Univerzita Hradec Králové

Fakulta informatiky a managementu

Katedra informačních technologií

Bezpečnost webových aplikací v PHP

Diplomová práce

Autor: Tomáš Janeček
Studijní obor: Aplikovaná informatika

Vedoucí práce: Ing. Zuzana Němcová, Ph.D.

Hradec Králové duben 2018

Prohlášení:

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

V Hradci Králové dne

18. 04. 2018 Tomáš Janeček

Poděkování:

Děkuji vedoucí diplomové práce Ing. Zuzaně Němcové, Ph.D. za příkladné a metodické vedení práce i významnou pomoc při jejím zpracování.

Anotace

Diplomová práce se zaměřuje na konkrétní způsoby řešení zabezpečení webových aplikací napsaných v jazyce PHP. Mezi server-side jazyky webových aplikací PHP přesvědčivě dominuje. Aplikace na webu nabývají stále většího významu. Zajišťují online nakupování, bankovnictví i další služby a zároveň hladký běh řady organizací, jejichž provoz závisí na online aplikacích. Není pochyb o tom, že je velmi důležité dbát na zabezpečení aplikací a dat v nich uložených. Práce obsahuje popis nejdůležitějších bezpečnostních rizik aplikací, rozdělení hackerů a základy penetračního testování. Hlavním přínosem práce je ale praktické rozvedení nejčastějších a nejnebezpečnějších typů útoků včetně praktických příkladů, možností jejich použití a návrhu efektivního řešení proti každému z nich s inspirací v aktuálních verzích populárních PHP frameworků Nette a Laravel. Součástí práce je i webová aplikace, ve které se názorně uplatňují navržené techniky zabezpečení. Na závěr práce je navržena strategie jejího testování a aplikace je podle ní podrobena úspěšnému testu zabezpečení.

Annotation

Title: PHP Web Application Security

This thesis is focused on specific means of assuring PHP web application security. PHP is tremendously dominant among web application server-side languages. Web applications keep growing in importance. They provide online shopping, banking and other services and ensure smooth operation of many organisations. There is absolutely no doubt that it is important to be mindful of web application and data security. The thesis contains an overview of most critical web application security risks, a classification of hackers and penetration testing basics. The main contribution of this work are extensive examples and proposed solutions to the most critical web application security risks. The solutions were inspired by the source code of Nette and Laravel PHP frameworks. To apply the solutions in practice, a web application using the solutions has been made as a part of this work. Furthermore, a security testing strategy for this work was designed. The strategy was put into practice and the application has been successfully tested.

Obsah

1 Úvod 1

2 Cíl a struktura práce 2

3 Bezpečnostní rizika aplikací 3

3.1 Autentizační útoky 3

3.2 SQL injection 5

3.3 Cross-site scripting (XSS) 5

3.4 Cross-site request forgery (CSRF) 6

3.5 File inclusion 7

3.6 Server-side request forgery 7

3.7 Denial of Service 8

3.8 Apache útoky 8

3.9 Další útoky 8

4 Testování zabezpečení aplikací 9

4.1 Pojmy a rozdělení 9

4.2 Metodiky testování zabezpečení 15

4.3 Nástroje k penetračnímu testování 19

5 Obrana proti hrozbám v PHP 20

5.1 Slabiny jazyka PHP 21

5.2 Autentizační útoky 25

5.3 Databázové a SQL injection útoky 34

5.4 Cross-site scripting (XSS) 46

5.5 Cross-site request forgery (CSRF, XSRF) 63

5.6 Chyby v řízení přístupu (Broken access control) 70

5.7 File inclusion 73

5.8 Path injection, command injection, code injection 75
5.9 Public files 78
5.10 Nahrávání souborů 80
5.11 Denial of service (DoS) 82
5.12 Chyby konfigurace 84
5.13 Ostatní 86
6 Testování zabezpečení webové aplikace 87
6.1 Strategie testování aplikace 87
6.2 Testování aplikace 88
7 Závěry a doporučení 94
8 Seznam použité literatury 95
9 Seznam obrázků 98
10 Seznam zdrojových kódů 99

1 Úvod

Téma bezpečnosti webových aplikací je rozsáhlé a především velmi důležité. Miliardy lidí každý den používají internet a webové aplikace a to nejenom ke čtení aktuálních zpráv, ale i k nákupům nebo zadávání platebních příkazů. S rozmachem sociálních sítí je na internetu více a více lidí, informace jsou sdíleny ve stále větším měřítku. Web je dále začleňován do každodenního života. Celá řada komerčních společností má business model závislý na internetu a webových aplikacích a bez nich nemůže generovat zisk.

Všudypřítomnost webových aplikací a další výše popsané důvody činí z webových aplikací, serverů, operačních systémů i celých sítí, které se na provozu webových aplikací podílí, populární terč útoků. Aplikace na celém světě jsou neustále vystaveny riziku prolomení vlastní bezpečnosti i dat v nich uložených.

V důsledku toho moderní webový vývoj přináší celou řadu výzev. Jednou z těch nejdůležitějších, přitom velice často podceňovaných, je dbát na bezpečnost webových aplikací už při jejich vývoji. Vývojem ale kapitola bezpečnosti nekončí. Neméně důležité je pak i pravidelné testování zabezpečení těchto aplikací a následné aktualizace a záplaty.

První knihy o hackování a penetračním testování byly vydány už před více než patnácti lety. Zároveň s jejich vydáním vyvstala kontroverzní otázka, zda je dobré psát něco, co zlepšuje znalosti útočníků, učí je nové techniky a dává za vznik dalším potenciálním útokům. Je ale jasné, že bezpečnost nesmí nikdy být postavena na utajování potenciálních slabin. Systém by se měl vždy kontinuálně zdokonalovat a přizpůsobovat případné nové situaci. Poskytnutí znalostí slouží totiž také k verifikaci, že zabezpečení aplikace je správné a kompletní.

K naplnění cílů práce popsaných v následující kapitole posloužila výhradně zahraniční literatura a webové zdroje. Z těch hlavních je vhodné jmenovat knihu Rafaye Balocha o etickém hackování a penetračním testování (2015), žebříček deseti největších rizik webových aplikací organizace OWASP (2017), dokumentaci PHP Group (2018) nebo sérii článků o bezpečnosti společnosti Acunetix (2018).

2 Cíl a struktura práce

Práce si dává za cíl poskytnout přehled nejčastějších technik prolomení zabezpečení webových aplikací, pokrýt typické způsoby penetračního testování těchto aplikací a po vzoru populárních PHP frameworků pak navrhnout optimální kód zajišťující zabezpečení aplikací napsaných v PHP vůči rozličným typům útoků. Dalším cílem je vytvořit důkladně zabezpečenou PHP aplikaci s využitím navržených způsobů zabezpečení. Pro verifikaci, že je vytvořená aplikace zabezpečená odpovídajícím způsobem, je sekundárním cílem práce navrhnout strategii otestování aplikace z bezpečnostního hlediska a při jejím testování strategii aplikovat.

Nejprve se v práci kapitola 3 zabývá základním teoretickým popisem konkrétních bezpečnostních rizik aplikací napsaných v jazyce PHP. Kapitola 4 popisuje rozdělení hackerů a způsoby testování zabezpečení webových aplikací, tzv. penetrační testování. Práce následně v kapitole 5 prakticky rozvádí nejrizikovější typy útoků na PHP webové aplikace, možnosti jejich použití a navrhuje efektivní řešení obrany proti každému z nich. Na závěr je představena webová aplikace, součást této práce, která vznikla pro názorné uplatnění navržených technik zabezpečení. Zabývá se jí kapitola 6. Ta dále obsahuje návrh strategie testování této aplikace a jsou představeny výsledky jejího testování. Závěry práce a další doporučení jsou shrnuty v kapitole 7.

3 Bezpečnostní rizika aplikací

Webové aplikace jsou místem, kde se v dnešní době odehrává většina útoků, píše Baloch (2015). Právě proto je web hacking jedním z hlavních témat této práce. Práce se dále zabývá prolomením autentizace, SQL injection, Cross-Site Scripting, Cross-Site Request Forgery, chybami v řízení přístupu, File Inclusion a DoS útoky, útoky na nahrávání souborů, public files zranitelnostmi a chybami v konfiguraci.

3.1 Autentizační útoky

Autentizace chrání soukromá či chráněná data, proto se k nim nesmí být možné dostat bez příslušných údajů. Penetrační testování jde hlouběji. Do takové míry, že namísto obcházení autentizace, což v drtivé případě není dost dobře možné, zajistí přihlašovací údaje a tím získá volný nenásilný přístup k soukromé části aplikace a datům.

3.1.1 Způsoby prolomení autentizací a obrana

Google již několik let ve svém page ranku resp. výsledcích vyhledávání bere v potaz, zda hodnocený web běží pod HTTP nebo HTTPS protokolem. Webům bez SSL page rank snižuje. Při přihlašování přes HTTP protokol lze totiž velmi snadno z té stejné sítě přihlašovací údaje včetně hesla zachytit pomocí „*man-in-the-middle*“ útoku.

Typickou slabinou autentizace jsou výchozí hesla. Hesla typu admin a 123456 nejsou už téměř dvě desetiletí ani považována za hesla vzhledem k jejich frekvenci. Ponechat výchozí heslo v zařízení nebo aplikaci tak často znamená nulové zabezpečení. Podobné platí i pro slabá krátká hesla, která lze uhodnout pomocí brute force („násilné“ zkoušení ohromného množství různých kombinací znaků) nebo slovníkových útoků.

Méně častým prohřeškem jsou pak nedokonale zpracované funkčnosti typu „zapomněl jsem heslo“, které v odstrašujících případech dovolí útočníkovi vygenerovat nové heslo pro účet, který se snaží napadnout. Některé přihlašovací formuláře útočníkovi dokonce konkrétně řeknou, zda špatně vyplnil heslo nebo uživatelské jméno.

Dalším prohřeškem je ukládání hesel na lokální úložiště, odkud mohou být hesla při využití vhodného útoku a nedostatečném zabezpečení extrahována. Z hlediska uživatele je neméně závažným pochybením ukládání hesel do předvyplnění v prohlížeči. V případě, že by se útočník dostal do takového počítače, bylo by pro něj, podle Balocha (2015), velmi snadné zjistit potřebná hesla.

Kim (2014) píše, že řada formulářů se dá obejít i pomocí SQL injection. K tomu stačí uhodnout nebo jinak zjistit uživatelské jméno a jako heslo napsat řetězec ' OR '1'='1. U špatně napsaného dotazu takové zadání projde kontrolou, protože podmínka, která kontroluje, že jde o správné heslo, je rozšířena o podmínku *1=1*. Heslo správné není, ale rovnost *1=1* samozřejmě platí vždy a útočník tak je vpuštěn do systému. V případě, že webová aplikace využívá jako úložiště XML soubory, lze za stejným účelem použít XPATH injection útok.

Nezřídka se daří autentizaci prolomit i zcizením dat cookie pro kontrolu přihlášení. Této problematice se věnuje kapitola 5.2.1. Také se lze zmocnit session ID pomocí tzv. session hijacking útoku, který je podrobně popsán v kapitole 5.2.2.

Obrana proti každé z těchto slabin je nasnadě. Existují mechanismy, které dokáží zabránit brute force útokům tím, že uživateli dají např. pouze pět pokusů na přihlášení za minutu. Pokud heslo splete pětkrát, účet je pro danou IP adresu na minutu zamknutý, uživatel tak musí vyčkat do další minuty, kdy má k dispozici nových 5 pokusů. Tím je zabráněno hromadnému zkoušení hesel roboty. Ti buď generují všechna možná hesla dané maximální délky z dané sady znaků (brute force útok), nebo zkouší hesla z databáze častých hesel – často i v řádech tisíců vyzkoušených hesel za minutu. Druhým řešením obrany je *CAPTCHA* kód, který je nutné opsat. Přes kód se dá ale opět dostat. A pro prevenci SQL/XPATH injection je třeba dbát na vhodný způsob psaní kódu při vytváření aplikace. Detailně je tato problematika rozebrána v kapitole 5.3 (Baloch, 2015).

3.2 SQL injection

Tento druh útoku, podle Engebretsona (2013), spočívá ve využití chyby kódu aplikace, kdy vstup od uživatele není filtrovaný a tak se ve své ryzí podobě dostane do spouštěného kódu. Uživatel tak může místo očekávaného vstupu vepsat kus SQL dotazu, který doplněním upraví původní zamýšlený SQL dotaz. Tímto je útočníkovi umožněno podniknout jeden z několika různých druhů SQL útoku. V případě, že web není proti SQL injection zabezpečený, musí útočník vědět nebo zjistit, o jaký typ SQL databáze webu se jedná a odhadnout podobu vykonávaného SQL dotazu, aby mohl dotaz vhodně doplnit a slabiny využít.

Nejčastějším typem SQL injection útoku je tzv. union-based SQL injection. Útočník využije SQL operátor „union“, aby z databáze dostal jakékoliv informace, které mu měly zůstat skryty. Nejsnazší SQL injection technika je naopak error-based SQL injection, ve které útočník donutí aplikaci zahlásit chybu, se kterou se vypíše i obsah z databáze, o který útočníkovi šlo. Třetí a poslední skupina SQL injection technik je blind SQL injection. Při takovém typu útoku databáze prostřednictvím aplikace útočníkovi nezobrazuje žádné chybové hlášky, které by hack usnadnily. Je tak třeba slepě (proto blind SQL injection) dotazovat databázi *true/false* dotazy a na základě zkušeností zkoušet různé možnosti SQL injection, které mohou a nemusí fungovat, dokládá Baloch (2015).

3.3 Cross-site scripting (XSS)

Baloch (2015) o XSS píše jako o metodě využívající problém analogický k SQL injection, kdy jde opět o neošetřené vstupy z formulářů. Tentokrát se ale nevyužívá doplnění SQL kódu, ale typicky JavaScriptu nebo jiného client-side jazyka (jazyk, jehož kód běží prohlížeči uživatele a ne na serveru), který pak u ostatních uživatelů udělá, co útočník zamýšlí. Dělí se většinou na 3 hlavní skupiny, nepersistentní, persistentní a DOM-based XSS. Nejčastěji se využívá k odcizení cookie s přihlašovací session, čímž si útočník zajistí přístup do aplikace, aniž by znal heslo uživatele. Této problematice se maximálně věnují kapitoly 5.2.1 a 5.2.2.

Nejčastější formou cross-site scriptování je nepersistentní XSS. Vstup se přímo vrací uživateli, neukládá se do databáze. Není snadné tohoto útoku zneužít, protože je nezbytné, aby uživatel klikl na útočníkem připravený odkaz. Ten totiž zneužívá formulářového vstupu a následně query string parametru URL. Jedná se o proměnné, které jsou v adrese stránky/skriptu za otazníkem, v případě více query string parametrů jsou odděleny znaménkém ampersand „*&*“, např. „*http://adresa.cz?parametr=hodnota&druhy_parametr=hodnota2*“. V hodnotě tohoto parametru pak zpravidla bývá útočníkem připravený skript (Lockhart, Sturgeon, 2017).

Méně častou, ale o to více nebezpečnou formou XSS je persistentní cross-site scripting. Principově je podobný nepersistentnímu, rozdíl je však v tom, že vstup se neukládá do query string parametru URL, ale do databáze, odkud se později opět načte a další postup je již stejný. Vzhledem k tomu, jak snadné je takový útok provést, jde o nejnebezpečnější formu XSS, uvádí Baloch (2015).

Obdobou obou předchozích forem XSS, tedy persistentního i nepersistentního cross-site scriptingu, je DOM-based XSS. Hlavní rozdíl spočívá v tom, že se odehrává pouze na straně klienta. Vzhledem k rozšíření JavaScriptu jsou moderní aplikace plné asynchronních (AJAX) požadavků a vzniká tak mnoho potenciálních mezer v zabezpečení.

3.4 Cross-site request forgery (CSRF)

CSRF je velmi specifický typ útoku, kdy útočník donutí prohlížeč oběti udělat nežádoucí požadavek na server, čímž jménem oběti aktuálně přihlášené v dané webové aplikaci může změnit heslo jejího účtu, poslat někomu zprávu, odhlásit svou oběť apod. Tyto útoky jsou často úspěšné, protože servery běžně neověřují, zda požadavek na server přišel od opravdového uživatele, kontrolují pouze, zda přišel z prohlížeče oprávněného uživatele. Uživatel se tak přihlásí na svém oblíbeném webu a útočník jej nějakým způsobem vláká na stránku, kterou potřebuje. Z té stránky pak útočník přímo prostřednictvím prohlížeče uživatele odesílá HTTP požadavky na server a tak může udělat vše, co může dělat onen přihlášený uživatel (Baloch, 2015).

Útoky mohou být založené na HTTP požadavcích na server typu GET i POST a využívat skryté formuláře, vytvořené obrázkové značky nebo asynchronní požadavky na server, aby si uživatel podivného chování aplikace snadno nevšiml. Nejčastější efektivní obranou jsou CSRF tokeny (náhodné řetězc) vytvořené jen pro session přihlášeného uživatele.

Zabezpečení proti CSRF útokům je velmi důležité, zejména v online banking aplikacích jde o absolutní nevyhnutelnost kvůli citlivým informacím a řadě možných nevratných změn. Online banking je velmi specifické téma. Rizika prolomení bezpečnosti online banking aplikací spočívají v celé škále útoků (zejména XSS). To ale nesnižuje důležitost zabezpečení u jiných, nebankovních aplikací. Ve vztahu k CSRF je naprosto klíčové, aby útočník nemohl např. podat platební příkaz jménem oběti (Aljawarneh, 2017).

3.5 File inclusion

Zranitelnosti vkládáním souborů již na webu nejsou nijak časté, v moderních aplikacích téměř neexistují. PHP povoluje pomocí funkcí *include()*, *include_once()*, *require()* a *require_once()* dynamicky načítat libovolný (i vzdálený) soubor, k čemuž dále pomáhají metody jako *file()*, *fopen()* a *file_get_contents()*. Všechny tyto metody lze při nefiltrovaném vstupu zneužít, ale aplikací s mezerou v zabezpečení v této oblasti již existuje skutečně jen poskrovnu (Baloch, 2015, Acunetix, 2018).

3.6 Server-side request forgery

SSRF je celá třída útoků, které využívají zranitelností v podobě nebezpečného použití funkcí, které otevírají sockety (síťová spojení) a stahují data (obrázky, text, jiný obsah) z webových serverů (zkráceně web server). Je třeba, aby vstupní pole i odpovědi ze serveru byly ošetřeny. Pokud nejsou, útočník může využít jiného web serveru, aby se choval dle jeho pokynů při komunikaci s web serverem a obejít tak firewall, scanovat externí webservery, interní aplikace na web serveru, číst soubory na serveru, způsobit DoS útok nebo zneužít interní zranitelné aplikace (Baloch, 2015).

3.7 Denial of Service

Velmi častá metoda útoků, která spočívá v jednoduché myšlence přetížení serveru požadavky. Zejména u UNIXových systémů je snadné vyvolat požadavky na malé soubory jako */dev/random* nebo */dev/zero*, které server nikdy nevrátí, nebo naopak požadavky na nějaký velký soubor. Tím útočník vytíží prostředky serveru na plno a dojde tak k odepření služby web serveru. Nebude mít totiž volné prostředky na to, aby se vypořádal s dalšími požadavky (Baloch, 2015).

3.8 Apache útoky

Poslední kapitolou jsou útoky na apache servery. Samotný zdrojový kód Apache byl přezkoumán řadou různých bezpečnostních profesionálů a drtivá většina zranitelností tak byla již vyřešena. Apache servery ale mohou načítat externí moduly jako PHP a CGI, které při nesprávné konfiguraci mohou být úspěšně napadeny.

3.9 Další útoky

Mezi další útoky na webové aplikace se řadí

- chyby v řízení přístupu,
- path injection (directory traversal), command injection, code injection,
- nahrávání souborů,
- chyby konfigurace.

Podrobnější informace o těchto metodách útoků jsou k dispozici v kapitolách 5.6 až 5.15.

4 Testování zabezpečení aplikací

Žádná aplikace po dokončení vývoje programátory nebude zaručeně odolná vůči všem typům útoků. Pokud u dané aplikace na bezpečnosti skutečně záleží, je nutné její bezpečnost otestovat. Tím se zabývají white hat hackeři, o kterých více referuje kapitola 4.1.1. Provádí tzv. penetrační testování aplikace, jehož cílem je odhalit slabiny zabezpečení a následně provést aktualizaci kódu, konfigurace či procesů tak, aby byly všechny zranitelnosti odstraněny.

4.1 Pojmy a rozdělení

Penetrační testování či hackování aplikací jsou specifické činnosti, kterými se zabývají určité skupiny lidí. Před popisem samotných metodik penetračního testování je vhodné vymezit klíčové pojmy z této oblasti.

4.1.1 Rozdělení hackerů

Hacker, jinými slovy kreativní člověk, který dokáže díky svým znalostem a schopnostem vymyslet komplexní logický postup k prolomení zabezpečení software, je vlivem médií dnes vnímán neprávem zpravidla negativně. Tito lidé přitom posouvají hranice možností zabezpečení stále dále a dále a mají velkou zásluhu na podobě a možnostech dnešních technologií spojených s bezpečností.

Hackeři se mohou dělit do několika kategorií dle různých kritérií. Rozdělení dle etičnosti Baloch (2015) popisuje následovně:

- *White hat hacker* – profesionál na zabezpečení, jehož práce spočívá v hledání slabin zabezpečení softwaru. Viz dále *etický hacker*.
- *Black hat hacker / cracker* – člověk zneužívající svých znalostí a schopností ohledně zabezpečení pro negativní účely. Často médii mylně označován jako „hacker“.
- *Grey hat hacker* – někdo na pomezí white and black hat hackera. Takový člověk typicky pracuje jako profesionál přes zabezpečení, ale vědomě zanechává v zabezpečení slabiny, aby se sám mohl později k datům dostat a zneužít je.

Baloch (2015) dále představuje rozdělení hackerů podle míry znalostí a náplně jejich práce:

- *Script kiddie, skid* – někdo bez dostatečných znalostí principů zneužití slabin. Nedokáže vytvořit zranitelnost, kreativně upravit hack (tedy postup zneužití konkrétní slabiny) nebo najít jinou možnost v případě nefunkčního hacku, pouze dokáže využívat již existující slabiny systému, které někdo vytvořil.
- *Elitní hacker, 133t, 1337* – člověk s hlubokými znalostmi o principech zneužívání zranitelností. Dokáže slabiny cíleně vytvářet i upravovat cizí kód k umožnění prolomení zabezpečení.
- *Hacktivist* – lidé nebo skupiny, které společně pro dobrou věc prolamují zabezpečení systémů, aby rozšířili důležité informace nebo z jiných eticky dobrých důvodů. Často prolamují zabezpečení např. webových stránek politických stran.
- *Etický hacker* – člověk najatý organizací, kterému bylo uděleno povolení a zároveň přidělen úkol zaútočit na systémy dané organizace za účelem nalezení slabin zabezpečení, kterých by mohli zneužít nežádoucím způsobem black hat hackeři. Jediný rozdíl mezi hackerem a etickým hackerem je v povolení, které je etickému hackerovi uděleno.

4.1.2 Hackování

Zjednodušeně je možné říci, že hackování je prolomení zabezpečení počítače nebo systému někoho cizího. To znamená, že se útočník dostane k informacím uvnitř, může tato data zcizit, upravit, změnit některé konfigurace nebo i sabotovat systém jako takový (Sharpe, 2015).

Hackování začalo jako nevinná zábava počítačových profesionálů, dnes lze tuto aktivitu v národních měřítkách označit dokonce jako druh terorismu.

4.1.3 Penetrační testování

Penetrační testování, zkráceně jen **pentesting**, je důkladný proces vyhodnocení zabezpečení. Je považováno za poslední a nejagresivnější formou testování bezpečnosti. Musí být vykonáno vysoce kvalifikovanými profesionály a může být

vykonáváno jak s předchozí znalostí systému, tak bez ní. Vyhodnocení zabezpečení se může týkat celé IT infrastruktury – aplikací, síťových zařízení, operačních systémů, komunikačních médií, fyzické bezpečnosti, ale i psychologie člověka. Výsledkem je podrobný report rozdělený do sekcí zahrnující všechny nalezené zranitelnosti v aktuálním stavu cílového prostředí, opatření proti těmto slabinám a další doporučení (Ali, Allen, 2014; Long, Mitnick, 2008)

Penetrační testování probíhá vždy dle předem vybrané metodiky. Těmi se zabývá kapitola 4.2.

