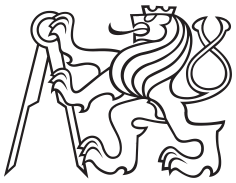
Bakalářská práce



České
vysoké
učení technické
v Praze

F3

Fakulta elektrotechnická
Katedra počítačové grafiky a interakce

Časté zranitelnosti webových aplikací

Zabezpečení webových aplikací v jazyce PHP

Tomáš Klouček

Vedoucí: RnDr. Ondřej Žára
Obor: Softwarové inženýrství
Studijní program: Enterprise
Květen 2024

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Klouček** Jméno: **Tomáš** Osobní číslo: **507670**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra počítačů**

Studijní program: **Softwarové inženýrství a technologie**

Specializace: **Enterprise systémy**

II. ÚDAJE K BAKALÁŘSKÉ PRÁCI

Název bakalářské práce:

Časté zranitelnosti webových aplikací

Název bakalářské práce anglicky:

Web application vulnerabilities

Pokyny pro vypracování:

Seznamte se s nejčastějšími zranitelnostmi webových aplikací:
- Broken Access Control: buď DoS, nebo Privilege escalation
- Injection: SQL injection, XSS
- Security misconfiguration: directory listing, directory traversal
- Insecure design: ověření vstupu vůči nesmyslným hodnotám
- CSRF

Tyto zranitelnosti popište a na ukázkách vysvětlete, jak fungují. Doplňte je o kvantitativní statistiky výskytu z webu OWASP. Dále navrhněte a sestavte webovou aplikaci, která bude obsahovat výše uvedené bezpečnostní zranitelnosti. (Účel: prostor pro výuku a aplikaci zabezpečení webových aplikací.) Vytvořte Dockerfile, pomocí kterého bude možné web snadno a opakovatelně spustit v takovém sandboxu, kterému nebude vadit potenciální napadnutí pomocí využití zmiňovaných zranitelností.
K aplikace vytvořte testovací program, který ověří, jsou-li předem definované zranitelnosti přítomny a zneužitelny.
Ukažte, jak se lze před těmito zranitelnostmi chránit, ve dvou kontextech:
1) rukou psaného ("vanilla") PHP kódu,
2) existujícího webového PHP frameworku dle vlastní volby
Ochranu v kontextu 1 předveďte na úpravách kódu vzorové zranitelné aplikace.

Seznam doporučené literatury:

https://owasp.org/www-project-top-ten/
https://www.php.net/manual/en/security.php
https://developer.mozilla.org/en-US/docs/Learn/Server-side/First_steps/Website_security

Jméno a pracoviště vedoucí(ho) bakalářské práce:

RNDr. Ondřej Žára Katedra počítačové grafiky a interakce

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) bakalářské práce:

Datum zadání bakalářské práce: **07.02.2024** Termín odevzdání bakalářské práce: **24.05.2024**

Platnost zadání bakalářské práce: **21.09.2025**

RNDr. Ondřej Žára podpis vedoucí(ho) práce	podpis vedoucí(ho) ústavu/katedry	prof. Mgr. Petr Páta, Ph.D. podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Student bere na vědomí, že je povinen vypracovat bakalářskou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací. Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v bakalářské práci.

. _____ _____

Datum převzetí zadání Podpis studenta

Poděkování

Děkuji panu doktorovi Ondřeji Žárovi za vedení bakalářské práce a cenné rady, které mi doporučil.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškerou použitou literaturu.

V Praze, 23. května 2024

Abstrakt

Cílem mé bakalářské práce je seznámit čtenáře se základním zabezpečením webových aplikací, předvést mu možné řešení těchto zranitelností v jazyce PHP a v neposlední řadě ukázat interaktivní aplikaci pro otestování nabitých zkušeností.

Klíčová slova: Webová aplikace, PHP, Zabezpečení, Zranitelnost

Vedoucí: RnDr. Ondřej Žára

Abstract

The aim of this project thesis is to familiarize the reader with the basic security of web applications, to demonstrate possible solutions to these vulnerabilities and last but not least it contains an application to test the acquired experience.

Keywords: Web Applications, PHP, Cybersecurity, Vulnerability

Title translation: Web applications vulnerabilities — Securing web applications in PHP

Obsah

1 Úvod 1

1.1 Motivace 1

1.2 Cíl práce 1

2 Současný stav 3

2.1 Zabezpečovací frameworky 3

2.1.1 CIA 3

2.1.2 NIST 3

2.1.3 OWASP 4

2.2 Nejčastější metody útoků 5

2.2.1 Broken Access Control 5

2.2.2 Cryptographic Failures 7

2.2.3 Injection 8

2.2.4 Insecure Design 9

2.2.5 Security Misconfiguration . . . 10

2.3 Frameworky v PHP 11

2.3.1 Laravel 12

2.3.2 Symfony 12

2.3.3 Code Igniter 13

2.4 Vzdělávání v bezpečnosti webových aplikací 14

2.4.1 Videa 14

2.4.2 Knihy 14

2.4.3 Vysoké školy 14

2.4.4 Kurzy 15

3 Analýza 17

3.1 Zadání 17

3.2 Funkční požadavky 18

3.3 Nefunkční požadavky 18

4 Návrh řešení 21

4.1 Technologie 21

4.1.1 Docker služby 21

4.1.2 Serverová strana aplikace 22

4.1.3 Klientská část aplikace 23

4.1.4 Skript pro ověření přítomnosti zranitelností 23

5 Implementace 25

5.1 Broken Access Control 26

5.1.1 IDOR 26

5.1.2 CSRF 27

5.2 Cryptographic Failures 30

5.2.1 Hashování hesel 30

5.3 Injection 32

5.3.1 SQL 32

5.3.2 XSS 34

5.4 Insecure Design 37

5.4.1 Nezabezpečení vstupů 37

5.5 Security Misconfiguration 38
5.5.1 Detailní chybové hlášky 39
5.5.2 Directory traversal 40
6 Nasazení 43
6.1 Dokumentace 44
6.1.1 Aplikace 44
6.1.2 Skript 44
6.2 Postup nasazení aplikace 44
6.3 Kontrola pomocí skriptu 45
6.3.1 Argumenty pro spuštění 45
6.3.2 Použití 46
7 Závěr 47
Listings 49
Slovník 51
Slovník 51
Literatura 53

Obrázky

2.1 Graf útoků na webové aplikace mezi lety 2012 a 2016 [1] 5

4.1 Podíl web serverů na trhu [20] . . 22

Tabulky

6.1 Podporované prohlížeče a jejich verze 43

Kapitola 1

Úvod

1.1 Motivace

V posledních letech počet útoků na webové aplikace klesá, ale dopad úspěšných útoků narůstá [1]. Webové aplikace zažívají obrovský rozmach. Jako příklad uvedu Microsoft Office balíček. Daný balíček už je zcela dostupný formou webové aplikace, a tak všechny jeho prvky jako jsou například Microsoft Word nebo Microsoft Excel, už můžeme používat na webu. Webové aplikace jsou většinou veřejně všem dostupné na webu. Proto jsou často obětmi útoků, které mohou oběť útoku stát jak peníze, tak hlavně důvěru uživatelů webu. Je tedy potřeba takovéto aplikace zabezpečit proti všem reálným útokům a zmenšit riziko útoku na minimum.

Jsem si rovněž vědom skutečnosti, že na podobné téma již existují jiné práce, ale jednak mi nepřišly dostatečně aktuální, a jednak nenabízely aplikaci, na které si může čtenář své nově nabyté schopnosti otestovat, a tím i lépe zapamatovat.

1.2 Cíl práce

Tento dokument si bere za cíl jednak čtenáře seznámit s tématikou zabezpečení webových aplikací, jednak čtenáři poskytnout ukázku, jak si zkontrolovat zabezpečení vlastní webové aplikace. Případně nabídne ukázky možného řešení. V neposlední řadě ukáže, co chybné zabezpečení může způsobit.

Vedlejším cílem projektu je studentům předmětu ZWA poskytnout dodatečný materiál ke studiu, který se týká zabezpečení webových aplikací,

a jednoduše rozvést témata probíraná na přednáškách i s jednoduchými příklady, která mohou využít ve svých semestrálních pracích.

Kapitola 2

Současný stav

2.1 Zabezpečovací frameworky

Zabezpečovací frameworku je dobrovolný návod založený na existujících standardech, pokynech a postupech pro organizace, aby mohly lépe řídit a snižovat rizika kybernetické bezpečnosti [2]. Zabezpečení by mělo být nedílnou součástí každá webové aplikace. K tomu nám slouží zabezpečovací frameworky určené pro organizace. Představím tedy dva nejčastěji využívané a jeden specifický pro zabezpečení webových aplikací.

2.1.1 CIA

CIA triádový model je přes čtyřicet let považován za základní myšlenku kybernetické bezpečnosti. Popisuje: C - Confidentiality (důvěrnost), I - Integrity (integrita), A - Availability (dostupnost) a slouží ke kvalitativnímu posouzení bezpečnostních na aplikaci [3].

2.1.2 NIST

Uvědomění, že národní a ekonomická bezpečnost v USA závisí na spolehlivém fungovaní kritické infrastruktury, vedlo k příkazu na vylepšení kyberbezpečnosti kritické infrastruktury [2]. A tak vznikl NIST. Framework, který je souhrnem všech standardů, praktik a znalostí pro mitigaci a snížení rizik.

2.1.3 OWASP

OWASP (The Open Worldwide Application Security Project) je nezisková organizace, jejímž cílem je zlepšit zabezpečení softwaru [4]. Organizace pravidelně pořádá konference a pravidelně vydává shrnutí nejaktuálnějších hrozeb.

OWASP Top Ten

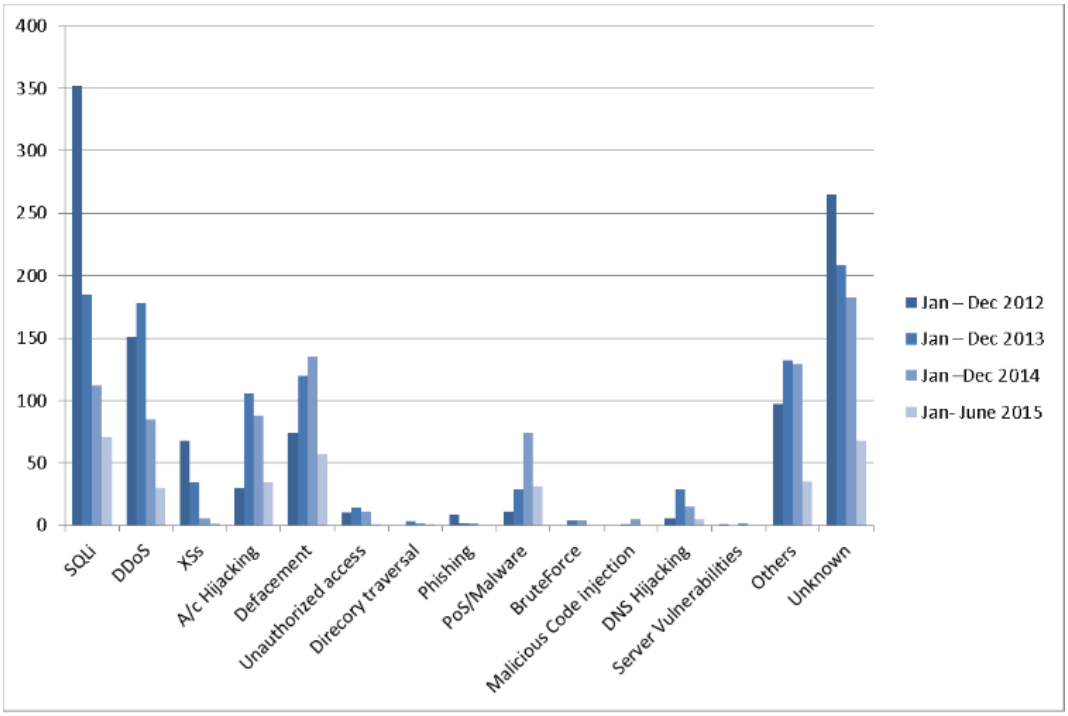
Cílem dokumentu OWASP Top Ten je zvýšit povědomí vývojářů webových aplikací o aktuálních bezpečnostních rizicích na webu. Firmy by měly tento dokument využít pro zabezpečení vlastních webových aplikací a minimalizovat rizika bezpečnostních rizik. Výstupem je seznam deseti vybraných rizik specialisty v oboru. Aktuální vydání je z roku 2021, které v práci představíme a z něhož bude aplikace vycházet.

