Masarykova univerzita
Fakulta informatiky

Detekce bezpečnostních chyb pomocí statické analýzy kódu

Diplomová práce

Bc. David Formánek

Brno, jaro 2016

Prohlášení

Prohlašuji, že tato diplomová práce je mým původním autorským dílem, které jsem vypracoval samostatně. Všechny zdroje, prameny a literaturu, které jsem při vypracování používal nebo z nich čerpal, v práci řádně cituji s uvedením úplného odkazu na příslušný zdroj.

Bc. David Formánek

Vedoucí práce: RNDr. Andriy Stetsko, Ph.D.

Poděkování

Děkuji RNDr. Andriymu Stetskovi, Ph. D. za odborné vedení práce a společnosti Y Soft za možnost otestovat vyvíjený nástroj v reálném prostředí.

Shrnutí

Práce se zabývá automatickou statickou analýzou kódu a jejím využitím pro detekci třídy bezpečnostních zranitelností nazývaných jako injekce. Kromě nezbytné teorie je především popsána vlastní implementace nového mechanismu, který pomocí tzv. taint analýzy umožňuje tyto chyby spolehlivě detekovat v Java aplikacích. Tento mechanismus byl integrován do rozšíření FindSecurityBugs pro volně dostupný nástroj FindBugs. Pokrok v analýze injekčních chyb byl ověřen na testovací sadě Juliet – přesnost detekce vzrostla na 95 % oproti 53 % u původního nástroje a rozšíření.

Klíčová slova

bezpečnost software, statická analýza, taint analýza, injekce, FindBugs

Obsah

1 **Úvod** 1
2 **Bezpečnostní chyby typu injekce** 3
  2.1 *Klasifikace chyb a motivace pro detekci injekcí* 3
  2.2 *SQL injekce* 4
  2.3 *Injekce příkazu* 6
  2.4 *XSS* 7
  2.5 *Procházení adresářem* 9
  2.6 *LDAP injekce* 10
  2.7 *XPath injekce* 10
  2.8 *CRLF injekce* 10
  2.9 *Neošetřené přesměrování* 12
3 **Taint analýza** 13
  3.1 *Základní princip a terminologie* 13
  3.2 *Automatické hledání bezpečnostních chyb* 14
  3.3 *Taint zdroje* 16
  3.4 *Ošetření taint zdrojů* 18
  3.5 *Analýza toku dat a propagace stavu taint* 20
4 **Nástroj FindBugs a možnosti rozšíření** 25
  4.1 *Nástroj FindBugs* 25
  4.2 *Bytekód* 26
  4.3 *Tvorba vlastních detektorů* 30
5 **Implementace taint analýzy** 37
  5.1 *Model bezpečnosti dat* 37
  5.2 *Konfigurace metod a automatické odvozování* 42
  5.3 *Detektory využívající taint analýzu* 47
  5.4 *Vymezení rozsahu práce* 50
6 **Spuštění a vyhodnocení úspěšnosti detekce** 53
  6.1 *Konfigurace a spuštění* 53
  6.2 *Vyhodnocení na testovací sadě Juliet* 54
  6.3 *Limity současné analýzy* 60
7 **Závěr** 63
A **Seznam zneužitelných metod** 69
B **Seznam taint zdrojů** 79
C **Elektronické přílohy práce** 83

Seznam tabulek

5.1 Počty typů detekovaných volání pro injekční chyby *52*

6.1 Citlivost a specificita detekce injekčních chyb *56*
6.2 Srovnání celkové úspěšnosti detekce injekcí *57*
6.3 Schopnost analýzy toku dat v sadě Juliet *59*

1 Úvod

Hlavní důraz při vývoji software je zpravidla kladen na naplnění funkčních požadavků a testována je zejména správná odpověď systému pro předepsané vstupy. Velmi důležitou otázkou je nicméně to, jak systém zareaguje na nestandardní data a okolnosti. Pokud jí není věnována dostatečná pozornost, nejenže může být narušena spolehlivost, ale jedinec s dostatečnými znalostmi může být schopen využít systém způsobem, který nebyl zamýšlen.

Klasickým bezpečnostním chybám jako *přetečení zásobníku* sice brání použití jazyků s automatickou správou paměti, jiné chyby ale stejně tak mohou vést až ke kompletnímu převzetí systému, časté a přitom závažné jsou zejména tzv. *injekce*. Jedná se o třídu zranitelností, kde může útočník pomocí speciálně upraveného parametru změnit logiku vykonávaného dotazu či příkazu uvnitř aplikace. Práce se zabývá nejen typy chyb, které mají slovo injekce v názvu (např. *SQL injekce*), ale také dalšími problémy s obdobným principem zneužití (např. chyba zvaná *XSS*).

Protože manuální detekce chyb je velmi zdlouhavá a nákladná, i pro hledání bezpečnostních problémů je velmi výhodné použít automatické nástroje. Statickou a dynamickou analýzou pro vysokoúrovňové jazyky jsem se zabýval už v bakalářské práci [1], zde se soustředíme hlouběji na statickou analýzu a jazyk Java. Pro detekci chyb typu injekce a příbuzných zranitelností se nabízí použití tzv. *taint analýzy*. Při ní jsou v kódu vyhledána potenciálně zranitelná místa (volání dotazů a příkazů) a dále je prověřována existence neošetřených cest mezi těmito místy a nedůvěryhodnými vstupy programu. Přitom není nutné analyzovaný software spouštět a chyby jsou hledány napříč celou aplikací, takže nehrozí riziko vynechání zranitelnosti v nedostatečně otestované části systému (jako při dynamické analýze).

Pravděpodobně nejpraktičtějším volně dostupným statickým analyzátorem je nástroj FindBugs [2]. Jeho zaměření na bezpečnost je však pouze okrajové a odhalena je jen malá část skutečných chyb. Naopak jeho rozšíření FindSecurityBugs [3] cílí přímo na usnadnění bezpečnostního auditu aplikací, i zde však mnoho typů vážných zranitelností zůstalo nepokryto. Velmi nízká byla také jeho schopnost rozlišovat

mezi problémy skutečnými a pouze potenciálními, takže bylo třeba vždy vynaložit další úsilí na potvrzení reportovaných chyb.

Cílem práce je zvýšit množství detekovaných injekčních zranitelností rozšíření FindSecurityBugs a omezit reportování volání s bezpečnými parametry. Toho je dosaženo zejména náhradou jednoduché detekce konstant vlastní implementací taint analýzy. Dále je nutné projít Java API a často používané knihovny a identifikovat v nich nejen místa náchylná na zneužití injekce, ale také určit zdroje nedůvěryhodných či naopak bezpečných dat, metody pro konverzi nebezpečných vstupů pro každý z typů zranitelností a do analýzy zahrnout i způsob, jakým důvěryhodnost návratové hodnoty různých volání závisí na jejich vstupech. Pro ověření zlepšení a srovnání s původními nástroji použijeme rozsáhlou testovací sadu *Juliet* [4].

Následující kapitola je věnována samotným chybám, které se snažíme detekovat, a konkrétním způsobům zneužití. Další část obecně popisuje principy taint analýzy a zmiňuje zdroje nedůvěryhodných dat a jejich validaci. Kapitola 4 se věnuje především možnostem rozšiřování nástroje FindBugs a v kapitole 5 je popsán nově implementovaný mechanismus pro detekci injekcí. Následuje vyhodnocení úspěšnosti a srovnání s původními nástroji. Závěr obsahuje zhodnocení celé práce a diskutuje možnosti dalšího pokračování.

2 Bezpečnostní chyby typu injekce

Jako *injekce* můžeme označit takové typy zranitelností, kde je útočník schopen změnit příkaz (popř. dotaz či datovou strukturu) volaný uvnitř aplikace. [5] To může obecně nastat v případě, kdy je příkaz závislý na datech přicházejících od uživatele (resp. vstupech popsaných v sekci 3.3), ta nejsou dostatečně ošetřena (viz sekce 3.4) a zamýšlený datový vstup je interpretován jako část příkazu. Konkrétní typy zranitelností jsou popsány v sekcích 2.2–2.9. Uvedeny jsou i chyby, které se obvykle jako injekce neklasifikují, ale princip jejich vzniku a zneužití je totožný a použijeme stejnou metodu (popsanou v kapitole 3) pro jejich automatickou detekci. Pro zjednodušení budeme v celé práci pod injekcemi uvažovat i další typy zranitelností z této kapitoly.

2.1 Klasifikace chyb a motivace pro detekci injekcí

Bezpečnostní chyby lze kategorizovat a posuzovat podle různých kritérií, takovou klasifikaci provádějí například neziskové organizace *MITRE* a *OWASP*. Obě společnosti se rovněž snaží vytipovat nejdůležitější bezpečnostní chyby, kterých by se měli (nejen) vývojáři vyvarovat. Při tom se řídí jak možnými důsledky v případě zneužití, tak i frekvencí výskytu nebo složitostí útoku. Z jejich analýz (viz dále) lze vyčíst, že automatický nástroj, jehož cílem je zvýšení bezpečnosti kódu, by se měl zaměřovat právě na chyby typu injekce.

2.1.1 MITRE a CWE

Společnost MITRE mj. vytváří seznam bezpečnostních slabin zvaný *CWE (Common Weakness Enumeration)* [6], kde má každá chyba svůj číselný CWE identifikátor, popis, možné důsledky zneužití, způsoby obrany apod. Chyby nejsou rozděleny do disjunktních kategorií na jedné úrovni abstrakce, ale uspořádány do hierarchické struktury a obsahují odkazy na související chyby. Například injekce[1] s identifikátorem *CWE-74* je označena jako rodič některých chyb zmíněných

1. Plným názvem *Improper Neutralization of Special Elements in Output Used by a Downstream Component*

v následujících podkapitolách a jako rodič této chyby je naopak zmíněna nedostatečná validace dat (*CWE-20*).

MITRE ve spolupráci s institucí SANS vydala v roce 2011 seznam 25 nejnebezpečnějších chyb v software. [7] Na prvním, resp. druhém místě se nachází *SQL injekce*, resp. *injekce příkazu* (viz sekce 2.2 a 2.3). Třetí místo obsazuje klasická chyba *přetečení zásobníku (buffer overflow)*, která ale není relevantní pro vysokoúrovňové jazyky s automatickou správou paměti jako je Java. Na čtvrtém místě je *XSS* (popsané v sekci 2.4) a v seznamu se nachází také *procházení adresářem* a *neošetřené přesměrování* (viz sekce 2.5 a 2.9). Některé nebezpečné chyby nelze spolehlivě identifikovat automaticky – například na pátém a šestém místě jsou *chybějící autentizace* a *autorizace*, pro jejich detekci by ale automatický nástroj potřeboval informaci o tom, která data a operace jsou citlivé a jakým způsobem má být přístup k nim omezen.

2.1.2 OWASP

Projekt OWASP (*Open Web Application Security Project*) se zaměřuje na bezpečnost webových aplikací. Publikuje návody pro jejich vývoj a testování, vytváří vlastní volně dostupné nástroje a vydává (naposledy 2013) seznam deseti nejnebezpečnějších chyb. [8] Pro každou z nich uvádí možnosti detekce a prevence, ukázku útoku, kategorizaci a další reference. Na prvním místě jsou obecně injekce (jako jedna chyba) – kromě SQL injekce a injekce příkazu jsou jsem zařazeny také *LDAP injekce* a *XPath injekce* (viz sekce 2.6 a 2.7). XSS je zařazeno zvlášť na třetím místě a neošetřené přesměrování je na místě desátém. Ostatní chyby jsou spíše generické a většinou obtížně automaticky detekovatelné.

2.2 SQL injekce

Pro přístup do databáze se obvykle používá SQL[2], následující dotaz například zajistí vybrání takového řádku tabulky, kde hodnoty polí `username` a `password` odpovídají hodnotám zapsaným v apostrofech:

```
SELECT * FROM users WHERE
    username='Karel42' AND password='123456'
```

2. *Structured Query Language*, dotazovací jazyk pro správu dat relačních databází

Pokud aplikace hodnoty místo toho přijme od uživatele (např. z polí přihlašovacího formuláře), lze dotaz využít pro autentizaci registrovaných uživatelů pomocí hesla. Část zjednodušeného kódu v jazyce Java by mohla vypadat takto:

```
ResultSet result = sqlStatement.executeQuery(
    "SELECT * FROM users WHERE username='"
    + request.getParameter("username")
    + "' AND password='"
    + request.getParameter("password") + "'"
);
if (!result.next()) {
    System.out.println("Invalid user or password");
    return;
}
```

V tomto kódu nedochází k žádnému ošetření vstupu a pokud útočník místo skutečného (či domnělého) hesla vyplní například řetězec `' OR ''='`, do databáze se pošle následující dotaz:

```
SELECT * FROM users WHERE
    username='Karel42' AND password='' OR ''=''
```

Protože výraz `''=''` je vždy pravdivý (rovnost dvou prázdných řetězců), autentizace proběhne i bez znalosti hesla. Kvůli tomu, že operátor `OR` má obvykle nižší prioritu než `AND`, může být přihlášen jiný než očekávaný uživatel, rozvinutím dotazu lze ale provést přihlášení za libovolného uživatele.[3] [9] Ještě jednodušší může být vložení znaků pro komentář (`#` či `--`) za znak `'`, takže zbytek dotazu bude ignorován.

Nejenže lze výše uvedenými způsoby obejít mechanismus autentizace, ale pomocí konstrukcí s výrazy `UNION SELECT` nebo `EXISTS` a operátoru `LIKE` či funkcí dostupných v daném dialektu jazyka SQL je možné vyextrahovat obsah tabulky uživatelů i celé databáze. To lze pomocí techniky *blind SQL injection* [10] množstvím dotazů učinit pouze na základě binární odpovědi systému (úspěšná a neúspěšná autentizace) a to dokonce i pokud z odpovědi serveru není vidět, kolik měl uvnitř volaný dotaz výsledků – využívá se časově nároč-

3. V případě, že by část dotazu s ověřením hesla byla uzávorkovaná, lze vyplnit řetězcem `' OR ''='` i hodnotu `username` a odstranit působnost podmínky bez ohledu na prioritu operací

ných funkcí v dotazech a následné analýzy doby odpovědi serveru. V případě zneužití aktualizačních dotazů lze data také měnit či mazat a v závislosti na systému je někdy možné zcela převzít kontrolu nad serverem, kde je webová aplikace spuštěna. Konkrétní postupy závisí na použitých technologiích a jejich popis je nad rámec této práce, s využitím automatických nástrojů jako např. *sqlmap* [11] lze ale chyby zneužít s relativně malým množstvím úsilí a znalostí.

Při kontrole výskytu této chyby je nutné ověřit všechna místa, kde lze nastavit SQL dotaz posílaný databázi. Kromě aplikačního rozhraní *JDBC*[4] dostupného ve standardní verzi jazyka Java je třeba uvážit specifikace *JDO*[5] a *JPA*[6] či známé knihovny *Spring* [12] a *Hibernate* [13]. Seznam volání pro kontrolu je uveden v sekci A.1 na straně 69. V JPA a Hibernate se nepoužívá přímo SQL, ale dotazovací jazyk s velmi podobnou syntaxí. Účinnou obranou před SQL injekcí je oddělení dotazu od dat pomocí tzv. *prepared statements*, i tak ale může být uživatelský vstup použit v připraveném dotazu (např. pokud mají být výsledky uspořádány podle daného parametru) a pak bude nutné provést validaci dat i zde.

2.3 Injekce příkazu

Některé aplikace pro svou funkcionalitu vyžadují volání externích příkazů (např. příkazový řádek operačního systému). Pokud je součástí volání také parametr pocházející od uživatele a není dostatečně validován, může dojít ke zneužití pomocí tzv. *Command injection*. [14] V důsledku může být zavolán jiný příkaz, než očekával autor aplikace, případně může útočník jako data vložit znak oddělovače (např. `;` nebo `&&`) následovaný dalším příkazem.

Uvažme například aplikaci, která by chtěla pro zjištění souborů v adresáři daného parametrem použít systémový příkaz (v systému Windows):

```
Process dirProcess = Runtime.getRuntime()
    .exec("cmd /C dir " + parameterDir);
// ... = dirProcess.getInputStream()
```

4. *Java Database Connectivity*, rozhraní pro práci klienta s databází
5. *Java Data Objects*, specifikace pro persistenci dat pomocí jednoduchých tříd
6. *Java Persistence API*, specifikace rozhraní pro práci s relačními daty

Útočník může jako parametr odeslat např. `adr && echo byl jsem zde > vzkaz.txt`, takže aplikace bude fungovat stejně jako s parametrem `adr`, ale jako vedlejší efekt dojde k vytvoření (přepsání) souboru v aktuálním adresáři. Takto je možné vykonat libovolný příkaz na vzdáleném serveru nebo útok zneužít pro eskalaci uživatelských práv na lokálním systému – vložený příkaz je vykonán s právy spuštěné aplikace.

Kromě metody `exec` je možné příkaz vykonat i voláním konstruktoru třídy `ProcessBuilder` či její metody `command`. Ideální obranou je vůbec systémová volání nevyužívat (např. v našem případě využít třídy `File`), jinak je nutné kontrolovat uživatelské vstupy (ideálně omezit na množinu předem daných hodnot), jako další vrstvu obrany je vždy dobré nastavit přístupová práva software na nutné minimum.

2.4 XSS

XSS neboli *Cross-site scripting*[7] je významná a specifická chyba, kterou i OWASP vyčleňuje mimo injekční zranitelnosti. Svým charakterem ale odpovídá injekci pro jazyk *HTML* [8], kde interpretem je přímo internetový prohlížeč návštěvníka zranitelné webové aplikace. [15] Problém nastane, pokud v sobě dynamicky generovaná stránka obsahuje neošetřený vstup od uživatele. Útočník tak může do stránky vložit nejen data, ale i HTML kód a pro následné zneužití klientský škodlivý kód (skript), obvykle v jazyce JavaScript. Přímo ohrožena není samotná aplikace, ale uživatel, který útočníkem změněnou stránku navštíví. Vložením útočníkova skriptu do stránky dojde k porušení tzv. *same-origin policy* prohlížeče a útočník získá přístup ke všem zdrojům, ke kterým může přistupovat původní stránka a její uživatelé. Lze tak libovolně číst a měnit obsah stránky (a např. vylákat citlivé údaje), vykonávat akce jménem přihlášeného uživatele, číst tzv. *cookies* (pokud nemají příznak *HttpOnly*) či podnikat další útoky z prohlížeče uživatele. Podle způsobu vložení skriptu do stránky rozlišujeme 3 typy XSS:

- Při *reflected XSS* (též *XSS typu 1*) pochází vstup přímo z HTTP požadavku a je vepsán bez uložení do HTTP odpovědi (např.

7. Někdy též *CSS*, lze přeložit jako *skriptování napříč stránkou*
8. *HyperText Markup Language*, značkovací jazyk pro tvorbu webových stránek

pokud stránka některé předané parametry zároveň zobrazuje). Útočník proto musí nejprve uživatele přimět navštívit speciálně upravenou adresu obsahující skript, který se pak v prohlížeči oběti spustí (popř. navštívit stránku, která teprve provede přesměrování).

- Naopak při *stored XSS (typ 2)* dochází k uložení útočníkova skriptu na server s aplikací a automatickému spuštění v případě návštěvy dané stránky. Příkladem může být diskusní fórum – pokud není správně ošetřen formulář pro odeslání příspěvku, je i útočníkův skript uložen např. do databáze a pokud ani při načítání existujících příspěvků nejsou odstraněny problematické znaky, skript bude spuštěn v prohlížeči každého uživatele, kterému bude útočníkův příspěvek zobrazen.
- Pojem *DOM-based XSS (typ 0)* se vztahuje zejména na moderní webové aplikace, kde se významná část aplikační logiky odehrává na straně klienta. V tomto případě se data často zpracují a zobrazí na stránce, aniž by došlo ke komunikaci se serverem. Jinak je princip stejný jako u typu *reflected*, skript může být obsažen v adrese za znakem `#`.

