VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INFORMAČNÍCH SYSTÉMŮ
DEPARTMENT OF INFORMATION SYSTEMS

DETEKCE KÓDU V JAZYCE JAVASCRIPT SE ZNÁMÝMI BEZPEČNOSTNÍMI CHYBAMI

DETECTING JAVASCRIPT CODE WITH KNOWN VULNERABILITES

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE **Bc. VOJTĚCH RANDÝSEK**
AUTHOR

VEDOUCÍ PRÁCE **Ing. LIBOR POLČÁK, Ph.D.**
SUPERVISOR

BRNO 2022

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav informačních systémů (UIFS) Akademický rok 2021/2022

Zadání diplomové práce

Student: **Randýsek Vojtěch, Bc.**
Program: Informační technologie a umělá inteligence
Specializace: Informační systémy a databáze
Název: **Detekce kódu v jazyce JavaScript se známými bezpečnostními chybami**
Detecting JavaScript Code with Known Vulnerabilites
Kategorie: Web

Zadání:

1. Seznamte se s balíčkovacím systémem npm a jeho bezpečnostními problémy.
2. Nastudujte možnosti nástrojů pro statickou a dynamickou analýzu JavaScriptového kódu.
3. Navrhněte nástroj, který dokáže rozpoznat kód se známými bezpečnostními problémy. Po dohodě s vedoucím může tento nástroj spolupracovat s nástrojem Selenium, může jít o rozšíření webového prohlížeče apod.
4. Návrh implementujte.
5. Implementaci otestujte, zaměřte se také na minifikovaný a obfuskovaný kód.
6. Shrňte výsledky práce, zhodnoťte její slabé stránky a navrhněte možnosti pokračování.

Literatura:

- Tobias Lauinger, Abdelberi Chaabane, Christo Wilson. 2017. Thou Shalt Not Depend on Me: Analysing the Use of Outdated JavaScript Libraries on the Web. NDSS '17, San Diego, CA, USA.
- Marius Musch, Marius Steffens, Sebastian Roth, Ben Stock, Martin Johns. 2019. ScriptProtect: Mitigating Unsafe Third-Party JavaScript Practices. In Asia CCS '19. Association for Computing Machinery, New York, NY, USA, str. 391-402.
- Markus Zimmermann, Cristian-Alexandru Staicu, Cam Tenny, Michael Pradel. 2019. Smallworld with high risks: a study of security threats in the npm ecosystem. In Proceedings of the 28th USENIX Conference on Security Symposium. USENIX Association, USA, str. 995-1010.
- Alexandre Decan, Tom Mens, Eleni Constantinou. 2018. On the impact of security vulnerabilities in the npm package dependency network. In Proceedings of the 15th International Conference on Mining Software Repositories, str. 181-191.

Při obhajobě semestrální části projektu je požadováno:

- Body 1 až 3 zadání.

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/

Vedoucí práce: **Polčák Libor, Ing., Ph.D.**
Vedoucí ústavu: Kolář Dušan, doc. Dr. Ing.
Datum zadání: 1. listopadu 2021
Datum odevzdání: 18. května 2022
Datum schválení: 20. října 2021

Abstrakt

Práce se zabývá problematikou detekce zranitelných JavaScriptových knihoven a NPM balíčků. Na základě existujících studií shrnuje technologický základ platformy Node.js a dále se hlouběji věnuje vybraným zranitelnostem systému NPM a stávajícím ochranným prostředkům. Bylo vytvořeno rozšíření prohlížeče Chrome, které má za cíl detekovat a opravit JavaScriptový kód se známými zranitelnostmi na straně webového prohlížeče. Vytvořený nástroj byl otestován průchodem 50 000 webovými stránkami. Bylo detekováno 8 129 zranitelných skriptů. Rozšíření bylo publikováno na *Chrome Web Store* pod názvem *JS Vulnerability Detector*.

Abstract

This thesis deals with the detection of vulnerable JavaScript libraries and NPM packages. Based on existing studies, it summarizes the technological core of the Node.js platform and further focuses on selected vulnerabilities of the NPM system and existing means of protection. A Chrome browser extension able to detect and fix JavaScript code with known vulnerabilities on the web browser had been introduced. The tool was tested in a crawl of 50 000 websites. 8 129 vulnerable scripts were detected. The extension has been published to the *Chrome Web Store* as *JS Vulnerability Detector*.

Klíčová slova

JavaScript, Node.js, NPM, detekce zranitelností, klientský JavaScript, abstraktní syntaktický strom, rozšíření webového prohlížeče, National Vulnerability Database, Snyk, Chrome, Manifest V3, crawl, hash, JSON

Keywords

JavaScript, Node.js, NPM, vulnerability detection, client-side JavaScript, abstract syntax tree, browser extension, pushdown automata, National Vulnerability Database, Snyk, Chome, Manifest V3, crawl, hash, JSON

Citace

RANDÝSEK, Vojtěch. *Detekce kódu v jazyce JavaScript se známými bezpečnostními chybami*. Brno, 2022. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Libor Polčák, Ph.D.

Detekce kódu v jazyce JavaScript se známými bezpečnostními chybami

Prohlášení

Prohlašuji, že jsem tuto diplomouvou práci vypracoval samostatně pod vedením pana Ing. Libora Polčáka, Ph.D. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Vojtěch Randýsek
16. května 2022

Poděkování

Rád bych poděkoval svému vedoucímu Ing. Liboru Polčákovi, Ph.D. za odborné vedení, ochotu při konzultacích a celkovou podporu. Dále bych chtěl poděkovat za ohleduplnost a trpělivost své rodině, přátelům a kolegům.

Obsah

Kapitola 1

Úvod

S novodobým vnímáním JavaScriptu jako univerzálního programovacího jazyku se pojí výrazný růst jeho popularity mezi vývojáři. Za přerod jazyka, který byl původně určen pro přidání doplňujících funkcí k oživení webových stránek, do současné podoby může z velké části příchod platformy Node.js [16]. JavaScript dnes není jen jazyk klientských webů, ale také jazyk rostoucí množiny webových aplikací a aplikačních serverů [23].

Okolo JavaScriptu vznikl rozsáhlý systém balíčků třetích stran, které vývojářům, bez vynaložení výrazného úsilí, poskytují téměř libovolnou funkcionalitu. Node.js prostřednictvím zabudovaného systému pro správu balíčků NPM[1] poskytuje přístup k více než 1,3 milionu balíčků třetích stran. NPM je napříč všemi technologiemi největší správce balíčků na světě, roste jak zájem vývojářů, tak i jeho obsah; denně přibývá více než 800 balíčků [16] a měsíčně proběhne přes 75 miliard stažení [12].

Weboví vývojáři jsou proslulí tím, že se k použití balíčků třetích stran uchylují častěji, než vývojáři jiných platforem. Nejznámější z rozšiřujících JavaScriptových knihoven je *jQuery*, která je použita na 78,4 % webových stránek [4]. Oproti takto rozšířeným knihovnám existuje nespočet knihoven poskytujících až triviální funkcionalitu. Vývojáři často spoléhají na to, že publikovaná knihovna je řádně otestovaná a spravovaná, neuvědomují si však, že tak do svého projektu zavádí externí tranzitivní závislosti a zvětšují vektor možného útoku [16].

Od roku 2014 narostl provoz na serverech NPM o 23 500 %, což ukazuje na rostoucí zájem mezi vývojáři. Množství balíčků na NPM umožňuje vývojářům velmi rychle vyvíjet aplikace a služby s využitím již existujících funkcí a komponent. Mnoho vědeckých prací ukazuje, že použití knihoven třetích stran s sebou nese vlastní bezpečnostní rizika [16, 10, 17, 30, 29]. Bylo ukázáno, že 93,2 % kódu průměrné Node.js aplikace bylo vyvinuto třetí stranou [16]. Jedním z častých důvodů, proč se vývojáři uchylují k použití knihoven je domněnka, že jsou dobře testovány a udržovány. Navzdory tomuto názoru, jedna ze studií uvádí, že pouze 45,2 % triviálních knihoven obsahuje testy [16]. Dalším znakem NPM je vysoký počet tranzitivních závislostí. Pokud použijeme populární webový rámec[2] *Spring* v Javě, zavedeme tranzitivní závislost na 10 dalších knihovnách. Pro porovnání podobný rámec *Express.js* zavádí 47 tranzitivních závislostí [10].

Kapitola 2 se podrobněji zabývá platformou Node.js z hlediska architektury a použitých technologií. Navazující kapitola 3 se blíže zaměřuje na NPM a zkoumá jej z hlediska bezpečnosti; zmiňuje některé z častých hrozeb a zranitelností, kterým platforma Node.js a NPM

[1] NPM – Node Package Manager, dále bude používána zkratka

[2] v původním znění *framework*

denně čelí. Teoretická část je zakončena kapitolou 4, která představuje souhrn vybraných teoretických praktik a existujících nástrojů, snažících se eliminovat či zmírnit popsané hrozby. Praktická část práce začíná kapitolou 5, ve které na základě zmapovaného aktuálního stavu navrhuji vytvořit rozšíření prohlížeče Chrome, schopné detekovat a opravit známé zranitelnosti. V navazující kapitole 6 popisuji implementaci tohoto rozšíření, pomocných knihoven a proces zpracování a uložení zranitelností. Kapitolou 7, zabývající se testováním, praktická část končí. Závěr práce, kapitola 8, shrnuje celý text práce, implementaci, výsledky testování a krátce zmiňuje slabiny práce a možnosti jejího pokračování.

Kapitola 2

Node.js a související technologie

Platforma Node.js byla představena v roce 2009 a od té doby si vybudovala ve vývojářské komunitě pevnou pozici. Platforma si klade za cíl zjednodušit a unifikovat implementaci rychlých, snadno škálovatelných, webových služeb a aplikací. V jádru se jedná o *open-source* projekt uložený na serveru GitHub, na jeho vývoji se však podílí i weboví giganti jako je Yahoo. Ostatní, jako například Microsoft, IBM a PayPal [16], zase integrují Node.js do svých *cloudových* systémů. Byť je projekt na poměry IT světa stále relativně mladý a nedosáhl ještě pomyslné zralosti, dokázal se díky svému inovativnímu přístupu, použití již dříve populárních technologií a dobrou škálovatelností stát schopným konkurentem tradičních platforem jako je Apache nebo Java [23].

Node.js je postaveno na neblokující událostmi řízené architektuře[1], která vede ke snadné škálovatelnosti aplikací [16]. Novost platformy a technického řešení za ní vedla k rychlé migraci vývojářů, kteří však obecně nevěnují mnoho pozornosti možným bezpečnostním hrozbám, které nová platforma přináší. Je tak často vnímána jako lépe škálovatelná náhrada existujících platforem. Pravdou však je, že Node.js s sebou nenese všechny bezpečnostní prvky platforem, které zdánlivě nahrazuje [23]. Volba konkrétních technologií a implementační detaily použité v Node.js mají přímý dopad na bezpečnostní faktory platformy. V Node.js lze identifikovat 4 hlavní technické složky:

1. architekturu,
2. jazyk pro implementaci aplikací,
3. běhové prostředí[2],
4. standardní knihovny a systém pro modulární rozšíření.

2.1 Architektura Node.js

Následující sekce popisuje shrnující přehled základních technických prvků platformy Node.js. Charakteristikou Node.js je událostmi řízený model vykonání[3]. Ten využívá jediné vlákno, tzv. vlákno událostí[4], k vykonávání kódu aplikace. Běh v jednom vlákně znamená, že na aplikační úrovni není podporován paralelismus a všechny operace jsou vykonávány stejným

[1] v původním znění *non-blocking event-based architecture*
[2] v původním znění *runtime environmnt*
[3] v původním znění *event-based execution model*
[4] v původním znění *event thread*

vláknem. Díky tomu se nemusí platforma zabývat zamykáním sdílených zdrojů a synchronizacemi, které jsou z pohledu výkonu drahé a náchylné k chybám. Takový přístup byl ve spojitosti s JavaScriptem použit poprvé, v ostatních technologiích je však již zaběhlý. Příkladem budiž *Apache MINA*, *Eventlet*, *Twisted*, nebo *vert.x*. Běh na jednom vlákně vede ke snadnějšímu vývoji aplikací, na druhou stranu ale časově náročné výpočty a blokující I/O[5] operace vykonávané jedním vláknem vedou k blokování vykonávání dalších požadavků. V Node.js je tento problém řešen pomocí asynchronních I/O operací. Častým jevem je, že jsou jednotlivé potenciálně blokující I/O požadavky (databázové čtení a zápisy, přístup k disku) doplněny o zpětná volání. Běhové prostředí v takovém případě zpracuje požadavek na jiném vlákně, ze své interní množiny vláken, neblokujícím způsobem. Po skončení požadované operace je vyvolána funkce zpětného volání na hlavním vlákně zpracovávajícím události. V klasických aplikacích typu Apache a PHP je použita architektura více vláken, každý příchozí požadavek je zpracován ve vlastním vlákně. Z hlediska výkonosti a využití zdrojů jsou oba přístupy při efektivní implementaci srovnatelné [23].

Ač se může přístup vlastního vlákna pro každý dotaz zdát vývojářům při návrhu své aplikace přirozenější, jeho škálovatelnost není optimální a při desítkách tisíc paralelně připojených klientů přestává fungovat spolehlivě a efektivně. V těchto případech jsou správci systémů často nuceni škálovat vertikálně, kvůli časté absenci možnosti škálovat horizontálně. Vertikální škálování znamená použití lepšího HW, který dokáže obsloužit víc dotazů. Tato možnost je však konečná. Přístup zvolený v Node.js naopak pracuje se zdroji daleko efektivněji. To pak v kontextu serverových aplikací znamená vyšší odolnost proti útokům typu DOS[6] [23].

2.2 Jazyk pro implementaci aplikací

Jako nativní jazyk platformy byl zvolen JavaScript. Jde o událostmi řízený skriptovací jazyk, který byl v červenci 1997 standardizován asociací ECMA. Standardizovaná verze se jmenuje ECMAScript. Dále je vhodné zmínit, že slovo Java je v názvu pouze z marketingových důvodů, svou syntaxí patří do rodiny jazyků C/C++/Java [5].

Před vznikem Node.js byl JavaScript vnímán jako základní skriptovací jazyk pro klientské webové stránky. Díky své popularitě, obrovské adopci komunitou a soutěživosti na poli webových prohlížečů získal jazyk za dobu své existence mnoho vylepšení. Spolu s jazykem se zlepšovaly i jeho běhová prostředí zabudovaná do prohlížečů [23].

Důvodů pro použití JavaScriptu v Node.js bylo několik. Prvním z nich je popularita. Pravděpodobně každý vývojář webových stránek se s JavaScriptem v nějaké podobě setkal. Použití stejného jazyka na straně klienta i serveru vede ke snadnějšímu přechodu, vývojáři se nemusí učit nový jazyk a jsou tak rychleji produktivní. Jak již bylo uvedeno v sekci 2.1, Node.js úzce spoléhá na zpětná volání, která jsou v JavaScriptu snadno proveditelná, protože součástí jazyka je podpora pro tvorbu anonymních funkcí. Jejich vhodným použitím lze v rámci platformy Node.js dosáhnout cílů krátkým a elegantním kódem. V neposlední řadě hrála důležitou roli při výběru JavaScriptu existence zavedeného efektivního běhového prostředí [23].

Jak nastínil předchozí odstavec, Node.js obsahuje i vlastní běhové prostředí, které plní obdobnou funkci jako například JVM[7] pro platformu Java. Běhové prostředí Node.js je napsáno v C++ a vychází z JavaScriptového interpretu V8 od firmy Google, který byl

[5] In/Out – čtecí a zápisové operace
[6] Denial of service, odepření přístupu
[7] JVM – Java Virtual Machine

původně vyvinut pro prohlížeč Chrome. V8 je prokazatelně rychlý, ověřený a obsahuje pokročilé techniky jako je JIT[8] překlad a efektivní práce s pamětí. Node.js pracuje s událostmi, které zpracovává na jednom dedikovaném vlákně. Dále má vnitřní množinu vláken, díky kterým se aplikacím zdá, že jsou I/O operace neblokující [23].

2.3 Rozšiřující knihovny

U vysokoúrovňových jazyků je obvyklé, že obsahují sadu rozšiřujících knihoven. Není tedy překvapením, že Node.js obsahuje API[9] pro nízkoúrovňovou síťovou komunikaci, základní funkce HTTP serveru, operace se souborovým systémem, kompresi a další běžné úlohy. Základní funkcionalitu Node.js lze rozšířit přídavnými knihovnami třetích stran, označovaných jako balíčky. Šíření těchto knihoven probíhá jak v soukromých tak veřejných registrech. Jejich struktura je pevně daná a odpovídá formátu *CommonJS*. Balíčky jsou do projektů instalovány pomocí NPM [23].

Oproti organizované struktuře aplikace Node.js, ve které jsou přímé závislosti uvedeny v souboru `package.json`, stojí zdánlivě až chaotický systém nativních JavaScriptových knihoven na webu. Weboví vývojáři běžně spoléhají na knihovny třetích stran, jako je například *jQuery*, pro vylepšení funkcionality svých stránek. JavaScriptová knihovna je ve svém jádru textový skript, obsahující kód, který vykonává relativně dobře definovanou funkcionalitu. Takový skript má plný přístup k DOMu[10], do kterého je vložen. V JavaScriptu neexistují jmenné prostory[11] a všechny konstrukty jsou globální. To může vést ke konfliktům ve jménech mezi knihovnami, proto některé zavádí různé mechanismy jejich prevence. Aby mohla být knihovna zavedena do webové stránky, vývojář nejčastěji využije příslušnou HTML značku a předá jí referenci na externě uloženou verzi knihovny, nebo její kopii na svém serveru. Knihovny jsou často dodávány v minifikovaných verzích, ze kterých jsou odstraněny komentáře a bílé znaky. Dále jsou v minifikovaných knihovnách jména a použití všech proměnných nahrazeny za co nejkratší tvary. Cílem minifikace je zmenšit konečnou velikost souborů knihovny. Vývojáři také někdy slučují soubory jedné i více knihoven, vytvářejí vlastní sestavení[12], případně používají vlastní pokročilé nástroje pro minifikaci.

V případě, že knihovny nejsou správně udržovány, znamená závislost na nich možnou zranitelnost pro webovou stránku, která je používá. Analýzou bylo zjištěno, že až 37 % analyzovaných stránek obsahuje zranitelnou verzi některé knihovny [17].

Zranitelnost je známá, uveřejněná bezpečnostní hrozba, která ovlivňuje některé z publikovaných verzí knihovny. Tato knihovna je označována jako zranitelná a ovlivněné publikované verze jsou označovány jako zranitelné verze. Zranitelnost knihovny je považována za opravenou, jakmile je publikována verze knihovny, která již není touto zranitelností ovlivněna. Pokud je knihovna závislá na zranitelné knihovně, hovoříme o zranitelné závislosti. Zranitelná knihovna může v takovém případě vystavit své zranitelnosti i knihovnu, či aplikaci, která na ní závisí. Dle typu závislosti tak lze dělit zranitelné závislosti na přímé a nepřímé (tranzitivní) [29].

Závislosti jsou do webových stránek zaváděny často ad-hoc a tranzitivně, což může vést k zavedení různých verzí stejné knihovny v jeden moment. Navíc analýzy ukazují, že

[8] JIT – Just In Time
[9] API – Application Programming Interface
[10] DOM – Document Object Model – stromová reprezentace webové stránky
[11] v původním znění *namespaces*
[12] v původním znění *build*

knihovny zavedené nepřímo, nebo skrz reklamní sledovače[13], jsou zranitelné častěji. Z toho nepřímo vyplývá, že bezpečnost webové stránky není pouze v rukou jejího správce, ale, díky dynamické architektuře webu, i v rukou třetích stran [17].

Bylo ukázáno, že webové stránky komunikují s rostoucím počtem třetích stran. Na počátku tohoto tisíciletí kontaktovalo více než 5 třetích stran pouze 5 % z 500 nejnavštěvovanějších stránek. V roce 2016 to bylo už 40 % [19].

2.4 Sémantické verzování

Je běžné, že v souvislosti se softwarem hovoříme o jeho jednotlivých verzích. Aby od sebe bylo možné verze snadno odlišit, bývají kódově označeny. U JavaScriptu se nejčastěji setkáváme s tzv. sémantickým verzováním. Tato konvence kóduje verze trojicí čísel `major.minor.patch`, přičemž každá složka má svůj význam a pravidla pro navýšení oproti předchozí verzi. Složka `major` je navýšena, pokud nová verze obsahuje zpětně nekompatibilní změny. V případě větších změn, které jsou zpětně kompatibilní je navýšena složka `minor`. Pokud nová verze obsahuje pouze zpětně kompatibilní opravy chyb, roste část `patch` [17, 29].

[13] v původním znění *ad-tracking*

Kapitola 3

Zranitelnosti systému NPM

Podstatou této kapitoly je nastínit problematiku bezpečnosti technologií souvisejících s platformou Node.js; systému NPM a klientského JavaScriptu. Kapitola ukazuje, že NPM přebírá některé ze zranitelností známých z klientského JavaScriptu a je možným terčem útoků podobného typu.

