Univerzita Hradec Králové
Fakulta informatiky a managementu
Katedra informačních technologií

Možnosti využití fuzz testingu

Diplomová práce

Autor: Bc. Kristýna Hnízdilová
Studijní obor: Aplikovaná informatika (Ai2-p)

Vedoucí práce: Mgr. Josef Horálek, Ph.D

Hradec Králové duben 2023

Prohlášení:

Prohlašuji, že jsem diplomovou práci zpracovala samostatně a s použitím uvedené literatury.

V Hradci Králové dne 27.4.2023 Kristýna Hnízdilová

Poděkování:

Ráda bych tímto poděkovala vedoucímu diplomové práce Mgr. Josefu Horálkovi, Ph.D za metodické vedení práce, cenné rady během konzultací a jeho pozitivní a vždy optimistický přístup. Dále bych chtěla vyjádřit velké díky mé rodině a přátelům, kteří mi byli podporou po celou dobu mého studia.

Anotace

Diplomová práce se zabývá problematikou testování, konkrétně fuzz testingem. Cílem diplomové práce je na základě podrobné rešerše navrhnout metodiku pro využití fuzz testingu pro testování stability, korektnosti a bezpečnosti aplikací a informačních systémů. V teoretické části je zpracována podrobná rešerše v oblasti testování softwaru, hledání chyb, fuzzingu a quality assurance v návaznosti na vývojový proces projektu. Na základě této rešerše je provedena analýza možností využití jednotlivých typů fuzz testování a zmapovány jednotlivé nástroje pro fuzzing. Dále je navržena metodika testování pro využití fuzz testingu, která bude ověřena sadou navržených testů a výstupního reportů pro vybraný projekt.

Annotation

The diploma thesis deals with the issue of testing, specifically fuzz testing. The aim of the thesis is to propose a methodology for the use of fuzz testing for testing the stability, correctness and security of applications and information systems based on detailed research. In the theoretical part, detailed research in the field of software testing, error finding, fuzzing and quality assurance is processed in connection with the development process of the project. Based on this research, an analysis of the possibilities of using individual types of fuzz testing is performed and individual tools for fuzz testing are mapped. Furthermore, a testing methodology using fuzz testing is proposed, which will be verified by a set of proposed tests and output report for the selected project.

Title: Possibilities of using fuzz testing

Obsah

Seznam obrázků

Seznam tabulek

1 Úvod

Během celosvětové pandemie se společnost ještě více uchýlila k životu online. Bylo masivně vyvinuto nepřeberné množství softwaru za poměrně krátkou dobu. S tím však roste i riziko chybovosti a náchylnosti na kybernetický útok. Společnost Check Point® Software Technologies Ltd. se svém dokumentu Každá druhá organizace hlásí během koronavirové pandemie nárůst kybernetických útoků [1] zabývala dotazníkovým šetřením ohledně kybernetických útoků. Uvádí například, že 58 % respondentů čelí vyšší míře kybernetických útoků v době pandemie, než tomu bylo před začátkem pandemie.

K vývoji software neodmyslitelně patří důraz na kvalitu a testovací procesy. Bez kontroly kvality a objevování bezpečnostních hrozeb by současný software v konkurenci neobstál. Stačí jedna chybová událost a firma se může potýkat s devastující finanční ztrátou. Například v roce 2021 se firma Tesla potýkala s problémy ve svém softwaru Full-Self Driving. Zákazníci reportovali falešná hlášení jejich vozidla o čelní srážce, kdy následkem bylo nouzové brždění vozidla. Tesla následně objevila chybu ve svém beta software, vydala aktualizaci a upozornila uživatele na tuto chybu. [2] Testeři v tomto případě měli za úkol odhalit bezpečnostní rizika a kritické chyby ještě před zavedením systému do ostrého provozu. Čím větší a komplexnější je však řešení, tím méně efektivní jsou nejčastěji používané manuální testy.

Při testování a odhalování bezpečnostních hrozeb se proto seniornější odborníci zaměřují na automatizaci, která nabízí rychlé a efektivní řešení. Na trhu je možné dohledat nepřeberné množství frameworků pro testovaní designu, funkcionalit i výkonu. Hojně využívaná a vysoce automatizovaná testovací technika se nazývaná fuzzing. Cílem fuzzy testingu je primárně zaslání neplatných dat přes vstupy do softwaru tak, aby došlo k pádu. Fuzzing je široce používán jak odborníky na bezpečnost, tak mezi odborníky na zajištění kvality (QA), ačkoli někteří lidé stále trpí mylnými představami o jeho schopnostech, efektivitě a praktické implementaci. [3]

Cílem práce je představit a přiblížit širokou oblast zvanou fuzzy testing. Důraz je kladen na principy fungování a souvislost s quality assurance v praxi, představení nástrojů a sady praktických ukázek.

2 Chyby softwaru

Testování softwaru lze definovat jako porovnání vyvíjeného software s vypracovanou a schválenou specifikací. Pokud software splňuje veškeré zadání, je možné ho označit jako vyhovující. Pokud jsou však odchylky oproti specifikaci velké, software požadavkům nevyhovuje, a je třeba ho upravit. [4]

Testy mají za úlohu odhalit co nejvíce chyb v aplikacích. Softwarové bugy jsou chyby v počítačových programech, které zavádějí programátoři. Tyto chyby mohou způsobit různé nechtěné efekty jako jsou pády, zablokování nebo nesprávné chování programu. Důsledky takových chyb sahají od malých nepohodlí při používání software až po katastrofické scénáře, při kterých je ztraceno mnoho životů nebo peněz. Existuje několik typů softwarových chyb, jelikož existuje mnoho různých programovacích paradigmat a platforem, na kterých je třeba vyvíjet. [5]

Softwarové chyby se obecně kategorizují dle severity, priority a povahy dané chyby.

Dělení softwarových chyb dle priority závisí na koncovém zákazníkovi. Ten určuje prioritu chyby podle závaznosti dopadu na jeho podnik. Do chyb s nejnižší prioritou spadají například zarovnání textu na tlačítku nebo špatné textace. Tyto chyby nemají žádný výrazný dopad na funkci softwaru. Naopak chyby s nejvyšší prioritou musejí být řešeny co nejdříve, zpravidla se stanovuje limit několika hodin na zavedení opravy do produkce. Jako příklad lze uvést nefunkčnost provádět platební transakce v bankovním systému. [6]

Severita určuje míru technického dopadu chyby na software. Stejně jako priorita, je odstupňován dle závažnosti. Zde však na rozdíl od priority nehraje roli pohled zákazníka. Vysoká severita ne vždy znamená kritická priorita. Příklad vysoké priority a nízké severity je špatné logo na webové stránce. Dle zákazníka by měl být tento defekt vyřešen v co nejkratším časovém úseku, dopady na systém jsou však minimální.

Dle povahy lze definovat chyby:

- Výkonové – chyby snižující rychlost, stabilitu a zvyšující čas odpovědi
- Unit chyby – chyby ve výpočtech
- Funkční chyby – určitá funkce nebo celý software nefunguje dle specifikace
- Chyby použitelnosti – chyba blokuje používání software, například přihlašování do aplikace
- Syntaktické chyby – chyba kompilace kódu
- Logické chyby – chyba špatně postaveného kódu, kdy program může uvíznout ve smyčce nebo poskytovat nesprávný výstup
- Bezpečnostní chyby

Skupina softwarových chyb, které vedou k porušení bezpečnostních vlastností, je nazývána jako eskalace oprávnění a zranitelnosti softwaru pro spuštění libovolného kódu. Chyba zabezpečení má často bezpečnostní důsledky, které jsou velmi snadno dosažitelné pro uživatele s úmyslem zneužít systém. Útok DoS může způsobit například masivní výpadky webových stránek nebo služeb, nicméně nemělo by dojít k úniku citlivých dat nebo infekce škodlivým malwarem. Závažnější situace nastává, pokud je možné provádět libovolný příkaz. Útočník převezme kontrolu nad strojem oběti, který si pak dovolí provést sekundární útok uvnitř sítě, do které stroj oběti patří. Aby se předešlo velkému poškození, které je způsobené zneužitím zranitelností, je nutné tyto zranitelnosti rychle opravit, nejlépe s předstihem. Nutné je tyto zranitelnosti najít. Existuje mnoho přístupů k nalezení zranitelnosti. V minulosti vývojáři a bezpečnostní výzkumníci ručně kontrolovali zdrojový kód, pokud byl zrovna k dispozici, nebo rozebrání binárních programů. Tato metoda se však neškáluje, protože program se komplikuje. Pokrok v automatizovaných systémech, které kontrolují zranitelnosti programu, rychle otevřel cestu k efektivnímu hledání chyb. [5]

2.1 Přístupy pro hledání chyb

Za pomocí několika výzkumů zaměřených na vývoj bylo pozorováno, jak zvýšit bezpečnost a odstranit chyby zavedené vývojáři. V roce 2020 byl proveden výzkum od Bhaveet Nagaria a Tracy Hall, který poukazuje na chyby z lidské

perspektivy. Autoři v článku poukazují na důležitost koncentrace a soustředění na úkol. Za pomocí praktického příkladu bylo zjištěno, že vývojářům nejvíce pomáhá dobrá informovanost o programátorské úloze, kontrolní seznamy nebo například trénink kognitivních funkcí. [7] Další výzkum od profesorů z Montclair State University se také zabývá lidskou chybovostí. Článek klasifikuje několik lidských chyb, mezi které je zařazeno přehlížení designové dokumentace, či přehlížení API dokumentace. Další oblasti jsou nevymazání debuggovácích souborů nebo neošetření zadních vrat v programu při přesunu na produkční prostředí. [8]

Najít veškeré chyby v robustních programech je takřka nemožné. Vědci s tímto zjištěním však přišli na několik technik, které pomohou odhalit co nejvíce softwarových chyb za krátký časový úsek. [5] Tyto techniky jsou všeobecně dělené na statickou a dynamickou analýzu. Statická analýza kódu zkoumá kód k identifikaci problémů v rámci logiky. Dynamická analýza kódu zahrnuje spouštění kódu a zkoumání výsledku, což také zahrnuje testování možných cest běhu kódu. [9] Obě z technik mají své výhody a nevýhody, což výzkumníky vedlo k využití kombinace statické a dynamické analýzy ([10], [11], [12]).

2.1.1 Statická analýza

Tato analýza zpravidla vyhledává chyby ještě před samotným spuštěním programu. Může být provedena specializovaným nástrojem, kompilátorem, ale i osobou, která přezkoumá použití kódovacích standardů a konvencí (Code Review). [9]

Statická analýza obsahuje další kategorizaci technik, kam mimo jiné spadá statická analýza kódu a binární analýza.

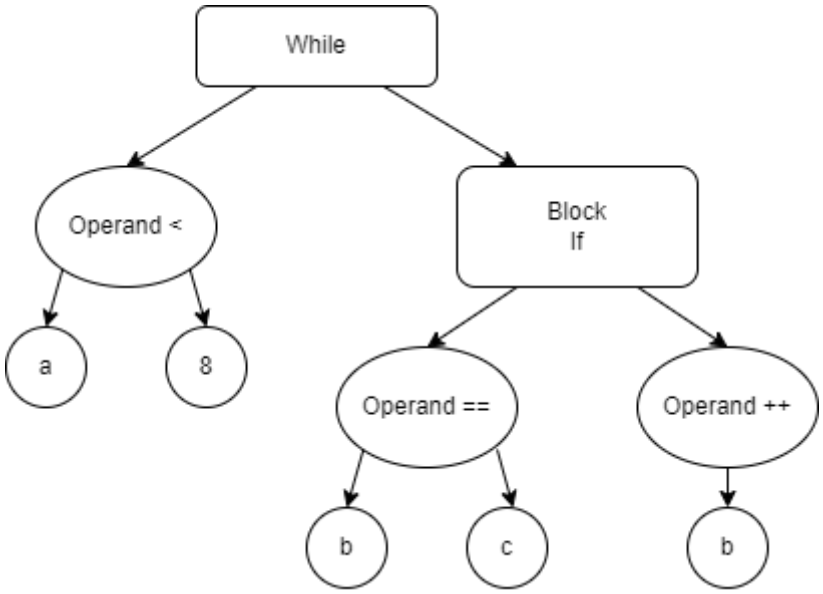
Statická analýza kódu

Statická analýza kódu má za úkol detekovat potencionální chyby ve zdrojovém kódu, aniž by bylo potřeba program spouštět. Analýza je obvykle prováděna za pomocí automatizovaných nástrojů, ve kterých jsou implementována pravidla. Tato pravidla se aplikují na zdrojový kód a ověřují, zda části kódu pravidla porušují. V závislosti na modelování může analýza statického kódu najít jednoduché a snadno přehlédnutelné chyby. Nachází ale i složité, které vyžadují pochopení

vztahu mezi souborem a třídou. [5] Statická analýza kódu je výkonná, levná a zároveň účinná. Může poskytnout detekci chyb v softwaru, které dynamická analýza nemůže snadno odhalit. Statická analýza kódu je obvykle implementována analýzou zdrojového kódu a vytvořením abstraktního syntaktického stromu (AST) programu. [13]

Pro demonstraci je uveden následující příklad:

```
WHILE (a < 8) {
IF (b == c)
      b++;
}
```



Obrázek 1 Abstraktní syntaktický strom (zdroj: vlastní)

Statická binární analýza

Statická binární analýza je založena na stejném principu jako statická analýza kódu, tedy bez exekuce programu. Kontroluje, zda v binárním kódu nejsou porušena pravidla. Platforma je však zcela odlišná. Statická analýza kódu byla zaměřena na zdrojový kód a vyšší programovací jazyk. Binární analýza se pohybuje spíše v kompilaci nízko úrovňových jazyků jako je assembler nebo na platformě nezávislých jako JAVA a .NET. Statická binární analýza může odhalit chyby, které jsou zaváděny kompilátory jako součást optimalizačního procesu. Statická analýza kódu nemůže chybu v tomto procesu odhalit. [5]

