UNIVERZITA PAVLA JOZEFA ŠAFÁRIKA V KOŠICIACH
PRÍRODOVEDECKÁ FAKULTA

TESTOVANIE ZRANITEĽNOSTÍ APLIKÁCIÍ

2018 Lucia KOKUĽOVÁ

UNIVERZITA PAVLA JOZEFA ŠAFÁRIKA
PRÍRODOVEDECKÁ FAKULTA

TESTOVANIE ZRANITEĽNOSTÍ APLIKÁCIÍ

BAKALÁRSKA PRÁCA

Študijný program:	informatika
Pracovisko (katedra/ústav):	Ústav informatiky
Vedúci bakalárskej práce:	RNDr. JUDr. Pavol Sokol, PhD.
Konzultant bakalárskej práce:	Mgr. Terézia Mézešová

Košice 2018

Lucia KOKUĽOVÁ

Univerzita P. J. Šafárika v Košiciach
Prírodovedecká fakulta

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta: Lucia Kokuľová
Študijný program: Informatika (Jednoodborové štúdium, bakalársky I. st., denná forma)
Študijný odbor: 9.2.1. informatika
Typ záverečnej práce: Bakalárska práca
Jazyk záverečnej práce: slovenský
Sekundárny jazyk: anglický

Názov: Testovanie zraniteľností aplikácií

Názov EN: Testing of applications´ vulnerabilities

Cieľ: 1) Analyzovať možnosti testovania zraniteľností aplikácii
2) Porovnať aktuálne prístupy a implementácie testovania zraniteľnosti aplikácií
3) Navrhnúť a implementovať rozhranie pre vybrané metódy testovania zraniteľnosti aplikácii

Literatúra:
[1] MUELLER, John Paul. Security for Web Developers: Using JavaScript, HTML, and CSS. " O'Reilly Media, Inc.", 2015.
[2] STUTTARD, Dafydd; PINTO, Marcus. The Web Application Hacker's Handbook: Finding and Exploiting Security Flaws. John Wiley & Sons, 2011.
[3] SHEMA, Mike. Hacking web apps: detecting and preventing web application security problems. Newnes, 2012.
[4] MEUCCI, Matteo; MULLER, Andrew. The OWASP Testing Guide 4.0. Open Web Application Security Project, 2014, 30.

Vedúci: RNDr. JUDr. Pavol Sokol, PhD.
Konzultant: Mgr. Terézia Mézešová
Ústav : ÚINF - Ústav informatiky
Riaditeľ ústavu: prof. RNDr. Viliam Geffert, DrSc.

Dátum schválenia: 09.05.2018

Poďakovanie

Týmto sa chcem poďakovať vedúcemu tejto bakalárskej práce RNDr. JUDr. Pavlovi Sokolovi, PhD. a konzultantke Mgr. Terézii Mézešovej za cenné rady a pomoc pri spracovaní problematiky testovania zraniteľností aplikácií.

Abstrakt v štátnom jazyku

Práca sa zaoberá jednou z dôležitých tém informačnej bezpečnosti, testovaním zraniteľností aplikácií. Každoročne sa na internete objavuje čoraz viac nových aplikácií. Spolu s nimi sa objavujú útočníci hľadajúci ich slabé stránky, teda zraniteľnosti, ktoré by využili vo svoj prospech. Je preto nutné neustále odhaľovať a odstraňovať zraniteľnosti aplikácií, napríklad testovaním zraniteľností. To zahŕňa niekoľko rôznych metód, ktoré pokrývajú rôzne typy zraniteľností. Pri testovaní zraniteľností aplikácií je prvým dôležitým krokom, rozhodnúť sa akým spôsobom sa tieto zraniteľnosti budú testovať. V našej práci popisujeme tieto spôsoby a tiež rôzne metódy, ktoré sa v súčasnosti najčastejšie používajú pri statickej analýze kódu, ktorou sa zaoberáme. Hlavným cieľom praktickej časti našej práce je implementovať jednu z týchto metód a takto prispieť do tvorby open source projektu SonarQube. Pravidlá ktoré sme implementovali, boli vybrané tak, aby sa v nástroji SonarQube ich implementácia ešte nenachádzala. V tomto nástroji sme takisto otestovali dve z aplikácií, ktoré boli vytvorené na našej univerzite.

Kľúčové slová: zraniteľnosť, informačná bezpečnosť, testovanie zraniteľností, penetračné testovanie, statická analýza

Abstrakt v cudzom jazyku

This paper deals with an important topic of information security, testing for application vulnerabilities. Every year, an increasing number of new applications appears on the Internet. Together with them, the attackers are looking for their weaknesses, the vulnerabilities they would use to their advantage. Therefore, it is necessary to continually detect and remove the vulnerabilities from applications, e.g. by vulnerability assessment. This includes several different methods that cover various types of vulnerabilities. When testing application vulnerabilities, the first important step is to decide how these vulnerabilities will be tested. In our work we describe these methods and the various methods that are currently most commonly used in the static analysis of the source code we are dealing with. The main objective of the practical part of our work is to implement one of these methods and thus to contribute to the formation of the open source project SonarQube. The rules we have implemented have been selected so that SonarQube does not involve them. In this tool, we also tested two of the applications that were created at our university.

Keywords: vulnerability, information security, vulnerability assessment, penetration testing, static analysis

Obsah

Zoznam ilustrácií **6**

Zoznam tabuliek **7**

Zoznam skratiek a značiek **8**

Úvod **9**

1 Bezpečnostné aspekty vývoja softvéru **11**

1.1 Fázy vývoja softvéru 11

1.2 Testovanie zraniteľností 12

1.2.1 OWASP 13

1.2.2 Penetračné testovanie 14

1.2.3 Statická analýza kódu 14

1.3 Metódy statickej analýzy 15

1.3.1 Taint analýza – control flow graph 16

1.3.2 Data flow diagram 18

1.3.3 Abstraktný syntaktický strom 19

1.3.4 Code property graph 21

1.3.5 Zhodnotenie 23

2 Statická analýza v praxi **26**

2.1 Požiadavky pre našu infraštruktúru 26

2.2 Nástroje na statickú analýzu 27

3 Nástroj SonarQube **29**

3.1 Tvorba pravidiel v SonarQube 29

3.2 Využitie abstraktného syntaktického stromu 30

3.3 Implementácia pravidiel 30

3.3.1 Pravidlo hľadajúce používanie zraniteľných komponentov 31

3.3.2 Pravidlo hľadajúce zraniteľnosť SQL Injection 32

3.3.3 Pravidlo hľadajúce zraniteľnosť XML External Entity Processing 33

3.3.4 Pravidlo hľadajúce generovanie bezpečných náhodných čísel 34

3.4 Analýza testovacích projektov 35

3.5 Analýza projektov zo služby GitHub 37

Záver **38**

Zoznam použitej literatúry **40**

Prílohy **43**

Zoznam ilustrácií

Obr. 1 Zmeny v zozname OWASP Top 10 v roku 2017 oproti roku 2013 13
Obr. 2 Metóda a k nej prislúchajúci control flow graph 16
Obr. 3 Rozdiel medzi data flow a control flow 18
Obr. 4 Metóda a k nej prislúchajúci abstraktný syntaktický strom 20
Obr. 5 Metóda a k nej prislúchajúci code property graph 22

Zoznam tabuliek

Tab. 1 Porovnanie prác využívajúcich rôzne metódy statickej analýzy 25
Tab. 2 Nástroje na statickú analýzu 27
Tab. 3 Počet nájdených zraniteľností v jednotlivých projektoch 37

Zoznam skratiek a značiek

SDL	The Security Development Lifecycle, cyklus vývoja bezpečnosti
IRP	Incident Response Plan, plán reakcie na (bezpečnostné) incidenty
CFG	Control Flow Graph, graf riadiaceho toku
CPG	Code Property Graph, graf vlastností kódu
PDG	Program Dependence Graph, graf programových závislostí
CG	Call Graph, graf volaní podprogramov určitého programu
DFD	Data Flow Diagram, diagram toku údajov
XSS	Cross-Site Scripting, zraniteľnosť webových aplikácií
PHP	Hypertext Preprocessor, open source skriptovací jazyk
CRF	Collision resistance function, funkcia odolná voči kolíziám
SHA-1	Secure Hash Algorithm 1, kryptografická hašovacia funkcia

Úvod

Dennodenne dochádza k rozvoju nových technológií, a teda aj technológií pre vývoj informačných systémov. Okrem kladného prínosu týchto technológií, má toto napredovanie aj negatívny dopad. Ten spočíva v tom, že vznikajú nové bezpečnostné zraniteľnosti a bezpečnostné riziká ohrozujúce tri základné atribúty informačnej bezpečnosti, a to dôvernosť, integritu a dostupnosť informácií. Testovaním zraniteľnosti aplikácií môžeme odhaliť bezpečnostné riziká danej aplikácie, a tým predísť využitiu bezpečnostnej zraniteľnosti zo strany útočníka. Príkladom je využitie alebo odcudzenie citlivých dát útočníkom. Najdôležitejším dôvodom testovania aplikácií je ochrana informácií, ktoré dané aplikácie uchovávajú a využívajú.

Ďalším, nemenej dôležitým dôvodom, prečo je testovanie zraniteľností dôležité, je rozsiahla dostupnosť informácií na internete. V dnešnej dobe informačných technológií sú na internete dostupné rôzne návody a tipy, ako si môže aj laik (napr. „amatérsky programátor") vytvoriť vlastnú aplikáciu. Samozrejmosťou je, že takto vytvorené aplikácie sú veľkým bezpečnostným rizikom, pretože laici (amatérski programátori) nemajú znalosti dostatočné na dosiahnutie bezpečného zdrojového kódu.

Testovanie zraniteľností aplikácií v rámci informačnej bezpečnosti obsahuje niekoľko metód, ako takúto činnosť vykonávať. Tieto metódy sú zahrnuté v dvoch hlavných skupinách, penetračnom testovaní a statickej analýze kódu. Penetračný test simuluje reálnu činnosť útočníka s cieľom odhaliť bezpečnostné a iné zraniteľnosti systému. Statická analýza kódu môže byť označená ako opak penetračného testovania. Testuje samotný zdrojový kód, avšak bez akéhokoľvek jeho spustenia a na rozdiel od penetračného testu, môže byť vykonávaná automaticky.

Metódy testovania zraniteľností aplikácií sú spravidla vykonávané buď nástrojmi alebo testermi. Existujú rôzne organizácie, ktoré ponúkajú svoje služby v oblasti bezpečnosti a zabezpečenia softvéru. Alternatívu predstavujú automatické nástroje voľne stiahnuteľné na internete. Jedným z takýchto nástrojov je nástroj SonarQube [1]. Je to open source (voľne šíriteľný) nástroj, do ktorého sú pravidlá implementované mnohými testermi. Jedným z cieľov našej práce je implementovať niektoré z pravidiel pre tento nástroj a prispieť tak k tvorbe tohto projektu s otvoreným kódom (open source projektu). Ďalším cieľom je takisto otestovať reálne aplikácie používané na Univerzite Pavla Jozefa Šafárika v Košiciach a taktiež na týchto aplikáciách vyskúšať funkčnosť nami vytvorených pravidiel.

Naša práca začína úvodom do testovania zraniteľností aplikácií. V prvých kapitolách sa venujeme popisu bezpečnostných aspektov vývoja softvéru, taktiež popisujeme fázy vývoja softvéru. V každej fáze stanovujeme, aké zraniteľnosti sa v nej môžu vyskytnúť.

Pred samotným testovaním aplikácií, sme si museli vybrať, ako budeme dané aplikácie testovať. Jednou z možností je penetračné testovanie, ktoré sa zaoberá testovaním aplikácií bez toho, aby tester poznal zdrojový kód. Druhou možnosťou je statická analýza kódu, ktorá skúma priamo daný zdrojový kód aplikácie. My sme si na testovanie aplikácií zvolili statickú analýzu kód. Hlavným dôvodom tohto výberu je to, že je pre nás dôležité odhalenie zraniteľností v aplikáciách ešte predtým, než sa začnú používať.

V ďalších podkapitolách práce sme rozobrali výhody a nevýhody rôznych metód, ktorí využíva statická analýza kódu. Popísali sme metódy využívajúce taint analýzu, data flow diagram, abstraktný syntaktický strom a code property graf. V kapitole Zhodnotenie, sme pomocou tabuľky porovnali tieto metódy a zhodnotili sme ich výhody a nevýhody.