1. A01:2021-Broken Access Control
2. A02:2021-Cryptographic Failures
3. A03:2021-Injection
4. A04:2021-Insecure Design
5. A05:2021-Security Misconfiguration
6. A06:2021-Vulnerable and Outdated Components
7. A07:2021-Identification and Authentication Failures
8. A08:2021-Software and Data Integrity Failures
9. A09:2021-Security Logging and Monitoring Failures
10. A10:2021-Server-Side Request Forgery

S vedoucím práce, panem doktorem Žárou, jsme se dohodli, že se budeme inspirovat při tvorbě práce jak dokumentem od OWASPu, tak i zkušenostmi pana Žáry z pohledu učitele předmětu ZWA. Shodli jsme se, že v práci zpracujeme z dokumentu OWASPu body 1 až 5. Z důvodu, že jednak tyto zranitelnosti se nejčastěji objevují na webových aplikací a jednak z důvodu, že na těchto zranitelnostech se dá jednoduše ukázat, jak je lze opravit.

2.2 Nejčastější metody útoků

Web je velmi oblíbeným cílem útočníků a to protože, webové aplikace často nejsou náležitě zabezpečeny. V kapitole si představíme nejčastější útoky mezi lety 2012 a 2016 dle analýzy z článku Empirical Analysis of Web Attacks [1]. V následujících podkapitolách si detailně představíme zranitelnosti z prvních pěti bodů dokumentu OWASP a jejich podtypy, které budou implementovány v přiložené aplikaci. V této práci budu používat zkratku `CWE`[1], kterou používá i OWASP pro konkrétní příklady zranitelností.



Obrázek 2.1: Graf útoků na webové aplikace mezi lety 2012 a 2016 [1]

2.2.1 Broken Access Control

Zranitelnost `Broken Access Control` umožňuje útočníkovi provést v aplikaci akce, které by uživatel se svými právy neměl býti schopen. Zmíněná zranitelnost často vede k odhalení, modifikaci či destrukci dat [6]. Dle OWASP bylo testováno na přítomnost této zranitelnosti 94 % s průměrnou incidencí 3.81 % [6]. Zranitelnosti, které dle OWASP je potřeba zmínit jsou [6] :

1. CWE-200: Exposure of Sensitive Information to an Unauthorized Actor

[1] seznam slabých míst softwaru a hardwaru vytvořený komunitou [5]

2. CWE-201: Insertion of Sensitive Information Into Sent Data
3. CWE-352: Cross-Site Request Forgery

Z těchto tří vyjmenovaných budou v mojí aplikaci implementovány konkrétní zranitelnosti jedna a tři.

IDOR

Insecure direct object references (IDOR) je zranitelnost, která útočníkovi umožňuje dostat se do míst, kam by neměl mít právo, pomocí přímého odkazu [7]. Uživatel se tedy pomocí parametrů dokáže probourat aplikací. Uvedu pro názornost příklad, máme stránku na url `/user`, kde se pomocí parametru `ID` zobrazí uživatel s daným `ID`. Když uživatel klikne na svůj profil, tak se mu zobrazí pouze jeho profil. Na zobrazení jiného profilu nemá dostatečná práva jako běžný uživatel. `URL` stránky uživatele by mohla vypadat například takto: `https://nebezpecnastranka.cz/user?ID=10`. Pokud bychom se pomocí změny parametru `ID` v `URL` dostali na profil jiného uživatele, pak je zranitelnost Insecure direct object references přítomna. Rovněž je třeba zmínit, že zranitelnost je pouze jedním typem zranitelnosti `CWE-200`.

CSRF

Cross-site request forgery (CSRF) je webová bezpečností zranitelnost, která umožňuje útočníkovi přimět uživatele provést akce, které nechtěli provést. To umožňuje útočníkovi částečně obejít politiku stejného původu, která je navržena, aby se různé webové stránky vzájemně neovlivňovaly [8]. Aby útočník mohl provést Cross-site request forgery, je potřeba splnit tyto tři podmínky [8]:

- Webová aplikace musí obsahovat funkcionalitu, kterou útočník může zneužít pro svůj prospěch.
- Webová aplikace zpracovává uživatelské relace přes `cookie`.
- `Request` nesmí obsahovat žádné náhodné parametry, které útočník nezná, nebo neumí odhadnout.

Je důležité zmínit, že se jedná o často přítomnou zranitelnost vzhledem ke statistikám zveřejněných organizcí OWASP.

2.2.2 Cryptographic Failures

Zranitelnost `Cryptographic Failures` lze rozdělit na dva lehce odlišné problémy. A to pomocí určení, kde data aktuálně jsou. Za prvé v transportu, kdy aplikace nepoužívá šifrovaný protokol `HTTPS`, a díky tomu je veškerá komunikace zobrazena komukoliv, kdo se dostane v připojení mezi nás a webovou aplikaci. Za druhé uložená data, kdy například má aplikace uložená hesla v databázi bez jakéhokoliv šifrování v lidsky čitelné podobě [9]. Podle statistik poskytnutých organizací OWASP, kde byla testována její přítomnost na 79 % testovaných subjektů, byla přítomna v 4.49 % případů [9]. Významnými podtypy této zranitelnosti dle OWASP jsou [9]:

1. CWE-327: Broken or Risky Crypto Algorithm
2. CWE-259: Use of Hard-coded Password
3. CWE-331: Insufficient Entropy

Zde jsem pro inspiraci použil bod jedna, ale místo riskantního či prolomeného algoritmu jsem nepoužil žádný algoritmus.

Hashování hesel

`Hashování hesel` je proces, při kterém aplikace ukládá hesla uživatelů do databáze v zašifrované podobě. To znamená, že hesla nejsou uložena v čitelné podobě, ale v podobě otisku, který je vygenerován pomocí hashovací funkce. Díky tomu, i když dojde k úniku databáze, útočníci nezískají hesla uživatelů v čitelné podobě a nemohou je tak snadno zneužít. Je však důležité poznamenat, že ne všechny hashovací funkce jsou si rovny. Bezpečnost uložených hesel závisí na volbě hashovací funkce, její konfiguraci a použití soli.

Volba hashovací funkce: Doporučuje se používat silné a moderní hashovací funkce, jako je bcrypt nebo Argon2 [10]. Tyto funkce jsou odolné proti útokům typu brute-force a rainbow table. Vyhněte se používání starších a zranitelných hashovacích funkcí, jako je MD5 nebo SHA-1.

Konfigurace hashovací funkce: Většina hashovacích funkcí umožňuje upravit parametry ovlivňující jejich sílu. Obecně platí, že čím více času a paměti funkce spotřebuje, tím je odolnější proti útokům. Doporučuje se nastavit parametry hashovací funkce na co nejvyšší úroveň, která je kompatibilní s hardwarem a softwarem vaší aplikace.

Použití soli: Salt neboli sůl je náhodný řetězec znaků, který se přidává k heslu před jeho shašováním. To znemožňuje útočníkům vytvářet hash

tabulky (tzv. rainbow table), které jim usnadňují prolomení hesel. Vždy bychom měli používat salt při hashování hesel. Mnoho hashovacích funkcí má vestavěnou podporu pro generování a přidávání soli.

Funkce password_hash a Hash::make: Funkce password_hash() v PHP je navržena pro bezpečné hashování hesel. Používá silnou hashovací funkci bcrypt a automaticky generuje a přidává salt. Funkce Hash::make() v Laravelu je dalším nástrojem pro bezpečné hashování hesel. Umožňuje nám vybrat si hashovací funkci a konfigurovat její parametry a automaticky generuje a přidává salt. Používáním silných hashovacích funkcí, správnou konfigurací a používáním soli můžeme zajistit, že hesla uživatelů budou uložena v bezpečí a chráněna před útoky.

2.2.3 Injection

Injection je typ útoku, kdy útočník do uživatelského pole (formulář, url) vloží škodlivý kód, u kterého nebyla provedena sanitizace, ale kód je aplikací zpracován a případně i proveden. Zde OWASP uvádí, že na zranitelnost otestoval 94% stránek a v 3% případů byla zranitelnost přítomná. [11] Nejvýznamnějšími zranitelnostmi z této kategorie dle OWASP jsou [11]:

1. CWE-79: Cross-site Scripting
2. CWE-89: SQL Injection
3. CWE-73: External Control of File Name or Path

V implementované aplikaci jsem zvolil, vzhledem k jejich rozšířenosti, body jedna a dva.

SQL

Útok `SQL Injection` se čtenáři nejčastěji vybaví při názvu `injection`. Spočívá v neošetřeném uživatelském vstupu, který není na serverové straně nijak kontrolován. To vede k situaci, kdy dotaz spustí jak původní dotaz, tak i dodatečný jenž vložil útočník. Berme v potaz situaci, kdy máme formulář, v němž uživatel vyplní email a heslo, a aplikace následně pošle dotaz, zda, existuje uživatel s takovým heslem a jménem. V tomto příkladu nebudeme brát v potaz správné či špatné řešení, ale budeme jej využívat pouze jako příklad. Útočník vyplnil formulář s daty: email:`a@a.com` a heslo: `' OR 1=1; --`. Taktéž ignorujme fakt, že by databáze ukládala hesla v zabezpečené formě.

```
SELECT COUNT() FROM users WHERE name = 'uzivatel@uzivatel.com'
    AND password = ' ' OR 1=1; -- ';
```

Listing 2.1: Útok SQL Injection

V ukázce vidíme, že místo toho, aby porovnal heslo, přihlásí útočníka pomocí vyplněného hesla na jakýkoliv účet, pouze znalostí uživatelova emailu. Když útočníkův vstup rozebereme znak po znaku, tak část `'` ukončí dotaz na heslo. `' OR 1=1;` zajistí, že dotaz vrátí `true` a `--` zajistí, že cokoliv by následovalo, bude ignorováno. Dalším problémem `SQL Injection` je, že útočník nemusí umět `SQL`, protože jsou veřejně dostupné nástroje. Postačí mu např. sqlmap, kterému stačí dodat jen adresu, na kterou má útočit. Zbytek vyřeší skript za útočníka.

XSS

Cross Site Scripting (XSS) je typ útoku, kdy je útočníkův skript vložen do jinak bezpečné stránky. Vzniká tak situace, kdy stránka zobrazí uživatelův vstup bez sanitizace, a tím spustí skript [12]. XSS se dělí na dva typy *stored* a *reflected*. Reflected XSS vzniká, když jeden `HTTP request`, který není nikde uložen na stránce, doručí a provede XSS payload [12]. Proti tomuto typu se prohlížeče brání pomocí `XSS auditory`, přes to útočníci stále dokáží obejít `XSS auditory` [12]. Druhým typem je *stored*, což je stav, kdy se vstup, který útočník zadal, uloží na serveru a při opětovném načtení stránky zůstává na stránce. Navíc pokud stránka, kde se tento kód zobrazuje, je veřejná, pak útočníkovi stačí poslat oběti email a každému, kdo tuto stránku otevře, se spustí útočníkův skript. Proto je tento typ považován za nebezpečnější [13]. Jako příklad uvedu jednoduchý skript, kterým si útočník ověřuje, zda je zranitelnost XSS na stránce přítomna:

```
<script>alert(1);</script>
```

Listing 2.2: Test přítomnosti XSS

2.2.4 Insecure Design

Insecure Design je široká kategorie představující různé nedostatky, vyjádřené jako *chybějící* nebo *neúčinný* návrh aplikace [14]. Tyto zranitelnosti tedy na rozdíl od ostatních v tomto seznamu nevznikají při vývoji, nýbrž již při návrhu aplikace jako takové. Tato kategorie vznikla v OWASP seznamu nově. Byla testována na 77 % stránkách a její přítomnost byla zjištěna v 3 % [14]. Nejvýznamnějšími zástupci z řad zranitelností dle OWASP jsou [14]:

1. CWE-209: Generation of Error Message Containing Sensitive Information

2. CWE-256: Unprotected Storage of Credentials
3. CWE-501: Trust Boundary Violation
4. CWE-522: Insufficiently Protected Credentials

Z těchto zranitelností jsem si pro implementaci do aplikace vybral body jedna a dva, které ale spadají i do jiných kategorií. Proto jsou rozepsány jinde a zde jsem zvolil pro implementaci příklady, které OWASP uvádí v popisu zranitelnosti.

