Univerzita Hradec Králové
Fakulta informatiky a managementu
Katedra informatiky a kvantitativních metod

Zabezpečení webových aplikací

Bakalářská práce

Autor: Ondřej Boura
Studijní obor: Informační management

Vedoucí práce: Ing. Monika Borkovcová

Hradec Králové duben 2015

Prohlášení:

Prohlašuji, že jsem bakalářskou práci zpracoval samostatně a s použitím uvedené literatury.

V Hradci Králové dne 27.4.2015 Ondřej Boura

Poděkování:

Děkuji vedoucímu bakalářské práce Ing. Monice Borkovcové, která mi byla celou dobu nápomocna, za její trpělivost a ochotu, za metodické vedení práce, užitečné rady a připomínky.

Anotace

Tato bakalářská práce se zabývá zabezpečením webových aplikací. Je určená především pro vývojáře webových aplikací a lidem starajících se o jejich zabezpečení a poskytuje přehled možností, jak aplikace zabezpečit a na co si dát při vývoji takové aplikace pozor.

V první části jsou představeny nejzávažnější zranitelností či bezpečnostní nedostatky dnešních webových aplikací, útoky využívající těchto slabin a rizika, které s sebou tyto útoky nesou. V druhé části jsou popsány základní způsoby, jak tyto zranitelnosti odstranit a webovou aplikaci tak zabezpečit, metody pro obranu před útoky třetích stran a výhody či nevýhody plynoucí z použití jednotlivých metod. V třetí části jsou představeny nástroje, kterými je možné zranitelnosti identifikovat. Ve čtvrté části je testována a zabezpečena vytvořená webová aplikace.

Annotation

Title: Web Application Security

This bachelor thesis deals with the security of web applications. It is primarily intended for web application developers and people who take care of their security and provide an overview of how to secure applications and what to be careful when developing web application.

The first part presents the most common vulnerabilities of today's web applications attacks, exploiting these weaknesses and some risks of these attacks. The second section describes the basic ways to eliminate these vulnerabilities and secure web application, methods for defense against attacks by third parties and the advantages or disadvantages of using different methods. In the third part introduces the tools which can be useful to identify vulnerabilities of web applications. In the fourth part is tested and secured particular web application.

Obsah

Úvod

V současné době dochází k neustále rostoucí dostupnosti připojení k internetu. Je to již 10 let, kdy byla překonána hranice jedné miliardy uživatelů internetu celosvětově. Od té doby se tento počet zvýšil na trojnásobek, což představuje již více než 40 procent světového obyvatelstva. A růst se stále zrychluje [10].

S tímto nárůstem přichází i zvyšující se počet aplikací, které tohoto trendu využívají a jsou vyvíjené a poskytované jako webové na úkor „klasických" desktopových aplikací. Ty se vyznačují především tím, že není potřeba jakékoli instalace na straně uživatele, protože jsou provozovány na nějakém serveru v internetu a je k nim přistupováno pomocí webového prohlížeče.

Mezi hlavní výhody tohoto typu aplikací je bezpochyby jejich multiplatformnost. Aplikaci stačí napsat pouze jednou a ta poté může běžet na všech typech zařízení vybavených webovým prohlížečem. Velkou nevýhodou je ale nutnost připojení k internetu, aby bylo možné tuto aplikaci používat.

Tím se však stávají cílem mnoha „hackerů", kteří se pomocí nejrůznějších útoků snaží tyto aplikace nabourávat a dostávat z nich citlivá data a informace. To s sebou nese potřebu zabezpečit aplikace, s kterou musí počítat každý vývojář.

Zranitelností, které dnešní webové aplikace obsahují je celá řada. V této bakalářské práci bude poskytnut přehled těch nejčetnějších a nejzávažnějších, jako je Cross-site scripting, SQL injection, Information Leakage a Session management. Jednotlivé zranitelnosti budou popsány a představeny budou také typy útoků, které z nich hrozí. Své místo v této práci dostanou také nástroje, pomocí nichž je možné mnohé ze závažných zranitelností identifikovat. Práce bude věnována také základním způsobům zabezpečení, které by správná webová aplikace měla mít implementována.

Některé způsoby zabezpečení se odvíjejí od použitých technologií, kterými byla aplikace vytvořena. Ačkoli u nejrozšířenějších programovacích jazyků jako PHP, .NET nebo Java se mohou některé postupy zabezpečení lišit, základní princip bývá stejný, nezávisle na jazyku. V této práci je pro praktické ukázky útoků a pro implementaci ochran rozhodl použit jazyk Java. V práci jsou také použity jazyky

SQL pro komunikaci s databází, HTML pro tvorbu webových stránek a JavaScript pro demonstraci základních skriptů.

Pro praktickou ukázku zranitelností a konkrétních útoků na webové aplikace bude vytvořena jednoduchá webová aplikace, založená na platformě Java, která bude podrobena různým testům za účelem nalezení a identifikace jednotlivých zranitelností. Nakonec do aplikace budou implementována ochranná opatření a bude provedeno závěrečné testování, čímž se vyzkouší účinnost aplikovaných bezpečnostních opatření.

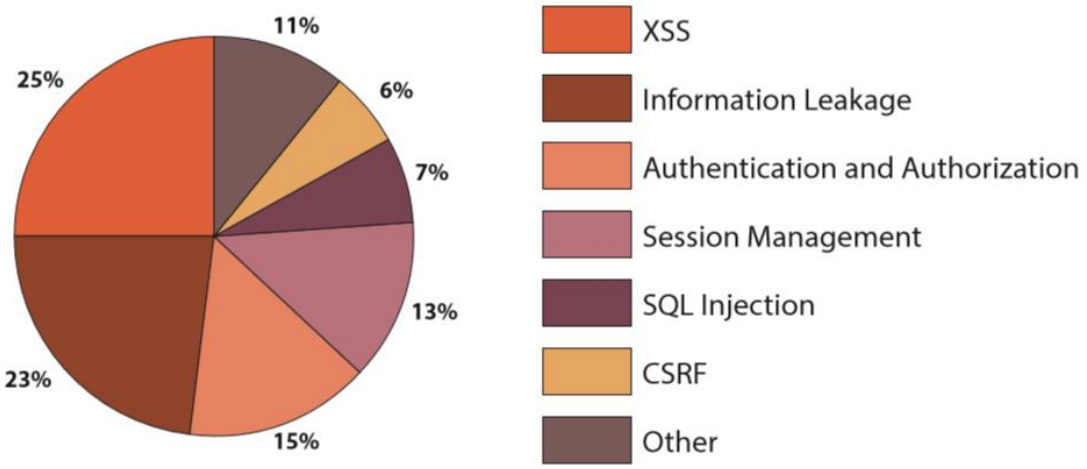
1 Zranitelnosti webových aplikací

Zranitelností webových aplikací je velké množství. Aplikací, které by byly chráněny proti všem možným typům útoků, je pouze malé procento. Podle [16] mělo 96% testovaných webových aplikací v roce 2013 alespoň jednu závažnou bezpečnostní zranitelnost. Medián počtu zranitelností na jednu webovou aplikaci je dokonce 14. Tento fakt nabízí útočníkům volbu toho, jakým způsobem provedou svůj útok s cílem získání hodnotných dat z webové aplikace [16].

Mnohé z těchto zranitelností bývá relativně jednoduché detekovat a opravit. Jsou zde však i zranitelnosti, jejichž objevení si žádá velké množství zkušeností programátorů webových aplikací. Častým zvykem mnohdy bývá jejich podceňování. Spousta odborníků přes zabezpečení mylně považuje některé zranitelnosti za neškodné a nevěnují jim proto příliš pozornosti.

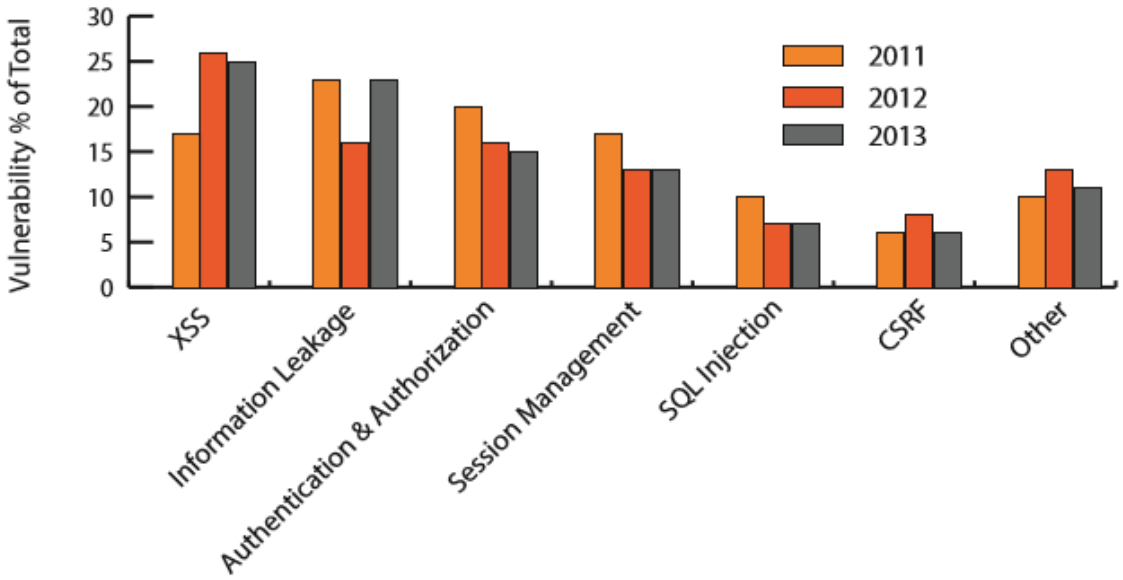
Nejvíce rozšířenou zranitelností v roce 2013 byla zranitelnost Cross-site scripting (XSS), která pokrývala celou čtvrtinu všech zranitelností v rámci testovaných aplikací. Podstatná část těchto aplikací měla navíc tuto zranitelnost obsaženu na více místech, což pro dotyčnou aplikaci znamená vážné bezpečnostní riziko [16].

Mezi další nejčastější zranitelnosti patří Information Leakage (23%), Authentication and Authorization (15%), Session Management (13%), SQL Injection (7%), Cross Site Request Forgery (CSRF) (6%), a další (11%).



Obr. 1: Podíl zranitelností webových aplikací v roce 2013, zdroj: [16]

Některé z těchto zranitelností jsou starší a jejich přítomnost je dobře známa už od počátků zrodu internetu a s ním spojených webových aplikací, jiné byli naopak objeveny relativně nedávno. Proti mnohým se daří nacházet účinnou obranu a částečně je tak možné se proti útokům hackerů bránit. Ruku v ruce s tím jsou však útočníci stále rafinovanější a vymýšlejí způsoby, jak tyto obrany obejít. Je tedy jasné, že podíl jednotlivých zranitelností se rok od roku mění. Následující graf ukazuje vývoj tohoto podílu od roku 2011 do roku 2013 [16].

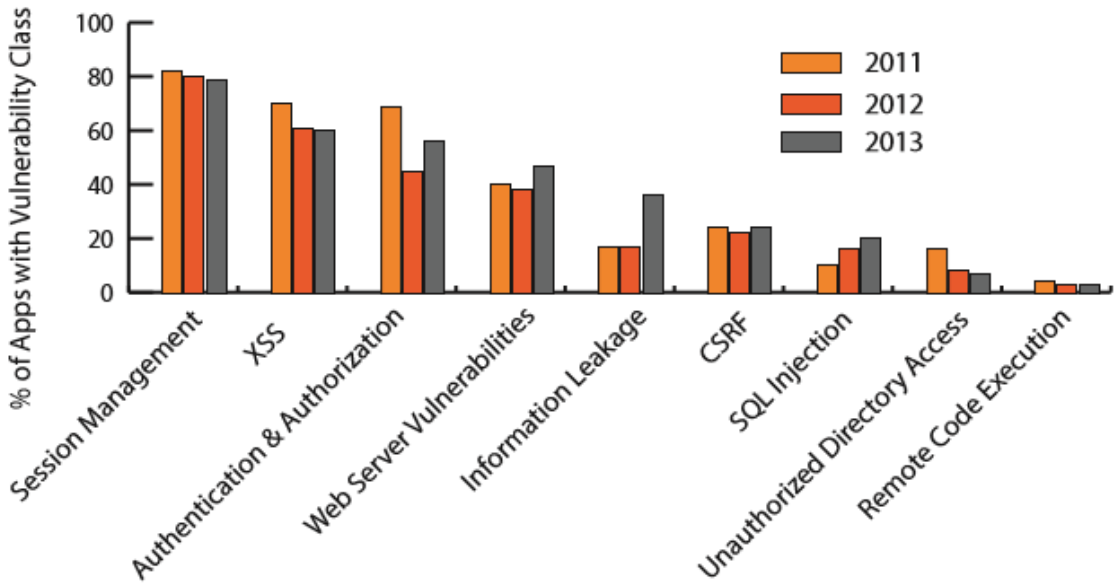


Obr. 2: Vývoj podílu zranitelností webových aplikací v letech 2011-2013, zdroj: [16]

Žádná z těchto nejrozšířenějších zranitelností nezaznamenala každoroční nárůst nebo pokles. Z uvedeného grafu je však možné vypozorovat zejména vysoký růst podílu zranitelnosti XSS, která se ze 17 procent v roce 2011 vyšplhala až na 25 procentní podíl v roce 2013. Naopak zranitelnosti jako Authentication and Authorization, Session Management a SQL Injection zaznamenali za sledované období jistý pokles. Tento trend ještě více zdůrazňuje potřebu zabývat se zranitelností XSS. Ačkoli je tato zranitelnost jedna z nejstarších, její nárůst v posledních letech láká stále více a více útočníků.

Tyto podíly zranitelností však neodpovídají podílům zranitelných webových aplikací. To je dáno zejména tím, že jeden konkrétní typ zranitelnosti se může vy-

skytovat vícekrát napříč jednou aplikací. Například zranitelností XSS nebo Information Leakage nemusí být postižena celá aplikace, ale tato bezpečnostní chyba se může týkat pouze dvou nebo tří webových stránek aplikace. Jiné se naopak mohou rozprostírat napříč celou aplikací. Větší vypovídající hodnotu o rozšířenosti webových zranitelností mají procentní podíly zranitelných aplikací zobrazené v následujícím diagramu.



