Masarykova univerzita
Fakulta informatiky

Fuzz testování webových aplikačních rozhraní

Diplomová práce

Bc. Jan Stárek

Brno, podzim 2019

Masarykova univerzita
Fakulta informatiky

Fuzz testování webových aplikačních rozhraní

Diplomová práce

Bc. Jan Stárek

Brno, podzim 2019

Na tomto místě se v tištěné práci nachází oficiální podepsané zadání práce a prohlášení autora školního díla.

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. Jan Stárek

Vedoucí práce: RNDr. Andrij Stecko, Ph.D.

Poděkování

Tímto bych chtěl poděkovat vedoucímu diplomové práce, jímž je RNDr. Andrij Stecko, Ph.D., za odbornou pomoc, rady, konzultace a připomínky při jejím psaní. Dále bych chtěl poděkovat také bezpečnostním konzultantům ze společnosti Y Soft, jmenovitě: Ing. Petr Štěpánek, Mgr. David Formánek a Mgr. Lenka Bačinská za odbornou konzultaci ohledně implementovaného nástroje.

Shrnutí

Diplomová práce se zaobírá implementací plně automatizovaného nástroje pro fuzz testování webových aplikačních rozhraní pro snadné nasazení v prostředí průběžné integrace. Tento nástroj je vyvíjen ve spolupráci se společností Y Soft, jenž jej využívá pro testování svých webových aplikačních rozhraní. Nástroj je na společnosti nezávislý a je volně dostupný. V práci jsou popsány existující nástroje a knihovny pro fuzz testování včetně jejich nedostatků vzhledem k řešené problematice. Druhá část práce pak rozebírá implementaci samotného nástroje, využité knihovny a dosažené výsledky.

Klíčová slova

fuzzing, fuzz testování, Boofuzz, CI/CD, web api, swagger, openapi, docker

Obsah

Úvod **1**

1 Fuzz testování **3**
1.1 *Vznik a historie* 3
1.2 *Whitebox přístup* 5

2 Nástroje pro fuzz testování **7**
2.1 *Burp Suite* 7
2.1.1 Komponenty 7
2.1.2 Fuzz testování 8
2.1.3 Průběžná integrace 9
2.2 *OWASP ZAP* 9
2.2.1 Možnosti fuzz testování 10
2.2.2 Průběžná integrace 10
2.3 *Boofuzz* 10
2.3.1 Vlastnosti knihovny 11
2.3.2 Monitor procesů 12
2.3.3 Důležité funkce a datové struktury 12
2.3.4 Vytváření definic požadavků 14
2.3.5 Vestavěná primitiva 15
2.3.6 Generování výsledných požadavků 17
2.4 *Shrnutí a zhodnocení popsaných nástrojů* 18

3 Systémy průběžné integrace **19**
3.1 *Travis CI* 20
3.1.1 Jak Travis CI funguje? 20
3.1.2 Konfigurace 21
3.1.3 Podpora kontejnerizace 22
3.2 *CircleCI* 23
3.2.1 Funkce platformy 23
3.2.2 Konfigurace 24
3.3 *Bamboo* 25
3.3.1 Základní koncepty a pojmy 26
3.3.2 Agenti 26
3.3.3 Kontejnerizace 27

3.4 *Jenkins* 28
3.4.1 Pipelines 28
3.4.2 Kontejnerizace 29
3.5 *Limitace kladené průběžnou integrací na implementované řešení* 29

4 Dokumentační formáty webových aplikačních rozhraní 31
4.1 *OpenAPI* 32
4.2 *API Blueprint* 32
4.3 *RAML* 33

5 Implementované řešení 35
5.1 *Návrh* 35
5.2 *Získávání dat z dokumentace* 37
5.2.1 Konverze mezi formáty 37
5.2.2 Výběr vstupního formátu 37
5.2.3 Implementace parseru 38
5.3 *Fuzzování dat* 38
5.3.1 Staticky předpřipravené vektory dat 39
5.3.2 Zhodnocení vestavěných primitiv 39
5.3.3 Vlastní primitiva 40
5.3.4 Generování JSON dat 42
5.3.5 Fuzzované části dotazů 43
5.4 *Zasílání dotazů a validace odpovědí* 45
5.4.1 Komunikace s testovanou aplikací 46
5.4.2 Vyhodnocení úspěšnosti testovacího scénáře . . 47
5.5 *Generování testových reportů* 48
5.5.1 Volba formátu pro reportování testovacích scénářů 48
5.5.2 Implementace generování JUnit výsledků 48
5.5.3 Vizualizace reportů 49
5.6 *Vstupy implementovaného nástroje* 49
5.7 *Zapouzdření do Docker obrazu* 50
5.7.1 Tvorba Docker obrazu 50
5.7.2 Spuštění a běh Docker kontejneru 51
5.8 *Implementace systémových testů* 51
5.8.1 Detekce zranitelnosti serveru pomocí SQL injekcí 51
5.8.2 Detekce zranitelnosti serveru pomocí vkládání příkazů 52

5.8.3 Detekce neexistující testované aplikace 52
5.8.4 Detekce špatného formátu odpovědi 52
5.8.5 Detekce selhání obsluhy požadavku 53

6 Zhodnocení řešení **55**
6.1 *Modelové použití na volně šiřitelném projektu* 55
6.2 *Použití v prostředí společnosti Y Soft* 57
6.3 *Náročnost na výpočetní zdroje* 57
6.4 *Problematika účinnosti fuzz testování* 58
6.5 *Návrhy na rozšíření* 59

7 Závěr **61**

Bibliografie **63**

A Nástroje pro fuzz testování síťových aplikací **69**

B Manuál k výslednému nástroji **71**
B.1 *Konfigurace implementovaného nástroje* 71
B.2 *Použití monitoru procesů* 71
B.3 *Spuštění pomocí běžně dostupných shellů* 71
B.4 *Spuštění pomocí Docker kontejneru* 72

C Elektronické přílohy **73**
C.1 *Struktura archivu s implementovaným nástrojem* 73

Úvod

Fuzz testování je čím dál více populární metodou testování softwarových aplikací. Tato forma testování může v dnešní podobě sloužit pro plně automatizované bezpečnostní testování, které může odhalit řadu zranitelností testované aplikace. Tuto metodiku testování, jenž bude detailně popsána v textu níže, aktuálně využívá řada velkých firem. Jedná se například Microsoft, Google, Twitter a mnoho dalších. Společnost Google uvádí, že pomocí fuzz testování odhalila zhruba 16 000 chyb jen v aplikaci Google Chrome[1]. [1]

Poptávka po nástrojích vhodných pro fuzz testování je relativně velká. Proto vzniklo a dále vzniká mnoho nástrojů, které se této problematice věnují. Fuzz testování je však velmi obecná technika testování, která se dá cílit na libovolný vstup aplikace. Jednotlivé nástroje se tedy specializují na určitou doménu testovaných aplikací. Cílem této práce je zmapovat, navrhnout a implementovat plně automatizované fuzz testování webových aplikačních rozhraní pro snadné nasazení v prostředí průběžné integrace.

Práce začíná představením metodiky fuzz testování, jejím vznikem a historickým vývojem. Následuje seznámení s několika existujícími řešeními fuzz testování. Jelikož cílem práce je implementovat nástroj pro snadné nasazení v prostředí průběžné integrace, jsou zde detailněji rozebráni typičtí zástupci těchto systémů. Následuje popis vybraných dokumentačních formátů pro webová aplikační rozhraní, které jsou taktéž klíčovým bodem této práce. Implementační část diplomové práce se pak zaobírá návrhem a samotnou implementací výsledného nástroje.

1. `https://www.google.com/chrome/`

1 Fuzz testování

Fuzz testování (neboli také fuzzing) je automatizovaná technika testování, která spočívá v přivádění nevalidních vstupů do testovacích aplikací. Fuzzing umožňuje vývojářům a testerům otestovat velké množství krajních případů tam, kde by běžné funkcionální testování nebylo efektivní. [2]

1.1 Vznik a historie

Fuzz testování bylo poprvé definováno na univerzitě Wisconsin Madison kolem roku 1988 profesorem Bartonem Millerem a jeho studenty. [3] Tento typ testování byl avšak znám již mnohem dříve. Gerald M. Weinberg, americký doktor v oblasti komunikačních systémů, uvedl, že tato technika byla používána již v 50 letech, kdy programátoři používali pro náhodná data odhozené děrné štítky, které nalezli například v odpadkovém koši. [4]

Jelikož zcela náhodné testování je časově náročné, přístup profesora Wisconsina Madisona, který je připisován k založení, pojmenování a definování techniky fuzz testování, bylo dodržení třech následujících charakteristik, které pospolu tvořily odlišný přístup od ostatních v té době používaných metodik testování. [5] Nutno podotknout, že tato tři charakteristická pravidla byla použita pro testování aplikací v 90. letech a na přelomu tisíciletí.

- Vstup je náhodný. Není použita znalost modelu programu, jeho chování ani znalost typu testovaného programu. Jedná se o typické blackbox testování.
- Spolehlivostní kritérium je jednoduché: pokud aplikace zamrzne či havaruje, test je považován za neúspěšný, jinak je úspěšný. Aplikace na předaný vstup nemusí nijak reagovat. I pokud se tiše ukončí bez jakékoliv jiné reakce, test je považován za úspěšný.
- Výsledkem prvních dvou charakteristik je, že test může být jednoduše automatizován a výsledky lze porovnat napříč aplikacemi, operačními systémy i dodavateli.

Roku 1988 byl zadán a v roce 1990 dokončen první projekt implementující softwarové fuzz testování, jehož cílem bylo otestovat různé unixové nástroje pomocí nepředvídatelných vstupních proudů. Vznikl protokol s názvem *An Empirical Study of the Reliability of UNIX Utilities* [3], který dokumentoval prováděný test a jeho závěry. Zajímavostí se pak může jevit samotná motivace k provedení takovéhoto projektu. Jeden z autorů uvádí, že byl přihlášen ke svému počítači pomocí vytáčeného spojení skrze 1200 baudový modem. Silný déšť pospolu s bouřkou pak ovlivňoval spojení natolik, že často docházelo ke změně přenášených dat. Autor tak mohl sledovat lexikální deformaci jím zadávaných příkazů. Autoři pak byli překvapeni, když zjistili, že takovýto deformovaný signál projevený změnou jednotlivých písmen přiváděných na vstup aplikace způsobil u spousty běžných unixových nástrojů jejich pád. Tato příhoda pak vedla k samotné motivaci uskutečnit projekt, který bude posílat náhodné znaky do běžných unixových nástrojů. Výsledek projektu pak ukázal, že pro 25% až 33% běžných nástrojů unixového systému bylo možné způsobit jejich pád přivedením náhodných dat na jejich vstup. Více o metodice a provedeném testování je možné nalézt na webových stránkách profesora Bartona. [5]

V roce 1995 bylo provedeno další a rozsáhlejší testování unixových nástrojů a služeb, z jehož výsledků čerpá následující odstavec. [6] Mezi testované služby patřil například X-Window[1] server a dále také typické síťové služby jako ftp démon, telnet démon a další. Výsledek byl velice obdobný projektu z roku 1990. Až u 40% základních unixových programů byl přivedením náhodných dat na jejich vstup způsoben jejich pád. To samé platilo pro více než 25% tehdejších X-Window aplikací. Výsledky pro samotný X-Server a síťové služby byly již příznivější. Nepovedlo se zastavit či shodit ani jednu testovanou službu. Testování také mimo jiné ukázalo, že nástroje GNU či Linuxu byly spolehlivější než většina komerčních variant.

V letech 2000 a 2006 pak byly podrobeny podobným testům operační systémy Windows NT a Mac OS X. Nástroje systému Windows NT dopadly velmi obdobně jako nástroje systému UNIX z roku 1995. Pod vlivem náhodného vstupu spadlo 21% aplikací a dalších 24% zamrzlo. Protokol z roku 2006 o testování systému Mac OS X ukazuje,

1. `https://en.wikipedia.org/wiki/X_Window_System`

že unixové programy spouštěné z příkazové řádky mají s náhodným vstupem problém pouze v 7% případů. Naopak, Mac OS X aplikace využívající grafické uživatelské rozhraní byly k pádu či zamrznutí daleko náchylnější. Pouze 8 aplikací z 30 testovaných se s náhodným vstupem dokázalo vypořádat jinak než zamrznutím či pádem. [7] [8]

1.2 Whitebox přístup

Whitebox fuzzing byl detailně popsán v článcích [9] [10], na kterých spolupracovalo několik výzkumníků ze společnosti Microsoft a Kalifornské univerzity v Berkeley. Tradičně je fuzz testováním myšleno blackbox testování, kde pomocí variace validních vstupů a jejich náhodné modifikace testujeme chování testované aplikace. Vezměme v potaz následující funkci:

```
int foo(int x)
{
  int y = x + 3;
  if (y == 13) abort; // error
  return 0;
}
```

Pokud pracujeme na běžné architektuře, kde je integer v jazyce C mapován na 32 bitovou hodnotu v paměti, pak při testování běžným blackbox přístupem máme 2^{32} možností, jak náhodně zvolit hodnotu proměnné x. Na tomto jednoduchém příkladu můžeme vidět, že pokud bychom chtěli pokrýt všechny možnosti, bude blackbox testování nereálné z důvodu jeho časové náročnosti.

Firma Microsoft vytvořila platformu SAGE[2] jako sadu programů pro whitebox souborové fuzz testovaní x86 Windows aplikací. Tento nástroj spustil testovanou aplikaci s nějakým počátečním vstupem, provedl tzv. symbolické provedení[3] pro získání sady omezení pro vstupní data na základě predikátů při větvení programu a následně tato omezení analyzoval a vytvořil další varianty testovacích vstupů, které zapříčinily běh programu v jiných větvích.

2. https://www.microsoft.com/en-us/research/publication/automated-whitebox-fuzz-testing/
3. https://www.cs.umd.edu/~mwh/se-tutorial/symbolic-exec.pdf

Toto testování se ukázalo jako mnohem efektivnější než běžné blackbox testování pomocí náhodné modifikace vstupních dat, nicméně je úzce závislé na analýze běhu programu a derivování nových vstupů na základě samotného kódu.

2 Nástroje pro fuzz testování

Nástrojů pro fuzz testování je celá plejáda. V tabulce A.1 jsou zaznamenány některé z nástrojů, které jsem analyzoval. V následujících kapitolách si rozebereme tři typické zástupce detailněji. Jedná se o nástroje Burp Suite, OWASP Zed Attack Proxy (dále jen OWASP ZAP) a Boofuzz.

2.1 Burp Suite

Burp Suite[1] je softwarové řešení od společnosti PortSwigger[2], která se zabývá bezpečnostním testováním webových aplikací. Jedná se o kolekci mnoha užitečných nástrojů pro pokročilé penetrační testování webových aplikačních rozhraní.

2.1.1 Komponenty

Výčet a detailní popis všech komponent je možné nalézt v dokumentaci [11], ze které budu vycházet. Mezi základní moduly aplikace patří:

- **Target** - obsahuje detailní informace o webové aplikaci, kterou testujeme. Obsahuje například přehlednou mapu dané stránky. Umožňuje nám se zaměřit pouze na konkrétní části stránky specifikováním URL[3], které nás při testování zajímají.
- **Proxy** - dovoluje nám operovat tzv. man-in-the-middle (dále jen MITM). Aplikace se ocitne mezi webovým serverem a webovým prohlížečem a tím nám umožňuje jednodušeji analyzovat provoz dat oběma směry.
- **Scanner** - pokročilá technika odhalení zranitelností, která automaticky prochází obsah a testuje na celou řadu zranitelností.