Systém NPM je otevřený, všichni uživatelé skrze něj mohou volně šířit a sdílet kód. Použití existujícího kódu je tak záležitost jednoho příkazu v terminálu, který stáhne a nainstaluje balíček a s ním i všechny další, na kterých tranzitivně závisí. Sdílení vlastního balíčku s komunitou je obdobně jednoduché, není nijak omezeno ani kontrolováno. Otevřenost NPM umožnila jeho rychlý růst a NPM nyní obsahuje balíčky pro téměř každou myslitelnou situaci, od malých nástrojů až po komplexní rámcová řešení pro tvorbu webových serverů a uživatelských rozhraní. Otevřenost NPM je však vyvážena bezpečnostními riziky, která s sebou přináší. Na úvod připomeňme některé z incidentů z posledních let [30]:

1. Aféra zmizelého balíčku *left-pad* je v NPM komunitě velmi známá. V březnu 2016 byl jeden z autorů NPM balíčků vyzván ke změně jména svého balíčku *Kik*, který kolidoval se jménem plánovaného projektu jisté firmy. Autorovou reakcí bylo, že z NPM odešel a ve vzteku smazal všech svých 250 publikovaných balíčků. Jedním z nich byl i *left-pad*, 11 řádkový skript, na kterém však závisely tisíce projektů, včetně Node.js a *babelu*. Správci NPM přistoupili k bezprecedentnímu kroku a proti vůli autora obnovili smazaný balíček v jeho poslední verzi [30, 28].

2. *Eslint-scope* je součástí JavaScriptového linteru a v roce 2018 byl cílem útoku, při kterém do něj byl nahrán škodlivý kód. Kód kradl uživatelům linteru jejich přístupové údaje k NPM. Odhaduje se, že bylo odcizeno až 4 500 účtů [14].

3. V roce 2019 byl do balíčku *event-stream* nahrán škodlivý kód, který kradl uživatelům bitcoiny. Škodlivý kód byl odhalen po 2,5 měsících a 8 milionech stažení. Původní správce předal kontrolu nad balíčkem útočníkovi, který sliboval, že balíček bude dále spravovat. Jedná se tedy o druh sociálního inženýrství [29, 14].

4. Dále v roce 2019 bezpečnostní tým NPM odhalil škodlivou verzi balíčku *electron-native-notify*. Jeho autor původně vydal užitečnou knihovnu. Poté, co si na knihovnu vytvořila závislost aplikace *Agama Wallet*, vydal aktualizaci se škodlivým kódem. Útočník ukradl přes 13 milionů dolarů v bitcoinech [14].

3.1 Životní cyklus zranitelnosti

Zranitelnost je v kontextu software jeho slabina, která umožňuje provedení neoprávněné akce nebo získání přístupu. Tyto akce jsou většinou dále využity k prolomení systému a porušení jeho bezpečnostních politik. Hrozba je pak označení potenciální příčiny incidentu, který může nastat využitím zranitelnosti systému. Zranitelnost v kontextu NPM balíčků, za předpokladu zodpovědného odhalení, prochází několika fázemi:

1. Zavedení – okamžik, kdy byla zranitelnost zavedena do kódu. V této fázi o zranitelnosti prakticky nikdo neví.
2. Nález – poté co je zranitelnost objevena by v případě zodpovědného odhalení měla být nahlášena bezpečnostnímu týmu NPM. Tým následně nález přezkoumá a ověří jeho platnost. V této fázi o zranitelnosti ví nálezce a tým NPM.
3. Oznámení – jakmile je nález zranitelnosti potvrzen týmem NPM, jsou o zranitelnosti informování správci příslušného balíčku. V této fázi o zranitelnosti ví jen nálezce, tým NPM a správci balíčku.
4. Zveřejnění bez opravy – po oznámení mají správci balíčku 45 dní, než tým NPM zranitelnost zveřejní. Spolu se zveřejněním může tým uvést ukázkový kód, demonstrující využití dané zranitelnosti.
5. Zveřejnění s opravou – druhá, častější, možnost je, že správci balíčku zranitelnost opraví před vypršením 45 denní lhůty. V tomto případě tým NPM zveřejní zranitelnost spolu s vydáním nové verze knihovny, která zranitelnost již neobsahuje. Uživatelé knihovny musí pro zvýšení bezpečnosti co nejdříve povýšit na opravenou verzi.

Zmíněné kroky se zdají sekvenční, není to tak však vždy. Jsou časté případy, kdy je oprava vydána ještě před formální fází nálezu, nebo zveřejnění [10].

3.2 Knihovny v klientském webu

Zvoleným jazykem platformy Node.js a také jazykem většiny balíčků na NPM je JavaScript. O nedokonalostech a zranitelnostech webu v souvislosti s využitím JavaScriptu bylo napsáno mnoho prací, jejichž závěry jsou důležitým kontextem pro popis zranitelností systému NPM.

Jedním z fenoménu moderních webových stránek je vložený obsah třetí strany, typicky reklamy. Ty jsou většinou implementovány JavaScriptem, který však může do hostující stránky zavádět libovolné další závislosti bez vědomí původního autora stránky. Tyto závislosti mohou být zastaralé, zranitelné a dokonce duplicitní k již použitým knihovnám [17]. V důsledku zavedení zranitelné knihovny se stává zranitelnou i samotná stránka [22].

JavaScript nemá žádné nástroje pro kontrolování a nastavení přístupu kódu na základě zdroje. Pro porovnání v Javě lze použít `SecurityManager`, který zamezí přístupu k citlivým API. To znamená, že veškerý kód třetích stran je vykonáván s plnými přístupovými právy k celé aplikaci. V případě Node.js tak získává tento kód přístup například k souborovému systému nebo k síti [10].

Průzkum z roku 2019 [22] ukázal, že až 25 % stránek zranitelných vůči XSS[1] útoku jsou zranitelné pouze kvůli zranitelnosti v knihovně zavedené třetí stranou. Tyto nálezy odpovídají predikcím starších studií a průzkumů z roku 2014, které uvádí, že zranitelné knihovny

[1] XSS – Cross site scripting

na webu přežívají i mnoho let po odhalení a opravení jejich zranitelností. Uváděný medián stáří vydání knihovny oproti datu vydání její nejnovější verze je po zaokrouhlení 3 roky. Studie [17] uvádí, že tranzitivně zavedený kód třetích stran má větší pravděpodobnost výskytu zranitelnosti, než zbytek kódu webu. Pokud kód třetí strany není uzavřen ve vlastním `frame` získává plný přístup a kontrolu nad webovou stránkou do které je zaveden.

Stejná studie dále uvádí, že ve světě JavaScriptu neexistuje unifikovaný systém pro správu závislostí. Knihovny jsou uloženy v různých CDN[2], často je však autoři stahují a kopírují do úložiště své webové prezentace. Studie dále upozorňuje, že neexistuje centralizovaná databáze zranitelností JavaScriptových knihoven a systém pro hlášení těchto zranitelností. Weboví vývojáři knihovny často modifikují, formátují, minifikují a obfuskují, což přidává na obtížnosti ve snahách knihovny rozpoznávat. Důsledkem všeho zmíněného je, že 4,2 % analyzovaných webů za běhu zavádělo na jedné stránce několikrát stejnou verzi knihovny *jQuery* z různých zdrojů, a že 10,9 % analyzovaných webů zavádělo několik odlišných verzí této knihovny.

Výsledek tohoto průzkumu ukazuje, že 37,8 % zkoumaných webů obsahuje alespoň jednu zranitelnou verzi nějaké knihovny, 9,7 % webů má takových knihoven vice než jednu. Z pohledu jednotlivých knihoven 36,7 % *jQuery*, 40,1 % *Angularu*, 86,6 % *Handlebars* a 87,3 % *YUI* bylo zavedeno ve zranitelné verzi. Práce dále přiznává, že použitím nástrojů *Bower* nebo v případě použití Node.js NPM, lze alespoň v rámci jednoho webu zajistit snadnou správu přímých závislostí a problém zmenšit [17].

3.3 Uzamčené závislosti

Závislosti jsou v NPM uváděny v konfiguračním souboru `package.json`. Každá závislost je dána jménem knihovny na kterou odkazuje a její verzí. Verze může být zapsána dvěma způsoby – buď je uvedena specifická verze, nebo rozsah verzí. Rozsah lze chápat jako „verze knihovny novější než verze X“. Pokaždé, když NPM stahuje balíčky, vyhodnocuje znova všechny závislosti a snaží se najít odpovídající verze všech knihoven. Stává se tak, že NPM ve dvou různých časech na základě totožného `package.json` souboru nainstaluje rozdílné verze knihoven. Tomuto se předchází existencí a sdílením souboru `package-lock.json`, který všechny závislosti uzamyká na konkrétní verze. Zajišťuje tak, že NPM stáhne stejné závislosti pro všechny vývojáře, na všech strojích a ve všech časech. Velkou daní za unifikaci je ale to, že pokud je v některé ze závislostí opravena zranitelnost a je navýšena verze této knihovny, nová verze nebude nainstalována, dokud se nepřegeneruje `package-lock.json`. Jinými slovy si vývojáři musí vybrat mezi automatickými aktualizacemi závislostí a konzistencí stažených závislostí mezi systémy. U větších projektů je častější druhá možnost, tedy konzistence, což vede ke značným prodlevám v zavádění nových verzí knihoven do projektů [30].

3.4 Publikace NPM balíčků

Proces hledání a hlášení zranitelností je v systému NPM stále mladý. Publikovaný kód neprochází ručním schvalováním a téměř veškeré zranitelnosti jsou hlášeny jednotlivci, kteří je odhalili v produkčním kódu [10]. Z procesu publikace mobilních aplikací je známý schvalovací proces, kdy před publikací je aplikace posouzena odbornými zaměstnanci obchodů.

[2]CDN – Content delivery network, v překladu *Síť pro doručování obsahu*, je soustava propojených serverů, jejichž cílem je typicky zvýšit dostupnost poskytovaných dat [13].

Spolu s aplikací vstupují do procesu i metadata, např. snímky obrazovky, popis aplikace, informace o přítomnosti reklam a plateb, zda je aplikace vhodná pro děti a jiné. Mezi časté důvody zamítnutí patří nalezení chyb a nekompletní funkcionality v aplikaci, nedostatečné odůvodnění požadovaných oprávnění, lživé a nepřesné snímky obrazovky, nekvalitní uživatelské rozhraní, klamání uživatelů a v neposlední řadě neužitečnost aplikace [6].

K tomu aby vývojář mohl publikovat svůj balíček na NPM si musí nejdříve založit účet na stránkách NPM. Po splnění tohoto kroku stačí zadat příkaz `npm publish` ve složce obsahující soubor `package.json`. Uživatel, který balíček publikuje se automaticky stává jeho správcem a může tak publikovat jeho další verze. Tento uživatel dále může ostatním uživatelům přidělovat roli správce. Zajímavé je, že tento systém nevyžaduje uvedení odkazu na veřejný systém pro správu verzí (jako GitHub) se zdrojovými kódy příslušné knihovny, ani nekontroluje, zda má nahrávající uživatel oprávnění kód balíčku zveřejnit [10].

3.5 Typy útoků

Vlastnosti NPM popsané v předchozích sekcích umožňují útočníkům provést mnoho různých útoků. Následující sekce uvádí výčet několik typů útoků na NPM a nebo jeho uživatele, které se v nedávné době staly, nebo které je možné očekávat v budoucnu [10].

3.5.1 Škodlivé balíčky

Díky otevřenosti NPM je možné nahrát škodlivý kód, vydávající se za legitimní knihovnu a klamně přimět uživatele, aby tuto knihovnu používali a nebo na ni měli ze své aplikace závislost. Příkladem takového útoku je incident s balíčkem *eslint-scope* z roku 2018. V tomto případě byl škodlivý kód zaveden při instalaci pomocí *post-install* skriptu. Lze si představit sofistikovanější metody, například stažení škodlivého kódu až při běhu aplikace za specifických podmínek. Do podobné kategorie spadají balíčky, které porušují soukromí uživatelů tím, že odesílají data o svých uživatelích třetím stranám. Příkladem takových balíčků jsou *insight* nebo *analytics-node*. Tyto balíčky jsou za určitých okolností legitimní, někteří uživatele si však nemusí být sledování vědomi, ačkoliv autoři chování balíčků popisují v dokumentaci. Pokud jsou však tyto balíčky zaneseny do aplikace pomocí tranzitivních závislostí, mohou vést k neočekávanému sledování [10].

3.5.2 Zneužití zranitelných knihoven

Jak popisuje sekce 3.1, životní cyklus zranitelnosti má fáze, ve kterých je známou zranitelnost možné využít k útoku. V případě NPM je situace horší v tom, že existuje velké množství zastaralých knihoven, jejichž správci nejsou aktivní a nahlášené zranitelnosti neopravují. Tyto knihovny jsou nasazeny v produkčních systémech, ať už přímou, nebo tranzitivní závislostí. V případě opravy zranitelnosti nová verze nutně nemusí být do systémů nasazena kvůli často uzamčeným závislostem v souboru `package-lock.json` [10]. Zneužití zranitelných knihoven je přímý důsledek uzamykání závislostí, což blíže rozvedla sekce 3.3.

3.5.3 Převzetí balíčku

Správce balíčku je oprávněn přidávat další správce. Útočník tak může přesvědčit současného správce, aby mu balíček předal. Příkladem budiž incident balíčku *event-stream*, kdy útočník použil sociální inženýrství k získání práv. Útočník po jejich získání odstranil původnímu vlastníkovi roli správce a stal se tak jediným vlastníkem balíčku. Variantou tohoto útoku

je vložení škodlivého kódu do jinak nepozměněného balíčku, například skrz *pull-request*, nebo skrze zneužití systému, ve kterém je kód knihovny uložen (například krádež účtu na GitHubu). Poté, co je kód knihovny takto pozměněn, musí nic netušící majitel vydat na NPM novou verzi.

Další variantou je tzv. *typo squatting*, kdy útočník vydá škodlivou verzi balíčku pod jménem podobným původní knihovně. Kdykoliv uživatel udělá překlep, omylem nainstaluje škodlivý kód. Útok typu *typo squatting* je velice jednoduchý na provedení [10].

3.5.4 Převzetí účtu

Bezpečnost balíčku závisí na bezpečnosti účtu správce. Útočník může odcizit přístupové údaje správce a pod jeho jménem nahrát libovolný zlomyslný kód. Útok na balíček *eslint-scope* spadá i do této kategorie. Existuje celá řada možných způsobů, jak získat něčí přístupové údaje – slabé heslo, sociální inženýrství, sdílení kompromitovaných hesel mezi systémy, nebo datový únik ze strany NPM [10].

3.5.5 Spiknutí

Všechny předchozí typy útoků předpokládají jeden cíl útoku, tedy jednu kompromitovanou knihovnu. Je však možné, že bude napadeno několik knihoven zároveň, ať už neoprávněným převzetím několika účtů, nebo tím, že jeden účet spravuje několik knihoven. Možnou hrozbou s nedozírnými následky je organizované spiknutí skupiny správců balíčků s cílem působení škody, případně neoprávněné převzetí jejich účtů útočníkem [10].

Kapitola 4

Nástroje a techniky pro zvýšení bezpečnosti NPM a webových stránek

V předchozí kapitole bylo uvedeno několik slabin systému NPM a klientského webu. Identifikace zranitelnosti nutně vede k otázce jejího řešení a bezpečnostního prostředku, který by danou slabinu odstranil, omezil její hrozbu, či alespoň zmírnil její dopady. Některé uvedené nástroje jsou použitelné v praxi, jiné jsou spíše teoretického ražení. Tato kapitola představuje výčet vybraných nástrojů a technik, vědeckých i praktických.

4.1 Databáze zranitelností balíčků v NPM

Nutným nástrojem pro jakoukoliv činnost související s bezpečností je databáze zranitelností daného systému. Existence centralizované databáze zranitelností je prvním krokem na cestě k analýze zranitelností a jejich řešení. V případě NPM byl takovou databází *NPM Advisories Dataset*, která je nyní součástí databáze *Github Advisory*. Každý záznam zranitelnosti obsahuje jméno zranitelného balíčku, výčet verzí v nichž se daná zranitelnost vyskytuje a případně bezpečné verze, ve kterých již daná zranitelnost není [10].

Další databáze zranitelností použité v literatuře [29] jsou soukromá databáze *Snyk* a *National Vulnerability Database* spravovaná vládou Spojených Států.

4.2 Aktualizace závislostí a `npm audit`

Součástí programové sady pro práci s NPM je nástroj `npm audit`, který na základě souboru `package.json` porovnává závislosti projektu se záznamy v databázi *Github Advisory*. Nástroj je vhodný pro vývojáře aplikací. Dává jim snadnou možnost ověřit, zda nepoužívají některou zranitelnou verzi balíčku. Prerekvizitami pro fungování nástroje je, že daná zranitelnost musí být nahlášena v databázi zranitelností, že vývojář použije příkaz `npm audit` a že na základě jeho výsledků zranitelnou závislost v projektu aktualizuje, nebo ji odstraní [30].

Bylo ukázáno, že jedna ze tří přímých zranitelných závislostí projektu může být odstraněna pouhou aktualizací závislostí projektu v rámci stejné `major` verze (jen pomocí zpětně kompatibilních změn). Nástroj `npm audit` tak dává vývojářům možnost výrazně zvýšit bezpečnost svého projektu v porovnání se zbytkem systému [29].

4.3 Verifikace kódu a správců

Jedním z plošných a aktivních opatření v boji proti zranitelnému a škodlivému kódu je proces schvalování. Tato praktika je známá například ze světa mobilních aplikací, ve kterém musí nová verze aplikace projít procesem schvalování, než je zveřejněna v obchodech s aplikacemi. Podobně, kdykoliv je vydána nová verze balíčku, by tým NPM mohl analyzovat její kód. Balíček by mohl být uveřejněn pouze v případě, že jej tým NPM shledá bezpečným. Lze očekávat, že tento proces by nebyl zcela automatizovaný a vyžadoval by manuální kroky. Do systému NPM by mohl být zaváděn postupně, od nejpopulárnějších balíčků.

Specialitou NPM oproti mobilním aplikacím je, že NPM balíčky mohou být použity na různých platformách majících různé bezpečnostní modely. Například XSS zranitelnost je relevantní pouze pokud je balíček použit na straně klienta. Naopak *command injection pomocí funkce exec* je problém pouze na straně serveru.

Do podobné kategorie aktivních opatření spadá školení a schvalování správců balíčků. Cílem tohoto opatření je zajistit integritu správců a bezpečnost jejich účtů. Toto opatření může zahrnovat ověření identity správce, vynucení více-faktorové autentizace, či jiných bezpečnostních praktik pro přístup k účtu [30].

4.4 Systém oprávnění

Dalším z bezpečnostních opatření známých ze světa mobilních aplikací je systém oprávnění. Mobilní aplikace ve svém manifestu deklaruje, která oprávnění potřebuje ke svému správnému fungování. Uživatel aplikace jí musí danou formou příslušné oprávnění udělit, jinak aplikace nemůže přistoupit k funkcím a API, které dané oprávnění vyžadují. V případě NPM bylo odhaleno, že velké množství balíčků provádí jen velice jednoduché výpočetní operace a nepotřebuje přístup k souborovému systému, nebo síti. To představuje příležitost k zavedení systému přístupových práv, který ochrání aplikace před škodlivými aktualizacemi jejich závislostí.

Práce [14] představila rozšíření běhového prostředí Node.js o podporu oprávnění. Autoři si nastavili několik cílů, které se jim v práci podařilo splnit:

- zmenšení zasažené plochy při útoku,
- řešení nesmí vyžadovat velké změny v infrastruktuře Node.js,
- řešení nesmí znemožnit funkci existujícího kódu,
- minimální dopad na výkon platformy.

V práci [14] prezentované řešení trpí dvěma nedostatky. Instalace znamená modifikaci běhového prostředí Node.js, což z pohledu uživatele není chtěná praktika. Alternativou by bylo začlenění do Node.js samotnými autory, což ale v dohledné době nelze očekávat. Druhým, závažnějším, problémem je, že balíčky nahrané do NPM nedeklarují potřebná oprávnění, prezentovaný nástroj je tak v praxi nepoužitelný.

Podobným nástrojem je *NodeSentry*, který zavádí systém minimálních oprávnění přímo do modulů Node.js [16].

4.5 Detekce na základě kódu

Většina prací a nástrojů zabývajících se detekcí zranitelného JavaScriptového kódu, jako je např. `npm audit`, využívá tzv. meta-detekci. Ta vychází z předpokladu, že metadata knihoven (jméno, verze) a zranitelností (technické detaily, seznam zranitelných komponent) jsou vždy známé a přesné. Skutečnost je však taková, že informace o použitých knihovnách jsou často nekompletní, nekonzistentní, či zcela chybí. Existující práce ukazují, že přístupy založené na meta-detekci nejsou zcela spolehlivé a jsou často zatíženy falešně pozitivními nálezy [7].