Podle toho, co chce organizace otestovat, je vybrána jedna ze tří kategorií penetračních testů (Baloch, 2015):

- *Black Box* – testování, kdy tester nemá žádné nebo jen minimum informací o svém cíli. Například v případě síťových penetračních testů by jedinou informací byl rozsah IP adres k testování, ale žádné informace o operačním systému či serverech.
- *White Box* – opak k Black boxu, kdy má penetrační tester k dispozici všechny informace o svém cíli. V případě síťového penetračního testu to znamená dokonce i informace o tom, jaké aplikace jsou spuštěny, jejich verze, operační systém, apod. V případě testů webových aplikací jde o přístup ke zdrojovým kódům. White box testování je velmi častý scénář zejména v interních informačních systémech kvůli obavám z úniku informací.
- *Gray box* – testování na pomezí black a white boxu, tedy některé informace jsou testerovi známé, jiné ne. V případě síťového penetračního testu je známá aplikace schovaná za IP, nikoliv však její konkrétní verze. U testování webové aplikace jsou známy informace jako testovací účty a servery, na nichž je spuštěný backend a databáze.

Podle toho, co přesně je v rámci penetračního testu testováno, jde o jeden či více typů testů z následujícího seznamu (Ali, Allen, 2014):

- síťový penetrační test,
- penetrační test webové aplikace,

- penetrační test mobilní aplikace,
- penetrační test sociálního inženýrství,
- fyzický penetrační test.

4.1.4 Oblasti a techniky penetračního testování

Penetrační testování je zajímavé zejména šíří svého záběru. Testování samotné začíná sběrem informací pomocí velmi široké škály technik, pokračuje vyhodnocením cíle a např. scanováním portů. Pak přijdou na řadu samotné penetrační testy a nakonec vyhodnocení zranitelností, jak uvádí Baloch (2015). Penetrační testování pak může zahrnovat následující typy útoků a řadu dalších:

- síťový sniffing
    - MITM útoky
    - ARP útoky
        - MAC flooding
        - ARP poisoning
    - DDoS útoky
    - odposlech komunikace
    - odposlech obrázků
    - zachytávání session
        - odposlech session cookie
    - SSL stripping
    - DNS spoofing (DNS cache poisoning)
    - DHCP spoofing
- vzdálené využití zranitelností
    - brute force útoky na vzdálené síťové služby
        - tradiční brute force útoky
        - slovníkové útoky
        - hybridní útoky
    - útoky na prolomení SSH
    - útoky na prolomení RDP
    - útoky na SMTP servery

    - útoky na SQL servery
    - útoky na slabou autentizaci
    - scanování portů
    - kompromitování Windows serverů
- zranitelnosti na straně klienta
    - e-maily se škodlivými přílohami
        - PDF útoky
    - e-maily se škodlivými odkazy
    - kompromitování aktualizací na klientovi
    - malware na USB úložištích
    - zranitelnosti prohlížečů
    - sbírání přihlašovacích údajů
    - vylákání přihlašovacích údajů
- útoky následující prolomení bezpečnosti systému
    - vytvoření výčtu dostupných částí systémů
    - zvýšení privilegií v systému
        - obejití řízení uživatelských účtů
        - zcizení bezpečnostního tokenu
    - získávání přístupu do dalších částí
    - udržení přístupu
        - tvorba backdoor
            - odstavení firewallu
            - odstavení antiviru
    - prolamování hesel / hashů
        - bruteforce
        - slovníkové útoky
        - salt útoky
        - duhová tabulka
    - data mining
        - sbírání informací o OS
        - sbírání uložených přihlašovacích údajů

    - identifikace dalších potenciálních cílů
        - mapování sítě
        - scanování portů, služeb a detekce OS
        - kompromitování dalších cílů na síti se stejným heslem
- hackování bezdrátových sítí
    - odhalení skrytých názvů sítí
    - sledování rámců, zachytávání paketů
    - obcházení MAC filtrů
    - prolamování WEP sítí
    - prolamování WPA/WPA2 sítí
    - podvrhování MAC adresy
- web hacking
    - viz kapitola 3

Všechny zmíněné oblasti a techniky by svojí rozsáhlostí vydaly na několik knih. To samé platí i pro web hacking samotný, kterému je věnována samostatná kapitola 3.2. Drtivá většina pentesterů, podle Beggse (2014), používá pro penetrační testování celý operační systém Kali Linux, což je debianový nástupce úspěšného ubuntu systému Backtrack 5, ve spojení s řadou dalších utilit.

4.2 Metodiky testování zabezpečení

Není sporu o tom, že testovat zabezpečení aplikací je žádoucí. Aby penetrační testování webových aplikací bylo systematické, vznikly metodiky penetračního testování. Metodiky pentestů neposkytují řešení jak testování provádět, ale teoreticky doporučují techniky a metody testování, pravidla, kterých se při testování držet a často celý proces penetračního testování dělí do fází.

4.2.1 Open Source Security Testing Methodology Manual (OSSTMM)

Tato metodika zahrnuje téměř všechny kroky každého penetračního testu. Je poměrně stručná a přesto velmi náročná na zvládnutí implementace do každodenního života. Penetrační testy v každém případě i přes svoji jednotvárnost vyžadují vysoké nároky na rozpočet firem, které jsou často s vykonáním takových testů neslučitelné (Muniz, 2013).

Rozdělení metodiky spočívá ve čtyřech skupinách – *scope, channel, index a vector* (Ali, Allen, 2014):

- *scope* – definuje proces sbírání informací na všech součástech systému,
- *channel* – udává typ komunikace mezi těmito součástmi, z typu pak vyplývá, co vše bude muset být otestováno,
- *index* – metoda utřídění součástí dle jejich indentifikátorů, např. MAC a IP adresy,
- *vector* – udává směr, kterým se tester může dostat ke všem součástem, které mají být testovány, a jakým směrem bude analyzovat každou součást.

Metodika OSSTMM dále rozděluje šest standardních typů testování, vhodných pro odlišné scénáře, jak uvádí Ali a Allen (2014):

- *slepé* – bez předchozí znalosti cílového systému, nicméně cíl ví, že je na něm vykonáváno penetrační testování,
- *dvojitě slepé* – bez předchozí znalosti cílového systému i bez vědomí cíle o penetračním testování,

- *gray box* – s limitovanou znalostí systému a vědomím cíle o penetračním testování,
- *dvojitý gray box* – stejné jako gray box, avšak s omezeným časem a bez testování channel a vector,
- *tandem* – testování s minimální znalosti cílového systému, cíl je o testování informován, testování je v tomto případě zcela úplné,
- *reversal* – testování, kdy tester ví vše o svém cíli a naopak cíl neví, jak nebo kdy bude testován.

4.2.2 Information Systems Security Assessment Framework (ISSAF)

Open source testovací framework je podobný metodice OSSTMM. Tento byl však rozkategorizovaný do několika domén týkajících se různých částí systému. Tyto domény mají adresovat vyhodnocování bezpečnosti v logickém pořadí. Jeho integrace do běžného životního cyklu v businessu dokáže poskytnout přesnost, kompletnost i efektivitu vyžadovanou v požadavcích testování (Ali, Allen, 2014)

Muniz (2013) uvádí, že ISSAF byl vyvinut se zaměřením na technickou a manažerskou část na rozdíl od OSSTMM. Manažerská část se týká zapojení managementu firmy do testování. Zaměření na technickou část znamená nejlepší postupy, které by měly být dodržovány během celého procesu testování. Výstupem je opět kompletní seznam zranitelností, které v systému existují, a navíc také provozní aktivity a bezpečnostní opatření. Proces vyhodnocování se zaměřuje na analýzu kritických zranitelností cíle, kterých může být zneužito s minimálním úsilím.

4.2.3 Open Web Application Security Project Testing Guide (OWASP)

Projekt, který přináší bezpečnost prostřednictvím principů a postupů bezpečného kódu. OWASP navíc poskytuje kompletního průvodce projektem testování aplikace. Přesto tento framework ukazuje obecné metody útoku nezávislé na použité technologii nebo platformě. Dává i specifické instrukce na to, jak testovat, ověřit a napravit zranitelné části aplikace. Hlavní zaměření projektu je spíše na vysoce

rizikové oblasti než na úplně všechny možné problémy bezpečnosti webových aplikací (Ali, Allen, 2014).

4.2.4 Web Application Security Consortium Threat Classification (WASC-TC)

Pro identifikaci bezpečnostních rizik je třeba přísných testovacích technik během celého životního cyklu vývoje aplikace. Web Application Security Consortium Threat Classification je dalším standardem k posouzení bezpečnosti webových aplikací. V kategorizaci útoků a slabin je velmi podobný OWASP standardu, ale zabývá se jimi mnohem detailněji (Ali, Allen, 2014).

Pro lepší a snazší porozumění celému postupu je nutné dodržovat přesnou terminologii, metodika jinak může působit jako příliš matoucí a nepřehledná. Ali a Allen (2014) uvádí rozdělení standardu do tří rozdílných pohledů pokrývajících rizika webových aplikací:

- *Výčtový pohled (Enumeration view)* – pohled věnovaný poskytnutí základů pro veškeré útoky a slabiny včetně přesné definice, typu a příkladů v různých programovacích jazycích a na různých platformách. Je popsáno 49 různých typů útoků.
- *Vývojový pohled (Development view)* – posouvá pohled vývojáře dále. Z kombinace útoků a slabin dělá zranitelnosti, které s vysokou pravděpodobností v jedné ze tří fází vývoje (návrh, implementace, nasazení) nastávají. V první fázi je to již během sběru požadavků, ve druhé nebezpečným kódem a ve třetí špatným nastavením aplikace, webového serveru nebo jiných externích systémů.
- *Taxonomický odkazový pohled (Taxonomy cross-reference view)* – umožňuje testerům / auditorům a vývojářům zmapovat terminologii napříč metodickými standardy. S dalším úsilím je možné dosáhnout dokonce na více bezpečnostních standardů najednou. Nicméně každá webová aplikace má svoje požadavky na bezpečnost dle různých kritérií. Pro dosažení každého standardu je tak zapotřebí různého úsilí v různých aplikacích dle výpočtů

rizik. WASC-TC útoky spadající do této kategorie jsou pak mapovány pomocí dalších standardních technik: Common Weakness Enumeration – CWE, Common Attack Pattern Enumeration and Classification – CAPEC atd.

4.2.5 Penetration Testing Execution Standard (PTES)

PTES byl vytvořen nejlepšími experty v průmyslu penetračního testování. Metodika popisuje sedm fází penetračního testování a může být použita jako penetrační test v jakémkoliv prostředí.

Jde o tyto fáze (Ali, Allen, 2014):

1. interakce před zahájením,
2. sbírání informací,
3. modelování hrozeb,
4. analýza zranitelností,
5. využití zranitelností,
6. útoky následující prolomení bezpečnosti,
7. informování o výsledcích.

4.3 Nástroje k penetračnímu testování

Ve většině případů bezpečnostní profesionálové, white hat hackeři, používají k penetračnímu testování naprosto stejné nástroje, jako používají black hat hackeři, crackeři – Kali Linux (operační systém vytvořený pro hackování a penetrační testování), Python (programovací jazyk) a další jazyky specifické pro každý typ aplikace a útoku, nejčastěji právě PHP a JavaScript (Sinha, 2017).

Používání stejných nástrojů oběma skupinami je podle Harrise logické, protože členové obou skupin dělají jednu a tu samou věc, jen za jiným účelem. Zároveň je nutné, aby odborníci na zabezpečení používali ty samé nástroje, jako útočníci, to proto, aby měli stejné možnosti a tedy stejnou šanci uspět v hledání slabin. Nástroje a postupy se ale v průběhu času mění a kdo zůstane pozadu, ve svém úkolu nejspíš selže. Je tedy v zájmu black hat i white hat hackerů používat co nejaktuálnější nástroje.

Není jednoduché rozhodnout, jaká pozice je výhodnější – útočníkovi stačí jeden nástroj a hluboká znalost několika málo útoků, dokonce i jeden jediný, nebo velké štěstí. Jeho oponent, který se stará o řádné zabezpečení aplikace, pak musí pokrýt zabezpečení kompletně a nevynechat jedinou mezeru, protože útočníci jsou zpravidla nevypočitatelní (Harris, 2008).

5 Obrana proti hrozbám v PHP

PHP je nejčastěji používaným server-side jazykem a vzhledem k jeho popularitě je naprosto zásadní, aby aplikace v něm psané byly dobře zabezpečené. Hackeři a crackeři umí být velmi inovativní a důmyslní ve svých postupech dobývání cíle. Často se dochází k závěru, že za většinou problémů se zabezpečením stojí lidský faktor a to i mimo oblast webových aplikací (Ludwig, 1995). Každému lidskému pochybení, ať už jde o programátora nebo administrátora hostingu, lze ale s větším či menším úsilím předejít a tato kapitola to názorně dokládá. V každém projektu, kde hrozí potenciální nebezpečí v nedostatečném zabezpečení, musí mít jeho bezpečnost na starost vždy jeden člověk nebo rovnou celý tým. Tím je zaručeno, že bezpečnost aplikace nezůstane upozaděna (Shema, 2011).

Kromě důrazu na zabezpečení při samotném psaní kódu, je důležité kód i testovat. To je všem vývojářům velmi dobře známý fakt. Častým problémem testování zůstává, že se aplikace testují jen na funkčnost, nikoliv na bezpečnost (neochráněné vstupy atp.). Testy pak jen ověří, že vše funguje, jak bylo požadováno, neověří ale, zda je aplikace dobře zabezpečná či nikoliv (Shema, 2011).

OWASP (2017) sestavil nový žebříček nejkritičtějších bezpečnostních rizik webových aplikací, který vypadá následovně:

1. injection,
2. chyby v autentizaci,
3. vystavení citlivých dat,
4. XML External Entities (XXE),
5. chyby v řízení přístupu,
6. špatně nakonfigurované zabezpečení,
7. XSS,
8. nebezpečná deserializace,
9. používání komponent se známými bezpečnostními riziky,
10. nedostatečné logování a sledování.

Těmito a dalšími body se zabývají následující kapitoly.

The PHP Group (2018) tento seznam dále rozšiřuje konkrétnějšími body vztahující se přímo k jazyku PHP:

- zabezpečení session,
- zabezpečení file systému,
- globální proměnné,
- maskování použitého programovacího jazyka.

PHP je velmi specifický jazyk a jako každý, má řadu silných i slabých stránek. O jeho slabých stránkách referuje následující kapitola.

5.1 Slabiny jazyka PHP

PHP (PHP: Hypertext Preprocessor, dříve Personal Home Page) je skriptovací programovací jazyk na straně serveru vytvořený v roce 1995 se zaměřením na webový vývoj. PHP je dostupné zdarma a jedná se o open source, tedy veškeré zdrojové kódy jazyka jsou zdarma volně dostupné. Od té doby ale uplynulo mnoho času a PHP se vyvinulo do víceúčelového jazyka s podporou struktur objektově orientovaného programování (OOP), closures, traits, jmenných prostorů a dalších – a tak už zdaleka neslouží jen pro weby. Jsou v něm psané SQL i NoSQL databázové systémy, servery, skripty na upravování fotografií, aplikace na rozpoznávání hlasu, API a řada dalších pro PHP netypických projektů. Weby samozřejmě dominují a to se v dohledné době nejspíš nezmění (Aley, 2016).

V listopadu roku 2017 bylo až 83% webových stránek včetně Wikipedie, Facebooku nebo Yahoo a aplikací napsaných v PHP. Takto hojně používaný jazyk znamená velké množství potenciálních cílů útoků, proto zde na bezpečnosti obzvlášť záleží. Mnoho webových projektů navíc používá open source knihovny třetích stran. Pokud se v takové knihovně vyskytne chyba, všechny weby používající takovou knihovnu automaticky obsahují tu samou zranitelnost, jak publikoval Acunetix (2018).

PHP je programovací jazyk

- interpretovaný,
- na straně serveru,
- nezávislý na platformě,
- dynamicky typovaný (většina typových kontrol je prováděna za běhu),
- slabě typovaný (typy proměnných jsou automaticky přiřazeny či převedeny).

Zejména z posledního bodu předchozího výčtu vychází potenciální problémy a ty popisuje následující podkapitola.

5.1.1 Slabé typování a ošetření vstupů

Při definici proměnné je její typ automaticky přiřazen podle hodnoty. Pokud data v nějaké proměnné budou mít nesprávný datový typ, jsou automaticky převedena na požadovaný typ. A tím se maskují chyby vývojáře a umožňuje se také vznik chyby, kdy jsou do vstupu v aplikaci (většinou přes formuláře nebo URL) vložena neočekávaná data. To může skrývat bezpečnostní problémy (Afooshteh, Hoffmann, Stock a Plant, 2017).

Pokud skript aplikace využívá GET proměnnou *$var* typu string a požaduje, aby byla shodná s další proměnnou vygenerovanou aplikací bez uživatelského vstupu, v nejjednodušší podobě bude skript vypadat následovně:

```
$var = $_GET['var'];
$expectedVar = getExpectedVar();

if (strcmp($var, $expectedVar) == 0) {
   …
}
```

Kód 1 - Kód podléhající neočekávanému vstupu

Adresa v prohlížeči bude v takovém případě např. jako v kódu 2.

```
https://adresa.cz?var=hodnota
```

Kód 2 - Adresa s očekávaným vstupem z parametru

Pokud útočník parametr v adrese změní jako v kódu 3,

```
https://adresa.cz?var[]=jakakoliv_hodnota
```

Kód 3 - Adresa s neočekávaným vstupem z parametru

tak proměnná *$var* ve skriptu už nebude typu řetězec (string), ale pole (array), tím pádem funkce *strcmp()* nevrátí ani 0, ani 1, jak je očekáváno, ale *NULL*. Pak díky slabému typování a využití operátoru *==* namísto *===* podmínka projde, i když by neměla, a to může mít nedozírné následky. Vývojář se tak vždy musí mít na pozoru a uvědomovat si potenciální vedlejší účinky již při psaní kódu.

Řešení je v tomto případě jednoduché, stačí v podmínce použít operátor identity namísto rovnosti a problém nenastane, protože platí:

```
null == 0
```

Kód 4 - Důvod umožnění úspěšného útoku

a zároveň platí i následující nerovnost

```
null !== 0
```

Kód 5 - Důvod neumožnění úspěšného útoku

Výsledný skript s vyřešeným bezpečnostním problémem tedy vypadá následovně:

```
$var = $_GET['var'];
$expectedVar = getExpectedVar();

if (strcmp($var, $expectedVar) === 0) {
   …
}
```

Kód 6 - Skript nepodléhající neočekávanému vstupu

5.1.2 PHP mechanismy routování

Vestavěné routování v jádru PHP spouští všechny soubory, které mají příponu php, v dostupné adresářové struktuře. Prakticky to znamená následující (Afoosteh, Naderi, Hoffmann, Stock a Plant, 2017):

- Existuje riziko spuštění škodlivého kódu s každým nahráním souboru na server, pokud není název souboru ošetřen.
- Zdrojové soubory aplikace a konfigurační soubory jsou ve veřejně dostupných adresářích spolu se soubory, které jsou dostupné ke stažení. Při špatném nastavení mohou být zdrojové soubory dostupné ke stažení útočníkům.
- Některé soubory, které nebyly navrženy jako vstupní soubory pro zahájení cyklu požadavku na server, mohou být právě tímto způsobem využity a to povede k nežádoucímu chování aplikace.
- Vývojáři využívají vlastní routovací mechanismy.

5.1.3 Další slabiny jazyka PHP

Ve většině vestavěných funkcí a řadě PHP knihoven nejsou využívány **výjimky**, ale pouze error reporting různých úrovní. Nastane-li neočekávané chování v PHP, pokud nenastane kritická chyba (výjimka nebo tzv. fatal error), kód běží dál a tak se mohou maskovat chyby. Jiné jazyky v takovou chvíli udělají to nejbezpečnější, co uznají za vhodné – běh aplikace zastaví.

PHP se od dob svého vzniku do dnešní doby zásadně vyvinulo a přesto řada funkcí (např. *addslashes()*) a sad funkcí (*MySQL* vs *MySQLi*), která v jazyce dodnes je, obsahuje chyby nebo poskytuje falešný pocit zabezpečení. Trvá dlouho, než jsou takové funkce a sady funkcí označeny jako „deprecated“ a následně odstraněny, kvůli zpětné kompatibilitě (Lockhart, Sturgeon, 2017).

Chování PHP silně závisí na konfiguračních hodnotách php.ini souboru, které mohou omezit, jaké funkce budou v dané instalaci PHP k dispozici. Proto je těžké psát kód, který bude fungovat za všech podmínek v různých prostředích. Knihovny třetích

stran mohou pak očekávat různá nastavení a tak nastává problém, jak knihovny správně využít.

PHP je zároveň i šablonovací jazyk, může tedy být využit uprostřed HTML šablon. A přesto s využitím základních funkcí neescapuje výstup (vyčištění dat k zobrazení od nežádoucích znaků, podrobně o této problematice pojednává kapitola 5.4) a je potřeba volat další funkce. Pokud to vývojář správně neudělá, vystavuje aplikaci riziku XSS.

Poslední slabou a silnou stránkou PHP zároveň je existence velkého množství knihoven a projektů třetích stran. Je skvělé, že vývojáři mají možnost zdarma rozšiřovat svoje aplikace o další funkcionality bez nutností psát veškerý kód od základu, knihovny ale mohou obsahovat celou škálu zranitelností využitelných útočníkem (Afooshteh, Hoffmann, Stock a Plant, 2017).

Aley (2016) tvrdí, že je důležité nezapomínat na aktualizace distribuce PHP. Každý den může být objevena chyba, ať už bezpečnostní chyba nebo chyba v chování některé z funkcí. Jen včasné aktualizace PHP distribuce zabrání zneužití takových chyb. Nové verze PHP přináší i nové možnosti a vyšší výkon a tak není důvod aktualizace odkládat.

5.2 Autentizační útoky

Autentizační útoky jsou útoky na mechanismus přihlašování uživatelů, tedy ověření toho, že uživatel je skutečně ten, za koho se přihlašuje. Na základě tohoto ověření pak uživatel získá přístup k dané části aplikace a příslušným datům v ní obsažených.

PHP jako takové ani knihovny, které jsou jeho nativní součástí, neobsahují ucelený systém autentizace. Frameworky tento neduh napravují a poskytují vlastní autentizační mechanismy, ne vždy jsou ale dokonalé. Pro bezpečné nasazení autentizačního mechanismu je třeba v každém případě dbát několika zásadních pravidel ohledně správy session, cookies a s tím související funkce pamatování přihlášení (Afoosteh, Naderi, Hoffmann, Stock a Plant, 2017).

Velký problémem dle OWASP (2017) zůstává, že na internetu jsou volně dostupné rozsáhlé uniklé databáze platných uživatelských jmen a k nim patřících hesel (včetně seznamů běžných výchozích kombinací). K těm mají samozřejmě přístup i útočníci, stejně jako k nástrojům, které tato data dokáží dobře zneužít – ať už jde o automatizované brute force útoky nebo slovníkové útoky. Není možné zcela automatizovaně zjistit, zda je autentizace špatně zabezpečená, ale když už se to útočníkovi podaří, může zranitelnosti automatizovanými nástroji využít.

Internetové vyhledávače jsou nejen dobrým pomocníkem pro běžné uživatele, ale i skvělým nástrojem pro útočníky. S použitím vyhledávačů je hledání stránek s autentizačními formuláři velmi snadné, stejně jako najít stránky, u kterých je velmi pravděpodobné, že používají konkrétní programovací jazyk. Stejně tak je snadné najít stránky, u kterých se dá odhadnout, že budou obsahovat alespoň trochu cenný obsah, nebo skutečně existující uživatelská jména na daném webu.

Stačí jednoduché dotazy jako v kódu 7 nebo v kódu 8 a útočník získá velkou část výchozích informací, které potřebuje k zahájení útoku.

```
username | userid | user.ID | "your username is"
```

Kód 7 - Příklad dotazu do vyhledávače pro nalezení webů s přihlašováním uživatelů

```
-ext:php
```

Kód 8 - Příklad dotazu do vyhledávače pro nalezení webů napsaných v PHP

Útočník takto dříve či později najde web napsaný v PHP s přihlášením a vytipuje si uživatelská jména, jejichž účty se může pokusit pomocí výše popsaných metod kompromitovat (Long, 2005).

OWASP (2017) ve svém dokumentu The Ten Most Critical Web Application Security Risks vypsal doporučení, jak autentizačním útokům předcházet:

- Vícefaktorová autentizace kdykoliv je to možné.
- Žádné výchozí přihlašovací údaje, zejména pro administrátorské účty.

