VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INTELIGENTNÍCH SYSTÉMŮ
DEPARTMENT OF INTELLIGENT SYSTEMS

BEZPEČNOST SLUŽBY TESTING FARM

SECURITY OF TESTING FARM SERVICE

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — **Bc. JAN HAVLÍN**
AUTHOR

VEDOUCÍ PRÁCE — **Mgr. JOZEF DRGA**
SUPERVISOR

BRNO 2022

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav inteligentních systémů (UITS) Akademický rok 2021/2022

Zadání diplomové práce

Student:	**Havlín Jan, Bc.**
Program:	Informační technologie a umělá inteligence
Specializace:	Kybernetická bezpečnost
Název:	**Bezpečnost služby Testing Farm** **Security of Testing Farm Service**
Kategorie:	Bezpečnost

Zadání:

1. Seznamte se se službou Testing Farm a jejími jednotlivými částmi.
2. Seznamte se s problematikou bezpečnosti síťových služeb a algoritmickými hrozbami, zejména neoprávněnému využití HW, a existujících řešení pro jejich ochranu.
3. Vytvořte threat model pro službu Testing Farm, zmapujte možnosti zneužití služby, a to především autorizovanými uživateli.
4. Navrhněte způsob detekce zneužití služby a navrhněte software pro identifikaci přítomnosti útoku a omezení jeho dopadu.
5. Implementujte preventivní a omezující bezpečnostní opatření proti zneužití služby Testing Farm.
6. Testujte software v reálném provozu a vyhodnoťte přínos implementovaného řešení.

Literatura:

- SHOSTACK, Adam. Threat Modeling: Designing for Security. Wiley, 2014. ISBN 978-1-118-80999-0
- Dokumentace k projektu Packit: https://packit.dev/docs/testing-farm/
- Dokumentace k projektu TMT: https://tmt.readthedocs.io/en/stable/
- OWASP top 10 zranitelností ve webových aplikacích: https://owasp.org/www-project-top-ten/

Při obhajobě semestrální části projektu je požadováno:

- Splněné body 1 až 3.

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

Vedoucí práce:	**Drga Jozef, Mgr.**
Konzultant:	Vadkerti Miroslav, RedHatCZ
Vedoucí ústavu:	Hanáček Petr, doc. Dr. Ing.
Datum zadání:	1. listopadu 2021
Datum odevzdání:	18. května 2022
Datum schválení:	3. listopadu 2021

Abstrakt

Práce se zabývá bezpečností služby Testing Farm ve firmě Red Hat. Konkrétně jde o možnost neoprávněného využití testovacích strojů k jiným účelům, než jsou určené. Potřeba pro implementaci bezpečnostního systému pramení z toho, že uživatelé této služby mohou na testovacích strojích spouštět libovolný kód jako uživatel root.

V implementační části práce byl vytvořen monitorovací agent, kterého se podařilo nasadit na testovací stroje v produkčním prostředí služby. Tento systém sleduje přenášené síťové pakety, systémové zdroje a konfiguraci. Na základě těchto pozorování vytváří metriky o chování systému, které odesílá na monitorovací server Prometheus.

Abstract

This thesis deals with security of Testing Farm Service in Red Hat company. Specifically, it is about unauthorized usage of testing machines for purposes which are not allowed. The need for implementing security measures comes from the fact that users are allowed to run arbitrary code on test machines as the user root.

In the implementation part of the thesis, a monitoring agent was created and deployed to the testing machines of the production environment of the service. This system watches transmitted packets, system resources and configuration. Based on these observations, it creates metrics about the system behavior and sends them over to monitoring server Prometheus.

Klíčová slova

Bezpečnost, Red Hat, eBPF, OWASP, Testing Farm, IDS, IPS, Go, cilium, bpf2go, skenování sítě, Prometheus

Keywords

Security, Red Hat, eBPF, OWASP, Testing Farm, IDS, IPS, Go, cilium, bpf2go, network scan, Prometheus

Citace

HAVLÍN, Jan. *Bezpečnost služby Testing Farm.* Brno, 2022. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Mgr. Jozef Drga

Bezpečnost služby Testing Farm

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana Mgr. Jozefa Drgy. Další informace mi poskytl technický konzultant pan Mgr. Miroslav Vadkerti. Uvedl jsem všechny literární prameny, publikace a další zdroje, ze kterých jsem čerpal.

.....
Jan Havlín
18. května 2022

Poděkování

Tímto bych chtěl poděkovat vedoucímu práce panu Mgr. Jozefu Drgovi za vedení práce, technickému konzultantovi panu Mgr. Miroslavu Vadkertimu za cenné rady při řešení této práce a svojí přítelkyni za podporu ve studiu.

Obsah

Kapitola 1

Úvod

Tato diplomová práce si klade za cíl zvýšit zabezpečení open-source testovací služby Testing Farm, a to konkrétně vytvořením systému pro detekci průniku, který by odhaloval přítomnost nežádoucí aktivity na strojích určených k vykonávání automatizovaných testů a zároveň zamezil útočníkovi v pokračování činnosti. Zadání práce vzniklo ve spolupráci s firmou Red Hat Czech, s.r.o.

Služba Testing Farm vznikla za účelem testování operačního systému. Uživatelům, kteří jsou oprávněni službu využívat, je umožněno spouštět libovolný kód na testovacích strojích. Toto je v současnosti akceptovatelné riziko, neboť všichni současní uživatelé služby jsou současně důvěryhodní zaměstnanci firmy Red Hat. Potřeba pro nasazení tohoto zabezpečovacího mechanismu pramení z důvodu plánovaného zveřejnění služby mezi širokou komunitu vývojářů distribuce Fedora, čímž by se mohla služba vystavit riziku neoprávněného užití výpočetních prostředků.

Práce je členěna do 7 kapitol, kde kapitola 2, následující po úvodu, se věnuje obecnému popisu principu fungování a architektuře služby Testing Farm, dále kapitola 3 uvádí do problematiky bezpečnosti software. Kapitola 4 vytváří threat model Testing Farm, tedy detailněji popisuje to, jakým bezpečnostním rizikům by služba mohla být vystavena. Kapitola 5 představuje návrh systému pro zlepšení bezpečnosti služby, kapitola 6 popisuje, jakým způsobem se systém implementoval. Konečně, kapitola 7 se věnuje nasazení tohoto bezpečnostního systému do produkčního prostředí služby a také testuje a vyhodnocuje přínos realizovaného řešení.

Kapitola 2

Služba Testing Farm

Tato kapitola se věnuje systému, který je předmětem této práce a na nějž bude implementováno bezpečnostní rozšíření. Následující sekce se věnují obecnému principu fungování služby a její architektuře.

2.1 Popis služby

Testing Farm je software s otevřeným zdrojovým kódem vydaný pod licencí Apache 2.0, který je současně nasazený v modelu software jako služba (angl. *Software as a Service*). V současnosti je jejím primárním využitím testování částí operačního systému GNU/Linux, a to konkrétně distribucí Red Hat Enterprise Linux, Fedora a CentOS. Nicméně lze tuto službu využít i pro generické testování libovolného softwaru. Princip fungování Testing Farm lze přirovnat k *Compile Farms*, ovšem zde je hlavním cílem služby vykonávat automatizované testy. Vstupním bodem služby je programové aplikační rozhraní založené na REST API. Testing Farm se rozléhá napříč několika infrastrukturami, včetně veřejných i neveřejných cloudů.

Za přibližně rok trvající provoz se stihlo vykonat v průměru 30 000 testovacích požadavků za měsíc v součtu pro všechny uživatele. Funkcionálně služba vykonala testy s 454 různými balíčky distribuce Fedora a 325 balíčky distribuce RHEL. Testovalo se 38 subsystémů distribuce RHEL a 34 projektů na serveru GitHub.

Není od věci také zmínit, že jako předchůdce služby Testing Farm lze považovat z dnešního pohledu spíše již legacy systém *BaseOS-CI*, který se v Red Hatu interně stále používá k průběžné integraci operačního systému RHEL. Tento systém s Testing Farm stále sdílí několik komponent. Tímto legacy systémem se zabývá například diplomová práce Ing. Martina Klusoně „Podpora průběžné integrace v rámci systému Copr“[16], která BaseOS-CI rozšířila o nové testovací vstupy. Tato práce si klade podobný cíl – vytvořit rozšíření projektu Testing Farm, ovšem v tomto případě se jedná o takový příspěvek, který by pomohl k bezpečnějšímu chodu služby.

Testing Farm lze popsat jako back-end k systémům průběžné integrace, na které se deleguje spouštění automatizovaných testů. Typickým uživatelem služby jsou tak vývojáři komponent operačního systému, kteří si přejí automaticky spouštět integrační testy například při vytvoření nového commitu na serveru verzovacího systému. Uživatelům služba přináší prospěch tím, že jim odpadá starost o udržování testovací infrastruktury, neboť testy mají obvykle specifické požadavky pro jejich běh, a to například konkrétní distribuci operačního systému či architekturu procesoru. Stačí jim tak na vstup Testing Farm zadat

definici testu, který chtějí vykonat a popis testovací prostředí, v kterém se bude tento test spouštět. O zbytek se následně postará služba, která uživatelům po dokončení testu předá výsledek.

2.2 Popis testovacích případů

Definici testovacích případů pro vykonání lze službě předat v podobě dvou různých standardů pro popis testů, a to *Flexible Metadata Format* a *Standard Test Interface*.

2.2.1 Flexible Metadata Format

Jak již zaznělo v úvodu sekce, jedná se o formát pro popis testovacích případů[26]. S tímto formátem se úzce váže nástroj TMT[27] (*Test Management Tool*), který bývá v kontextu Testing Farm často zaměňován s pojmem FMF.

Test Management Tool vytvořila společnost Red Hat a vydala pod MIT licencí. Nástroj poskytuje uživatelsky přívětivý způsob pro práci s testy ve formátu FMF. Pomocí nástroje lze vytvářet nové testy, bezpečně a jednoduše je spouštět v různých prostředích, prohlížet si výsledky testování a spouštění testů průběžné integrace pomocí konzistentní konfigurace. Nástrojem TMT se zabývala také diplomová práce Ing. Ondreje Dubaje „Systém pro správu výsledků testů doplňující nástroj tmt“[13].

Specifikace FMF definuje několik úrovní metadat:

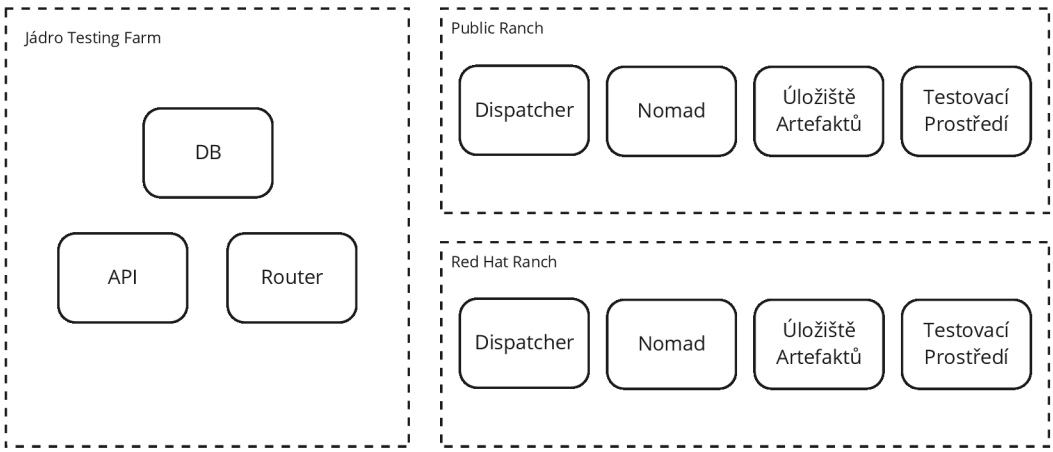
- *Core* vlastnosti, jako například shrnutí nebo popis, jsou společné napříč všemi testy. Tyto atributy se označují jako úroveň L0.
- *Tests*, označovány jako L1 metadata, definují atributy, které blízce souvisí s jednotlivými testovacími případy jako například testovací skript a framework, cesta k pracovnímu adresáři pro spouštění testů, maximální možná doba běhu testu nebo závislosti potřebné k vykonání testu.
- *Plans*, které odpovídají metadatům úrovně L2, se používají k seskupování souvisejících testů a umožňují jejich spouštění v průběžné integraci. Popisují, kde hledat testy pro vykonávání, jak získat testovací prostředí, jak toto prostředí připravit k testování, jak vykonat testy a jak vytvořit výsledný report z testování.
- *Stories* implementují L3 metadata a lze je využít k sledování pokrytí implementace, testů a dokumentace pro jednotlivé vlastnosti nebo požadavky. Pomocí stories lze jednoduše sledovat různé vlastnosti, včetně postupu implementace, sdružené do jednoho místa.

2.2.2 Standard Test Interface

V současnosti se spíše jedná o legacy formát pro popis testů, nicméně se stále využívá v integračním testování RPM balíčků pro Fedora Linux, proto jej Testing Farm stále podporuje. Standard Test Interface vymezuje rozhraní mezi testovací sadou a Continuous Integration serverem, na kterém se vykonávají. Tento formát se příliš nevyužívá a do budoucna se spíše plánuje úplný přechod na formát FMF[5].

2.3 Architektura služby Testing Farm

Tato sekce popisuje všechny významné komponenty tvořící službu Testing Farm, včetně jejich vzájemných vztahů. Architekturu lze logicky rozdělit do dvou částí zvaných *Jádro* a *Ranče*. Toto rozdělení zobrazuje obrázek 2.1.



Obrázek 2.1: Architektura služby Testing Farm.

2.3.1 Jádro

V části označené jako Jádro se nacházejí následující komponenty:

- REST API – tvoří vstupní bod do služby Testing Farm. Pomocí API lze vytvářet nové testovací požadavky a dotazovat se na jejich stav.
- Databáze – ukládá záznamy o uživatelích a jejich testovacích požadavcích.
- Router – směruje nově příchozí požadavky na jejich odpovídající ranče ke zpracování.

REST API

Aplikační programové rozhraní založené na protokolu HTTP vytváří uživatelům vstupní bod do služby Testing Farm. Kompletní popis rozhraní se nachází na adrese `https://testing-farm.gitlab.io/api/`. Implementace API vznikla v programovacím jazyce Python s použitím frameworku FastApi[1].

V současné verzi API 0.1 jsou k dispozici následující vstupní body:

- `/requests/{ID}` (metoda *GET*) – endpoint pro získání informací o existujícím testovacím požadavku, kde hodnota `{ID}` odpovídá unikátnímu identifikátoru daného testovacího požadavku. Odpověď pak obsahuje informace jako je stav požadavku či výsledek testu.
- `/requests` (metoda *POST*) – endpoint pro vytvoření nového testovacího požadavku. Mezi povinné parametry, které je nutno dodat, patří tajný API klíč uživatele, definice testu ve formátu TMT nebo STI a definice testovacího prostředí (distribuce operačního systému a architektura procesoru),

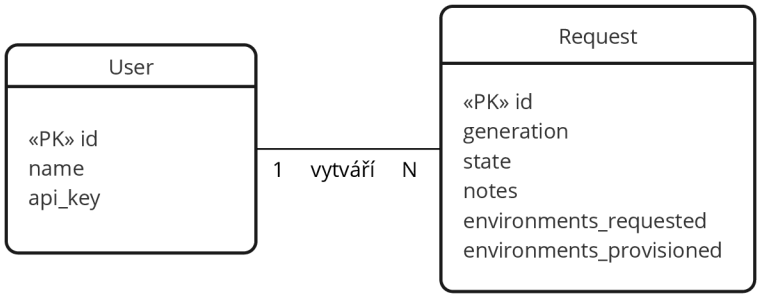
[1] `https://fastapi.tiangolo.com/`

- `/composes` (metoda *GET*) – endpoint pro získání seznamu dostupných distribucí operačního systému a architektur, na kterých lze vykonávat testovací požadavky.
- `/about` (metoda *GET*) – endpoint pro získání informací o službě.

Databáze

Jako databázový systém je nasazena distribuovaná SQL databáze *CockroachDB*[2]. Mezi její přednosti patří vysoká stabilita, škálovatelnost a rychlost.

V databázi se nacházejí 2 tabulky, a to *User*, ukládající uživatele služby Testing Farm a jejich testovací požadavky v tabulce *Request*, viz ER diagram na obrázku 2.2.



Obrázek 2.2: ER diagram databáze.

Za zmínku stojí položka *state* v tabulce *Request*, která reprezentuje stav testovacího požadavku a která může nabývat jedné z následujících hodnot:

- *NEW* – implicitní stav při vytvoření nového požadavku,
- *QUEUED* – požadavek, který dispatcher přijal a zařadil do fronty,
- *RUNNING* – aktuálně vykonávající se test,
- *COMPLETE* – dokončený test,
- *ERROR* – neúspěšně zpracovaný testovací požadavek.

Router

Router je komponenta, která zpracovává nově vytvořené požadavky (požadavky ve stavu *NEW*). Jeho úlohou je každý tento nový požadavek klasifikovat a správně nasměrovat do odpovídajícího ranče, který je určený pro jeho zpracování. Tohoto docílí tím, že každému požadavku přidělí značku, na základě které lze rozhodnout, do jakého ranče patří.

Uživatelské rozhraní

V současnosti služba nedisponuje plnohodnotným uživatelským rozhraním, zlepšení této části je tak plánováno do budoucna. Mimo samotné REST API však v současnosti existuje několik možností, kterými může uživatel komunikovat se službou pohodlnějším způsobem. Výčet některých těchto variant je následující:

[2] https://www.cockroachlabs.com/

- CLI[3] – pro snazší práci s API existuje utilita s rozhraním v příkazovém řádku. Pomocí tohoto nástroje lze vytvářet testovací požadavky, dotazovat se na jejich stav a získávat jejich výsledky.
- Webový server s artefakty[4] – server se základním uživatelským rozhraním, jež umožňuje si prohlížet jednotlivé testovací požadavky včetně artefaktů vytvořených během testování.
- GitHub Action[5] – na serveru GitHub lze automatizovaně spouštět testovací požadavky, například při provedení operace *push* do repozitáře.

2.3.2 Ranče

Jak již bylo zmíněno v předchozích kapitolách, každý testovací požadavek se vykonává v tzv. Ranči. Tato architektura poskytuje službě určitou flexibilitu v tom, kde se bude daný test vykonávat. V současnosti existuje soukromý Ranč (*Red Hat Ranch*), který je přístupný pouze v interní síti Red Hatu a využívá se např. k testování distribuce RHEL. Naopak veřejný Ranč (*Public Ranch*) je dostupný ve veřejném internetu a používá se k testování programů s otevřeným zdrojovým kódem.