Nezabezpečení vstupů

Zranitelnost spočívá v nesprávném ošetření vstupů a následném dopadu na aplikaci. Jako příklad uvedu e-shop, kde uživatel si může rezervovat 1000 kusů zboží na dobu 30 dní bez jakéhokoliv poplatku. Pak e-shop bude tratit, protože ostatní zákazníci, kteří by si zboží zakoupili, si ho koupit nemohou, protože všechny kusy jsou zarezervované.

Session hijack

Session hijack je situace, kdy webová aplikace řeší uživatelské relace pomocí `cookie` a útočník jakýmkoliv způsobem se zmocní této `cookie`. Pak se může vydávat za uživatele, jemuž byla `cookie` odcizena.

2.2.5 Security Misconfiguration

Nastavení všech částí aplikace je důležité. I proto je zranitelnost pátá v dokumentu OWASP 2021 a od verze dokumentu z roku 2017 se posunula o jedno místo níže [15]. Představím tři různé zranitelnosti, které vzniknou, když se nedostatečně věnuje pozornost nastavení aplikace. Rovněž zmíním incidenci dle OWASP. Testováno bylo 90% aplikací a v 4% byla nalezena [15]. Jako nejvýznamější zmiňuje OWASP:

1. CWE-16 Configuration
2. CWE-611 Improper Restriction of XML External Entity Reference

První bod zahrnuje množství špatně nastavené konfigurace. Mezi tyto případy spadají i níže vyjmenované, které jsou v aplikaci přítomny.

Detailní chybové hlášky

Chybové hlášky nabízejí vývojáři při vývoji informace o tom, kde se chyba objevila a co ji mohlo způsobit. Jednou z informací může být například verze serveru či jiná citlivá informace, kterou může útočník využít ve svůj prospěch při útoku na aplikaci.

Directory traversal

Directory traversal také známý jako Path traversal je zranitelnost, při které se dokáže uživatel pomocí změny `URL` dostat na soubory na serveru, na kterém webová aplikace běží. Pro představu předvedu jednoduchý příklad [16]. Máme aplikaci která, zobrazuje obrázky. `URL` jedné z takových stránek by vypadala takto:

```
1 https://nezabezpecastranka.cz/loadImage?filename=img.jpg
```

Listing 2.3: Directory traversal v URL

Pokud by útočník změnil parametr *filename* na jinou hodnotu např:`../../../etc/passwd`, a tento soubor by byl čitelný pro aktuálního uživatele, pod nímž běží webový server, a pokud bude tato zranitelnost v aplikaci přítomna, bude útočníkovi soubor zobrazen.

Zanechání přednastavených hesel

Má-li má útočník informace o systémech, které používáme jako například databázový server, pak se nejdříve zkusí přihlásit pomocí přednastaveného uživatelského jména a hesla, která jsou veřejně dostupná v dokumentaci systému.

2.3 Frameworky v PHP

Použijeme-li při vývoji webové aplikace použijeme framework, předpokládáme, že framework již nějak bezpečnost řeší. V kapitole si ukážeme jednak nejčastěji používané frameworky v PHP na tvorbu webu a jednak i to, jak tyto frameworky zajišťují bezpečnost za vývojáře. Dle Stack Overflow Developer Survey pro rok 2023, jsou třemi nejpoužívanějšimi PHP frameworky: Laravel, Symfony a CodeIgniter [17]. Každý z nich v následujících podkapitolách krátce představím a uvedu jak každý z nich řeší zranitelnosti, kterými se, v této práci budeme zabývat.

2.3.1 Laravel

Laravel je framework sloužící pro rychlejší a jednodušší vývoj webových aplikací. Je založen na návrhovém vzoru MVC a umožňuje psát čistější a udržovatelnější kód. Obsahuje většinu vlastností, které musí programátor do aplikace jinak sám složitě implementovat. Jsou to např: Autentizace, Autorizace, ORM. Nás zajímají vlastnosti spojené s bezpečností. V následujících podkapitolách si popíšeme, co obsahuje balíček `Bezpečnost` v Laravelu.

1. Broken Access Control - Zranitelnost CSRF je ošetřena v šablonovacím enginu `blade`, ve kterém nám stačí pouze přidat direktivu `@csrf`.
2. Cryptographic Failures - Hesla musí vývojář stále hashovat. Jen Laravel mu nabízí vlastní metody hashování navíc `Hash::make('LaravelIsCool');`.
3. Injection - V Laravelu jsou SQL Injections řešené pomocí bindování a query builder. XSS Injection neřeší. Musíme tedy aktivně řešit buď pomocí šablonovacího enginu nebo na serveru pomocí PHP funkcí jako je například `striptags`.
4. Insecure Design - tuto problematiku za nás žádný framework nevyřeší, protože vzniká již při návrhu aplikace.
5. Security Misconfiguration - Laravel zobrazení chybových hlášek řeší přes `.ENV` a vlastnosti `APP_ENVIRONMENT` a `APP_DEBUG`, které určují, jak moc detailní chybové hlášky se mají zobrazovat. Proto vývojář musí tyto hodnoty změnit, když aplikaci zveřejní pro veřejnost. V základním nastavení vystavuje pouze složku `public`, proto jakýkoliv *traversal* mimo složku `public` je vyloučen.

2.3.2 Symfony

Symfony je množina znovupoužitelných PHP komponent a framework pro tvorbu webových aplikací [18]. Oproti Laravelu se hodí na velké projekty, kdežto Laravel v tomto ohledu je spíše na menší projekty, to ale neznamená, že frameworky nemůžeme použít jinak. Symfony je použit na obrovských službách jako jsou Trivago, Vogue a nebo Spotify. Jako u Laravelu nás především zajímá, jak je na tom framework s bezpečností.

1. Broken Access Control - Symfony pro autorizaci uživatelů používá role. Uživatelé mají přiřazené svoje role a tyto role pro uživatele získáme voláním funkce `getRoles()`. Ochranu proti CSRF má framework zabudovanou a stačí jí pouze nastavit v konfiguračním souboru.

2. Cryptographic Failures - hashování hesel podporuje v základu a stejně jako ochrana proti CSRF nám stačí pouze nastavit, jaké hashování chceme použít v konfiguračním souboru.
3. Injection - v základu není žádná injection řešena, je tedy na vývojáři, aby tuto zranitelnost vyřešil. Můžeme ale využít Doctrine DQL, kterou Symfony podporuje.
4. Insecure Design - tuto problematiku za nás žádný framework nevyřeší, protože vzniká již při návrhu aplikace.
5. Security Misconfiguration - Pro obranu proti Directory Traversal útokům musíme použít základní funkce PHP, kterými jsou `realpath()` a `basename()`, jejichž kombinací můžeme zajistit, aby tato zranitelnost nebyla zneužitelná. Chybové hlášky zobrazuje detailní pouze v situaci, kdy je aplikace označena v konfiguračním souboru jako ve vývoji. Jinak zobrazuje pouze chybu bez zneužitelných detailů.

2.3.3 Code Igniter

Velice malý a tedy i jednoduchý framework. Jeho jednoduchost tkví v jeho minimální konfiguraci. Jeho velikost je aktuálně k datu 4. 2. 2024 pouze 1.1MB. Doporučuje vývojáři použít MVC, ale není povinné ho použít. Vzhledem k velikosti je zde méně kladen důraz na bezpečnost na straně frameworku a vývojář musí zranitelnosti, které jsou u jiných frameworků řešené frameworkem, zabezpečit sám.

1. Broken Access Control - Autorizaci framework řeší podobně jako předchozí dva, ale dle mého názoru méně sofistikovaně pomocí polí v konfiguračním souboru. Ochranu proti CSRF framework obsahuje a stačí ji pomocí postupu v dokumentaci správně nastavit.
2. Cryptographic Failures - hashování hesel podporuje v základu a stejně jako ochrana proti CSRF nám stačí pouze nastavit jaké hashování chceme použít v konfiguračním souboru.
3. Injection - framework za nás tento typ chyby neřeší. Vývojář tedy musí každý uživatelský vstup *očistit*, jak bude v této práci vysvětleno. U SQL injection můžeme použít například `bindování`.
4. Insecure Design - tuto problematiku za nás žádný framework nevyřeší, protože vzniká již při návrhu aplikace.
5. Security Misconfiguration - Pro obranu proti Directory Traversal útoku dokumentace CodeIgniteru doporučuje použít funkci `sanitizeFilename()`. Jako v předešlých frameworcích je zde detailnost chybové hlášky řešena pomocí nastavení parametru, zda je aplikace ve vývoji či nikoliv.

2.4 Vzdělávání v bezpečnosti webových aplikací

Daná práce slouží čtenáři jako vhled do složité problematiky bezpečnosti. Věnuje se pouze konkrétní problematice a tou je zabezpečení webových aplikací v jazyce PHP. V této kapitole, vám chci představit další možné zdroje, které mohou pomoci lépe pochopit či zlepšit bezpečnost webových aplikací napsaných v PHP. Bezpečnost webových aplikací je stále aktuálním a komplexním tématem, a je nezbytné přistupovat k ní s odpovídající vážností.

2.4.1 Videa

Internet či konkrétněji platforma YouTube nabízí v dnešní době mnoho kvalitního obsahu zdarma. Obecně na platformách, kam může nahrávat videa každý uživatel, je problém najít kvalitní informace. Zmiňme tedy pár videí či kanálů, za jejichž kvalitou si stojím.

- Web Application Ethical Hacking - Penetration Testing Course for Beginners
- Série videí zaměřených na konkrétní zranitelnosti webových aplikací
  - Cross Site Request Forgery - Computerphile
  - Hacking Websites with SQL Injection - Computerphile
  - Cracking Websites with Cross Site Scripting - Computerphile

2.4.2 Knihy

V tomto odstavci uvedu literaturu/publikace, které lze využít při zpracování tématu.

- Real-World Bug Hunting, o danou knihu se opírá má práce, čerpám z ní.
- Web Security for Developers

2.4.3 Vysoké školy

Otázkou zabezpečení se zabývá i mnoho vysokých škol v České republice. Moje osobní zkušenosti jsou pouze s ČVUT FEL, kde jsem úspěšně absolvoval předmět Základy webových aplikací. V němž se základům bezpečnosti

webových aplikací věnuje jen část předmětu, protože objem vyučované látky je sám o sobě objemný. Následný seznam je tedy pouze informativní, protože s danými vysokými školami nemám vlastní zkušenost.

- ČVUT FIT - bakalářské studium - informační bezpečnost
- MUNI - bakalářský obor Kyberbezpečnost
- ČVUT FEL - magisterské studium - Kybernetická bezpečnost

2.4.4 Kurzy

V informatické bezpečnosti je často kladen důraz na placené certifikáty a kurzy. Velice kvalitními a uznávanými jsou certifikáty od firmy Offensive Security. Kvalitě odpovídá jejich vysoká cena. Poskytují certifikáty z oblasti penetračního testování a jejich certifikace pokrývají množství podskupin jako jsou: webové aplikace, wifi bezpečnost a vývoj. Zde jsou dva kurzy/certifikáty zaměřené na webovou bezpečnost.

- WEB-200: Foundational Web Application Assessments with Kali Linux
- WEB-300: Advanced Web Attacks and Exploitation

Dalším kurzem, který můžu jen doporučit, je kurz: Burp Suite Certified Practitioner od vývojářů nástroje na penetrační testování Burp Suite. Kurz nemá takovou kvalitu jako předešlé zmíněné kurzy, ale kompenzuje to svojí cenou. V práci využívám výukové materiály tohoto kurzu.