- Funkcionalita při tvorbě hesla identifikující příliš slabá hesla.
- Délka, složitost i pravidla pro pravidelnou změnu hesla by se měla řídit některou z moderních sad pravidel, která reflektuje skutečnou situaci ve světě hackerů.
- Chybové hlášky stejné pro všechny vstupy neumožňující přihlášení.
- Limitování počtu neúspěšných pokusů pro zadání hesla a postupné prodlužování mezery mezi dalšími pokusy.
- Server-side bezpečná správa session s novým náhodným session ID uloženém na bezpečném místě (rozhodně ne v URL), které je úspěšně anulováno po odhlášení.

Následují znaky aplikací, které většinou znamenají, že daná aplikace není dostatečně dobře zabezpečená:

- dovoluje využití automatizovaných útoků typu zkoušení seznamu uživatelských jmen a příslušných hesel,
- umožňuje brute force nebo jiné automatizované útoky,
- povoluje výchozí, slabá nebo typická velmi dobře známá hesla,
- využivá nedostatečně zabezpečené procesy na obnovu ztracených údajů, např. otázky, na které může útočník získat nebo uhodnout odpověď,
- hesla nehashuje vůbec, pouze je šifruje nebo je hashuje nedostatečně nebo nevhodným algoritmem,
- nemá vůbec nebo má neefektivní vícefaktorovou autentizaci,
- vystavuje session ID v URL,
- nemění session ID po každém úspěšném přihlášení,
- správně nezneplatňuje session ID po určité době nebo odhlášení.

5.2.1 Problematika hesel

Pokud by se stalo, že se útočníkovi podaří dostat do databáze aplikace, je naprosto nezbytné, aby mu to nepřineslo více užitku než je nutné. Hesla proto musí být uložena v hashované podobě. Hash je jednosměrová funkce aplikovaná na heslo každého uživatele, která každé heslo nevratně transformuje na vždy stejný řetězec

znaků, nezávisle na délce původního hesla. Při přihlašování uživatele se pak zadávané heslo také hashuje a porovnává se s uloženým hashem hesla daného uživatele (Lockhart, Sturgeon, 2017).

Hashovací funkce *H* přijímá vstupní zprávu *m* a transformuje ji do hašované podoby *h*, která je funkcí vstupní zprávy *h = H(m)*. Vstupní zpráva je variabilní délky a hašovaná podoba má konstantní délku. Hashované podobě se říká také „*fingerprint*", otisk původní zprávy, protože je velmi nepravděpodobné, aby dvě zprávy vyprodukovaly stejný hash, stejně jako u otisků prstů. Důležitá je zejména náročnost získání původní zprávy z hashe – u zatím neprolomených hashovacích metod nelze původní zprávu získat v reálném čase (Mogollon, 2007).

Hashovací funkce by měla mít následující vlastnosti:

- zpráva může být jakékoliv délky,
- výsledný hash musí být konstantní délky,
- je celkem snadné spočítat hashovací funkci *H(m)* pro jakoukoliv zprávu *m,*
- není možné v reálném čase:
    - najít původní zprávu m z hashovací funkce *H(m)*,
    - mít dvě různé zprávy m1 a m2, pro které platí *H(m1) = H(m2)*,
    - najít dvě různé zprávy m1 a m2, u kterých platí H(m1) = *H(m2)*.

Hashovací funkce se dle Mogollona (2007) nazývá silná hashovací funkce, pokud splňuje všechny podmínky ve výše uvedeném seznamu.

Již v roce 1995 vznikl Secure Hash Standard s prvním secure hash algoritmem – SHA-1. Postupně vznikly další, SHA-256, SHA-384 nebo SHA-512, které se liší v maximální délce zprávy, délce hashe a tím i celkové bezpečnosti daného hashe.

Kryptograficky bezpečné hashovací algoritmy jsou navrženy tak, aby bylo možné rychle vypočítat hash. Útočník toho ale může využít také a snadno si předpočítat hashe pro všechna běžná a krátká hesla a uložit je do veliké tabulky (duhová tabulka, tzv. rainbow table). Najít odpovídající heslo v tabulce získané z uniklé databáze uživatelských účtů a hashů hesel je pak velmi snadné. Proti tomuto útoku

ale existuje účinná metoda obrany – přidávání kryptografické soli k heslu před vypočítáním hashe. V praxi to znamená, že se k heslu vždy před výpočtem hashe přidává náhodně zvolený řetězec znaků, a to i při přihlašování uživatele (Acobs, 2016).

The PHP Group (2018) označila hashovací algoritmy MD5 nebo SHA-1 či SHA256 za nevhodné pro hashování hesel – byly navrženy tak, aby byly velmi rychlé a účinné. S moderními výkonnými počítači se ale stalo trivialitou využít brute force metody útoku ke zjištění původního vstupu.

PHP bylo ve verzi 5.5 obohaceno o „*native password hashing API*" obsahující mimo jiné metodu *password_hash()*. Dodnes využívá hashovací algoritmus **BCrypt**, který je na rozdíl od MD5 dosud neprolomený a je aktuálně nejsilnějším hashovacím algoritmem, který PHP nativně podporuje. Funkce *password_hash()* se automaticky stará o přidání kryptografické soli a sůl, algoritmus i výpočetní náročnost výpočtu se ukládá jako součást hashe (Lockhart, Sturgeon, 2017).

Řešení ve frameworku Nette 2.4

Výňatek klíčového kódu verifikace hesla při přihlášení výchozím autentifikátorem:

```
if (strcasecmp($name, $username) === 0) {
    if ((string) $pass === (string) $password) {
        return new Identity($name, isset($this->usersRoles[$name])
            ? $this->usersRoles[$name]
            : null);
    } else {
        throw new AuthenticationException('Invalid password.',
self::INVALID_CREDENTIAL);
    }
}
```

Kód 9 - Ukázka kódu výchozího autentifikátoru v Nette 2.4 využívající striktní porovnání řetězců

Nette také obsahuje pomocnou třídu Passwords, která využívá volání funkce *password_verify()*, vestavěný *SimpleAuthenticator* ale využívá metodu striktního porovnávání řetězců.

To je vše, čím je předpřipravený autentifikátor Nette frameworku vybaven. Pro odolávání brute force útokům limitováním počtu neúspěšných pokusů a případně i postupným prodlužování mezery mezi dalšími pokusy o přihlášení je nutné napsat vlastní kód, nebo se poohlédnout po jiném řešení.

Řešení ve frameworku Laravel 5.5

Klíčový kód verifikace hesla při přihlašování výchozím autentifikátorem:

```
if (strlen($hashedValue) === 0) {
    return false;
}

return password_verify($value, $hashedValue);
```

Kód 10 - Ukázka kódu výchozího autentifikátoru v Laravelu 5.5 využívající PHP Password API a metodu password_verify()

Laravel jde cestou doporučovanou přímo PHP Group a pro ověření správnosti hesla využívá PHP Password API funkci *password_verify()*.

Celý autentifikátor je ale daleko propracovanější než v Nette 2.4, tělo metody *login()* ukazuje kód 11.

```
$this->validateLogin($request);

if ($this->hasTooManyLoginAttempts($request)) {
    $this->fireLockoutEvent($request);

    return $this->sendLockoutResponse($request);
}

if ($this->attemptLogin($request)) {
    return $this->sendLoginResponse($request);
}

$this->incrementLoginAttempts($request);

return $this->sendFailedLoginResponse($request);
```

Kód 11 - Tělo metody login() autentifikátoru frameworku Laravel 5.5

Nejprve jsou zvalidovány *$_POST* proměnné na přítomnost, že nejsou prázdné a že jsou typu řetězec, to vše v metodě *validatesLogin()*.

Následně se metodou *hasTooManyLoginAttempts()* zkontroluje, po kolikáté se daný uživatel (identifikovaný uživatelským jménem a IP adresou) snaží přihlásit a v případě, že překročí stanovený limit pěti neúspěšných pokusů o přihlášení během jedné poslední minuty, jsou mu po následující jednu minutu od prvního pokusu další pokusy o přihlášení odepřeny. Pokud se do pěti pokusů přihlásí úspěšně, je počítadlo neúspěšných pokusů vynulováno. Pokud však všech pět pokusů na přihlášení selže, minutu vyčká a zkouší se přihlásit znovu, platí stejné pravidlo jako v předchozím procesu přihlašování – pět pokusů na přihlášení a v případě pěti neúspěšných pokusů minutová přestávka, než budou další pokusy o přihlášení vůbec vyhodnocovány na správnost údajů.

Za předpokladu, že uživatel výše zmíněnými kroky projde, je heslo zkontrolováno na správnost metodou v kódu č. 10 a v případě, že je správné, je uživatel přesměrován na výchozí stránku po přihlášení. V opačném případě je zobrazena chyba o neúspěšném přihlášení a v případě překročení pěti pokusů během jedné

minuty zobrazena informace, co se stalo a kdy se uživatel může pokusit přihlásit znovu.

Toto řešení je navíc konfigurovatelné (vlastní hodnoty, kolik neúspěšných pokusů je třeba pro odepření přihlašování a jak dlouho odepření přihlašování trvá) a jednoznačně lepší co se bezpečnosti týká než řešení Nette 2.4. Je ale jen na vývojářích, aby zkontrolovali způsoby přihlašování ve svých aplikacích a zabezpečili přihlašování na takovou úroveň, jakou je třeba. Není dobré spoléhat se na vestavěná hotová řešení frameworků, pokud nesplňují bezpečnostní politiku organizace, pro kterou je aplikace vyvíjena. I v autentizaci v Laravelu 5.5 je rezerva, jak by bezpečnost přihlašování mohlo být vylepšena – např. po každých pěti neúspěšných pokusech by se mohla inkrementálně zvětšovat přestávka mezi dalšími pěti pokusy, i aktuální řešení je ale velmi dobré a pro většinu aplikací zcela dostatečné.

5.2.2 Útoky na session (session hijacking)

Výchozí API pro práci se sessions v PHP se obecně považuje za bezpečné, zejména díky náhodnosti *PHPSessionID*, identifikátoru session. Bezpečné už ale nemusí být uložiště session – což je zpravidla */tmp* adresář na serveru, který může podléhat tzv. *local file inclusion* útokům (jsou rozvedeny v kapitole 5.8). Je možné přesunout session do paměti nebo implementovat vlastní mechanismus session s úložištěm např. v databázi (PHP.earth, 2018).

Většině session hijacking útoků lze předejít vázáním session na IP adresy. To ale může znamenat problém pro uživatele, kteří přistupují k webu prostřednictvím anonymizačních prostředků jako je síť TOR, a také problémy v případě přístupu z lokální sítě, kdy je třeba takovou IP ošetřit podmínkou. Existuje řada dalších doporučení, jak se sessions pro maximální bezpečnost nakládat (Afooshteh, Hoffmann, Stock a Plant, 2017):

- Nastane-li porušení pravidel konkrétní session (2 různé IP se hlásí k jednomu účtu), je třeba invalidovat celou session – smazat cookie, uložiště

této session a vše s ní spojené. V ideálním případě uživatele informovat o tom, co se stalo, jako to dělá např. Google ve svých službách.

- Měnit session ID při každé významné akci, např. při přihlášení, nebo i po každém požadavku.
- Nikde nevystavovat session ID, obzvlášť ne tehdy, kdy je vázané na přihlášeného uživatele.
- Pokud je to třeba, přenášet session ID bezpečně (TLS).
- Session musí mít omezenou platnost, může platit po omezenou dobu neaktivity i po omezenou dobu aktivity a session řádně invalidovat a odstranit – a to i při odhlášení.
    - Pokud se požadavek provede více než X vteřin po posledním požadavku, session bude expirovat (typicky 30 minut).
    - Pokud je aktuální session aktivní určitou dobu (většinou den až týden), bude invalidována a smazána.

5.2.3 Útoky na cookies

Autentizace, session a cookies úzce spolupracují (Stock, Gonçalves, Correa, 2017). Pravidla pro nakládání s nimi jsou velmi prostá. Data v cookie se nesmí serializovat, protože mohou být příliš snadno změněna, např. přidány proměnné a podobně. Cookie je vždy třeba mazat následujícím způsobem:

```
setcookie ($name, "", 1);
setcookie ($name, false);
unset(_COOKIE[$name]);
```

Kód 12 - Ukázka kódu, jak řádně a bezpečně smazat cookie

Jedině tak je zaručeno, že se cookie opravdu smaže (první řádek zajistí expiraci v prohlížeči, druhý je standardní způsob odstranění cookie a třetí smaže cookie z PHP skriptu).

Posledním pravidlem je dobrý zvyk programátora využívat HTTP-only cookies. HTTP-only cookies jsou cookies, které jsou dostupné jen skrz HTTP požadavky a ne

přes JavaScript. Skripty XSS útoků k nim nemají přístup – bohužel však bezpečnost zcela nezajistí. Důvodem je, že některé prohlížeče bohužel obsahují chyby v implementaci HTTP-only cookies nebo vlastnosti těchto cookies nemají vůbec implementované. To má za následek vystavení HTTP-only cookies pro client-side skripty a primární cíl těchto cookies tak nemusí být naplněn (Afooshteh, Hoffmann, Stock a Plant, 2017).

5.3 Databázové a SQL injection útoky

Jedná se o útoky na databázové servery prostřednictvím webové aplikace. Webová aplikace při nich využívá nedůvěryhodná data (tedy data pocházející od uživatele) přímo při tvorbě SQL dotazu a celý dotaz včetně vstupu od uživatele v původní podobě předá databázovému serveru. Databázový server dotaz spustí a aplikace se může v tu chvílí ocitnout pod kontrolou útočníků. Útočníci tak mohou získat cenná data, data z databáze smazat nebo je jinak znehodnotit (Messier, 2016).

Spolu s dalšími injection útoky (path, code injection, command injection…) jde dlouhodobě o nejrizikovější oblast zabezpečení všech webových aplikací, následovanou autentizačními útoky, uvedl OWASP (2017).

K takovému útoku zpravidla dochází, když:

- Vstupní data od uživatele (útočníka) nejsou validována, filtrována ani vyčištěna od nežádoucího obsahu.
- Aplikace využívá dynamické databázové dotazy bez escapování vzhledem ke kontextu a vstupní data jsou tak neescapovaná používána v dotazech.
- Vstupní data od uživatele (útočníka) jsou vhodně využita ve vyhledávacích parametrech objektově relačního mapování, aby se útočník dostal k dalším, citlivým datům.
- Vstupní data od uživatele (útočníka) se přímo používají v dotazech na databázový server prostřednictvím aplikace napřímo nebo vložením do existujícího dotazu nebo uložené procedury.

Podstatné je, že koncept je vždy stejný, nezávisle na typu interpreta (ORM, SQL, NoSQL…).

Základem úspěšného řešení v PHP je vhodné používání MySQLi (nikoliv MySQL) nebo PDO rozšíření PHP, popř. dobrý ORM framework. ORM frameworky jsou součástí řady populárních PHP frameworků (Mitchell, 2010). Za takovými frameworky se rozšíření MySQLi nebo PDO zcela jistě nachází. Samotné používání PDO nebo ORM frameworku ale bezpečnost nezaručí – chyba, která v důsledku umožní úspěšný SQL injection útok, je méně pravděpodobná, ale stále velmi dobře možná, proto je třeba dbát, aby i ORM framework byl používaný správně – a to tak, aby tato rizika potlačil. Vhodné používání těchto rozšíření nebo ORM je naprosto klíčové, protože při jejich nevhodném užití je aplikace zranitelná SQL injection útoky, jak ukazují příklady v následujících kapitolách (OWASP, 2017).

5.3.1 Příklad SQL injection a řešení v případě použití PDO rozšíření PHP

```
$id = $_GET['id'] ?? null;

$dbh = new PDO('mysql:dbname=testdb;host=127.0.0.1', 'username',
'password');

$sql = 'SELECT username, e-mail FROM users WHERE id = ' . $id;

foreach ($dbh->query($sql) as $row) {
    printf ($row['username'] . ' ' . $row['e-mail']);
}
```

Kód 13 - Příklad kódu s PDO, který může umožnit SQL injection

V kódu 13 je stránka, která zobrazuje informace (uživatelské jméno a heslo) o konkrétním uživateli. Z GET parametru HTTP požadavku je přejata proměnná *$id* a není dál zpracována (validace, filtrování, escapování). Je ale přímo použita na konci SQL dotazu, kam byla vložena pomocí spojování řetězců. To je chyba, stačí GET parametr v URL změnit a to tak, že obsahem parametru je vhodný SQL kód. Tento kód při spuštění na databázovém serveru ukončí první dotaz do databáze

pomocí doplnění SQL kódu, který je součástí aplikace. Po ukončení předešlého SQL dotazu útočník naváže dalším dotazem, ve kterém může udělat naprosto cokoliv. Může třeba celou databázi smazit, upravit některé záznamy apod., většinou ale bývá výhodnější dostat databázi nebo aplikaci pod svoji kontrolu, např. přidáním vlastního uživatelského účtu. Příkladem takové změny GET parametru může být následující kód 14.

```
1; DROP TABLE users
```

Kód 14 - Vhodná hodnota GET parametru URL pro SQL injection

Jestliže URL této stránky vypadá standardně jako v následující ukázce kódu 15,

```
http://example.com/get-user.php?id=1
```

Kód 15 - URL stránky zranitelné SQL injection tak, jak ji uživatel standardně vidí

po úpravě kódu 14 bude vypadat následovně:

```
http://example.com/get-user.php?id=1; DROP TABLE users
```

Kód 16 - URL stránky zranitelné SQL injection po vhodném doplnění útočníkem

Ve chvíli, kdy se tato stránka načte, útočníkovi se naposledy zobrazí a pak bude celá tabulka uživatelů smazána, přesně jak útočník zamýšlel, a to jen proto, že se dotaz doplní tak, jak ukazuje následující kód 17.

```
SELECT username, e-mail FROM users WHERE id = 1;
DROP TABLE users
```

Kód 17 - Výsledný dotaz do databáze obsahující i dotaz útočníka

Řešení je přitom nasnadě, v případě PDO rozšíření stačí využít takzvané *prepared statements*, jak popisuje kód 18.

```
$sql = 'SELECT username, e-mail FROM users WHERE id = :id';

$statement = $dbh->prepare($sql, [PDO::ATTR_CURSOR =>
PDO::CURSOR_FWDONLY]);

$statement->execute([':id' => $id]);

$users = $statement->fetchAll();
```

Kód 18 - Řešení SQL injection v případě využití PDO rozšíření PHP

Místo spojování řetězců pomocí tečky je v tomto případě proměnná v dotazu nahrazena parametrem dotazu *:id*. V dalším kroku je vytvořen samotný statement a parametr je při spuštění prepared statementu naplněn proměnnou *$id*. Pokud by se v tomto případě vyskytl v proměnné celý SQL dotaz nebo jiný škodlivý kód, prepared statement by jeho vykonání zabránil (PHP.earth, 2018).

5.3.2 Příklad SQL injection a řešení s využitím MySQLi rozšíření PHP

```
$id = $_GET['id'] ?? null;

$mysqli = new mysqli('localhost', 'username', 'password',
'testdb');

if ($mysqli->connect_error) {
    die('Connection error (' . $mysqli->connect_errno . ') ' .
$mysqli->connect_error);
}

$query = 'SELECT username, e-mail FROM users WHERE id = ' . $id;

if ($result = $mysqli->query($query)) {
        while ($row = $result->fetch_row()) {
        print($row[0] . ' ' . $row[1]);
    }
    $result->close();
} else {
    die($mysqli->error);
}
```

Kód 19 - Příklad kódu s MySQLi, který může umožnit SQL injection

V tomto případě jde o naprosto totožný způsob provedení útoku, jako popisuje kapitola 5.3.1. Opět je zde zcela nezpracovaný vstup z GET parametru HTTP požadavku, který je přesně v takové podobě použitý pouhým spojováním řetězců v SQL dotazu. Tím je umožněno útočníkovi GET parametr změnit na takovou hodnotu, která vhodně doplní onen SQL dotaz a útočníkovi je tak umožněno volat vlastní SQL dotazy nad databází svého cíle.

Vzhledem k jinému použitému rozšíření PHP pro přístup k databázi je zásadní změna v řešení. Stejně jako v předchozím případě ideální řešení spočívá ve využití *prepared statements*, tentokrát ale jiným způsobem, jak, názorně ukazuje kód 20.

```
$query = 'SELECT username, e-mail FROM users WHERE id = ? ';

$statement = $mysqli->stmt_init();

if ($stmt->prepare($query)) {
    $statement->bind_param('i', $id);
    $statement->execute();

    $result = $statement->get_result();

    while ($row = $result->fetch_array(MYSQLI_NUM)) {
        print($row[0] . ' ' . $row[1]);
    }
}
```

Kód 20 - Řešení SQL injection v případě využití MySQLi rozšíření PHP

Stejně jako při využití PDO rozšíření PHP, místo spojování řetězců pomocí tečky je v tomto případě proměnná v dotazu nahrazena parametrem. Parametr dotazu je ale v tomto případě označený otazníkem, není tedy pojmenovaný. V dalším kroku je opět vytvořen samotný statement, pouze na jiném objektu. Pak parametr opět nabyde hodnoty proměnné *$id*. Stále platí, že pokud by se v tomto případě vyskytl v proměnné *$id* celý SQL dotaz nebo jiný škodlivý kód, prepared statement by jeho vykonání zabránil (PHP.earth, 2018).

5.3.3 Řešení SQL injection v ORM frameworku Nette

Připojení k databázi v Nette frameworku pomocí vestavěného ORM znázorňuje kód 21 (metoda *connect()* třídy *Nette\Database\Connection*).

```
public function connect()
{
	if ($this->pdo) {
		return;
	}

	try {
		$this->pdo = new PDO($this->params[0], $this->params[1], $this->params[2], $this->options);
		$this->pdo->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
	} catch (PDOException $e) {
		throw ConnectionException::from($e);
	}

	$class = empty($this->options['driverClass'])
		? 'Nette\Database\Drivers\\' . ucfirst(str_replace('sql', 'Sql', $this->pdo->getAttribute(PDO::ATTR_DRIVER_NAME))) . 'Driver'
		: $this->options['driverClass'];
	$this->driver = new $class($this, $this->options);
	$this->preprocessor = new SqlPreprocessor($this);
	$this->onConnect($this);
}
```

Kód 21 - Ukázka připojení k databázi ve vestavěném ORM frameworku Nette

Pokud chce programátor přes toto ORM zadat příkaz do databáze, použije metodu *query($sql, ...$params)*, kde *$sql* je SQL dotaz v podobě s otazníky jako zástupnými znaky proměnných a *$params* jsou pak samotné proměnné, které mají být v dotazu použity. ORM framework pak vytvoří novou instanci třídy *Nette\Database\ResultSet*, kde řádně využívá PDO a *prepared statements*, jak ukazuje kód 22.

```
public function __construct(Connection $connection, $queryString,
array $params)
{
	$time = microtime(true);
	$this->connection = $connection;
	$this->supplementalDriver = $connection-
>getSupplementalDriver();
	$this->queryString = $queryString;
	$this->params = $params;

	try {
		if (substr($queryString, 0, 2) === '::') {
			$connection->getPdo()->{substr($queryString,
2)}();
		} elseif ($queryString !== null) {
			static $types = ['boolean' =>
PDO::PARAM_BOOL, 'integer' => PDO::PARAM_INT,
				'resource' => PDO::PARAM_LOB, 'NULL'
=> PDO::PARAM_NULL, ];
			$this->pdoStatement = $connection->getPdo()-
>prepare($queryString);
			foreach ($params as $key => $value) {
				$type = gettype($value);
				$this->pdoStatement-
>bindValue(is_int($key) ? $key + 1 : $key, $value,
isset($types[$type]) ? $types[$type] : PDO::PARAM_STR);
			}
			$this->pdoStatement-
>setFetchMode(PDO::FETCH_ASSOC);
			$this->pdoStatement->execute();
		}
	} catch (\PDOException $e) {
		$e = $this->supplementalDriver-
>convertException($e);
		$e->queryString = $queryString;
		throw $e;
	}
	$this->time = microtime(true) - $time;
}
```

Kód 22 - Ukázka kódu předávající SQL dotaz databázi v ORM frameworku Nette

Klíčové využití parametrických *prepared statements* je vidět v *elseif* větvi uvnitř *try* bloku. V případě Nette tak lze shrnout, že jeho ORM framework opravdu řádně brání SQL injection i dalším útokům a využívá PDO rozšíření PHP a parametrické prepared statements.