Výhody statické analýzy

- Přesné určení místa v kódu, kde se nachází slabina
- Umožňuje rychlejší opravy chyb
- Chyby lze detekovat v rané fázi vývoje, což snižuje náklady na další opravy
- Méně závad v pozdějším testování

Limitace statické analýzy

- Časová náročnost v případě výkonu osobou
- Nedostatek vyškoleného personálu k důkladnému provedení
- V běhovém prostředí nenalezne chybu
- Automatizované nástroje mohou produkovat falešné výsledky
- Automatizované nástroje mohou mít nedostatečně definovaná pravidla

[9]

2.1.2 Dynamická analýza

Pro překonání omezení statické analýzy se navrhlo další paradigma programové analýzy – dynamická programová analýza. [5] Dynamická programová analýza je populární technika pro analýzu programů. Analyzuje vlastnosti počítačového programu za jeho běhu. [14] Dynamická analýza je přesná, protože není třeba provádět žádnou aproximaci nebo abstrakci. Analýza zkoumá skutečné chování programu během spuštění. S jistotou je možné popsat jaké cesty řídicího toku byly použity a jaké hodnoty byly vypočteny. Dále je také možné přesně určit kolik paměti bylo spotřebováno a jak dlouho trvalo provedení programu. Dynamická analýza může být stejně rychlá jako běh programu. Některé statické analýzy probíhají poměrně rychle, ale obecně platí, že získání přesných výsledků vyžaduje velké množství výpočtů a dlouhé čekání, zejména při analýze velkých programů. [15]

Dynamická analýza toku dat

Dynamická analýza toku dat je orientována na typový stav a spojuje abstraktní hodnotu s objekty v programu. Na rozdíl od typů se typový stav objektu může během provádění změnit. Například popisovač souboru nebo řetězec si zachovává stejný typ po celou dobu své životnosti, ale jeho typový stav – otevřený nebo uzavřený, poskvrněný nebo neposkvrněný – se může během běhu programu měnit. [16]

Symbolická exekuce

Symbolická exekuce je další z technik dynamické analýzy. Používá se jako prostředek ověření programu k prokázání správnosti. Symbolická exekuce funguje tak, že dodává a sleduje abstraktní symboly namísto konkrétních hodnot. Očekává, že se přes vstupní symbol vygenerují symbolické vzorce. [5] Symbolická exekuce se stala atraktivní technikou ve výzkumu softwarového inženýrství pro své dvě vlastnosti. Netrpí významnými falešnými poplachy, protože spouštěný program poskytuje dostatek informací za běhu. Typické nástroje symbolické exekuce, jako jsou SAGE [17] a CUTE [18], produkují alarmy pouze tehdy, když za běhu nastanou softwarové výjimky. Druhou významnou vlastností je, že každý nový vstup vygenerovaný je schopen uplatnit novou cestu, což vede k vysokému pokrytí kódu procházením všech cest programu. [19]

Fuzzing

Fuzzing je metoda dynamické analýzy, která efektivně odhaluje zranitelnosti softwaru tím, že do programů cíleně vkládá chybná nebo neočekávaná data. Hlavní motiv je odhalit případy, které programátoři při testování přehlédli, jako jsou například extrémní nebo nesmyslné hodnoty. Fuzzing dosahuje svého cíle vkládáním generovaných nebo mutovaných vstupů do programu a monitoruje dopad na cílové programy, zejména pokud jde o pády. [5]

Výhody dynamické analýzy

- Identifikace slabiny za běhu programu
- Analýza aplikace bez nutnosti znát zdrojový kód
- Identifikace falešných výsledků ze statické analýzy
- Ověření výsledků statické analýzy
- Může být spuštěna vůči jakékoliv aplikaci bez ohledu na platformu

Limitace dynamické analýzy

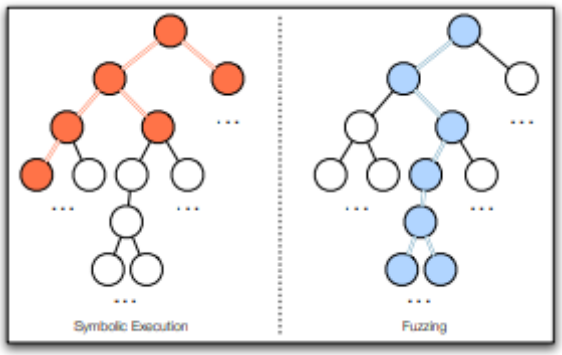
- Nelze zaručit úplné pokrytí testy
- Automatizované nástroje mohou produkovat falešné výsledky
- Automatizované nástroje mohou mít nedostatečně definovaná pravidla
- Obtížné dohledat zpětně přesné místo chyby v kódu
- Časová náročnost opravy je vyšší, než u statické analýzy

2.1.3 Hybridní fuzzing

V úvodu kapitoly bylo zmíněno několik výzkumů, které se zabývají kombinací statické a dynamické analýzy. Jako zástupce kombinace několika technik dynamické analýzy je uveden hybridní fuzzing. Autor Brian S. Pak se ve své práci Hybrid Fuzz Testing: Discovering Software Bugs via Fuzzing and Symbolic Execution zaměřuje na kombinaci symbolické exekuce a fuzzingu [5].

Symbolická exekuce je schopna objevit a prozkoumat všechny možné cesty v programu, ale není prakticky škálovatelná, protože počet cest se rychle stává exponenciální. Zatímco fuzzing je mnohem rychlejší než symbolické spouštění a tím zaručuje prozkoumání hlubší části kódu. [5]

Cílem je využít výhody symbolické exekuce k rozprostření do jedinečných cest a poté použít fuzzing k rychlému otestování každé cesty. Nelze pokrýt celý program, protože symbolická exekuce se v programu zastaví pro prozkoumání částí všech cest. Až poté přichází řada na fuzzing, který zkoumá zvolenou cestu do hloubky. Neznamená to však, že jsou prozkoumány úplně všechny cesty. [5]



Obrázek 2 Rozdíl mezi symbolickou exekucí a fuzzingem (zroj: [5])

Hybridní fuzzing se skládá ze čtyř fází: základní profilování bloků, symbolická exekuce, generování vstupů a řízené náhodné fuzzování. V prvních dvou fázích jsou shromažďovány predikáty cest pro cílový program, pomocí kterých je popisován vstupní prostor pro každou cestu. Poté jsou efektivně generovány náhodné testovací případy, které respektují predikát cesty. V poslední fázi jsou přivedeny vygenerované vstupy do cílového programu a sleduje se jeho chování. [5]

3 Fuzz testing

Fuzz testing, zkráceně fuzzing, lze definovat jako metodu pro objevování chyb v softwaru a jejich zaznamenávání za pomocí vkládání neočekávaného či zcela náhodného vstupu. Typicky se jedná o vysoce automatizovaný nebo poloautomatizovaný proces, který zahrnuje manipulaci a dodávání dat do cílového softwaru ke zpracování. [20]

Název pochází z tendence aplikací modemu selhat kvůli náhodnému vstupu způsobenému šumem na „fuzzy“ telefonních linkách. [3]

3.1 Typy fuzz testingu

Blackbox fuzzing

Fuzzer si v tomto typu fuzzingu neuvědomuje vnitřní obsah cílového programu. Vygenerované soubory jsou vkládány do programu a fuzzer sleduje výstup a chování programu. V mnoha dostupných fuzzerech se většinou používá jako výchozí přístup kvůli jednoduchosti a výkonu. Blackbox fuzzing má však velký problém s pokrytím kódu, protože se neustále dívá na stejnou část kódu. [5]

Greybox fuzzing

Fuzzer má odlehčený přístup ke generování testů. Tento přístup efektivně odhaluje chyby a slabá místa zabezpečení. Greybox fuzzery náhodně mutují vstupy programu, aby exekuovaly nové cesty. To však ztěžuje pokrytí kódu, který je střežen úzkými kontrolami. [21] Jako jeden z nejznámějších zástupců pro greybox fuzzing je považován nástroj AFL [22].

Whitebox fuzzing

Fuzzer využívá při tomto typu fuzzingu výhodu vnitřních struktur nebo designu cílového programu. Obvykle sleduje stav a každý běh má za cíl iterativně otestovat správnost určité části kódu. Cílem může být maximalizace pokrytí kódu nebo testování extrémních hodnot pro určitou funkci. Whitebox fuzzingu obvykle pomáhají jiné techniky, jako je symbolická exekuce, aby bylo možné určit konkrétnější chyby. [5]

3.2 Fáze fuzz testování

Toby Clarke ve svém vydání Fuzzing for software vulnerability Discovery uvádí pro jednotlivé fáze následující model:

1. Identifikace cíle
2. Identifikace vstupů
3. Generování fuzz dat
4. Monitorování výjimek
5. Stanovení využitelnosti

Identifikace cíle

Tato fáze je volitelná, protože cíl již mohl být vybrán. Útočníci si obvykle vybírají své cíle, zatímco testeři nemusí. Riziko, dopad a uživatelská základna jsou primárními faktory, které ovlivňují výběr cíle pro ty, kteří si cíl mohou vybrat. Nasazení zdrojů je primární faktor pro ty, kteří si cíl nevybírají. [13]

Atraktivní cíle pro útočníky mohou být:

- Aplikace, které přijímají vstup přes síť a které usnadňují snadné spuštění kódu vzdáleně. Jsou potencionální cíl pro internetové červy.
- Aplikace využívající vyšší oprávnění než běžný uživatel. Útočník může spustit kód na vyšší úrovni oprávnění, známé také jako eskalace oprávnění.
- Aplikace, které zpracovávají hodnotné informace. Útočník může obejít kontrolu a narušit integritu, důvěrnost a dostupnost cenných dat.
- Aplikace, které zpracovávají osobní údaje. Útočník by mohl obejít kontroly a narušit integritu, důvěrnost a dostupnost soukromých a osobních dat.

Cíle, které je možné definovat dvěma a více možnostmi z výše uvedených, jsou obzvláště ohrožené. Služba, která běží s oprávněním na úrovni Windows SYSTEM a také přijímá vstup ze sítě, je pro útočníka velmi významná. Velká uživatelská základna, tedy široce nasazená aplikace nebo výchozí komponenta

komerčního operačního systému představuje zvýšené riziko, protože dopad úspěšného útoku zvyšuje velká uživatelská populace. [13]

Identifikace vstupů

Identifikace vstupů má za cíl zanalyzovat a popsat plochu pro možný útok. Existuje mnoho podob vstupů do aplikace: síť, soubory, registry, proměnné prostředí, příkazy z příkazového řádku. Byly vyvinuty skupiny nástrojů, které seskupují uváděné vstupy:

- Argumenty příkazového řádku
- Proměnné prostředí (ShareFuzz)
- Webové aplikace (WebFuzz)
- Formáty souborů (FileFuzz)
- Síťové protokoly (SPIKE)
- Paměť
- Objekty COM (COMRaider)
- Meziprocesová komunikace

Generování fuzz dat

Účelem fuzzeru je testovat zranitelnost, která je přístupná prostřednictvím vstupů v aplikaci. Proto musí fuzzer generovat testovací data, která by měla vytyčit cílový vstupní prostor, který pak může být předán vstupu cílové aplikace. Testovací data mohou být buď generována jako celek před testováním, nebo častěji iterativně generována na vyžádání na začátku každého z balíku testů. Obecným přístupem k fuzz testování je opakované poskytování testovacích instancí cíli a monitorování odezvy. Pokud se během testování zjistí, že testovací případ způsobil chybu aplikace, představuje kombinace konkrétního testovacího případu a informace o povaze selhání, report o závadě. Zprávy o defektech lze považovat za finální výstup z fuzz testování pro předání k vývojářům. [13]

Monitorování výjimek

Generovaní a spouštění testovacích dat, která spouštějí projevy softwarových defektů, je nutné detekovat. Toho je dosaženo pomocí orákula, obecného termínu pro softwarovou komponentu, která monitoruje cíl a hlásí selhání nebo výjimku. Orákulum může mít formu jednoduché kontroly nebo může být ladicí program běžící na cíli, který monitoruje výjimky a shromažďuje podrobné informace o protokolování. [13]

Stanovení využitelnosti

Jakmile jsou identifikovány defekty na aplikaci, je nutné tento report zpracovat a předat na opravu k vývojářům. Tester může dané chyby také prošetřit a určit míru rizika zneužitelnosti.

3.3 Výhody fuzzingu

Fuzzing je obecně velmi výhodný a dokáže ušetřit čas, práci i finance za opravy chyb. Mezi největší výhody dle [23] patří:

- Zlepšení bezpečnosti: Fuzzing je účinný způsob, jak detekovat chyby a zranitelnosti v softwaru, které by jinak mohly být zneužity útočníky.
- Automatizace testování: Fuzzing umožňuje automatizaci procesu testování, což usnadňuje a urychluje detekci a opravu chyb.
- Zlepšení kvality kódu: Fuzzing pomáhá vývojářům najít a opravit chyby v jejich kódu, což pomáhá zlepšit jeho kvalitu a stability.

3.4 Nevýhody fuzzingu

Fuzzing má také své nevýhody. Čím rozsáhlejší je testovaný objekt, tím větší vzniká náročnost pokrytí celého softwaru a čas se naopak prodlužuje a testování vyžaduje kvalifikované odborníky. Zdroj [23] uvádí několik nevýhod, mezi které patří:

- Náročnost na zdroje: Fuzzing může být náročný na zdroje, jako jsou CPU, RAM a disk, což může vést ke zpomalení počítače a jiných aplikací běžících na stejném stroji.
- Složitost: Fuzzing může být složitý proces, který vyžaduje znalosti a zkušenosti s testováním software.
- Neplatné výstupy: Fuzzing může generovat neplatné výstupy, které nejsou relevantní pro testovanou aplikaci, což může způsobit ztrátu času a námahy.