Kapitola 3

Analýza

3.1 Zadání

Zadáním bakalářské práce je:

1. Seznámit se s nejčastějšími zranitelnostmi webových aplikací:
   - Broken Access Control: buď DoS, nebo Privilege escalation
   - Injection: SQL injection, XSS
   - Security misconfiguration: directory listing, directory traversal
   - Insecure design: ověření vstupu vůči nesmyslným hodnotám
   - CSRF
2. Popsat uvedené zranitelnosti a na ukázkách vysvětlit, jak fungují. Doplnit kvantitativní statistiky z výskytu z webu OWASP.
3. Navrhnout a sestavit webovou aplikace, která bude obsahovat výše uvedené bezpečnostní zranitelnosti.
4. Vytvořit Dockerfile, pomocí kterého bude možné web snadno a opakovatelně spustit v `sandboxu`[1], kterému nebude vadit potencionální napadnutí pomocí zmiňovaných zranitelností.
5. Vytvořit k aplikaci testovací program, který ověří, jsou-li předem definované zranitelnosti přítomny a zneužitelné.
6. Ukázat, jak se lze před těmito zranitelnostmi chránit, ve dvou kontextech:

[1] oddělené prostředí pro program který běží v bezpečném prostředí proti zneužití [19]

a. rukou psaného ("vanilla") PHP kódu
b. existujícího webového PHP frameworku dle vlastní volby

Ochranu v kontextu 1 předveďte na úpravách kódu vzorové zranitelné aplikace.

3.2 Funkční požadavky

Funkční požadavky vycházejí ze zadaní vedoucího projektu. Jejich cílem je vydefinovat, co systém uživateli umožní, aby aplikace splnila zadání.

1. FR-1 - Uživatel má možnost se na stránce zaregistrovat.
2. FR-2 - Uživatel, který je zaregistrován, má možnost se přihlásit.
3. FR-3 - Uživatel, který je `adminem`, má možnost přidat nový článek.
4. FR-4 - Uživatel, který je přihlášen, má možnost přidat k existujícímu článku.
5. FR-5 - Uživatel má možnost zobrazit se články po kategoriích či celkově.
6. FR-6 - Uživatel, který je přihlášen, má možnost se odhlásit.
7. FR-7 - Uživatel, který je přihlášen, má si zobrazit vlastní profil.
8. FR-8 - Uživatel, který je přihlášen, má možnost změnit si heslo.
9. FR-9 - Uživatel, který je přihlášen, má možnost zobrazit si obrázky, které jsou uloženy na serveru.

3.3 Nefunkční požadavky

Ze zadání práce jsem vyvodil také nefunkční požadavky, které aplikace musí pokrývat, aby splnila zadání.

1. NFR-1 - Systém bude obsahovat bezpečnostní zranitelnosti předem určené vedoucím práce.
   - Implementovány budou zranitelnosti z prvních pěti bodů dokumentu OWASP Top 10 pro rok 2021.
   - Zranitelnosti budou implementovány tak, aby se daly zásahem do kódu či nastavení opravit.

2. NFR-2 - Zdrojový kód systému bude dodržovat základy čistého kódu.
   - Přestože má být aplikace děravá, stále musí udržet čistý kód, aby mohl uživatel chyby nalézt.
   - Netriviální části kódu aplikace by měly být patřičně okomentovány.
3. NFR-3 - Systém bude jednoduše a opakovatelně spustitelný.
   - Uživatel musí být schopen si *rozbitou* aplikaci znovu spustit.
4. NFR-4 - Systém bude dostupný na veřejném úložišti.
   - Uživatel musí mít možnost, jak získat zdrojový kód aplikace.

Kapitola 4

Návrh řešení

Navrhnutým řešením je vytvořit aplikaci k této bakalářské práci, která čtenáři umožní vyzkoušet si, zda dokáže zranitelnosti v kódu najít a posléze opravit. Aplikace bude rozdělená na dvě části. A to: na webovou aplikaci, kde uživatel může najít chyby z pohledu útočníka. Následně webovou aplikaci doplní skript, který automaticky projde konkrétní webovou aplikaci a uživatele upozorní, kde se nachází neopravené chyby. Funkční požadavek `FRQ-3` bude řešen pomocí docker kontejnerů, které se spustí pomocí `compose`.

4.1 Technologie

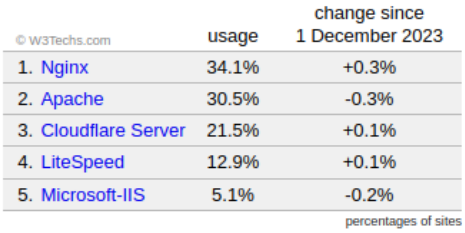
Aplikaci lze spustit přes příkaz `docker-compose up`, který spustí tři služby, které zajistí chod celé aplikace. Jsou popsané v následující kapitole.

4.1.1 Docker služby

Webová aplikace musí běžet na několika serverech. V následujících podkapitolách každý server potřebný pro aplikaci a službu, na které tento server běží, vysvětlím a přiblížím jeho důležitost ve fungování aplikaci.

Webová služba

Pro webový server, na kterém aplikace poběží, jsem zvolil nginx. Jednak z důvodu snadné konfigurace a jednoduchosti, jednak z důvodu, že je nejpoužívanějším webovým serverem k datu 11. ledna roku 2024 [20].



© W3Techs.com	usage	change since 1 December 2023
1. Nginx	34.1%	+0.3%
2. Apache	30.5%	-0.3%
3. Cloudflare Server	21.5%	+0.1%
4. LiteSpeed	12.9%	+0.1%
5. Microsoft-IIS	5.1%	-0.2%

percentages of sites

Obrázek 4.1: Podíl web serverů na trhu [20]

Databázová služba

Jako databázový server jsem vybíral ze dvou možností: MariaDB a PostgresSQL. Nakonec jsem zvolil MariaDB ze dvou důvodů. Jedná se o open source, je tedy zdarma, a tak se na ni nevztahují žádná autorská práva. Zadruhé je zcela kompatibilní s komerční MySQL, který je aktuálně druhým nejpoužívanějším databázovým serverem [21].

4.1.2 Serverová strana aplikace

Pro aplikaci serverovou stranu webové aplikace jsem si vybral programovací jazyk PHP. Je nejpoužívanějším jazykem na serverové straně webových aplikací [20]. Zvolil jsem základní PHP verze 8.2.12 bez frameworků, protože si myslím, že vývojář by měl vědět, jak věci fungují na té nejnižší vrstvě, tedy bez použití frameworku. Frameworky totiž mnoho zabezpečení dělají „za nás“ a tak v jiném frameworku, který bezpečnost neřeší jako první, bychom mohli ponechat bezpečnostní zranitelnost. V této kapitole popíšu návrh aplikace pouze obecně a nebudu zacházet do detailů. Takto jsem se rozhodl, protože aplikace je již z pohledu bezpečnosti navržena záměrně špatně. Proto si myslím, že je zbytečné do detailů rozebírat návrh aplikace. Zmíním tedy pouze místa návrhu, která souvisí s bezpečnostními riziky.

Architektura

Aplikace má monolitickou architekturu. Toto rozhodnutí jsem učinil, protože jsem chtěl, aby aplikace byla co nejjednodušší na pochopení a aby čtenář více času věnoval zranitelnostem a ne pochopení, jak aplikace funguje. Aplikace rovněž nevyužívá žádný známý návrhový vzor jako je MVC ze stejného důvodu. Může to být rovněž vnímáno jako zranitelnost aplikace, neboť kód je méně přehledný a je větší pravděpodobnost, že vývojář nějakou zranitelnost v kódu přehlédne. Smyslem aplikace je být zranitelná a zároveň špatně navržená. Pokud bychom tedy chtěli podobnou aplikaci vyvíjet, měli bychom

využít některý návrhový vzor a nejlépe aplikaci rozdělit do microslužeb. Toto rozhodnutí by aplikaci udělalo přehlednější pro vývojáře a zajistilo by vyšší škálovatelnost a modularitu. Zdrojový kód aplikace je rozdělen dle funkčnosti na dvě hlavní části v serverové části. Jednou z nich je `/pages`. Každý PHP soubor v této složce reprezentuje různé stránky aplikace, jako jsou přihlášení, registrace, zobrazení článků atd. Druhou složkou je `/helpers`. Ta obsahuje pomocné funkce, například pro manipulaci s články, uživateli nebo s databází.

Databáze

Pro aplikaci byl vytvořen databázový model. Jak již bylo řečeno, aplikace funguje na bázi novinkových webů. Aplikace má velice jednoduchý databázový model, protože jej tvoří pouze čtyři tabulky.

4.1.3 Klientská část aplikace

Klientská část je řešena přes HTML, CSS a Javascript. Pro ikonky je použita knihovna fontawesome. Veškeré obrázky pro klientskou část byly vygenerovány pomocí neuronové sítě Dall E. Klientská část taktéž využívá moderní komponenty ze standartu HTML5.

4.1.4 Skript pro ověření přítomnosti zranitelností

Skript bude pracovat na principu scrapovaní HTTP endpointu a následném procházení parsovaného HTML. Tato metoda nám umožní otestovat stránku rychle a bez užití prohlížeče jako jiné metody. Nevýhodou této metody je, že spoléhá na hledání elementů ne vizuálně, ale podle `id` a jiných parametrů, které může vývojář kdykoliv změnit. Struktura kódu je organizována tak, aby byla jednoduše upravitelná i čitelná.

Zdrojový kód tohoto skriptu je oddělen od zbytku webové aplikace a nachází se ve složce: `vulnerability_checker`. V této složce se nachází soubor `requirements.txt`. Tento soubor slouží k instalaci všech potřebných knihoven, které jsou potřeba ke spuštění skriptu. Ve zmíněné složce se ještě nachází složka `src`, která obsahuje veškerý zdrojový kód. Zde jsou tyto soubory. V adresáři `crawler` se nachází implementace webového crawleru v souboru `web_crawler.py`. Tento modul je zodpovědný za procházení webové stránky pomocí HTTP požadavků. Složka `enums` obsahuje všechny třídy typu `Výčet` pro definování konstantních hodnot v aplikaci. Konkrétně soubor `enum_request_constants.py` definuje konstanty, které se vícekrát používají

v HTTP požadavcích. V adresáři `helpers` jsou umístěny pomocné funkce používané v aplikaci. Například soubor `network_helper.py` obsahuje funkce pro práci se sítí, jako je ověření, zda zařízení na dané adrese skutečně má vystavenou webovou stránku. Dalším příkladem je `string_parsing_helper.py`, jenž obsahuje funkce pro řetězců získávání hodnot z řetězců. Ve složce `vulnerabilities` jsou umístěny implementace detekce zranitelností. Hlavním souborem je `abstract_vulnerability.py`, který obsahuje abstraktní třídu, od které všechny konkrétní implementace dědí. Každý jiný soubor ve složce obsahuje implementaci jedné konkrétní zranitelnosti, jako například `broken_access_control_vulnerability.py` nebo `injection\_vulnerability.py`.

Po konzultaci s panem doktorem Žárou jsme rozhodli, že skript bude v aplikaci kontrolovat přítomnost zranitelnosti pouze na jednom místě. Aplikace jako taková obsahuje tentýž typ zranitelnosti na více místech, protože při vývoji se nestane, aby aplikace na jednom místě měla zranitelnost ošetřenou, a na jiném ne. Tím spíše u aplikace této velikosti, kterou zvládne vytvořit jeden vývojář. Pro jednoduchost skriptu a porozumění, jak přesně funguje, je kontrola zranitelnosti pouze na konkrétním místě v aplikaci, které bylo vybráno tak, aby byl ve skriptu co nejlépe vidět postup. Pokud bychom tedy korektně chtěli udělat penetrační testování, pak bychom otestovali všechny uživatelské vstupy a jiná potencionálně náchylná místa k útoku.