Obr. 3: Vývoj pravděpodobností výskytu zranitelností v letech 2011-2013, zdroj: [16]

Některé z těchto zranitelností budou dále podrobněji rozebrány. Na praktických příkladech bude demonstrováno, jak by mohl vypadat útok využívající těchto slabin webových aplikací.

1.1 Cross Site Scripting

Zranitelnost Cross-Site Scripting je často označována zkratkou XSS (není označována CSS, protože tato zkratka slouží pro označení kaskádových stylů) a do češtiny je překládána jako skriptování napříč weby. V současné době jde o nejrozšířenější zranitelnost webových aplikací. Zároveň se jedná o jednu z nejstarších zranitelností vůbec. [1]

Obecně je tato velká rozšířenost spojována s nedostatečnou informovaností provozovatelů webových stránek. Druhým důvodem je fakt, že se tato zranitelnost

dostala do podvědomí uživatelů jako nepříliš nebezpečná. Mnozí si totiž myslí, že možnosti XSS spočívají pouze v zobrazení výstražného okna na webové stránce, jak bude předvedeno na pár příkladech v následujících kapitolách. To je však velký omyl. Skrze tuto zranitelnost mají útočníci možnost získat plnou kontrolu nad webovým prohlížečem nic netušícího uživatele zranitelné webové aplikace. Útočník má tímto způsobem možnost využít a získat uživatelovu session a pomocí ní se jednoduše dostat na jeho účet a napáchat tam velké škody, jak na uživatelově jméně nebo uložených datech, tak i v jeho financích. [2]

Základním předpokladem pro vykonání XSS útoku je povolení skriptovacího jazyka na straně uživatele. V dnešní době spousta webových aplikací vyžaduje ke správné funkčnosti, aby měl klient skriptovací jazyky povoleny. A to je další důvod toho, že jsou útoky tímto způsobem tak hojně rozšířené a lákají velké množství útočníků. Tato metoda je založena na injektování JavaScriptu do kódů webových stránek. Útoky využívající zranitelnosti XSS se rozdělují na tři základní typy právě podle způsobu injektáže skriptů. Těmito způsoby jsou perzistentní útok, non-perzistentní útok a útok založený na Document Objekt Modelu. Všechny zmíněné typy útoků budou dále podrobněji rozebrány v následujících kapitolách. [2]

1.1.1 Perzistentní XSS

První typ XSS útoku je nazývám jako perzistentní neboli trvalý. Vyznačuje se tím, že při něm dochází k přímé a trvalé modifikaci stránky. V případě injektáže JavaScriptu se jedná o uložení kódu JavaScriptu na webový server tak, aby se při každém zobrazení webové stránky načetl i injektovaný skript. Toto tvrzení však odporuje principu samotného XSS, který uvádí, že při XSS útoku není zasahováno do kódů uložených na webovém serveru. Jedná se totiž o metodu, při které JavaScriptový kód skutečně není vkládán do kódů webové aplikace, ale je ukládán do datového úložiště dané aplikace, která poté sama během každého zobrazení webové stránky tyto skripty přečte a zakomponuje je do vráceného obsahu HTML[1]. [2]

[1] HTML - HyperText Markup Language

S perzistentní typem XSS útoku se můžeme nejčastěji setkat na diskusních fórech, komentářích pod články nebo obecně tam, kde mají uživatelé možnost uložit trvale na webový server svůj vlastní řetězec znaků.

Máme-li například webovou stránku obsahující článek a pod tímto článkem jsou příspěvky uživatelů a formulář pro přidání nového příspěvku. Výpis příspěvků je realizován pomocí jednoduchého JSTL[2] tagu *<c:foreach>*, který vypisuje přezdívku autora a text každého příspěvku, jak ukazuje následující kód:

```
<c:forEach items="${prispevky}" var="prispevek"
   varStatus="status">
   <b>${prispevek.autor.jmeno}: </b> ${prispevek.text} <br>
</c:forEach>
```

Kód 1: Výpis všech příspěvků JSTL cyklem foreach

Načtená webová stránka pak každému uživateli, který tuto stránku zobrazí, vygeneruje skript s výpisem všech doposud vytvořených příspěvků v následující podobě:

```
<b>Petr:</b> Velmi kvalitní článek. <br>
<b>Honza:</b> Jen více takových článků!! :) <br>
<b>Jirka:</b> Slabší článek. <br>
```

Kód 2: Vygenerovaný skript všech příspěvků v HTML

Uživateli se podle očekávání zobrazí seznam příspěvků společně s jejich autory. Pokud však uživatel „Hacker“ vytvoří nový příspěvek a namísto běžného příspěvku „Ahoj všichni. Jak se máte?“ vloží text „Ahoj všichni. <h1>Jak se máte?</h1>“, vygenerovaný skript poté bude mít následující podobu:

```
<b>Petr:</b> Velmi kvalitní článek. <br>
<b>Honza:</b> Jen více takových článků!! :) <br>
<b>Jirka:</b> Slabší článek. <br>
<b>Hacker:</b> „Ahoj všichni. <h1>Jak se máte?</h1>“ <br>
```

Kód 3: Vygenerovaný HTML skript s injektovaným kódem

V tomto případě se do stránky injektuje část HTML kódu, který se následně při generování výsledné stránky vloží do výsledného kódu. Na stejném principu je

[2] JSTL - JavaServer Pages Standard Tag Library

založen i perzistentní XSS útok. Namísto injektování HTML kódu skrze formulář je možné vložit do příspěvku skript, např.:

```
<script>alert(„Toto je útok XSS!“);</script>
```

Kód 4: Skript jazyka JavaScript

Jakmile je do databáze aplikace uložen příspěvek s tímto skriptem, při každém načítání dat (tj. při načítání webové stránky s tímto diskuzním fórem) dojde se spuštění vloženého skriptu a uživateli se zobrazí výstražné okno s nápisem „Toto je útok XSS!“.

Vložení skriptu do nechráněné webové aplikace je tedy velice jednoduché a nemusí jít pouze o jednoduchý skript s metodou *alert()* z předchozího příkladu. Útočník může tímto typem útoku docílit mnoha různých výsledků, protože má k dispozici všechny možnosti, které JavaScript nabízí webovým vývojářům a záleží pouze na jeho schopnostech a vynalézavosti, jakým způsobem svůj útok provede. Nejčastějšími cíli útočníků bývají různá přesměrování na jinou webovou stránku, smazání nebo nahrazení obsahu webové stránky, změna cíle v přihlašovacím formuláři, krádež uživatelských session (cookies) nebo odeslání nevyžádaných e-mailů. [2]

Diskuzní fóra však nemusí být jediným místem, kde je možné se s perzistentním XSS útokem setkat. Obecně se tento typ útoku může vyskytnout všude tam, kde dochází k zobrazování údajů, které byly vytvořeny a vloženy samotnými uživateli webové aplikace. Příkladem takových míst může být:

- webové rozhraní e-mailových účtů
- zobrazení uživatelského profilu
- zobrazení tabulky nejlepších hráčů

Trvalý XSS útok je ve většině případů pro administrátora napadené webové aplikace snadno rozpoznatelný. Změna obsahu webové stránky nebo přesměrování je evidentně patrné na první pohled a webmaster má možnost ji ihned odstranit. Perzistentního XSS však může být zneužito i nenápadným způsobem, například ke krádeži cookies. Útočný skript v tomto případě může zůstat dlouho skryt. Takovýto útok bývá velmi závažný a nebezpečný. Napaden je totiž každý z návštěvníků we-

bové aplikace, který se proti XSS sám aktivně nechrání (například vypnutím cookies nebo zakázáním JavaScriptu). 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. [2]

1.1.2 Non-perzistentní XSS

Druhým typem XSS útoku je tzv. non-perzistentní útok. Tato metoda, někdy označována také jako reflektované XSS, je zároveň označována za nejčastější XSS zranitelnost. Představuje totiž základní způsob injektování skriptů do obsahu webových stránek. [2]

Také v tomto případě se jedná o vkládání skriptu do obsahu webové stránky. Od perzistentního útoku se odlišuje především tím, že při něm nedochází k trvalému uložení skriptu do databáze na straně serveru. Skript je předáván webovému serveru při každém odeslání požadavku na zaslání obsahu webové stránky. Server předaný skript zakomponuje jednorázově do obsahu HTML dokumentu a výsledek vrátí k zobrazení webovému browseru klienta. 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. [2]

Způsob, jakým se skripty během non-perzistentního XSS dostávají od uživatele na server, se různí podle toho, zda jsou proměnné HTTP[3] požadavku odesílány metodou GET, nebo zda je pro jejich odesílání použita metoda POST.

Metoda GET

Při použití metody GET jsou všechny proměnné požadavku doručovány jako součást URI[4]. Tento způsob předávání proměnných je například využíván v případě všech požadavků vzniklých při přímém zápisu URL[5] adresy do adresního řádku nebo u požadavků vzniklých kliknutím na běžný odkaz tvořený HTML tagem *<a>*. Od adresy URL se při použití této metody odděluje část proměnných znakem „?“

[3] HTTP - Hypertext Transfer Protocol
[4] URI - Uniform Resource Identifier
[5] URL - Uniform Resource Locator

a konkrétní proměnné mezi sebou znakem „&“. URI takového požadavku pak může mít podobu:

http://obchod.cz?kat=elektronika&cena=5000

Tohoto předávání proměnných je možné zneužít pro předání útočného skriptu. Takový skript je útočníkem zakomponován do URI odkazu a ten je předán oběti útoku. Té již pouze stačí, aby dotyčný odkaz následovala, přešla na odkazovanou stránku, čímž by došlo k odeslání připraveného skriptu webovému serveru. Aplikace na serveru převezme proměnnou obsahující útočný skript, vloží jej do obsahu vyžádaného dokumentu a následně tento dokument vrátí oběti. Oběti je tento HTML dokument prohlížečem zobrazen a současně s vlastním zobrazením dokumentu je také vykonán vložený skript. [2]

Způsobů, jak útočníci předávají své útočné odkazy obětím je mnoho. Příkladem může být zaslání odkazu e-mailem, soukromou zprávou uvnitř webové aplikace nebo zveřejnění odkazu kdekoli na webu (takovým místem často bývají různá diskuzní fóra).

Příklad reflektovaného XSS útoku předvedu na jednoduchém vyhledávači internetového obchodu. Takový vyhledávač je v rámci webové stránky realizován pomocí jednoduchého formuláře využívajícího metodu GET:

```
<form name="formular" method="get">
   <input type="text" name="dotaz">
   <input type="submit" value="Hledej">
</form>
```

Kód 5: HTML formulář I

Odeslání tohoto formuláře přesměruje uživatele na webovou stránku, která zobrazí výsledky hledání. Tato stránka začíná obvyklou hláškou: „Pro hledanou frázi: *dotaz* byly nalezeny tyto výsledky:“. Samotné výsledky hledání momentálně nejsou důležité. Klíčovou informací pro útočníka je fakt, že uživatelem zadaný dotaz se celý zobrazí na stránce. Z pohledu útočníka to indikuje potenciálně zranitelné místo v aplikaci i bez znalosti samotného kódu aplikace. Problém je v tom, že se uživatelský vstup objevuje na výstupu aplikace. Kód zobrazující uvedenou hlášku může vypadat nějak takto:

```
<b> Pro hledanou frázi: ${dotaz} byly nalezeny tyto
výsledky:</b>
```

Kód 6: HTML kód obsahující vstup uživatele

Problém tedy nastává v případě, když útočník namísto hledaného dotazu zadá kód JavaScriptu, podobně jako u výše popsaného perzistentního XSS. Odešle-li uživatel formulářem hledaný dotaz „<script>alert(„Toto je útok XSS!“);</script>“, webový server vrátí stránku s výsledky, a společně s jejím načtením dojde i ke spuštění vloženého skriptu, který do kódu injektuje výraz ${dotaz}. Uživateli se tak zobrazí vyskakovací okno, jako tomu bylo v předchozí kapitole při perzistentním útoku XSS.

Z tohoto příkladu je patrné, že injektovaný skript opravdu nemusí být uložen v žádném úložišti. Aplikace pouze přejímá hodnotu z předané proměnné. Otázkou však je, jak útočník přinutí svojí oběť k tomu, aby na server odeslala právě kód jeho skriptu. Pravděpodobně nebude svoji oběť přesvědčovat o tom, že má daný skript přepsat přímo do vyhledávacího pole. Jak bylo zmíněno výše, v praxi se takový útok provádí nejčastěji pomocí odkazů umístěných volně na webu nebo v diskuzním fóru. Takový odkaz by mohl vypadat takto:

http://obchod.cz/hledat?dotaz=<script>alert(„Toto je útok XSS!“);</script>

Metoda POST

U metody POST nedochází k předávání proměnných prostřednictvím URI jako u metody GET, ale v samotném těle požadavku HTTP. Tato metoda se zcela výhradně využívá k odesílání požadavků s citlivými údaji jako je tomu například u přihlašovacích a registračních formulářů. Zranitelnost XSS je i v tomto případě možné odhalit stejnými postupy jako u metody GET, ale pro útočníka již není možné jednoduše podstrčit útočný odkaz se skriptem své oběti. Útočník se musí postarat o odeslání požadavku POST ze strany své oběti jiným způsobem. Tato situace se řeší přesměrováním na jinou webovou stránku, která se o odeslání útočného POST požadavku sama postará.

Útočník tak nejprve kamkoli na web umístí odkaz, který neobsahuje žádné parametry ani žádný útočný skript. Takový odkaz pouze směřuje na „nakaženou“ webovou stránku, která obsahuje mechanismy pro okamžité odeslání připrave-

ných dat zahrnujících útočný skript. Tato data se odesílají metodou POST ke zranitelné webové aplikaci, na které již dojde ke spuštění útočného skriptu shodně jako je tomu u metody GET.

Tento způsob má oproti odeslání přímého odkazu metodou GET výhodu především v tom, že je daleko méně nápadný a důvěryhodnější, protože odkaz na zdánlivě bezpečnou stránku ve svém těle neobsahuje parametr s vloženým skriptem.