V posledných dvoch kapitolách sa venujeme priamo využitiu statickej analýzy v praxi. Stanovili sme požiadavky na nástroj pre testovanie zraniteľností aplikácií na našej univerzite a porovnali sme viacero nástrojov na statickú analýzu zdrojového kódu. Popisujeme nástroj SonarQube, ktorý sme si vybrali na statickú analýzu a takisto uvádzame dôvody, prečo sme si vybrali práve tento nástroj.

Posledné podkapitoly práce popisujú našu implementáciu, teda pridanie pravidiel na hľadanie zraniteľností v zdrojovom kóde do nástroja SonarQube. Implementovali sme 4 pravidlá v jazyku Java, ktoré sú schopné nájsť chyby alebo zraniteľnosti v 2 programovacích jazykoch, a to Java a PHP. Naše pravidlá sme otestovali na testovacích projektoch z našej univerzity, v ktorých sme odhalili rôzne chyby. V práci popisujeme výsledky testovania pomocou našich pravidiel a takisto výsledky analýzy testovacích projektov pravidlami nástroja SonarQube.

1 Bezpečnostné aspekty vývoja softvéru

Bezpečnosť je neoddeliteľnou súčasťou vývoja softvéru, ktorá zabezpečuje zachovanie základných atribútov informačnej bezpečnosti, a to dôvernosť, integritu a dostupnosť informácií. Táto kapitola popisuje jednotlivé fázy vývoja softvéru a rozoberá v ktorej z týchto fáz je dôležité dbať na bezpečnosť. Takisto sa zaoberá testovaním zraniteľností aplikácií vo všeobecnosti a metódami statickej analýzy.

1.1 Fázy vývoja softvéru

Pre rozdelenie vývoja softvéru do jednotlivých fáz je dôležité oboznámiť sa s pojmom **cyklu vývoja bezpečnosti** (The Security Development Lifecycle, SDL). Je to proces vývoja softvéru, ktorý pomáha vývojárom tvoriť bezpečnejší softvér, riešiť dodržiavanie bezpečnostných požiadaviek a zároveň znižuje vývojové náklady [2]. Cyklus vývoja bezpečnosti je určený pre každý softvér ktorý aktívne spolupracuje s internetom, spracováva citlivé údaje alebo je aktívne využívaný nejakou firmou či inou organizáciou. Je zjavné, že táto selekcia popisuje takmer všetky aplikácie ktoré poznáme. Z tohto dôvodu, je pre vývojárov nevyhnutné porozumieť pojmu SDL a naučiť sa ho využívať pri tvorbe softvéru.

Cyklus vývoja bezpečnosti rozdeľuje vývoj softvéru do siedmich fáz [3]. Prvá z nich sa nazýva **tréningová fáza** (training phase), ktorá zahŕňa len teoretické poučenie všetkých účastníkov vývoja softvéru o tom, ako vyvíjať softvér s využitím SDL.

Druhou fázou je **fáza požiadaviek** (requirements phase), ktorá sa delí na tri časti. V tejto fáze sa určujú požiadavky na bezpečnosť a ochranu osobných údajov, čo minimalizuje narušenie plánov vývoja softvéru. Taktiež sa tu určuje minimálna prijateľná úroveň zabezpečenia softvéru a vykonávajú sa hodnotenia rizík bezpečnosti a súkromia. To znamená, že sa identifikuje, ktoré časti softvéru budú vyžadovať modelovanie hrozieb (threat modeling).

Ďalšou fázou vo vývoji je **fáza dizajnu** (design phase). Tá takisto zahŕňa tri časti, a to určenie požiadaviek na dizajn, analýzu možných útokov a modelovanie hrozieb. Pre nás je najdôležitejšia časť zaoberajúca sa analýzou možných útokov, pretože je v nej možné znížiť príležitosti na zneužitie slabých či zraniteľných miest útočníkom. Dôležitým je skutočnosť, že táto analýza zahŕňa aj blokovanie alebo obmedzenie prístupu

k systémovým službám, či uplatňovanie princípu udeľovania čo najmenej privilégií, všade kde je to možné.

Štvrtá fáza vo vývoji softvéru - **implementačná fáza** (implementation phase), je najdôležitejšia. Tá zahŕňa spísanie a použitie vhodných nástrojov. Inými slovami, ide o kontrolu aktuálnosti nástrojov používaných pri vývoji softvéru, odstránenie nebezpečných funkcií a statickú analýzu kódu. V tejto fáze je statická analýza kódu samozrejmosťou k dosiahnutiu bezpečného zdrojového kódu, keďže pomáha zaistiť dodržiavanie bezpečných zásad programovania a taktiež môže včas odhaliť bezpečnostné zraniteľnosti, ktoré sa v kóde môžu nachádzať.

Po statickej analýze kódu sa prechádza na **verifikačnú fázu** (verification phase). Úlohou tejto fázy je vykonať ďalšie bezpečnostné testy, konkrétne **dynamickú analýzu** a **fuzz testing**. Dynamickú analýzu môžeme chápať ako opak ku statickej analýze, pretože sa nezaoberá zdrojovým kódom. Aplikáciu testuje po jej spustení a snaží sa odhaliť chyby počas jej behu [4]. Fuzz testing je testovanie aplikácií vykonávané podávaním náhodných alebo zámerne zlých dát ako vstup aplikácii, pričom takisto nevyužíva zdrojový kód aplikácie [4]. Spojenie týchto dvoch metód často odhalí ďalšie zraniteľnosti, ktoré statická analýza odhaliť nedokázala a poskytne tak softvéru ďalšiu vrstvu zabezpečenia.

Poslednými dvoma fázami vo vývoji softvéru sú **fáza vydania** (release phase) softvéru a **fáza odozvy** (response phase). Tieto dve finálne fázy zahŕňajú tvorbu plánu na riešenie možných bezpečnostných incidentov (Incident Response Plan, IRP), vykonanie záverečnej bezpečnostnej kontroly a tiež implementáciu IRP v praxi, čo je nutné na otestovanie tohto plánu.

1.2 Testovanie zraniteľností

V oblasti informačnej bezpečnosti, **zraniteľnosť** je slabosť systému, ktorá umožňuje útočníkovi znížiť odolnosť systému, teda ho napadnúť a získať z neho požadované údaje [5]. Zraniteľnosť je priesečníkom troch prvkov [5]:

- citlivosti alebo chyby systému,
- útočníkovho prístupu k tejto chybe a
- schopnosti útočníka túto chybu zneužiť.

Zraniteľnosti sú charakterizované predovšetkým pomocou dvoch faktorov:

- **citlivosť** (náchylnosť k spôsobeniu rizika hrozbou) a
- **kritickosť** (význam aktíva pre organizáciu, jednotlivca či systém).

Ľubovoľný počítačový program môže mať jedno ale aj viac zraniteľných miest, umožňujúcich prístup útočníkovi. Keďže takéto sprístupnenie systému môže spôsobiť ohrozenie **aktíva** (niečoho hodnotného pre entitu ktorá ho vlastní) [6]. Je dôležité systémové zraniteľnosti priebežne odhaľovať a čím skôr ich odstrániť, ak je to možné. K tomuto účelu bolo vyvinutých mnoho metód a nástrojov. Ako bolo spomínané, najdôležitejšími z nich sú penetračné testovanie a statická analýza kódu.

1.2.1 OWASP

Keďže sa internet každým dňom vyvíja, pribúdajú nové zraniteľnosti, ktoré je potrebné preskúmať a zaznamenávať. Jedným z projektov, ktoré sa zaoberajú zraniteľnosťami a bezpečnosťou aplikácií je projekt OWASP [7]. Stránky tohto projektu informujú verejnosť o tom, aké sú najčastejšie typy útokov a aké zraniteľnosti v aplikáciách tieto útoky využívajú. Okrem týchto informácií taktiež popisujú postupy, ako tieto zraniteľnosti odhaliť, ako ich odstrániť a ako aplikácie vyvíjať tak, aby tieto zraniteľnosti neobsahovali. Spoločnosť OWASP opakovane vydáva dokument OWASP Top Ten (Obr. 1), ktorý poskytuje zoznam a popis 10 najkritickejších bezpečnostných chýb webových aplikácií.

OWASP Top 10 - 2013	→	OWASP Top 10 - 2017
A1 – Injection	→	A1:2017-Injection
A2 – Broken Authentication and Session Management	→	A2:2017-Broken Authentication
A3 – Cross-Site Scripting (XSS)	↘	A3:2017-Sensitive Data Exposure
A4 – Insecure Direct Object References [Merged+A7]	∪	A4:2017-XML External Entities (XXE) [NEW]
A5 – Security Misconfiguration	↘	A5:2017-Broken Access Control [Merged]
A6 – Sensitive Data Exposure	↗	A6:2017-Security Misconfiguration
A7 – Missing Function Level Access Contr [Merged+A4]	∪	A7:2017-Cross-Site Scripting (XSS)
A8 – Cross-Site Request Forgery (CSRF)	☒	A8:2017-Insecure Deserialization [NEW, Community]
A9 – Using Components with Known Vulnerabilities	→	A9:2017-Using Components with Known Vulnerabilities
A10 – Unvalidated Redirects and Forwards	☒	A10:2017-Insufficient Logging&Monitoring [NEW,Comm.]

Obr. 1 Zmeny v zozname OWASP Top 10 v roku 2017 oproti roku 2013 [7].

1.2.2 Penetračné testovanie

Penetračné testovanie (**pentesting)** zahŕňa simuláciu skutočného útoku na posúdenie rizika spojeného s potenciálnym narušením bezpečnosti. Pri vykonávaní penetračného testu, testeri nielen odhaľujú zraniteľnosti, ktoré môžu byť útočníkmi použité, ale taktiež sa snažia použiť zraniteľné miesta, ak je to možné. Cieľom je zhodnotiť, čo by mohol útočník získať po úspešnom napadnutí systému. Penetračný test môže byť [8]:

- **interný**, vykonávaný z pohľadu zamestnanca danej firmy, ktorá si takýto test vyžiada, alebo
- **externý**, vykonávaný z pohľadu útočníka.

Test začína **predbežnou fázou**, ktorá spočíva v rozhovore testera a klienta v ktorom sa stanovujú hranice a ciele testovania. Keď sa títo dvaja účastníci dohodnú na rozsahu testovania, formáte výslednej správy a iných dôležitých veciach, začína sa samostatné testovanie. V ďalšej fáze testu tester vyhľadáva verejne dostupné informácie o klientovi a identifikuje potenciálne spôsoby pripojenia k systému. Vo **fáze modelovania hrozieb** tester využíva tieto informácie na určenie hodnoty každej nájdenej informácie a dopad na klienta, ak tieto informácie umožnia útočníkovi preniknúť do systému. Toto vyhodnotenie umožňuje testerovi vypracovanie akčného plánu a metód útoku. Predtým ako tester začne simulovať útok, vykoná analýzu zraniteľností. V tejto fáze sa pokúša odhaliť zraniteľnosti, ktoré by mohol využiť. Úspešné odhalenie zraniteľností môže viesť k nájdeniu ďalších informácií o klientovi, k odhaleniu citlivých údajov alebo k prístupu k iným systémom. V poslednej fáze, **fáze predávania správ**, tester zosumarizuje všetky jeho zistenia pre jeho vedúcich a takisto pre technických pracovníkov systému.

1.2.3 Statická analýza kódu

Samotný názov tejto metódy naznačuje, že testovanie zraniteľností aplikácií vykonáva bez jej spustenia. Táto metóda vyhľadáva chyby v zdrojovom kóde aplikácie. Tento termín sa zvyčajne používa na označenie automatizovaného nástroja. Znamená to teda, že statická analýza je postupnosť presne preddefinovaných krokov, ktoré majú za úlohu nájdenie chýb a zraniteľností v danom zdrojovom kóde. Táto metóda využíva takzvanú „white box" techniku testovania. To znamená, že skúma hlavne implementáciu,

štruktúru kódu [9]. Statická metóda ako vstup dostane zdrojový kód (alebo niekedy binárny kód) aplikácie. Na výstupe poskytuje správu o nájdených zraniteľnostiach. Táto správa ubezpečuje testera o správnom chode aplikácie, alebo naopak, varuje ho pred nesprávnym správaním aplikácie. Súčasne môže navrhovať riešenia zistených problémov.

