Univerzita Komenského v Bratislave

Fakulta matematiky, fyziky a informatiky

Detekcia zneužívania bezpečnostných zraniteľností v binárnych programoch

diplomová práca

2017

Bc. Jozef Brandys

Univerzita Komenského v Bratislave

Fakulta matematiky, fyziky a informatiky

Detekcia zneužívania bezpečnostných zraniteľností v binárnych programoch

diplomová práca

Študijný program:	Informatika
Študijný odbor:	2508 Informatika
Školiace pracovisko:	Katedra informatiky
Školiteľ:	Mgr. Peter Košinár

Bratislava, 2017

Bc. Jozef Brandys

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta: Bc. Jozef Brandys
Študijný program: informatika (Jednoodborové štúdium, magisterský II. st., denná forma)
Študijný odbor: informatika
Typ záverečnej práce: diplomová
Jazyk záverečnej práce: slovenský
Sekundárny jazyk: anglický

Názov: Detekcia zneužívania bezpečnostných zraniteľností v binárnych programoch
Detection of Exploiting Security Vulnerabilities in Binary Programs

Cieľ: Cieľom práce je analyzovať možnosti detekcie zneužívania zraniteľností v prípadoch, keď zdrojový kód cieľovej aplikácie nemusí byť k dispozícii. Práca poskytne základný prehľad existujúcich metód a ich vyhodnotenie v reálnych scenároch pri softvéri so známymi zraniteľnosťami. Cieľom práce bude zlepšiť možnosti takejto detekcie, či už zlepšením existujúcich metód alebo návrhom nových, ktoré budú následne implementované a prakticky otestované.

Vedúci: Mgr. Peter Košinár
Katedra: FMFI.KI - Katedra informatiky
Vedúci katedry: prof. RNDr. Martin Škoviera, PhD.

Dátum zadania: 23.05.2016

Dátum schválenia: 23.05.2016 prof. RNDr. Rastislav Kráľovič, PhD.
garant študijného programu

.....
študent

.....
vedúci práce

Poďakovanie: Rád by som na tomto mieste vyjadril vďaku svojmu diplomovému vedúcemu Petrovi Košinárovi, za odborné konzultácie, ako aj za cenné rady a pripomienky pri písaní tejto práce. Taktiež by som rád poďakoval mojej rodine a priateľke Barbare Bublákovej, za ich podporu a pomoc počas celého môjho štúdia.

Abstrakt

V našej práci sa venujeme detekcii exploitácie v binárnych programoch. Popisujeme rôzne druhy zraniteľností a spôsoby detekcie, ktoré sa používajú. Popisujeme či už riešenia založené na kontrole toku programu, hľadaní anomálií alebo analýze vplyvu vstupu na beh programu. Navrhli sme riešenie pomocou klasterizácie behov a hľadania prerekvizít a postrekvizít udalostí. Následne hľadáme anomálie, ktoré nepatria do žiadneho klastra alebo nespĺňajú našu kontrolu integrity. Toto riešenie sme prakticky implementovali a otestovali na reálnych zraniteľnostiach.

Kľúčové slová: binárny program, exploitácia, detekcia, klasterizácia, cfi, zraniteľnosť

Abstract

In this work we focus on detection of exploiting in binary programs. We describe different types of vulnerabilities and types of detections which are used. We describe detections based on control flow integrity, anomaly detection and taint analysis of the input. We have designed a solution which uses clusterization of the traces and search for prerequisites and post-requisites between events. In detection phase we check if trace belongs to any cluster and passes out integrity check. We have practically implemented this solution and tested it on real vulnerabilities.

Keywords: binary program, exploitation, detection, clustering, cfi, vulnerability

Obsah

Úvod

Útoky na programy sú bežnou súčasťou každodenného života. Už niekoľko desaťročí sa zvádza súboj medzi útočníkmi a vyvojármi bezpečnostných riešení. Veľká časť výskumu sa zaoberá vývinom bezpečnostných riešení, ktoré sa aplikujú na aplikácie, ku ktorým je dostupný zdrojový kód. Naša práca sa zameriava na prípady keď nie je dostupný zdrojový kód, čo je bežná prax pri proprietárnych aplikáciach, alebo keď už nie je dostupný pri starších aplikáciach. Používaním takýchto aplikácií nechceme oslabiť celkovú bezpečnosť aplikácií, ale chceme ich uchrániť pre útočníkmi. V našej práci sa budeme venovať takýmto riešeniam, pričom prvé dve kapitoly sú všeobecný úvod, a následne tretia kapitola je popis nami navrhovaného riešenia.

V prvej kapitole sa venujeme útokom na programy. Popisujeme základné kategórie útokov a spôsob vykonávania záškodníckej činnosti. Útočníci menili svoje technológie a metódy postupom času tak, aby sa dokázali vyhnúť detekcii. Formy detekcie sa stále vyvíjajú a sú stále sofistikovanejšie, a tak aj metódy ktoré si popíšeme, sú rôznej komplexnosti.

Metódy detekcie exploitácie sú témou druhej kapitoly. V našej práci sme sa rozhodli priniesť prehľad metód detekcie exploitácie v binárnych programoch. Metódy, ktoré si spomíname v tejto kapitole sú v niektorých prípadoch priamym rozšírením metód používaných kompilátormi na zvýšenie bezpečnosti. Tieto metódy bývajú deterministické, a komplikujú exploitáciu určitých druhov exploitov. Ďalšie metódy sú všeobecnejšie a detegujú jednak stav, či je program už exploitovaný a útočník prebral nad ním kontrolu, alebo vytvárajú model normálneho správania sa programu jeho pozorovaním, a ná-

sledne hľadajú anomálie, ktoré sa vymykajú normálnemu správaniu. Úlohou tejto kapitoly je priblížiť čitateľovi túto problematiku a ukázať mu riešenia, ktoré sa ukázali ako účinné.

Témou tretej kapitoly je návrh a implementácia nášho riešenia. Popíšeme si bezpečnostný model, ktorý hovorí o tom aké druhy chýb detegujeme. Následne si popíšeme naše riešenie, jeho jednotlivé časti a spôsob implementácie nášho prototypu. Implementáciu, ktorú sme realizovali, sme otestovali a vyhodnotili na niekoľkých zraniteľnostiach. Výsledky nášho testovania sú taktiež súčasťou tejto kapitoly. Nakoniec si v závere kapitoly zhrnieme vlastnosti nášho riešenia a porovnáme ho s predchádzajúcimi návrhmi.

Cieľom našej diplomovej práce bolo priniesť prehľad metód detekcie exploitácie v binárnych programoch a navrhnúť vlastné riešenie, ktoré sa implementuje a prakticky otestuje.

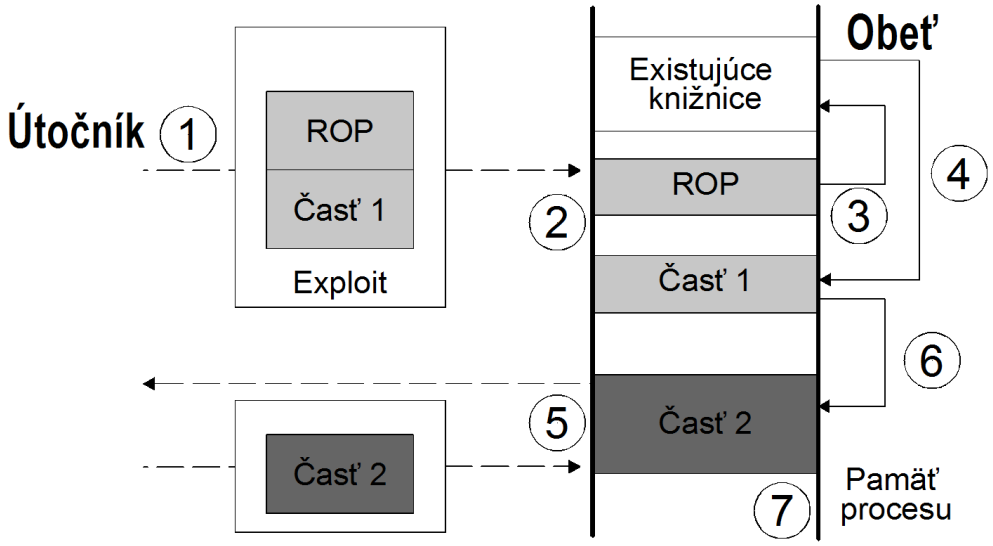
Kapitola 1

Zneužívanie chýb v programoch

Počítačové systémy sú pod stálou hrozbou útoku, keď útočník napríklad pomocou špeciálne pripravených vstupných dát (napríklad internetový paket) dokáže prebrať kontrolu nad programom a ďalej na cielenom počítači vykonávať svoj vlastný kód. Informačná bezpečnosť sa už dlho snaží zabraňovať úspešným útokom, a tak ochrániť užívateľov. V tejto kapitole si najprv popíšeme typický postup exploitácie a následne niektoré konkrétne techniky fungovania exploitov (programov zneužívajúcich softvérovú zraniteľnosť).

V najbližších odsekoch si popíšeme typický postup exploitácie podľa obr. 1.1. Na začiatku v fáze č.**1** útočník pripraví špeciálny vstup obsahujúci exploit. Tento exploit sa skladá z dvoch častí. Prvá časť je kód, ktorý postupne prevezme riadenie programu potom ako využije zraniteľnosť. Druhá je krátky kód pripravujúci proces exploitácie na ďalšie fázy. V **2.** fáze je exploit nahraný do programu (ako škodlivý paket/webová stránka/vstup/obrázok) a aktivuje chybu, ktorá bola útočníkovi vopred známa. V **3.** fáze exploit preberie riadenie programu pomocou niektorej z techník, ktoré sú neskôr popísané v tejto kapitole a po jej skončení sa v kroku číslo **4.** presmeruje riadenie do krátkeho kódu, ktorý býva typicky veľmi malý. Obmedzenie na veľkosť je niekedy iba praktická záležitosť, inokedy je kvôli ľahšiemu vytvoreniu exploitu. Kratší kód sa dá jednoduchšie skryť a znižuje sa riziko prepísania časti pamäte, v ktorej sa nachádza, počas predchádzajúcich fáz - využitia chyby alebo prebe-

rania riadenia. Typicky je napísaný multiplatformovou formou a v asembleri s použitím malého počtu základných inštrukcií, tak aby bol pamäťovo pozične nezávislý. Úlohou tohto kódu je v **5.** fáze stiahnuť a nasledovne vo fáze **6.** previesť kontrolu na ďalší už finálny kód. Tento kód sa zvyčajne nachádza iba v pamäti a pomocou reflektívneho načítania knižnice sa pripojí, ako knižnica bez potreby vytvárať súbory v súborovom systéme. Po tomto má už útočník pod kontrolou beh programu a môže vykonávať ďalšie úlohy, ktoré potrebuje na svoju záškodnícku činnosť[25].

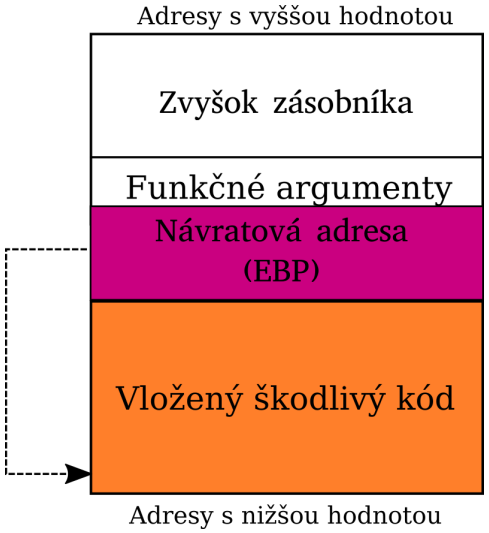


Obr. 1.1: Typický postup exploitácie

1.1 Vloženie vlastného kódu

Vloženie vlastného kódu je jednou z prvých metód používaných exploitami. Útočník pri tejto technike vloží kód priamo do pamäte vykonávaného programu a následne presmeruje beh vykonávania programu tak, aby sa vykonal aj tento, ním vložený kód. Takúto príležitosť môže vytvoriť mnoho situácií,

napríklad chyby typu pretečenia zásobníka niekedy umožňujú, tak ako je to zobrazené na obrázku (1.2), vložiť kód na zásobník a prepísať aj návratovú adresu funkcie. Následne pri návrate z funkcie sa presmeruje vykonávanie do vloženého kódu, ktorý vykonáva už ďalšie útočníkom mienené operácie. Vloženie vlastného kódu nie je výsada nízkoúrovňového jazyka. Príkladom sú chyby *SQL injection*, ktoré boli v roku 2012 najviac rozšírenými zraniteľnosťami vo webových aplikáciách[1]. Aj mnohé ďalšie jazyky (napr. aj Python, PHP, javascript, Html, ...) pri nesprávnom používaní môžu dovoľovať útočníkovi vložiť kód (najmä pri nesprávnom používaní citlivých funkcií a nedostatočnom kontrolovaní ich argumentov).



Obr. 1.2: Vkladanie kódu

V zásade platí pri ochrane pred vložením vlastného kódu pravidlo dostatočného overovania a precíznej kontroly vstupu od užívateľa. Takéto kontroly sa ľahšie odporúčajú, ako vykonávajú a preto sa používa niekoľko techník, ktoré zabraňujú ich zneužitiu. Pri *SQL injection* to môže byť výlučné používanie predpripravených SQL príkazov, dostatočné obmedzenie nebezpeč-

[1]Spot the Web Vulnerability: `http://www.slideshare.net/stamparm/spot-the-web-vulnerability`

ných znakov[5]. Môžu sa používať aj zoznamy povolených znakov/príkazov (prípadne zakázaných), aby sa predišlo zneužitiu citlivých funkcií. V binárnych programoch sa používa, napríklad metóda zabránenia vloženia kódu na zásobník alebo metóda W^X, keď všetky stránky pamäte môžu byť buď zapisovateľné, alebo exekutovateľné, ale nikdy nie oba naraz [10].

1.2 Zneužitie existujúceho kódu

Vývojom bezpečnosti sa vytvorilo viacero metód na zabránenie vkladania vlastného kódu a jeho následného spustenia. Útočníci boli vynaliezavý a podarilo sa im vyvinúť niekoľko techník, ktoré využívajú kód nachádzajúci sa v pamäti na to, aby ho využili na svoje záškodnícke účely. V najbližších častiach si popíšeme niekoľko kategórií útokov, ktoré zneužívajú kód prítomný v pamäti na to, aby prebrali kontrolu nad programom.

1.2.1 Skočenie do štandardnej knižnice

Skočenie do štandardnej knižnice je jednou z odpovedí, na zavedené opatrenia proti vkladaniu vlastného kódu na zásobník, alebo všeobecne do pamäte. V prvom prípade mohol útočník zmeniť obsah zásobníka tak aby sa najprv pomocou knižničnej funkcie *strcpy* (na ktorú presmeroval chod programu) skopíroval kód zo zásobníka do dátovej oblasti a následne sa odtiaľ vykonal. V druhom prípade, ak žiadna stránka pamäte, v ktorej sa potenciónálne mohol nachádzať škodlivý kód, nebola exekutovateľná, mohol útočník priamo zavolať štandardnú funkciu *system* s vlastnými parametrami. Napríklad *system("/bin/sh")* [10].

1.2.2 Využívanie inštrukcie return

Využívanie inštrukcie return sa nazýva Return-oriented programming (skrátene ROP). Je to technika, pri ktorej útočník efektívne využíva kód aplikácie tak, aby získal kontrolu nad programom. Zvyčajne pri tejto technike dokáže

útočník modifikovať obsah zásobníka bežiaceho programu (prípadne zmeniť hodnotu registra *%esp*). Pozmenený zásobník sa modifikuje tak, aby program postupne vykonával krátke sekvencie inštrukcií, ktoré sa končia pomocou inštrukcie return. Každá časť končiaca inštrukciou *return* sa nazýva *gadget* a vykonáva zvyčajne iba krátku sekvenciu inštrukcií. Zreťazením *gadgetov* dokáže útočník vykonať zložitejšie operácie. Dokonca sa ukázalo, že vo väčšine prípadov je táto technika turingovsky kompletná [9].

V najjednoduchších prípadoch má útočník prístup k binárnemu súboru a dokáže si vopred predpripraviť reťaz *gadgetov*, ktorá sa vykoná. Pri použití znáhodnenia programu, napr. pomocou ASLR (kap. 2.1.1), potrebuje útočník navyše poznať adresy jednotlivých modulov v pamäti v programe a prepočítavať reťaz pri každom použití. Mnohokrát nie sú takéto predpoklady naplnené, ale rôzne kombinácie schopností útočníka môžu viesť k technikám, ktoré obchádzajú aj pomerne sofistikované ochrany. My si teraz niektoré popíšeme.

JIT-ROP

Predpripravená reťaz *gadgetov* nie je vždy funkčná možnosť. Aplikácie môžu využívať znáhodňovanie pamäti nielen na úrovní modulov, ale na úrovní jednotlivých funkcií, či dokonca základných blokov alebo inštrukcií. *JIT-ROP* [24] je vhodný práve v takýchto prípadoch. Pri tejto technike je predpoklad úspešného útoku niekoľko informácií o programe - odhalenie aspoň jedného funkčného ukazovateľa v pamäti, možnosť presmerovať tok programu do reťaze *gadgetov* a možnosť čítať ktorékoľvek miesto v pamäti na absolútnej adrese. Môže sa to zdať ako veľmi silný predpoklad, ale napríklad moderné prehliadače sú dobrý príkladom softvéru, u ktorých sa podobné situácie vyskytujú.