Testovací požadavky se na odpovídající ranč směrují podle tajného API klíče uživatele. Každý API klíč je pevně přiřazen k jednomu ranči.

Rozdíly mezi těmito dvěma nasazeními spočívají také v různých testovacích prostředích, jež poskytují k vykonávání testů. Public Ranch umožňuje testovat pouze na virtuálních strojích v cloudu AWS s architekturami *aarch64* a *x86_64*. Naproti tomu Red Hat Ranch poskytuje architekturu *x86_64* v platformách OpenStack a AWS, dále ve službě Beaker je umožněno testovat na *aarch64*, *ppc64le* a *s390x*.

Každý ranč tvoří následující komponenty:

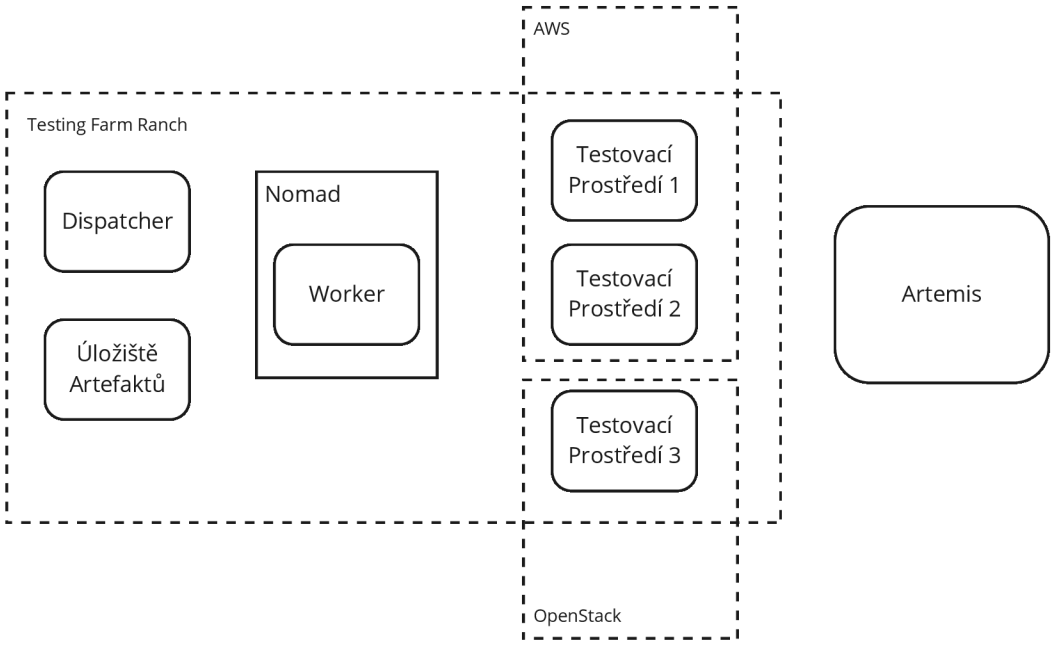
- Dispatcher – přijímá testovací požadavky cílené pro svůj ranč, které odesílá dál na Nomad pro jejich vykonání.
- Nomad – fronta postupně zpracovávaných testovacích požadavků. Proces zpracování zahrnuje získání testovacího stroje, spuštění testů, archivaci výsledků a zahození stroje.
- Testovací prostředí – (virtuální) stroje, na nichž se provádí testování.
- Úložiště artefaktů – webový server s výsledky testovacích úloh.

Pohled na jednotlivé části zobrazuje obrázek 2.3. Počet jednotlivých testovacích prostředí a jejich příslušnost ke cloudovým platformám je pouze orientační.

Dispatcher

Cíl této komponenty spočívá ve zpracování všech nově vytvořených požadavků (nacházejících se ve stavu *NEW*), které jsou určené pro tento ranč. Tyto požadavky rozeznává na základě značky v požadavku, kterou nastavil Router. Následně tyto požadavky odesílá ke zpracování do fronty na serveru Nomad.

[3] https://gitlab.com/testing-farm/cli
[4] https://artifacts.dev.testing-farm.io/
[5] https://github.com/sclorg/testing-farm-as-github-action



Obrázek 2.3: Architektura ranče.

Nomad

Software Nomad[6] od společnosti HashiCorp je open-source flexibilní plánovač pro orchestraci kontejnerů a vykonávání úloh. Ve službě Testing Farm se Nomad používá k provádění kroků potřebných k vykonání každé testovací úlohy. Obrázek 2.4 zobrazuje frontu na webovém rozhraní této komponenty. Tento proces zahrnuje následující:

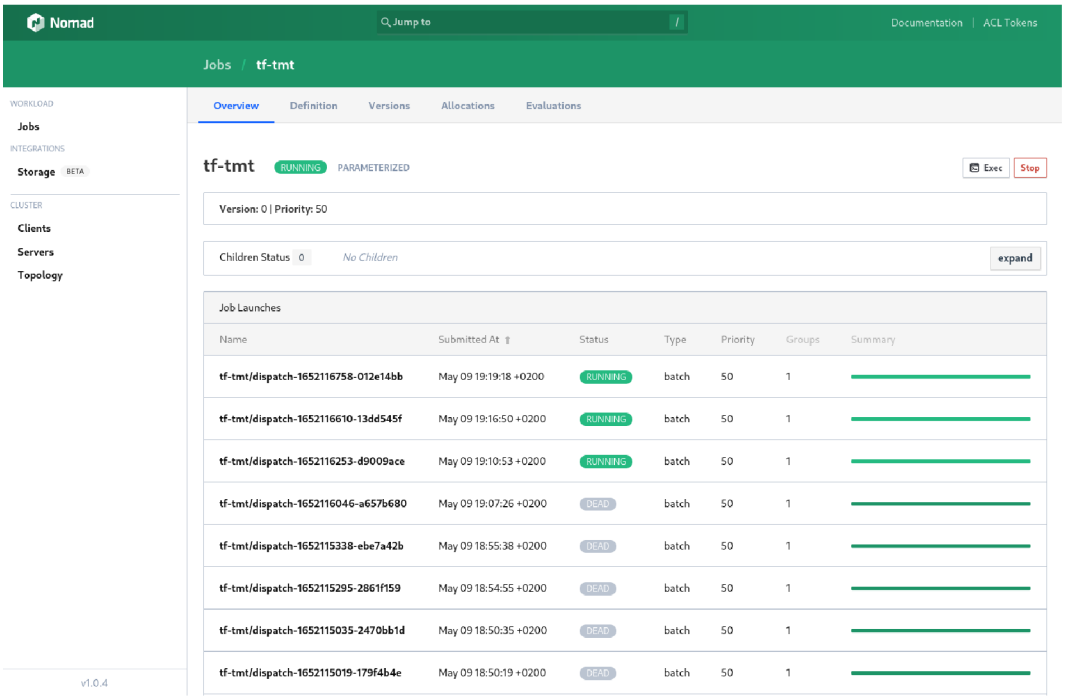
- Získání testovacího virtuálního stroje – využívá se služba Artemis, které se věnuje sekce 2.4.1.
- Zjištění testovacích plánů nástroje TMT.
- Instalace testovacích artefaktů na stroje – využívá se nástroj Ansible[7] pro automatizovanou instalaci na testovací stroj.
- Spuštění testování.
- Zpracování výsledků testování a uklizení testovacího stroje.

Výše uvedené jednotlivé kroky jsou implementované v modulech frameworku gluetool, kterému se věnuje sekce 2.4.2.

Testovací prostředí

Jak již bylo zmíněno, podobu testovacích prostředí pro vykonávání testů uživatele tvoří virtuální stroje. Pro každý testovací požadavek se dynamicky vytvoří a zahodí čistý stroj s vybranou distribucí operačního systému, kterou si uživatel zvolil. Pro poskytování těchto

[6] https://www.nomadproject.io/
[7] https://www.ansible.com/



Obrázek 2.4: Webové rozhraní Nomad zobrazující frontu testovacích požadavků ve specifikaci TMT.

strojů se využívá několik cloudových infrastruktur, v současnosti se používají platformy AWS a OpenStack. Jako abstrakce nad cloudy se využívá služba *Artemis*, která poskytuje jednotné rozhraní pro získávání těchto strojů. Celý tento proces – od požadavku na vytvoření stroje a instalaci testovacích artefaktů až po vykonání testů a archivaci výsledků orchestruje výše uvedený Nomad prostřednictvím frameworku *Gluetool*.

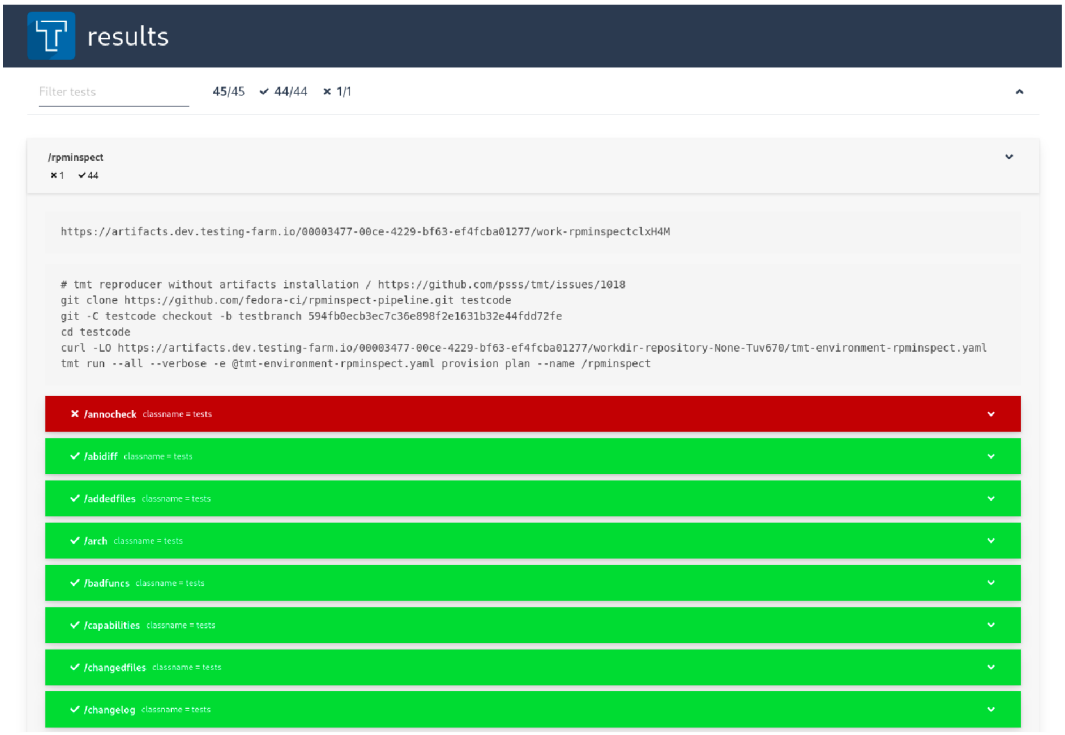
Vhodné je také zmínit, že na testovacích strojích může uživatel spouštět libovolný kód jako privilegovaný uživatel `root`. Právě obava ze zneužití této skutečnosti přispěla ke vzniku této práce.

Úložiště artefaktů

Úložiště artefaktů je webový server, na který se v průběhu vykonávání testovací úlohy ukládají informace o průběhu testování. Úložiště je pro veřejný ranč dostupné na adrese `https://artifacts.dev.testing-farm.io/`. Webový server disponuje jednoduchým uživatelským rozhraním, které se nachází na obrázku 2.5.

Shrnutí

Na závěr této části je vhodné ilustrovat komunikaci jednotlivých komponent. Tu vystihuje obrázek 2.6 se sekvenčním diagramem zobrazující vykonání nového testovacího požadavku v daném ranči.



Obrázek 2.5: Webové rozhraní úložiště artefaktů zobrazující výsledky testovacího požadavku.

2.4 Související služby a frameworky

Předchozí sekce popsala klíčové komponenty Testing Farm, nicméně fungování služby stojí na několika dalších prvcích, které lze považovat za natolik generické, že si zaslouží zařadit samostatně. Konkrétně byla zmíněna služba Artemis a framework Gluetool. Právě těmito dvěma nástroji se tato část zabývá.

2.4.1 Artemis

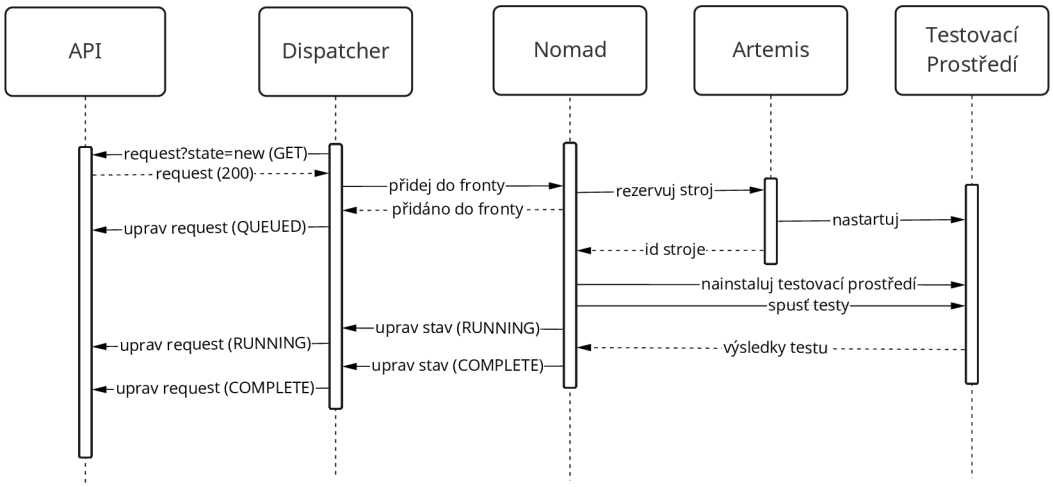
Artemis je samostatná webová služba, která prostřednictvím svého REST API umožňuje poskytování strojů z různých cloudových infrastruktur. Cílem Artemis je nabízet uživatelům abstrakci nad několika existujícími platformami pro poskytování (virtuálních) strojů, jako jsou například OpenStack či AWS.

Artemis je open-source software vyvíjený společností Red Hat a vydaný pod licencí Apache 2.0. Zdrojový kód Artemis lze nalézt na serveru GitLab[8].

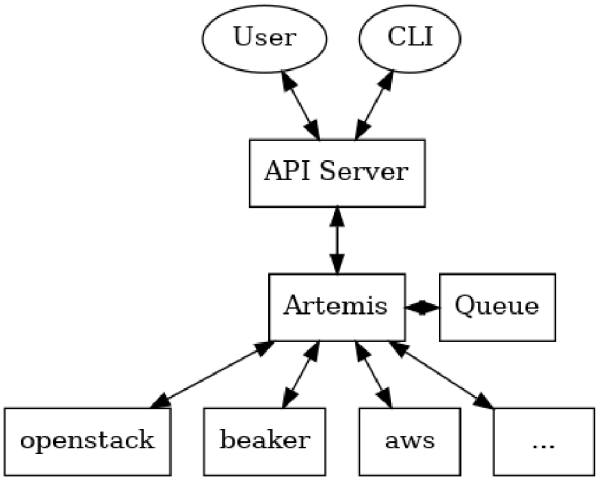
Mezi další funkce lze také zařadit například monitorování a zotavování při chybě v infrastruktuře. Při výpadku jedné cloudové platformy umožňuje přestup na jinou infrastrukturu dle nastavených podmínek. Potřeba pro vznik takovéto služby pramenila z boje s nestabilní infrastrukturou pro poskytování strojů OpenStack. Časté výpadky celkově zpomalovaly testovací proces velkého množství testovacích požadavků.

Z vysokoúrovňového pohledu, jak zobrazuje obrázek 2.7, komponenty Artemis tvoří server s aplikačním rozhraním, fronta požadavků a jednotlivé infrastruktury, z nichž je možné poskytovat jednotlivé (virtuální) stroje.

[8] https://gitlab.com/testing-farm/artemis



Obrázek 2.6: Sekvenční diagram zpracování nového testovacího požadavku v části Ranč.



Obrázek 2.7: Vysokoúrovňové schéma služby Artemis.

Aplikační rozhraní

Na nejnižší úrovni jako vstupní bod pro komunikaci se službou existuje REST API. Je možné používat toto rozhraní napřímo například pomocí `curl`. Nicméně pro pohodlnější práci se službou vzniklo několik nástrojů.

Doporučeným nástrojem ke komunikaci se službou je nástroj s rozhraním v příkazové řádce *artemis-cli*, který lze po nakonfigurování použít s daným nasazením Artemis. Druhou možnost, kterou využívá služba Testing Farm při vykonávání svých testovacích úloh, poskytuje nástroj *gluetool* s použitím modulu *artemis*. Tato varianta je naopak vhodná pro skriptování, proto ji využívá Testing Farm ve svých automatizovaných testovacích požadavcích. Oba tyto přístupy ukazuje výpis 2.1.

```
artemis-cli guest create --keyname ci-key --arch x86_64 \
    --compose Fedora-Rawhide

gluetool artemis --provision 1 --arch x86_64 --compose Fedora-Rawhide
```

Výpis 2.1: Ukázka vytvoření nového stroje pomocí nástrojů *artemis-cli* a *gluetool*.

2.4.2 Gluetool

Gluetool je generický open-source framework napsaný v jazyce Python cílený do prostředí Linuxového shellu. Jeho zdrojový kód je volně přístupný na serveru GitLab[9]. Framework vyvinula společnost Red Hat a vydala pod licencí Apache 2.0. Gluetool umožňuje konstruování tzv. pipeline v příkazovém řádku pomocí dílčích modulů, v kterých je implementována business logika dané aplikace.

Moduly lze charakterizovat jako logicky oddělené celky obsahující funkcionalitu a jejichž poskládáním za sebe vznikne pipeline, která vykonává nějaký větší celek. Takto navržená architektura založená na modulech umožňuje snadnou změnu funkcionality pipeline tím, že se zamění složení modulů, za předpokladu, že bude dodržena jejich kompatibilita.

Gluetool je v současné době nejvíce používán v Red Hatu pro vykonávání požadavků Testing Farm a pro průběžnou integraci operačního systému, která byla také hlavním motivátorem pro vznik tohoto frameworku. V průběžné integraci se v různých testovacích případech vyskytují drobné odlišnosti, které lze jednoduše obsloužit výběrem vhodného modulu.