1. `https://portswigger.net/burp`
2. `https://portswigger.net/`
3. `https://en.wikipedia.org/wiki/URL`

- **Intruder** - nástroj pro automatizované testování webových aplikací. Jeho předností je velká přizpůsobitelnost, která nám umožňuje provádět například i pokročilé fuzz testování.
- **Repeater** - nástroj pro manuální úpravu a znovu zaslání HTTP požadavků pospolu s validací přijaté odpovědi.
- **Sequencer** - sofistikovaný nástroj pro analyzování kvality náhodnosti relačních tokenů či jiných datových položek, které mají být nepředvídatelné.
- **Decoder** - nástroj pro dekódování a enkódování aplikačních dat.
- **Comparer** - vizualizace populárního nástroje GNU Diffutils[4] známého především z unixových operačních systémů pod příkazem `diff`.
- **Extender** - slouží pro načítání přídavných rozšíření.
- **Clickbandit** - generování tzv. clickjacking útoků. Jedná se o útoky mířené na uživatele webových stránek, kdy uživatel po provedení zdánlivě nechtěné činnosti může spustit škodlivý kód.
- **Mobile Assistant** - nástroj pro testování mobilních aplikací pomocí platformy Burp Suite.

2.1.2 Fuzz testování

Jak již bylo zmíněno ve výčtu komponent, fuzz testování lze nejjednodušeji provést pomocí modulu Intruder. V tomto modulu specifikujeme cílovou URL adresu, číslo portu a především pak konkrétní validní požadavek, který na server zasíláme. V požadavku pak můžeme do libovolných míst vložit proměnné. Tyto proměnné jsou následně automaticky plněny pomocí definovaných seznamů. Seznam si můžeme specifikovat ručně či vybrat některý předdefinovaný a Intruder pak za nás ve výchozím nastavení automaticky skládá permutace všech vstupních kombinací do námi specifikovaných proměnných.

4. `https://www.gnu.org/software/diffutils/`

Pokud chceme upravit jak jsou jednotlivé permutace skládány, můžeme definovat množinu pravidel, jak má Intruder dané proměnné ze vstupního seznamu plnit. V neposlední řadě můžeme nastavit ještě specifické kódování, prodlevy mezi jednotlivými požadavky, počet vláken, které modul využije a další detailní nastavení. Důležitou součástí je pak i vyhodnocení výsledků. To se v modulu Intruder dělá pomocí funkcionality velmi podobné známého nástroje GNU Grep[5]. Stačí tedy definovat seznam slov, frází či regulárních výrazů, které budou porovnány s obdrženou odpovědí a pokud se v odpovědi objeví, bude konkrétní test považován za neúspěšný.

2.1.3 Průběžná integrace

V dokumentaci produktu [11] je možné nalézt informace o podpoře průběžné integrace. Produkt se nabízí ve třech různých edicích. Jedná se o edice Community, Professional a Enterprise. Verze Community a Professional nabízí pouze manuální grafické nástroje a nenabízí žádnou vestavěnou možnost průběžné integrace. Verze Enterprise se pak liší tím, že již nenabízí manuální nástroje, nýbrž přístup k REST aplikačnímu rozhraní, na které všechny testy deleguje. Pro průběžnou integraci pak využívá zásuvných modulů pro platformy Jenkins a TeamCity, případně generického modulu, který je nezávislý na použitém prostředí průběžné integrace. Tento generický modul pak přináší nástroj pro příkazovou řádku, díky kterému je možno se k danému REST aplikačnímu rozhraní připojit a testy provádět.

2.2 OWASP ZAP

V této sekci vycházím z oficiální dokumentace [12], zdrojového kódu [13] a osobních zkušeností s produktem. OWASP ZAP je open-source[6] nástroj pro bezpečnostní testování šířený pod Apache 2.0[7] licencí. Pomáhá automatizovaně hledat bezpečnostní hrozby ve webových aplikacích již během vývoje samotné aplikace, nicméně je také užitečným nástrojem pro ruční penetrační testování již hotových aplikací.

5. `https://www.gnu.org/software/grep/`
6. `https://en.wikipedia.org/wiki/Open-source_software`
7. `https://en.wikipedia.org/wiki/Apache_License`

Na vývoji nástroje se podílejí velké korporace jako například Google, Microsoft, The Linux Foundation, Mozilla a další. Vývoj nástroje OWASP ZAP je stále velmi aktivní.

2.2.1 Možnosti fuzz testování

OWASP ZAP nabízí pro fuzz testování velmi podobné možnosti, jako výše zmíněný modul Intruder ze sady nástrojů Burp Suite. Opět zde můžeme specifikovat konkrétní dotaz a v něm jednoduše vybrat část dotazu, která bude dynamicky generována. Opět můžeme specifikovat seznam hodnot, které se na námi vyznačených místech budou nahrazovat. Je zde i velmi široký výběr předdefinovaných seznamů pro různé konkrétní účely. Z pohledu bezpečnosti zde nalezneme velmi zajímavé a vyčerpávající seznamy frází pro konkrétní protokoly, služby či s konkrétními útoky a exploity.

2.2.2 Průběžná integrace

OWASP ZAP nabízí pro účely automatizace a průběžné integrace vestavěné webové aplikační rozhraní, kterým je možno nástroj ovládat. V rámci vestavěného aplikačního rozhraní jsme schopni komunikovat pomocí serializačních formátů JSON, HTML či XML. Toto aplikační rozhraní je součástí standardní instalace a ihned po spuštění nástroje je dostupné na portu 8080. Bohužel, modul pro fuzz testování není aktuálně aplikačním rozhraním podporován. Uživatelé již v červnu roku 2015 vyjádřili na platformě GitHub[8], kde je projekt spravován, prosbu o podporu této komponenty aplikačním rozhraním, nicméně do dnešního data nebyla nikým tato funkcionalita implementována.

2.3 Boofuzz

Boofuzz je dalším z mnoha projektů snažící se zjednodušit programátorům a penetračním testerům fuzz testování síťových aplikací. Jedná se opět o open-source nástroj, který je možné nalézt na platformě GitHub [14], odkud pospolu s oficiální dokumentací [15] a osobními

8. `https://github.com/zaproxy/zaproxy/issues/1689`

zkušenostmi čerpám informace pro tuto sekci. Boofuzz je pokračovatelem projektu Sulley [16], který byl aktivně vyvíjen a používán zhruba do roku 2015.

Během psaní této práce byla knihovna přidána i do kyberbezpečnostního inventáře[9] nástrojů skupiny Rawsec[10].

2.3.1 Vlastnosti knihovny

Jak již bylo zmíněno výše, Sulley je předchůdce dnes používaného nástroje Boofuzz. [16] Jedná se o knihovnu pro jazyk Python 2, která je šířena pomocí platformy PyPA[11]. Sulley je určen pro fuzz testování síťových aplikací a zaměřuje se především na jednoduchost a zapouzdření celého procesu. Jak je uvedeno v dokumentaci projektu, spousta nástrojů pro fuzz testování je zaměřena z velké části na generování dat. Sulley tuto funkcionalitu obohacuje o samotné zasílání požadavků a metodické uchování záznamů v databázových souborech. Dále je schopen detekovat, sledovat a kategorizovat zjištěné problémy, případně restartovat cíl, pokud dojde k chybě, aby bylo zamezeno ovlivnění dalších testů.

Projekt Boofuzz pak oproti svému předchůdci přidává následující vlastnosti:

- Online dokumentaci.
- Podpora pro libovolná komunikační média.
- Vestavěnou podporu pro sériové fuzz testování.
- Lepší správu testovacích dat.
- Export výsledků do formátu CSV.
- Rozšiřovatelnou detekci chyb.
- Jednodušší instalaci a použití.
- Méně chyb.

9. `https://inventory.rawsec.ml/tools.html`
10. `https://rawsec.ml/about/rawsec/`
11. `https://www.pypa.io/`

Boofuzz je aktuálně možné spustit na Pythonu verze 2 i 3. Podpora pro Python 3 byla přidána v průběhu psaní diplomové práce. Jelikož se jedná o knihovnu jazyka Python, je použití v prostředí průběžné integrace bez problémů.

2.3.2 Monitor procesů

Jelikož k monitoru procesů není žádná dokumentace ani jiný zdroj relevantních informací, budu v následující sekci vycházet převážně ze zdrojového kódu [14].

Monitor procesů je nástroj, který běží na systému, kde se nachází subjekt našeho testování. Tento nástroj slouží k monitorování testovaného procesu, primárně pak pro detekci pádů a restartů testované aplikace. Pokud chceme při fuzz testování pomocí Boofuzz knihovny využít monitor procesů, stačí při testování specifikovat doménové jméno či případně IP adresu, kde monitor naslouchá, číslo portu na kterém naslouchá a v neposlední řadě příkaz v příkazové řádce, kterým je testovaná služba spouštěna, aby mohlo dojít ke znovuspuštění služby po jejím pádu.

Monitor procesů je součástí knihovny a je možné jej nalézt v repozitáři v souborech `process_monitor.py` a `process_monitor_unix.py`. Tyto dva soubory, obsahující skripty jazyka Python 2, jsou určené pro běh na systému, kde se nachází subjekt testování. Ve verzi pro Windows jsou navíc pro korektní běh vyžadovány další předinstalované nástroje:

- **pydbg**[12] (striktně 32 bitová verze) - rozhraní sloužící k ladění chyb pro win32 aplikace.
- **pydasm**[13] - knihovna jazyka C pro jednoduché parsování Intel x86 instrukcí.

2.3.3 Důležité funkce a datové struktury

Knihovna Boofuzz se skládá z několika základních tříd, rozhraní a funkcí, které poskytují hlavní funkcionalitu pro fuzz testování sí-

12. `https://github.com/Fitblip/pydbg`
13. `https://github.com/jtpereyda/libdasm`

ťových protokolů. Jmenovitě jde například o třídy `Session`, `Target`, `SocketConnection` či `FuzzLoggerCsv`.

Pro samotné fuzzování konkrétních datových entit pak slouží takzvaná primitiva. Primitiva se dělí převážně dle typu dat, které modifikují. Jedná se o funkce, kterými specifikujeme konkrétní část požadavku a tato funkce se nám pak může starat o mutování předané výchozí položky.

`Session`

Třída reprezentující relaci mezi uzlem provádějícím testování a množinou takzvaných cílů, tedy definic uzlů, na které je útok prováděn. Konstruktor třídy má mnoho parametrů s výchozími hodnotami, které můžeme při instanciaci upřesnit. Můžeme jí například předat námi definované logovací objekty, specifikovat různé časové intervaly použité při provádění útoků či například definovat callback funkci. Ta se provede po provedení každého testu a v té se můžeme dostat k odpovědi, kterou nám testovaný server pro každý dotaz zaslal.

`Target`

Třída reprezentující cíl testování. Hlavním parametrem konstruktoru je objekt reprezentující spojení s testovanou aplikací, který implementuje rozhraní `ITargetConnection`. Dalšími volitelnými parametry jsou pak informace o tom, zda na testovaném systému běží monitor procesů, případně na jakém portu jej Boofuzz může nalézt.

`SocketConnection`

`SocketConnection` je jednou z předdefinovaných implementací rozhraní `ITargetConnection` a reprezentuje konkrétní server, který testujeme. Jeho hlavními parametry jsou doménové jméno či IP adresa testovaného zařízení. Následně port, na kterém server naslouchá a protokol, přes který chceme požadavky zasílat.

K dispozici jsou následující předdefinované protokoly: tcp, udp ssl, raw-l2, raw-l3. TCP a UDP jsou klasické spojení skrze zmíněné transportní protokoly. SSL je pak nadstavba nad TCP o šifrovanou

komunikaci pomocí SSL/TLS. Raw-l2 a Raw-l3 pak označují rámce či datagramy na linkové či síťové vrstvě.

`IFuzzLogger`

Rozhraní popisující logovací objekt. Implementací tohoto rozhraní můžeme vytvořit libovolnou logovací třídu dle našich potřeb. Boofuzz nabízí několik předdefinovaných logovacích tříd pro čitelné textové logování či logování ve formátu CSV.

Jelikož Boofuzz knihovna automaticky bez ohledu na další definovaná logování zároveň loguje veškeré informace do databázového souboru, není třeba vždy rozšiřujícího logování využít.

2.3.4 Vytváření definic požadavků

Nyní se podíváme na způsob vytváření definic požadavků, nad kterými se následně generují mutace a vytvářejí se tak konkrétní požadavky na testovanou síťovou aplikaci.

- **`s_initialize`** - započne tvoření celé definice požadavku. Jediným povinným parametrem je název dané definice požadavku, který musí být unikátní v rámci celého běhu programu. Pokud není unikátní, je vyvolána příslušná výjimka. Tato funkce nevrací referenci na vytvořenou definici, pouze jej globálně inicializuje.
- **`s_get`** - na základě jména definice požadavku vrátí objekt, který tuto definici popisuje. Pokud jméno není specifikováno, funkce vrátí naposledy utvářenou, tedy aktuálně aktivní, definici.
- **`s_mutate`** - mutuje aktivní definici požadavku do následující varianty. Pokud jsou mutace vyčerpány, vrátí funkce hodnotu `False`.
- **`s_num_mutations`** - vrátí počet mutací, které lze s danou definicí požadavku provést. Tedy kolik různých požadavků lze z dané definice požadavku vygenerovat.
- **`s_render`** - slouží spíše pro ladící účely. Vrátí řetězec, který popisuje aktuálně vytvářenou definici požadavku. Neukazuje

nicméně žádné mutace vstupních dat, pouze validní hodnoty požadavku.

- `s_switch` - přepne aktivní definici požadavku na jinou podle zadaného jména.

Knihovna Boofuzz také umožňuje práci s tzv. bloky. Blok je možné použít, pokud chceme periodicky opakovat či znovupoužít určitou část definice požadavku. Konkrétní funkce, kterými lze bloky vytvářet a modifikovat lze nalézt v dokumentaci [15].

2.3.5 Vestavěná primitiva

Boofuzz také nabízí množinu vestavěných primitiv, které lze použít při samotné definici dat vytvářených definic požadavků. Tedy po inicializaci požadavku pomocí `s_initialize` můžeme zavolat jednu z následujících metod a do vytvářené definice požadavku se vloží blok dat. Každá z těchto funkcí má atribut `value`, který slouží pro definici řetězce, ten by měl být na daném místě validního dotazu. Pro lepší pochopení použití popsaných primitiv následuje příklad v sekci 2.3.6.

- `s_binary` - převede řetězec bajtů zadaný pomocí hexadecimálního formátu, který je jí předán, do binární podoby a vloží jej do definice požadavku.
- `s_delim` - slouží pro definici oddělovače. Jejím povinným parametrem je hodnota, která má být obsažena ve validním požadavku a z této hodnoty se následně vytvářejí různé mutace. Vytváří se celkem 44 mutací pro libovolně zadaný validní oddělovač. Mutace spočívá například v opakování více oddělovačů za sebou, přidáváním různých bílých znaků či vkládáním znaků, které by mohly potenciálně způsobit chybu zpracování takového oddělovače.
- `s_group` - reprezentuje množinu statických řetězců. Jedna mutace odpovídá jednomu řetězci z předaného pole. Samy o sobě se žádné mutace nevytváří.
- `s_random` - zcela náhodně mutuje vstupní řetězec. Využívá knihovnu `Random`, která je standardně dodána s instalací Pythonu.

Výstupem nemusí být jen validní ASCII řetězec, ale i libovolná sekvence bajtů. Je možné nastavit počet mutací, které jsou vytvářeny. Ve výchozím stavu je vytvářeno 25 zcela náhodných mutací.

- `s_static` - vloží staticky předaný řetězec do definice požadavku. Bez jakýchkoliv mutací.

Primitivum `s_group` obsahovalo chybu, kdy prázdný řetězec nebyl akceptován jako výchozí hodnota dané části dotazu. Při mutování jiného primitiva tak v části dotazu popsaném pomocí primitiva `s_group` nebyla použita výchozí hodnota, tedy prázdný řetězec, nýbrž první z definovaných mutací. To vyústilo v situaci, kdy generované testovací scénáře obsahovaly mutace dvou částí zároveň a nedocházelo tak ke korektnímu postupnému testování. Pokud například testovaná aplikace spadla již při zpracovávání první mutované části, nebyla tak odhalena chyba způsobená druhou mutovanou částí zaslaného požadavku. Tuto chybu jsem v knihovně Boofuzz opravil[14].