JIT-ROP sa skladá z niekoľkých častí, ktoré postupne v niekoľkých fázach vykonávajú útok. Jednotlivé ukazovatele nám zvyčajne dovolia prečítať bezpečne (bez prístupu mimo alokovanú pamäť) 4 kb pamäte. Tieto jednotlivé úseky pamäte môžu obsahovať ďalšie indície o ďalších alokovaných blokoch

(napríklad vetviace inštrukcie odhaľujú ďalšie bloky). Zbieraním takýchto blokov sa dá získať dostatočný počet informácii o programe na úspešný útok. Po získavaní dostatočného množstva informácií, hľadajú nástroje ako komunikovať s operačným systémom. Na to sú vhodné knižničné funkcie, ktoré môžu byť odhalené rôznymi cestami - napr. čítaním PEB tabuľky, nájdenie adresy na zásobníku, ... Teraz máme už dostatok informácií a treba vytvoriť exploit. Exploit sa kompiluje postupne, aby vykonal svoju úlohu, pričom sa využívajú informácie v odhalených častiach pamäte. Nájdene kúsky pamäte sa disasemblujú, identifikujú sa *gadgety* a následne sa pospájajú do reťaze. Podľa tejto reťaze sa presmerováva tok programu a útočník môže začať vykonávať záškodnícku činnosť.

Blind Rop

Dlhotrvajúce serverové aplikácie sú vystavené aj útokom metódy *Blind ROP* [8], ktorá slúži na automatické vytváranie exploitu, aj ak útočník nemá prístup k binárnej podobe programu a dokonca ani k zdrojovému súboru, ale pozná nejakú chybu pretečenia zásobníka, ktorá sa dá spoľahlivo opakovane zneužívať. Predpokladom pre úspešný útok je používanie systémového volania *fork* na vytváranie nových procesov, ktoré jednotlivo spracúvajú požiadavky užívateľa. Problém pri používaní *fork-u* spočíva v tom, že 2 procesy majú rovnaké pamäťové rozloženie, používajú rovnaké kanáriky na zásobníku. Dlhotrvajúce serverové aplikácie ponúkajú množstvo pokusov na to, aby sa postupne útočník zistil informácie potrebné na úspešnú exploitáciu a keďže sa nemenia, tak ich môže použiť na vytvorenie exploitu.

Metóda *Blind ROP* popisovaná v článku [8] sa skladá z niekoľkých fáz. Útočník postupne obchádza ochrany (ASLR, kanáriky na zásobníku) a hľadá zaujímavé miesta v pamäti obsahujúce správnu postupnosť inštrukcií na to, aby z nich vytvoril reťaz, ktorú použije na exploitáciu. Hlavným cieľom spomínanej metódy bolo vytvoriť reťaz, volajúcu systémové volanie *write*, aby si útočník vedel posielať cez už otvorené pripojenie naspäť dáta z pamäte. Následne s takýmito dátami vie vykonať už klasické druhy útokov - napr. spo-

mínaný *JIT-ROP*. Reťaz vytvára postupne, najprv nájde inštrukcie, ktoré priamo vykonajú alebo "emulujú" `pop rdi`, `pop rsi`, ..., `syscall`. Na to aby našli inštrukciu `pop X` používajú mechanizmus, ktorý vie detegovať, či bola zavolaná inštrukcia *pop*, alebo nie. Následne sa pokúsia nájsť relokačnú tabuľku, ktorú využijú na nájdenie funkcie *write* a *strcmp* (ktorú využívajú na emuláciu `pop rdx`). Nakoniec už iba pospájajú všetky dôležité časti do jednej reťaze, aby vykonali úspešný útok.

Útok *Blind ROP* môže byť vhodný vo viacerých situáciách - útok na proprietárne aplikácie, ku ktorým máme vzdialený prístup, útok na open-source aplikácie používané v proprietárnych aplikáciách, útok na open-source aplikácie ku ktorým máme zdrojový kód, ale nemáme konkrétnu verziu binárneho súboru (napr. nachádzajúcich sa v distribúciach linuxu, ktoré si zvyknú užívatelia sami kompilovať). Pri praktických testoch potrebovali na úspešný útok približne 2000 požiadaviek na server na to, aby úspešne vykonali volanie *write*.

1.3 Útoky bez získania kontroly

Útoky, ktoré sme si doteraz spomínali, všetky získavali priamu kontrolu nad programom. Menili výpočet programu, pričom priamo ovplyvňovali hodnotu inštrukčného ukazovateľa. Chyby pretečenia zásobníka, chyby formátovania reťazcov a ďalšie sa dajú zneužiť aj bez toho, aby útočník získal kontrolu nad programom. V článku [11] tvrdia, že útoky, ktoré nepreberajú kontrolu nad programom sú rovnako nebezpečné, ako tie ktoré ju preberajú. Navyše tieto chyby sú reálnou hrozbou a aj my si neskôr v tejto kapitole popíšeme reálne prípady.

Ako môžu prebiehať takéto typické útoky? Typické útoky bez kontroly behu sa zameriavajú hlavne na tieto kategórie údajov (podľa [11]):

- Konfiguračné dáta
- Vstup užívateľa
- Dáta o užívateľovi

- Dáta používajúce sa na rozhodovanie o toku programu

Pri exploitovaní chyby sa môžu tieto druhy údajov modifikovať tak, aby útočník splnil svoj záškodnícky úmysel. Rôznymi chybami prepíšeme tieto dáta na to, aby sme ovplyvnili beh programu a získali nad ním kontrolu. Predtým, ako si popíšeme reálne útoky, si ešte popíšeme prečo sú jednotlivé kategórie zaujímavé.

Konfiguračné dáta obsahujú mnoho nastavení programu. Tieto údaje môžu obsahovať informácie o pravidlách, ako sa má kontrolovať prístup k systémovým častiam. Napríklad webové servery si uchovávajú prístupové cesty k súborom servera a k ním prislúchajúce pravidlá prístupu. Podobné súbory si ukladajú aj napríklad SSH a FTP servery. Tieto údaje sa málokedy menia počas behu servera a zostávajú statické až do reštartu. Ich poškodením môže útočník získať prístup na inak neprístupne miesta. Napríklad v niektorých prípadoch zmenením adresy k podporným modulom, môže útočník spustiť ľubovoľný program.

Vstup užívateľa sa taktiež môže potencionálne používať na útok. Ak dokážeme zmeniť vstup užívateľa potom, ako sa skontrolovala jeho správnosť, tak v správnych situáciach môžeme vykonať útok.

Dáta o užívateľovi, ktoré sa najprv načítajú do pamäte a následne sú útočníkom zmenené, môžu taktiež spôsobiť škody v programe. Napríklad zmenou identity, ktorá bude následne použitá na bezpečnostnú kontrolu môže byť použité na obídenie kontroly prístupu. Takto môže útočník zmeniť citlivé údaje, prípadne získať priamy privilegovaný prístup k svojmu cieľu.

Správnosť dát používajúcich sa na rozhodovanie o toku programu je mnohokrát kritická. Často sa takéto kritické binárne premenné nachádzajú v pamäti a pamätajú si priebežné hodnoty výsledkov. Napríklad binárna hodnota o tom či je užívateľ úspešne autentifikovaný, alebo nie môže byť disjunkciou čiastkových výsledkov. Jej prepísaním môže byť výsledok pozitívny napriek tomu, že všetky čiastkové výsledky sú negatívne.

Teraz si ukážeme niekoľko ukážok (prevažne prebratých z článku [11]) zneužite chýb na útok bez kontroly behu programu.

WU-FTP

Je ftp server, ktorý obsahoval chybu vo formátovacom reťazci. Táto chyba sa dala zneužiť iba po prihlásení užívateľa, ale podarilo sa ju zneužiť tak aby útočník získal administrátorské práva. Tento server zvyčajne bežal s efektívnym užívateľským ID podľa prihláseného užívateľa, ale reálne ID bolo ID užívateľa *root*. Pri niektorých operáciach sa nastavil dočasne privilegovaný užívateľ a následne sa po vykonaní akcie nastavil správne späť na práve prihláseného užívateľa. Môžeme si pozrieť ukážku kódu:

```
FILE * getdatasock(...) {
    ...
    seteuid(0);
    setsockopt(\ldots);
    ...
    seteuid(pw->pw_uid);
    ...
}
```

Útok spočíval v prepísaní dát o užívateľovi. V štruktúre *pw* sa zmení hodnota *pw_uid* na 0. Takto zostanú užívateľovi práva root aj naďalej a môže, napríklad zmenit súbor */etc/passwd*, tak aby mal perzistentne administrátorský účet.

SSH

V tomto prípade obsahoval server SSH chybu pretečenia celočíselnej premennej. Táto chyba sa prejavila pri dlhých paketoch a dovoľovala zmeniť hodnotu veľkosti v 16 bitovej premennej na hodnotu 0, čo malo za následok možnosť prepísať akékoľvek miesto v pamäti. Útok, ktorý sa podaril skupine výskumníkov, dovoľuje obísť autentifikáciu v programe, a tak sa prihlásiť aj bez potreby autentifikácie (či už pomocou hesla, alebo súkromného kľúča). SSH, ktoré túto chybu obsahovalo používalo premennú *authenticated*, ktorá bola

inicializovaná na 0. Táto hodnota sa používala na zapamätanie si úspešnej autentifikácie. Pri autentifikácií sa vybrala metóda (podľa dohody v komunikácií) a následne ak bola úspešná, tak nastavila hodnotu *authenticated* na 1. Ak bola neúspešná, tak hodnotu *authenticated* nemenila. Po autentifikácií sa kontrolovala hodnota v premennej *authenticated*, na to či bola autentifikácia úspešná alebo nie. Útok, ktorý realizovali, prepísal hodnotu *authenticated* na 1, čím SSH server pokladal útočníka za úspešne autentifikovaného a dovolil mu prístup k serveru. Ani tento útok teda nezískal priamu kontrolu nad tokom programu, ale útočníkovi sa podarí získať prístup k serveru, kde môže ďalej vykonávať svoju záškodnícku činnosť.

Kapitola 2

Prehľad metód detekcie

Postupne ako sa vyvíjali možnosti exploitácie, vyvíjali sa aj riešenia, ktoré sa im snažili zabrániť. Tieto riešenia boli zvyčajne odpoveďou na konkrétne druhy útokov, napríklad neexekutovateľný zásobník, bol odpoveď na vkladania kódu na zásobník pri chybách typy pretečenie zásobníka. Niektoré vyvinuté riešenia sa snažili sťažiť vytváranie exploitov pridávaním náhodnosti (kap. 2.1.1), iné sa snažili obmedziť schopnosti programu (kap. 2.1.2). Mnoho riešení vyžaduje na svoje využitie zdrojový kód, ale mnoho si vystačí iba s binárnou podobou, čo ale zväčša vedie k menej efektívnym riešeniam. Metódy detekcie v našom prípade rozdelíme na 2 skupiny. Prvá skupina sú metódy, ktoré sa snažia priamo zabrániť exploitáciu, zmeneným správaním programu, čím sa deteguje exploit, ktorý sa používa na nezmenenú verziu programu. Druhá skupina vytvorí model ideálneho správania sa programu, ktorý sa následne používa na rozdelenie sledovaných behov programu do dvoch skupín - normálne a anomálne behy.

2.1 Detekcia zabránením exploitácie

V tejto časti našej práce spomenieme niekoľko techník, ktoré sa snažia zabrániť úspešnej exploitácií programu.

2.1.1 ASLR

ASLR (z anglického Address space layout randomization), je často používaná technika, ktorá zabraňuje veľkému radu exploitov. Cieľom je vložiť náhodnosť do pamäťového rozloženia, keď sa pri spúšťaní nezávislé časti programu, uložia na od seba nezávislé náhodné miesta. Náhodné umiestnenie pamäte sa nanovo vykonáva pri každom spustení programu a teda exploity, ktoré rátajú s fixným rozložením pamäte, majú omnoho nižšiu šancu, že exploitácia prebehne úspešne, čím môžu spôsobiť pád programu, ktorý môže byť potencionálne detegovaný, následne analyzovaný a chyba, ktorá tento pád zapríčinila bude opravená [4].

2.1.2 CFI

(Control flow integrity) je technika, pri ktorej sa kontroluje integrita behu programov pomocou overenia cieľov vetviacich inštrukcií. Väčšina implementácií CFI vyžaduje prekompilovať zdrojový kód, keď sa pri kompilácií statickou analýzou určia, pre každé vetviace miesto s dynamickým cieľom miesta, kde môže vetvenie programu pokračovať. Príkladom je CFG od Microsoft [2]. Veľké množstvo útokov, ktoré využívali kód aplikácie (kap. 1.2.2), motivovali výskumníkov vyvinúť riešenia, ktoré si vystačia s binárnym kódom. Tieto vytvárajú graf programu, pomocou ktorého sa kontroluje integrita programu. Taktiež sa používajú osobitné zásobníky, ktoré sa používajú na kontrolu inštrukcií return, aby skákali iba na miesta, odkiaľ bola funkcia zavolaná. Tieto riešenia zabezpečujú pomerne vysokú ochranu, ale príliš spomaľujú vykonávanie programu. Ďalšie riešenia sa snažili zjemniť pravidlá, podľa ktorých sa kontrolovala integrita programu. Napríklad pri kontrole skoku inštrukcie return, môže byť cieľom akékoľvek potencionálne miesto volania programu (nemuselo byť striktne to, ktoré naozaj funkciu volalo). Ukázalo sa, že uvoľnenie pravidiel viedlo k jednoduchému obídeniu ochrany.

Chyby v programe, ktoré umožňujú prečítanie časti pamäti sú takisto veľký problém, pretože umožňujú útočníkovi pripraviť sa a obísť ochrany,

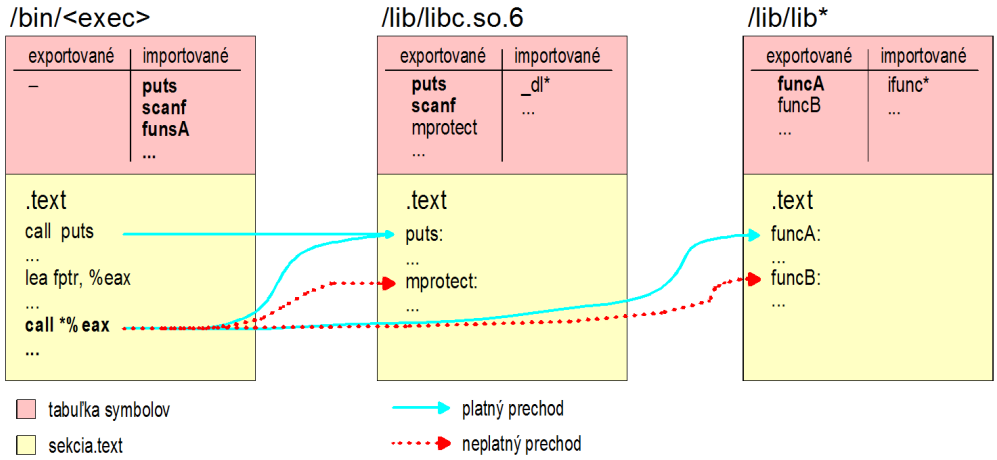
ktoré kontrolujú integritu programu, ako sme si už spomínali pri JIT-ROP (kap. 1.2.2). Súčasťou CFI sa stáva aj skrytie informáciu o kontrolovanom toku, či detekcia pokusov hľadania úspešného exploitu, aby sme znovu znáhodňovali beh programu. Vtedy útočník nemá viacej pokusov na to aby postupne získal dostatok informácií o programe [17].

Lockdown

Zaujímavým ukážkovým príkladom, ako vynucovať integritu behu programu pomocou overovania vetviacich inštrukcií, je práve nástroj Lockdown[20].V najbližších odsekoch popíšeme ako zabezpečuje kontrolu integrity a ako vytvára pravidlá na základe ktorých vynucuje správnosť behu programu. Najprv si popíšeme ako bez prístupu k zdrojovému kódu vytvárajú pravidlá pre správny beh programu. Lockdown využíva vierohodný čítač knižníc, ktor7 ho informuje o pridaní/odobraní jednotlivých knižníc a dynamicky prispôsobuje pravidlá práve načítanému kódu. Pri každom načítaní sa analyzuje načítaná knižnica a prepočítajú sa pravidlá, pri odobraní knižníc z behu programu sa následne tieto pravidlá odstránia. Z knižníc sa získavajú informácie o začiatkoch a koncoch funkcií za tabuľky **symtab**, ktorá obsahuje informácie o jednotlivých funkciách v rámci knižnice. Navyše sa aj prezerajú aj záznamy v tabuľke **dynsym**, ktorá obsahuje záznamy o všetkých exportovaných funkciách. Prvá zo spomínaných tabuliek sa nemusí vždy nachádzať v knižnici, vtedy sa pravidlá závisle na tejto informácií nevynucujú.

Informácie o importovaných a exportovaných funkciách spolu so záznamami v tabuľke *symtab* sa využívajú nasledovne. Popíšeme si obrázok 2.1, na ktorom vidíme, ktoré skoky/volania sú dovolené alebo zakázané. Ak je vetviaca inštrukcia volanie funkcie, tak je povolené volanie iba do funkcií v rámci tej istej knižnice, volanie importovaných funkcií z danej knižnice. Ak nemá informácie z tabuľky *symtab*, tak sa povolia aj skoky mimo začiatkov funkcií v rámci tej istej knižnice. Skoky pomocou inštrukcie *jmp* môžu viesť iba v rámci jednej funkcie (ak nie sú informácie o začiatkoch a koncoch funkcie, tak v rámci jednej knižnice) a ešte na začiatky funkcií (kvôli chvostovej

rekurzii). Navyše, všetky skoky musia byť na začiatky inštrukcií, ktoré sú dosiahnuteľné zo začiatkov funkcií.