Moduly

Jak již naznačil úvod do frameworku, základními stavebními kameny frameworku jsou moduly. Všechny moduly dědí z třídy `gluetool.glue.Module`, která definuje jejich standardní rozhraní. Mezi klíčové vlastnosti, které každý modul nabízí, patří:

- Sdílené funkce – jedná se o hlavní způsob, kterým si moduly předávají informace. Každý modul v sobě může definovat funkce, které poskytne ostatním modulům tím, že jejich názvy uloží do seznamu `shared_functions`. Ostatní moduly pak mohou tuto sdílenou funkci z cizího modulu volat pomocí metody `self.shared('NAME')`. Pokud několik modulů definuje sdílenou funkci se stejným jménem, použije se ta nejpozději definovaná.

- Konfigurace modulů – každému modulu lze přidat parametry, které mohou být dodány prostřednictvím příkazového řádku shellu nebo pomocí textových souborů, které se hodí pro uchování konfigurace neměnného charakteru. Výchozí adresář pro hledání konfiguračních souborů se nachází v `~/.gluetool.d/config`.

- Logování – každý modul umožňuje logování informací unifikovaným způsobem. Dostupné jsou tři úrovně logování, a to v několika úrovních (*debug*, *info*, *warn*, *error*). V případě vyvolání neočekávané výjimky umožňuje logger zaslání informací do služby Sentry.

[9] https://gitlab.com/testing-farm/gluetool/

Příklad jednoduché pipeline

Uvažme soubor `modules.py`, jež definuje tři moduly s názvy `weekday`, `time` a `hello`. Obsah tohoto souboru ukazuje výpis 2.2.

```
import gluetool
import datetime

class Weekday(gluetool.glue.Module):
    name = 'weekday'
    shared_functions = ['message']

    def message(self):
        return 'Today is {}.'.format(datetime.datetime.now().strftime('%A'))

class Time(gluetool.glue.Module):
    name = 'time'
    shared_functions = ['message']

    def message(self):
        return 'It is {}.'.format(datetime.datetime.now().strftime('%H:%M'))

class Hello(gluetool.glue.Module):
    name = 'hello'

    def execute(self):
        self.require_shared('message')
        self.info('Hello! {}'.format(self.shared('message')))
```

Výpis 2.2: Ukázka jednoduchých gluetool modulů.

Moduly `weekday` a `time` definují sdílené funkce se stejným pojmenováním `message`, které vrací řetězec. Funkci s tímto názvem pak ve své metodě `execute()` vyžaduje modul `hello`, který ji zavolá a vytiskne její obsah. Tuto pipeline lze spustit například s moduly `weekday` a `hello` následovně (přepínač `gluetool --module-path .` určuje, že se moduly budou hledat v aktuálním adresáři):

```
$ gluetool --module-path . weekday hello
[20:50:53] [+] [hello] Hello! Today is Thursday.
```

Pokud se však zamění modul `weekday` za `time`, změní se obsah zprávy, kterou tiskne modul `hello`:

```
$ gluetool --module-path . time hello
[20:51:27] [+] [hello] Hello! It is 20:51.
```

Cíl této ukázky spočíval v demonstraci toho, jak lze měnit funkcionalitu pipeline záměnou modulů na úrovni příkazové řádky.

Využití gluetool v Testing Farm

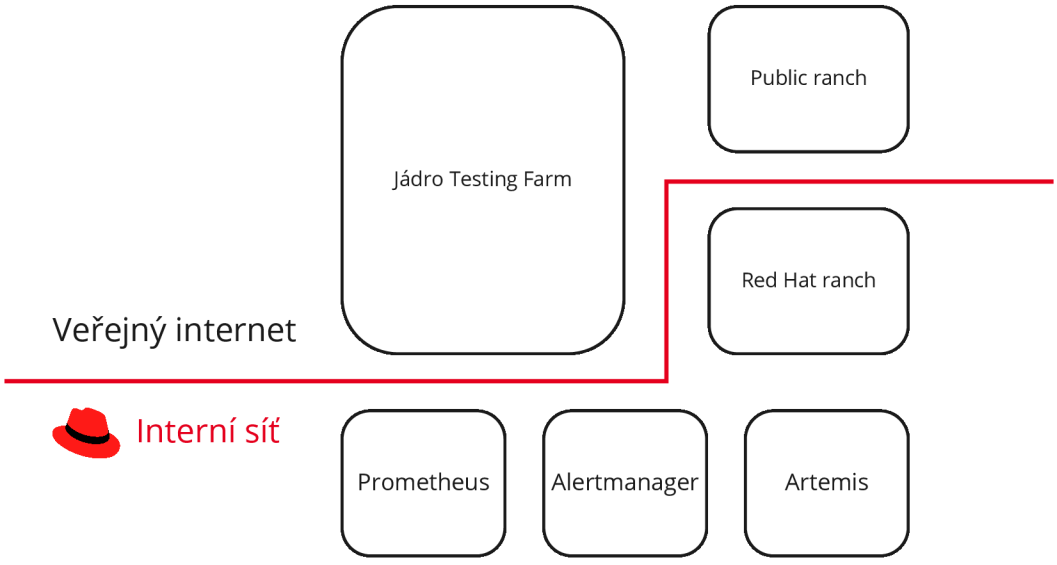
Jak již bylo zmíněno v předchozí sekci, framework se používá v systému Nomad pro orchestraci procesu testování. Pro tento účel vznikla celá řada open-source modulů, jež jsou také k dispozici na veřejném serveru GitLab[10] .

Mezi klíčové moduly, které Testing Farm využívá, patří:

- `artemis` – modul pro získání (virtuálního) testovacího stroje,
- `test-schedule-tmt` – modul pro plánování a spouštění testů na testovacích strojích ve specifikaci TMT,
- `test-schedule-runner-sti` a `test-scheduler-sti` – moduly pro plánování a spouštění STI testů na testovacích strojích,
- `testing-farm-request` – modul pro práci s programovým aplikačním rozhraním služby Testing Farm.

2.5 Monitorování služby

Mezi neoddělitelné součásti provozování každé webové služby patří také její monitorování. K tomuto účelu se používá platforma Prometheus společně s upozorňovacím subsystémem Alertmanager. Obrázek 2.8 zobrazuje nasazení tohoto monitorovacího systému. Také lze poukázat na to, že obrázek ukazuje, které komponenty Testing Farm se nacházejí v interní síti Red Hatu a které ve veřejném internetu.



Obrázek 2.8: Zasazení nástrojů Prometheus a Alertmanager do kontextu služby Testing Farm.

[10] https://gitlab.com/testing-farm/gluetool-modules/

2.5.1 Prometheus

Prometheus je open-source monitorovací systém. Tento server je nasazen na Kubernetes clusteru v interní síti Red Hatu. Mezi jeho klíčovou vlastnost patří to, že získává metriky, které ukládá společně s jejich časovou značkou. Metriky jsou číselná měření, která byly naměřena v průběhu nějaké trvající doby.

Systém Prometheus se skládá z několika dílčích částí:

- Hlavní Prometheus server, který stahuje a ukládá do databáze metriky ze zdrojů, které je vystavují.
- Klientské knihovny pro psaní aplikací komunikujících se serverem Prometheus.
- Systém Alertmanager pro správu upozornění.

Prometheus podporuje následující typy metrik:

- *Counter* – akumulační metrika reprezentující jedno monotónně zvyšující se počítadlo.
- *Gauge* – metrika reprezentující číselnou hodnotu, která se může libovolně zvyšovat a snižovat.
- *Histogram* – komplexnější metrika, jež vzorkuje pozorování a rozděluje je do konfigurovatelných intervalů.
- *Summary* – podobně jako histogram, *summary* vzorkuje pozorování a rozděluje je do konfigurovatelných intervalů. Dále také počítá konfigurovatelné kvantily s klouzavým časovým oknem.

2.5.2 Alertmanager

Alertmanager zpracovává upozornění, která vytvoří klientské aplikace jako například Prometheus server. Systém zajišťuje deduplikaci, seskupení a správné směrování upozornění na komunikační kanály jako například email. Alertmanager je stejně jako Prometheus nasazen na Kubernetes clusteru v interní síti.

V současnosti je Alertmanager zintegrován s IRC kanálem týmu Testing Farm, kam zasílá upozorňující zprávy při splnění nastavených pravidel. Pravidla jsou různorodého charakteru napříč softwarovými službami, které Testing Farm tým spravuje. Mezi tato pravidla patří například příliš dlouhá fronta testovacích požadavků nebo pokud sondy živosti služeb detekovaly nějaký výpadek.

Kapitola 3

Bezpečnost v informačních technologiích

Tato kapitola se zabývá základy bezpečnosti v informačních technologiích. Ve svém úvodu popisuje základní pojmy z bezpečnosti. Dále se kapitola zabývá častými bezpečnostními chybami ve webových aplikacích dle OWASP. Popsány jsou také systémy pro detekci a prevenci průniku. V závěru kapitoly je uvedena technologie eBPF, jež nachází využití, mimo jiné, v bezpečnostních aplikacích.

3.1 Úvod do informační bezpečnosti

V úvodu do informační bezpečnosti je vhodné si ujasnit základní pojmy z této oblasti. Informace v této sekci jsem čerpal z [10] a [14].

Počítačová bezpečnost stojí na třech základních konceptech. Prvním z nich je *důvěrnost*. Ta vyjadřuje utajení informací nebo zdrojů. Příkladem systému, jenž implementuje důvěrnost, je systémy pro řízení přístupu.

Dalším klíčovým pojmem je *integrita*. Ta zajišťuje důvěryhodnost informací a obvykle se vysvětluje jako ochrana dat před jejich neoprávněnou změnou. Mechanismy pro zajištění integrity se dělí do dvou tříd, a to mechanismy pro *prevenci* porušení integrity a mechanismy pro *detekci* porušení integrity.

Poslední bezpečnostní cíl v této sekci je *dostupnost*. Dostupnost úzce souvisí se spolehlivostí a vyjadřuje možnost použít informace nebo data. Z pohledu bezpečnosti se snažíme zamezit tomu, že by někdo úmyslně odepřel možnost přistupovat k informacím nebo službám tím, že by je učinil nedostupnými. Pokusy o porušení dostupnosti se nazývají odepření služby.

Veškeré statky a prvky, které představují hodnotu se nazývají *aktiva*. Může se tak jednat například o citlivá uživatelská data, která by měla být utajována nebo také samotný software a hardware.

Obecně cokoliv, co může potenciálně narušit bezpečnost se nazývá *hrozba*. Hrozba nastává i v případě, kdy k samotnému útoku nedošlo. *Zranitelná místa* jsou slabiny v systému, která mohou být zneužita k provedení útoku. Systém je vystaven *riziku*, vyskytne-li se současně zranitelné místo a hrozba ve stejný časový okamžik.

Tato práce se zabývá bezpečností počítačů s operačním systémem GNU/Linux. Tyto počítače v sobě také nesou aktiva, která by pro útočníka mohla potenciálně představovat nějakou hodnotu. Mezi příklady aktiv v Linuxových počítačích lze považovat například:

- Konektivita – prostřednictvím konektivity může útočník uskutečnit útok.
- Procesor – útočník může zneužít výpočetní výkon počítače ku svému prospěchu.
- Disk – útočník může využít diskový prostor počítače například k ukládání nelegálních dat a jejich případnou distribuci.
- Data – útočník se snaží nalézt citlivá data nacházející se na počítači.

Pokud by se útočníkovi povedlo cíleně kompromitovat cizí počítač za účelem jeho zneužití, pak z pohledu útočníka je žádoucí skrýt svoji nekalou aktivitu na počítači, neboli zamazat svoje stopy. K tomuto účelu existují specializované software, kterým se říká *rootkity*.

3.2 Nejčastějších 10 zranitelností ve webových aplikacích dle OWASP

The Open Web Application Security Project (zkráceně *OWASP*)[1] je nezisková organizace s více než dvacetiletou historií, která si klade za cíl zlepšit bezpečnost softwaru. Organizace OWASP se svou komunitou, jež zahrnuje desítky tisíc členů, stojí za několika open-source softwarovými projekty a pořádáním edukačních konferencí. OWASP vytváří a zdarma poskytuje články, dokumentace, nástroje a technologie v oblasti kybernetické bezpečnosti webových aplikací. Mezi nejznámější publikovaný dokument se řadí *OWASP Top 10*, který popisuje deset nejčastějších bezpečnostních chyb ve webových aplikacích. Poprvé byl vydaný v roce 2003 a od té doby pravidelně prochází aktualizacemi, které průběžně mění složení těchto deseti nejčastějších zranitelností na základě aktuální situace ve světě bezpečnosti webových aplikací. V současnosti je nejaktuálnější verze z roku 2021. Cílem tohoto dokumentu je rozšířit povědomí o zranitelnostech v softwaru mezi každého vývojáře, a tím tak minimalizovat bezpečnostní rizika všech aplikací.

Tato sekce se zabývá popisem zranitelností z dokumentu *OWASP Top 10 – 2021* [7].

1. *Chybné řízení přístupu* – řízení přístupu vynucuje politiku, která zamezuje uživatelům jednat mimo jejich přidělená práva. Chyby v řízení přístupu obvykle vedou k neautorizovanému prozrazení informací, modifikaci nebo destrukci dat, nebo vykonání akce mimo dosah jejich přidělených práv.

2. *Chyba kryptografie* – kryptografie se používá k utajování dat, a to jak uložených, tak přenášených. Důležité je dbát na to, aby se používaly aktuální šifrovací a hashovací algoritmy, dále je také důležité vyvarovat se používání nezabezpečených protokolů jako např. HTTP či FTP. Mezi další rizika patří také použití slabých šifrovacích klíčů, jejich nevhodná správa, či chybějící obměna klíčů.

3. *Injektování* – injektování typicky spočívá v interpretování nějakého škodlivého kódu dodaného uživatelem. Účinnou obranou proti tomuto útoku je správné očištění a použití escape sekvencí nad vstupními daty od uživatele.

4. *Nezabezpečený návrh* – tato kategorie se zaměřuje na rizika spojená s návrhem a architektonickými vadami. Nezabezpečený návrh je široký pojem zahrnující různá zranitelná místa. Existuje rozdíl mezi nezabezpečeným návrhem a nezabezpečenou implementací, neboť obě vady mají různé příčiny a možné nápravy. Bezpečně navržená

[1] https://owasp.org/

aplikace může být implementovaná zranitelným způsobem, ovšem pro nezabezpečený návrh nelze vytvořit zabezpečenou implementaci. Bezpečný design je metodologie, která periodicky vyhodnocuje hrozby a zajišťuje, aby byl zdrojový kód aplikace navržen robustně a odolně proti známým útokům. Threat modeling by měl být integrovaný do vývojového procesu.

5. *Chybná konfigurace zabezpečení* – mnoho software v dnešní době poskytuje spoustu konfiguračních možností, proto se stále častějším jevem stává chybná konfigurace, která tvoří aplikaci zranitelnou. Mezi časté případy této zranitelnosti patří používání výchozích autentizačních údajů k aplikaci, odesílání potenciálně citlivých informací uživateli jako je například obsah zásobníku při vyvolané výjimce nebo nevyužívání bezpečnostních hlaviček v aplikačních protokolech.

6. *Zranitelné a zastaralé komponenty* – tato zranitelnost nastává v případě provozování zranitelné, nepodporované nebo zastaralé verze software. To zahrnuje mnoho komponent, a to například operační systém, aplikační server, databázi, knihovny, atd. Mezi způsoby, kterými se lze bránit, patří nemít v systému nadbytečný nepotřebný software, pravidelně kontrolovat provozovaný software, zda je stále aktuální včetně všech jeho závislostí a pravidelná aktualizace z důvěryhodných zdrojů.

7. *Chyba identifikace a autentizace* – správná autentizace je zásadní část pro bezpečné fungování celé aplikace. Aplikace může být zranitelná například v případech, kdy dovoluje přihlašovat se pomocí hádání hesel hrubou silou, chybně ukládá hesla do databáze, případně pokud chybně uklízí autentizační tokeny.

8. *Chyba softwaru a integrity dat* – tato kategorie cílí převážně na procesy instalace a aktualizace softwaru. Bezpečnostní incident může nastat v případech, kdy se stahují a instalují moduly či knihovny z nedůvěryhodných zdrojů. Také automatizovaná aktualizace software může přinést tuto chybu. Způsoby, jak se proti této zranitelnosti bránit, zahrnují využívání digitálních podpisů pro verifikaci pravosti dat, zajištění získávání softwaru z důvěryhodných zdrojů a využívání nástrojů pro kontrolu zranitelností v modulech.

9. *Chybné bezpečnostní logování a monitorování* – cílem bezpečnostního logování a monitorování aplikace je detekovat, eskalovat a provést odezvu k aktivním průnikům do systému. Bez těchto procesů nelze zjistit, že byl systém napaden nebo zda došlo k jinému selhání. V tomto případě je jako obranný mechanismus žádoucí provádět penetrační testování systému, pro ověření, zda detekční mechanismus bezpečnostních incidentů je schopný správně reagovat.

10. *Padělání požadavku na straně serveru* – tento bezpečnostní problém nastává v případě, kdy webová aplikace přijímá data, aniž by validovala URL dodané uživatelem. Tímto je útočníkům umožněno donutit aplikaci, aby odeslala útočníkem vytvořený požadavek na neočekávanou destinaci. Tento typ zranitelnosti je v současnosti na vzestupu, jelikož získávání dat z URL bývá v moderních aplikacích často přítomná vlastnost.

3.3 Systémy pro detekci průniku a systémy pro prevenci průniku

Tato sekce čerpá informace z [9] a [12].

Systém pro detekci průniku (angl. *Intrusion Detection System*, dále označován zkratkou *IDS*) je software nebo hardware, který umožňuje detekovat nebo identifikovat útoky proti zdrojům v síti jako například servery, směrovače. IDS je pasivní obranný mechanismus, což znamená, že poslouchá síťový provoz a na základě něj informuje, že nastala nějaká nežádoucí událost.

Systém pro prevenci průniku (angl. *Intrusion Prevention Systém*, dále označován zkratkou *IPS*) je aktivní obranný systém, který má podobné vlastnosti jako IDS systém, tedy je schopen detekovat útok na základě pozorování, ale zároveň umožňuje i tento útok zastavit.

Motivace pro nasazení IDS/IPS systému je snaha detekovat a zabránit šíření malware a útokům na aplikační vrstvě. Standardní nástroj ze síťového světa, paketový filtr v podobě například firewallu, často nestačí na zastavení pokročilých útoků, neboť filtruje pouze na vrstvě L3 a případně L4. Z toho plyne, že na základě informací z IP hlavičky nebo hlaviček transportních protokolů nelze získat dostatečné množství informací, aby se dalo rozhodnout, zdali se jedná o legitimní provoz či nikoliv.

Příkladem jednoduchého IDS/IPS systému je software *fail2ban*. Tento nástroj lze nainstalovat například na server, na kterém současně běží nějaká webová služba. Systém na tomto serveru monitoruje logovací soubory (například `/var/log/auth.log`) a v případě, že zaznamená nadměrnou síťovou aktivitu z jedné konkrétní IP adresy, zablokuje ji na určitou dobu. Tímto mechanismem se zabraňuje útočníkům například v hádání hesel na serverech.