4 Druhy a rozdělení fuzzerů

Fuzzer je možné definovat jako program, který náhodně vkládá data do programu a detekuje chyby. Část generující data je tvořena generátory a identifikace zranitelnosti se opírá o ladicí nástroje. Generátory obvykle používají kombinace statických fuzzing vektorů (známé jako nebezpečné hodnoty) nebo zcela náhodná data. Fuzzery nové generace používají genetické algoritmy k propojení vložených dat a pozorovaného dopadu. Takové nástroje zatím nejsou veřejné. [24]

Pro fuzzing existují klasifikace v závislosti na útočných vektorech, cílech, fuzzing metodě a mnoho dalšího. Mezi aplikační cíle patří formáty souborů, síťové protokoly, argumenty příkazového řádku, proměnné prostředí, webové aplikace a mnoho dalších. [25] Za zmínku stojí i rozdělení dle známé organizace OWASP, která se zabývá bezpečností aplikací a každoročně vydává seznam deseti nejkritičtějších chyb zabezpečení webových aplikací. OWASP foundation na svých stránkách rozdělila fuzzery následovně [26]:

- Fuzzing webových aplikací – URL adresy, formuláře, uživatelsky generovaný kontent
- Fuzzing protokolů – zasílání podvržených paketů
- Fuzzing pro formát souborů – zasílání špatně formátovaných souborů

Při následujícím dělení fuzzerů je třeba klást si otázky [27]:

- Jaká data se mají používat na vstupech? Jak jsou generovány testovací případy?
- Co je známo o testovaném objektu? (Whitebox, greybox a blackbox, představené v předchozí kapitole)
- Rozumí fuzzer struktuře dat či je jen náhodně generuje bez potřebných znalostí?
- Zajišťuje fuzzer pokrytí a upravuje dle toho sadu testovacích dat?

Druhy dle generování testovacích případů

Mutation-based: Mutační fuzzer generuje testovací data na základě existujícího vzoru. Tyto testovací data jsou následně upravována (mutována) na různé způsoby, aby bylo zjištěno, jak software reaguje na neobvyklé vstupy. [23] Mutační fuzzery jsou schopny za velmi krátký čas vygenerovat velký objem testovacích případů. [5] Jako zástupce mutation-base nástrojů je SAGE [17].

Generation-based: Generační fuzzer tvoří testovací data od nuly na základě specifikací protokolu, formátu dat atd. Tyto testovací data jsou navrženy tak, aby otestovaly co nejvíce vstupních bodů a funkcí v software. [23] Místo mutování dostupných počátečních souborů se soubory automaticky generují na základě šablony (gramatiky), která popisuje syntaxi souboru. Tento typ generování vstupu vyžaduje dobré modelování vstupu, což vyžaduje důkladné pochopení specifikace. [5] Zástupce generation-based nástrojů je SPIKEfile [28].

Grammar-based: Tento druh fuzzeru je všestrný, hodí se na testování webových aplikací, konkrétně HTML stránek a stránek postavených na JavaScriptu. Testuje se s ním ale i správnost kompileru nebo webové protokoly. Funguje na principu definování vstupní gramatiky a vstupního formátu uživatelem. Obvykle uživatel i specifikuje konkrétní vstupy, které mají být fuzzované a jak. Z této gramatiky pak fuzzer generuje další nové vstupy. [29] Příkladem jsou nástroje SPIKE [28] a Sulley [30].

Protocol-based fuzzer: Protocol-based fuzzer se používá k testování specifických protokolů, jako jsou například HTTP, DNS nebo FTP. Tyto fuzzery generují testovací data na základě specifikace protokolu a testují, jak software reaguje na různé neplatné vstupy. [23]

A. Tekanen ve své knize [3] zmiňuje také kategorizaci dle složitosti testovacího případu.

Statické a náhodné fuzzery založené na šablonách obvykle testují pouze jednoduché protokoly žádost – odpověď nebo formáty souborů. Nezahrnuje žádnou dynamickou funkci. Povědomí o protokolu není téměř žádné.

Blokově založené fuzzery implementují základní strukturu pro jednoduchý protokol žádost – odpověď a mohou obsahovat některé základní dynamické funkce, jako je výpočet kontrolních součtů a hodnot délek.

Fuzzery založené na dynamickém generování nebo evoluci nemusí nutně rozumět protokolu nebo formátu souboru, který je fuzzován, ale učí se to na základě zpětné vazby od cílového systému. Mohou nebo nemusí narušit sekvence zpráv.

Fuzzery založené na modelování nebo na simulaci implementují testované rozhraní buď prostřednictvím modelu nebo simulace, nebo mohou být také plnou implementací protokolu. Nejen struktury zpráv jsou fuzzované, ale také mohou být generovány neočekávané zprávy v sekvencích.

5 Analýza dostupných nástrojů

Dnes je dostupná celá škála nástrojů umožňující automatizaci. Mnoho z nich lze nalézt i v základní výbavě Linuxových distribucí pro penetrační testování. Konkrétně se jedná o Kali Linux, Parrot Security nebo Arch Linux. [31]

5.1 Přehled dostupných fuzzerů

Crashme

Kde nalézt	https://github.com/legendecas/crashme
Druh	Generation-based fuzzer
Zaměření	Testuje stabilitu a odolnost jádra OS

Tabulka 1 Nástroj Crashme (zdroj: [27])

American Fuzzy Loop (AFL)

Kde nalézt	https://github.com/google/AFL
Druh	Mutation-based fuzzer
Zaměření	Brute-force nástroj, generuje testovací případy a dokáže pokrýt velké množství testovaného kódu Testuje jakýkoliv typ aplikace – binární systém
Nároky	Linux x86, MacOS x86, Solaris x86 (Pro Windows existuje alternativa WinAFL)

Tabulka 2 Nástroj AFL
(zdroj: https://afl-1.readthedocs.io/en/latest/index.html)

Hypothesis

Kde nalézt	https://github.com/HypothesisWorks/hypothesis
Druh	Generation-based fuzzer
Zaměření	Druh jednotkových testů – fuzzuje zdrojový kód a testuje krajní případy
Nároky	Jakýkoliv OS s CPython verze 3.7+

Tabulka 3 Nástroj Hypothesis (zdroj: [32])

PythonFuzz

Kde nalézt	https://github.com/fuzzitdev/pythonfuzz
Druh	Generation-based fuzzer
Zaměření	Testování Python balíčků. Testuje neošetřené výjimky, logické a bezpečnostní chyby
Nároky	OS s Python verze 3.6+

Tabulka 4 Nástroj PythonFuzz (zdroj: https://github.com/fuzzitdev/pythonfuzz)

Gofuzz

Kde nalézt	https://github.com/google/gofuzz
Druh	Mutation-based fuzzer
Zaměření	Náhodné plnění objektů v jazyce GO Doplněk pro následující nástroj go-fuzz
Nároky	OS s instalovaným jazykem GO verze 1.6+

Tabulka 5 Nástroj GoFuzz (zdroj: https://github.com/google/gofuzz)

Go-fuzz

Kde nalézt	https://github.com/dvyukov/go-fuzz
Druh	Mutation-based fuzzer
Zaměření	Testování balíčků GO Specializuje se na analýzu složitých vstupů – textových i binárních
Nároky	OS s instalovaným jazykem GO verze 1.6+

Tabulka 6 Nástroj go-fuzz (zdroj: https://github.com/dvyukov/go-fuzz)

JQF

Kde nalézt	https://github.com/rohanpadhye/JQF
Druh	Generation-based fuzzer
Zaměření	Platforma pro Javu Testování pomocí vstupů, které jsou generovány ve fuzzingové smyčce řízené pokrytím Bylo odhaleno několik bugů v openJDK nebo Apache Maven
Nároky	Java verze 8+

Tabulka 7 Nástroj JQF (zdroj: [33], [34])

Peach

Kde nalézt	https://peachtech.gitlab.io/peach-fuzzer-community/
Druh	Generation-based i mutation-based fuzzer
Zaměření	Framework – vytváření si vlastních fuzzerů Poskytuje monitorovací systém pro bugy Testování souborů, protokolů a streamů
Nároky	Windows: Microsoft .NET v4 Linux/OS X: Mono 2.10+

Tabulka 8 Nástroj Peach
(zdroj: https://peachtech.gitlab.io/peach-fuzzer-community/)

SAGE

Kde nalézt	https://github.com/microsoft/SymEx/tree/master/symcc/tools/sage
Druh	Mutation-based fuzzer
Zaměření	Whitebox testy velkých aplikací
Nároky	Windows x86

Tabulka 9 Nástroj SAGE (zdroj: [17])

Spike

Kde nalézt	https://github.com/guilhermeferreira/spikepp
Druh	Protocol fuzzer
Zaměření	Síťové protokoly Na bázi jazyku C
Nároky	Linux OS – BackTrack 4, Kali Linux, BlackArch Windows, MacOS Bez speciálních nároků

Tabulka 10 Nástroj Spike (zdroj: [35], [28])

FileFuzz

Kde nalézt	https://filefuzz.software.informer.com/2.0/
Druh	Generation-based fuzzer
Zaměření	Testování formátů Windows souborů Detekce výjimek způsobených fuzzovanými formáty souborů.
Nároky	Windows OS

Tabulka 11 Nástroj FileFuzz
(zdroj: https://filefuzz.software.informer.com/2.0/)

Sulley

Kde nalézt	https://github.com/OpenRCE/sulley
Druh	Generation-based fuzzer
Zaměření	Testování formátů souborů, síťových protokolů, argumentů příkazového řádku a dalších kódů Sledování zdraví cíle a možnost ho obnovit
Nároky	Windows, Linux, MacOS Instalovaný Python 2.7+

Tabulka 12 Nástroj Sulley
(zdroj: https://github.com/OpenRCE/sulley, [30])

Honggfuzz

Kde nalézt	https://github.com/google/honggfuzz
Druh	Mutation-based fuzzer
Zaměření	Whitebox testování softwarových aplikací
Nároky	Linux – BFD knihovna, libunwind, clang v.5.0+ Windows – CygWin Android – Android SDK/NDK

Tabulka 13 Nástroj Honggfuzz (zdroj: https://github.com/google/honggfuzz)

libFuzzer

Kde nalézt	https://github.com/llvm/llvm-project/
Druh	Mutation-based fuzzer
Zaměření	Testování knihoven – párovače regulárních výrazů, analyzátory textového nebo binárního formátu, komprese, síť, šifrování
Nároky	Linux, MacOS, Windows Nutná instalace Clang 9

Tabulka 14 Nástroj libFuzzer (zdroj: https://llvm.org/docs/LibFuzzer.html)

XMLFuzz

Kde nalézt	https://github.com/llvm/llvm-project/
Druh	Mutation-based fuzzer
Zaměření	Testování knihoven – párovače regulárních výrazů, analyzátory textového nebo binárního formátu, komprese, síť, šifrování
Nároky	Linux, MacOS, Windows Nutná instalace Clang 9

Tabulka 15 Nástroj libFuzzer (zdroj: https://llvm.org/docs/LibFuzzer.html)

Přehled nástrojů pro Linuxové distribuce

Název nástroje	Systém	Zařazení fuzzeru
wfuzz	KALI Linux, Black Arch	Bruteforce pro webové aplikace
sfuzz	KALI Linux, Black Arch	Simple fuzzer
Spike	KALI Linux, Black Arch	IMMUNITYsec's fuzzer creation kit in C
afl++	KALI Linux, Black Arch	American Fuzzy loop
Ssdeep – libfuzzy	KALI Linux	Fuzzy hashing – porovnání souborů
thc-ipv6 – atk-fuzz	KALI Linux	IPv6 fuzzer
Sloth-fuzzer	Black Arch	File fuzzer
uniofuzz	Black Arch	Browser, web services
Sulley	Black Arch	Python automated fuzzer
frisbeelite	Black Arch	USB fuzzer
Sshfuzz	Black Arch	SSH2 fuzzer
dharma	Black Arch	Generation-based fuzzer
peach	Black Arch	Mutation-based i generation-based fuzzer
zzuf	Black Arch	Transparent application input fuzzer
sandsifter	Black Arch	The x86 processor fuzzer.
trinity	Black Arch	Linux system call fuzzer

Tabulka 16 Vybrané fuzzery z Linuxových distribucí (zdroj: [36], [35])

Crashme

Tento nástroj je již poměrně letitý, nicméně stále je možné ho nalézt na moderních Linuxových distribucích. Crashme je generátor náhodné sekvence bajtů, které jsou následně spuštěny. Díky tomu se blíží k hledání neznámých instrukcí. Testuje se tak hlavně operační systém a jeho stabilita a odolnost. Předpokládá se, že náhodné sekvence by neměly narušit chod jádra OS. [27]

American Fuzzy Lop – AFL

Jedná se o jeden z nejslavnějších a nejpoužívanějších fuzzerů. AFL je využíván na jakýkoliv typ aplikace k zjištění potencionálních bezpečnostních problémů a neošetřených vstupů. Tento fuzzer je sada binárních aplikací a přípustná je i například kombinace s programovacím jazykem Python. AFL testuje různé vstupy a na základě chování aplikace se pokouší vygenerovat taková data, které způsobí aplikační chybu. [27]

Hypothesis

Tento nástroj byl původně zamýšlen pro aplikace psané v programovacím jazyce Python, ale postupně se rozšířil dál. Hypothesis generuje testy pouze na základě pár pravidel, která tvoří programátor. V případě klasických testů je třeba scénáře zapisovat explicitně. Fuzzeru stačí specifikovat jeden scénář a rozsah parametrů, díky kterým jsou testy následně generovány. [27]

Pro demonstraci je představen příklad, kdy je třeba provést fuzzing funkcí pro kódování a dekódování textu. Předpokládá se, že text, který je zakódován a následně dekódován se shoduje.

```
@given(s=st.text())
@example(s="")
def test_decode_inverts_encode(s):
    assert decode(encode(s)) == s
```

Funkce text() vrací to, co hypotéza nazývá vyhledávací strategií. Anotace @given pak vezme testovací funkci a přemění ji na parametrizovanou, která, když je zavolána, spustí testovací funkci v širokém rozsahu odpovídajících dat z této strategie. Anotace @example slouží k regresním testům. Obsahuje validní vstupy, které budou při každém spuštění testovány. [32]

Pythonfuzz

Pythonfuzz je určený pro použití v programovacím jazyce Python. Tento fuzzer má za cíl co největší pokrytí. Specialitou je použití takzvaného korpusu, který zajišťuje, že budoucí testování bude probíhat za stejných podmínek jako předchozí.