Statická analýza kódu sa vykonáva buď pomocou samostatných aplikácií, alebo môže byť integrovanou súčasťou vývojového prostredia. Zložitosť takýchto nástrojov sa pohybuje od jednoduchých skenerov (lineárne podľa veľkosti kódu) až po drahé nástroje vykonávajúce podrobnú analýzu kódu, odhaľujúce každú jednu situáciu, ktorá v ňom môže nastať. Statická analýza je v princípe veľmi dobrým nástrojom pre informačnú bezpečnosť, pretože prehľadáva každý jeden riadok kódu a berie do úvahy všetky možné vstupy. Výhodou je tiež to, že takáto analýza môže nájsť chyby v programe predtým, než je program po prvýkrát spustený. Takto má programátor možnosť opraviť chyby, ktoré neodhalí beh aplikácie, hoci znamenajú bezpečnostné riziko. Je dôležité podotknúť, že ostáva na testerovi, aby určil povahu chýb nájdených testovacím nástrojom. Môže sa totiž stať, že chyba nájdená statickou analýzou predstavuje zraniteľnosť a potrebuje byť odstránená (tzv. **true positive**) [10]. Opakom je, ak nájdená chyba nepredstavuje hrozbu a nemôže byť využitá útočníkom (tzv. **false positive**) [10]. Podobne, ak nástroj na statickú analýzu nenašiel žiadny problémy, môže to mať dva dôvody. Samozrejme prvým môže byť, že testovaný zdrojový kód je naozaj bezpečný (tzv. **true negative**) [10]. Druhým, nebezpečnejším dôvodom, môže byť že zdrojový kód obsahuje bezpečnostné zraniteľnosti, ale kvôli rôznym limitom testovacieho nástroja, ich nástroj neodhalil, alebo ich odhalil ale neoznačil ich za problém (tzv. **false negative**) [10].

1.3 Metódy statickej analýzy

Testovanie zraniteľností aplikácií je činnosť, ktorá je vykonávaná dennodenne pri vývoji aplikácií. Je preto zrejmé, že od jeho začiatkov vzniklo mnoho rôznych metód, ako takéto testovanie vykonávať. Tieto metódy sa dajú deliť podľa viacerých kritérií, či už podľa nástrojov ktoré využívajú, alebo podľa formy výsledku, ktorú poskytujú. V našej práci sa zaoberáme štyrmi metódami, ktoré sa pri statickej analýze využívajú najčastejšie, a tými sú:

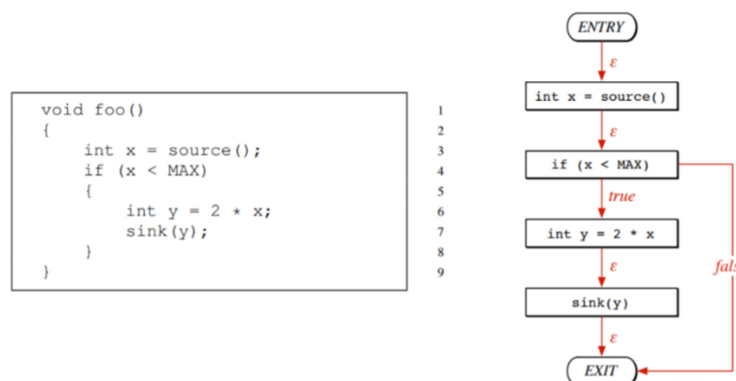
- taint analýza,

- analýza pomocou data flow diagramu,
- analýza pomocou abstraktného syntaktického stromu a
- analýza pomocou code property grafu.

V nasledujúcich podkapitolách popisujeme niekoľko prác venovaných týmto metódam. Súčasne v rámci týchto podkapitol popisujeme, akým spôsobom sú tieto metódy využité na účely statickej analýzy.

1.3.1 Taint analýza – control flow graph

Hlavným cieľom **taint analýzy** je v aplikácii sledovať tok procesov medzi vstupmi (napríklad od používateľa) a výstupmi aplikácie [11]. Na sledovanie tohto toku, taint analýza využíva takzvaný graf riadiaceho toku (control flow graph, CFG), ktorý graficky znázorňuje všetky možné cesty, možnosti ktoré môžu nastať počas behu programu. Príkladom control flow graphu je graf znázorňujúci metódu `foo()` (Obr. 2). V ľavej časti na Obr. 2 môžeme vidieť jednoduchú metódu `foo()`, ktorej vstupom je premenná `x` (označená ako source, zdroj) a výstupom je premenná `y` (označená ako sink, výstup). Táto metóda v tele obsahuje podmienkový príkaz a príkazy priradenia. Na pravej časti obrázku je znázornený control flow graf, vytvorený na dátovú reprezentáciu metódy `foo()`. Červené šípky v grafe označujú ohodnotenie jednotlivých hrán, ktoré je buď ε alebo true a false pri podmienkovom príkaze (kladná a záporná vetva podmienky).



Obr. 2 Metóda a k nej prislúchajúci control flow graph [12].

Taint analýza [12, 13, 14] priradí každému zdroju údajov, ktorý je považovaný za nedôveryhodný (napríklad neoverený vstup od používateľa), hodnotu „tainted“ (označený, poškvrnený). Yamaguchi a spol. [12] a Xu a spol. [13] v prácach popisujú,

ako taint analýza spracováva hodnotu tainted pri hľadaní zraniteľností. Túto hodnotu je vďaka jej označeniu ľahšie sledovať v toku procesov control flow graphu a vieme preto určiť, ktoré iné hodnoty ovplyvnila. Všetky hodnoty, ktoré prišli do styku s pôvodným nedôveryhodným zdrojom informácií, teda ktorých výpočet závisí na tomto údaji, sú takisto označené ako tainted. Každá implementácia taint analýzy obsahuje taktiež validačnú funkciu, ktorá môže hodnote odstrániť označenie tainted. Princípom taint analýzy je, že ak sa tainted hodnota dostane na výstup programu, tak je označená ako zraniteľnosť. Ako popisuje Chen a spol. [14], taint analýza nemusí byť vždy spoľahlivá. Pri tejto metóde môžu nastať dve chyby – hodnota je označená ako tainted, hoci označená byť nemala (**overtainting**) alebo hodnota označená nebola, hoci je ovplyvnená inou tainted hodnotou (**undertainting**).

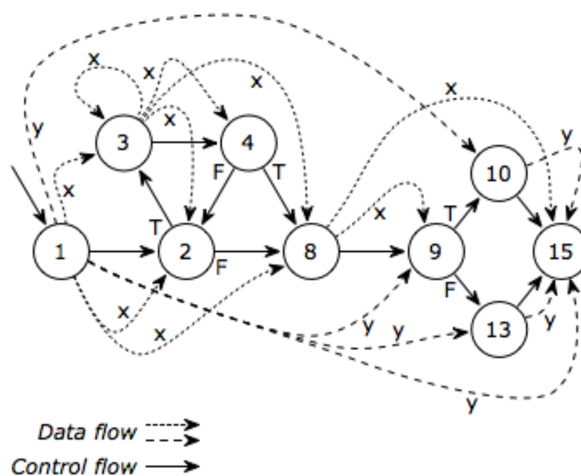
Kang a spol. [11] vo svojej práci predstavili vlastnú jedinečnú metódu statickej analýzy DTA++, založenú na taint analýze. Táto vylepšená technika dodatočne šíri označenie „tainted" pozdĺž vybranej podmnožiny riadiacich závislostí. Práca sa od iných líši v tom, že poskytuje nový pohľad na dynamickú taint analýzu. Ako autori píšu, najčastejším nedostatkom implementácie tejto metódy je, že nešíri označenie tainted pozdĺž riadiacich závislostí (control dependencies). Riadiaca závislosť je situácia, kedy sa príkaz v programe vykoná vtedy a len vtedy, ak sa nejaký konkrétny príkaz pred ním vykoná s očakávaným výstupom [15]. Platí to napríklad pre obe vetvy podmienkového príkazu, teda vetvy takéhoto príkazu sú riadiaco závislé (control dependent) na príkaze určujúcom podmienku. Riadiace závislosti v ponímaní autorov označujú také časti programu, kde dáta označené taint analýzou ovplyvňujú tok procesov, teda aj iné dáta. Tento vplyv môže viesť ku undertainting-u, čomu sa autori svojou metódou snažia predísť. Autori svoj nový algoritmus DTA++ implementovali pomocou platformy na binárnu analýzu BitBlaze. Svoju metódu otestovali na 8 aplikáciách pracujúcich s textami, medzi nimi napríklad Microsoft Word, WordPad, IntelliEdit, či AngelWriter. Ich metóda odhalila vo všetkých týchto aplikáciách neošetrené riadiace závislosti, ktorých úpravou sa podarilo predísť undertainting-u pri taint analýze.

Inou ukážkou využitia dynamickej taint analýzy je práca, **McClurg a spol.** [16]. Autori sa v tejto práci zamerali na testovanie aplikácií určených pre operačný systém Android. Otestovali 125 aplikácií napísaných v jazyku Java. Pri hľadaní zraniteľností sa zamerali na úniky citlivých údajov prostredníctvom SMS správ a internetového pripojenia pri používaní týchto aplikácií používateľmi. Implementovali teda desktopovú

aplikáciu v jazyku Java, ktorej princípom bola klasická taint analýza. Ich systém pri 2 testovacích aplikáciách odhalil, že umožňujú prístup k citlivým údajom používateľa bez nutnosti prihlásenia sa do zariadenia, na ktorom je aplikácia nainštalovaná. Autori týchto aplikácií boli na chybu upozornení a svoje aplikácie dodatočne opravili. V 8 ďalších aplikáciách sa im podarilo zachytiť vytvorenie spojenia medzi zariadením a internetom pri používaní daných aplikácií, avšak toto spojenie nebolo vytvorené za účelom odosielania citlivých údajov.

1.3.2 Data flow diagram

Diagram toku údajov (data flow diagram, DFD) je grafická reprezentácia toku údajov v programe. Na rozdiel od CFG sa už nezameriava na samotné procesy (Obr.3). Na obrázku je znázornený rozdiel medzi DFD (prerušované šípky) a CFG (plné šípky). Graf znázorňuje cyklus v programe, ktorý prechádza metódami označenými číslami. Označenia *x* a *y* pri prerušovaných šípkach označujú premenné, údaje, ktoré znázorňuje data flow diagram. Označenia *T* a *F* pri plných šípkach, označujú kladnú a zápornú vetvu (napríklad pri podmienkových príkazoch). Na obrázku môžeme vidieť, že data flow znázorňuje omnoho väčší tok, keďže sa zameriava na údaje, nielen na procesy.



Obr. 3 Rozdiel medzi data flow a control flow [17].

Pomocou data flow diagramu, môžeme podobne ako pri taint analýze sledovať, ktoré hodnoty programu sa navzájom ovplyvňujú. Túto vlastnosť tak vieme využiť pri statickej analýze, na zistenie pôvodu zraniteľnej časti kódu.