3.3.1 Typy IDS/IPS systémů

Typicky se systémy IDS/IPS rozdělují do dvou kategorií na základě toho, na kterém místě v síťové topologii jsou nasazeny, a to *host-based IDS/IPS* (zkráceně HIDS/HIPS) a *network-based IDS/IPS* (zkráceně NIDS/NIPS):

- Host-based IDS je software, který běží na koncových stanicích či serverech a na nich monitoruje například činnosti na počítači, běžící uživatelské procesy, přístup k systémovým souborům. Ve skutečnosti se jedná o antivirus s tím rozdílem, že odesílá data o dění na zařízení nějakou centrální stanici. Tento typ IDS vyžaduje nainstalování monitorovacího agenta na každém cílovém zařízení.
- Network-based IDS/IPS odposlouchává síťový provoz a na základě toho detekuje útoky. Tato zařízení mohou být nasazena například v podobě samostatného fyzického síťového prvku. V tomto systému je způsob nebo poloha umístění v síti zásadní, neboť v různých částech sítě se přenáší různé informace, což může ovlivnit výkon IDS/IPS systému. Pokud by se nasadil IDS/IPS systém například před směrovač provádějící překlad adres (NAT), docházelo by zde ke ztrátě informace (IP adresy), což by mělo vliv na výkon a docházelo by k častějším falešným poplachům.

3.3.2 Zapojení network-based IDS/IPS systémů do sítě

Pro zapojení NIDS/NIPS systémů do síťové topologie existuje několik možností, kam je umístit. Každá varianta s sebou přináší různé výhody a nevýhody.

- Pokud je nutné, aby IDS/IPS systémy prováděly filtraci dat tekoucích skrz síť, je zapotřebí je zapojit inline, tedy jako zařízení, které přijímá a přeposílá pakety od sousedících zařízení. Mezi hlavní nevýhody tohoto zapojení patří to, že z IDS/IPS systému se stává jediný bod selhání, pokud by došlo k výpadku tohoto systému, současně by došlo k výpadku této části sítě. Dále může takto zapojené zařízení znatelně snižovat propustnost sítě.

- Jinak je možné IDS/IPS systém provozovat „bokem“, ať už v hardwarové či softwarové podobě. Hlavní výhoda tohoto zapojení plyne z toho, že zařízení nijak neovlivňuje chod sítě, takže výpadek IDS/IPS systému neomezí provoz v síti. Při tomto způsobu je nutné „nakopírovat“ provoz v síti na IDS/IPS systém, ať už ve fyzické podobě např. zařízením *network tap* nebo pomocí tzv. zrcadlení portu (*SPAN port*).

3.3.3 Detekce útoků v IDS/IPS systémech

IDS/IPS systémy detekují útoky na základě pravidel, které se označují jako signatury. Signatury existují dvojího typu, a to atomické a složené:

- Atomické signatury jsou informace, pro jejichž detekci stačí jeden samostatný paket. Nepotřebují tak uchovávat žádný stav. Příkladem využití tohoto pravidla může být při validování ARP zpráv.

- Složené signatury naopak vyžadují při detekci posloupnost několika kroků a dochází zde k uchovávání stavové informace

Poté, co se detekuje signatura útoku, dojde ke spuštění nějaké definované akce, například alarmu. Při reakci na útok může dojít k chybám dvojího typu, a to chybě pozitivní, při které se normální provoz nahlásí jako útok a chybně negativní, při které probíhá v síti reálný útok, který se nedetekoval. Míra těchto dvou veličin patří mezi některá z možných kritérií používaných pro porovnání kvality IDS/IPS systémů.

Metod pro detekci signatury existuje několik:

- Detekce založená na vzoru – hledá se specifický vzor v datech.

- Detekce založená na politice – administrátor si ručně nadefinuje vzor, který považuje za nežádoucí.

- Detekce založená na anomáliích – nejprve se definuje normální profil provozu a při odchylkách se hlásí problém. Útoky se detekují na základě statistických či nestatistických (na základě expertní znalosti se ručně nastaví hranice, po jejíž překročení se hlásí útok) metodách v nasbíraných datech. Tímto způsobem je možné odhalit ještě neznámé útoky.

- Detekce založená na honey potu – detekce útočníků v cíleně nezabezpečených systémech, které se snaží nalákat útočníka.

3.4 Existující host-based intrusion detection systémy

IDS/IPS systémů existuje celá řada, proto je vhodné se podívat na již existující řešení a porovnat jejich možnosti. Tato část popisuje některé z nich.

- *OSSEC*[2] – *Open Source Security Event Correlator* je host-based IDS napsaný v jazyce C. Podporuje několik platforem, např. Linux, FreeBSD, Windows. Umožňuje analyzovat logy, kontrolovat integritu, detekovat rootkity, zasílat upozornění na vyskytnuté incidenty a aktivně reagovat. OSSEC se skládá z dvou komponent, a to Server a Agent.
- *Tripwire*[3] – jedná se open-source host-based IDS. Mezi jeho hlavní schopnosti patří zajištění integrity dat a vytváření upozornění při detekci jejich korupce.
- *Samhain*[4] – další open-source HIDS, který poskytuje monitorování logovacích souborů, kontrolu integrity a detekci skrytých procesů.
- *Security Onion*[5] – jedná se o kolekci software a hardware. Software existuje v podobě open-source linuxové distribuce, jejíž součástí jsou IDS systémy jako *Snort*, *Wazuh* a *Suricata*.

3.5 Extended Berkley Packet Filter

Tato sekce se zabývá úvodem do technologie *Extended Berkeley Packet Filter*, dále označováno jen jako *eBPF*. Do této části práce je eBPF zařazeno, neboť nachází své uplatnění v bezpečnostních nástrojích pro operační systém GNU/Linux a rovněž je využito v implementační části této práce. Informace do této sekce jsem čerpal z [11] a [1].

Technologii eBPF lze využít i k mnoha dalším účelům – např. k trasování, pozorování systému, profiling. Programy eBPF se zavádějí do kernel space pomocí systémového volání `bpf()`.

3.5.1 Historie BPF a eBPF

Předchůdcem eBPF je BPF. Jak již samotný název napovídá, *Berkeley Packet Filter* byl vytvořen ve světě operačního systému BSD. V roce 1992 vzniknul vědecký článek od autorů Steven McCanne a Van Jacobso „The BSD Packet Filter: A New Architecture for User-Level Packet Capture“[18], který přinesl inovace ve zpracování síťových paketů, jako například mnohonásobně rychlejší filtrování paketů než tehdejší state-of-the-art paketové filtry. BPF se postupem času rozšířilo i do dalších operačních systémů v Unixové rodině, ale i do Windows.

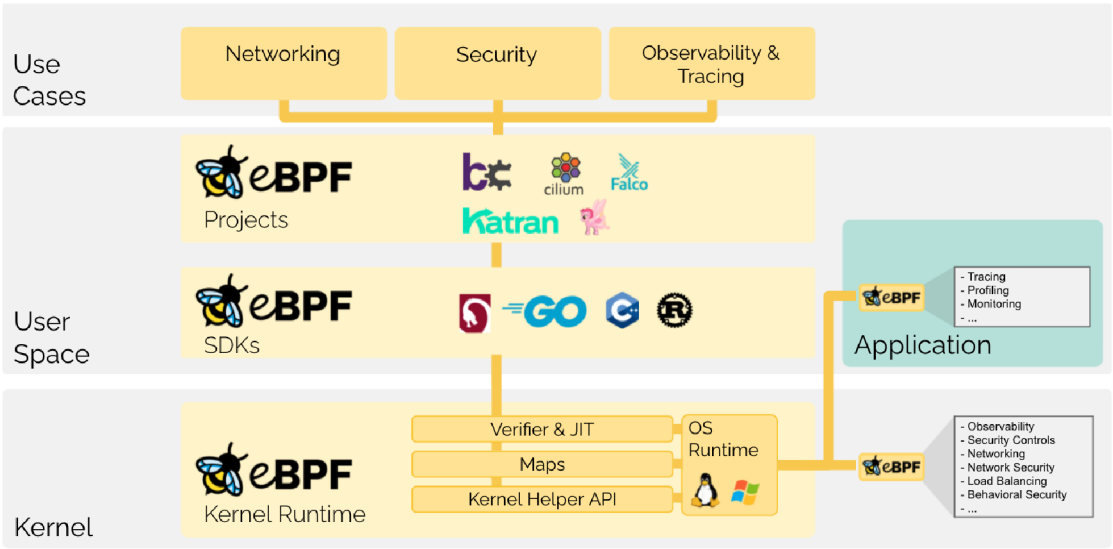
Technologie eBPF byla uvedena do Linuxového kernelu verze 3.18, který vyšel koncem roku 2014. V této verzi eBPF zásadně vylepšilo schopnosti dosavadního BPF tím, že umožnilo psát více komplexní programy a také došlo až k čtyřnásobnému zrychlení oproti BPF. Také se navýšil počet registrů z dvou 32bitových na deset 64bitových.

3.5.2 Architektura eBPF

Obrázek 3.1 zobrazuje vysokoúrovňovou architekturu eBPF, která lze rozdělit na user-space a kernel-space části:

[2] https://www.ossec.net/
[3] https://www.tripwire.com/
[4] https://la-samhna.de/samhain/index.html
[5] https://securityonionsolutions.com/

- User-space část tvoří uživatelské programy a nástroje poskytující programátorovi pohodlnější rozhraní pro práci s eBPF. Mezi tyto projekty patří například *cilium*[6] a *bcc*[7].
- Kernel-space část obsahuje komponenty potřebné pro běh eBPF programů – verifikátor, mapy a programové rozhraní.



Obrázek 3.1: Architektura technologie eBPF. Obrázek převzat z [1].

3.5.3 Typy eBPF programů

Tato část popisuje několik typů eBPF programů. Existuje mnoho typů programů, nicméně obecně je lze rozdělit do dvou kategorií:

- Trasovací programy – jejich cíl spočívá v poskytnutí informací o tom, co se děje uvnitř operačního systému na úrovni jádra. Pomocí těchto typů programů je umožněno sledovat například trasování volání funkcí v procesech nebo sledovat alokované zdroje jednotlivých procesů, jako jejich využití paměti či CPU.
- Síťové programy – tyto programy umožňují nahlížet a manipulovat s tím, co se děje v síťovém provozu v systému. Je tak možné filtrovat nebo zahazovat pakety na síťovém rozhraní. Pakety je možné zpracovávat na různých úrovních, lze se tak například napojit na události následující ihned po zpracování síťovou kartou, anebo se lze napojit na události těsně předcházející tomu, než je kernel předá do user-space.

Socket filter programy

Tyto programy patří mezi první typ eBPF programů, které byly přidány do Linuxového jádra. Definovat je lze pomocí `BPF_PROG_TYPE_SOCKET_FILTER`. Programy umožňují se napojit na síťové sockety a získat tak informace o všech paketech, které tímto socketem procházejí. Dovoleno je pouze sledování paketů, nikoliv jejich modifikace.

[6] https://cilium.io/
[7] https://github.com/iovisor/bcc

Kprobe programy

Programy *kprobe* jsou funkce, které se napojí na libovolnou funkci kernelu a spustí se tak se zavoláním této vybrané funkce. Programy lze definovat pomocí `BPF_PROG_TYPE_KPROBE`. Lze také zvolit, zdali se eBPF program vykoná před nebo po volání vybrané funkce.

Tracepoint

Tento typ programů se umožňuje napojit na tracepoint handler poskytovaný Linuxovým jádrem. Definují se pomocí `BPF_PROG_TYPE_TRACEPOINT`. V těchto místech lze injektovat kód pro trasování a debugování. Tento typ programů lze přirovnat ke *kprobe* programům, nicméně tento případ poskytuje menší flexibilitu.

3.5.4 Verifikátor

Oproti např. kernel modulům, eBPF programy mají určitou garanci toho, že jsou bezpečné na spuštění. Tyto vlastnosti zajišťuje eBPF verifikátor, který kontroluje, že program splňuje následující kritéria:

- Při běhu nedojde k selhání programu nebo jinému poškození systému.
- Program vždy doběhne do konce, nesmí uváznout nebo donekonečna cyklit.
- Programy nesmí využívat neinicializované proměnné a přistupovat do paměti mimo přidělený rozsah.
- Velikost eBPF programu nesmí přesáhnout limit operačního systému, není tak možné spouštět programy neomezené velikosti.

3.5.5 Mapy

Důležitým aspektem eBPF programů jsou takzvané *mapy*. Mapy existují v podobách několika datových struktur a využívají se tedy k ukládání a získávání dat v eBPF programech. Tyto datové struktury lze sdílet mezi user-space a kernel-space, umožňují tak přenášet data ven z eBPF programů[4].

Celkově se v eBPF nachází mnoho typů map, řádově se jedná okolo několika desítek, a tak nemá smysl zde vyjmenovávat všechny z nich. Podrobný výčet map lze vyčíst z literatury, případně ze zdrojových souborů. Mezi nejzákladnější patří následující typy:

- `BPF_MAP_TYPE_HASH`, `BPF_MAP_TYPE_ARRAY` – hashovací tabulka a pole,
- `BPF_MAP_TYPE_QUEUE`, `BPF_MAP_TYPE_STACK` – fronta a zásobník,
- `BPF_MAP_TYPE_LRU_HASH` – least recently used,
- `BPF_MAP_TYPE_RINGBUF` – kruhový buffer,
- `BPF_MAP_TYPE_STACK_TRACE` – trasování zásobníku,
- `BPF_MAP_TYPE_LPM_TRIE` – longest prefix match.

Kapitola 4

Threat model Testing Farm

Threat modeling spočívá ve využití modelů k nalezení bezpečnostních děr. Model v tomto případě reprezentuje abstrakci nad mnoha detaily, která umožňuje podívat se na systém v širším měřítku. Tato kapitola tvoří threat model pro službu Testing Farm. Existuje literatura o tom, jaké přístupy volit při provádění této aktivity. Informace jsem čerpal z [23].

4.1 Profil útočníka

Jedním z prvních kroků při vytváření threat modelu spočívá v identifikaci útočníka. Cílem této sekce je tedy ujasnění toho, proti komu je vlastně službu třeba bránit. V úvodu práce zaznělo, že plánovaná práce na službě Testing Farm zahrnuje její rozšíření mezi širší skupinu lidí. Tato část přibližuje, kdo se za touto skupinou lidí skrývá.

Jak již bylo zmíněno, v současné době jsou autorizovaní k používání služby výhradně zaměstnanci Red Hatu, od kterých se neočekává, že by úmyslně využili službu k cílům jiným, než ke kterým je určena.

Je zapotřebí zohlednit skutečnost, že aplikační rozhraní služby je vystavené do veřejného internetu. Únik autentizačního API klíče legitimního uživatele, byť neúmyslný, by mohl vyústit k nežádoucímu zneužití výpočetních prostředků služby. Nicméně tento scénář, ačkoliv reálný, nepředstavuje tak významnou hrozbu, aby stálo za to vynaložit nemalé úsilí pro omezení dopadů toho, že někdo „omylem ztratil API klíč“.

Daleko významnější hrozbu představuje výhledový plán do budoucna, který by rozšířil uživatelskou základnu služby Testing Farm i mimo zaměstnance Red Hatu, a to konkrétně mezi komunitu vývojářů distribuce Fedora. Neznamená to sice, že by ke službě mohla přistupovat libovolná osoba připojená k internetu, stále bude přítomen manuální proces přidělování API klíčů pro vybrané uživatele. Nicméně z této skutečnosti plyne, že rozšířením uživatelské základny Testing Farm o tyto ne zcela důvěryhodné uživatele dojde k potenciálnímu rozšíření vektoru útoku.

Každý vývojář přispívající do projektu Fedora musel podepsat smlouvu[1], že souhlasí s podmínkami pro přispěvatele do projektu Fedora. Smlouva technicky samozřejmě nikoho žádným způsobem neomezuje v provádění počítačových útoků, nicméně smluvní podmínky také patří mezi určitý typ bezpečnostního opatření.

[1] https://fedoraproject.org/wiki/Legal:Fedora_Project_Contributor_Agreement

4.2 Zneužitelnost testovacích strojů

Bezpečnost týkající se testovacích strojů služby patří mezi hlavní důvody, kvůli kterým tato práce vznikla. Každý uživatel oprávněný k užívání Testing Farm má možnost vyžádat si dedikovaný virtuální stroj, ke kterému má k dispozici plný přístup jako privilegovaný uživatel `root`. Uživateli je tak umožněno spouštět libovolný kód na těchto strojích.

První myšlenka pro omezení potenciálních útočníků spočívá v omezení práv uživatele na virtuálních strojích poskytujících testovací prostředí. Ovšem povaha testovacích případů, které uživatelé služby spouští, vyžaduje k jejich vykonávání právě přístup uživatele `root`. Je tak snadné se vžít do role útočníka a představit si některé způsoby, kterými je možné tyto stroje zneužít k jiným účelům, než jsou určené. Z těchto skutečností plyne, že je zapotřebí nasadit monitorovací systém, který by sledoval aktivitu jednotlivých testovacích strojů samotných včetně jejich síťové aktivity, kterou produkují či přijímají.

4.3 Možné způsoby zneužití testovacích strojů útočníkem

Jak již zaznělo v předchozí kapitole 3, v Linuxových počítačích se nacházejí aktiva, která by pro útočníka mohla představovat nějakou hodnotu. Cílem této sekce je tedy zamyslet se nad tím, co konkrétního by mohl útočník chtít napáchat na testovacích strojích služby.

Dle technického konzultanta, stručný výčet některých typů provozu, které lze považovat jako nežádoucí nebo podezřelé na testovacích strojích služby Testing Farm, je následující (Pozn.: pojmem *test* se rozumí definice potenciálně nebezpečného testovacího případu dodaného od uživatele-útočníka.):

- Test se pokusí o skenování otevřených portů.
- Test se pokusí o provedení Denial of Service útoku.
- Test nahraje do systému kernel modul.
- Test otevře síťový port na veřejném síťovém rozhraní.
- Test vypne monitorovací nástroj nebo ho neoprávněně zmanipuluje.
- Test změní initrd na testovacím stroji.
- Test restartuje testovací stroj.
- Test významně zatíží CPU, IO nebo paměť testovacího stroje.
- Test spustí program na těžbu kryptoměn.
- Test přenese velké množství dat do internetu.

Negativní dopady provádění výše uvedených aktivit jsou zjevné – například nadměrné zneužití výpočetního výkonu stroje se projeví ve finanční stránce provozu daného stroje, skenováním sítě Red Hatu může neúmyslně vyzradit síťovou topologii, čímž by se pro útočníky mohl rozšířit vektor útoku.