Pro příklad poslouží jednoduchá webová stránka s přihlašovacím formulářem, která očekává vložení jména a hesla.

```
<form name="formular" method="post">
   <input type="text" name="jmeno“>
   <input type="password" name="heslo">
   <input type="submit" value="Přihlásit">
</form>
```

Kód 7: HTML formulář II

V případě chybně zadaného hesla při odeslání tohoto formuláře je o tom uživatel informován zprávou: „Chybně zadané heslo pro login *jmeno*“, která je v kódu realizována podmínkou:

```
<c:if test=“${not empty jmeno}“>
   „Chybně zadané heslo pro login ${jmeno}“,
</c:if>
```

Kód 8: Podmínka if jazyka JSTL

Při odeslání formuláře s vyplněným polem pro jméno již známým skriptem <script>alert(„Toto je útok XSS!“)</script>, dojde ke spuštění skriptu a zobrazí se, podobně jako v předešlých příkladech, výstražné okno s nápisem „Toto je útok XSS!“. Zbývá však ještě vyřešit druhou část útoku a tou je předání parametrů pomocí odkazu. K tomuto účelu slouží stránka, která se postará o automatické přesměrování a odeslání patřičných proměnných. Tato stránka obsahuje následující formulář s metodou POST a nastavenou akcí na cílovou webovou stránku s přihlašovacím formulářem:

```
<form name="form" method="post" action=" www.obchod.cz/login">
   <input name="jmeno" value="<script> alert(„Toto je útok
   XSS!")</script>">
   <input type="hidden" name="heslo" value="abcd">
</form>
```

Kód 9: HTML formulář III

Tento formulář je automaticky odesílán pomocí skriptu nacházejícího se na stejné stránce a obsahující metodu „*submit()*“ [Kód 10], která způsobí okamžité odeslání příslušného formuláře po načtení stránky.

```
<script>formular.submit();</script>
```

Kód 10: Skript jazyka JavaScript II

Poté již stačí tuto stránku nazvat například *post.html* a umístit následující tag na jakoukoli webovou stránku:

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

Kód 11: Tag HTML

1.1.3 DOM-based XSS

Třetí typ XSS útoku je na rozdíl od prvních dvou útoků založen na Document Object Modelu. To má za následek, že využívá rozdílné hodnoty parametrů, které se zakomponují do obsahu HTML stránky. U perzistentního a non-perzistentního XSS útoku se tyto hodnoty vkládaly do obsahu stránky již na straně serveru a ten poté vracel uživateli hotový dokument. U XSS útoku založeném na DOM dochází ke vkládání hodnot parametrů až na straně webového prohlížeče a to pomocí JavaScriptu. [2]

Příkladem tohoto typu zranitelnosti může být webová stránka obsahující formulář pro vyplnění jména, který následně data předává metodou GET jako součást URI. Součástí této stránky je i JavaScript, který následně pomocí parametru v URI získá tyto data. Hodnota tohoto parametru je tak převedena z URL kódování a vložena do obsahu stránky. Zdrojový kód těla takové stránky by mohl vypadat následovně:

```
<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>
   <form method="get">
       Zadej své jméno:
       <input type="text" name="jmeno">
       <input type="submit" value="Odešli">
   </form>
</body>
```

Kód 12: Ukázka HTML stránky se zranitelností DOM-based XSS

Poněvadž v tomto příkladu se vyskytuje absence jakékoliv kontroly uživatelského vstupu, má útočník možnost vložit do formuláře řetězec v podobě útočného skriptu podobně, jako jsem demonstroval u non-perzistentního typu XSS využívajícího metodu GET v kapitole 1.1.2.

Aplikace s DOM-based XSS zranitelností mohou být také ty, u nichž se pomocí JavaScriptu čte a zobrazuje obsah cookies. Právě cookies v sobě mohou obsahovat například uživatelské jméno. Také u aplikací, u nichž JavaScript zjišťuje a zobrazuje informace jako použitý webový prohlížeč nebo verze operačního systému uživatele. Už z tohoto popisu je zřejmé, že zneužití této zranitelnosti je mnohem komplikovanější, protože útočník nejprve musí modifikovat tyto hodnoty na straně své oběti. [2]

1.2 SQL Injection

SQL Injection je druh injection útoku, jehož cílem je změnit dotaz na databázi aplikace. Tato zranitelnost je podobně jako u XSS způsobena vstupy formulářů webové aplikace, do kterých však útočník namísto nebezpečných skriptů vkládá SQL dotazy, které pak aplikace odesílá na databázový server a může tam napáchat značné škody.

Touto zranitelností již netrpí takové množství webových aplikací, jako tomu bylo před několika lety. V posledních letech se podařilo tuto zranitelnost omezit

pomocí nejrůznějších způsobů obrany (některé z nich budou zmíněny v kapitole 2). Navzdory tomu se však SQL Injection stále řadí mezi nejčastější zranitelnosti webových aplikací vůbec a pravděpodobně tomu tak ještě dlouhou dobu bude (alespoň dokud se pro ukládání dat budou používat relační databáze tak, jako tomu je dnes). Této problematice je však zapotřebí věnovat stále velkou pozornost z důvodu, že se rozhodně jedná o jednu z nejzávažnějších zranitelností, a útoky, s ní spojené, mají často fatální následky. Jejich cílem je totiž nejcitlivější část každé webové aplikace, tedy uložená data v databázi. [11]

1.2.1 Rizika útoku SQL Injection

Z rizik spočívajících v útoku SQL Injection je jasné, že ochrana webové aplikace je nutností. Možná rizika jsou [14]:

- Neoprávněný přístup k citlivým datům webové aplikace (uživatelská hesla, atd.)
- Přístup k účtům aplikace
- Modifikace dat v databázi
- Smazání dat nebo celé databáze
- Ovládnutí celé webové aplikace

1.2.2 Způsoby útoku SQL Injection

Při útoku SQL Injection využívá útočník vstupních prvků formulářů v aplikaci, jejichž hodnoty jsou odesílány na databázový server. Shodný princip využívají také dříve popsané útoky XSS. Pro vyhledání zranitelných vstupů je možné použít nástroje schopné automatické detekce těchto náchylných vstupů, jako jsou Sqlmap[6], BSQL Hacker[7] nebo Sqlninja[8]. [7]

Jakmile útočník nalezne zranitelné vstupy v aplikaci, může provést svůj útok. Možné způsoby útoků budu dále demonstrovat na příkladu s webovou stránkou obsahující jednoduchý přihlašovací formulář (Kód 13).

---

[6] http://sqlmap.org/
[7] https://github.com/portcullislabs/bsql-hacker
[8] http://sqlninja.sourceforge.net/

```
<form name="formular" method="post">
   <input type="text" name="jmeno">
   <input type="password" name="heslo">
   <input type=" submit" value="Přihlásit">
</form>
```

Kód 13: HTML formulář IV

Tento formulář je při kliknutí uživatele na tlačítko „Přihlásit“ odesílán na webový server a spolu s ním i uživatelem zadané hodnoty „*jmeno*“ a „*heslo*“. Webový server tyto hodnoty přijme a vloží je do SQL dotazu, který odešle na databázový server (viz. Kód 14).

```
Connection con = pool.getConnection();
Statement st = con.createStatement();
String dotaz = "SELECT * FROM user WHERE username = '" +
   jmeno +"' AND password = '" + heslo + "'";
ResultSet rs = st.executeQuery(dotaz);
return (rs.next()) ? true : false;
```

Kód 14: Komunikace s databází v jazyce Java

Při přihlašování se porovnají uživatelem zadané přihlašovací údaje s daty uloženými v tabulce *user*. Jestliže výsledek dotazu je kladný, znamená to, že se přihlašovací údaje shodují s databází a uživatel je přihlášen. Pokud však databázový server nevrátí žádnou hodnotu, ověření údajů se nezdařilo a uživatel přihlášen nebude.

Toto je typický příklad neošetřeného požadavku na databázi. Útočník má totiž možnost vložit do vstupu pro jméno část svého kódu a změnit tak původní SQL dotaz. Dále jsou uvedeny tři základní způsoby, kterými je možné tento dotaz upravit.

Využití komentářů

Tato technika spočívá ve vkládání znaků či slov, kterými se v jazyce SQL značí začátek komentáře. Interpret jazyka SQL pak ignoruje všechny znaky dotazu, které jsou za komentářem a vykoná tak pouze část dotazu. Nejčastěji se tímto způsobem zakomentovávají části dotazů za podmínkou „WHERE“. Mezi tato značení patří [4]:

- Dvojitá pomlčka („--“)
- Hashtag („#”)

- Lomítko + hvězdička („/*“)

Pokud do vstupního pole pro jméno uživatele zadáme „admin’ --“, následující dotaz bude vypadat takto:

```
SELECT * FROM user WHERE username = 'admin’ -- ' AND
password='heslo'
```

Kód 15: SQL dotaz I

Protože „--“ je slovo, kterým začíná v jazyce SQL komentář, provede se pouze část dotazu před ním. Takový dotaz vrátí libovolného uživatele i bez znalosti jeho hesla, v tomto případě uživatele „admin“.

Spojování dotazů

Technika spojování dotazů spočívá ve vykonání více dotazů v rámci jedné transakce. Oddělujícím prvkem rozlišující jednotlivé dotazy je znak „;“. Vložíme-li do vstupu pro jméno řetězec „admin’; DROP TABLE user --“, dostaneme konečný dotaz ve formě:

```
 SELECT * FROM user WHERE name = ’admin’; DROP TABLE user -- ’
AND password = ’heslo’
```

Kód 16: SQL dotaz II

V tomto případě dojde po odeslání dotazu do databáze nejprve k vyhledání uživatele „admin“ a poté se druhým dotazem smaže celá tabulka *user*.

Vkládání uvozovek

Technika vkládání uvozovek je založena na ukončení jedné podmínky dotazu a následném přidání nové podmínky. Příkladem může být vložení řetězce „admin’ OR ’1’=’1“ do vstupního pole pro jméno. Výsledný dotaz bude vypadat následovně:

```
SELECT * FROM user WHERE name = ’admin’ OR ’1’=’1’ AND
password = ’heslo’
```

Kód 17: SQL dotaz III

Databáze po obdržení tohoto dotazu vyhodnotí složenou podmínku jako „true“ a opět vrátí webovému serveru uživatele „admin“.

1.3 Information Leakage

Infomation Leakage (únik informací), je zranitelnost webových aplikací spočívající v odhalování citlivých dat. Těmi mohou být různé technické detaily aplikace, její prostředí nebo uživatelská data. Takové údaje mohou být využity útočníkem k prozkoumání a bližšímu pochopení funkčnosti své cílové aplikace. Únik těchto důvěrných dat by měl být limitován a pokud možno zcela odstraněn. Ačkoli tato zranitelnost aplikace samotná neznamená bezprostřední riziko, bývá často nápomocna při odhalení dalších slabin jako například výše popsané SQL Injection. Ve své nejběžnější formě mívá únik informací některou z těchto podob [9]:

- Zapomenuté smazání HTML/JavaScript komentářů, které obsahují citlivá data aplikace
- Nesprávná konfigurace na straně aplikace a serveru, na které běží
- Rozdíly v chování webových stránek aplikace při zadání platný a neplatných dat

1.3.1 Zapomenuté komentáře

Ponechání některých komentářů jazyka HTML nebo JavaScript během provozu aplikace může vést k úniku citlivých, kontextových informací. Mezi ně se řadí zejména podoba vnitřní adresářové struktury, podoba konstrukce SQL dotazů nebo vnitřní síťové informace. Častým zvykem vývojáře webové aplikace bývá ponechání komentářů pro případ pozdější úpravy kódu nebo vyskytnutí případných chyb a následné debuggování.

Příkladem chybného ponechání řádkového komentáře v kódu HTML je následující ukázka. [17]

```
<TABLE border="0" cellPadding="0" cellSpacing="0" height="59"
width="591">
    <TBODY>
        <TR>
    <!--Pokud se obrázek nenačte, zkontroluj 192.168.0.110 -->
            <TD bgColor="#ffffff" colSpan="5" height="17"
                width="587"> </TD>
        </TR>
```

Kód 18: HTML tabulka s vloženým komentářem

Na této ukázce je možné vidět komentář zanechaný vývojářem naznačující, co je potřeba udělat v případě, když se nezobrazí obrázek. Tato volně přístupná informace poskytuje vnitřní IP adresu serveru, což může znamenat potenciální výhodu pro útočníka.

1.3.2 Nesprávná konfigurace serveru

Softwarové verze nebo upovídané chybové hlášky jsou typickým příkladem nesprávné konfigurace serveru. Tyto informace jsou útočníkům užitečné poskytováním detailního pohledu na strukturu aplikace zahrnující použité nástroje jako různé jazyky či frameworky. Nejzákladnější serverová konfigurace poskytuje vývojáři čísla verzí aplikace a obsáhlé chybové hlášky, které mu jsou nápomocny při debuggování a hledání chyb v aplikaci.

Příkladem obsáhlé chybové hlášky je odezva na neplatný SQL dotaz. Útok SQL Injection je možné snadněji provést, pokud má útočník k dispozici informace o struktuře a formátu SQL dotazů použitých v cílové aplikaci. Informace uniklá skrze obsáhlou chybovou hlášku poskytuje útočníkovy znalost, jak sestavit útočný SQL dotaz na databázi. [17]

Následující chybová hláška byla zobrazena uživateli po vložení znaku apostrofu do vstupního pole pro jméno v rámci přihlašovacího formuláře [17]:

```
An Error Has Occurred.
Error Message:
System.Data.OleDb.OleDbException: Syntax error (missing
operator) in query expression 'username = ''' and password =
'g''. at
System.Data.OleDb.OleDbCommand.ExecuteCommandTextErrorHandlin
g ( Int32 hr) at…
```

Kód 19: Výpis chybové hlášky

Chybové hlášení varuje před neplatnou syntaxí a odhaluje parametry použitého SQL dotazu, „username“ a „password“. Tyto uniklé informace mohou být vítanou pomocí pro útočníka při sestavování útočného SQL dotazu.