Kapitola 5

Implementace

Jak již bylo zmíněno v předchozí kapitole, implementace obsahuje dvě důležité části. Zranitelnou webovou aplikaci napsanou v jazyce PHP a skript v jazyce Python, který webovou aplikaci otestuje, zda obsahuje zmíněné zranitelnosti. Výše uvedený skript nebude testovat všechny zranitelnosti zmíněné v kapitole Nejčastější metody útoků. Tato skutečnost je z důvodu, že ne všechny zranitelnosti lze pomocí skriptu a posílání požadavků na stránku otestovat. Mezi zranitelnosti, které touto formou otestovat nelze, patří například `Hashování hesel`. To nemáme možnost zjistit jako uživatel, pokud i aplikace není špatně navržená, ale tím by se jednalo o kombinaci, což by bylo proti myšlence skriptu. Úkolem skriptu je zranitelnosti otestovat nezávisle na sobě, tak aby test jedné neovlivnil test druhé. V kapitole se již nebudu věnovat vysvětlování, jak zranitelnosti fungují, poněvadž tomu již byla věnována celá kapitola Nejčastější metody útoků. Tato kapitola tedy bude věnována implementacím zranitelností ve webové aplikaci, implementaci jejího otestování ve skriptu a v neposlední řadě ukázka možného opravení chyby. V této kapitole se budou nacházet dva typy ukázky kódu. Ukázky kódu aplikace v PHP a ukázky skriptu v Pythonu. Níže ukazuji jaké barvy pou69v8 pro zv7razn2n9 syntaxe jaký jazyk.

```
1 def test():
2     # comment
3     return "true" == True
```

Listing 5.1: Ukázkový kód v jazyce Python

```
1 function test() {
2     // comment
3     return "true" == true;
4 }
```

Listing 5.2: Ukázkový kód v jazyce PHP

5.1 Broken Access Control

V kategorii Broken Access Control jsou vybrány dvě zcela odlišné zranitelnosti, kde každá z nich vyžaduje jiný postup. Jednak zranitelnost `IDOR`, která se jednoduše testuje i vysvětluje, jednak zranitelnost `CSRF` jejíž pochopení i otestování je dle mého názoru principiálně složitější. Obě zranitelnosti jsou implementované v aplikaci a jejich přítomnost je testována pomocí skriptu.

5.1.1 IDOR

Pro zopakování: zranitelnost spočívá v nedostatečném zabezpečení zobrazení dat. Umožní tedy uživateli pomocí pouhé změny parametru zobrazit si obsah, k němuž by neměl mít přístup.

Implementaci v aplikaci

V aplikaci je stránka uživatelský profil, kam se po kliknutí na nápis *Profile* v menu, uživatel dostane. Uživateli se zobrazí jeho profil se zobrazeným heslem a formulářem na změnu hesla. URL na této stránce vypadá následovně: `http://nebezpecnastranka.cz/user?id=1`. Můžeme tedy se pokusit změnit parametr `id`. Po změně parametru se nám zobrazí profil jiného uživatele a jeho heslo, které můžeme i změnit. V kódu aplikace tedy na stránce profilu chybí jakákoliv autorizace, která by zajistila přístup pouze těm uživatelům, kteří mají práva.

Nalezení pomocí skriptu

Funkce `try_idor()` přijímá dva parametry: url stránky a uživatelské *id*. Skript funguje tak, že zaregistruje uživatele s emailem *script@script.com* a uloží si jeho *id*. Toto *id* je poté využito v této funkci, kde jako parametr pošle *id* o jedno menší, jelikož uživatel s tímto *id* by měl existovat. Poté si pomocí funkce `create_html_tree()` vytvoří z odpovědi `DOM`, ve kterém ověří zda email, který se na stránce zobrazil, je jiný než uživatele, se kterým jsme se registrovali, a také zda vůbec se nám zobrazil nějaký email.

```
1 @staticmethod
2     def try_idor(user_id, url):
3         url = "http://" + url + "/user"
4 
5         response = requests.get(url, params={"id": int(user_id)
    - 1})
```

```
6         tree = create_html_tree(response.text)
7         usermail = tree.find("input", id="email")
8         if usermail is not None and usermail.get("value") != "
      script@script.com":
9             return True
10        return False
```

Listing 5.3: Kód pro ověření přítomnosti zranitelnosti IDOR

Ukázka možné opravy zranitelnosti

Zmíním dva možné postupy jak tuto zranitelnost opravit. Jedním způsobem je nemít identifikátory jako sekvence čísel, ale náhodné sekvence znaků jako je například hash nebo *UUID*. Tento přístup sice tento problém částečně řeší. Útočník s velkým výpočetním výkonem může tyto identifikátory náhodně zkoušet pomocí útoku hrubou silou, a pokud nebudou dostatečně dlouhé, pak útočník tuto zranitelnost zneužije. Druhým a dle mé zkušenosti lepším řešením je zavedení autorizace. Ukázka kódu kontroluje pomocí funkce `getUserEmailById()`, do níž vložíme *id* uživatele, jehož si chceme zobrazit, získáme email uživatele, jehož profil si chce chceme zobrazit. Ten porovná s emailem, který je uložen v relaci, pod kterou je uživatel přihlášen. Pokud se shoduje, pak se uživateli zobrazí jeho profil, pokud nikoli, pak se mu zobrazí chybová hláška, že není autorizován pro zobrazení tohoto obsahu.

```
1 <?php if($_GET["id"] == -1 || $_SESSION["email"] !=
      getUserEmailById($_GET["id"])) { ?>
2     <!-- Displaying a message for unauthorized access -->
3 <?php } else { ?>
4     <!-- Displaying user profile information and password change
       form -->
5 <?php } ?>
```

Listing 5.4: Kód pro autorizaci při IDOR

5.1.2 CSRF

Zranitelnost spočívá v možnosti zneužití HTTP požadavku s údaji jiného uživatele. Pokud vývojář neimplementuje adekvátní ochranná opatření, může být tato zranitelnost v aplikaci přítomna pro zneužití útočníky.

Implementaci v aplikaci

V aplikaci je tato zranitelnost implementována všude, kde je jakákoliv akce od uživatele. Je tedy přítomna u registrace, přihlášení, změny hesla či napsání

komentáře. U změny hesla je nejvíce zneužitelná, protože útočník může změnit heslo uživatele.

Nalezení pomocí skriptu

Zde je kontrola přítomnosti zranitelnosti, vzhledem k tomu, jak útok funguje, lehce komplikovaná. Skript pošle dotaz jako *oběť* pro získání cookies a vygeneruje mu náhodný mail. Poté pomocí HTTP požadavků zaregistruje oběť s tímto emailem a přihlásí ji. Vygeneruje se mu náhodné heslo, abychom mohli porovnat, zda byl útok úspěšný. Skript poté pošle HTTP požadavek s cookie uživatele. Tedy jakoby by tato *oběť* klikla na odkaz na stránce nasazené útočníkem. Poté už jen ověří, zda se heslo uživatele změnilo, a pokud ano, pak byl útok úspěšný.

```
@staticmethod
    def try_csrf(url, crawler):
        victim_cookie = crawler.get_cookies(url)
        victim_mail = random_char(5)+"victim@victim.com"
        crawler.register(url, victim_mail, "victim",
    victim_cookie)
        crawler.login(url, victim_mail, "victim", victim_cookie)
        new_url = "http://" + url + "/user"
        random_password = "".join(random.choices(string.
    ascii_uppercase + string.digits, k=10))
        headers = {
            headers
        }
        data = {
            "password": random_password
        }
        requests.post(new_url, headers=headers, data=data)
        victim_id = WebCrawler.get_user_id(url, victim_cookie)
        url = "http://" + url + "/user?id="+victim_id
        response = requests.get(url)
        if random_password in response.text:
            return True
        else:
            return False
```

Listing 5.5: Funkce pro kontrolu přítomnosti CSRF

Ukázka možné opravy zranitelnosti

Mechanismů obrany je více, ale zmíním dva které jsou od sebe odlišné a pouze jeden z nich musí vývojář implementovat.

V roce 2016 přišla společnost Google s novým parametrem cookie: *SameSite*. Tato cookie má tři možné hodnoty: *Lax*, při které se cookie přenáší po celé

síti se stejným prefixem, *Strict*, kdy cookie pouze platí pro konkretní stránku, a *None*, kdy je přenášena po všech stránkách. Chrome se tedy v roce 2020 rozhodl cookie, které parametr *SameSite* neměly vyplněný, považovat, jako by měly nastavenou hodnotu *Lax*. Je rovněž důležité zmínit, že v aktuální době již všechny hlavní prohlížeče považují hodnotu *Lax* za výchozí. Prohlížeče tedy brání před touto zranitelností, ale žádná ochrana není stoprocentní, a tak i zde lze tuto ochranu obejít, pokud je *SameSite* cookie nastavena na hodnotu *Lax*. Zde přikládám článek, kde se tomuto tématu věnují více do detailů: Obejití *SameSite* cookie.

Druhým řešením je použití CSRF tokenu. Tedy tokenu, který byl přímo vymyšlen jako obrana proti tomuto útoku. Stejně jako předešlé i toto se dá obejít, ale kombinace více zabezpečení zmenšuje riziko útoku. Tyto tokeny musí být dostatečně náhodné, aby se nedaly útočníkem uhádnout a zároveň se musí přenášet ve formuláři. Níže si předvedeme ochranu na formuláři, který slouží ke změně hesla.

Zde je formulář na změnu hesla bez jakéhokoliv zabezpečení:

```
<form method="POST" action="/user">
    <input type="password" id="password" name="password">
    <button type="submit">Change</button>
</form>
```

Listing 5.6: Formulář bez zabezpečení proti CSRF

Tento formulář se odesílá na stránku `/user`. Pro zabezpečení proti zranitelnosti CSRF přidáme do formuláře skryté pole CSRF token s náhodným tokenem, který útočník nemá matematicky šanci v reálném čase uhodnout. Do stránky při načtení vložíme tento kód pro vygenerování CSRF tokenu:

```
if (!isset($_SESSION["csrf_token"])) {
    $_SESSION["csrf_token"] = bin2hex(random_bytes(32));
}
```

Listing 5.7: Vygenerování CSRF tokenu

Formulář tedy pak bude vypadat následovně:

```
<form method="POST" action="/user">
    <input type="hidden" name="csrf_token" value="<?=
    htmlspecialchars($_SESSION["csrf_token"]) ?>">;
    <input type="password" id="password" name="password">
    <button type="submit">Change</button>
</form>
```

Listing 5.8: Formulář s CSRF tokenem

Po odeslání formuláře musíme tedy ještě ověřit, zda token, který nám dorazil ve formuláři, je správný. Pro dané ověření můžeme využít jednoduchou funkci, co ověří přítomnost obou tokenů a rovněž ověří zda mají stejné hodnoty.

```php
function validate_csrf_token() {
    // Check if CSRF token exists in both POST data and session
    if (isset($_POST["csrf_token"]) && isset($_SESSION["
    csrf_token"])) {
        // Validate token
        if ($_POST["csrf_token"] === $_SESSION["csrf_token"]) {
            return true; // Token is valid
        } else {
            return false; // Token is invalid
        }
    } else {
        return false; // Token does not exist
    }
}
```

Listing 5.9: Ověření správnosti CSRF tokenů

Jak již bylo řečeno, obě tyto metody mohou útočníci obejít, ale jejich kombinace zmenšuje pravděpodobnost úspěšně provedeného útoku.

5.2 Cryptographic Failures

Zranitelnosti z této kategorie nelze prakticky testovat pomocí automatizovaných skriptů. Nalezení tedy spočívá v kombinaci se špatným návrhem aplikace. Zde tedy implementace ve skriptu bude chybět. Přesto si ukážeme, jak poznáme, zda se hesla hashují, a jak je hashovat a následně i porovnávat.