Dále lze XSS rozlišovat podle kontextu, do kterého se vstupní data vkládají. Podle toho vypadá i úsek HTML kódu, který útočník může použít pro vložení skriptu. V nejjednodušším případě jsou vkládány parametry přímo do těla HTML stránky – pak útočníkovi stačí vložit např. řetězec `<script>alert(1)</script>` pro otestování existence zranitelnosti a nahradit příkaz `alert` (který jen zobrazí dialogové okno) skriptem s požadovanou funkcionalitou pro zneužití. Aplikace ale také např. může umožnit uživatelům vkládat externí obrázky určením HTML atributu `src` u tagu `img` – pokud pak útočník vloží místo odkazu např. řetězec `x"onerror="alert(1)`, výsledný HTML kód může vypadat takto:

```
<img src="x" onerror="alert(1)" />
```

Ten způsobí pokus o načtení obrázku z neexistující adresy, vyvolání události `onerror` a spuštění kódu ve stejnojmenném atributu. Existuje velké množství způsobů, jak škodlivý skript spustit pro různé kontexty a jak výsledek zakódovat, aby obešel nesystematické způsoby obrany.

Je proto důležité převést všechny speciální znaky pro daný kontext na HTML entity (např. znak < nahradit sekvencí `<`), aby ztratily svůj speciální význam a byly vykresleny jako prostý text, více informací je uvedeno v podsekci 3.4.1.

Hledání XSS v kódu může být problematičtější než u jiných chyb, protože dynamická webová stránka je často generována s využitím šablon a identifikace zranitelného místa není tak přímočará. Při využití technologie *JSP*[9] se každá stránka nejprve převede na spustitelnou třídu (tzv. *servlet*), její metody pro zápis parametrů do stránky jsou uvedeny v sekci A.3.

2.5 Procházení adresářem

Uvažme aplikaci, která umožňuje přístup k souborům v předem daném adresáři. Pokud je cesta k souboru vytvářena zřetězením cesty k adresáři a neošetřeným názvem souboru od uživatele, může útočník pomocí sekvencí `../` přejít do nadřazených adresářů a pak uvést cestu k libovolnému souboru. [16] Tímto způsobem se lze dostat k souborům, ke kterým by normálně uživatel neměl přístup, např. zřetězení cesty `/data/public/` a vstupu `../../etc/passwd` se vyhodnotí jako `/etc/passwd` a může být přečten (popř. modifikován) soubor s přístupovými údaji.

Popsaná chyba se někdy blíže specifikuje jako *Relative path traversal*. Pak *Absolute path traversal* značí variantu, kdy nedochází ke zřetězení cest, ale je použita přímo hodnota od uživatele. V takovém případě aplikace může předpokládat určení cesty k souboru relativně k aktuálnímu adresáři, ale útočník může specifikovat přímo absolutní cestu k souboru (např. `/etc/passwd`). Útoky mohou být kombinovány i se zneužitím dalších zranitelností – pokud například webová aplikace umožňuje nahrát vlastní soubor do úložiště, lze se později odkázat na takový HTML soubor a jeho načtením vyvolat XSS (viz sekce 2.4). Při hledání této chyby v kódu je nutné zejména zkontrolovat ošetření řetězců, které jsou parametrem konstruktorů tříd pro práci se soubory, jejich výčet je obsažen v sekci A.4.

9. *JavaServer Pages*, technologie pro dynamické generování webových stránek pomocí kombinace HTML šablon a kódu jazyka Java

2.6 LDAP injekce

LDAP[10] je protokol pro síťový přístup k adresářovému serveru (kde je spuštěna adresářová služba, např. *Microsoft Active Directory*). Pokud požadavek na server obsahuje neošetřený uživatelský vstup, útočník je schopen měnit LDAP dotazy podobně jako SQL dotazy u SQL injekce (viz sekce 2.2). Může tak obejít autentizaci, dostat se k citlivým informacím nebo i jejich změně. [17] Metody, pro které je nutné parametry validovat, jsou uvedeny v sekci A.5.

2.7 XPath injekce

Jazyk *XPath* umožňuje snadný výběr (vyhledání) dat v *XML*[11] dokumentech, např. výraz `//uzly/uzel[@atr="x"]/text()` vrátí textový obsah všech uzlů s názvem `uzel`, které se nacházejí v XML tagu `uzly` a zároveň mají atribut `atr` s hodnotou `x`. Obsahuje-li XPath dotaz neošetřená data pocházející od uživatele (např. místo konstanty `x` je použita hodnota z formulářového pole), útočník může modifikovat dotaz a získat přístup k jiné než zamýšlené části dokumentu nebo změnit aplikační logiku. [18] Pokud je např. pro ověření přístupu využita XML databáze s uživatelskými jmény a hesly, lze využít operátor `or` a vždy pravdivého výrazu pro obejití autentizace podobně jako u SQL injekce (viz sekce 2.2). Pro detekci injekce je nutné zkontrolovat metody `compile` a `evaluate` ze třídy `XPath` balíku `javax.xml.xpath` a všechny XPath volání s využitím externích knihoven, např. od nadace *Apache*, seznam je uveden v sekci A.6.

2.8 CRLF injekce

Jako oddělovač řádků se v závislosti na platformě využívají kontrolní znaky `CR` (*carriage return, \r*), `LF` (*line feed, \n*), nebo kombinace obou. Pokud aplikace zapisuje data s neošetřenými částmi od uživatele do řádku, může jí útočník předat znaky `CR` nebo `LF` a zbytek řetězce bude interpretován jako nový řádek. [19] Ukážeme dva konkrétnější typy této chyby, které mohou být zneužity.

---

10. *Lightweight Directory Access Protocol*
11. *Extensible Markup Language* , obecný značkovací jazyk pro výměnu dat

2.8.1 Falšování záznamů logu

Aplikace běžně ukládají do logu výjimečné i předpokládané stavy systému pro případnou pozdější analýzu. Jednotlivé záznamy se obvykle skládají z časového razítka, názvu komponenty provádějící zápis, určení závažnosti a samotné zprávy. Pokud zpráva obsahuje neošetřená data od uživatele, útočník může za znaky nového řádku vložit řetězec stejného tvaru jako skutečné záznamy a díky tomu je schopen uložit falešné záznamy s libovolným obsahem. Takto lze manipulovat se statistikami a například odvést pozornost od skutečného útoku či přímo zavést podezření na třetí stranu. [20]

V případě, že záznamy jsou dále strojově zpracovávány, lze také zneužít chyby v software, který je bude interpretovat. Zde lze naopak využít záměrně špatný formát záznamu, který způsobí chybu při čtení nebo dokonce povede ke spuštění útočníkovi kódu – např. injekce příkazu (viz sekce 2.3) v případě zpracování záznamů přes systémový nástroj nebo XSS (viz sekce 2.4) při použití webového nástroje pro analýzu. Pro detekci chyby je nutné zkontrolovat volání mechanismu záznamu do logu, který aplikace používá, kromě použití výchozí třídy `Logger` z balíku `java.util.logging` to mohou být např. volání knihoven *Apache Commons Logging*, *SLF4J*, *log4j* nebo *tinylog*, seznam metod je uveden v sekci A.7.

2.8.2 Rozdělování HTTP odpovědi

Když server posílá klientovi odpověď přes *HTTP*[12] a obsahuje neošetřená data se znaky `CR` či `LF` a identifikátorem nové HTTP hlavičky (např. `HTTP/1.1 200 OK`), může dojít k chybě známé jako *HTTP response splitting*. [21] Pak by byla odpověď serveru rozdělena na dvě samostatné, kde druhá z nich je zcela pod kontrolou útočníka. Za správných okolností tak může být uživateli podstrčen libovolný obsah a důsledky jsou podobné jako u XSS (viz sekce 2.4). Aby byl útok úspěšný, musí být kromě zranitelného kódu použita i zranitelná platforma umožňující vkládání znaků `CR` a `LF`. Moderní aplikační servery Java EE by touto chybou již neměly být ovlivněny, [22] přesto je vhodné kód zabezpečit bez ohledu na prostředí, ve kterém bude spuštěn.

12. *Hypertext Transfer Protocol*, hlavní internetový protokol aplikační vrstvy

2.9 Neošetřené přesměrování

Webové aplikace často obsahují přesměrování na jinou stránku, chyba vznikne, pokud je cíl přesměrován na libovolnou absolutní adresu danou parametrem (tzv. *Open redirect*). [23] Spíše než o injekci se jedná přímo o použití nedůvěryhodného parametru jako adresy. To sice nepředstavuje přímé riziko pro aplikaci, ale důvěryhodnost webu je zneužita pro usnadnění útoku na uživatele. Útočníci se často snaží přimět oběť přistoupit na falešnou stránku, která vzhledem imituje původní web a snaží odcizit např. přihlašovací údaje (tzv. *phishing*). Může také přímo obsahovat skript, který se pokusí instalovat škodlivý software do počítače oběti. Zranitelný kód vypadá např. takto:

```
response.sendRedirect(request.getParameter("url"));
```

Útočník pak může použít adresu jako

```
http://banka.cz/redir.php?url=http://f4k3b4nk.ru
```

Obezřetný uživatel před kliknutím na odkaz zkontroluje doménu stránky, zda nevypadá podezřele, neošetřené přesměrování ale umožňuje použít důvěryhodnou doménu, ze které je oběť však ihned přesměrována na stránky útočníka. Kromě metody `sendRedirect` (třídy `HttpServletResponse`) je třeba zkontrolovat také nastavení hlavičky *Location* (metody `addHeader` a `setHeader`).

OWASP do této kategorie chyb řadí i tzv. neošetřený *forward* [24] – zde parametr obsahuje naopak název lokální podstránky (např. pro technologii JSP), která je použita pro vygenerování aktuální stránky (bez přesměrování). Chyba nastává, pokud je umožněno použít i podstránku, ke které by aktuálně přihlášený uživatel neměl mít přístup, jedná se tedy o chybu autorizace.

3 Taint analýza

Pro detekci chyb z kapitoly 2 lze využít tzv. *taint analýzu*, díky níž lze určit, která místa jsou ovlivněna nedůvěryhodnými vstupy. Tato kapitola uvádí její obecné principy a srovnává ji s jinými metodami testování bezpečnosti software.

3.1 Základní princip a terminologie

Každá prakticky použitelná aplikace načítá při svém běhu nějaké vstupy. Pokud je takový vstup závislý na hodnotě od uživatele nebo komponentě, jejíž výstupy mohou být nepředvídatelné, budeme ho označovat jako *taint zdroj*[1]. Naopak bezpečné jsou konstanty uvnitř aplikace a z našeho pohledu např. také numerické hodnoty (s libovolným původem) převedené na řetězce. Problémy s neošetřenými uživatelskými vstupy z kapitoly 2 se ve skutečnosti vztahují i na všechny neošetřené taint zdroje, co je za ně považováno, je dále diskutováno v sekci 3.3. Jedním ze způsobů, jak testovat přítomnost injekčních zranitelností v aplikaci, je identifikovat veškeré vstupy, posílat do nich předem připravené řetězce obsahující speciální znaky [25] a snažit se rozpoznat výstup se známkami projevené zranitelnosti. To lze provádět i vzdáleně bez přístupu k (např. webové) aplikaci a nezávisle na implementaci (tzv. *black-box analýza*).

Pokud ale máme přístup ke zdrojovému kódu (tzv. *white-box analýza*) nebo dokážeme využít i přeložený tvar aplikace, lze se naopak zaměřit na potenciálně nebezpečné metody (uvedené v příloze A) a analyzovat, s jakými parametry mohou být volány. Tato volání, resp. pouze jejich potenciálně nebezpečné parametry, budeme označovat jako *taint sink*[2]. V těchto místech nemusí být použit pouze taint zdroj nebo bezpečný zdroj, ale také data, která vznikla kombinací a transformací různých zdrojů. Rovněž bychom rádi rozpoznali, zda data prošla validací pro daný typ chyby (viz sekce 3.4). Která data se k volání dostanou je také závislé na tom, jak se vyhodnotí podmíněné výrazy uvnitř programu. Úkolem *taint analýzy* je právě rozeznat, pro

1. Anglicky *taint source*, slovo taint lze přeložit např. jako *poskvrnění* či *nákaza*, v této práci ale zachováme originální terminologii
2. *Sink* lze přeložit jako *stok* či *dno*, nadále ale budeme používat originální termín

který taint sink existuje možnost zavolání s neošetřeným parametrem závislým na některém taint zdroji a který taint sink je naopak bezpečný. [26]

3.2 Automatické hledání bezpečnostních chyb

Manuální kontrola kódu a ruční testování běžící aplikace mohou být nezastupitelné, bez využití automatizovaných nástrojů by však náklady na detailní analýzu rozsáhlého software byly enormní. Míra autonomie může být různá – od jednoduchého vyhledávání řetězců v kódu až po zcela samostatné nástroje (některé dokáží dokonce zaznamenat simulovaný útok). Automatizace testování umožňuje s nižšími náklady výrazně rychleji prověřit celou aplikaci, navíc způsobem, který může být snadno deterministicky zopakován a použit k regresnímu testování či kontrole kvality a analýze trendů.

3.2.1 Dynamická analýza

Při dynamické bezpečnostní analýze musí být obvykle sestavena a spuštěna kompletní aplikace. Jak už bylo zmíněno v sekci 3.1, tradičně jsou jí zasílány předem připravené vstupy a zkoumány jsou pouze výstupy. Výhodou tohoto přístupu je především jeho reálnost, detekce jsou obvykle skutečné problémy a problematické vstupy lze snadno ověřit. Naopak dynamickým analyzátorům často unikají chyby, které nastávají pouze za velmi specifických podmínek, o to ale mohou být nebezpečnější. Metoda *fuzzing* [27] spočívá v zasílání velkých množství vstupů, které jsou do určité míry náhodné (vznikly např. modifikací validního vstupu nebo jsou generovány dle určitých pravidel), a následné analýzy, zda nedošlo k pádu programu nebo zneužití zranitelnosti. Moderní techniky dokáží dále dynamickou analýzu kombinovat s white-box pohledem a statickou analýzou (viz dále), například volit jen vstupy, pro které existuje v aplikaci potenciálně zranitelný taint sink, popř. je měnit tak, aby bylo dosaženo vysokého pokrytí kódu[3]. Relativně nová technika *IAST*[4] [28] nejprve umístí do testované aplikace

3. *Code coverage*, podle metodiky určuje např. kolik řádků v kódu bylo při testech spuštěno a kolik podmíněných výrazů bylo vyhodnoceno na `true` i `false`
4. *Interactive Application Security Testing*, interaktivní testování aplikací

senzory, které sledují požadavky na externí komponenty (databáze, souborový systém apod.) a spolupracují s dynamickou komponentou na detekci problémů.

Programovací jazyk *Perl* přímo obsahuje tzv. *taint mode* a provádí taint analýzu za běhu. [29] Všechny hodnoty z externích vstupů jsou označeny příznakem *tainted* a stejně tak hodnoty v proměnných obsahujících přiřazení výrazu s takovou hodnotou – během vykonávání programu se tak příznak rozšíří do všech míst ovlivněných neošetřeným vstupem. Perl toto označení odstraní, pokud se hodnota používá v podmíněné větvi příkazu, kde podmínkou je kontrola dané hodnoty regulárním výrazem. Pokud jsou neošetřená data parametrem volání známého jako taint sink, není umožněno jej vykonat (je vyvolána chyba za běhu se zdůvodněním). Nejenže takto může být zranitelnost nalezena, ale zároveň brání útočníkovi, aby ji zneužil.[5] Samotné regulární výrazy ale nejsou nijak kontrolovány a příznak není šířen za všech okolností, Perl tedy negarantuje bezpečný běh, spíše programátory nutí uvažovat o nebezpečných vstupech a zabraňuje určitým chybám z nepozornosti. Podobnou funkcionalitu nabízí i jazyk *Ruby*. [30]

3.2.2 Statická analýza

Při statické analýze naopak nedochází ke skutečnému spuštění kódu, analyzátor nahlíží dovnitř aplikace a pokouší se nalézt problematická místa vyhledáváním určitých vzorů, případně abstraktní simulací vykonávání kódu. Největší výhodou oproti dynamickému testování je teoretické pokrytí celého analyzovaného kódu, aniž by bylo nutné hledat rozmanité vstupy, vyhodnocen je celý program. Naopak není nutné, aby byla celá aplikace funkční, analýza může být prováděna už od začátku vývoje a chyby mohou být nalezeny i v neúplných částech. Navíc je nástrojem obvykle přímo označeno místo chyby a popis nebezpečí, vývojáři je tak poskytována i určitá forma vzdělání. Kvůli tomu, že aplikace není doopravdy spuštěna, ale mohou analyzátoru chybět informace o tom, jakým způsobem bude software využíván a s jakými technologiemi použit, a ne všechny detekce jsou relevantní pro daný kontext. Obecně se statická a dynamická analýza dobře do-

5. Ve verzi 5.8 byl přidán parametr, který umožňuje v případě nepovoleného použití pouze výpis varování místo zastavení programu

plňují (viz podsekce 3.2.1), ve zbytku této práce už se dále zaměříme pouze na automatickou statickou analýzu.

Samotné detekci chyb obvykle předchází lexikální a sémantická analýza kódu, která vede k vytvoření modelu – reprezentaci programu vhodné pro další analýzu. [31] Nejjednodušším způsobem předzpracování je odstranění komentářů a sjednocení formátování, pokročilejší analyzátory vytváří vlastní struktury, které zachycují sémantické vztahy mezi různými částmi kódu. Analýza pak kombinuje model se sadou požadovaných či naopak nežádoucích vlastností či pravidel a označuje problematická místa.

3.3 Taint zdroje

V kapitole 2 jsme o taint zdrojích mluvili jako o uživatelských vstupech, takové označení ale není zcela přesné, ve skutečnosti se může jednat o:

1. Skutečně vstupy, o které je uživatel požádán, typicky vstupy z formulářů ve webových nebo desktopových aplikacích, případně data zadaná v rozhraní pro příkazovou řádku (či přímo argumenty programu)

2. Vstupy, které předvyplní aplikace, ale pokročilým uživatelem mohou být snadno změněny – např. webová aplikace vygeneruje odkaz určitého tvaru, ale útočník může před návštěvou dané stránky upravit data vložená v adrese

3. Vstupy z externích komponent a systémů, o nichž můžeme předpokládat, že jsou validní, ale sama aplikace nemůže validitu garantovat – jedná se např. o data ze souborů, databáze či sítě

I méně zkušení vývojáři obvykle provádí nějakou kontrolu dat vyžádaných od uživatele, nemusí si ale uvědomit, že také ostatní vstupy představují bezpečnostní riziko. U webových aplikací jsou to především všechny informace odesílané na server webovým prohlížečem, který je zcela pod kontrolou uživatele (útočníka) a může zaslat libovolná data (toho lze docílit např. úpravou běžného prohlížeče). Kromě částí URL adresy sem patří i obsah skrytých formulářových polí, hodnoty cookies a zasílaných hlaviček a všechny výstupy jazyka JavaScript

běžícího na straně klienta. Na tyto hodnoty tedy nelze spoléhat při žádném bezpečnostním rozhodnutí, [32] např. zcela chybný způsob autentizace by bylo pouhé uložení cookie `prihlasen=1`. Další rozměr problém nabývá, pokud je na těchto hodnotách závislý nějaký taint sink, takové vstupy musí být každopádně ošetřeny (viz sekce 3.4).

U dalších externích vstupů si jako vývojáři obvykle nejsme zcela jisti jejich původem a mírou ochrany před modifikací. Při čtení ze souboru (např. konfiguračního) nelze určit všechny zdroje, které do něho kdy zapisovaly. Podobně u dat z databáze je lepší nepředpokládat, že jsou ošetřena[6], i pokud všechny zápisy v aplikaci do ní ošetření zajišťují – do stejné databáze může např. přistupovat i další strana (která navíc může vzniknout až v budoucnu). Rovněž při čtení ze síťového spojení bychom neměli zbytečně spoléhat na důvěryhodnost druhé strany, ta může sama být terčem útoku a při neautentizovaném spojení lze také data cíleně modifikovat během přenosu. Dosud nediskutovanou kategorií jsou systémové proměnné (proměnné prostředí). Pokud je lokální aplikace spuštěna pod systémovým uživatelem s vyššími právy než má uživatel aplikace, může útočník přepsat hodnoty systémových proměnných, které zranitelná aplikace využívá, a jejím prostřednictvím vykonat operace, na které by jako samotný uživatel neměl systémová práva.