Obr. 2.1: Ukážka pravidiel v Lockdown

Okrem volania funkcií a skokov pomocou inštrukcie *jmp* Lockdown vynucuje aj integritu pri návrate z funkcií. Na to, aby udržal integritu volania a návratu z funkcií, implementuje skrytý zásobník, ktorý nie je dostupný z aplikácie. Inštrumentovaním funkcií volania a návratu z funkcie si vkladá a vyberá hodnoty z tohto zásobníka, pričom kontroluje integritu s aplikačným zásobníkom. Pri správnom behu programu sa očakáva, že posledne pridaná hodnota na skrytý zásobník bude prvá vybraná hodnota. Toto nie je univerzálny prípad, napríklad keď sa využívajú C++ výnimky. V implementácií Lockdown sa rozhodli sledovať tieto výnimky a po každej zosynchronizujú skrytý zásobník s aplikačným zásobníkom, pričom môžu adresy iba vyberať.

V nasledujúcich odsekoch si popíšeme druhú časť Lockdown [20] a to je spôsob zabezpečenia integrity a vynucovania pravidiel. Lockdown je implementovaný pomocou knižnice *libdetox*[21], ktorá umožňuje binárny preklad programu pre architektúru x86. *Libdetox* rozdelí program do dvoch častí - aplikačnej a dôveryhodnej časti. Tieto dve časti sú striktne oddelené, pričom z aplikačnej vrstvy neexistuje cesta do dôveryhodnej. *Libdetox* umožňuje in-

štruovať niektoré časti programu a využiť to na bezpečnostné kontroly. Využívali to napríklad pri spomínanej implementácií skrytého zásobníka. Všetky statické skoky a volania funkcií sú skontrolované už pri preklade binárneho kódu. Dynamické skoky sa prirodzene nedajú skontrolovať už pri preklade, keďže cieľ sa nachádza v pamäti, ktorá ešte nie je známa a navyše sa môže hocikedy zmeniť. Binárny prekladač teda pri každom volaní takýchto inštrukcií zavolá výnimku, ktorá spustí kontrolu skoku, ktorý je skontrolovaný (čím sa spomaľuje beh programu). Aby zbytočne nevyhľadávali stále tie iste informácie dookola, používajú cache pamäti na urýchlenie kontroly.

Výkon *Lockdownu* sa zdá byť celkom zaujímavý, keď bez potreby zdrojového kódu a vynucovania integrity toku programu pre jednotlivé knižnice, dosahuje spomalenie iba na úrovní 20%. Pričom značnú časť z tohto tvorí samotné binárne prepisovanie programu.

Opaque CFI

O-CFI [17]} v preklade priehľadné CFI. Bezpečnostný model tohto riešenia, predpokladá útočníka, ktorý má neobmedzený prístup k čítaniu pamäte programu a aj napriek tejto veľkej aj keď nie nereálnej schopnosti útočníka dokáže program s vysokou pravdepodobnosťou uchrániť pred útočníkom a zabrániť mu vykonať úspešný útok. Hlavným cieľom tvorcov O-CFI bolo vytvoriť riešenie, ktoré by bolo odpoveďou na útoky, ktoré dokázali obísť znáhodňovanie programu po malých úsekoch, podobne ako sme popísali pri JIT-ROP útokoch v kapitole 1.2.2. Rovnako ako pri *Lockdown* si postupne popíšeme časti *O-CFI*.

Na to, aby sme pochopili význam jednotlivých častí popíšeme si celkovú architektúru riešenia. Hlavnou otázkou je dosiahnuť riešenie, ktoré je účinné aj pri možnosti útočníka prečítať všetky dostupné časti pamäte. *O-CFI* využíva kontrolu intervalov, keď pre každý cieľ pre konkrétny skok skontroluje, či sa nachádza v konkrétnom intervale, ktorý je pre neho určený. Kontrola intervalov je veľmi rýchla a aj sa jednoduchšie schováva oproti zložitejším pravidlám na overovanie korektnosti. Samotná kontrola, samozrejme, nestačí, a

preto použili techniku, pri ktorej sa znáhodni rozloženie programu na úrovni základných blokov. Pre každý skok majú zoznam cieľov, kam sa môže skákať a interval musí zahŕňať všetky z nich. Zoznam možných cieľov získajú konzervatívnym reverzovaním binárneho súboru, pričom sa riadia pravidlom, že je lepšie mať viacej cieľov, ako keby mali niektoré chýbať. Prečo to teda funguje a je bezpečné? Najprv sa vypočítajú všetky možné ciele pre každý skok, zároveň sa celý program znáhodní a po základných blokoch je náhodne uložený do pamäte. Celý program je teda pri každom behu úplne náhodný a intervaly, ktoré sú skryté pred útočníkom, narozdiel od zvyšku programu, sú vždy iné. Ešte s ďalšími malými úpravami útočník nie je schopný vyprodukovať sériu adries, kam by mohol pri ROP útoku skákať.

Najprv si ujasníme, čo mysleli autori tým, že útočník vie prečítať celú pamäť. Pod čítaním celej pamäte sa predpokladá že útočník vie rekurzívne prečítať ktorékoľvek miesto v pamäti, referencované z iného miesta, takže celý pôvodný program. Tvorcovia ale schovávajú citlivé informácie - intervaly, na náhodnom mieste v pamäti tak, aby sa nedali prečítať. Aby útočník nemohol prečítať hodnoty intervalov, tak ukazovatele na tabuľku s intervalmi sa nachádza iba v registri *gs*. Keďže inštrukcie, ktoré sa nachádzajú v programe nereferencujú tento register, útočník je odkázaný na hádanie adresy, čo je pravdepodobnostne zanedbateľné. Navyše, aby zvýšili ochranu, pri každom čítaní pamäte z nedostupnej pamäte, vydá program signál, aby sa nanovo znáhodnil (zabráni sa útoku popísanom pri *JIT-ROP* v kapitole 1.2.2).

Priamočiare vytváranie portálov vedie k veľmi voľným intervalom, čím by sa bezpečnosť veľmi nezvýšila. Nový prístup, s ktorým prišli, pridal pri znáhodňovaní programu aj tzv. portály. Portály sú miesta, ktoré sa vkladajú navyše do programu a obsahujú iba priamy skok na iné miesto (takže už nepotrebujú kontrolu správnosti). Pri náhodnom usporiadaní programu sa snažia, čo najviac zúžiť veľkosť intervalov, aby čo najviac obmedzili útočníka, a tak vytvorili bezpečnejšie kontroly. Toto dosiahli tým, že pre každý nepriamy skok vytvoria osobitný klaster. Pri vkladaní základného bloku majú tri možnosti - buď rozšíria jednotlivé klastre, ktoré smerujú do tohto bloku,

aby obsahovali aj tento blok alebo vytvoria viacero kópií toho istého bloku vo viacerých klastroch, alebo treťou možnosťou je, že sa vytvorí nový portál v danom klastri. Každá z možností má iný vplyv na výsledné správanie programu. Rozšírenie intervalov znižuje bezpečnosť. Kópia blokov zväčšuje množstvo potrebnej pamäte. Použitie portálu zase znižuje výkon programu - keďže pridáva jeden medziskok navyše. Na to, aby zvolili kompromis, tak pre každý klaster dovoľujú fixný počet portálov. Vo väčšine prípadov používajú 12 portálov na jeden klaster, čím nespôsobia príliš veľký dopad na rýchlosť programu a zároveň značne zmenšia priemerné veľkosti intervalov.

Navyše, ku kontrole intervalov, vynucujú aj skok na miesta zarovnané na okraj krátkych blokov dĺžky mocniny čísla 2 (zvyčajne na 16 bajtov). Znižujú tým pravdepodobnosť, že útočník nájde miesto vo vnútri intervalu, ktoré nie je začiatok žiadneho bloku. Toto miesto nachádzajúce sa potencionálne mimo začiatku nejakej inštrukcie, by po vykonaní takýchto posunutých inštrukcií nevynucovalo správnosť skokov. Pri implementácií dbajú na to, aby nimi vytvorené portály neobsahovali valídne inštrukcie na adresách začiatkov blokov. Taktiež počas binárneho prekladu (ktorý si následne popíšeme) prepisujú binárny kód tak, aby zarovnal inštrukcie na okrajoch týchto blokov. Takto je nútený útočník nájsť správne miesto vo vnútri intervalu, ktoré by vyhovovalo jeho požiadavkám pri tvorbe exploitu, čo je teraz omnoho ťažšie.

Identifikácia cieľov je pomerne zložitá vec, od ktorej značne závisí či bude *O-CFI* funkčné, alebo nie. Ak by neidentifikovali všetky ciele, vytvorili by falošné hlásenia, čo by mohlo byť vo veľa prípadoch neprijateľné. Na analýzu kódu napísali *IDA Python* program. Pri hľadaní potencionálnych cieľov postupujú tak, aby identifikovali všetky ciele aj za cenu toho že identifikujú niečo navyše.

Časťou *O-CFI*, ktorú si teraz popíšeme, je binárne prepisovanie programu, ktoré umožňuje znáhodnenie behu programu ako aj pridanie kontrol. Na začiatku pri prepisovaní programu vytvoria dve sekcie *.told* a *.tnew*. Prvá menovaná obsahuje pôvodný kód, zatiaľ čo do druhej sa zapíše už prepísaný. Sekcia obsahujúce pôvodný kód je nastavená ako nespúšťateľná, aby sa za-

bránilo jej zneužitiu. Všetky možné ciele skoku v pôvodnej binárnej podobe sú prepísané na špeciálnu značku *0xf4* a za nimi sa napíše adresa prepísaného kódu v *.tnew*. Táto značka s adresou sa využívajú na efektívne presmerovanie skoku počas behu (kontrola intervalov a zarovnania sa vykonáva až následne, aby sa takto nedala obísť zabezpečenie). Pri prepisovaní pôvodného kódu sa postupuje konzervatívne - každý potencionálny kód sa prepíše do novej sekcie. Aj prípadné dáta sa prepíšu, aby sa predišlo chybe počas behu programu, ak by bol kód označený omylom za dáta. Všetky priame skoky sú automaticky prepísané na skoky s novými adresami v *.tnew*. Nepriame skoky sa vyhodnocujú za behu, pričom ak adresa je už z novej sekcie, tak to netreba nijak špeciálne riešiť. Ak je z pôvodnej sekcie, tak sa skontroluje cieľ, či je označený značkou *0xf4* a následne sa načíta adresa za touto značkou.

Samotné znáhodnenie vykonávania sa musí samozrejme diať pri každom behu programu. Keďže ich riešenie funguje iba na platforme Windows, využili knižnicu `kernel32.dll`, ktorú v importe prepísali na ich vlastnú knižnicu - tá exportuje rovnaké symboly ako pôvodná a navyše sa postará o znáhodnenie celého načítania.

Celé riešenie *O-CFI* má spomalenie iba približne 4.7%, čo sa veľmi približuje aj riešeniam, ktoré majú prístupný zdrojový kód. Dokonca sa spomalenie malo zlepšiť na úroveň cca. 4.17% po pridaní nových inštrukcií na narábanie s intervalmi. Veľkosť binárnych súborov po prepísaní sa zvýšila približne o 137%, čo je taktiež prijateľné číslo keďže pamäť zvyčajne nebýva problém a je to iba fixná zmena, a nemení sa počas vykonávania. Úspešne sa im podarilo skontrolovať bezpečnosť riešenia vo všetkých prípadoch, ktorými sa chceli zaoberať.

CFIMon

Mnohé riešenia, ktoré sme si spomínali, využívajú binárne prepisovanie kódu alebo binárnu inštrumentáciu programu. V prvom aj druhom prípade nastáva zásah do behu programu, ktorý zvyčajne znamená netriviálne spomalenie programu. Procesory obsahujú nástroje, ktoré sa používajú na analýzu kódu

a monitorovanie behu systému alebo jednotlivých programov. Tieto nástroje sú bežnou súčasťou komerčne dodávaných procesorov, čím sa ponúkajú ako pomocný nástroj bez potreby špeciálneho hardvéru a zásahu do systému. Keďže monitorovanie prebieha na úrovni procesoru s pomocou operačného systému, programy, ktoré chceme monitorovať, nemusíme špeciálne upravovať a zároveň hardvérová podpora nám ponúka riešenie s iba malým spomalením.

Nástroj *CFIMon* [28] používa konkrétne nástroje, ktoré umožňujú sledovať riadiaci beh programu. Tieto nástroje sa používajú na monitorovanie všetkých udalostí kedy sa mení tok programu a detegujú porušenie pravidiel pre správny tok programu, ktoré si predvypočítavajú. Nástroje na monitorovanie zmeny toku programu fungujú v dvoch režimoch. Prvým prípadom je automatické prerušenie behu programu a predanie riadenia funkcii, ktorá bola na to určená. Táto funkcia, ale nemusí byť zavolaná ihneď po zmene toku a nemôžeme sa spoliehať na inštrukčný ukazovateľ, pretože procesory vo všeobecnosti nevykonávajú inštrukcie v tom istom poradí, ako sú napísané v pamäti a môže sa stať, že prerušenie nastane až 72 inštrukcií neskôr. Takéto informácie by, samozrejme, veľa nepomohli a potrebujeme presnejšie informácie. Navyše je aj možnosť zaznamenávať vetvenia programu do registrov procesora, ktorých počet je obmedzený, ale v tomto prípade to môže stačiť a vyčítame odtiaľ informácie o skoku, ktoré by sa podrobili ďalšej analýze. Našťastie existuje aj druhý prípad, ako získavať tieto informácie a tento si vybrali tvorcovia nástroju *CFIMon*. Ako jeden z argumentov pre sledovanie vetvenia sa akceptuje zásobník, do ktorého postupne procesor zapisuje hodnoty. Zároveň sa aktivuje počítadlo, ktoré počíta počet vetviacich inštrukcií a keď dosiahne určitú hodnotu, procesor vytvorí prerušenie ešte pred pretečením celého zásobníka. Pri prerušení zavolá funkciu, ktorá môže ďalej analyzovať tieto dáta a vyhodnotiť správanie programu. Skvelou výhodou takéhoto skupinového spracovávania informácií je dramatické zmenšenie počtu prerušení, čím sa znižuje spomalenie programu.

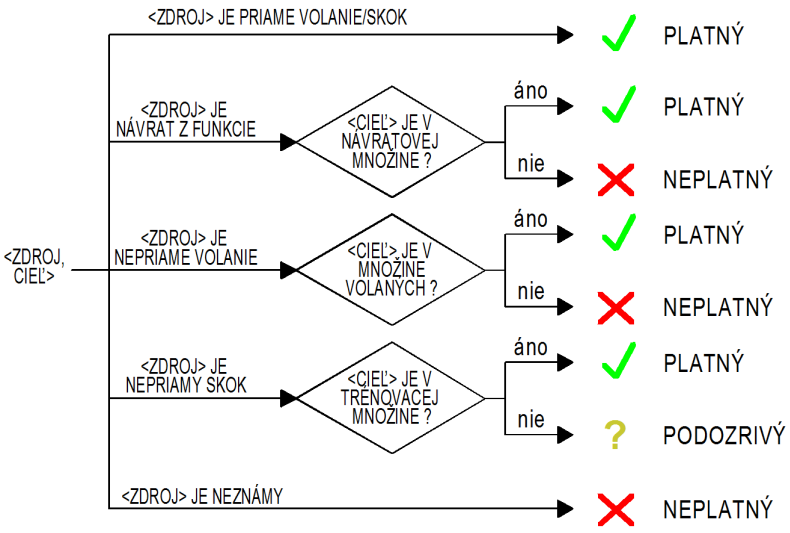
Už sme si spomenuli, ako získava CFIMon informácie o toku riadenia, druhou, takisto dôležitou, časťou je spôsob akým sa analyzuje tok. Ako sme

spomínali presné sledovanie toku, je veľmi náročné, či už vytvoriť pravidlá alebo na rýchlu implementáciu. Pri sledovaní sa zameriavame na tieto skupiny skokov: *Priamy skok, priame volanie funkcie, návrat z funkcie, nepriamy skok, nepriame volanie funkcie*. Každú z týchto skupín riešili osobitne a riešili ich vytvorením niekoľkých množín miest, kam mohli jednotlivé druhy inštrukcie skákať. Následne ako vidieť na obrázku 2.2, pozreli sa do správnej množiny a vyhodnotili, či je skok platný/neplatný/podozrivý. Množinou podozrivých skokov sa budeme zaoberať neskôr. Na obrázku vidieť, že ak je skok/volanie funkcie priame, tak je skok automaticky povolený. Množina pre návrat z funkcie je vytvorená statickou analýzou, zanalyzuje sa binárny súbor a povolia sa iba skoky bezprostredne za inštrukciami na volanie funkcie, čím sa vyrieši väčšina prípadov (ostatné sa riešia špeciálnymi pravidlami). Množina pre volanie funkcií sú adresy začiatkov funkcií, ktoré sú taktiež získané analýzou binárneho súboru. Poslednou množinou je trénovacia množina. Táto množina je získaná dynamickou analýzou - analyzujú reálne behy programov, pri ktorých si zapamätávajú adresy, ktoré majú patriť do trénovacej množiny. Keďže, ale získať dostatočný počet vstupov a dobre pokryť celú škálu možných prípadov je už notorický problém rozhodli sa akceptovať isté riziko, že adresy kontrolované pri nepriamych skokoch sa nebudú nachádzať v množine. Aby nemali príliš veľa falošných detekcií využívajú metódu posúvneho okna a v každom takomto okne povolia iba fixný počet tzv. podozrivých inštrukcií.