5.2.1 Hashovaní hesel

Implementaci v aplikaci

Pro nehashování hesel nemusíme dělat nic. Jak je vidět v části kódu níže, stačí nám pouze uživatelovo heslo v původní podobě uložit do databáze.

```php
// Establishing a~database connection
$dbh = getDbConnection();
// Inserting user data into the database
$query = "INSERT INTO user (email, admin, password) VALUES ('" .
    $email . "',$admin,'" . $password . "')";
return $dbh->exec($query);
```

Listing 5.10: Kód pro uložení nehashovaného hesla

Zde vidíme, že heslo není nijak upraveno. V databázi tedy bude heslo uloženo stejně jako ho uživatel zadal do formuláře. Ověřování správnosti hesla

uživatele tedy může být provedeno pouhým porovnáním řetězců, jak lze vidět v následující ukázce.

```php
$dbh = getDbConnection();
// Query the database to check if the user exists with the
    provided email and password
$stmt = $dbh->query("SELECT * from user where email='" . $_POST[
    "email"] . "' AND password='".$_POST["password"]."'");
$result = $stmt->fetch();
if ($result !== false) {
// Passwords are same
} else {
// Passwords are not same
}
```

Listing 5.11: Kód pro kontrolu nehashovaného hesla

Ukázka možné opravy zranitelnosti

Pro odebrání této zranitelnosti z aplikace, nám stačí víceméně dvě funkce. A to tyto: `password_hash()`, která slouží k zahashování hesla a druhá `password_verify()`, pomocí které dokážeme ověřit, zda zadané heslo sedí s vygenerovaným hashem. Výhodou je, že nám funkce `password_hash()` již automaticky přidává do hesla *sůl* a využívá algoritmy, které jsou bezpečné pro hashování hesel. V našem případě by implementace vypadala následovně:

```php
// Establishing a database connection
$dbh = getDbConnection();
// Inserting user data into the database
$query = "INSERT INTO user (email, admin, password) VALUES ('" .
     $email . "',$admin,'" . password_hash($password) . "')";
return $dbh->exec($query);
```

Listing 5.12: Kód pro uložení zahashovaného hesla

Pro ověření správnosti hesla musíme původní kód pozměnit a to následovně:

```php
$dbh = getDbConnection();
// Query the database to check if the user exists with the
    provided email and password
$stmt = $dbh->prepare("SELECT password FROM user WHERE email = :
    email");
$stmt->bindParam(":email", $_POST["email"]);
$stmt->execute();
$result = $stmt->fetch();
$hashedPassword = $result["password"];
$inputPassword = $_POST["password"];
if (password_verify($inputPassword, $hashedPassword)) {
    // Passwords are same
} else {
    // Passwords are not same
}
```

Listing 5.13: Kód pro porovnání zahashovaného hesla

Zde jsem opravil i SQL injection, která byla přítomna u pole email, protože v sekci `Ukázka opravy` by neměla být žádná zranitelnost.

5.3 Injection

Zranitelnosti v této kategorii spočívají v nesprávném zpracování uživatelského vstupu. Jejich oprava spočívá v *escapování* nebezpečných znaků ve vstupu. Tudíž jejich implementace neboli přítomnost je závislá na absenci kontroly uživatelského vstupu.

5.3.1 SQL

Zranitelnost, jak již bylo řečeno, je přítomna v polích, kde se očekává uživatelský vstup. Tímto vstupem tedy může být například URL nebo vyhledávací pole. Ve vytvořené webové aplikaci je zranitelnost přítomná na více místech. To jednak v URL, jednak ve formulářovém poli, kde je nejlépe vidět nebezpečí této zranitelnosti.

Implementaci v aplikaci

Jak již bylo napsáno, tato zranitelnost je přítomná na více místech ve webové aplikaci, ale pro ukázku její opravy a implementace jsem si vybral místo, které je pro útočníka nejcennější. Tím je přihlášení jako administrátorský uživatel bez vědomosti, jak emailu tak hesla. Přihlášení do aplikace na straně serveru vypadá následovně:

```
1  $stmt = $dbh->query("SELECT * from user where email='" . $_POST[
       "email"] . "' AND password='".$_POST["password"]."'");
2  $result = $stmt->fetch();
3  if ($result !== false) {
4      // Set session variables upon successful login
5      $_SESSION["login"] = true;
6      $_SESSION["email"] = $_POST["email"];
7      $_SESSION["admin"] = $result["admin"] == 1;
8      // Redirect to the welcome page
9      header("Location: welcome");
10 } else {
11     // Set error flag if login credentials are incorrect
12     $error = 1;
13 }
```

Listing 5.14: Přihlášení s SQL Injection

Zde je vidět že aplikace přihlásí uživatele, pokud zadá existující dvojici email a heslo. Útočník pomocí tohoto vstupu `1' or '1' = '1`, který napíše do obou polí, ho přihlásí. SQL navíc funguje tak, že vrátí první výsledek, který splní podmínku. V tomto případě to útočníka přihlásí jako administrátora, protože většinou uživatel, který je první vytvořen v systému, je administrátor. Tato kombinace dělá z SQL injection jedním z nejnebezpečnějších útoků přítomných na webu.

Nalezení pomocí skriptu

Funkce ve skriptu implementuje ve skutečnosti popis, který je v předchozím odstavci popsán jako útok. Využívá k tomu pouze HTTP požadavky. Jedním požadavkem získá session:`PHPSESSID` a poté se pomocí druhého požadavku, tentokrát POST na adresu `/login`, pokusí přihlásit s tímto uživatelským emailem a heslem: `1' or '1' = '1`. Pokud se v odpovědi nachází řetězec: *successful-login*, pak víme, že skript se úspěšně přihlásil a že SQL Injection je v aplikaci přítomná.

```
@staticmethod
    def try_sqlinjection(url):
        try:
            response = requests.get("http://" + url)
            if response.status_code == 200:
                sessid = response.cookies.get("PHPSESSID")
            else:
                return False
        except requests.exceptions.RequestException:
            return False
        headers = { some data}
        data = {
            "email": "1' or '1' = '1",
            "password": "1' or '1' = '1"
        }
        response = requests.post("http://" + url + "/login",
    headers=headers, data=data)
        if "successful-login" in response.text:
            return True
        else:
            return False
```

Listing 5.15: Funkce pro otestování přítomnosti SQL Injection

Ukázka možné opravy zranitelnosti

Mitigace této zranitelnosti je přitom velice jednoduchá. Místo spojování řetězců v případě PHP pomocí teček jako zde:

```
$stmt = $dbh->query("SELECT * from user where email='" . $_POST[
    "email"] . "' AND password='".$_POST["password"])
```

```
$stmt->execute();
$result = $stmt->fetch();
```

Listing 5.16: Ukázka dotazu do databáze náchylného k SQL Injection

Musíme využít takzvané *bindování* parametrů. To je součástí použití prepared statements. Díky nim nepotřebujeme použít žádnou funkci, neboť slouží jako dostatečná ochrana proti SQL injection. V PHP můžeme využít funkci `bindParam()`, která slouží přesně k těmto případům. Níže je ukázka, jak kód výš můžeme přepsat do bezpečné podoby:

```
$stmt = $dbh->prepare("SELECT * FROM user WHERE email = :email
    AND password = :password");
$stmt->bindParam(":email", $_POST["email"]);
$stmt->bindParam(":password", $_POST["password"]);
$stmt->execute();
$result = $stmt->fetch();
```

Listing 5.17: Ukázka dotazu do databáze zabezpečeného proti SQL Injection

Tento kód je v tomto upravení bezpečný vůči SQL Injection a to jak zmíněného tak i obecně jakéhokoliv útoku SQL Injection.

5.3.2 XSS

Do aplikace jsem se rozhodl vložit *Stored* XSS, která je, jak jsme si popsali v kapitole o XSS, nebezpečnější. Stejně jako u SQL Injection i u XSS závisí na důvěře uživateli, že do vstupu nevloží pro aplikaci nebezpečný kód. Tím si však nemůžeme být jisti, a tak musíme všechna místa, kam uživatel může vložit vstup, který aplikace zpracovává, náležitě ošetřit.

Implementaci v aplikaci

V aplikaci je nezabezpečených míst více, ale rozhodl jsem se ukázat jediné místo, kde je přítomna *Stored* XSS. Toto místo se nachází v článku a je to formulář pro komentář. Komentář se ukládá stejně, jako ho uživatel zadal. Není tedy nijak *sanitizován*, když se podíváme na kód.

```
$dbh->exec("INSERT INTO comment (text, user_id, article_id)
    VALUES ('".$_POST["text"]."', '".$_POST["user"]."', '".
    $_POST["article"]."')");
$dbh = null;
```

Listing 5.18: Kód náchylný k XSS

Uživatelský vstup tedy není nijak upraven a tedy pokud útočník napíše do komentáře například tento řetězec:`<script>alert(1)</script>`, pak si ověří

přítomnost XSS. Toto je typický řetězec, jakým útočník hledá XSS. V našem případě se tato část kódu, kterou útočník napsal, zobrazí každému, kdo si článek, ke kterému byl komentář napsán, zobrazí. Každý uživatel, který si zobrazí tento článek, je tak potencionální obětí. Tímto způsobem se oběti spustí jakýkoliv skript útočník napsal. Může to být například krádež cookies, díky kterým je například oběť přihlášena. Komentář se na obrazovku uživatele v aplikaci vypisuje těmito řádky:

```
<article>
    <img src="https://ui-avatars.com/api/?name=<?=
    getUserEmailById($comment["user_id"])?>"/>
    <div>
        <p><?= $comment["text"] ?></p>
        <p><?= $comment["creation_date"] ?></p>
    </div>
</article>
```

Listing 5.19: Výpis komentáře

Nalezení pomocí skriptu

Hledání zranitelnosti ve skriptu je na stejném místě jak bylo ukázáno výše, tedy v komentářích. Zde jsem musel vyřešit problém ohledně toho, jak poznat, zda komentář už nebyl vytvořen v minulém průchodu skriptu a tak by nedokázal zkontrolovat aktuální stav. Proto je zde v této funkci parametr `xss_comment`, který rozhoduje, zda se bude tvořit nový komentář či nikoliv. Tato funkce slouží ke tvorbě komentáře, který zkouší vložit nebezpečný kód do stránky.

```
def try_xss(url, xss_comment, sess_id, user_id):
    if not xss_comment:
        print("Not creating new xss comment")
        return False
    headers = {
        some data
    }
    data = {
        "text": "<script id="findme">alert("XSS works!")</script
    >",
        "user": user_id,
        "article": 1
    }
    response = requests.post("http://" + url + "/article?id=1",
    headers=headers, data=data)
    return response
```

Listing 5.20: Funkce na tvorbu komentáře pro kontrolu XSS

Poté v druhé funkci níže, kde je funkce výše volána, se ověřuje, zda aplikace obsahuje zranitelnost XSS. To se ověří tak, že se vezme odpověď, kterou vrátí dotaz na článek, ke kterému byl vytvořen nebezpečný komentář. Poté se

vytvoří *DOM* z odpovědi, ve kterém se pokusí skript najít prvek `<script>` s id *findme*, které jsme schválně zvolili tak, aby pravděpodobnost, že takový prvek je již na stránce, byla minimální.

```
self.crawler.try_xss(self.url, self.crawler.xss_comment, self.
    crawler.sess_id, self.crawler.user_id)
response = requests.get("http://" + self.url + "/article?id=1")
tree = create_html_tree(response.text)
tag = tree.find("script", id="findme")
```

Listing 5.21: Ukázka použití funkce na tvorbu nebezpečného komentáře

Ukázka možné opravy zranitelnosti

Stejně jako u jiných zranitelností zde představím více možných řešení jak opravit zranitelnost, ale obdobně jako u jiných, kombinace těchto mitigací zmenšuje pravděpodobnost úspěšného útoku. Základem opravy této zranitelnosti je filtrovat uživatelský vstup jak na klientské části tak i na serverové straně. Na serverové straně je ochrana důležitá. Neboť ochrana na klientské části je nedostatečná z důvodu, že se dá vypnout, když se v prohlížeči vypne javascript. A i proto se obrana proti XSS dělá kombinací funkcí na úpravu vstupu na serverové straně a také pomocí javascriptu na straně klienta. Zde je ukázka, jak vypadá zabezpečení na straně serveru. Je rovněž potřeba zmínit, že zpracování nebezpečných vstupů se objevuje jednak při výpisu na stránku, jednak při ukládání do databáze . Pro první popisovanou situaci se používá se funkce: `htmlspecialchars()`. Tato funkce převede veškeré speciální HTML znaky do jejich HTML kódování. To znamená, že při výpisu v prohlížeči nebudou brány jako standartní HTML, a tak se nepřevedou na případné HTML elementy na stránce. Pro ukládání do databáze máme funkce specificky pro tyto účely. Jednou z nich je funkce `mysqli_real_escape_string()`, která se používá, pokud používáme knihovnu `mysqli` pro komunikaci s databází. Pokud však používáme knihovnu `PDO`, tímto ekvivalentem je funkce `PDO::quote`. Tyto dvě funkce odstraní nebezpečné znaky tak, aby řetězec při ukládání do databáze nebyl nebezpečný. V našem příkladu by oprava mohla vypadat takto:

```
$dbh->exec("INSERT INTO comment (text, user_id, article_id)
    VALUES ('".PDO::quote($_POST["text"])."', ".PDO::quote(
    $_POST["user"])."', '".PDO::quote($_POST["article"])."')');
$dbh = null;
```

Listing 5.22: Ukázka serverové ochrany proti XSS

Jak bylo řečeno ochrana se dělá i na klientské straně aplikace. To se znovu může dělat vícero způsoby, mezi nimiž jsou například sanitizace výstupu, nebo enkódování. Právě ukázku kódu, který může být použit pro sanitizaci nabízím níže:

```
function sanitizeInput(input) {
    return input.replace(/<[^>]*>?/gm, "");
}
```

Listing 5.23: Ukázka klientské ochrany proti XSS

Tento kód ze vstupu odstraní nebezpečné znaky, které by mohly způsobit XSS. Těmito znaky můžou býti: $<$, $>$ a jiné. Chceme se hlavně zbavit možnosti, aby útočník mohl do stránky vložit skript. Od toho poslední S v názvu XSS.