Následující části této kapitoly se podrobněji zabývají některými z výše uvedených jevů za účelem vysvětlení toho, jakým způsobem fungují. Posléze kapitoly 5 a 6 navrhují a implementují způsob, jak tato chování systému detekovat.

4.4 Skenování sítě

Jedním z prvních kroků, které mohou útočníci na kompromitovaném počítači provést, je skenování sítě. Tím lze získat informace o síťové topologii, dalších počítačích v síti, o dostupných službách, atd. Z podstaty služby Testing Farm plyne, že během normálního provozu si každý testovací stroj vykonává svou vlastní činnost nezávisle na ostatních strojích, tedy že síťová komunikace mezi těmito stroji nepatří mezi běžný jev. Ačkoliv skenování sítí samo o sobě nepatří mezi počítačové útoky, v tomto prostředí se však jedná o podezřelý síťový provoz, který může útoku předcházet. Z tohoto plyne, že tento typ provozu by bylo nanejvýš vhodné detekovat.

Technik pro skenování sítí existuje vícero. Švýcarským nožem v oblasti skenování sítí je nástroj *nmap*[2], proto jsem pro tuto část práce čerpal informace z [17] a [19]. Následující sekce představí princip fungování skenování sítí a vysvětlí některé techniky.

Skenování sítí se skládá z několika po sobě navazujících fází, mezi které patří:

1. *Target enumeration* – zjištění IP adres zadaných skenovaných cílů.
2. *Host discovery* – zjištění, které počítače ve skenované síti jsou zapnuté.
3. *Reverse-DNS resolution* – po nalezení adres skenovaných počítačů se vyhledají reverzní DNS záznamy.
4. *Port scanning* – odeslání skenovacích paketů na cílové porty pro zjištění, v jakém stavu se nacházejí (otevřený, uzavřený, filtrovaný).
5. *Version detection* – pokud se nějaký port nachází v otevřeném stavu, pak se vyhodnocuje, která služba a jaká její verze za daným portem běží.
6. *OS detection* – různé operační systémy implementují síťové standardy různými způsoby, a tak lze z těchto drobných rozdílů v některých případech odvodit, o jaký operační systém se jedná.

Následující podsekce popisují fáze *host discovery* a *port scanning* včetně různých technik, které se používají pro skenování.

4.4.1 Host discovery

Mezi jednu z prvních fází při skenování sítí patří vyhledávání běžících počítačů v dané síti. Vzhledem k tomu, že testovací stroje by se ani neměly snažit získat tyto informace, je tak i tato fáze *host discovery* považována za nežádoucí provoz.

TCP SYN ping

V této variantě host discovery se odesílá prázdný TCP datagram s nastaveným příznakem SYN a s libovolným portem. Tímto vzdálený server nabývá dojmu, že si odesilatel přeje ustanovit TCP spojení. Pokud je port uzavřen, přijde odpověď ve formě TCP datagramu s nastaveným příznakem SYN. V opačném případě, když je port otevřen, přijde TCP odpověď s nastavenými příznaky SYN a ACK. Vzhledem k tomu, že smyslem host discovery fáze je zjištění existence vzdáleného počítače s nějakou IP adresou, je libovolná příchozí odpověď známkou toho, že vzdálený server je spuštěný.

[2] https://nmap.org/

Pro odesílání raw TCP datagramů jsou zapotřebí práva privilegovaného uživatele `root`. Pro neprivilegovaného uživatele existuje varianta využití systémového volání `connect()`.

TCP ACK ping

TCP ACK ping se podobá variantě TCP SYN ping. Rozdílem je, že první paket, který se odesílá na vzdálený server, tvoří TCP datagram s nastaveným příznakem ACK. Takovýto datagram budí dojem, že již existuje navázané TCP spojení, což ovšem server vyvrátí zasláním TCP datagramu s nastaveným příznakem RST.

Pro tento typ skenování je nezbytný `root` přístup na počítači, ze kterého se skenuje, neboť volání `connect` neumožňuje odeslání TCP datagramu s nastaveným ACK. Vždy se iniciuje spojení s příznakem SYN.

Výhoda této techniky oproti TCP SYN ping spočívá ve vyšší pravděpodobnosti pro obejití pravidel firewallu cílového serveru. V bezstavových firewallech může být nastaveno zahazování TCP SYN datagramů, které směřují na uzavřené porty. TCP ACK datagram by toto pravidlo obešlo a dorazilo by na cílový port. Ovšem pokud je na vzdáleném serveru nasazen stavový firewall, tak detekuje, že příchozí TCP ACK datagram nepatří k žádnému TCP spojení, a zahodí ho.

UDP ping

Další host discovery možností je UDP ping. Ten spočívá v odeslání UDP datagramu na nějaký port, přičemž je vhodné volit takový port, pro který je pravděpodobné, že je uzavřený. Pokud by paket dorazil na otevřený port, s velkou pravděpodobností by ho daná služba zahodila bez jakékoliv odpovědi. Naopak pokud dorazí tento prázdný paket na uzavřený port, vrátí se *ICMP port unreachable* zpráva nazpět odesilateli. Tato zpráva značí, že vzdálený server běží. Ostatní typy ICMP zpráv indikují, že vzdálený server není spuštěný.

Výhodou této volby pro host discovery může být, podobně jako u předchozí možnosti TCP ACK Ping, obejití pravidel firewallu v případě, že jsou pravidla nastavena pouze pro TCP protokol.

ICMP ping

Standardní způsob pro provedení host discovery je s využitím protokolu ICMP, konkrétně pomocí typu 8, *echo request*. Očekávaná odpověď na tuto zprávu je paket ICMP typu 0, *reply*. Odesílání tohoto typu zpráv podporuje známá utilita *ping*.

Ačkoliv ICMP zpráva echo request je standardní způsob, jak objevit vzdálené počítače, ještě lze použít několik dalších typů zpráv pro tento účel. Těmi jsou ICMP *timestamp request* (typ 13) a *address mask request* (typ 17). Očekávané odpovědi na tyto zprávy jsou *timestamp reply* (typ 14) a *address mask reply* (typ 18). Tyto různé varianty mají svoje opodstatnění opět v případě, kdy základní varianta, echo request, je blokovaná, ale blokování ostatních typů zpráv je opomenuto.

IP ping

Další varianta pro host discovery je odeslání IP paketu se specifikovaným číslem protokolu v IP hlavičce. Tato metoda hledá odpovědi buď ve formě stejného protokolu, který byl specifikován v odeslaném paketu, nebo ICMP zprávu typu *unreachable*, která značí nedostupnost daného protokolu na vzdáleném serveru. V každém případě libovolná odpověď znamená úspěšné objevení hledaného serveru.

ARP sken

Mezi časté jevy také patří skenování lokální sítě (LAN). Před samotným odesláním např. ICMP zprávy pro zjištění počítačů je ve většině takovýchto sítí nutné nejprve zjistit hardwarovou adresu síťového rozhraní pro správné sestavení ethernetového rámce. K tomuto účelu se používá protokol ARP. Na broadcast adresu dané sítě se odešle požadavek *ARP request*, který se dotazuje, jaká HW adresa odpovídá zvolené IP adrese.

4.4.2 Skenování portů

Skenování portů probíhá za účelem zjištění, v jakém stavu se nachází TCP a UDP porty na vzdáleném serveru. Porty se mohou nacházet ve 3 stavech:

- Otevřený – port je dosažitelný pro TCP spojení nebo UDP komunikaci a naslouchá na něm nějaká aplikace. Nelezení tohoto typu portu bývá při skenování sítí nejžádanější.
- Uzavřený – port je dosažitelný, nicméně na něm neposlouchá žádná aplikace.
- Filtrovaný – port je nedosažitelný, takže není jasné, zda na něm nějaká aplikace poslouchá. Port může být filtrovaný například z důvodu existence nějakého paketového filtru, který pakety zahazuje.

Útočníci tuto techniku využívají, neboť otevřený port na nějakém počítači může být příležitost pro provedení útoku. Pro kompromitaci serveru stačí, aby na něm byla spuštěna jediná zranitelná služba s otevřeným portem pro poslech. Tato část popisuje několik technik, kterými lze skenování provádět.

TCP SYN sken

Tento typ skenu patří mezi nejpopulárnější, neboť ho lze provádět rychle. Na rychlé síti bez striktního firewallu lze skenovat tisíce portů za sekundu. TCP SYN sken je spíše nenápadný, jelikož nedokončuje ustanovení TCP spojení.

Jako první paket se vzdálenému serveru odešle TCP datagram s nastaveným příznakem SYN. Následují tři možnosti v závislosti na tom, v jakém stavu se port nachází:

1. Při otevřeném portu vzdálený server odpoví TCP datagramem s nastavenými příznaky SYN a ACK. Tato zpráva dává informaci, že port je otevřený a server souhlasí se zahájením TCP spojení. Nicméně potřebnou informaci odesílatel již získal a o navázání TCP spojení nestojí, takže serveru odešle paket s nastaveným příznakem RST.
2. Naopak u zavřeného portu server odpoví TCP zprávou s příznakem RST. Tím se získá informace, že port je uzavřený.
3. Jako filtrovaný se port považuje, pokud žádná odpověď od serveru nedorazí, nebo také v případě, kdy dorazí vybrané typy chybových zpráv ICMP.

TCP connect sken

Tento typ skenu lze považovat za méně účinnou variantu (s drobnými změnami) předchozího TCP SYN skenu, který vyžaduje přístupová práva privilegovaného uživatele `root` pro vytváření raw paketů. Tato varianta je tedy dostupná pro neprivilegované uživatele a využívá systémového volání `connect()`, což je ovšem z několika důvodů méně účinné. Jednak

přenechává větší režii operačnímu systému, a tak uživatel provádějící sken nemá takovou moc nad samotným odesíláním paketů. Dále toto systémové volání vždy ustanovuje TCP spojení (ve třetím kroku TCP handshake odešle příznak ACK namísto RST), což způsobuje pomalejší rychlost skenování, neboť je potřeba odeslat větší počet paketů.

UDP sken

UDP sken spočívá v odeslání prázdného UDP datagramu (prázdný znamená, že je vyplněna UDP hlavička a datová část je ponechána prázdná) na nějaký cílový port vzdáleného serveru. V závislosti na odpovědi lze stav portu rozhodnout následovně:

1. Port je otevřený, pokud server odpoví UDP datagramem.
2. Port je otevřený nebo filtrovaný, pokud se nevrátí žádná odpověď. Tato situace je obzvlášť zákeřná, neboť nelze jednoznačně rozhodnout, v jakém stavu se port nachází. Běžící služby za otevřenými porty obvykle zahazují prázdné pakety, což je stejné chování jako u filtrovacích zařízení.
3. Port je uzavřený, pokud se vrátí zpráva *ICMP port unreachable error*.
4. Port je filtrovaný, pokud se vrátí ostatní zprávy typu *ICMP unreachable error*.

TCP FIN, FIN, a Xmas sken

Tyto tři typy skenů využívají vlastnost uvedenou v RFC 793[6], která umožňuje rozlišení mezi otevřeným a uzavřeným portem. Dle RFC 793, pokud na uzavřený port cílí TCP datagram bez příznaku RST, pak server jako odpověď odešle TCP datagram s nastaveným RST. Dále pokud na otevřený port dorazí TCP datagram bez SYN, RST, nebo ACK příznakem, pak se tyto pakety zahodí bez odpovědi.

Jak již bylo zmíněno, pro tento typ skenu se používají tři varianty TCP datagramu (ale lze využít i jiné kombinace, dle nastavených TCP příznaků):

1. Null sken – odešle se TCP datagram s žádnými příznaky.
2. FIN sken – odešle se TCP datagram s nastaveným příznakem FIN.
3. Xmas sken – odešle se TCP datagram „rozzářený jako vánoční stromek“ s nastavenými příznaky FIN, PSH a URG.

Na základě odpovědi serveru pak lze rozhodnout stav portu následovně:

1. Port je otevřený nebo filtrovaný, pokud server nevrátí žádnou odpověď.
2. Port je uzavřený, pokud server vrátí odpověď v podobě TCP RST datagramu.
3. Port je filtrovaný, pokud se vrátí *ICMP unreachable error* zpráva.

Jak již nastínil předchozí výčet, toto skenování nepatří mezi nejefektivnější, neboť nelze jednoznačně určit otevřenost portu. Ovšem výhoda spočívá v možnosti obejití nestavových firewallů, které se snaží zabránit vzniku TCP spojení. Nevýhoda spočívá v tom, že skenovaný systém se musí chovat přesně v souladu RFC 793, což nebývá pravidlem. Obvykle lze tento sken použít na zařízení založených na Unixových operačních systémech.

4.5 Provádění Denial of Service útoku

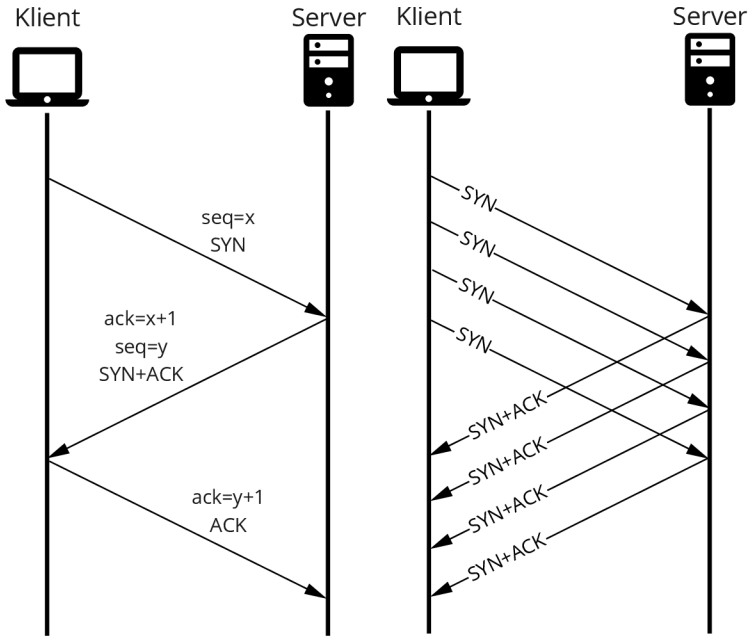
V této části práci chceme docílit toho, aby se zamezilo možnosti využít testovací stroje k provádění útoků typu odepření služby. Jak již zaznělo v kapitole 3, odepření služby (angl. *Denial of Service*, zkr. *DoS*) je populární typ síťového útoku, jehož cílem je porušit bezpečnostní cíl dostupnost. Tento útok bývá směřován na síťové služby vzdáleného serveru, například webového.

Známou variantou je také *DDoS* (*Distributed DoS*), který se liší pouze tím, že k vykonávání útoku se používá více než jeden počítač. Existuje několik způsobů, kterými lze tento typ útoku provádět. Tento útok bývá populární v případech, kdy dojde ke kompromitaci počítače a jeho následné zapojení do botnetu, který následně synchronizovaně provádí útoky či jiné nežádoucí aktivity. Informace o DoS útocích jsem čerpal z [22].

4.5.1 SYN Flood

Pro pochopení tohoto typu útoku je nezbytné nejprve pochopit fungování protokolu TCP, zejména navazování spojení *3-way handshake*. V legitimním provozu se ustanovuje TCP spojení následujícím způsobem:

1. Klient nejprve zašle serveru zprávu s nastaveným příznakem SYN v hlavičce TCP datagramu. Sekvenční číslo je nastavené na libovolnou hodnotu x.
2. Poté server odpoví TCP datagramem s nastavenými příznaky SYN a ACK v hlavičce. Acknowledgement číslo je nastavené na hodnotu $x + 1$ a sekvenční číslo na libovolnou hodnotu y.
3. V posledním kroku klient odešle TCP datagram s nastaveným příznakem ACK. Acknowledgement číslo je nastavené na hodnotu $y + 1$.



Obrázek 4.1: Normální TCP 3-way handshake a SYN flood útok.

Mezi jednu z možností, jak vykonat DoS útok, se řadí SYN Flood. V tomto typu útoku dochází ke zneužití navazování spojení takovým způsobem, kde klient záměrně opakovaně žádá o navázání TCP spojení, nicméně v posledním kroku neodesílá s příznakem ACK, čímž nedokončí ustanovení TCP spojení. Tímto způsobem dojde k zahlcení serveru, který čeká, až klient dokonční navázání TCP spojení. Princip navázání spojení a útoku SYN Flood zachycuje obrázek 4.1.

4.5.2 UDP Flood

Dalším způsobem DoS útoku je provádění *UDP Flood*. Jeho princip fungování je na rozdíl od TCP Flood jednodušší, což plyne z vlastností UDP protokolu, který není spojovaný. Nelze tak využít stejný trik jako u TCP Flood, kde princip útoku spočívá v nedokončení ustanovení spojení.

4.6 Nahrání kernel modulu

Kernel moduly představují způsob, jak za běhu systému modifikovat chování kernelu bez nutnosti jeho změny a restartu systému. Kernel modul představuje způsob, kterým lze zavést rootkit do systému, a proto je nežádoucí, aby se o to někdo pokoušel na strojích Testing Farm. Informace o kernel modulech jsem čerpal z [21] a [20].

Pomocí kernel modulů lze například implementovat rootkity a jiné programy, kterým je umožněno:

- Libovolně měnit hodnoty, které se prezentují v pseudofilesystému `/proc` za účelem skrytí toho, co se v počítači děje.
- Skrýt soubory a adresáře na filesystému, skrýt síťové sockety.
- Změnit chování systémových volání.

Výpis 4.1 ukazuje zdrojový kód jednoduchého kernel modulu. Jsou v něm definované dvě funkce, které se zavolají při nahrání modulu do kernelu. Vzhledem k tomu, že se program pohybuje v kernel-space, není tak možné tisknout na standardní výstup jako u user-space programů. Proto je v programu využita funkce `printk()`, která tiskne výstup do kernel logu.

Program uvedený ve výpise 4.1 lze přeložit příkazem `make -C /lib/modules/‘uname -r‘/build M=$(PWD) modules`, čímž vznikne spustitelný soubor ve formátu ELF, který lze do kernelu nahrát příkazem `insmod example.ko`. Pomocí příkazu `dmesg` lze vypsat kernel log a zkontrolovat, že byl modul skutečně nahrán. Odebrat modul lze pomocí příkazu `rmmod example`.

```c
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>

MODULE_LICENSE("GPL");

static int __init example_init(void)
{
    printk(KERN_INFO "Hello, world!\n");
    return 0;
}

static void __exit example_exit(void)
{
    printk(KERN_INFO "Goodbye, world!\n");
}

module_init(example_init);
module_exit(example_exit);
```

Výpis 4.1: Ukázka jednoduchého kernel modulu `example.c`

4.7 Vypnutí SELinux

SELinux (Security Enhanced Linux) je open-source projekt rozšiřující Linuxový kernel, jež vytvořila americká NSA. Informace do této části jsem čerpal z [24] a [25]. SELinux lze nainstalovat na libovolnou Linuxovou distribuci, nicméně na distribucích od Red Hatu je SELinux ve výchozím stavu spuštěný.