Hlavnou motiváciou používanie hardvérovej podpory bola rýchlosť analýzy, čo sa aj podarilo a výsledne riešenie má spomalenie v intervale 2.3% - 8.4%

HA-CFI

Podobne ako pri *CFIMon* aj tvorcovia *HA-CFI* [12] sa rozhodli využívať hardvérovú podporu pri ich implementácií, ale využili aj ďalšie možnosti, ktoré hardvérové nástroje na sledovanie ponúkajú. Základným predpokladom pri tvorbe ich nástroja je, že pri exploitácií sa tok riadenia nielen nebude



Obr. 2.2: Pravidlá v CFIMon[28]

správať podľa pravidiel, ale navyše, kvôli netradičnému toku, ktorý sa bude pri exploitácií vykonávať aj predikčná jednotka, v procesore nesprávne predikuje konkrétne vetvenie. Keď už majú predpoklad nesprávneho vetvenia, môžu využiť schopnosť filtrovať udalosti, ktoré hardvér odkladá do zásobníka a upozorňuje na ne funkciu na sledovanie. Pri používaní hardvérovej pomoci sa rozhodli, narozdiel od CFIMon, používať prvú možnosť sledovanie - procesor, čo najskôr vytvorí výnimku a zavolá funkciu na sledovanie.

Autori práce sa rozhodli sledovať a analyzovať iba nepriame volania funkcie, či už kvôli lepšiemu výkonu (menšie spomalenie), jednoduchšiemu overovaniu platnosti skoku, ale aj kvôli ich vysokému relatívnemu počtu oproti podmieneným skokom. Takto navrhnuté riešenie vytvára približne 1000 prerušení za sekundu v Internet Exploreri počas nečinnosti a počas náročnejších štandardizovaných testov to bolo aj približne 83000 prerušení za sekundu. Tieto prerušenia dokázali obsluhovať a analyzovať dostatočne rýchlo na to, aby spomalenie bolo na úrovni okolo 10% počas náročnejších testov a počas bežného používania značne menej. Tvrdia, že počas bežného používania si

nevšimli vplyv na rýchlosť prezerania webových stránok.

Ostáva nám vyriešiť problém, ako kontrolovať či je nepriamy skok, ktorý bol zle predikovaný predikčnou jednotkou dovolený, alebo nie. V práci popisujú, že tento problém riešili postupne metódou pokus-omyl. Najprv si vytvorili množinu skokov, ktorú potrebovali pokryť a následne iteratívne vytvárali zoznam povolených skokov, až kým nevyriešili všetky skoky. Konečné riešenie dynamicky pri načítavaní všetkých knižníc a samotného programu analyzuje vkladaný kód. Pri každom vložení sa pozrie funkcie, ktoré knižnica exportuje. Ďalej využívajú dáta z relokačnej tabuľky, aby hľadali funkčné ukazovatele smerujúce do sekcie `.text`. Sekciu `.text`, ešte zanalyzujú a hľadajú pomocou hľadania vzoru adresy začiatkov funkcií.

HA-CFI je riešenie, ktoré má veľký potenciál kvôli jednoduchému použitiu bez predtrénovania programu. Zároveň spôsobuje nízke spomalenie behu. Oproti *CFIMon*, ako hlavnú výhodu autori vyzdvihujú skorú detekciu, keď ich riešenie robí analýzu automaticky po vetvení, zatiaľ čo pri *CFIMon* až po naplnení zásobníka, čím sa vytvára okno pre útočníka, robiť záškodnícku činnosť. Dokonca pri porovnávaní na reálnych exploitov bola schopnosť detekcia vyššia ako nástroja *EMET* [6].

2.1.3 Analýza toku škodlivého vstupu

Analýza toku škodlivého vstupu (Taint analysis) sleduje tok vstupných dát počas exekúcie programu. Tieto dáta sú v programe zvyčajne buď kopírované, alebo ovplyvňujú iné dáta a tok programu. Sledovaním, ako sa šíri vplyv vstupu v programe môžeme detegovať exploitáciu. Aby sme sledovali, čo všetko je ovplyvnené vstupom (napr. register, pamäťové miesta) používa sa systém indukcie, nazýva sa to aj šírenie skazy. Pri analýze škodlivého toku sa počas vykonávania programu zároveň udržiavajú informácie, ktoré časti pamäte a registrov sú ovplyvnené vstupom. V prípade, ak nejaká citlivá časť pamäte alebo register je ovplyvnený vstupom (napr. inštrukčný pointer, argument systémového volania), detegovali sme exploitáciu a môžeme ju ďalej analyzovať. Existuje aj spätná analýza toku škodlivého vstupu, keď sa po-

čas exekúcie vytvárajú obrazy programu a pri prípadnom páde programu sa spätne vyhľadá, či bol vyvolaný škodlivým vstupom. Spätná analýza predchádza problémom, keď sa pri doprednej analýze toku označí ako potencionálne škodlivý až príliš veľká časť dát[3]. Pri oboch typoch analýzy sa používajú rovnaké pravidlá na šírenie vplyvu.

Systém indukcie funguje podľa ďalej popísaných pravidiel. Ak pri priradený už bola priraďovaná hodnota označená, tak aj miesto kde sa priraďuje bude ďalej označené. Pri matematických operáciach väčšinou stačí, ak jeden z operandov bol ovplyvnený a výsledok bude takisto ovplyvnený. Pri operácií logické alebo, ak neovplyvnený operand je pravda, tak nezáleží na druhom operande a výsledok je neovplyvnený. Podobne pri operácií logický súčet, ak neovplyvnený operand je nepravda, tak výsledok je neovplyvnený, napriek tomu, že druhý operand môže byť ovplyvnený. Ovplyvnené môžu byť aj výsledky rôznych operácií pôsobiacich na reťazcoch, napr. operácia na nájdenie pozície špeciálneho znaku, vypočítanie dĺžky reťazca a podobne. Slabinou tohto prístupu je, napríklad: `if (x == 0) y = 0; else if (x ==1) y = 1; ...`, keď výsledkom tohto kódu je `y = x`, ale keďže hodnota y nie je priamo ovplyvnená, tak ani nebude označená. Ak by sa v ďalšej časti vykonávania programu táto hodnota vyskytla v niektorej citlivej časti nedetegovali by sme to a útočník by mohol takto uniknúť detekcii[19].

Ako sme už spomenuli, cieľom je zistiť, či má vstup vplyv na niektoré z citlivých miest. Ako citlivé miesta sa väčšinou považujú argumenty systémových volaní, ako argument pre inštrukcie skoku/volania funkcie/návratu z funkcie. Ďalšie miesta môžu byť aj niektoré pamäťové adresy (čo je, netradičné pre analýzu vstupu bez prístupu k zdrojovým súborom), argumenty knižničných funkcií a podobne. Môže sa stať, že vstup naozaj ovplyvní citlivé miesto, ale nemalo by sa to považovať za anomáliu, pretože program predtým vykoná dostatočnú kontrolu vstupných dát. Takéto správanie sa musí vopred detegovať a uviesť ako výnimka pri spracovaní vstupu. Pri reálnych testoch sa ukázalo, že väčšina takýchto miest sa dá nájsť už pri testovaní s normálnymi vstupmi. Používatelia analýzy vstupu iba musia pred reálnym

spustením nájsť tieto výnimky, čím znížia počet falošných detekcií a zjednodušia reálne používanie tohto druhu ochrany.

V reálnom nasadení sa môže, ako veľkou výhodou ukázať možnosť automatického vytvárania signatúr zlých vstupov, ktoré môžu byť automaticky poskytované ďalším systémom detegujúcich útoky, a tak ochrániť mnoho ďalších systémov bez nutnosti použitia dynamickej analýzy [19].

2.2 Detekcia pomocou diverzity

Detekcia pomocou diverzity je technika, pri ktorej sa deteguje s vysokou pravdepodobnosťou veľké množstvo rôznych tried útokov. Namiesto toho, aby sa spúšťala iba jedna inštancia programu a tá bola potencionálne napadnutá, a následne detegovaná exploitácia. Spúšťa sa väčšie množstvo procesov s rovnakým vstupom, a následne sa detegujú odlišnosti a vyhodnocuje sa, či mohli byť spôsobené normálnym správaním, alebo prebiehajúcou exploitáciou. Výhodou tejto metódy je, že útočník na úspešný útok potrebuje naraz exploitovať niekoľko procesov pomocou rovnakého vstupu. Ďalšou výhodou je nezameriavanie sa na konkrétne triedy exploitov, a tak má vyšší potenciál oproti technikám, ktoré sa snažia zabrániť iba konkrétnym triedam.

Aby sa znížila šanca na úspešnú exploitáciu viacerých inštancií naraz, využívajú sa rôzne techniky. Hlavnou technikou je vkladanie rôznych odlišností do exekúcie programu. Tieto odlišnosti bývajú niekedy kontrolované kľúčmi, aby sa uľahčila diverzifikácia jednotlivých behov. Jedným z prvých možnosti odlíšenia je odlišné rozloženie pamäte, čo môže byť automaticky zabezpečené pomocou ASLR. Útočník by teda musel vytvoriť exploit, ktorý by bol nezávislý na rozložení pamäte. Ďalším spôsobom je zmena inštrukčnej sady. Rôzne exekúcie budú virtualizované pomocou odlišných inštrukčných sád, čím sa znefunkčnia binárne kódy vložené behu programu. Tiež môžeme náhodne iniciovať systémové volania, názvy súborov, volacie konvencie, konfigurácie systému, mená užívateľov, heslá a iné nástroje, ktoré by mohli potencionálne ovplyvniť vykonávanie programu.

Problémom pri tejto technike je aj vyhodnocovanie pokusu o exploitáciu. Musíme si uvedomiť, že veľa výpočtov býva znáhodnených operačným systémom, a preto väčšinou nestačí iba porovnávať výstupy. V takomto prípade treba napísať špeciálny validátor vstupu, ktorý je odolný voči týmto odlišnostiam. Niekedy je ale výstup príliš stručný, aby sa tam dali detegovať nejaké odlišnosti, prípadne je výpočet príliš dlhý a nechceme čakať s detekciou až na ukončenie výstupu a jeho výpis. Potrebujeme použiť iný spôsob detekcie (môže nastať aj z iných dôvodov). Ďalší využívaný spôsob je, napríklad porovnávanie postupností systémových volaní a ich argumentov, čím sa môžu porovnávať 2 rôzne inštancie programu navzájom [13].

2.3 Detekcia hľadaním anomálií

Pri hľadaní anomálií sledujeme program počas jeho vykonávania a hľadáme anomálie, ktoré by nám pomohli detegovať exploitáciu útočníkom. Pri hľadaní anomálií používame niekoľko spôsobov sledovania programu pričom ich zvyčajne rozdeľujeme do 3 kategórií:

- **Sledovanie čiernej skrinky** zahrňuje v provom rade komunikáciu programu s operačným systémom. Táto metóda je najhrubšia zo všetkých a má najnižšiu úroveň detailu. Výhodou je relatívne malý počet výskytov, čím sa pri sledovaní nevytvára príliš veľa zdržania. V druhom rade sa v nedávnej dobe na detekciu anomálií začali používať informácie z procesora, o čom si viac povieme v ďalších kapitolách.
- **Sledovanie na úrovní inštrukcií** je najpresnejšie možné meranie. Vzhľadom na to, že obsahuje príliš veľa často zbytočných detailov a spôsobovalo by to veľké spomalenie vykonávania programu. Pre predídenie prílišného spomalenie sa zvyčajne monitoruje iba podmnožina všetkých inštrukcií. Medzi podmnožiny môžu patriť, napríklad volania funkcií, návrat z funkcií či inštrukcie podmienených, alebo nepodmienených skokov.

- **Sledovanie na vyšších úrovniach** zahrňuje sledovanie programu, keď ako výstupná postupnosť sledovania je, napríklad postupnosť volaní knižničných funkcií, systémových volaní, potencionálne aj s argumentami.

V najbližších častiach si postupne spomenieme reálne príklady hľadania anomálií. Spomenieme si sa dá vytvárať model správania programu na základe udalostí (hlavne systémových volaní). Tieto modely sú od najjednoduchších - modelovanie pomocou n-gramov až k zložitejším keď modelujeme program pomocou pravdepodobnostných metód. Ukážeme si tiež metódu na hľadanie anomálií v dlhých postupnostiach, ktorá dokáže kontrolovať súvislosti medzi udalosťami, ktoré sa vykonávajú ďaleko od seba, a využíva to na detekciu exploitácie. Zaujímavým spôsobom na hľadanie anomálií sa využívajú aj rôzne druhy hardvérových ukazovateľov. Pomoc funkcionality zabudovanej v hardvéri je dôležitým aspektom na detekciu ROP útokov, ktorý nám umožňuje efektívne získavanie informácií o behu a viac si o tom spomenieme v tejto kapitole.

2.3.1 Hľadanie anomálií v postupnosti systémových volaní

Vhodným kandidátom na zaznamenávanie udalostí pri behu programu je komunikácia programu s operačným systémom pomocou systémových volaní. Zaznamenávať komunikáciu s operačným systémom je väčšine systémov jednoduché, má jasné pravidlá a nevyžaduje žiadny zásah do spúšťaného kódu. Zároveň sú systémové volania zväčša nie príliš časté udalosti, aby vznikalo príliš veľké spomalenie vykonávania procesov. V neposlednom rade väčšina exploitov vyžaduje nejakú komunikáciu s operačným systémom, aby využila potenciál prístupu k programu a vykonala záškodnícku činnosť, na ktorú bola určená.

Väčšina riešení, ktoré hľadajú anomálie v postupnosti systémových volaní porovnáva pozorované správanie so správaním získaním jedného z dvoch

spôsobov - statickou analýzou binárneho kódu alebo získaním modelového správania. V oboch prípadoch sa pred každým systémovým volaním skontroluje správanie programu a hľadajú sa postupnosti, ktoré sú dostatočne odlišné od očakávaných postupností. Výhodou takéhoto modelovania je, že sa nespolieha na konkrétne formy správania exploitov, ktoré by sa snažil detegovať, ale keďže toto správanie nie je explicitne definované, môžu sa potencionálne detegovať aj neznáme spôsoby exploitácie.

Jednou z nevýhod hľadania anomálií je akceptácia postupnosti volaní, ktoré nepatria k normálnemu správaniu. Dôvodom môže byť či už príliš optimistická statická analýza zdrojového kódu, voľná definícia netradičného správania, alebo malá detailnosť zaznamenávaných dát. Útočník môže, napríklad zavolať očakávané systémové volanie s neočakávanými argumentami, umelo pridať sekvenciu systémových volaní tak, aby nevytvárala žiadne anomálie v správaní. Tento druh obídenia detekcie anomálií sa volá *mimicry* útok.

Výskumníci si veľmi rýchlo uvedomili, že modelovať správanie iba na základe nejakého diania, je pomerne jednoduché a pridali k čistému modelovaniu systémových volaní aj ďalšie informácie. Napríklad namiesto toho, aby iba zaznamenali, že sa udialo systémové volanie, zaznamenali aj jeho adresu v pamäti. Ďalej pri modelovaní začali používať aj argumenty systémových volaní, takže už nevedeli iba to, že sa zapisuje do nejakého súboru, ale vedeli aj kontrolovať rôzne ďalšie údaje (napr. adresa súboru (ak zvykla byť fixná), prístupové práva, ...). Takto mohli modelovať postupnosti systémových volaní ešte podrobnejšie.

Niektoré riešenia prehľadávajú jednotlivé rámce na zásobníku, aby si vytvorili postupnosť vnorení funkcií, ktorú taktiež používali na modelovanie správania. Iné porovnávajú iba zmeny v postupnosti volaní (najmenšiu zmenu z jednej postupnosti funkcií na zásobníku ku druhej). Väčšina týchto vylepšení modelu vedie k presnejšiemu modelu, čím sa zmenšuje priestor pre *mimicry* útoky a zlepšuje tak ochranu. Problémom sa ale mohlo stať, že príliš presným modelom, ktorý bol vytvorený pozorovaním predchádzajúcich behov programu, sme mohli dostať riešenie s príliš veľkým počtom falošných

hlásení, ktoré sa ťažko filtrovali [18].

V nasledujúcich častiach podkapitoly si spomenieme niekoľko spôsobov, ako modelovať systémové volania[23].

Modelovanie pomocou konečných deterministických automatov

Najjednoduchší model, môžeme ho vytvoriť buď statickou analýzou programu, alebo jeho dynamickým pozorovaním. Jednotlivými stavmi automatu je číslo systémového volania a inštrukčný pointer miesta, odkiaľ bol zavolaný. Hrany sa následne vytvoria medzi vrcholmi, ktoré boli nájdené pri dynamickom pozorovaní bezprostredne za sebou alebo existuje medzi nimi priama cesta nájdená pri statickej analýze [15].