Při statické taint analýze budeme všechna zmíněná volání detekovat a nebezpečnost vstupu propagovat na další místa (viz sekce 3.5). Vstupy z prohlížeče v Javě většinou získáme ze tříd implementující rozhraní `ServletRequest`, rozpoznat se ale dají také parametry označené anotacemi, které nabízí např. framework Spring. Čtení ze souboru nebo síťového proudu je obvykle převedeno na `BufferedReader` a pak stačí detekovat volání `readLine`, alternativně lze použít i `DataInputStream` nebo `LineNumberReader` (vše z balíku `java.io`), popř. třídu `Properties` pro konfigurační soubory. Další taint zdroje jsou uvedeny v příloze B.

6. Výjimkou by mohly být hodnoty ze sloupců, kde je dané omezení na možné hodnoty (*constraint*) nastaveno už na úrovni databáze

3.4 Ošetření taint zdrojů

Kromě volání s bezpečnými parametry je v aplikaci často nutné použít taint sink, který na taint zdrojích skutečně závisí a nelze jej oddělit od dotazu či příkazu – v tomto případě musí být data zkontrolována nebo upravena tak, aby ke zneužití nemohlo dojít. U webových aplikací musí ošetření probíhat na straně serveru (popř. duplicitně), protože klientský kód je pod kontrolou útočníka a vždy může být obejit. Jednou z možností je validace – pokud jsou přijata data obsahující znaky se speciálním významem nebo data nemohou odpovídat požadované informaci, taint sink není zavolán a je zahlášena chyba (popř. je vstup ignorován). Před samotnou validací může být nutné data nejprve dekódovat do podoby, která bude skutečně použita jako vstup pro taint sink (např. z URL převést znaky s `%`), a provést *kanonizaci* (převedení do kanonického tvaru, např. vyhodnocení posloupnosti `../` v cestě k souboru), při špatném pořadí těchto operací (či implicitní kanonizaci následující až po validaci) by útočník mohl obejít kontrolu vhodným kódováním nebezpečného vstupu. [33]

Tzv. *webový aplikační firewall* umožňuje detekovat a blokovat podezřelé vstupy, nedokáže ale zabránit všem útokům a měl by být použit jen jako další vrstva obrany. [34] Základem by měla být validace přímo v aplikaci a to za použití konzervativnějšího *whitelist* přístupu, kdy je specifikováno, jaká data jsou bezpečná, a ostatní jsou zamítnuta. Podobně jako u principu přidělování nejnižších možných práv uživatelům je vhodné povolit co nejmenší množinu vstupních dat. Tu lze specifikovat např. výčtem prvků nebo regulárním výrazem.

3.4.1 Kódování dat

V některých případech není možné takto vstupy omezit, protože aplikace musí přijmout i data obsahující nebezpečné znaky (např. i jméno může obsahovat apostrof a příspěvek v diskusi libovolné tisknutelné znaky). Pak je nutné provést konverzi (sanitizaci), která zaručí, že speciální symboly ztratí svůj řídící význam a budou interpretovány jako běžné znaky. Tato konverze je závislá na systému, který bude data interpretovat, například znak `'` (apostrof) je v SQL dotazu potřeba konvertovat podle použité databáze na `\'` nebo `''` a v kontextu HTML na `'` (popř. `'`). V Javě lze pro jednoduchou náhradu

znaků použít např. metodu `replace` třídy `String`. Je ale důležité skutečně nahradit všechny znaky se speciálním významem pro daný kontext a podobně jako při validaci preferovat whitelist přístup (převést všechny znaky kromě bezpečných). Obecně (a zvláště např. pro XSS) se jedná o proces náchylný na chyby a stejně jako se nedoporučuje vlastní implementace kryptografie i při validaci je vhodné spoléhat na existující řešení. Pro jazyk Java to jsou například volně dostupná:

- *ESAPI (The OWASP Enterprise Security API)* je knihovna od organizace OWASP (viz sekce 2.1.2), která usnadňuje tvorbu bezpečných aplikací tím, že poskytuje rozhraní a referenční implementaci pro zapouzdření operací kritických pro bezpečnost jako je autentizace a řízení přístupu, kryptografie nebo právě validace a kódování pro různé jazyky a kontexty. [35] Rozhraní `Encoder` z balíku `org.owasp.esapi` nabízí metody jako `encodeForHTML` s parametrem a návratovou hodnotou typu řetězec. Některé metody jako `encodeForSQL` mají zároveň parametr typu `Codec`, který určuje způsob kódování (zde už implementace obsahuje kodek pro MySQL a databáze od firmy Oracle).

- *OWASP Java Encoder* je novější a čistě sanitizační knihovna, která se zaměřuje na snadno použitelnou a výkonnou ochranu před XSS. [36] Třída `Encode` v balíku `org.owasp.encoder` poskytuje metody i pro specifické kontexty – např. `forCssUrl` (pro URL adresu použitou v kaskádových stylech) nebo `forHtmlUnquotedAttribute` (pro hodnotu HTML atributu, která není uzavřena v uvozovkách). Součástí projektu je také knihovna tagů pro použití v JSP s jazykem EL[7].

- *OWASP Java HTML Sanitizer* je další sanitizační knihovna od OWASP, tato se nepoužívá pro konverzi dat, která se do HTML vkládají, ale specificky přímo pro ošetření nedůvěryhodného HTML (kde musí být zachován význam značek). [37] Volání metody `sanitize` třídy `PolicyFactory` z balíku `org.owasp.html` zajistí ošetření dat, takže budou obsahovat jen definované HTML elementy, atributy a jejich hodnoty.

7. *Expression Language*, jednoduchý jazyk, který usnadňuje zápis dat do JSP stránek

- Známá knihovna *Apache Commons Lang* obsahuje také třídu `StringEscapeUtils` [38] s množstvím metod pro konverzi chránící před XSS (např. `escapeHtml4`). Starší verze (< 3) obsahuje také metodu pro ošetření SQL, ta ale pouze zdvojuje výskyt každého apostrofu, její použití tak obecně není spolehlivé.

- Jednoduché ošetření XSS nabízí také framework *Spring* ve třídě `HtmlUtils` [39] prostřednictvím volání `htmlEscape`, sanitizovat je možné i JavaScript s využitím třídy `JavaScriptUtils`.

3.5 Analýza toku dat a propagace stavu taint

Nejjednodušším způsobem detekce potenciálních injekčních zranitelností je označení všech volání známých jako taint sink. I taková analýza může značně urychlit manuální kontrolu na dané chyby, protože omezí množství kódu, které je nutné projít, od automatického nástroje ale očekáváme schopnost rozlišení mezi bezpečným a zneužitelným voláním. K tomu je potřeba nejen detekovat a klasifikovat taint zdroje, ale hlavně analyzovat možné toky dat v aplikaci (viz dále). Dále vezmeme v úvahu kódování nebezpečných dat (viz podsekce 3.4.1) – pokud je provedeno správně, není žádoucí, aby volání s těmito daty byla dále detekována jako bezpečnostní chyby. Označit ošetřená data za bezpečná (jako kdyby nepocházela od uživatele) by však nebylo zcela správné:

1. Úprava je často platná pouze pro danou bezpečnostní chybu a použití v jiném kontextu stále může vést ke zneužití (např. vstup bezpečný pro SQL může obsahovat HTML a být zneužit pro XSS, viz sekce 2.2 a 2.4).

2. Pokud je provedeno kódování, data mohou být později opět dekódována na původní nebezpečný vstup, pouhé označení dat na bezpečná by vedlo ke ztrátě této informace.

Místo toho je proto vhodnější označit data speciálním příznakem pro ošetření podle typu zranitelnosti a považovat je případně za bezpečná teprve až bude dosažen konkrétní taint sink.

3.5.1 Formální verifikační metody

Na rozdíl od běžného testování statická analýza neuvažuje postupně jednotlivé vstupní hodnoty, ale snaží se vyvodit vlastnosti programu zvažováním kombinací všech možných vstupů a běhů programu. Pro tento účel také bylo vyvinuto množství formálních metod:

- Při *symbolickém spouštění (symbolic execution)* je konkrétní vstup nahrazen symbolem a další hodnoty uvnitř programu výrazy obsahující tyto symboly. Každá podmínka v programu vede k rozvětvení stromu, kde uzel tvoří místo v programu, výraz se symboly a také podmínka pro danou cestu. Analýzou splnitelnosti těchto podmínek lze přímo detekovat určité chyby nebo generovat vstupy, které pokryjí celý kód. [40]

- *Deduktivní verifikace* umožňuje ověřit korektnost programů pomocí automatického dokazování vztahu mezi vstupem a výstupem (pro parciální korektnost) a nalezení klesající sekvence zdola ohraničeného výrazu (jako důkaz konvergence), často je ale nutné ručně určit invarianty cyklů apod. [41]

- Technika *model checking* ověřuje specifikaci definovanou formulemi temporální logiky na modelu systému reprezentovaném stavy, přechody mezi nimi a tvrzeními. Z důvodu efektivity se často využívá varianta *bounded model checking*, která vlastnost systému ověřuje jen do omezeného počtu provedených kroků. [42]

- *Abstraktní interpretaci* lze považovat za zobecnění výše zmíněných metod. Všechny konkrétní hodnoty budou reprezentovány jen menším počtem abstraktních stavů (konkrétní číslo by mohlo být abstrahováno např. do kategorie *záporné*, *0* nebo *kladné*). To samozřejmě zavádí do analýzy nepřesnost (např. inkrementované záporné číslo se může stát nulou, ale nemusí), na druhou stranu ale umožňuje podstatně efektivnější analýzu a při správném použití stále umožňuje prokázat přítomnost či naopak absenci konkrétních chyb. Při větvení programu se také zaznamenává predikát platný pro hodnoty v dané větvi programu, na rozdíl od symbolického spouštění se větve ale opět

spojují a výsledkem je konečný graf toku, který lze iterativně aktualizovat, dokud nedojde k ustálení informací o datech. [43]

Formální metody umožňují verifikovat vlastnosti systému s jistotou, popř. za daných podmínek (model odpovídá skutečnosti, chyba se projeví do *k* kroků apod.), nástroje však obvykle nejsou jednoduché na použití a vyžadují správnou konfiguraci pro daný systém. Hlavním problémem je ale často rychlý nárůst času a paměti potřebných pro analýzu rozsáhlejších systémů a tedy praktická neschopnost verifikace běžných programů.

3.5.2 Efektivní analýza

Pro prakticky použitelný nástroj naopak budeme vyžadovat snadné použití s minimem konfigurace a hlavně škálovatelnost, která umožní analyzovat reálné aplikace (např. se stovkami tisíc řádků kódu). Formální metody mohou sloužit jako inspirace (zejména abstraktní interpretace), je ale nutné se vzdát jejich přesnosti a garance. Kvůli aproximacím nemusí být všechny chyby nalezeny a reportované problémy naopak nemusí být skutečné. Při analýze toku dat pro účely taint analýzy budeme abstrahovat hodnoty do stavu *tainted* reprezentujícího potenciálně nebezpečné řetězce a *safe* reprezentujícím bezpečné vstupy. Pokud pro danou hodnotu nemůžeme rozhodnout, zda může být bezpečná, nebo ne, můžeme ji považovat za bezpečnou za cenu neodhalení všech chyb, nebo za nebezpečnou za cenu falešných detekcí, lze ale také použít nový stav *unknown* a toto rozhodnutí tak odložit na dobu reportování výsledků.

Důležitou strukturou (používanou i formálními metodami) je tzv. *graf řízení toku (control flow graph)*. Je to reprezentace programu (v našem případě pouze jedné metody) orientovaným grafem, kde uzly odpovídají příkazům v kódu a hrany značí možné přechody na následující uzel, [44] nebo se uzly už skládají ze základních bloků instrukcí (takových, kde je následovník jednoznačný). [45] Kromě toho graf obsahuje také speciální vstupní a výstupní uzel bez instrukcí. Pro uzly v grafu si chceme pamatovat určitá *fakta*, která budeme modifikovat při průchodu grafem podle instrukcí. Pokud obsahuje uzel více vstupních hran, musí být provedeno spojení faktů. Pro smysluplné spojování musí být množina všech možných hodnot fakt částečně uspořádána

a spolu s operací spojení tvořit matematickou strukturu *svaz* [46]. Pro každou dvojici faktů musí v množině existovat jejich supremum a infimum, spojování také bude asociativní, komutativní a idempotentní. Pro taint analýzu popisují fakta bezpečnost aktuálních hodnot – tvoří tím úplné uspořádání a supremem (resp. infimem) dvojice faktů je jednoduše ta méně (resp. více) bezpečná hodnota. V praxi ale budou fakta obsahovat i další informace a také jejich spojování bude komplikovanější. Na rozdíl od formální abstraktní interpretace budeme při větvení ignorovat samotnou podmínku a do obou větví vstoupí stejná fakta.

Teoreticky lze nahradit volání metod jejich definicemi a vytvořit globální graf, analýza by se ale takto snadno opět mohla stát příliš náročnou na zdroje. Navíc počítáme s použitím nástroje už od iniciálních fází vývoje (oprava chyb už během implementace je mnohem snadnější) a globální pohled nemusí být vždy výhodou (můžeme chtít včas odhalit budoucí problémy, přestože aktuálně přímé riziko nepředstavují). Proto budeme provádět analýzu lokálně a globálně zohledníme pouze shrnutí výsledku lokální analýzy (viz kapitola 5).

4 Nástroj FindBugs a možnosti rozšíření

Před popisem samotné implementace vylepšené detekce injekčních chyb (v kapitole 5) je nutné se seznámit s nástrojem, který bude použit pro základ analýzy, a především možnostmi jeho rozšíření (viz sekce 4.3). Protože analýza probíhá nad přeloženou formou souborů jazyka Java, část kapitoly (sekce 4.2) je také věnována tomuto formátu.

4.1 Nástroj FindBugs

FindBugs [2] je nástroj pro detekci chyb v Java programech pomocí statické analýzy kódu. Vyvinut byl na Marylandské univerzitě, distribuován je pod svobodnou licencí LGPL[1] a jeho vývoj stále probíhá, aktuální verze je 3.0.1. Kromě vlastního grafického rozhraní a rozhraní pro příkazový řádek nabízí integraci s vývojovými prostředími (Eclipse, NetBeans, IntelliJ IDEA), nástroji pro automatizaci sestavování aplikace (Ant, Maven, Gradle) i jiným software (Jenkins, SonarQube).

FindBugs dokáže detekovat širokou škálu chyb – obecně například kód, který se pravděpodobně chová jinak, než vývojář zamýšlel, porušuje předepsané či osvědčené postupy, je neefektivní nebo zbytečný. Obvykle se jedná o relativně jednoduchá pravidla (např. chybějící příkaz `break` ve větvi výrazu `switch`, nepoužitá lokální proměnná nebo třída s vlastní metodou `equals`, ale bez `hashCode`) a síla nástroje je především v jejich množství. FindBugs ale dokáže použít i složitější analýzu toku dat (viz podsekce 4.3.2) a víceprůchodovou analýzu k detekci problémů, které nemusí být snadno zřetelné ani při manuální kontrole kódu zkušenějším vývojářem. Toho využívá například při detekci možných dereferencí hodnoty `null`. Mezi volně dostupným software se jedná jistě o nejlépe použitelný statický analyzátor, je-li cílem skutečná detekce chyb a ne pouze prohřešků proti formátování apod. [1] Obsahuje také kategorii bezpečnostních chyb a dovede odhalit některé případy SQL injekce, XSS nebo procházení adresářem, bohužel přesnost těchto detektorů je velmi nízká (viz podsekce 6.2.1).

1. *GNU Lesser General Public License*, na rozdíl od GPL umožňuje i použití v proprietárním software, modifikace ale musí být zveřejněny

Protože jako častý důvod, proč statické analyzátory stále nejsou při vývoji hojně využívány, bylo označeno velké množství falešných detekcí (planých poplachů), filosofií FindBugs je tento problém potlačit i za cenu toho, že některé skutečné problémy odhaleny nebudou. To je pravděpodobně dobrá strategie, protože rozsáhlý software bude pravděpodobně i tak obsahovat velké množství chyb, které nástroj detekovat dokáže a použití analyzátoru může být velkým přínosem. Pro oblast bezpečnosti ale požadujeme vyšší garanci, že aplikace neobsahuje nebezpečné zranitelnosti, a není zde možné spoléhat na nástroj, který odhalí jen zlomek skutečně přítomných chyb.

4.1.1 Rozšíření FindSecurityBugs

FindBugs lze však rozšířit o další sady detektorů (včetně vlastních, viz sekce 4.3), jednou z nich je i *FindSecurityBugs*, jejíž cílem je usnadnění bezpečnostního auditu kódu. [3] Tomu odpovídá i přítomnost detektorů, které nehledají přímo chyby, ale také některé samotné taint zdroje (viz sekce 3.3) nebo místa, kde k bezpečnostním problémům často dochází. Projevilo se to i v detekci injekčních zranitelností – jako potenciální chyba byl původně značen každý taint sink, kromě případů, kdy vstupem zjevně byla konstanta známé hodnoty. Převažující falešná hlášení byla motivací pro implementaci výrazně pokročilejší analýzy popsané v kapitole 5. Další detekované chyby často souvisí se špatným použitím kryptografie nebo slabými algoritmy, přítomny jsou také detektory pro OS Android. Rozšiřující sady lze použít či integrovat s téměř libovolným nástrojem, který už dokáže spolupracovat s původním FindBugs (viz sekce 6.1).

4.2 Bytekód

Analýza ve skutečnosti neprobíhá nad textovým zdrojovým kódem, ale FindBugs analyzuje rovnou přeložené třídy (`.class`) v podobě tzv. *bytekódu*, který už *JVM*[2] za běhu kompiluje do strojového kódu pro danou platformu. Kromě aplikací v Javě lze tak hledat chyby i v jazycích Scala, Groovy nebo Closure a dalších, pro něž existuje implementace pro JVM (např. Python a Ruby). Všechny detektory

2. *Java Virtual Machine*, virtuální stroj jazyka Java

vyvíjené pro Javu ale nemusí být smysluplně funkční, protože tyto jazyky mohou využívat jiná volání a také vytvořit bytekód, pro který neexistuje odpovídající kód jazyka Java (ten např. nevyužívá instrukci pro dynamické volání metod). Další výhodou je, že stejný bytekód odpovídá více variantám zápisu téhož programu a to navíc takové variantě, ze které je explicitně vidět, co program skutečně vykonává. Není tak nutné řešit formátování a různý zápis se stejným významem a tzv. syntaktický cukr. Dále máme jistotu, že se jedná o syntakticky validní program, a překladač také provede určitá zjednodušení (jako vyhodnocení konstantních výrazů), které mohou analýzu usnadnit.

Teoretickou nevýhodou je určitá ztráta informace při překladu, nelze tak například detekovat blok kódu, který je jinak odsazen, než odpovídá jeho významu (viz *CWE-483*). Na rozdíl od strojového kódu je ale při běžném překladu úbytek informace minimální a bytekód může být relativně snadno převeden zpět do podoby zdrojového kódu (dekompilátor je např. integrován ve vývojovém prostředí IntelliJ IDEA). V souborech zůstávají i názvy identifikátorů a mapování mezi instrukcemi a čísly řádků, což lze využít pro zobrazení problematických míst přímo v původním zdrojovém kódu. Za nevýhodu lze považovat i nutnost překladu, stačí ale analyzovat syntakticky korektní třídy, aplikace nemusí být funkční a detekovat chyby lze i bez dostupnosti využitých knihoven (za cenu možného snížení přesnosti). Zůstává tedy zachována jedna z výhod statické analýzy – možnost detekovat chyby už v brzké fázi vývoje (kdy je oprava snadnější a méně nákladná). Pokud chceme vytvořit nový detektor (nebo vylepšit stávající), narazíme na praktickou překážku tohoto přístupu – prvním předpokladem bude seznámit se s instrukční sadou bytekódu a základní principem fungování JVM (viz dále). [47]

4.2.1 Struktura class souboru a značení datových typu

Při překladu zdrojového kódu třídy (nebo rozhraní) s koncovkou `.java` (jinými jazyky se už dále nebudeme zabývat) vznikne minimálně soubor stejného názvu s koncovkou `.class`, případně další soubory, pokud v sobě třída obsahovala definice jiných tříd.[3] Každý soubor tedy bude obsahovat informace právě o jedné třídě (popř. roz-

3. Pak se bude název skládat z názvu veřejné třídy, znaku $ a pojmenování vnitřní či lokální třídy, popř. pouze čísla pro anonymní třídy

hraní nebo výčtovém typu). Za hlavičkou souboru (s identifikační konstantou a číslem verze) následuje tzv. *constant pool*, který kromě číselných a řetězcových konstant obsahuje všechny identifikátory tříd, metod a typů. Další části kódu se pak místo těchto identifikátorů odkazují přímo na index v constant poolu (a záznamy v něm se často odkazují na další záznamy pro bližší určení).