Alternativním přístupem je analýza kódu, osvědčeně použitá v nástroji *Eclipse-Steady* pro jazyky Java a Python. Detekce na základě kódu snižuje podíl falešně pozitivních i falešně negativních výsledků detekce, protože detekuje skutečnou dosažitelnost zranitelných konstruktů, nejen referenci na zranitelnou knihovnu v metadatech analyzované aplikace.

4.5.1 *Eclipse-Steady*

Eclipse-Steady je *open-source* řešení uveřejněné pod *Apache License V2.0,* které se snaží odstranit jedno z *OWASP Top 10* bezpečnostních rizik webových aplikací – *Použití komponent se známými zranitelnostmi* [9]. V současnosti podporuje jazyky Python a Java, podpora pro Node.js a JavaScript je ve vývoji. Projekt je vyvíjen v inkubátoru firmy SAP, která je součástí *Eclipse Foundation*. Řešení je složeno ze dvou komponent. *Steady backend* je aplikace běžící v *Dockeru*, která uchovává informace o známých zranitelnostech a provedených skenech. Autoři doporučují aplikaci nainstalovat na vlastním vyhrazeném serveru. Pokud backend neběží, klient nedokáže samostatně skenovat.

Klient je rozšíření nástroje Maven a musí být spuštěn po každé změně v kódu aplikace, nebo v jejích závislostech. Jedná se tedy o mechanismus použitelný při vývoji, ne o nástroj pro koncové uživatele. Projekt je stále rozšiřován, v době psaní práce je do repositáře otevřený *pull-request* zavádějící podporu JavaScriptu [7, 8].

Jednotlivá rozšíření *Eclipse-Steady* jsou popsána v článcích a publikována. Tento typ detekce vychází z identifikace použití zranitelných konstrukcí. Zranitelná konstrukce je taková konstrukce, která byla modifikována v *commitu* opravujícím zranitelnost. Na základě správné identifikace těchto konstruktů lze určit, zda jsou tyto konstrukty dosažitelné v rámci aplikace a lze tak určit dopad zranitelnosti na aplikaci. Článek popisující rozšiřování *Eclipse-Steady* uvádí, že jde o vůbec první pokus o detekci tohoto typu nad JavaScriptovým kódem. Výsledky provedených testů ukazují, že je tento přístup použitelný. Byla provedena analýza 65 Node.js aplikací vyvíjených firmou SAP, která odhalila 5 zranitelností v 18 aplikacích.

Zásadním krokem v rozšiřování *Eclipse-Steady* je definice konstrukcí jazyka. Použité konstrukce ukazuje tabulka 4.1. Cílem je s dostatečnou granularitou popsat všechny prvky jazyka. Samotnou statickou detekci dosažitelnosti zranitelností provádí *Eclipse-Steady* na základě dvou vstupů:

1. *bill of materials* – seznam použitých konstrukcí v analyzované aplikaci,
2. databáze zranitelností.

BOM je získán analýzou zdrojových kódů aplikace a zdrojových kódů jejích závislostí stažených z NPM díky souboru `package-lock.json`. Ke zpracování zdrojových kódu byl použit nástroj *ANTLR-v4* spolu s JavaScriptovou gramatikou. Databázi zranitelností autoři budovali sami na základě *National Vulnerability Database*. Použili pouze zranitelnosti

s uvedenými opravami, které se navíc týkají 100 nejpoužívanějších NPM balíčků. K těmto zranitelnostem autoři ručně dohledali příslušné *commity* s opravami. V závěru měla použitá databáze 60 zranitelností. Nástroj dokázal zanalyzovat 42 ze 65 aplikací a dokázal odhalit 5 zranitelností [12].

Typ konstrukce	Celá cesta
Balíček (PACK)	ProjectA
Modul/Soubor (MODU)	ProjectA.utils.util_b
Funkce (FUNC)	ProjectA.utils.util_b.buy(item)
Třída (CLAS)	ProjectA.utils.util_b.Car()
Metoda (MeETH)	ProjectA.utils.util_b.Car().drive(distance, direction)
Konstruktor (CONS)	ProjectA.utils.util_b.Car().constructor(name, age)
Objekt (OBJT)	ProjectA.utils.util_b.item_list

Tabulka 4.1: Hierarchie JavaScriptových konstrukcí v rozšíření *Eclipse-Steady*

4.6 CSP Trusted Types

CSP (Content-Security-Policy) je nepovinná hlavička HTTP odpovědí, díky které mohou weboví administrátoři kontrolovat a omezovat přístup klientských aplikací ke zdrojům daného webu. Hlavním cílem je zamezení XSS útokům [2].

Jednou z možných hodnot CSP hlavičky je `Trusted Types`, která je nová a označovaná jako experimentální. Principem je, že každá nebezpečná operace nad DOMem musí být explicitně ošetřena voláním zabezpečujícího API. Konkrétně tedy každý řetězec, který je předán API manipulujícímu s DOMem, musí odpovídat bezpečnému typu (například `TrustedHtml` pro `innerHTML` atribut), jinak prohlížeč vyvolá výjimku. Bezpečné typy vznikají prací s `Policy` objektem, který provede sofistikovanou sanitaci vstupu. Nepříjemností `Trusted Types`, stejně jako dalších možných CSP, je nutný přepis webové aplikace do podoby, která by nové API používala. Ke skutečnému bezpečnostnímu významu je nutné, aby na `Trusted Types` přešel i kód třetích stran [22].

V současnosti jsou `Trusted Types` experimentálně podporovány například v prohlížečích Chrome, Edge a Opera, nejsou podporovány v prohlížečích Firefox, Internet Explorer a Safari [2]. Podpora je pro rozšíření CSP mezi širokou veřejností zásadní.

4.7 *ScriptProtect*

Minimalistickým řešením XSS z pohledu administrátora webu je nástroj *ScriptProtect*. Ten dává zodpovědným administrátorům možnost, jak zabezpečit své webové stránky proti zranitelnostem zavedeným za běhu třetí stranou (zejména provozovateli reklam). Principem je, že je do zdrojových kódů webu zaveden nový JavaScriptový soubor, který obaluje vybrané, potenciálně nebezpečné, API (např. `innerHTLM`, `document.write`) sanitačním kódem. Mírným omezením je instrumentace funkce `eval`, která může vést k neočekávaným vedlejším efektům [22]. *ScriptProtect* plní prakticky stejnou bezpečnostní funkci jako `Trusted Types`, nespoléhá však s instrumentací na prohlížeč a staví se defenzivně ke kódu třetích stran.

4.8 Ochranné prostředky na straně klienta

Předchozí sekce se zabývaly obecnými koncepty, možnými úpravami architektury Node.js a NPM, detekcí a prevencí zranitelností z pohledu programátora. Tato sekce představuje tři nástroje použitelné na straně koncového uživatele, jehož cílem je bezpečně procházet web.

4.8.1 *NoScript*

Absolutně účinným mechanismem pro odstranění zranitelností webu z pohledu koncového uživatele je rozšíření *NoScript*, které je v určité podobě dostupné ve většině moderních webových prohlížečů (v prohlížeči Tor je zabudované). Toto rozšíření blokuje spouštění JavaScriptových souborů na základě seznamu výjimek. Vzhledem k tomu, že prakticky každá webová stránka JavaScript ke své funkci využívá, stává se pro uživatele stránka v mnoha případech nepoužitelnou. Uživatel sice má možnost jednotlivé skripty přidat na seznam výjimek a přenést tak roli zajištění bezpečnosti na sebe, musel by ale procházet a analyzovat z pohledu bezpečnosti všechny skripty na všech stránkách, což je v praxi nemožné. *NoScript* je tak převážně nástrojem nadšenců pro soukromí a bezpečnost [27].

4.8.2 *JShelter*

Druhým zástupcem skupiny rozšíření webového prohlížeče, jejichž cílem je zvýšení bezpečnosti a soukromí běžných uživatelů, je *JShelter*. Rozšíření vzniklo původně na VUT FIT jako diplomová práce pana Červinky [26] pod názvem *JavaScript Restricter* a původně bylo určeno pouze pro prohlížeč Mozilla Firefox. Od té doby prošlo dalšími úpravami, zejména přidáním pokročilé ochrany proti snímání otisku prohlížeče. Rozšíření se snaží zachovávat původní funkcionalitu stránek, je aktivně vyvíjeno a rozšiřováno o bezpečnostní prvky prohlížečů a rozšíření zaměřených na bezpečnost (Brave, Chrome Zero) [27, 3, 26, 24].

4.8.3 *Retire.js*

Open-source projekt *Retire.js*[1], na který jsem narazil až v závěrečné fázi této práce, se zaměřuje na detekci známých zranitelných knihoven na webu. Na vývoji se podílelo 87 autorů a rozšíření prohlížeče Chrome[2], které je na knihovně *Retire.js* založeno, má přes 10 000 uživatelů. Mimo rozšíření pro Chrome a Firefox jsou součástí i implementace pro *Grunt*, *Gulp* a NPM. Principem je meta-detekce, stejně jako u mnoha podobných nástrojů. Nástroj se při analýze snaží detekovat známé JavaScriptové knihovny a identifikovat jejich verze, které následně porovnává s databází zranitelných verzí. Tento přístup se jeví jako dostatečně účinný, je však náchylný na falešně pozitivní výsledky. Každá známá zranitelnost obsahuje seznam `extractors`, což jsou většinou regulární výrazy, které *Retire.js* aplikuje na názvy souborů, URL, jména funkcí a jejich obsah aby získalo číslo verze dané knihovny. Nástroj neobsahuje podporu pro opravy, nebo blokování zranitelného kódu. Rozšíření pro Chrome je napsáno pod zastaralým *Manifestem V2*. Migrace na novou verzi *Manifest V3* může být pro rozšíření tohoto rozsahu komplikovaná.

[1] https://github.com/RetireJS/retire.js
[2] https://chrome.google.com/webstore/detail/retirejs/moibopkbhjceeedibkbkbchbjnkadmom

Manifesty rozšíření prohlížeče Chrome

Podle statistiky za poslední rok[3] je Chrome s obrovským náskokem nejrozšířenější webový prohlížeč. V době vzniku této práce dochází v oblasti tvorby rozšíření prohlížeče Chrome k zásadním změnám. Již v současné době musí každé rozšíření obsahovat `manifest.json`, soubor, který krom metadat obsahuje verzi standardu, který rozšíření implementuje. Existují rozšíření, která implementují *Manifest V2* (dále jen *MV2*) a *Manifest V3* (dále jen *MV3*). Rok vydání této práce – 2022 a rok 2023 obsahují zásadní milníky v přechodu z *MV2* na *MV3*:

- 17.1.2022 – Nové rozšíření implementující *MV2* nebude možné publikovat na *Chrome Web Store*. Stávající *MV2* rozšíření bude stále možné aktualizovat.
- Leden 2023 – Chrome přestane používat rozšíření v *MV2*. Stávající *MV2* rozšíření již nebude možné aktualizovat.

Podle autorů *MV3* [20] se jedná o další krok směrem k vyššímu soukromí, bezpečnosti a výkonu. Z pohledu tvůrců rozšíření jde však o značné omezení dosavadních praktik a postupů používaných v rozšířeních. Skutečnost je taková, že nový manifest značně omezuje možnosti rozšíření, zejména těch zaměřených na bezpečnost, soukromí, blokování reklam a trackerů. Proti novému manifestu se vzedmula vlna odporu a skepse z řad expertů a tvůrců. Obavy panují z toho, že Google je mnoho let největší reklamní společností a novým manifestem se snaží získat nad soukromí-střežícími rozšířeními větší převahu [21].

Nový manifest nutí všechna rozšíření podstoupit migraci, nebo zaniknout. Mezi hlavní změny patří nutnost použití *service workerů*. V *MV2* bylo možné pro práci na pozadí použít *background page*. Ta mohla fungovat ve dvou režimech – persistentním, nebo řízeném událostmi. Druhý zmíněný funguje tak, že Chrome stránku spouští při registrování události a po jejím zpracování stránku ukončí. *MV3* odstraňuje možnost volby a nutí autory použít událostmi řízenou možnost, respektive jejího nástupce, kterým je *service worker*, známý z webových aplikací. Rozdílem je to, že *service worker* nemá přístup ke standardním API, ke kterým měla dříve přístup *background page*. Tato změna je pokládána za nejzásadnější a nejbolestivější. Existuje rozsáhlý seznam[4] případů užití, které jsou díky novému manifestu nemožné, či obtížně dosažitelné, například:

- přehrávání audia na pozadí,
- parsování HTML,
- vyžádání geolokačních oprávnění,
- komunikace přes *WebRTC*,
- spuštění nového *service workeru*.

Z pohledu práce je zásadní dopad na běžné praktiky používané v rozšířeních analyzujících obsah webu a blokujících obsah. Tato rozšíření v *MV2* spoléhají na `webRequest` API. V *MV3* toto API není dostupné a jeho nová náhrada `declarativeNetRequest` API zdaleka nepokrývá původní funkcionalitu. Toto API dokáže stejně jako v *MV2* jednotlivá HTTP volání prozkoumat, ale ne modifikovat. V *MV2* může rozšíření zachytit každý jeden odchozí

[3] https://gs.statcounter.com/browser-market-share
[4] https://github.com/w3c/webextensions/issues/72

HTTP požadavek, zpracovat jej a upravit jeho výsledek. Nové API je však deklarativní, což nutí vývojáře už v čase překladu jasně specifikovat, co bude jejich rozšíření s konkrétními HTTP požadavky dělat. K dispozici mají jen velmi omezený výčet příkazů a pravidel. Pokud potřebuje rozšíření pracovat s HTTP požadavky jinak, než umožňují definovaná pravidla, nejde to [21].

Kapitola 5

Návrh nástroje pro rozpoznání zranitelného JavaScriptu

V předchozích kapitolách byla popsána platforma Node.js, s ní související technologie a některé ze zranitelností, které se v nich vyskytují. Následující sekce stručně shrnuje dříve popsané zranitelnosti související s NPM a použitím JavaScriptu ve webových stránkách. Na existující nástroje nahlíží z pohledu týmu NPM, vývojářů a běžných uživatelů a nachází prostor, který tyto nástroje příliš nepokrývají. Tuto nalezenou oblast pokrývá soustavou nástrojů prezentovaná v sekci 5.2.

5.1 Shrnutí současného stavu

Hlavními technickými složkami platformy Node.js jsou jazyk JavaScript a systém pro správu balíčků NPM. Práce v sekci 3.2 zmiňuje některé z problémů souvisejících s použitím JavaScriptu v klientských webech a ukazuje hrozby plynoucí z chování webových vývojářů. Hlavními problémy jsou používání zastaralých, zranitelných knihoven a spouštění předem neznámého kódu třetích stran prostřednictvím reklam. Uživatelé Node.js a NPM vykazují vzhledem k práci s balíčky podobné charakteristiky, jako weboví vývojáři. Zbytek kapitoly 3 se blíže věnuje zranitelnostem systému NPM.

Zbytek této sekce stručně kategorizuje dříve popsané zranitelnosti, hrozby a nástroje. Cílem je najít prázdné místo v systému zranitelností a ochranných prostředků, které by zaplnila tato práce. Sekce 4.3 a 4.4 popisují bezpečnostní problémy na globální úrovni NPM, zejména typových útoků ze sekce 3.5, a představují jejich možné řešení. Do systému NPM lze zanést preventivní mechanismy, známé z jiných platforem, které dokáží eliminovat či alespoň zmírnit většinu hrozeb. Implementace těchto procesních opatření však vyžaduje rozsáhlé kroky ze strany NPM i ze strany komunity. Přínos do této úrovně se z pohledu práce zdá nereálný.

Optikou vývojáře, snažícího se zabezpečit svůj web, nebo Node.js aplikaci, jsou viditelné jiné bezpečnostní hrozby a rizika, umožněné nedostatky v bezpečnosti na zmíněné vyšší úrovni. Vývojář se obává útoků ze sekce 3.5, jejichž společným důsledkem je, že je do výsledné aplikace zanesen škodlivý kód. Hlavní ochranou v této oblasti jsou časté aktualizace a nástroj `npm audit`. Sofistikovanější přístup slibují nástroje jako je *Eclipse-Steady*, které mimo soubor `package-lock.json` zkoumají i samotný kód aplikace a dosažitelnost zranitelných konstruktů. Podpora JavaScriptu v *Eclipse-Steady* je stále ve vývoji, lze však předpokládat, že po nasazení předčí nástroj `npm audit`. Pro odhalení zranitelné závislosti

je však potřeba, aby byla zranitelnost uveřejněna v databázích zranitelností. Na skupinu webových vývojářů, do které částečně spadají i vývojáři Node.js, cílí nástroj *ScriptProtect* a CSP hlavičky, zejména `Trusted Types`. Tyto a mnohé další nástroje se specializují na XSS zranitelnosti klientského webu. Specifikem webu je, že zranitelnosti mohou být zaneseny za běhu třetí stranou (externí provozovatel reklam, či sběru analytických dat) a je mimo možnosti správce webu takový kód opravit. Na základě průzkumu existujících nástrojů se zdá, že nástrojů, které může správce webu použít pro zvýšení bezpečnosti svého webu, je dostatek a pokrývají většinu zásadních oblastí. Úkolem vývojáře je tak udržovat závislosti aktuální, případně zabezpečit web některými ze zmíněných omezujících prostředků. Příčinu zavádění škodlivého kódu je však potřeba řešit na vyšší úrovni.

Koncový uživatel, přistupující k webovým stránkám a Node.js aplikacím pomocí webového prohlížeče je obecně vystaven zastaralým a zranitelným verzím knihoven a tudíž i škodlivému kódu. V extrémním případě se uživatel může rozhodnout vypnout JavaScript v prohlížeči rozšířením *NoScript*, čímž nejspíš omezí funkce dané webové stránky. Potenciálně uživatelsky příjemnějším řešením může být použití rozšíření *JShelter*, které oproti *NoScript*u funguje v různých úrovních zabezpečení a snaží se zachovat webovou stránku funkční. *JShelter* pracuje převážně tak, že mění odpovědi API webového prohlížeče za obecnější, čímž omezuje, co může webová stránka o uživateli zjistit [15]. Jako velmi vhodné se jeví rozšíření *Retire.js*, které je však implementováno v zastaralém *MV2* a obsahuje pouze funkce pro detekci zranitelností, ale ne pro jejich opravu či zablokování.

Jako nejvhodnější oblast, do které tato práce může nějakým způsobem skutečně přispět, považuji oblast zranitelností klientského JavaScriptu z pohledu kocnového uživatele webu. Po zpracování nástrojů v kapitole 4 se zdá, že existující nástroje kód publikovaných webových stránek pouze analyzují, ale neopravují. Jejich publikované verze jsou technicky zastaralé a spoléhají na fakt, že se jim podaří korektně identifikovat knihovnu i její verzi. Mnoho existujících detekčních nástrojů pracuje na úrovni zdrojových kódů a využívá metadata v souboru `package.json`. Tyto nástroje je netriviální přenést na stranu klienta, protože soubor `package.json` standardně není součástí výsledné sestavené webové prezentace a není tak ze strany klienta dostupný. Pominu-li meta-detekci, zbývá možnost analyzovat kód, podobně jako *Eclipse-Steady* v sekci 4.5. Autoři uvádí, že na poli detekce škodlivého kódu ze zdrojových kódů aplikace je jejich přístup nový. Na základě provedené analýzy existujících nástrojů se zdá, že podobný přístup ze strany klienta zatím nebyl vyzkoušen. Prerekvizitou pro tento druh analýzy je databáze zranitelných konstruktů, která v případě *Eclipse-Steady* byla autory vytvořena ručně na základě existujících databází zranitelností.

5.2 Návrh

Na základě poznatků získaných v teoretické části práce bylo navrženo rozšíření webového prohlížeče, jehož funkcí je detekovat a opravit JavaScriptový kód se známými bezpečnostními chybami. Součástí návrhu jsou dále podpůrné knihovny, skripty a procesy zpracování zranitelností. Jádrem navrženého nástroje je program, který dokáže na základě *commitu* vybudovat abstraktní syntaktický strom původního a v *commitu* upraveného kódu. Nástroj tento abstraktní syntaktický strom následně převede na zásobníkový automat, který tento strom přijímá. Je vhodné v této části dodat, že výsledná implementace se skutečnými konečnými automaty nepracuje, jejich funkce je nahrazena jednoduchým procházením stromem. Tato návrhová rozhodnutí jsem učinil ještě před tím, než jsem se obeznámil s rozšířením *Retire.js*. Zpětně bych na návrhu nic neměnil, stále dává smysl zranitelnosti detekovat i opravovat. Za zvážení by stálo přispění implementací oprav do *Retire.js*, čemuž ale brání

zastaralost *Manifestu* použitého v *Retire.js*. Prerekvizitou by tedy byla migrace *Retire.js* na *MV3*, což se jeví jako velké riziko pro to, aby tato práce byla dokončena v termínu.

Analyzovaný JavaScriptový kód je převeden na abstraktní syntaktický strom, který je vložen na vstup sestrojeného automatu. Pokud automat strom přijme, znamená to, že se v analyzovaném kódu vyskytuje známá zranitelnost [11]. Schéma procesu a toku dat v systému je na obrázku 5.1.