5.3.4 Řešení SQL injection v ORM frameworku Laravel

Připojení k databázi ve vestavěném ORM frameworku frameworku Laravel je velmi stručné, neboť zdrojový kód frameworku Laravel je velmi fragmentovaný. Je rozdělen do velkého počtu menších tříd, z nichž se využívají jen ty, které odpovídají aktuálnímu nastavení a kde každá má svoji jednu jedinou zodpovědnost spíše menší velikosti. Samotné připojení má na starost třída *Illuminate\Database\Connectors\Connector* a tělo metody *createPdoConnection()* je zobrazeno v kódu 23.

```
protected function createPdoConnection($dsn, $username, $password,
$options)
{
        if (class_exists(PDOConnection::class) && ! $this-
>isPersistentConnection($options)) {
                return new PDOConnection($dsn, $username, $password,
$options);
        }

        return new PDO($dsn, $username, $password, $options);
}
```

Kód 23 - Ukázka připojení k databázi ve vestavěném ORM frameworku Laravelu

Práce s databází je zde více abstraktní, pokud si to vývojář nevynutí, využívá abstraktní metody jednotlivých modelových tříd a k samotnému psaní SQL dotazů se ani nemusí dostat. Interface veřejných metod ORM frameworku ale není náplní této práce, to podstatné je, zda jsou také použity parametrické prepared statements nebo jiný přístup, vzhledem k využití PDO je ale více pravděpodobná první varianta. Klíčový kód je k nalezení v třídě *Illuminate\Database\Connection* a v ukázce kódu 24.

```
public function select($query, $bindings = [], $useReadPdo = true)
{
        return $this->run($query, $bindings, function ($query,
$bindings) use ($useReadPdo) {
                if ($this->pretending()) {
                        return [];
                }

                $statement = $this->prepared($this-
>getPdoForSelect($useReadPdo)->prepare($query));

                $this->bindValues($statement, $this-
>prepareBindings($bindings));

                $statement->execute();

                return $statement->fetchAll();
        });
}
```

Kód 24 - Ukázka kódu předávající SQL dotaz databázi v ORM frameworku Laravelu

Přestože kód se malinko liší a vyskytuje se v něm volání dalších metod ve zmiňované třídě, samotné provedení, resp. funkce použité v pozadí tohoto ORM frameworku jsou shodné jako v Nette a plně dostatečné pro bezpečné dotazování do databáze.

5.3.5 Kompletní řešení SQL injection útoků

Základem prevence SQL injection spočívá v používání bezpečného API, které poskytuje parametrický interface, ať už je to PDO nebo MySQLi rozšíření PHP či ORM framework využívající jedno z rozšíření v pozadí. Tuto problematiku řeší kapitoly 5.3.1 – 5.3.4.

To je ale pouze nutný přepoklad zabezpečení, ne zabezpečení samotné natož kompletní. V první linii praktického zabezpečení webové aplikace stojí **validace vstupních dat** pocházejících od uživatele. Zranitelnosti typu SQL injection, HTML injection (a s tím spjatá část XSS útoků) jsou zcela založeny na schopnosti útočníka vložit na vhodné místo (proto injection) taková škodlivá vstupní data,

která aplikace neočekává. V případě, že se s nimi aplikace nedokáže vypořádat, je úspěšně napadena. Tomu by ale měla validace zabránit tím, že u vstupních dat zajistí (Scambray, Liu, Sima, 2011):

1. správný formát,
2. správný datový typ,
3. správnou délku,
4. popř. i správný rozsah.

Perfektním příkladem je poštovní směrovací číslo, které má uživatel zadat do formuláře. Pro rychlé doručení pošty je tato informace klíčová, proto toto pole musí být vždy vyplněno. Také je známý tvar PSČ – pět číslic, volitelně může být po třetí číslici mezera, celkem tedy 5 nebo 6 znaků. Aplikace zkontroluje délku vstupních dat na 5 nebo 6 znaků. Pak zkontroluje, že se jedná pouze o číslice a mezery. Je-li délka 5 znaků, musí být všechny znaky číslice. Pokud je délka 6 znaků, pak musí jít o 3 číslice, mezeru a 2 číslice. Každá číslice bude vždy v rozsahu od 0 do 9. To by samo o sobě byla velmi dobrá a účinná obrana, lze ale využít známého faktu, že poštovních směrovacích čísel je omezená množina a všechna jsou dohledatelná. Po splnění všech předchozích podmínek tak lze do dat ještě jednu, která ověří, že zadané PSČ je v množině existujících PSČ. PSČ typu 00000 by pak zcela jistě neprošlo validací (Bosworth, Kabay, Whyne, 2014).

Ne vždy je nutné, aby byla validace naprosto striktní, může být i o něco volnější, ale vždy musí být provedena správně. Často se stávalo, že tyto testy byly naprogramovány jen v JavaScriptu a validace tak probíhala **pouze na straně klienta**. Klient ale může JavaScript vypnout a validaci obejít. Proto je třeba tuto validaci naprogramovat především na straně serveru, v tomto případě v PHP. Hlavní účel JavaScriptové validace by tak měl být hlavně výkon, tzn. rychlost, nikoliv bezpečnost. U velkých aplikací se díky ní může nezanedbatelně snížit zátěž na servery (Scambray, Liu, Sima, 2011).

Druhým krokem správného zabezpečení musí být bezpečné **escapování vstupních dat od uživatele**, tedy nahrazení některých znaků jinými (i více) znaky, pokud jsou

nežádoucí nebo neočekávané v datech. Některým útokům by předešla jednoduchá nahrazení třeba jednoho jediného znaku (rovnítka), kompletní řešení je ale samozřejmě složitější (Cannings, Dwivedi, Lackey, 2008).

The PHP Group (2018) píše, že rozšíření PHP MySQLi i PDO a stejně tak dobré ORM frameworky tento problém kompletně řeší za programátora. I proto není tomuto kroku často věnována tak velká pozornost, jakou by si zasloužil. K řešení jsou využívány tzv. *prepared statements*. V případě, že je použita hodnota od uživatele přímo v dotazu pomocí spojování řetězců, je manuální escapování naprosto zásadní pro bezpečnost aplikace. Pro escapování pro MySQL databázi je v PHP funkce *mysqli_real_escape_string()*, ale ani její použití nemusí být dostatečné – podtržítka a procenta, což jsou znaky, které mohou být snadno zneužity ke zmanipulování dotazu (*like* operátor), escapovány nejsou a je třeba je escapovat extra navíc ve své vlastní funkci.

Kromě samotného escapování jako takového je žádoucí odepřít uživateli použití některých znaků a zkrátka je ze vstupu filtrem odebrat. Na základě konkrétního vstupu se dá předpokládat, jaká skupina znaků bude potřeba. V případě e-mailu jistě stačí alfanumerické znaky, tečka, podtržítko a zavináč. Lze tedy vytvořit buď white list povolených znaků nebo black list zakázaných znaků, podle toho, co je v konkrétním případě výhodnější. Pokud je cílem zakázat HTML, odebrání znaků větší než „>“ a menší než „<“ pomocí black listu bude jistě jednodušší, než vyjmenovávat všechny ostatní povolené znaky (Scambray, Liu, Sima, 2011).

5.3.6 Dělení SQL útoků

Jak v Nette, tak v Laravelu jsou ORM frameworky jednou z mnoha výhod jejich používání – a přesto stále volitelnou součástí, kterou vývojáři využívat nemusí. Tak či tak, podstatné je předcházet SQL injection útokům, které se mohou dělit do několika skupin dle toho, jak útočník k útoku přistoupí. Předchozí kapitoly se zabývaly SQL injection v jeho nejtypičtější podobě, která nebývá označována žádným vlastním názvem.

Union-based SQL injection útoky spočívají v nastavení hodnoty parametru GET proměnné na hodnotu v podobě SQL dotazu s využitím operátoru *union*. Tento operátor naváže na dotaz další dotaz, ve kterém lze získat data i z jiných sloupců a tabulek databáze oproti původnímu dotazu, a výsledky vrátí najednou. Příkladem může být např. kód 25.

```
?id=1 UNION ALL SELECT card_number,validity,code FROM credit_cards
```

Kód 25 - Příklad hodnoty GET parametru v případě UNION based SQL injection

Error-based SQL injection útoky, jak jejich název sám napovídá, využívají toho, že na databázi bude provedena taková operace, která povede k chybě. Cílem je získat nějaká data z databáze a zobrazit je v chybové zprávě namísto běžného výstupu.

Blind SQL injection útoky se použlívají v kombinaci s Boolean SQL injection útoky. Útočník prostřednictvím chyby v aplikaci pokládá na její databázi ano/ne dotazy a z odpovědi serveru pak určí, jak databáze odpověděla. Tento typ útoku je často používaný tehdy, když je aplikace náchylná k SQL injection, ale zobrazuje obecné chybové hlášky (HTTP stavové kódy - nejčastěji 500 a 404, redirect, SQL error…). Cíl těchto útoků je stejný jako u error-based SQL injection, získat data ve výpisu chyby aplikace, pouze k nim vede jiná cesta (Stock, Gonçalves, Correa, 2017).

Z celé kapitoly 5.3 vyplývá, že obrana proti SQL injection je naprosto nezbytná a tvoří základ pro bezpečnost aplikace. Vzhledem k povaze útoků, kdy většina z nich je typu pokus a omyl, je dle Messiera (2016) nutné zkoušet různé varianty injektovaného kódu a sledovat, zda a jak aplikace reaguje a je šance na úspěch.

5.4 Cross-site scripting (XSS)

K XSS dochází, kdykoliv se do výstupu aplikace dostanou nedůvěryhodná data bez správné validace nebo escapování a tato data obsahují kód, který běží na straně klienta, zpravidla JavaScript, nebo obsahuje skript, který aktualizuje obsah stránky

daty od uživatele opět bez řádného ošetření. Nejčastěji se tak děje přes URL (PHP.earth, 2018).

OWASP (2017) popisuje, že cross-site scripting umožní útočníkům spustit skripty v prohlížeči oběti, které jim mohou pomoci zmocnit se session nebo cookies oběti, změnit vzhled aplikací zranitelných XSS nebo přesměrovat uživatele na takové stránky, které mohou oběti způsobit nakažení počítače zákeřnými viry a následnou škodu.

Nedůvěryhodná data mohou být cokoliv, nejedná se pouze o GET a POST data z formulářů uživatele. Patří sem data ze $_SERVER superglobální proměnné PHP serveru, data z těla HTTP požadavku získaná přes *fopen('php://input', 'r')*, nahrané a stažené soubory, hodnoty session, data v cookies i data z webových služeb třetích stran (Lockhart, Sturgeon, 2017).

Podle Easttoma (2016) je tento typ zranitelnosti aplikace velmi rozšířený. Kdekoliv, kde lze na webu vložit komentář, recenzi produktu nebo jiný vstup od uživatele, je potenciál pro XSS. A tak jsou velmi ohroženy zejména e-shopy, ale nikoliv výlučně. V dnešní době jsou všechny velké nebo známé weby dobře zabezpečeny a XSS nepodléhají. Zabezpečení v tomto ohledu je totiž nesmírně důležité a přitom velmi jednoduché až triviální, stačí pouze filtrovat všechen vstup od uživatele. Největší problém je neinformovanost vývojářů v oblasti zabezpečení, kteří pak nevědomě napomáhají XSS ke snadnému uplatnění a hojnému rozšíření na webu.

Samotné prohlížeče obsahují určité zabezpečení, klíč k úspěšnému napadení aplikace spočívá v nalezení chyby v takovém zabezpečení nebo jeho obejití. Všechny prvky zabezpečení se snaží být na sobě nezávislé, ale pokud útočník dokáže vložit svůj JavaScriptový kód na vhodné místo, veškerá tato zabezpečení padnou. Prohlížeče mají několik bezpečnostních modelů, které se o jednu vrstvu zabezpečení starají (Cannings, Dwivedi, Lackey, 2008).

- **Same origin/domain policy**, tedy pravidlo stejného původu, je hlavní prvek zabezpečení ve všech prohlížečích. Origin, původ, je definován kombinací protokolu, adresy a portu. V praxi toto pravidlo znamená, že dynamický

obsah může číst obsah pouze z domény, na které se nachází a žádné jiné. Řízení toho, kam dynamický obsah může zapisovat, součástí zabezpečení není. To lze vyřešit omezeními na *cookies* nebo *headers*, které se o posílání požadavků z aplikace starají.

- **Cookie security model**. Cookies vznikly jako jedna z hlaviček HTTP požadavku, když bylo třeba v bezstavovém protokolu HTTP uchovávat některá data mezi jednotlivými požadavky. Cookies jsou dostupné skrz JavaScript, ale servery by měly být jejich hlavní řídicí jednotkou. Cookie security model zahrnuje omezení:
    - Domény (domain) - cookie jsou tak odeslány na takový požadavek, který je na stejnou doménu (ale může být i na jinou subdoménu, je-li zdrojem původního požadavku).
    - Cesty URL (path) – cookie je odeslána jen na stejnou cestu/URL jako je volitelně nastaveno v atributu path.
    - Protokolu HTTP/HTTPS (secure) – pokud je tento atribut nastaven, je cookie odeslána buď jen na HTTP nebo jen na HTTPS požadavky.
    - Platnosti (expires) – běžně cookie platí, dokud není uzavřen prohlížeč, pak je smazána. Cookie může být nastaveno datum, do kdy platí, v takovém případě je uchována až do daného data. Pro mazání cookies okamžitě stačí nastavit expires atribut na uplynulé datum.
    - Omezení pouze na HTTP (HttpOnly) – původně výborný nápad, který ale nebyl implementován stejně všemi prohlížeči, a tak jeho využití v praxi pokulhává. Pokud byl tento atribut nastaven, cookie nemohla být přečtena JavaScriptem skrz *document.cookie*, ale pouze serverem. Tak mělo být zamezeno zcizení cookies využitím XSS.
- **Flash security model** – protože se Flash stal populárním pluginem téměř všech prohlížečů a Flash vždy obsahoval chyby s možností zneužití, vznikl bezpečnostní model pro Flash. S nástupem HTML5 je Flash zcela na ústupu, a tak tomuto modelu nebude věnována další pozornost (Cannings, Dwivedi, Lackey, 2008).

5.4.1 Příklad XSS

XSS se dá rozdělit do několika skupin, princip útoku se ale v zásadě neliší.

- Nepersistentní XSS (reflected) – jedná se o takové útoky, kdy je útočný kód vložen na stránku dynamicky, nejčastěji prostřednictvím URL adresy odkazu např. v e-mailu nebo na pochybných či napadených webových stránkách. Nic netušící uživatel na tento odkaz klikne a stránka zranitelná XSS útoky po návštěvě z takového odkazu provede, co útočník zamýšlel (Messier, 2016).
- Persistentní XSS (stored) – stejné útoky jako nepersistentní, jen s tím rozdílem, že škodlivý kód není schovaný v URL adrese, ale útočníkovi se jej podařilo uložit do databáze dané aplikace (např. prostřednictvím nezabezpečeného systému komentářů atp.). Tím pádem není třeba, aby uživatel klikl na odkaz s připojeným skriptem v adrese, ale stačí, aby navštívil stránku, kde se daný záznam z databáze se skriptem zobrazuje – např. článek s komentáři, pokud je skript uložený jako součást komentáře (Messier, 2016).
- DOM-based XSS – jsou velmi podobné XSS formě nepersistentních XSS útoků, s tím rozdílem, že HTML stránka může být statická a skript útočníka není vložen do stránky samotné. Pokud je skript v URL za hashem *(#)*, skript se nikdy nedostane na server, a tak kód na serveru nemůže tuto formu XSS přímo ovlivnit ani detekovat (OWASP, 2017).

Typický příklad XSS útoku je jednoduché vložení skriptu do výstupu PHP skriptu. Tam se dostane buď z URL adresy (pak se jedná o nepersistentní, reflected XSS), nebo z databáze, kam se útočníkovi skript podařilo uložit (v tom případě se jedná o persistentní, stored XSS). Nejjednodušší případ může vypadat tak, jak znázorňuje kód 26.

```
$search = $_GET['search'] ?? null;
echo 'Výsledky hledání pro výraz ' . $search;
```

Kód 26 - Ukázka PHP skriptu, který podléhá nepersistentnímu XSS útoku

Kód 27 je URL adresa pro hledaný výraz „auto“ na webu, který zobrazuje stránku tak, jak znázorňuje kód 26.

```
http://example.com/search.php?search=auto
```

Kód 27 - Ukázka běžné URL adresy a GET parametru skriptu náchylného k XSS

Pokud útočník tuto URL adresu změní tak, jak znázorňuje kód 28, PHP skript zpracovávající zobrazení této stránky (kód 26) zobrazí stránku, která zobrazí *„Výsledky hledání pro výraz“* a místo výrazu, který měl následovat, bude vložen skript *<script>alert('Pozor, XSS')</script>*.

```
http://example.com/search.php?search=<script>alert('Pozor,
XSS')</script>
```

Kód 28 - Ukázka URL adresy s pokusem o jednoduchý XSS útok

Skript se samozřejmě spustí na stránce a v prohlížeči návštěvníka, který se přes upravený odkaz na stránku dostane a zobrazí se mu hláška *„Pozor, XSS“*. To by bylo od útočníka zcela zbytečné, takže se v praxi stane spíše to, že tento skript bude důmyslnější. Například přečte nějaká důvěrná data přihlášeného uživatele a odešle je útočníkovi, nebo v horším případě dokonce odešle jeho session cookie a útočník tak získá přístup do dané aplikace.

A přitom řešení je velmi jednoduché, na původní proměnnou *$search* stačí před jejím zobrazením v HTML stránky zavolat funkci *htmlspecialchars()*.

```
$search = htmlspecialchars($_GET['search'] ?? null, ENT_QUOTES |
ENT_HTML5);
echo 'Výsledky hledání pro výraz ' . $search;
```

Kód 29 - Ukázka PHP kódu pro zobrazení HTML stránky, který nepodléhá XSS útokům

PHP.earth (2018) uvádí, že tato funkce převede všechny speciální znaky na jejich příslušné HTML entity a tak se skript nikdy nemůže spustit, neboť např. znak „<“ je převeden na *<*, znak „>“ pak na *>*. Platný skript ale musí začínat skutečným znakem „menší než“, nikoliv jeho HTML entitou – a tak je XSS úspěšně zabráněno. Druhý parametr *ENT_QUOTES* je použitý k tomu, aby byly escapovány i apostrofy a uvozovky, což je doporučené nastavení escapování.

Přestože toto řešení je dostatečné, není vhodné jej používat takovým způsobem, jak je zde nastíněno, proto se mu věnuje celá samostatná následující kapitola.

5.4.2 Prevence XSS

Jak ukazuje kód 29 v kapitole 5.4.1, předejít XSS je velmi snadné – escapovat data při výstupu. Většinu času není vůbec nutné, aby data pocházející od uživatele nebyla escapována, protože se nejčastěji pouze vypisuje nějaká hodnota do příslušného HTML elementu. Stejně snadné je ale udělat chybu a jednou tento kód zapomenout použít – a ihned je aplikace zranitelná XSS útoky. Z toho důvodu není opakované použití této funkce považováno za zcela bezpečné nebo doporučené řešení. Pokud ale vývojář nezapomene takovou funkci použít (viz kód 30), je dostatečné a užitečné např. v případech, kdy z nějakého důvodu není možné použít lepší řešení (Stock, Gonçalves, Correa, 2017).

```
function xssafe($data,$encoding='UTF-8')
{
   return htmlspecialchars($data, ENT_QUOTES |
ENT_HTML5,$encoding);
}

function xecho($data)
{
   echo xssafe($data);
}
```

Kód 30 - Ukázka PHP funkcí, které zabrání XSS útokům u webových aplikací, kde není možné použít šablonovací systémy

Doporučené řešení spočívá v použití **šablonovacích systémů**, které pří výpisu dat aplikují escapovací funkce automaticky. Veškeré výstupy aplikace projdou skrz šablonovací systém, resp. aplikace pro zobrazení všech svých stránek použije šablonovací systém a pak se nemůže stát, že vývojář zapomene data escapovat. Jediný problém spočívá v tom, že pokud je vstup od uživatelel využíván v elementech jako je *style*, *script*, *src* atribut obrázku, *a* (odkaz), což se zpravidla nestává, není možné tento obsah uchránit před XSS takto jednoduchou metodou a je třeba použít důmyslnější způsob vhodný pro konkrétní aplikaci. V ideálním stavu se vývojář takové situaci zcela vyhne a najde jiné řešení pro daný problém. Zajímavé je, že bezpečnost vůči XSS je jen vedlejší efekt šablonovacích systémů – primární cíl, se kterým vznikly, je vylepšení či usnadnění způsobu psaní stránek s výstupem (*view*), jejich designování. Pokud vývojář chce, aby proměnná byla vypsána bez escapování, musí o to šablonovací systém explicitně požádat (Stock, Gonçalves, Correa, 2017).

Řešení nastíněné výše je ale vhodné jen pro situace, kdy je žádoucí z dat pro výstup odstranit veškeré HTML tagy, resp. není povoleno žádné HTML. To ale neplatí vždy. Může nastat situace, kdy uživatelé mohou využívat HTML značky ve svém vstupu – ať už jde o komentáře na blozích, příspěvky v diskuzních fórech. V takových případech je třeba používat knihovny bezpečného kódování a to je náročnější i z časového hlediska, proto řada aplikací obsahuje XSS zranitelnosti. Možností je ale v tomto případě opět celá řada, ať už OWASP ESAPI, OWASP ANtiSAmmy nebo populární HTMLPurifier přímo pro PHP (Stock, Gonçalves, Correa, 2017).

Jak již bylo uvedeno, ideální prevencí je escapování. Escapování by ale mělo být přizpůsobeno kontextu, ve kterém je daná proměnná vypsána. Platí jiná pravidla pro escapování proměnných v odstavci HTML a jiná pravidla pro výpis proměnných do hodnot atributů elementu odstavce (OWASP, 2017).

Posledním a velmi účinným krokem k potlačení XSS je zavedení CSP (Content Security Policy), tedy pravidel hluboko v jádru aplikace pro bezpečnost obsahu. Tato pravidla mohou omezit, které domény budou považovány za bezpečné a důvěryhodné pro spouštění skriptů a které protokoly mohou být použity – často

to bude pouze HTTPS. Pokud se pak útočník pokusí spustit inline JavaScript nebo JavaScript načtený z jeho domény, prohlížeč to zkrátka neudělá. Po nastavení těchto pravidel je v požadavcích serveru Content-Security-Policy navíc HTTP hlavička, kterou většina dnešních prohlížečů bude dodržovat. Prohlížeče, které by CSP nepodporovaly, by toto pravidlo pouze ignorovaly.

Na závěr kapitoly prevence je vhodné uvést některá další obecná doporučení pro bezpečnost PHP webových aplikací co se prevence XSS týká (Stock, Gonçalves, Correa, 2017):

- Aplikace by neměla mít žádnou sekci, kde se předpokládá, že je bezpečná a útočník se do ní nedostane, a tak nemusí být zabezpečená proti XSS.
- Všechny výskyty volání funkci *echo*, *print* a *printf* by měly být nahrazeny šablonovacím systémem.
- HTTP-Only cookies, byť nejsou podporovány vždy a všude, jsou už nyní *good-practice*, doporučovaný způsob ukládání cookies.
- Aby escapování mohlo fungovat přesně tak, jak je zamýšleno, je nutné dodržovat standardy validního HTML.
- Nepersistentní XSS je stejně nebezpečné jako persistentní XSS.
- Ne každá instalace PHP nutně musí mít povolená rozšíření *mhash* a *mcrypt*, bez nich nebude fungovat hashing / *SHA-256* resp. *AES*.

5.4.3 Řešení XSS v Nette frameworku

Nette framework používá vlastní šablonovací systém nazvaný Latte. Poskytuje vývojáři širokou škálu funkcí, které jsou v Latte zvané makra. Makra pak vývojář může využít ve všech souborech, ve kterých dochází k výpisu dat z webové aplikace - tzv. šablony (proto šablonovací systém). Název souborů Latte šablon musí mít strukturu *nazev-sablony.latte*. Velká výhoda šablonovacího systému Latte spočívá v tom, že kromě běžných výhod specifických pro šablonovací systémy dovoluje psát i „čisté“ PHP, tedy PHP v podobě, jak se běžně používá mimo šablony, musí být však mezi značkami *{php* a *}*, nikoliv *<?php* a *?>*, jak je v PHP zvykem. Obsah všech šablon je kompilován do PHP kódu při každé změně obsahu šablony. To znamená, že obsah

každé šablony je pomocí speciálních funkcí kompilátoru transformován do nativního PHP kódu. Pro optimalizaci výkonu aplikace je možné v produkčním prostředí funkci kontroly aktuálnosti kompilované šablony vypnout a tím je znegována hlavní nevýhoda šablonovacích systémů.