Pro účely tvorby detektorů je důležitý zejména formát těchto údajů. Třída (popř. rozhraní nebo výčtový typ) je identifikována jejím názvem včetně balíku, kde místo znaku . je použito /, např. java/lang/Object pro třídu Object z balíku java.lang. Metoda je identifikována identifikátorem třídy, ke které patří, názvem metody a její *signaturou*, která určuje počet a typ parametrů a návratovou hodnotu. Místo jména konstruktoru se používá řetězec <init> a místo statického konstruktoru (část kódu třídy v bloku static) <cinit>. Primitivní typy byte, char, double, float, int, long, short a boolean jsou reprezentovány popořadě písmeny B, C, D, F, I, J, S a Z. Referenční typ má stejný identifikátor jako třída, ale na začátek je přidáno písmeno L a na konec znak ;. Identifikátor typu pole vznikne přidáním znaku [před jeho typ (lze i vícekrát pro vícerozměrné pole, např. [[B značí dvojrozměrné pole typu byte). Signatura metody se skládá ze zřetězených identifikátorů typů parametrů v závorkách následovaných identifikátorem návratového typu (pro typ void se použije písmeno V), např. pokud má metoda jeden parametr typu int, jeden typu String a nevrací hodnotu, bude mít signaturu (ILjava/lang/String;)V.

V další části souboru se nachází binární příznaky, které např. určují, zda je třída veřejná, abstraktní apod. Následují odkazy do constant poolu určující samotnou třídu, nadtřídu a případná implementovaná rozhraní (jedině třída Object nemá žádnou nadtřídu). Dále je v souboru seznam atributů třídy (zde máme na mysli datové položky třídy, jako atributy se také nazývají informace na konci souboru). Každá obsahuje jméno, identifikátor typu (deskriptor), příznaky určující vlastnosti (např. viditelnost) a další informace (např. jestli je atribut označen anotací @Deprecated). Poté následují všechny metody, oproti atributům obsahují záznamy především tělo metody – posloupnost instrukcí (viz dále). Také přidaných informací může být větší množství, např. vyhazované výjimky nebo tabulku, která mapuje instrukce v metodě na řádky zdrojového souboru. Celý class soubor je uzavřen sekcí s méně podstatnými informacemi, které se vztahují k celé třídě.

4.2.2 Instrukční sada

Důležitým aspektem fungování JVM je práce ze zásobníkem operandů. Na začátku metody je zásobník prázdný a s inkrementováním programového čítače se postupně vykonávají instrukce. Každá instrukce může odebrat několik hodnot z vrcholu zásobníku (nejprve musí odebírat ty, které byly vloženy naposled) a také uložit hodnoty na zásobník. Pro uložení číselné konstanty na zásobník slouží instrukce `bipush` resp. `sipush`, které vloží hodnotu uvedenou na dalším bytu resp. dvěma byty. Jako zkratku pro hodnoty od -1 do 5 lze použít instrukce `iconst` bez parametru (podobné existují i pro typy s plovoucí desetinnou čárkou). Pro vysoké číselné hodnoty a hlavně hodnoty referenčního typu se využívá instrukce `ldc`, která vloží na zásobník hodnotu v constant poolu (kam je automaticky uložena při překladu).

Pro přesun dat mezi zásobníkem a lokálními proměnnými slouží sada instrukcí `load` (hodnotu proměnné vloží na zásobník) a `store` (odebere vrchol zásobníku a hodnotu uloží do proměnné). Tyto instrukce existují pro každý typ a kromě variant, kdy je index požadované proměnné specifikován hodnotou bytu za instrukcí, existují opět varianty bez parametru pro proměnné na indexu od 0 do 3. Nejvyšší index lokální proměnné a také maximální hloubka zásobníku jsou předem známé pro každou metodu. Na začátku provádění kódu metody jsou na nižších indexech uloženy hodnoty parametrů. Pro přesun mezi zásobníkem a atributy třídy slouží instrukce `getfield` a `putfield`, na zásobníku musí být před zavoláním také reference na instanci třídy (pokud se nejedná o statické atributy, pak se ale použijí instrukce `getstatic` a `putstatic`).

Pro volání metod se podle jejího typu používá instrukce `invokestatic`, `invokevirtual`, `invokeinterface` nebo `invokespecial` (`invokedynamic` nemá využití pro jazyk Java). Před jejich zavoláním musí být na zásobníku všechny jejich parametry včetně implicitního parametru *this* s referencí na instanci třídy, která metodu volá (kromě `invokestatic`, která se používá pro statické metody). Nová instance třídy se vytváří instrukcí `new`, následně je potřeba zavolat konstruktor. Návratová hodnota je po vykonání metody vložena na vrchol zásobníku (pokud není typu void). Někdy je nutné použít instrukce pro přímou manipulaci se zásobníkem – `pop` odstraní vrchol zásobníku bez využití (např. pokud ignorujeme návratovou hodnotu), a `dup` nao-

pak zdvojí (např. před voláním konstruktoru, který odstraní referenci na instanci), `swap` prohodí dva vrchní prvky. Metody není nutné volat pro základní aritmetické a bitové operace nebo konverzi typů, pro ně existuje množství instrukcí. Pole se deklaruje voláním `newarray`, resp. `anewarray` pro primitivní, resp. referenční typy, další instrukce existují pro práci s ním.

Důležité jsou instrukce pro řízení běhu programu. Instrukce `goto` (která v samotném jazyce Java není, ale v bytekódu je nutná) slouží k nastavení programového čítače na adresu danou dvěma následujícími byty, skoky jsou však omezeny na aktuální metodu. Instrukce `ifeq` a dalších pět slouží k podmíněnému skoku podle porovnání celočíselného vrcholu zásobníku s nulou ($=$, $\neq$, $<$, $\leq$, $>$, $\geq$). Pro zjednodušení existuje další šestice instrukcí, kde nedochází k porovnání s nulou, ale dvou horních hodnot zásobníku. Instrukce `if_acmpeq` a `if_acmpne` slouží pro srovnání referenčních typů (pouze $=$, $\neq$) a speciální instrukce `ifnull` a `ifnonnull` pro porovnání s hodnotou `null`. Pro konstrukci `switch` existuje instrukce `tableswitch` a také `lookupswitch`, použije se ta, jejíž implementace bude v daném případě efektivnější. Pro návrat z metody slouží instrukce `return` a její varianty podle typu návratové hodnoty (bere se opět z vrcholu zásobníku). Tok může ovlivnit také instrukce `athrow` pro vyhození výjimky.

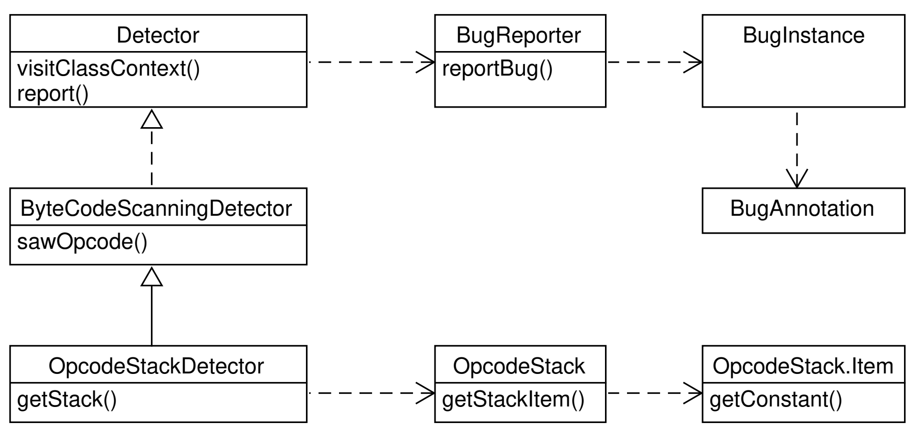
4.3 Tvorba vlastních detektorů

Nejpřínosnějším nalezeným informačním zdrojem o tvorbě detektorů je prezentace [45] z konference *PLDI*[4] spolu se stručnou dokumentací [48] aplikačního rozhraní, pro dosažení potřebné úrovně znalostí však bylo často nutné procházet přímo zdrojové kódy nástroje FindBugs [49]. Vlastní detektor musí být přidán do rozšíření, což je JAR soubor obsahující nejméně jednu třídu s detektorem a dva konfigurační XML soubory. Ve `findbugs.xml` je uvedena identifikace rozšíření, seznam chyb (*bug pattern*) se zařazením do kategorie a seznam tříd detektorů s odkazem na ně. Soubor `messages.xml` obsahuje další informace o rozšíření a detektorech a především podrobný popis detekovaných chyb, který může být zobrazen v rozhraní FindBugs.

4. *Programming Language Design and Implementation*, tradiční každoroční konference o návrhu a implementaci programovacích jazyků

Samotný detektor je třída implementující rozhraní `Detector` (popř. `Detector2`) z balíku `edu.umd.cs.findbugs`. To předepisuje metodu `visitClassContext`, která je zavolána jedenkrát pro každou analyzovanou metodu. Je na daném detektoru, aby předanou instanci typu `ClassContext` zpracoval, např. si postupně vyžádal bytekód jednotlivých metod a vyhledal problematické posloupnosti instrukcí. Aby mohly být nalezené chyby reportovány (a např. zobrazeny v rozhraní po skončení analýzy), detektor musí mít konstruktor s parametrem typu `BugReporter`, který si uchová v atributu a při detekci chyby zavolá jeho metodu `reportBug`. Rozhraní `Detector` dále předepisuje metodu `report`, která je zavolána při ukončení analýzy a uvnitř ní je poslední možnost chybu reportovat (to je výhodné zvláště pokud problém nesouvisí pouze s jednou třídou), lze to ale udělat kdykoliv během analýzy. Jediným parametrem při reportování je instance třídy `BugInstance`. Její konstruktor má tři parametry – detektor (stačí použít klíčové slovo `this`), typ chyby (zde se použije název chyby z konfigurace) a prioritu, která by měla odpovídat míře jistoty, že se jedná o skutečnou chybu. Jako její hodnotu lze použít číselnou konstantu ze třídy `Priorities` (většinou *high*, *normal* nebo *low*), záleží na rozhraní, jak s prioritou naloží (např. detekce s nízkou prioritou často nejsou zobrazovány). K chybě je vhodné přidat identifikaci místa problému (metody `addClassAndMethod` či `addSourceLine`), případně libovolnou poznámku pomocí instancí třídy `BugAnnotation`. Analýza chyb tedy zjednodušeně probíhá takto:

1. FindBugs načte rozšíření a ze souboru `findbugs.xml` zjistí umístění detektorů.
2. Je vytvořena nová instance každého detektoru s předaným objektem typu `BugReporter`.
3. Z každé analyzované třídy je vytvořena instance typu `ClassContext` a je zavolána metoda `visitClassContext` každého detektoru s tímto parametrem.
4. Detektor třídu libovolně analyzuje a případně volá metodu `reportBug` předaného objektu typu `BugReporter` s novou instancí `BugInstance` obsahující informace o chybě.
5. Analýza končí zavoláním metody `report` pro každý detektor.



Obrázek 4.1: Zjednodušený diagram tříd pro detekci

4.3.1 Třídy pro usnadnění tvorby detektorů

Přímou implementací rozhraní lze dosáhnout libovolné analýzy kódu, pro snadnější tvorbu detektorů ale existuje množství tříd s užitečnou funkcionalitou, které detektor může rozšířit (místo přímé implementace), popř. jen použít během analýzy. Jednodušší typy chyb lze obvykle detekovat jako posloupnost instrukcí bytekódu v rámci jedné metody. V tomto případě je vhodné, aby detektor rozšiřoval třídu `BytecodeScanningDetector`, která kromě množství užitečných funkcí obsahuje především metody volané při průchodu různými částmi kódu (např. třída, metoda, pole nebo přímo instrukce). Detektory tak lze programovat s využitím návrhového vzoru *návštěvník (visitor)*. Místo implementace metody `visitClassContext` může detektor přepsat přímo metodu `sawOpcode`, která bude zavolána pro každou analyzovanou instrukci spolu s číslem, které ji identifikuje. Při analýze např. často potřebujeme detekovat volání určité metody – nejprve musíme ověřit podle čísla, že se jedná o jednu z `invoke` metod, a pak ověřit jméno volané metody pomocí `getNameConstantOperand`. Protože nás pravděpodobně zajímá konkrétní metoda a ne všechny metody daného názvu, použijeme volání `getClassConstantOperand` pro zjištění třídy (včetně balíku), ke které metoda patří. Pokud má

třída více metod stejného názvu, můžeme použít ještě volání getSigConstantOperand pro zjištění signatury metody, formát je stejný jako v class souborech (viz podsekce 4.2.1).

Při detekci volání metody nás mohou zajímat také hodnoty jejích parametrů. Pokud je ve zdrojovém kódu použita přímo konstanta (konkrétní číslo či řetězec) v místě volání, lze detekovat instrukce jako ldc těsně před invoke instrukcemi a přes dostupné metody vyčíst hodnotu z constant poolu (popř. přímo z instrukcí). Pokud je ale konstanta nejprve uložena do proměnné a v místě volání použita, je detekce podstatně složitější. Kromě sledování hodnot proměnných je pak nutné zaznamenávat větvení v programu a ověřit, zda je hodnota skutečně jednoznačně dána (a nezávisí na vstupech dostupných až za běhu). Pokročilejší detektory lze ale snadněji vytvářet rozšířením třídy OpcodeStackDetector (která sama rozšiřuje dříve zmíněný BytecodeScanningDetector). Ta přidává metodu getStack s návratovou hodnotou typu OpcodeStack, která se snaží modelovat zásobník operandů v místě volání (modelem platným nezávisle na skutečném běhu programu). Pro pasivní použití je nejdůležitější metoda getStackItem, která vrátí instanci vnořené třídy Item s informacemi o hodnotě na zásobníku na specifikovaném indexu. Z tohoto objektu lze konečně mj. vyčíst hodnotu konstanty pomocí metody getConstant (je-li v dané položce konstanta a bylo-li ji možné jednoznačně určit). Závislosti zmíněných tříd jsou vyznačeny na obrázku 4.1.

4.3.2 Pokročilá analýza toku dat

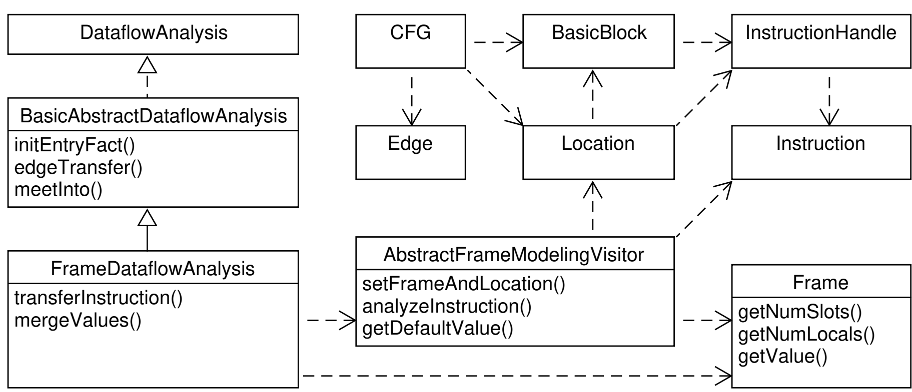
Jednoduchou analýzu toku provádí už OpcodeStackDetector, když vyvozuje určitá fakta o hodnotách na zásobníku. Existuje i možnost vlastního rozšíření tohoto mechanismu, pokud je ale analýza toku dat pro detekci chyby zásadní, je výhodnější použít odlišný mechanismus, který zajistí vyšší přesnost, flexibilitu a znovupoužitelnost analýzy. FindBugs umožňuje předzpracovat kód metod a vytvořit *graf kontrolního toku* (zmíněný v podsekci 3.5.2, dále jen *CFG*), jehož uzly jsou reprezentovány instancemi třídy BasicBlock a hrany objekty typu Edge. Instance BasicBlock obsahuje seznam instancí InstructionHandle, z nichž každá se odkazuje na konkrétní instrukci (reprezentovanou instancí třídy Instruction). Kvůli způsobu, jakým graf ovlivní instrukce jsr, je možné, že se více instancí BasicBlock odkazuje na

stejnou instanci InstructionHandle. Konkrétní lokace v kódu proto reprezentují instance třídy Location, které se odkazují na svoje instance BasicBlock a InstructionHandle.

Detektory, které využívají CFG, rozšiřují CFGDetector nebo implementují rozhraní Detector přímo a z instance ClassContext obdrží iterátor pro metody reprezentované instancemi třídy Method. Třídy Method, InstructionHandle a Instruction pochází z knihovny *BCEL*[5] [50], na které je FindBugs závislý[6]. Smyslem tohoto přístupu je, že pro každou metodu si lze vyžádat tok dat pro určitý typ analýzy, který může být využit ve více detektorech. Prakticky se jedná o instanci potomka třídy Dataflow, kterou lze obecně obdržet od třídy AnalysisCache (dostupnou ze třídy Global) voláním getMethodAnalysis (pro zjednodušení uvažujeme jen tento typ analýzy). Parametrem je třída vyžadovaného toku a deskriptor metody, pro kterou objekt požadujeme (deskriptor je instance MethodDescriptor a lze ji vytvořit např. přes třídu BCELUtil). Ve vytvářeném detektoru lze získat iterátor lokací pro danou metodu, během iterování zavolat na toku metodu getFactAtLocation (kde parametrem je právě lokace) a obdržet objekt, který nese fakta pro dané místo. O jaký fakt se jedná, záleží na typu toku, pro zjištění hodnot parametrů lze použít ConstantDataflow).

Důležitá je možnost vytvoření vlastní analýzy toku dat, samotný detektor pak bude pracovat pouze s potomkem generické třídy Dataflow. Ta má dva typové parametry – libovolný typ reprezentující určitý fakt na dané lokaci a vlastní implementaci rozhraní DataflowAnalysis, která provádí samotnou analýzu (často s využitím dalších tříd). Existuje hierarchie tříd (nejvýše je BasicAbstractDataflowAnalysis), která implementaci analýzy ulehčuje, stačí pak implementovat metody jako initEntryFact pro inicializaci na začátku vstupu CFG, edgeTransfer pro případnou změnu faktu při přechodu přes hranu grafu nebo meetInto, která na začátku bloku kódu spojí fakta pocházející z různých vstupních hran do jednoho. Základní analýza mění fakta v uzlech jen na úrovni celého bloku pomocí metody transfer, potomci AbstractDataflowAnalysis i na úrovni instrukcí v závislosti na implementaci metody transferInstruction.