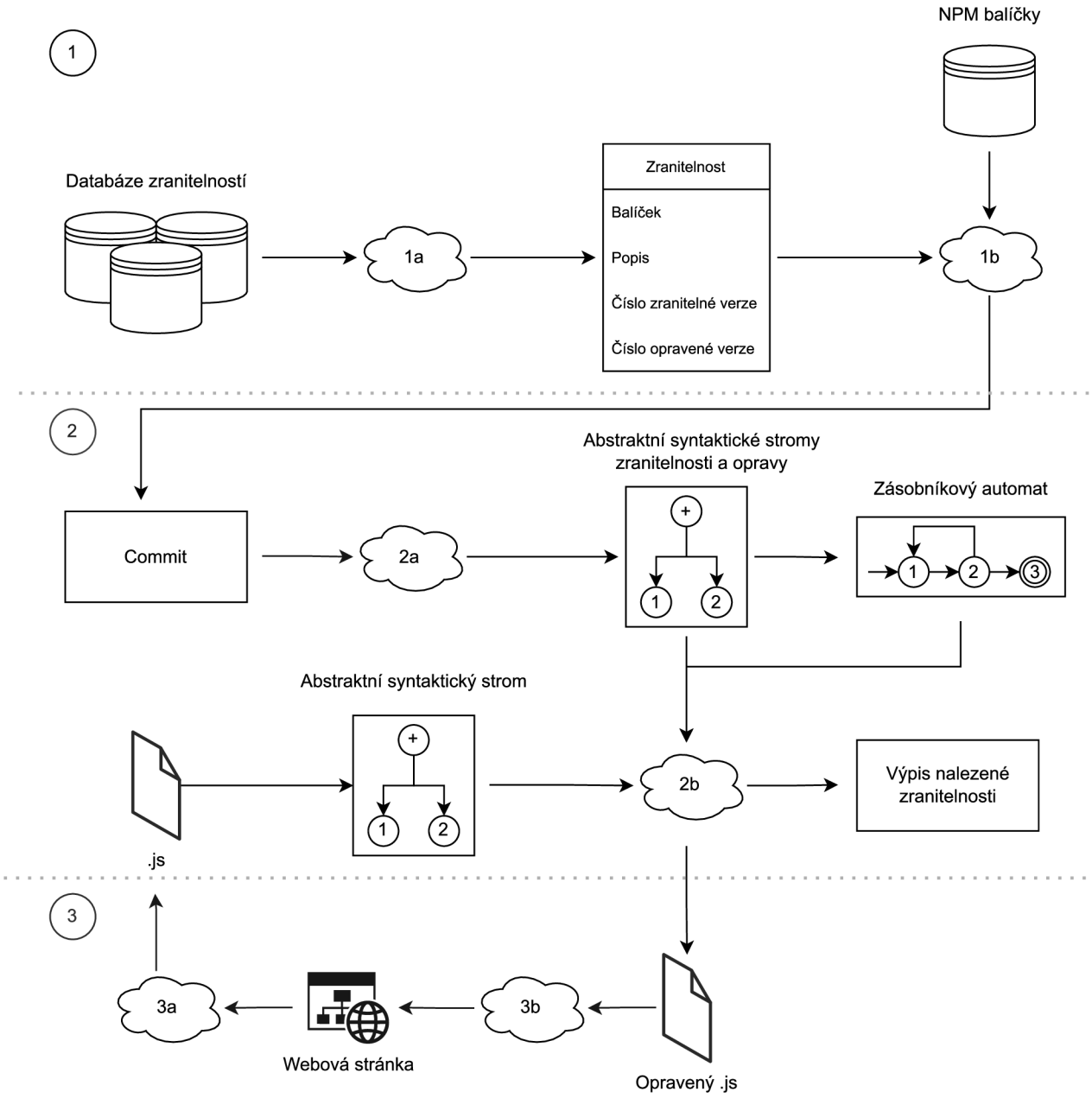
Navržená soustava je pomyslně rozdělena do tří kroků. V obrázku 5.1 jsou kroky odděleny tečkovanými horizontálními čarami a označeny čísly 1, 2 a 3. Přeneseně je po řadě lze chápat jako zpracování dat, detekci s případnou opravou a použití koncovým uživatelem.

Zpracovávanými vstupy jsou metadata balíčků v systému NPM a existující databáze zranitelností, které práce zmiňuje v sekci 4.1. Výstupem jsou metadata zranitelností rozšířená o zdrojové kódy balíčku a obsah *commitů*, které zranitelnost opravují. Prvek `1a` představuje nástroj, který z databáze zranitelností získá metadata zranitelností. Prvek `1b` páruje metadata zranitelností s metadaty balíčků v NPM a dohledává příslušné *commity*. V existujících studiích je role prvků `1a` i `1b` často vykonávána ručně.

Zásadní částí procesu je detekce zranitelného kódu. Prerekvizitou detekce je dokončení zpracování *commitů* z předchozího kroku. Prvek `2a` přetransformuje zranitelný kód i jeho opravu na abstraktní syntaktické stromy. Tyto stromy dále převede obecně známými procesy na zásobníkové automaty, které je přijímají. Proces detekce je zahájen převodem analyzovaného JavaScriptového kódu na abstraktní syntaktický strom. Prvek `2b` vloží tento strom na vstup automatů přijímajících stromy odpovídající různým zranitelnostem. Pokud některý z automatů vstup přijme, je odhalena zranitelnost. Další funkcí, kterou prvek `2b` vykonává je nahrazení zranitelného podstromu podstromem bez dané zranitelnosti. Ten je dřívějším výstupem prvku `2a`.

Práce cílí na použitelnost koncovým uživatelem ve webovém prohlížeči. Prvky `3a` a `3b` mohou být realizovány rozšířením webového prohlížeče. Jejich role je získat JavaScriptový kód z webové stránky a poskytnout ho jako vstup prvku `2b`. V případě pozitivních nálezů je komunikují uživateli a nahrazují zranitelný kód za kód s opravou.

Návrh se snaží udržet mezi pomyslnými vrstvami jasné rozhraní a zajistit tak znovupoužitelnost, rozšiřitelnost a v rámci práce odolnost proti návrhovým chybám. Prvky `2a` a `2b` na obecné rovině hledají v analyzovaném kódu dané konstrukce. Hledaný kód nemusí být nutně zranitelný, nástroj lze použít i pro jiné sofistikované nahrazování. Neváže se navíc ani na NPM, vstupem může být obecně libovolný JavaScriptový kód, je tedy možné rozšířit vrstvu zpracování dat i o zranitelnosti nativních JavaScriptových knihoven. Použití představené ve vrstvě 3 také nemusí být nutně jediné možné. Prvky `2a` a `2b` lze použít jako součást procesu postupné integrace systému, či jako samostatný nástroj pro detekci nahrazující `npm audit`. Předpokládaným výstupem práce je rozšíření webového prohlížeče Chrome, které pokryje funkci většiny aktivních prvků navržené soustavy.



Obrázek 5.1: Schéma navržené soustavy nástrojů. Tečkované horizontální čáry oddělují pomyslné ucelené kroky procesu. Šipky mezi komponentami označují směr toku dat.

Kapitola 6

Implementace

Následující sekce popisují jednotlivé implementované prvky, iterace jejich vývoje a testování. Nejzásadnější změnou oproti návrhu je odstranění porovnání abstraktních syntaktických stromů pomocí zásobníkových automatů. Místo automatu byl kvůli lepšímu výkonu použit *preorder*[1] průchod stromem společně s *hash* funkcí[2]. Implementace pracující se skutečným zásobníkovým automatem se ukázala jako zbytečně složitá. Popis prvků soustavy začíná sekcí 6.1, v návrhu 5.1 se jedná o vrstvu č. 2 a z pohledu systému jde o jeho jádro. Podpůrnou součástí systému jsou skripty použité k tvorbě a zpracování zranitelností, kterým se věnuje sekce 6.2. Tyto skripty společně s manuální prací zastávají roli první vrstvy návrhu. Poslední vrstva z návrhu, uživatelské rozhraní, je rozebrána v sekci 6.3. Implementačním výstupem je rozšíření prohlížeče Chrome schopné detekovat některé známé zranitelnosti JavaScriptových knihoven, případně je i za běhu opravit. Zvolené zranitelnosti se týkají především knihovny *jQuery*. Jedním z přínosů oproti mnoha existujícím rozšířením je úspěšná implementace *MV3*, který znatelně omezuje dřívější možnosti rozšíření prohlížeče Chrome.

6.1 Knihovna pro práci s abstraktními syntaktickými stromy

První komponentou, kterou považuji za jádro představeného nástroje, je knihovna označená jako `js-to-ast`. Interně se jedná o NPM balíček, implementačním jazykem je JavaScript. Tyto technologie jsem volil kvůli předpokládanému použití v rozšíření prohlížeče Chrome, které se standardně implementuje v JavaScriptu a ve kterém za použití pomocných nástrojů lze NPM balíčky používat.

Hlavním účelem této knihovny je zapouzdřit celou práci s abstraktním syntaktickým stromem. Následující sekce popisují knihovnu ze tří pohledů, které se částečně prolínají:

- tvorba abstraktního syntaktického stromu ze zdrojového kódu,
- formát uložení zranitelností a oprav,
- algoritmus detekce známých zranitelností v daném kódu.

[1] *Preorder* je jeden z možných průchodů stromem, kdy je nejprve zpracován kořen a poté po řadě jeho synové.

[2] Matematická funkce pro převod vstupních dat do (relativně) malého čísla. Výstup této funcke se často označuje jako otisk, v textu bude dále použita anglická varianta *hash* a od ní odvozená slova jako *hashovat*.

6.1.1 Tvorba abstraktního syntaktického stromu

Prerekvizitou pro práci se zranitelnostmi je tvorba abstraktního syntaktického stromu. Převod kódu na AST je proces, který provádí každý interpret JavaScriptu. *ESTree Specifikace*[3] sama sebe označuje jako lingua franca[4] nástrojů pracujících s JavaScriptovým kódem. Agregovaná podoba aktuální *ESTree* specifikace je uvedena v příloze B.

Překladač *Acorn*[5] implementuje specifikaci *ESTree* a má na NPM téměř dvojnásobek stažení než jeho hlavní konkurent *Esprima*[6]. Kvůli předpokládané snadné zaměnitelnosti za *Esprimu* v případě problémů jsem se rozhodl využít překladač *Acorn*. Jedinou konfigurací, kterou *Acorn* vyžadoval, bylo nastavení parametrů `ecmaVersion: "latest"` a `sourceType: "module"`. Příklady formátu výstupu překladače *Acorn* jsou na výpisech D.1 a D.2.

Práce s překladačem *Acorn* je jediná část nástroje, která oproti prvotní verzi implementace nemusela projít téměř žádným vývojem. Jedinou úpravou bylo přidání normalizace získaného AST, jak popisuje sekce 6.1.5. Volba se tak zpětně jeví jako vhodná.

6.1.2 Formát uložení zranitelností

Zásadní součástí prezentovaného řešení je definice vhodného formátu ukládání dat zranitelností a oprav. Tato část prošla největším vývojem, vyzkoušel jsem několik různých struktur a formátů dat, které podrobněji rozebírá sekce 6.1.4. Zranitelnosti jsou uloženy jako jediný objekt v souboru `generated_vulnerabilities.json`. Tento soubor lze pro zjednodušení chápat jako více-úrovňový slovník. V nejvyšší úrovni slovníku jsou jednotlivé typy uzlů specifikace *ESTree*, jako je `Program`, `Literal` a `BlockStatement`. V nižší úrovni jsou data zranitelností příslušného uzlu. Zranitelnosti jsou uloženy pod klíči, které odpovídají `SHA1` *hashům* AST zranitelného kódu. Obsahují vlastní unikátní identifikátor a identifikátor opravy. Identifikátor je vypočítán jako `SHA1` *hash* AST opravy a je nepovinný. Ne každá zranitelnost je v rámci nástroje opravitelná. Výňatek z dat zranitelností je na výpise 6.1.

```
{
    "Program": {
        "afade3b13f04486b802c73c2046549f6548a9a0b": {
            "id": "a45afe8b1ffa62c92bb1c910981a19060ec2a1a5",
            "patch": "9844f81e1408f6ecb932137d33bed7cfdcf518a3"
        },
    ...
    }
}
```

Výpis 6.1: Zkrácená ukázka zápisu zranitelností

Ve vlastním slovníku v souboru `generated_vulnerabilities_meta.json` jsou uložena metadata zranitelností. Klíče ve slovníku jsou identifikátory zranitelností. O každé zranitelnosti je známý název, popis, vážnost a URL odkazující na další informace o zranitelnosti. Výňatek z metadat zranitelností je na výpise 6.2.

[3] https://github.com/estree/estree

[4] Jazyk, který má funkci společného komunikačního prostředku mluvčích různých mateřských jazyků; česky někdy též lingva franka.

[5] https://www.npmjs.com/package/acorn

[6] https://www.npmjs.com/package/esprima

```
{
    "a45afe8b1ffa62c92bb1c910981a19060ec2a1a5": {
        "id": "a45afe8b1ffa62c92bb1c910981a19060ec2a1a5",
        "title": "Unsafe defaults in remark-html",
        "description": "The documentation of remark-html ... ",
        "reference_url": "https://github.com/advisories/GHSA-9q5w-79cv-947m",
        "severity": 1
    },
    ...
}
```

Výpis 6.2: Zkrácená ukázka metadat zranitelnosti

Poslední ze slovníků, `generated_patches.json`, obsahuje data oprav. Opravy jsou ve slovníku adresovány pomocí svých identifikátorů a obsahují celé AST opravy v normalizované podobě. Výňatek z dat oprav je na výpise 6.3.

```
{
    "9844f81e1408f6ecb932137d33bed7cfdcf518a3": {
        "type": "Program",
        "body": [
            {
                "type": "ImportDeclaration",
                "specifiers": [
                    {
                        "type": "ImportSpecifier",
                        "imported": {
                            "type": "Identifier",
                            "name": "toHtml"
                        },
                    ...
                ...
            ...
        ]
    },
    ...
}
```

Výpis 6.3: Zkrácená ukázka struktury dat oprav

6.1.3 Detekce zranitelností a tvorba opravy

Hlavním úkolem knihovny je ve vstupním kódu nalézt známé zranitelnosti. Tuto roli zastává funkce `findMatches(input, vulnerabilities, patches, meta)` ve skriptu `finder.js`. Význam parametrů je následující:

- `input` – vstupní skript v textové podobě,
- `vulnerabilities` – data zranitelností v podobě uvedené v sekci 6.1.2,
- `patches` – data oprav v podobě uvedené v sekci 6.1.2,

- `meta` – metadata zranitelností v podobě uvedené v sekci 6.1.2.

Návratovou hodnotou je objekt ve formátu na výpise 6.4.

```
{
    foundVulnerabilities: VulnerabilityMeta[],
    patchedCode: string
}
```

Výpis 6.4: Datový typ návratové hodnoty funkce `findMatches`

Ke generování výstupního kódu z opraveného AST jsem využil knihovnu *Escodegen*[7]. Algoritmus 1 je pseudokód finálního procesu detekce.

Algoritmus 1 Průběh detekce

```
1:  Get AST from Acorn
2:  Normalize AST
3:  for Every node do
4:      Get vulnerabilities for node type
5:      if Vulnerabilities for node type are not empty then
6:          Stringify node
7:          Calculate hash
8:          if Vulnerabilities contain hash then
9:              Replace node with patch
10:             Add vulnerability metadata to result list
11:         end if
12:     end if
13: end for
14: Generate patched code with Escodegen
15: Return found vulnerabilities and patched code
```

6.1.4 Optimalizce výkonu

Původní, značně naivní, implementace držela v databázi (soubory ve formátu JSON) zranitelností celé abstraktní syntaktické stromy zranitelností i jejich oprav. Fáze detekce byla provedena prakticky třemi vnořenými for-cykly. První cyklus procházel všemi uzly analyzovaného stromu, druhý cyklus procházel všemi možnými zranitelnostmi a třetí cyklus sloužil k porovnání uzlů na totožnost atributů. Při testování na malých souborech dat se neprojevovaly žádné problémy související s využitým prostorem nebo spotřebovaným výpočetním časem. Za součást práce jsem považoval ověření konceptu na větším objemu dat a případnou optimalizaci implementovaných algoritmů a struktur. Tato podsekce se hlouběji zaobírá provedeným testováním, odhalenými slabinami a procesem jejich řešení vedoucím k výsledné optimalizované verzi. Pro testování byly použity dvě sady známých zranitelností:

- sada A – „velká"“, obsahující 175 zranitelností, převážně různé verze *jQuery*,
- sada B – „malá"“, obsahující 13 zranitelností, převážně různé verze *jQuery*.

[7] https://www.npmjs.com/package/escodegen

Cílem testování bylo ověřit, jak se budou použité konstrukce a datové struktury škálovat, a sledovat, jaký dopad mají případné úpravy na dobu vykonávání. Náplní testu, provedeného pomocí jednotkového testu v nástroji *Jest*, bylo 1000krát v kódu minifikované knihovny *jQuery* `v3.4.0` identifikovat zranitelnost. Stanovil jsem 4 měřitelné parametry:

- $size_a$ – velikost dat zranitelností v případě použití sady A,
- $size_b$ – velikost dat zranitelností v případě použití sady B,
- $time_a$ – průměrná doba trvání jednoho běhu v ms za použití sady A,
- $time_b$ – průměrná doba trvání jednoho běhu v ms za použití sady B.

Výsledky měření prvotní implementace a následných optimalizací jsou uvedeny v tabulce 6.1. Je patrné, že datové i algoritmické struktury použité v původní implementaci by byly v produkčním prostředí, ve kterém předpokládám tisíce zranitelností, nepoužitelné. S vyšším počtem zranitelností značně narostla jak potřeba místa, tak i doba výpočtu. V první optimalizaci jsem se zaměřil na formát dat oprav. Všechny zranitelnosti knihovny *jQuery* mají stejnou opravu – aktualizaci na nejnovější verzi. Zavedl jsem tedy podporu pro sdílení oprav mezi zranitelnostmi a vyčlenil jsem data oprav do vlastního souboru. Dle očekávání došlo k téměř polovičnímu zmenšení velikosti, bez zásadního dopadu na rychlost vykonávání. Tato situace je však specifická pro data použitá v testu, dopad na produkční data závisí na množství zranitelností se stejnou opravou.

Předchozí optimalizace původní problém jen oddálila. Data zranitelností mají stále neúnosnou velikost. Řešením je neukládat celé abstraktní syntaktické stromy zranitelností, ale jen jejich *hashe*. Experimentoval jsem s několika knihovnami a *hashovacími* algoritmy, nakonec jsem zvolil knihovnu *Crypto.js*[8] a algoritmus `SHA1`, který na mém lokálním prostředí dosahoval nejlepších výsledků. Ostatní knihovny byly buď pomalejší, nebo po nasazení nestabilní. Očekávaný výsledek byl extrémní zmenšení velikosti dat zranitelností, za cenu režie výpočtu *hashů*.

Zmenšení velikosti se povedlo docílit, ze stovek MB byla velikost zmenšena na jednotky. Cenou však bylo podstatné zpomalení vykonávání. Test ukázal, že délka běhu přímo nesouvisí s počtem zranitelností, je téměř konstantní. Po bližším přezkoumání jsem zjistil, že zpomalení bylo způsobeno zavedením *hashování*, což je obecně výpočetně náročná operace. Podezřelou byla také serializace uzlů pomocí funkce `JSON.stringify`, která se však ukázala být dostatečně rychlá a v porovnání s výpočtem *hashe* zanedbatelná. Proces detekce v této fázi optimalizace je zjednodušeně popsán algoritmem 2.

Algoritmus 2 Průběh detekce po zavedení *hashování*

```
1: for Every node do
2:     Stringify node
3:     Calculate hash
4:     Search known vulnerabilities for the hash
5:     if Hash found then
6:         Add vulnerability to list of found vulnerabilities
7:     end if
8: end for
```

[8] https://www.npmjs.com/package/crypto-js

Poslední provedená optimalizace se zaměřuje na dobu vykonávání, která v minulém kroce značně narostla. Jako úzké hrdlo algoritmu se ukázalo *hashování*, každý uzel analyzovaného skriptu je samostatně *hashován*. Čím rozsáhlejší je analyzovaný skript, tím delší je pak doba vykonávání. V případě provedeného testu byl použit poměrně rozsáhlý skript, celá knihovna *jQuery*. Podstata další optimalizace spočívá v uvědomění si, že každá zranitelnost může nést informaci, jakého typu uzlu se týká. Uzel odpovídající typu, který nemá žádnou známou zranitelnost, není třeba *hashovat* a analyzovat. Provedená změna tedy roztřídila známé zranitelnosti do finálního formátu uvedeného v podsekci 6.1.3. Dosažené výsledky jsou opět značně zatíženy tím, že většina použitých zranitelností zasahuje uzel `Program`. V předpokládaném produkčním nasazení je tato optimalizace užitečná až do bodu, kdy bude uzel každého typu zranitelný. Z toho lze vyvodit doporučení, že ne všechny uzly nutně musí mít přiřazené zranitelnosti, naopak je vhodné zranitelnosti abstrahovat na úroveň vyšších uzlů jako jsou `Program`, `BlockStatement` nebo `Function`. Tabulka 6.1 porovnává výsledky měření jednotlivých, výše uvedených, fází optimalizace.