Výpis proměnných, tedy hlavní situaci, kdy se XSS útoky především hlásí o slovo, se v Latte šablonách Nette frameworku provádí jedním jediným způsobem. Kód 31 ukazuje, jak proměnnou v šabloně vypsat. V tomto případě je proměnná bezpečně escapována.

```
{$promenna}
```

Kód 31 - Ukázka výpisu proměnné v Latte šablonovacím systému frameworku Nette

Výjimečně mohou nastat situace, kdy je escapování nežádoucí. V tom případě je třeba přidat filtr noescape, viz kód 32 pro názornou ukázku.

```
{$promenna|noescape}
```

Kód 32 - Ukázka výpisu proměnné bez escapování v Latte šablonovacím systému frameworku Nette

Navíc je Latte tak důmyslné, že dokáže rozpoznat kontext, ve kterém je makro použito. To znamená, že kompilátor detekuje, kde přesně v šabloně se makro pro vypsání obsahu proměnné nachází.

1. Pokud je to mezi dvěma HTML značkami, dochází ke standardnímu escapování proměnné pomocí *htmlspecialchars()* funkce a většina zvláštních znaků v proměnné je převedena na své HTML entity, jak ukazuje kód 33.

```
public static function escapeHtml($s)
{
	return htmlspecialchars((string) $s, ENT_QUOTES, 'UTF-8');
}

public static function escapeHtmlText($s)
{
	return $s instanceof IHtmlString || $s instanceof
\Nette\Utils\IHtmlString
		? $s->__toString(true)
		: htmlspecialchars((string) $s, ENT_NOQUOTES, 'UTF-
8');
}
```

Kód 33 - Ukázka escapování v Latte, proměnné vypsané mezi dvě HTML značky

2. Je-li výpis proměnné na místě, kde bývá hodnota atributu HTML elementu, tedy např. jako hodnota atributu alt u obrázku *<img alt="{$promenna}">*, je escapování složitější. Hodnota k vypsání je převedena na proměnnou typu string (řetězec) a pokud se vyskytnou znaky „`" nebo mezera, znaménka větší či menší, uvozovky, zpětné lomítko nebo apostrof, jsou nahrazeny mezerou. K tomu všemu dojde ke standardnímu escapování pomocí *htmlspecialchars()* funkce jako v předchozím případě. Tato funkce je vyobrazena v kódu 34.

```
public static function escapeHtmlAttr($s, $double = true)
{
	$double = $double && $s instanceof IHtmlString ? false :
$double;
	$s = (string) $s;
	if (strpos($s, '`') !== false && strpbrk($s, ' <>"\'') ===
false) {
		$s .= ' ';
	}
	return htmlspecialchars($s, ENT_QUOTES, 'UTF-8', $double);
}
```

Kód 34 - Ukázka escapování v Latte, proměnné vypsané do hodnot atributů HTML elementů

3. V případě výpisu proměnné na místě názvu atributu HTML elementu, je hodnota proměnné opět převedena na string. Pokud obsahuje pouze alfanumerické znaky, je přímo vypsána, pokud i další znaky, je escapována jako běžný obsah. Viz kód 35.

```
public static function escapeHtmlAttrUnquoted($s)
{
	$s = (string) $s;
	return preg_match('#^[a-z0-9:-]+$#i', $s)
		? $s
		: '"' . self::escapeHtmlAttr($s) . '"';
}
```

Kód 35 - Ukázka escapování v Latte proměnných vypsaných jako název atributů HTML elementů

4. Pokud je proměnná k vypsání na místě v HTML komentáři, je escapována dost specifickým způsobem, kdy jsou některé znaky přímo nahrazeny, jiné nahrazeny jen tehdy, pokud se vyskytují na začátku řetězce. Vše je zobrazeno v kódu 36.

```
public static function escapeHtmlComment($s)
{
	$s = (string) $s;
	if ($s && ($s[0] === '-' || $s[0] === '>' || $s[0] === '!'))
{
		$s = ' ' . $s;
	}
	$s = str_replace('--', '- - ', $s);
	if (substr($s, -1) === '-') {
		$s .= ' ';
	}
	return $s;
}
```

Kód 36 - Ukázka escapování v Latte, proměnné vypsané v HTML komentářích

5. Výpis proměnné v XML šabloně se řídí striktně standardy XML a zakázané znaky jsou z původní hodnoty proměnné přímo odebrány, ukazuje kód 37.

```
public static function escapeXml($s)
{
	return htmlspecialchars(preg_replace('#[\x00-
\x08\x0B\x0C\x0E-\x1F]+#', '', (string) $s), ENT_QUOTES, 'UTF-8');
}
```

Kód 37 - Ukázka escapování Latte, proměnné vypsané v XML šablonách

6. V XML názvu atributu je při escapování dodržen analogický postup jako v HTML názvech atributu. Jediný rozdíl je v tom, že je tento postup doplněn o escapovací pravidla pro XML z předchozího příkladu. Viz kód 38.

```
public static function escapeXmlAttrUnquoted($s)
{
	$s = (string) $s;
	return preg_match('#^[a-z0-9:-]+$#i', $s)
		? $s
		: '"' . self::escapeXml($s) . '"';
}
```

Kód 38 – Ukázka escapování Latte, proměnné vypsané jako atribut v XML šablonách

7. V CSS šabloně je escapování potřeba také. Latte využívá funkci *addcslashes()*, s jejíž pomocí escapuje všechny znaky vypsané jako druhý parametr této funkce v kódu 39.

```
public static function escapeCss($s)
{
	return addcslashes((string) $s,
"\x00..\x1F!\"#$%&'()*+,./:;<=>?@[\\]^`{|}~");
}
```

Kód 39 - Ukázka escapování Latte, proměnné vypsané v CSS šablonách

8. Pokud se makro nachází v JavaScriptu, tedy uvnitř *<script></script>* značek, escapování proběhne kódováním hodnoty proměnné do JSON formátu a nahrazením některých znaků znaky jinými. Vše znázorňuje kód 40.

```
public static function escapeJs($s)
{
	if ($s instanceof IHtmlString || $s instanceof
\Nette\Utils\IHtmlString) {
		$s = $s->__toString(true);
	}

	$json = json_encode($s, JSON_UNESCAPED_UNICODE);
	if ($error = json_last_error()) {
		throw new \RuntimeException(PHP_VERSION_ID >= 50500
? json_last_error_msg() : 'JSON encode error', $error);
	}

	return str_replace(["\xe2\x80\xa8", "\xe2\x80\xa9", ']]>',
'<!'], ['\u2028', '\u2029', ']]\x3E', '\x3C!'], $json);
}
```

Kód 40 - Ukázka escapování Latte, proměnné vypsané jako proměnné v JavaScriptu

9. Jestliže dochází k výpisu proměnné přímo v elementu *script* nebo *style*, je žádoucí, aby vypsaná hodnota neukončila aktuální otevřený tag, ať *script* nebo *style*. Pokud se v části řetězce manipulace s tímto tagem objeví, je z obsahu proměnné odstraněna a nevypsána, jak je vidět v kódu 41.

```
public static function escapeHtmlRawText($s)
{
	return preg_replace('#</(script|style)#i', '<\\/$1',
(string) $s);
}
```

Kód 41 - UKázka escapování Latte, proměnné vypsané jako text do script nebo style elementů

10. Přestože to není typické použití šablon, Latte počítá i s escapováním proměnných vypsaných v šabloně formátu *iCal*. Nahrazuje nepovolené části řetězce jinými, ukazuje kód 42.

```
public static function escapeICal($s)
{
       return addcslashes(preg_replace('#[\x00-\x08\x0B\x0C-
\x1F]+#', '', (string) $s), "\";\\,:\n");
}
```

Kód 42 - Ukázka escapování Latte, proměnné vypsané jako text v šabloně iCal

Na základě kontextu Latte volí nejvhodnější escapovací strategii, jak ukázky kódu 33 – 42 ukazují. Toto je velmi zdařilá a pokročilá funkcionalita a přitom funguje zcela automaticky. Laravel tuto funkcionalitu nemá a ani u jiných frameworků není rozšířená. Právě zde Nette ukazuje svoji silnou stránku, důraz na bezpečnost zejména co se týká XSS.

5.4.4 Řešení XSS v Laravel frameworku

Laravel také používá vlastní šablonovací systém. Byl pojmenován Blade. Blade dává vývojářům k dispozici řadu dodatečných funkcí využitelných v šablonách. Šablony musí mít podobu názvu souboru a přípon *nazev-souboru.blade.php*. Velká výhoda šablonovacího systému Blade spočívá v tom, že kromě běžných výhod specifických pro šablonovací systémy dovoluje psát i „čisté“ PHP, tedy PHP v podobě, jak se běžně používá mimo šablony, měl by však být mezi značkami *@php* a *@endphp*, nikoliv *<?php* a *?>*, jak je v PHP zvykem. Veškerý obsah blade souborů je kompilovaný do nových souborů s ryzím PHP kódem a pouze s příponou *.php*, část *.blade* zde již chybí. K nové kompilaci dochází vždy, když se obsah souboru Blade šablony změní. Dokud zůstává obsah šablony nezměněný, šablona se znovu nekompiluje. Díky tomu užití šablonovacího systému nemá téměř žádný dopad na výkon aplikace. To platí za předpokladu, že Laravel aplikace běží v tzv. vývojovém režimu. V tomto režimu v aplikaci běží více kódu (např. při kontrole, zda zkompilovaný soubor šablony existuje, se navíc oproti produkčnímu režimu kontroluje, zda se původní kód šablony nezměnil) a vypisují se chybové hlášky, které jsou pak v produkčním

režimu skryty – jednak jsou běžnému uživateli nesrozumitelné a jednak se informací v nich dá zneužít.

XSS se v Blade šabloně týká výpis proměnných. Proměnná může být vypsána několika způsoby. Standardní způsob zobrazený v kódu 43 přepokládá využití dvojitých složených závorek, které jsou pak zkompilovány do volání funkce *e()*, která pak aplikuje funkci *htmlspecialchars()*.

```
{{ $promenna }}
```

Kód 43 - Ukázka výpisu proměnné v Blade šablonovacím systému frameworku Laravel

V případě, že vývojář chce vědomě vypsanou proměnou neescapovat, použije složenou závorku se dvěma vykřičníky. To je velmi příhodné označení vzhledem k potencionálnímu riziku, které za použitím této funkcionality stojí. Příkladem budiž kód 44.

```
{!! $promenna !!}
```

Kód 44 - Ukázka výpisu proměnné bez escapování v Blade šablonovacím systému frameworku Laravel

Blade je kompilován velmi čitelným způsobem a je vhodné podívat se, jak kompilace dvou volání z kódu 43 a 44 vypadá. Escapovaný výstup, tedy výstup z kódu 43, je kompilován následující funkcí v kódu 45. Podstatné je, že výstup obsahuje volání *echo e($promenna)*, escapovací funkce e je tedy skutečně automaticky zavolána.

```
protected function compileEscapedEchos($value)
{
	$pattern = sprintf('/(@)?%s\s*(.+?)\s*%s(\r?\n)?/s', $this->escapedTags[0], $this->escapedTags[1]);

	$callback = function ($matches) {
		$whitespace = empty($matches[3]) ? '' : $matches[3].$matches[3];

		return $matches[1] ? $matches[0] : "<?php echo e({$this->compileEchoDefaults($matches[2])}); ?>{$whitespace}";
	};

	return preg_replace_callback($pattern, $callback, $value);
}
```

Kód 45 - Funkce Laravel frameworku kompilující značky {{ }} pro výpis proměnných do PHP

Kód 44 by při výstupu escapovaný být neměl. Kompilace probíhá téměř totožně jako v předchozím případě, hlavní a jediný rozdíl ve výstupu spočívá v tom, že výstup neobsahuje volání escapovací funkce *e()*, jak ukazuje tučná část kódu 46.

```
protected function compileRawEchos($value)
{
	$pattern = sprintf('/(@)?%s\s*(.+?)\s*%s(\r?\n)?/s', $this->rawTags[0], $this->rawTags[1]);

	$callback = function ($matches) {
		$whitespace = empty($matches[3]) ? '' : $matches[3].$matches[3];

		return $matches[1] ? substr($matches[0], 1) : "<?php echo {$this->compileEchoDefaults($matches[2])}; ?>{$whitespace}";
	};

	return preg_replace_callback($pattern, $callback, $value);
}
```

Kód 46 - Funkce Laravel frameworku kompilující značky {!! !!} pro neescapovaný výpis proměnných do PHP

Samotná escapovací funkce v kódu 47 pak opravdu volá doporučenou funkci *htmlspecialchars()*, která nahrazuje potenciálně závadné znaky jejich HTML entitami. Výsledkem je, že všechny výpisy proměnných, u kterých není explicitně určeno, aby byly vypsány bez escapování, escapovány jsou a je to tak nejbezpečnější (Cannings, Dwivedi, Lackey, 2008).

```
function e($value)
{
	if ($value instanceof Htmlable) {
		return $value->toHtml();
	}

	return htmlspecialchars($value, ENT_QUOTES, 'UTF-8', false);
}
```

Kód 47 - Funkce Laravel frameworku volaná na všechny escapované výstupy dat pro prevenci XSS

5.5 Cross-site request forgery (CSRF, XSRF)

CSRF útokům se také říká XSRF, „*útok jedním kliknutím*“ nebo „*session riding*“. Podstata těchto útoků spočívá v tom, že útočník prostřednictvím aktuálně přihlášeného uživatele webové aplikace v ní provede nechtěné operace. CSRF útoky se zaměřují na změnu stavu (přesun financí, změna e-mailové adresy či hesla uživatele apod.) ve webové aplikaci, nikoliv krádež dat. Zcizení dat není z principu CSRF útoků možné, útočník totiž nemůže nijak vidět odpověď na požadavek odeslaný na server (OWASP, 2017).

Acunetix (2018) píše, že útoky cross-site request forgery jsou považovány za velmi mocný, ale nepříliš často používaný nástroj útočníků a často jim není věnována dostatečná pozornost z bezpečnostního hlediska. V OWASP seznamu deseti největších hrozeb se CSRF útoky dlouhodobě pohybovaly a teprve v loňském roce byly ze seznamu odstraněny právě z důvodu nepříliš častého výskytu. To však neznamená, že riziko pominulo. Není pochyb o tom, že XSS útoky jsou mnohem větším bezpečnostním rizikem a to z důvodu, že při XSS útocích může útočník získat odpověď serveru, což při CSRF útocích neplatí. Proto lze XSS používat pro odcizení citlivých dat. Druhým omezením CSRF útoků je skutečnost, že jejich prostřednictvím lze ve webové aplikaci dělat jen takové změny, které dovolí práva přihlášeného uživatele. Povede-li se útočníkovi CSRF útok na běžného uživatele aplikace, možnosti škody budou značně omezené. Pokud se však takový útok útočníkovi podaří uskutečnit prostřednictvím administrátora, možná rizika jsou mnohem větší. I z důvodu bezpečnosti vznikly různé úrovně přístupových práv pro různé uživatele a zde se ukazuje, že to bylo moudré rozhodnutí.

Existují dva základní typy CSRF útoků (Gollmann, 2011):

1. Nepersistentní CSRF (reflected CSRF) – v případě tohoto typu útoku útočník nějakým způsobem donutí oběť (většinou prostřednictvím sociálního inženýrství, což je zjednodušeně řečeno zmanipulování oběti do provedení určité akce) kliknout na nějaký odkaz nebo načíst nějakou stránku. Je nezbytné, aby v tu chvíli byla oběť na dané stránce aktuálně přihlášená. Pak zbývá už jen odeslat útočníkův požadavek na server, což se stane

automaticky poté, co prohlížeč oběti stránku načte. Server takový požadavek přijme, protože si myslí, že pochází od přihlášené oběti.

2. Persistentní CSRF (stored CSRF) – v tomto případě CSRF útoku je na serveru uložená zlomyslná stránka, resp. její obsah obsahuje kód útočníka. Nejde o nic složitého, stačí na první pohled nevinná HTML značka *img* nebo *iframe* s atributem *src* obsahujícím URL, která provede nežádoucí GET požadavek na server. Když oběť navštíví takovou stránku, požadavek je okamžitě vyvolán a pokud má uživatel dostatečná oprávnění na akci, která se za požadavkem skrývá, je provedena a to pokaždé, když uživatel stránku načte.

Při odeslání požadavku na server webové aplikace, kde je uživatel přihlášen, ve většině případů prohlížeč automaticky připojí autentizační a autorizační údaje přihlášeného uživateli k požadavku. To zahrnuje session cookie uživatele, IP adresu a popř. i některé další údaje. Požadavek je tak považován za legitimní požadavek tohoto uživatele, přestože to byl útočník, kdo zvolil, jaký požadavek bude na server odeslán a v konečném důsledku jaká akce ve webové aplikaci bude provedena. V takovém případě aplikace ani server opravdu nemá šanci rozpoznat, zda jde o legitimní požadavek od oběti, nebo požadavek, který je dílem CSRF útoku. K tomu je třeba tajný token (náhodný řetězec znaků), o kterém ví jen prohlížeč a server (OWASP, 2017).

5.5.1 Příklad GET metody CSRF útoků

Požadavky na server typu GET by měly být ze své podstaty idempotentní, to znamená, že by žádný GET požadavek neměl mít za následek žádnou stavovou změnu. Změna stavu v aplikaci znamená změnu dat. Toto je ideální stav a doporučení, jak by to mělo být, aplikace se tohoto pravidla ale nemusí držet a GET požadavky měnit data mohou – přidávat uživatelské účty, měnit hesla apod.

Za předpokladu, že na webu *aplikace.cz* je webová aplikace, ve které jsou uživatelské účty a každý má své finanční konto, kód 48 zobrazuje URL adresu GET požadavku, která na účet „Petr“ převede částku 999999 jednotek z účtu aktuálně přihlášeného uživatele.

```
https://aplikace.cz/prevod?castka=999999&account=Petr
```

Kód 48 - Ukázka GET požadavku v imaginární aplikaci pro převod peněz mezi účty

Útočník může navést své oběti na svoji vlastní webovou aplikaci, která může mít totožný design jako aplikace, na které se pokouší o CSRF útok. Pouhé navštívení této aplikace pak může v prohlížeči oběti vyvolat výše zmíněný GET požadavek na server skutečné webové aplikace (jak, ukazuje kód 49) a obrat oběť o částku specifikovanou v odkazu ve prospěch účtu taktéž specifikovaného v odkazu. Oběť přitom vůbec nemusí zjistit, že k takovému požadavku došlo a přitom web server tento požadavek považuje za požadavek iniciovaný obětí. Cookie přihlášeného uživatele, pomocí které server ověří, že přihlášený uživatel je ten, za koho se vydává, je totiž k požadavku připojena automaticky, jak uvádí Acunetix (2018).

```
<img src="https://aplikace.cz/prevod?castka=999999&ucet=Petr"
style="width:1px;height:1px">
```

Kód 49 - Ukázka img HTML tagu, který může automaticky provést CSRF útok metodou GET

Zbývá zodpovědět, jak se stane, že požadavek na server se odešle automaticky, ihned jak oběť navštíví útočníkovu webovou aplikaci (může to být jen jedna HTML stránka). Messier (2016) píše, že prohlížeče automaticky načítají všechny obrázky na stránce. Toho se dá velmi snadno zneužít a místo validní cesty k obrázku lze do *src* atributu umístit odkaz, který iniciuje škodlivý GET požadavek a tím automaticky dokončí CSRF útok, jak ukazuje kód 49. Cookie přihlášeného uživatele je i v tomto případě automaticky odeslána spolu s požadavkem. Pokud je žádoucí útok co nejvíce maskovat, je možné dát obrázku styl výšky i šířky 1 pixel, čímž se obrázek stane prakticky neviditelným. V případě, že by požadavek musel být typu POST, by toto řešení nebylo možné, protože prohlížeče obrázky načítají výhradně prostřednictvím GET požadavků.

5.5.2 Příklad POST metody CSRF útoků

Provést CSRF útok prostřednictvím GET požadavků je poměrně snadné, proč tedy cross-site request forgery útoky provádět prostřednictvím POST požadavků? Právě proto, že většina požadavků, které změní stav aplikace, změní data v aplikaci, je prováděna právě prostřednictvím POST požadavků. Taková byla hlavní myšlenka POST požadavků – GET k prohlížení, POST ke změně stavu. To pro útočníka ale není nijak zásadní změna, nejpodstatnější rozdíl mezi GET a POST CSRF je v tom, že parametry a jejich hodnoty při odeslání požadavku na server nebudou uloženy v URL, ale přímo v těle samotného požadavku.

Scénář se částečně opakuje, útočník vláká svoji oběť na svoji stránku nebo aplikaci vystavenou na webu. V tuto chvíli měl útočník hotovo, neboť prostřednictvím *img* HTML značky se CSRF útok automaticky dokončil. V případě POST metody CSRF útoku je situaci trochu složitější, protože k požadavku je třeba připojit tělo s formulářovými daty. Nejvhodnější cestou pro útočníka je použití JavaScriptu, který právě toto udělá automaticky po načtení stránky jako u GET metody CSRF útoku. Jak takový skript může vypadat, ukazuje kód 50 (Acunetix, 2018).

```
<body onload="document.csrf.submit()">

<form action="https://aplikace.cz/ prevod " method="POST"
name="csrf">
       <input type="hidden" name="castka" value="999999">
       <input type="hidden" name="ucet" value="Petr">
</form>
```

Kód 50 – Ukázka JavaScriptu, který může automaticky provést CSRF útok metodou POST

Stránka útočníka obsahuje skrytý formulář, resp. formulář se skrytými poli, která definují parametry a jejich hodnoty nastavené dle chuti útočníka. Tyto parametry a jejich hodnoty budou JavaScriptem odeslány na server prostřednictvím POST požadavku. V atributu *onload body* HTML elementu je skript, který formulář odešle. Atribut *onload* definuje skript, který se spustí automaticky po načtení stránky, i tento CSRF útok je tak po vlákání oběti na stránku zcela

automatický. Jde použít analogický postup a místo odeslání formuláře načíst *iframe*, ale metodu s formulářem uvádí Acunetix (2018) jako naprosto dostačující.

5.5.3 Prevence CSRF

CSRF je útok, jehož prostřednictvím mohou útočníci napáchat škody. Přesto existují situace, kdy není zcela nutné CSRF předcházet (Afoosteh, Naderi, Hoffmann, Stock a Plant, 2017):

- Stránka, na které by byl CSRF útok proveden, je veřejně dostupná, k akci není třeba přihlášeného uživatele (spamování skrz kontaktní formulář webu apod.). V takovém případě může útočník rovnou provést útok sám, což je rychlejší, spolehlivější i jednodušší řešení.
- Požadavek nedělá žádné změny v systému a operace probíhající za tímto požadavkem jsou rychlé. Pokud požadavek dělá změny v systému, je nutné CSRF předejít. Pokud změny nedělá, obecně je zbytečné CSRF útokům předcházet. To ale neplatí v případě, kdy je operace, kterou požadavek spustí, výpočetně náročná, i v takovém případě je výhodné CSRF nedopustit.

OWASP (2017) vydal řadu doporučení, jak omezit možnosti CSRF útoků:

- Pokud jde o kritickou operaci v aplikaci, je doporučeno **vyžadovat reautentizaci**. To znamená, že se uživatel musí znovu přihlásit nebo dalším způsobem dokázat aplikaci svoji totožnost. Např. v případě podání platebního příkazu na počítači v online bankovnictví je před odesláním platební příkaz potvrdit telefonem, buď přes SMS zprávu nebo v mobilní aplikaci.
- Není-li jasné, zda je CSRF zabráněno v dostatečné míře, je možné přidat *CAPTCHA* (Completely Automated Public Test to tell Computers and Humans Apart). Jedná se o turingův test, který dokáže rozlišit člověka a robota (tedy počítač), např. pomocí vyplnění rychlého kvízu nebo opsání kódu. CSRF útok by musel být mnohem důmyslnější, aby dokázal CAPTCHA obejít. Řešení je to ale nešťastné v tom, že znepříjemňuje práci s aplikací řádným uživatelům.

- Pokud aplikace provádí operace i na základě jiných částí požadavků, např. cookies nebo HTTP hlaviček, je vhodné užít CSRF token i tam.
- Formuláře, které jsou odesílány asynchronně (AJAX) a využívají CSRF tokeny, musí své tokeny po každém odeslání znovu vygenerovat a to prostřednictvím server-side skriptu, nikdy na client-side.
- Používání CSRF ochrany v cookies a GET požadavcích není pro vývojáře pohodlné. Pokud je tato možnost potřeba, je třeba zvážit, zda nezměnit návrh kódu a architekturu aplikace (Kanat-Alexander, 2012).