5. *Byte Code Engineering Library*, knihovna nadace *Apache* pro práci s bytekódem
6. Detektory implementující rozhraní Detector2 na BCEL závislé nejsou



Obrázek 4.2: Zjednodušený diagram tříd pro analýzu toku

4.3.3 Modelování proměnných a zásobníku operandů

Je nutné si uvědomit, že pokud chceme modelovat informace o jednotlivých hodnotách v programu, použitým faktem analýzy nemůže být abstrakce jedné hodnoty na lokaci, ale model lokálních proměnných a všech hodnot na zásobníku operandů. Pro tento účel je výhodné rozšířit abstraktní třídu `FrameDataflowAnalysis` a pro reprezentaci faktu využít potomka generické třídy `Frame`, kde teprve typový parametr reprezentuje fakt už pouze pro jednu hodnotu. Instance potomka třídy `Frame` se skládá z `getNumSlots` *slotů* (velikost je určena už na začátku analýzy podle maximální hloubky zásobníku), kde prvních `getNumLocals` pozic zabírají proměnné a zbytek zásobník. Hodnotu na konkrétní pozici lze obdržet voláním `getValue`. Potomci `FrameDataflowAnalysis` definují metodu `mergeValues`, která má jako parametr kromě dvou instancí potomka `Frame` i index slotu a stačí tedy definovat spojování faktů po jednotlivých hodnotách. Pro zjednodušení implementace `transferInstruction` je vhodné volání delegovat na vlastního potomka třídy `AbstractFrameModelingVisitor` (na něm stačí zavolat metody `setFrameAndLocation` a následně `analyzeInstruction`). Bude se pravděpodobně jednat o nejdůležitější třídu analýzy, která zajistí modifikaci faktů podle dané instrukce, opět s využitím vzoru návštěvník. Výchozí implementace už zajišťuje přesun

faktů mezi proměnnými a zásobníkem při instrukcích typu `load` a `store` a pro ostatní instrukce aspoň modeluje odebírání a vkládání správného počtu hodnot na zásobník. Při vkládání použije hodnotu faktu definovanou metodou `getDefaultValue`. Chování pro každou instrukci lze opravit předefinováním odpovídající metody (např. `visitLDC` pro instrukci `ldc`), je ale vždy nutné odebrat či vložit správný počet hodnot na zásobník podle specifikace JVM. Závislosti zmíněných tříd lze vidět z obrázku 4.2

Aby byla analýza funkční, je nutné ještě implementovat rozhraní `IMethodAnalysisEngine`, především metodu `analyze`, která vytvoří instance tříd pro analýzu a tok, vše spustí a vrátí výsledek. Poslední nutnou třídou je implementace rozhraní `IAnalysisEngineRegistrar` s metodou `registerAnalysisEngines`, která vytvoří instanci implementaci `IMethodAnalysisEngine` a zaregistruje ji s instancí `IAnalysisCache`. To zajistí, že při prvním vyžádání toku je analýza spuštěna a uložena do paměti, což je důvodem relativně vysoké paměťové náročnosti nástroje. Aby byla použita samotná implementace `IAnalysisEngineRegistrar`, musí být třída přidána do `findbugs.xml` pod elementem `EngineRegistrar`. Veškerá zmíněná analýza probíhá pouze lokálně v rámci jedné metody, pro pokročilou detekci závislou na více metodách (nebo i třídách) se předpokládá vlastní sumarizace dříve navštívených metod a její využití pro globální identifikaci chyby. Z tohoto důvodu FindBugs umožňuje chytře zvolit pořadí analyzovaných metod a to topologicky podle grafu vzájemného volání – pokud metoda a ve svém těle volá metodu b, pak dříve proběhne analýza metody b (je-li to možné, např. ve volání nejsou cykly).

5 Implementace taint analýzy

Mechanismus detekce injekčních zranitelností využívá možnosti FindBugs (viz sekce 4.3) a principy z kapitoly 3 (zejména podsekce 3.5.2). Skládá se ze dvou hlavních částí:

- Taint analýza (viz dále) proběhne před samotnou detekcí a pro každou lokaci v kódu uloží informace pro hodnoty na zásobníku a proměnných, dle kterých už lze relativně snadno rozhodnout o přítomnosti chyby.
- Hierarchie detektorů (viz sekce 5.3) využívá výsledky taint analýzy, v závislosti na konkrétní chybě je zpřesňuje a provádí reportovaní chyb s informacemi o toku dat.

5.1 Model bezpečnosti dat

Původní detektory injekcí ve FindSecurityBugs používaly existující `ConstantDataflow` s faktem `Constant` (uvnitř `ConstantFrame`), který pomocí metod jako `getConstantString` umožňuje zjistit konkrétní hodnotu. Pokud je hodnota známá, zjevně se nejedná o taint zdroj, analýza ale nezachytí případy, kdy sice hodnota známá není, přesto je zjevně konstantní, uvažme např. následující ukázku kódu (konkrétní podmínka a dotaz jsou nepodstatné):

```
String dotaz;
if (pouzitSql1) {
    dotaz = "sql1";
} else {
    dotaz = "sql2";
}
sqlStatement.executeQuery(dotaz);
```

V místě spuštění dotazu bude konstanta neznámá a bude reportována chyba, přestože k SQL injekci nemůže dojít. Důvodem je metoda `merge` třídy `Constant` – v každé z větví podmínky je hodnota známá, při spojení faktů ale hodnota zůstane nastavena jen tehdy, pokud se spojované hodnoty rovnají (jinak je výsledkem `NOT_CONSTANT`). Násle-

dující kód by také nebyl úspěšně analyzován, přestože hodnota ve skutečnosti známá je:

```
String dotaz = null;
if (pouzitSql) {
    dotaz = "sql";
}
if (dotaz != null) {
    sqlStatement.executeQuery(dotaz);
}
```

Důvodem je to, že analýza už za prvním podmíněným výrazem označí hodnotu jako neznámou (může být `null`) a protože je analýza pouze dopředná, druhý podmíněný výraz nic nezmění.

Místo použití `ConstantDataflow` tedy implementujeme vlastní `TaintDataflow`, kde fakt bude reprezentovat třída `Taint` (uvnitř `TaintFrame`) s hlavními stavy *tainted* (nebezpečný ve smyslu taint analýzy), *unknown* (neznámý), *safe* (bezpečný, opak *tainted*) a *null* (pro hodnoty rovné `null`). Z implementačních důvodů je použit ještě stav *invalid*, pro zjednodušení ho dále nebudeme zmiňovat. Vytváření instancí `Taint` řídí zejména třída `TaintFrameModelingVisitor` volaná v metodě `transferInstruction` třídy `TaintAnalysis`. Instance se stavem *safe* je na zásobník vložena např. v případě instrukce `ldc`, hodnota *null* je použita pro instrukci `aconst_null`. V případě jedné z instrukcí `invoke` je vytvořen plný název volané metody (tj. včetně jména třídy, balíku a signatury) a je zjištěno, zda toto volání patří mezi taint zdroje (viz sekce 3.3) resp. zdroje bezpečných dat. Pokud ano, je použit stav *tainted*, resp. *safe*, jinak může být výsledný stav zjištěn na základě parametrů (viz podsekce 5.1.1), popř. podle volání rodičovské třídy (viz podsekce 5.2.2). V případě neznámého volání či instrukcí nepodstatných pro analýzu je použit výchozí stav *unknown*. Spojování těchto hodnot probíhá vždy konzervativně – výsledný stav je vždy nejméně tak nebezpečný jako každá ze vstupních hodnot (např. spojením stavů *unknown* a *safe* bude opět *unknown*). Hodnota `null` je z hlediska taint analýzy bezpečná a proto spojení stavu *null* a *safe* bude *safe*. Pokud bychom stavy uspořádali jako $tainted > unknown > safe > null$, pak výsledkem spojením stavů bude vždy jejich maximum. Takové nastavení zajistí, že v obou dříve zmíněných ukázkách bude proměnná s dotazem ve stavu *safe*. Dalším místem, kde může být vlo-

žena instance `Taint` se stavem *tainted*, je metoda `initEntryFact` třídy `TaintAnalysis`, která je zodpovědná za prvotní naplnění instance `TaintFrame` na začátku metody. Parametry metody (část proměnných) jsou označeny za nebezpečné, pokud obsahují vybrané anotace nebo se jedná o vstupní bod programu – veřejnou statickou metodu `main` s polem řetězců jako argument.

5.1.1 Modelování volání metod

Kromě taint zdrojů existují naopak metody, jejichž výstup lze vždy považovat za bezpečný a v analýze dále postupovat stejně jako pro konstanty. Například metody `toString` u objektových číselných typů (jako `Integer` obalující primitivní typ `int`) budou obsahovat jen číselné znaky, které nemají speciální význam v žádné z injekčních chyb. Také kontrolujeme typ návratové hodnoty metod a pokud se jedná o některou z předdefinovaných bezpečných tříd, je automaticky použit stav *safe*, i pokud je metoda neznámá, stejně tak u čtení bezpečných atributů třídy. U mnohých volání však bezpečnost návratové hodnoty závisí na jejích parametrech (včetně samotné instance, která je používaná jako implicitní parametr každé nestatické metody). Např. metody `trim` či `toLower` třídy `String` přímo zachovávají stav na výstupu, metoda `concat` pak musí provést spojení stavů instance a parametru, to můžeme provést stejným způsobem jako při spojování větví v CFG.

Analýzu řetězců usnadňuje fakt, že třída `String` je neměnitelná (tzv. *immutable*), tzn. každá její metoda vytvoří nový řetězec a původní zůstane stejný (tedy i jeho stav). Velmi časté je ale spojování řetězců pomocí operátoru +, např.:

```
sqlStatement.executeQuery(
    "SELECT * FROM t WHERE id=" + number
);
```

Takové případy nejsou přeloženy na volání `concat`, ale z důvodu efektivity je vytvořena nová instance měnitelné třídy `StringBuilder`, na ní následně zavolány metody `append` pro každou část oddělenou znakem + a na závěr je vytvořen řetězec pomocí volání `toString`. Aby byla analýza funkční, musí být instance `Taint` vytvořená spojením informace o instanci třídy `StringBuilder` a parametru `append` nejen

uložena na zásobník jako návratová hodnota, ale také musí být změněn stav samotné instance. S tímto se však pojí několik problémů:

1. Ukázalo se, že není vhodné instanci `Taint` přímo modifikovat a využít toho, že na stejný objekt je odkazováno z více míst paměti, ale naopak zajistit, že je každá instance unikátní, jinak může selhat analýza cyklů, kterou provádí FindBugs. Proto jsou v kódu na mnoha místech vytvářeny defensivní kopie a instance `Taint` jsou modifikovány jen lokálně před uložením do `TaintFrame`.

2. Je-li změněna informace pro instanci měnitelné třídy (resp. vytvořena nová odvozením), která byla již dříve uložena v lokální proměnné, kromě změny na zásobníku musí být také aktualizována hodnota odpovídající proměnné. Proto při každé instrukci typu `store` zaznamenáme do instance `Taint` také index proměnné, ve které je hodnota uložena. Když je později hodnota načtena, je index uchován i při manipulacích ze zásobníkem a pokud dojde ke změně odpovídající instance `Taint`, je aktualizována stejným způsobem i informace o odpovídající proměnné.

3. Aby byl mechanismus funkční, musí být vhodně definována bezpečnost nově vytvořené instance. Při použití bezparametrického konstruktoru třídy `StringBuilder` to např. bude stav *safe* (odpovídá prázdnému řetězci). Při vytvoření nové instance budou pravděpodobně v bytekódu po sobě následovat instrukce `new`, `dup` a `invokespecial` – to znamená, že nová instance uložená na zásobník je nejprve zduplikována a následně volaný konstruktor odebere vrchol zásobníku jako svůj implicitní parametr, ale žádnou hodnotu už neuloží zpět (konstruktory nemají návratovou hodnotu). Z tohoto důvodu je nutné aktualizovat instanci `Taint` ještě pro hodnotu na zásobníku o jedna níž než je obvyklé, aby mohla být propagována k dalším instrukcím.

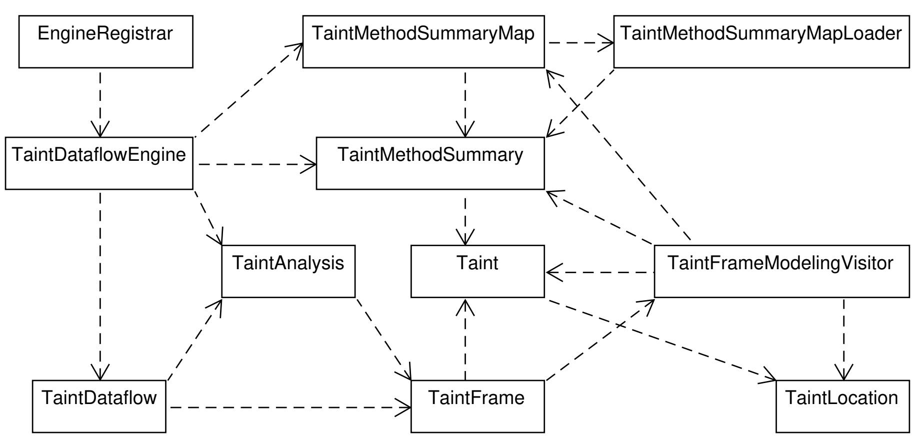
4. V souvislosti s měnitelnými třídami je potřeba si uvědomit, že výstup metody není jen návratová hodnota, ale mohou být měněny také parametry. Aby byly reportovány všechny potenciální chyby, je nutné pro neznámé metody předpokládat, že i svoje

> parametry mění neznámým způsobem. To provedeme tak, že instanci `Taint` pro každý z parametrů nahradíme jeho spojením s hodnotou *unknown*, tedy nebezpečné stavy zůstanou nebezpečnými, ale bezpečné budou neznámé. Abychom ale omezili zbytečné označování bezpečných hodnot za neznámé, dojde nejprve ke zpracování signatury a jsou zjištěny indexy parametrů s referenčními typy, které zároveň nejsou na seznamu známých neměnitelných tříd (zejména `String`). Dalším opatřením je, že pro známé metody není tato transformace prováděna a metodu lze přidat čistě z důvodu, aby parametry nebyly modifikovány – takto je například obsaženo množství volání pro logování (kde parametry jsou často typu `Object`, ale je z nich pouze čteno).

Stejně jako pro třídy `String` a `StringBuilder` jsou obsaženy i konfigurace metod pro `StringBuffer` a méně známé `Appendable`, `CharSequence` či `StringTokenizer`. Význačnou skupinu tvoří informace pro analýzu bezpečnosti kolekcí (implementace rozhraní `Collection` a `Map` z balíku `java.util`) – prázdnou kolekci rovněž inicializujeme stavem *safe* a vložení hodnoty z taint zdroje změní přiřazený stav instanci též na *tainted* (opětovného označení za *safe* pak docílí pouze metoda `clear`). Každý prvek v kolekci je považován za stejně nebezpečný, jako značí stav celé kolekce, to je nutná aproximace pro zachování efektivní analýzy, v praxi ale jen zřídka může dojít k situaci, kdy je hodnota z kolekce obsahující nějaké nebezpečné vstupy zaručeně bezpečná. Stejný princip se uplatní i v případě polí. I s mechanismem zabraňujícím nutnosti konfigurovat metody potomků (viz podsekce 5.2.2) obsahuje analýza informace o stovkách metod.

5.1.2 Značení dat ošetřených konverzí

Jak už bylo naznačeno v sekci 3.5, aby se omezilo reportování volání s ošetřenými daty, specifické metody dále aktualizují instanci `Taint` přidáním informace o provedené konverzi. Jsou definovány *tagy* (implementačně výčtový typ), které značí provedení konverze určitého znaku (např. `CR_ENCODED`) nebo přímo konverze zabraňující určité chybě (např. `XSS_SAFE`), v budoucnu by mohly přibýt také jiné binární příznaky nesouvisející s konverzí. Toto jsou jediné závislosti taint analýzy na konkrétních chybách, jinak je analýza obecná a detektory



Obrázek 5.1: Diagram tříd implementované taint analýzy

mohou být tvořeny bez její modifikace. U známých konverzních metod (viz podsekce 3.4.1) definujeme tagy, které daná metoda přidává či odebírá a instance `Taint` uchovává informaci, které tagy aktuální hodnota obsahuje. Na detektorech záleží, jak informaci o přítomnosti tagů využijí při reportování chyb. Kromě toho analýza podporuje také jednoduchou konverzi pomocí náhrady znaků (metody `replace` a `replaceAll` třídy `String`). Protože je nutné přečíst hodnotu argumentu (obsahující znaky, které budou nahrazeny), instance `Taint` ukládají i hodnotu konstanty (podobně jako v `ConstantAnalysis`). `TaintFrameModelingVisitor` pak obsahuje mapování znaku na konkrétní tag (pro každý z těchto znaků vyskytujících se v konstantním parametru metody je přidán odpovídající tag k instanci `Taint` pro návratovou hodnotu).

5.2 Konfigurace metod a automatické odvozování

S předchozího textu (podsekce 5.1.1) je zřejmé, že analýza spoléhá na množství informací o známých metodách – ty jsou uloženy v mapování `TaintMethodSummaryMap`, kde klíčem je plný název metody (tzn. včetně třídy, balíku a signatury) a hodnotou instance třídy `TaintMethodSummary`, která uvnitř obsahuje především instanci `Taint` urču-

jící stav (většinou návratové hodnoty). Závislosti mezi třídami taint analýzy jsou zaneseny v obrázku 5.1. Kvůli tomu, že výstup metody může být závislý na vstupních parametrech, obsahuje třída `Taint` také kolekci indexů těchto parametrů. Pro případy, kdy nestačí výslednou instanci vložit na zásobník, ale přenést na objekt, který metodu volá, nebo jeho parametry, obsahuje `TaintMethodSummary` kolekci indexů do zásobníku, kde je nutné aktualizovat stav.

5.2.1 Formát konfiguračních souborů

Pro snazší a přehlednější přidávání informací o dalších metodách obsahuje `TaintMethodSummary` statickou metodu `load`, která vytvoří novou instanci podle konfigurace dané řetězcem. `TaintMethodSummaryMap-Loader` pak slouží k načtení celého mapování ze souboru, kde na jednom řádku je plné jméno metody následované znakem `:` a konfiguračním řetězcem (řádky začínající znakem `-` jsou považovány za komentář a přeskočeny, stejně jako prázdné řádky). Protože je formát relativně komplikovaný, před načtením je každý řádek syntakticky zkontrolován pro omezení překlepů.

Povinnou částí konfiguračního řetězce jsou indexy do zásobníku operandů (oddělené čárkou), které určují položky pro sloučení k vytvoření nové instance `Taint`. Vrchol zásobníku má index 0 a odpovídá poslednímu parametru metody s n parametry, první má obvykle index $n - 1$ a na indexu n se nachází instance objektu volající nestatickou metodu. Jen obvykle proto, že typy `double` a `long` zabírají na zásobníku dvě pozice (souvisí s použitím knihovny BCEL a formátem class souborů), při přechodu o parametr doleva je tedy nutné zvýšit index o dva. Místo indexu je možné uvést do konfigurace také přímo stav (např. `SAFE`) a to i v kombinaci s indexy, pak bude výsledkem spojení všech těchto hodnot, proto obsahuje třída `Taint` ještě další atribut se stavem. Lze použít např. konfiguraci `0,UKNOWN`, která značí, že je metoda závislá na vrcholu zásobníku, ale také na volání neznámé metody – pro vstup stavu *tainted* tedy bude výstup *tainted*, ale pro *safe* bude *unknown*. U ručně konfigurovaných metod tato možnost nebude příliš využívána, ale podstatná je především pro automaticky odvozené konfigurace (viz podsekce 5.2.3). Ručně přidaná konfigurace má informační hodnotu i v případě, že je pouze `UKNOWN` – říkáme tím, že parametry metody nejsou uvnitř měněny.

Vzniklá instance `Taint` je ve výchozím stavu uložena na zásobník (nejedná-li se o konstruktor nebo metodu typu `void`). Pokud ji dále chceme použít i na hodnoty hlouběji na zásobníku (na parametry či samotný objekt), jsou tyto indexy specifikovány za znakem `#` a odděleny čárkou. U konstruktorů je pak nutné přidat ještě jeden index níže (důvod je uveden v podsekci 5.1.1), např. `StringBuilder` inicializovaný řetězcem přebírá jeho stav a má konfiguraci `0#1,2` (specifické případy, kdy je na zásobníku pouze jedna hodnota, jsou ošetřeny). Poslední (rovněž nepovinnou) částí konfiguračního řetězce je seznam tagů (oddělený čárkou) uvedených za znakem `|`. Názvu každého tagu předchází buď znak `+`, nebo `-`, podle toho, jestli metoda tag přidává, nebo odebírá, např. konfigurace `0|+XSS_SAFE` značí přebrání stavu z vrcholu zásobníku a přidání označení, že je hodnota bezpečná pro použití v HTML. Konfigurační soubory pro mnoho metod jsou obsaženy přímo uvnitř nástroje, pomocí parametru `findsecbugs.taint.customconfigfile` lze zvenku konfiguraci přepsat nebo přidat vlastní metody bez nutnosti rekompilace nástroje.

5.2.2 Vliv dědičnosti a polymorfismu

Při konstrukci plného názvu metody pro vyhledání odpovídající instance `TaintMethodSummary` je použit název třídy (či rozhraní) z instrukce typu `invoke`. Kvůli dědičnosti a polymorfismu ale nemusí dojít k volání metody dané třídy (ani nemusí existovat) – zavolán může být potomek třídy deklarovaný jako jeho rodič (popř. implementované rozhraní) nebo také rodič třídy, pokud daná třída metodu sama neimplementuje. Protože tyto informace jsou vyhodnocovány až za běhu, nelze zcela spolehlivě zjistit kód, který bude skutečně vykonán. Jedním z opatření je uložení názvu skutečně použité třídy do instance `Taint` při volání instrukce `new` a přednostní použití této třídy při vyhledávání konfigurace metody.