Modelovanie pomocou n-gramov

Modelovanie pomocou n-gramov je metóda ([14]), ktorá používa fragmenty vykonávanej sekvencie systémových volaní na to, aby detegovala anomáliu. N-gram v našom prípade je n volaní dlhá súvislá pod sekvencia systémových volaní. Využívanie n-gramov sa spolieha na to, že krátke podpostupnosti stačia na detegovanie anomálnych úsekov od zlých. Test anomálie sa robí postupne pri vykonávaní programu, pričom sa po každom systémovom volaní kontroluje, či posledných n systémových volaní netvorí anomálnu sekvenciu. Test zvyčajne pozostáva z overenia, či už bola zaznamenaná pri vytváraní modelu programu.

Modelovanie pomocou zásobníkových automatov

Modelovanie pomocou zásobníkových automatov je metóda, ktorá má väčšiu vyjadrovaciu silu ako konečné automaty. Výhoda zásobníkových automatov je ich schopnosť simulovať zmeny na zásobníku programu. Deterministické automaty nedokážu modelovať priame alebo nepriame volanie funkcií. Na druhej strane zásobníkové automaty sa môžu využiť na simuláciu všetkých volaní, a tak dokážu zaznamenať a overiť aj rekurzívne vzťahy.

Problémom však je konštrukcia zásobníkových automatov, keďže iba znalosť zásobníka programu pri jednotlivých systémových volaniach nie je dostatočná. Napríklad v článku [27] spomínajú tvorbu zásobníkového automatu pomocou statickej analýzy kódu.

Modelovanie pomocou pravdepodobnostných metód

Pravdepodobnostné metódy na modelovanie vykonávania programu a následné hľadania anomálií je jednou z ďalších možností, ako detegovať exploitáciu. Ako prostriedky pre modelovanie sa používali, napríklad skrytý markovov model, rôznych klasifikačných pravdepodobnostných metód, metód strojového učenia, prípadne ďalších. Keďže pravdepodobnostné modely sú často obmedzené vo vyjadrovaní rekurzívnych vzťahov, majú vyjadrovaciu silu približne na úrovni deterministického konečného automatu.

V skrytom markovovom modeli sa zvyčajne ako skryté stavy používajú jednotlivé udalosti, ktoré môžu nastať (napr. systémové volanie), pričom aj ako pozorované symboly sa používajú tie isté udalosti, kde sa v stave reprezentujúci konkrétny symbol môže pozorovať iba zodpovedajúci symbol [29].

Hybridný prístup pravdepodobnostného modelovania a statickej analýzy kódu

Výhodou pravdepodobnostného modelovania je zachytenie informácie o tom, ako často sa jednotlivé udalosti vyskytujú v postupnostiach, ktoré sú dostupné na trénovanie. Problémom sú podpostupnosti, ktoré sa nenachádzajú v trénovacej sade a pri ich vyhodnocovaní nám spôsobujú problémy. Konkrétne zvyšujú počet falošných hlásení, čo ma značne nepriaznivý účinok na využitie týchto metód. Naopak, metódy, ktoré modelujú program iba na základe statickej analýzy, neobsahujú informácie o tom, ako vyzerá normálny a beh a preto považujú za normálny beh, akýkoľvek ktorý mohol byť potencionálne vytvorený programom - čím sa vytvára veľký priestor na nedetegovanie anomálneho behu programu.

V práci [29] sa skupine výskumníkov podarilo úspešne vytvoriť riešenie, ktoré spája statickú analýzu programu a pravdepodobnostné modelovanie do jedného systému, čím vytvorili riešenie nazývané *STILO*. *STILO* najprv staticky analyzuje program a vytvorí graf programu. Tento graf programu sa využije na vypočítanie odhadovaných pravdepodobností na prechod medzi jednotlivými časťami v programe. Najprv pre jednotlivé funkcie osobitne, a následné všetky tieto informácie spoja do globálnej tabuľky prechodov. Všetky doteraz získané informácie v agregovanej podobe sú použité na inicializovanie skrytého markovového modelu, ktorý je následne dotrénovaný klasickým postupom.

Pri testovaní sa im podarilo detegovať množstvo rôznych útokov a zároveň sa im podarilo zvýšiť presnosť, oproti klasickému modelu iba so skrytým markovovým modelom 11-28 krát, čo naznačuje, že využitie predspracovania programu, môže mať veľký vplyv na kvalitu výstupného modelu.

2.3.2 Využívanie hardvérových ukazovateľov

Mnohé procesory podporujú sledovanie aplikácií na úrovní procesora a poskytujú rôzne informácie o tom, čo sa udialo. Medzi informácie, ktoré poskytujú patrí napríklad: počet načítavacích inštrukcií, počet ukladacích inštrukcií, počet aritmetických/vetviacich inštrukcií, počet volaní funkcií (blízkych, priamych, nepriamych), počet zavolaní návratových inštrukcií. Medzi ďalšie patria počet nenájdených údajov v rôznych cache pamätiach, procesorom zle predpovedané inštrukcie návratu/volania funkcie/vetvenia. Tieto, ale aj ďalšie informácie sa zdajú byť vhodným kandidátom na hľadanie anomálií spôsobených škodlivým kódom. Prvým dôvodom je predpoklad, že škodlivý kód bude mať iné vlastnosti, ktoré by sa mohli dať namerať pomocou skôr popísaných počítadiel a druhým je relatívne malé spomalenie vykonávanie programu. Existuje viacero pokusov, ktoré využívajú hardvérovú pomoc, ako sme už aj spomínali pri *CFIMon* alebo *HA-CFI* v kapitole 2.1.2 na stranách 20. a 22. . Teraz si spomenieme niekoľko spôsobov, ktoré sa nezameriavali iba kontrolu toku programu, ale hľadali anomálie.

Detekcia pomocou strojového učenia a hardvérových ukazovateľov

Prvá metóda je popísaná v článku [25], kde vyskúšali niekoľko možností detekcie anomálií. Riešenie, ktoré vytvorili sa muselo vysporiadať s niekoľkými problémami - procesor dokáže merať iba niekoľko počítadiel naraz (v ich prípade to bolo 4), jednotlivé počítadla nedokázali detegovať anomálie, vonkajšie vplyvy na beh programu (napríklad počet nenájdených hodnôt v pamäti cache je závislý aj od ostatných programoch bežiacich na tom istom procesore). Výsledné riešenie je dostatočne robustné, aby zohľadnilo tieto problémy a dosahuje nízke množstvo falošných označení normálnych behov za anomálie, pričom dokáže detegovať reálne exploity v programoch ako je Internet Explorer, či prehliadač pdf súborov. Navrhli taktiež metódu, ktorá by výrazne sťažila tvorbu mimicry útokov. Vzhľadom na obmedzenie počtu sledovaných počítadiel, ich môžu náhodne kombinovať a zároveň meniť dĺžku jednotlivých úsekov, v ktorých vyhodnocujú anomálie tak, aby mali útočníci ťažšie predikovať vypočítaný model a vytvoriť skrytý útok, ktorý by nebol detegovaný.

Eunomia

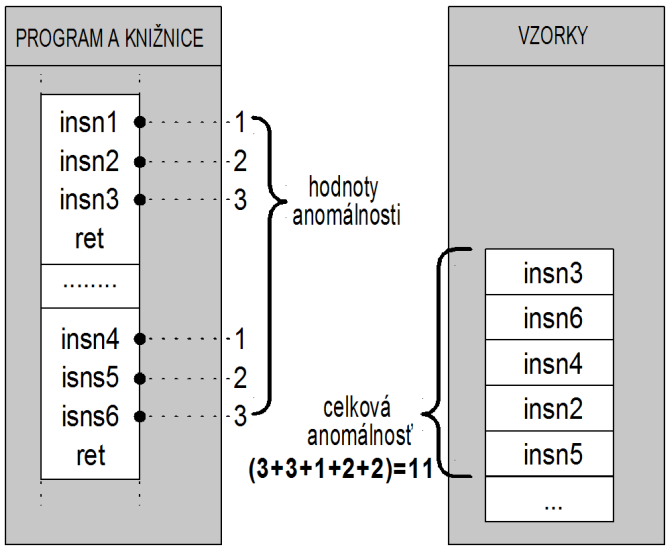
Eunomia [30] je nástroj, ktorý ako prvý využíval hardvérovú pomoc na detekciu anomálií v programe. V tejto práci sa zameriavajú na 3 rôzne druhy útokov a na každý používajú mierne iný prístup. Prvým je vkladanie kódu z kapitoly 1.1, druhým je útok s použitím inštrukcií return z kapitoly 1.2.2 a posledným - tretím je skákanie do štandardnej knižnice z kapitoly 1.2.1. My si v krátkosti popíšeme, ako sa snažili detegovať jednotlivé anomálie.

Na detekciu útoku s vkladaním kódu, používajú možnosť zaznamenávať nepredikované skoky, ak takéto skoky smerujú do dátovej časti, tak je to s vysokou pravdepodobnosťou skok do vloženého kódu. Takéto riešenie je približne rovnako dobré, ako zabránenie vykonávať kód z dátovej časti, jediný rozdiel môže byť schopnosť prispôsobiť sa konkrétnym programom a povoliť pre konkrétne aplikáciu im špecifické skoky, ktoré sa dajú získať analýzou behu.

Útoku skokom do štandardnej funkcie sa snažia zabrániť už klasickým spôsobom pri hľadaní anomálií. Najskôr spustia program do predpripravených vstupoch a pre každú knižnicu získajú množinu adries, odkiaľ sa tam môže skákať. Následne detegujú všetky skoky do štandartných funkcií a overujú, či sa nachádzajú v získaných množinách, a ak nie, tak vyhlásia detekciu exploitácie. Jednoducho by sa to dalo obísť, medzi skokom na miesto odkiaľ sa zvykne skákať do konkrétnej štandardnej funkcie. To by nebolo veľmi bezpečné, a preto ešte navyše kontrolujú, či predchádzajúci skok nebol na to konkrétne miesto. Príklad je postupnosť skokov `addr0 -> addr1 a addr1 -> addrKniznica`, takéto medziskoky sa detegujú a až na výnimky (získane dynamickou analýzou) sú všetky zakázané. Na získavanie informácií o skokoch používajú už skôr popísanú možnosť získavania postupnosti skokov z procesora, pričom preferujú metódu, pri ktorej sa sekvencia skokov zapíše do predpripraveného zásobníka a po naplnení zásobníka sa všetky skoky skontrolujú.

Detekcia využívania inštrukcií return na vytvorenie sekvencie skokov pomocou návratových hodnôt je asi najzaujímavejšia časť nástroja *Eunomia*. Táto metóda je ukázaná na obrázku 2.3. *Eunomia* pre každý zásobník, ktorý kontroluje na prítomnosť ROP útoku, vypočíta hodnotu podozrenia útoku, ktorá je súčtom týchto hodnôt pre jednotlivé skoky. Ukazuje sa, že väčšina jednotlivých častí v *ROP* útoku je dĺžky 1,2 alebo 3 (väčšie sú nepraktické a ťažko sa používajú). V programe si prepvypočítajú všetky miesta, ktoré sa nachádzajú tesne pred inštrukciami return . Následne, pre každý skok vypočítajú jeho vzdialenosť od ďalšej inštrukcie return a obodujú pre vzdialenosť 1 3 body, pre vzdialenosť 2 2 body a pre vzdialenosť 3 1 bod, pre ostatné sa dáva 0 bodov. Ak celkový súčet v zásobníku prekročí kritickú hodnotu (v testoch používajú hodnotu 15), tak označia beh ako anomálny.

Nástroj *Eunomia* je praktický nástroj, ktorý sa dá pomerne jednoducho bez úpravy programu použiť na detekciu viacerých druhov útokov. Jeho výhoda je tiež v nízkom spomalení programu iba na úrovní 4.7%. Výhodou použitia zásobníka je aj možnosť získať cenné informácie o útoku, keď si po



Obr. 2.3: Detekcie ROP v Eunomia[30]

detekcii môžeme prečítať informácie, kde a ako postupoval útočník (samozrejme, iba pokiaľ sa to stále nachádza v zásobníku).

2.3.3 Anomálie v extrémne dlhých postupnostiach

Nie všetky útoky vytvárajú nelegálny tok programu. Niektoré útoky dokážu prepísať riadiace premenné programu tak, aby tok programu išiel iba po normálnych cestách, ale útočník aj tak dosiahne svoj záškodnícky cieľ. Takéto útoky sme popisovali, napríklad v kapitole 1.3. Tento druh útokov sa nemusí dať lokálne detegovať a na to, aby sme ich detegovali, potrebujeme korelovať údaje, ktoré sa nachádzajú veľmi ďaleko od seba v behu programu. Priamočiare rozšírenia predchádzajúcich riešení nefungujú, a preto sa v článku [22] pokúsili adresovať tento problém.

Predtým, ako sa dostaneme k funkčnému riešeniu, popíšeme si problémy priamočiarych rozšírení. Rozšírením veľkosti okna pri kontrole n-gramov (alebo pravdepodobnostných metód využívajúcich markovov model) sa do-

staneme k problémom s trénovaním. Metódy, ktoré pozorujú krátke okná a v nich vyhodnocujú či sú, alebo nie sú anomálne, narážajú pri zväčšovaní okna na problém, získať dostatočné množstvo trénovacích dát. Zvyčajne sa počet potrebných dát na dostatočné trénovanie zvyšuje exponenciálne, a tak už aj modely s n o veľkosti približne 40 sú nepraktické a majú príliš veľké množstvo falošných hlásení.

Rozšíriť riešenia môžeme aj získavaním informácie o frekvencií výskytov jednotlivých častí (napr. n-gramov, prechodov v deterministickom automate). Takéto riešenia dokážu úspešne detegovať napríklad DOS útoky. Problémom je, že nedokážu jednotlivo korelovať údaje medzi udalosťami, čím nedokážu detegovať väčšinu útokov skrývajúcich sa v dlhej postupnosti. Podobne aj iné prístupy sa ukazujú ako nedostatočné.

Riešenie, ktoré navrhli v článku [22] využíva rozdelenie všetkých behov do klastrov. Táto klasterizácia sa ukázala ako vhodný nástroj na detekciu útokov, ktoré sa skrývajú v dlhých behoch. Údaje, ktoré sa používajú na detekciu anomálií, sú frekvencie prechodov medzi funkciami v programe. Tieto prechody medzi funkciami si ukladajú do matice prechodov. Následne pomocou nej trénujú model.

Trénovanie modelu postupuje v dvoch fázach. V prvej fáze rozdelia všetky trénovacie behy do klastrov a v druhej fáze vykonávajú frekvenčnú analýzu volaní funkcií v rámci jednotlivých klastrov. Na klasterizáciu využívajú binárnu podobu frekvenčnej matice. Matica klastra sa priebežne počas procesu mení, pričom aktuálna matica klastra je disjunkcia všetkých matíc behov, ktoré sa v ňom nachádzajú. Pri detekcii anomálií hľadajú klastre, ktoré obsahujú všetky hodnoty `1` v matici klastra na miestach, kde sa nachádzajú v maticu behu. Takáto kontrola pomôže nájsť útoky, ktoré sa snažia spojiť 2 nekompatibilné časti spolu. Príkladom môže byť zraniteľnosť v SSH, ktorú sme popísali v kapitole 1.3.

V druhej fáze využívajú frekvenčnú analýzu. Táto frekvenčná analýza sa skladá z dvoch častí. Prvá časť pre každú dvojicu funkcií v matici skontroluje, či sa frekvencia výskytov nachádza v intervale výskytov trénovacích

dát patriacich do daného klastra. Pre všetky možné dvojice funkcií sa nájde minimálna a maximálna hodnota v maticiach behov a tieto sa použijú ako začiatok a koniec intervalu, do ktorého sa očakáva, že budú patriť všetky behy normálnych programov.

Druhá časť koreluje všetky hodnoty v matici pomocou SVM modelu. Model trénuje na predpripravených dátach. Dáta sú predpripravené tak, že všetky hodnoty v matici sú modifikované funkciou $f(x) = log(x+1)$, aby sa znížila rozptyl hodnôt v tabuľke. Následne ešte používajú metódy PCA alebo FVA na zníženie počtu dimenzií vstupných dát. Následne sa natrénujú na týchto dátach SVM. Pri kontrole sa zistí, či beh b patrí do SVM modelu, ktorý sa vytvoril pri trénovaní.

V druhej časti detekcie anomálií sa riešenie zameriava na útoky, ktoré zvláštnym spôsobom menia frekvencie výskytov volaní funkcií.

Výsledné 2 časti fungujú nezávisle, keď kontrola behu programu prebieha až po jeho ukončení. Riešenie, ktoré implementovali bolo schopné detegovať rôzne druhy útokov - napr. spomínany útok na SSH, DOS útok na regulárne výrazy a iné.

Kapitola 3

Popis vytvoreného riešenia

Cieľom našej práce je vytvoriť nové riešenie, ktoré by pomohlo detegovať exploitáciu v binárnych programoch. V predchádzajúcich častiach práce sme si spomínali množstvo prístupov, ktoré už boli navrhnuté, implementované a otestované. Možné prístupy, ktoré sa nám ponúkali, sa dajú zaradiť do 2 hlavných kategórií.

Prvou kategóriou je detekcia a zabránenie úspešnej exploitácií popísanej v kapitole 2.1. Tieto prístupy ([17], [28], [12], [20]) sa zvyčajne zameriavajú na konkrétne druhy útokov, ktoré preberú tok programu, kde útočník využíva implementačné chyby, ktoré vedu k chybe pri spracovaní a umožňujú útočníkovi vykonávať jeho vlastný kód (či už priamo, alebo nepriamo pomocou využívania existujúceho kódu).