	Naivní	Znovupoužití oprav	*Hashování* všech	*Hashování* jen možných
$size_a$	383,88 MB	197,79 MB	1,28 MB	1,28 MB
$size_b$	23,59 MB	12,82 MB	1,24 MB	1,24 MB
$time_a$	228,46 ms/běh	253,44 ms/běh	304,81 ms/běh	61,87 ms/běh
$time_b$	38,36 ms/běh	37,71 ms/běh	289,60 ms/běh	61,55 ms/běh

Tabulka 6.1: Jednotlivé sloupce obsahují naměřená data v odpovídajících fázích optimalizace. Po řadě jsou uvedeny výsledky naivní implementace, vytknutí a znovupoužití dat oprav, *hashování* všech uzlů a finální implementace *hashující* pouze uzly se známými zranitelnostmi.

Dosažené výsledky považuji za dostatečné pro produkční nasazení. Zajímavé je, že zavedení *hashování* prakticky odstranilo závislost složitosti na počtu a velikosti zranitelností. Jediná metrika, kterou je vhodné dále sledovat je počet uzlů analyzovaného kódu. Další možná optimalizace, kterou jsem však již neimplementoval, je ukončit procházení podstromu v případě, že je v něm nalezena zranitelnost. Podmínkou této optimalizace je, že všechny známé kódy oprav neobsahují vlastní chyby. V případě, že by byla nalezena a opravena zranitelnost, není třeba dále opravený podstrom procházet. Pro některé analytické úlohy by bylo navíc možné ukončit procházení při nalezení první zranitelnosti ve skriptu, místo aktuální implementace, která v analyzovaném kódu hledá všechny zranitelnosti.

6.1.5 Minifikace

Existuje řada transformací kódu, počínaje prostým přejmenováním lokálních proměnných a konče složitými změnami ovlivňujícími tok programu a dat. Tyto transformace, které cílí na zmenšení celkové velikosti zdrojových souborů, jsou běžně označovány jako minifikace. Oproti tomu obecně složitější techniky, které mají za cíl udělat kód hůře pochopitelným a analyzovatelným, jsou označovány jako obfuskace. Oba zmíněné procesy jsou často chápány jako narušení bezpečnosti. V legitimních případech jsou používány ke zmenšení velikosti zdrojových souborů a ochraně duševního vlastnictví. Ve všech případech nicméně minifikační a obfuskační techniky způsobují vyšší složitost bezpečnostních analýz kódu.

Nedávná studie [25] ukazuje, že minifikace je široce rozšířená. Až 38 % všech klientských skriptů je minifikováno, oproti tomu pouze 1 % je obfuskováno. Minifikace většinou nemá negativní dopad na rychlost vykonávání kódu, ve spoustě případů naopak přispívá

k vyšší rychlosti. Stejná studie dále zmiňuje, že některé minifikační nástroje nezachovávají úplnou sémantiku původního kódu. Přesnou sémantiku zachovává překvapivě jen polovina zkoumaných nástrojů. Zajímavým zjištěním je, že skripty třetích stran jsou téměř dvakrát častěji transformovány než skripty první strany (55,38 % proti 30,18 %) [25].

Jedním z nástrojů, které v dokumentaci alespoň částečně popisují prováděné transformace, je *Closure Compiler* společnosti Google. Znalost možných transformací je klíčová pro analytické nástroje, které s takto upraveným kódem pracují. *Closure Compiler* pracuje v jednom ze tří režimů.

Režim `WHITESPACE_ONLY` odstraňuje z kódu komentáře, zalomení řádků, zbytečné mezery, přebytečné závorky a středníky a ostatní bílé znaky. Výstupní JavaScript je funkčně totožný s původním kódem.

Standardní úrovní minifikace *Closure Compileru* je režim `SIMPLE_OPTIMIZATIONS`, který rozšiřuje zmíněný režim `WHITESPACE_ONLY` a plně pokrývá všechny jeho transformace. Navíc však optimalizuje výrazy a funkce. Mění jména lokálních proměnných a parametrů funkcí za krátká kódová označení. Zkrácení jmen značně snižuje celkovou velikost kódu. Tento režim zasahuje pouze do lokálních proměnných a funkcí, nijak tedy neovlivňuje interakci mezi upraveným kódem a okolním JavaScriptem. Stejně jako u režimu `WHITESPACE_ONLY` je i v režimu `SIMPLE_OPTIMIZATIONS` zachovaná sémantika, pokud kód nepřistupuje k lokálním proměnným prostřednictvím textových řetězců jejich původních jmen (například ve funkci `eval()`, nebo voláním `toString()` nad funkcemi).

Režim `ADVANCED_OPTIMIZATIONS` provádí všechny transformace jako předchozí úroveň, navíc však přidává řadu podstatně agresivnějších globálních transformací k dosažení nejvyšší možné komprese. K dosažení očekávaných výsledků je potřeba, aby zpracovávaný kód splňoval sadu nutných podmínek *Closure Compileru*. V případě, že tyto podmínky nejsou splněny, nelze kód přeložený v režimu `ADVANCED_OPTIMIZATIONS` vykonat. Prováděné operace spadají do tří kategorií [1]:

- Agresivní přejmenování – překlad se `SIMPLE_OPTIMIZATIONS` přejmenuje pouze parametry funkcí a lokální proměnné. `ADVANCED_OPTIMIZATIONS` navíc přejmenuje globální proměnné, jména funkcí a vlastností objektů.
- Odstranění mrtvého kódu – pokud je kód nedosažitelný, je odstraněn. Tato funkce je velmi praktická při použití jednotlivých funkcí z rozsáhlých knihoven. Překlad dokáže z takových knihoven odstranit vše krom použité funkce.
- Globální vkládání[9] – překlad nahradí volání některých funkcí jejich tělem. Překladač provede vložení jen pokud dojde k závěru, že je tato operace bezpečná a ve výsledku ušetří místo. Překlad s `ADVANCED_OPTIMIZATIONS` dále provádí vkládání některých konstant a proměnných.

Původní návrh i implementace se minifikací zdrojových kódů okrajově zabývají a částečně s ní počítají. Návrh předpokládal, že minifikovaný kód má s původním kódem velmi podobný, až totožný, abstraktní syntaktický strom. Při implementaci se však ukázalo, že předpokládané tvrzení neplatí. V závislosti na druhu provedené minifikace se abstraktiní syntaktické stromy značně liší. Pozitivem navrženého řešení je odolnost vůči komentářům a bílým znakům, které se na výsledném stromu nijak nepodílí. Stromy jsou však náchylné na změny v pořadí jednotlivých prvků, přejmenování funkcí a proměnných, typ uvozovek literálů textových řetězců a další ze zmíněných technik minifikace. Příkladem je minifikace

[9] v původním znění *global inlining*

```
var a = "hello";
var b = 'world';
```

Výpis 6.5: Vstup bez minifikace

```
var a="hello",b="world";
```

Výpis 6.6: Minifikovaný vstup

vstupu na výpise 6.5 za použití nástroje *JavaScript Minifier*[10], jejíž výstup je na výpise 6.6. Abstraktní syntaktický strom vygenerovaný pomocí nástroje *AST explorer*[11] pro uvedené případy je po řadě na obrázku 6.1 a obrázku 6.2. Podklady pro tvorbu těchto obrázků jsou uvedeny v příloze D.

Kvůli značnému dopadu minifikace na prezentované postupy detekce jsem do knihovny přidal podporu pro implementaci normalizačních funkcí, jejichž cílem je zajistit, aby byl ekvivalentní kód převeden na stejný strom. Analyzovaný kód je ihned po převodu na abstraktní syntaktický strom normalizován. Implementovaná normalizace má spíše demonstrativní charakter, pokročilejší normalizace jsou možným rozšířením této práce. Konkrétně se jedná o tyto transformace:

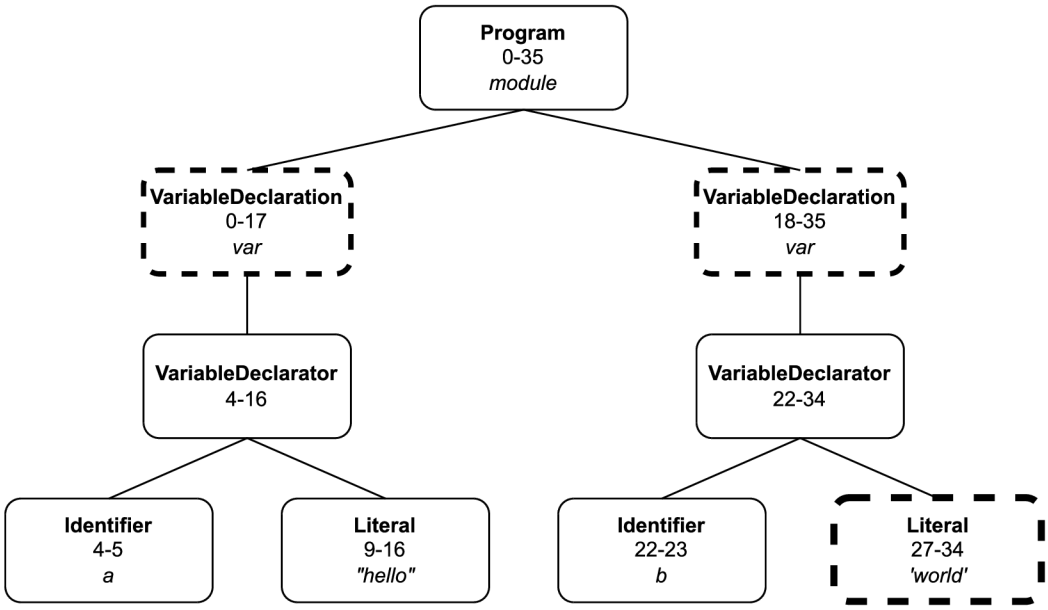
1. odstranění informací o poloze uzlů a typu vstupního kódu (atribut `sourceType` uzlu typu `Program`),
2. převod uvozovek řetězců na složené uvozovky,
3. sloučení po sobě jdoucích deklarací proměnných do jedné deklarace.

Informace o poloze jednotlivých uzlů jsou odstraněny kvůli zvýšení odolnosti. Knihovna porovnává kód na základě přesné shody vytvořených stromů. Pokud by údaje o poloze zůstaly součástí porovnání, stačilo by přidat kamkoliv do kódu libovolný bílý znak a zranitelnost by nebyla rozpoznána. Atribut `sourceType` je odstraněn, protože v předpokládaném praktickém použití knihovny jsou všechny vstupy předávány knihovně uměle prostřednictvím argumentů a informace o typu původního zdroje je ztracena. Převod uvozovek a sloučení deklarací proměnných byly vybrány jako vhodný demonstrativní případ. Na obrázku 6.3 je výsledná podoba AST vytvořeného prezentovaným nástrojem za použití normalizace. Strom je totožný se stromem na obrázku 6.2, navíc jsou z něj ale odstraněna nepotřebná metadata. Použitá data jsou uvedena v příloze D. Vstupy z výpisu 6.5 a výpisu 6.6 mají po aplikace normalizace totožné abstraktní syntaktické stromy.

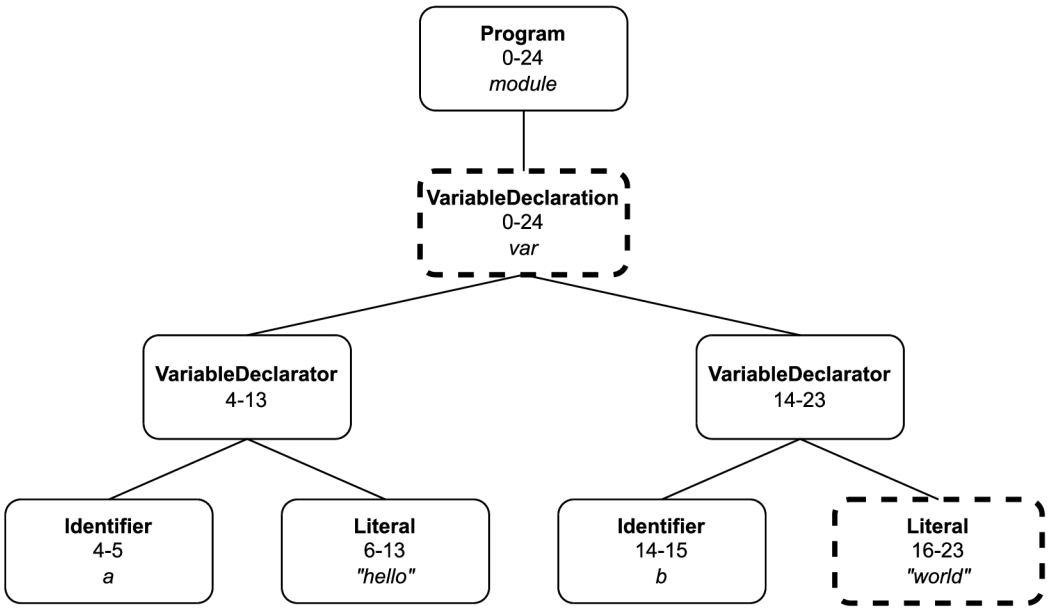
6.1.6 Obfuskace

Tato práce se obfuskovanému kódu věnuje jen okrajově. Jak zmiňuje předchozí podsekce 6.1.5, pouze 1 % skriptů je obfuskováno. Studie [25] navíc ukazuje, že naprostá většina obfuskace je prováděna pomocí *open-source* online nástroje *Daft Logic Obfuscator*[12]. Zajímavým zjištěním studie dále bylo, že obfuskovaný kód je častější v některých kategoriích webových stránek, jmenovitě například ve stránkách s obsahem pro dospělé. Dalším zjištěním je,

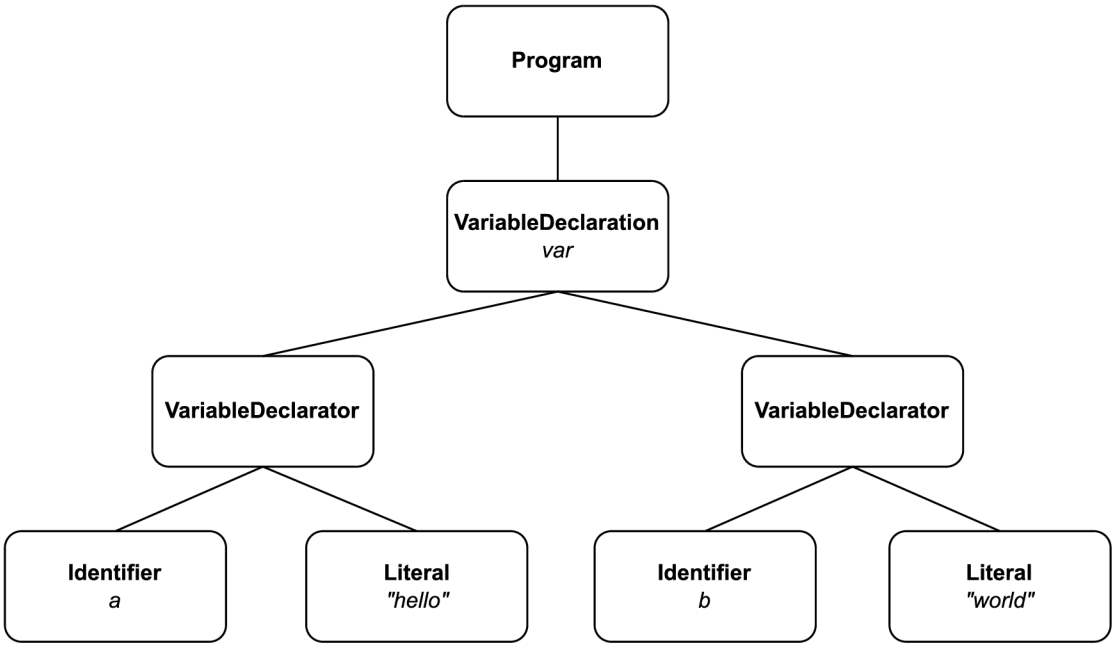
[10] https://www.toptal.com/developers/javascript-minifier/
[11] https://astexplorer.net/
[12] https://www.daftlogic.com/projects-online-javascript-obfuscator.htm



Obrázek 6.1: Grafická reprezentace AST. V každém uzlu je uveden jeho typ, informace o pozici v kódu a v některých případech doplňující data. Uzly ohraničené tučnou přerušovanou čarou obsahují změny oproti stromu na obrázku 6.2.



Obrázek 6.2: Minifikovaná reprezentace stromu z obrázku 6.1. Uzly ve druhé úrovni stromu byly sloučeny, hodnota literálu *world* má změněný typ uvozovek. Ve všech uzlech se dále změnila data o poloze v kódu, který se minifikací zkrátil.



Obrázek 6.3: Grafická reprezentace AST vstupu z výpisu 6.5 vytvořený prezentovaným nástrojem s použitím normalizace.

že obfuskovaný kód často přistupuje k API sloužícím ke sledování, *fignerprintingu*, prací s *cookies* a jejich krádežemi. Oproti minifkaci má obfuskace negativní dopad na rychlost vykonávání kódu. Z pohledu jednotlivých technik je nejčastější spouštění obsahu skriptu za běhu pomocí funkce `eval()`.

Domnívám se, že podpora normalizace představená v podsekci 6.1.5 je vhodná i pro zlepšení analýzy obfuskovaného kódu. Práce si však neklade za cíl detekci obfuskovaného a minifikovaného kódu plně pokrýt a při detekci soupeřit s autory, snažícími se zranitelný kód před nástrojem provádějícím analýzu skrýt. Vytvořený nástroj však s transformovaným kódem dokáže bez problému pracovat a pokud je transformovaný kód známým vstupem, nástroj ho dokáže rozpoznat. Příkladem budiž minifikované verze knihovny *jQuery*, které byly v nástroji použity. Pokud je kód zranitelnosti ve fázi učení minifikovaný anebo obfuskovaný, dokáže jej nástroj v režimu analýzy rozeznat. Použité normalizační techniky pouze zvětšují množinu potenciálně rozpoznatelných skriptů. Budoucí vývoj nástroje by se mohl ubírat směrem rozšíření normalizačních technik i o techniky obfuskace a v případě plného pokrytí nejčastějších transformací by nástroj mohl dosáhnout lepších výsledků.

6.2 Známé zranitelnosti a pomocné skripty

Významnou oblastí, kterou jsem se v této práci zabýval, je problematika zpracování známých zranitelností. Za známou zranitelnost považuji takovou zranitelnost, která byla uveřejněna v některém z veřejných registrů. Pro účely implementace jsem zvolil *Github Advisory*, které zmiňuji již v sekci 4.1. Většina nahlášených zranitelností je totiž nahlášena v databázích *Snyk*, *NVD* i *Github Advisory*. Poslední zmíněná má jednoduché webové uživatelské rozhraní s funkcemi vyhledávání a filtrování pomocí platformy. Z pohledu návrhu pokrývá tato sekce celou vrstvu č.1 a horní poloviny vrstvy č.2, tedy od záznamu v databázi zranitelností až po uloženou reprezentaci abstraktního syntaktického stromu.

Pomocné skripty se v repositáři nachází ve složce `vulnerability-processing`, podrobněji se struktuře věnuje příloha A. Zaměřuji se na podporu tří typů zpracování zranitelností:

- automatizovaně se zdrojovými soubory verzovanými nástrojem Git,
- automatizovaně se zdrojovými soubory staženými přes FTP,
- ručním zpracováním.

Zásadní rozhodnutí, které jsem musel v této fázi práce udělat, byl výběr granularity při určování zranitelného kódu. Stejně jako nástroj *Eclipse-Steady*, zmíněný v podsekci 4.5.1, považuji za zranitelný takový kód, který byl změněn v *commitu*, který je označen jako oprava nějaké zranitelnosti. Je nutné se rozhodnout, jaká je optimální jednotka kódu, kterou budeme považovat za atomickou a zranitelnou. Může se jednat o literál, funkci, výraz, či o celý skript. Pro jednoduchost jsem se rozhodl při automatizovaném zpracování považovat celý soubor za atomickou jednotku. Chápání souboru atomicky vede ke snadné podpoře oprav, celý skript je ve fázi opravy nahrazen jiným skriptem.

Po průchodu webem, popsaným v sekci 7.2, se tato volba jeví jako vhodná pro případy, kdy ve fázi zpracování zranitelností je vstupem distribuovaná verze knihovny a ne zdrojové kódy. Pokud je zranitelnost „naučena“ na zdrojových kódech, je nutné, aby se i ve fázi detekce zdrojové kódy objevovaly ve stejné podobě, což v praxi neplatí. Pokud je naopak zranitelnost „naučena“ na všech možných sestavách dané knihovny, je úspěšnost detekce podstatně vyšší.