Vogt a spol. [18] spojili metódu využitia data flow diagramu s metódou taint analýzy a využili svoj systém na hľadanie zraniteľnosti cross-site scripting (XSS)

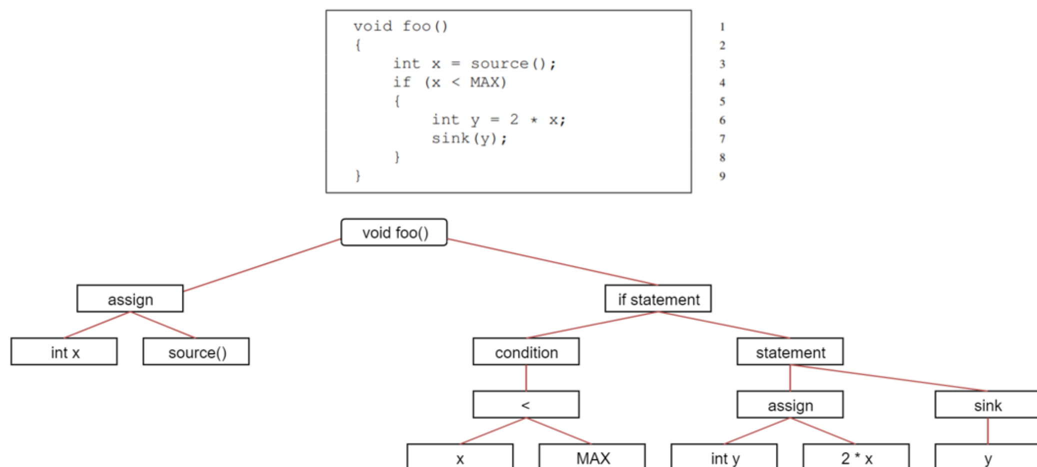
vo webových aplikáciách. **Cross-Site Scripting (XSS)** je metóda napadnutia webových aplikácií, spočívajúca vo vložení skriptovacieho kódu (teda kódu napísaného v skriptovacom jazyku) od útočníka do zdrojového kódu vygenerovaného webovou stránkou. Táto metóda využíva hlavne bezpečnostné chyby týkajúce sa neošetrených vstupov. Autori v práci navrhli mechanizmus schopný sledovať tok citlivých dát, napríklad cookies, používaných webovým prehliadačom. V tomto procese, rovnako ako pri postupe taint analýzy, sú citlivé dáta označené ako tainted a ak týmto dátam pristúpi skript bežiaci v prehliadači, ich použitie je dynamicky sledované systémom autorov. Označenie tainted následne pomáha sledovať tok potenciálne nebezpečných dát v procesoch webového prehliadača. Ak sa citlivé dáta majú odoslať tretej strane, je na používateľovi, či tento prenos umožní alebo nie. Autori implementáciu svojej metódy otestovali na troch rozšírených open source webových aplikáciách phpBB, myBB a WebCal. V práci sa uvádza, že systém vyprodukoval veľmi málo false positive zraniteľností, pričom tie ktoré nájdené boli, boli zapríčinené zberom informácií webovými stránkami za účelom evidovania návštevnosti stránok. Túto anomáliu autori hodnotia ako pozitívnu, keďže systém správne odhalil vyžiadaný prístup k citlivým údajom zo strany prehliadača.

Jones a Muchnick [19] vo svojej práci opisujú ako sa dá data flow diagram efektívne využiť na analýzu zdrojového kódu. V zdrojovom kóde sa zamerali hlavne na to, ako efektívne analyzovať rekurzívne metódy. Autori v práci popisujú, ako sa v rekurzívnej metóde dá pozrieť na všetky jej vnorené volania a následne kontrolovať vstupy týchto volaní. Systém, ktorý navrhli však neotestovali na žiadnej reálnej aplikácii, iba na pseudokódoch, ktoré simulujú správania rekurzívnych metód a ich volaní. Ako uvádzajú, tento všeobecný prístup zvolili preto, aby ich metóda bola aplikovateľná na rôzne programovacie jazyky. Autori v práci ďalej neuvádzajú, či sa im tento cieľ podarilo splniť.

1.3.3 Abstraktný syntaktický strom

Pojem **abstraktný syntaktický strom** (abstract syntax tree, AST), alebo krátko syntaktický strom, je označenie pre minimálnu stromovú reprezentáciu zdrojového kódu napísaného v ľubovoľnom programovacom jazyku [20]. Obr. 4 popisuje abstraktný syntaktický strom pre metódu `foo()`.

Táto dátová štruktúra reprezentuje zdrojový kód bez komentárov a je vytváraná podľa gramatiky príslušného programovacieho jazyka [21]. Zameriava sa skôr na pravidlá daného jazyka než na prvky, akými sú zátvorky či bodkočiarky, ktoré ukončujú vety v niektorých jazykoch. Abstraktné syntaktické stromy sa často využívajú v kompilátoroch na kontrolu správnosti kódu.



Obr. 4 Metóda a k nej prislúchajúci abstraktný syntaktický strom.

Každý uzol stromu označuje konštrukciu vyskytujúcu sa v zdrojovom kóde. Abstraktný syntaktický strom je „abstraktný“ v tom zmysle, že niektoré skutočné znaky používané v „konkrétnom“ programe sa neobjavujú v tomto strome, teda nereprezentuje každý detail, ktorý sa nachádza v skutočnej syntaxi. Vnútorné uzly takéhoto stromu sú operátory (napríklad binárne operátory pre sčítanie, násobenie, či príkazy priradenia, cyklov a podmienok) a jeho uzly sú operandy (napríklad premenné, konštanty).

Abstraktný syntaktický strom využili **Yamaguchi, Lottmann a Rieck** [22], ktorí navrhli metódu na pomoc bezpečnostným analytikom pri statickej analýze kódu. Vo svojej práci využívajú takzvanú extrapoláciu zraniteľností [23]. Táto metóda používa automatickú detekciu na preskúmanie využitia API (rozhranie pre programovanie aplikácií) v zdrojovom kóde a v tomto využití hľadá nejaké vzory. Príkladom na vzor využitia API je časté volanie deklarácie nejakej premennej, či volanie nejakej metódy. Následne, začínajúc zo známej zraniteľnosti, tieto vzory môžu byť v zdrojovom kóde odhalené a môžu viesť k nájdeniu potenciálne zraniteľnej časti kódu. S využitím myšlienky extrapolácie zraniteľností, metóda autorov pokračuje extrahovaním abstraktných syntaktických stromov zo zdrojového kódu. V týchto syntaktických

stromoch autori hľadajú štrukturálne vzory na základe toho, že každá funkcia v zdrojovom kóde sa dá popísať ako súbor vzorov vopred nájdených extrapoláciou zraniteľností. Najdôležitejšou časťou ich práce je identifikácia štrukturálnych vzorov. Na túto identifikáciu využívajú **techniku latentnej sémantickej analýzy** [24], ktorá sa používa na určenie podobnosti textov, berúc do úvahy ich význam. Je to klasická metóda na analýzu veľkého množstva textu, čo je aj prípad zdrojového kódu. Účinnosť navrhnutej finálnej metódy autori otestovali na štyroch známych open source projektoch LibTIFF, FFmpeg, Pidgin a Asterisk, v ktorých odhalili predtým nenájdené bezpečnostné zraniteľnosti.

Xie a Aiken [25] použili abstraktný syntaktický strom na hľadanie zraniteľností v skriptovacích jazykoch, konkrétne v jazyku PHP. Otestovali 6 open source aplikácií v jazyku PHP. Implementovali architektúru, ktorá zdrojový kód rozdelí do abstraktného syntaktického stromu a taktiež do control flow grafu, čím docielili pokrytie väčšieho množstva zraniteľností. Jednou z aplikácií, ktoré analyzovali bola aplikácia PHP-Fusion [26], čo je open source redakčný systém, ktorý využíva jazyk PHP a databázu MySQL. Autorom práce sa v tejto aplikácii podarilo nájsť 2 zraniteľnosti SQL Injection a niekoľko false positive zraniteľností. Využitím jednej z nich je možné manipulovať s dátami v databáze pod administrátorským prihlásením. Autori práce nájdené zraniteľnosti nahlásili vývojárom aplikácie PHP-Fusion, ktorí zraniteľnosti v aplikácii odstránili a následne vydali novú verziu danej aplikácie.

1.3.4 Code property graph

Grafovú reprezentáciu dát nazývanú **graf vlastností kódu** (code property graph, CPG) prvýkrát predstavil vo svojej práci Yamaguchi [12] v roku 2014 ako spojenie vlastností abstraktného syntaktického stromu, control flow grafu a grafu programových závislostí (program dependence graph, PDG) (Obr. 5). Obr. 5 znázorňuje metódu `foo()` a k nej prislúchajúci CPG. Zelené šípky v grafe reprezentujú hrany abstraktného syntaktického stromu, modré šípky hrany program dependence grafu a červené šípky sú hranami control flow grafu. Hrany CFG sú ohodnotené ε (ak hrana nie je ohodnotená) alebo *true* a *false* (podmienkové príkazy). Tento typ grafu je výhodný pri hľadaní zraniteľností pri statickej analýze, pretože má všetky vlastnosti AST, CFG a PDG. Dokáže teda tie isté zraniteľnosti, ktoré by odhalili tieto tri dátové štruktúry samostatne, avšak odhalí ich všetky naraz. Využitie vlastností AST mu umožňuje odhaliť

zraniteľnosti syntaktického pôvodu a vlastnosti CFG a PDG umožňujú code property grafu odhaliť zraniteľnosti dynamického charakteru.

```
void foo()
{
    int x = source();
    if (x < MAX)
    {
        int y = 2 * x;
        sink(y);
    }
}
```

Obr. 5 Metóda a k nej prislúchajúci code property graph [12].

Backes a spol. [27] navrhli spojenie Yamaguchiho CPG a takzvaného grafu volaní (call graph, CG) do jednotnej štruktúry. Graf volaní je reprezentácia control flow grafu, ktorá graficky znázorňuje volania medzi jednotlivými podprogramami počítačového programu. Ako v práci popisujú, toto vylepšenie viedlo k zvýšeniu pravdepodobnosti nájdenia zraniteľného miesta v programe. Svoju metódu navrhli na hľadanie zraniteľností ako na strane servera, tak aj na klientskej strane. Ich hlavným cieľom bolo navrhnúť kontrolu rozsiahlych PHP (Hypertext Preprocessor, skriptovací jazyk) projektov, ktorá by bola plne automatizovaná, teda žiadnu z transformácií na jednotlivé dátové štruktúry by nebolo potrebné vykonávať manuálne. Svoju metódu autori otestovali na viac než 1500 projektoch v jazyku PHP, ktoré si stiahli z webovej služby GitHub [28]. V práci autori uvádzajú že sa im lepšie darilo odhaliť syntaktické zraniteľnosti (napríklad SQL Injection s 25% úspešnosťou) než zraniteľnosti spojené s vlastnosťami webových aplikácií (napríklad XSS len s 4% úspešnosťou).

Yamaguchi a spol. [29] vo svojej práci využili na hľadanie zraniteľností iba code property graf. V praktickej časti tento graf otestovali hľadaním zraniteľností v zdrojovom kóde Linuxového jadra, čo je jadro open source operačného systému Linux. Tento zdrojový kód si vybral, lebo sa rozhodli testovať rozsiahle zdrojové kódy a svoj systém chceli odskúšať na čo najväčšom projekte. V prvej časti práce preskúmali, aké zraniteľnosti boli v tomto zdrojovom kóde objavené doteraz. Následne, v druhej časti

práce, popisujú svoju implementáciu code property grafu a testujú ju na spomínanom zdrojovom kóde. V kóde sa im podarilo objaviť 18 dovtedy nenájdených zraniteľností, ktoré boli následne potvrdené a opravené programátormi tohto zdrojového kódu. Z týchto zraniteľností sa väčšina týkala pretečenia zásobníka a chybne alokovanej, prípadne dealokovanej pamäte.

1.3.5 Zhodnotenie

Každá z prác, ktoré sme popísali v predchádzajúcich podkapitolách využívala iný spôsob statickej analýzy a mala svoje výhody aj nevýhody. Kang a spol. [11] pre hľadanie zraniteľností použili metódu taint analýzy a control flow grafu, pričom predstavili svoju vlastnú metódu DTA++. Táto upravená taint analýza má lepšie výsledky než klasická taint analýza, pretože poskytuje prevenciu proti undertainting-u. Autori však svoju metódu testovali len na desktopových aplikáciách (konkrétne na textových editoroch), čo je množina, ktorá nemusela poskytnúť dostatočne objektívne informácie.

McClurg a spol. [16] využili taint analýzu na hľadanie bezpečnostných chýb v Java aplikáciách určených pre operačný systém Android. Pomocou ich prístupu sa im podarilo odhaliť únik citlivých údajov. Ich systém môže používateľovi pomôcť odhaliť, či aplikácia ktorú používa je alebo nie je dostatočne zabezpečená. Veľkou výhodou ich implementácie je takisto možnosť aplikácie na rozsiahlych zdrojových kódoch a to nielen v jazyku Java.

Vogt a spol. [18] vo svojej práci využívajúcej data flow diagram poskytli výhodu, ktorú žiadna iná práca neposkytla, a to komunikáciu programu s používateľom. Metóda ktorú navrhli, produkuje malé množstvo false positive chýb, avšak je zameraná len na hľadanie jednej konkrétnej zraniteľnosti (cross-site scripting).

Vo svojej práci Jones a Muchnick [19] predstavili využitie data flow diagramu na hľadanie chýb pri používaní rekurzívnych metód v zdrojovom kóde. Ich metóda dokáže prehľadať pomocou data flow diagramu všetky volania danej rekurzívnej metódy a takisto kontrolovať správnosť parametrov daných vnorených metód. Veľkou nevýhodou tejto práce je, že nebola otestovaná na žiadnej reálnej aplikácii, iba na pseudokódoch. Hoci autori uvádzajú že ich metóda je tým pádom aplikovateľná na rôzne aplikácie a jazyky, toto tvrdenie nemusí byť pravdivé.