5.4 Insecure Design

Do této kapitoly spadají veškeré chyby, které vývojáři aplikace při vývoji ignorují. Níže ukážu pouze dvě různé zranitelnosti, které jsou v aplikaci přítomné.

5.4.1 Nezabezpečení vstupů

V kapitolách o XSS a SQL Injection bylo předvedeno, jak je chybějící zabezpečení vstupů nebezpečné. V této kapitole jde o nesmyslné hodnoty ve formulářových polích.

Implementaci v aplikaci

Tato zranitelnost je v aplikaci přítomna ve formuláři pro potvrzení administrátorských práv. Pokud uživatel v registračním formuláři zvolil možnost administrátor. Je přesměrován na další formulář, kde je po něm vyžadován číselný kód, pro ověření zda může být administrátor. Zde nebezpečný design aplikace spočívá v absenci kontroly jakékoliv automatizace. A tedy útočník může *tipovat* správnou hodnotu, díky které získá administrátorská práva. To může zautomatizovat, takže pomocí útoku hrubou silou může prolomit vstup do administrátorské sekce díky nepřítomnosti `CAPTCHA` či jiné obdobné technologie.

```
<form id="registration-form" method="POST" action="">
    <label for="code">Code:</label>
    <input type="number" id="code" name="code">
    <input class="rounded-input submit-button register-button"
    type="submit" value="Confirm">
</form>
```

Listing 5.24: Formulář pro ověření administrátorských práv

Nalezení pomocí skriptu

Zde se pokusíme pomocí skriptu získat přístup k administrátorské sekci bez toho, aniž bychom znali kód. Zabezpečení by tedy mělo být natolik dostatečné, abychom ho nedokázali v uspokojivém čase prolomit. Skript tedy vezme množinu velkých, malých písmen a číslic a pokouší se prolomit pomocí posílání HTTP požadavků, kde se mění parametr *code*, který se porovnává s kódem, který byl napevno nastaven v kódu PHP aplikace. Níže je kód, který ukazuje jak může probíhat útok hrubou silou na tento formulář, pokud aplikace nemá žádnou obranu.

```
1         characters = string.digits + string.ascii_letters
2         for length in range(1, 7):  # Generate combinations for
    lengths 1 to 6
3             for combination in itertools.product(characters,
    repeat=length):
4                 code = ''.join(combination)
5                 data = {'code': code}
6                 response = requests.post(f'http://{url}:{
    CONST_PORT}/admin_rights', headers=headers, data=data)
7                 if 'Congratulations, you are registered' in
    response.text:
8                     return True
9         return False
```

Listing 5.25: Brute force parametru code v adminské sekci

Ukázka možné opravy zranitelnosti

Jak již bylo výše uvedeno, skript zkouší hesla pouze do 6 znaků. Pokud jsme se již tedy rozhodli, že chceme administrátorské uživatele tvořit touto cestou, pak musíme zajistit, aby byl kód co nejhůře odhadnutelný. Můžeme tedy zvolit, například náhodně vygenerovaný řetězec délky aspoň 32 znaků. Můžeme také požadavky zpožďovat proti útoku hrubou silou či můžeme dokonce požadavky z konkrétní IP adresy zablokovat. Nejlepším řešením je se tomuto stylu registrace administrátorského pracovníka naprosto vyhnout a tvorbu nových administrátorských uživatelů povolit pouze existujícím administrátorům s tím, že alespoň jeden administrátor bude vždy existovat.

5.5 Security Misconfiguration

Zranitelnosti v této kapitole spočívají, jak název napovídá, v nesprávné konfiguraci. V aplikaci se konkrétně jedná o detailní chybové hlášky a directory traversal. Existuje jistě mnoho různých špatných konfigurací, ale zde jsem se

inspiroval možnými ukázkami, které byly poskytnuty dokumentem OWASP 2021.

5.5.1 Detailní chybové hlášky

V aplikaci je zapnut ladící mód a tedy veškeré hlášky z PHP jsou velice detailní a prozrazují veškeré informace o tom, kde se chyba stala. Zde jsem se ale rozhodl popsat vypisování chybových hlášek z databáze na obrazovku uživateli.

Implementaci v aplikaci

V aplikaci je chyba v dotazu do databáze řešena tak, že se vypíše uživateli na obrazovku.

```
} catch (PDOException $e) {
        print "Error!: " . $e->getMessage() . "<br/>";
        die();
}
```

Listing 5.26: Zobrazení databázových chybových hlášek

To umožňuje útočníkovi prozkoumat strukturu databáze pomocí špatných dotazů do databáze. Například při dotazu na článek s tímto id v URL: `1 UNION select text FROM article` nám aplikace prozradí touto hláškou:

```
Error!: SQLSTATE[21000]: Cardinality violation: 1222 The used
   SELECT statements have a different number of columns
```

Listing 5.27: Chybová hláška SQL

Aplikace nám tedy prozradila, že máme přidat sloupce, aby náš SQL Injection útok byl úspěšný. Tato zranitelnost, jak jsme si ukázali, není sama o sobě nebezpečná, ale útočníkovi prozradí informace, které může zneužít k útoku na jinou zranitelnost.

Nalezení pomocí skriptu

Skript se pokusí poslat dotaz na stránku s články s tímto parametrem: `1 UNION select text FROM article`, a pokud odpověď obsahuje řetězec *SQLSTATE*, pak považujeme zranitelnost za přítomnou.

```
@staticmethod
    def try_detailed_errors(url):
        url = "http://" + url + "/article"
```

```
4         response = requests.get(url, params={"id": "1 UNION
      select text FROM article"})
5         if "SQLSTATE" in response.text:
6             return True
7         return False
```

Listing 5.28: Kód pro ověření přítomnosti databázové chybové hlášky

Ukázka možné opravy zranitelnosti

Jedním z možných řešení je místo výpisu konkrétní chybové hlášky, která uživateli prozradí informace o aplikaci, vypsat pouze obecnou chybou hlášku. Možné řešení je tedy nabídnuto níže v ukázce kódu:

```
1 } catch (PDOException $e) {
2         print "Error!: Something unexpected happened";
3         die();
4 }
```

Listing 5.29: Zobrazení obecné chybové hlášky

Možnou nevýhodou může být, když klientovi nebude aplikace fungovat a jediným vodítkem bude tato chybová hláška. To je ale lépe řešitelné než aplikace obsahující zranitelnost.

5.5.2 Directory traversal

Přítomnost této zranitelnosti závisí na nastavení webového serveru a přístupů z různých souborů. Umožňuje útočníkovi pohybovat se v souborech aplikace pomocí příkazů místo názvu souboru.

Implementaci v aplikaci

V aplikaci je prohlížeč fotek, kam uživatel zadá název fotky, kterou si chce zobrazit. Název fotky zadá do formuláře aplikace, pokud fotka existuje, zobrazí ji, jinak zobrazí chybovou hlášku, že takový soubor neexistuje. Výpis fotky v kódu vypadá následovně:

```
1 <?php
2             if (isset($_GET["img"])) {
3                 $imgFile = "../public/img/".$_GET["img"];
4                 if (file_exists($imgFile)) {
5                     echo file_get_contents($imgFile);
6                 } else {
7                     echo "File not found";
```

```
                }
            }
        ?>
```

Listing 5.30: Kód pro zobrazení fotky

Problém je zde ve funkci `file_get_contents()`, která zobrazí veškerý text, který je v souboru. To znamená, že může zobrazit i kód. A druhým problémem je nulová kontrola uživatelského vstupu.

Nalezení pomocí skriptu

Funkce pro ověření přítomnosti zranitelnosti path traversal se pokusí poslat dotaz na fotku s následujícím názvem fotky: `../../src/helpers/db.php`. Neboť aplikace neprovádí žádnou kontrolu vstupu a vypisuje celý obsah souboru, tak funkce v odpovědi hledá řetězec s připojením do databáze.

```
@staticmethod
    def try_traversal(url):
        url = "http://" + url + "/image_viewer?img=..%2F..%2Fsrc
    %2Fhelpers%2Fdb.php"
        try:
            response = requests.get(url)
            if response.status_code == 200:
                pattern = r'return new PDO\("mysql:host=mysql;
    port=3306;dbname=bachelor", "redactor", "secret"\);'
                matches = re.findall(pattern, response.text)
                if matches:
                    return True
                else:
                    return False
            else:
                return False
        except Exception:
            return False
```

Listing 5.31: Kód pro ověření přítomnosti path traversal

Ukázka možné opravy zranitelnosti

Zaprvé musíme zpracovat uživatelský vstup. Toho můžeme docílit například escapováním nebezpečných znaků jako jsou například znaky . a /. Poté co zpracujeme uživatelský vstup a víme, že neobsahuje žádné nebezpečné znaky, pak stejně musíme ověřit cestu, na kterou se dotazuje.

```
$basePath = BASE_DIRECTORY;
$filePath = $basePath . $userInput;

if (strpos(realpath($filePath), $basePath) === 0) {
```

```
5       // working with file
6   }
7   ?>
```

Listing 5.32: Kód pro kontrolu cesty

Zde do proměnné *BASE_DIRECTORY* můžeme nastavit složku *img*, a tak máme jistotu, že se uživatel k jinému obsahu než námi dodanému nedostane.

Kapitola 6

Nasazení

V kapitole se budeme věnovat třem hlavním problémům nasazení aplikace. K problémům náleží: získání aplikace, zprovoznění, užívání aplikace. Jednotlivě se jim budu věnovat v kapitole *Postup nasazení*.

Podporované prohlížeče

Zaručuji se za fungování v prohlížečích a jejich verzích uvedených v tabulce:

Webový prohlížeč	Verze
Chrome	118.0.5993.88
Firefox	121.0

Tabulka 6.1: Podporované prohlížeče a jejich verze

6.1 Dokumentace

Součástí aplikace je dokumentace obou dostupných částí. Tato dokumentace je vygenerována automaticky z komentářů v zdrojovém kódu. Celý kód je tedy doplněn komentáři, které dovysvětlují použití daných funkcí a metod.

6.1.1 Aplikace

Pro dokumentaci zdrojového kódu PHP aplikace jsem si vybral nástroj phpDocumentor pro jeho jednoduché používání a uživatelsky přívětivou vygenerovanou dokumentaci ve formě HTML stránky. Tato dokumentace je dostupná ve složce `docs`, kde pouze stačí otevřít soubor `index.html` v jakémkoliv prohlížeči. Zde jsou zdokumentovány veškeré metody a funkce, které jsou v aplikaci. Pro bližší informace stačí navštívit dokumentaci nástroje.

6.1.2 Skript

Stejně jako u aplikace i zdrojový kód skriptu je okomentován tak, aby ze zdrojového kódu mohla být vygenerována dokumentace. Zde je jako nástroj pro generování zdrojového kódu použit pdoc. Umožňuje z komentářů zdrojového kódu vygenerovat dokumentaci v HTML, abych zachoval stejný formát jako v případě dokumentace PHP aplikace. Pro skript je dokumentace dostupná ve složce `vulnerability_checker/html`. Zde poté stačí pouze otevřít soubor `index` a zobrazí se nám veškerá dokumentace ke skriptu.

6.2 Postup nasazení aplikace

Pro nasazení aplikace, která je součástí této práce se, budeme řídit návodem popsaném v této kapitole. Tento návod byl úspěšně otestován na zařízeních s operačním systémem Windows 11 a na linuxových distribucích Manjaro KDE 6.6 a Ubuntu 24.04.

1. Zdrojový kód aplikace získáme z této URL adresy ze služby gitlab pomocí aplikace git [1]. Můžeme si vybrat, zda aplikaci stáhneme jako ZIP nebo pomocí příkazů git.