Hlavním klíčem v prevenci CSRF útoků jsou CSRF (resp. antiCSRF) tokeny, náhodné řetězce, které jsou spjaty s konkrétním uživatelem a jsou součástí formulářů. Jen pokud má pole formuláře s tokenem správnou hodnotu, je požadavek serverem přijat. Pro útočníka je velmi těžké až nemožné hodnotu CSRF tokenu uhodnout. Díky úspěšnosti tohoto řešení se jedná zároveň o řešení nejpopulárnější. Acunetix (2018) uvedl, že podstata tokenů spočívá v tom, že:

1. web server vygeneruje token,
2. token je nastaven jako skryté pole každého formuláře, který má být chráněn proti CSRF, a zároveň uložen do session, aby aplikace mohla token porovnat,
3. uživatel odešle formulář,
4. token je součástí dat z formuláře,
5. aplikace porovná token odeslaný z formuláře s vygenerovaným tokenem, který byl uložen do session,
6. pokud se oba tokeny shodují, požadavek je validní,
7. pokud se tokeny neshodují, požadavek je považován za CSRF útok a odmítnut.

Při práci s tokeny se uplatňují další pravidla. Každý token by měl mít časově omezenou platnost a měl by být zneplatněn, pokud se uživatel z aplikace odhlásí. Token by měl být také kryptograficky bezpečný, aby nemohl být snadno uhodnut, což není pravděpodobné, ale možné ano. V každém případě je doporučeno neexperimentovat s vlastní implementací a využít buď frameworků (**Nette i Laravel CSRF tokeny shodně a správně implementují**), které obsahují

kompletní řešení, nebo se řídit doporučením implementace OWASP (Acunetix, 2018).

Klíčová implementace CSRF tokenu v PHP7 dle doporučení OWASP (2017) je nastíněna níže v kódu 51.

```
session_start();

function store_in_session($key, $value)
{
        if (isset($_SESSION)) {
                $_SESSION[$key]=$value;
        }
}

function unset_session($key)
{
        $_SESSION[$key] = ' ';
        unset($_SESSION[$key]);
}

function get_from_session($key)
{
        if (isset($_SESSION[$key])) {
                return $_SESSION[$key];
        } else {
                return false;
        }
}

function generate_token($unique_form_name)
{
        $token = random_bytes(64);
        store_in_session($unique_form_name, $token);
        return $token;
}




function validate_token($unique_form_name, $token_value)
```

```
{
        $token = get_from_session($unique_form_name);
        if (!is_string($token_value)) {
                return false;
        }
        $result = hash_equals($token, $token_value);
        unset_session($unique_form_name);
        return $result;
}
```

Kód 51 - Doporučená implementace CSRF ochrany prostřednictvím tokenů v PHP7

První tři funkce, *store_in_session()*, *unset_session()* a *get_from_session()* pouze usnadňují práci se session. Klíčové jsou funkce *generate_token()* a *validate_token()*. První jmenovaná vygeneruje náhodný token kryptograficky bezpečným způsobem pomocí PHP funkce *random_bytes()*, uloží jej do session a vrátí jeho hodnotu. Druhá jmenovaná funkce už jen bezpečně ověří shodu tokenu s tokenem v session pro daný formulář. To je velmi podstatné. Pokud by byl v session token ukládaný jen pro daného uživatele nezávisle na formuláři, v případě, že by uživatel otevřel více různých formulářů v dané aplikaci v několika oknech nebo záložkách prohlížeče, pouze poslední otevřený formulář by měl v session uložený token. Tím pádem by jen poslední otevřený formulář šel odeslat, všechny ostatní dříve otevřené by hlásily nesprávný CSRF token.

5.6 Chyby v řízení přístupu (Broken access control)

Řízení přístupu slouží k tomu, aby každý uživatel mohl vykovánat jen takové akce ve webové aplikaci, ke kterým má svolení. Chyby v řízení přístupu jsou podle OWASP (2017) 5. nejdůležitějším bodem v seznamu deseti největších hrozeb. Jedná se o chyby, kde řízení přístupu potažmo funkční systém přístupových práv buď zcela chybí, nebo nefunguje korektně a umožní útočníkovi přístup tam, kam by přístup mít neměl. To vede v nejmírnějším případě k tomu, že se někdo dostane k datům, ke kterým by přístup mít neměl. V horším případě tato data i modifikuje nebo dokonce všechna data smaže. Také se může stát, že bude nepovolaná osoba vykonávat funkce, které neodpovídají jeho přístupovým právům.

Nejčastější zranitelnosti v oblasti chyb v řízení přístupu podle OWASP žebříčku deseti nejkritičtějších zranitelnosti za rok 2017 zahrnují:

- Obcházení kontrol řízení přístupu modifikací URL, vnitřního stavu aplikace, HTML stránky nebo pomocí vlastního API dané aplikace.
- Umožnění změny primárního klíče uživatele. To zapříčiní, že si někdo může prohlížet nebo upravovat aplikaci či profil jménem někoho jiného.
- Chování uživatele jako by měl více přístupových práv, než jich ve skutečnosti má. Může jít o nepřihlášeného uživatele, který se chová, jako by byl přihlášený, nebo o řadového uživatele, který se vydává za administrátora.
- Možnost manipulace metadat, neoprávněné zásahy do JSON Web Tokenu (JWT), který slouží jako autentizační a autorizační token uživatele, neoprávněné zásahy do cookies nebo skrytých formulářových polí s účelem získání vyšších přístupových práv v aplikaci.
- Špatná konfigurace přístupových práv k volání API aplikace (tzv. CORS, Cross-Origin Resource Sharing, mechanismy, které při správné konfiguraci dovolí prohlížeči (nebo jinému user agentovi) provádět požadavky na jiné domény než je doména aplikace samotné.

Velmi jednoduchým **příkladem** takového zneužití chyby v řízení přístupu budiž následující situace. Předpoklady této situace jsou:

- přihlášený uživatel má přístup v aplikaci jen na stránku přihlášení a stránku se články, která se nachází pod URL v kódu 52,
- přihlášený uživatel nemá přístup do nastavení, které se nachází pod URL v kódu 53,
- přístupová práva jsou řešena příliš jednoduše, odkaz do nastavení je pro tohoto uživatele skrytý, ale kód stránky samotné už nekontroluje, zda uživatel, který si chce nastavení prohlížet nebo je upravovat, má příslušná přístupová práva.

```
https://aplikace.cz/clanky
```

Kód 52 - Příklad URL adresy ve fiktivní aplikaci pro stránku se články

```
https://aplikace.cz/nastaveni
```

Kód 53 - Příklad URL adresy ve fiktivní aplikaci pro stránku s nastavením

Pokud tento uživatel nějakým způsobem, ať už uhodnutím, přečtením z obrazovky kolegy nebo jiným způsobem zjistí, jaká je URL stránky s nastavením aplikace, může ji zadat do prohlížeče. Pokud bude uživatel přihlášen a budou platit předpoklady uvedené výše, dostane se na stránku nastavení aplikace i přes to, že mu nebyl udělen přístup.

Prevence chyb v přístupových právech je naprosto nezbytná. Přístupová práva jsou efektivní jen tehdy, pokud je vynucuje kód na serveru (nikoliv v prohlížeči) a když útočník nemůže změnit, jak jsou přístupová práva kontrolována nebo metadata spjata s řízením přístupových práv. Spolu s těmito doporučeními OWASP (2017) dále doporučuje:

- Přístup ke všemu, co nemá být veřejně dostupné, musí být ve výchozím stavu nepřístupné.
- Systém řízení přístupových práv by měl být implementován pouze jednou a používán v celé aplikaci.
- Obecně by přístupová práva měla být modelována tak, že uživatel může manipulovat se záznamy, které sám vytvořil nebo se na nich jinak podílel (vztah prostřednictvím cizích klíčů k danému modelu). Opakem tohoto modelu je model, kde uživatel může prohlížet, vytvářet, editovat nebo mazat libovolné záznamy daného typu.
- Specifické požadavky doménových modelů na přístupová práva by měl zajišťovat každý model sám.
- Je důležité zakázat „web server directory listing“, funkci, která dovolí zobrazit soubory v adresářích na serveru. Také soubory s metadaty a adresáře (např. adresáře *.git* nebo *.idea*, zpravidla začínající tečkou, by neměly být v kořenovém adresáři webového serveru)

- Každé selhání řízení přístupových práv musí být logováno a administrátoři upozorněni, že k porušení přístupových práv došlo.
- API by mělo implementovat rate limiting, tedy omezovat, kolik požadavků může být za určitou dobu provedeno (zpravidla za sekundu, za hodinu a za kalendářní měsíc).
- JWT tokeny by měly být zneplatněny po odhlášení uživatele.

Ve frameworku Nette není připravené žádné vestavěné řešení a tak je na vývojářích, aby implementovali své vlastní řešení dodržující výše zmíněná doporučení. Ve frameworku Laravel je připravena sada funkcí na ověřování přístupových práv a model vyšší úrovně s názvem *Gate*. Tento model po vývojářích vyžaduje implementaci konkrétních přístupových práv pro každý model v aplikaci, který má podléhat přístupovým právům. K tomu slouží tzv. „Policy classes".

5.7 File inclusion

Útoky typu local nebo remote file inclusion, tedy lokální nebo vzdálené vložení souboru, jsou útoky, kdy útočník zmanipuluje webovou aplikaci tak, že načte buď jiný soubor ze serveru, na kterém webová aplikace běží, nebo úplně jiný soubor z některého vzdáleného serveru útočníka. Acunetix (2018) říká, že tento útok může nastat jen tehdy, když aplikace využívá volání *include* nebo *require* pro načtení dalšího PHP skriptu na základě nefiltrovaného a nevalidovaného vstupu od uživatele.

Jak vypadá kód aplikace náchylné na útok local file inclusion, zobrazuje kód 54, v případě náchylnosti na remote file inclusion pak kód 55 (PHP.earth).

```
$file = $_GET['file'] ?? 'home';

include 'adresar/' . $file . '.php';
```

Kód 54 - Ukázka kódu aplikace náchylné na local file inclusion útoky

```
$file = $_GET['file'] ?? 'home.php';
```

```
include $file;
```

Kód 55 - Ukázka kódu aplikace náchylné na remote file inclusion útoky

Oba tyto kódy jsou velmi podobné. V obou případech je do proměnné *$file* načten název souboru z GET proměnné požadavku. Tento soubor je následně načten do PHP skriptu. V prvním případě, při local file inclusion, je kód v ukázce více omezující, načítá se pouze název souboru bez přípony, ta je pevně nastavena na *.php*. Ve druhém případě, při remote file inclusion, není přípona přednastavena, takže tento způsob napsání stejného kódu dává útočníkovi více prostoru pro jeho akce. Fakt, že jsou v prvním případě povoleny pouze PHP soubory, na bezpečnostních rizicích skriptu nic nemění. Útočník může nastavit jakoukoliv hodnotu a načíst tak jakýkoliv soubor. Rozdíl mezi local a remote metodami file inclusion útoku je jen v tom, že v local file inclusion je již přednastavená část cesty, takže není možné načíst skript z jiného serveru. V druhém případě ta možnost je a možnost načíst jiný lokální skript zůstává (Acunetix, 2018).

Pro úplnost je vhodné doplnit, jaké potenciální hodnoty může útočník vepsat do GET parametru pro úspěšný útok. Dva příklady ukazují kódy 56 a 57.

```
https://aplikace.cz/index.php?page=http://utocnik.net/utocny-skript
```

Kód 56 - Ukázka hodnoty, kterou je vhodné použít pro file inclusion útok

```
https://aplikace.cz/?file=../../../../etc/passwd
```

Kód 57 - Ukázka hodnoty, kterou je vhodné použít pro file inclusion útok a directory traversal útok zároveň

V kódu 56 útočník využívá remote file inclusion útok a načítá skript ze svého serveru, který pak udělá, co útočník chce. Může smazat některé soubory a aplikaci tak vyřadit z provozu, nebo např. nahrát některý svůj soubor a získat tím trvalý přístup na server.

V kódu 57 se jedná o local file inclusion útok a direktory traversal útok zároveň, protože útočník načítá soubor mimo adresář webu. Ze souboru *passwd* získá seznam uživatelů na serveru. Útok ale vždy může přizpůsobit své potřebě a dalším cílům (Acunetix, 2018).

Prevencí proti file inclusion útokům je zcela se vyhnout dynamickému načítání souborů na základě vstupu od uživatele. To ale bohužel není vždy schůdné řešení, u starších aplikací to může znamenat nutnost jejich kompletního přepsání. Existují naštěstí i další možnosti, které tento problém řeší. Základem řešení remote file inclusion je zakázání direktivy *allow_url_fopen* a *allow_url_include*, tzn. nastavení jejích hodnot v php.ini souboru na *off*. Tím server-side část PHP aplikace kompletně přijde o možnost načítat data ze vzdálených serverů a to je pro bezpečnost žádoucí. Pro kompletní řešení a pokrytí local file inclusion útoků je nevyhnutelné vedení white listu, tj. seznamu povolených hodnot, kterých parametr může nabývat.

Nette i Laravel frameworky se dynamickému načítání souborů na základě vstupu zcela vyhýbají a při jejich používání tak nebezpečí v tomto ohledu nehrozí.

5.8 Path injection, command injection, code injection

Útoky path injection, také zvané directory traversal nebo *../ (dot dot slash)* útoky, jsou velmi podobné file inclusion a často mají v postupu útočníka přímou návaznost. Jde o http útok, který dovolí útočníkovi přistupovat do adresářů, kam by vůbec neměl mít přístup. K tomu stačí použít znaky „../“ v cestě k nějakému souboru (Acunetix, 2018).

PHP.earth (2018) považuje command injection za stejný typ útoku. Jediný rozdíl spočívá v tom, že neošetřený vstup je vložený do volání příkazů pomocí PHP funkce *exec()*. Ten útočníkovi dává možnost spouštět příkazy mimo kořenový adresář webového serveru. Příklad ukazuje kód 58. Code injection je taktéž podobný útok, neošetřený vstup od uživatele je vložený do volání funkce *eval()*, která převede řetězec znaků na PHP kód a spustí jej. Příkladem je kód 59.

```
exec('rm -rf '.$GET['path']);
```

Kód 58 - Příklad PHP kódu aplikace podléhající command injection útokům

```
eval('include '.$_GET['path']);
```

Kód 59 - Příklad PHP kódu aplikace podléhající code injection útokům

Path, command i code injection útoky vyžadují, aby byla cesta k určitému souboru načítána dynamicky ze vstupu od uživatele a aby tento vstup nebyl nijak validován ani filtrován. V praxi takový kód podléhající direktory traversal útoku může vypadat např. tak, jako v ukázce kódu 60.

```
$page = $_GET['page'] ?? 'home.php';

require $page;
```

Kód 60 - Ukázka kódu aplikace podléhající directory traversal útoku

V GET parametru v URL se může nacházet parametr *page*. Pokud tam není, načte se domovská stránka ze skriptu *home.php*. Pokud v URL je, načte se taková stránka, kterou parametr určí. Např. kód 61 ukazuje, pod jakou URL se nachází stránka kontaktů.

```
https://aplikace.cz/index.php?page=contacts.php
```

Kód 61 - Ukázka URL, která načte stránku s kontakty

Jak PHP.earth (2018) uvádí, útočník může hodnotu *contacts.php* přepsat na *„../secret“* nebo třeba *„/var/www/secret“*. Takový případ ukazuje kód 62. Tím se může dostat k souborům mimo kořenový adresář web serveru, ke kterým by rozhodně neměl mít přístup.

```
https://aplikace.cz/index.php?page=../secret
```

Kód 62 - Ukázka URL, která načte soubor mimo root web serveru a může tak způsobit directory traversal útok

Prevence proti tomuto typu útoku je nasnadě. Nejlepším způsobem je zcela se vyhnout dynamickému načítání souborů na základě vstupu od uživatele. Pokud to není možné, aplikace by měla udržovat whitelist souborů, které mohou být dynamicky načteny a efektivně tak filtrovat vstup od uživatele. Pro kompletní řešení je možné zároveň kontrolovat, zda se útočník nesnaží do názvu souboru k načtení vložit znaky „../“ nebo „file://“ a white list tak obejít (PHP.earth, 2018). Pokud dynamické načítání souborů na základě vstupu od uživatele v aplikaci musí být, zabezpečení zahrnující výše popsané techniky názorně ukazuje kód 63.

```
$page = $_GET['page'] ?? 'home';

if (strstr($_GET['page'], '../') !== false) {
    throw new \Exception("Directory traversal útok!");
}

if (strstr($_GET['page'], 'file://') !== false) {
    throw new \Exception("Remote file inclusion útok!");
}

$whiteList = ['home', 'blog', 'gallery', contacts];
$page = (in_array($page, $ whiteList)) ? $page : 'home';

Require '../pages/' . $page . '.php';
```

Kód 63 - Ukázka PHP implementace zabezpečení proti directory traversal a file inclusion útokům

Nette i Laravel frameworky se dynamickému načítání souborů na základě vstupu zcela vyhýbají a při jejich používání tak nebezpečí v tomto ohledu nehrozí.

5.9 Public files

Public files zranitelnost spočívá v tom, že soubory, které nemají být veřejně dostupné, veřejně dostupné jsou. Ať už jde o soubory v podsložce, která má být veřejně nedostupná, nebo soubory mimo kořenový adresář veřejné části aplikace (většinou adresář *www* nebo *public*).

Příkladem může být struktura aplikace napsané ve frameworku Laravel v kódu 64.

```
app/
boootstrap/
config/
database/
public/
  .htaccess
  index.php
  robots.txt
  web.config
resources/
routes/
storage/
tests/
vendor/
.env
.giattributes
.gitignore
.htaccess
artisan
composer.json
composer.lock
package.json
phpunit.xml
server.php
webpack.mix.js
```

Kód 64 - Struktura adresářů aplikací napsaných ve frameworku Laravel

Zdrojový kód aplikace znamená všechny soubory a adresáře v kódu 64. Veřejná část aplikace je pouze obsah adresáře *public*. V souboru *.env* je nastavení připojení do databáze nebo SMTP nastavení e-mailu včetně uživatelských jmen a hesel. Řada

dalších důvěrných nastavení je v adresáři *config*. Proto je naprosto nutné, aby *.env* soubor i adresář *config* nebyly volně dostupné online. V souborech *composer.json, composer.lock a package.json* se útočník může dočíst o tom, jaké knihovny a jiné produkty třetích stran aplikace využívá. I takovou informaci je dobré skrýt. Proto je nevyhnutelné, aby byl server nakonfigurovaný tak, že veřejně dostupné budou jen a pouze soubory a adresáře v adresáři *public*.

V kořenovém adresáři je soubor .htaccess, ve kterém je třeba nakonfigurovat, aby při přístupu k webovému serveru byl návštěvník přesměrován do adresáře *public*, kde se načte soubor *index.php* a iniciuje celý framework. To lze udělat tak, jak ukazují kódy 65 a 66.

```
<IfModule mod_rewrite.c>
        RewriteEngine On
        RewriteRule ^$ /public/ [L]
        RewriteCond %{REQUEST_FILENAME} !-f
        RewriteCond %{REQUEST_FILENAME} !-d
        RewriteCond %{REQUEST_URI} !^/public/
        RewriteRule ^(.*)$ /public/$1
</IfModule>
```

Kód 65 - Příklad obsahu souboru .htaccess aplikace v Laravel v kořenovém adresáři aplikace, který vede k public files zranitelnosti

```
<IfModule mod_rewrite.c>
        RewriteEngine On
        RewriteBase /
        RewriteCond %{THE_REQUEST} /public/([^\s?]*) [NC]
        RewriteRule ^ %1 [L,NE,R=301]
        RewriteRule ^((?!public/).*)$ public/$1 [L,NC]
</IfModule>
```

Kód 66 - Příklad obsahu souboru .htaccess aplikace v Laravel v kořenovém adresáři aplikace, který řeší zranitelnost public files

Kód 65 skutečně návštěvníka přesměruje do složky *public* a aplikace bude řádně fungovat. Bezpečná ale nebude. Jestliže uživatel zadá URL ukázanou v kódu 67, prohlížeč mu zobrazí obsah *.env* souboru mimo jiné s heslem do databáze.

```
https://aplikace.cz/.env
```

Kód 67 - Příklad URL k pokusu o získání obsahu konfiguračního .env souboru ve frameworku Laravel

Až kód 66 je správné řešení tohoto problému. Návštěvníka webu taktéž správně přesměruje do složky *public* a aplikace bude také řádně fungovat. Rozdíl je v tom, že při zadání URL v kódu 67 dostane návštěvník, v tu tuto chvíli už útočník, odpověď serveru 404 (nenalezeno) namísto obsahu souboru a tak je to správně.

5.10 Nahrávání souborů

Nahrávání souborů (uploading files) na server představuje hned několik bezpečnostních rizik. Každý soubor, se kterým server pracuje jako s HTML souborem, může být použitý pro XSS útok. Každý soubor, se kterým server pracuje jako s PHP souborem, může útočník použít pro extrémně závažný útok – remote execution, obdobu code injection popsanou v kapitole 5.8 (Afoosteh, Naderi, Hoffmann, Stock a Plant, 2017):

```
<form method="post" enctype="multipart/form-data"
action="upload.php">
    File: <input type="file" name="pictures[]" multiple="true">
    <input type="submit">
</form>
```

Kód 68 - HTML formulář pro nahrávání souborů (PHP.earth, 2018)

```
foreach ($_FILES['pictures']['error'] as $key => $error) {
    if ($error == UPLOAD_ERR_OK) {
        $tmpName = $_FILES['pictures']['tmp_name'][$key];

        $name = basename($_FILES['pictures']['name'][$key]);
        move_uploaded_file($tmpName,
"/var/www/project/uploads/$name");
    }
}
```

Kód 69 - Příklad jednoduchého zpracování nahrání souborů v PHP, bez validace (PHP.earth, 2018)

Nahrávat soubory v PHP lze tak, jak ukazují kódy 68 a 69. Tento způsob ale neobsahuje dostatečné zabezpečení. Existuje hned několik opatření, které je třeba při nahrávání souborů provádět. PHP.earth (2018) mezi ně řadí:

- Validace nahraných souborů, která zahrnuje hned několik kroků:
    - Pro přečtení názvu souboru použít funkci *basename()*, předejde se tak directory traversal útokům.
    - Přejmenovávat nahrané soubory na náhodně generované názvy pomocí *microtime()* nebo např. *hash_file()* či *sha1_file()* funkcí. Jednak se tím předejde nežádoucímu přepsání souborů se stejným názvem a taktéž předchází directory traversal útokům. Pokud je třeba uchovat i původní název, lze jej uchovat v databázi.
    - Kontrolovat typy nahraných souborů. Je však důležité nespoléhat na příponu souboru, která může být cíleně změněna. Pro získání typu souborů je třeba užít funkci *finfo_file()*.
    - Kontrolovat nebo limitovat maximální velikost souboru pomocí klíče *size* v příslušném klíči dvoudimenzionálního pole *$_FILES*.
    - Ukládání souborů do veřejně nedostupných adresářů je považováno za *good practice* (dobrý zvyk).
- Správná konfigurace serveru pro vynucení typu nahraných souborů.

```
if (!empty($_FILES['upload']) && $_FILES['upload']['error'] ==
UPLOAD_ERR_OK) {
    if (is_uploaded_file($_FILES['upload']['tmp_name']) === false)
{
        throw new \Exception('Chyba při nahrávání souboru');
    }

    $uploadName = $_FILES['upload']['name'];
    $ext = strtolower(substr($uploadName, strripos($uploadName,
'.') +1));
    $filename = round(microtime(true)) . mt_rand() . '.' . $ext;

    move_uploaded_file($_FILES['upload']['tmp_name'], __DIR__.
'../uploads/' . $filename);
}
```

Kód 70 - Příklad bezpečného nahrávání PHP souborů v PHP (PHP.earth, 2018)

Implementací pravidel ve výčtu výše vznikne kód 70. Nejprve je zkontrolováno, zda se soubor nahrál v pořádku celý. Poté se zkontroluje pomocí *is_uploaded_file()*, zda byl soubor nahrán pomocí HTTP požadavku. Nahraný soubor se následně přejmenuje na náhodný název se zachováním jeho přípony. Nakonec je přejmenovaný soubor nahraný na server do veřejně nedostupného adresáře, popisuje PHP.earth (2018).