Druhým je vyhledávání metod postupně pro všechny rodiče a také implementovaná rozhraní, dokud není nalezena nějaká se známou konfigurací (nebo není použito výchozí chování se stavem *unknown*). Nalezení konkrétní třídy podle názvu umožňuje FindBugs pomocí třídy `Repository` a zjištění rodiče, resp. implementovaných rozhraní pomocí metod `getSuperClasses`, resp. `getAllInterfaces`. V konfiguračních souborech stačí uvádět metodu stejného názvu (a signatury)

jen u tříd či rozhraní, které jsou nejvýše v hierarchii dědičnosti (pokud by měly stejné konfigurace). To podstatně zkrátí konfiguraci metod přítomných v hlubších hierarchiích (např. pro kolekce) a umožní případně konfiguraci snadno změnit na všech místech současně. Další výhodou je, že jsou tak automaticky pokryti potomci a implementace neznáme v době psaní konfigurace (např. i nové či nestandardní kolekce pravděpodobně budou implementovat rozhraní `Collection`). Pro konfigurace metod z externích knihoven však raději uvádíme i metody potomků, protože pokud by daná knihovna nebyla přidána k analýze (z důvodu efektivity či neznalosti), rodič by nebyl nalezen a došlo by zbytečně ke snížení přesnosti analýzy.

Shrňme nyní pro přehlednost kroky, které se dějí ve třídě `TaintFrameModelingVisitor` při analýze instrukce typu `invoke`. Popis je zkrácen o přidávání informací pro účely přehlednějšího reportování chyb a ladění, ošetření výjimečných stavů a implementační detaily:

1. Ze signatury metody je zjištěn návratový typ a jedná-li se o bezpečný objektový typ (`Integer` apod.), je i výsledek bezpečný.

2. Pokud instance `Taint` odpovídající položce na zásobníku pro instanci volané nestatické metody obsahuje informaci o skutečném typu objektu, považuje se dále za třídu, ve které se pravděpodobně nachází aktuálně volaná metoda, jinak se využije přímo parametr instrukce typu `invoke`.

3. Z názvu třídy (včetně balíku), názvu metody a signatury je sestaven identifikátor metody a je zjištěno, zda pro ni existuje instance `TaintMethodSummary`.

4. Pokud se jedná o metodu `replace` či `replaceAll` třídy `String`, jsou vyšetřeny její parametry a případně přidány tagy značící konverzi daných znaků.

5. Pokud metoda není známá, je mapování s instancemi `TaintMethodSummary` prohledáno na výskyt klíčů, kde název třídy je postupně nahrazován názvy všech rodičovských tříd a implementovaných rozhraní.

6. Jedná-li se o neznámý konstruktor, je vytvořena instance `TaintMethodSummary`, která nastaví stav vytvářeného objektu na *unknown* (to je nutné, protože nová instance má stav *safe*).

7. Je vytvořena instance `Taint` podle konfigurace (popř. výchozí), případně je provedeno sloučení pro vybrané vstupní parametry, jsou také přidány či odebrány tagy značící konverzi.

8. Pokud se nejedná o metodu uvedenou v konfiguraci či jejího rodiče, je pro každou instanci `Taint` odpovídající položce na zásobníku pro parametr metody, jehož typ je referenční a není uveden v seznamu známých neměnitelných typů, provedena aktualizace stavu spojením se stavem *unknown*.

9. Pokud instance `TaintMethodSummary` aktualizuje také stávající hodnoty na zásobníku (tzn. ne pouze návratovou hodnotu), jsou odpovídající instance `Taint` spojeny s informací o nebezpečnosti metody a výsledek je také propagován do odpovídajících lokálních proměnných.

10. Z modelu zásobníku se odebere počet hodnot podle parametrů a na vrchol se vloží instance `Taint` získaná z instance `TaintMethodSummary` (nebo výchozí hodnota se stavem *unknown*).

5.2.3 Automatické odvozování konfigurací metod

Veškerá taint analýza probíhá pouze lokálně v rámci jedné metody, tok dat ale v aplikacích často prochází skrz více metod i tříd. Proto analýza kromě využití předem nakonfigurovaného chování také odvozuje a ukládá instance `TaintMethodSummary` za běhu. Za tímto účelem si instance `Taint` se stavem *unknown* udržují kolekci indexů vstupních parametrů právě analyzované metody, na jejichž stavu je výsledná instance závislá. Toho je docíleno tak, že na začátku analýzy je tato kolekce inicializována pouze indexem parametru pro odpovídající část proměnných a při každém spojení instancí `Taint` je výstupní množina dána sjednocením množin vstupních instancí. V případě sloučení s instancí, která není závislá na argumentech, je provedeno také sloučení stavu této instance a dalšího vnitřního neparametrického stavu druhé instance, takže výsledek nemusí být závislý pouze na

argumentech, ale i na dalších konkrétních stavech (zřejmě si stačí pamatovat jen jejich spojení – nejnebezpečnější stav).

Metoda s referenčním návratovým typem je běžně zakončena instrukcí `areturn`. Instanci `Taint`, která se v době jejího volání nachází na vrcholu zásobníku, lze přidat do `TaintMethodSummary` a uložit do mapování s původně jen ručně nakonfigurovanými metodami pod klíčem aktuálně analyzované metody. Protože metoda může vrátit hodnotu na více místech, dochází samozřejmě ke spojení všech možných návratových hodnot. Takto mohou být odvozeny nové taint zdroje či bezpečné zdroje (pokud hodnota nezávisí na argumentech), ale i metody, jejichž nebezpečnost závisí na vstupech, a metody provádějící konverzi. Jedině přenos stavu na další položky zásobníku zatím není automaticky odvozován, naopak oproti ruční konfiguraci je navíc ukládán název skutečné třídy instance a také taint zdroje, na kterých je metoda závislá. Výsledek analýzy metody je uložen, pokud zatím neexistuje (ruční konfigurace má přednost) a pokud má nějakou informační hodnotu (např. instance `Taint` není výchozí hodnota). Protože metody i třídy analyzujeme v pořadí podle grafu volání, i v jednom průchodu je dosaženo relativně přesné globální analýzy. Odvozené instance `TaintMethodSummary` mohou být vypsány ve formátu konfigurace, takže lze snadněji ověřit správnost taint analýzy. Toho bylo využito pro testování během vývoje – po každé změně můžeme spustit nástroj nad neměnným rozsáhlým kódem a zkontrolovat všechna odvození, u kterých došlo ke změně od minulého běhu, ani významné změny se totiž nemusí projevit v počtu nahlášených chyb.

5.3 Detektory využívající taint analýzu

Dosud zmíněné třídy analýzy pouze uloží potřebná fakta o všech lokacích v kódu, pro samotné reportování využíváme hierarchii detektorů. Nejvýše je `AbstractTaintDetector`, který přímo implementuje rozhraní `Detector`, pro každou metodu získá instanci `TaintDataflow` a pro každou lokaci CFG volá svoji abstraktní metodu `analyzeLocation` s odkazem na analyzovanou instrukci, instancí `TaintFrame` a dalšími parametry. Přímým potomkem je `AbstractInjectionDetector`, který rozšiřuje analýzu toku dat pro konkrétní typy chyb (viz podsekce 5.3.1) a provádí reportování chyb způsobem, který umožní zobrazit nejen

zranitelný taint sink, ale pokud možno také taint zdroje a místa kódu, která značí cestu mezi taint zdroji a konečným voláním. K tomu slouží instance dosud nezmíněné třídy `TaintLocation`, která globálně určuje místo v kódu (pomocí odkazu na `MethodDescriptor` a pozici). Ty jsou vytvářeny na vhodných místech během taint analýzy a vkládány do instancí `Taint` spolu s informací, zda se připojují data se stavem *tainted*, nebo pouze *unknown*. Pokud byly nějaké lokace přidány pod stavem *tainted*, využijí se při reportování pro přehlednost pouze ty. Předtím jsou převedeny na konkrétní řádky, uspořádány a jsou odstraněny duplicity.

K vytvoření konkrétního detektoru je nutné implementovat metodu `getInjectionPoint`, která pro konkrétní volání instrukce typu `invoke` vrátí v instanci `InjectionPoint` případný typ chyby a indexy argumentů, které mohou sloužit jako taint sink. Pro velmi snadnou tvorbu nových injekčních detektorů slouží ještě abstraktní třída `BasicInjectionDetector`, která s využitím třídy `SinksLoader` načítá potenciálně zranitelná volání z konfigurace – na každém řádku je plné jméno metody a za znakem `:` indexy argumentů oddělené čárkou. Jejím potomkům (např. `SqlInjectionDetector`) pak stačí v konstruktoru zavolat metodu `loadConfiguredSinks` s názvem konfiguračního souboru a typem chyby. Často je také vhodné použít vlastní implementaci metody `getPriority`, která určuje, s jakou prioritou bude nahlášena chyba pro danou instanci `Taint`. Výchozí chování (ze třídy `AbstractInjectionDetector`) používá vysokou prioritu pro stav *tainted*, normální pro *unknown* a jinak chyby nehlásí. V této metodě mohou potomci také zjistit přítomnost tagů pro konverzi a případně danou chybu nereportovat (přestože stav není *safe*) nebo pouze s nízkou prioritou (není totiž vždy možné automaticky ověřit použití správné konverze pro danou situaci).

5.3.1 Změna priority podle volání metod

Taint analýza sice dokáže odvodit nové konfigurace, problém však nastane, pokud je taint sink volán s hodnotou, jejíž stav závisí na argumentu právě analyzované metody. Přímočarým řešením by mohlo být uložit metodu jako odvozený taint sink, kde sledovaným argumentem by byl právě argument, na kterém vnitřní taint sink závisí, to by však neumožnilo přehledné oddělení taint analýzy od samotné detekce

a zkomplikovalo reportování chyb. Taint sink závislý na argumentu místo toho nebude ihned reportován, ale budou odchytávány volání metody, která taint sink obsahuje, a pro hodnoty nezávislé na dalších argumentech bude priorita chyby aktualizována (podle metody `getPriority`). Pokud je i metoda obsahující taint sink volána s argumenty, které závisí na argumentech metody obsahující toto volání, je dále sledováno i volání této metody a při volání s instancemi `Taint` nezávislých na argumentech je změna propagována až po skutečný taint sink. Stejně jako při taint analýze je i v detektorech použito pořadí analýzy metod podle grafu volání. Podobně je také před zjištěním, zda se jedná o konkrétní taint sink, využita informace z objektu `Taint` o skutečném typu instance volající sink a provedeno hledání v rodičovských třídách a implementovaných rozhraních.

Protože ale volání metody nelze vždy spolehlivě detekovat (např. při komplikovaných voláních využívajících polymorfismus nebo použití reflexe), nebylo by vhodné chybu vůbec nereportovat, pokud volání odpovídající metody není nalezeno. Navíc může být žádoucí hledat i chyby v momentálně nevyužívaném kódu. Pokud je ale detekován nenulový počet volání s bezpečnou hodnotou a žádná s nebezpečnou či neznámou, předpokládáme, že volání detekovat dokážeme a o zranitelnost se pravděpodobně nejedná. V takovém případě je chyba reportována pouze s nízkou prioritou. Informace o tom, jaká volání byla nalezena, je navíc přidána do výsledků analýzy. Pro každý taint sink na konkrétním místě v kódu je uložena instance `InjectionSink`, která obsahuje všechny informace a vytváří instanci `BugInstance` pro reportování chyby. `AbstractInjectionDetector` obsahuje mapování metod na množinu instancí `InjectionSink`, které její argumenty ovlivňují. Konkrétní indexy argumentů a další informace (např. tagy pro provedení konverze) jsou uloženy v mapování instancí `MethodAndSink` (s odkazem na název metody a `InjectionSink`) na instanci `Taint`. Na konci analýzy jsou sjednoceny množiny instancí `InjectionSink` z mapování a pro každou jedinečnou hodnotu provedeno reportování s aktualizovanou prioritou.

5.4 Vymezení rozsahu práce

Protože rozšiřující sada detektorů FindSecurityBugs nebyla vyvíjena od počátku a během vylepšování do kódu v menší míře zasahoval také původní autor (*Philippe Arteau*), následující sekce vymezuje úpravy provedené v rámci této práce. Všechny příspěvky kódu i s popisem a diskusí o jejich začlenění jsou zveřejněny na stránce projektu na serveru GitHub. [51] Podle tohoto systému bylo mnou přidáno téměř 13 tisíc řádků kódu ve 130 příspěvcích (*commit*).

Náplní práce byla implementace taint analýzy a vylepšení detektorů pro injekční zranitelnosti:

- Samotný mechanismus, tak jak je popsán v sekcích 5.1 a 5.2, byl vytvořen zcela od počátku, skládá se z deseti tříd
- Součástí taint analýzy jsou ručně vytvořené konfigurace pro více než 750 volání pokrývajících jak základní Java API, tak vybrané knihovny třetích stran
- Zcela nově byla vytvořena také hierarchie detektorů popsaná v sekci 5.3 (převzata byla pouze jednoduchá třída `InjectionPoint`), zlepšuje výsledky analýzy a umožňuje snadnou tvorbu nových detektorů
- Původní detektory měly každý taint sink uveden v konstantách v kódu rozdělen na název třídy, metody a signaturu, všechny byly přepsány, aby zranitelná voláni načítala přehledně z konfigurace
- Při přepisu detektorů byly nalezeny a odstraněny chyby, kvůli kterým nebyla detekována některá volání pro LDAP injekci a injekci příkazu (přestože se vyskytovaly v kódu)
- Nově byly vytvořeny detektory pro externí kontrolu konfigurace (CWE-15), rozdělování HTTP odpovědi (CWE-113) a falšování záznamů logu (CWE-117)
- U detektorů ostatních injekčních chyb byla přidána další volání známá jako taint sink, jejich počty jsou zobrazeny v tabulce 5.1

- Pro většinu kódu byly vytvořeny také odpovídající testy automaticky spouštěné při sestavování aplikace

Původní autor provedl následující zásahy:

- Do taint analýzy byla přidána možnost ukládat další informace pro účely ladění
- Byla přidána detekce taint zdroje přes anotace argumentu metody
- Byla vyextrahována 11. třída taint analýzy pro načítání konfigurací metod
- Byly vytvořeny detektory pro XSS využívající taint analýzu (v rámci práce ale byly vylepšeny)

Naopak mimo hlavní náplň této práce mnou byly provedeny následující vylepšení detekce ostatních bezpečnostních problémů:

- Byl vytvořen detektor pro hesla a kryptografické klíče, které se přímo vyskytují v kódu (jsou tzv. *hard coded*), chyba se pod CWE číslem 798 vyskytuje na 7. místě CWE-Top25 (viz sekce 2.1.1), konstanty jsou detekovány v 35 voláních a dále heuristicky podle názvu atributu třídy
- Byla vylepšena detekce použití kryptografických algoritmů bez zajištění integrity (není-li specifikován mód operace, použije se výchozí ECB[1], což nebylo při detekci reflektováno)
- Byla přidána méně známá volání pro detekci požití kryptograficky slabých hašovacích funkcí
- Byl sjednocen popis chyb a ke všem byly přidány CWE identifikátory, které jsou pak dostupné v rozšířeném exportu výsledků analýzy

1. *Electronic Codebook*, nejjednodušší mód pro symetrické blokové šifry

CWE	FSB 1.4.0	FSB 1.4.6	Přidáno
15	0	1	1
22	11	12	1
78	8	14	6
79	0	38	4
89	17	160	24
90	4	45	9
113	0	11	11
117	0	337	337
601	2	6	4
643	2	29	27
celkem	44	653	424

Tabulka 5.1: Počty typů detekovaných volání pro injekční chyby
Tabulka zobrazuje, kolik volání známých jako taint sink byla analýza ve Find-SecurityBugs schopná detekovat v původní a vylepšené verzi. Sloupec *Přidáno* značí počet přidaný v rámci této práce (zbytek rozdílu přidal původní autor rozšíření).

6 Spuštění a vyhodnocení úspěšnosti detekce

Je důležité ověřit, jaký je praktický dopad implementovaných vylepšení. Tato kapitola srovnává úspěšnost původní a nové verze FindSecurityBugs a také samostatného nástroje FindBugs s využitím testovací sady (viz sekce 6.2). Rozbor výsledků také umožňuje odhalit některé limity analýzy (viz sekce 6.3).

6.1 Konfigurace a spuštění

FindSecurityBugs ke spuštění vyžaduje původní FindBugs pro požadované prostředí (viz sekce 4.1). V některých případech (FindBugs pro IntelliJ IDEA nebo SonarQube) je FindSecurityBugs přímo součástí instalace (nemusí být ale dostupná nejnovější verze) a stačí jej povolit v nastavení, ve zbylých případech je potřeba stáhnout Java archiv z oficiálních stránek [3] (popř. specifikovat artefakt v systému Maven). Distribuován je s licencí LGPL stejně jako původní analyzátor. Pro použití se samostatným nástrojem FindBugs s vlastním rozhraním stačí vložit archiv do složky `plugin` v adresáři instalace, nalezené chyby jsou reportovány v kategorii *security* (spolu s původními detektory pro bezpečnostní chyby). Pro vyvolání grafického rozhraní lze spustit přímo `findbugs.jar` ve složce `lib` nebo zavolat obalující skript ze složky `bin` (pro Windows `findbugs.bat`), který usnadňuje předávání parametrů pro analýzu. V rozhraní lze snadno vytvořit nový projekt, nastavit cestu k přeložené podobě aplikace pro analýzu, použitým knihovnám (budou analyzovány, ale nebudou v nich hledány chyby) a zdrojovým kódům (pro zobrazení reportované chyby přímo v kódu). V souvislosti s taint analýzou lze FindSecurityBugs spustit s následujícími parametry:

- `findsecbugs.taint.taintedsystemvariables` umožňuje specifikovat, zda mají být systémové proměnné považovány za taint zdroje, výchozí hodnota je `false`, protože nástroj je častěji používán pro analýzu webových aplikací, kde lze odpovídající volání zpravidla považovat za bezpečná

- `findsecbugs.taint.customconfigfile` je zobecněním předchozího parametru – umožňuje libovolně přepsat či doplnit konfigurace metod souborem formátovaným dle podsekce 5.2.1
- `findsecbugs.taint.taintedmainargument` určuje, zda jsou argumenty vstupní metody `main` považovány za taint zdroje (výchozí hodnota je `true`)
- `findsecbugs.taint.outputsummaries` při nastavení na hodnotu `true` způsobí vytvoření souboru, který po skončení analýzy bude obsahovat všechny odvozené konfigurace metod, což je výhodné zejména při ladění analýzy
- `findsecbugs.injection.sources` umožňuje přidat také volání známá jako taint sink, jedná se ale o původní parametr se složitým formátem, je plánováno přidat parametr pro snadné načítání ve formátu, který používají konfigurovatelné detektory

6.2 Vyhodnocení na testovací sadě Juliet

Pro praktické ověření schopností analýzy vylepšených detektorů FindSecurityBugs a srovnání s jejich původní verzí a samostatným nástrojem FindBugs byla zvolena sada *Juliet Test Suite* pro jazyk Java (ve verzi 1.2). Jedná se o testovací sadu agentury NSA[1] vyvinutou přímo pro vyhodnocení schopností nástrojů pro statickou analýzu kódu, dostupná je ze stránek laboratoře NIST[2]. [4] Sada se skládá z více než 25 tisíc testovacích případů (*test cases*) rozdělených do 112 skupin podle identifikátoru CWE, z toho 12 skupin s celkem necelými 8 tisíci případy odpovídá chybám typu injekce.

Každý případ se skládá z jedné či více tříd a kromě chyby pokud možno obsahuje také odpovídající kód bez zranitelnosti. Úspěšnost budeme posuzovat podle *citlivosti*, tedy toho, zda nástroj v daném případu problém odhalí (*true positive*), ale také podle tzv. *specificity* – zda chyba nebude reportována v místě, kde se ve skutečnosti nenachází (*true negative* – nereportovaná neexistující chyba). Z těchto

1. *National Security Agency*, Národní bezpečnostní agentura, vládní organizace USA
2. *National Institute of Standards and Technology*, Nár. institut standardů a technologie

metrik se dají odvozovat další jako *přesnost (accuracy)* udávající poměr počtu správných rozhodnutí o přítomnosti dané chyby oproti rozhodnutím nesprávným, *preciznost (precision)* značící relativní četnost skutečných chyb mezi detekcemi nebo F_1 *skóre* vypočítané jako harmonický průměr citlivosti a preciznosti, které může být použito jako ukazatel celkové úspěšnosti pouze pomocí jedné hodnoty. [52]

Protože sada jako taint zdroj používá i systémové proměnné (ve výchozím stavu považovány za bezpečné), byla analýza spuštěna s parametrem `-Dfindsecbugs.taint.taintedsystemvariables=true`, který odpovídající volání přidá jako taint zdroje.