1.3.3 Neošetřené neplatné hodnoty vstupů

Webové stránky poskytující rozdílné odpovědi v závislosti na platnosti vložených dat mohou vést k úniku informací. Obzvláště pokud data, považována za důvěrné,

jsou odhalována jako výsledek designu aplikace. Citlivá data uniklá touto formou mohou být různá. Od čísel účtů, nejrůznějších uživatelských údajů (čísla občanského nebo řidičského průkazu, cestovního průkazu, čísla kreditní karty, atd.) až po uživatelské informace jako hesla nebo emailové adresy. Únik informací v této souvislosti se tedy zabývá odhalováním klíčových uživatelských dat, které by neměly být vystavovány nikde na očích, a to ani pro dotyčného uživatele.

Příklad rozdílné odezvy v závislosti na platnosti vložených dat je funkce „zapomenuté heslo“ při přihlašování uživatele do systému, která dělá danou aplikaci uživatelsky přívětivější. Avšak kvůli veřejnému přístupu této funkce má útočník možnost využít ji k získání platných uživatelských jmen nebo emailových adres. Služba zapomenutého hesla mívá tyto tři kroky [17]:

1. Zadání uživatelského jména (adresy)
    a. Pokud je uživatelské jméno (adresa) platné, pokračuje se krokem 2
    b. Pokud uživatelské jméno (adresa) není platné, je zobrazena chybová hláška: „Zadané uživatelské jméno (adresa) je neplatné!“
2. Zobrazení hlášky uživateli o odeslání emailu na adresu daného účtu
3. Odeslání odkazu umožňujícího změnu hesla.

V tomto případě je únik informací způsoben odlišným chováním aplikace při zadání platného a neplatného uživatelského jména (adresy). Tento rozdíl umožňuje útočníkům odvodit platné emailové adresy nebo jména uživatelských účtů.

1.4 Session management

Jak vyplývá z grafu [Obr. 3] uvedeného v kapitole 1, zranitelnost Session management je nejrozšířenější zranitelností, která postihuje téměř 80% webových aplikací. Jedná se zároveň o jednu z nejzávažnějších zranitelností, protože výsledkem útoků využívajících tuto zranitelnost je úplné ovládnutí uživatelských účtů webové aplikace.

Aby bylo možné porozumět zranitelnosti způsobenou špatnou správou relací (session management), musí být nejprve vysvětleno, jaký smysl mají tzv. relace (sessions) v rámci webové aplikace.

Komunikace přes internet využívá HTTP protokol. HTTP je bezstavový protokol, což znamená, že neposkytuje informace o stavech jednotlivých požadavků. Jinak řečeno, webové servery nemají možnost zjistit, odkud požadavky přicházejí, jestli od nového klienta nebo od klienta, který již s daným server komunikoval. Z pohledu webového serveru tedy každý příchozí požadavek vypadá jako nový. Z tohoto důvodu existují relace a s nimi spojená správa relací. Jde o proces sledování uživatelských aktivit, které se zahrnují do relací jako interakce s webovým prohlížečem klienta. [4]

Tohoto principu využívají webové aplikace, které tak umožňují uživateli například trvalé přihlášení v aplikaci z jednoho webového prohlížeče. Pokud se uživatel přihlásí do aplikace, vytvoří se nová relace mezi webovým serverem a webovým prohlížečem klienta, která je označena unikátním identifikátorem (tzv. session ID, dále SID). Při každém novém požadavku (načtení webové stránky) na webový server aplikace zjistí uživatelovo SID a ten tak může pokračovat ve své relaci.

Existují tři možné způsoby, jak může být SID sledováno [4]:

- **Cookies** – Veškerá SID jsou spravována serverem a jsou odesílána uživatelům ve formě cookies. Ty jsou poté uložena na uživatelově pevném disku a následně využívána při každém požadavku na načtení webové stránky. Toto je nejpoužívanější způsob sledování SID.
- **Přepis URL** – Hodnoty SID jsou odesílána jako parametr URL při každém novém požadavku.
- **Skrytá pole** – Jedná se o elementy, které jsou pro uživatele neviditelné. Tato skrytá pole jsou obsažena ve webové stránce a obsahují hodnoty SID a jsou odesílána na server při každém novém požadavku. Tato metoda však již v dnešní době není běžně využívána.

Útok využívající zranitelnosti špatné správy relací se nazývá fixace relací (session fixation). Definice tohoto útoku podle [5]: “Fixace relací je útok, který umožňuje útočníkům unést uživatelskou relaci. Při ověřování uživatele se nevytváří nové SID, ale použije se jiné, již existující SID. Útok se skládá ze získání platného SID

(přihlášením se do aplikace) a následného únosu uživatelské relace za použití ukradeného SID."

Útočník se tedy nejprve sám přihlásí do aplikace, čímž získá SID a to následně podstrčí své oběti, která se taktéž do aplikace přihlásí. Poté již pouze stačí útočníkovi aktualizovat stránku webové aplikace, čímž docílí přihlášení na účet své oběti. Podle způsobu předání své SID oběti existují tři běžné techniky fixace relací.

1.4.1 Fixace parametrem URL

Útočník může předat oběti své SID pomocí parametru URL, které jí podstrčí podobně, jako u non-perzistentního typu útoku XSS využívajícího metody GET popsaného dříve, tedy například odkazem odeslaným pomocí emailu. Útočný odkaz by mohl vypadat následovně.

http://obchod.cz/login?sid=p6f2ac1th3j8qw7s9hzu2kl3s

Jakmile oběť použije tento odkaz, vyplní přihlašovací údaje a úspěšně se přihlásí, útočník získá přístup do aplikace na stejný účet, na kterém je přihlášena jeho oběť.

1.4.2 Fixace skrytým polem v HTML

Na stejném příkladu je možné demonstrovat i druhý způsob fixace relací a to pomocí skrytého pole. Tento útok využívá podobný princip jako typ non-perzistentního útoku XSS využívajícího metodu POST, tedy využití nakaženého formuláře obsahujícího SID ve skryté poli jako jeden z parametrů a přesměrovávajícího uživatele na přihlašovací stránku zranitelné aplikace. Cílová webová aplikace poté využije SID zaslané formulářem pro přihlášení oběti tohoto útoku. [5]

1.4.3 Fixace pomocí cookies

Třetím a posledním způsobem předání SID uživateli je využití JavaScriptové funkce pro nastavení cookies ve webovém prohlížeči oběti útoku. Tato metoda využívá zranitelnosti XSS. Jejím principem je vložení skriptu s funkcí JavaScriptu pro nastavení SID do uživatelských cookies jako parametr URL následným způsobem: [5]

http://obchod.cz/login?txt=<script>document.cookies="SESSIONID=
p6f2ac1th3j8qw7s9hzu2kl3s"</script>

Po kliknutí na tento odkaz se uživateli nastaví cookies SessionID na SID útočníka a útočník má možnost po přihlášení uživatele využít jeho účet.

2 Metody ochrany a jejich implementace

Na všechny výše popsané typy útoků existuje alespoň částečná obrana. Tato kapitola je věnována metodám, jak svojí webovou aplikaci chránit a jak odstranit jednotlivé druhy zranitelností.

2.1 Ochrana před útoky typu XSS

I když před útoky typu Cross-site scripting je možné se do určité míry bránit také na straně uživatele, tato práce má za cíl představit a popsat především ochranu na straně aplikace. Problematice zranitelnosti XSS byla věnována kapitola 1.1. Z uvedeného popisu je jasné, že zranitelnosti XSS jsou důsledkem špatně ošetřeného výstupu. Každý výstup, který webový server prezentuje v rámci výsledného kódu HTML stránky, by měl být před jeho vložením zkontrolován, aby se zabránilo injektování kódu HTML a různým spustitelným skriptům. [2]

2.1.1 Zákaz veškerého HTML

V případě, že chceme zakázat, aby vstup uživatelů obsahoval jakýkoli HTML nebo JavaScriptový kód, je řešení velice jednoduché. Injektáž skriptů je totiž možná pouze v případě, že se útočníkovi podaří vložit do kódu HTML stránky vlastní tagy. Z toho vyplývá, že pro takovou injekci je nutné použít některé znaky, které mají speciální význam, jak uvádí tabulka (Tabulka 1).

Znak	Význam
<	Závorka značící začátek tagu
>	Závorka značící konec tagu
“	Uvozovka ohraničující textový řetězce
`	Apostrof ohraničující textový řetězec
\	Zpětné lomítko umožňující Unicode zápis znaků

Tabulka 1: Speciální významy nebezpečných znaků, zdroj: [2]

Zabránění injektáži skriptů je tak možné docílit pomocí filtrování uvedených znaků a jejich následným nepouštěním do výstupů aplikace. Filtrace těchto

potenciálně nebezpečných znaků se provádí nahrazením za jiné, bezpečné znaky, uvedené v tabulce (Tabulka 2).

Znak	Bezpečný zápis znaku
<	&lt;
>	&gt;
“	&quot;
`	' nebo '

Tabulka 2: Alternativní zápisy nebezpečných znaků, zdroj: [2]

Náhradu potenciálně nebezpečných znaků za jejich bezpečné ekvivalenty je možné provést vytvořením vlastní funkce, kterou poté budeme volat při každém výstupu. To však bývá většinou zbytečné. Většina nejrozšířenějších programovacích jazyků již má takovou funkci implementovanou. V PHP je to například funkce „*htmlspecialchars()*“. Java má ve své výbavě dokonce dva způsoby zápisu této funkce. Jedním je využití JSTL tagu *<c:out>* a tím druhým je funkce „*fn:escapeXml()*“ sady Expression Language. [2]

Použití tohoto principu bude demonstrováno na příkladu uvedeného v kapitole 1.1.1. Výpis jednotlivých příspěvků a jejich autorů by v ošetřené podobě vypadal následovně:

```
<c:forEach items="${prispevky}" var="prispevek"
   varStatus="status">
   <b><c:out value="${prispevek.autor.jmeno}"></b>
      ${fn:escapeXml(prispevek.text)}<br>
</c:forEach>
```

Kód 20: Ošetřený výpis cyklem foreach jazyka JSTL

Výsledkem takto ošetřeného vstupu je výstup, kde veškeré nebezpečné znaky byly nahrazeny bezpečnými.

```
&lt;script&gt;alert(&quot; Toto je útok
XSS!&quot;)&lt;/script&gt;
```

Kód 21: Výsledek ošetřeného výstupu

2.1.2 Content Security Policy

Ošetřování všech výstupů je sice velmi účinné, avšak velice zdlouhavé a snadno se může stát, že vývojář webové aplikace nějaké zranitelné místo přehlédne. Před několika lety proto začal vývoj ochrany, která by lépe zabránila útoků XSS. V roce 2012 tak vznikl koncept nového nástroje založeného na rozlišování původu vkládaného obsahu a jeho následného legitimování. Tímto nástrojem je bezpečnostní politika Content Secutity Policy. [2]

Použití tohoto nástroje umožňuje vývojářům definovat zdroje dat vkládaných do obsahu webové stránky a zamezit tak injektáži zakázaného obsahu. Webové aplikace tak zavádějí základní omezení jako je například [2]:

- Zákaz veškerých skriptů vkládaných tagem *<script>* přímo do obsahu HTML dokumentu
- Zákaz definování ovladačů událostí přímo u jednotlivých prvků HTML
- Zákaz použití konstruktoru
- Zákaz použití pseudoprotokolu javascript

Podpora Content Secutity Policy musí být implementována samotnou webovou aplikací, a zároveň také webovým prohlížečem uživatele. 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[9]. Tato informace se předává HTTP hlavičkou „X-Content-Security-Policy“. V defaultní nastavení má webový prohlížeč zakázáno spuštění jakéhokoli kódu a je dále na vývojáři, aby explicitně sdělil, co má být povoleno. Pro toto vyjmenování povolených zdrojů slouží různé direktivy, například [2]:

- **default-src** - Definuje zdroje, které mohou být defaultně použity pro všechny prvky
- **script-src** - Definuje zdroje, ze kterých mohou být načítány externí skripty
- **img-src** - Definuje zdroje, ze kterých mohou být načítány externí obrázky

[9] CSP - Content Secutity Policy

Příkladem http hlavičky dokumentu pak může být následující ukázka:

```
X-Content-Security-Policy: default-src 'self'; img-src *
```

Kód 22: HTTP hlavička dokumentu

Tato hlavička povoluje načítání čehokoliv ze stejného hostitele, ze kterého pochází dotyčný chráněný dokument a zároveň umožňuje načítání obrázků odkudkoliv.

2.2 Ochrana před útoky typu SQL Injection

Typům útoků SQL Injection a rizik, které z nich vyplývají, byla věnována kapitola 1.2. Pro vývojáře webové aplikace naštěstí existuje hned několik způsobů, jak těmto útokům zamezit a aplikaci ochránit. Zároveň je možné říci, že implementace této obrany je velice snadná. Nejpoužívanějšími metodami jsou Parametrizace SQL dotazů a Escapování vstupních dat. [8]

2.2.1 Parametrizace SQL dotazů

Parametrizace SQL dotazů se provádí pomocí tzv. Prepared Statementů (Parametrizované dotazy). Jejich používání se v posledních letech stalo standardem pro správné psaní databázových SQL dotazů. Jejich vytváření je velice snadné a zároveň jsou snadněji pochopitelné než klasické dotazy. [12]

Parametrizované dotazy nutí programátora nejprve definovat celý SQL dotaz a až poté se zpětným dosazováním do tohoto dotazu doplní jednotlivé hodnoty vstupů. Tento styl psaní SQL dotazů dovoluje databázi rozlišovat mezi samotným kódem a daty, bez ohledu na to, jaká data uživatel aplikaci dodal. Tyto dotazy zároveň zajišťují, že není možné změnit dotaz, i kdyby došlo ke vložení některého z SQL kódu na vstupu aplikace popsaného v kapitole 1.2.2. [12]

Pokud tuto techniku aplikujeme na příklad z kapitoly 1.2.2, výsledný kód bude vypadat následovně:

```
Connection con = pool.getConnection();
String dotaz = "SELECT * FROM user WHERE username = ? AND
password = ?;
PreparedStatement pst = con.prepareStatement(dotaz);
pst.setString(1, jmeno);
pst.setString(2, heslo);
```

```
ResultSet rs = pst.executeQuery();
return (rs.next()) ? true : false;
```

Kód 23: Ošetřená komunikace s databází v jazyce Java

Zadá-li uživatel do vstupního pole pro jméno řetězec „admin' --" stejně jako v příkladu v kapitole 1.2.2, z důvodu parametrizovaného dotazu nedojde ke změně výsledného SQL dotazu, ale místo toho databázový server bude hledat doslovně zadané jméno, tedy uživatele se jménem, shodujícím se s celým předloženým řetězcem „admin' --".