5.11 Denial of service (DoS)

Pro porozumění DDoS útokům a jejich následkům je nutné seznámit se s DoS útoky. DoS útok je útok, který způsobí odepření služby web serveru, aplikace tak bude v nejhorším případě zcela nedostupná všem svým klientům, v mírnějším případě pak bude reagovat pomaleji nebo kvalita služby poskytované aplikací bude jinak snížena. Klienti v tomto smyslu nemusí být jen fyzičtí návštěvníci, mohou to být i roboti nebo jiné aplikace. Podstatný rozdíl mezi tímto a ostatními útoky popsanými v práci je v tom, že DoS útok nevyžaduje jakékoliv nabourání do aplikace nebo na její server. Cílem tohoto útoku nemusí být nutně jediný server, může to být i celá síť. Podstata útoku je ale jednoduchá. Může jít o přetížení sítě nebo serveru do takové míry, aby nebyly schopny zvládnout příchozí síťový provoz. Druhou možností je

odesílání takových paketů na server, aby se cílový systém zasekl nebo se zcela zhroutil. Dle těchto způsobů Schmied a Shimonski (2003) DoS útoky dělí na:

- Resource consumption útoky – útoky, které zaberou prostředky serveru nebo sítě
- Malformed packet útoky – útoky, které na server posílají neplatné pakety

Schmied a Shimonski dále správně uvádí, že DoS útok může být iniciován i z lokální sítě. DoS útoky nemohou vést ke zcizení citlivých dat, ale i proto jsou často využívány jako doprovodné během jiných útoků, aby na sebe strhly pozornost, zatímco se útočník snaží proniknout do systému. V případě, že útočník provede pouze DoS útok, i tak může způsobit dost škody. Jde zpravidla o případy, kdy je daná aplikace závislá na internetovém provozu, např. e-shopy.

DDos útok je pouze variací DoS útoku. Rozdíl spočívá v tom, že útočník použije více strojů k útoku na svůj cíl. Stroje nemusí nutně znamenat počítače, mohou to být různá chytrá zařízení připojená k internetu včetně např. televizorů nebo aut, uvádí Bell (2016) a Song, Fink a Jeschke (2017). Velká část moderních webů a aplikací je provozována ve velkých síťových clusterech nebo v cloudu, proto je pro jeden stroj často nemožné vyvolat na serveru tolik provozu, aby na něj měl skutečný dopad. Pokud se ale do DDoS útoku spojí stovky až tisíce strojů, situace se náhle mění. DDoS útoky mívají zpravidla dvě fáze. Během té první útočník cílí na různé stroje připojené k internetu a nainstaluje na ně specializovaný software, díky kterému se budou moci i tyto napadené stroje podílet na útoku. V druhé fázi všechny napadané stroje (tzv. zombies) společně zaútočí na hlavní cíl. Díky množství počítačů zapojených do útoku je iniciátor útoku téměř nedohledatelný (Easttom, 2016, Schmied, Shimonski, 2003, Decker, Zúquete, 2014).

Nejznámější nástroj pro provedení DoS útoku je aplikace Low Orbit Ion Cannon, uvádí dále Easttom (2016). Je zdarma ke stažení a každý bez technických znalostí může snadno zaútočit na libovolný server, stačí pouze zadat IP adresu nebo URL cíle.

5.12 Chyby konfigurace

Chování PHP do velké míry závisí na konfiguraci v souboru *php.ini*, která určuje, jaké funkce budou v dané instalaci PHP k dispozici. Další konfigurační direktivy se nastavují na Apache serveru. Řada z těchto nastavení přímo ovlivňuje bezpečnost aplikace, jak ukazuje kapitola 5.7 o file inclusion útocích (Afooshteh, Hoffmann, Stock a Plant, 2017).

Mezi hlavní hodnoty, které je třeba důkladně nastavit v konfiguračním souboru *php.ini*, dle PHP.earth (2018) bezesporu patří:

- display_errors = off, aby se nevypisovaly chybové hlášky, kterých by mohl útočník zneužít,
- log_errors = on, aby o všech chybových událostech v aplikaci administrátor věděl,
- expose_php = off, aby web server s odpovědí neodesílal hlavičku X-Powered-By s hodnotou PHP a jeho verzí (např. PHP/7.0.27),
- allow_url_fopen = off,
  allow_url_include = off, aby server neměl přístup ke vzdáleným souborům,
- open_basedir = "/var/www/test/uploads", aby PHP mělo přístup ke čtení a zápisu jen tam, kam je to žádoucí; omezení se týká funkcí (*fopen*, *file_get_contents()* i volání *include* a *require*),
- session.use_cookies = on, aby se session ukládala do cookie,
- session.use_only_cookies = on, aby session ID nemohlo být přes GET parametr PHPSESSID změněno a používáno případné session ID z HTTP požadavků nebo URL,
- session.use_trans_sid = off, aby i při nefunkčních cookies nebylo session ID předáváno do URL a předcházelo se session ID injection nebo uniknutí,
- session.use_strict_mode = on, aby server důvěřoval jen session ID vytvořené serverem, nikoliv session ID od uživatele,
- session.cookie_httponly = on, aby byly serverem přijmuty jen HttpOnly cookies, což zabrání následkům XSS, které by cookie zcizilo a zneužilo

- session.cookie_domain = aplikace.cz, aby aplikace přijímala cookies jen z dané domény, což je velmi žádoucí a není ve výchozím stavu nastaveno,
- session.cookie_secure = on, aby v aplikaci běžící pod HTTPS protokolem web server přijímal pouze cookies obdržené z požadavku poslaného přes HTTPS.

The PHP Group (2018) tento výčet dále rozšiřuje o session.cookie_lifetime = off, aby session ID bylo smazáno se zavřením prohlížeče a nemohlo tak být zneužito jiným uživatelem.

Podstatná část nastavení se liší ve vývoji, potamžmo na vývojovém a lokálním serveru, s nastavením na produkčním serveru. Stále ale platí pravidlo, že dojde-li v aplikaci na produkčním serveru k chybě, uživatel se nesmí dozvědět, o jakou chybu se jedná a už vůbec nesmí být při chybě zobrazen zdrojový kód, ve kterém se chyba naskytla. Těchto informací by totiž mohl útočník zneužít, vždy je těžší penetrovat neznámé prostředí než prostředí známé, uvádí Lockhart a Sturgeon (2017) a doporučují následující nastavení *php.ini* souboru pro vývojové prostředí:

- display_errors = on,
- display_startup_errors = on,
- error_reporting = -1,
- log_errors = on.

Tak bude zajištěno, že vývojář se vždy dozví o každé chybě, která nastane. Naproti tomu v produkčním prostředí je potřeba nastavení změnit tak, aby se informace o chybách stále ukládaly do logů, ale nezobrazovaly se uživatelům a to pomocí následujících hodnot nastavení (Lockhart, Sturgeon, 2017):

- display_errors = off,
- display_startup_errors = off,
- error_reporting = -1,
- log_errors = on.

Posledním bodem konfiguračních chyb webových aplikací je bezesporu nepoužití HTTPS protokolu pro přístup k aplikaci. Každá aplikace by dnes měla využívat

zabezpečený protokol, neexistuje dobrý důvod tomu bránit. Hlavním důvodem dříve byla cena certifikátů, které jsou pro HTTPS potřeba. Dnes je ale dostupný certifikát Let's Encrypt zcela zdarma a tak i poslední důvod padl (PHP.earth, 2018).

5.13 Ostatní

Wang a Ledley (2012) považují cloud computing za velmi zajímavá odnož webu z hlediska bezpečnosti, neboť přitahuje velké množství útočníků. To je přirozené, v cloudu je obrovské množství dat a velká část jich je velmi cenná. Data v cloudu nejsou na serverech v jedné doméně, jsou na platformě, takže je složité až nemožné zjistit, kde se data fyzicky nachází. To ani jiné nevýhody cloudu z hlediska bezpečnosti paradoxně nejsou takový problém, protože velcí poskytovatelé cloudových služeb jako Amazon nebo Google mají rozpočet na nejlepší profesionály v oboru bezpečnosti a tak data v cloudu mohou být uložena bezpečněji než např. na serverech vládních agentur.

6 Testování zabezpečení webové aplikace

Součástí této práce je řádně zabezpečená webová aplikace napsaná v PHP frameworku Laravel verze 5.5, dostupná z https://gitlab.com/TeeJay.net/diploma-thesis.git (online). Podstata této aplikace spočívá v názorném uplatnění navržených technik zabezpečení. Tato kapitola představuje strategii testování zabezpečení vytvořené aplikace, průběh jejího testování a samozřejmě výsledky.

Na hackování a penetrační testování se velmi často používají skripty psané v jazyce Python a operační systém Kali linux. Jak se domnívá Sinha (2017), testování aplikace v této práci se omezuje pouze na provedení základních testů prostřednictvím uživatelského prostředí aplikace. Tyto testy proběhnou podle vybraných scénářů potenciálních útoků pro potvrzení teorie správnosti zabezpečení webové aplikace.

6.1 Strategie testování aplikace

Pro testování v rámci práce je aplikace nainstalována na PHP hosting společnosti Active24, balíček Linux Komplet. Aplikace na serveru běží pod protokolem HTTPS prostřednictvím certifikátu *Let's Encrypt*, verze PHP je v době psaní 7.1.16. Aplikace je testována v několika krocích. V každém kroku se testuje jiná zranitelnost, dochází k simulaci konkrétních útoku. Testovány jsou jen ty útoky, které se na aplikaci vztahují, proto např. *file inclusion* útoky testovány nebudou – neexistuje místo, kde by aplikace mohla obsahovat tuto zranitelnost. Všechny formuláře v aplikaci napsané v Laravel frameworku obsahují CSRF tokeny, bez kterých žádný formulář nelze odeslat, ani *CSRF* útoky tak nemá smysl testovat. Strategie je inspirována **reversal** metodou metodiky OSSTM z kapitoly 4.2.1, tedy útočník ví vše o svém cíli. Po vymezení cílů strategie testování zahrnuje testování těchto položek:

1. autentizace,
2. databázové útoky – SQL injection,
3. XSS,
4. public files,
5. řízení přístupu,
6. nahrávání souborů.

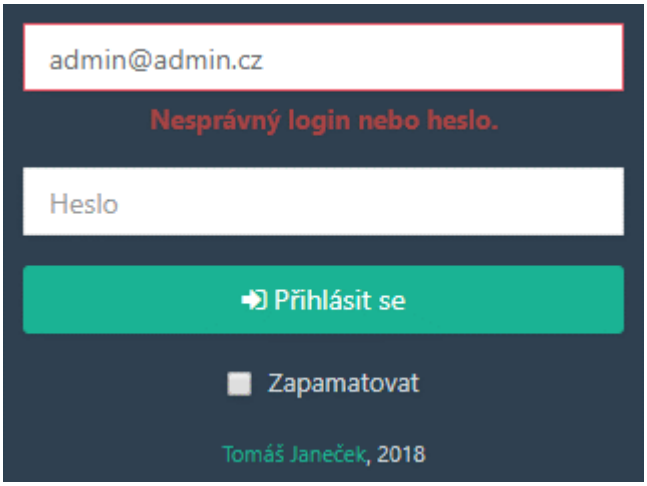
První čtyři body výčtu se týkají veřejně dostupné části aplikaci, zbylé dva neveřejné části aplikaci dostupné jen administrátorům po přihlášení.

6.2 Testování aplikace

Aplikace je otestována na šest zranitelností. Tato kapitola popisuje způsob ověření správností zabezpečení proti každému útoku bez znalosti zdrojového kódu. Test každé zranitelnosti popisuje veřejně dostupná místa, kde hrozí zneužití zranitelnosti, provedení samotného útoku a reakci aplikace na simulovaný útok.

6.2.1 Testování autentizace

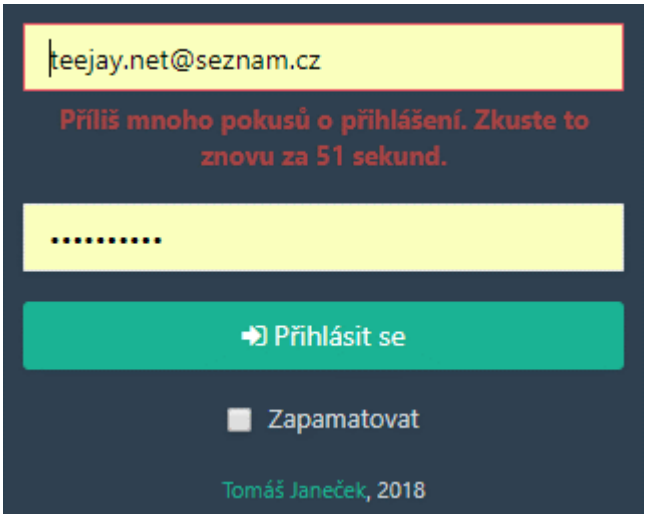
Aplikace nenabízí možnost přihlášení pro běžné uživatele. Jediné místo, kde se lze příhlásit, je na stránce */admin*, kde se přihlašují administrátoři aplikace. V případě testování autentizace je v této aplikaci vhodné vyzkoušet pouze ochranu proti slovníkovým nebo brute force útokům, tedy zkoušet různé kombinace hesel a jmen. Funkce „zapomněl jsem heslo“ zde není a běžné kombinace výchozích uživatelských jmen a hesel pokryjí testy slovníkových útoků.



Obrázek 1 - Přihlašovací formulář aplikace upozorňující na nesprávné heslo nebo login

V rámci testu mohou být do aplikace zadávány různé kombinace emailů a hesel, jak napovídá formulář sám. Typických kombinací různých emailů typu „admin@admin.cz“ a hesel typu „admin“, „123456“ může být vyzkoušeno velké množství. Zda je nesprávný email nebo heslo, aplikace neprozradí, ukazuje obrázek 1. Navíc po pěti pokusech v jedné minutě je útočník do další minuty

zastaven, jak ukazuje obrázek 2. Vyzkoušet dostatečně velké množství kombinací, aby se pravděpodobnost úspěchu dostala do přirozených čísel, by tak trvalo velmi dlouho. Proto aplikace v testu obstála.



Obrázek 2 - Přihlašovací formulář aplikace upozorňující, že do další minuty se nelze pokusit znovu přihlásit

6.2.2 Testování databázových útoků

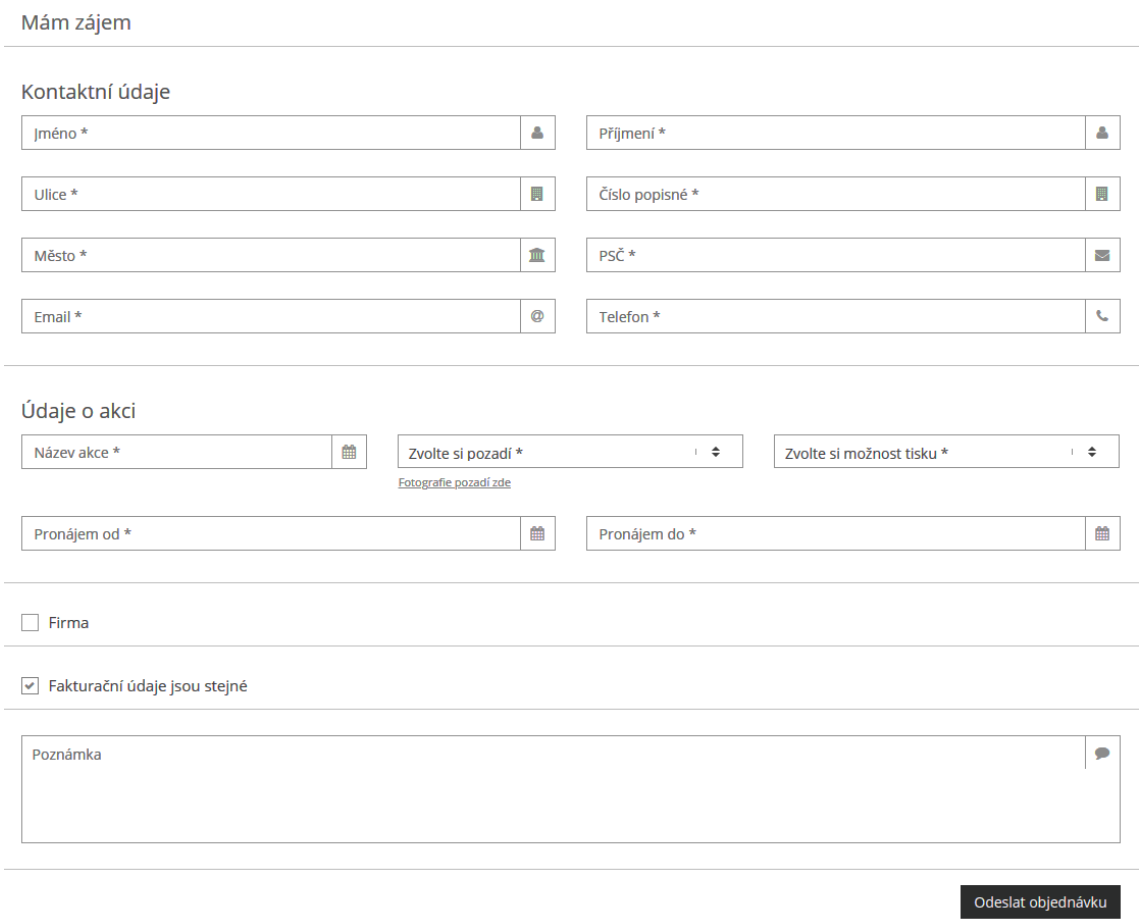
Teoretická možnost SQL injection útoku je na několika místech v aplikaci. Jednak opět v přihlašovacím formuláři administrátora, kde by pomocí SQL injection mohlo dojít k obejití přihlašovacího formuláře i bez znalosti hesla nebo při využití parametrů stránkování v seznamu článků, položka menu „blog“. Stránkování se zobrazuje jen v případě, že je v aplikaci více článků, než kolik jich je nastavených k zobrazení na jednu stránku, tzn. v tomto případě více než 10 článků. To ale útočníkovi nebrání tento parametr do URL ručně přidat.

Aplikace využívá ORM framework s názvem Eloquent patřící k frameworku Laravel, takže veškeré dotazy jsou na databázi odesílány přes PDO rozšíření PHP a prepared statements. Žádný dotaz v aplikaci není ručně psaný, každý je vytvořený pouze pomocí funkcí ORM bez vlastního SQL kódu. Na pozadí v kódu ORM frameworku při konstrukci finálních dotazů samozřejmě platí, že žádná proměnná není vložena do dotazu prostřednictvím spojováním řetězců, každá proměnná je vložena parametricky pomocí funkcí k tomu určených. I tak ale stojí za to ověřit, zda se někde nestala chyba. V přihlašovacím formuláři se nabízí do pole „Heslo“ zkusit vepsat

hodnoty, které by se potenciálně mohly navázat na stávající dotaz a přihlášení obejít. Jedná se o následující hodnoty – „' or '1'='1“, „' or 1='1“, „1' or 1=1 -- -“, „' or '1'='1“, nebo např. „' or ' 1=1“. Při vyzkoušení všech těchto hodnot aplikace dle očekávání reaguje stejně jako na obrázku 1 – hlásí nesprávný login nebo heslo. Aplikace tak v testu obstála.

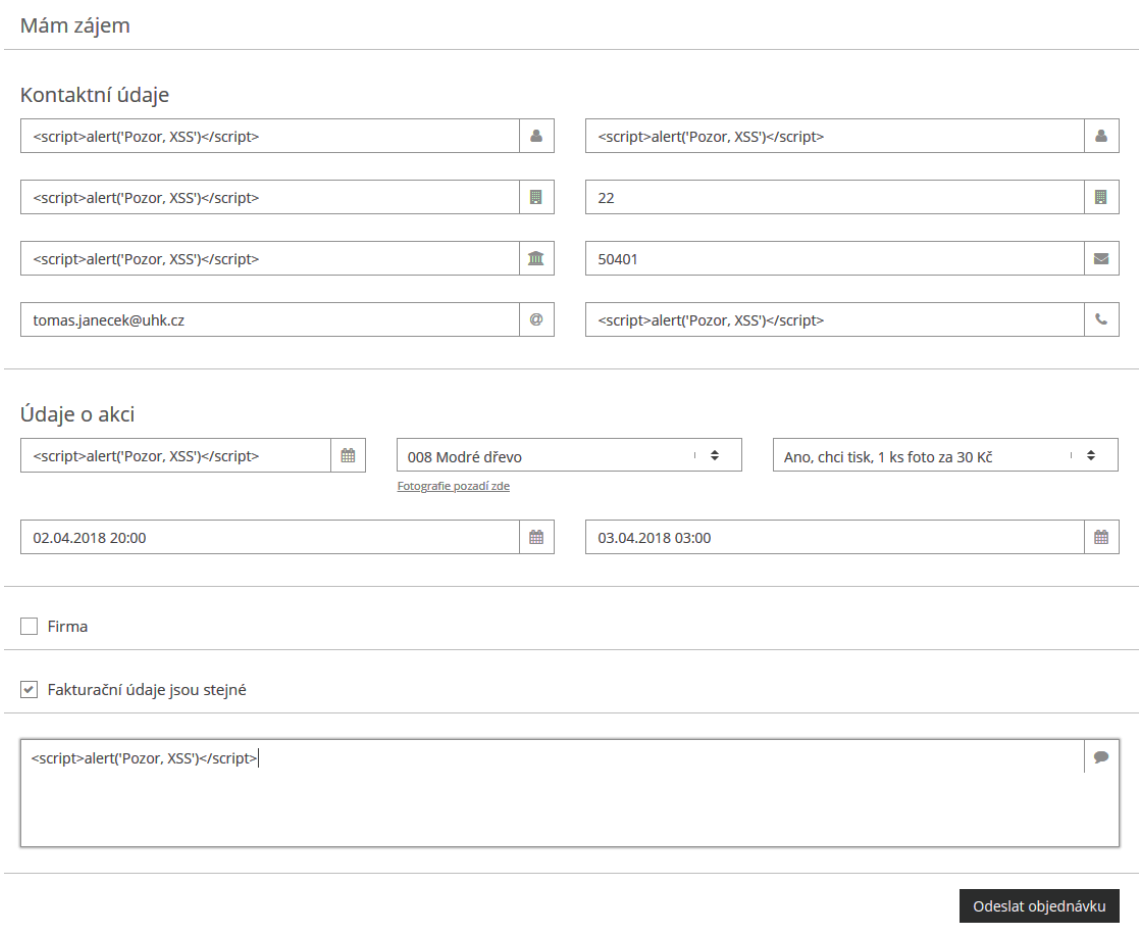
6.2.3 Testování XSS

Veřejně dostupná část aplikace obsahuje pouze jeden formulář, který by mohl být využit pro persistentní XSS. Pro nepersistentní XSS by bylo třeba GET parametru, jehož hodnota by se na webu zobrazovala v nefiltrované nebo špatně filtrované podobě. Aplikace ale obsah GET parametrů do výstupu nikde neodesílá, tato možnost tedy není přípustná. Jedinou možností by mohlo být stránkování, to má ale předdefinovanou podobu a whitelist filtr jiné hodnoty nepustí. Vhodným kandidátem na test je tedy objednávkový formulář na obrázku 3.



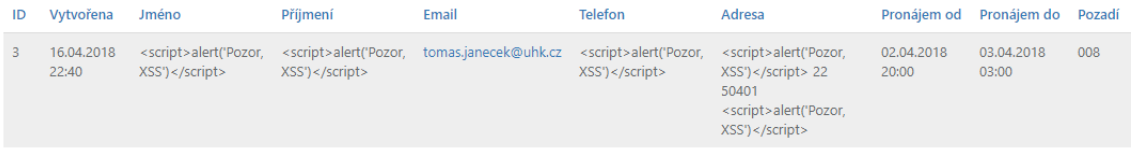
Obrázek 3 - Objednávkový formulář v aplikaci

Perfektní volbou pro zjištění, zda aplikace XSS podléhá, je vepsání krátkého JavaSkriptu do některých polí formuláře. Cílem tohoto testu je pouze zjistit, zda aplikace XSS podléhá, nikoliv nějak uškodit, postačí proto jednoduchý skript *<script>alert('Pozor, XSS')</script>*. Aplikace obsahuje řadu validačních pravidel, do všech polí, která to vzhledem k validačním pravidlům dovolí, je však dobré tento skript vložit. Pokud je aplikace dobře zabezpečená proti XSS, jsou řádně vyčištěny od škodlivých dat všechny vstupy, nikoliv jen některé. Formulář v aplikaci lze vyplnit tak, jak zobrazuje obrázek 4.



Obrázek 4 – Objednávkový formulář aplikace vyplněný skripty pro pokus o XSS útok

Skript je ve všech polích, která mají validaci volnější a limitují pouze rozsah, nikoliv použité znaky (číslo popisné musí být přirozené číslo, PSČ musí obsahovat 5 číslic apod.). Formulář lze takto odeslat. Co pak vidí administrátor, je následek správného escapování všech proměnných a zobrazuje to obrázek 5.