Yamaguchi, Lottmann a Rieck [22] využívajúc abstraktný syntaktický strom navrhli metódu, ktorá pracuje s rozsiahlymi zdrojovými kódmi. Je preto využiteľná v praxi, čo je veľkou výhodou tejto práce. Nevýhodou môže byť, že autori v práci neposkytujú automatizované riešenie, ale iba vylepšujú riešenie manuálne. Čo je však prekvapivé, ich metóda je nepoužiteľná bez vopred známej zraniteľnosti v kóde. Dôvodom je, že autori svoj algoritmus spúšťajú v tej časti zdrojového kódu, kde už bola nejaká zraniteľnosť nájdená. Hoci táto zraniteľnosť sa dá ľahko do zdrojového kódu doplniť, považujeme to za nevhodné riešenie hľadania zraniteľností.

Ďalšou prácou, ktorá využila abstraktný syntaktický strom je práca, ktorú napísali Xie a Aiken [25]. Svoju implementáciu otestovali na 6 open source PHP aplikáciách, pričom v jednej z nich sa im podarilo odhaliť zraniteľnosť SQL Injection. Veľkou výhodou tejto práce je otestovanie systému na reálnych aplikáciách a taktiež potvrdenie správnosti systému odhalením danej zraniteľnosti. Nevýhodou môže byť zameranie sa autorov len na skriptovacie jazyky. Pre tieto jazyky však otestovali a dokázali správnu funkčnosť ich metódy.

Metódu využívajúcu code property graf použili vo svojej práci Backes a spol. [27], ktorí sa zamerali na analýzu aplikácií v jazyku PHP. Výhodou tejto práce je automatická detekcia a parsovanie projektov v jazyku PHP z portálu GitHub [28]. Najväčšou nevýhodou je, že autori sa zamerali iba na hľadanie syntaktických zraniteľností.

Ďalšou z prác využívajúcich túto metódu je práca autorov Yamaguchi a spol. [29]. Veľkou výhodou tejto práce je schopnosť analyzovať rozsiahle kódy, čo autori odskúšali na zdrojovom kóde jadra operačného systému Linux. Ako je v práci popísané, nevýhodou tohto prístupu je hľadanie len tých zraniteľností, ktoré sú v zdrojovom kóde priamo viditeľné, namiesto tých, ktoré sa ukážu po spustení kódu. To však nepovažujeme za chybu implementácie autorov, pretože je to vlastnosť statickej analýzy kódu, ktorú využívajú.

Tab.1 Porovnanie prác využívajúcich rôzne metódy statickej analýzy

Metóda	Autori, spôsob statickej analýzy	Testované aplikácie	Úspešnosť
Abstraktný syntaktický strom	Yamaguchi, Lottmann a Rieck; Generalized Vulnerability Extrapolation	Desktopové aplikácie v jazyku C	Áno, rôzne zraniteľnosti v 3 aplikáciách (okrem aplikácie LibTIFF)
	Xie a Aiken; Abstraktný syntaktický strom a control flow graf	Webové aplikácie v jazyku PHP	Áno, odhalenie 105 predtým nenájdených zraniteľností
Code property graph	Backes a spol., Code property graf	Webové aplikácie v jazyku PHP	Áno, odhalenie zraniteľností XSS (menšia úspešnosť) aj SQL Injection
	Yamaguchi a spol.; Code property graf	Zdrojový kód Linuxového jadra v jazyku C	Áno, odhalenie 18 predtým nenájdených zraniteľností
Data flow diagram	Vogt a spol.; Dynamic Data Tainting a taint analýza	Webové aplikácie v jazyku PHP a Perl	Áno, cross-site scripting vo všetkých aplikáciách, ak sa tam nachádzal
	Jones a Muchnick; Data flow diagram	Pseudokódy vlastných vzorových aplikácií	Nepriamo, odhalením vzorov v pseudokódoch
Taint analýza, control flow graph	Kang a spol.; DTA++	Desktopové aplikácie v jazyku C++	Nepriamo, pomocou odhalenia under-taintingu
	McClurg a spol.; Dynamic Taint Analysis	Open source aplikácie pre Android v jazyku Java	Áno, odhalený možný únik citlivých údajov

2 Statická analýza v praxi

Pri testovaní zraniteľností aplikácií, teda pri statickej analýze kódu sa zväčša využívajú automatizované nástroje. Keďže týchto nástrojov je na trhu veľa, je na bezpečnostnom analytikovi, ktorý z nich pri analýze použije. V tejto kapitole si rozoberieme, podľa čoho sa dajú jednotlivé nástroje rozdeliť. Súčasne popíšeme postup a podmienky, podľa ktorých sme si vybrali nástroj na statickú analýzu my.

2.1 Požiadavky pre našu infraštruktúru

Praktická časť našej práce spočíva v testovaní zraniteľností vybraných aplikácií používaných na Univerzite Pavla Jozefa Šafárika v Košiciach. Tieto aplikácie sú v rôznych jazykoch, majú rozličnú funkcionalitu a bežia na rôznych operačných systémoch. Naším cieľom z tohto dôvodu bolo nájsť nástroj, ktorý nám bude vyhovovať v mnohých požiadavkách. Na naplnenie tohto cieľa sme stanovili nasledujúce požiadavky na nástroj:

- open source,
- spoľahlivosť existujúcej analýzy,
- možnosť dopĺňať vlastné pravidlá na analýzu,
- podpora operačných systémov Windows a Linux a
- podpora open source databáz.

Tieto požiadavky sú navrhnuté na základe toho, že hľadáme riešenie na testovanie aplikácií, ktoré by nadväzovalo na už existujúcu univerzitnú infraštruktúru. Open source nástroj a takisto podporu open source databáz požadujeme najmä z dôvodu finančnej dostupnosti a možnosti úprav nástroja pre analýzu. Spoľahlivosť existujúcej analýzy spočíva v hlavne vo vydávaní nových aktuálnych verzií, ktoré pokrývajú čo najväčšie množstvo známych zraniteľností. Za spoľahlivý nástroj taktiež považujeme taký nástroj, ktorý využíva spoľahlivé metódy. Podľa nášho prieskumu rôznych prác využívajúcich metódy statickej analýzy, považujeme za najspoľahlivejšie z nich metódy využívajúce control flow graf alebo abstraktný syntaktický strom. Takisto je pre nás dôležitá možnosť dopĺňania vlastných pravidiel na analýzu, aby sme si nástroj vedeli prispôsobiť vlastným špecifickým požiadavkám. Požiadavka na podporu operačných systémov Windows a Linux je spojená s infraštruktúrou univerzity, ktorá tieto dva operačné systémy používa.

2.2 Nástroje na statickú analýzu

Nástroje na testovanie zraniteľností aplikácií sa delia podľa rôznych kritérií [30]. Najdôležitejším z nich je prirodzene programovací jazyk, pre ktorý je daný nástroj určený. Ten môže byť buď jeden, alebo ich môže byť viacero. Iným kritériom je delenie podľa ceny nástrojov. Podľa tohto kritéria sa delia na nástroje s otvoreným kódom (open source), bezplatné (freeware, community edition), čiastočne platené (napr. základná verzia je bezplatná, platí sa len za doplnky) a komerčné. V našom prieskume sme sa zamerali na nástroje určené na statickú analýzu viacerých programovacích jazykov. Porovnanie siedmych odlišných nástrojov sme spracovali v tabuľke nižšie (Tab. 2). Nástroje, ktoré sme porovnávali, boli vyberané podľa rôznych, vopred stanovených kritérií. Pre rôznorodosť sme vyberali komerčné nástroje a aj nástroje s otvoreným kódom. Následne sme porovnali, v akých vlastnostiach sa líšia. Zamerali sme sa napríklad na to, či v danom nástroje je možné implementovať vlastné pravidlá. Ide o jednu z požiadaviek pre našu infraštruktúru. Tiež sme sledovali vydanie poslednej verzie, čo je dôležité pre aktuálnosť daného nástroja.

Tab. 2 Nástroje na statickú analýzu [31]

Nástroj	Open source	Dokumentácia	Vlastné pravidlá	Posledná verzia	Spôsob statickej analýzy	Požadované programovacie jazyky
AppChecker [32]	Nie	Nie	Nie	Neznáme	Regulárne výrazy	JavaScript nie
DevSkim [33]	Áno	Áno	Áno	0.1.19, marec 2018	Regulárne výrazy	Áno
Fortify [34]	Nie	Áno	Áno, ale len pre vlastnú inštanciu	17.2, november 2017	DFD	Áno
Checkmarx [35]	Nie	Áno	Áno	8.5, marec 2018	Rôzne grafy	Angular nie
Infer [36]	Áno	Áno	Áno	0.14, marec 2018	Separačná logika	Iba Java a C
Security Code Scan [37]	Áno	Nie	Nie	2.7, apríl 2018	Neznáme	Iba C#
SonarQube [1]	Áno	Áno	Áno	7.0, február 2018	AST	Áno

Cieľom porovnania vybraných nástrojov, bolo rozhodnúť ktorý z nich by bol najvhodnejší na použitie na našej univerzite. Po analýze jednotlivých nástrojov sme sa rozhodli pre open source nástroj **SonarQube**, ktorý popisujeme v nasledujúcej kapitole. Pre naše použitie bol tento nástroj najvhodnejší, pretože spĺňa všetky naše požiadavky. Dajú sa doň lokálne dopĺňať ľubovoľné špecifické pravidlá a je široko škálovateľný. Ďalšou výhodou SonarQube je, že pri analýze zdrojového kódu využíva abstraktný syntaktický strom, čím spĺňa našu požiadavku na spoľahlivosť existujúcej analýzy. Spoľahlivosť statickej analýzy pomocou abstraktného syntaktického stromu sme popísali pri prácach Yamaguchi, Lottmann a Rieck [22] a Xie, Aiken [25], kde sa ukázalo, že táto metóda má vysokú mieru úspešnosti.

3 Nástroj SonarQube

SonarQube [1] (predtým nazývaný Sonar) je open source nástroj pre priebežnú kontrolu kvality kódu na vykonávanie automatických kontrol statickou analýzou kódu. Tento nástroj slúži na detekciu chýb, „code smells“ (akýkoľvek príznak v kóde, ktorý môže naznačovať problém, avšak nie je to technická chyba) a bezpečnostných zraniteľností. Je navrhnutý pre viac ako 20 programovacích jazykov vrátane jazykov Java, C#, PHP, JavaScript, Python a mnoho ďalších. Túto detekciu vykonáva implementovanými pravidlami, ktoré sú napísané v jazyku Java. SonarQube ponúka správy o duplicitnom kóde, kódovacích štandardoch, UNIT testoch, zložitosti kódu, komentároch, chybách a bezpečnostných zraniteľnostiach. Navyše dokáže zaznamenávať históriu a poskytuje grafy vývoja testovaného projektu.

Najväčším prínosom SonarQube je, že poskytuje plne automatizovanú analýzu a integráciu s rôznymi nástrojmi. Tiež je možné ho integrovať s obľúbenými vývojovými prostrediami, ako je napríklad Eclipse, Visual Studio a IntelliJ IDEA prostredníctvom zásuvných modulov SonarLint. Keďže je tento nástroj open source, znamená to, že jeho zdrojové kódy sú verejne prístupné a koncoví používatelia majú právo ho voľne používať, modifikovať a šíriť. Tým sa ponúka možnosť implementovať do tohto nástroja vlastné pravidlá.

Na stránkach nástroja SonarQube je dôkladne vypracovaná podrobná dokumentácia [38] o tom, ako takéto pravidlá vytvárať, čo musia a čo nemôžu obsahovať. Súčasne je v nej popísané ako pravidlá implementovať do tohto nástroja. Cieľom našej práce je implementovať niektoré z týchto pravidiel pre taký jazyk a takú zraniteľnosť, ktoré ešte v nástroji SonarQube implementované nie sú.

3.1 Tvorba pravidiel v SonarQube

SonarQube po prihlásení sa do systému umožňuje používateľovi nechať si skontrolovať svoju aplikáciu, ak je napísaná v niektorom z podporovaných jazykov. Po kontrole nástrojom si používateľ vie prezrieť všetky detaily kontrol. Napríklad je možné si pozrieť údaj, na koľko percent považuje SonarQube podľa jeho kritérií aplikáciu za bezpečnú. Tiež je možné zobraziť počet a názvy zraniteľností, ktoré boli v aplikácii nájdené. Nástroj súčasne poskytuje vysvetlenie všetkých nájdených problémov, ktoré

pomáha pochopiť zistené problémy a vysvetľuje, prečo je daný problém rizikom pre aplikáciu.