2.2.2 Další metody ochrany

Vedle parametrizovaných dotazů existují i jiné způsoby ochrany před SQL Injection, ale ty nebývají příliš používané, jednak z důvodu, že nejsou vždy zcela účinné, a zároveň, protože parametrizace dotazů je nejjednodušší a dostatečný způsob. Další možné metody ochrany tedy pouze ve zkratce.

Escapování vstupních dat

Jednou z možností je ošetření nepovolených znaků tak, aby po vložení do dotazu nebyly považovány za klíčová slova jazyka, ale pouze za řetězec. Tato technika je nazývaná escaping a jejím principem je modifikování řetězců tak, že před znaky s jiným než textovým významem vloží lomítko. Nástroje na tuto změnu řetězců poskytuje většina jazyků, v Javě tuto funkcionalitu nabízí třída *StringEscapeUtils*.[8]

Výhoda použití této metody oproti parametrizovaným dotazům se projeví zejména v případě, aplikujeme-li ochranu před SQL Injection na již hotové aplikaci, protože není zapotřebí přepisovat veškeré SQL dotazy na databázi. Poněvadž ale zabezpečení touto formou nebývá vždy zcela spolehlivé, je upřednostňována metoda použití Prepared Statementů.[8]

Princip nejmenších privilegií

Jednou z metod, jak snížit rizika SQL Injection, je i využití pravidla nejmenších privilegií. Jeho principem je nastavení uživatelům databáze co nejmenšího oprávnění v rámci databáze. Výhoda pak spočívá v lepší ochraně neuživatelských dat. I kdyby

k napadení databáze došlo, útočník se dostane pouze k datům aplikace, na která má oprávnění a nemá tak možnost narušit celou databázi.[8]

Uložené procedury

Využití uložených procedur (Stored Procedures) je dalším řešením, jak zabezpečit aplikace před SQL Injection. Metoda spočívá v prvotním definování SQL kódu a následném dosazením parametrů podobně, jako u parametrizovaných dotazů. Tyto dva přístupy se liší především v rozdílném skladování SQL kódů. Procedury jsou totiž uloženy na databázovém serveru a aplikace si je volá pouze v případě potřeby.[8]

2.3 Ochrana před útoky typu Session Fixation

Typu útoku fixace relací byla věnována kapitola 1.4, v rámci zranitelnosti způsobenou špatnou správou relací. Z uvedeného popisu je zřejmé, že se jedná o útok využívající identifikátorů relací (SID) k získání přístupu k uživatelským účtům aplikace. [6]

Nejlepším řešením ochrany je vytváření nových SID při každém úspěšném přihlášení do aplikace. To má za následek, že každý uživatel použije pouze takové SID, které zná pouze on sám. Z pohledu serveru pak proces vytváření SID při nově příchozím HTTP požadavku má následující kroky [15]:

1. Získat z HTTP požadavku SID a označit ho jako „OLD_SID“
2. Pokud žádné takové OLD_SID neexistuje, vytvořit novou relaci
3. Vygenerovat nové SID jako „NEW_SID“
4. Označit relaci HTTP požadavku novým NEW_SID
5. Odeslat nové SID požadavku klientovi

Kromě vytváření nových SID při každém přihlášení je doporučováno chránit aplikaci také dalšími pomocnými technikami, které pomáhají snižovat škody v případě tohoto útoku. Mezi doporučované techniky patří[6]:

- Vypršení platnosti relace po určitém časovém okamžiku.
- Implementování funkce automatického odhlášení uživatele
- Využití bezpečnostního protokolu SSL pro přenos SID

3 Automatické nástroje pro testování zranitelností webových aplikací

V kapitole 2 byly popsány základní metody a postupy ochrany, kterými je možné jednotlivé zranitelnosti odstranit. Testování webových aplikací spočívá v hledání takových míst v aplikaci, které by potenciálně mohly obsahovat některou ze zranitelností. Například u zranitelnosti XSS jde o vkládání znaků, které mohou mít v určitém kontextu zvláštní význam, do všech vstupů, na které v aplikaci narazíme. Po vložení vstupu je pak potřeba prozkoumat vrácený HTML kód, a případně ošetřit daný výstup. [2]

Vstupů, které by při testování webové aplikace neměli být opomenuty, jsou mimo jiné [2]:

- 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

Existuje tedy mnoho variant vstupů, které je nutné prověřit. V případě, že se jedná o rozsáhlejší aplikaci, je takovéto ruční testování velmi zdlouhavé a velice snadno se na některý ze vstupů zapomene. Z toho důvodu bylo vytvořeno mnoho nástrojů, které testování usnadňují a do jisté míry i automatizují. Tyto nástroje jsou také někdy nazývány jako skenery zranitelností webových aplikací a vyznačují se především schopností otestovat velké množství funkcionality aplikace v relativně krátkém čase a ve většině běžných aplikací umožňují identifikovat spoustu nejzásadnějších zranitelností. [3]

Skenery pracují napříč celou funkcionalitou aplikace na principu předkládání široké škály různých řetězců vstupům jednotlivých formulářů, přičemž analyzují veškeré výstupy a pátrají po známkách běžných aplikačních zranitelností. Uži-

vatelům pak poskytují hlášení o výskytu jednotlivých zranitelností v aplikaci. Tato hlášení obvykle obsahují:

- **Vstup** (řetězec), který byl využit k objevení zranitelnosti.
- **Výstup**, který daný výstup vyvolal.
- **Umístění** v aplikaci, kde se zranitelné místo vyskytuje.
- **Rizika** plynoucí z dané zranitelnosti.
- **Návrh** řešení ochrany.

	Name	Module
+	Blind SQL Injection (5)	Scripting (Blind_Sql_Injection.script)
+	Cross site scripting (verified) (1)	Scripting (XSS.script)
+	Directory traversal (1)	Scripting (Directory_Traversal.script)
+	Microsoft IIS tilde directory enumeration (1)	Scripting (IIS_Tilde_Dir_Enumeration.script)
+	Script source code disclosure (1)	Scripting (Script_Source_Code_Disclosure.script)
+	SQL injection (verified) (15)	Scripting (Sql_Injection.script)
+	Weak password (2)	Scripting (Html_Authentication_Audit.script)
+	Application error message (11)	Scripting (Error_Message.script)
+	HTML form without CSRF protection (3)	Crawler
+	User credentials are sent in clear text (2)	Crawler
+	ASP.NET version disclosure (1)	Scripting (ASP_NET_Error_Message.script)
+	Clickjacking: X-Frame-Options header missing (1)	Scripting (Clickjacking_X_Frame_Options.script)
+	Cookie without HttpOnly flag set (1)	Crawler
+	Cookie without Secure flag set (1)	Crawler
+	Login page password-guessing attack (4)	Scripting (Html_Authentication_Audit.script)
+	OPTIONS method is enabled (1)	Scripting (Options_Server_Method.script)

Obr. 4: Ukázka hlášení o nalezených zranitelnostech skeneru Acunetix, zdroj: autor

Ukázku takové zprávy, která obsahuje jednotlivá hlášení o zranitelnosti webové aplikace, obsahuje obrázek [Obr. 4]. Zde je vidět ukázka vygenerované zprávy o ukončené analýze, která obsahuje některé závažné položky jako výskyt zranitelnosti XSS, SQL Injection a další. Po rozkliknutí položky se uživateli zobrazí detailnější informace popsané výše.

3.1 Odhalení zranitelností

Skenery zranitelností webových aplikací jsou schopné odhalit mnohé zranitelnosti s vysokou mírou spolehlivosti. Jedná se především o ty zranitelnosti, které nabízí jen omezené množství podob, ve kterých se vyskytují. Zde je uveden přehled těch zranitelností, které mohou být snadno detekovány automatickým nástrojem a zároveň způsoby, jimiž je toho dosaženo [3]:

- Reflektované (non-perzistentní) XSS zranitelnosti vznikají, když uživatelem dodané vstupy se odráží v odpovědích aplikace bez jakéhokoli ošetření. Automatické skenery obvykle odesílají testovací řetězce obsahující skripty a hledají odpovědi, které jim pomohou odhalit mnohé nedostatky.
- Některé zranitelnosti SQL Injection a další injekce mohou být odhaleny například předložením řetězce *'; waitfor delay '0:0:30',* což může mít za následek zpoždění odpovědi aplikace,
- Zranitelnosti jako přenášení hesla v podobě prostého textu nebo automaticky odesílané formuláře jsou spolehlivě odhaleny pomocí kontroly požadavků a odpovědí aplikace.
- Úniky výpisů adresářové struktury mohou být identifikovány prostřednictvím dotazování se na cestu k adresáři a výsledným výpisem adresáře.

Na druhou stranu existují i takové zranitelnosti, které se nevyznačují žádnou standardní podobou a nelze je vystopovat pomocí žádného obecného řetězce. Obecně platí, že automatické skenery jsou při odhalování takových zranitelností neúčinné. Mezi ty se řadí například [3]:

- Zranitelnosti využívající úpravu hodnot těch parametrů, které mají specifický význam v rámci aplikace. Například skrytá pole reprezentující cenu nebo stav objednávky.
- Porušení přístupových práv, což uživateli umožňuje přístup k datům jiných uživatelů nebo k datům určených pro administrátory. Skener nerozezná požadavky na přístupová práva aplikace ani nemá možnost posoudit význam uložených dat.
- Úniky některých citlivých dat jako například seznamy uživatelů obsažených v session.
- Logické chyby v aplikaci jako je například porušování limitů transakcí pomocí záporné hodnoty nebo obcházení stádia procesu obnovy účtu vynecháním klíčového parametru požadavku.

3.2 Příklady nástrojů

Ačkoli většina skenerů webových aplikací je založena na stejném funkčním základu, existují odlišnosti v různých přístupech detekce některých zranitelností. Na základě provedené rozsáhlé studie skenerů [3], která měla za cíl přiřadit každému skeneru skóre na základě schopnosti identifikovat různé typy zranitelností, se mezi nejlépe hodnocené nástroje řadí Acunetix[10], WebInspect[11], Burp Scanner[12] a N-Stalker[13]. Výsledky studie zobrazuje tabulka [Tabulka 3].

SCANNER	SCORE	PRICE
Acunetix	14	$4,995 to $6,350
WebInspect	13	$6,000 to $30,000
Burp Scanner	13	$191
N-Stalker	13	$899 to $6,299
AppScan	10	$17,550 to $32,500
w3af	9	Free
Paros	6	Free
HailStorm	6	$10,000
NTOSpider	4	$10,000
MileSCAN	4	$495 to $1,495
Grendel-Scan	3	Free

Tabulka 3: Porovnání výkonnosti skenerů zranitelností webových aplikací, zdroj: [3]

Studie mimo jiné zjistila, že neexistuje příliš velká souvztažnost mezi cenou a výkonností testovaných skenerů. Některé volně dostupné nebo nepříliš drahé nástroje dokázaly svým výkonem zdatně soupeřit i s těmi výrazně nákladnějšími. Pro spolehlivé testování zranitelností webové aplikace tak vývojáři nemusí investovat do pořízení drahých skenerů. [3]

[10] https://www.acunetix.com/
[11] http://www8.hp.com/cz/cs/software-solutions/webinspect-dynamic-analysis-dast/
[12] http://portswigger.net/burp/scanner.html
[13] http://www.nstalker.com/

4 Vytvoření a testování vlastní webové aplikace

Pro účel analýzy a testování výše popsaných zranitelností bude v této kapitole vytvořena vlastní webová aplikace. Bude se jednat o aplikaci, která bude zajišťovat základní možnosti komunikace v podobě jednoduchého diskuzního fóra.

4.1 Cíle aplikace

Tato aplikace bude záměrně vytvářena bez implementace jakýchkoli ochranných opatření, které by zabraňovaly výskytu zranitelností. Jako ukázkovou aplikaci, jejíž vytvoření je součástí této práce a na které budou ukázány zranitelnosti a způsoby ochrany popsané v kapitole 2, byla zvolena aplikace, sloužící pro základní potřeby týmové komunikace z důvodu, že se jedná o dostatečně jednoduchou aplikaci na to, aby neobsahovala redundantní funkce, které nejsou předmětem této práce, a zároveň je dostatečně složitá na to, aby na ní bylo možné otestovat všechny typy zranitelností, popsaných v této práci.

Vytvořená aplikace bude nejprve podrobena automatickému skeneru zranitelností webových aplikací za použití vybraného nástroje, který poskytne základní report o stavu aplikace. Bude tak získáno hlášení o potenciálních výskytech nebezpečných zranitelností, na základě kterých bude následně provedeno manuální testování.

Manuální testy budou provedeny zejména metodou vkládání různých řetězců do vstupů aplikace a jejich cílem bude potvrzení konkrétních případů zranitelných míst a ohodnocení jejich závažnosti na chod aplikace. Na zranitelných místech poté budou implementována ochranná opatření a otestována jejich funkčnost. Takto zabezpečená aplikace poté bude znovu otestována pomocí automatického skeneru a manuálního testování.

4.2 Základní funkcionalita

Základním účelem bude vytvoření takové aplikace, která bude obsahovat základní možnosti každého diskuzního fóra. Základní funkcionalitu této aplikace lze shrnout do těchto bodů:

- Každý uživatel se bude moci zaregistrovat.
- Každý zaregistrovaný uživatel se bude muset přihlásit pod svým jedinečným jménem a heslem.
- Přihlášeným uživatelům se zobrazí seznam všech příspěvků.
- Uživatelé budou moci přidávat nové příspěvky.
- Uživatelé budou moci filtrovat příspěvky podle konkrétního jména uživatele.