Příkladem je funkce, která je opakovaně volána s náhodnými daty.

```
from html.parser import HTMLParser
from pythonfuzz.main import PythonFuzz


@PythonFuzz
def fuzz(buf):
    try:
        string = buf.decode("ascii")
        parser = HTMLParser()
        parser.feed(string)
    except UnicodeDecodeError:
        pass
```

Pythonfuzz zaznamená každou výjimku kromě UnicodeDecodeError. Testování probíhá přes metodu HTMLParser.feed(), která veškeré výjimky může vyhazovat. [27]

Gofuzz a go-fuzz

Tyto nástroje jsou zaměřené na testy v programovacím jazyku Go. Go-fuzz se využívá k testování korektnosti vstupů, které jsou předávány funkcím a metodám. Našel již mnoho chyb, které jsou uvedeny na oficiálním githubu nástroje (https://github.com/dvyukov/go-fuzz#trophies).

Gofuzz je jednoduchý balík pro generování pseudonáhodných testovacích dat. Na rozdíl od Go-fuzz není zajištěna zpětná vazba založená na zjišťování pokrytí testy. [27]

JQF

Tento fuzzer je jedním z nejpokročilejších zástupců fuzz testing platforem pro Javu.

JQF používá abstrakci testování založeného na vlastnostech, díky čemuž je příjemné psát fuzz ovladače jako testovací metody JUnit. JQF umožňuje spouštění jednotkových testů s výkonem fuzzingových algoritmů řízených pokrytím, jako je Zest.

Cílem Zest je najít hluboké sémantické chyby, které nelze nalézt fuzzingovými nástroji, které kladou důraz pouze na logiku zpracování chyb.

Peach

Jedním z příkladů fuzzing frameworku je Peach. Peach je multiplatformní fuzzing framework napsaný v Pythonu. Mezi hlavní atributy Peach patří krátká doba vývoje, opětovné použití kódu, snadnost použití a flexibilita. Peach dokáže fuzzovat téměř cokoli, včetně .NET, SQL, sdílených knihoven, webu. [3]

Pro příklad je uveden následující kód. Je zapotřebí otestovat pole hesla protokolu.

```
group = Group()
gen = Block([
        Static('USERNAME: BOB'),
        Static('PASSWORD: '),
        Repeater(group, Static('B'), 2, 3),
         Static('\r\n'),
        ])
while True:
        print gen.getValue()
        group.next();
```

Static() se používá k vytvoření pevného řetězce v rámci Block(). Funkce opakovače je užitečná pro vytváření iterativně delších řetězců. Výsledky z tohoto kódu by měly být například tyto řetězce:

```
USERNAME: BOB
PASSWORD: BBBBBB

USERNAME: BOB
PASSWORD: BBBBBBBBBBBB
```

SPIKE

Spike je komplexní sada nástrojů, která slouží k vlastnímu vytváření nových fuzzerů. Fuzzové zprávy jsou nazývané jako Spikes a mohou být zasílány do síťové služby, aby vyvolávaly chyby. Nástroj je na bázi jazyka C a umožňuje jednoduché skriptování, ve kterém je možné použít dostupná primitiva, která Spike také nabízí.

Nástroj lze najít na BackTrack 4 Linuxu nebo populárním Kali Linuxu, případně se dá stáhnout na jakýkoliv Linux nebo Windows systém. [28]

Následující ukázka skriptu demonstruje zasílání proměnné za pomocí POST metody na example.com:

```
s_string("POST /test.php HTTP/1.1rn");
s_string("Host: example.comrn");
s_string("Content-Length: ");
s_blocksize_string("block1", 5);
s_string("rnConnection: closernrn");
s_block_start("block1");
s_string("inputvar=");
s_string_variable("inputvalue");
s_block_end("block1");
```

Sulley

Sulley je fuzzing framework, založený na bázi programovacího jazyku Python, který lze použít k fuzzování formátů souborů, síťových protokolů, argumentů příkazového řádku a dalších kódů. [30] Tento framework se skládá z několika komponent a cílem je zjednodušení reprezentaci dat, přenos dat a sledování cílů. Sulley má dle oficiálního návodu [37] několik výhod:

- Sleduje síť a metodicky o ní vede záznamy
- Monitoruje stav cíle a je schopen ho vrátit do předchozího bezzávažného stavu
- Detekuje, sleduje a také kategorizuje zjištěné závady
- Umožňuje paralelní fuzzing a šetří tak čas

SAGE

SAGE (Scalable automated Guided Execution) je první zástupce fuzzerů pro testy typu white-box. SAGE cílí na velké aplikace a jedno provedení může obsahovat až stovky milionů instrukcí. Proto je jeho část symbolické exekuce časově nejnáročnější. [17]

Od roku 2007 společnost SAGE objevila mnoho chyb souvisejících se zabezpečením v mnoha velkých aplikacích společnosti Microsoft, včetně obrázkových procesorů, přehrávačů médií, dekodérů souborů a dokumentových

parserů. SAGE našel zhruba jednu třetinu všech chyb objevených fuzzingem souborů během vývoje Windows 7 od Microsoftu. Protože SAGE je obvykle spouštěn jako poslední, všechny ostatní, včetně statické analýzy programu a blackbox fuzzingu, tyto chyby přehlédly. [17]

6 Fuzzing a quality assurance

Problémy s kvalitou software jako jsou chyby v návrhu nebo programátorské chyby jsou hlavním důvodem většiny známých zranitelností softwaru. [3] Kvalita softwaru je výsledkem souboru několika závislých činností, mezi které patří například analýzy a příprava testování. Testovací a analytické činnosti by měly probíhat již během vývoje, nikoliv na konci nebo být označovány pouze jako samostatná jedna fáze před procesem doručení softwaru. Obecně kvalita software však závisí na celém procesu vývoje softwaru a všech jeho dílčích částech. [38]

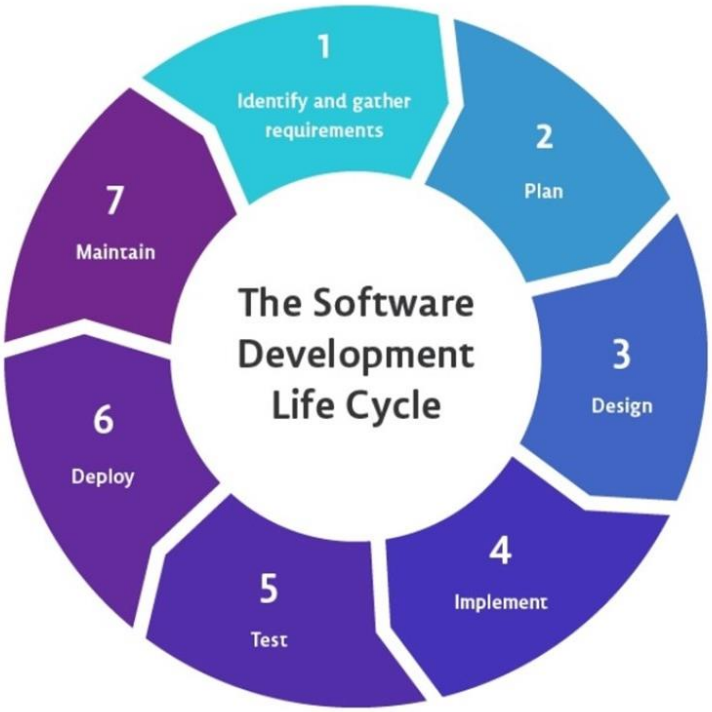
6.1 Proces vývoje software

Mnoho projektů během svého vývoje selžou a málokteré firmy mají veškeré své projekty úspěšné. [39] Když projekt selže, může to ohrozit pověst organizace u jejích zákazníků a její jméno na trhu. Kromě toho mohou být selhání projektů pro organizace nákladné, pokud jde o plýtvání penězi a zdroji. Pointa je, že projekty nadále selhávají z mnoha důvodů. Obecně se uznává, že k selháním softwarových projektů dochází převážně v důsledku překročení nákladů nebo harmonogramu a problémů s kontrolou kvality. [40]

Dle [40] jsou také uvedeny jako nejčastější příčiny následující:

- Špatné plánování
- Špatně definované požadavky
- Nedostatek času
- Zkrácená doba pro testování a zaručení kvality
- Neefektivní komunikace
- Nereálné projektové cíle
- Výběr technologií

Pro efektivní plánování se využívá životní cyklus projektu, který pomáhá udržet zaměření a dynamiku projektu na vysoké úrovni. Životní cyklus projektu rozděluje projektové řízení do několika fází, které jsou důležité pro plánování a vedou ke kontrole projektu, zda pokračuje dle zadání. [41] Vývojový proces dle [42], [43] a [44] zahrnuje těchto 7 fází:



Obrázek 3 Fáze procesu vývoje software (zdroj: [44])

Získávání požadavků

Počáteční fáze, kdy klient vyžaduje software na míru a koná se několik schůzek. Je třeba shromáždit veškeré jedinečné požadavky, aby bylo známo, proč je software tvořen, jak má fungovat a cíle, kterých se má pomocí něj dosáhnout. [42]

Analýza požadavků a plánování

Po shromáždění požadavků začíná analýza, tvoří se předběžná datová struktura a identifikují se rizika. Dále je třeba naplánovat časovou osu projektu. [42] Projektový a produktový manažer odhadují náklady, poskytování rezerv, alokaci zdrojů (lidských i materiálních) a plánují kapacity. Aby bylo zajištěno, že budou zastoupeny všechny perspektivy, měli by projektoví manažeři a vývojoví pracovníci spolupracovat s pracovníky provozu a bezpečnosti, aby odpovídali za všechny potřeby oddělení. [44]

Design a prototypování

Se splněnými požadavky je čas začít navrhovat, jak bude tento software vypadat a jak bude fungovat. Během procesu jsou navrhovány drátové modely anebo plnohodnotnější prototypy, které si zákazník může otestovat. Tato fáze je svěřena softwarovým architektům, kteří řeší vhodnost použitých technologií, problémy s algoritmy a návrhové vzory. [43] [44]

Implementace

Během této fáze vzniká samotný software a je zároveň označována za nejrizikovější. Vývojový tým může tuto fázi provádět v časově ohraničených „sprintech“ (Agile) nebo jako jeden blok (Waterfall). Bez ohledu na použitou metodu by vývojové týmy měly vytvořit funkční software co nejrychleji. Jako výsledek této fáze by měl být funkční testovatelný software. [44]

Testování

Již při implementaci by mělo docházet k pravidelnému testování, sledování a opravování chyb. Jakmile jsou dokončeny klíčové funkce, je třeba do hloubky otestovat celý software. [43] Tato část je považována za jednu z nejdůležitějších, je potřeba se ujistit, že zákazníkovi nebude odevzdán chybný software. K měření kvality je potřeba vykonat několik druhů testů [44]:

- Výkonnostní testy
- Funkční testy
- Unit testy
- Integrační testy
- Bezpečnostní testování (včetně fuzzingu)

Automatizace testů je nejlepší způsob, jak zajistit, aby byly spouštěny pravidelně a nikdy nebyly vynechány z důvodu účelnosti. Vyvinutý software je po náležitém otestování připraven k nasazení v produkčním prostředí. [44]

Nasazení a předání

K nasazení dochází, když je plánováno vydání, jedno vydání může mít více sprintů, plán vydání je plánován předem se všemi zúčastněnými stranami. Fáze nasazení by měla být v nejlepším případě vysoce automatizovaná za pomocí takzvaných Application Release Automation nástrojů. [43], [44]

Údržba a podpora zákazníka

Po spuštění produktu cyklus není u konce. Požadavky a potřeby zákazníků se vyvíjí a po uvedení produktu na trh se nepochybně najdou také chyby. Výrobní vady a chyby musí být hlášeny a řešeny, což často vrací práci zpět do procesu a celý cyklus začíná nanovo. [44]

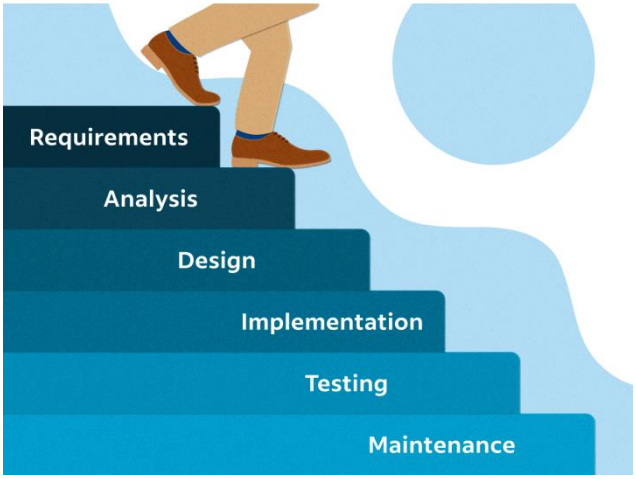
Výše nastíněný proces je obecným vodítkem. Kvalitní software lze díky těmto fázím zaručit, záleží však na druhu projektu, jak budou tyto fáze kontrolovány, kdy a v jakém pořadí. V průběhu let byla formalizována řada různých procesů vývoje softwaru, aby bylo možné řešit stále složitější projekty. [43]

Vodopádový přístup

Přístup vodopádového řízení projektu zahrnuje jasně definovanou posloupnost provádění s fázemi projektu, které nepokračují, dokud některá fáze nezíská konečné schválení. Jakmile je fáze dokončena, může být obtížné a nákladné vrátit se k předchozí fázi. Jediná zmeškaná lhůta nebo změna rozsahu během vodopádového projektu může způsobit nadměrný dopad na následná vydání. [45]

Je určen především pro týmy s pevnými strukturami a potřebami dokumentace. Vodopádový přístup díky své pevné struktuře a velkému počátečnímu času plánování funguje nejlépe, pokud se cíle, požadavky a zásobník technologií nezmění během procesu vývoje nezmění.
Vodopádový přístup má následující fáze:

- Plánování
- Upřesnění požadavků
- Návrh systému a softwaru
- Implementace
- Testování
- Rozvinutí
- Údržba – podpora



Obrázek 4 Vodopádový přístup (zdroj: [46])

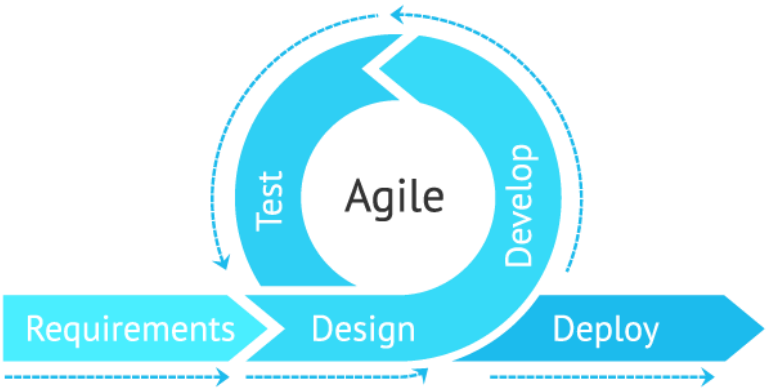
Agilní přístup

Agile byl poprvé přijat softwarovými týmy, které přešly od tradičního, sekvenčního vodopádového přístupu k metodě, která získávala konzistentní zpětnou vazbu a úpravy v průběhu životního cyklu vývoje. Agilní projektové řízení využívá iterativní přístup k vývoji vytvořením několika postupných kroků s pravidelnými intervaly zpětné vazby. [45] Na rozdíl od přísného sekvenčního vodopádového procesu pracují v Agile týmy ve „sprintech“ trvajících 2 týdny až 2 měsíce, aby vytvořili a uvolnili použitelný software zákazníkům pro zpětnou vazbu. [43]

Díky své dynamické a uživatelsky zaměřené povaze je Agile procesem vývoje softwaru, který upřednostňuje většina začínajících a technologických společností, které testují nové produkty nebo provádějí průběžné aktualizace již existujících produktů.

Dynamická povaha Agile znamená, že projekty mohou snadno překročit svůj původní časový harmonogram nebo rozpočet a mohou vytvářet konflikty se stávající architekturou nebo mohou být vykolejeny v důsledku špatného řízení. [43]
Agile metodika v jednom sprintu obsahuje následující:

- Analýza požadavků
- Design
- Implementace
- Testování
- Nasazení
- Plánování dalšího sprintu



Obrázek 5 Agilní přístup (zdroj: [47])

Následující kapitoly budou zaměřené na fáze testování, které jsou nedílnou součástí procesů vývoje software.

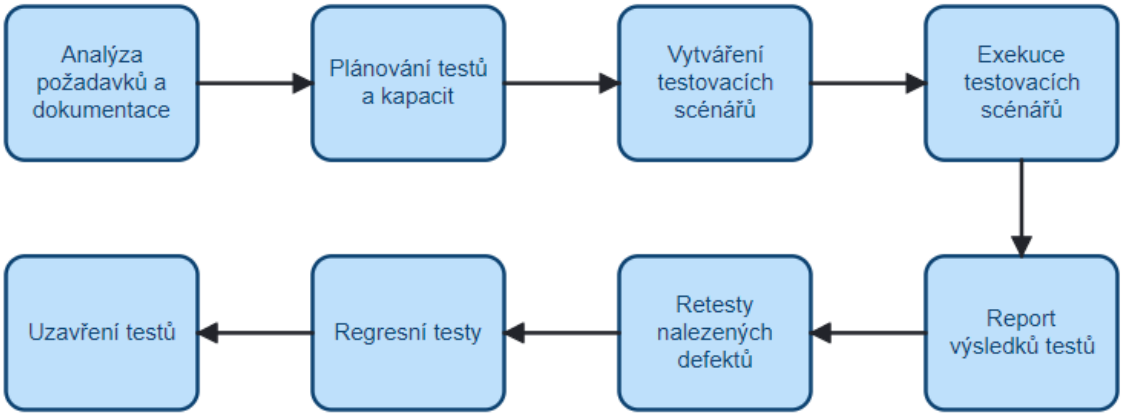
6.2 Software quality assurance

Software Quality Assurance (SQA) je opakovatelný proces, který je integrován s projektovým řízením a životními cykly vývoje software, aby přezkoumal mechanismy vnitřní kontroly a zajistil dodržování softwarových standardů a postupů. Cílem je zajištění schody s požadavky, snížení bezpečnostních rizik a zlepšení kvality při dodržování stanoveného harmonogramu a rozpočtu. [40]

6.2.1 Testovací proces

Testování softwaru lze považovat za činnost založenou na riziku. Během procesu testování je důležité, aby testeři softwaru pochopili, jak minimalizovat velký počet testů na zvládnutelnou sadu testů, a dělat rozhodnutí o rizicích, která jsou důležitá pro testování či která nejsou. Testovací cyklus se skládá z několika fází, od plánování testu po analýzu výsledků testů.

Během první fáze provádí tým kontroly kvality přezkoumání softwarových požadavků, během kterého nemálo dochází i ke koordinaci testovacího a vývojového týmu. Během analýzy požadavků téměř vždy vznikají otázky ohledně správné funkčnosti. Plánování testů, které jsou součástí druhé fáze, je především plán všech testovacích aktivit, které mají být provedeny v celém testovacím procesu. Vývoj testů je třetí fáze životního cyklu testování, kde se vyvíjejí testovací případy, které mají být použity v procesu testování. Během tohoto procesu jsou navrhovány testovací případy, testovací scénáře a testovací instrukce. Exekuce testu je následující fází testovacího cyklu, která zahrnuje provádění testovacích případů, a příslušné chyby jsou hlášeny v návazné fázi, kterou je fáze reportu. Při této příležitosti jsou zadávány bugy na vývojáře, které jsou následně opětovně testovány. Uzavření testování pak zařídí sada regresních testů, které jsou povětšinou automatizované. [48]



Obrázek 6 Testovací proces (zdroj: vlastní)

6.2.2 Quality assurance a fuzzing

Tradiční postupy zajišťování bezpečnosti v rámci quality assurance se obvykle odehrávají v pozdní fázi životního cyklu vývoje softwaru. Zaměřují se však na ochranu před již známými zranitelnostmi v již nasazených systémech. Tradiční hodnocení bezpečnosti se nesnaží hledat nic nového a jedinečného, nicméně se stále dobře hodí pro procesy nasazení. Fuzzing je velmi odlišný od tradičního zajištění bezpečnosti, protože jeho účelem je nalézt nové a dříve nezjištěné nedostatky. Hodnocení bezpečnosti pomocí fuzzingu se téměř vždy provádí na dokončeném nebo dokonce nasazeném produktu. Fuzzing obvykle provádějí pouze odborníci na hodnocení zranitelnosti. [13]

7 Fuzzing v umělé inteligenci

Tradiční techniky fuzzingu jakožto zástupce technologie zajištění bezpečnosti mohou mít mnoho problémů. Objevují se otázky, jak mutovat při počátečním vstupu, jak zvýšit pokrytí kódu a jak efektivně obejít ověřování. Technologie neuronových sítí a strojového učení byla integrována do fuzzingu, aby pomohla zmírnit tyto problémy. [49]

Jako nový přístup byl popsán Neural fuzzing, který se spoléhá právě na strojové učení a neuronové sítě. Je to relativně nový koncept, který se prosadil se zavedením hluboko učících se neuronových sítí, čímž se otevřely možnosti pro vývoj metod k identifikaci zranitelností v softwaru a sítích pomocí technik strojového učení ke generování vstupů. Například díky tomu, že umělá inteligence rychle získává na popularitě, vědci společnosti Microsoft byli schopni zlepšit techniky fuzzingu pomocí hlubokých neuronových sítí a strojového učení, aby lépe odhalovali chyby tím, že se poučili z předchozích zkušeností. Neuronové modely vyvíjejí funkci pro předvídání příznivých a špatných pozic ve vstupních souborech, aby mohly dělat fuzz mutace na základě mutací předchozích a relevantních dat. [50]

Hlavní výhodou neuronového fuzzingu je, že může projít různé části systému, včetně cest, které testeři přehlédli. Tímto způsobem může neurální fuzzing najít chyby, které lidský faktor nezaznamenal, a zvýšit efektivitu a přesnost procesu testování. [50]

Tato technika může rychle najít mnoho problémů s částí softwarového kódu, ale může také způsobit nestabilitu kvůli neustálému padání a opětovnému otevírání aplikací, což má za následek ztrátu dat a zranitelnosti zabezpečení. [50]

7.1 Průběh fuzzingu za pomocí technik strojového učení

Fuzzing za pomocí neuronových sítí dodržuje dle [49] následující postup, který se však výrazně neliší od klasického postupu fuzzingu:

Generování počátečního souboru

Vstupní soubor je mutován do vstupního vzorku pomocí různých mutačních operací. Kvalita vstupního souboru zásadně ovlivní celý zbytek procesu testování. Pomocí technik strojového učení lze na základě společných vlastností různých počátečních souborů vygenerovat více nových vstupních souborů, které povedou k vyššímu pokrytí kódu.

Generování testovacího případu

Vygenerování testovacího případu lze provést mutací vstupního souboru, a to na základě známých formátů vstupních souborů. Poslední vstup bude přímo ovlivňovat obsah testovacího případu a to, zda se chyba spustí či nikoliv. Proto sestavení testovacího případu s vysokým pokrytím kódu nebo zaměřeným na zranitelnost může účinně zlepšit efektivitu detekce zranitelnosti ve fuzzeru.

Filtrování testovacího případu

Během fuzz testingu se naskytne velké množství vzorků a je velmi časově náročné a neefektivní provést veškeré vzorky. Filtrování účelně vybere testovací vstup, který s vyšší pravděpodobností objeví nové cesty a nové zranitelnosti. Vstupní vzorky lze analyzovat a klasifikovat, aby bylo možné určit, které vzorky by měly být dále zpracovány k nalezení slabých míst zabezpečení pomocí technik strojového učení.

Výběr mutace operátora

Koncept mutačních operátorů ve fuzzingu je odvozen z biologického genetického algoritmu. Mutační operátor zahrnuje operace jako přidání, úpravu a odebrání. Různé mutace na různých místech mohou mít různé účinky. Strategie výběru operátora mutace je dosáhnout cílů zlepšení efektivity fuzzingu, jako je zvýšení pokrytí kódem nebo zahrnutí zranitelných cest.

Fitness funkce

Fitness funkce označuje metodu hodnocení používanou k rozlišení uspokojivých a neuspokojivých standardů testovacích případů. Mezi běžné fitness funkce patří pokrytí kódem a potenciální pozice zranitelnosti.

Analýza využitelnosti

Využitelnost zranitelnosti označuje možnost, že tato objevená zranitelnost bude napadena a zneužita útočníkem. Ve fuzzingu dochází k mnoha pádům a chybovým zprávám, ale ne všechny lze označit za zranitelnosti. Jak najít skutečné zranitelnosti z těchto havárií je výzva. Běžně používané metody analýzy zranitelnosti jsou statická analýza a dynamická analýza, které již byly v této práci zmíněny.

7.2 AI fuzzing nástroje

One Fuzz

Od společnosti	Microsoft
Oficiální stránky	https://github.com/microsoft/onefuzz
Dostupnost	Open Source – zdarma
Dokumentace	https://github.com/microsoft/onefuzz/tree/main/docs
Zaměření	Tvorba vlastních fuzzerů a správa vstupních dat Snadné pozorování průběhu a ladění při testech Plná integrace s Microsoft Azure a MS Teams
Nároky	Windows 10 nebo Windows Server 2019 Linux Podpora virtualizace Minimálně 8 GB RAM a 100 GB HDD

Tabulka 17 Nástroj One Fuzz (zdroj: https://github.com/microsoft/onefuzz)

OSS Fuzz

Od společnosti	Google
Oficiální stránky	https://github.com/google/oss-fuzz
Dostupnost	Open Source – zdarma
Dokumentace	https://google.github.io/oss-fuzz/
Zaměření	Podpora integrace na nástroje libFuzzer, AFL++, Honggfuzz Rychlá a efektivní automatizace testování Podpora jazyků C/C++, Rust, Go, Python and Java/JVM code
Nároky	Linux, Python 2.7, Git Minimálně 4 jádra CPU a 4 GB RAM

Tabulka 18 Nástroj OSS Fuzz (zdroj: https://google.github.io/oss-fuzz/)

Defensics Fuzz

Od společnosti	Synopsys
Oficiální stránky	https://www.synopsys.com/software-integrity/security-testing/fuzz-testing.html
Dostupnost	Pouze předplatné
Zaměření	Generační fuzzer Pokročilý fuzzing šablon souborů a protokolů pro vytváření vlastních testovacích případů Sada SDK umožňuje zkušeným uživatelům používat rámec Defensics k vývoji vlastních testovacích případů.
Nároky	Windows i Linux Minimálně 4 jádrový procesor Minimálně 8 GB RAM a 200 GB HDD

Tabulka 19 Nástroj Defensics Fuzz (zdroj: https://www.synopsys.com/software-integrity/security-testing/fuzz-testing.html)

FuzzBuzz

Od společnosti	FUZZBUZZ
Oficiální stránky	https://fuzzbuzz.io/
Dostupnost	Zdarma s omezením Předplacené plány pro týmy a společnosti
Dokumentace	https://docs.fuzzbuzz.io/
Zaměření	Fuzzer plně integrovaný s GIT nebo CI/CD Chyby nalezené za běhu, údajně žádná falešná pozitiva 20+ typů chyb Důsledně reprodukovatelné reporty o chybách
Nároky	Linux, Python 2.7 nebo 3.x, Docker Minimálně 2 jádra CPU a 2GB RAM

Tabulka 20 Nástroj FuzzBuzz (zdroj: https://docs.fuzzbuzz.io/)

8 Návrh metodiky

V praktické části diplomové práce bude zanalyzována situace a testovací procesy pro firmu spravující bankovní systémy. Následně bude navržen testovací plán a konkrétní fuzzery, kterými bude aplikace skenována. Na testování je třeba nastavit testovací prostředí a provést následující body:

- Identifikovat vstupní body
- Vygenerovat/vytvořit náhodná data
- Spustit fuzz testování
- Analyzovat výsledky
- Případné opakování testů

Po provedení testů bude sestaven report a bude zhodnoceno, jak fuzz testing celkově ovlivnil stav projektu a jak moc se vyplatí jej používat v testovacích procesech.