Okrem testovania vlastnej aplikácie, SonarQube umožňuje prehliadanie všetkých predošlých aplikácií, ktoré boli testované inými používateľmi. To ale len v prípade, že títo nezakázali ich aplikácie zverejňovať. Používateľ si teda môže pomocou filtra vyhľadať, ako vyzerá aplikácia vo zvolenom jazyku, so zvolenou kvalitou a spĺňajúca špecifické vlastnosti, ktoré daného používateľa zaujímajú.

3.2 Využitie abstraktného syntaktického stromu

Vo všeobecnosti sa abstraktné syntaktické stromy používajú hlavne na sémantickú analýzu, čo je analýza postupne prechádzajúca symboly či skupiny symbolov získané zo syntaktického stromu priradzujúc im význam. Počas tejto analýzy kompilátor generuje tabuľku symbolov podľa syntaktického stromu a následný prechod stromom umožňuje overiť správnosť syntaktickej štruktúry programu.

V nástroji SonarQube sa abstraktné syntaktické stromy využívajú pri tvorbe pravidiel. Pred spustením nejakého pravidla v programovacom jazyku Java, nástroj SonarQube Java Analyzer analyzuje daný Java kód a vytvorí ekvivalentnú dátovú štruktúru, syntaktický strom. Každý typ konštrukcie jazyka Java môže byť reprezentovaná špecifickým druhom syntaktického stromu, ktorý podrobne opisuje každú z jeho vlastností. Pri analýze kódu, SonarQube Java Analyzer prechádza celým novovytvoreným stromom. Kódovacie pravidlo tu hrá úlohu návštevníka, ktorý je schopný navštíviť všetky uzly daného abstraktného syntaktického stromu. Akonáhle pravidlo pre zdrojový kód navštívi nejaký uzol, môže prechádzať strom v okolí tohto uzla a v prípade potreby zaznamenávať nájdené problémy.

3.3 Implementácia pravidiel

Implementácia vlastných pravidiel v nástroji SonarQube je vopred definovaná dokumentáciou [39]. Pravidlá sa do SonarQube môžu pridávať dvoma spôsobmi. My sme sa rozhodli pre písanie pravidiel v jazyku Java a ich následné pridanie pomocou SonarQube pluginu. Následne je nutné rozhodnúť sa pre aký jazyk bude dané pravidlo písané. My sme si vybrali jazyky PHP Java, v ktorých máme k dispozícii testovacie

aplikácie z Univerzity Pavla Jozefa Šafárika. Vyhľadávali sme zraniteľnosti zo zoznamu OWASP Top 10 [7].

SonarQube pri tvorbe abstraktného syntaktického stromu zo zdrojového kódu delí jednotlivé podstromy do viacerých kategórií. Tieto kategórie sa označujú **typy** a zodpovedajú štruktúre jednotlivých príkazov. Podmienkový príkaz, napríklad, má typ *IF_STATEMENT*, cyklus while má typ *WHILE_LOOP* a podobne. Pred každou implementáciou pravidla je teda nutné rozhodnúť sa podľa štruktúry hľadaného výrazu, ktoré kategórie budeme v zdrojovom kóde prehľadávať.

3.3.1 Pravidlo hľadajúce používanie zraniteľných komponentov

Ako prvé sme implementovali pravidlo pre **A9 – Using Components with Known Vulnerabilities**, teda používanie komponentov so známymi zraniteľnosťami, keďže o testovacej aplikácii vieme, že bola naprogramovaná v roku 2014. Existuje mnoho nástrojov, ktoré sa používajú pri vývoji aplikácií napriek tomu, že v nich boli nájdené zraniteľnosti. Dôvodom tohto problému je najčastejšie neaktuálnosť nástrojov a knižníc využívaných programátormi. Nie je vždy zvykom aktualizovať v aplikáciách komponenty, ktoré používajú. Je preto dôležité si vždy kontrolovať aktuálnosť aplikácií a sťahovať novšie, avšak stabilné verzie softvéru, používať aktuálne API metódy.

Pri implementácii pravidla na hľadanie komponentov so známymi zraniteľnosťami, sme sa nezamerali na aktuálnosť, ale na ich priame použitie v zdrojovom kóde. Keďže takéto nástroje sa v zdrojovom kóde volajú vo forme metód a funkcií, museli sme všetky takéto metódy v kóde vyhľadať. Pri implementácii pravidla sme sa zamerali na typ abstraktného syntaktického stromu *METHOD_INVOCATION* (volanie metódy). Je to spôsob ako sa v jazykoch PHP a Java volajú vopred definované metódy a funkcie. Toto volanie môže mať tvar *objekt.metóda* alebo *objekt = metóda(iný objekt).* Naše pravidlo teda prehľadáva všetky volania rôznych metód a funkcií v zdrojovom kóde a kontroluje, či niektorá z nich nie je metódou so známymi zraniteľnosťami.

Našou implementáciou sa podarilo v zdrojovom kóde testovacej aplikácie odhaliť použitie kryptografických hašovacích funkcií MD5 a SHA-1, ktoré sa v súčasnosti považujú za zraniteľné. Pre hašovaciu funkciu MD5 existujú známe CRF (collision resistance function) kolízie [40], teda sa nepovažuje za bezpečnú hašovaciu funkciu. Hašovacia funkcia SHA-1 (Secure Hash Algorithm 1) má takisto známe kolízne útoky od roku 2017 [41], avšak stále sa považuje za bezpečnú hašovaciu funkciu. Napriek tomu,

ak jej použitie nie je nevyhnutné, sa pri hašovaní odporúča používať iné funkcie [42]. Jazyk PHP, v ktorom je napísaná naša testovacia aplikácia, ponúka aj využitie iných hašovacích funkcií. Z tohto dôvodu je možné nepoužiť funkcie so známymi kolíznymi útokmi. Nasledujúce dva riadky zdrojového kódu, znázorňujú príklad použitia hašovacích funkcií MD5 a SHA-1 v testovacej aplikácii.

```
1. $passwordDB = sha1($data[0]['login'].'*'.$password);
2. $passwordDB = sha1($bForm->get('email')->getData()
                 .'*'
                 .md5(rand(1,9999).microtime()));
```

V jazyku Java naše pravidlo kontroluje použitie takýchto hašovacích funkcií pri volaní metódy `MessageDigest.getInstance`, ktorej parametrom sú práve hašovacie funkcie. Naše pravidlo je schopné odhaliť, ak funkcia v tomto parametri je jednou z funkcií, ktoré sa nepovažujú sa bezpečné hašovacie funkcie, ako napríklad funkcia MD5 z prechádzajúceho príkladu.

3.3.2 Pravidlo hľadajúce zraniteľnosť SQL Injection

Ďalším pravidlom, ktoré sme implementovali je hľadanie zraniteľnosti SQL Injection, jedna z popísaných v riziku **A1 – Injection** zo zoznamu OWASP Top 10. SQL Injection je zraniteľnosť, ktorej využitie môže umožniť odcudzenie citlivých údajov. Je založená na neošetrení vstupu od používateľa. Používateľ teda môže pomocou svojho vstupu manipulovať s dátami v databáze bez nutnosti vlastníctva prihlasovacích údajov.

Pri implementácii tohto pravidla sme sa v oboch jazykoch zamerali na kontrolu typov abstraktného syntaktického stromu, ktoré sa nazývajú *ASSIGNMENT* (priradenie), *PLUS_ASSIGNMENT* (priradenie znamienkom plus) a *PLUS* (konkatenácia). Priradenie má tvar *premenná = výraz*, priradenie znamienkom plus má tvar *premenná += výraz* a konkatenácia má tvar *premenná = výraz + výraz*. Po nájdení takéhoto typu sme kontrolovali ako vyzerá daný výraz, ktorý bol priradený do premennej. V prvom kroku sme overovali, či výraz obsahuje niektorý z príkazov jazyka SQL manipulujúcich s dátami [43] (*SELECT*, *DELETE*, *INSERT*, ...). Keď sme takýto príkaz našli, druhým krokom bola kontrola, či výraz obsahuje nejakú formu konkatenácie (napríklad pomocou znamienka plus a metódy `concat()` v jazyku Java, alebo pomocou znamienok plus

a bodka v jazyku PHP). Ak sme našli spojenie týchto dvoch vlastností, výraz sme označili ako zraniteľnosť. Príklad takéhoto výrazu v jazyku Java môžeme vidieť v riadku zdrojového kódu nižšie.

```
String query = "SELECT data FROM user_data WHERE user_name = " +
                request.getParameter("customerName");
```

Týmto pravidlom sa nám podarilo odhaliť výskyty v testovacom projekte v jazyku Java, ktoré naše pravidlo označilo ako zraniteľnosti. V testovacom projekte bola nájdené konkatenácia stringov (alebo stringových premenných) používaných na manipuláciu dát z databázy. Následná analýza daných riadkov zdrojového kódu však ukázala, že nájdené výskyty zraniteľnosťou neboli. V daných riadkov sa vyskytovala konkatenácia so stringovou premennou, ktorá pochádzala z parametra metódy. Táto premenná však bola do parametra danej metódy vložená z databázy. Z toho vyplýva že je to údaj pochádzajúci zo systému, nie vstup od používateľa. Z tohto dôvodu tieto konkatenácie nebudú viesť k zraniteľnosti SQL Injection. To či údaj je, alebo nie je vstupom od používateľa sa pri implementácii pravidiel v nástroji SonarQube nedá zistiť. Tieto pravidlá totiž kontrolujú samotný zdrojový kód a podľa syntaxe, či názvu premennej sa tento fakt nedá overiť.

3.3.3 Pravidlo hľadajúce zraniteľnosť XML External Entity Processing

Zraniteľnosť XML External Entity Processing (spracovanie externej entity XML) nadväzuje na riziko **A4 – XML External Entities (XXE)** zo zoznamu OWASP Top 10. XML External Entity útok je typ útoku zameraný na aplikácie, ktoré parsujú XML súbory [44]. Tento útok sa vyskytuje, ak XML vstup obsahujúci referenciu na nejakú externú entitu je spracovávaný zle nakonfigurovaným XML parserom. Využitie tejto zraniteľnosti môže viesť napríklad k odcudzeniu citlivých údajov alebo odopretiu služby.

Pri implementácii sme rovnako ako v kapitole 3.3.2 využili vyhľadávanie abstraktného syntaktického stromu typu *METHOD_INVOCATION*. Prevenciou proti riziku XXE je zakázanie deklarácie externých entít v XML vstupe. To sa v jazyku Java a takisto v jazyku PHP vykonáva volaním metód, ktoré sú na to určené. V jazyku Java je to metóda `setFeature` triedy `DocumentBuilderFactory` [45] a v jazyku PHP je to metóda `libxml_disable_entity_loader` [46], samozrejme obe musia mať správne

nastavené parametre. Príklady použitia týchto metód môžeme vidieť v zdrojovom kóde nižšie – príklad v jazyku Java v prvom a druhom riadku a v jazyku PHP v treťom riadku.

```
1. DocumentBuilderFactory dbf =
DocumentBuilderFactory.newInstance();
2. dbf.setFeature("http://xml.org/sax/features/external-general-
entities", false);
3. libxml_disable_entity_loader(true);
```

Tu však nastáva problém pri implementácii nášho pravidla. Pravidlá jazyka SonarQube sú totiž schopné vyhľadávať len chybnú implementáciu, nie tú chýbajúcu. Naše pravidlo teda dokáže detegovať, ak tieto metódy majú nesprávne nastavené parametre, ale nedokáže upozorniť používateľa, ak ich nepoužil vôbec. Toto je jedna z nevýhod implementácie pravidiel v SonarQube pomocou abstraktného syntaktického stromu. Pri implementácii tohto pravidla by bola výhodnejšia reprezentácia zdrojového kódu napríklad pomocou code property grafu, v ktorom by sme priamo vedeli skontrolovať, či bola daná metóda volaná alebo nie.