6.2.1 Výsledky

Výsledky pro samostatný FindBugs a FindSecurityBugs v různých verzích a nastaveních jsou pro jednotlivé typy chyb zobrazeny v tabulce 6.1 a průměrný výsledek i s dalšími metrikami pak v tabulce 6.2. Častou překážkou při nasazování automatické statické analýzy je vysoká míra falešných poplachů a FindBugs se proto soustředí na vysokou specificitu (hlášené chyby by měly být skutečné). Výjimkou je zjevně detektor pro SQL injekce (CWE-89), který za cenu nižší specificity odhalil většinu chyb. Důvodem je pravděpodobně častý výskyt a vážné důsledky zranitelností tohoto typu. Pokud připočítáme i detekce s nižší prioritou (nejsou zobrazeny v tabulce), citlivost se ještě zvýší, ale přesnost i preciznost klesnou na 50 %. Naopak FindSecurityBugs je zamýšlen jako nástroj pro pomoc při bezpečnostním auditu kódu a cílí na vysokou citlivost i za cenu nižší specificity. Bohužel detekce množství injekčních zranitelností byla implementována pouze pro případy, které se v testovací sadě nenachází (jiná rozhraní pro SQL, LDAP a XPath injekce, XSS v JSP), popř. vůbec. Celková přesnost detekce je shodou náhod pro FindBugs i původní FindSecurityBugs téměř stejná (53 %), F_1 skóre je však pro FindBugs výrazně nižší kvůli nízké citlivosti (19 % oproti 41 %).

Nová verze FindSecurityBugs (1.4.6) přistupuje k analýze všech chyb typu injekce jednotně (i když specifika chyb jako XSS vyžadovala přizpůsobení) a úspěšnost detekce jednotlivých typů chyb v sadě je shodná. Cílem bylo vytvořit analýzu s ideálně absolutní citlivostí, přitom ale podstatně omezit množství falešných poplachů a pro ještě lepší praktickou použitelnost také potvrdit vybrané detekce, u nichž

CWE-ID	Případů	FB	FSB 1.4.0	FSB 1.4.6 V	FSB 1.4.6 V+N
15	444	0 / 100	0 / 100	89 / 100	100 / 89
22*	888	5 / 100	100 / 16	89 / 100	100 / 89
78	444	0 / 100	100 / 0	89 / 100	100 / 89
79**	1332	4 / 100	0 / 100	89 / 100	100 / 89
89	2220	84 / 49	0 / 100	89 / 100	100 / 89
90	444	0 / 100	0 / 100	89 / 100	100 / 89
113	1332	4 / 100	0 / 100	89 / 100	100 / 89
601	333	0 / 100	100 / 30	89 / 100	100 / 89
643	444	0 / 100	0 / 100	89 / 100	100 / 89

Tabulka 6.1: Citlivost a specificita detekce injekčních chyb
Tabulka zobrazuje citlivost a specificitu analýzy v procentech pro jednotlivé chyby typu injekce, které umožňuje sada Juliet testovat, a to pro samostatný FindBugs bez rozšíření (v nejnovější verzi 3.0.1), původní verzi FindSecurityBugs (před zapojením se do vývoje) a vylepšenou verzi FindSecurityBugs s různým nastavením hranice priority (pouze vysoká, nebo vysoká a normální, detekce s nízkou prioritou jsou ignorovány). Hodnota *0 / 100* např. značí, že nebyla hlášena žádná chyba, naopak *100 / 0* značí detekci všech skutečných chyb, ale také považování všech bezpečných volání za chybu.
*Výsledek pro CWE-22 byl získán sjednocením skupin pro CWE-23 a CWE-36
**CWE-79 je dáno sjednocením CWE-80, CWE-81 a CWE-83
Pozn.: CWE-113 je kvůli nižší nebezpečnosti celkově reportováno s nižší prioritou, pro zjednodušení však budeme předpokládat stejný systém jako pro ostatní chyby.

je vysoká pravděpodobnost zneužitelné chyby. Ty chceme reportovat s vyšší prioritou pro možnost použití nástroje jako analyzátoru s vysokou precizností. Z výsledků vidíme, že všechny detekce s vysokou prioritou jsou chybami skutečnými (100% specificita a preciznost) a to při zachování vysoké citlivosti 89 %. Po přidání detekcí s normální prioritou je citlivost absolutní a přesto 90 % hlášení odpovídá skutečným chybám. Přesnost analýzy byla výrazně zvýšena na 95 % pro oba způsoby nastavení. Podobné jsou také hodnoty F_1 skóre, to vychází mírně vyšší pro nastavení zahrnující také problémy s normální prioritou (95 % oproti 94 %).

Pro hlubší porozumění výsledkům je však vhodné se seznámit se strukturou testovací sady. Testovací případy nebyly vytvořeny ručně,

	FB	FSB 1.4.0	FSB 1.4.6 V	FSB 1.4.6 V+N
Citlivost	11	33	89	100
Specificita	94	72	100	89
Přesnost	53	53	95	95
Preciznost	65	54	100	90
F_1 skóre	19	41	94	95

Tabulka 6.2: Srovnání celkové úspěšnosti detekce injekcí
Zde je zobrazena celková úspěšnost detekce pro FindBugs a různé verze FindSecurityBugs vypočtená z (neváženého) průměru citlivosti a specificity uvedených v tabulce 6.1, zbývající metriky jsou odvozeny z nich.

ale vygenerovány za použití šablon kombinujících různé varianty toku (viz podsekce 6.2.2) a funkční varianty chyby. V případě injekčních zranitelností funkční variantu určuje konkrétní taint sink a taint zdroj a testový případ existuje pro každou kombinaci, takže jejich počet je relativně vysoký. Pro úspěšnou analýzu případu proto musí nástroj rozpoznat taint zdroj (popř. bezpečný zdroj) i taint sink a porozumět použité variantě toku. Při testu falešných detekcí je jako zdroj použit konstantní řetězec, nebo je daný taint zdroj ošetřen konverzí. Množství chyb nebylo odhaleno už proto, že daný taint sink nebyl analýze známý (popř. neexistoval vůbec detektor pro odpovídající chybu). Tam, kde je citlivost v tabulce 6.1 nenulová, byl ale každý testovaný taint sink podporován (i když samostatný FindBugs přidal podporu pro SQL sink `executeBatch` teprve v poslední verzi).

Pro FindBugs je hlavním důvodem nízké citlivosti to, že kromě SQL injekce považuje za taint zdroj pouze metodu `getParameter` (třídy `HttpServletRequest`) a ignoruje dalších 11 testovaných zdrojů (detektor pro SQL injekci počítá se všemi nekonstantními řetězci, ale před reportováním chyby používá další heuristiku). Naopak FindSecurityBugs taint zdroje nerozlišoval a za bezpečné považoval pouze snadno detekovatelné konstanty, takže nenulovou specificitu má pouze v případech, kdy taint sink přímo přejímá hodnotu taint zdroje (resp. konstanty). Nová verze FindSecurityBugs zná všechny testované taint zdroje (a množství netestovaných) pro detekci s vysokou prioritou a obsahuje pokročilou detekci konstantních i ošetřených vstupů.

6.2.2 Varianty toku

Obtížnost analýzy toku někdy ovlivňuje už samotná funkční varianta – např. chyba typu procházení adresářem se v sadě nachází jako *relativní* (CWE-23) i *absolutní* (CWE-36), při první variantě ale taint sink obsahuje zřetězení zdroje a konstanty (navíc závislé na vyhodnocení podmínky pro vyhodnocení operačního systému), takže je detekce složitější. V testu některých chyb jsou bezpečné zdroje dat tvořeny konverzí, což je těžší případ než pouhé volání s konstantou. Kromě toho je ale každý testový případ ovlivněn jednou ze 37 variant toku.

Prvních 17 variant obsahuje jednoduché kontrolní větvení (vždy pravdivé podmínky a cykly) a protože kompilátor jazyka Java už provádí nejjednodušší optimalizace, 8 variant je reprezentováno stejným bytekódem (bez větvení). I další tyto varianty analýzu téměř neovlivňují, protože FindBugs i FindSecurityBugs zvažují vždy všechny varianty toku a navíc větvení obsahují obvykle už funkční varianty (ošetření výjimek apod.). Všechny analyzátory si rovněž poradí s kopií dat v rámci metody (varianta 31), pro ostatní varianty je ale nutné sledovat tok skrz více metod, popř. i tříd. Zde jsou např. taint zdroje zapouzdřeny v jiné metodě nebo je taint sink v metodě, která je teprve volána s nebezpečným argumentem (a to až přes 4 třídy ve variantě 54). Data nejsou vždy poslána přímo, ale někdy nejprve uložena do pole či kolekce, popř. přetypována. Obsaženo je také volání abstraktní metody, kde zneužitelnost chyby závisí na volbě instance konkrétní třídy (varianta 81). Jedinou variantu, kterou v některých detektorech dokáže analyzovat FindBugs a nová verze FindSecurityBugs zatím ne, je předání dat přes privátní atribut třídy. Tři varianty nedokázal správně analyzovat žádný detektor – data poslaná uvnitř instance vlastní třídy nebo v serializované podobě a předání přes veřejný atribut. Zbývajících 33 variant nová verze FindSecurityBugs správně rozpozná, pokud ignorujeme detekce reportované s nízkou prioritou (s nimi by byly hlášeny chyby i v bezpečných voláních pro 15 variant toku). Výsledky jsou zobrazeny v tabulce 6.3. Sada Juliet samozřejmě netestuje schopnost analýzy toku vyčerpávajícím způsobem a množství technik přidaných nově do FindSecurityBugs nevedlo automaticky k vyšší přesnosti v testu, pro analýzu reálného kódu je významná např. podpora velkého množství rozhraní a automatické odvozování přenosu nebezpečných dat nejen pro taint zdroje, ale i další metody.

Varianta	FindBugs	FSB (jen vysoká)	FSB (v.+norm.)
1-17*	ano	ano	ano
21	ne (chybně)	**ano**	ano
22	ne (chybně)	**ano**	ano
31	ano	ano	ano
41	ne (chybně)	**ano**	ano
42	ne (chybně)	**ano**	ano
45	**ano** (chybně)	ne	chybně
51	ne (chybně)	**ano**	ano
52	ne (chybně)	**ano**	ano
53	ne (chybně)	**ano**	ano
54	ne (chybně)	**ano**	ano
61	ne (chybně)	**ano**	ano
66	ne (chybně)	**ano**	ano
67	ne (chybně)	ne	**chybně**
68	ne	ne	**chybně**
71	ne	**ano**	ano
72	ne	**ano**	ano
73	ne	**ano**	ano
74	ne	**ano**	ano
75	ne	ne	**chybně**
81	ne (chybně)	**ano****	ano
celkem ano	19 / 18	33	33
vč. chybných	19 / 31	33	37

Tabulka 6.3: Schopnost analýzy toku dat v sadě Juliet

Tabulka zobrazuje, které z 37 zdrojových variant jsou detekovány samotným nástrojem FindBugs a novým FindSecurityBugs (pro dvě nastavení hranice reportované priority). Kolonka *chybně* značí detekci varianty, ale také nesprávné reportování bezpečných volání. Druhá hodnota ve sloupci pro FindBugs značí úspěšnost pro detekce SQL injekce s vysokou prioritou.

*Zdrojové varianty 2, 3, 4, 6, 9, 13 a 16 mají stejný bytekód jako 1

**Ve výjimečných případech není varianta rozpoznána (reportována s nižší prioritou)

6.3 Limity současné analýzy

S téměř každým vylepšením detekce se zároveň objevovaly krajní případy, pro které je zvolený přístup nedostačující, nebo se díky pokroku odkryly nové překážky limitující analýzu. Uveďme aspoň některé oblasti, které působí v analýze nepřesnost, přitom se ale nejedná pouze o omezení plynoucí ze statického přístupu obecně (tzn. chybějící informace o běhovém prostředí apod.). Stručně zmíněny jsou zároveň i návrhy na vylepšení.

- Automatické odvozování konfigurací metod (viz podsekce 5.2.3) funguje zatím pouze pro návratovou hodnotu, nelze odvodit změnu stavu měnitelného parametru či instance. To je možné pouze u ruční konfigurace, i tam ovšem platí omezení – stav přenesený na parametr či instanci je vždy shodný i pro návratovou hodnotu. Konfigurace by mohla být zobecněna tak, že by každá položka na zásobníku mohla být na parametrech závislá jiným způsobem, v praxi to ale zpravidla není nutné (výjimkou je např. metoda `push` třídy `Stack`, která vrací hodnotu parametru). Analýza by šla vylepšit kontrolou změny stavu proměnných odpovídajících parametrům metody a přidáním odpovídajících indexů do automaticky odvozené konfigurace.

- Pouze čtení atributů třídy bezpečného typu (např. `Integer`) je považováno za bezpečné, jinak je stav neznámý a zápis do atributů je analýzou ignorován, proto nemohl být správně detekován problém ve třech variantách toku sady Juliet (viz podsekce 6.2.2). Analýza by mohla být snadno rozšířena aspoň jednosměrně – jakmile by byl detekován zápis dat se stavem *tainted* do atributu, jeho čtení by dále byla považována za nebezpečná, to by ale v sadě způsobilo snížení specificity. V některých případech by pomohlo automaticky odvozovat konfigurace metod pro nové typy, které mohou přenášet data z taint zdroje ve své instanci. Ještě lepší by bylo neukládat pro instance jen jednu hodnotu, ale stav pro každý atribut a provázat jej s voláním metod. I tak by ale bezpečnost byla závislá na pořadí volání a chyby by musely být potvrzovány podobně jako pro metody (viz podsekce 5.3.1), což už by analýzu velmi komplikovalo.

- Zbývající nedetekovaná varianta (serializovaná data) nemůže být správně analyzována hlavně proto, že objekt, který nebezpečná data přijímá, je přenáší do jiného objektu (předaného v konstruktoru). Pokud je stejná oblast paměti odkazována z více míst, bez globální analýzy nelze při změně z jednoho místa aktualizovat i ostatní reference. Částečným řešením by mohlo být provázání lokálních proměnných s objekty aspoň v rámci jedné metody.

- Současná analýza ignoruje samotnou podmínku při rozvětvení programu (modeluje jej jako nedeterministickou volbu jedné z větví), takže nelze detekovat kontrolu potenciálně nebezpečných dat pomocí validace (viz sekce 3.4). Pokud by podmínka obsahovala např. kontrolu regulárním výrazem nebo testovala přítomnost hodnoty v množině obsahující jen bezpečné prvky, mohl by být k dané hodnotě přidán dočasný příznak bezpečnosti platný minimálně pro jeden základní blok CFG (viz podsekce 3.5.2).

- Při analýze udržujeme pro každou proměnnou či položku zásobníku jen jeden stav i pokud se uvnitř skládá z více prvků. Kromě toho, že z kolekce obsahující nebezpečný prvek už nemůže být vrácena bezpečná hodnota, také např. není detekována konverze pole či kolekce po jednotlivých prvcích. Uchovávání pouze jednoho stavu je důležité z důvodu efektivity, bylo by však možné detekovat přímo určité používané konstrukce (např. cyklus aplikující metodu na všechny její prvky) a na základě toho určit výsledný stav složené hodnoty.

7 Závěr

Tato práce řešila problematiku využití automatické statické analýzy pro detekci injekčních zranitelností. Nejprve jsme se zabývali samotnými injekcemi – pro každou z 9 typů chyb byl zmíněn princip zneužití a možné důsledky, u vybraných jsou uvedeny také ukázky kódu a doplňující informace. V další části byly stručně rozebrány přístupy k automatické analýze software a vysvětleny zásadní pojmy jako *taint zdroj*, *taint sink* a *taint analýza*. Zmíněny byly také způsoby ošetření potenciálně nebezpečných dat a při srovnání s formálními verifikačními metodami už je naznačen princip analýzy toku dat. Po představení nástroje FindBugs a rozšíření FindSecurityBugs jsme se věnovali formátu přeložených tříd jazyka Java a instrukcím bytekódu, což je důležitý předpoklad pro tvorbu vlastních detektorů chyb a vylepšování analýzy. Shrnuty by měly být nejdůležitější informace potřebné pro zapojení se do vývoje FindSecurityBugs či jiných rozšíření, uvedeny jsou i principy pro detektory využívající pokročilou analýzu toku dat.

Implementovaná taint analýza modeluje hodnoty dat v zásobníku operandů a lokálních proměnných pro každé místo v kódu, při čemž naráz uvažuje všechny možnosti větvení. Pro každou položku si pamatuje především to, zda data v ní mohou být nebezpečná (alespoň při určitém větvení) a zda byla ošetřena konverzí. Jednotlivé instrukce tyto příznaky mění a přenášejí mezi sebou, takové chování je definováno i pro instrukce typu *invoke* a více než 750 nakonfigurovaných metod (navíc automaticky převzato potomky odpovídajících tříd či rozhraní). Konfigurace jsou také automaticky odvozovány pro neznámé metody na základě výsledku jejich analýzy, takže chyby mohou být hledány napříč celým programem, tomu napomáhá vhodná volba pořadí analýzy metod.

Samotná detekční část taint analýzu ještě vylepšuje sledováním volání s potenciálními zranitelnostmi a umožňuje snadnou tvorbu nových a přehledných detektorů. V rámci práce bylo také rozšíření FindSecurityBugs obohaceno o detekci tří dalších typů injekčních zranitelností (a jedné další chyby) a opraveno několik chyb. Existující detektory injekcí byly přepsány a provedena byla revize potenciálně zranitelných volání – pro každou chybu jsme nově přidali nejméně jeden taint sink a i s novými detektory byl jejich počet navýšen o více

než 400. Podrobnější informace o implementovaných vylepšeních jsou uvedeny v sekci 5.4. Všechny změny byly přijaty do oficiální distribuce FindSecurityBugs [51] a novější verzi nástroje jsme spolu s původním autorem ke konci roku 2015 představili také na prestižní bezpečnostní konferenci *Black Hat Europe* [53].

Úspěšnost analýzy byla vyhodnocena na části sady Juliet – otestována byla detekce 12 typů injekčních chyb v necelých 8 tisících testovacích případech. Analýzu vylepšené verze FindSecurityBugs limitovaly pouze 4 náročné varianty toku (z 37), které byly rozeznány až při nižší prioritě (avšak za cenu reportování také bezpečných volání v těchto variantách). Nástroj je proto vhodné použít tak, že nejprve budou ověřeny chyby s vysokou prioritou, kde je nízká míra falešných detekcí (pro sadu Juliet nulová). Následně můžeme zkontrolovat detekce s nižší prioritou, které by měly obsahovat všechny skutečné chyby daného typu, ale pravděpodobně i bezpečná volání (ovšem v mnohem menší míře než u původního rozšíření). Celková přesnost analýzy vzrostla na 95 % (pro obě nastavení hranice priority) oproti 53 % u původní verze rozšíření i samostatného nástroje FindBugs.

Inspirací pro budoucí vývoj může být sekce 6.3 popisující limity současné analýzy. Pokročilejší odvozování konfigurací, analýza atributů tříd či detekce validace dat mohou úspěšnost detekce dále zvýšit, rovněž množství používaných knihoven není dosud podporováno. Před implementací vylepšení by však bylo vhodné vyhodnotit výsledky nad reálným kódem. Nástroj sice byl úspěšně testován i na skutečném software, analýza výsledků ale zatím není dostatečná pro rozhodnutí, na která vylepšení je třeba se zaměřit pro co nejefektivnější zvýšení použitelnosti nástroje.

Dalším směrem může být použití taint analýzy i pro jiné než injekční a příbuzné chyby. Například implementovaný detektor pro hesla a kryptografické klíče uvedené v kódu by ji mohl využít pro vyšší přesnost, pokud bude přidána podpora pro další datové typy (např. pole hodnot typu `byte`). Dále by šlo analýzu použít pro detekci úniku informací (CWE-200), místo taint zdrojů by byly specifikovány zdroje citlivých dat a jako taint sink by mohl sloužit libovolný výstup.