4.3 Použité technologie a architektura aplikace

Pro vytvoření této aplikace byl jako hlavní programovací jazyk použit jazyk Java, konkrétně platforma Java EE (Java Enterprise Edition). Jedná se o součást jazyka Java, která je určena pro vytváření a vývoj webových aplikací. Tato platforma využívá principy architektury MVC[14], což mimo jiné znamená, že uživatelské rozhraní je striktně odděleno od řídící logiky a datového modelu aplikace.

4.3.1 Klientská část

Uživatelské rozhraní aplikace běží v prohlížeči klienta a skládá se z jednoduchých JSP stránek, které jsou založeny na jazyce HTML a vkládaném kódu jazyka Java. O design jednotlivých stránek se starají kaskádové styly a framework Twitter Bootstrap. Tyto použité technologie dále krátce popíši.

HTML5

HTML5 je dosud poslední verzí značkovacího jazyka HTML. Jedná se o standard sloužící pro tvorbu webového obsahu. Základem jazyka HTML jsou tzv. tagy (značky), které slouží pro definování webového obsahu. Základními částmi takového HTML dokumentu jsou oblasti ohraničené značkami *head*, které slouží pro definici meta informací o webovém dokumentu a dále oblast ohraničená značkami *body* pro definici samotného webového obsahu.

[14] MVC - Model View Controller

JavaServer Pages

JavaServer Pages (JSP) je technologie pro vývoj dynamických HTML stránek založená na jazyce Java a je součástí platformy Java EE. Při vytváření dynamických stránek se v tomto případě používá jazyk HTML, do kterého je vkládán kód jazyku Java. Ten je od samotného HTML kódu oddělen tagy *<% … %>*. Tyto bloky kódu se nazývají skriptlety a slouží k přijímání a zpracování dat přijatých ze serveru. Soubory obsahující JSP mají koncovku *.jsp*.

Technologie JSP využívá knihovnu JSTL (JSP Standard Tag Library). Jedná se o set tagů pro zjednodušení vývoje JSP stránek, ve kterých není potřeba používat scriptlet tagy. V aplikaci jsou z této knihovny využity základní tagy pro práci s proměnnými, s URL a kontrolu a správu stránek.

CSS3

O design jednotlivých JavaServer Pages se starají kaskádové styly. CSS (Cascading Style Sheets) je jazyk pro definování vzhledu HTML aplikací. Tento jazyk byl navržen organizací W3C a aktuálně existuje již třetí specifikace tohoto jazyka označována jako CSS3.

Syntaxe tohoto jazyka je velmi jednoduchá a skládá se ze dvou částí, a to selektoru a bloku deklarace. Jako selektor se nejčastěji používá samotný název HTML tagu, jeho atribut id nebo atribut class. V bloku deklarace je poté popsán samotný způsob zobrazení daného elementu.

Twitter Bootstrap[15]

Kromě kaskádových stylů je na design uživatelského rozhraní aplikace použit v dnešní době velice populární framework Twitter Bootstrap.

Jde o velmi jednoduchý a volně dostupný soubor nástrojů pro vytváření moderního webu a webových aplikací. Nabízí podporu nejrůznějších webových technologií jako použité HTML, CSS, JavaScript a mnoho prvků, které je možné snadno implementovat do své stránky. Pro použití Twitter Bootstrap jsou nutné pouze základní znalosti HTML a CSS. Interaktivní prvky jako jsou tlačítka, boxy,

[15] http://getbootstrap.com/

menu a další kompletně nastavené a graficky zpracované elementy je možné vložit pouze pomocí HTML a CSS.

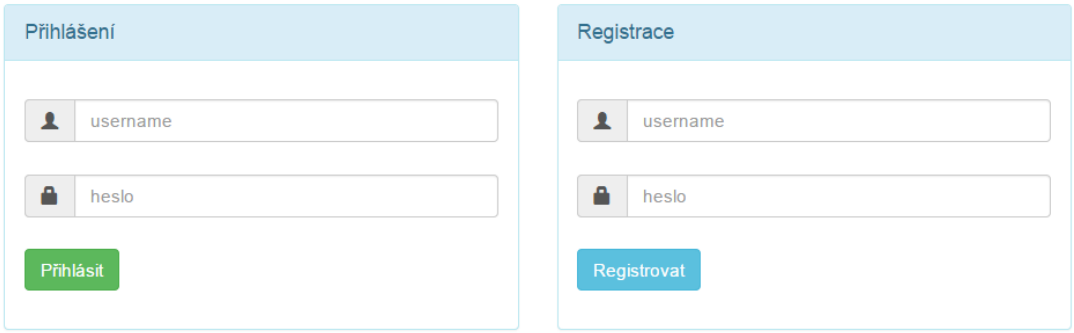
4.3.2 Serverová část

Serverová část má na starosti řídící logiku a datový model aplikace. Aplikace je založena na principu běžných webových aplikací, kdy na konkrétní URL adresu je namapován potřebný controller. O funkci controllerů se starají Java Servlety, které zpracovávají příchozí požadavky na server a odesílají odpovědi ve formě JSP ke klientovi.

Datový model je realizován prostřednictvím datových tříd a DAO tříd, které obstarávají komunikaci s databází MySQL pomocí SQL dotazů.

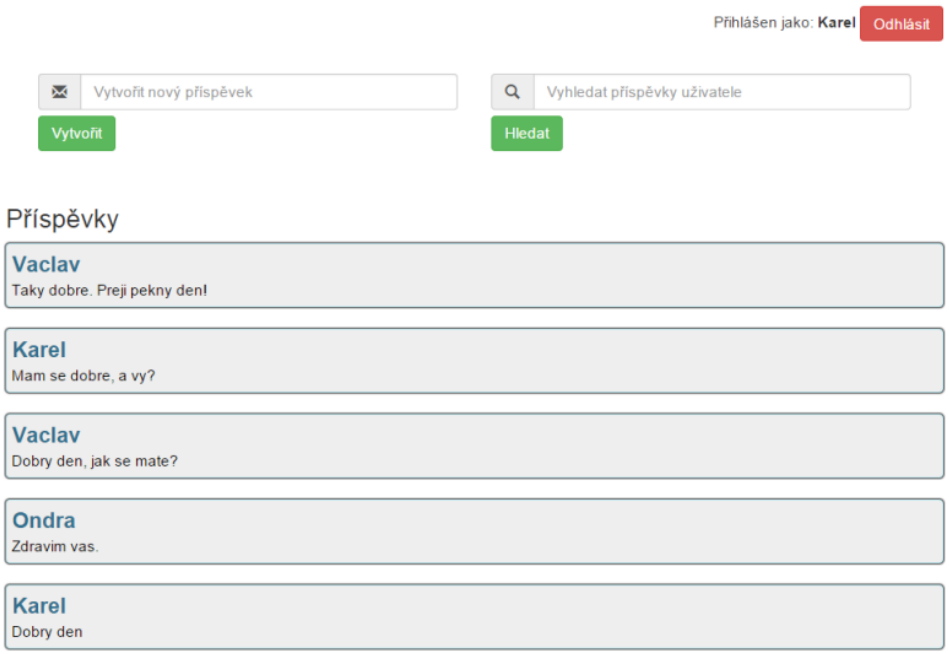
4.3.3 Ukázky z vytvořené aplikace

Následující obrázky ukazují uživatelské rozhraní vytvořené aplikace. Na obrázku [Obr. 5] lze vidět úvodní stránku aplikace obsahující dva formuláře pro přihlášení uživatele a vytvoření nového účtu.



Obr. 5: Úvodní stránka vytvořené aplikace, zdroj: autor

Na obrázku [Obr. 6] je hlavní stránka aplikace, která se uživateli zobrazí po úspěšném přihlášení. Tato stránka obsahuje výpis všech příspěvků, pole pro vytvoření nového příspěvku a pole pro filtrování příspěvků podle jména uživatele.



Obr. 6: Hlavní stránka vytvořené aplikace, zdroj: autor

4.4 Počáteční testování aplikace

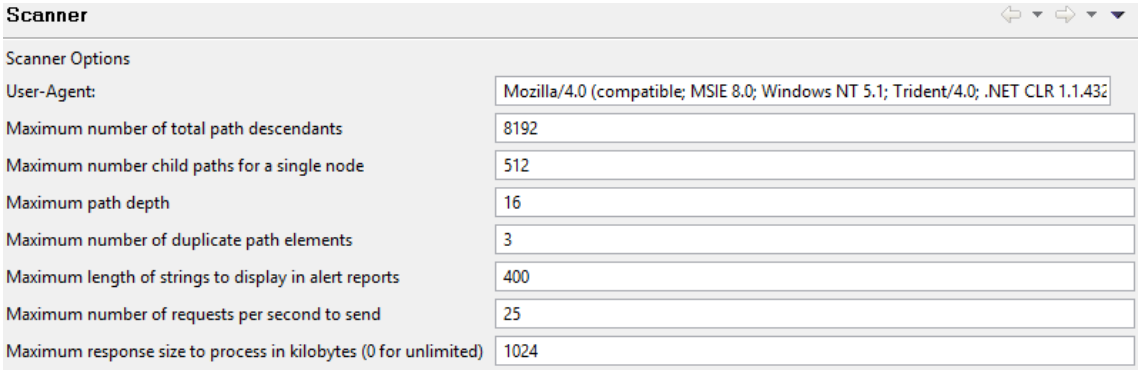
Jak již bylo uvedeno, při vytváření testovací aplikace nebyla věnována přílišná pozornost jejímu zabezpečení. Nyní bude provedeno několik testů, pomocí nichž se zjistí, zda obsahuje některé závažné zranitelnosti.

4.4.1 Automatický skener

Aplikace bude nejprve podrobena testu pomocí automatického nástroje Vega[16]. Vega je volně dostupný open source skener a testovací platforma pro testování bezpečnosti webových aplikací. Umožňuje vyhledat a ověřit zranitelnosti typu SQL Injection, Cross-site scripting, úniky citlivých informací a jiné. Nástroj byl vyvinut společností Subgraph v Montrealu. Program je napsán kompletně v programovacím jazyku Java a je k dispozici pro operační systémy Windows, Linux a OS X.

[16] https://subgraph.com/

Výhodou tohoto nástroje je možnost rozšíření pomocí silného API[17] napsaného v JavaScriptu. Skener Vega umožňuje nastavení šíře a hloubky skenování aplikace, počet dotazů za sekundu a mnoho dalšího. Nastavení skeneru ukazuje obrázek [Obr. 7].



Obr. 7: Nastavení skeneru v programu Vega, zdroj: autor

Skener Vega disponuje také přehledným uživatelským rozhraním a kvalitně zpracovanou dokumentací.

Výsledky vlastního testování

Aplikace byla otestována v celé její šíři a hloubce. Obrázek [Obr. 8] ukazuje výsledný report o objevených zranitelnostech vygenerovaný po ukončení automatického skenu.

Obr. 8: Report skeneru Vega o zjištěných zranitelnostech aplikace, zdroj: autor

[17] API - Application Programming Interface

Z výsledného reportu [Obr. 8] je možné vyčíst, že skener dokázal identifikovat některé potenciální zranitelnosti. Ty rozřadil do tří kategorií podle jejich závažnosti a rizika, které z nich hrozí. Jednotlivé výskyty zranitelností budou dále blíže popsány.

Cleartext Password over HTTP

Prvním identifikovaným závažným nedostatkem je posílání hesel ve formě prostého textu skrze HTTP. Skener dokázal nalézt formulář obsahující vstupní pole pro heslo, který je odesílán prostřednictvím protokolu HTTP. Problém je v tom, že tento protokol neumožňuje jakékoli šifrování ani zabezpečení integrity dat.

Tato chyba zabezpečení aplikace může vést k prozrazení posílaných hesel v případě odposlouchání komunikace mezi klientem a serverem. Hodnoty hesel by proto nikdy neměly být odesílány pomocí nezabezpečeného protokolu HTTP. Namísto toho by měly vždy využívat bezpečného protokolu HTTPS[18].

Session Cookie Without Secure Flag

Druhá nebezpečná chyba zabezpečení detekována skenerem se týká zranitelnosti Session management, které byla věnována kapitola 1.4. Varování poukazuje na to, že hodnoty cookies mohou být odposlouchány třetí stranou a při jejich vytváření by měly vždy být opatřeny tzv. bezpečnou vlajkou. [13]

Bezpečná vlajka (secure flag) je volitelná forma zabezpečení, jenž může být nastavena aplikačním serverem v případě odesílání cookies uživateli v rámci odpovědi serveru na požadavek uživatele. Účelem tohoto zabezpečení je ochránit cookies před odposloucháním během komunikace mezi serverem a klientem v důsledku přenosu cookies v podobě prostého textu. Jedná se o podobný princip jako výše popsaný problém s posíláním nešifrovaného hesla. [13]

Cross Site Scriptnig

Problém zranitelnosti Cross Site Scripting byl vysvětlen v kapitole 1.1. Tato zranitelnost byla v aplikaci nalezena ve dvou různých podobách:

- Při vytváření nového příspěvku jako perzistentní forma XSS.

[18] HTTPS - Hypertext Transfer Protocol Secure

- Při neúspěšném pokusu o filtrování příspěvků podle jména ve formě non-perzistentního XSS.

SQL Injection

Dalším závažným detekovaným problémem aplikace je zranitelnost proti SQL Injection. Ta byla rozebrána v kapitole 1.2. V testovací aplikaci se vyskytla hned na třech místech, a to v případě:

- registrace uživatele
- přihlášení uživatele
- neúspěšného pokusu o filtrování příspěvků podle jména

Java Debug Output Detected

Poslední zjištěná zranitelnost, která je zařazena mezi ty středně závažné, se týká úniku citlivých informací, konkrétně nesprávné konfigurace serveru. Této problematice byla věnována kapitola 1.3.2. V aplikaci se tato zranitelnost vyskytuje při vytváření nového příspěvku. Jak může k takovému okamžiku dojít, bude vysvětleno v kapitole 4.4.2.

4.4.2 Ruční testování

Nyní budou na aplikaci provedeny vlastní testy za účelem prezentování některých zranitelností popsaných v kapitole 1, nebo těch, které byly zjištěny pomocí automatického nástroje v předcházející kapitole. Testy budou provedeny simulováním útoků, které by mohly takovou webovou aplikaci postihnout.