Poslední a nejdůležitější předpřipravené primitivum je reprezentováno funkcí `s_string`. Jedná se o primitivum, které je vytvořeno pro mutování libovolných textových řetězců. Lze nastavit i kódování, pro které jsou mutace vytvářeny. Zde probíhá mutace nejčastěji aplikováním předdefinovaných řetězců, které jsou vkládány a nahrazeny za původní validní řetězec. Výčtově se jedná o následující řetězce a mutace:

- Řetězce pro Directory traversal[15] útoky.
- Vkládání sekvencí speciálních znaků (např. znaky s ASCII hodnotou menší než 32).
- Různé sekvence formátovacích řetězců[16].
- Command injections[17] řetězce.
- Sekvence binárních řetězců.

14. https://github.com/jtpereyda/boofuzz/pull/291
15. https://www.owasp.org/index.php/Path_Traversal
16. https://www.owasp.org/index.php/Format_string_attack
17. https://www.owasp.org/index.php/Command_Injection

- Řetězce obsahující známé exploity z existujících aplikací (například exploit z programu sendmail[18]).
- Generované velmi dlouhé řetězce znaků, do kterých jsou náhodně vkládány různé binární sekvence.

Ve výchozím stavu vytváří `s_string` přibližně 1440 mutací. Toto lze výrazně ovlivnit zadáním maximální délky vygenerovaných mutací, jelikož jedna z nejčastějších mutací je právě generování dlouhých řetězců. Mutace dlouhých řetězců jsou generovány jako násobky jednotlivých předdefinovaných znaků. Tyto sekvence se v testovací množině velmi často opakují, pouze s odlišnou délkou. Po zarovnání vygenerovaných dlouhých řetězců na zadanou délku se smažou duplicitní mutace, čímž se výrazně zredukuje jejich celkový počet.

2.3.6 Generování výsledných požadavků

Definice požadavku pro jednoduché fuzz testování HTTP komunikace může vypadat následovně:

```
s_initialize(name="Request")
s_group("Method", ['GET', 'HEAD', 'POST', 'PUT', 'DELETE'])
s_delim(" ", name='space-1')
s_string("/index.html", name='Request-URI')
s_delim(" ", name='space-2')
s_string('HTTP/1.1', name='HTTP-Version')
s_static("\r\n", name="Request-Line-CRLF")
s_static("\r\n", "Request-CRLF")
```

Nejprve pomocí funkce `s_initialize` vytvoříme novou definici požadavku. Následně pomocí `s_group` definujeme množinu statických řetězců, které v tomto případě reprezentují různé HTTP metody[19]. Následuje definice oddělovače, kde správným oddělovačem je jedna mezera, která je předána funkci `s_delim`. Tato funkce následně vytváří mutace pro předaný oddělovač. Poté dochází k mutaci URI[20] následované mutováním oddělovače. Následně mutujeme samotnou

18. `http://www.postfix.org/sendmail.1.html`
19. `https://tools.ietf.org/html/rfc7231#page-21`
20. `https://tools.ietf.org/html/rfc3986`

verzi HTTP požadavku a nakonec vložíme již staticky, bez mutací, dva oddělovače pro oddělení HTTP těla od hlavičky.

Jednotlivé požadavky jsou generovány tak, že se všechny mutované části definice požadavku zafixují na validní hodnotu a mutuje se postupně pouze jedno primitivum. Počet mutovaných požadavků tedy roste s přibývajícími primitivy lineárně.

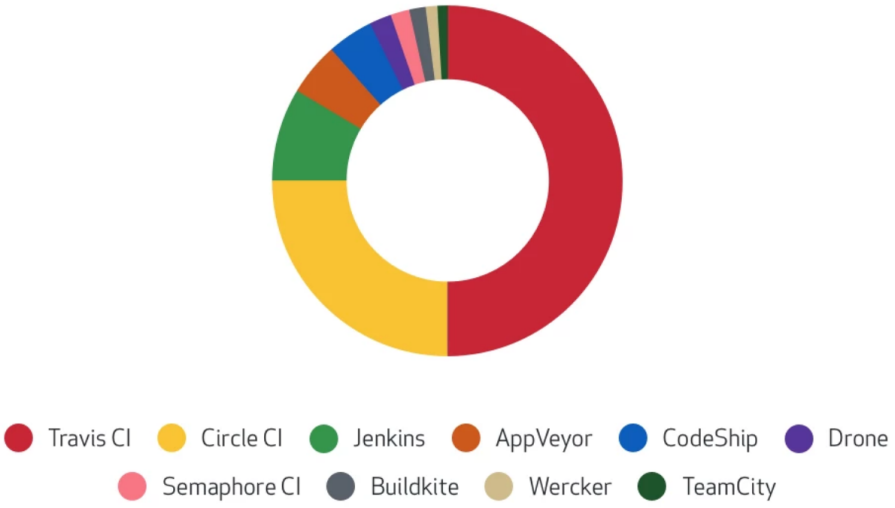
2.4 Shrnutí a zhodnocení popsaných nástrojů

Aplikace Burp Suite umožňuje provádět fuzz testování webových aplikačních rozhraní, nicméně v neplacených variantách neumožňuje jednoduché použití v systémech pro průběžnou integraci. Nástroj OWASP ZAP umožňuje taktéž fuzz testování vhodné pro náš účel, nicméně jeho aplikační rozhraní neumí modul provádějící fuzz testování ovládat. Tudíž také nelze jednoduše použít v prostředí pro průběžnou integraci. Boofuzz je knihovnou, kterou lze pro fuzz testování webových aplikačních rozhraní použít a jelikož se jedná pouze o knihovnu, není použití v prostředích průběžné integrace problém.

3 Systémy průběžné integrace

Dle slov Martina Fowlera[1], autora několika knih o vývoji softwarových produktů, je průběžná integrace technika vývoje softwaru, při které všichni členové teamu pravidelně integrují jejich práci. Obvykle pak minimálně jednou denně. Každá integrace je pak verifikována automatickým překladem pospolu se spuštěním testů pro co nejdřívější detekci integračních chyb. Mnoho týmů potvrzuje, že tento přístup výrazně redukuje problémy s integrací a dovoluje týmu vyvíjet soudržný software výrazně rychleji. [17]

V roce 2017 zveřejnila služba GitHub žebříček nejpoužívanějších systémů pro průběžnou integraci.



Obrázek 3.1: Žebříček nejpoužívanějších systémů průběžné integrace platformy GitHub [18]

Na obrázku 3.1 můžeme vidět, že v roce 2017 byl nejpopulárnějším nástrojem průběžné integrace Travis CI. Hned za ním, ovšem s relativně velkou ztrátou se umístil CircleCI, na třetím místě pak Jenkins. Nutno podotknout, že jelikož se jedná o statistiky nasbírané

1. `https://martinfowler.com/`

pouze na platformě GitHub, některá často používaná prostředí pro průběžnou integraci zde nejsou zastoupena, jelikož mohou být více spjata s jiným správcem Git repozitářů. To se týká například prostředí Bamboo, které spravuje společnost Atlassian[2], která zároveň poskytuje správce repozitářů Bitbucket[3] a je častěji používáno v lokálním prostředí společnosti.

3.1 Travis CI

Travis CI[4] je SaaS[5] (software jako služba) platforma pro open-source software úzce spolupracující s platformou GitHub. [19]

Tato služba je provozována zdarma pro všechny veřejné GitHub repozitáře. Pro privátní repozitáře je pak zpoplatněna pomocí několika různých plánů. Jednou z výhod platformy Travis CI je velmi přehledná a kompletní dokumentace, která ukrývá mnoho konfiguračních příkladů. [20]

3.1.1 Jak Travis CI funguje?

Jakmile vývojář nahraje na GitHub repozitář nový přírůstek, GitHub pomocí webhooku [21] pošle webový dotaz na služby platformy Travis CI, která začne provádět následující operace:

- Naklonování repozitáře a tzv. `checkout` na nově nahraný commit.
- Přečtení konfiguračního souboru `travis.yml`.
- Nastartování kontejnerů, ve kterých se další prováděné akce provádějí.
- Spuštění překladu, testů a případných dalších skriptů.
- Reportování úspěchu či selhání prováděných operací.

2. `https://www.atlassian.com/`
3. `https://bitbucket.org/`
4. `https://travis-ci.org/`
5. `https://en.wikipedia.org/wiki/Software_as_a_service`

- Pokud je tak nastaveno a pokud předchozí kroky dopadly úspěšně, provede se nasazení přeložené a otestované aplikace.

Veškeré výše uvedené kroky se dají nastavit v konfiguračním souboru, který bude vzápětí popsán. [22] [20]

Tuto službu využívá aktuálně přes 900 tisíc repozitářů. Přes 600 tisíc uživatelů využívá služby Travis CI pro automatizované testování jejich aplikací. [23]

3.1.2 Konfigurace

Pro počátek je zapotřebí se přihlásit skrze GitHub autentizaci na portálu `https://travis-ci.org` a aktivovat webhook pro repozitář, pro který chcete Travis CI použít. Pak již jen přidáte konfigurační soubor do kořenové složky vašeho adresáře. [20]

Srdcem konfigurace celé průběžné integrace je soubor `.travis.yml`. V tomto souboru můžeme specifikovat následující údaje. [20]

- Primární programovací jazyk projektu.
- Konkrétní verze (běhová prostředí) vybraného jazyka.
- Další potřebné závislosti.
- Případné služby a jejich konfigurace.
- Specifikování překladu aplikace.
- Specifikování spuštění testů.
- Kroky k nasazení aplikace.
- Případné další položky specifické k vybranému jazyku.

Jak název konfiguračního souboru napovídá, konfigurace je uchována v serializačním formátu YAML.

Travis CI obsahuje předpřipravené prostředí pro více než 30 známých programovacích jazyků či frameworků. Také podporuje přes 40 známých prostředí pro nasazení webových či jiných aplikací. [22]

Příklad konfiguračního souboru:

```
language: python
python:
  - '3.5'
  - '3.4'
  - '2.7'
matrix:
  include:
    - python: '3.5'
      env: EXTRA_TESTS=true
    - python: '3.4'
      env: EXTRA_TESTS=true
script: env $EXTRA_TESTS ./test.py $TEST_SUITE
```

Tento příklad pochází z oficiální dokumentace [22]. Jedná se o spouštění testů pro různé verze jazyka Python, které jsou spouštěny souborem `test.py`. V příkladu je vidět, že pomocí klíče `env` můžeme jednoduše nastavovat systémové proměnné operačního systému. Klíč `script` poté slouží ke spuštění uživatelsky definovaného skriptu spuštěného ve výchozím shellu.

Operační systém, ve kterém se průběžná integrace provádí, lze v konfiguraci specifikovat také. Slouží k tomu klíč `os`, který lze nastavit na hodnoty `windows`, `linux` či `osx`. Pokud tento klíč nespecifikujeme, výchozím systémem je Linux, konkrétně distribuce Ubuntu ve verzi 14.04 (známá pod pojmem Trusty). Konkrétní verzi operačního systému lze specifikovat[6]. [22]

3.1.3 Podpora kontejnerizace

Podpora pro platformu Docker[7] je dnes pro většinu nejpoužívanějších systémů pro průběžnou integraci standardem. To platí samozřejmě i pro Travis CI. Podporu pro Docker lze aktivovat v konfiguračním souboru pod klíčem `services`. Jakmile pod tento klíč přidáme položku `docker`, můžeme stejnojmenný příkaz v sekci uživatelských skriptů začít používat. Jediné omezení je aktuálně pro systém macOS, na který prozatím tato služba není povolena.

6. `https://docs.travis-ci.com/user/reference/overview/`
7. `https://www.docker.com/`

3.2 CircleCI

CircleCI je, stejně jako Travis CI, SaaS (software jako služba) platforma pro open-source software. Pokrývá stejné případy užití jako výše zmíněný Travis CI. CircleCI přináší velkou výhodu spočívající v podpoře více platforem pro správu repozitářů.

Zatímco Travis CI podporuje pouze platformu GitHub, CircleCI umožňuje integraci i s dalšími správci repozitářů. Jedná se pak například o platformu Bitbucket[8] [24] nebo GitLab[9] [25]. Podpora operačních systémů je ovšem u CircleCI více limitována. Aktuálně je podporován pouze Linux, Android a iOS. Windows aktuálně není podporován. Aktuálně je také podporována pouze `amd64` architektura.

3.2.1 Funkce platformy

Jak již bylo zmíněno, princip činnosti a funkcionalita je velmi podobná výše popsané platformě Travis CI. Konkrétní výčet poskytovaných funkcí je následující: [24]

- Široká podpora programovacích jazyků.
- Konfigurovatelná prostředí.
- Flexibilní alokace zdrojů.
- Přístup přes SSH pro jednoduché ladění.
- Možnost využití lokálních zdrojů pro ještě snadnější ladění.
- Zdokonalené využití mezipaměti.
- Zabezpečená prostředí.
- Možnost paralelizace.

CircleCI podporuje také virtualizaci aplikací pomocí platformy Docker.

8. `https://bitbucket.org/`
9. `https://about.gitlab.com/`

3.2.2 Konfigurace

Konfigurace je velmi podobná jako v případě platformy Travis CI. Opět se jedná o definici jediného YAML souboru, který je pro CircleCI verze 2.0 a vyšší umístěn na následující relativní cestě (relativní vzhledem ke kořenovému adresáři repozitáře) `.circleci/config.yml`. Pro verzi CircleCI 1.0 pak stačilo mít soubor `config.yml` v kořenové složce adresáře.

Důležité klíče konfiguračního souboru:

- `version` - verze konfiguračního souboru.
- `orbs` - definice takzvaných orbů. Jedná se o znovupoužitelné balíčky CircleCI konfigurací sdílené napříč projekty. Můžeme použít již předdefinované orby, případně si vytvořit vlastní. Více v dokumentaci [26].
- `commands` - definice příkazů. Jeden příkaz má své argumenty a skládá se z několika kroků, které reprezentují příkazy ve zvoleném shellu.
- `executors` - pomocí spouštěčů definujeme prostředí, ve kterém se jednotlivé kroky budou provádět. Můžeme zde definovat konkrétní operační systém, docker image či případně shelly, které na daném systému budou nainstalovány a mnoho dalšího.
- `jobs` - seznam prací, které se mají vykonat. Práce je kolekce kroků, podobně jako v případě příkazů (`commands`). Jednotlivé práce se pak dají použít v takzvaných workflows, viz níže.
- `workflows` - jedná se o orchestraci prací. Každé workflow obsahuje seznam prací a takzvané triggery, tedy specifikaci situací, při kterých se má daný workflow spustit.

Detailněji je možné si konfiguraci projít v dokumentaci [26]. Je zde celá řada příkladů pro konkrétní a často řešené situace. Pojďme si tedy jeden příklad ukázat:

```
version: 2
jobs:
  build:
    docker:
      - image: circleci/<language>:<version TAG>
    steps:
      - checkout
      - run: <command>
  test:
    docker:
      - image: circleci/<language>:<version TAG>
    steps:
      - checkout
      - run: <command>
workflows:
  version: 2
  build_and_test:
    jobs:
      - build
      - test
```

V tomto příkladu vidíme definované dvě práce a jeden workflow, které tyto práce spojuje do paralelní kompozice. Můžeme si všimnout, že v definicích jednotlivých prací je využito platformy docker.

3.3 Bamboo

Bamboo je prostředí pro průběžnou integraci od společnosti Atlassian. V této kapitole budu čerpat především z oficiální dokumentace [27] a ze zkušeností s platformou ze své lokální instalace produktu.

Platformu lze použít pro překlad, testování a nasazení vyvíjené aplikace. Bamboo disponuje vestavěnou integrací se všemi dalšími produkty společnosti Atlassian pro podporu a usnadnění vývoje softwaru jakými jsou například Jira[10], Bitbucket[11] či Fisheye[12].

10. https://www.atlassian.com/software/jira
11. https://cs.atlassian.com/software/bitbucket
12. https://cs.atlassian.com/software/fisheye

Od předchozích dvou typů průběžné integrace se liší tím, že není distribuován jako SaaS, nicméně je šířena jako klasická serverová aplikace. To přináší mnoho odlišností.