Po manuální identifikace *commitu*, který opravuje zranitelnost, jsou tři možnosti postupu:

1. manuální zpracování,
2. zpracování na základě URL Git repositáře a *hashe commitu*,
3. zpracování na základě URL zranitelné knihovny a opravy.

Hlavní výhodou manuálního zpracování je svoboda při označení kódu zranitelnosti a opravy. Postup manuálního přidání další zranitelnosti je následovný:

1. vlož zranitelný kód a kód opravy do skriptu `show_ast_manual.js`,
2. nastav příznak `showAstOnly` na true a spusť skript,
3. vlož získané AST ze standardního výstupu do skriptu,
4. nastav příznak `showAstOnly` na false a spusť skript,
5. vlož získané *hashe* ze standardního výstupu společně s metadaty zranitelnosti do skriptu `js-to-ast/src/run.js`.

Do testovací sady bylo přidáno 5 manuálně vytvořených zranitelností. Ve 4 případech jde o zranitelné literály regulárních výrazů, v jednom případě o for-cyklus. Automatizované zpracování má podobný postup, je však celý abstrahovaný do skriptů `run.sh`, `show_ast.js`, `process_ftp.sh` a `process_git.sh`. `Shell` skripty používám kvůli snadné integraci s Gitem, ve všech ostatních případech se držím JavaScriptu, kvůli použití knihovny ze sekce 6.1. Zpracování konkrétních souborů přes FTP je oproti Gitu snazší. Soubory zranitelnosti i opravy jsou staženy a převedeny na AST. Dále jsou spočítány jejich *hashe*, které jsou

společně s metadaty přidány do generovaných souborů `js-to-ast/src/generated_*.json`. V případě Gitu je situace složitější. Prvním krokem je stažení repositáře. Následuje průchod soubory, který byly v *commitu* změněny, mimo soubory testů. Původní verze je považována za zranitelnou, verze po změně je považována za opravenou. Zbytek procesu je stejný jako u stažení přes FTP.

S pomocí Gitu jsem v rámci práce automatizovaně zpracoval 19 zranitelností, skrz FTP 152 zranitelných verzí a sestav knihovny *jQuery*, vybraných 5 zranitelností jsem zpracoval ručně. Všechny zpracované zranitelnosti jsou uvedeny v příloze C. Prostor pro návaznou práci je v oblasti automatizovaného zpracování z Gitu, které nevede k produkčně použitelné podobě zranitelností. Možné vylepšení spočívá v bližším určení zranitelného kódu, například na úrovni funkcí, či jiných celků. Problémem, kvůli kterému jsem toto vylepšení neimplementoval, je složitý výběr kódu opravy. Další slabinou použití *commitů* je ignorace historie. To, že je nějaká část kód opravena, nutně neznamená, že je zranitelná jen předchozí implementace. Zranitelné mohou být i všechny dřívější podoby dané části kódu. Pro úplnost je tedy nutné zranitelnosti individuálně a hluboce posuzovat.

6.3 Rozšíření prohlížeče Chrome

Hlavním výstupem práce je rozšíření prohlížeče Chrome. Budoucí pokračování práce může zahrnovat migraci rozšíření i na ostatní, méně rozšířené, prohlížeče. Z pohledu návrhu (sekce 5.2) pokrývá rozšíření celou vrstvu č.3 a částečně přebírá některé z funkcí spodní poloviny vrstvy č.2.

Implementovat nové rozšíření v *MV2* již nedává smysl. Hlavní náplní práce v této části se tak stalo hledání způsobu, jak v novém *manifestu* analyzovat webovou stránku a vyměnit za běhu její zranitelné skripty. V *MV2* by stačilo prozkoumat všechny odchozí požadavky z prohlížeče. V novém *manifestu* je proces složitější. Implementované rozšíření má následující souborovou strukturu:

```
js-vuln-det
├── browser
│   ├── popup.js
│   └── popup.html
├── background.js
├── content_script.js
└── evaluate.js
```

Proces zpracování v režimu opravy (další režimy jsou popsány níže, režim opravy je nejkomplikovanější) je následovný:

1. Do stránky je při spuštění zaveden `content_script.js`. Jeho prvním úkolem je zastavit veškeré běžící vykonávání na stránce a zobrazit indikátor průběhu. Pomocí `XMLHttpRequest` stáhne obsah otevřené webové stránky a začne ji zpracovávat. Projde všechny uzly typu `script` a odešle jejich obsah, případně URL, do skriptu `background.js`, který zastává roli *service workeru*.

2. Skript `background.js` reaguje na příchozí události. Pokud je obsahem příchozí události URL skriptu, pak daný skript z URL prostředníctvím `fetch` API stáhne. Po získání obsahu skriptu volá funkce z knihovny `js-to-ast` (sekce 6.1), které ve skrip-

tech hledají zranitelnosti a vrací opravený kód. Informace o nalezených zranitelnostech a kódy oprav vrací v odpovědích na zprávy do `content_script.js`.

3. `Content_script.js` v paměti vytváří DOM nové stránky, ve které nahrazuje původní zranitelné skripty jejich získanými opravami.

4. Dále agreguje metadata zranitelností pro danou stránku a posílá je do `background.js` k uložení.

5. Na závěr je nový DOM vložen do zastavené stránky v prohlížeči, na které dosud běžel ukazatel průběhu.

6. Zásadním nedostatkem řešení bylo, že při celkovém nahrazení DOMu nebyly jednotlivé skripty vykonány. To však lze obejít postupným přidáváním skriptů do stránky. Proto je do načteného DOMu vložen skript `evaluate.js`, který je okamžitě vykonán. Jeho úkolem je po jednom projít přes všechny skripty v DOMu a postupně je z DOMu odstranit a zase přidat. Tato operace v prohlížeči vynutí jejich vykonání. Alternativou je použití funkce `eval`, která však nedokáže všechny skripty vykonat správně.

Rozšíření podporuje 4 režimy běhu, které se liší v chování k analyzované stránce a zranitelným skriptům:

- `Disabled` – rozšíření je načteno, ale `content_script.js` nezastaví vykonání stránky ani neprochází její obsah.

- `Analyze` – vykonávání není zastaveno, nalezené zranitelnosti jsou ukládány a reportovány, ale zranitelné skripty nejsou na stránce nijak omezeny.

- `Block` – vykonávání je zastaveno, zranitelné skripty jsou odstraněny z DOMu, zranitelnosti jsou reportovány stejně jako v režimu `Analyze`. Zranitelné skripty nejsou vůbec vykonány, pokud stihne rozšíření zafungovat před jejich vykonáním. Tato *race condition* je popsána níže.

- `Repair` – vykonávání je zastaveno, zranitelné skripty se známými opravami jsou opraveny, zranitelnosti jsou reportovány stejně jako v režimu `Analyze`.

Nutnost použití `content_script.js` k zatavení prvního načtení stránky s sebou nese nepříjemnou *race condition* v režimech `Block` a `Repair`. Z hlediska bezpečnosti je chtěné, aby zranitelné skripty nebyly vůbec vykonány. K tomu je potřeba zastavit vykonávání stránky. Problémem je, že z pohledu stránky je `content_script.js` jen další ze skriptů, dochází tak k závodu mezi zranitelnými skripty na stránce a `content_script.js`. Je tak časté, že skripty, které budou následně označeny jako zranitelné, jsou hned po spuštění vykonány. Odstranění této slabiny je jedním z možných vylepšení a pokračování této práce.

Soubory `popup.js` a `popup.html` obsahují implementaci uživatelského rozhraní (obrázek 6.4). Rozhraní pokrývá pouze nutné funkce pro demonstraci a ovládání rozšíření. Konkrétně se jedná o tyto funkce:

- výběr módu,

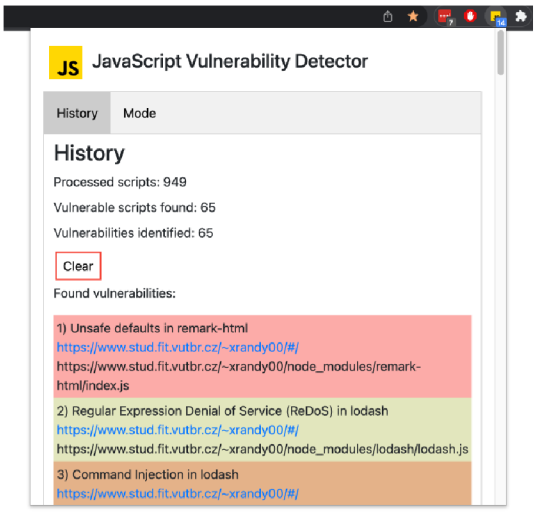
- zobrazení souhrnných statistik,

- vymazání všech dat,

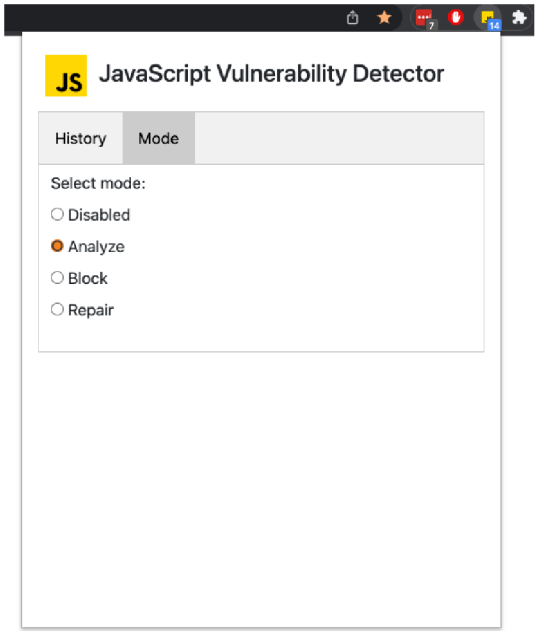
- kompletní výčet nalezených zranitelností,
- notifikace uživatele, že na stránce byly nalezeny zranitelnosti.

Nalezené zranitelnosti jsou společně se souhrnnou statistikou ukládány do uložiště `chrome.storage.local`, které nesdílí data mezi více prohlížeči, ve kterých je uživatel přihlášen. Kompletní uložená data jsou pro účely hlubší analýzy a vývoje vypsány do konzole při každém otevření *popupu* rozšíření. Každá položka ve výpisu zranitelností obsahuje následující informace:

1. pořadové číslo zranitelnosti,
2. URL webu, na kterém byla zranitelnost nalezena,
3. URL zranitelného skriptu, případně informaci, že se jedná o *inline* skript,
4. barva záznamu reprezentuje závažnost zranitelnosti, od nejméně závažné: bílá, zelená, oranžová, červená.



(a) Karta Historie obsahuje souhrnné statistiky, tlačítko pro vymazání dat a kompletní výčet nalezených zranitelností.



(b) Karta výběru módu obsahuje pouze jednoduchý přepínač. Modrý *badge* s číslem 14 na ikoně rozhraní v horním černém rámci označuje počet zranitelností nalezených na aktuálně zobrazené webové stránce.

Obrázek 6.4: Uživatelské rozhraní rozšíření

Kapitola 7

Testování a nasazení

Nedílnou součástí každého softwarového projektu je testování. V případě této práce jsou některé základní funkce pokryty jednotkovými testy, hlavní důraz jsem však kladl na manuální a integrační testy v produkčním prostředí. Pro ověření základních funkcionalit a jejich demonstraci jsem vytvořil jednoduchou webovou stránku obsahující kód, který by rozšíření mělo umět označit za zranitelný. Blíže se mu věnuje sekce 7.1. Dále jsem rozšíření pro účely testování modifikoval tak, aby fungovalo jako *web crawler*[1]. Rozšíření jsem nechal automatizovaně projít část nejnavštěvovanějších webových stránek. Výsledky průchodu prezentuji v sekci 7.2.

7.1 Ukázkový web

Jedním z prvních kroků při implementaci bylo vytvoření webové prezentace, která by demonstrovala funkčnost rozšíření. Tuto prezentaci jsem postupně upravoval, aby vždy vhodně demonstrovala funkčnost rozšíření, a její poslední verzi jsem umístil veřejně na server `merlin`[2]. Rozšíření na tomto webu detekuje 14 zranitelností ve 14 různých skriptech, jak ukazuje obrázek 7.1.

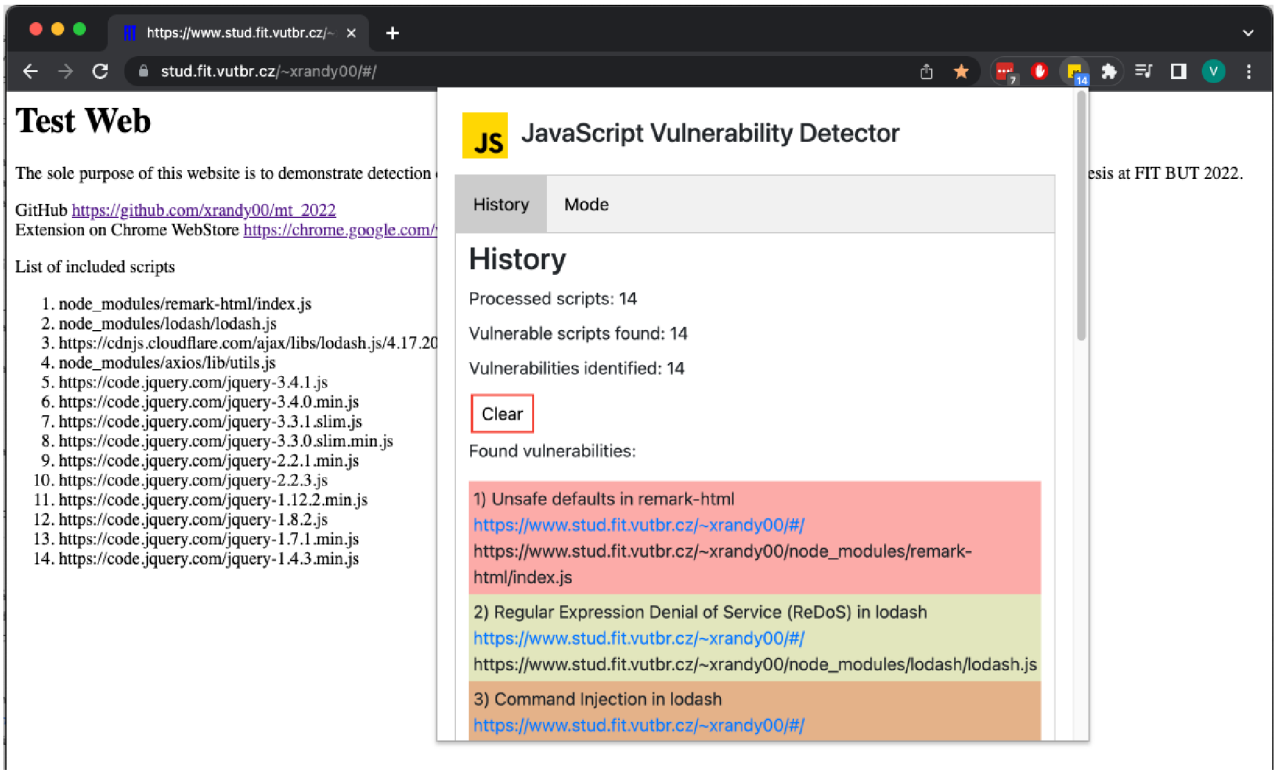
V kořenovém adresáři webu jsem založil Node.js, abych pomocí nástroje NPM mohl snadno do adresáře webové prezentace instalovat zranitelné verze NPM balíčků. Tyto balíčky jsem plánoval nasadit jako součást testovacího webu a odkazovat je z něj. Tento přístup nebyl moc úspěšný, kód stažený z NPM do složky `node_modules` nešel do stránky rozumným způsobem vložit. Většina NPM balíčků má nějaký definovaný vstupní bod, například soubor `index.js`, nebo `nazev_knihovny.js`, který importuje ostatní soubory z balíčku. Problém je, že importy z Node.js nefungují v JavaScriptu v prohlížeči, který potřebuje skripty odkazovat funkcí `require`. I přes tato omezení se mi povedlo najít 2 NPM balíčky, které rozšíření na testovacím webu detekuje – *remark-html* a *lodash*.

Obrázek 7.1 zachycuje na pozadí testovací web, který obsahuje výpis vložených skriptů a knihoven. V popředí se nachází vyskakovací okno rozšíření ukotvené v pravém horním rohu ikonou s číslem 14, ukazujícím počet detekovaných zranitelností na dané stránce. Pokud je rozšíření v režimech `Block` nebo `Repair` lze správnou funkčnost ověřit nahlédnutím do DOMu stránky. V režimu `Block` jsou všechny skripty webu označeny za zranitelné a jsou z DOMu odstraněny. Pro ověření chování rozšíření v režimu `Repair` je například možné

[1] Automatizovaný proces, který prochází dané webové stránky a online je zpracovává, či ukládá jejich obsah pro offline zpracování.

[2] https://www.stud.fit.vutbr.cz/~xrandy00

v DOMu prozkoumat opravené skripty knihovny *jQuery*, které v dokumentační hlavičce po opravě uvádí verzi `3.5.0`.



Obrázek 7.1: Snímek stavu rozšíření po analýze ukázkového webu

7.2 Automatizovaný průchod weby (*crawl*)

Nutným pokračování testování v umělém prostředí, popsaném v minulé sekci, bylo testování v prostředí produkčním. Po celou dobu psaní této práce jsem měl rozšíření nainstalované a spuštěné v režimu `Analyze` a odhalil jsem mnoho zranitelností bez jakéhokoliv negativního dopadu na běžnou, každodenní činnost. V rámci práce jsem však chtěl dosáhnout i měřitelných výsledků.

Rozšíření jsem snadno upravil na *web crawler*, který jsem použil k automatizovanému průchodu 50 000 stránkami ze seznamu 1 milionu nejnavštěvovanějších stránek [18]. Abych vhodně reprezentoval skutečný stav webu, vzal jsem vždy prvních 5 000 z každého statisíce nejnavštěvovanějších stránek. Podle indexů jsem tedy zahrnul stránky s indexy 0 až 5 000, 100 000 až 105 000, atd. Do uživatelského rozhraní rozšíření jsem přidal další kartu, ve které bylo možné zadat počáteční a koncové indexy a průchod z ní spustit. Tato funkcionalita není součástí výsledného rozšíření, byla před publikací odstraněna.

Pro podporu paralelního běhu jsem si založil několik uživatelských profilů v prohlížeči Chrome a do každého z nich jsem rozšíření nainstaloval. Následně jsem každý profil otevřel ve vlastním okně. Každé okno si spravuje svá data izolovaně od ostatních a může tak snadno procházet a analyzovat vlastní podmnožinu stránek, aniž by bylo ostatními okny nějak ovlivněno.

Je nutno zmínit, že na všechny stránky se nepovedlo přistoupit, ať už kvůli neexistenci domény, firewallu či problémům s DNS. Navštěvoval jsem pouze úvodní stránky každé domény. Pokud se stránku nepodařilo do 5 s kompletně načíst, pokračoval jsem na další. Data ze stránek, jejichž načítání nebylo plně dokončeno byla i přesto zahrnuta do statistik.

Cílem průchodu bylo v prvé řadě ukázat, že rozšíření je i ve skutečném nasazení schopné zranitelnosti detekovat. Dále mě zajímala četnost výskytu konkrétních zranitelností, hlavně vztah mezi detekcemi zranitelností knihovny *jQuery* definovanými ručně a zranitelnostmi automatizovaně naučenými na celých, často minifikovaných, sestavách této knihovny. V neposlední řadě mě zajímalo, jestli je možné najít korelaci mezi návštěvností webu a počtem jeho zranitelností.

Po prvotním prozkoumání dat jsem odhalil, že se v nich objevovaly zranitelnosti pocházející ze stránek, které nebyly součástí analyzovaného seznamu stránek.Domnívám se, že některé stránky obsahovaly kód, který prohlížeč při průchodu přesměroval jinam. Tyto záznamy jsem z dat před dalším zkoumáním odstranil. Nalezené zranitelnosti jsem dále setřídil do jednoho seznamu podle domény a do druhého seznamu podle zranitelnosti. Získal jsem tak četnosti detekce jednotlivých zranitelností, 5 nejčastějších uvádím v tabulce 7.1, kompletní data jsou součástí přílohy C.

Ne všechny známé zranitelnosti se povedlo detekovat, což otevírá otázku, jak ve fázi zpracování zranitelností ověřit, že je zranitelnost zpracována správně. Zkoumání prostředků verifikace je jedním z možných pokračování práce. Nepodařilo se najít především zranitelnosti získané ze systému pro správu verzí Git, které se v produkčním prostředí zřejmě vyskytují v jiné podobě než v jaké jsou uloženy v repositáři. Zranitelnou knihovnu *jQuery* se z konkrétních sestav podařilo detekovat v 3 505 případech. Oproti tomu v 4 836 případech se zranitelnou knihovnu *jQuery* povedlo detekovat pouze na základě přítomnosti zranitelného regulárního výrazu. To nepřímo potvrzuje, že má smysl pokoušet se detekovat zranitelnosti jak pomocí celých sestav knihoven, tak i pomocí konkrétních částí kódu. Dále v tabulce 7.2 uvádím 10 webů s nejvíce detekovanými zranitelnostmi. Tyto weby lze použít pro rychlou kontrolu správnosti fungování rozšíření, pokud by testovací web (sekce 7.1) nebyl dostatečný.