ID	Vytvořena	Jméno	Příjmení	Email	Telefon	Adresa	Pronájem od	Pronájem do	Pozadí
3	16.04.2018 22:40	<script>alert('Pozor, XSS')</script>	<script>alert('Pozor, XSS')</script>	tomas.janecek@uhk.cz	<script>alert('Pozor, XSS')</script>	<script>alert('Pozor, XSS')</script> 22 50401 <script>alert('Pozor, XSS')</script>	02.04.2018 20:00	03.04.2018 03:00	008

Obrázek 5 - Zobrazení dat z odeslaného formuláře s pokusem o XSS v administraci

Vyskakovací okno s textem „Pozor, XSS“ na administrátora nevyskočilo, všechny znaky skriptů ve všech případech byly správně escapovány a skripty tak nejsou spustitelné. Aplikace je zde odolná vůči XSS a tak v testu obstála.

6.2.4 Testování zranitelnosti public files

Aplikace napsané v Laravel frameworku obsahují v kořenovém adresáři některé soubory, u kterých je důležité, aby nebyly veřejné. Jedná se zejména o soubor *.env*, který obsahuje údaje pro připojení aplikace k databázi včetně IP adresy databázového serveru, jména databáze, uživatelského jména i hesla databáze. Také zde mohou být údaje k SMTP připojení do emailových schránek, klíč pro šifrování obsahu v aplikaci a řada dalších klíčových nastavení aplikace. Test spočívá v zadání URL do prohlížeče obsahující cestu */.env*, popř. */../.env* a ani v jednom případě nesmí dojít k tomu, aby aplikace soubor zobrazila nebo snad jen naznačila, že tento soubor existuje. Aplikace na toto zadání reaguje odpovědí HTTP 404, tedy nenalezeno. Test je tak úspěšný.

6.2.5 Testování řízení přístupu

Testování řízení přístupu spočívá v otestování, že neoprávněný uživatel nemá přístup do částí aplikace, které neodpovídají jeho přístupovým právům. To znamená, že nepřihlášený uživatel nesmí mít přístup do žádné části aplikace, která vyžaduje přihlášení, a přihlášený uživatel může mít přístup jen do míst odpovídajících jeho přístupovým právům.

URL hlavního panelu administrace, kam má přístup každý přihlášený uživatel, je */admin/dashboard*. Pokud tuto URL uživatel zadá do prohlížeče, je přesměrován na stránku */admin/* s přihlašovacím formulářem. Jedná se o žádoucí chování aplikace.

Druhý neméně důležitý test lze provést s libovolnou stránkou. Spočívá v tom, že uživatel se pokusí prostřednictvím vhodné změny URL dostat na stránku, na kterou jinak nemá přístup. Pokud uživatel, který nemá právo přistupovat k administraci článků, které jsou pod URL /admin/articles, zadá tuto URL přímo do prohlížeče, dostane se mu odpovědi HTTP 403 – přístup zamítnut. To platí pro všechny položky aplikace, které podléhají řízení přístupu. A je skutečně podstatné ověřit, podobně jako u většiny potenciálních zranitelností, že problém je vyřešený na všech místech, kde by mohl nastat, nikoliv jen na jednom z několika. Může se stát, že vývojář autorizaci na jednom místě zapomene, zatímco jinde v kódu bude autorizace v pořádku. Aplikace nicméně oběma testy řádně prošla.

6.2.6 Testování nahrávání souborů

Aplikace obsahuje několik shodných formulářů, které umožňují administrátorům nahrávat obrázky na server. Kapitola 5.10 popisuje hlavní rizika nahrávání souborů. Hlavní pravidlo spočívá v tom, že server-side kód zpracování formuláře musí verifikovat, že typ nahraného souboru je v souladu s validačními pravidly formuláře. V žádném případě nesmí být dopuštěno, aby na server mohl být nahrán PHP nebo HTML soubor, ale ani žádný jiný soubor, který neodpovídá účelu formuláře. Formulář pro nahrávání obrázků musí pečlivě kontrolovat, že jsou skutečně nahrávány obrázky a ne pouze soubory, které mají příponu jako obrázkové soubory, ale ve skutečnosti se o obrázkové soubory nejedná.

Součástí testu je hned několik podtestů. Jedná se o pokusy o nahrání:

- JPG souboru s příponou *.jpg*,
- PHP souboru s příponou *.php*,
- JPG souboru s příponou *.txt* namísto *.jpg*,
- PHP souboru s příponou přepsanou na *.jpg*.

Validace typu *image* vestavěná v Laravel frameworku dokáže i překvapit. Samozřejmě neumožní nahrát maskovaný PHP nebo jiný neobrázkový soubor. Navíc ale správně umožní nahrát obrázkový soubor, který se vydává za neobrázkový. Test je úspěšný.

7 Závěry a doporučení

Software je stále důmyslnější, komplexnější a provázanější s dalšími systémy a s tím exponenciálně roste i náročnost cesty k dosažení stavu, kdy je takový software považovaný za bezpečný. Rapidní tempo vývoje moderních aplikací znamená, že odhalení a vyřešení jejich bezpečnostních problémů musí být provedeno rychle a spolehlivě. V dnešní době si už není možné dovolit tolerovat relativně prosté bezpečnostní problémy, které popisuje tato práce. Nedostatečně zabezpečené aplikace mohou být v důsledku nespolehlivé a ztratit důvěru svých uživatelů.

Práce popisuje rozličné typy útoků na webové aplikace v PHP, které útočníci používají nejčastěji. Mezi nejfrekventovanější a nejnebezpečnější typy útoků už dlouhou dobu patří zejména SQL injection a cross-site scripting. Tyto útoky se uplatňují i na aplikace mimo web. Bezpečnost webových aplikací totiž spočívá v uplatnění obecných principů bezpečnosti aplikací na internet a aplikace na webu. Dále se práce zaměřuje na problematiku testování zabezpečení webových aplikací, tzv. penetrační testování. Následně jsou detailně rozebrány principy výše popsaných i dalších způsobů útoků a navržena řešení pro aplikace psané v PHP, mj. s inspirací z frameworků Nette verze 2.4 a Laravel verze 5.5. Na základě navržených způsobů obrany webových aplikací je jako součást práce vytvořena PHP aplikace ve frameworku Laravel (dostupná online z https://gitlab.com/TeeJay.net/diploma-thesis.git), která tyto bezpečnostní principy uplatňuje v praxi. Závěr tvoří navržená strategie testování vytvořené aplikace z hlediska bezpečnosti a aplikace je s její pomocí úspěšně otestována.

Ukazuje se, že **u většiny bezpečnostních slabin** webových aplikací, se kterými je možné se na webu setkat nejčastěji, **je řešení relativně jednoduché**. Za většinou chyb v zabezpečení stojí sami vývojáři aplikací. Při značném zjednodušení platí, že stejně jako u sítí, i webové aplikace jsou jen tak bezpečné, jak je jejich tvůrci udělají. Tato práce by proto mohla být dále rozšířena hned několika směry. Jednak směrem automatizovaného testování zabezpečení, směrem zaměření projektových manažerů na bezpečnost a směrem k edukaci vývojářů.

8 Seznam použité literatury

ACOBS, Stuart. Engineering Information Security: The Application of Systems Engineering Concepts to Achieve Information Assurance. Second Edition. New Jersey: IEEE Press, 2016. ISBN 978-1-119-10160-4.

Acunetix: Web Site Security: What can you do about it? [online]. London: Acunetix, 2018 [cit. 2018-01-28]. Dostupné z: http://www.acunetix.com/websitesecurity/.

AFOOSHTEH, Abbas Naderi, Achim HOFFMANN, Andrew STOCK a Luke PLANT. PHP Security Cheat Sheet. OWASP [online]. 2016 [cit. 2018-01-28]. Dostupné z: https://www.owasp.org/index.php/PHP_Security_Cheat_Sheet.

ALEY, Rob. PHP Beyond the Web. New York, NY: Springer Science Business Media, 2016. ISBN 978-1-4842-2480-9.

ALI, Shakeel a ALLEN, Lee. Kali Linux: Assuring Security by Penetration Testing. Packt Publishing, 2014. ISBN 978-184-9519-489.

ALJAWARNEH, Shadi. Online Banking Security Measures and Data Protection. Hershey, PA: [Information Science Reference], An Imprint of IGI Global, 2017. ISBN 978-152-2508-656.

BALOCH, Rafay. Ethical Hacking and Penetration Testing Guide. Boca Raton: CRC Press, Taylor, 2015. ISBN 14-822-3161-1.

BEGGS, Robert. Mastering Kali Linux for Advanced Penetration Testing: A Practical Guide to Testing Your Network's Security with Kali Linux, the Preferred Choice of Penetration Testers and Hackers. Packt Publishing, 2014. ISBN 978-1-78216-312-1.

BELL, Charles. MySQL for the Internet of Things. New York: Apress, 2016. Expert's voice in big data. ISBN 978-148-4212-943.

BOSWORTH, Seymour., Michel E. KABAY a Eric WHYNE. Computer Security Handbook. Sixth edition. Hoboken, New Jersey, 2014. ISBN 978-1-118-13411-5.

CANNINGS, Rich, Himanshu DWIVEDI a Zane LACKEY. Hacking Exposed Web 2.0 Web 2.0 Security Secrets and Solutions. New York: McGraw-Hill, 2008. ISBN 00-715-9548-1.

DECKER, Bart De a André ZÚQUETE. Communications and multimedia security: 15th IFIP TC6/TC11 International Conference, CMS 2014, Aveiro, Portugal, September 25-26, 2014. Proceedings. New York: Springer, 2014. ISBN 978-3-662-44884-7.

EASTTOM, William. Computer Security Fundamentals. 3rd edition. Indianapolis, IN: Pearson Education, 2016. ISBN 978-0-7897-5746-3.

ENGEBRETSON, Pat. The Basics of Hacking and Penetration Testing: ethical hacking and penetration testing made easy. Second Edition. Boston: Syngress, an imprint of Elsevier, 2013. ISBN 978-0-12-411644-3.

GOLLMANN, Dieter. Computer Security. 3rd ed. Chichester, West Sussex: Wiley, 2011. ISBN 978-0-470-74115-3.

HARRIS, Shon. Gray Hat Hacking the Ethical Hacker's Handbook. 2nd ed. New York: McGraw-Hill, 2008. ISBN 00-715-9553-8.

KANAT-ALEXANDER, Max. Code Simplicity. Sebastopol, CA: O'Reilly, c2012. ISBN 978-1-449-31389-0.

KIM, Peter. The Hacker Playbook: Practical Guide to Penetration Testing. North Charleston, South Carolina: Secure Planet, LLC, 2014. ISBN 978-149-4932-633.

LOCKHART, Josh a Phil STURGEON. PHP The Right Way: Your guide to PHP best practices, coding standards and authoritative tutorials. PHP The Right Way [online]. 2017, 2017-11-20 [cit. 2018-01-28]. Dostupné z: http://www.phptherightway.com/.

LONG, Johnny. a Kevin D. MITNICK. No Tech Hacking: A Guide to Social Engineering, Dumpster Diving, and Shoulder Surfing. Oxford: Elsevier Science [distributor], c2008. ISBN 978-1-59749-215-7.

LONG, Johnny. Google Hacking: For Penetration Testers. Rockland: Syngress Publishing, 2005. ISBN 19-318-3636-1.

LUDWIG, Mark A. The Giant Black Book of Computer Viruses. Show Low, Ariz.: American Eagle, 1995. ISBN 09-294-0810-1.

MESSIER, Ric. Penetration Testing Basics: A Quick-start Guide To Breaking Into Systems. New York, NY: Apress, 2016. ISBN 978-1-4842-1856-3.

MOGOLLON, Manuel. Cryptography and Security Services: Mechanisms and Applications. Hershey, PA: CyberTech Pub., c2007. ISBN 978-1-59904-837-6.

MUNIZ, Joseph a LAKHANI, Aamir. Web Penetration Testing with Kali Linux: A Practical Guide to Implementing Penetration Testing Strategies on Websites, Web Applications, and Standard Web Protocols with Kali Linux. Birmingham, England: Packt Publishing, 2013. ISBN 978-1-78216-317-6.

OWASP. OWASP Top 10 - 2017: The Ten Most Critical Web Application Security Risks. OWASP [online]. 2017 [cit. 2018-01-28]. Dostupné z: https://www.owasp.org/images/7/72/OWASP_Top_10-2017_%28en%29.pdf.pdf.

PHP.EARTH. How to secure PHP web applications and prevent attacks? PHP.earth [online]. 2018 [cit. 2018-01-28]. Dostupné z: https://php.earth/docs/security/intro.

SCAMBRAY, Joel, Vincent LIU a Caleb SIMA. Hacking Exposed Web Applications: Web Application Security Secrets and Solutions. 3rd ed. New York: McGraw-Hill, 2011. ISBN 978-007-1740-425.

SHARPE, Isaac. Hacking: Guide to Basic Security, Penetration Testing and Everything Else Hacking. CreateSpace Independent Publishing Platform, 2015. ISBN 978-1-512-3007-72.

SHEMA, Mike. Web Application Ssecurity for Dummies. Chichester: A John Wiley, 2011. ISBN 978-111-9994-879.

SCHMIED, Will a Robert J. SHIMONSKI. : Study Guide And DVD Training System. [Online-Ausg.]. Rockland, Mass: Syngress, 2003. ISBN 19-318-3684-1.

SIMPSON, Michael T., Kent. BACKMAN a James E. CORLEY. Hands-on Ethical Hacking and Network Defense. 2nd ed., international ed. Boston, MA: Course Technology, Cengage Learning, c2011. ISBN 978-1-4354-8609-6.

SINHA, Sanjib. Beginning Ethical Hacking with Python. West Bengal, India: Apress, 2017. ISBN 978-148-4225-400.

SONG, Houbing, Glenn A. FINK a Sabina JESCHKE. Security and Privacy in Cyber-physical Systems: Foundations, Principles, and Applications. Hoboken, NJ, 2017. ISBN 978-111-9226-048.

STOCK, Andrew, Ismael Rocha GONÇALVES a Jorge CORREA. OWASP Top Ten Cheat Sheet. OWASP [online]. 2017 [cit. 2018-01-28]. Dostupné z: https://www.owasp.org/index.php/OWASP_Top_Ten_Cheat_Sheet.

THE PHP GROUP. Safe Password Hashing. PHP.net[online]. 2018, 2001-2018 [cit. 2018-01-28]. Dostupné z: http://php.net/manual/en/faq.passwords.php.

THE PHP GROUP. Security. PHP.net [online]. 2018, 2001-2018 [cit. 2018-01-28]. Dostupné z: http://php.net/manual/en/security.php.

WANG, Shuangbao Paul a Robert Steven LEDLEY. Computer architecture and security: Fundamentals of Designing Secure Computer Systems. Hoboken, NJ, 2012. ISBN 978-1-118-16881-3.

9 Seznam obrázků

Obrázek 1 - Přihlašovací formulář aplikace upozorňující na nesprávné heslo nebo login 88

Obrázek 2 - Přihlašovací formulář aplikace upozorňující, že do další minuty se nelze pokusit znovu přihlásit 89

Obrázek 3 - Objednávkový formulář v aplikaci 90

Obrázek 4 – Objednávkový formulář aplikace vyplněný skripty pro pokus o XSS útok 91

Obrázek 5 - Zobrazení dat z odeslaného formuláře s pokusem o XSS v administraci 92

10 Seznam zdrojových kódů

Kód 1 - Kód podléhající neočekávanému vstupu 22

Kód 2 - Adresa s očekávaným vstupem z parametru 22

Kód 3 - Adresa s neočekávaným vstupem z parametru 23

Kód 4 - Důvod umožnění úspěšného útoku 23

Kód 5 - Důvod neumožnění úspěšného útoku 23

Kód 6 - Skript nepodléhající neočekávanému vstupu 23

Kód 7 - Příklad dotazu do vyhledávače pro nalezení webů s přihlašováním uživatelů 26

Kód 8 - Příklad dotazu do vyhledávače pro nalezení webů napsaných v PHP 26

Kód 9 - Ukázka kódu výchozího autentifikátoru v Nette 2.4 využívající striktní porovnání řetězců 29

Kód 10 - Ukázka kódu výchozího autentifikátoru v Laravelu 5.5 využívající PHP Password API a metodu password_verify() 30

Kód 11 - Tělo metody login() autentifikátoru frameworku Laravel 5.5 31

Kód 12 - Ukázka kódu, jak řádně a bezpečně smazat cookie 33

Kód 13 - Příklad kódu s PDO, který může umožnit SQL injection 35

Kód 14 - Vhodná hodnota GET parametru URL pro SQL injection 36

Kód 15 - URL stránky zranitelné SQL injection tak, jak ji uživatel standardně vidí 36

Kód 16 - URL stránky zranitelné SQL injection po vhodném doplnění útočníkem .36

Kód 17 - Výsledný dotaz do databáze obsahující i dotaz útočníka 36

Kód 18 - Řešení SQL injection v případě využití PDO rozšíření PHP 37

Kód 19 - Příklad kódu s MySQLi, který může umožnit SQL injection 38

Kód 20 - Řešení SQL injection v případě využití MySQLi rozšíření PHP 39

Kód 21 - Ukázka připojení k databázi ve vestavěném ORM frameworku Nette 40

Kód 22 - Ukázka kódu předávající SQL dotaz databázi v ORM frameworku Nette .. 41

Kód 23 - Ukázka připojení k databázi ve vestavěném ORM frameworku Laravelu 42

Kód 24 - Ukázka kódu předávající SQL dotaz databázi v ORM frameworku Laravelu 43

Kód 25 - Příklad hodnoty GET parametru v případě UNION based SQL injection ... 46

Kód 26 - Ukázka PHP skriptu, který podléhá nepersistentnímu XSS útoku 49

Kód 27 - Ukázka běžné URL adresy a GET parametru skriptu náchylného k XSS 50

Kód 28 - Ukázka URL adresy s pokusem o jednoduchý XSS útok 50

Kód 29 - Ukázka PHP kódu pro zobrazení HTML stránky, který nepodléhá XSS útokům 50

Kód 30 - Ukázka PHP funkcí, které zabrání XSS útokům u webových aplikací, kde není možné použít šablonovací systémy 51

Kód 31 - Ukázka výpisu proměnné v Latte šablonovacím systému frameworku Nette 54

Kód 32 - Ukázka výpisu proměnné bez escapování v Latte šablonovacím systému frameworku Nette 54

Kód 33 - Ukázka escapování v Latte, proměnné vypsané mezi dvě HTML značky .. 55

Kód 34 - Ukázka escapování v Latte, proměnné vypsané do hodnot atributů HTML elementů 55

Kód 35 - Ukázka escapování v Latte proměnných vypsaných jako název atributů HTML elementů 56

Kód 36 - Ukázka escapování v Latte, proměnné vypsané v HTML komentářích 56

Kód 37 - Ukázka escapování Latte, proměnné vypsané v XML šablonách 57

Kód 38 – Ukázka escapování Latte, proměnné vypsané jako atribut v XML šablonách 57

Kód 39 - Ukázka escapování Latte, proměnné vypsané v CSS šablonách 57

Kód 40 - Ukázka escapování Latte, proměnné vypsané jako proměnné v JavaScriptu 58

Kód 41 - UKázka escapování Latte, proměnné vypsané jako text do script nebo style elementů 58

Kód 42 - Ukázka escapování Latte, proměnné vypsané jako text v šabloně iCal 59

Kód 43 - Ukázka výpisu proměnné v Blade šablonovacím systému frameworku Laravel 60

Kód 44 - Ukázka výpisu proměnné bez escapování v Blade šablonovacím systému frameworku Laravel 60

Kód 45 - Funkce Laravel frameworku kompilující značky {{ }} pro výpis proměnných do PHP 61

Kód 46 - Funkce Laravel frameworku kompilující značky {!! !!} pro neescapovaný výpis proměnných do PHP 62

Kód 47 - Funkce Laravel frameworku volaná na všechny escapované výstupy dat pro prevenci XSS 62

Kód 48 - Ukázka GET požadavku v imaginární aplikaci pro převod peněz mezi účty 65

Kód 49 - Ukázka img HTML tagu, který může automaticky provést CSRF útok metodou GET 65

Kód 50 – Ukázka JavaScriptu, který může automaticky provést CSRF útok metodou POST 66

Kód 51 - Doporučená implementace CSRF ochrany prostřednictvím tokenů v PHP7 70

Kód 52 - Příklad URL adresy ve fiktivní aplikaci pro stránku se články 72

Kód 53 - Příklad URL adresy ve fiktivní aplikaci pro stránku s nastavením 72

Kód 54 - Ukázka kódu aplikace náchylné na local file inclusion útoky 73

Kód 55 - Ukázka kódu aplikace náchylné na remote file inclusion útoky 74

Kód 56 - Ukázka hodnoty, kterou je vhodné použít pro file inclusion útok 74

Kód 57 - Ukázka hodnoty, kterou je vhodné použít pro file inclusion útok a directory traversal útok zároveň 74

Kód 58 - Příklad PHP kódu aplikace podléhající command injection útokům 76

Kód 59 - Příklad PHP kódu aplikace podléhající code injection útokům 76

Kód 60 - Ukázka kódu aplikace podléhající directory traversal útoku 76

Kód 61 - Ukázka URL, která načte stránku s kontakty 76

Kód 62 - Ukázka URL, která načte soubor mimo root web serveru a může tak způsobit directory traversal útok 77

Kód 63 - Ukázka PHP implementace zabezpečení proti directory traversal a file inclusion útokům 77

Kód 64 - Struktura adresářů aplikací napsaných ve frameworku Laravel 78

Kód 65 - Příklad obsahu souboru .htaccess aplikace v Laravel v kořenovém adresáři aplikace, který vede k public files zranitelnosti 79

Kód 66 - Příklad obsahu souboru .htaccess aplikace v Laravel v kořenovém adresáři aplikace, který řeší zranitelnost public files 79

Kód 67 - Příklad URL k pokusu o získání obsahu konfiguračního .env souboru ve frameworku Laravel 80

Kód 68 - HTML formulář pro nahrávání souborů (PHP.earth, 2018) 80

Kód 69 - Příklad jednoduchého zpracování nahrání souborů v PHP, bez validace (PHP.earth, 2018) 81

Kód 70 - Příklad bezpečného nahrávání PHP souborů v PHP (PHP.earth, 2018) 82

Univerzita Hradec Králové
Fakulta informatiky a managementu
Akademický rok: 2017/2018

Studijní program: Aplikovaná informatika
Forma: Prezenční
Obor/komb.: Aplikovaná informatika (ai2-p)

Podklad pro zadání DIPLOMOVÉ práce studenta

PŘEDKLÁDÁ:	ADRESA	OSOBNÍ ČÍSLO
Bc. Janeček Tomáš	Řehoty 22, Králíky - Řehoty	I1500688

TÉMA ČESKY:

Bezpečnost webových aplikací v PHP

TÉMA ANGLICKY:

PHP Web Application Security

VEDOUCÍ PRÁCE:

Ing. Zuzana Němcová, Ph.D. - KIT

ZÁSADY PRO VYPRACOVÁNÍ:

Cíl práce:
Poskytnout přehled nejčastějších technik prolomení zabezpečení webových aplikací, pokrýt typické způsoby penetračního testování a navrhnout optimální strategii zabezpečení webových aplikací v PHP."

Osnova:
1) Úvod
2) Cíl a struktura práce
3) Bezpečnostní rizika aplikací
4) Testování zabezpečení aplikací
5) Obrana proti hrozbám v PHP
6) Závěry a doporučení

SEZNAM DOPORUČENÉ LITERATURY:

1. BALOCH, Rafay. Ethical Hacking and Penetration Testing Guide. CRC Press, Taylor, 2015. ISBN 14-822-3161-1.
2. BEGGS, Robert. Mastering Kali Linux for Advanced Penetration Testing: A Practical Guide to Testing Your Network's Security with Kali Linux, the Preferred Choice of Penetration Testers and Hackers. 2014. ISBN 978-1-78216-312-1.
3. BRYAN SULLIVAN, Vincent Liu a Michael Howard. TECHNICAL EDITOR. Web application security a beginner's guide. New York: McGraw-Hill, 2012. ISBN 978-0-07-177612-7.
4. OWASP. OWASP Top 10 - 2017. Dostupné z: https://www.owasp.org/images/7/72/OWASP_Top_10-2017_%28en%29.pdf.pdf

Podpis studenta: ..... **Datum:** 17.4.2018

Podpis vedoucího práce: ..... **Datum:** 17.4.2018

(c) IS/STAG , Portál - Podklad kvalifikační práce , I1500688 , 17.04.2018 12:12