Veškeré komponenty Bamboo jsou psány v programovacím jazyce Java a běží na JVM[13]. Tedy jak pro server, tak pro agentské stanice, je potřeba mít nainstalovaný JRE[14]. Co to jsou agentské stanice, dále jen agenti, bude vysvětleno níže.

3.3.1 Základní koncepty a pojmy

Z dokumentace [27] můžeme vyčíst následující základní pojmy a konstrukty, které slouží k provádění operací na serveru kontinuální integrace platformy Bamboo.

- **Projekt** - poskytuje reportování a nastavení oprávnění. Může obsahovat několik plánů.
- **Plán** - obsahuje několik etap, které následně sekvenčně vykonává. Specifikuje kdy a za jakých podmínek se celý plán spouští.
- **Etapa** - může obsahovat jednu nebo více prací. Veškeré práce jsou zpracovány paralelně na více agentech (pokud jsou k dispozici). Může produkovat artefakty, které jsou dostupné již po skončení dané etapy.
- **Práce** - množina úloh, které jsou vykonávány sekvenčně na jednom konkrétním agentovi. Definuje artefakty, které jsou v rámci etapy vytvářeny.
- **Úloha** - malá diskrétní operace. Příkladem může být naklonování repozitáře, spuštění skriptu či provedení Maven cíle.

3.3.2 Agenti

Výše jsme uvedli, že jednotlivé práce jsou spouštěny na takzvaných agentech. Agent je v pojetí platformy Bamboo služba, který poskytuje schopnosti pro spuštění nějaké práce. [27]

13. https://en.wikipedia.org/wiki/Java_virtual_machine
14. https://www.javaworld.com/article/3304858/what-is-the-jre-introduction-to-the-java-runtime-environment.html

Agenti se dělí do dvou základních kategorií:

- **Lokální agent** - běží v rámci procesu Bamboo serveru jako jednotlivá vlákna ve stejné instanci JVM.
- **Vzdálený agent** - běží na jiném systému než Bamboo server. Na takovém systému musí běžet nástroj pro vzdáleného agenta, který lze získat z nainstalované instance Bamboo serveru. Jedná se o běžný JAR[15] balík, kterému stačí v parametru předat jméno hosta aktivního Bamboo serveru. Každý vzdálený agent tedy běží ve své instanci JVM a s Bamboo serverem komunikuje pomocí JMS[16].

Jak je uvedeno v definici výše, agent definuje takzvané schopnosti. Agent je tedy schopen pouštět pouze takové práce, pro které má schopnosti. I proto je vhodné a doporučené používat vzdálené agenty, jelikož nejsou limitováni zdroji a schopnostmi systému, na kterém běží samotný Bamboo server.

Schopnosti agentů se kategorizují do 4 základních skupin.

- **Spustitelnost** - jakýkoliv spustitelný binární soubor (například Maven[17]).
- **JDK** - konkrétní verze frameworku Java Development Kit.
- **Verzovací systém** - klientská aplikace verzovacího systému (například Git[18]).
- **Vlastní schopnosti** - uživatelsky definované schopnosti, například: `operating.system=ubuntu16.04`.

3.3.3 Kontejnerizace

Systém pro průběžnou integraci Bamboo podporuje virtualizační platformu Docker a to hned na několika úrovních:

- Celý Bamboo server může běžet v Docker kontejneru.

15. `https://en.wikipedia.org/wiki/JAR_(file_format)`
16. `https://en.wikipedia.org/wiki/Java_Message_Service`
17. `https://maven.apache.org/`
18. `https://git-scm.com/`

- Bamboo práce mohou běžet v Docker kontejnerech.
- Samotné úlohy mohou používat Docker kontejnery.

Při návrhu celého procesu a architektury průběžné integrace Bamboo v souvislosti s platformou Docker je ovšem potřeba myslet na nevýhody [28], které může přinést situace, kdy budeme v rámci Docker kontejneru používat znovu platformu Docker.

3.4 Jenkins

Jenkins je architektonicky podobný platformě Bamboo. Opět se nejedná o SaaS řešení, nicméně o software běžící lokálně. Taktéž se jedná o cross-platform aplikaci založenou na programovacím jazyku Java. Pro běh platformy Jenkins nám tedy stačí nainstalované běhové prostředí pro Javu 8 a Docker. V této kapitole budu čerpat především z oficiální dokumentace [29].

Platforma Jenkins je velice populární. V roce 2018 byl proveden průzkum pro více než 10 000 Java vývojářů, ze kterého vyplynulo že 57% z nich používá pro prostředí průběžné integrace právě Jenkins. [30] Velký podíl na tom má fakt, že platforma Jenkins je kompletně open-source a umožňuje vývojářům tvořit vlastní přídavné moduly.

3.4.1 Pipelines

Jenkins uchovává své procesy v takzvaných *pipelines*. Jenkins Pipeline (nebo zkráceně *pipeline*) je sada pluginů které pomáhají implementovat a integrovat takzvané *continuous delivery pipelines* do systému Jenkins. [29]

Continuous delivery pipeline (v kontextu platformy Jenkins) je pak automatizovaný proces přetavení softwaru z verzovacího systému k uživatelům. Proces zahrnuje spolehlivý a opakovatelný překlad pospolu s jeho testováním a doručením. *Pipeline* je možné specifikovat pomocí speciální syntaxe. Ta bývá obvykle zapsána v souboru `Jenkinsfile` v kořenové složce repozitáře. [29]

Jenkins poskytuje dva hlavní způsoby jakým můžeme *pipelines* specifikovat. Jedná se o deklarativní a skriptovací syntax. Deklarativní způsob je novější a preferovanější způsob, který obsahuje bohatší

možnosti a je navržen pro jednodušší čtení i psaní popisu *pipeline* funkcionality. [29]

Deklarativní způsob zapisování *pipeline* spíše připomíná strukturou konfigurační soubor podobný konfiguraci pro Travis CI a CircleCI. Jednotlivé kroky a výrazy však poté podléhají Apache Groovy[19] syntaxi[20].

Skriptovací způsob je zcela postaven nad Apache Groovy syntaxí, ve které je zapsána. Většinu funkcionality, kterou Groovy poskytuje, je dostupná i ve skriptovacím způsobu zápisu *pipeline*.

3.4.2 Kontejnerizace

Co se týče kontejnerů, zde máme z pohledu platformy Jenkins volnou ruku. Záleží jen na nás jakou formu virtualizace aplikací si zvolíme. Jelikož Jenkins pracuje se systémem, na kterém běží a umožňuje přistupovat k jeho zdrojům, nejsou zde žádné limitace.

Opět zde máme pohled z více úrovní, kdy samotný Jenkins je doručován i v podobě Docker obrazu. Samotné *pipelines* pak umožnují pracovat s platformou Docker skrze svou syntaxi. Stačí specifikovat klíč `docker`, do kterého vložíme konkrétní Docker obraz, ve kterém chceme jednotlivé kroky pouštět.

3.5 Limitace kladené průběžnou integrací na implementované řešení

Jak je vidět v tabulce 3.1, všechny čtyři zkoumané nástroje pro průběžnou integraci umožňují pouštět skripty a podporují aplikačně virtualizační platformu Docker.

Pokud tedy bude vyvíjený nástroj zapouzdřen v Docker obraze, neměl by být problém s jeho užíváním na žádném z výše vypsaných prostředí pro průběžnou integraci.

19. `http://groovy-lang.org/`
20. `http://groovy-lang.org/syntax.html`

Tabulka 3.1: Podporované funkce a vlastnosti průběžné integrace

Služba průběžné integrace	**Docker**	**PowerShell**	**Bash**
Travis CI	Ano	Ano	Ano
CircleCI	Ano	Ano	Ano
Bamboo	Ano*	Ano*	Ano*
Jenkins	Ano**	Ano**	Ano**

* musí podporovat klientští agenti
** musí podporovat systém, na kterém Jenkins běží

4 Dokumentační formáty webových aplikačních rozhraní

API Description Languages (zkráceně API DL) je třída jazyků určená pro dokumentaci webových aplikačních rozhraní. Podle prezentace [31] existují 4 základní účely, ke kterým tyto jazyky slouží:

- **Blueprint** - podrobný plán a předpis pro samotnou implementaci webového aplikačního rozhraní.
- **Kontrakt** - dohoda o tom, jak spolu budou jednotlivé části systému pracovat.
- **Metadata** - souhrn doplňujících informací, které mohou být použity při vývoji či používání aplikačního rozhraní.
- **Dokumentace**

Již v roce 2009 vznikl jazyk WADL [32] za účelem popisu webových aplikací od společnosti Sun Microsystems Inc. Tento jazyk je založen na serializačním formátu XML. Dle Laury Heritage [32] existuje hned několik důvodů proč je tento formát, případně další z něho derivované, nevhodný pro popis webových aplikačních rozhraní. Jedná se o následující faktory:

- XML se ukázal být pro člověka těžko zpracovatelným.
- Obvykle bývá strojově generován, tedy není vhodný jako blueprint pro netechnické uživatele.
- Jeho gramatika není schopna vyjádřit veškeré informace o webovém aplikačním rozhraní založené na architektuře REST[1].

Z těchto důvodů začaly vznikat nové jazyky a formáty, které známe dnes. [33]

1. `https://en.wikipedia.org/wiki/Representational_state_transfer`

4.1 OpenAPI

Prvním ze zástupců dokumentačních formátů je zmiňované OpenAPI, dříve známo také pod pojmem Swagger. Pojem Swagger je aktuálně vyhrazen pro sadu open-source nástrojů, které nějakým způsobem operují s dokumentačním formátem OpenAPI. [34]

OpenAPI nám umožňuje kompletně popsat webové aplikační rozhraní pospolu s:

- Definicí koncových bodů rozhraní a operacemi nad nimi.
- Parametry jednotlivých operací.
- Autentizační metodou.
- Kontaktními informacemi, licencí, podmínkami užití a dalšími informacemi.

OpenAPI využívá serializačních formátů YAML či JSON. [34] Vývojář si může vybrat jaký formát použít, nicméně jak v návodech v dokumentaci tak mezi vývojáři je preferovanější formát YAML, díky jeho obecně příznivější čitelnosti.

Aktuálně jsou aktivně používány dvě verze OpenAPI specifikace. Jedná se o verzi 2 a 3. Jelikož mezi těmito verzemi jsou nemalé rozdíly, aktuálně ještě všechny nástroje nepodporují verzi OpenAPI 3 a naopak některé nově vzniklé nástroje již nejsou zpětně kompatibilní s verzí 2. Seznam nástrojů doporučených pro práci s OpenAPI dokumentací je možné nalézt na webu[2] dvou nadšenců do OpenAPI projektu.

4.2 API Blueprint

V této sekci budu vycházet z oficiální dokumentace API Bluperint jak na jejich hlavním webu [35], tak v dokumentaci hlavního repozitáře API Blueprint [36] a také z vlastních zkušeností s tímto formátem.

Syntax dokumentačního formátu API Blueprint je založena na populární syntaxi Markdown[3], kterou můžeme znát například z prostředí GitHub, Bitbucket, Reddit, Stack Exchange a dalších populár-

2. `https://openapi.tools/`
3. `https://daringfireball.net/projects/markdown/`

ních webových nástrojů. Konkrétní rozdělení a detailní popis pro jednotlivé sekce dokumentu lze nalézt v dokumentaci [36].

API Blueprint dále definuje tzv. MSON (*Markdown Syntax for Object Notation*) syntax, což je rozšíření jazyka Markdown zaměřené na popis objektů. Je zaměřena na popis datových struktur známých serializačních formátů jako například JSON, XML či YAML. MSON může být také konvertováno mezi výše vypsanými serializačními formáty.

Stejně jako v případě OpenAPI, i pro API Blueprint existuje celá řada volně šířených nástrojů, které je možné nalézt na oficiálním webu [35].

4.3 RAML

RAML je zkratka pro *RESTful API Modeling Language*. Ve své nejnovější verzi 1.0 používá RAML jako základní formát YAML[4] verze 1.2. Volně šiřitelné nástroje pro práci s formátem RAML je možné nalézt na webových stránkách věnovaným tomuto formátu [37]. Jelikož se společnost MuleSoft, autor formátu RAML, rozhodla také připojit [5] k Open API Initiaitve, open-source komunitě spravující OpenAPI formát, nebudu tento formát dále rozepisovat.

4. `https://yaml.org/spec/1.2/spec.html`
5. `https://blogs.mulesoft.com/dev/api-dev/open-api-raml-better-together/`

5 Implementované řešení

Implementované řešení se skládá ze tří hlavních modulů. Jedná se o `parser`, který se stará o zpracování dokumentace webového aplikačního rozhraní a generování seznamu cílů. Druhým modulem je pak samotný `fuzzer`, který dostane na vstupu vygenerovaný seznam cílů a začne provádět samotné fuzz testování webového aplikačního rozhraní. Třetím modulem je pak `reporter`, který převezme výsledky testování komponenty `fuzzer` a vygeneruje výsledky testování v uživatelsky přívětivé podobě.

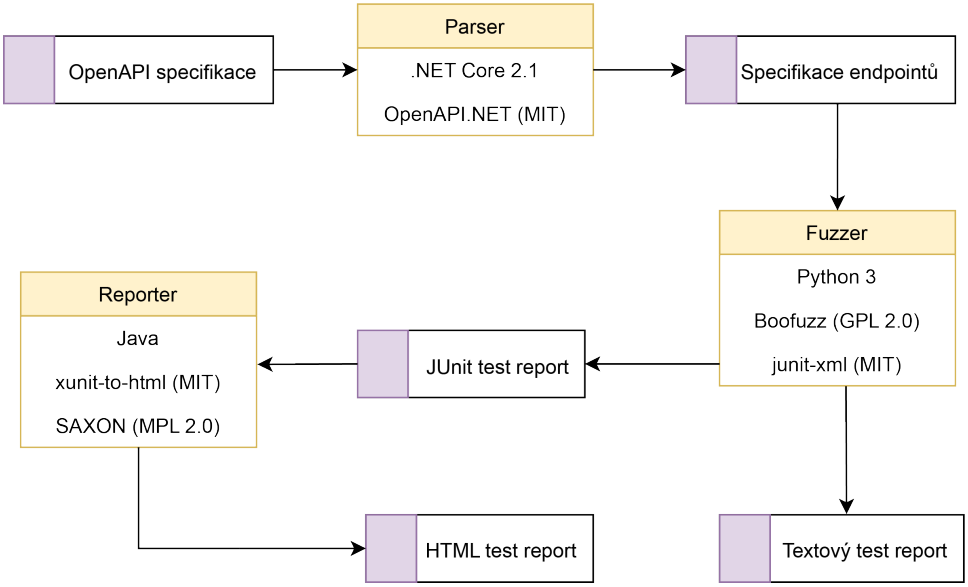
5.1 Návrh

Proces návrhu prošel iterativním vývojem. Návrh architektury jsem postupně upravoval a zlepšoval pomocí získaných zkušeností s problematikou a upřesněním specifikací na vytvářený nástroj. V první verzi návrhu se mělo řešení skládat ze 4 oddělených částí a to jmenovitě z `konvertoru` jednotlivých API dokumentací, `parseru`, `fuzzeru` a `reporteru`.

Nejprve jsem během vývoje zjistil, že komponenta `reporter` není potřeba, jelikož `fuzzer` může rovnou obsahovat funkcionalitu pro jednoduché generování univerzálního formátu, který může být jednoduše vizualizován již existujícími nástroji. Toto tvrzení je sice stále pravdivé a mnoho prostředí pro průběžnou integraci umožňuje zobrazení univerzálních formátů pro popis výsledků testů, avšak ukázalo se, že například modul v prostředí Bamboo parsuje výsledky testů tzv. in-memory. To znamená, že načte veškeré testy do paměti a následně se je pokusí zpracovat do vhodné formy pro zobrazení uživateli. Jelikož implementovaný nástroj může pro některá aplikační rozhraní generovat a provádět statisíce až miliony testů, ukázalo se, že toto řešení nemusí být funkční z důvodů omezených prostředků v použitém prostředí pro průběžnou integraci. Komponenta `reporter` tak byla nakonec implementována tak, aby dokázala bezpečně převést testovací výsledky do lidsky přívětivé podoby.