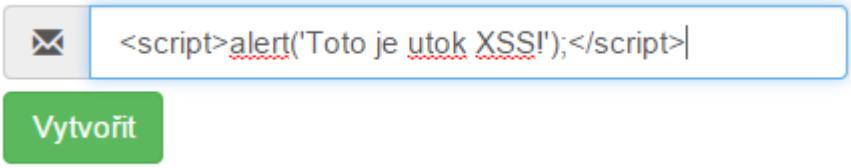
4.4.2.1 Testování Cross Site Scripting

Zranitelnost XSS byla testována na formuláři pro vytváření nového příspěvku. Ten se skládá z jednoho vstupního pole pro zadání textu příspěvku a tlačítka pro odeslání [Kód 24].

```
<form action="<c:url value='/vytvoritPrispevek'/>"
   method="GET">
   <input type="text" name="text" value=""
      placeholder="Vytvořit nový příspěvek">
   <button type="submit">Vytvořit</button>
</form>
```

Kód 24: HTML formulář aplikace

Namísto klasického textu má však útočník k dispozici možnost vložit do vstupního pole formuláře útočný skript. Vkládání tohoto skriptu do vstupního pole formuláře z pohledu uživatele je zachyceno na obrázku [Obr. 9].



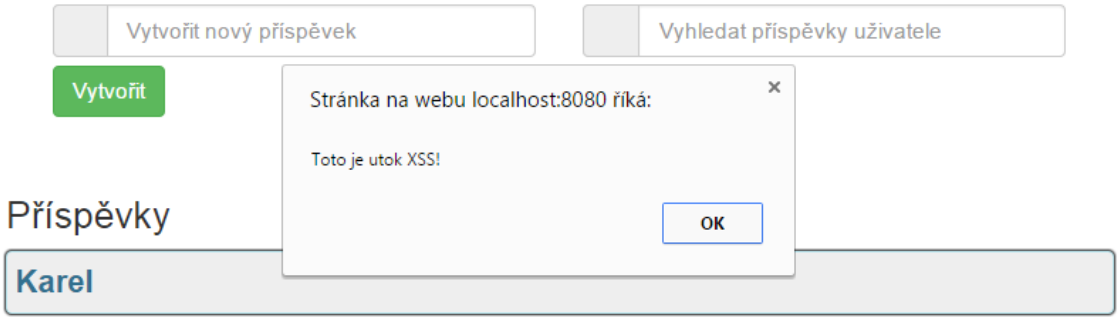
Obr. 9: Vkládání útočného skriptu do vstupního pole aplikace, zdroj: autor

Po odeslání formuláře s tímto skriptem se vložený řetězec uloží na databázový server do tabulky *prispevky*. Při následném obnovení hlavní stránky dojde k načtení útočného skriptu z databáze a ten je spolu s ostatními příspěvky zakomponován do obsahu stránky JSTL cyklem foreach [Kód 25].

```
<c:forEach items="${prispevky}" var="prispevek">
   <div class="alert alert-info">
      <b>${prispevek.uzivatel.login}</b> </br>
      <p>${prispevek.text}</p>
   </div>
</c:forEach>
```

Kód 25: JSTL cyklus pro výpis příspěvku aplikace

Načítaná stránka aplikace se tak začne zobrazovat a ve chvíli, kdy dojde k načtení příspěvku s injektovaným skriptem, se zobrazí vyskakovací okno [Obr. 10]. To se následně zobrazuje při každém novém požadavku na tuto stránku. Důvodem tohoto chování je uložení skriptu na databázovém serveru v tabulce *prispevky*. Jedná se tedy o perzistentní XSS.



Obr. 10: Ukázka aplikace napadené perzistentním XSS, zdroj: autor

Stejného výsledku (zobrazení vyskakovacího okna) je možné docílit taktéž vložením útočného skriptu do vstupního pole pro jméno uživatele formuláře pro filtrování příspěvků [Kód 26].

```
<form action="<c:url value='/home'/>" method="GET">
   <input type="text" name="filterUsername" value=""
      placeholder="Vyhledat příspěvky uživatele">
   <button type="submit" >Hledat</button>
</form>
```

Kód 26: HTML formulář aplikace II

V takovém případě aplikace nenalezne žádného uživatele a informuje o tom prostřednictvím výpisu: „Hledaný uživatel „“ nenalezen!“. To je v kódu stránky realizováno pomocí podmínky [Kód 27].

```
<c:if test="${param.status == 'filterError2'}">
   Hledaný uživatel "${param.username}" nenalezen!
</c:if>
```

Kód 27: JSTL podmínka aplikace

Protože hodnotou parametru *username* je vložený skript, namísto běžného výpisu hledaného uživatele dojde ke spuštění tohoto skriptu a zobrazení vyskakovacího okna podobně, jako v případě načítání příspěvku uvedeného výše. Tento způsob vložení útočného skriptu je však odlišný tím, že není trvale uložen do databáze a při obnovení stránky již je vše v pořádku. Jedná se tedy o non-perzistentní XSS.

4.4.2.2 Testování SQL Injection

Pro ukázku zranitelnosti SQL Injection je využito jednoho z míst, na které upozornil automatický skener, konkrétně funkce přihlášení uživatele. K tomu je potřeba zadat správné jméno a heslo. Pokud se však využije jedna z metod útoku SQL Injection, popsaná v kapitole 1.2, tedy namísto běžně vyplněných přihlašovacích údajů se dosadí připravený řetězec „Karel'; DROP TABLE prispevky --", docílí se zneužití zranitelnosti SQL Injection.

Tuto zranitelnost způsobuje neošetřená komunikace mezi aplikačním a databázovým serverem při vyhledávání uživatele. Komunikace je zobrazena ve výpisu kódu [Kód 28].

```
Statement st = connection.createStatement();
String query = "select id, login, password from uzivatel
where login = '" + username + "'";
ResultSet rs = st.executeQuery(query);
if (rs.next()) {
   User user = new User(rs.getInt(1), rs.getString(2),
   rs.getString(3));
   return user;
}
```

Kód 28: Neošetřená komunikace s databázovým serverem aplikace

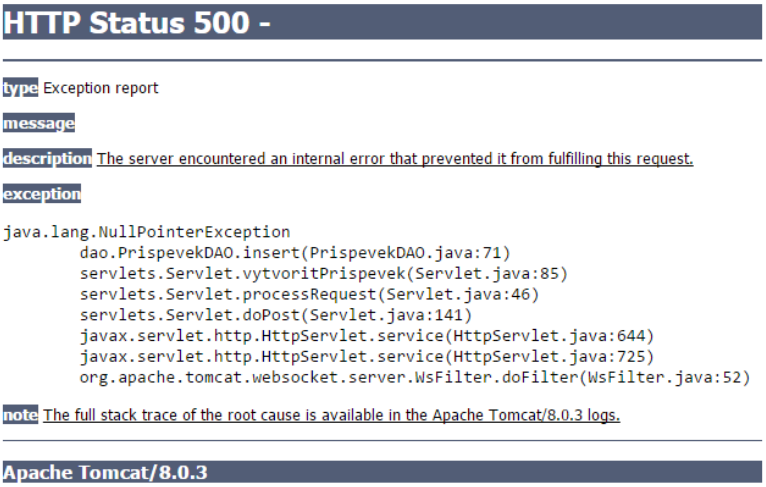
Po dosazení vloženého řetězce do SQL dotazu dojde ke spojení dvou dotazů a k následnému smazání tabulky *prispevky.* Po přihlášení tedy nedojde k žádnému zobrazení příspěvků.

4.4.2.3 Testování Information Leakage

Zranitelnost Information Leakage byla v kapitole 4.4.1 zjištěna v místě vytváření nového příspěvku. K tomu může dojít ve chvíli, kdy uživatel vytváří příspěvek a není při tom přihlášen. Tato situace může nastat dvěma způsoby:

- Po přihlášení do aplikace a delší nečinnosti (30 minut) dojde k automatickému odhlášení (smazání uživatele ze session).
- Po zadání URL adresy stránky s příspěvky bez předchozího přihlášení.

Po zadání příkazu pro vytvoření nového příspěvku bez předchozího přihlášení se zobrazí následující chybová stránka:



Obr. 11: Ukázka špatné konfigurace aplikace, zdroj: autor

Tato stránka obsahuje výpis chyby obsahující informace o konfiguraci serveru. Tyto údaje má možnost potenciální útočník využít k lepšímu pochopení fungování aplikace.

4.5 Implementace ochranných opatření

Předchozí kapitola byla věnována zkoumání a testování vytvořené aplikace a hledání jednotlivých závažných zranitelností. Pro přehled je uveden seznam potřebných bezpečnostních opatření, které je potřeba vyřešit:

- Šifrování hesel při přihlášení a registraci uživatele.
- Zamezení odposlechu cookies použitím bezpečných vlajek.
- Ošetření zranitelnosti Cross Site Scripting při vytváření nového příspěvku a filtrování příspěvků.
- Zabránění SQL Injection v případě přihlášení, registrace uživatele a filtrování příspěvků.
- Zamezení úniku informací při vytváření příspěvku bez přihlášení.

Na všechny tyto výskyty byla postupně aplikována konkrétní ochranná opatření za účelem zabezpečení aplikace. Dále jsou uvedeny způsoby, kterými toho bylo dosaženo.

4.5.1 Šifrování hesel při komunikaci

Aby se zabránilo potenciálnímu odposlouchání a prozrazení hesel, bylo nutné nastavit, aby komunikace mezi úvodní stránkou, která obsahuje přihlašovací a registrační formulář, byla realizována prostřednictvím bezpečného protokolu HTTPS. Toho bylo docíleno pomocí vytvoření nového bezpečnostního omezení v konfiguračním souboru *web.xml [Kód 29]*.

```
<security-constraint>
    <web-resource-collection>
        <web-resource-name>Bezpečná stránka</web-resource-
          name>
        <url-pattern>/prihlasit</url-pattern>
        <url-pattern>/registrace</url-pattern>
    </web-resource-collection>
    <user-data-constraint>
        <transport-guarantee>CONFIDENTIAL</transport-
          guarantee>
    </user-data-constraint>
</security-constraint>
```

Kód 29: Vytvoření bezpečnostího omezení v souboru web.xml

Toto omezení zaručuje, že při každém příchozím požadavku na server, který obsahuje v adrese URL cestu */prihlasit* nebo */registrace*, bude použit protokol HTTPS. V tom případě dojde k znemožnění prozrazení hesel při přihlašování a při registraci z důvodu jejich zašifrování.

4.5.2 Využití bezpečných vlajek

Zranitelnost Session Management, spočívající v možném prozrazení hodnot cookies (Session ID a jiné), je vyřešena nastavením bezpečných vlajek. Jediné, co k tomu bylo potřeba, bylo nastavení konfigurace cookies v souboru *web.xml*, přidáním následujícího kódu.

```
<session-config>
 <cookie-config>
      <secure>true</secure>
  </cookie-config>
</session-config>
```

Kód 30: Nastavení bezpečných vlajek

Nastavením bezpečných vlajek se zaručuje, že hodnoty cookies budou při komunikaci mezi serverem a klientem zašifrovány. Stejného výsledku by bylo možné dosáhnout i využitím bezpečného protokolu HTTPS, který šifruje veškerá přenášená data. Při šifrování souborů cookies je však upřednostňován postup manuální konfigurace z důvodu delší latence při použití protokolu HTTPS. [13]

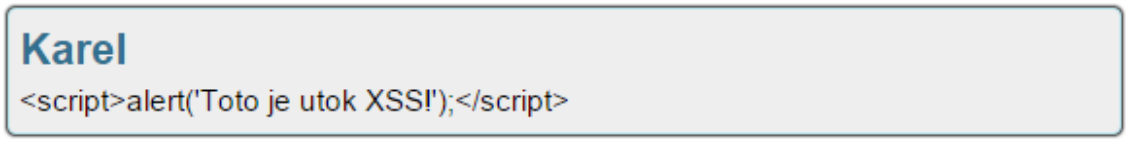
4.5.3 Ošetření zranitelnosti XSS

Zranitelnost XSS se v aplikaci vyskytovala u vkládání příspěvku a při jejich neúspěšném filtrování. Tato místa byla ošetřena zákazem veškerého HTML při výstupu aplikace. K tomu byla využita speciální funkce *fn:escapeXml()*, kterou nabízí sada Expression Language [Kód 31].

```
<c:forEach items="${prispevky}" var="prispevek">
   <div class="alert alert-info">
      <b>${prispevek.uzivatel.login}</b></br>
      <p>${fn:escapeXml(prispevek.text)}</p>
   </div>
</c:forEach>
```

Kód 31: Ošetřený výpis cyklu foreach

Při takto realizovaném výpisu příspěvků již nedochází ke spuštění uloženého skriptu v databázi. Ten se pouze vypíše jako každý jiný příspěvek, jak je ukázáno na obrázku [Obr. 12].



Obr. 12: Ukázka ošetřeného výstupu zobrazení příspěvku, zdroj: autor

Stejného způsobu zabezpečení výstupu bylo aplikováno i na výpis zprávy o neúspěšném filtrování příspěvků.

4.5.4 Zabránění SQL Injection

K zabránění útokům SQL Injection byla využita parametrizace SQL dotazů. Tato metoda ochrany byla detailně popsána a demonstrována v kapitole 2.2.1. Tímto způsobem byla ochráněna databáze v případě přihlášení uživatele, registrace no-

vého uživatele i filtrování příspěvků podle jména uživatele, kde byla aplikace náchylná na útok SQL Injection.