Druhou kategóriou je hľadanie anomálií, kde sa viacero riešení ([30], [19]) snažilo detegovať prevzatie toku riadenia útočníkom (podobne ako v predchádzajúcej kategórií). Ďalšie modelovali správanie programu a kontrolovali či je, alebo nie je anomálny. Robili to na základe malého okna, ktoré sledovali (n-gramy, modelovanie pomocou skrytého markovového modelu), alebo individuálnych nelegálnych zmien toku programu (napr. modelovanie pomocou automatov). Zaujímavým riešením bolo tiež klasifikovanie anomálnosti pomocou aktuálneho stavu a vývinu hardvérových ukazovateľov (popísovali sme ho na strane 33).

Takmer osobitnou časťou v druhej kategórií bola práca z kapitoly 2.3.3, ktorej cieľom bolo detegovať útoky, ktoré sa nedajú detegovať iba s lokálnou informáciou, ale je potrebná korelácia vzdialených udalostí a informácia o dlhej časti behu programu. Útoky, ktoré sú hlavným záujmom práce, bývajú typicky také, ktoré nepreberajú riadenie programu, ale pozmenia ho pomocou zmeny lokálnych premenných alebo využívajú logické chyby programu, ktoré dovolia útočníkovi rôznym spôsobom získať, alebo pozmeniť dôležité informácie (niektoré príklady sú popísané v kapitole 1.3).

My sme sa rozhodli vytvoriť riešenie z druhej kategórie a konkrétne sme sa snažili vytvoriť nástroj schopný detekcie útokov a chýb, ktoré nepreberajú tok programu. Ako inšpiráciu a niektoré techniky sme prebrali z práce spomínanej v článku [22] (a v tejto práci v kapitole 2.3.3).

V tejto kapitole si postupne popíšeme všetky dôležité aspekty nášho riešenia. Začneme popisom bezpečnostného modelu, ktorý bude popisovať aké druhy chýb a útokov budú stredobodom nášho záujmu. Ďalej popíšeme naše samotné riešenie a implementačné detaily. Súčasťou popisu bude taktiež popis spôsobu testovania, výsledky testovania a následne popíšeme výhody a nevýhody nášho riešenia.

3.1 Bezpečnostný model

Cieľom je detegovať zneužívanie programov, ktoré nevkladá vlastný kód do toku programu (či už priamo pomocou vkladania kódu, alebo zneužitím už existujúceho kódu ako pri metódach skákajúcich do štandartných funkcií, metódach využívajúcich inštrukcie návratu z funkcie, alebo inštrukcie skoku).

Takéto zneužívanie programu nemusí vytvárať ilegálny tok programu ani ilegálne sekvencie systémových volaní. Zameriavame sa na zraniteľnosti, ktoré sú anomálne nie v krátkych úsekoch, ale zneužívajú program na účely, ktoré sa tradične nepoužíva alebo sú nekonzistentné s normálnym tokom programu. Kategóriou útokov, ktorú chceme detegovať, sú útoky nepreberajúce tok riadenia programu popísané v kapitole 1.3.

3.2 Popis riešenia

Podobne ako pri väčšine riešení hľadajúcich anomálie sa aj naše riešenie skladá z dvoch nezávislých častí, každá má inú úlohu.

Prvá časť sa zameriava na zbieranie údajov o bežiacom programe. Pod zbieraním údajov rozumieme sledovanie programu na to, aby sme získali sekvencie udalostí, ktoré chceme skúmať a v ktorých chceme hľadať anomálie.

Následne úlohou druhej časti nášho riešenia je získané dáta analyzovať a natrénovať model programu. Model programu je následne použitý pri ostatných behoch ako referenčný model a skúma sa či aktuálny model je alebo nie je anomálny.

My si v tejto podkapitole postupne rozoberieme podstatné časti nášho riešenia. Najprv si popíšeme rôzne druhy dát, ktoré môžeme zbierať a sú vhodnými kandidátmi ako dáta na modelovanie správania. Následne si popíšeme spôsob trénovania a vyhodnocovania anomálií.

3.2.1 Pozorovanie behu programu

Prirodzený spôsob, ako sledovať program, je sledovať postupnosť systémových volaní, volaní funkcií alebo základných blokov nachádzajúcich sa v programe. Tieto postupnosti sú postupne od najmenej podrobných po najpodrobnejšie. My sme sa rozhodli sledovať na úrovni základných blokov, pričom jednotlivými udalosťami sú prechody medzi jednotlivými základnými blokmi. Výhodou sledovania na úrovní základných blokov je presný popis behu programu.

Podobne ako v práci [22] sme sa rozhodli pre každý beh pamätať iba konštantné množstvo informácie pre každú udalosť, ktorá nastala. Nepamätáme si celú postupnosť behu programu, ale pri každom výskyte udalosti si môžeme aktualizovať informácie o nej. Základnou informáciou, ktorú si budeme pamätať je či sa daná udalosť vyskytla, alebo nevyskytla.

Keďže si nepamätáme celé postupnosti udalostí, tak sme sa rozhodli pamätať prechody medzi základnými blokmi, namiesto iba jednotlivých blo-

koch osobitne. Sledovanie dvojíc nám pomôže odlíšiť, napríklad nasledujúce 2 behy:

- `A -> B -> C -> D -> E` (obsahuje dvojice: AB, BC, CD, DE)
- `A -> B -> D -> C -> E` (obsahuje dvojice: AB, BD, DC, CE)

Ak by sme sledovali iba jednotlivé bloky v obidvoch behoch by sme získali množinu `A, B, C, D, E`. Takto je možné odlíšiť volania tých istých funkcií z iných častí kódu a zvýšime pravdepodobnosť detekcie zneužitia programu.

Binárna informácia o každom bloku je veľmi zaujímavá a už v predchádzajúcich prácach sa ukázala veľmi užitočná, a dostatočná na odhalenie rôznych útokov. V našej práci sme sa rozhodli k binárnej informácií zaznamenávať aj informáciu o prvom/poslednom výskyte danej udalosti. Pre každú udalosť si uložíme sekvenčné číslo prvého a posledného výskytu. Táto informácia nám pomôže pri detekcii, keď previaže aj udalosti, ktoré nie sú inak priamo prepojené. Hlavné vzťahy medzi udalosťami, ktoré nám tieto informácie poskytujú, sú náväznosť - ktorá iná udalosť sa vždy vyskytuje pred konkrétnou udalosťou, alebo, ktorá iná udalosť sa vždy vyskytuje po pozorovanej udalosti. Takáto udalosť sa dá uložiť do konštantného množstva pamäte.

Sledovanie základných blokov môže spôsobiť či už veľké časové nároky na ich spracovanie, alebo veľké pamäťové nároky na ich uloženie. My sme sa rozhodli inšpirovať programom na fuzzovanie - *American fuzzy loop* [1] (skrátene *AFL*). Základných blokov v programe býva zvyčajne na úrovni niekoľko tisíc až niekoľko desaťtisíc. Potencionálnych prechodov medzi nimi môže byť kvadraticky veľa, čo je niekoľko miliónov. Obrovské množstvo z nich sa v programe nevyskytuje a výsledne programy obsahujú rádovo iba lineárne veľa rôznych prechodov (vzhľadom na celkový počet základných blokov). *AFL* pri zbieraní informácií o hranách priradí každej hrane náhodné číslo, toto číslo sa následne používa na indexovanie do tabuľky, kde si ukladá svoje dáta. Náhodné indexy do tabuľky sú generované nezávisle od seba, a tak sa môže stať, že 2 rôzne udalosti majú rovnaký index. Tento problém je ale daňou, za jednoduchšie a potencionálne rýchlejšie zbieranie údajov. Navyše

veľkosť tabuľky do ktorej sa ukladajú dáta môže byť parametrizovaná tak, aby sa prispôsobila počtu prechodov v programe, a tak budú vznikať kolízie iba sporadicky. Navyše v neskoršej kapitole 3.5.4 ukážeme, že takéto kolízie nemajú veľký vplyv na naše výsledky.

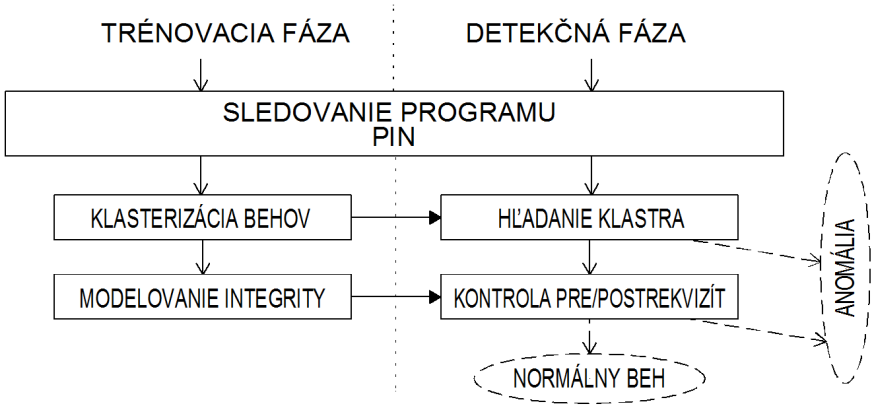
3.2.2 Architektúra modelu

Náš model sa skladá z dvoch častí, každá časť má trénovaciu zložku a detekčnú zložku.

- Prvá časť využíva iba binárnu informáciu o udalostiach. Informácie o tom či konkrétna udalosť nastala, alebo nenastala v konkrétnom behu sa uloží do množiny výskytu. Najprv máme pre každý beh programu osobitnú množinu výskytu. Tieto množiny následne spájame do klastrov. V tejto časti riešenia detegujeme útoky, ktoré spájajú niekoľko častí programu do behu, ktorý sa síce skladá iba z jednotlivo normálnych častí, ale ich spoločný výskyt v jednom behu je anomálny.
- Druhá časť nášho riešenia operuje v každom klastri osobitne. Využíva či už binárne informácie, alebo informácie o prvom/poslednom výskyte konkrétnych udalostí. V tejto časti vykonávame kontroly integrity programu na základe relatívneho usporiadania udalostí. Kontrolujeme vzájomné vzťahy medzi udalosťami pričom využívame aj relatívne usporiadanie medzi prvými/poslednými výskytmi udalostí. Taktiež, navyše oproti predchádzajúcej fáze kontrolujeme pre jednotlivé udalosti, či sa vždy vyskytovali/nevyskytovali za prítomnosti ostatných udalostí.

Celkovú architektúru môžeme vidieť na obr. 3.1. Na pravej strane obrázka vidieť trénovacie zložky a na pravej strane vidieť detekčné zložky. Do jednotlivých častí idú získané informácie o behoch programu. Pri trénovaní programu sa najprv pozoruje dostatočne veľká množina behov, ktorá sa naraz poskytne trénovaciemu programu. Ten v prvej časti programu rozdelí všetky behy do klastrov a v druhej časti trénuje jednotlivo každý klaster osobitne.

Pri detekcii sa jednotlivé behy v prvej časti nášho riešenia pokúsia nájsť vhodný klaster, do ktorého potencionálne patria, ak sa žiadny taký klaster nenájde, tak beh označíme za anomálny. Ak sa nejaké klastre nájdu, vyberieme najvhodnejší z nich a vykonáme kontroly, ktoré si neskôr popíšeme. Ak je niektorá z kontrol neúspešná tak beh označíme za anomálny, inak za normálny.



Obr. 3.1: Architektúra navrhovaného riešenia

Klasterizácia behov

Klasterizácia behov, ktorú implementujeme adresuje 2 problémy:

- Veľké množstvo malých klastrov
- Problémy behov, ktoré sú na rozhraní klastrov

Malé klastre neposkytujú dostatok informácií pre trénovanie v ďalších fázach, čo zníži kvalitu výsledného riešenia. Niektoré štandardné riešenia klasterizácie taktiež nerátajú so situáciou, keď jeden beh programu môže byť zaradený do jedného klastra, ale následne bude detekčná fáza prebiehať v inom klastri. My používame spôsob klasterizácie behov z článku [22]. Tento

spôsob adresuje problémy, ktoré chceme riešiť a už bol osvedčený v predchádzajúcich riešeniach.

Každý klaster C sa skladá z niekoľkých behov $C = \{b_i | 0 \leq i \leq k\}$ a je reprezentovaný množinou O_C, ktorá nám hovorí, ktoré rôzne udalosti môžu nastať naraz v jednom behu na to, aby beh patril do klastra C. Množina O_C je zjednotením množín všetkých behov patriacich do klastra C.

$$O_C = \bigcup_{k}^{i=0} O_{b_i} \tag{3.1}$$

Každú množinu si môžeme predstavovať aj ako binárny vektor B_C, ktorý obsahuje `1` ak sa udalosť nachádza v množine a `0` ak sa nenachádza. Jednotlivé zložky vektora reprezentujú všetky potencionálne udalosti, ktoré môžu nastať pri behu programu. Vzdialenosť medzi dvoma klastrami, klastrom a behom alebo medzi dvoma behmi sa vypočítava nasledovne:

$$vzdialenost(O_C, O_{b_i}) = \frac{HammingovaVzdialenost(B_C, B_{b_i})}{minimum(|O_C|, |O_{b_i}|)} \tag{3.2}$$

Navyše okrem Hammingovej vzdialenosti sa váhuje aj počtom udalostí v jednotlivých množinách, lebo práve tie ovplyvňujú výslednú podobu klastrov. Klasterizácia navyše zahŕňa aj nasledujúce funkcie, ktoré jej pomáhajú riešiť problémy spomenuté v úvode.

- Odstránenie veľkého počtu malých klastrov sa rieši pridaním penalizácie. Táto penalizácia odzrkadľuje veľkosť klastru a tým sa pri klasterizácií preferujú malé klastre oproti veľkým. Penalizáciu vypočítavame nasledujúcim vzťahom $penalizacia(C_A, C_B) = maximum(log|C_A|, log|C_B|)$. Táto penalizácia sa uplatňuje na vypočítanú hodnotu vzdialenosti, keď sa ním následne prenásobí a takto pozmenená vzdialenosť sa používa ďalej vo výpočtoch.
- Problémom výskytu na rozhraní klastrov (keď $vzdialenosť(O_A, O_b) = vzdialenosť(O_B, O_b)$) môže spôsobovať problémy pre behy, ktoré sú veľmi blízke tomuto behu. Tieto behy by mohli spadnúť do klastra A aj

B a v prípade ak by sme trénovali jednotlivé klastre bez behu b, tak by sme zbytočne generovali falošnú anomáliu. Tieto prípady riešime tým, že beh b použijeme pri trénovaní oboch klastrov (duplikujeme ho).

Klasterizácia behov - detekcia

Pri detekcii kontrolujeme či beh b je súčasťou nejakého klastra. Postupne prechádzame cez všetky klastre a hľadáme taký, ktorý je konzistentný s behom b. To nastáva vtedy, keď všetky udalosti sa mohli spolu udiať v rámci niektorého klastra a to platí práve vtedy keď $C_{b_i} \subseteq C_A$. Ak sa žiadny taký klaster nenájde, tak sa to vyhlási za anomáliu. Zo všetkých klastrov do ktorých patrí beh b si vyberieme taký, ktorý je najbližší k behu b. Vzdialenosť sa vypočíta podľa vzorca (3.2). Takýchto klastrov môže byť viacej a ďalšiu fázu detekcie vykonávame vo všetkých naraz, pričom hľadáme aspoň jeden, ktorý daný beh akceptuje.

Kontrola integrity v klastri

Trénovanie v klastri dáva príležitosť na podrobnejšie analýzy. Predpokladom je že behy sú si trochu podobné a ich zadelením do klastrov budú môcť byť výsledné pravidlá špecifickejšie, bez toho aby sme spôsobili neefektívnosť hľadania anomálií. Ako sme už skôr spomínali, tak v tejto časti budeme používať navyše udalosti o relatívnom usporiadaní prvých/posledných výskytov jednotlivých udalostí. Spomenieme si, aké informácie si vypočítame a tie následne používame na detekciu anomálií.

Množina C_A pre klaster A nám dáva "horné" ohraničenie na veľkosť behu. Pri kontrole integrity vypočítame aj "dolné" ohraničenie pre veľkosť behu. Takto zabránime útokom, ktoré dokážu vynechať pri výpočte niektoré dôležité časti (napríklad kontrolu vstupu/autentifikáciu). Počítanie "dolnej" hranice je veľmi podobné počítaniu "hornej" hranice. Dolnú hranicu pre klaster

C označíme D_C a vypočítame ju nasledujúcim vzťahom:

$$D_C = \bigcap_{k}^{i=0} O_{b_i} \tag{3.3}$$

Ďalšia kontrola integrity, ktorú budeme realizovať v rámci klastrov, sa bude diať pre každú udalosť osobitne. V momente tejto kontroly máme približne rovnaké behy a máme zaručené aj "minimálne" množstvo udalostí v každom behu, ktorý je platný. Pre každú udalosť si vypočítame množinu prerekvizít a postrekvizít. Prerekvizita je udalosť, ktorá sa vykoná vždy pred danou udalosťou a postrekvizita je udalosť, ktorá sa vykoná po poslednom výskyte danej udalosti. Pre vyjadrenie prerekvizít a postrekvizít si zavedieme niekoľko pomocných označení.