Konvertor dokumentací jsem nakonec vynechal z důvodu zaměření se na konkrétní typ dokumentačního formátu. Jelikož existuje

velké množství existujících konvertorů mezi dokumentačními formáty, není potřeba tento scénář pokrýt v rámci implementovaného řešení.



Obrázek 5.1: Diagram datových toků implementovaného nástroje

Na diagramu datových toků 5.1 můžeme vidět tok dat implementovaným nástrojem. Na tomto diagramu budu popisovat základní architekturu tohoto nástroje. Následně budu jednotlivé komponenty a úkony podrobněji rozvádět v dalších sekcích této kapitoly.

Na vstup `parseru` přivedeme dokumentaci ve formátu OpenAPI 2 nebo OpenAPI 3. Ta je následně `parserem` převedena na seznam koncových bodů, což jsou jednotlivé URI adresy testovaného aplikačního webového rozhraní. Každý takový koncový bod může mít několik konkrétních HTTP metod a pro každou z těchto metod může existovat více ukázkových dotazů i odpovědí s konkrétními daty a jejich strukturou. Samotné URI adresy mohou mít navíc proměnné, u nichž máme definován datový typ a případně ukázkové hodnoty. Specifikace koncových bodů je uložena v serializačním formátu JSON.

Specifikace koncových bodů je následně načtena komponentou `fuzzer`. Tato komponenta vytvoří na základě specifikace koncových bodů testovací sadu a následně začne provádět samotné testování. Komponenta `fuzzer` produkuje dva důležité výstupy. Je jím report

z provedené testovací sady ve formátu JUnit a také kompletní textový log sloužící k dohledání komplexních situací, ke kterým mohlo při testování dojít.

Testovací report ve formátu JUnit je pak ještě zpracován komponentou `reporter`, která pomocí jednoduchých XSL transformací[1] vygeneruje uživatelsky přívětivý report ve formátu jedné HTML stránky.

5.2 Získávání dat z dokumentace

K získávání dat z dokumentace slouží komponenta `parser`. V následující sekci jsou obsaženy informace o výběru dokumentačního formátu a o implementaci této komponenty.

5.2.1 Konverze mezi formáty

Pro každý výše zmíněný formát pro dokumentaci webových aplikačních rozhraní existuje řada nástrojů pro konverzi tohoto formátu do několika jiných. Je tedy možné si na oficiálních stránkách vždy dohledat patřičný konvertor a použít jej.

Další možností je využití univerzálního webového konvertoru API Transformer[2] od společnosti APIMatic. Tento nástroj nám umožňuje převádět mezi velkým množstvím dokumentačních formátů.

Služba API Transformer vydala za rok 2017 zprávu o statistikách převodů mezi jednotlivými formáty. Z těchto statistik vyplývá že mezi nejvíce konvertované formáty se řadí OpenAPI verze 2 a formát pro nástroj Postman, tedy takzvaný Postman Collection[3]. [38]

5.2.2 Výběr vstupního formátu

Jelikož je relativně snadné mezi jednotlivými formáty převádět, není výběr vstupního dokumentačního formátu pro nástroj na fuzzování tak kruciální. Zvolil jsem tedy formát OpenAPI verze 2 i verze 3, které budou fuzzovacím nástrojem akceptovány.

Formát OpenAPI se těší popularitě, jen do repozitáře se samotnou specifikací přispívá 144 vývojářů, nehledě na množství lidí, kteří při-

1. `https://en.wikipedia.org/wiki/XSLT`
2. `https://www.apimatic.io/transformer`
3. `https://www.getpostman.com/collection`

spívají k vývoji volně šiřitelných nástrojů, jenž s OpenAPI nějakým způsobem operují. OpenAPI Initiative[4], jak se jmenuje název komunity starající se o OpenAPI, je jedním z projektů The Linux Foundation[5], která se stará o zajímavé volně šiřitelné projekty[6]. Pod jejími křídly se udržují projekty jako Linux, GraphQL, IoTivity, Jenkins, Mocha a mnoho dalších.

5.2.3 Implementace parseru

`Parser` byl implementován v prostředí .NET Core verze 2.1. Tuto platformu jsem zvolil převážně kvůli existujícím knihovnám pro práci s OpenAPI formátem. Pro účely deserializace dokumentačního souboru do shluku objektů jsem využil NuGet[7] balíček `OpenAPI.NET`[8] šířeným pod MIT licencí[9]. Pro zpětnou serializaci načtených a zpracovaných informací v podobě seznamu koncových bodů do formátu JSON byl využit populární [39] NuGet balíček `Newtonsoft.Json`[10], šířený rovněž pod MIT licencí.

Komponenta `parser` načte vstupní dokumentaci a převede ji do interní struktury definované pomocí knihovny `OpenAPI.NET`. Následně jsou data vyextrahována do podoby interní struktury seznamu koncových bodů, která je následně pomocí knihovny `Newtonsoft.Json` serializována do formátu JSON pro další použití komponentou `fuzzer`.

5.3 Fuzzování dat

Pro provádění fuzz testů jsem použil knihovnu Boofuzz, která jako jedna z mála vyhovovala definovaným požadavkům. Konkrétně se jedná o požadavky na aktivitu komunity vyvíjející daný nástroj, možnost použití v prostředí průběžné integrace, volně šiřitelné licence a v neposlední řadě také nezávislost na konkrétní platformě či architektuře.

4. `https://www.openapis.org/`
5. `https://www.linuxfoundation.org/`
6. `https://www.linuxfoundation.org/projects/`
7. `https://docs.microsoft.com/en-us/nuget/what-is-nuget`
8. `https://github.com/microsoft/OpenAPI.NET`
9. `https://github.com/microsoft/OpenAPI.NET/blob/master/LICENSE`
10. `https://www.newtonsoft.com/json`

Jedná se o knihovnu pro jazyk Python. V průběhu psaní diplomové práce se intenzivně pracovalo na verzi pro Python 3, která byla dne 26.5.2019 vydána ve verzi knihovny 0.1.5[11].

5.3.1 Staticky předpřipravené vektory dat

Při analýze nástrojů pro fuzz testování síťových aplikací v teoretické části jsem zjistil, že drtivá většina nástrojů nepoužívá náhodné vektory jak bylo v původním blackbox fuzz testování definováno, nýbrž jde cestou whitebox přístupu, který využívá znalosti testované aplikace a volí takové řetězce dat, o kterých je dopředu empiricky zjištěno, že mohou poškodit daný typ aplikace. Toto testování je efektivnější a i proto jsem tento princip zvolil pro implementovaný nástroj. Jednotlivé zasílané řetězce tak nebudou generovány náhodně či pseudonáhodně, ale budou vybírány z předem předdefinované množiny, jako je tomu u jiných nástrojů pro fuzz testování síťových aplikací.

5.3.2 Zhodnocení vestavěných primitiv

Jak je popsáno v sekci 2.3.5, Boofuzz disponuje vlastními primitivy. Při hlubší analýze těchto primitiv jsem zjistil, že nejsou vhodná pro použití ve vyvíjeném nástroji. Primitivum, které se stará o modifikaci řetězců a je tedy tak z pohledu testování datových položek zasílaných v rámci HTTP dotazů důležité, nemohlo být nastaveno jako deterministické, což je vhodné pro denní testování. Využívá totiž pseudonáhodné vkládání nulových ASCII hodnot, jenž nelze vypnout či ovlivnit. Pokud chce konzument našeho nástroje denně testovat svoji aplikaci, je žádoucí aby testované scénáře byly pokaždé stejné a on tak měl přehled o tom, zda například s další novou verzí předchozí chybu úspěšně opravil. Dalším zjištěným problémem bylo, že testovací statické řetězce jsou zde často příliš obecné a často nejsou cílená či vhodná pro webové aplikační rozhraní. Dalším nedostatkem byla absence SQL injekcí, které bych rád do mutací řetězců zahrnul. Posledním, neméně důležitým aspektem pak bylo generování tzv. dlouhých řetězců, které mohou způsobit přetečení bufferů[12]. Těchto řetězců bylo generováno

11. `https://github.com/jtpereyda/boofuzz/releases/tag/v0.1.5`
12. `http://www.cis.syr.edu/~wedu/seed/Book/book_sample_buffer.pdf`

příliš, s různým obsahem a významným způsobem navyšovaly počet testů, čímž se testování složitějších dotazů stalo neakceptovatelně pomalým. Implementace těchto primitiv navíc není ani jednoduše rozšiřitelná či modifikovatelná. Z těchto důvodů jsem se rozhodl napsat si primitiva vlastní.

5.3.3 Vlastní primitiva

Mnou navržená primitiva jsou deterministická, tedy v každém běhu programu generují totožné mutace, jako v bězích předcházejících. Vkládané řetězce použité při tvorbě primitiv jsou typické zranitelnosti, které se používají při penetračním testování webových rozhraní. Tyto zranitelnosti pak mohou být uživateli nástroje jednoduše rozšířena pomocí definování vlastních souborů s novými řetězci. Více o formě takového souboru je možné nalézt v kořenové složce nástroje v souboru `README.md`.

Předem definované zranitelnosti můžeme řadit do následujících kategorií:

- **Numerické řetězce** - řetězce obsahující číslice a často používaná rezervovaná slova pro vyjádření číselných hodnot. Jedná se často o sekvence, které by mohly způsobit tzv. přetečení bufferu[13] či použití nestandardních formátů pro zápis číselných hodnot.

- **Command injection** - tato kategorie obsahuje řetězce, které by mohly být interpretovány v některém ze shellů[14]. Příkazy, které se v těchto řetězcích nacházejí, se pokouší uspat vykonávaný proces tak, aby jejich skutečné provedení bylo možné testujícím nástrojem detekovat.

- **Path traversal** - často nazývaná také jako zranitelnost procházením adresářů. Tato kategorie obsahuje řetězce, které mohou vést k souboru, ke kterému aplikace nemá přístup a tím se dostane do potenciálně neošetřených potíží. Je využito jak relativních, tak absolutních cest.

13. `https://en.wikipedia.org/wiki/Buffer_overflow`
14. `https://en.wikipedia.org/wiki/Shell_(computing)`

- **Speciální znaky** - zde se nacházejí řetězce s nejrůznějšími speciálními znaky. Jedná se kupříkladu o netisknutelné znaky, bytové reprezentace nulových znaků a mnoho dalších.

- **SQL Injection** - řetězce, které se snaží útočit na aplikaci pomocí vkládání SQL příkazů. Nachází se zde obecné SQL injection, ale i útoky specifické a směrované na konkrétní databázové systémy.

- **Unicode sekvence** - platné i neplatné znaky reprezentované v unicode[15] kódování mohou také negativně ovlivnit testovanou aplikaci a způsobit její pád. Řetězce obsahují nestandardní sekvence znaků, kupříkladu řetězce psané zprava doleva či jednotlivá písmena obrácená vzhůru nohama.

- **XML, XPath** - řetězce obsahující XML a XPath[16] zranitelnosti.

Obecně existuje celá řada dalších zranitelností, jakými jsou například XSS[17] a jiné polygloty. Je třeba se ovšem zamyslet nad tím, že data bude zpracovávat webový server, který pravděpodobně nebude například klientský JavaScript vykonávat. Tedy spousta typů zranitelností byla z testování vyřazena z důvodu efektivity testování.

Nápady a náměty na konkrétní podobu řetězců se zranitelnostmi jsem čerpal z volně dostupných zdrojů. Existuje spousta veřejných projektů a repozitářů se seznamy takovýchto řetězců. Přímo pro problematiku fuzz testování existuje veřejný repozitář FuzzDB [40], který se snaží vybrat zranitelnosti vhodné pro fuzz testování z jiných, veřejně dostupných, seznamů zranitelností. Pro textové řetězce například čerpá z projektu BLNS[18] (*big-list-of-naughty-strings*). FuzzDB projekt jsem použil jako zdroj příkladů útoků a zranitelností, pro které jsem následně vyhodnocoval jejich vhodnost při testování webových aplikačních rozhraní. Například u SQL injekcí bylo nutné upravit jejich konkrétní tvar tak, aby jejich případné provedení bylo možné detekovat. Proto se zde využívá například uspání databáze k odložení

15. `https://en.wikipedia.org/wiki/Unicode`
16. `https://en.wikipedia.org/wiki/XPath`
17. `https://en.wikipedia.org/wiki/Cross-site_scripting`
18. `https://github.com/minimaxir/big-list-of-naughty-strings`

provedení dotazu na takovou dobu, aby vypršel časový limit pro odpověď a bylo tak možné detekovat úspěšné provedení útoku.

Nová primitiva jsou implementována použitím vestavěného primitiva `s_group`, které obstarává samotné mutování statických řetězců při generování jednotlivých požadavků skrze knihovnu Boofuzz.

Aktuálně jsou implementována tři nová primitiva, konkrétně:

- `s_http_string` - použitelné ve všech částech požadavku, kde je očekávána libovolná forma textu.
- `s_http_number` - použitelné ve všech částech požadavku, kde je očekáváno číslo. Například při fuzz testování JSON datových struktur je toto primitivum použito pro numerické primitivní datové typy této datové struktury.
- `s_http_boolean` - použitelné ve všech částech požadavku, kde je očekávána booleovská hodnota. Typickým příkladem je opět primitivní datový typ boolean v datové struktuře JSON.

Primitivum `s_http_string` obsahuje veškeré výše vypsané typy zranitelností, kdežto primitiva `s_http_number` a `s_http_boolean` pouze jejich podmnožiny. Mapování konkrétních kategorií zranitelností na konkrétní implementovaná primitiva lze dokonce změnit v konfiguračním souboru, popsaném v souboru `README.md` v kořenové složce nástroje.

5.3.4 Generování JSON dat

Jedním z problémů, který bylo potřeba vyřešit, je generování datových částí požadavků v serializačním formátu JSON. Datové struktury, které se vyskytují v samotných dotazech na webové aplikační rozhraní, jsou v OpenAPI dokumentaci popsány pomocí formátu JSON Schema[19] verze 4 či 5, dle verze OpenAPI dokumentace. [41] Tato schémata mohou být poměrně komplexní a obsahovat spoustu referencí na schémata jiná. Toto schéma je tedy zapotřebí celé projít, včetně rekurzivního průchodu všech referencí, a vygenerovat z něj konkrétní příklad ve formátu JSON odpovídající specifikovanému schématu. Tento proces jsem rozdělil na dvě části.

19. `https://json-schema.org/`

Nejprve v komponentě `parser`, ve třídě `SchemaParser`, rekurzivně procházím ve schématu všechny atributy popisovaného JSON objektu a tvořím topologickou strukturu totožnou s výsledným JSON příkladem. Tato struktura obsahuje pro každý atribut každého objektu jeho datový typ, jméno, formát či případně konkrétní příklad, pokud je definován. Následně je struktura serializována komponentou `parser` a je součástí definice cílových bodů.

Komponenta `fuzzer` následně při generování konkrétních definic požadavků předanou strukturu použije a vygeneruje podle ní konkrétní JSON tak, že prochází jednotlivé popisy atributů objektů a generuje z nich konkrétní JSON datovou strukturu. Tuto funkcionalitu je možné nalézt v modulu `json_schema_parser`. Pokud pro nějakou primitivní JSON hodnotu nebyl v dokumentaci uveden konkrétní příklad, je vložena jedna z výchozích definovaných hodnot dle datového typu primitiva. Pro číselná primitiva se jedná o hodnotu 0, pro řetězce se jedná o hodnotu `string`, apod. Důležité je, aby tento vygenerovaný příklad odpovídal datovému typu. Na konkrétní hodnotě už tolik nezáleží.

Ve zkratce by se tedy dalo shrnout, že `parser` nejprve projde a rekurzivně zjistí všechny reference daného schématu a vybere pouze důležitá data. Komponenta `fuzzer` následně již dle této redukované a ucelené struktury vygeneruje konkrétní příklad, který je následně mutován a posílán v jednotlivých požadavcích na testovanou aplikaci.