9 Návrh testování

9.1 Vyhodnocení procesu testování

Aplikace pod projektem se věnuje internetovému bankovnictví investiční banky. Jedná se o webovou aplikaci s mikroservisní architekturou. Mobilní verze k dispozici není. Psaná je v poměrně starých technologiích. Pro backend byla zvolena JAVA 8, frontend je psán v CoffeeSkriptu, který je v dnešní době ojedinělý. Pro datovou vrstvu je využívána databáze Oracle.

Jako první je potřeba vyhodnotit zaběhnuté standardy a postupy společnosti. Projektové řízení je řešeno v softwaru JIRA a veškeré dokumentace, postupy a také testovací případy jsou vedeny v Confluence. Testovací procesy se liší tým od týmu v závislosti na spravovaných aplikacích. Testy pro naši aplikaci jsou nyní prováděny především manuálně, existuje však sada automatizovaných regresních testů psaných v Katalonu, která se vždy spustila po nasazení nové verze aplikace na testovací prostředí. Tato sada čítá dohromady přes sto testů. Spravovaná aplikace prošla poslední rok velkou změnou a automatizované testy je třeba přizpůsobit.

Pravidelně probíhá funkční testování a velká pozornost je věnována i regresním testům. Další nedílnou součástí je integrační testování, vzhledem k tomu, že aplikace komunikuje s několika externími systémy, konkrétně s 5 systémy. Na to je využíván nástroj Postman. Občas je potřeba také přetestovat opravy z penetračního testování, které však vždy provádí specializovaná společnost na penetrační testy. Naopak nefunkčním testům, testům UX a designu věnována pozornost příliš není.

Testovací případy jsou vedeny v již zmíněné Confluence. Zde by byl prostor k vylepšení, kdy by bylo vhodnější zavést sofistikovanější nástroj určený pro vedení veškerých testovacích případů. V Confluence je veden vždy seznam verzí a pod ním příslušné testovací scénáře, testovací případy a testovací instrukce. Speciální prostor je pak věnován regresním testům (kritické scénáře), kde jsou popsané jak manuální testy, tak i automatizované a jejich přesné kroky. Tyto testy by měly projít vždy před odevzdáním nové verze zákazníkovi.

Shrnutí posledních vydávaných verzí:

Verze	Počet opravených bugů z minulých verzí:	Počet změnových požadavků:	Počet testovacích scénářů:	Počet MD na testy:
5.19	7	2 (+ 3 z penetračních testů	14	3,5
6.00	7	6	25	4,5
6.01	8	9	42	12

Tabulka 21 Shrnutí posledních vydaných verzí (zdroj: vlastní)

9.2 Návrhy fuzz testů

V této části budou navrhnuty jednotlivé testy pro různé oblasti. Využity budou fuzzery, které byly popsány v teoretické části práce. Probíhat bude white-bow testing, jelikož testovaná aplikace je známá k dispozici jsou i zdrojové kódy. Cílem je ověřit funkčnost aplikace a najít případné neodhalené chyby.

Při fuzz testování lze narazit na několik druhů odpovědí na requesty zasílané fuzzery:

Kód	Popis
200	Úspěšná odpověď – server zpracovat požadavek
204	Úspěšná odpověď bez obsahu, pouze s hlavičkou
301	Trvalé přesměrovánní na URL uvedené v hlavičce oddpovědi
302	Požadavek byl přijat a byl přesměrován na jinou URL adresu
303	Přesměrování na nové URL, zpravidla na POST požadavek
401	Neautorizovaný přístup
403	URL adresa nebyla rozpoznaná
405	Použitá metoda (GET, POST, PUT, DELETE) není povolena
500	Server nezpracoval požadavek

Tabulka 22 Popisy kódů odpovědí (zdroj: vlastní)

Návrh testů

Testovaná aplikace obsahuje sadu formulářů, které by mohly obsahovat potencionální chyby. Dále je navrhnut sken URL adresy a skrytých složek, které by útočník mohl zneužít. Fuzzovány budou také cookies.

9.2.1 Test 1 - Fuzz testing formulářů

Pro testy formulářů bude použit nástroj wfuzz. Jako cíl je identifikována aplikace pro internetové bankovnictví. Po nezbytné instalaci pythonu, pycurl a colorama modulů je možné nástroj spustit.

Nejprve je potřeba dostat se přes autorizace. To lze provést za pomocí několika metod, například provolat curl s POST požadavkem a uvedením přihlašovacích údajů. Následně je možné zaměřit se na samotné formuláře.

V případě tohoto testu budou fuzzovány formuláře pro:

A) Změna uživatelského jména

Identifikace vstupů: textový input ve formuláři – pole pro uživatelské jméno
Generování dat: Vytvořen slovník usernames.txt, který obsahuje nevalidní hodnoty. Nové uživatelské jméno musí splňovat následující podmínky:

- Délka musí být mezi 3 a 20 znaky
- Uživatelské jméno může obsahovat malá písmena, velká písmena, číslice, a speciální znaky
- Uživatelské jméno nesmí být totožné s jiným uživatelským jménem

Spuštění testu: Pomocí následujícího příkazu lze validovat formulář pro změnu uživatelského jména:

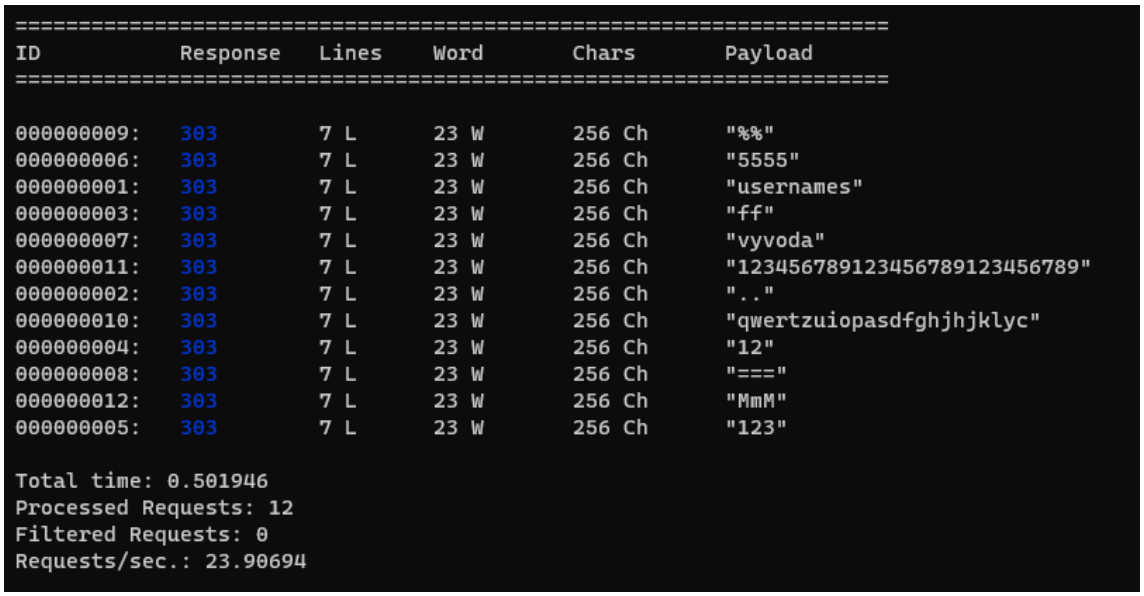
```
Wfuzz -c -z file,wordlist/general/usernames.txt -d "alias=FUZZ"
url_adresa
```

`-c` – defaultní nastavení připojení na server

`-z` – zpoždění požadavku na server, v kombinaci se slovníkem posílá postupně se zpožděním jednotlivá hesla a minimalizuje tak riziko blokování IP adresy serveru

`-d` – nastavení dat pro požadavek na server, pokud se jedná o POST požadavek

Wfuzz následně provolá request na server, kdy do inputu alias vkládá postupně slova z vytvořeného slovníku.



Obrázek 7 Výsledek fuzzingu pro změnu uživatelského jména (zdroj: vlastní)

Výsledek testování

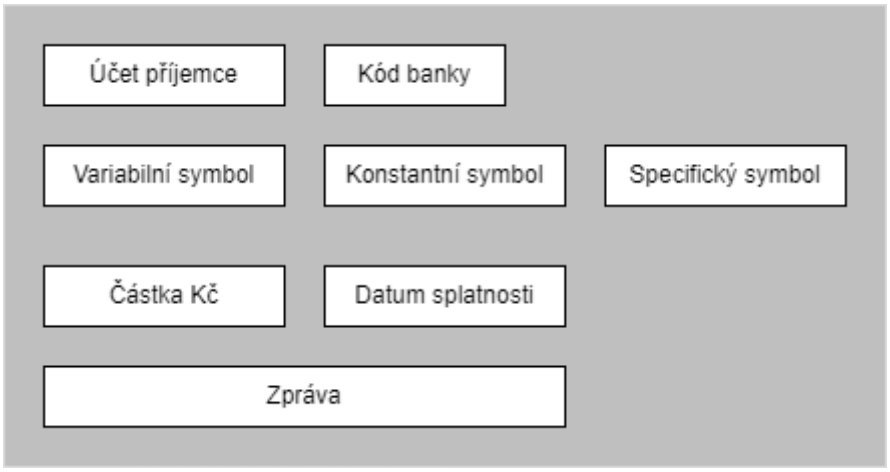
Z výsledku je patrný pokus o redirect. V tomto případě se jedná o redirect na autorizační komponentu a až poté je request zpracováván na serveru. Výsledek tohoto testu nelze považovat za zcela platný. Vzhledem k zabezpečení aplikace tedy konečné testy musely probíhat manuálně.

B) Odeslání transakce

Pro formulář transakce bude ověřováno následující:

- Ověřit formát čísla účtu
- Ověřit formát variabilního symbolu
- Ověřit pole pro částku transakce
- Ověření délky zprávy – maximální počet znaků

Identifikace vstupů: formulářové inputy pro účet příjemce, variabilní symbol, částku a zprávu



Obrázek 8 Formulář pro transakce (zdroj: vlastní)

Generování dat

Pro účty příjemce byl vytvořen slovník bankaccounts.txt, pro variabilní symbol varsymbols.txt a pro částku amount.txt. Testování délky znaků pro zprávu bude probíhat pouze v příkazu, kde bude zadán neúměrně dlouhý řetězec znaků.

Spuštění testu:

```
wfuzz -f output.txt -c -z file,wordlist/general/bankaccounts.txt -z
file,wordlist/general/varsymbols.txt -z
file,wordlist/general/amount.txt -d
"accountNumberCreditInput=FUZZ&bankCodeCredit=0100&variableSymbol=FUZ2
Z&amount=FUZ3Z&textarea=longlongtexttextextetxtextetxtetextetxtettxtet
txfttextextextextextextextetxetxtextetxetxtextetxtetxtetxtext" Url_adr
esa
```

`-f` – uložení výstupu do souboru

`-c` – defaultní nastavení připojení na server

`-z` – zpoždění požadavku na server, v kombinaci se slovníkem posílá postupně se zpožděním jednotlivá hesla a minimalizuje tak riziko blokování IP adresy serveru

`-d` – nastavení dat pro požadavek na server, pokud se jedná o POST požadavek

`FUZZ, FUZ2Z, FUZ3Z` – místo pro vkládání dat, byly použity veškeré kombinace ze tří slovníků

Wfuzz následně sestaví všechny možné kombinace slov ze tří slovníků. Výsledek je možné najít v přílohách. Opět je možné se setkat s redirectem na

autorizační komponentu. Na výsledku je však možné vypozorovat efektivní kombinaci všech slov a rychlost provolávání.

Výsledek testování

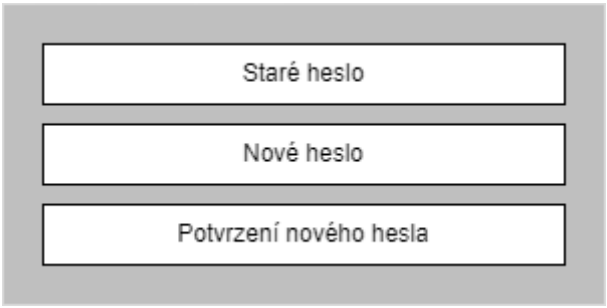
Výsledek je přílohou této diplomové práce. Je v něm patrný pokus o redirect. V tomto případě se jedná o redirect na autorizační komponentu a až poté je request zpracováván na serveru.