Jméno	Počet
Potential XSS vulnerability in jQuery < 3.5.0	4 300
Cross-Site Scripting in jquery < 1.9.0	536
Vulnerable jQuery version 3.3.1	302
Vulnerable jQuery version 3.4.1	299
Vulnerable jQuery version 1.12.4	252

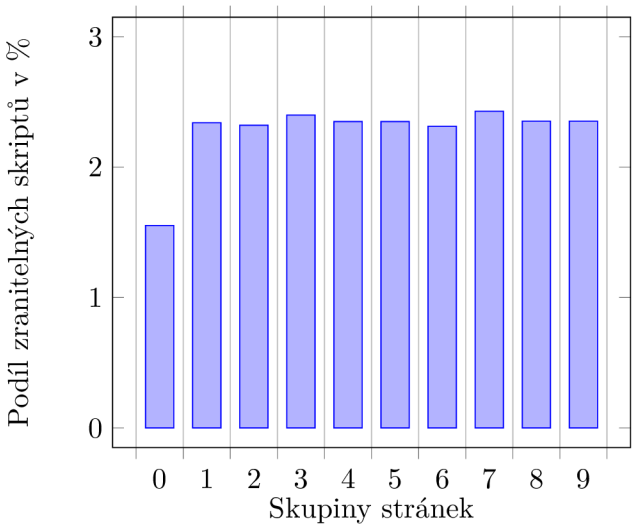
Tabulka 7.1: Nejčastěji detekované zranitelnosti při automatizovaném průchodu. Jména zranitelností jsou ponechána v původním anglickém znění kvůli případnému dohledání v Github Advisory, či ve zdrojových kódech.

Pro každou skupinu stránek (prvních 5 000 z každého statisíce) jsem spočítal podíl detekovaných zranitelných skriptů a všech analyzovaných skriptů a zanesl ho do grafu 7.2. Očekával jsem, že histogram bude mít rostoucí tendenci, tedy, že méně navštěvované stránky budou víc zranitelné. Rozšíření tuto hypotézu nedokáže potvrdit, ani vyvrátit. Lze argumentovat, že stránky s nižší návštěvností skutečně obsahují víc zranitelností, jen je rozšíření nezachytí a opačně pro stránky s nižší návštěvností. I tak je však z grafu 7.2 patrné, že na

Stránka	Index	Počet
timbercubes.com	304 101	4
socialking.in	103 498	3
ideamart.io	300 495	3
robtus.ru	401 465	3
a2zmarketresearch.com	404 678	3
ifminvestors.com	601 098	3
forex-pips.co	604 046	3
ad-vz.ru	801 488	3
tokyo-tabearuki.com	804 087	3
achkayen.com	100 774	2

Tabulka 7.2: 10 stránek, na kterých bylo nalezeno nejvíce zranitelností. Za zmínku stojí, že v těchto 10 stránkách není žádná z 5 000 nejnavštěvovanějších stránek.

stránkách s vyšší návštěvností bylo detekováno méně zranitelností. Od indexu 100 000 výš už je procento zranitelných skriptů bez větších odchylek.



Graf 7.2: Histogram podílu detekovaných zranitelných skriptů v jednotlivých skupinách analyzovaných stránek. Skupiny jsou vytvořeny na základě indexu v seznamu nejnavštěvovanějších stránek. Číslo sloupce odpovídá číslu statisíce, ze kterého skupina pochází, např. ve sloupci 4 jsou souhrnná data stránek z indexů 400 000-405 000.

Celkově jsem zpracoval 354 944 skriptů, z toho 8 129 bylo zranitelných, což odpovídá 2,29 %. Odhaleno bylo 9 748 zranitelností, což je víc, než kolik bylo odhaleno zranitelných skriptů. Důvodem jsou záměrně zanesené duplicity ve zranitelnostech knihovny *jQuery* – rozšíření detekuje jak celé zranitelné sestavení, tak i konkrétní zranitelné konstrukce v něm.

7.3 Zhodnocení a možnosti pokračování práce

Během implementace jsem postupně došel k řešení, které se použitými technikami může měřit s konkurenčními nástroji. Hlavním rozdílem a slabinou, která brání rozšíření mezi

širokou veřejnost je malý objem známých zranitelností. Pokusil jsem se zpracovat 93 zranitelností, mnoho z nich v různých sestavách, včetně minifikovaných. Z toho se však 35 vůbec nepodařilo při produkčním běhu detekovat. Součástí těchto 35 zranitelností jsou všechny zranitelnosti NPM balíčků získané ze zdrojových kódů knihoven v systému Git. Je možné, že jsem zvolil knihovny, které se na klientském webu vůbec nevyskytují. Pravděpodobnější ale je, že se na webu vyskytují jako součásti webových aplikací. U těchto aplikací je časté použití minifikačních nástrojů, které detekci na úrovni celých skriptů znemožní. Jako velmi úspěšná se ukázala detekce celých sestav knihoven, zranitelnosti z této kategorie se podařilo detekovat 3 505krát. Za velmi zajímavé považuji zranitelnosti, které jsem zpracoval ručně. V této kategorii byly zranitelnosti literálů regulárních výrazů, které jsou odolné vůči minifikaci. Celkově byly tyto zranitelnosti detekovány 5 058krát. Na základě těchto výsledků považuji práci za úspěšnou. Výstupem je funkční rozšíření prohlížeče Chrome, volně dostupné na *Chrome Web Store*[3], implementované pod novým, omezujícím, *Manifestem V3*, které dokáže detekovat, blokovat a opravit známé zranitelnosti klientského JavaScriptu.

Poměrně jasným pokračováním práce může být rozšiřování databáze zranitelností. Může jít o aktivitu jedince, či komunity, jako je tomu v případě *Retire.js*. Domnívám se, že je vhodné nadále zachovat podporu všech uvedených způsobů definování zranitelností. Ručně zpracované zranitelnosti umožňují pro každou zranitelnost zvlášť vhodně zvolit úroveň stromu, ve které chceme zranitelnost detekovat a opravit. Automatizovaně zpracované zranitelnosti zase zmenšují úsilí, které je nutné vynaložit. Hlavním problémem současného řešení při automatizovaném zpracování je, že bere skript jako atom, tedy zranitelnosti detekuje na úrovni uzlu typu `Program`, což v praxi funguje pouze při detekci celých sestav knihoven. Jako vhodné rozšíření práce považuji úpravu automatizovaného zpracování zranitelností tak, aby byla granularita uzlů větší. Úkolem je tedy umět z *commitu*, či páru souborů „před“ a „po“, získat nejmenší možný abstraktní syntaktický strom změny. Je možné, že bude potřeba takových stromů z jednoho souboru získat několik, pokud je v souboru několik oddělených změn. Podobně se může stát při zpracování AST opravy, že bude nutné vytvořit mapování M:N mezi abstraktními syntaktickými stromy zranitelností a oprav. Takové úpravy by vyžadovaly zásah do všech úrovní práce a považuji je za náročné.

O něco snadnější se jeví rozšíření o další možnosti normalizace, jak nastiňuje podsekce 6.1.5. Především jde o analýzu existujících minifikačních nástrojů a migraci jejich technik do rozšíření představeného v této práci. K výsledku této práce lze přistupovat jako k sofistikovanému *search and replace* nástroji a lze uvažovat o jeho využití i pro jiné účely, než je detekce zranitelností. Jako zajímavá oblast studia a zdroj doplňujících informací se jeví detekce plagiátů a související úlohy. Úlohu detekce zranitelností můžeme zobecnit na hledání předem známého kódu a jeho variací, v takové abstrakci je pak detekce plagiátů jednou z podúloh, které představené řešení může vykonávat.

Samotné rozšíření má několik známých nedostatků. Prvním je *race condition* při práci v režimu `Block`, jak zmiňuje sekce 6.3. Řešení existuje [24] v knihovně *NSCL*[4], jeho implementace je však nad rámec této práce. Druhým nedostatkem je strohé uživatelské rozhraní rozšíření. Z funkčních požadavků rozhraní nepodporuje stránkování a vyhledávání ve zranitelnostech, zajímavou by mohla být možnost exportu dat. Z uživatelského pohledu se lze zaměřit na typografii, rozložení prvků, celkový design, schéma použitých barev a práci s volným prostorem. Posledním bodem, který stojí za zmínku je migrace řešení na další prohlížeče, ale i platformy, jako je *Azure Devops*.

[3] https://chrome.google.com/webstore/detail/js-vulnerability-detector/bmcojnncgfmglejiinbdnahmkmbgifhk

[4] NoScript Commons Library – https://github.com/hackademix/nscl

Kapitola 8

Závěr

Tato práce se zabývá tvorbou nástroje schopného detekovat známé zranitelnosti v kódu v jazyce JavaScript. Navrhl jsem soustavu nástrojů, která s použitím známých databází zranitelností a abstraktních syntaktických stromů dokáže známé zranitelnosti detekovat a opravit. Navrženou soustavu jsem implementoval a testoval z pohledu časové a paměťové složitosti. Odhalil jsem slabiny v návrhu a prvotní implementaci, které jsem vyřešil zavedením *hashování*. Vše jsem zapouzdřil do rozšíření prohlížeče Chrome, které jsem veřejně publikoval na *Chrome Web Store* pod názvem *JS Vulnerability Detector*. Vytvořené rozšíření dokáže zranitelné skripty nejen detekovat, ale i blokovat anebo opravit.

Vytvořil jsem databázi zranitelností, která vychází z dat v Github Advisory. Tato databáze obsahuje 69 zranitelných verzí knihovny *jQuery* ve 152 sestavách, včetně minifikovaných verzí, 19 náhodně vybraných zranitelností NPM balíčků a 5, vhodně zvolených, ručně zpracovaných zranitelností. S rozšířením jsem provedl manuální testování i automatizovaný průchod 50 000 webovými stránkami. V tomto průchodu jsem zanalyzoval 354 944 skriptů. V 8 129 z nich jsem identifikoval zranitelnost, což odpovídá 2,29 %. Z 69 zranitelných verzí knihovny *jQuery* se podařilo 64 detekovat, celkem v 3 505 případech. Bohužel se nepovedlo detekovat žádnou ze zranitelností NPM balíčků, což může být způsobeno tím, že se v očekávané podobě NPM balíčky na straně klienta nevyskytují. Z 5 ručně zpracovaných zranitelností se 4 podařilo detekovat, celkem 5 058krát.

Rozšíření je implementováno v novém *Manifestu V3*, čímž se odlišuje od konkurenčních nástrojů, které mohou být migrací na novou verzi značně zasaženy. Nástroj je díky použití abstraktních syntaktických stromů generický a lze uvažovat nad rozšířením i do jiných oblastí jako je například detekce plagiátů. Jádrem nástroje je NPM balíček, který je možné použít i samostatně jako konzolovou aplikaci, či jej lze zapojit do jiných nástrojů.

Budoucím pokračováním práce může být úprava zpracování zranitelností NPM balíčků, které se nepodařilo detekovat. Dále je jeden z režimů běhu rozšíření zatížen chybou typu *race condition*, jejíž odstranění by navýšilo přínos rozšíření k bezpečnosti uživatele. V neposlední řadě si zaslouží pozornost uživatelské rozhraní rozšíření, které by mohlo obsahovat další funkční prvky jako je vyhledávání, řazení, stránkování a export dat.

Literatura

[1] Closure compiler nbsp;|nbsp; google developers.
URL https://developers.google.com/closure/compiler

[2] Content-Security-Policy. [Online; accessed 19-January-2022].
URL https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Security-Policy

[3] JavaScript Restrictor. [Online; accessed 19-January-2022].
URL https://polcak.github.io/jsrestrictor/

[4] Usage statistics of JavaScript libraries for websites. [Online; accessed 19-January-2022].
URL https://w3techs.com/technologies/overview/javascript_library

[5] JavaScript. Dec 2021, [Online; accessed 19-January-2022].
URL https://cs.wikipedia.org/wiki/JavaScript

[6] App Review. 2022, [Online; accessed 19-January-2022].
URL https://developer.apple.com/app-store/review/

[7] Eclipse Steady. 2022, [Online; accessed 19-January-2022].
URL https://projects.eclipse.org/proposals/eclipse-steady

[8] Eclipse Steady (Incubator Project). 2022, [Online; accessed 19-January-2022].
URL https://github.com/eclipse/steady

[9] OWASP Top 10 Web Application Security Risks. 2022, [Online; accessed 19-January-2022].
URL https://owasp.org/www-project-top-ten/

[10] Alfadel, M.; Costa, D. E.; Mokhallalati, M.; aj.: On the Threat of npm Vulnerable Dependencies in Node.js Applications. 2020, 2009.09019.

[11] Blanc, G.; Miyamoto, D.; Akiyama, M.; aj.: Characterizing Obfuscated JavaScript Using Abstract Syntax Trees: Experimenting with Malicious Scripts. In *2012 26th International Conference on Advanced Information Networking and Applications Workshops*, 2012, s. 344–351, doi:10.1109/WAINA.2012.140.

[12] Chinthanet, B.; Ponta, S. E.; Plate, H.; aj.: *Code-Based Vulnerability Detection in Node.Js Applications: How Far Are We?* New York, NY, USA: Association for Computing Machinery, 2020, ISBN 9781450367684, str. 1199–1203.
URL https://doi.org/10.1145/3324884.3421838

[13] Dilley, J.; Maggs, B. M.; Parikh, J.; aj.: Globally distributed content delivery. Jun 2018, doi:10.1184/R1/6605972.v1.
URL https://kilthub.cmu.edu/articles/journal_contribution/Globally_distributed_content_delivery/6605972/1

[14] Ferreira, G.; Jia, L.; Sunshine, J.; aj.: Containing Malicious Package Updates in npm with a Lightweight Permission System. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, 2021, s. 1334–1346, doi:10.1109/ICSE43902.2021.00121.

[15] Horňák, P.: Přenos bezpečnostních opatření z Chrome Zero do JavaScript Restrictor. 2020.
URL https://www.fit.vut.cz/study/thesis/22374/

[16] Koishybayev, I.; Kapravelos, A.: Mininode: Reducing the Attack Surface of Node.js Applications. In *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*, San Sebastian: USENIX Association, Říjen 2020, ISBN 978-1-939133-18-2, s. 121–134.
URL https://www.usenix.org/conference/raid2020/presentation/koishybayev

[17] Lauinger, T.; Chaabane, A.; Arshad, S.; aj.: Thou Shalt Not Depend on Me: Analysing the Use of Outdated JavaScript Libraries on the Web. *CoRR*, ročník abs/1811.00918, 2018, 1811.00918.
URL http://arxiv.org/abs/1811.00918

[18] Le Pochat, V.; Van Goethem, T.; Tajalizadehkhoob, S.; aj.: Tranco: A Research-Oriented Top Sites Ranking Hardened Against Manipulation. In *Proceedings of the 26th Annual Network and Distributed System Security Symposium*, NDSS 2019, Únor 2019, doi:10.14722/ndss.2019.23386.

[19] Lerner, A.; Simpson, A. K.; Kohno, T.; aj.: Internet Jones and the Raiders of the Lost Trackers: An Archaeological Study of Web Tracking from 1996 to 2016. In *25th USENIX Security Symposium (USENIX Security 16)*, Austin, TX: USENIX Association, Srpen 2016.
URL https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/lerner

[20] LiProduct, D.: The transition of chrome extensions to Manifest V3.
URL https://developer.chrome.com/blog/mv2-transition/

[21] Miagkov, A.; Cyphers, B.: Google's manifest V3 still hurts privacy, security, and Innovation. Dec 2021.
URL https://www.eff.org/deeplinks/2021/12/googles-manifest-v3-still-hurts-privacy-security-innovation

[22] Musch, M.; Steffens, M.; Roth, S.; aj.: ScriptProtect: Mitigating Unsafe Third-Party JavaScript Practices. In *AsiaCCS*, July 2019.
URL https://publications.cispa.saarland/2894/

[23] Ojamaa, A.; Düüna, K.: Assessing the security of Node.js platform. In *2012 International Conference for Internet Technology and Secured Transactions*, 2012, s. 348–355.

[24] Polčák, L.; Saloň, M.; Maone, G.; aj.: JShelter: Give Me My Browser Back. 2022, doi:10.48550/ARXIV.2204.01392.
URL https://arxiv.org/abs/2204.01392

[25] Skolka, P.; Staicu, C.-A.; Pradel, M.: Anything to Hide? Studying Minified and Obfuscated Code in the Web. In *The World Wide Web Conference*, WWW '19, New York, NY, USA: Association for Computing Machinery, 2019, ISBN 9781450366748, str. 1735–1746, doi:10.1145/3308558.3313752.
URL https://doi.org/10.1145/3308558.3313752

[26] Červinka, Z.: *Rozšíření pro webový prohlížeč zaměřené na ochranu soukromí*. Diplomová práce, Vysoké učení technické v Brně, Fakulta informačních technologií, 2018.
URL https://www.fit.vut.cz/study/thesis/21274/

[27] Švancár, M.: *Přenos bezpečnostních opatření z prohlížeče Brave do rozšíření JavaScript Restrictor*. Diplomová práce, Vysoké učení technické v Brně, Fakulta informačních technologií, 2021.
URL https://www.fit.vut.cz/study/thesis/23310/

[28] Williams, C.: How one developer just broke node, Babel and thousands of projects in 11 lines of JavaScript. Jun 2020, [Online; accessed 19-January-2022].
URL https://www.theregister.com/2016/03/23/npm_left_pad_chaos/

[29] Zerouali, A.; Mens, T.; Decan, A.; aj.: On the Impact of Security Vulnerabilities in the npm and RubyGems Dependency Networks. 2021, 2106.06747.