5.3.5 Fuzzované části dotazů

Fuzzování HTTP dotazů jsem rozdělil do třech samostatných částí, které jsou děleny jak na úrovni zdrojového kódu, tak ve vygenerovaných reportech. Jedná se o následující kategorie:

- Testování samotných HTTP hlaviček jednotlivých dotazů.
- Testování URI proměnných.
- Testování datové části HTTP dotazů.

Testování datových částí HTTP požadavků se pak ještě dělí na dvě menší kategorie, které budou blíže popsány níže.

Pojďme si nyní fuzzování HTTP hlavičky ukázat na následujícím příkladu:

```
GET /path HTTP/1.1 \r\n
Host: 127.0.0.1 \r\n
Content-Length: 0\r\n
User-Agent: WFuzz
\r\n\r\n
```

Červeně obarvené části dotazů označují fuzzování řetězců a oranžově zbarvené části pak označují fuzzování oddělovačů. Fuzzer tedy nahrazuje postupně jednotlivá červená pole mutacemi definovaných v příslušných primitivech pro fuzzování řetězců tak, že zafixuje všechny části požadavku na validní hodnoty a mutuje pouze jednu část, postupně všemi řetězci, které jsou v rámci daného primitiva specifikované. Pro oranžová pole pak používá primitivum `s_delim`, která bylo popsáno v sekci 2.3.5.

Fuzzování URI atributů demostruje následující příklad:

```
POST /et/v1/1/jobs/?param=value HTTP/1.1\r\n
Content-Type: text/plain\r\n
Host: 127.0.0.1 \r\n
Content-Length: 4\r\n
\r\n\r\n
data
```

URI jednotlivých dotazů mohou obsahovat parametry. Tyto parametry jsou detailně popsány v dokumentačních formátech včetně datových typů a ukázkových příkladů. Proto je možné snadno tyto parametry fuzzovat pomocí primitiv pro fuzzování řetězců a testovat tak validaci a parsování parametrů na straně serveru. V příkladu výše pak můžeme vidět, že fuzzována je pouze hodnota atributu `param`.

Fuzzování datových částí se liší podle přenášených informací. Pokud je v datové části přenášen typ `text/plain` či jakýkoliv typ jiný nežli `application/json`, je celý obsah datové části dotazu fuzzován jako celek a mutace se tedy provádí nad celou datovou částí. Pokud je však v datové části obsažen serializační formát, například JSON, je tento typ fuzzování velice neefektivní. Důvod je prostý, při modifikaci

celé datové části je velká pravděpodobnost, že nezůstane zachována správná syntaxe serializačního jazyka a server pak dotaz zamítne ihned, jakmile se nepovede korektně načíst datovou strukturu z datové části zaslaného požadavku. Proto je vhodné fuzzovat jen jednotlivé primitivní datové typy či klíče, které jsou v rámci serializačních formátů přenášeny. Tím efektivně otestujeme zpracování jednotlivých datových položek v rámci serializačního formátu.

```
POST /et/v1/1/jobs/?param=value HTTP/1.1\r\n
Content-Type: application/json\r\n
Host: 127.0.0.1 \r\n
Content-Length: 4\r\n
{"name": "John", "age": 30}
```

Jak můžeme vidět na příkladu výše, fuzzovány jsou pouze primitivní datové typy, v tomto případě jeden řetězec a jedno číslo. Testujeme tedy, zda testovaná aplikace nezhavaruje, pokud ji bude v nějakém atributu předána nečekaná hodnota, například nějaké vkládání příkazů či SQL injekcí.

Jak bylo již zmíněno výše, fuzzování datových části se ještě dělí na dvě menší kategorie. První z nich byla právě popsána. Druhá forma testování datových částí přináší menší obměnu, kdy za všechny primitivní datové typy jazyka JSON, které nejsou řetězci, vkládáme zranitelnosti v uvozovkách. Děláme to proto, že většina zranitelností je nějakou formou řetězce. Tedy pro zvýšení šance korektního formátu datové struktury JSON tyto zranitelnosti zabalíme do uvozovek, díky nimž se pak pro testovaný server tváří jako řetězce, i když byl na původním místě dle vstupní dokumentace očekáván jiný primitivní datový typ.

5.4 Zasílání dotazů a validace odpovědí

V rámci této sekce popisuji, jakým způsobem testovací nástroj komunikuje s testovanou aplikací. Dále zde rozebírám, jakým způsobem jsou validovány odpovědi ze serveru a jak je vyhodnocena jejich správnost.

5.4.1 Komunikace s testovanou aplikací

Navazování komunikace s testovaným serverem realizuje knihovna Boofuzz, konkrétně třída `SocketConnection`. Jak jsem již uvedl v sekci 2.3.3, pomocí třídy `SocketConnection` je možné komunikovat pod několika různými protokoly. Jelikož náš nástroj má za úkol testovat webová aplikační rozhraní, byl pro komunikaci použit protokol transportní vrstvy TCP. Vyvíjený nástroj umožňuje také testovat šifrované spojení pomocí SSL, které je taktéž realizováno předdefinovanou třídou `SocketConnection`.

Třída `SocketConnection` ke své funkci využívá unixové aplikační rozhraní *Berkeley sockets*[20]. *Berkeley sockets* je implementováno všemi moderními operačními systémy a je psáno v programovacím jazyce C. Jazyky vyšší úrovně, jakým je například jazyk Python, pak obvykle zabalují tuto funkcionalitu do takzvaných wrapperů. Ve standardní knihovně jazyka Python se jedná o modul `socket`.

Pro vytvoření socketu pro TCP komunikaci je tedy potřeba vytvořit socket, naplnit jej inicializačními informacemi a navázat TCP spojení. Tuto funkcionalitu obsahuje metoda `open` v třídě `SocketConnection`. V rámci mé práce jsem z knihovny Boofuzz odstranil chybu[21], kdy docházelo k špatnému odchycení a ošetření chybového stavu, které vyústilo v situaci, kdy při nenavázání spojení z důvodu vypršení časového limitu nebyl tento incident korektně reportován do patřičného testu.

Pro každý testovací scénář je tímto způsobem vytvořen socket a navázáno spojení. Tímto docílíme izolovanosti jednotlivých testů. Následně již stačí pomocí vytvořeného socketu zaslat vygenerovanou mutaci požadavku a čekat na případnou reakci serveru.

Jelikož testování i poměrně malých aplikačních rozhraní s několika málo cílovými body může vyústit ve velké množství testovaných scénářů, může se nám stát, že vyčerpáme množinu socketů, které nám jádro operačního systému povolí otevřít. Stává se tak především kvůli stavu `TIME_WAIT` [42], do kterého se socket dostane po aktivním uzavření. V tomto stavu musí setrvat po dobu dvou MSL (*Maximum Segment Lifetime*). Tato hodnota se na různých operačních systémech liší. Dle RFC 739 [42] je vhodné použít pro MSL hodnotu 2 minut.

20. `https://en.wikipedia.org/wiki/Berkeley_sockets`
21. `https://github.com/jtpereyda/boofuzz/pull/288`

Microsoft se této hodnoty ve verzi pro Windows Server 2000 držel a nastavil tak konstantu `TIME_WAIT` na 4 minuty, tedy dvakrát 2 minuty, jak doporučuje RFC. [43] Ostatní verze operačního systému Windows a všechny mnou testované unixové distribuce mají tuto hodnotu nižší. Testování jsem prováděl na operačních systémech Ubuntu 19.04, Fedora 28 a Debian 9.9.

Boofuzz situaci, kdy není možné otevřít socket, dříve neošetřoval a proto se mohlo stát, že při velmi rychlém provádění testovacích scénářů nástroj spadl z důvodu nemožnosti vytvoření socketu pro komunikaci. Proto jsem situaci nahlásil[22], analyzoval[23] a navrhl řešení[24], které v této situaci čeká po maximální dobu 4 minut, ve které průběžně po 5 sekundách zkouší, zda již je nějaký socket volný. Tím nedojde k pádu testujícího nástroje, pouze k odložení testů do doby, než bude možné otevřít socket.

5.4.2 Vyhodnocení úspěšnosti testovacího scénáře

Po zaslání požadavku následuje validace několika scénářů, které mohou nastat. Aktuálně jsou sledovány následující stavy:

- Přijde odpověď se stavovým kódem 500 nebo vyšším.
- Přijatá odpověď není validní.
  - Zpráva neodpovídá HTTP standardu.
  - Datový obsah zprávy neodpovídá udávanému typu obsahu v hlavičce HTTP zprávy.
- Server po přijetí zprávy uzavře TCP spojení.
- Vyprší časový limit pro obdržení odpovědi.

Ve všech výše vypsaných stavech se jedná o chybu, která je patřičně zaznamenána.

---

22. `https://github.com/jtpereyda/boofuzz/issues/296`
23. `https://github.com/jtpereyda/boofuzz/issues/296#issuecomment-548774369`
24. `https://github.com/jtpereyda/boofuzz/pull/342`

5.5 Generování testových reportů

Strategicky důležitým prvkem celého nástroje je i generování reportů o testování umožňující předání relevantní zpětné vazby, která povede k rychlému pochopení chyby a její případné nápravě. Proto bylo důležité nalézt vhodný formát pro podání zpětné vazby tak, aby byla pro uživatele snadno čitelná i v prostředí průběžné integrace nebo alespoň, aby bylo triviálně snadné ji do snadno čitelné podoby převést.

5.5.1 Volba formátu pro reportování testovacích scénářů

Při hledání formátu pro reporty testovacích scénářů pro mě důležitým kritériem bylo, zda jej prostředí pro průběžnou integraci dokáží sama, případně s podporou zásuvných modulů, vizualizovat a vyhodnocovat. Jedním z formátů splňující toto kritérium je JUnit[25]. Například prostředí pro průběžnou integraci Bamboo disponuje zásůvným modulem[26], který je schopen takovýto formát zpracovat a zobrazit v uživatelsky přívětivé podobě. Prostředí Jenkins má pro podobné účely také k dispozici zásuvný modul[27].

Existují i jiné formáty pro reportování testovacích scénářů, pro které by mohly existovat zásuvné moduly do prostředí průběžné integrace, nicméně JUnit formát je jednoduchý, přímočarý a nepřináší žádné znevýhodňující atributy.

5.5.2 Implementace generování JUnit výsledků

Jelikož knihovna Boofuzz, použitá pro samotné spouštění testů, disponuje rozhraním pro implementaci vlastní logovací třídy, byla implementace poměrně snadná. Využil jsem knihovny `junit-xml`[28], která slouží ke generování testovacích případů pro jazyk Python. Následně jsem pomocí této knihovny implementoval rozhraní `IFuzzLogger` a instanci této třídy předal do konstruktoru třídy `Session`, popsané v kapitole 2.3.3.

25. `https://junit.org/`
26. `https://confluence.atlassian.com/bamboo/junit-parser-289277056.html`
27. `https://wiki.jenkins.io/display/JENKINS/JUnit+Plugin`
28. `https://pypi.org/project/junit-xml/`

5.5.3 Vizualizace reportů

Jak již bylo zmíněno v úvodu kapitoly, některá prostředí pro průběžnou integraci obsahují parsery a vizualizéry JUnit formátu a je tedy možné vidět výsledky testů v lidsky přívětivé podobě přímo v nativním prostředí průběžné integrace.

Bohužel, ne vždy tyto parsery dokáží zpracovat tak velké množství dat, jaké je implementovaný nástroj schopen vygenerovat. Proto jsem implementoval komponentu `reporter`, která vygeneruje report ve formátu HTML. Jelikož je JUnit formát serializovaný do formátu XML, lze využít jednoduchých XSL tranformací k jeho převodu do podoby HTML. V rámci implementovaného řešení je tedy možné nalézt XSL soubor popisující transofrmaci dvou zmíněných formátů. Následně je využit existující modul jazyka Java SAXON[29], který samotnou XSL transformaci provede.

5.6 Vstupy implementovaného nástroje

Nástroj potřebuje pro svou funkci dva následující vstupní soubory:

- Dokumentaci aplikačního rozhraní.
- Konfiguraci serializovanou do formátu JSON.

Dokumentace aplikačního rozhraní musí být ve formátech OpenAPI 2 či OpenAPI 3 a musí obsahovat veškeré datové struktury, které jsou v rámci jednotlivých koncových bodů používány. Výhodou je, pokud tyto datové struktury mají uveden příklad takové struktury. Díky tomu se nemusí generovat neurčité statické řetězce a čísla, ale mohou být použity hodnoty z příkladů, které mohou mít pro daný kontext větší význam.

Konfigurační soubor obsahuje řadu nastavení pro implementovaný nástroj. Nejdůležitějším prvkem konfigurace je položka `target`, která obsahuje doménové jméno či IP adresu testovaného zařízení, číslo portu, na kterém běží testovaná aplikace a následně volbu, zda má být využito testování přes šifrovanou komunikaci pomocí SSL. Zbytek konfigurace je popsán v souboru `README.md` v kořenové složce adresáře nástroje. V kořenové složce je také ukázkový konfigurační soubor,

29. `http://saxon.sourceforge.net/`

pro uvedení příkladu konkrétní konfigurace. Nachází se v souboru `config_example.json`.

5.7 Zapouzdření do Docker obrazu

Docker[30] je aktuálně jedním z nejpoužívanějších nástrojů pro virtualizaci aplikací. Tato platforma je velmi často užívána v prostředích průběžné integrace, proto jsem se rozhodl vytvořit Docker obraz zapouzdřující vyvíjený nástroj pospolu s prostředím potřebným pro jeho běh.

5.7.1 Tvorba Docker obrazu

V kořenové složce repozitáře je k naleznutí soubor `Dockerfile`, který obsahuje popis tvorby obrazu zapouzdřujícího implementované řešení. Nejprve je vybrán obraz, na který se budou nalepovat další vzniklé vrstvy. Tím je obraz s .NET Core závislostmi, jelikož je to jedním ze základních požadavků pro komponentu zpracovávající vstupní dokumentaci. Dále následuje vystavení portu 26002, aby následně vzniklý kontejner mapoval port hostujícího systému na stejné číslo portu vzniklého kontejneru. Tento port je následně využíván monitorem procesů. Číslo portu, po kterém jsou zasílány samotné dotazy na webové aplikační rozhraní je takzvaně otevřeno až při spouštění samotného kontejneru z již existujícího obrazu. Poté následuje instalace dalších závislostí, jakým je například Python 3.7, ale také OpenJDK, která je potřeba pro běh komponenty provádějící XSLT transformace. V neposlední řadě se již jen do obrazu překopírují skripty a jiné soubory implementovaného nástroje.

Jelikož jsem nakonfiguroval automatické vytváření obrazu na platformě DockerHub[31], které čerpá z aktuální verze implementovaného nástroje ve veřejném GitHub repozitáři[32], není tak třeba nástroj lokálně sestavovat a je možné použít již sestavený obraz z platformy DockerHub.

---

30. `https://www.docker.com/`
31. `https://cloud.docker.com/u/starek4/repository/docker/starek4/wfuzz`
32. `https://github.com/ysoftdevs/wfuzz`

5.7.2 Spuštění a běh Docker kontejneru

Vstupem pro běh kontejneru jsou soubory, a ty musíme kontejneru nějakým způsobem předat. K tomuto účelu můžeme využít Docker Volumes[33], kdy jsme schopni připojit složku z běžícího systému do souborového systému kontejneru. Poté může nástroj přistoupit k souborům, které jsou obsaženy v připojeném adresáři. Soubory, námi považované za výstupní, pak můžeme taktéž generovat do připojené složky v rámci běhu kontejneru a hostující systém k nim má přístup i po skončení běhu daného kontejneru. Tímto způsobem jsme schopni předat kontejneru potřebné soubory a získat výsledky z provedené testovací sady.

Více o spuštění samotného nástroje pomocí Docker kontejneru naleznete v příloze B.4.

5.8 Implementace systémových testů

Nástroj je také doručován s několika sadami testů. Jedná se o jednotkové testy pro komponenty `fuzzer` a `parser` a také sadou systémových (*end-to-end*) testů celého nástroje. Jednotkové testy pro komponentu `reporter` postrádají význam, jelikož komponenta pouze pouští zapouzdřený Java modul SAXON.