C) Změna hesla

Měněné heslo zadávané do formuláře musí splňovat následující podmínky:

- Délka musí být mezi 8 a 50 znaky
- Heslo musí obsahovat malá písmena, velká písmena a číslo
- Opakovaně stejné heslo

Identifikace vstupů: input pro nové heslo a input pro opakované zadání tohoto hesla



Obrázek 9 Formulář pro změnu hesla (zdroj: vlastní)

Generování dat: Pro změnu hesla byl vytvořen slovník passwords.txt, kdy heslo porušuje zmíněné podmínky

Spuštění testu:

```
wfuzz -f outputPasswords.txt -c -z file,wordlist/general/passwords.txt
-d "currentPassword=Heslo123&newPassword=FUZZ&reNewPassword=FUZZ"
```

`-f` – uložení výstupu do souboru

`-c` – defaultní nastavení připojení na server

`-z` – zpoždění požadavku na server, v kombinaci se slovníkem posílá postupně se zpožděním jednotlivá hesla a minimalizuje tak riziko blokování IP adresy serveru

`-d` – nastavení dat pro požadavek na server, pokud se jedná o POST požadavek

`FUZZ` – na místo inputu pro heslo byl použitý stejný slovník vždy 1:1, aby se heslo shodovalo

Obrázek 10 Výsledek fuzzingu pro změnu hesla (zdroj: vlastní)

Vyhodnocení výsledků

Nástroj Wfuzz se zdá být velmi užitečným nástrojem pro fuzzování formulářů, nicméně je zde jistá limitace. Pokud je aplikace již nasazena a obsahuje autorizační komponentu i pro úspěšné odeslání formuláře, nelze na výsledky testů brát zřetel. V tento moment je již mnohem výhodnější využít implementovaných automatizovaných testů a testovat krajní hodnoty formulářů. Použití wfuzzu je doporučitelné využít spíše během počátečního vývoje, kdy testeři mohou generovat slovníky s krajními hodnotami.

9.2.2 Test 2 - Fuzz testing URL

Fuzzování URL bylo také provedeno skrz nástroj wfuzz. Jako cílová aplikace byla tentokrát zvolena konzole pro operátory, která nedisponuje dvoufázovou autentizací.

Identifikace vstupu: URL konzole pro operátory

Generování dat: defaultní slovník common.txt, obsahující klíčová slova

Spuštění testu A)

V prvním kroku byl fuzzován odkaz před autentizací.

```
Wfuzz – z file,wordlist/general/common.txt --hc 404,403
url_adresa/FUZZ
```

`-z` – zpoždění požadavku na server, v kombinaci se slovníkem posílá postupně se zpožděním jednotlivá hesla a minimalizuje tak riziko blokování IP adresy serveru

--hc – znamená filtraci odpovědí 404 a 403, které jsou serverem automaticky zahazovány, tyty odpovědi se nezobrazí ve výsledku

Obrázek 11 Fuzzing URL (zdroj: vlastní)

Výsledek testování

Výsledkem pro tento test jsou 3 potencionálně zneužitelné adresy:

- https://url_adresa/client
- https://url_adresa/css
- https://url_adresa/js

Adresy s klíčovými slovy byly však prověřeny a jedná se o redirect, který je následně blokován kvůli nedostatečnému oprávnění.

Spuštění testu B)

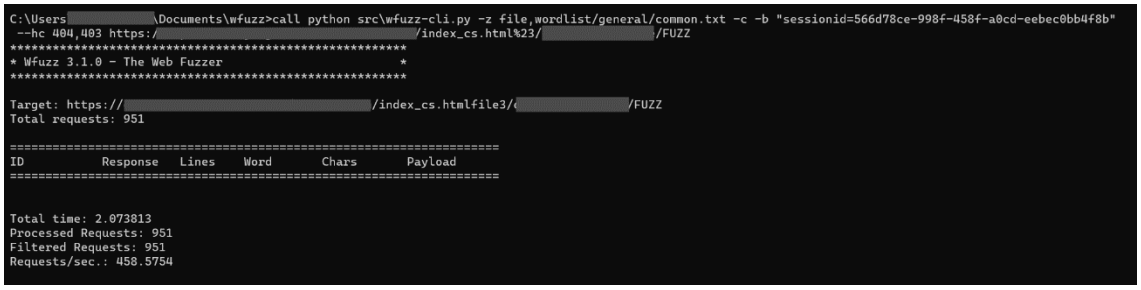
Dále se test soustředil na objevení URL po autorizaci do aplikace. Před samotným testem je třeba zjistit platné sessionid přihlášeného uživatele a následně ho použít.

```
Wfuzz –z file,wordlist/general/common.txt --hc 404,403 -b
“sessionid=566d78ce-998f-458f-a0cd-eebec0bb4f8b“ url_adresa/FUZZ
```

`-z` – zpoždění požadavku na server, v kombinaci se slovníkem posílá postupně se zpožděním jednotlivá hesla a minimalizuje tak riziko blokování IP adresy serveru

`--hc` – znamená filtraci odpovědí 404 a 403, které jsou serverem automaticky zahazovány, tyty odpovědi se nezobrazí ve výsledku

`-b` – autentizace do webové aplikace za pomocí platného session id přihlášeného uživatele



Obrázek 12 Fuzzing URL po autorizaci (zdroj: vlastní)

Výsledek testování: V tomto případě naštěstí nebyla odhalena potencionální adresa pro zneužití.

9.2.3 Test 3 - Fuzz testing cookies

Pro testing cookies bude využit opět nástroj wfuzz. Nástroj obsahuje defaultní wordlist, který obsahuje generované hodnoty.

Identifikace vstupu: URL adresa aplikace

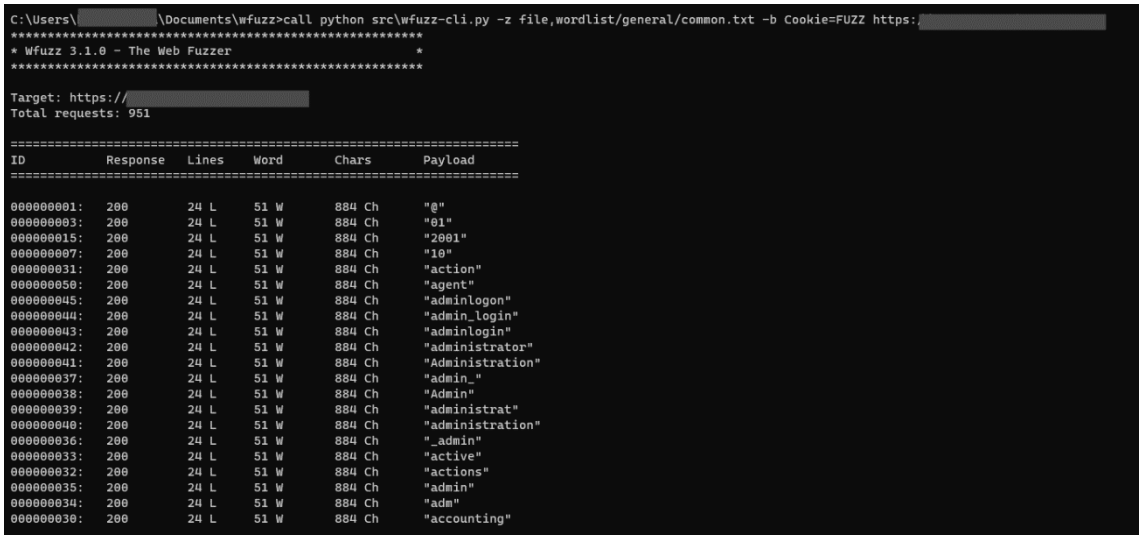
Generování dat: defaultní slovník common.txt, obsahující klíčová slova

Spuštění testu

Příkaz `wfuzz -z file,wordlist/general/common.txt -b "cookie=FUZZ" url_adresa` otestuje vkládání hodnoty cookies ze souboru common.txt

`-z` – zpoždění požadavku na server, v kombinaci se slovníkem posílá postupně se zpožděním jednotlivá hesla a minimalizuje tak riziko blokování IP adresy serveru

`-b` – specifikace parametru v hlavičce požadavku, v tomto případě cookies



Obrázek 13 Fuzzing cookies (zdroj: vlastní)

Výsledek testu

Při použití na aplikaci je možné vidět odpověď 200. Server tedy odpověděl na volání. Byly nalezeny platné hodnoty cookies, je však potřeba brát v potaz i další faktory jako šifrování https a doba platnosti cookies.

9.2.4 Objevení skrytých stránek a složek

Pro tento typ testů byl zvolen nástroj ffuf, jehož instalace je nenáročná. Funguje na základě generování requestů ze slovníku.

Identifikace vstupů: URL adresa aplikace

Generování dat: slovník admin-panel, který lze nalézt z přílohách

Spuštění testu

Příkaz `ffuf -c -w admin-panels.txt -u url_adresa` protestuje všechny možné případy.

`-w` – identifikace použitého slovníku

`-u` – URL adresa

`-c` – defaultní počet požadavků dle slovníku

Obrázek 14 Fuzzing skrytých stránek a složek (zdroj: vlastní)

Výsledek testování

V tomto případě byly nalezeny dvě potencionální zranitelnosti. Nutné podotknout, že se jedná o status 302. Stránky byly následně prověřeny ručně a obě byly přesměrovány na defaultní 404 stránku.

10 Analýza výsledků

Následující tabulka shrnuje konečné výsledky testů.

TEST	VÝSLEDEK
TEST FORMULÁŘŮ A)	Neodhalena kritická chyba
TEST FORMULÁŘŮ B)	Neodhalena kritická chyba
TEST FORMULÁŘŮ C)	Neodhalena kritická chyba
TEST URL	Neodhalena kritická chyba
TEST COOKIES	Nález nízké priority
TEST SKRYTÝCH SLOŽEK	Neodhalena kritická chyba

Tabulka 23 Výsledky testů (zdroj: vlastní)

Testování formulářů

V první sadě testů byly ověřovány formuláře za pomocí nástroje wfuzz. Nastavení nástroje je časově nenáročné a příkazy jsou intuitivní. Nástroj také obsahuje sadu již vygenerovaných slovníků, které lze používat. V testování bylo potřeba dostat se přes prvotní autentizaci a následně offuzovat formuláře. Formuláře jsou však zabezpečeny další autorizační komponentou. Proto musely být dotestovány automatizovanými testy a manuálně.

V minulosti se projekt setkával s chybami validací datumů a povinností vyplnit vstupy. Zde by byl prostor pro fuzz testing formuláře na vývojovém prostředí.

Testování URL

Testování URL probíhalo s pomocí wfuzz na aplikaci pro operátory internetového bankovnictví, kde není třeba procházet dvoufázovou autentizací. Testování odhalilo 3 místa, která nebyla serverem zamítnut, nicméně byla přesměrovaná na bezpečnou stránku. Nalezená místa byla ověřena a byla vyloučena chyba.

Testování URL adres je specifické spíše pro blackbox testing za účelem skenu zranitelných míst. Pro testy URL by bylo vhodnější použít specializované nástroje pro penetrační testy, například OWASP ZAP.

Testování cookies

Test cookies odhalil zajímavé výsledky, kdy do cookies lze vložit jakoukoliv hodnotu. Zde je ale nutné brát ohled na https šifrování. Tento nález byl předán na prošetření a bude do budoucna řešen.

Pro testy různě generovaných hlaviček požadavků je nástroj efektivní a může během krátké chvilky objevit nevalidní vstupy, které projdou. V kombinaci s nástrojem Burp Suite, který je používaný v testování bezpečnosti a dají se přes něj zasílat požadavky s upravenou hlavičkou, lze dosáhnout identifikaci chyb v hlavičkách.

Testování skrytých složek

Při testování skrytých složek byl použit nástroj ffuf. Nástroj lze velmi rychle uvést do provozu a začít testovat. Nabízí však velmi podobné možnosti jako wfuzz a neobsahuje v základu vygenerované slovníky. Při tomto testu byly odhaleny dvě potencionální skryté stránky, které byly ihned prověřeny. Odkazy však mají redirect na stránku 404 a nelze se přes ně kamkoliv dostat.

Na sken složek by bylo opět efektivnější použít specializovaný nástroj pro sken. Vygenerované slovníky v tomto případě nemohou stačit na pokrytí všech míst a bylo by vhodnější skenovat místo fuzzingu.

Fuzz testing webové aplikace pro internetové bankovnictví je velmi specifický a mnoho nástrojů na něj nelze použít. Zavádění nových technologií by znamenalo také negativní dopady na financování celého projektu. Vzhledem k tomu, že nebyla objevena kritická chyba, bylo by obtížné rozhodnutí o fuzzerech obhájit. Nenalezení kritické chyby může souviset také s tím, že aplikace je již několik let nasazena do ostrého provozu a pravidelně je testována i odborníky na penetrační testování a bezpečnost.

11 Závěr a doporučení

Cílem práce bylo zpracovat detailní rešerši ohledně problematiky fuzz testingu, porovnat jednotlivé nástroje a jejich zaměření a následně navrhnout sadu praktických testů, které ověřují užitečnost fuzzingu jako druhu testování ve vývojovém procesu.

V teoretické části práce bylo představeno několik přístupů k hledání chyb. Následně byl popsán fuzzing, jednotlivé fáze fuzzingu, výhody a nevýhody. Bylo uvedeno rozdělení fuzzerů a představeno několik nástrojů pro fuzzing. V závěru teoretické části byl vysvětlen proces vývoje softwaru, návaznost na quality assurance a zařazení fuzzingu do procesu. V poslední kapitole byla nastíněno propojení fuzzingu a umělé inteligence.

V praktické části byly shrnuty testovací procesy vybrané společnosti. Navrženo bylo několik testů, které využívají fuzzing nástroje.

Testování webové aplikace za pomocí fuzzerů odhalilo spoustu úskalí a nástrah. Z pohledu řízení projektu může být zavádění fuzz nástrojů velmi časově náročné a ze strany zákazníka může být zavádění těchto typů testů nesmyslné z finančních důvodů. Kvalitní fuzz testy vyžadují kvalifikované odborníky a mnoho času k výběru vhodných nástrojů, instalaci a vytvoření speciálních testů. Některé základní fuzzování a generování vstupů lze však využít již během samotného vývoje, před nasazením aplikace do ostrého provozu.

Nástrojů pro fuzz testing existuje celá řada, jejich použití je však velmi specifické. Mnoho z nich je omezeno pouze na jeden konkrétní jazyk nebo pouze generují data. Použití ve vývojovém procesu je užitečné, netřeba však trávit zbytečně několik hodin vymýšlením stejných testů. Fuzzing je nezbytnou součástí také softwarové bezpečnosti. Zde bych však nechala prostor pro specializované nástroje pro penetrační testování, které samy o sobě dokážou generovat data dle zadaných kritérií a například prolamovat hesla či generovat škodlivé soubory.