Literatura

[1] David Formánek. „Nástroje pro statickou a dynamickou analýzu kódu se zaměřením na bezpečnostní chyby". Bakalářská práce. Masarykova univerzita, 2014.

[2] *FindBugs – Find Bugs in Java Programs*. 2015. URL: http://findbugs.sourceforge.net/ (cit. 08. 05. 2016).

[3] *Find Security Bugs*. 2015. URL: https://find-sec-bugs.github.io/ (cit. 08. 05. 2016).

[4] *Test Suites*. 2015. URL: https://samate.nist.gov/SRD/testsuite.php (cit. 08. 05. 2016).

[5] *CWE-74: Improper Neutralization of Special Elements in Output Used by a Downstream Component ('Injection')*. 2015. URL: https://cwe.mitre.org/data/definitions/79.html (cit. 08. 05. 2016).

[6] *CWE – Common Weakness Enumeration*. 2015. URL: https://cwe.mitre.org/index.html (cit. 08. 05. 2016).

[7] Steve Christey. *2011 CWE/SANS Top 25 Most Dangerous Software Errors*. 2011. URL: https://cwe.mitre.org/top25/index.html (cit. 08. 05. 2016).

[8] *Top 10 2013*. 2016. URL: https://www.owasp.org/index.php/Top_10_2013 (cit. 08. 05. 2016).

[9] *CWE-89: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')*. 2015. URL: https://cwe.mitre.org/data/definitions/89.html (cit. 08. 05. 2016).

[10] *CAPEC-7: Blind SQL Injection*. 2015. URL: https://capec.mitre.org/data/definitions/7.html (cit. 08. 05. 2016).

[11] Bernando Damele a Miroslav Stampar. *sqlmap: Automatic SQL injection and database takeover tool*. 2016. URL: http://sqlmap.org/ (cit. 08. 05. 2016).

[12] *Spring Framework*. 2016. URL: http://projects.spring.io/spring-framework/ (cit. 08. 05. 2016).

[13] *Hibernate ORM*. 2016. URL: http://hibernate.org/orm/ (cit. 08. 05. 2016).

[14] *CWE-77: Improper Neutralization of Special Elements used in a Command ('Command Injection')*. 2015. URL: https://cwe.mitre.org/data/definitions/77.html (cit. 08. 05. 2016).

[15] *CWE-79: Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')*. 2015. URL: https://cwe.mitre.org/data/definitions/79.html (cit. 08. 05. 2016).
[16] *CWE-22: Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')*. 2015. URL: https://cwe.mitre.org/data/definitions/22.html (cit. 08. 05. 2016).
[17] *CWE-90: Improper Neutralization of Special Elements used in an LDAP Query ('LDAP Injection')*. 2015. URL: https://cwe.mitre.org/data/definitions/90.html (cit. 08. 05. 2016).
[18] *CWE-643: Improper Neutralization of Data within XPath Expressions ('XPath Injection')*. 2014. URL: https://cwe.mitre.org/data/definitions/643.html (cit. 08. 05. 2016).
[19] *CWE-93: Improper Neutralization of CRLF Sequences ('CRLF Injection')*. 2015. URL: https://cwe.mitre.org/data/definitions/93.html (cit. 08. 05. 2016).
[20] *CWE-117: Improper Output Neutralization for Logs*. 2014. URL: https://cwe.mitre.org/data/definitions/117.html (cit. 08. 05. 2016).
[21] *CWE-113: Improper Neutralization of CRLF Sequences in HTTP Headers ('HTTP Response Splitting')*. 2015. URL: https://cwe.mitre.org/data/definitions/113.html (cit. 08. 05. 2016).
[22] *HTTP Response Splitting*. 2016. URL: https://www.owasp.org/index.php/HTTP_Response_Splitting (cit. 08. 05. 2016).
[23] *CWE-601: URL Redirection to Untrusted Site ('Open Redirect')*. 2015. URL: https://cwe.mitre.org/data/definitions/601.html (cit. 08. 05. 2016).
[24] *Top 10 2013-A10-Unvalidated Redirects and Forwards*. 2013. URL: https://www.owasp.org/index.php/Top_10_2013-A10-Unvalidated_Redirects_and_Forwards (cit. 08. 05. 2016).
[25] *OWASP Testing Guide Appendix C: Fuzz Vectors*. 2014. URL: https://www.owasp.org/index.php/OWASP_Testing_Guide_Appendix_C:_Fuzz_Vectors (cit. 08. 05. 2016).
[26] *Static Code Analysis*. 2016. URL: https://www.owasp.org/index.php/Static_Code_Analysis (cit. 08. 05. 2016).
[27] *Fuzzing*. 2016. URL: https://www.owasp.org/index.php/Fuzzing (cit. 08. 05. 2016).
[28] Irene Abezgauz. *Introduction to Interactive Application Security Testing (IAST)*. 2014. URL: http://www.quotium.com/resources/interactive-application-security-testing/ (cit. 08. 05. 2016).

[29] *perlsec – Perl Security*. 2016. URL: http://perldoc.perl.org/perlsec.pdf (cit. 08. 05. 2016).

[30] Addison Wesley Longman. *Locking Ruby in the Safe*. 2001. URL: http://phrogz.net/ProgrammingRuby/taint.html (cit. 08. 05. 2016).

[31] Brian Chess, Jacob West a Gary McGraw. „Secure Programming with Static Analysis“. In: Addison-Wesley, 2007. Kap. 4, s. 71–105. ISBN: 978-0321424778.

[32] *CWE-807: Reliance on Untrusted Inputs in a Security Decision*. 2014. URL: https://cwe.mitre.org/data/definitions/807.html (cit. 08. 05. 2016).

[33] *CWE-180: Incorrect Behavior Order: Validate Before Canonicalize*. 2014. URL: https://cwe.mitre.org/data/definitions/180.html (cit. 08. 05. 2016).

[34] Jim McMillan. *IDFAQ: What is the Difference Between an IPS and a Web Application Firewall?* 2009. URL: https://www.sans.org/security-resources/idfaq/what-is-the-difference-between-an-ips-and-a-web-application-firewall/1/25 (cit. 08. 05. 2016).

[35] *Category:OWASP Enterprise Security API*. 2015. URL: https://www.owasp.org/index.php/ESAPI (cit. 08. 05. 2016).

[36] *OWASP Java Encoder Project*. 2015. URL: https://www.owasp.org/index.php/OWASP_Java_Encoder_Project (cit. 08. 05. 2016).

[37] *OWASP Java HTML Sanitizer Project*. 2016. URL: https://www.owasp.org/index.php/OWASP_Java_HTML_Sanitizer_Project (cit. 08. 05. 2016).

[38] *Class StringEscapeUtils*. 2015. URL: https://commons.apache.org/proper/commons-lang/apidocs/org/apache/commons/lang3/StringEscapeUtils.html (cit. 08. 05. 2016).

[39] *Class HtmlUtils*. 2016. URL: http://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/web/util/HtmlUtils.html (cit. 08. 05. 2016).

[40] Cristian Cadar a Koushik Sen. *Symbolic Execution for Software Testing: Three Decades Later*. 2013. URL: http://srl.cs.berkeley.edu/~ksen/papers/cacm13.pdf (cit. 08. 05. 2016).

[41] Jean-Christophe Filliâtre. *Deductive software verification*. 2011. URL: http://link.springer.com/article/10.1007/s10009-011-0211-0 (cit. 08. 05. 2016).

[42] Ranjit Jhala a Rupak Majumdar. *Software Model Checking*. 2009. URL: http://goto.ucsd.edu/~rjhala/papers/software_model_checking_survey.pdf (cit. 08. 05. 2016).

[43] Michael I. Schwartzbach. *Lecture Notes on Static Analysis*. URL: http://lara.epfl.ch/w/_media/sav08:schwartzbach.pdf (cit. 08. 05. 2016).

[44] Michael I. Schwartzbach. *CS252r Spring 2011. Data Flow Analysis*. 2011. URL: http://www.seas.harvard.edu/courses/cs252/2011sp/slides/Lec02-Dataflow.pdf (cit. 08. 05. 2016).

[45] *Using FindBugs for Research*. 2007. URL: https://findbugs-tutorials.googlecode.com/files/uffr-talk.pdf (cit. 08. 05. 2016).

[46] Matt Insall a Eric W. Weisstein. *Lattice*. From MathWorld – A Wolfram Web Resource. URL: http://mathworld.wolfram.com/Lattice.html (cit. 08. 05. 2016).

[47] Tim Lindholm et al. *The Java Virtual Machine Specification. Java SE 8 Edition*. 2015. URL: http://docs.oracle.com/javase/specs/jvms/se8/html/index.html (cit. 08. 05. 2016).

[48] *FindBugs API Documentation*. 2015. URL: http://findbugs.sourceforge.net/api/index.html (cit. 08. 05. 2016).

[49] *The new home of the FindBugs project*. 2016. URL: https://github.com/findbugsproject/findbugs (cit. 08. 05. 2016).

[50] *Apache Commons BCEL*. 2014. URL: https://commons.apache.org/proper/commons-bcel/ (cit. 08. 05. 2016).

[51] *Find Security Bugs*. 2016. URL: https://github.com/find-sec-bugs/find-sec-bugs (cit. 08. 05. 2016).

[52] Damien François. *Binary classification performances measure cheat sheet*. 2009. URL: http://www.damienfrancois.be/blog/files/modelperfcheatsheet.pdf (cit. 08. 05. 2016).

[53] *Black Hat Europe 2015. Arsenal*. 2015. URL: https://www.blackhat.com/eu-15/arsenal.html#findsecuritybugs (cit. 08. 05. 2016).

A Seznam zneužitelných metod

Kapitola obsahuje seznam volání (dostupných ve FindSecurityBugs v době dokončení práce), která jsou považována za taint sink (viz sekce 3.1) pro každý typ chyby zmíněný v kapitole 2. Uveden je vždy název třídy včetně balíku a názvy jeho metod. Pro přehlednost nejsou uvedeny parametry a návratové hodnoty a více metod stejného názvu (lišící se počtem či typem parametrů) je uvedeno jen pod názvem metody a počtem variant v závorce. Rovněž není uvedeno, které parametry metod jsou zranitelné, tyto detailní informace lze nalézt v konfiguraci nástroje.

A.1 SQL injekce

JDBC

- java.sql.Statement
  - executeQuery
  - execute (4 varianty)
  - executeUpdate (4 varianty)
  - executeLargeUpdate (4 varianty)
  - addBatch

- java.sql.PreparedStatement
  - executeQuery
  - execute (4 varianty)
  - executeUpdate (4 varianty)
  - executeLargeUpdate (4 varianty)
  - addBatch

- java.sql.Connection
  - prepareCall (3 varianty)
  - prepareStatement (6 variant)
  - nativeSql

JDO

- javax.jdo.PersistenceManager
  - newQuery (5 variant)
- javax/jdo/Query
  - setFilter
  - setGrouping

JPA

- javax.persistence.EntityManager
  - createQuery (2 varianty)
  - createNativeQuery (3 varianty)

Spring

- org.springframework.jdbc.core.PreparedStatementCreatorFactory
  - *konstruktor* (3 varianty)
  - newPreparedStatementCreator
- org.springframework.jdbc.core.JdbcOperations
  - batchUpdate (5 variant)
  - execute (3 varianty)
  - query (15 variant)
  - queryForList (7 variant)
  - queryForMap (3 varianty)
  - queryForObject (8 variant)
  - queryForRowSet (3 varianty)
  - queryForInt (3 varianty)
  - queryForLong (3 varianty)
  - update (4 varianty)
- org.springframework.jdbc.core.JdbcTemplate
  - batchUpdate (5 variant)
  - execute (3 varianty)
  - query (15 variant)
  - queryForList (7 variant)

  - queryForMap (3 varianty)
  - queryForObject (8 variant)
  - queryForRowSet (3 varianty)
  - queryForInt (3 varianty)
  - queryForLong (3 varianty)
  - update (4 varianty)

Hibernate

- org.hibernate.criterion.Restrictions
  - sqlRestriction (3 varianty)
- org.hibernate.Session
  - createQuery
  - createSQLQuery

A.2 Injekce příkazu

- java.lang.Runtime
  - exec (6 variant)
- java.lang.ProcessBuilder
  - *konstruktor* (2 varianty)
  - command (2 varianty)

A.3 XSS

Servlet

- java.io.PrintWriter (jen pro určité analyzované podtřídy)
  - write (4 varianty)
  - format (2 varianty)
  - print (4 varianty)
  - println (4 varianty)
  - printf (2 varianty)
  - append (3 varianty)

JSP

- javax.servlet.jsp.JspWriter (jen pro určité analyzované podtřídy)
  - write (4 varianty)
  - append (3 varianty)
  - print (4 varianty)
  - println (4 varianty)

A.4 Procházení adresářem

- java.io.File
  - *konstruktor* (4 varianty)
- java.io.RandomAccessFile
  - *konstruktor*
- java.io.FileReader
  - *konstruktor*
- java.io.FileInputStream
  - *konstruktor*
- java.io.FileWriter
  - *konstruktor* (2 varianty)
- java.io.FileOutputStream
  - *konstruktor* (2 varianty)

A.5 LDAP injekce

- javax.naming.ldap.LdapName
  - *konstruktor*
- javax.naming.directory.Context
  - lookup
- javax.naming.directory.DirContext

  - lookup
  - search (4 varianty)

- javax.naming.directory.InitialDirContext
  - lookup
  - search (4 varianty)

- javax.naming.ldap.LdapContext
  - lookup
  - search (4 varianty)

- javax.naming.ldap.InitialLdapContext
  - lookup
  - search (4 varianty)

- javax.naming.event.EventDirContext
  - lookup
  - search (4 varianty)

- com.sun.jndi.ldap.LdapCtx
  - lookup
  - search (4 varianty)

- com.unboundid.ldap.sdk.LDAPConnection
  - search

A.6 XPath injekce

- javax.xml.xpath.XPath
  - compile
  - evaluate (4 varianty)

Apache

- org.apache.xpath.XPathAPI
  - eval (3 varianty)
  - selectNodeIterator (2 varianty)

  - selectNodeList (2 varianty)
  - selectSingleNode (2 varianty)

- org.apache.xpath.internal.XPathAPI
  - eval (3 varianty)
  - selectNodeIterator (2 varianty)
  - selectNodeList (2 varianty)
  - selectSingleNode (2 varianty)

- org.apache.xml.security.utils.XPathAPI
  - evaluate
  - selectNodeList

- org.apache.xml.security.utils.JDKXPathAPI
  - evaluate
  - selectNodeList

- org.apache.xml.security.utils.XalanXPathAPI
  - evaluate
  - selectNodeList

A.7 CRLF injekce

Falšování záznamů logu

- java.util.logging.Logger
  - config
  - entering (3 varianty)
  - exiting (2 varianty)
  - fine
  - finer
  - finest
  - info
  - log (4 varianty)
  - logp (5 variant)
  - logrb (6 variant)
  - severe

  - throwing
  - warning

- org.apache.commons.logging.Log
  - debug (2 varianty)
  - error (2 varianty)
  - fatal (2 varianty)
  - info (2 varianty)
  - trace (2 varianty)
  - warn (2 varianty)

- org.slf4j.Logger
  - debug (10 variant)
  - error (10 variant)
  - info (10 variant)
  - trace (10 variant)
  - warn (10 variant)

- org.apache.logging.log4j.Logger
  - debug (12 variant)
  - error (12 variant)
  - fatal (12 variant)
  - info (12 variant)
  - log (12 variant)
  - printf (2 varianty)
  - trace (12 variant)
  - warn (12 variant)

- org.apache.log4j.Category
  - debug (2 varianty)
  - error (2 varianty)
  - fatal (2 varianty)
  - info (2 varianty)
  - 17dlog (2 varianty)
  - log (3 varianty)
  - warn (2 varianty)

- org.apache.log4j.Logger

  - debug (2 varianty)
  - error (2 varianty)
  - fatal (2 varianty)
  - info (2 varianty)
  - 17dlog (2 varianty)
  - log (3 varianty)
  - warn (2 varianty)
  - trace (2 varianty)

- org.pmw.tinylog.Logger
  - debug (4 varianty)
  - error (4 varianty)
  - info (4 varianty)
  - trace (4 varianty)
  - warn (4 varianty)

Rozdělování HTTP odpovědi

- javax.servlet.http.Cookie
  - *konstruktor*
  - setValue
- javax.servlet.http.HttpServletResponse
  - addHeader
  - setHeader
- javax.servlet.http.HttpServletResponseWrapper
  - addHeader
  - setHeader

A.8 Neošetřené přesměrování

- javax.servlet.http.HttpServletResponse
  - addHeader (jen s argumentem `"Location"`)
  - sendRedirect
  - setHeader (jen s argumentem `"Location"`)

- javax.servlet.http.HttpServletResponseWrapper
  - addHeader (jen s argumentem `"Location"`)
  - sendRedirect
  - setHeader (jen s argumentem `"Location"`)

A.9 Ostatní

Externí kontrola konfigurace

- java.sql.Connection
  - setCatalog

EL injekce

- javax.el.ExpressionFactory
  - createValueExpression
  - createMethodExpression

SPEL injekce

- org.springframework.expression.ExpressionParser
  - parseExpression
- org.springframework.expression.spel.standard.
  SpelExpressionParser
  - parseExpression
- org.springframework.expression.common.
  TemplateAwareExpressionParser
  - parseExpression

SEAM-EL injekce

- org.jboss.seam.log.Log
  - debug (2 varianty)
  - error (2 varianty)
  - fatal (2 varianty)

  - info (2 varianty)
  - trace (2 varianty)
  - warn (2 varianty)

Injekce pro Script engine

- javax.script.ScriptEngine
  - eval

B Seznam taint zdrojů

Kapitola obsahuje seznam volání považovaných za taint zdroj (podle FindSecurityBugs v době dokončení této práce). Formát je stejný jako v příloze A, navíc jsou uvedena jen volání třídy či rozhraní nejvýše v hierarchii podle dědičnosti, metody potomků jsou podporovány automaticky. Kromě toho je za taint zdroj považován argument vstupní funkce `main` a parametry metod označených jednou ze 14 anotací z balíků:

- org.springframework.web.bind.annotation
- org.springframework.ws.server.endpoint.annotation
- javax.ws.rs

B.1 Servlety

- javax.servlet.ServletRequest
  - getContentType
  - getLocalAddr
  - getLocalName
  - getParameter
  - getParameterMap
  - getParameterNames
  - getParameterValues
  - getRemoteHost
  - getServerName
- javax.servlet.http.HttpServletRequest
  - getCookies
  - getHeader
  - getHeaderNames
  - getHeaders
  - getPathInfo
  - getPathInfoTranslated
  - getQueryString
  - getRemoteUser
  - getRequestURI

  - getRequestURL
  - getServletPath

- javax.servlet.http.Cookie
  - getComment
  - getDomain
  - getName
  - getPath
  - getValue

B.2 Ostatní

Grafické rozhraní (AWT a Swing)

- javax.awt.TextComponent
  - getSelectedText
  - getText

- javax.swing.text.JTextComponent
  - getSelectedText
  - getText (2 varianty)

Čtení souborů (a obecně proudů), konzole a obsahu databáze

- java.io.BufferedReader
  - readLine (2 varianty)

- java.io.Console
  - readLine (2 varianty)

- java.io.DataInputStream
  - readLine
  - readUTF (2 varianty)

- java.io.LineNumberReader
  - readLine

- java.sql.ResultSet

  - getNString (2 varianty)
  - getString (2 varianty)

- java.util.Properties
  - load (2 varianty)
  - loadFromXml

Systémové proměnné

- java.lang.System (jen pro lokální aplikace)
  - clearProperty
  - getenv (2 varianty)
  - getProperty (2 varianty)

C Elektronické přílohy práce

Součástí práce jsou také následující elektronické přílohy:

- `findbugs.zip` obsahuje nástroj FindBugs ve verzi 3.0.1 s předinstalovaným rozšířením FindSecurityBugs pro snadné spuštění
- `FindSecurityBugs-source.zip` obsahuje zdrojové kódy vylepšeného nástroje FindSecurityBugs
- `latex.zip` obsahuje zdrojové soubory systému LaTeXpro text této práce