`Pred(udalosť u, beh b)` označuje množinu udalostí, ktoré sa udiali pred prvým výskytom udalosti u v behu b. `Po(udalosť u, beh b)` označuje množinu udalostí, ktoré sa udiali po poslednom výskyte udalosti u v behu b. Pripomeňme si, že klaster sme si označili ako $C = \{b_i | 0 \leq i \leq k\}$.

$$Prerekvizita(u) = \bigcap_{k}^{i=0} Pred(u, b_i) \tag{3.4}$$

$$Postrekvizita(u) = \bigcap_{k}^{i=0} Po(u, b_i) \tag{3.5}$$

Udalosti, ktoré sa vyskytujú zriedka nám môžu spôsobovať problémy. Nemáme pre ne dostatok informácie a ak by sme využili dáta iba z malého počtu behov v ktorých nastali tak by sme vyvolávali veľa falošných hlásení, preto udalostiam, ktoré sa nachádzajú iba malý počet krát v klastri, nastavíme postrekvizity a prerekvizity ako prázdne množiny.

Kontrola integrity v klastri - detekcia

Kontrolovať integritu vo vnútri klastra je priamočiare. Na to aby bol beh b normálny, musí splniť všetky kritéria. Medzi kritéria patria:

- $D_C \subseteq b$, táto podmienka skontroluje minimalitu behu - t.j. či obsahuje všetky udalosti ktoré sú v "dolnej" hranici klastra.
- $\forall u \in b : Prerekvizita(u) \subseteq Pred(u, b)$, pre každú udalosť, ktorá v behu nastala musí platiť že pred tým ako prvýkrát nastala udalosť u už nastali všetky udalosti, ktoré sú v množine prerekvizít udalosti u.
- $\forall u \in b : Postrekvizita(u) \subseteq Po(u, b)$, pre každú udalosť, ktorá v behu nastala musí platiť, že potom ako posledný krát nastala udalosť u nastali všetky udalosti, ktoré sú v množine postrekvizít udalosti u.

Ak sú splnené všetky tieto kritéria, tak beh je označený za normálny. Ak nie, tak beh je označený za anomálny.

3.3 Implementácia

V nasledujúcej časti si bližšie popíšeme niektoré detaily implementácie. Najprv si spomenieme spôsob, akým získavame informácie o behu a následne si v krátkosti spomenieme ako sme implementovali trénovanie modelu a detegovanie anomálií. Náš prototyp sme implementovali na Linuxe (Ubuntu 16.04).

3.3.1 Pozorovanie behu programu

Na pozorovanie behu programu používame nástroj **PIN** ([7]).

PIN

Nástroj na inštrumentáciu binárnych súborov. Používa sa na tvorbu nástrojov na dynamickú analýzu programov. PIN za behu programu pridáva inštrumentáciu do skompilovaných binárnych súborov. Poskytuje API, ktoré umožňuje jednoducho inštrumentovať program. Využíva JIT kompiláciu - inštrumentuje program tesne predtým, ako sa ide spustiť.

Nástroje používajúce PIN môžu byť napísané použitím jazyka C, C++ alebo asembleru. Podporuje operačné systémy Windows, Linux, OSX aj Android, pričom funguje na architektúrach IA-32 a Intel64.

Inštrumentovanie v nástroji prebieha pomocou volaní, ktoré sa volajú tesne pred prvým vykonávaním danej časti kódu. PIN môže zavolať funkcie na inštrumentovanie na rôznych úrovniach - sledov, základného bloku alebo jednotlivých inštrukcií. Sled je časť binárneho kódu, ktorá na jednom vstup a potencionálne niekoľko výstupov. Základný blok má jeden vstup a práve jeden výstup.

Naša inštrumentácia funguje na úrovní základných blokov. Ako sme už spomínali, tak chceme zaznamenávať informácie o prechodoch medzi základnými blokmi, pričom každému prechodu chceme priradiť pseudonáhodné číslo (v nasledujúcich častiach ho budeme nazývať ID). Aby sme mohli spájať a porovnávať údaje medzi jednotlivými behmi, musí byť toto generovanie konzistentné medzi behmi. Pomocou nástroja PIN vieme zistiť adresu inštrukcií, zároveň vieme zistiť začiatok virtuálnej adresy knižnice, v ktorej sa nachádza. Pomocou týchto dvoch adries vieme vypočítať adresu pamäti v rámci knižnice, čo zostáva konzistentné medzi dvoma behmi programu. Hashovaním adresy inštrukcie v rámci knižnice spolu s názvom knižnice, v ktorej sa nachádza, vieme vygenerovať pre každý základný blok (prvú inštrukciu v ňom) ID, ktoré je konzistentné medzi behmi.

Pri navštívení základného bloku vypočítame ID hrany, ktorou sme sa dostali do aktuálneho bloku pomocou nasledujúceho vzťahu:

$$ID = (ID\_PREDCHÁDZAJÚCI << 1) \oplus ID\_AKTUÁLNY \quad (3.6)$$

Bitový posun ID_PREDCHÁDZAJÚCI nám zruší symetriu, aby sme vedeli rozlíšiť skoky `A -> B a B -> A`. Program si teda musí pamätať adresu predchádzajúceho bloku.

Dáta o jednotlivých prechodoch medzi základnými blokmi si pamätáme v hashovacej tabuľke o veľkosti mocniny čísla 2. Túto veľkosť prispôsobujeme počtu prechodov, ktoré sú dosiahnuteľné v jednotlivých programoch. Kolízie sa riešia zjednotením dvoch rôznych prechodov do jedného, pričom veľkosť

sa nastaví tak aby ich nevznikalo príliš veľa. Indexovanie do tabuľky je teda jednoduchá operácie xor s maskou $2^n - 1$.

Pre každý prechod si chceme pamätať, či bol navštívený a usporiadanie prvých a posledných výskytov jednotlivých prechodov. Počas behu programu si udržiavame počítadlo počtu vykonaných blokov. Pre každý prechod si zapamätáme hodnotu počítadla pri prvom a posledom výskyte. Tieto hodnoty porovnáme po ukončení programu na výpočet relatívneho usporiadania.

3.3.2 Trénovanie modelu

Trénovanie a testovanie modelu sme implementovali v jazyku C++. Množiny udalostí, ktoré nastali, sme v klastrovacej fáze implementovali pomocou bitového poľa umožňujúce efektívne binárne operácie slúžiace na implementovanie funkcií zjednotenia a prieniku množín.

Počas klasterizácie používame binárnu haldu na udržiavanie vzdialeností jednotlivých klastrov. Keďže sa klastre neustále počas výpočtu menia, menia sa aj vzdialenosti medzi nimi. My sme sa rozhodli neprepočítavať tieto vzdialenosti priebežne, ale pri spojení dvoch klastrov vytvoríme úplne nový klaster. Vypočítame nové vzdialenosti k aktívnym klastrom a tie vložíme do haldy. Ak vyberieme z haldy dvojicu klastrov, z ktorých aspoň jedna už nie je aktívna, tak takúto dvojicu ignorujeme. Časová zložitosť takéhoto riešenia je $O(n^2 * log(n) * M)$, pamäťová náročnosť je O (n * M), kde n je počet trénovacích dát a M je veľkosť hashovacej tabuľky.

Druhá časť výpočtu vo vnútri klastrov je priamočiara, pre každú udalosť si spočítame koľkokrát sa pred ňou/ po nej udiala iná udalosť (počítame vo všetkých behoch daného klastra). Ak je tento počet rovný celkovému počtu behov v danom klastri, tak označíme, že daná udalosť za prerekvizitu resp. postrekvizitu.

Vypočítané informácie o klastroch si ukladáme do textového súboru.

3.3.3 Testovanie behu

Pri zisťovaní či je beh anomálny skontrolujeme, či je beh konzistentný s modelom. Na zistenie či beh patrí do klastra, ktorého udalosti sú uložené v bitovom poli $B_{klaster}$ a z behu máme bitové pole $*B_{beh}$ nám stačí skontrolovať rovnosť: $B_{beh}|B_{klaster} = B_{klaster}$. V rámci behu musíme urobiť kontrolu prerekvizít/postrekvizít pre každú udalosť osobitne. Táto kontrola má časovú zložitosť rádovo $O(M^2)$.

3.4 Testovanie

Na praktické testovanie sme sa rozhodli otestovať niekoľko reálnych bezpečnostných zraniteľností. Na týchto zraniteľnostiach si ukážeme aké chyby dokážeme detegovať a aké má naše riešenie výhody/nevýhody. Všetky sledované programy sme spúšťali na Ubuntu 16.04 s Intel Core i7-4720HQ procesorom a 16 Gib RAM pamäte. Implementácia, ktorú sme realizovali nevyužíva viac jadier procesora. Aby sme si uľahčili trénovanie využívame nástroj *GNU Parallel* ([26]), ktorým dokážeme testovať viacero rôznych modelov naraz. Navyše je trénovanie úplne nezávislé od sledovania programu a môže prebiehať nezávisle. Pri testovaní viacerých alternatív sme taktiež využívali *Google cloud*, kde sme na virtuálnych serveroch s mnohými jadrami testovali a trénovali modely.

3.4.1 Testovanie SSH

V kapitole 1.3, sme spomínali zraniteľnosť v SSH, ktorá viedla k obídeniu autentifikácie. My ju využijeme ako ukážku zraniteľnosti, ktorá dokáže byť identifikovaná pomocou klasterizácie behov. Dôležitý úryvok kódu, ktorý sa nás týka, je zobrazený v listingu č.1. Na riadku 4. sa nachádza zraniteľná verzia funkcie packet_read, ktorá sa môže použiť na prepísanie hodnoty v **authenticated**. Aby sme imitovali útok útočníka sme hodnotu *authenticated* nainicializovali takto: `int authenticated = (strcmp(pw->pw_name, ‘evil’) == 0)`.

Základné bloky, ktoré sa vykonávajú pri inicializácií sme sa rozhodli ignorovať (aby nám nenapomáhali pri detekcii).

```
1  void do_authentication(...) {
2      for(;;) {
3          int authenticated = 0;
4          type = packet_read(...); // zraniteľný
5          switch(type) {
6              ...  // ine metody
7              case SSH_CMSG_AUTH_PASSWORD:
8                  password = packet_get_string(&dlen);
9                  if (auth_password(pw, password)) {
10                     log("Autentifikacia uspesna.");
11                     memset(password, 0, strlen(password));
12                     xfree(password);
13                     authenticated = 1;
14                     break;
15                 }
16                 log("Autentifikacia neuspesna.");
17                 memset(password, 0, strlen(password));
18                 xfree(password);
19                 break;
20             ...
21         }
22         if (authenticated) break;
23     }
24 }
```

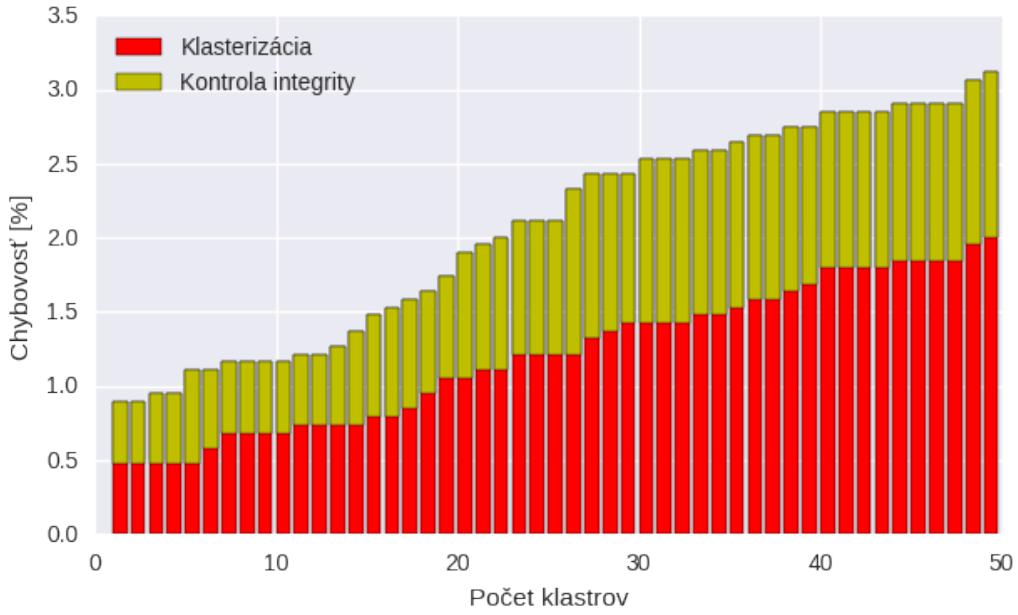
Listing 1: Zraniteľnosť v SSH

Na trénovanie sme vygenerovali náhodných 4000 rôznych pripojení na server. Každé pripojenie na server vyskúšalo niekoľko hesiel. Ak sa podarilo pripojiť tak následne náhodne vykonal niekoľko príkazov a ukončil spojenie. Server sme spúšťali v testovacom móde. V tomto móde príjme program iba jedno spojenie, a následne sa ukončí. My sledovali sme celé jeho vykonávanie.

Tie sme rozdelili do 2 skupín. Jednu skupinu sme využili na natrénovanie modelu a druhú skupinu sme využili na jeho ohodnotenie.

Typický beh obsahoval niekoľko desaťtisíc udalostí, pričom v rámci týchto udalostí obsahoval približne 2000 rôznych blokov a 2500 rôznych prechodov medzi blokmi.

V grafe 3.2 môžeme vidieť ako sa vyvíja počet falošných označení. Na x-ovej osi je znázornený počet klastrov v modely. Na y-ovej osi vidieť percentuálne množstvo falošných označení v testovacej skupine vzoriek.



Obr. 3.2: Vývoj počtu falošných označení

Pri detekcii sme dostali 100% detekciu chyby už pri počte klastrov **1**. Spôsob detekcie sa ale líši od toho, či sa model skladá iba z jedného klastru alebo z viacerých.

Ak sa nachádza v modeli iba jeden klaster, tak takmer všetky (až na malý počet vzoriek) boli detegované až v druhej fáze testovania behu. Dôvod je pomerne zrejmý, a zistili sme ho analýzou kódu v listingu č.1 a výsledkov z nášho nástroja. Problémom je príkaz *break* na riadku 22. Aby sa mohol

vykonať tak v normálnych behoch museli byť predtým vykonané príkazy nachádzajúce sa v základných blokoch pokrývajúce riadky 10-14. To sa ale pri útoku nedeje, a tak je porušená prerekvizita pre príkaz *break*.

Pri viacerých klastroch je útok taktiež detegovaný v druhej fáze. Tentokrát, ale skôr ako kontrola prerekvizít deteguje útok kontrola "dolného" ohraničenia na veľkosť behu. Behy programu SSH sa už pri dvoch klastroch rozdelia na tie pri ktorých sa užívateľ úspešne prihlásil, a tie pri ktorých sa úspešne neprihlásil. Pri útoku sa útočník úspešne prihlási (pomocou exploitovania chyby). Takýto beh sa zaradí do klastra s úspešnými prihláseniami. Dolné ohraničenie v klastri obsahuje základne bloky pokrývajúce riadky 10 - 14 (v listingu č. 1), tieto ale nie sú pokryté v útočníkovom behu a naše riešenie deteguje útok.

3.4.2 Testovanie Linksys

Chyba, ktorú ideme testovať v tejto časti sa nachádzala v routroch spoločnosti Linksys (informácie o chybe [16]). Na zneužitie chyby sa útočilo na časť firmvéru spracúvajúcu HTTP požiadavky. Požiadavky sú najprv spracovavané funkciou *handle_request()*. Táto spracuje HTTP požiadavku a podľa informácií v poli *mime_handlers[]* vyberie správnu funkciu na spracovanie a autentifikáciu. Príklad jedného riadku v *mime_handlers[]*:

```
{ "apply.cgi*", "text/html", no_cache, NULL, do_apply_cgi, do_auth }
```

Tento riadok nastavuje to, aby sa požiadavky na *apply.cgi* spracovávali funkciou *do_apply_cgi()*, a autentifikovali pomocou *do_auth()*. Ak sa niekde nachádza *NULL*, tak sa táto položka ignoruje. Funkcia *do_apply_cgi* obsahuje chybu, vďaka ktorej vie útočník spustiť ľubovoľný príkaz. Na jej exploitáciu útočník nastaví parameter **StartEPI=1**, čím sa spustí funkcia *Start_EPI()*, ktorá prečíta ešte ďalšie parametre napr. *ttcp_ip*, a vykoná príkaz v systéme ktorý obsahuje hodnotu tohto parametra. Napríklad zavolaním s parametrami **StartEPI=1&ttcp_ip=|telnetd**, sa spustí *telnetd*, na ktorý sa môže útočník ďalej prihlásiť.

Táto jedna zraniteľnosť sama o sebe nie je kritická, lebo útočník by sa stále potreboval autentifikovať. Zoznam *mime_handlers[]* obsahuje aj nasledujúce položky:

```
{ "tmUnblock.cgi*", "text/html", no_cache, do_apply_post, do_apply_cgi, NULL }
{ "hndUnblock.cgi*", "text/html", no_cache, do_apply_post, do_apply_cgi, NULL }
{ "tmBlock.cgi**", "text/html", no_cache, NULL, do_apply_cgi, NULL }
{ "hndBlock.cgi**", "text/html", no_cache, NULL, do_apply_cgi, NULL }
```

Tieto položky otvárajú aj neautentifikovanému útočníkovi pristupovať do funkcie *do_apply_cgi*, a exploitovať prvú zraniteľnosť.