3.3.4 Pravidlo hľadajúce generovanie bezpečných náhodných čísel

Toto pravidlo priamo nenadväzuje na žiadne z rizík zo zoznamu OWASP Top 10. Avšak, generovanie náhodných čísel môžeme časti vidieť v zdrojových kódoch. Využitie je rôzne, či už na kryptografické účely (na generovanie kľúčov) alebo na jednoduchý výber náhodného indexu v poli.

V jazykoch Java aj PHP existujú rôzne metódy na vygenerovanie náhodného čísla. Najčastejšie používané z nich, nie sú však v dnešnej dobe považované za kryptograficky bezpečné. Dôvodom je, že ich výsledky sú predpovedateľné a teda nie je vhodné ich využívať napríklad na už spomínané generovanie kľúčov. Naše pravidlo teda vyhľadáva využitie takýchto metód a odporúča použitie tých, ktoré sú bezpečné. V jazyku PHP naše pravidlo neodporúča používať funkcie `rand()` a `mt_rand()` a odporúča ich zmeniť na `random_int()`, `random_bytes()`, alebo `openssl_random_pseudo_bytes()`. V jazyku Java pravidlo odporúča namiesto funkcie `Math.random()` a triedy `Random` použiť triedu `SecureRandom`.

Týmto pravidlom sa nám podarilo odhaliť chybu v testovacom projekte z našej univerzity. Na viacerých miestach v zdrojovom kóde aplikácie sa nášmu pravidlu podarilo nájsť použitie funkcie `rand()`. Namiesto tejto funkcie sa v zdrojovom kóde odporúča použiť napríklad funkciu `random_int()`, ktorá sa považuje za kryptograficky bezpečnú. V riadku zdrojového kódu nižšie môžeme vidieť konkrétne použitie v zdrojovom kóde testovacieho projektu. Úlohou funkcie `rand()` je v tomto prípade vygenerovanie nového náhodného hesla. To sa však v tomto prípade dá aj jednoduchšie, napríklad pomocou konkatenácie mena používateľa, jeho mailovej adresy a nejakého náhodného slova.

```
$passwordDB = sha1($bForm->get('email')->getData()
                .'*'
                .md5(rand(1,9999).microtime()));
```

3.4 Analýza testovacích projektov

Testovacie projekty sme otestovali aj pravidlami nástroja SonarQube, ktoré už sú implementované. V SonarQube je pre jazyk PHP implementovaných celkom 147 pravidiel. Pre jazyk Java je implementovaných 458 pravidiel.

Ako prvý sme analyzovali testovací projekt v jazyku PHP. SonarQube v tomto projekte odhalil 2 zraniteľnosti, 20 chýb a takmer 4000 code smells. V testovacom projekte v jazyku Java, SonarQube nenašiel ani jednu zraniteľnosť, ale našiel 4 chyby a 21 code smells. Rozdiel v týchto číslach je veľmi veľký aj vďaka faktu, že v jazyku PHP sme otestovali približne 52 000 riadkov zdrojového kódu, zatiaľ čo v jazyku Java sme mali k dispozícii na testovanie len približne 6 000 riadkov zdrojového kódu.

Väčšina nájdených chýb v jazyku PHP sa týkala uvádzania kódu po tom, čo bola metódou vrátená nejaká hodnota. Táto chyba nie je bezpečnostného charakteru, ale iba upozorňuje na zbytočné riadky v kóde, nie je teda prioritou ju upravovať. Zvyšné chyby sa týkali neinicializovaných premenných, čo taktiež nie je záležitosťou bezpečnosti. V Java projekte boli 2 zo 4 nájdených chýb funkcie `BigDecimal()`. SonarQube uvádza, že podľa JavaDocs [47], je táto funkcia nepresná a nepredvídateľná a preto sa odporúča namiesto nej použiť funkciu `BigDecimal.valueOf()`. Zvyšné dve chyby sa týkali neošetrenej výnimky a podmienky, ktorá bude stále pravdivá.

Ďalšími výskytmi v projektoch boli code smells, teda príznaky nájdené v zdrojovom kóde, ktoré však nie sú technickou chybou. Tie boli nájdené na mnohých riadkoch v zdrojovom kóde testovacej aplikácie v jazyku PHP. Väčšina z nich sa týkala zakomentovaných riadkov zdrojového kódu, rozdelenia príliš dlhých metód do viacerých a pridania kučeravých zátvoriek tam, kde chýbali. Všetky tieto príznaky nepredstavujú chyby, ale nástroj SonarQube ich odporúča opraviť, aby zdrojový kód mal prehľadnejšiu štruktúru. V jazyku Java sa väčšina týkala stringových premenných viacnásobne použitých v kóde, ktoré však neboli uložené do premennej ale písané pomocou úvodzoviek. SonarQube odporúča kvôli prehľadnosti kódu takéto viacnásobne použité výrazy uložiť do premennej.

Pri analýze projektu nás najviac zaujímalo, či SonarQube objaví v zdrojovom kóde zraniteľnosti. Výsledkom analýzy boli podľa SonarQube 2 zraniteľnosti v projekte v jazyku PHP, avšak po našej kontrole sme zistili, že tento výsledok bol chybný. Časti kódu, ktoré SonarQube označil ako zraniteľnosti, boli na rôznych miestach v aplikácii, avšak SonarQube ich označil ako tú istú zraniteľnosť. Jeho výstupom bolo odporúčanie, aby sa v kóde nenachádzali premenné, ktoré obsahujú vo svojom mene slovo **password** (heslo). Takéto premenné SonarQube vyhodnotil ako potenciálne napevno zapísané heslo v zdrojovom kóde, čo samozrejme označil ako zraniteľnosť **A1 – Injection** zo zoznamu OWASP Top 10. Avšak po preskúmaní konkrétnych častí kódu, sme zistili že sa nejedná o túto zraniteľnosť, keďže v aplikácii bola správne použitá metóda `bindParam()` [48] jazyka PHP. Táto metóda predchádza zraniteľnosti A1 – Injection a preto sme nájdenú zraniteľnosť vyhodnotili ako false positive. V nasledujúcom zdrojovom kóde je výsek zdrojového kódu v ktorom je použitá metóda `bindParam()` a premenná obsahujúca slovo password. Nasledujúci kód znázorňuje proces zmeny hesla používateľa v aplikácii a uloženie tejto zmeny do databázy. Heslo a identifikátor sa do databázy ukladajú bezpečnou metódou `bindParam()`.

```
1. $st = $this->sqlConnection->prepare('UPDATE `users` SET
2. password=:password WHERE id=:id');
3. $st->bindParam(':password', $password, \PDO::PARAM_STR);
4. $st->bindParam(':id', $userId, \PDO::PARAM_INT);
5. return $st->execute();
```

3.5 Analýza projektov zo služby GitHub

Pre dodatočné otestovanie našej implementácie, sme spustili analýzu v nástroji SonarQube, ktorá zahŕňala len testovanie pomocou našich 4 pravidiel. Analyzovali sme desať najobľúbenejších projektov v jazyku Java. Obľúbenosť projektov v službe GitHub sa dá zistiť pomocou počtu pridelených hviezdičiek, ktoré má daný projekt [49]. Touto selekciou sme vybrali 10 projektov, na ktorých sme následne v SonarQube vykonali analýzu zdrojového kódu. Podľa štatistík nástroja SonarQube sme v týchto desiatich projektoch otestovali dokopy 597 000 riadkov zdrojového kódu, z čoho približne 3 000 riadkov kódu bolo v značkovacom jazyku XML. Nami implementované pravidlá teda hľadali zraniteľnosti v 594 000 riadkoch zdrojového kódu napísaného v jazyku Java.

Našim pravidlám sa v desiatich analyzovaných projektoch podarilo nájsť dokopy 58 zraniteľných miest. Každé z týchto miest, bolo označené ako zraniteľné pravidlom hľadajúcim generovanie bezpečných náhodných čísel. Na každom riadku označenom nástrojom SonarQube sa teda používala buď funkcia `Math.random()` alebo trieda `Random()` jazyka Java. V mnohých prípadoch, ktoré sme preverovali sa tieto funkcie generovali v testovacích triedach pri potrebe vygenerovať náhodné číslo. Tento prípad nie je zraniteľnosťou v kóde, preto tieto chyby v projektoch nie je prioritou opraviť.

Zvyšné 3 pravidlá nenašli v otestovaných projektoch žiadne výskyty zraniteľných miest. Po následnej analýze zdrojových kódov projektov sme hľadali dôvod, prečo zvyšné pravidlá v projektoch nenašli žiadne chyby. Zistili sme, že napríklad pravidlo hľadajúce zraniteľnosť SQL Injection pravdepodobne nenašlo žiadne výskyty preto, že spolupráca s databázou sa vyskytovala len v malom počte tried testovaných projektov. Iným dôvodom samozrejme môže byť to, že tieto projekty žiadnu z nami hľadaných zraniteľností neobsahujú. Iné zraniteľnosti v testovaných projektoch by bolo možné nájsť analýzou všetkými pravidlami nástroja SonarQube. Naším cieľom však bolo otestovať pravidlá ktoré sme implementovali my, preto sme celkovú analýzu na týchto projektoch nespúšťali.

Tab.3 Počet nájdených zraniteľností v jednotlivých projektoch

Projekt	Počet nájdených zraniteľností
1.	1
2.	7
3.	13
4.	37
5. - 10.	0

Záver

Aplikácie ktoré každodenne používame, majú čoraz väčší prístup k našim citlivým údajom. V bankovej online aplikácii zapisujeme svoje číslo účtu, do sociálnych sietí si ukladáme svoje meno, priezvisko, telefónne číslo a mnoho iných informácií. Je preto dôležité, aby tieto programy, ktorým zverujeme naše osobné údaje, boli dobre zabezpečené voči možným útokom z internetu. V práci sme sa venovali metódam, ktoré sú schopné odhaliť slabé miesta aplikácií, vedia nájsť ich zraniteľnosti. Popísali sme, v čom sú jednotlivé metódy výhodné a aké sú ich nedostatky. Viac sme sa venovali statickej analýze, pri ktorej sme tiež popísali štyri najvyužívanejšie metódy, ktoré využíva. Pre našu prácu najvýznamnejší je postup využívajúci abstraktný syntaktický strom, stromovú reprezentáciu zdrojového kódu. Na splnenie cieľa našej práce sme si vybrali túto konkrétnu metódu statickej analýzy kódu, hlavne preto že je škálovateľná a prispôsobiteľná pre naše potreby.

Hlavným cieľom našej práce bolo implementovať pravidlá do open source nástroja SonarQube. Tým sme sa oboznámili s mnohými typmi zraniteľností aplikácií a tiež s metódami statickej analýzy, ktoré využíva. V práci sme rozoberali aj iné nástroje na statickú analýzu, pričom sme uviedli niekoľko dôvodov, prečo sme si pre naše potreby vybrali práve SonarQube. Uviedli sme tiež požiadavky pre naše riešenie, ktoré zabezpečili, aby nadväzovalo na už existujúcu infraštruktúru našej univerzity. Popísali sme výhody ale aj nevýhody vybratých open source a komerčných nástrojov. Najväčšou výhodou open source je samozrejme možnosť úpravy nástroja podľa vlastných požiadaviek. Komerčné nástroje, na druhej strane, môžu mať funkcionality, ktoré open source nástroje neponúkajú, napríklad časté a hlavne spoľahlivé aktualizácie pravidiel.

Sekundárnym cieľom takisto patriacim do praktickej časti práce bolo otestovanie aplikácií vytvorených na našej univerzite. Tieto aplikácie sme otestovali v už spomínanom nástroji SonarQube. Jedna z nich bola napísaná v skriptovacom jazyku PHP a druhá v jazyku Java. Následne sme rozobrali výsledky testovania. V poslednej kapitole sme popísali našu implementáciu pravidla do nástroja SonarQube a tiež výsledky, ktoré táto implementácia priniesla. Pomocou nášho nového pravidla sa nám podarilo odhaliť v testovacej aplikácii zraniteľnosť zo zoznamu OWASP Top 10, konkrétne zraniteľnosť A9 – Using Components with Known Vulnerabilities (používanie komponentov so známymi zraniteľnosťami) a taktiež použitie generátora náhodných čísel, ktorý sa nepovažuje za bezpečný.

V závere práce popisujeme otestovanie našich pravidiel na reálnych open source aplikáciách, ktoré nepochádzali z našej univerzity. Pomocou nástroja SonarQube a nami implementovaných pravidiel sme otestovali 10 najobľúbenejších projektov zo služby GitHub. Testovali sme aplikácie v jazyku Java. Naše pravidlá skontrolovali spolu 594 000 riadkov zdrojového kódu v týchto aplikáciách a podarilo sa nám odhaliť celkovo 58 zraniteľných miest v týchto zdrojových kódoch. Podrobné výsledky tejto analýzy popisujeme v predposlednej kapitole práce.