SELinux implementuje *mandatory access control* – tedy řízení přístupu mezi *subjekty* a *objekty* vynucené jádrem operačního systému. Mezi objekty lze zařadit například uživatele, procesy a vlákna. Subjekty jsou pak například soubory, adresáře, adresářové svazky, kernel moduly, síťová rozhraní, sokety.

Mezi základní pojem patří *policy*. Jedná se o pravidlo, jež vynucuje bezpečnostní politiku mezi objektem a subjektem. Každý objekt jako například soubor či adresář obsahuje nějakou značku. Prostřednictvím těchto značek se definují politiky, které dovolují vzájemnou interakci mezi objekty. Tyto bezpečnostní politiky pak přímo vynucuje jádro.

Mezi základní typy bezpečnostních politik patří následující:

- *Type enforcement* – zajišťuje politiky mezi různými programy. Pokud by například došlo ke kompromitování procesu s běžícím webovým serverem, pak by SELinux nedovolil přistupovat ke zdrojům v systému, ke kterým tento webový server nemá „označkovaný“ přístup, a to i v případě, že kompromitovaný proces běží s právami `root`.
- *MCS enforcement* – zajišťuje politiky mezi stejnými programy. Umožňuje tak vytvořit oddělená prostředí pro běh několika instancí stejných programů. Příkladem je virtualizační systém *libvirt*. V případě kompromitování jednoho virtuálního stroje SELinux zamezí pokusům o útok na ostatní virtuální stroje.

Z výše uvedeného popisu je zřejmé, že tento systém poskytuje účinnou obranu proti narušitelům. Je tak žádoucí mít tento systém spuštěný na testovacích strojích služby Testing Farm.

4.8 Otevření síťového portu

Jako další možnost podezřelého typu provozu je neočekávané otevření TCP nebo UDP portu do stavu *LISTEN*. Provedení této akce neočekávaně může znamenat, že se útočník pokouší o to vytvořit si tzv. zadní vrátka (*backdoor*) pro jeho snadnější přístup k počítači.

Útočníkova motivace, proč by tohoto chtěl docílit, spočívá v tom, že na stroje Testing Farm neexistuje příliš „uživatelsky přívětivý“ způsob, jak je přímo ovládat. Uživatel/útočník dodá kód k vykonání prostřednictvím například gitového repozitáře, který se naklonuje na testovací stroj a posléze spustí kód uvnitř repozitáře. Útočník tedy ve výchozím stavu nemá pohodlný přístup na stroje v podobě například SSH spojení.

Otevírání zadních vrátek je typickým jevem u škodlivého software, který se snaží rekrutovat počítač oběti do botnetu. Existuje celá řada zadních vrátek, např. *Beagle backdoor*, *NetDevil backdoor*, *Kuang backdoor*. Typickým jevem těchto zadních vrátek je, že otevírají TCP port pro naslouchání[22]. Z těchto skutečností tedy plyne, že je žádoucí monitorovat stavy síťových portů na strojích Testing Farm.

Kapitola 5

Návrh zabezpečení Testing Farm

Cílem této kapitoly spočívá v popisu chování systému, který by měl tvořit realizační výstup práce. Tato implementační část se bude skládat z několika částí, první je implementace monitorovacího agenta – host-based IDS. Jsou zde popsány systémové zdroje, které by měl nástroj sledovat. Z těchto zdrojů budou generovány metriky, z nichž se bude vyhodnocovat legitimnost chování systému. Další část tvoří způsob integrace tohoto nástroje s již nasazenou platformou pro monitorování systémů Prometheus.

5.1 Požadavky na monitorovací nástroj

Ještě před samotnou implementací je namístě si specifikovat a vysvětlit požadavky na monitorovací nástroj, který bude náplní implementace této práce.

Dle požadavků technického konzultanta by monitorovací nástroj měl být jeden binární spustitelný soubor. K tomuto účelu je tedy zapotřebí vybrat vhodný kompilovaný jazyk pro implementaci. Každý stroj testovacího prostředí v sobě bude mít nainstalovaný tento spustitelný binární soubor, který bude monitorovat dění v systému.

Další důležitý požadavek cílí na přenositelnost řešení, tedy nezávislost na cloudové platformě, na které testovací stroje Testing Farm běží. Ačkoliv v současnosti se stroje provozují v cloudu AWS a ten již poskytuje některé nástroje související s monitorováním a bezpečností virtuálních strojů, v budoucnu se však počítá s možností poskytovat část výpočetního výkonu například na platformě OpenShift. Proto je zapotřebí vytvořit nezávislé řešení spustitelné na různých platformách.

5.1.1 Zdroje pro získávání dat o systému

Implementovaný systém by měl z počítače získávat data, ze kterých pak lze vyhodnotit, zdali se na stroji děje nelegitimní provoz. Pro získávání informací o systému lze využít následující zdroje dat.

Souborový systém /proc

Adresář `/proc` poskytuje rozhraní pro datové struktury jádra. Lze ho tak využít k vyčtení informací o dění na operačním systému. Obsahuje podadresář pro každý spuštěný proces v systému[2].

Výčet některých údajů, které lze vyčíst z tohoto filesystému, je následující:

- Základní údaje o systému, jako využití CPU.

- Údaje o každém spuštěném procesu zvlášť jako například využití CPU a RAM.
- Využití sítě, množství přijatých a odeslaných dat, otevřené síťové porty.

Hlavní výhoda tohoto přístupu je jednoduchost na implementaci. Mezi jednu z nevýhod však patří možnost podvržení těchto údajů. Jak již zaznělo v předchozí kapitole, prostřednictvím rootkitů lze ovlivnit výstupy z těchto souborů.

Konkrétně, monitorovací nástroj by měl z tohoto souborového systému číst následující informace:

- Celkové množství přenesených dat – ze souboru `/proc/net/dev`.
- Otevřené síťové porty ve stavu LISTEN – ze souboru `/proc/net/tcp`.
- Využití CPU – ze souborů `/proc/stat` a `/proc/loadavg`.

Knihovna libpcap

Knihovna *libpcap* poskytuje systémově nezávislé programové rozhraní pro zachytávání paketů v user-space programech. Knihovna zachycuje pakety na síťové vrstvě L2 v Unixových operačních systémech. Při zachycování paketů lze použít BPF filtry (zde je myšlena původní technologie BPF, nikoliv eBPF). Knihovnu využívá například populární nástroj pro zachytávání paketů *tcpdump*[1].

Monitorovací nástroj by měl tuto knihovnu využít pro zachycování paketů. Zachycování paketů je potřebné pro detekci skenování sítě a DoS útoků. Skenování sítě pomocí TCP SYN a SYN Flood lze detekovat obdobně – pomocí detekce 3-way-handshake. Program by si měl uchovávat stav o (ne)uskutečněných ustanovení TCP spojení a v případě, kdy zaznamenal velké množství nedokončených 3-way-handshake, by měl vyvolat alarm. Dále by měl nástroj zachytávat rovněž ARP pakety a UDP datagramy pro detekci ostatních skenovacích technik.

Programy eBPF

Technologii eBPF se podrobněji věnuje kapitola 3.5. Tento přístup pro získávání dat o systému je nejvíce preferovaný.

Mezi nevýhody patří obtížnější implementace, neboť tato technologie disponuje strmější učící křivkou. Pro účely této práce postačí implementace pouze malého podsystému monitorovacího nástroje s využitím této technologie.

Monitorovací nástroj může využít technologii eBPF pro detekci zavádění kernel modulů. Při spuštění příkazu `insmod` ze zavolá systémové volání `finit_module()`. Prostřednictvím eBPF programu se lze napojit na volání této funkce.

5.1.2 Metriky vytvořené nástrojem

Ze získaných dat lze generovat metriky o chování systému. Na základě těchto metrik pak lze rozhodnout o legitimnosti provozu na systému. Bezpečnostní systém pak může nežádoucí provoz detekovat pomocí detekce založené na anomáliích nebo podle detekce založené na signaturách.

Následující výčty se snaží popsat několik metrik, které umožňují charakterizovat taková nežádoucí chování systému, jež popsala předchozí kapitola 4.

[1] https://www.tcpdump.org/

Metriky založené na anomáliích

Tyto metriky jsou specifické v tom, že pro detekci nežádoucího provozu pomocí anomálií je nejprve nutné získat znalost normálního provozu systému. Na základě této znalosti pak lze vybrat prahovou hodnotu, po jejíž překročení se provoz klasifikuje jako nežádoucí.

Výčet těchto metrik je následující:

- Počet nedokončených TCP 3-way handshake – detekce skenování sítě pomocí TCP SYN datagramů a DoS útoku SYN flood.
- Počet TCP paketů, které neobsahují ani jeden z flagů RST, ACK nebo SYN – detekce skenování sítě.
- Počet unikátních IP adres, na které byly odeslány UDP datagramy – detekce skenování sítě a UDP flood.
- Počet různých UDP portů na jedné IP adrese, na kterou se poslaly UDP datagramy – detekce skenování sítě a UDP flood.
- Celkové přenesené množství dat, a to jak přijímaná data, tak i odeslaná – detekce nadměrného využití konektivity.
- Celkové využití CPU, v user-space i kernel-space – detekce nadměrného využití výpočetních zdrojů systému.

Metriky založené na signaturách

Jednodušší situace je u tohoto typu metrik, kde lze o legitimnosti provozu rozhodnout okamžitě:

- Zavedení kernel modulu – pokus o zavedení rootkitu.
- Otevření síťového portu ve stavu LISTEN – pokus o vytvoření zadních vrátek do systému.
- Vypnutí SELinux.

5.2 Integrace s monitorovacím systémem Prometheus

Prometheus je v současnosti již nasazené řešení pro monitorování několika služeb uvnitř Red Hatu včetně Testing Farm. Z tohoto důvodu se tak jedná o vhodné řešení pro integraci s metrikami vytvořeného monitorovacího nástroje.

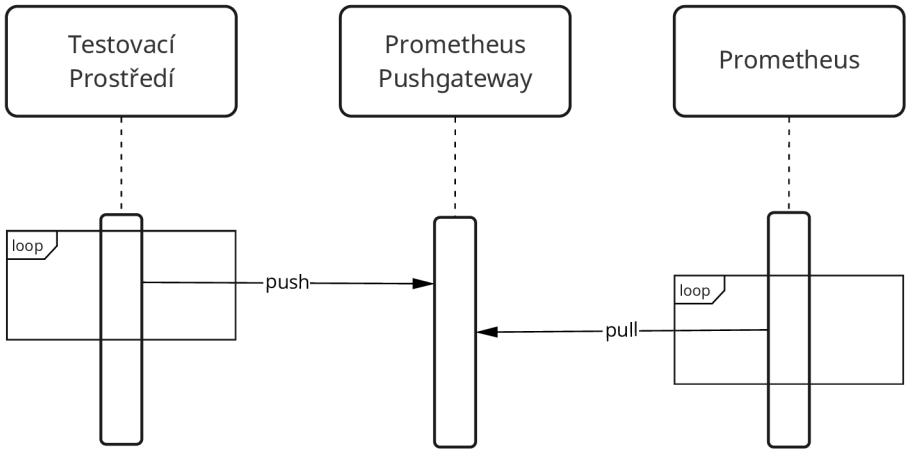
Veškeré metriky přichází do systému Prometheus tím způsobem, že si je sám stahuje ze zdrojů, které metriky vystavují. Je tedy zapotřebí tyto metriky nějakým způsobem vystavit.

Pokud by si Prometheus stahoval metriky přímo z testovacích strojů, znamenalo by to, že by na nich musel být otevřený síťový port, který by vystavoval vybrané metriky. Jelikož mezi jeden z požadavků patří co nejmenší zásah na chod testovacích strojů, bylo by vhodné se nutnosti otevření síťového portu vyvarovat.

5.2.1 Prometheus Pushgateway

Výše uvedené problémy řeší právě *Prometheus Pushgateway*. Tuto bránu lze využít jako dočasné úložiště pro metriky ze zdrojů, které jsou dočasného charakteru, což je případ testovacích strojů, jelikož v čase dynamicky vznikají a zanikají. Princip dočasného úložiště je založen na tom, že Prometheus sám metriky stahuje, není tak nutné data uchovávat dlouhou dobu. Jedná se o server, kam lze skrze HTTP požadavky odesílat metriky pro Prometheus bez nutnosti vystavování metrik na daném stroji[8].

Obrázek 5.1 zobrazuje sekvenční diagram, který ukazuje, jak by se měly metriky předávat směrem od testovacího prostředí do serveru Prometheus – na každém stroji testovacího prostředí poběží program, který v nekonečné smyčce bude odesílat metriky na Prometheus Pushgateway. Obdobně, Prometheus periodicky tyto metriky bude stahovat z Pushgateway.



Obrázek 5.1: Sekvenční diagram předávání metrik mezi testovacím prostředím, Prometheus Pushgateway a serverem Prometheus.

5.2.2 Alertmanager

Další prvek, s nímž by řešení mělo být integrované, je Alertmanager. Prostřednictvím tohoto subsystému lze vytvářet alarmy na základě pravidel, a ty následně eskalovat na IRC kanál.

Pomocí tohoto systému lze také implementovat omezující bezpečnostní opatření. Prostřednictvím Alertmanager Actions[2] lze vytvářet skripty pro vykonání v případě porušení bezpečnostního pravidla.

[2] https://github.com/little-angry-clouds/alertmanager-actions

5.3 Nedostatky navrženého řešení

Z povahy IDS/IPS přímo vychází, že neposkytují ochranu proti 100 % útoků, a ani tento systém nebude výjimkou. Je tedy vhodné se zamyslet nad nedostatky navrženého řešení.

5.3.1 Bezpečný provoz monitorovacího agenta

Jak již bylo zmíněno, uživatel spouštějící testy v službě Testing Farm má přístup uživatele `root` na testovací stroje. Současně na těchto strojích poběží monitorovací systém, který bude dohlížet na chod daného systému.

Z této skutečnosti plyne, že monitorovacího agenta lze jen obtížně provozovat v takovémto prostředí bezpečně – uživatel `root` samozřejmě má pravomoc provádět cokoliv v systému, včetně modifikace dat, které nástroj přenáší.

Jako možné řešení se zdá být to, že by se metriky odesílané na Prometheus kryptograficky podepisovaly. Nicméně tato možnost neřeší problém toho, že `root` by si mohl klíč k podepisování zjistit, a to i v případě, že by se nacházel pouze v paměti procesu monitorovacího agenta. Jediné vhodné řešení spočívá v omezení pravomocí uživatele `root`. Otázkou pro vývoj práce v budoucnu nicméně zbývá, zdali je toto vůbec technicky proveditelné.

Může se zdát, že technicky vhodnější řešení by bylo implementovat network-based IDS/IPS systém. Nicméně toho lze ve službě Testing Farm docílit obtížně. Jedním z důvodů je ten, že vývojáři Testing Farm nespravují cloudovou síť, ve které testovací stroje běží. Z těchto důvodů bylo rozhodnuto o vytvoření host-based IDS, navzdory jeho technickým nedostatkům.

Kapitola 6

Implementace monitorovacího nástroje

Tato kapitola se zabývá popisem realizačního výstupu práce. Jsou zde popsány technické detaily jednotlivých částí vytvořeného programu včetně použitého programovacího jazyka k implementaci a využitých knihoven. Dále jsou zde popsány některé problémy, které jsem během implementace řešil.

V rámci implementace vzniknul nástroj pojmenovaný *doggo* – „Guard Dog for Testing Farm written in Go“ s otevřeným zdrojovým kódem dostupným na veřejném serveru GitLab[1].

6.1 Výběr programovacího jazyka pro implementaci

Jak bylo zmíněno v předchozí kapitole 5, pro implementaci monitorovacího nástroje je vhodné vytvořit jednu binárku, která bude spuštěná na testovacích strojích. Pro tento účel je nezbytné využít některý kompilovaný jazyk. Prvními kandidáty vhodnými pro tuto práci jsou jazyky C/C++, u nichž lze ocenit zejména jejich efektivitu. Nicméně vzhledem ke službám, jako je například Prometheus, se kterými má program komunikovat, je tak vhodnou volbou pro implementaci jazyk *Go* ve verzi `1.16.14`.

Mezi některé základní vlastnosti jazyka Go patří například:

- Kompilovaný jazyk, jehož zdrojový kód se překládá do strojového kódu, a tak není nutný virtuální stroj pro běh programů.
- Podporuje vícero programovacích paradigmat – imperativní, strukturované a objektově orientované.
- Bohatá podpora funkcí ve standardních knihovnách jazyka a široký výběr z existujících knihoven třetích stran.
- Dobrá podpora souběhu pomocí gorutin.
- Výkon lze přirovnat k programům napsaných v C/C++.

[1] https://gitlab.com/testing-farm/doggo

6.1.1 Popis využitých knihoven třetích stran

Tato část popisuje moduly jazyka Go, které byly použity pro implementaci nástroje a které současně nepatří mezi standardní knihovny tohoto jazyka.

github.com/google/gopacket

Knihovna `github.com/google/gopacket`[2] slouží k odchytávání síťových paketů. K tomuto účelu využívá knihovnu *libpcap*. Tato knihovna běží v user-space, nicméně využívá BPF (Berkeley Packet Filter), který běží v kernel-space. Tato skutečnost poskytuje velmi efektivní přístup k paketům, jelikož se kopírují přímo z paměti jádra, do které fyzická síťová zařízení zapisují.

github.com/prometheus/procfs

Modul `github.com/prometheus/procfs`[3] poskytuje abstrakci nad virtuálními filesystémy `/proc` a `/sys`. Pomocí funkcí jazyka *Go* je tak možné získat informace o systému právě z těchto zdrojů, jako například vytížení CPU, paměti, data o běžících procesech, využití sítě, apod.

github.com/prometheus/client_golang

Knihovna `github.com/prometheus/client_golang`[4] poskytuje klientské rozhraní pro práci s monitorovacím systémem Prometheus. Pomocí knihovních funkcí je tak možné například komunikovat s Prometheus Pushgateway, tedy odesílat metriky prostřednictvím jejího HTTP API.

github.com/cilium/ebpf

Balík `github.com/cilium/ebpf`[5] poskytuje sadu nástrojů pro práci s eBPF. Pomocí kódu jazyka Go je tak umožněno zavádět eBPF programy do kernel-space a také s nimi za běhu komunikovat prostřednictvím map. Představením technologie eBPF se podrobněji zabývá kapitola 3.5.