My sme implementovali podmnožinu spracovávania HTTP požiadavky (môžete ho nájsť v prílohe práce). Inšpirovali sme sa implementáciou ktorú sme našli v zdrojových súboroch[1] Linksys routra. Tieto neobsahujú druhu z vyššie popísaných chýb, a preto sme ich upravili tak, aby ju obsahovali. Následne sme si vygenerovali 3000 náhodných požiadaviek na server, a náhodne si vybrali 2000 vzoriek ktoré sme použili na vytvorenie normálneho správania routru. Ostatné sme použili na vyhodnotenie vlastností routra. Ďalej sme si vygenerovali 500 požiadaviek exploitujúcich popísané zraniteľnosti.

Náš zjednodušený program obsahuje iba približne 500 rôznych prechodov. Preto sme si zvolili za veľkosť tabuľky do ktorej ukladáme prechody 1024. Chybovosť modelu je 0% pri počte klastrov menšom ako 14. V prípadoch 15 až 50 klastrov to je 0.6%. Nižšiu chybovosť ako pri SSH si vysvetľujeme jednoduchším programom, ktorého vykonávanie obsahuje menej šumu.

Naše riešenie dokázalo detegovať všetky behy, keď sme spracovávali vygenerované škodlivé pakety. Naša detekcia prebiehala zaujímavým spôsobom. Spôsob detekcie môžeme vidieť na obrázku 3.3. Vidíme, že pri menšom počte klastrov, nie je model schopný detegovať zraniteľnosti v prvej fáze, až neskôr, keď sú behy rozdelené dostatočne na to, aby neautentifikované behy spracovávané v funkcii *apply_cgi* boli oddelené od tých, ktoré sú autentifikované.

[1] `http://downloads.linksys.com/downloads/gplcode/WRT54GL-US_v4.30.13.tgz`

Ak sú oddelené, tak neautentifikovaný beh nikdy nie je v klastri s behom, ktorý vošiel do funkcie *Start_EPI*, a model dokáže detegovať útok.

Pri menšom počte klastrov taktiež úspešne detegujeme útok. Nezávisle od toho aké behy sa nachádzajú v jednotlivých klastroch, tak tam platí, že užívateľ musí byť autentifikovaný pomocou *do_auth* už predtým, ako vošiel do funkcie *Start_EPI*.

Obr. 3.3: Spôsob detekcie zraniteľnosti v Linksys

3.5 Zhrnutie a diskusia

V nasledujúcej podkapitole si zhrnieme schopnosti a dosiahnuté výsledky. Popíšeme si detekčnú schopnosť nášho riešenia, a následne sa budeme zaoberať dĺžkou detekcie jedného správania, a vplyvom monitorovania na výkon programu. Ukážeme si tiež vplyv veľkosti tabuľky na detekčnú schopnosť.

Detekcia anomálií, ktoré sme navrhli funguje nezávisle od toho kedy bol program vykonávaný. Kontrola behu sa môže vykonávať offline, a s malými úpravami implementácie vieme zo zaznamenaných dát zistiť zaujímavé informácie o tom, ako prebiehal útok, a kde presne nastala anomália. Tieto dáta nám poskytujú informácie o celom behu programu, čo môže byť zároveň veľkou výhodou, ale aj nevýhodou.

3.5.1 Detekčná schopnosť

Ukázali sme si, že naše riešenie vie detegovať rôzne druhy anomálií. Úspešne sme detegovali útok na SSH, aj na zraniteľnú verziu routra Linksys.

Anomálie, ktoré dokáže naše riešenie detegovať, sú také, pri ktorých beh obsahuje dvojice udalostí, ktoré sa zvyčajne nevyskytujú naraz v jednom behu. Prípadne existujú dvojice behov, ktoré sa vždy vyskytujú pri sebe, ale v anomálnych sa nevyskytujú.

3.5.2 Spomalenie vykonávania programu

Spomalenie programu je momentálne veľmi výrazné, kvôli používaniu nástroj *PIN*. Aj jednoduché nástroje (rátanie počtu vykonaných inštrukcií), ktoré sme vyskúšali, spomaľovali beh programu 10-15x. Náš nástroj spôsobuje spomalenie len o málo viac. Nami namerané spomalenie sa väčšinou pohybovalo okolo 2000%.

Toto spomalenie by mohlo byť značne zredukované pomocou používania iných nástrojov, napríklad WinAfl[2] používa nástroj DynamoRio[3], a sleduje podobné informácie ako nami navrhované riešenie. Ich spomalenie je približne 200%, a tak tu vidíme veľký priestor na zlepšenie.

3.5.3 Čas detekcie

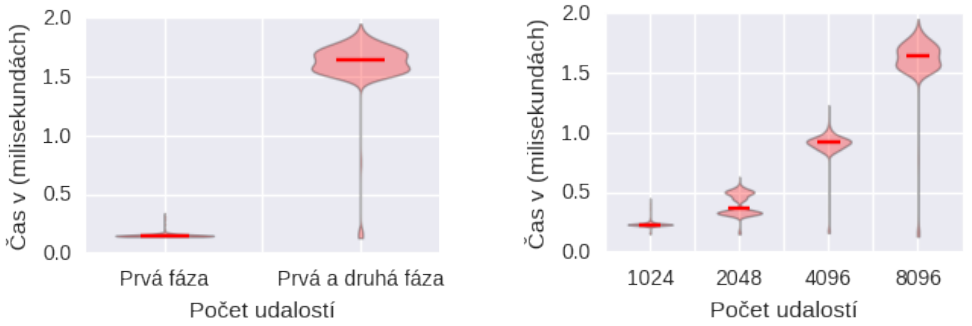
Čas detekcie v našej implementácií značne záleží od toho, či sa beh nachádza v aspoň jednom klastri. Ak sa v žiadnom nenachádza, tak čas detekcie sa skráti. Na obrázku č.3.4a sme si zobrazili časy detekcie pre iba prvú fázu, a pre obe fázy naraz. Udalosti sme v tomto prípade ukladali do poľa o veľkosti 8096. Vidíme, že priemerná dĺžka detekcie je *1.6 ms*.

Dôležitým faktorom pre čas detekcie je aj počet rôznych udalostí, ktoré si ukladáme. Na obrázku č.3.4a môžeme vidieť, že dĺžka detekcie sa zvyšuje približne kvadratickou rýchlosťou. Konkrétne priemerné časy detekcie sú

[2] `https://github.com/ivanfratric/winafl`

[3] `http://dynamorio.org/`

0.23ms, 0.40ms, 0.90ms, 1.60ms postupne pre veľkosti klastrov 1024, 2048, 4096 a 8096.



(a) Čas podľa fázy

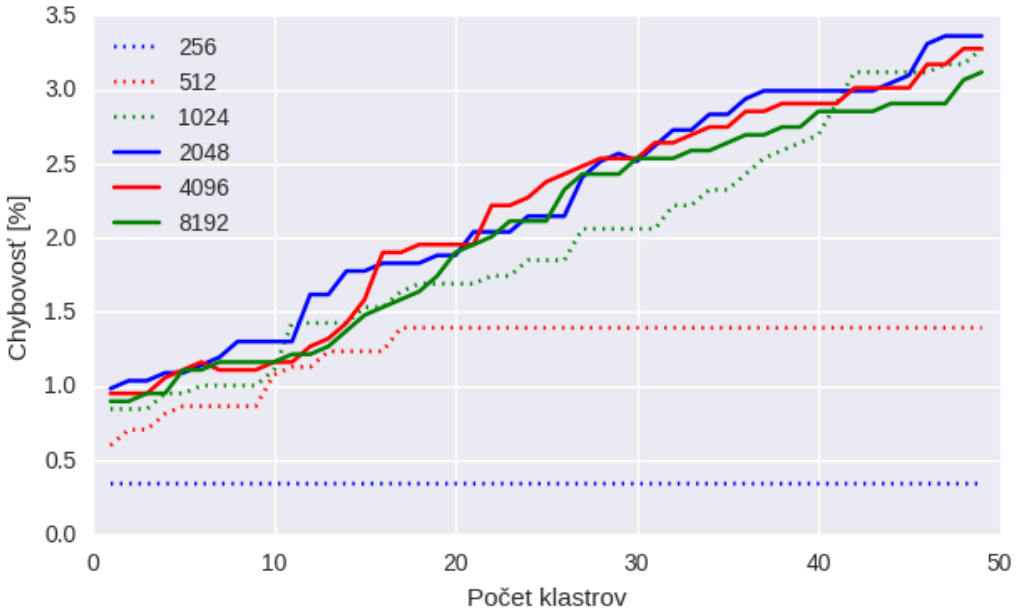
(b) Časy v závislosti od počtu udalostí

Obr. 3.4: Čas detekcie pre jeden beh programu.

3.5.4 Analýza veľkosti tabuľky udalostí

Veľkosť tabuľky v prípade, keď je vhodne zvolená, nemá veľký vplyv na výsledne vlastnosti riešenia. Vybrali sme si dáta zo sledovania programu SSH. Spomínali sme približný počet rôznych prechodov medzi základnými blokmi, ktorý je 2500. Na grafe z obrázku 3.5 môžeme vidieť, že chybovosť sa medzi modelmi s veľkosťami väčšími ako je počet rôznych udalostí veľmi nelíši. Otestovali sme aj detekčnú schopnosť, ktorá zostala zachovaná až ku veľkosti tabuľky *256*, ostatné veľkosti tabuľky detegovali všetky naše vzorky zaznamenávajúce škodlivé behy.

Používanie sledovania na úrovní základných blokov vytvára taktiež limity pri detekcii. Môžeme vidieť na grafe z obrázka 3.5, že chybovosť detekcie sa nedá jednoducho znížiť pod ľubovoľnú potrebnú úroveň. Časť z celkového počtu chýb sa vytvára tým, že beh obsahuje ešte nenavštívenú hranu. Pri veľkostiach klastrov 256, 512, 1024, 2048, 4096, 8192 to je postupne 0.052% ,0.052%, 0.21%, 0.31%, 0.42%, 0.47% behov v testovacej skupine.



Obr. 3.5: Vývoj počtu falošných označení

Záver

Naša práca popisuje spôsoby detekcie v binárnych programoch, ku ktorým z rôznych dôvodov nie je dostupný zdrojový kód. Najprv si všeobecne popisujeme prístupy, ktoré sa používajú, a aké druhy útokov používajú útočníci na vykonanie svojej záškodníckej činnosti. Práca priniesla prehľad metód útokov, a následne sme si rozdelili metódy detekcie do niekoľkých kategórií.V každej kategórií sme popísali niekoľko riešení, ktoré boli implementované a otestované v rôznych vedeckých prácach.

Používanie takýchto riešení môže mať netriviálny dopad na zvýšenie bezpečnosti v komerčnej sfére, ktorá mnohokrát nemá možnosť nahradiť softvér za iný, a preto je záujem o tieto riešenia. Hlavným kritériom je samozrejme schopnosť detegovať útoky, a ďalšími sú spomalenie vykonávania programu, či počet falošných hlásení. Tieto aspekty sme sa snažili brať do úvahy v našom riešení, ktoré sme prakticky vyskúšali a ohodnotili.

Naše riešenie úspešne detegovalo niekoľko druhov útokov s relatívne malým počtom falošných hlásení (čo samozrejme záleží od použitia). Počet falošných hlásení je na úrovní 2%, čo môže byť stále praktické pre niektoré aplikácie. Naša implementácia ukázala potenciál, ktorý má naše riešenie, a navrhli sme jej vylepšenia pre zníženie spomalenia, ktoré je momentálne veľmi vysoké kvôli dynamickému sledovaniu programu programom *PIN*.

Nadväzujúce práce sa môžu vybrať rôznymi smermi. Môžu si vybrať úplne iné druhy spôsobu detekcie, prípadne vylepšiť implementáciu niektorých aktuálnych riešení, za cieľom zníženia spomalenia, alebo počtu falošných hlásení. Priamou nadväznosťou na našu prácu by mohla byť implementácia po-

mocou binárneho prepisovania aplikácie, tak aby sa znížilo spomalenie. Alebo vyskúšať a otestovať sledovanie iných druhov udalostí. Taktiež zmenením časti riešenie, ktorá kontrolovala integritu v rámci klastra môžeme dostať potencionálne riešenie s lepšími vlastnosťami.

Literatúra

[1] American fuzzy lop. [Citované 2016-01-9] Dostupné z http://lcamtuf.coredump.cx/afl/.

[2] Control flow guard. [Citované 2016-01-9] Dostupné z https://msdn.microsoft.com/en-us/library/windows/desktop/mt637065(v=vs.85).aspx.

[3] Dynamic program analysis and software exploitation. [Citované 2016-01-9] Dostupné z http://phrack.org/issues/67/10.html.

[4] Pax address space layout randomization (aslr). [Citované 2016-01-9] Dostupné z https://pax.grsecurity.net/docs/aslr.txt.

[5] Sql injection prevention cheat sheet. [Citované 2016-01-9] Dostupné z https://www.owasp.org/index.php/SQL_Injection_Prevention_Cheat_Sheet#Defense_Option_2:_Stored_Proceduresowaspinjection.

[6] The enhanced mitigation experience toolkit, 2016. [Citované 2017-9-4] Dostupné z https://support.microsoft.com/en-us/kb/2458544.

[7] Pin - a dynamic binary instrumentation tool, 2016. [Citované 2017-9-4] Dostupné z https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool.

[8] Andrea Bittau, Adam Belay, Ali Mashtizadeh, David Mazières, and Dan Boneh. Hacking blind. In *Security and Privacy (SP), 2014 IEEE Symposium on*, pages 227–242. IEEE, 2014.

[9] Tyler Bletsch, Xuxian Jiang, Vince W Freeh, and Zhenkai Liang. Jump-oriented programming: a new class of code-reuse attack. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, pages 30–40. ACM, 2011.

[10] Peter Buchlovsky and Adam Butcher. Buffer overflow vulnerabilities.

[11] Shuo Chen, Jun Xu, Emre Can Sezer, Prachi Gauriar, and Ravishankar K Iyer. Non-control-data attacks are realistic threats. In *Usenix Security*, volume 5, 2005.

[12] Kenneth Fitch Cody Pierce, Matthew Spisak. Capturing 0day exploits with perfectly placed hardware traps, 2016. [Citované 2017-9-4] Dostupné z `https://www.blackhat.com/docs/us-16/materials/us-16-Pierce-Capturing-0days-With-PERFectly-Placed-Hardware-Traps-wp.pdf`.

[13] Benjamin Cox, David Evans, Adrian Filipi, Jonathan Rowanhill, Wei Hu, Jack Davidson, John Knight, Anh Nguyen-Tuong, and Jason Hiser. N-variant systems: A secretless framework for security through diversity. In *Usenix Security*, volume 6, pages 105–120, 2006.

[14] Stephanie Forrest, Steven Hofmeyr, and Anil Somayaji. The evolution of system-call monitoring. In *Computer Security Applications Conference, 2008. ACSAC 2008. Annual*, pages 418–430. IEEE, 2008.

[15] Andrew P Kosoresow and SA Hofmeyer. Intrusion detection via system call traces. *IEEE software*, 14(5):35–42, 1997.

[16] Peter Košinár. Unauthenticated code execution in multiple linksys routers, 2017. Osobná korenšpodencia.

[17] Vishwath Mohan, Per Larsen, Stefan Brunthaler, Kevin W Hamlen, and Michael Franz. Opaque control-flow integrity. In *NDSS*, 2015.

[18] Darren Mutz, William Robertson, Giovanni Vigna, and Richard Kemmerer. Exploiting execution context for the detection of anomalous system calls. In *International Workshop on Recent Advances in Intrusion Detection*, pages 1–20. Springer, 2007.

[19] James Newsome and Dawn Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. 2005.

[20] Mathias Payer, Antonio Barresi, and Thomas R Gross. Lockdown: Dynamic control-flow integrity. *arXiv preprint arXiv:1407.0549*, 2014.

[21] Mathias Payer and Thomas R Gross. Fine-grained user-space security through virtualization. 2011.

[22] Xiaokui Shu, Danfeng Yao, and Naren Ramakrishnan. Unearthing stealthy program attacks buried in extremely long execution paths. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 401–413. ACM, 2015.

[23] Xiaokui Shu, Danfeng Daphne Yao, and Barbara G Ryder. A formal framework for program anomaly detection. In *International Workshop on Recent Advances in Intrusion Detection*, pages 270–292. Springer, 2015.

[24] Kevin Z Snow, Fabian Monrose, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, and Ahmad-Reza Sadeghi. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 574–588. IEEE, 2013.

[25] Adrian Tang, Simha Sethumadhavan, and Salvatore J Stolfo. Unsupervised anomaly-based malware detection using hardware features. In *International Workshop on Recent Advances in Intrusion Detection*, 2014.

[26] O. Tange. Gnu parallel - the command-line power tool. *;login: The USENIX Magazine*, 36(1):42–47, Feb 2011.

[27] David Wagner and R Dean. Intrusion detection via static analysis. In *Security and Privacy, 2001. S&P 2001. Proceedings. 2001 IEEE Symposium on*, pages 156–168. IEEE, 2001.

[28] Yubin Xia, Yutao Liu, Haibo Chen, and Binyu Zang. Cfimon: Detecting violation of control flow integrity using performance counters. 2012.

[29] Kui Xu, Danfeng Daphne Yao, Barbara G Ryder, and Ke Tian. Probabilistic program modeling for high-precision anomaly classification. In *Computer Security Foundations Symposium (CSF), 2015 IEEE 28th*, 2015.

[30] Liwei Yuan, Weichao Xing, Haibo Chen, and Binyu Zang. Security breaches as pmu deviation: detecting and identifying security attacks using performance counters. In *Proceedings of the Second Asia-Pacific Workshop on Systems*, page 6. ACM, 2011.