[1] Nástroj pro správu zdrojových kódů

2. Pokud jsme v minulém kroku vybrali archiv ZIP, pak jej rozbalíme. V případě gitu repozitář naklonujeme.

3. Pokud nemáme docker a *docker compose*, pak jej musíme nainstalovat z těchto odkazů: docker a docker compose. Verze docker-compose pro kterou byla aplikace testována je v2.27.0.

4. Ve složce, kterou jsme stáhli, je soubor `docker-compose.yml`, který budeme spouštět pro restart či start aplikace.

5. Zde spustíme příkaz: `sudo docker compose up` pro start aplikace. Aplikace je nastavena tak, aby běžela na lokálním portu 81.

6. Pro zobrazení aplikace si vybereme jeden z doporučených prohlížečů z předešlé tabulky. V prohlížeči do url řádku zadáme adresu aplikace: `http://localhost:81`

7. Pokud aplikaci dostaneme do nefunkčního stavu, projdeme tento seznam krok po kroku znovu pro znovu nasazení aplikace.

8. V konfiguraci webového serveru `nginx.conf` si můžeme zapnout autorizaci. Přihlašovací údaje do aplikace jsou `authorized` a heslo `SemestralniProjekt2023`.

6.3 Kontrola pomocí skriptu

Pro kontrolu bezpečnosti aplikace slouží přiložený skript ve složce `vulnerability_checker`. Obsah skriptu byl popsán v minulých kapitolách a v této kapitole si popíšeme jeho použití.

6.3.1 Argumenty pro spuštění

Skript jakožto aplikace spustitelná přes příkazovou řádku požaduje po uživateli vstup. Pro skript jsem se rozhodl zvolit co nejmenší počet parametrů, aby byla zachována jednoduchost použití pro uživatele na úkor škály použití. Těmito argumenty jsou:

- `-url` je povinný argument a slouží k výběru url adresy stránky, na které nám běží aplikace.
- `-xss-comment` je volitelný argument, který určuje, zda vytvořit komentář obsahující xss. Pokud ho nepoužijeme, pak se nový komentář nevytvoří.

6.3.2 Použití

Pokud chceme spustit skript, pak nejdříve spustíme webovou aplikaci podle návodu výše. Pokud pouštíme skript poprvé, pak vyplníme i parametr `-xss-comment`, aby skript otestoval přítomnost XSS.

Pro použití skriptu potřebujeme mít na zařízení nainstalován Python verze 3.11. Po spuštění webové aplikace, jak bylo uvedeno v návodu výše, můžeme použít i skript. Bez spuštěné webové aplikace skript nebude správně fungovat.

Po spuštění webové aplikace si otevřeme příkazový řádek ve složce, ve které je aplikace. Jak bylo zmíněno v úvodu kapitoly, skript se nachází ve složce `vulnerability_checker`. V případě, že máme nainstalován python, tak musíme nainstalovat všechny potřebné balíčky pro spuštění skriptu. Ty jsou uložené v souboru `requirements.txt` a nainstalujeme je pomocí tohoto příkazu: `pip install -r /path/to/requirements.txt`. Po nainstalování všech knihoven můžeme skript poprvé spustit takto: `python vulnerability_checker/src/run_check.py -url adresa-aplikace -xss-comment`.

Po vykonání skriptu dostaneme počet zranitelností v aplikaci, které je potřeba ještě opravit. Skript nám neřekne, kde se zranitelnosti v kódu nachází, ale pouze sdělí, zda daný typ zranitelnosti je v aplikaci přítomný.

Pokud zranitelnost opravíme a znovu spustíme skript, zranitelnost se nám ukáže jako opravená. Takto pokračujeme, dokud skript nachází zranitelnosti. Pokud skript nenajde zranitelnost, pak můžeme říci, že aplikace je zabezpečená proti zranitelnostem, které byly v této práci zmíněné a v aplikaci implementované.

Kapitola 7

Závěr

Cílem práce bylo seznámit čtenáře se zabezpečením webových aplikací.

V teoretické části jsme si představili, jak lze ve webové aplikaci najít časté chyby jako uživatel a jako vývojář v kódu. Dále jsme se seznámili s aplikací, ve které v praktické části ukážeme chyby a na které si můžeme vyzkoušet opravy těchto zranitelností.

V praktické části jsme se seznámili s metodami, díky kterým útočníci ve webových aplikacích hledají a zneužívají zranitelnosti. Metody můžeme využít také na vyhledání chyb v naší aplikaci. Následně jsme si společně popsali, jak tyto chyby v kódu najít a opravit. Na aplikaci, kterou jsme si v teoretické části ukázali, můžeme vyzkoušet nabyté znalosti a ověřit, zda jsme chyby v aplikaci nalezli a opravili.

Jak bylo předvedeno v teoretické a následně i potvrzeno v praktické části mnoho chyb ve webových aplikacích lze poměrně jednoduše nalézt a opravit. Práce by se dala dále rozšířit o zbylé zranitelnosti z OWASP Top Ten, tedy 6 až 10. Popřípadě reagovat na aktualizovaný dokument.

Listings

2.1 Útok SQL Injection 9

2.2 Test přítomnosti XSS 9

2.3 Directory traversal v URL 11

5.1 Ukázkový kód v jazyce Python 25

5.2 Ukázkový kód v jazyce PHP 25

5.3 Kód pro ověření přítomnosti zranitelnosti IDOR 26

5.4 Kód pro autorizaci při IDOR 27

5.5 Funkce pro kontrolu přítomnosti CSRF 28

5.6 Formulář bez zabezpečení proti CSRF 29

5.7 Vygenerování CSRF tokenu 29

5.8 Formulář s CSRF tokenem 29

5.9 Ověření správnosti CSRF tokenů 30

5.10 Kód pro uložení nehashovaného hesla 30

5.11 Kód pro kontrolu nehashovaného hesla 31

5.12 Kód pro uložení zahashovaného hesla 31

5.13 Kód pro porovnání zahashovaného hesla 31

5.14 Přihlášení s SQL Injection 32

5.15 Funkce pro otestování přítomnosti SQL Injection 33

5.16 Ukázka dotazu do databáze náchylného k SQL Injection . . . 33

5.17 Ukázka dotazu do databáze zabezpečeného proti SQL Injection 34

5.18 Kód náchylný k XSS 34

5.19 Výpis komentáře 35

5.20 Funkce na tvorbu komentáře pro kontrolu XSS 35

5.21 Ukázka použití funkce na tvorbu nebezpečného komentáře . . 36

5.22 Ukázka serverové ochrany proti XSS 36

5.23 Ukázka klientské ochrany proti XSS 37

5.24 Formulář pro ověření administrátorských práv 37

5.25 Brute force parametru code v adminské sekci 38

5.26 Zobrazení databázových chybových hlášek 39

5.27 Chybová hláška SQL 39

5.28 Kód pro ověření přítomnosti databázové chybové hlášky . . . 39

5.29 Zobrazení obecné chybové hlášky 40

5.30 Kód pro zobrazení fotky 40

5.31 Kód pro ověření přítomnosti path traversal 41

5.32 Kód pro kontrolu cesty 41

Slovník

bindování Je bezpečnostní technika, která slouží k prevenci SQL Injection útoků. Technika zabraňuje neoprávněnému vkládání SQL kódu do dotazů, protože hodnoty parametrů jsou odděleny od samotného SQL dotazu a jsou bezpečně escapovány nebo zpracovány tak, aby nedocházelo k nebezpečným injekcím

cookie Informace, kterou si o uživateli web udržuje v prohlížeči

framework Je sada nástrojů, které pomáhají vývojáři vyvinout kvalitní aplikaci

NIST The National Institute of Standards and Technology je agentura Ministerstva obchodu Spojených států amerických, jejímž posláním je podporovat inovace a konkurenceschopnost amerického průmyslu

payload Část škodlivého kódu, která je navržena k provedení škodlivé aktivity v cílovém systému

query builder Nástroj umožňující psaní dotazů pomocí PHP kódu, což zvyšuje bezpečnost a čitelnost kódu

sanitizace Je proces, při kterém očišťujeme vstupní informaci předtím, než bude uložena či dále zpracována

XSS auditor Zastaralá funkcionalita, která byla v prohlížečích Chrome a Safari, aby zabránila XSS útokům

Literatura

1. KAUR, Daljit; KAUR, Parminder. Empirical Analysis of Web Attacks. *Procedia Computer Science*. 2016, roč. 2016, č. 78, s. 298–306. ISSN 1877-0509.
2. *Getting Started* [online]. 2023. [cit. 2023-11-25]. Dostupné z: `https://www.nist.gov/cyberframework/getting-started`.
3. SHOUFAN, Abdulhadi; DAMIANI, Ernesto. On inter-Rater reliability of information security experts. *Journal of Information Security and Applications* [online]. 2017, roč. 37, s. 101–111 [cit. 2024-01-14]. ISSN 22142126. Dostupné z DOI: `10.1016/j.jisa.2017.10.006`.
4. *About the OWASP Foundation* [online]. 2023. [cit. 2023-11-12]. Dostupné z: `https://owasp.org/about/`.
5. *Common Weakness Enumeration: CWE* [online]. c2006. [cit. 2024-02-06]. Dostupné z: `https://cwe.mitre.org/`.
6. *A01:2021 – Broken Access Control* [online]. c2021. [cit. 2024-01-13]. Dostupné z: `https://owasp.org/Top10/A01_2021-Broken_Access_Control/`.
7. *Insecure direct object references (IDOR)* [online]. c2004. [cit. 2024-01-13]. Dostupné z: `https://portswigger.net/web-security/access-control/idor`.
8. *Cross-site request forgery (CSRF)* [online]. c2004. [cit. 2024-01-13]. Dostupné z: `https://portswigger.net/web-security/csrf`.
9. *A02:2021 – Cryptographic Failures* [online]. c2021. [cit. 2024-01-13]. Dostupné z: `https://owasp.org/Top10/A02_2021-Cryptographic_Failures/`.
10. *Password Storage Cheat Sheet* [online]. 2024. [cit. 2024-05-17]. Dostupné z: `https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html`.

11. *A03:2021 – Injection* [online]. c2021. [cit. 2024-01-13]. Dostupné z: https://owasp.org/Top10/A03_2021-Injection/.
12. YAWORSKI, Peter. *Real-World Bug Hunting: A Field Guide to Web Hacking*. 2nd. USA: No Starch Press, 2019. ISBN 9781593278618.
13. KHAZAL, Iman; HUSSAIN, Mohammed. Server Side Method to Detect and Prevent Stored XSS Attack. *Iraqi Journal for Electrical and Electronic Engineering* [online]. 2021-7-17, roč. 17, č. 2, s. 58–65 [cit. 2024-01-09]. ISSN 2078-6069. Dostupné z DOI: 10.37917/ijeee.17.2.8.
14. *A04:2021 – Insecure Design* [online]. c2021. [cit. 2024-01-13]. Dostupné z: https://owasp.org/Top10/A04_2021-Insecure_Design/.
15. *A05:2021 – Security Misconfiguration* [online]. c2021. [cit. 2024-01-13]. Dostupné z: https://owasp.org/Top10/A05_2021-Security_Misconfiguration/.
16. *Path traversal* [online]. c2004. [cit. 2024-01-10]. Dostupné z: https://portswigger.net/web-security/file-path-traversal.
17. *Developer Survey 2023* [online]. 2023. [cit. 2023-11-26]. Dostupné z: https://survey.stackoverflow.co/2023/#most-popular-technologies-webframe-prof.
18. *Symfony, High Performance PHP Framework for Web* [online]. c2005. [cit. 2024-02-04]. Dostupné z: https://symfony.com/.
19. *Sandbox* [online]. San Francisco (CA): Wikimedia Foundation, 2001- [cit. 2024-05-20]. Dostupné z: https://cs.wikipedia.org/wiki/Sandbox.
20. *W3Techs - World Wide Web Technology Surveys* [online]. 2009. [cit. 2024-01-11]. Dostupné z: https://w3techs.com/.
21. *Stack Overflow Developer Survey 2023* [online]. 2023. [cit. 2024-01-12]. Dostupné z: https://survey.stackoverflow.co/2023/%5C#most-popular-technologies-language.