Systémové testy tedy pracují s předem vytvořenou referenční dokumentací a testují reálný webový server, který je v některých případech doplněn o in-memory databázi. Tento testovací webový server je součástí implementace systémových testů. Pojďme se nyní podívat na konkrétní testovací scénáře.

5.8.1 Detekce zranitelnosti serveru pomocí SQL injekcí

Testovaný server disponuje jednoduchou SQLite[34] in-memory databází. Zasílané URI atributy jsou vkládány do SQL dotazu prováděném vůči in-memory databázi bez jakéhokoliv ošetření či validace. Primitivum pro testování řetězců obsahuje, mimo jiných, i mutaci, která se pomocí SQL injekcí snaží uspat databázi. Jakmile dojde ke zpracovávání této mutace, SQL příkaz čeká, až skončí uspání databáze

33. `https://docs.docker.com/storage/volumes/`
34. `https://www.sqlite.org/index.html`

a stejně tak s ním i webový server, jenž tento SQL příkaz inicioval. Touto cestou dojde k vypršení časového limitu na odpověď a test je reportován jako neúspěšný.

5.8.2 Detekce zranitelnosti serveru pomocí vkládání příkazů

Velmi podobné výše uvedenému testování detekce zranitelnosti v podobě SQL injekcí. Server již nedisponuje SQL databází, ale používá hodnoty atributů pro jejich interpretaci ve výchozím shellu. Jelikož primitivum pro testování řetězců obsahuje i pokus o uspání skrze vkládání příkazů, dojde zde podobně jako v předchozím příkladu čekání na dokončení tohoto příkazu a opět vyprší časový limit a test je označen jako neúspěšný.

5.8.3 Detekce neexistující testované aplikace

Neexistující testovaná aplikace je jinou situací, než situace popsána v předchozím příkladě, kdy došlo k vypršení časového intervalu pro zaslání odpovědi. Implementovaný nástroj totiž nemůže vůbec navázat TCP spojení na daném portu s testovanou aplikací. Tento scénář je v systémových testech také zahrnut.

5.8.4 Detekce špatného formátu odpovědi

V tomto případě se testují dva odlišné scénáře. Jedním z nich je situace, kdy webový server vrátí odpověď, která neodpovídá HTTP specifikaci. Testuje se pak, zda tuto situaci nástroj správně detekuje a patřičně na ni zareaguje.

Druhým scénářem je, že server sice vrátí odpověď validní dle HTTP specifikace, ovšem formát datové části nesouhlasí s avizovaným typem obsahu v hlavičce. Konkrétně se zde testuje situace, při které je v hlavičce uveden typ obsahu `application/json` (což odpovídá serializačnímu formátu JSON), ale v odpovědi přijde neformátovaný text, který odpovídá obsahu `text/plain`. Opět se testuje, zda implementovaný nástroj tuto chybu odhalí.

5.8.5 Detekce selhání obsluhy požadavku

Při zpracovávání požadavku může dojít k nespecifikované chybě, na kterou by měl webový server zareagovat zasláním odpovědi se stavovým kódem 500 nebo vyšším, dle okolností, za jakých k chybě došlo. Webový server tedy v tomto scénáři vrací stavový kód 500 a následně je vyhodnoceno, zda implementovaný nástroj tuto situaci detekuje.

6 Zhodnocení řešení

V této sekci hodnotím implementovaný nástroj na základě jeho používání jak ve společnosti Y Soft, tak na volně šiřitelném projektu.

6.1 Modelové použití na volně šiřitelném projektu

Pro otestování vyvíjeného nástroje jsem si vybral, kromě komponent společnosti Y Soft, i volně šiřitelný projekt Harbor. Harbor je služba pro správu Docker obrazů v zabezpečeném prostředí. Umožňuje uchovávat, podepisovat a skenovat Docker obrazy pro zranitelnosti. Na tomto open-source projektu spolupracovalo již přes 150 lidí, obsahuje více než 8 000 inkrementů a používá jej například i pojišťovna Axa[1]. [44] [45]

Harbor je on-premise[2] služba, kterou si může kdokoliv nainstalovat. V rámci této služby je dostupné i webové aplikační rozhraní pro správu Docker obrazů. Toto aplikační rozhraní je popsáno pomocí formátu OpenAPI 2. Aplikační rozhraní navíc konzumuje datové položky dotazů ve formátu JSON, je tedy ideálním cílem pro vyvíjený nástroj.

Službu Harbor jsem tedy pomocí nástroje otestoval. Ze vstupní dokumentace bylo vygenerováno 866 836 testovacích scénářů. Testovaná aplikace běžela v lokálním virtuálním prostředí, nebylo zde tedy téměř žádné spoždění způsobené přenosem dat. Generování testovacích scénářů je otázka několika sekund. Celé testování zabralo bezmála 2 hodiny.

První zajímavou zjištěnou skutečností bylo chování samotného webového serveru, na kterém API běží. Jedná se o server nginx[3]. Při fuzzování samotného HTTP dotazu, jehož výsledkem může být i dle specifikace nevalidní dotaz, webový server odpověděl taktéž nevalidní HTTP odpovědí. Místo HTTP hlavičky a těla server odpoví pouze textový ASCII řetězec s popisem chyby, bez jakékoliv informace o zaslaném obsahu či jiných metadatech, které jsou běžnou součástí HTTP hlavičky.

---

1. `https://www.axa.com/`
2. `https://en.wikipedia.org/wiki/On-premises_software`
3. `http://nginx.org/`

Při testování URI atributů aplikačního rozhraní produktu Harbor bylo nalezeno několik chyb. Cílový bod `/api/projects` obsahuje, mimo jiných, i volitelný atribut `page`, který se stará o stránkování výsledků. Datovým typem tohoto atributu je dle dokumentace číslo. Formát čísla je nastaven na hodnotu `int32`, která dle OpenAPI 2 specifikace [41] znamená znaménkové 32 bitové číslo. Z výsledků testování je patrné, že aplikační rozhraní akceptuje i daleko vyšší čísla. Aplikace akceptuje čísla až do hodnoty 2^{62}. Číslo $2^{62} + 1$ je první číslo, při kterém obsluha požadavku zhavaruje a je navrácen stavový kód 500. Tato situace se ze vzrůstající hodnotou atributu `page` opakuje až do hodnoty 2^{63}, což je první hodnota, pro kterou je již korektně navrácen stavový kód 400, který upozorňuje klienta, že neposílá validní požadavek. Podobná situace se pak opakovala i v jiných cílových bodech pro atributy označující celočíselný identifikátor nějaké entity, příkadem nechť je `user_id`. Dalším zjištěním bylo, že cílový bod `/api/repositories/{repo_name}/tags/{tag}/scan` vracel stavový kód 500 pro každý neexistující repozitář specifikovaný atributem `repo_name`.

Fuzzování datových položek serializačního formátu JSON také nedopadlo bez chyb. Některé URI parametry, které nejsou korektně validovány, se v určitých situacích používají i v datových částech dotazů. Pokud například chceme přes webové rozhraní aktualizovat uživatele, který v datové části dotazu obsahuje i svůj jednoznačný identifikátor `user_id`, setkáváme se při fuzzování této hodnoty se stejnou chybou, jako v případě URI atributů. Je tedy patrné, že je na straně implementace webového rozhraní použita stejná logika pro zpracování identifikátoru uživatele bez ohledu na to, zda je tento atribut předán v rámci URI atributu či je předán v datové části požadavku. Při testování datových položek bylo zjištěno ještě několik dalších chyb, které se týkaly opět validace dat. Příkladem nechť je položka `group_name`, která určuje uživatelskou skupinu. Pokud byla tato JSON položka nastavena na velmi dlouhý řetězec, aplikační server jej nedokázal zpracovat a byla opět navrácena odpověď se stavovým kódem 500, jako ve všech předchozích příkladech.

Všechny odhalené chyby v aplikaci Harbor se týkaly chybné validace vstupních dat a byly odhaleny pomocí navráceného stavového kódu 500. Ani v jednom případě nedošlo k nalezení nějaké závažné zranitelnosti. Na všechny dotazy server odpověděl a nedošlo k žádnému

pádu či restartu testované aplikace. Některé výše zmíněné příklady chyb jsem nahlásil[4] komunitě v oficiálním GitHub repozitáři.

6.2 Použití v prostředí společnosti Y Soft

Implementovaný nástroj je ve společnosti Y Soft součástí vývojového cyklu. Jelikož je takovéto testování časově náročné a může zabrat jednotky, často i desítky hodin, není spouštěno pro každý inkrement, nýbrž pouze jednou týdně.

Nástroj běží v prostředí průběžné integrace Bamboo, která byla popsána v teoretické části. Nástroj je zde schopen běžet jak na agentech používající operační systém Windows, tak na agentech, jenž používají Linux. Nástroj je také používán manuálně, mimo prostředí průběžné integrace, tam je pak často využívána varianta Docker obrazu, kterým lze testování spustit i bez předchozí instalace závislostí.

Nástroj již v implementaci testovaného webového aplikačního rozhraní detekoval několik validačních chyb, které jsem následně v testované aplikaci opravil. Menší chyby byly nalezeny také v dokumentaci. Zatím ovšem nebyla nalezena žádná bezpečnost kompromitující zranitelnost.

6.3 Náročnost na výpočetní zdroje

Testování náročnosti na výpočetní zdroje proběhlo na čisté instalaci systému Ubuntu 19.04. Nástroj byl spuštět standardně pomocí Bash skriptu, který je součástí nástroje. Sledování náročnosti probíhalo při testování projektu Harbor, při kterém bylo vytvořeno 866 836 testovacích scénářů.

Nástroj při zpracovávání dokumentace a vytváření všech testovacích scénářů naalokoval zhruba 200 MB paměti. Tato hodnota při provádění samotných testů pomalu roste, v našem případě rychlostí zhruba 1 MB za minutu, a po dokončení každé testované kategorie klesne zpět na 200 MB. Toto je způsobeno ukládáním výsledků pro generování JUnit testovacích reportů pro každou kategorii testů. Až po dokončení všech testovacích scénářů jedné kategorie, například

4. `https://github.com/goharbor/harbor/issues/9976`

testování URI atributů, jsou JUnit výsledky serializovány do souboru a paměť je tak uvolněna. Vzhledem k běžně dostupným kapacitám pamětí v prostředích průběžné integrace takovýto dočasný nárust nepovažuji za problematický. Nástroj neměl na žádných z používaných platforem průběžné integrace, jmenovitě Bamboo, Travis CI a CircleCI, s pamětí žádný problém i při velmi dlouhém testování.

Jelikož nástroj záměrně neumožňuje provádění více testovacích scénářů najednou, není zde žádna forma paralelismu. I kdyby na úrovni zdrojového kódu nějaká forma paralelizace byla, na situaci by to nic neměnilo, jelikož CPython používá GIL[5] mechanismus. Je tedy vytěžováno pouze jedno procesorové vlákno. Vytížení tohoto vlákna se odvíjí především od rychlosti zaslání odpovědi od testované aplikace. Nicméně i při testování velmi lehké implementace webového serveru, který běží lokálně pospolu s implementovaným nástrojem, je vytížení tohoto vlákna minimální. Konkrétně se jedná o jednotky procent. Testování bylo prováděno na procesoru i7-8850H[6] od společnosti Intel.

6.4 Problematika účinnosti fuzz testování

Při fuzz testování webových aplikačních rozhraní jsem se setkával často se scénářem, kdy webový server zamítl většinu požadavků z důvodu chybějící autentizace či kvůli neexistující entitě, identifikované některým parametrem URI adresy. V těchto situacích pak dochází k velké neefektivitě celého testování, jelikož aplikační logika serveru často vůbec nemusí zpracovávat části dotazu, které jsou modifikovány fuzzerem.

Tato situace byla zjištěna již při prvních prováděných testováních a bylo potřeba se s ní nějak vypořádat. Z těchto důvodu vznikly v konfiguračním JSON souboru, mimo jiné, dva klíče, které umožňují uživateli zadat statické a fuzzerem neměnné hlavičky a také zafixovat některé hodnoty parametrů, které taktéž nebudou fuzzerem modifikovány.

---

5. `https://en.wikipedia.org/wiki/Global_interpreter_lock`
6. `https://ark.intel.com/content/www/us/en/ark/products/134899/intel-core-i7-8850h-processor-9m-cache-up-to-4-30-ghz.html`

Jejich typické použití je právě pro zvýšení účinnosti prováděného testování. Příkladem pak mohou být často používané autentizační tokeny, které mohou sloužit k autentizaci uživatele k aplikačnímu rozhraní a bez jejichž uvedení jsou dotazy ihned odmítnuty. Před samotným testováním pak tedy stačí provést autentizaci, vygenerovat si autentizační token a ten vložit do konfiguračního souboru mezi ostatní statické hlavičky. Tímto zajistíme, že každý dotaz je autentizován a server jej ihned neodmítne. Zvýšíme tak pravděpodobnost, že jej některé z fuzzovaných částí dotazu přivedou k pádu či neočekávané reakci. Tuto možnost jsem uplatnil i při testování služby Harbor, která využívá standardní HTTP autentizační metodu. Před samotným testováním jsem si vygeneroval autentizační hlavičku, která je vložena do konfiguračního souboru nástroje a je tak odeslána v každém požadavku.

Statické URI atributy pak mohou být použity, pokud je celé aplikační rozhraní svázáno s konkrétní entitou, bez jejíž korektní přítomnosti bude vetšina požadavků zamítnuta. Identifikace takovéto entity je pak často předávána v parametru URI adresy. Tuto možnost jsem využil i při testování aplikačního rozhraní služby Harbor. Tato služba pracuje s takzvanými projekty. Každý projekt má svůj celočíselný identifikátor. Spousta cílových bodů aplikačního rozhraní obsahuje jako parametr právě tento celočíselný identifikátor, který určuje, jakého projektu se konkrétní požadavek týká. Proto jsem si ve službě Harbor vytvořil jeden projekt, kterému bylo automaticky přidělen identifikátor s hodnotou 1. Tento atribut s konkrétní hodnotou jsem pomocí konfiguračního souboru nástroje zafixoval tak, aby v dotazech byla použita tato konkrétní hodnota a zvýšila se tak účinnost testování. Testovaný server pak neodmítne většinu dotazů kvůli neexistujícímu projektu, ale dále vyhodnocuje předaná data, která mohou případně způsobit uplatnění nějaké zranitelnosti.

6.5 Návrhy na rozšíření

Implementovaný nástroj je aktuálně udržován na platformě GitHub, kde může kdokoliv přispět k vývoji, návrhům vylepšení nebo hlášení chyb. Z mé strany jsou však aktuálně v přípravě, či se již pracuje, na několika rozšířeních.

Jedním z nich je umožnění porovnávat starší testy stejného aplikačního rozhraní. Tímto můžeme sledovat jednotlivé chyby a jejich průběh v čase. Tedy jestli a kdy byla tato chyba odstraněna a jak dlouho se v produktu nacházela. Dále to umožní označit některé scénáře jako minoritní a v dalších prováděných testech je případně ignorovat.

Druhým plánovaným vylepšením je monitorování zdrojů systému, na kterém běží testované aplikační rozhraní. Tedy periodicky sbírat informace o stavu systému a případné větší výkyvy se snažit namapovat na konkrétní testované scénáře. Jedná se například o monitorování zátěže procesoru, využití virtuální paměti procesu, využití operační paměti a další vhodné čítače.

A v neposlední řadě bych rád rozšířil testovací nástroj o podporu detailního fuzzování i jiných serializačních formátů, než pouze JSON. Jmenovitě se jedná o XML a poslední dobou stále populárnější YAML.

7 Závěr

Cílem mé práce bylo zjistit aktuální možnosti pro automatizované fuzz testování síťových aplikací, specificky webových aplikačních rozhraní. Zmapovat nástroje, které by bylo možné použít pro automatizované fuzz testování takovýchto rozhraní, případně sloužících jako základ pro implementovaný nástroj.