4.5.5 Zamezení úniku informací

V aplikaci docházelo k úniku citlivých informací z důvodu špatné konfigurace serverové části, která umožňovala vytváření příspěvků ve chvíli, kdy nebyl přihlášen žádný uživatel. Tento bezpečnostní nedostatek byl vyřešen vytvořením filtru v podobě třídy UserCheckFilter, který implementuje rozhraní Filter balíčku javax.servlet, a umožňuje ověřit, zda je v případě požadavku na zobrazení hlavní stránky aplikace uživatel přihlášen. Pokud tomu tak není, je automaticky přesměrován na stránku, ze které byl požadavek přijat. Filtr UserCheckFilter obsahuje následující kód:

```
private String contextPath;

@Override
public void init(FilterConfig fc) throws ServletException {
        contextPath=fc.getServletContext().getContextPath();
}

@Override
public void doFilter(ServletRequest request, ServletResponse
   response, FilterChain fc) throws IOException,
   ServletException {
   HttpServletRequest req = (HttpServletRequest) request;
   HttpServletResponse res = (HttpServletResponse) response;
   if (req.getSession().getAttribute("user") == null) {
      res.sendRedirect(contextPath);
   }else{
      fc.doFilter(request, response);
   }
}
```

Kód 32: Bezpečnostní filtr aplikace

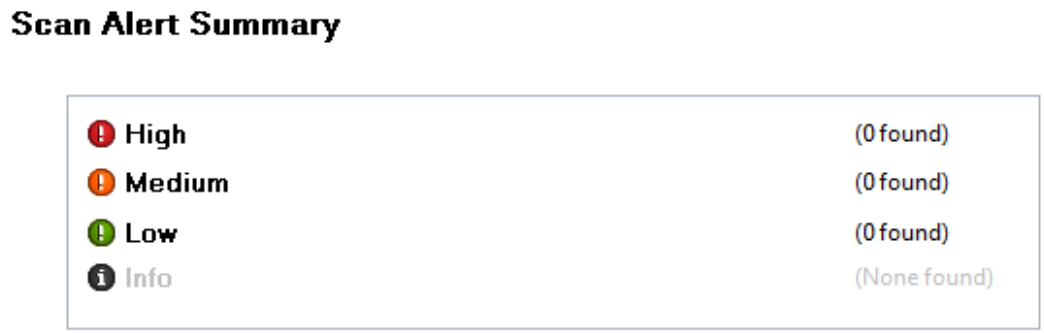
Tento filtr zachytává veškeré požadavky, které směřují na cílovou stránku s příspěvky. Jediné, co je zapotřebí obstarat, je namapování filtru na příslušný servlet, který takové požadavky zpracovává. To se provádí v souboru *web.xml*, ve kterém se daný filtr registruje a přiřadí určité adrese. Mapování filtru ukazuje kód [Kód 33].

```
<filter>
   <filter-name>UserCheckFilter</filter-name>
   <filter-class>UserCheckFilter</filter-class>
</filter>
<filter-mapping>
   <filter-name>UserCheckFilter</filter-name>
   <url-pattern>/home</url-pattern>
</filter-mapping>
```

Kód 33: Registrace a mapování bezpečnostního filtru

4.6 Závěrečné testování

Po implementaci všech ochranných opatření byla aplikace podrobena stejným testům, jako její původní, nezabezpečená verze. Použitím automatického skeneru Vega bylo dosaženo vygenerování pozitivní zprávy [Obr. 13], která již neobsahuje žádné hlášení o výskytech závažných zranitelností.



Obr. 13: Závěrečný report skeneru Vega, zdroj: autor

Manuální testování taktéž potvrdilo účinnost použitého zabezpečení a žádné náznaky zranitelností již nebyly zaznamenány. Závěrem je tedy možné konstatovat, že se testovací aplikaci podařilo zabezpečit proti všem zranitelnostem popsaných v této práci.

5 Výsledky a doporučení

V rámci kapitoly 4 jsem vytvořil webovou aplikaci v podobě diskuzního fóra. Tu jsem založil na platformě Java EE, která je součástí klasické Javy. Tuto aplikaci jsem záměrně vytvářel bez implementace jakýchkoli ochranných opatření a její cílem bylo identifikovat nejčastější a nejzávažnější zranitelnosti, které webové aplikace obsahují a demonstrovat způsoby, kterými je možné takovou aplikaci zabezpečit.

K identifikaci zranitelností jsem vedle manuálního testování využil volně dostupný automatický skener Vega od společnosti Subgraph. Tento nástroj bych doporučil zejména pro provádění základních testů zabezpečení webových aplikací. Navzdory tomu, že je volně dostupný, nabízí přehledné grafické uživatelské rozhraní, spolehlivou detekci nejzávažnějších zranitelností a kvalitně zpracovanou dokumentaci.

Pomocí skeneru se mi podařilo odhalit tyto zranitelnosti:

- Posílání hesel ve formě prostého text skrze HTTP
- Soubory cookies bez bezpečných vlajek (Session management)
- Cross-Site Scripting a SQL Injection
- Únik citlivých informací

Abych zabránil potenciálním útokům na aplikaci, musel jsem implementovat tato ochranná opatření:

- Využít bezpečného protokolu HTTPS při posílání hesel mezi klientem a serverem.
- Šifrovat hodnoty cookies při komunikaci pomocí bezpečných vlajek.
- Zakázat veškeré HTML při výstupu aplikace.
- Využít parametrizované dotazy při komunikaci s databází.
- Filtrovat požadavky na server.

Z důvodu, že všechna provedená opatření byla účinná, je mohu doporučit programátorům webových aplikací jako základní metody, kterými je možné jejich aplikaci zabezpečit před nejčastějšími útoky.

6 Závěr

Cílem této práce bylo představit a analyzovat nejčastější a nejzávažnější zranitelnosti webových aplikací a útoky, které těchto zranitelností využívají. Práce se taktéž věnovala nástrojům, pomocí nichž je možné tyto zranitelnosti identifikovat a způsobům zabezpečení webových aplikací.

Zranitelností webových aplikací je velké množství. Takových aplikací, které by byly chráněny proti všem možným typům útoků, je pouze malé procento. Mnohé z nich naopak obsahují celou řadu bezpečnostních nedostatků. Důvodem je zejména neznalost vývojářů o této problematice a v mnohých případech také podceňování možných útoků.

Nejrozšířenější zranitelností je Cross-Site Scripting, která se vyznačuje vkládáním skriptů do neošetřených vstupů aplikace. Útoky, s ní spojené, mají mnoho podob, přičemž mohou vést až k získání plné kontroly nad webovým prohlížečem nic netušícího uživatele zranitelné webové aplikace. Způsoby, kterými je možné aplikaci proti Cross-Site Scripting zabezpečit jsou bezpečnostní politika Content Security Policy a zakázání znaků HTML na výstupu aplikace.

Další zranitelností je SQL Injection. Ta se vyznačuje možností změnit dotaz na databázi aplikace. Tato zranitelnost je podobně jako u XSS způsobena vstupy formulářů webové aplikace, do kterých však útočník namísto nebezpečných skriptů vkládá SQL dotazy, které pak aplikace odesílá na databázový server a může tam napáchat značné škody. Této zranitelnosti se dá zabránit použitím parametrizací SQL dotazů, escapováním vstupních dat nebo uloženými procedurami.

Únik informací je zranitelnost webových aplikací spočívající v odhalování citlivých dat jako například různé technické detaily aplikace, její prostředí nebo uživatelská data. Bývá způsobena zejména zapomenutými komentáři, nesprávnou konfigurací serveru a z důvodu neošetření neplatných hodnot vstupů.

Poslední zranitelností, které jsem se v této práci věnoval, byla zranitelnost způsobená špatnou správou relací. Útok proti ní je nazýván fixace relace a spočívá v krádeži cizího uživatelského účtu. Obranou proti tomu je vytváření nových session ID při každém novém požadavku na aplikaci.

V této práci jsem se věnoval také způsobům, jak jednotlivé zranitelnosti v aplikaci identifikovat – automatickým skenerům. Ty umožňují poměrně spolehlivě zjistit, které zranitelnosti a v jaké míře jsou v aplikaci obsaženy. Zajímavým zjištěním v tomto případě bylo, že některé volně dostupné nástroje svojí funkčností příliš nezaostávaly za těmi drahými a skenování vlastní aplikace tak nemusí znamenat žádný výdaj.

Jako součást této práce jsem vytvořil testovací webovou aplikaci, ve které jsem identifikoval několik zranitelností pomocí vybraného automatického nástroje a potvrdil jsem účinnost některých způsobů zabezpečení.

Ačkoli zranitelnosti zmíněné v této práci jsou ty nejzávažnější, nejedná se ani zdaleka všechny slabiny dnešních webových aplikací, z kterých hrozí vážná bezpečnostní rizika. Na mnohé z nich se daří nacházet účinná ochranná opatření, avšak s neustálým vývojem technologií v oblasti webových aplikací se budou stále objevovat nové zranitelnosti, kterým bude nutné věnovat pozornost při vytváření webové aplikace.

7 Použité zdroje

Literární zdroje:

[1] GROSSMAN, Jeremiah. XSS attacks: cross site scripting exploits and defense. Burlington: Syngress, 2007, xiv, 448 s. ISBN 15-974-9154-3.

[2] Kümmel, Roman. Cross-Site Scripting v praxi: o reálných zranitelnostech ve virtuálním světě. 1. vyd. Zlín : Tigris spol. s r o., 2011. 331 str. ISBN 978-80-86062-34-1.

[3] PINTO, Marcus a STUTTARD Dafydd. The Web Application Hacker's Handbook: Finding and Exploiting Security Flaws. 2. vyd. Indianapolis: John Wiley & Sons, Inc.,2011. 853 str. ISBN 978-1-118-02647-2.

Internetové zdroje:

[4] AVYAAN. Session Management Vulnerability in Web Application Part 1. *Avyaan.com* [online]. 2015 [cit. 2015-04-15]. Dostupné z: http://www.avyaan.com/blog/session-management-vulnerability-web-application-part-1/

[5] BELANS, Joseph. Common Web Application Vulnerabilities - Part 8. *Fishnetsecurity.com* [online]. 2014 [cit. 2015-04-15]. Dostupné z: https://www.fishnetsecurity.com/6labs/blog/common-web-application-vulnerabilities-part-8

[6] DALTON, Michael, Christos KOZYRAKIS a Nickolai ZELDOVICH. *Nemesis: Preventing Authentication & Access Control Vulnerabilities in Web Applications* [online]. 2010[cit. 2015-04-14]. Dostupné z: http://people.csail.mit.edu/nickolai/papers/dalton-nemesis.pdf

[7] EFYTIMES. 10 Powerful SQL Injection Tools That Penetration Testers Can Use. *EFYTimes.com* [online]. 2014 [cit. 2015-04-15]. Dostupné z: http://www.efytimes.com/e1/fullnews.asp?edid=132535

[8] GADGIL, Sampada, Sanoop PILLAI a Sushant POOJARY. SQL INJECTION ATTACKS AND PREVENTION TECHNIQUES. *Academia.edu* [online]. 2013 [cit. 2015-04-15]. Dostupné z: http://www.academia.edu/3713854/SQL_INJECTION_ATTACKS_AND_PREVENTION_TECHNIQUES

[9] INFOSECPRO. Information Leakage. *InfoSecPro* [online]. 2013 [cit. 2015-04-15]. Dostupné z: http://www.infosecpro.com/applicationsecurity/a52.htm

[10] Internetlivestats.com Internet live stats: Internet Users. Internet Live Stats [online]. 2014 [cit. 2014-08-14]. Dostupné z: http://www.internetlivestats.com/internet-users/

[11] NETSPARKER LTD. What you need to know about SQL Injection Web Application Vulnerability. *Netsparker* [online]. 2015 [cit. 2015-04-14]. Dostupné z: https://www.netsparker.com/web-vulnerability-scanner/vulnerability-security-checks-index/sql-injection/

[12] *Open Web Application Security Project*. Preventing SQL Injection in Java. Owasp.com [online]. 2008 [cit. 2015-04-15]. Dostupné z: https://www.owasp.org/index.php/Preventing_SQL_Injection_in_Java

[13] *Open Web Application Security Project*. SecureFlag. *Owasp.com* [online]. 2014 [cit. 2015-04-15]. Dostupné z: https://www.owasp.org/index.php/SecureFlag

[14] SAJJADI, Sayyed Mohammad Sadegh a POUR, Bahare Tajalli. Study of SQL Injection Attacks and Countermeasures. In: *International Journal of Computer and Communication Engineering* [online]. 2013 [cit. 2015-04-14]. DOI: 10.7763/IJCCE.2013.V2.244. Dostupné z: http://www.ijcce.org/papers/244-E091.pdf

[15] RAUSHAN, Ritesh. Web application security for java. *Slideshare* [online]. 2014 [cit. 2015-04-15]. Dostupné z: http://www.slideshare.net/RiteshRaushan2/web-application-security-39337749

[16] TRUSTWAVE HOLDINGS, INC. Application Vulnerability Trends Report : 2014. *Trustwave: Smart Security On Demand* [online]. 2014 [cit. 2015-04-15]. Dostupné z: https://www.trustwave.com/Resources/Library/Documents/Cenzic-Application-Vulnerability-Trends-2014/

[17] WEBAPPSEC. Information Leakage. *The Web Application Security Consortium* [online]. 2009 [cit. 2015-04-14]. Dostupné z: http://projects.webappsec.org/w/page/13246936/Information%20Leakage

Seznam obrázků

Seznam tabulek

Seznam zdrojových kódů

Přílohy

Vytvořená testovací webová aplikace je veřejně přístupná na serveru GitHub, kde je možné nahlédnout do zdrojových kódů.

https://github.com/TheHumr/BP

UNIVERZITA HRADEC KRÁLOVÉ
Fakulta informatiky a managementu
Rokitanského 62, 500 03 Hradec Králové, tel: 493 331 111, fax: 493 332 235

Zadání k závěrečné práci

Jméno a příjmení studenta: **Ondřej Boura**

Obor studia: Informační management (3)
Jméno a příjmení vedoucího práce: **Monika Borkovcová**

Název práce:
Zabezpečení webových aplikací

Název práce v AJ:
Web Application Security

Podtitul práce:

Podtitul práce v AJ:

Cíl práce: Účelem této práce je poskytnout lidem zabývajícím se zabezpečením webových aplikací přehled možných útoků na webové aplikace a přehled možností, jak zabezpečení aplikace realizovat.

Osnova práce:
1. Úvod
2. Typy útoků
3. Způsoby zabezpečení
4. Shrnutí
5. Závěr
6. Seznam použitých zdrojů

Projednáno dne: 16. 10. 2014

Podpis studenta

Podpis vedoucího práce