6.2 Popis implementace modulů

Nástroj je členěn do několika logicky oddělených celků – hlavní modul a moduly implementující sondy, které sbírají metriky ze systému. Tato sekce se věnuje jejich popisu.

6.2.1 Modul main

Tok hlavního modulu implementovaného ve funkci `main()` je následující:

1. Zkontrolují se přepínače příkazového řádku, které se použijí pro ustanovení spojení s Prometheus Pushgateway.
2. Inicializují se metriky pro Prometheus Pushgateway.

[2] https://pkg.go.dev/github.com/google/gopacket@v1.1.19
[3] https://pkg.go.dev/github.com/prometheus/procfs@v0.7.3
[4] https://pkg.go.dev/github.com/prometheus/client_golang@v1.12.1
[5] https://pkg.go.dev/github.com/cilium/ebpf@v0.8.1

3. Inicializují se sondy zavoláním jejich metody `Init()`.
4. Spustí se gorutina paketového filtru.
5. Spustí se nekonečný cyklus s opakováním iterací každých pět sekund. V každé iteraci dojde k aktualizaci jednotlivých sond pomocí jejich metod `Update()` a dále také k odeslání metrik na Prometheus Pushgateway.

Gorutina paketového filtru pak probíhá následovně:

1. Z filesystému `/proc` se získají názvy všech síťových rozhraní.
2. Pro každé toto rozhraní (kromě *loopback* a rozhraní ve stavu *down*) se vytvoří handler modulu `pcap` pro odposlouchávání na rozhraní.
3. Pro každý handler se zavolá gorutina která donekonečna čeká na příchozí pakety, které následně předá odpovídajícím síťovým sondám pro zpracování.

6.2.2 Sonda pro detekci otevřených síťových portů

Sondu implementuje struktura `ProbeListeningTCPPorts` v souboru `probe_net_ports.go`. Struktura ve své metodě `Fetch()` čte prostřednictvím modulu `procfs` ze souboru `/proc/net/tcp`, jež obsahuje informace o všech otevřených TCP portech. Sonda si vyfiltruje pouze ty, které se nachází ve stavu 10, neboli *LISTEN*.

Struktura periodicky exportuje následující metriky:

- `doggo_metric_total_listening_tcp_ports` – celkový počet otevřených TCP portů ve stavu *LISTEN* na stroji,
- `doggo_metric_new_listening_tcp_ports_since_start` – počet otevřených TCP portů ve stavu *LISTEN* od počátku spuštění nástroje.

6.2.3 Sonda detekující skenování sítě

Tuto sondu implementuje struktura `ProbeNetScan` v souboru `probe_net_scan.go`. Sonda umožňuje detekci celkem dvou skenovacích technik, a to ARP sken a TCP NULL sken.

Sonda si v sobě uchovává data o tom, kolikrát za minutu byly na síti zachyceny ARP pakety nebo TCP datagramy, jež neobsahují ani jeden z příznaků RST, SYN nebo ACK. Tyto dvě hodnoty si ukládá do pomocných struktur.

Pomocnou datovou strukturu tvoří cyklický buffer implementovaný ve strutuře `RingBufferInt`, nad kterou jsou implementované následující metody:

- `Init(size int)` – inicializuje strukturu a nastaví jí velikost bufferu.
- `Rotate()` – provede rotaci ukazatele na následující pozici a danou pozici vynuluje.
- `Inc(n int)` – inkrementuje prvek, na který ukazuje ukazatel, o hodnotu `n`.
- `GetAll() int` – vrátí sumu všech hodnot uvnitř bufferu.

Prometheus metriky vytvářené strukturou jsou následující:

- `doggo_metric_arp_packets_per_minute` – počet zachycených ARP paketů za minutu
- `doggo_metric_tcp_null_packets_per_minute` – počet zachycených TCP datagramů za minutu, které neobsahovaly ani jeden z flagů *ACK*, *RST* a *SYN*.

6.2.4 Sonda zachycující navázání TCP spojení

Sondu implementuje struktura `TCPHandshake` v souboru `probe_net_tcp_handshake.go`. Sonda detekuje počet TCP spojení za poslední minutu, která nebyla uskutečněná do konce, tedy pokud mezi klientem a serverem neproběhly kroky TCP 3-way-handshake výměnou datagramů s příznaky SYN, SYN+ACK, ACK. Tímto způsobem lze detekovat několik typů nežádoucího provozu, a to skenování sítě pomocí TCP a také SYN Flood útok.

Sonda využívá dvě pomocné struktury, první z nich je `TCPHandshake`, která obsahuje tři položky, každá reprezentující jednu část 3-way-handshake. Pomocí struktury tak lze rozlišit, zda se spojení ustanovilo do konce, či nikoliv.

Tato struktura je vkládána opět do cyklického bufferu `RingBufferTCPHandshake`. Tímto způsobem se ukládají všechny pokusy o ustanovení spojení za posledních 60 sekund. Tato pomocná struktura implementuje stejné chování jako struktura `RingBufferInt` z předchozí sondy, ovšem v tomto případě namísto datového typu `int` ukládá typ `[]*TCPHandshake`.

Může se tak zdát, že implementace stejného chování dvakrát je zbytečně redundantní. Jazyk Go však ve verzi 1.16 nepodporuje způsob pro implementaci funkcí stejného chování nad odlišnými datovými typy, což je poněkud překvapující u takto rozšířeného jazyka. Tuto vlastnost přidává až nová verze 1.18 v podobě generických datových typů.

Sonda zpracovává každý TCP datagram v metodě `ParseTCPDatagram()`, kde může nastat jedna z tří možností, pokud se jedná o paket ustanovující spojení:

1. Datagram obsahuje příznak SYN, ale ne ACK – datagram se uloží do nově vytvořené struktury typu `TCPHandshake` a tato struktura se přidá do cyklického bufferu.

2. Datagram obsahuje příznak SYN a ACK – dle acknowledgement čísla se v kruhovém bufferu nalezne odpovídající struktura `TCPHandshake`. Pokud byla úspěšně nalezena, vloží se tam aktuálně příchozí datagram jako druhý datagram tvořící handshake.

3. Datagram obsahuje příznak ACK – opět dle acknowledgement čísla se v kruhovém bufferu nalezne odpovídající struktura `TCPHandshake`. Pokud byla nalezena, přidá se tam i tento datagram, čímž došlo k ustanovení TCP spojení.

Sonda exportuje následující metriku:

- `doggo_metric_incomplete_tcp_handshakes_per_minute` – počet neúplných ustanovení TCP spojení za minutu.

6.2.5 Sonda zachycující UDP datagramy

Sonda je implementovaná ve struktuře `ProbeUDP` v souboru `probe_net_udp.go`. Sonda má za úkol sledovat, na jaké IP adresy a na jaké jejich porty se odesílají UDP datagramy. Tato sonda umožňuje detekovat skenování sítě pomocí UDP.

Tyto informace se ukládají do hashovací tabulky, která je v jazyku Go implementovaná v datovém typu `map`.

Odchozí UDP datagramy se ukládají do datové struktury `map`, která má jako vyhledávající klíč typ `uint32` a jako hodnotu další typ `map`. Klíč `uint32` reprezentuje IPv4 adresu. Hodnoty, tedy vnořené mapy, jako klíč používají typ `uint16` reprezentující číslo UDP portu a jako hodnotu typ `bool`, který nese informaci o tom, zda byl na daný port odeslán datagram.

Sonda exportuje dvě metriky:

- `doggo_metric_udp_unique_target_ips` – počet unikátních cílových IP adres, na které se posílaly UDP datagramy,
- `doggo_metric_udp_max_unique_target_ports` – maximální počet cílových portů v rámci jedné cílové IP adresy, na kterou se zasílaly UDP datagramy.

6.2.6 Sonda detekující zavádění kernel modulů

Tato sonda využívá technologii eBPF a je tak rozdělena do dvou částí:

- Struktura `ProbeKernelModule` definovaná v souboru `probe_ebpf.go`,
- eBPF program v souboru `probe_ebpf.c`.

BPF program je typu *kprobe*, který se při spuštění napojí na volání kernelové funkce `sys_finit_module()`. Program definuje jedinou mapu, a to typu `BPF_MAP_TYPE_ARRAY` – tedy obyčejné pole, ve kterém se nachází pouze jediná položka typu `unisgned int`. Právě při každém zaznamenaném volání systémové funkce se tato číslená hodnota inkrementuje.

Struktura `ProbeKernelModule` běžící v user-space při každém volání metody `Update()` přistoupí do eBPF mapy a přečte si tuto jedinou hodnotu, která se v ní nachází. Posléze exportuje následující metriku pro Prometheus:

- `doggo_metric_times_sys_finit_module_called` – počet volání systémové funkce `sys_finit_module()`.

V tuto chvíli bylo také nutné vyřešit způsob překládání implementovaného programu. Jak již zaznělo v sekci 3.5, eBPF programy se kompilují do bytekódu, který je posléze nutný z user-space zavést pomocí systémového volání `bpf()`. Z tohoto výkladu plyne, že výsledný program se skládá minimálně ze dvou souborů – user-space program a bytekód s eBPF programem, který se čte a zavádí do kernel-space při runtime user-space programu. Tento přístup by však byl v rozporu s návrhem, který by si přál zabalit výsledné řešení do jediné binárky.

Pro jazyk Go však existuje utilita `bpf2go`[6], která tento problém řeší. Tento nástroj umožňuje vkládat eBPF programy do Go programů. Nástroj nejprve přeloží eBPF program (napsaný v jazyku C) do bytekódu a současně vygeneruje Go soubory, které tento bytekód načítají. Posléze lze teprve spustit Go překladač, který vše přeloží do jediné výsledné binárky.

6.2.7 Sonda detekující vypnutí SELinux

Sondu implementuje struktura `ProbeSELinux` v souboru `probe_selinux.go`. Struktura při zavolání své metody `Update()` přečte soubor `/sys/fs/selinux/enforce`, jež obsahuje informaci o tom, zda je SELinux zapnutý. Jediným obsahem tohoto souboru je ASCII hodnota `0` nebo `1`.

Sonda exportuje jednu metriku:

- `doggo_metric_selinux_enabled` – hodnota 0 nebo 1 udávající stav SELinux.

[6] https://pkg.go.dev/github.com/cilium/ebpf/cmd/bpf2go

6.2.8 Další metriky

Program exportuje několik metrik, jejichž získání je dostatečně jednoduché na to, aby byly zapouzdřeny ve struktuře s metodami. Výčet těchto metrik je následující:

- `doggo_metric_load_avg_1`, `doggo_metric_load_avg_5`, `doggo_metric_load_avg_15` – procentuální vytížení procesoru za posledních 1, 5 a 15 minut,
- `doggo_metric_cpu_time_user` a `doggo_metric_cpu_time_kernel` – počet sekund, které procesor strávil vykonáváním user-space, resp. kernel-space programů,
- `doggo_metric_received_megabytes` a `doggo_metric_transmitted_megabytes` – počet přijatých, resp. odeslaných dat v MB.

6.3 Rozhraní příkazového řádku

V jazyku Go lze snadno implementovat přepínače pro příkazovou řádku prostřednictvím standardní knihovny `flag`. Při spouštění nástroje *doggo* pomocí příkazové řádky je zapotřebí dodat dva povinné přepínače:

- `-pushgateway-url URL` – URL Prometheus Pusghateway, na kterou budou odesílány metriky.
- `-request-id ID` – unikátní identifikátor stroje v rámci Pushgateway pro označení metrik z jednoho zdroje. Na produkčním nasazení se zde bude používat identifikátor testovacího požadavku.

Pozn.: U dlouhých přepínačů příkazové řádky (jejichž název je delší než jeden znak) bývá zvykem psát dvě pomlčky (např. `--option`), což potvrzuje například referenční manuál knihovny GNU C[3]. Nicméně konvence jazyka Go jdou proti proudu tohoto zvyku a v dokumentacích uvádí u přepínačů pomlčku jednu. Nicméně obě varianty zápisu (s jednou i dvěma pomlčkami) jsou v Go programech přijímané.

6.4 Integrace s monitorovacím systémem Prometheus

Jak již bylo zmíněno v předchozích sekcích, integrace s Prometheus Pushgateway probíhá prostřednictvím modulu `github.com/prometheus/client_golang`, jež poskytuje rozhraní pro práci s Prometheus Pushgateway.

Volání `pusher := push.New(*pushgateway_url_ptr, *request_id_ptr)` předá URL běžícího Prometheus Pushgateway a unikátní identifikátor (v Testing Farm se bude používat identifikátor testovacího požadavku) vytvoří strukturu umožňující přidávat a odesílat metriky na Pushgateway.

Metoda `prometheus.NewGauge(prometheus.GaugeOpts{Name: "example_name"})` vytváří Prometheus metriku typu *Gauge*. Všechny metriky v nástroji jsou tohoto typu.

Voláním metody `pusher.Collector(metric)` lze přidat metriku k odeslání na Pushgateway.

Konečně, všechny metriky se na Pushgateway odesílají metodou `pusher.Push()`, která se periodicky volá v nekonečném cyklu hlavní funkce `main()`.

Kapitola 7

Nasazení a testování implementovaného řešení

V předchozí kapitole 6 byla popsána implementace monitorovacího nástroje *doggo* a způsob, kterým se integruje s monitorovací platformou Prometheus. Neodlučitelnou součástí vývoje programů je také jejich nasazení a provoz. Proto se tato kapitola věnuje jednak překladu programu tak, aby byl spustitelný na všech potřebných prostředích testovacích strojů a také způsob, kterým se na testovací stroje nasazuje. Následně se také vyhodnocuje přínos vytvořeného řešení pro službu Testing Farm.

7.1 Postup nasazení nástroje do Testing Farm

Tato sekce se skládá ze dvou kroků. V první řadě je zapotřebí mít co nasadit, proto se první část věnuje překladu programu. Druhá část sekce pak popisuje samotné nasazení.

7.1.1 Překlad nástroje

Vzhledem k tomu, že služba Testing Farm poskytuje testovací stroje ve dvou architekturách (*x86_64* a *aarch64*), je zapotřebí program překládat do dvou architektur. Překladač jazyka Go poskytuje křížové překládání pomocí proměnných prostředí `GOOS` a `GOARCH`, které v tomto případě budou nastavené na `GOOS=linux` a `GOARCH=amd64` nebo `GOARCH=arm64` (pozn.: *x86_64* a *amd64* jsou různá označení pro stejnou architekturu procesoru, stejně tak jako *aarch64* a *arm64*).

Tento křížový překlad do cílové architektury *arm64* se však neobejde bez problémů, neboť modul `github.com/google/gopacket` při překladu využívá systémové knihovny *libpcap* a *libpcap-devel*, které vyžadují přítomnost zkompilované verze cílové architektury.

Další problém nastal při běhu programu z důvodu běhových závislostí programu. Implicitně, překladač jazyka Go kompiluje programy se statickým linkováním knihoven, pokud je to možné. Nicméně mnou vytvořený program nelze linkovat staticky z důvodu přítomnosti knihoven *libpcap* a *glibc*. Implicitně se tedy kompiluje s dynamickým linkováním knihoven, což s sebou nese určitá úskalí:

- Na čisté instalaci distribuce CentOS se nenachází knihovna *libpcap*.
- Verze knihovny *glibc* mezi dvojicí distribucí např. CentOS a Fedora jsou natolik rozdílné, že vytvořit jednu binárku kompatibilní s oběma operačními systémy je obtížný úkol.

Vzhledem k požadavkům z návrhu je však preferována varianta se statickým linkováním, neboť je tak možné vytvořit přenositelnou binárku, která je v rámci jedné architektury spustitelná na všech distribucích.

Ovšem statické linkování knihovny *glibc*, která implementuje standardní funkce jazyka C a implicitně se nachází v distribucích např. Fedora a CentOS, je obtížné, a to záměrně kvůli licenčním podmínkám knihovny. Naštěstí k *glibc* existuje alternativa v podobě knihovny *musl*[1], která implementuje standardní knihovní funkce jazyka C. Tuto knihovnu již adoptovaly některé distribuce namísto *glibc*, a to například *Alpine Linux*[15].

Výsledný překlad nástroje probíhá přibližně v následujících několika krocích:

1. Z `https://musl.libc.org/releases` stáhni a zkompiluj *musl-gcc* toolchain.
2. Z `https://www.tcpdump.org/release/` stáhni a přelož zdrojové kódy knihovny *libpcap* pomocí *musl-gcc*.
3. Zkompilovanou knihovnu *libpcap.a* zkopíruj do *musl-gcc*.
4. Z `https://musl.cc/` stáhni křížový překladač pro *arm64* *aarch64-linux-musl-cross*.
5. Pomocí překladače *aarch64-linux-musl-cross* zkompiluj ještě jednou knihovnu *libpcap* a přeloženou knihovnu *libpcap.a* zkopíruj do knihoven překladače *aarch64-linux-musl-cross*.
6. Přelož nástroj *doggo* se statickým linkováním knihoven do architektur *amd64* a *arm64*.

Pro automatizovaný překlad nástroje do binárních souborů pro různá běhová prostředí a zároveň pro správu vydaných verzí se používá GitLab CI/CD. Definice jednotlivých úloh se nachází v souboru `.gitlab-ci.yaml`. Výše uvedený postup překladu nástroje se nachází v sekci *build*.

7.1.2 Nasazení do produkčního prostředí

Předchozí část popsala kompilaci binárky se staticky linkovanými knihovnami. Další krok je nasadit tento program do produkčního prostředí.

Pro automatizovanou instalaci potřebných závislostí na testovacích stroje v produkčním prostředí Testing Farm se využívá nástroj Ansible. Pro tyto účely existují dvě konfigurace ansible-playbook – *pre-artifact-installation* a *post-artifact-installation*, tedy podle toho, zda se jedná o instalaci před nebo po instalaci testovaných artefaktů. Jelikož je žádoucí nainstalovat a spustit monitorovací nástroj až jako poslední krok instalace, využita je první možnost, a to z důvodu, aby se monitorovala pouze aktivita uživatele, nikoliv režie Testing Farm. Výpis 7.1 tuto instalaci zobrazuje. Pro spuštění nástroje, jak udává 6.3, je zapotřebí dodat dva parametry. První je adresa běžící služby Prometheus Pushgateway – tato URL je statická pro celou službu, a může tak být pevně zadaná. Druhým parametrem je identifikátor testovacího požadavku – ten lze dodat pomocí proměnné `TESTING_FARM_REQUEST_ID`, která se v tomto Ansible playbook nachází.

```
- name: Download doggo binary
  get_url:
    url: "https://gitlab.com/testing-farm/doggo/-/releases/permalink/
      latest/downloads/doggo-amd64"
```

[1] `https://musl.libc.org/`

```
      dest: "/usr/local/bin/doggo"
      timeout: 30
      mode: +x
    retries: 5
    until: result_download_doggo is succeeded
    register: result_download_doggo

- name: "Start doggo"
  shell: nohup /usr/local/bin/doggo \
    --pushgateway-url http://pushgateway.dev.testing-farm.io:9091/ \
    --request-id {{ TESTING_FARM_REQUEST_ID }} &
```

Výpis 7.1: Část konfigurace ansible-playbook zobrazující instalaci monitorovacího nástroje na testovací stroje.