Zoznam použitej literatúry

1. SonarQube, 2015 [online] Dostupné z: https://www.sonarqube.org/

2. HOWARD, Michael, Steve LIPNER, 2006. The Security Development Lifecycle, Microsoft Press.

3. Microsoft, 2016 [online] Dostupné z: https://www.microsoft.com/en-us/SDL

4. SHASTRY, Bhargava, 2016 [online] Dostupné z: https://www.isti.tu-berlin.de/fileadmin/fg214/teaching/slides/SS16_YY_FuzzTesting.pdf

5. HUGHES, Jeff, George CYBENKO, 2014. Three Tenets for Secure Cyber-Physical System Design and Assessment. Dartmouth College.

6. O'SULLIVAN, Arthur, Steven M. SHEFFRIN, 2003. Economics: Principles in Action. Prentice Hall.

7. Open Web Application Security Project: OWASP Top Ten Project [online] Dostupné z: https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project

8. WEIDMAN, Georgia, 2014. Penetration Testing: A Hands-On Introduction to Hacking. No Starch Press, s. 1-9.

9. CHESS, Brian, Jacob, WEST, 2007. Secure Programming with Static Analysis, Pearson Education Inc.

10. BRUCKER, Achim D., Uwe SODAN, 2014. Deploying Static Application Security Testing on a Large Scale.

11. GYUNG KANG, Min, Stephen McCAMAN, Pongsin POOSANKAM, Dawn SONG, 2011. DTA++: Dynamic Taint Analysis w/ Targeted Control-Flow Propagation.

12. YAMAGUCHI, Fabian, Nico GOLDE, Daniel ARP, Konrad RIECK, 2014. Modelling and Discovering Vulnerabilities with Code Property Graphs.

13. XU, Wei, Sandeep BHATKAR, Ramachandran SEKAR, 2005. Practical Dynamic Taint Analysis for Countering Input Validation Attacks on Web Applications.

14. CHEN, Zhe, Xiao Juan WANG, Xin Xin ZHANG, 2011. Dynamic Taint Analysis with Control Flow Graph for Vulnerability Analysis.

15. Wikipédia, 2018 [online] Dostupné z: https://en.wikipedia.org/wiki/
Data_dependency

16. McCLURG, Jedidiah, Jonathan FRIEDMAN, William NG, 2012. Android Privacy Leak Detection via Dynamic Taint Analysis.

17. Julio Auto, 2017 [online] Dostupné z: http://www.julioauto.com/project/visual-data-tracer.html

18. VOGT, Philipp, Florian NENTWICH, Nenad JOVANOVIC, Engin KIRDA, Christopher KRUEGEL, Giovanni VIGNA, 2007. Cross-Site Scripting Prevention with Dynamic Data Tainting and Static Analysis.

19. JONES, Neil D., Steven S. MUCHNICK, 1982. A Flexible Approach to Interprocedural Data Flow Analysis and Programs with Recursive Data Structures.

20. JONES, Joel, 2003. Abstract Syntax Tree Implementation Idioms.

21. SLONNEGER, Kenneth, Barry L. KURTZ, 1995. Formal Syntax and Semantics of Programming Languages: A Laboratory Based Approach. Addison-Wesley Publishing Company.

22. YAMAGUCHI, Fabian, Markus LOTTMANN, Konrad RIECK, 2012. Generalized Vulnerability Extrapolation using Abstract Syntax Trees.

23. YAMAGUCGI, Fabian, Felix LINDNER, Konrad RIECK, 2011. Assisted Discovery of Vulnerabilities using Machine Learning.

24. DEERWESTER, Scott, Susan T. DUMAIS, Richard HASRSHMAN, 1990. Indexing by Latent Semantic Analysis.

25. XIE, Yichen, Alex AIKEN, 2006. Static Detection of Security Vulnerabilities in Scripting Languages.

26. PHP-Fusion, 2005 [online] Dostupné z: https://www.php-fusion.co.uk/home.php

27. BACKES, Michael, Konrad RIECK, Malte SKORUPPA, Ben STOCK, Fabian YAMAGUCHI, 2017. Efficient and Flexible Discovery of PHP Application Vulns.

28. GitHub, 2008 [online] Dostupné z: https://github.com/

29. YAMAGUCHI, Fabian, Nico GOLDE, Daniel ARP, Konrad RIECK, 2014. Modeling and Discovering Vulnerabilities with Code Property Graphs.

30. OWASP: Source Code Analysis Tools, 2018 [online] Dostupné z: https://www.owasp.org/index.php/Source_Code_Analysis_Tools

31. Awesome Static Analysis, 2017 [online] https://github.com/mre/awesome-static-analysis

32. AppChecker, 2016 [online] Dostupné z: https://npo-echelon.ru/en/solutions/appchecker.php

33. DevSkim, 2017 [online] Dostupné z: https://github.com/microsoft/devskim

34. Fortify, 2003 [online] Dostupné z: https://software.microfocus.com/en-us/products/static-code-analysis-sast/overview

35. Checkmarx, 2006 [online] Dostupné z: https://www.checkmarx.com/

36. Infer, 2015 [online] Dostupné z: https://github.com/facebook/infer

37. Security Code Scan, 2017 [online] https://security-code-scan.github.io/
38. GUMOWSKI, Michael, 2017. Writing Custom Java Rules 101 [online] Dostupné z: https://docs.sonarqube.org/display/PLUG/Writing+Custom+Java+Rules+101
39. Adding Coding Rules, 2018 [online] Dostupné z: https://docs.sonarqube.org/display/DEV/Adding+Coding+Rules
40. KUZNETSOV, Anton A., 2014. An algorithm for MD5 single-block collision attack using highperformance computing cluster.
41. SHAttered, 2017 [online] Dostupné z: https://shattered.io/
42. How secure is SHA1? 2017 [online] Dostupné z: https://crypto.stackexchange.com/questions/48289/how-secure-is-sha1-what-are-the-chances-of-a-real-exploit
43. Data Manipulation Statements, 2018 [online] Dostupné z: https://dev.mysql.com/doc/refman/5.7/en/sql-syntax-data-manipulation.html
44. XML External Entity (XXE) Processing, 2017 [online] Dostupné z: https://www.owasp.org/index.php/XML_External_Entity_(XXE)_Processing
45. DocumentBuilderFactory setFeature() Method, 2018 [online] Dostupné z: https://www.tutorialspoint.com/java/xml/javax_xml_parsers_documentbuilderfactory_setfeature.htm
46. PHP Manual: libxml_disable_entity_loader, 2018 [online] Dostupné z: http://php.net/manual/en/function.libxml-disable-entity-loader.php
47. JavaDocs: Class BigDecimal, 2018 [online] Dostupné z: https://docs.oracle.com/javase/7/docs/api/java/math/BigDecimal.html#BigDecimal(double)
48. PHP Manual: PDOStatement:bindParam, 2018 [online] Dostupné z: http://php.net/manual/en/pdostatement.bindparam.php
49. GitHub: About stars, 2018 [online] Dostupné z: https://help.github.com/articles/about-stars/

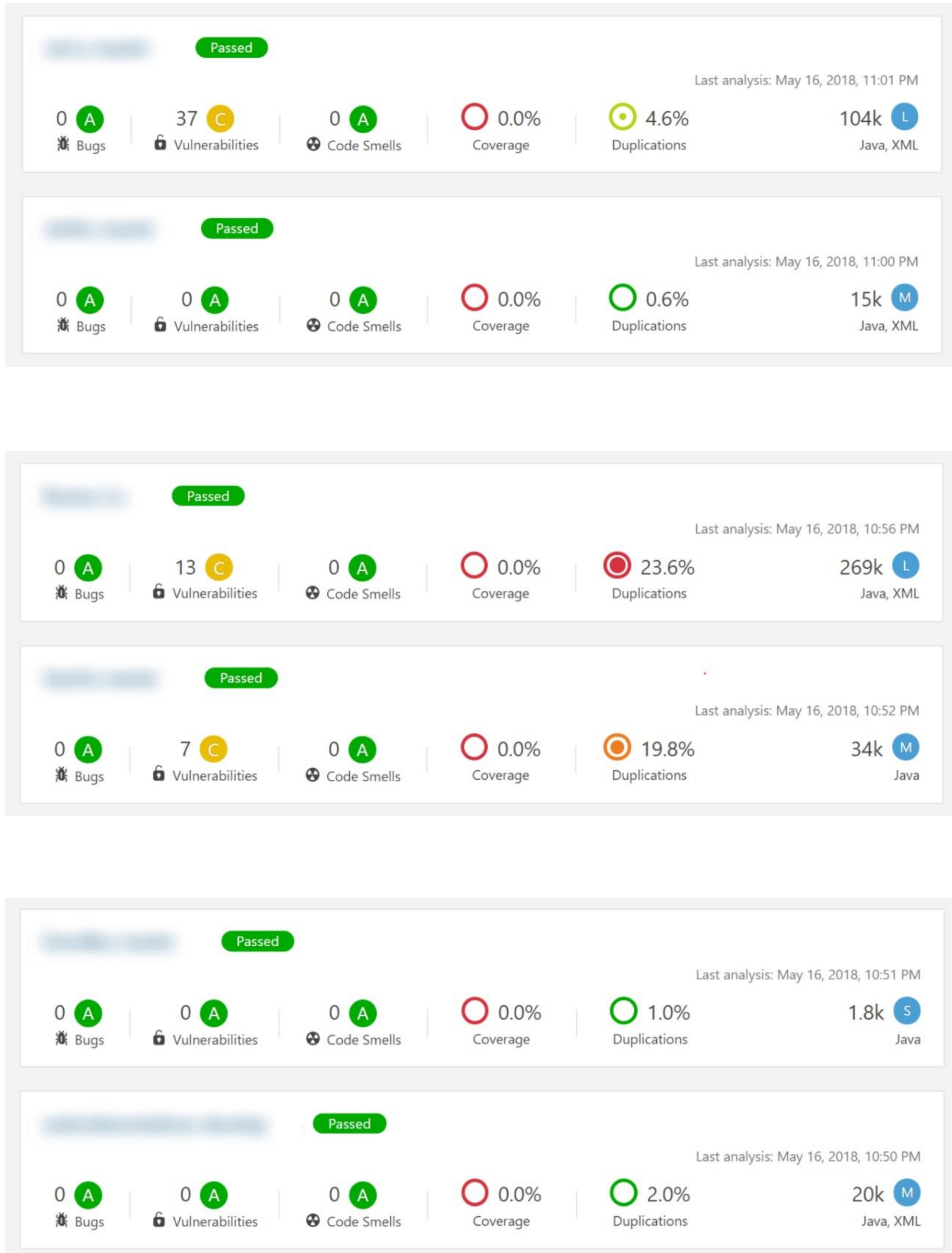
Prílohy

Príloha A: CD médium – bakalárska práca v elektronickej podobe, zdrojový kód pravidiel implementovaných v práci

Príloha B: Snímky obrazovky z analýzy open source projektov zo služby GitHub v nástroji SonarQube

Príloha C: Snímky obrazovky z analýzy testovacích projektov z našej univerzity

Príloha B: Snímky obrazovky z analýzy open source projektov zo služby GitHub v nástroji SonarQube



Passed

Last analysis: May 16, 2018, 5:05 PM

0 A Bugs | 0 A Vulnerabilities | 0 A Code Smells | 0.0% Coverage | 0.0% Duplications | 236 XS Java

Passed

Last analysis: May 16, 2018, 5:02 PM

0 A Bugs | 1 C Vulnerabilities | 15 A Code Smells | 0.0% Coverage | 2.2% Duplications | 135k L Java, XML, ...

Passed

Last analysis: May 16, 2018, 4:58 PM

0 A Bugs | 0 A Vulnerabilities | 0 A Code Smells | 0.0% Coverage | 0.0% Duplications | 1k S XML, Java

Passed

Last analysis: May 16, 2018, 4:56 PM

0 A Bugs | 0 A Vulnerabilities | 0 A Code Smells | 0.0% Coverage | 2.4% Duplications | 18k M Java, XML, ...

Príloha C: Snímky obrazovky z analýzy testovacích projektov z našej univerzity