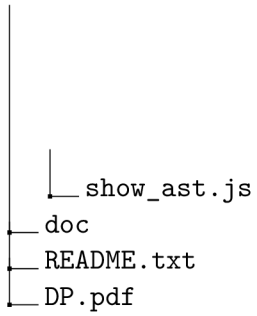
[30] Zimmermann, M.; Staicu, C.-A.; Tenny, C.; aj.: Small World with High Risks: A Study of Security Threats in the npm Ecosystem. In *28th USENIX Security Symposium (USENIX Security 19)*, Santa Clara, CA: USENIX Association, Srpen 2019, ISBN 978-1-939133-06-9, s. 995–1010.
URL https://www.usenix.org/conference/usenixsecurity19/presentation/zimmerman

Příloha A

Obsah přiloženého paměťového média

Struktura adresářů na přiloženém CD odpovídá jednotlivým prvkům nástroje. Složka `js-to-ast` obsahuje Node.js knihovnu popsanou v sekci 6.1. `Js-vuln-detector` je adresář zdrojových kódu rozšíření prohlížeče Chrome ze sekce 6.3. V jeho podsložce `browser` se nachází zdrojové kódy uživatelského rozhraní rozšíření. Složka `test-web` obsahuje zdrojový kód webové stránky uvedené v sekci 7.1. Ve složce `vulnerability-processing` jsou pomocné skripty pro zpracování zranitelností, kterým se blíže věnuje sekce 6.2. V `doc` jsou zdrojové soubory této zprávy ve formátu LaTeX a použitá grafika. Mimo adresáře jsou na nejvyšší úrovni soubory `DP.pdf`, což je tento dokument, a `README.txt` s návodem k lokální instalaci rozšíření, spuštění testovacího webu, přegenerování zranitelností a adresami testovacího webu, stránky rozšíření na *Chrome Web Store* a GitHub repositáře.

```
CD
├── js-to-ast
│   ├── finder.js
│   ├── generated_patches.json
│   ├── generated_vulnerabilities.json
│   ├── generated_vulnerabilities_meta.js
│   └── run.js
├── js-vuln-det
│   ├── browser
│   │   ├── popup.js
│   │   └── popup.html
│   ├── background.js
│   ├── content_script.js
│   ├── evaluate.js
│   └── manifest.json
├── test-web
│   └── index.html
└── vulnerability-processing
    ├── process_ftp.sh
    ├── process_git.sh
    ├── run.sh
    └── show_ast_manual.js
```


show_ast.js
doc
README.txt
DP.pdf

Příloha B

ESTree specifikace

Jak zmiňuje kapitola 6, použitý překladač *Acorn* splňuje specifikaci *ESTree*[1]. Tato příloha obsahuje agregovanou specifikaci *ESTree Spec es5* spolu s rozšířeními `es2015`, `es2016`, `es2017`, `es2018`, `es2019`, `es2020`, `es2021` a `es2022`. Výpis B.1 obsahuje úplný výčet typů a atributů uzlů abstraktního syntaktického stromu, se kterými vytvořená knihovna pracuje. Tato příloha představuje ucelený pohled na aktuální podobu *ESTree* specifikace, která je jinak dostupná pouze jako sada souborů popisujících změny v jednotlivých verzích oproti předchozí verzi. Při tvorbě této přílohy jsem na původní verzi `es5` postupně aplikoval všechny seznamy změn. V případě pokračování této práce může být nutná aktualizace na novou verzi knihovny *Acorn*, resp. knihovny *Escodegen*, resp. *ESTree* specifikace. Je vhodné začít rozšířením této přílohy tak, aby odpovídala nejnovější verzi specifikace a následně aktualizovat potřebné knihovny. Pokud budou zavedeny do specifikace zpětně nekompatibilní změny, bude potřeba znovu vytvořit databázi zranitelností, které se blíže věnuje sekce 6.2.

```
Node
    type: string;
    loc: SourceLocation | null;
SourceLocation
    source: string | null;
    start: Position;
    end: Position;
Position
    line: number; // >= 1
    column: number; // >= 0
Identifier <: Expression, Pattern
    type: "Identifier";
    name: string;
Literal <: Expression
    type: "Literal";
    value: string | boolean | null | number | RegExp | bigint;
RegExpLiteral <: Literal
  regex:
    pattern: string;
    flags: string;
Program <: Node
```

[1] https://github.com/estree/estree

```
    type: "Program";
    sourceType: "script" | "module";
    body: [ Statement | ModuleDeclaration ];
Function <: Node
    id: Identifier | null;
    generator: boolean;
    params: [ Pattern ];
    body: FunctionBody;
Statement <: Node
ExpressionStatement <: Statement
    type: "ExpressionStatement";
    expression: Expression;
Directive <: ExpressionStatement
    expression: Literal;
    directive: string;
BlockStatement <: Statement
    type: "BlockStatement";
    body: [ Statement ];
FunctionBody <: BlockStatement
    body: [ Directive | Statement ];
EmptyStatement <: Statement
    type: "EmptyStatement";
DebuggerStatement <: Statement
    type: "DebuggerStatement";
WithStatement <: Statement
    type: "WithStatement";
    object: Expression;
    body: Statement;
ReturnStatement <: Statement
    type: "ReturnStatement";
    argument: Expression | null;
LabeledStatement <: Statement
    type: "LabeledStatement";
    label: Identifier;
    body: Statement;
BreakStatement <: Statement
    type: "BreakStatement";
    label: Identifier | null;
ContinueStatement <: Statement
    type: "ContinueStatement";
    label: Identifier | null;
IfStatement <: Statement
    type: "IfStatement";
    test: Expression;
    consequent: Statement;
    alternate: Statement | null;
SwitchStatement <: Statement
    type: "SwitchStatement";
```

```
    discriminant: Expression;
    cases: [ SwitchCase ];
SwitchCase <: Node
    type: "SwitchCase";
    test: Expression | null;
    consequent: [ Statement ];
ThrowStatement <: Statement
    type: "ThrowStatement";
    argument: Expression;
TryStatement <: Statement
    type: "TryStatement";
    block: BlockStatement;
    handler: CatchClause | null;
    finalizer: BlockStatement | null;
CatchClause <: Node
    type: "CatchClause";
    param: Pattern | null;
    body: BlockStatement;
WhileStatement <: Statement
    type: "WhileStatement";
    test: Expression;
    body: Statement;
DoWhileStatement <: Statement
    type: "DoWhileStatement";
    body: Statement;
    test: Expression;
ForStatement <: Statement
    type: "ForStatement";
    init: VariableDeclaration | Expression | null;
    test: Expression | null;
    update: Expression | null;
    body: Statement;
ForInStatement <: Statement
    type: "ForInStatement";
    left: VariableDeclaration | Pattern;
    right: Expression;
    body: Statement;
Declaration <: Statement
FunctionDeclaration <: Function, Declaration
    type: "FunctionDeclaration";
    id: Identifier;
VariableDeclaration <: Declaration
    type: "VariableDeclaration";
    declarations: [ VariableDeclarator ];
    kind: "var" | "let" | "const";
VariableDeclarator <: Node
    type: "VariableDeclarator";
    id: Pattern;
```

```
    init: Expression | null;
Expression <: Node
ThisExpression <: Expression
    type: "ThisExpression";
ArrayExpression <: Expression
    type: "ArrayExpression";
    elements: [ Expression | SpreadElement | null ];
ObjectExpression <: Expression
    type: "ObjectExpression";
    properties: [ Property | SpreadElement ];
Property <: Node
    type: "Property";
    key: Expression;
    value: Expression;
    method: boolean;
    shorthand: boolean;
    computed: boolean;
    kind: "init" | "get" | "set";
FunctionExpression <: Function, Expression
    type: "FunctionExpression";
UnaryExpression <: Expression
    type: "UnaryExpression";
    operator: UnaryOperator;
    prefix: boolean;
    argument: Expression;
enum UnaryOperator
    "-" | "+" | "!" | "~" | "typeof" | "void" | "delete"
UpdateExpression <: Expression
    type: "UpdateExpression";
    operator: UpdateOperator;
    argument: Expression;
    prefix: boolean;
enum UpdateOperator
    "++" | "--"
BinaryExpression <: Expression
    type: "BinaryExpression";
    operator: BinaryOperator;
    left: Expression | PrivateIdentifier;
    right: Expression;
enum BinaryOperator
    "==" | "!=" | "===" | "!=="
         | "<" | "<=" | ">" | ">="
         | "<<" | ">>" | ">>>"
         | "+" | "-" | "*" | "/" | "%"
         | "|" | "^" | "&" | "in" | "**"
         | "instanceof"
AssignmentExpression <: Expression
    type: "AssignmentExpression";
```

```
    operator: AssignmentOperator;
    left: Pattern;
    right: Expression;
enum AssignmentOperator
    "=" | "+=" | "-=" | "*=" | "/=" | "%="
        | "<<=" | ">>=" | ">>>="
        | "|=" | "^=" | "&=" | "**=" |
        "||=" | "&&=" | "??="
LogicalExpression <: Expression
    type: "LogicalExpression";
    operator: LogicalOperator;
    left: Expression;
    right: Expression;
enum LogicalOperator
    "||" | "&&" | "??"
MemberExpression <: ChainElement
    type: "MemberExpression";
    object: Expression;
    property: Expression | PrivateIdentifier;
    computed: boolean;
ConditionalExpression <: Expression
    type: "ConditionalExpression";
    test: Expression;
    alternate: Expression;
    consequent: Expression;
CallExpression <: ChainElement
    type: "CallExpression";
    callee: Expression | Super;
    arguments: [ Expression | SpreadElement ];
NewExpression <: Expression
    type: "NewExpression";
    callee: Expression;
    arguments: [ Expression | SpreadElement ];
SequenceExpression <: Expression
    type: "SequenceExpression";
    expressions: [ Expression ];
Pattern <: Node
ForOfStatement <: ForInStatement
    type: "ForOfStatement";
    await: boolean;
Super <: Node
    type: "Super";
SpreadElement <: Node
    type: "SpreadElement";
    argument: Expression;
ArrowFunctionExpression <: Function, Expression
    type: "ArrowFunctionExpression";
    body: FunctionBody | Expression;
```

```
    expression: boolean;
    generator: false;
    async: boolean;
YieldExpression <: Expression
    type: "YieldExpression";
    argument: Expression | null;
    delegate: boolean;
TemplateLiteral <: Expression
    type: "TemplateLiteral";
    quasis: [ TemplateElement ];
    expressions: [ Expression ];
TaggedTemplateExpression <: Expression
    type: "TaggedTemplateExpression";
    tag: Expression;
    quasi: TemplateLiteral;
TemplateElement <: Node
    type: "TemplateElement";
    tail: boolean;
    value:
        cooked: string | null;
        raw: string;
AssignmentProperty <: Property
    type: "Property"; // inherited
    value: Pattern;
    kind: "init";
    method: false;
ObjectPattern <: Pattern
    type: "ObjectPattern";
    properties: [ AssignmentProperty | RestElement ];
ArrayPattern <: Pattern
    type: "ArrayPattern";
    elements: [ Pattern | null ];
RestElement <: Pattern
    type: "RestElement";
    argument: Pattern;
AssignmentPattern <: Pattern
    type: "AssignmentPattern";
    left: Pattern;
    right: Expression;
Class <: Node
    id: Identifier | null;
    superClass: Expression | null;
    body: ClassBody;
ClassBody <: Node
    type: "ClassBody";
    body: [ MethodDefinition | PropertyDefinition | StaticBlock ];
MethodDefinition <: Node
    type: "MethodDefinition";
```

```
    key: Expression | PrivateIdentifier;
    value: FunctionExpression;
    kind: "constructor" | "method" | "get" | "set";
    computed: boolean;
    static: boolean;
ClassDeclaration <: Class, Declaration
    type: "ClassDeclaration";
    id: Identifier;
 ClassExpression <: Class, Expression
    type: "ClassExpression";
MetaProperty <: Expression
    type: "MetaProperty";
    meta: Identifier;
    property: Identifier;
ModuleDeclaration <: Node
ModuleSpecifier <: Node
    local: Identifier;
ImportDeclaration <: ModuleDeclaration
    type: "ImportDeclaration";
    specifiers: [
        ImportSpecifier |
        ImportDefaultSpecifier |
        ImportNamespaceSpecifier ];
    source: Literal;
ImportSpecifier <: ModuleSpecifier
    type: "ImportSpecifier";
    imported: Identifier | Literal;
ImportDefaultSpecifier <: ModuleSpecifier
    type: "ImportDefaultSpecifier";
ImportNamespaceSpecifier <: ModuleSpecifier
    type: "ImportNamespaceSpecifier";
ExportNamedDeclaration <: ModuleDeclaration
    type: "ExportNamedDeclaration";
    declaration: Declaration | null;
    specifiers: [ ExportSpecifier ];
    source: Literal | null;
ExportSpecifier <: ModuleSpecifier
    type: "ExportSpecifier";
    local: Identifier | Literal;
    exported: Identifier | Literal;
AnonymousDefaultExportedFunctionDeclaration <: Function
    type: "FunctionDeclaration";
    id: null;
AnonymousDefaultExportedClassDeclaration <: Class
    type: "ClassDeclaration";
    id: null;
ExportDefaultDeclaration <: ModuleDeclaration
    type: "ExportDefaultDeclaration";
```

```
    declaration: AnonymousDefaultExportedFunctionDeclaration |
    FunctionDeclaration | AnonymousDefaultExportedClassDeclaration |
    ClassDeclaration | Expression;
ExportAllDeclaration <: ModuleDeclaration
    type: "ExportAllDeclaration";
    exported: Identifier | Literal | null;
    source: Literal;
AwaitExpression <: Expression
    type: "AwaitExpression";
    argument: Expression;
BigIntLiteral <: Literal
  bigint: string;
ChainExpression <: Expression
  type: "ChainExpression";
  expression: ChainElement;
ChainElement <: Node
  optional: boolean;
ImportExpression <: Expression
  type: "ImportExpression";
  source: Expression;
PropertyDefinition <: Node
    type: "PropertyDefinition";
    key: Expression | PrivateIdentifier;
    value: Expression | null;
    computed: boolean;
    static: boolean;
PrivateIdentifier <: Node
    type: "PrivateIdentifier";
    name: string;
StaticBlock <: BlockStatement
    type: "StaticBlock";
```

Výpis B.1: ESTree specifikace

Příloha C

Výčet naučených zranitelností

Tato příloha obsahuje výčet zranitelností, které rozšíření dokáže za ideálních podmínek detekovat. První je uveden seznam zranitelností, které jsou generovány pomocnými skripty z kódu na GitHubu, případně z CDN a pro úspěšnou detekci vyžadují, aby se celý obsah souboru použitého pro naučení zranitelnosti objevil v analyzovaném skriptu, jak je uvedeno v sekci 6.2. U každé zranitelnosti je uvedeno, kolikrát byla detekována v rámci průchodu webů v sekci 7.2. Pokud u zranitelnosti či verze nic uvedeno není, znamená to, že nebyla nalezena ani jednou.

1. Zranitelné verze *jQuery*, včetně minifikovaných, `pack` a `slim` buildů:
   1.0.1, 1.0.2, 1.0.3, 1.0.4,
   1.1.1, 1.1.2, 1.1.3, 1.1.4,
   1.2.1, 1.2.2, 1.2.3, 1.2.4, 1.2.5, 1.2.6 (8),
   1.3.1 (1), 1.3.2 (16),
   1.4.1 (3), 1.4.2 (25), 1.4.3 (1), 1.4.4 (18),
   1.5.1 (4), 1.5.2 (7),
   1.6.1 (3), 1.6.2 (5), 1.6.3 (4), 1.6.4 (4),
   1.7.0 (8), 1.7.1 (51), 1.7.2 (79).
   1.8.0 (8), 1.8.1 (14), 1.8.2 (54), 1.8.3 (115),
   1.9.0 (536), 1.9.1 (142),
   1.10.0 (1), 1.10.1 (18), 1.10.2 (137),
   1.11.0 (97), 1.11.1 (193), 1.11.2 (89), 1.11.3 (135),
   1.12.0 (61), 1.12.1 (10), 1.12.2 (16), 1.12.3 (8), 1.12.4 (252),
   2.0.0 (5), 2.0.1 (24), 2.0.2 (6), 2.0.3 (24),
   2.1.0 (21), 2.1.2, 2.1.3 (57), 2.1.4 (73),
   2.2.0 (26), 2.2.1 (8), 2.2.2 (7), 2.2.3 (18), 2.2.4 (188),
   3.0.0 (11), 3.1.0 (45), 3.1.1 (62),
   3.2.0 (2), 3.2.1 (217), 3.3.0, 3.3.1 (302),
   3.4.0 (15), 3.4.1 (299).
   Tyto knihovny jsou detekovány jako celek, jejich opravou je náhrada za verzi `3.5.0`. Zranitelností v *jQuery* je více, jako hlavní pro jednoduchost uvádím možný XSS útok[1], který postihuje všechny verze od `1.0.1` až po `3.5.0`. V součtu byly zranitelnosti z této kategorie detekovány 3 505krát, což potvrzuje, že tento přístup je možné k úspěšné detekci použít. Zbytek tohoto seznamu obsahuje zranitelnost NPM balíčků. Žádnou z nich se však nepodařilo na produkčních webech detekovat. To může být způsobeno

[1] https://github.com/advisories/GHSA-jpcq-cgw6-v4j6

tím, že se NPM balíčky na straně klienta vyskytují pouze jako zkomprimovaná součást webových aplikací.

2. *remark-html* – Unsafe defaults in ‘remark-html‘
   https://github.com/advisories/GHSA-9q5w-79cv-947m
3. *lodash* – Command Injection in lodash
   https://github.com/advisories/GHSA-35jh-r3h4-6jhm
4. *lodash* – Prototype Pollution in lodash
   https://github.com/advisories/GHSA-p6mc-m468-83gw
5. *request* – Remote Memory Exposure in request
   https://github.com/advisories/GHSA-7xfp-9c55-5vqj
6. *underscore* – Arbitrary Code Execution in underscore
   https://github.com/advisories/GHSA-cf4h-3jhx-xvhq
7. *axios* – Server-Side Request Forgery in Axios
   https://github.com/advisories/GHSA-4w2v-q235-vp99
8. *axios* – Incorrect Comparison in axios
   https://github.com/advisories/GHSA-cph5-m8f7-6c5x
9. *minimist* – Prototype Pollution in minimist
   https://github.com/advisories/GHSA-vh95-rmgr-6w4m
10. `shelljs` – Improper Privilege Management in shelljs
    https://github.com/advisories/GHSA-4rq4-32rv-6wp6
11. *js-yaml* – Denial of Service in js-yaml
    https://github.com/advisories/GHSA-2pr6-76vf-7546
12. *js-yaml* – Code Injection in js-yaml
    https://github.com/advisories/GHSA-8j8c-7jfh-h6hx
13. *handlebars* – Prototype Pollution in handlebars
    https://github.com/advisories/GHSA-765h-qjxv-5f44
14. *socket.io* – Insecure defaults due to CORS misconfiguration in socket.io
    h ttps://github.com/advisories/GHSA-fxwf-4rqh-v8g3
15. *socket.io* – Insecure randomness in socket.io
    https://github.com/advisories/GHSA-qv2v-m59f-v5fw
16. *ws* – ReDoS in Sec-Websocket-Protocol header
    https://github.com/advisories/GHSA-6fc8-4gx4-v693
17. *ws* – Denial of Service in ws
    https://github.com/advisories/GHSA-5v72-xg48-5rpm
18. *ejs* – High severity vulnerability that affects ejs
    https://github.com/advisories/GHSA-6x77-rpqf-j6mw

19. *mongoose* – Improper Input Validation in Automattic Mongoose
    https://github.com/advisories/GHSA-8687-vv9j-hgph
20. *redis* – Potential exponential regex in monitor mode
    https://github.com/advisories/GHSA-35q2-47q7-3pc3

Druhý seznam obsahuje zranitelnosti, které byly zpracovány a přidány ručně a za zranitelnou považují pouze specifickou část analyzovaného skriptu, typicky literál nebo jeden cyklus. Zranitelnosti z této kategorie byly detekovány celkem 5 058krát.

1. *jQuery* – Cross-Site Scripting in jquery < 1.9.0 (536)
   https://github.com/advisories/GHSA-2pqj-h3vj-pqgw
2. *jQuery* – Potential XSS vulnerability in jQuery < 3.5.0 (4300)
   https://github.com/advisories/GHSA-gxr4-xjj5-5px2
3. *lodash* – Regular Expression Denial of Service (ReDoS) in lodash (40)
   https://github.com/advisories/GHSA-x5rq-j2xg-h7qm
4. *moment* – Regular Expression Denial of Service in moment (182)
   https://github.com/advisories/GHSA-446m-mv8f-q348
5. *jQuery* – XSS in jQuery <3.4.0 as used in Drupal, Backdrop CMS, and other products
   https://github.com/advisories/GHSA-6c3j-c64m-qhgq

Příloha D

Data abstraktiních syntaktických stromů

V podsekci 6.1.5 zmiňující minifikaci byly uvedeny obrázky 6.1, 6.2 a 6.3. Tyto obrázky obsahují grafickou reprezentaci abstraktních syntaktických stromů před a po minifikaci. Kompletní data ve formátu JSON byla pro tělo práce příliš dlouhá, proto jsou uvedena v této příloze.


```
{
  "type": "Program",
  "start": 0,
  "end": 35,
  "body": [
    {
      "type": "VariableDeclaration",
      "start": 0,
      "end": 17,
      "declarations": [
        {
          "type": "VariableDeclarator",
          "start": 4,
          "end": 16,
          "id": {
            "type": "Identifier",
            "start": 4,
            "end": 5,
            "name": "a"
          },
          "init": {
            "type": "Literal",
            "start": 9,
            "end": 16,
            "value": "hello",
            "raw": "\"hello\""
          }
        }
```

```
      ],
      "kind": "var"
    },
    {
      "type": "VariableDeclaration",
      "start": 18,
      "end": 35,
      "declarations": [
        {
          "type": "VariableDeclarator",
          "start": 22,
          "end": 34,
          "id": {
            "type": "Identifier",
            "start": 22,
            "end": 23,
            "name": "b"
          },
          "init": {
            "type": "Literal",
            "start": 27,
            "end": 34,
            "value": "world",
            "raw": "'world'"
          }
        }
      ],
      "kind": "var"
    }
  ],
  "sourceType": "module"
}
```

Výpis D.1: AST vstupu bez minifikace

```
{
  "type": "Program",
  "start": 0,
  "end": 24,
  "body": [
    {
      "type": "VariableDeclaration",
      "start": 0,
      "end": 24,
      "declarations": [
        {
          "type": "VariableDeclarator",
          "start": 4,
          "end": 13,
```

```
        "id": {
          "type": "Identifier",
          "start": 4,
          "end": 5,
          "name": "a"
        },
        "init": {
          "type": "Literal",
          "start": 6,
          "end": 13,
          "value": "hello",
          "raw": "\"hello\""
        }
      },
      {
        "type": "VariableDeclarator",
        "start": 14,
        "end": 23,
        "id": {
          "type": "Identifier",
          "start": 14,
          "end": 15,
          "name": "b"
        },
        "init": {
          "type": "Literal",
          "start": 16,
          "end": 23,
          "value": "world",
          "raw": "\"world\""
        }
      }
    ],
    "kind": "var"
  }
 ],
 "sourceType": "module"
}
```


Výpis D.2: AST minifikovaného vstupu


```
{
   "type":"Program",
   "body":[
      {
         "type":"VariableDeclaration",
         "declarations":[
            {
               "type":"VariableDeclarator",
```

```
            "id":{
               "type":"Identifier",
               "name":"a"
            },
            "init":{
               "type":"Literal",
               "value":"hello",
               "raw":"\"hello\""
            }
         },
         {
            "type":"VariableDeclarator",
            "id":{
               "type":"Identifier",
               "name":"b"
            },
            "init":{
               "type":"Literal",
               "value":"world",
               "raw":"\"world\""
            }
         }
      ],
      "kind":"var"
   }
]
}
```


Výpis D.3: AST vytvořený pomocí knihovny implementované v této práci s použitím normalizace