7.2 Nasazení Prometheus Pushgateway

V předchozí kapitole se implementoval nástroj, jež komunikuje s Prometheus Pushgateway, nicméně dosud nebylo vysvětleno, jakým způsobem byla tato komponenta nasazena. Stejně jako samotný Prometheus server a Alertmanager, i v tomto případě se Pushgateway nasadil v podobě kontejneru na Kubernetes cluster dostupný z veřejného internetu.

Vypisovat zde celou konfiguraci nasazení nedává příliš smysl, nicméně mezi podstatnou část konfigurace patří verze image, tu tvoří `quay.io/prometheus/pushgateway:v1.4.2`.

Webové rozhraní, které je již naplněno metrikami z testovacích požadavků produkčního prostředí, ukazuje obrázek 7.1.

7.2.1 Integrace se serverem Prometheus

Předchozí část nasadila Pushgateway na Kubernetes cluster, další krok je propojit Prometheus server a Pushgateway, aby začal shrabovat metriky. Pro integraci postačí přidat záznam do konfiguračního souboru[2] serverové části, to zobrazuje výpis 7.2.

```
- job_name: 'tft-public-pushgateway'
  scheme: 'http'
  metrics_path: '/metrics'
  static_configs:
    - targets:
      - 'pushgateway.dev.testing-farm.io:9091'
```

Výpis 7.2: Integrace Prometheus serveru s Prometheus Pushgateway.

7.3 Vyhodnocení normálního provozu

Tato část ilustruje normální provoz služby Testing Farm vyjádřený v metrikách, které generuje monitorovací nástroj *doggo*. V této sekci byl porovnáván provoz 3470 testovacích požadavků.

[2] https://prometheus.io/docs/prometheus/latest/configuration/configuration/

Obrázek 7.1: Ukázka webového rozhraní Prometheus Pushgateway.

Tabulka 7.1 zobrazuje naměřené hodnoty různých metrik z normálního provozu. Uvedená data byla získána dotazy na Prometheus server pomocí jazyka *PromQL*. Příklad takovéhoto dotazu je následující: `avg(max_over_time(doggo_metric_cpu_time_kernel[1h]))`

Postup výpočtu jednotlivých agregovaných hodnot pro každou metriku je následující:

1. Z průběhu metriky v čase každého testovacího požadavku zjisti maximální hodnotu (nejhorší případ).

2. Na tyto hodnoty aplikuj zvolenou agregační funkci.

Při analýze naměřených hodnot bylo zjištěno, že hodnoty reprezentující množství přenesených dat nejsou přesné. Nevystihují totiž pouze využití systému, které způsobil samotný uživatel. Ačkoliv se monitorovací nástroj spouští až na samotném konci instalace testovacího prostředí, stále běží až do vypnutí testovacího stroje.

Název metriky	Průměr	Max	Medián	Min
cpu_time_user	468.44	23776.22	376.26	13.95
cpu_time_kernel	256.48	23313.02	122.78	12
received_megabytes	1008.26	171449	489	18
transmitted_megabytes	78.59	5588	4	0
udp_unique_target_ips	5.06	31	6	0
udp_max_unique_target_ports	0.92	1	1	0
incomplete_tcp_handshakes_per_minute	0.08	20	0	0
arp_packets_per_minute	1.70	30	2	0
tcp_null_packets_per_minute	0	0	0	0

Tabulka 7.1: Agregované hodnoty metrik všech naměřených testovacích požadavků.

7.3.1 Nastavení upozornění v Alertmanager

Nyní se metriky dostávají na Prometheus server, jako další krok lze nastavit upozornění na metriky překračující určité meze. Ukázka definice jednoho takového upozornění se nachází ve výpise 7.3.

```
- alert: TFT_DoggoDoS
  expr: doggo_metric_incomplete_tcp_handshakes_per_minute > 100
  labels:
    severity: warning
    service: testing-farm
  annotations:
    summary: |
      Possible DoS or network scan attempt detected for
      https://api.dev.testing-farm.io/v0.1/requests/{{ $labels.job }}
    description: |
      Detected more than 100 incomplete TCP handshakes. This could mean
      a Denial of Service attack or network scan from the test machine.
```

Výpis 7.3: Ukázka konfigurace upozornění nad hodnotami metrik v systému Prometheus.

Obdobným stylem lze nastavit upozornění nad hodnotami dalších metrik. Na základě měření provedeného v předchozí části byly prahové hodnoty zvoleny následovně (pokud je porušena podmínka, vytvoří se upozornění):

- `doggo_metric_udp_max_unique_target_ports > 50`,
- `doggo_metric_incomplete_tcp_handshakes_per_minute > 100`,
- `doggo_metric_arp_packets_per_minute > 100`,
- `doggo_metric_tcp_null_packets_per_minute > 10`,
- `doggo_metric_times_sys_finit_module_called != 0`,
- `doggo_metric_selinux_enabled != 1`,
- `doggo_metric_new_listening_tcp_ports_since_start != 0`.

Úplná podoba testovacích pravidel se nachází v souboru `testing-farm.rules`.

Jak je z výše uvedeného výčtu patrné, ne všechny metriky implementované v kapitole 6 byly využity pro vytváření upozornění. Přidání dalších upozornění je tak předmětem budoucího vývoje této práce. Proces vytváření pravidel pro upozorňování je dlouhodobého charakteru, bude tak nějakou dobu trvat, než se pravidla odladí.

7.4 Testování implementovaného systému

Nyní jsou všechny potřebné části propojeny včetně generování alarmů. Poslední chybějící krok je vyzkoušet implementovaný systém, zda funguje na produkčním nasazení dle očekávání. Pro tento účel lze vytvořit test ve formátu FMF a následně ho odeslat na API Testing Farm. Tento test, snažící se o spuštění co nejvíce upozornění, se nachází ve výpise 7.4.

```
summary: Trigger all alerts
provision:
  how: virtual
  image: Fedora-Rawhide
discover:
  how: shell
  tests:
    - name: install
      test: dnf install -y nmap kernel-devel-$(uname -r)

    - name: open-port
      test: nohup python -m http.server &> ~/python.out &

    - name: disable-selinux
      test: setenforce 0

    - name: scan-tcp
      test: nmap scanme.nmap.org

    - name: scan-tcp-null
      test: nmap -sN scanme.nmap.org

    - name: scan-udp
      test: nmap -sU -p0-150 scanme.nmap.org

    - name: insert-module
      test: |
        cd module && make && insmod example.ko
        dmesg | grep Hello
        rmmod example
        dmesg | grep Goodbye

execute:
  how: tmt
```

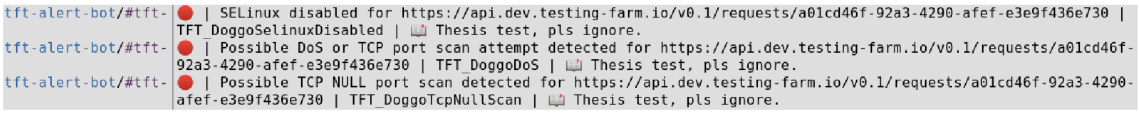
Výpis 7.4: Testovací úloha s několika typy nežádoucích provozů.

Výše uvedenou testovací úlohu lze odeslat na Testing Farm, podobu HTTP požadavku na API zobrazuje výpis 7.5. Jako HTTP klient je využita utilita *HTTPie*[3]. Tajný API klíč uživatele se nachází v proměnné `key`.

```
http POST https://api.dev.testing-farm.io/v0.1/requests \
    api_key=$key
    test[fmf][url]=https://gitlab.com/janhavlin/tf-tests/ \
    test[fmf][ref]=main \
    environments[0][arch]=x86_64 \
    environments[0][os][compose]=Fedora-Rawhide
```

Výpis 7.5: Odeslání požadavku na API Testing Farm.

Odesláním výše uvedeného testovacího požadavku na službu došlo ke spuštění několika alarmů. Ukázku několika těchto upozornění odeslaných na IRC kanál zobrazuje obrázek 7.2.



Obrázek 7.2: Ukázka upozornění na IRC kanále.

Při testování systému na produkčním nasazení služby, navzdory korektnímu chování při testování na lokálním počítači, se odhalily nedostatky v podobě falešně pozitivního a falešně negativního upozornění. První varianta se týká UDP portů, kdy monitorovací nástroj v některých testovacích případech hlásí mnoho různých UDP portů používaných pro komunikaci s jednou IP adresou. Falešně negativní upozornění spočívá v chybné detekci zavedení kernel modulu.

Nezbývá než konstatovat, že v produkčním nasazení se vytvořený systém nechová zcela dle očekávání. Proces nasazení změn na každý produkční systém bývá téměř vždy doprovázen neočekávaným chováním.

7.4.1 Zhodnocení

Mezi významný přínos této práce patří zlepšení monitorování služby Testing Farm. Prostřednictvím implementovaného systému lze získávat znalosti o dění ve službě. Množina současně získávaných metrik z testovacích strojů však není konečná, počítá se s rozšířením monitorovacího systému o další metriky.

Budoucí vývoj této práce zahrnuje implementaci omezujícího bezpečnostního opatření v případě detekce útoku. Ze strany tohoto implementovaného systému se nejedná o obtížný úkol – lze využít modul *Alertmanger Actions* pro přidání akcí při detekci podezřelého chování systému. Překážku v současné době tvoří aplikační rozhraní Testing Farm, které neumožňuje provést omezení dopadu. Zapotřebí je tedy rozšířit aplikační rozhraní Testing Farm, aby podporovalo zrušení právě vykonávaného testovacího požadavku a zamezení uživateli ve vytváření nových testovacích požadavků.

Další vize do budoucna tohoto systému spočívá ve větším rozšíření systému prostřednictvím eBPF sond. Současné řešení, ačkoliv funkční, spočívá převážně v implementaci programu, který běží v user-space. Využitím eBPF, tedy implementací a zavedením dalších programů do kernel-space, by obecně došlo ke zkvalitnění monitorovacího systému a zároveň vytvoření obtížnějších překážek k prolomení útočníkem.

[3] https://httpie.io/

Kapitola 8

Závěr

Práce se zabývala bezpečností služby Testing Farm ve firmě Red Hat. Konkrétně se jednalo o analýzu možností zneužití strojů testovacího prostředí a následnému implementování monitorovacího systému pro detekci tohoto nežádoucího provozu. Možnost zneužití testovacích strojů pramení z toho, že uživatelům je umožněno ve službě spouštět libovolný kód jako privilegovaný uživatel root.

V rámci implementační části práce se podařilo navrhnout a vytvořit monitorovací nástroj pro detekci několika způsobů zneužití testovacích strojů. Tento systém sleduje přenášené síťové pakety, systémové zdroje a nastavení. Na základě získávání informací z těchto zdrojů generuje metriky, které se odesílají na monitorovací server Prometheus. Z metrik je možné detekovat například síťové útoky jako SYN Flood nebo skenování sítí. Prostřednictvím technologie eBPF se na úrovni jádra monitoruje zavádění kernel modulů, což se považuje za nežádoucí akce.

Mezi úspěchy práce lze jistě zařadit skutečnost, že se implementovaný systém podařilo nasadit do produkčního prostředí služby Testing Farm, navzdory tomu, že se v něm objevily nedostatky řešení v podobě falešně pozitivního a falešně negativního poplachu. Dále se povedlo systém provázat se současně využívanými systémy pro monitorování a správu upozornění, konkrétně Prometheus a Alertmanager. Odesláním testovacího požadavku s nežádoucím kódem na Testing Farm API se podařilo ověřit, že upozornění fungují a odesílají se na IRC kanál Testing Farm týmu.

Jedna z věcí, která nebyla implementována, je implementace omezujících bezpečnostních opatření při detekci nežádoucího provozu na testovacím stroji. Ačkoliv reakce na útok lze v systému Alertmanger provést poměrně snadno, problém spočíval v tom, že služba Testing Farm v současnosti neobsahuje aplikační rozhraní pro zrušení aktuálně probíhajících testovacích úloh. Tato část práce tak zbývá pro budoucí vývoj systému.

Další z možných směrů budoucího vývoje spočívá ve zvýšení granularity monitorování systému. V současnosti se sleduje každá testovací úloha individuálně, nicméně vhodný přístup by byl sledovat najednou všechny testovací úlohy od jednoho uživatele. To by mohlo snížit dopady potenciálního útoku, pokud by se útočník rozhodl distribuovat útok do několika menších testovacích požadavků, které by individuálně nemusely být vyhodnoceny jako nežádoucí.

Literatura

[1] *EBPF Documentation* [online]. [cit. 2022-04-28]. Dostupné z: https://ebpf.io/what-is-ebpf.

[2] *Filesystems in the Linux kernel: The /proc Filesystem* [online]. [cit. 2022-05-05]. Dostupné z: https://www.kernel.org/doc/html/latest/filesystems/proc.html.

[3] *The GNU C Library Reference Manual* [online]. [cit. 2022-04-29]. Dostupné z: https://www.gnu.org/software/libc/manual/html_node/index.html.

[4] *Linux Programmer's Manual: BPF* [online]. [cit. 2022-05-05]. Dostupné z: https://man7.org/linux/man-pages/man2/bpf.2.html.

[5] *Standard Test Interface Documentation* [online]. [cit. 2022-02-08]. Dostupné z: https://docs.fedoraproject.org/en-US/ci/standard-test-interface/.

[6] *Transmission Control Protocol* [RFC 793]. RFC Editor, září 1981 [cit. 2022-03-29]. DOI: 10.17487/RFC0793. Dostupné z: https://www.rfc-editor.org/info/rfc793.

[7] *OWASP Top 10 Project* [online]. 2021 [cit. 2021-12-30]. Dostupné z: https://owasp.org/www-project-top-ten/.

[8] Authors, P. *Prometheus Documentation* [online]. 2022 [cit. 2022-04-17]. Dostupné z: https://prometheus.io/docs/.

[9] Axelsson, S. Intrusion Detection Systems: A Survey and Taxonomy. Duben 2000. Dostupné z: https://www.researchgate.net/publication/2597023_Intrusion_Detection_Systems_A_Survey_and_Taxonomy.

[10] Bishop, M. *Introduction to Computer Security*. Addison-Wesley, 2005 [cit. 2022-03-19]. ISBN 0-321-24744-2.

[11] Calavera, D. a Fontana, L. *Linux Observability with BPF*. O'Reilly, 2020 [cit. 2022-03-20]. ISBN 978-1-492-05020-9.

[12] Dorosz, P. K. P. Intrusion Detection Systems (IDS) Part I – (network intrusions; attack symptoms; IDS tasks; and IDS architecture). Duben 2003. Dostupné z: https://techgenix.com/Intrusion_Detection_Systems_IDS_Part_I__network_intrusions_attack_symptoms_IDS_tasks_and_IDS_architecture/.

[13] Dubaj, O. *Systém pro správu výsledků testů doplňující nástroj tmt*. 2021. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Dostupné z: https://www.vutbr.cz/studenti/zav-prace/detail/136799.

[14] Hanáček, P. a Staudek, J. *Bezpečnost informačních systémů*. 2000. 127 s. ÚSIS. ISBN 80-238-5400-3.

[15] Honnef, D. Statically compiled Go programs, always, even with cgo, using musl. Červen 2015, [cit. 2022-04-29]. Dostupné z: https://honnef.co/posts/2015/06/statically_compiled_go_programs__always__even_with_cgo__using_musl/.

[16] Klusoň, M. *Podpora průběžné integrace v rámci systému Copr*. 2018. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Dostupné z: https://www.vutbr.cz/studenti/zav-prace/detail/114915.

[17] Lyon, G. *Nmap Network Scanning*. 2008 [cit. 2022-03-20]. ISBN 978-0-9799587-1-7.

[18] McCanne, S. a Jacobso, V. Intrusion Detection Systems (IDS) Part I – (network intrusions; attack symptoms; IDS tasks; and IDS architecture). Prosinec 1992. Dostupné z: https://www.tcpdump.org/papers/bpf-usenix93.pdf.

[19] Messer, J. *Secrets of Network Cartography: A Comprehensive Guide to nmap*. 2005 [cit. 2022-03-20].

[20] Phillips, H. Linux Rootkits. Srpen 2020. Dostupné z: https://xcellerator.github.io/posts/linux_rootkits_01/.

[21] Salzman, P. J., Burian, M., Pomerantz, O., Mottram, B. a Huang, J. *The Linux Kernel Module Programming Guide*. 2022 [cit. 2022-05-01]. Dostupné z: https://sysprog21.github.io/lkmpg/.

[22] Schiller, C. a Binkley, J. R. *Botnets: The Killer Web App*. Syngress, 2007 [cit. 2022-03-12]. ISBN 978-1597491358.

[23] Shostack, A. *Threat Modeling: Designing for Security*. Wiley, 2014 [cit. 2022-03-12]. ISBN 978-1-118-80999-0.

[24] Tevault, D. *Mastering Linux Security and Hardening*. Packt, 2018 [cit. 2022-05-08]. ISBN 978-1-78862-030-7.

[25] Walsh, D. Your visual how-to guide for SELinux policy enforcement. 2013, [cit. 2022-05-04]. Dostupné z: https://opensource.com/business/13/11/selinux-policy-guide.

[26] Šplíchal, P. *Dokumentace k projektu Flexible Metadata Format* [online]. 2019 [cit. 2022-01-12]. Dostupné z: https://fmf.readthedocs.io/.

[27] Šplíchal, P. *Dokumentace k projektu Test Management Tool* [online]. 2019 [cit. 2022-01-12]. Dostupné z: https://tmt.readthedocs.io/en/stable/.

Příloha A

Obsah paměťového média

- `./bin/` – Adresář se spustitelnými soubory programu *doggo*.
- `./src/doggo/` – Adresář se zdrojovými kódy monitorovacího nástroje *doggo*.
- `./src/tests/` – Adresář s testy ve formátu FMF pro spuštění alarmů ve službě.
- `./src/latex/` – Adresář se zdrojovými soubory diplomové práce v LaTeXu.
- `./src/testing-farm.rules` – Konfigurační soubor pro Alertmanager nastavující upozorňovací pravidla na základě metrik.
- `./src/install-doggo.yaml` – Ansible playbook pro instalaci nástroje *doggo* na testovací stroje.
- `./xhavli47-DP.pdf` – Práce ve formátu *pdf*.