V první části práce jsem uvedl princip fuzz testování, za kterým následuje popis některých vybraných zástupců nástrojů pro tento typ testování. Nechybí ani přehled a pojednání o nástrojích průběžné integrace a vymezení požadavků na vyvíjený nástroj. Následuje popis dokumentačních formátů, které jsou důležitým vstupem implementovaného nástroje. Pátá kapitola pojednává o implementaci výsledného nástroje a důležitých rozhodnutích, které bylo třeba při návrhu a vývoji učinit.

Implementovaný nástroj je úspěšně používán ve vývojovém cyklu společnosti Y Soft. Díky nástroji se povedlo odhalit několik validačních a dokumentačních chyb, které jsem v rámci spolupráce z aplikací a dokumentací odstranil. Obdobné chyby byly také nalezeny v open-source projektu Harbor, který čítá již přes 8 000 inkrementů a na jeho vývoji spolupracuje přes 150 uživatelů. [45] Tyto chyby byly předány komunitě, která se stará o její vývoj. V rámci práce jsem taktéž pomohl zdokonalit open-source projekt Boofuzz, jenž aktuálně čítá přes 140 000 stažení[1] a je implementovaným nástrojem používán. Opravil jsem několik chyb znemožňujících testování v určitých podmínkách a také jsem testoval a konzultoval další opravy či vylepšení této knihovny.

Jelikož se implementovaný nástroj ve společnosti Y Soft osvědčil, navrhl jsem nástroj vydat jako volně šiřitelný nástroj a začít jej spravovat ve veřejném prostoru, kterým je správce repozitářů GitHub včetně automatického generování docker obrazů pro službu Docker Hub. Nyní je tak nástroj volně dostupný pro širokou veřejnost a jeho použití je pomocí Docker kontejneru snadné.

1. statistiky jsem získal z vereřného datasetu platformy Google BigQuery

Bibliografie

1. *ClusterFuzz* [online] [cit. 2019-10-04]. Dostupné z: https://google.github.io/clusterfuzz/.
2. OEHLERT, P. Violating assumptions with fuzzing. *IEEE Security Privacy* [online]. 2005, roč. 3, č. 2, s. 58–62 [cit. 2019-03-02]. ISSN 1540-7993. Dostupné z DOI: 10.1109/MSP.2005.55.
3. MILLER, Barton P.; FREDRIKSEN, Louis; SO, Bryan. An Empirical Study of the Reliability of UNIX Utilities. *Commun. ACM* [online]. 1990, roč. 33, č. 12, s. 32–44 [cit. 2019-03-02]. ISSN 0001-0782. Dostupné z DOI: 10.1145/96267.96279.
4. KUDERSKI, Jakub. Automated Test-Case Generation: Fuzzing [online]. 2019, s. 3 [cit. 2019-03-02]. Dostupné z: https://ece.uwaterloo.ca/~agurfink/stqam/assets/pdf/W04-Fuzzing.pdf.
5. MILLER, Barton P. Fuzz Testing of Application Reliability [online]. 2008 [cit. 2019-03-03]. Dostupné z: http://pages.cs.wisc.edu/~bart/fuzz/.
6. MILLER, Barton P.; KOSKI, David; LEE, Cjin Pheow; MAGANTY, Vivekananda; MURTHY, Ravi; NATARAJAN, Ajitkumar; STEIDL, Jeff. Fuzz Revisited: A Re-examination of the Reliability of UNIX Utilities and Services [online]. 1995, s. 1–23 [cit. 2019-03-03]. Dostupné z: http://ftp.cs.wisc.edu/paradyn/technical_papers/fuzz-revisited.pdf.
7. FORRESTER, Justin E.; MILLER, Barton P. An Empirical Study of the Robustness of Windows NT Applications Using Random Testing. In: *Proceedings of the 4th Conference on USENIX Windows Systems Symposium - Volume 4* [online]. Seattle, Washington: USENIX Association, 2000 [cit. 2019-03-04]. WSS'00. Dostupné z: http://dl.acm.org/citation.cfm?id=1267102.1267108.
8. MILLER, Barton P.; COOKSEY, Gregory; MOORE, Fredrick. An Empirical Study of the Robustness of MacOS Applications Using Random Testing. In: *Proceedings of the 1st International Workshop on Random Testing* [online]. Portland, Maine: ACM, 2006 [cit.

2019-03-04]. RT '06. ISBN 1-59593-457-X. Dostupné z DOI: 10.1145/1145735.1145743.

9. GODEFROID, Patrice; LEVIN, Michael Y.; MOLNAR, David. Automated Whitebox Fuzz Testing. In: [online]. 2008 [cit. 2019-03-08]. Dostupné z: https://www.microsoft.com/en-us/research/publication/automated-whitebox-fuzz-testing/.
10. BOUNIMOVA, Ella; GODEFROID, Patrice; MOLNAR, David. Billions and Billions of Constraints: Whitebox Fuzz Testing in Production. In: *Proceedings of the 2013 International Conference on Software Engineering* [online]. San Francisco, CA, USA: IEEE Press, 2013, s. 122–131 [cit. 2019-03-08]. ICSE '13. ISBN 978-1-4673-3076-3. Dostupné z: http://dl.acm.org/citation.cfm?id=2486788.2486805.
11. *Burp Suite documentation* [online] [cit. 2019-03-24]. Dostupné z: https://portswigger.net/burp/documentation.
12. *OWASP ZAP* [online] [cit. 2019-03-24]. Dostupné z: https://github.com/zaproxy/zaproxy/wiki.
13. *OWASP ZAP* [online] [cit. 2019-03-24]. Dostupné z: https://github.com/zaproxy/zaproxy.
14. *boofuzz: Network Protocol Fuzzing for Humans* [online] [cit. 2019-03-25]. Dostupné z: https://github.com/jtpereyda/boofuzz.
15. *boofuzz* [online] [cit. 2019-03-25]. Dostupné z: https://boofuzz.readthedocs.io/en/latest/.
16. *OpenRCE/sulley: A pure-python fully automated and unattended fuzzing framework* [online] [cit. 2019-03-25]. Dostupné z: https://github.com/OpenRCE/sulley.
17. FOWLER, Martin; FOEMMEL, Matthew. Continuous integration. *ThoughtWorks* [online]. 2006, roč. 122, s. 14 [cit. 2019-04-06]. Dostupné z: https://martinfowler.com/articles/continuousIntegration.html.
18. NICOLAI, Johannes. *GitHub welcomes all CI tools* [online] [cit. 2019-04-06]. Dostupné z: https://github.blog/2017-11-07-github-welcomes-all-ci-tools/.

19. BELLER, Moritz; GOUSIOS, Georgios; ZAIDMAN, Andy. Oops, my tests broke the build: An explorative analysis of Travis CI with GitHub. In: *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)* [online]. 2017, s. 356–367 [cit. 2019-04-06].

20. HRONČOK, Miro. Travis CI [online]. 2015 [cit. 2019-04-06]. Dostupné z: `https://installfest.cz/if15/slides/hroncok_travis.pdf`.

21. DAVIS, Tony. What is a WebHook? *webhooks. pbworks. com, retrieved Mar* [online]. 2017, roč. 28 [cit. 2019-04-06]. Dostupné z: `https://webhooks.pbworks.com/w/page/13385124/FrontPage`.

22. *Travis CI User Documentation* [online] [cit. 2019-04-06]. Dostupné z: `https://docs.travis-ci.com/`.

23. *Travis CI - Test and Deploy Your Code with Confidence* [online] [cit. 2019-04-06]. Dostupné z: `https://travis-ci.org/`.

24. *Companies that use CircleCI & CircleCI Integrations | StackShare* [online] [cit. 2019-04-06]. Dostupné z: `https://stackshare.io/circleci`.

25. *CircleCI vs. GitLab | GitLab* [online] [cit. 2019-04-06]. Dostupné z: `https://about.gitlab.com/devops-tools/circle-ci-vs-gitlab.html`.

26. *Welcome to CircleCI Documentation* [online] [cit. 2019-04-06]. Dostupné z: `https://circleci.com/docs/`.

27. *Bamboo documentation - Atlassian Documentation* [online] [cit. 2019-04-08]. Dostupné z: `https://confluence.atlassian.com/bamboo`.

28. *Using Docker-in-Docker* [online] [cit. 2019-11-23]. Dostupné z: `https://jpetazzo.github.io/2015/09/03/do-not-use-docker-in-docker-for-ci/`.

29. *Jenkins User Documentation* [online] [cit. 2019-04-14]. Dostupné z: `https://jenkins.io/doc/`.

30. MAPLE, Simon; BINSTOCK, Andrew. JVM Ecosystem report 2018 - About your Tools [online]. 2018 [cit. 2019-04-14]. Dostupné z: https://snyk.io/blog/jvm-ecosystem-report-2018-tools/.

31. HERITAGE, Laura. API Description Languages [online]. 2014 [cit. 2019-05-03]. Dostupné z: https://www.slideshare.net/SOA_Software/api-description-languages.

32. HADLEY, Marc J. *Web Application Description Language (WADL)* [online]. Mountain View, CA, USA: Sun Microsystems, Inc., 2006 [cit. 2019-05-03]. Technická zpráva.

33. SANDOVAL, Kristopher. *Top Specification Formats for REST APIs* [online]. 2016 [cit. 2019-05-03]. Dostupné z: https://nordicapis.com/top-specification-formats-for-rest-apis/. Technická zpráva.

34. *Swagger Documentation | Swagger* [online] [cit. 2019-05-07]. Dostupné z: https://swagger.io/docs/.

35. *API Blueprint | API Blueprint* [online] [cit. 2019-05-08]. Dostupné z: https://apiblueprint.org/.

36. *api-blueprint/API Blueprint Specification.md at master · apiaryio/api-blueprint* [online] [cit. 2019-05-08]. Dostupné z: https://github.com/apiaryio/api-blueprint/blob/master/API%20Blueprint%20Specification.md.

37. *Projects | RAML* [online] [cit. 2019-11-20]. Dostupné z: https://raml.org/projects.

38. REHMAN, Faria. A Year with API Transformer [online]. 2018 [cit. 2019-05-08]. Dostupné z: https://blog.apimatic.io/a-year-with-api-transformer-c6f821b7bbf.

39. *Package Downloads for Newtonsoft.Json* [online] [cit. 2019-05-20]. Dostupné z: https://www.nuget.org/stats/packages/Newtonsoft.Json?groupby=Version.

40. *FuzzDB* [online] [cit. 2019-10-04]. Dostupné z: https://github.com/fuzzdb-project/fuzzdb.

41. *OpenAPI Specification - Data Types* [online] [cit. 2019-11-22]. Dostupné z: https://github.com/OAI/OpenAPI-Specification/blob/master/versions/2.0.md#data-types.

42. POSTEL, Jon. *Transmission Control Protocol* [Internet Requests for Comments]. RFC Editor, 1981 [cit. 2019-11-20]. ISSN 2070-1721. Dostupné z: https://tools.ietf.org/html/rfc793. STD. RFC Editor. https://tools.ietf.org/html/rfc793.

43. *TcpTimedWaitDelay | Microsoft Docs* [online] [cit. 2019-05-20]. Dostupné z: https://docs.microsoft.com/en-us/previous-versions/windows/it-pro/windows-2000-server/cc938217(v=technet.10).

44. *Harbor* [online] [cit. 2019-11-22]. Dostupné z: https://goharbor.io/.

45. *Harbor* [online] [cit. 2019-05-07]. Dostupné z: https://github.com/goharbor/harbor.

A Nástroje pro fuzz testování síťových aplikací

Tabulka A.1: Výběr z aktuální nabídky nástrojů pro fuzz testování síťových aplikací

Nástroj	**Licence**	**Autor**	**Aktivita**	**Platforma**	**CI ***
Peach API Security	Komerční	Peach Tech	-	Windows, Linux, Mac OS	Ano
Burp Suite Community	Volné použití	PortSwigger	-	Windows, Linux, Mac OS	Ne
Burp Suite Enterprise	Komerční	PortSwigger	-	-	Ano
OWASP ZAP	Apache 2.0	OWASP	Aktivní	Windows, Linux, Mac OS	Ne
Boofuzz	GNU GPL v2.0	-	Aktivní	Windows, Linux, Mac OS	Ano
sfuzz	Volné použití	Aaron Conole	Neaktivní	Windows, Linux	Ano
SPIKE	Volné použití	Dave Aitel	Neaktivní	Windows, Linux	Ano

* Použitelnost pro fuzzing v prostředí průběžní integrace.

B Manuál k výslednému nástroji

B.1 Konfigurace implementovaného nástroje

Nástroj je nutné konfigurovat pomocí konfiguračního souboru zapsaném v serializačním formátu JSON. Jeho ukázku je možné nalézt v kořenové složce nástroje v souboru `config_example.json`. Detailní popis jednotlivých konfiguračních položek je následně možné nalézt v souboru `README.md`, který je uložen v kořenovém adresáři.

B.2 Použití monitoru procesů

Monitor procesů byl popsán v kapitole 2.3.2. Jak již bylo uvedeno v teoretické části, monitor procesů potřebuje běžet na testovaném operačním systému a pro svůj korektní běh potřebuje některé závislosti. Jejich podrobnější popis a postup instalace je možné nalézt v dokumentaci [15].

Samotný Python skript obsahující monitor procesů pak stačí spustit. Typicky jej stačí spustit pouze s parametrem `-p`, následovaný jménem procesu, ve kterém běží námi testovaní aplikační rozhraní. Monitor procesů je pak schopen kontrolovat, zda je tento proces stále běžící a hlásit tento stav testujícímu zařízení.

B.3 Spuštění pomocí běžně dostupných shellů

Implementovaný nástroj potřebuje pro svůj běh následující závislosti:

- **Shell** - Powershell nebo Bash.
- **Python** - ve verzi 3.6 nebo vyšší.
- **.NET Core** - ve verzi 2.1.
- **JRE nebo Docker** - prostředí pro běh Java aplikací ve verzi 8 nebo vyšší, případně pokud není Java interpret k dispozici, stačí aplikace Docker schopná sestavit a spustit linuxový kontejner.

Pokud máme na testujícím systému nainstalované všechny výše vypsané závislosti, spuštění testování je pak snadné. Stačí spustit buď skript `run.sh` (pro Bash) nebo `run.ps1` (pro PowerShell). Požadované parametry a jejich pořadí lze opět nalézt v souboru `README.md`, případně pokud uživatel spustí daný skript bez parametrů, je navigován k jejich korektnímu užití.

B.4 Spuštění pomocí Docker kontejneru

Při spouštění Docker kontejneru je třeba správně nastavit složky pro připojení z aktuálního do kontejnerizovaného souborového systému. Zároveň je třeba takzvaně otevřít takové porty, které jsou nutné pro komunikaci testovaného nástroje s testovaným aplikačním rozhraním.

Validní příkaz pro spuštění kontejneru je taktéž možné nalézt v souboru `README.md`, kde je detailně popsán.

C Elektronické přílohy

Součástí práce je i samotný implementovaný nástroj, který je zabalen v ZIP[1] archivu s názvem `wapifuzz.zip`.

C.1 Struktura archivu s implementovaným nástrojem

Po rozbalení ZIP archivu dostaneme následující strukturu adresářů a důležitých souborů:

- **fuzzer** - zdrojové kódy komponenty `fuzzer`.
  - **payloads** - obsahuje složky s konkrétními zranitelnostmi.
  - **unit_tests** - jednotkové testy komponenty `fuzzer`.
- **parser** - zdrojové kódy komponenty `parser`.
- **procmon** - monitor procesů pro Windows a Linux.
  - `README.md` - soubor popisující návod k použití monitoru procesů.
- **reporter** - zdrojové kódy komponenty `reporter` pospolu s knihovnou SAXON.
- **tests** - implementace systémových testů.
- `config_example.json` - ukázkový konfigurační soubor nástroje.
- `Dockerfile` - předpis pro vytvoření Docker obrazu.
- `README.md` - základní informace a návod k použití implementovaného nástroje.
- `run.ps1` - spouštěcí skript pro systémy Windows.
- `run.sh` - spouštěcí skript pro UNIX systémy.

1. `https://en.wikipedia.org/wiki/Zip_(file_format)`