Cíl práce byl splněn – dle praktického využití zmíněných fuzzerů bylo vyhodnoceno, že fuzzing je do jisté míry pomocná technika při vývoji, nicméně nelze pokrýt všechny případy testování. Také jsou potřeba znalosti vysoce kvalifikovaných odborníků. Fuzz testing je vhodné kombinovat s manuálními a automatizovanými testy. Za zmínku stojí také nástroje pro penetrační testování, které obsahují funkčnosti jako samotné fuzzery a nabízejí mnohem více funkcí navíc. Jejich dokumentace jsou mnohdy podrobnější a uživatelské ovládání je přívětivější.

12 Seznam použité literatury

[1] Ř. David, „Každá druhá organizace hlásí během koronavirové pandemie nárůst kybernetických útoků". Check Point® Software Technologies Ltd., 5. únor 2021. [Online]. Dostupné z: https://www.dns.cz/sites/default/files/210205_check_point_vyzvy_2021_tz.docx

[2] M. Stojmanovska, „10 Biggest Software Bugs and Tech Fails of 2021", *TestDevLab*, 27. prosinec 2021. https://www.testdevlab.com/blog/10-biggest-software-bugs-and-tech-fails-of-2021

[3] A. Takanen, J. D. Demott, a C. Miller, *Fuzzing for software security testing and quality assurance*. in Artech House information security and privacy series. Norwood, MA: Artech House, 2008.

[4] P. Tsankov, M. T. Dashti, a D. Basin, „SECFUZZ: Fuzz-testing security protocols", in *2012 7th International Workshop on Automation of Software Test (AST)*, Zurich, Switzerland: IEEE, čer. 2012, s. 1–7. doi: 10.1109/IWAST.2012.6228985.

[5] B. S. Pak, „Hybrid Fuzz Testing: Discovering Software Bugs via Fuzzing and Symbolic Execution", 2012, [Online]. Dostupné z: http://ra.adm.cs.cmu.edu/anon/2012/CMU-CS-12-116.pdf

[6] ThinkSys, „16 Types of Bugs in Software Testing", *ThinkSys*, 14. květen 2022. https://www.thinksys.com/qa-testing/types-software-testing-bugs/

[7] B. Nagaria a T. Hall, „How Software Developers Mitigate Their Errors When Developing Code", *IEEE Trans. Softw. Eng.*, roč. 48, č. 6, s. 1853–1867, čer. 2022, doi: 10.1109/TSE.2020.3040554.

[8] V. Anu, K. Z. Sultana, a B. K. Samanthula, „A Human Error Based Approach to Understanding Programmer-Induced Software Vulnerabilities", in *2020 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, Coimbra, Portugal: IEEE, říj. 2020, s. 49–54. doi: 10.1109/ISSREW51248.2020.00036.

[9] T. Nole, „Static and dynamic code analysis: Complementary techniques", *TechTarget*, 10. prosinec 2021.

https://www.techtarget.com/searchsoftwarequality/tip/Static-and-dynamic-code-analysis-Complementary-techniques

[10] T. Avgerinos, S. K. Cha, B. L. T. Hao, a D. Brumley, „AEG: Automatic Exploit Generation".

[11] A. Hanna, H. Z. Ling, X. Yang, a M. Debbabi, „A Synergy between Static and Dynamic Analysis for the Detection of Software Security Vulnerabilities", in *On the Move to Meaningful Internet Systems: OTM 2009*, R. Meersman, T. Dillon, a P. Herrero, Ed., in Lecture Notes in Computer Science, vol. 5871. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, s. 815–832. doi: 10.1007/978-3-642-05151-7_5.

[12] M. Harman a P. O'Hearn, „From Start-ups to Scale-ups: Opportunities and Open Problems for Static and Dynamic Program Analysis", in *2018 IEEE 18th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, Madrid: IEEE, zář. 2018, s. 1–23. doi: 10.1109/SCAM.2018.00009.

[13] T. Clarke, „Fuzzing for software vulnerability discovery", úno. 2009, [Online]. Dostupné z: https://www.ma.rhul.ac.uk/static/techrep/2009/RHUL-MA-2009-04.pdf

[14] A. Gosain a G. Sharma, „A Survey of Dynamic Program Analysis Techniques and Tools", in *Proceedings of the 3rd International Conference on Frontiers of Intelligent Computing: Theory and Applications (FICTA) 2014*, S. C. Satapathy, B. N. Biswal, S. K. Udgata, a J. K. Mandal, Ed., in Advances in Intelligent Systems and Computing, vol. 327. Cham: Springer International Publishing, 2015, s. 113–122. doi: 10.1007/978-3-319-11933-5_13.

[15] M. D. Ernst, „Static and dynamic analysis: synergy and duality".

[16] W. Chang, B. Streiff, a C. Lin, „Efficient and extensible security enforcement using dynamic data flow analysis", in *Proceedings of the 15th ACM conference on Computer and communications security - CCS '08*, Alexandria, Virginia, USA: ACM Press, 2008, s. 39. doi: 10.1145/1455770.1455778.

[17] P. Godefroid, M. Y. Levin, a D. Molnar, „SAGE: Whitebox Fuzzing for Security Testing".

[18] K. Sen, D. Marinov, a G. Agha, „CUTE: A Concolic Unit Testing Engine for C".

[19] T. Chen, X. Zhang, S. Guo, H. Li, a Y. Wu, „State of the art: Dynamic symbolic execution for automated test generation", *Future Gener. Comput. Syst.*, roč. 29, č. 7, s. 1758–1773, zář. 2013, doi: 10.1016/j.future.2012.02.006.

[20] M. Sutton, A. Greene, a P. Amini, *Fuzzing: Brute Force Vulnerability Discovery*. Pearson Education, 2007. [Online]. Dostupné z: https://books.google.cz/books?id=DPAwwn7QDy8C

[21] V. Wüstholz a M. Christakis, „Harvey: A Greybox Fuzzer for Smart Contracts", in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, in ESEC/FSE 2020. New York, NY, USA: Association for Computing Machinery, 2020, s. 1398–1409. doi: 10.1145/3368089.3417064.

[22] V.-T. Pham, M. Bohme, a A. Roychoudhury, „AFLNET: A Greybox Fuzzer for Network Protocols", in *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, Porto, Portugal: IEEE, říj. 2020, s. 460–465. doi: 10.1109/ICST46399.2020.00062.

[23] T. Hamilton, „Fuzz Testing (Fuzzing) Tutorial", *GURU99*, 21. leden 2023. https://www.guru99.com/fuzz-testing.html

[24] „Fuzzing". [Online]. Dostupné z: https://owasp.org/www-community/Fuzzing

[25] P. Garg, „Fuzzing: Mutation vs. generation", *Infosec*, 1. duben 2012. https://resources.infosecinstitute.com/topic/fuzzing-mutation-vs-generation/

[26] „Fuzzing", *OWASP*. https://owasp.org/www-community/Fuzzing

[27] P. Tišnovský, „Úvod do problematiky fuzzingu a fuzz testování", *Root.cz*. https://www.root.cz/clanky/uvod-do-problematiky-fuzzingu-a-fuzz-testovani/

[28] S. Bradshaw, „An introduction to fuzzing: using fuzzers (SPIKE) to find vulnerabilities", *Infosec*, 12. listopad 2010. https://resources.infosecinstitute.com/topic/intro-to-fuzzing/

[29] P. Godefroid, „Fuzzing: hack, art, and science", *Commun. ACM*, roč. 63, č. 2, s. 70–76, led. 2020, doi: 10.1145/3363824.

[30] D. Lukan, „Sulley Fuzzing Framework Intro", *Infosec*, 17. červenec 2010. https://resources.infosecinstitute.com/topic/sulley-fuzzing/

[31] K. Hnízdilová, „Linuxové distribuce pro penetrační testování a forenzní analýzu".

[32] „Quick start guide", *Hypothesis official documentation*. https://hypothesis.readthedocs.io/en/latest/quickstart.html

[33] R. Padhye, „JQF + Zest: Semantic Fuzzing for Java", *github.com*. https://github.com/rohanpadhye/JQF

[34] R. Padhye, C. Lemieux, a K. Sen, „JQF: coverage-guided property-based testing in Java", in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, Beijing China: ACM, čvc. 2019, s. 398–401. doi: 10.1145/3293882.3339002.

[35] „All Kali Tools", *kali.org*. https://www.kali.org/tools/all-tools/

[36] „Black Arch tools", *Black Arch Linux*. https://blackarch.org/fuzzer.html

[37] P. Amini a A. Portnoy, „Sulley: Fuzzing Framework". fuzzing.org. [Online]. Dostupné z: http://www.fuzzing.org/wp-content/SulleyManual.pdf

[38] „Software testing and analysis: process, principles, and techniques", *Choice Rev. Online*, roč. 46, č. 02, s. 46-0935-46–0935, říj. 2008, doi: 10.5860/CHOICE.46-0935.

[39] D. Winkowski, „Over 70% of tech projects fail but yours doesn't have to.", *LEOCODE*, 11. srpen 2021. https://leocode.com/development/over-70-of-tech-projects-fail/

[40] D. M. Owens a D. Khazanchi, „Software Quality Assurance", *Softw. Qual. Assur.*.

[41] Kantata Blog, „Why is the Project Lifecycle Important?", *KANTATA*, 14. červenec 2020. https://www.kantata.com/blog/article/why-is-the-project-lifecycle-important

[42] S. Gray, „Software Development: 7 Phases of Design & Development Process", *Custom Software Lab*. https://www.customsoftwarelab.com/software-development-7-phases-of-design-development-process/

[43] J. McKay, „Software Development Process: How to Pick The Process That's Right For You", *PLANIO*, 2. říjen 2019. https://plan.io/blog/software-development-process/

[44] J. Hall, „Everything You Should Know About Secure Software Development Life Cycle Processes (SDLC)", *CloudEmployee*, 3. únor 2022.

https://cloudemployee.co.uk/blog/code/everything-you-should-know-about-secure-software-development-life-cycle-processes-sdlc

[45] D. Radigan, „Agile vs. waterfall project management", *Atlassian*. https://www.atlassian.com/agile/project-management/project-management-intro

[46] „A Complete Guide to the Waterfall Methodology", *Indeed*, 4. únor 2023. https://www.indeed.com/career-advice/career-development/waterfall-methodology

[47] „Agile Methodology for Software Development", *Amplifyre*, 12. říjen 2022. https://www.amplifyre.com/articles/agile-methodology-for-software-development

[48] M. A. Jamil, M. Arif, N. S. A. Abubakar, a A. Ahmad, „Software Testing Techniques: A Literature Review", in *2016 6th International Conference on Information and Communication Technology for The Muslim World (ICT4M)*, Jakarta, Indonesia: IEEE, lis. 2016, s. 177–182. doi: 10.1109/ICT4M.2016.045.

[49] Y. Wang, P. Jia, L. Liu, C. Huang, a Z. Liu, „A systematic review of fuzzing based on machine learning techniques", *PLOS ONE*, roč. 15, č. 8, s. e0237749, srp. 2020, doi: 10.1371/journal.pone.0237749.

[50] John Iwuozur, „Neural Fuzzing: A Faster Way to Test Software Security", *eSecurity Planet*, 21. srpen 2021. https://www.esecurityplanet.com/applications/neural-fuzzing-software-security-testing/

13 Přílohy

1) Struktura přiloženého souboru UHK-FIM-Hnizdilova-Kristyna.zip
2) Podklad pro zadání diplomové práce

Struktura souboru UHK-FIM-Hnizdilova-Kristyna.zip

/Test 1 A) – obsahuje slovník usernames.txt

/Test 1 B) – obsahuje slovníky amount.txt, bankccounts.txt, varsymbols.txt a výsledek testu outputTransactions.txt

/Test 1 C) – obsahuje slovník passwords.txt

/Test 2 – obsahuje slovník common.txt

/Test 3 – obsahuje slovník common.txt

/Test 4 – obsahuje slovník admin-panels.txt

UNIVERZITA HRADEC KRÁLOVÉ
Fakulta informatiky a managementu
Akademický rok: 2021/2022

Studijní program: Aplikovaná informatika
Forma studia: Prezenční
Obor/kombinace: Aplikovaná informatika (ai2-p)

Podklad pro zadání DIPLOMOVÉ práce studenta

Jméno a příjmení: **Bc. Kristýna Hnízdilová**
Osobní číslo: **I2000039**
Adresa: Wolkerova 1384, Ústí nad Orlicí, 56201 Ústí nad Orlicí 1, Česká republika

Téma práce: Možnosti využití fuzz testingu
Téma práce anglicky: Possibilities of using fuzz testing

Vedoucí práce: Mgr. Josef Horálek, Ph.D.
Katedra informačních technologií

Zásady pro vypracování:

Cílem diplomové práce je na základě podrobné rešerše navrhnout metodiku pro využití fuzz testingu pro testování stability, korektnosti a bezpečnosti aplikací a informačních systémů. V teoretické části bude zpracována podrobná rešerše v oblasti fuzzingu, fuzz testingu a quality assurance. Na základě této rešerše bude provedena analýza možností využití jednotlivých typů tohoto testování a zmapovány jednotlivé nástroje pro fuzz testování. Dále bude navržena metodika testování za využití fuzz testingu, která bude ověřena na základě sadou navržených testů a výstupních reportů pro vybrané projekty.
Osnova:

1. Úvod
2. Principy softwarové bezpečnosti
3. Úvod to fuzz testingu (rešerše)
4. Typy testů (fuzzerů)
5. Analýza dostupných nástrojů
6. Fuzzing a quality assurence
7. Návrh metodiky
8. Návrh testů a reportů pro ověření metodiky
9. SWOT analýza fuzz testingu
10. Závěr

Seznam doporučené literatury:

Takanen, A.; DeMott, J.; Miller, C.: Fuzzing for Software Security Testing and Quality Assurance. Artech House, 2008, ISBN 978-1-59693-214-2.
Fuzzing for Software Security Testing and Quality Assurance. 2. Norwood, United States: Artech House Publishers, 2018. ISBN 9781608078509.

Podpis studenta: Datum:

Podpis vedoucího práce: Datum: