VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

NÁSTROJ NA GENEROVÁNÍ POLYMORFNÍCH SÍŤOVÝCH ÚTOKŮ

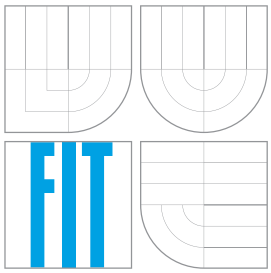
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

DAVID BUCHTA

BRNO 2015

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

NÁSTROJ NA GENEROVÁNÍ POLYMORFNÍCH SÍŤOVÝCH ÚTOKŮ

TOOL FOR GENERATING POLYMORPHIC NETWORK ATTACKS

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

DAVID BUCHTA

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. DANIEL OVŠONKA

BRNO 2015

Abstrakt

Tato bakalářská práce se zabývá návrhem a tvorbou desktopové GUI aplikace pro generování polymorfních síťových útoků. Vytvořená aplikace implementuje několik technik pro vyhýbání se detekci. Dále umožňuje uživatelovi definovat vlastní techniky. Aplikace pak ve velkém množství zasílá útoky na cíle a zkoumá jejich funkčnost a vyhnutí se detekci pomocí některého NIDS.

Abstract

This bachelor thesis presents design and implementation of desktop GUI application for generating polymorphic network attacks. Created application implements several evasion techniques. This application also allow user to create custom techniques and use it in application. Application sends large amount of attacks in purpose to find successful NIDS evasion.

Klíčová slova

polymorfní síťové útoky, vyhnutí se detekci, NIDS, IDS, C#, obfluskace, GUI aplikace, kodování NOP, obfluskace řetězců, přetečení bufferu

Keywords

polymorphic network attacks, detection evasion, NIDS, IDS, C#, obfuscation, GUI application, NOP encoding, string obfuscation, buffer overflow

Citace

David Buchta: Nástroj na generování polymorfních
síťových útoků, bakalářská práce, Brno, FIT VUT v Brně, 2015

Nástroj na generování polymorfních síťových útoků

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Daniela Ovšonky.

.....
David Buchta
3. května 2015

Poděkování

Děkuji vedoucímu mojí bakalářské práce za odbornou pomoc při její tvorbě.

Obsah

Kapitola 1

Úvod

Dnešní svět je zcela závislý na informačních technologiích. Počítač ke své práci denně používají miliony lidí. Počítače nebo jejich formy v podobě mikropočítačů můžeme najít prakticky v každém zařízení, jako např. lednička nebo auto, ale i v moha dalších. Kromě pracovních využití počítačů existuje nemalá skupina lidí využívající je k zábavě.

Ruku v ruce s rozšiřujícím se světem počítačů a jejich propojení přes internet jdou i vzrůstající požadavky na zabezpečení. Mnohé se změnilo od dob prvotních počítačových nadšenců, kterým se ani nesnilo o objednávání jídla přes internet. I na tomto jednoduchém případu můžeme vidět potřebu v zabezpečení ohledně přenášených údajů kreditní karty. V nechráněném chybovém prostředí může útočník odchytit od dat zadávaných uživatelem na klávesnici, až po pakety proudící v internetu obsahující tyto údaje. Kromě kradení dat se dají počítače zneužít k sestavení velkých farem například k těžení kryptoměn nebo jiných výpočtů různého typu.

Na ochranu před tímto přispěchaly mnohé firmy s programy pro zabezpečení počítačů. Jednou skupinou těchto programů jsou systémy detekce průniku neboli IDS. Tyto systémy dokážou uživatele, při komunikaci s nezabezpečeným a nebezpečným světem internetu, informovat o útocích a při spojení s prevenčními systémy zvládnou i ochránit před úspěchem takových útoků. Avšak aby mohl IDS správně fungovat, musí se naučit rozpoznat hrozby.

Cílem této práce je prozkoumat současné techniky obcházení systémů a následně pár vybraných implementovat a otestovat proti dnešním IDS. K tomu, aby šlo techniky testovat, bude zapotřebí najít chyby v softwarech, třeba i zdokumentované a dávno opravené, a implementovat tyto útoky. Avšak ani tohle nestačí, další nedílnou součástí bude vytvoření aplikace, která dokáže upravit útok určitou mutační technikou a zaslat ho na cílový stroj.

Jelikož se hrozby neustále vyvíjí a vyvíjí se i samotné systémy průniku detekce, tak by aplikace měla umožnit uživateli definovat vlastní nové útoky a techniky obcházející detekci. Cílem tedy bude vytvořit uživatelsky přívětivou aplikaci, která už v základu umožní otestovat současné systémy, ale zároveň si ji uživatel bude moci a chtít rozšířit a používat ji k testování IDS, testování zabezpečení sítě, popřípadě i k samotnému učení IDS novým hrozbám. Tohoto bude dosáhnuto i možností generování velkého množství útoků s minimální uživatelskou interakcí.

Kapitola 2

Instruction detection systems

Intrusion detection system je obranný prostředek, sbírající informace o určitém informačním systému nebo síti, které následně podrobuje analýze. Cílem této činnosti je odhalit prolomení bezpečnosti, pokusy o prolomení bezpečnosti, případně nalézt zranitelnosti, které by mohly vést k narušení bezpečnosti daného systému nebo sítě. IDS si můžeme představit jako bezpečnostní kameru v bance. [13][1]

V obecné rovině lze IDS popsat jako detektor, zpracovávající informace přicházející ze systému, který má chránit. Jeho úlohou může být eliminovat nepotřebné informace ze záznamů pro audit. Ten pak představuje souhrnný pohled na akce související s bezpečností zaznamenané během normálního používání nebo souhrnný pohled na aktuální stav bezpečnosti systému. K tomuto využívá trojici druhů informací. Dlouhodobé informace související s detekční technikou (např. znalostní bázi útoků), informace o aktuálním stavu konfigurace systému a auditní záznamy popisující právě probíhající události v systému.[13]

Úkolem IDS tedy je na základě jednotlivých pohledů určit pravděpodobnost, zdali právě probíhající akce nebo aktuální stav stroje má být považován jako příznak průniku, nebo nalezené zranitelnosti. Jestliže IDS vyhodnotí tuto skutečnost jako pravdivou, informuje o ní předem určeným způsobem, např. IPS systém.

Instrusion prevention system (IPS) je software, který na základě informací od senzorů realizuje opatření proti právě nastalému stavu např. zablokování provádění aktuálního kódu, nebo návrat do předchozího bezpečného nastavění. Často se IDS a IPS kombinuje do jediného systému IDPS (intrusion detection and prevention system). V praxi se však můžeme setkat, že jediným pojmem IDS je označován zároveň IDS a IPS.

2.1 Typy IDS systémů

Tak jak se v posledních třiceti letech vyvinuly techniky pro detekci průniků, vzniklo více způsobů realizace vlastního rozpoznání. IDS můžeme rozdělit dle tří konceptů, jejichž popisu se budu věnovat v následujících podkapitolách. Ačkoliv existuje mnoho prací, které do hloubky definují a popisují jednotlivé techniky a způsoby, pro účely této práce bude stačit toto hrubé dělení a popis.

2.2 Dělení dle detekční metody

Detekční metoda určuje způsob, který detektor použije pro kvalifikování dané situace. V technice detekce průniků existují dva, doplňující se trendy pro detekci.

2.2.1 Metoda založena na znalostní bázi

Metoda založena na znalostní bázi, často označována taky jako detekce zneužití, nebo detekce na základě chování, využívá pro detekci vědomosti získané o specifickém útoku. Systémy detekce průniků obsahují databázi otisků těchto útoků, nebo zranitelností a kdykoliv je nalezena shoda mezi aktuálně porovnávanými daty a záznamem v databázi dojde k vyvolání alarmu. Všechny akce, pro které není nalezen odpovídající otisk v databázi, jsou považovány za bezpečné. Přesto jsou bázově založené metody označovány jako velice přesné. Nicméně jejich přesnost je ovlivněna rozsahem a četností aktualizací databáze útoku. Samotné otisky pak mohou být vytvářeny dvěma způsoby - za pomoci expertních systémů, nebo analýzy signatury.

Expertní systémy obsahují sadu pravidel, které obsahují množinu pravidel popisující útok. Události auditu jsou následně přeloženy na fakta obsahující jejich sémantický význam v expertních systémech. Nakonec jsou tato fakta a pravidla zpracovány dedukčním členem. Expertní systémy zvyšují úroveň abstrakce auditních záznamů, jelikož připojují sémantické data.

Analýza signatury pro získávání informací sleduje stejný princip jako expertní systémy. Sémantická informace z útoku je přetransformována na informaci, která může být přímo nalezena v auditních záznamech, kupříkladu útok může být přeložen na sekvenci generovaných auditních událostí. Tato technika umožňuje velice výkonově účinnou transformaci, čehož se využívá v komerčních systémech.[4]

Otisky mohou nabývat více forem. Nejjednodušším otiskem může být URL zaslána na webový server, kde určité řetězce, jako např. "cmd.exe"nebo pokus o telnet připojení s uživatelským jménem "root", jsou příznaky útoku. Povaha komunikace může taky sloužit jako signatura, např. délka sezení nebo množství přenesených dat.[5]

Velikou výhodou detekce založené na znalostní bázi je teoreticky malá míra falešně vyvolaných poplachů. Dále pak kontextová analýza problému je velmi detailní, čímž zjednodušuje pochopení nastalé situace. Na druhou stranu problémem může být shromažďování informací a vytváření z nich databáze pro detekci. Neustále je třeba hledat nové zranitelnosti, následně je analyzovat a aktualizovat databáze. K těmto účelům lze využít buď různé způsoby vyhnutí se detekci a každý nezaznamenaný útok přidat k již známým. Dalším způsobem je pak vytváření tzv. honey-potů což jsou systémy tvářící se jako regulérní systémy s citlivými nebo jinak pro útočníka zajímavými informacemi, které monitorují útočníkovo chování při pokusu získaní těchto domněle cenných informací.

2.2.2 Metoda detekce na základě chování

Behaviorálně založené metody používají pro detekci útoku profily. Tyto systémy předpokládají, že každý pokus o průnik může být zjistitelný za pomocí pozorování odchylek od běžného uživatelova nebo systémového chování. Profily chování se vytvářejí z různým způsobem posbíraných informací. Následně se porovnávají s aktuálně odchycenými informacemi, a pokud není nalezen profil chování, dojde k vygenerování poplachu.

Z tohoto chování vyplývá veliká bezpečnost, jestliže pro dané chování neexistuje profil, není povolené. Další výhodu, kterou zde můžeme vidět oproti metodě založené na znalostní bázi je, že tyto metody jsou odolné i vůči novým ještě neznámým útokům. Na druhou stranu zde dochází k velmi častému vyvolání falešných poplachů i z důvodu drobných změn chování, jež by mělo být povolené a taky ne všechno povolené chování bylo obsáhnuto v tréninkové fázi modelu. Proto je třeba provádět retrénování modelu.

Při vytváření jednotlivých profilů chování ještě hrozí bezpečnostní riziko, způsobené trénováním modelu na již napadeném stroji a tak zanesení tohoto chování do databáze, což způsobí imunnost takového útoku vůči detekci.

Nejrozšířenějším způsobem vytváření profilu je používání statistik. Při tomto způsobu jsou monitorovány různé veličiny používání systému, jako jsou čas přihlášení a odhlášení, množství spotřebovaného procesorového času, množství používané paměti, disku, objem přenesených dat po sítí, používané protokoly a další. Tyto údaje se sbírají po předem stanovený čas (od jednotek minut až po měsíce), následně jsou zpracovány průměrné hodnoty, ze kterých se vytvoří detekční prahy. Detekční algoritmus pak na základě standartní odchylky porovnává, jestli nedošlo k překročení daného prahu. Ve skutečnosti je však tento model pro věrné zobrazení dat příliš jednoduchý. Tudíž byly vytvořeny mnohem komplexnější modely, které zohledňují jak krátkodobou, tak dlouhodobou uživatelskou aktivitu.

I pro behaviorálně založené metody lze použít expertních systémů. Tento přístup je použitelnější pro profily založené na právech, avšak je méně účinný při zpracování velkého množství auditních záznamů než statistický přístup.

Dále lze pro vytváření modelů použít neuronové sítě. Neuronová síť je algoritmus, který se učí vztahům mezi trénovanými množinami, tyto vztahy pak zobecňuje, aby byl schopen vytvářet nové smysluplné páry. Teoreticky by se neuronové sítě daly použít i při metodách na základě vědomostních bází, kdy nejprve identifikují útok a ten pak hledají v auditních záznamech. Při vytváření modelů chování se používají k učení chování běžných uživatelů. Výhoda oproti statistickým metodám leží v jednodušším vyjádření nelineárně souvisejících informací a v automatickém retrénování modelu.

Existují i tendence nevytvářet modely na základě chování uživatele, ale na základě chování systému jako jsou sekvence systémových volání procesů.

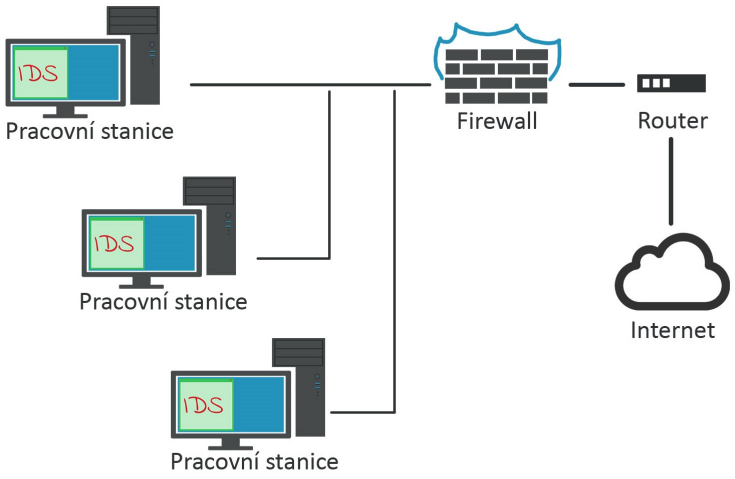
Samotné vzory mohou být i protokolově založeny. Zde se detekují fyzické rozdíly mezi daty jednotlivých paketů a protokolu odpovídajícímu RFC, např. pole hlaviček mohlo překročit RFC určenou maximální velikost.

Tato metoda detekce průniku nabývá nejlepších výsledků, a proto nalézá nejlepší uplatnění v předvídatelném prostředí s neměnící se aktivitou.[4][5][1]

2.3 Dělení dle umístění IDS

Dle oblasti působení můžeme rozdělit IDS do dvou kategorií. Z historického hlediska při prvních návrzích IDS fungovaly počítače jako mainframy, ke kterým přistupovali uživatelé, kteří z pohledu systému vypadali jako lokální, a tak byl kladen důraz na rozvoj uzlově orientovaných systémů detekce průniku. Až s pozdějším rozvojem výpočetní techniky a komunikačních sítí došlo k vytvoření pracovních uživatelských stanic, které komunikovaly nejen mezi sebou, ale i díky většímu rozvoji internetu komunikovaly tyto stanice s vnějším světem. To vedlo k zaměření vývoje detekčních systému i na univerzální jednotky, které ochraňují celou síť, a tak vznikla druhá kategorie působnosti - síťově orientované systémy detekce průniku. Bližšímu popisu principu a vlastnímu fungování budu věnovat následující dvě podkapitoly.

2.3.1 Host-based Intrusion Detection system (HIDS)



Obrázek 2.1: Pracovní síť s HIDS

Uzlově orientované detekční systémy byly sice vyvíjeny v době sálových počítačů, ale jejich uplatnění lze nalézt dodnes. HIDS jsou zaměřeny na sbírání a analýzu aktivity na jednom konkrétním počítači viz ilustrace 2.1. Jejich použití se nabízí typicky v situaci, kdy víme o jednom konkrétním stroji, že je náchylný k útokům. Tyto systémy přímo monitorují a přistupují k cílům útoků, které obvykle tvoří datové soubory nebo systémové procesy.

Díky svému umístění - přímo na hostitelském stroji, mohou získávat přesnější a spolehlivější informace o útocích. Dokážou tedy přímo odhalit uživatele a proces, jenž se na útoku podílel. HIDS umožňují monitorovat lokální události a tak uspět při rozpoznání útoků, které by za pomoci NIDS byly nerozpoznatelné, např. zvládnou odhalit trojské koně, které se projeví jen jako nekonzistence otisku aktuálního vykonávaní programu s běžným chováním. Dále dokážou fungovat i v šifrovaném prostředí, jelikož přistupují k datům před nebo ihned po dešifraci a také nejsou ovlivňovány přepínatelnými sítěmi.

K práci se síťovým tokem se zde přistupuje na vyšších úrovních abstrakce, což dovoluje kontextovou práci s tímto tokem a přiřazování ho ke konkrétním relacím. Na druhou stranu přicházíme o znalosti událostí probíhajících na síti. Další nevýhoda vyplývá ze samotného umístění. Jelikož jsou tyto systémy nainstalovány na pracovních stanicích, tak ubírají těmto stanicím určitý výpočetní výkon a taky jsme omezeni platformou dané stanice.

Množství těchto systémů ovlivňuje následnou správu, která musí být prováděna na každém stroji zvlášť, proto jsou některé HIDS navrhovány s podporou jednoho centrálního IDS a infrastrukturou pro zasílání zpráv. Tento přístup umožňuje centrální správu HIDS za pomoci jediné konzole. Můžeme se taky setkat se systémy generujícími zprávy v kompatibilním formátu se síťově zpravovanými systémy.[2][4][10]

Příklady HIDS:

- Dragon Squire
- Emerald eXpert-BSM
- NFR HID
- Intruder Alert

HIDS obvykle pracují na systémové úrovni a tak k monitorování stroje využívají záznamy auditu operačního systému, ten obvykle bývá generován jádrem, je proto velmi detailní a dobře zabezpečen. Dále se využívají systémové záznamy, které jsou univerzálnější, menší a jednodušší na pochopení. Konkrétně se pak jedná o C2 security audit a Syslog.[4]

C2 security audit C2 označuje bezpečnostní úroveň přístupu k daným záznamům. Potřebná zabezpečení takovýchto záznamů byla definovány americkým ministerstvem obrany (DoD). K nejdůležitějším požadavkům pro splnění úrovně C2 patří možnost udělit či odejmout zdroje počítače individuálnímu uživateli nebo pojmenovaným skupinám, paměť musí být chráněna proti neoprávněnému čtení, jestliže ji proces už uvolnil, uživatelé se musí systému identifikovat unikátním způsobem např. heslem a mnohé další. Veškeré bezpečnostní záznamy této úrovně využívají základního principu zaznamenávání křížení instrukcí prováděných v uživatelském prostoru a instrukcí vykonávaných v TCB prostoru.[4]

Syslog je auditní nástroj poskytovaný operačním nástrojem pro standardizování zpráv programů. Všechny záznamy obsahují datum, čas, úroveň a vlastní zprávu záznamů. Ke komunikaci se používá obyčejný textový řetězec, jehož formát je dán dle RFC 3164, tyto řetězce jsou za pomoci UDP protokolu zasílány na port 514 [7].

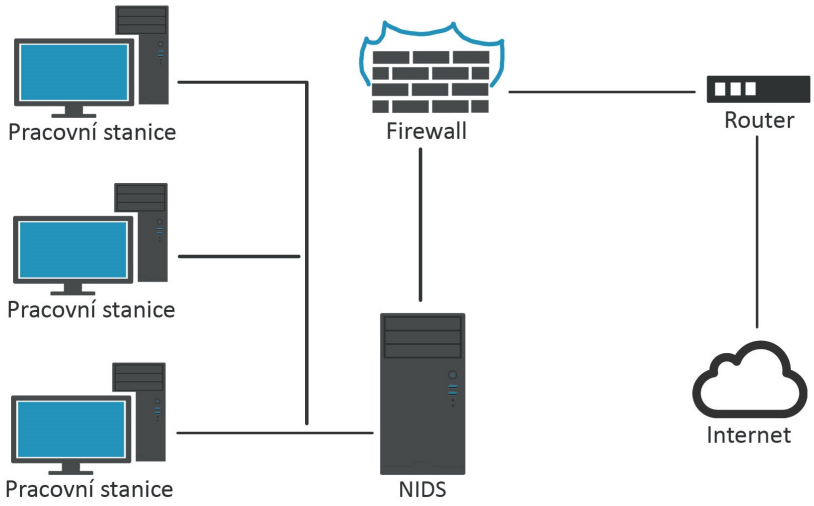
Zprávy na Syslog server pak mohou vypadat takto:

```
<134>Nov 17 14:27:15 127.0.0.1 ircbot <nick>: hola#015
<134>Nov 17 14:38:50 127.0.0.1 ircbot <Aveline>: :-)#015
```

Příklad Syslog záznamů:

```
Nov 17 14:27:15 localhost nick>: hola#015
Nov 17 14:38:50 localhost Aveline>: :-)#015
Nov 17 14:39:25 isa2014 dhclient: DHCPREQUEST of 192.168.10.15 on eth0
to 192.168.10.1 port 67 (xid=0x373bac6f)
Nov 17 14:39:25 isa2014 dhclient: DHCPACK of 192.168.10.15
from 192.168.10.1
Nov 17 14:39:25 isa2014 dhclient: bound to 192.168.10.15 -- renewal in
209 seconds.
Nov 17 14:42:54 isa2014 dhclient: DHCPREQUEST of 192.168.10.15 on eth0
to 192.168.10.1 port 67 (xid=0x373bac6f)
```

2.3.2 Network Intrusion Detection system (NIDS)



Obrázek 2.2: Pracovní síť s NIDS

Jak už bylo zmíněno výše, druhým typem jsou sítově orientované systémy detekce průniků. Na tomto principu funguje většina komerčních systémů. NIDS jsou buď přímo speciální hardwarová zařízení, nebo běžné stanice, které splní určité kvalitativní požadavky a mohou být k tomu účelu vyhrazeny.

Svou funkci systémy plní sledováním provozu v dané síti. Jak může být patrné z ilustrace 2.2 lze za pomoci jediného zařízení zabezpečit více stanic. Ale z toho taky vyplývá větší zatížení těchto zařízení, jelikož musí zvládat zpracovávat vysoký tok dat. Na druhou stranu však nasazení NIDS do existující sítě ji nijak neovlivní, neboť NIDS jsou většinou pasivní systémy, které komunikaci na síti pouze naslouchají, a pár dobře umístěných senzorů dokáže monitorovat rozsáhlé sítě.

Pro analýzu sítě musí systémy zachytávat všechny pakety procházející sítí, proto jejich síťové karty pracují v promiskuitním režimu, který dovoluje přijímat i data, která nemají hardwarovou adresu dané síťové karty. Z toho můžeme vidět hned několik problémů. Prvním problémem jsou přepínané sítě, kdy každé spojení funguje jako bod-bod spojení mezi přepínačem a stanicí. V tomto okamžiku by byly pro kontrolu k dispozici pouze broadcastova data, která jsou zasílaná všem stanicím. Proto je třeba využívat přepínačů s možností zrcadlení portu, jenž nám zajistí přeposílání veškerého toku do senzoru, nebo musí veškerý tok probíhat přes senzor. Dalším problémem jsou samotná data. V dnešní době se velice hojně používá šifrování pro komunikaci přes internet. Toto šifrování se provádí na stroji odesílajícím a přijímacím požadavek, tudíž veškerá taková zachycená data není možno podrobit analýze, na rozdíl od dříve zmíněného uzlového přístupu. Dále zde máme fragmentaci dat, ne všechny systémy používají (někdy to není ani možné) sestavování TCP toku. Jelikož NIDS pouze naslouchají, může u nich útočník použít např. (D)DoS útoků pro zahlcení, čímž se následný tok stává neanalyzovaným, přesto považovaným za bezpečný.

Samotné senzory jsou limitovány na běh IDS, to nám dovoluje je mnohem lépe zabezpečit proti útokům. Navíc velké množství senzoru je navrhováno pro běh v "nenápadném"režimu, díky čemuž se stávají mnohem hůře zjistitelné pro potencionálního útočníka. [4][2]

Příklady NIDS:

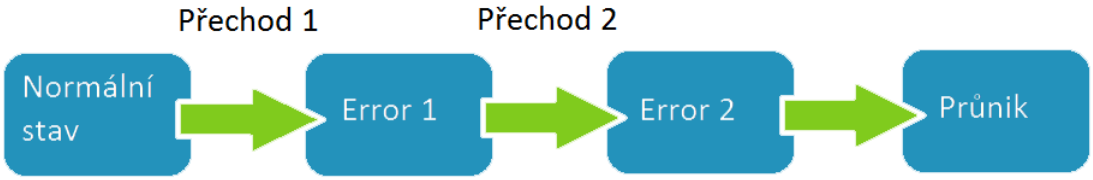
- Snort
- Cisco Secure IDS (dříve NetRanger)
- Hogwash
- Dragon
- E-Trust IDS

Kromě zachycených paketů se pro analýzu sítě používají ještě informace získané pomocí SNMP protokolu. Simple Network Management Protocol popsaný v RFC 1157 nám umožňuje např. získat směrovací tabulky, adresy, statistické údaje o pohybu dat v síti a mnoho dalšího.

K detekci se potom používá kombinace detekce na základě signatur, detekce na bázi anomálií a pro běžné protokoly se využívá jejich stavová analýza. Některé systémy dokážou při rozpoznání útoku v TCP komunikaci zajistit ukončení takové komunikace zasláním příslušných příkazů oběma stranám figurujícím v komunikaci.

2.4 Dělení dle detekčního paradigmatu

Ve světě detekce průniků se používají dva paradigmata pro detekci. První z nich spočívá ve stavech, ve kterých se systém nachází, druhé pak na přechodech mezi jednotlivými stavy. Pro bližší vysvětlení si pomohu obrázkem 2.3.



Obrázek 2.3: Stavy systému

Stavy můžeme definovat následovně. Normální stav odpovídá situaci, kdy je systém zcela v pořádku bez žádných zranitelností. Avšak i při rozpoznání normálního stavu, nemáme žádnou informaci o umístění toho stavu vůči stavu selhání. Stav Error 1 je situace, ve které byla na ochraňovaném počítači nalezena aplikace obsahující určitou zranitelnost. Stav Error 2 by mohlo být zneužití takové chyby útočníkem a získaní např. souboru s hesly. Nakonec stav průniků nastane, když útočník použije získané heslo administrátorského účtu pro přihlášení se k danému stroji.

Jelikož, jak bylo zmíněno výše, z rozpoznání normálního stavu nedokáže systém určit, kdy nastane problém, využívají stavově založené systémy chybových stavů pro určení cesty, případně vzdálenosti k cíli označujícím průnik. Chybový stav nastane, pokud je v systému přítomná zranitelnost nebo chyba konfigurace. Pro zjištění se používá pravidelné dotazování sytému na verze programu nebo konfiguraci, a pokud výsledek odpovídá záznamu v databázi, pak IDS přejde do příslušného chybového stavu. Některé systémy pro detekci průniku

kontrolují i signatury důležitých souborů.

IDS používající paradigma přechodů mezi stavy čekají na specifické události např. útok nebo podvodný paket. Tyto události jsou známy tím, že způsobují přechod mezi jednotlivými stavy v sytému.[4]

2.5 Použité NIDS

Jelikož program vytvořený v této práci se bude testovat proti NIDS Snort[3] a Suricata[14], chtěl bych se nyní zmínit o jejich vlastnostech. Jako první začnu programem Snort . Snort je open source, multiplatformní, jednovláknový IDS používající techniku detekce založenou na znalostní bázi. Snort při provozu odchytává veškerý síťový provoz a na základě porovnání s pravidly v databázi rozhodne, zda se jedná o legitimní nebo podvodný síťový tok. První verze byla vydaná již v roce 1998 a tak má k dispozici širokou uživatelskou základnu podílející se na vytváření pravidel.

Naproti tomu Suricata je poněkud mladší, konkrétněji od roku 2009. Suricata je taky open source, multiplatformní IDS, ale na rozdíl od Snortu je vícevláknová, díky tomu je rychlejší a umožňuje odhalovat útoky mnohem náročnější na výpočetní výkon. K detekci se opět využívá znalostní báze.

2.6 Doplňující informace

Uvedené rozdělení systému pro detekci průniku je do určité míry pouze formální, v praxi se můžeme setkat s tím, že se mnohá řešení navzájem překrývají. I samotné dělení je do určité míry ovlivněné autorem, proto lze nalézt mnoho rozdílných způsobů dělení.

I zdánlivě odlišné řešení za pomoci specializovaného hardwaru, nebo za čistě softwarového řešení, se může překrývat v okamžiku, kdy použijeme specializovanou stanici pouze za účelem detekce průniku. V tomto okamžiku jsme limitováni pouze možnostmi hostitelského operačního systému.

Samotnou kapitolou jsou rozsáhlá řešení za pomoci kombinace nejrůznějších zařízení a aplikací pod jednotnou správou určené k zabezpečení dané sítě. Tento druh IDS se zcela vymyká jakékoliv kategorizaci.

Kapitola 3

Techniky vyhýbání se detekci

Jako již od nepaměti existuje souboj dobra a zla, tak v počítačovém světě můžeme vidět jeho obdobu mezi útočníky a obránci. Jako útočníci mohou vystupovat různí jedinci nebo skupiny a to buď s cílem získat cenné informace, nebo si jen dokázat sobě, případně okolí, své schopnosti. Avšak útoky často provádějí samotní bezpečnostní experti za účelem hledaní chyb v aktuální síti, kterých by mohl útočník zneužít. Ale ať už se jedná o "padouchy", nebo bezpečnostní techniky, všichni se snaží najít takový útok, který unikne zájmu NIDS.

Jako nejjednodušší postup pro nalezení takového útoku by se mohlo jevit prozkoumání databáze vzorů, jednotlivých detekčních systémů, identifikovat slepá místa a ty využít pro útok. Bohužel málo komerčních systému poskytuje přístup ke svým databázím, a i když tyto databáze získáme, hledání jejich slepých míst je značně zdlouhavé a neefektivní což degraduje celou metodu.[16]

Mnohem lepší postup je zvolit jednoduchou úpravu podpisu útoku pro vyhnutí se detekci. Úpravy můžeme provádět na více vrstvách, a pro každou vrstvu existuje bezpočet technik, v následujících podkapitolách bych proto rád vyjmenoval a popsal princip funkce těch nejznámějších a nejpoužívanějších technik. Jak už nám mohl ukázat přehled uvedený v předchozí kapitole, záběr IDS je velmi široký, proto se ve zbytku této práce zaměřím jen na síťové systémy detekce průniku, pro něž bude výsledná aplikace testována.

3.1 Zahlcení

V počítačovém světě se často můžeme setkat se dvěma pojmy fail-open a fail-closed. Pojmy vysvětlím na výtahu. Každý klasický výtah je konstruovaný jako fail-closed, tohle pocítíme v situaci, když se přetrhnou nosná lana. Při tomto selhání okamžitě zareagují brzdy na výtahu, výtah se zastaví a nespadne. V případě fail-open konstrukce výtah nemá brzdy a při přetržení lan závisí stání výtahu pouze na tom, jestli se nalézal v dolní poloze. Stejná situace nastává v počítačovém světě v případě použití fail-open ochrany - bezpečnost je při nehodě garantována pouze pokud nikdo na daný systém neútočí.

Ale vraťme se k systémům detekce průniku. Často se lze setkat se sítově orientovanými detektory pracujícími v pasivním režimu, o kterých jsem se již zmínil v předchozí kapitole. Jen připomenu, že tyto systémy pracují za pomoci zrcadlení komunikace přepínačem. Z principu tyto detektory fungují jako fail-open. Tedy útočníkovým primárním cílem je zahltit detektor legitimními požadavky až do úrovně, kdy dojde k vyhladovění - detektor nebude stíhat kontrolovat všechna data, až k případnému selhání a pádu.

Pro generování útoku zahlcením lze využít chyby v aplikacích. Dále lze pak s využitím

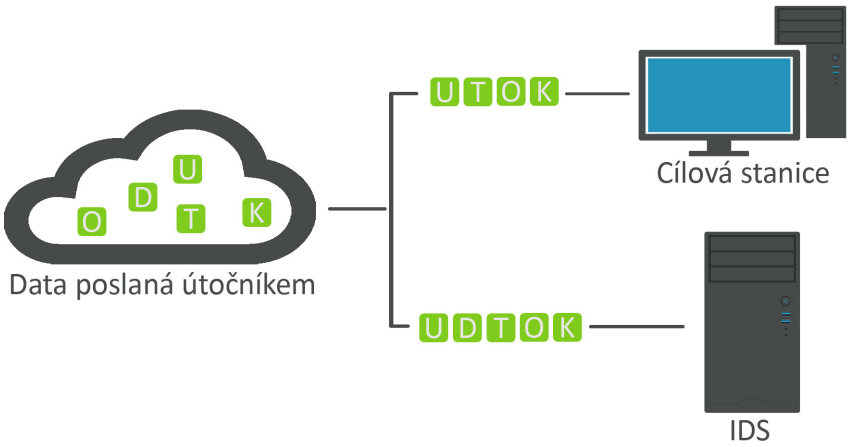
distribuované sítě tzv. DDoS útoku použit techniku zahlcení běžnými daty např. příkazem PING. Při úspěchu získává útočník volný vstup do nechráněné sítě. Bohužel obrana proti DoS útokům je velice složitá.[8]

3.2 Modifikace paketového proudu

Většina technik má společnou vlastnost - snaží se chytře obejít detekční systém a ponechat ho v přesvědčení, že se nic neděje. K tomuto lze využít úpravu toku odesílaného v rámci útoku. Tato technika je poněkud složitější, jelikož je zde potřeba pracovat na nízké úrovni síťové komunikace. Avšak na rozdíl od spoofing útoků je lehčí v tom, že upravovaný datový tok generuje samotný útočník.

3.2.1 Insertion

Pro techniku insertion, která prokládá proud paketů obsahující sekvenci dat rozpoznatelnou detektorem pakety narušující tato data, se předpokládá situace nepřijetí všech paketů koncovou stanicí. Samozřejmě IDS přijme celou sekvenci, tu porovná s databází, kde díky vloženým šumovým paketům nenastane shoda s nějakým otiskem, a jelikož předpokládá od koncové stanice přijetí stejného toku, jako přijal on, tak nevyvolá poplach. Útočník tedy vkládá data do IDS, avšak žádný jiný systém na síti nezpracovává tyto špatné pakety.[8]



Obrázek 3.1: Insertion

Pro lepší představu zde uvedu obrázek 3.1. Útočník chce použít útok, jenž nám symbolizuje řetězec UTOK. Tenhle řetězec má samozřejmě IDS uloženo jako škodlivý vzor, tudíž útočník musí vložit do proudu paket s daty D. Výsledný řetězec odeslaný do cílové sítě tedy má podobu UDTOK. Tento řetězec je zpracován IDS, avšak paket s daty D již nedorazí na cílovou stanici.

K zajištění stavu, kdy koncová stanice nepřijme všechna data, lze použit následující tři způsoby. Prvním z nich je využití rozdílné doby vypršení časovače detekčního systému a clonového systému. Defaultně je časovač systému Snort nastavený na 60 sekund, kdežto časovač Linux/FreeBSD jen na 30 sekund. Důsledkem je zahození vypršených paketů tudíž odfiltrování šumových dat.[11]

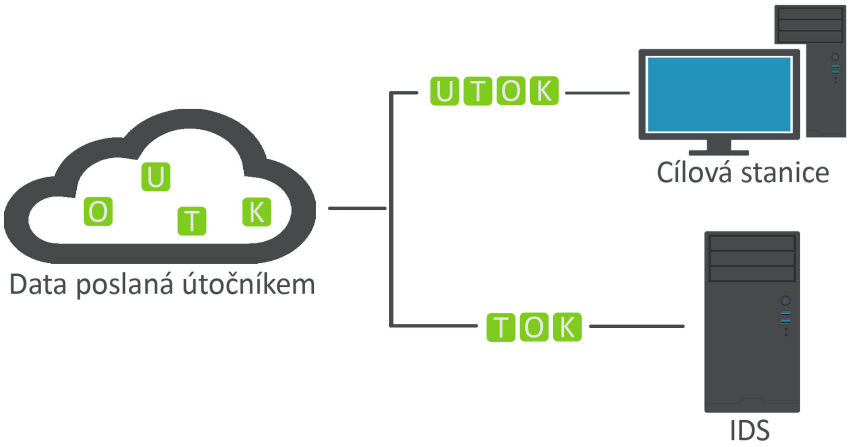
Pro další dva způsoby je pro útočníka nutná určitá znalost topologie cílové sítě. U předchozího útočník potřeboval znát pouze platformu, na niž útočí, a detekční systém. Informace o platformě lze zjistit z pasivního odposlouchávání komunikace z ní probíhající na základě různých sekvencí příkazů zasílaných pří sestavování TCP spojení nebo na základě různých hodnot TTL v hlavičkách paketů, jelikož rozdílné systémy používají rozdílné defaultní hodnoty.[18] K analýze získaných údajů lze použít nástroj p0f[17] od Michala Zalewskiho. Topologii sítě lze zjistit posíláním ICMP paketů Echo a odchytávání zpráv Echo Reply popřípadě Time Exceeded. Avšak zprávy Echo Reply bývají, často právě z důvodu skenování sítě, blokovány firewally. Jestli se na síti nacházejí i zařízení připojené přes linku podporující menší velikost rámců než IDS, lze zjistit nastavením příznaku DF (nefragmentovat) v IP hlavičce, a následně čekat na ICMP zprávu Fragmentation Needed. Dále lze opět použít nástroj p0f, kdy na základě znalosti defaultních hodnot TTL jsme schopni určit pozici stroje v síti.

Konkrétně šumové pakety mají nastavenou jinou hodnotu TTL, díky které je detektor sice příjme, ale na dalších směrovačích po cestě k cílové stanici klesne hodnota na nulu a pakety jsou zahozeny. Jelikož však internetová síť je paketová síť a ne okruhová síť, může každý paket být přenášen jinačí cestou a tak šumové pakety mohou být zahozeny již před senzorem. Druhou nevýhodou, společnou i pro další metodu, je nutnost určité specifické topologie sítě.[11]

Poslední zde uvedu metodu zneužívající MTU podsítě mezi IDS a stanicí. Pokud v této oblasti je MTU nižší než v sítí, ke které je připojen senzor, lze zaslat pakety s DF příznakem. Tyto pakety budou opět za IDS zahozeny a cílové stanici již nepřijdou.

3.2.2 Evasion

V případě, že cílem metody insertion bylo přidat do toku pakety, které příjme jen IDS, pak technika evasion dělá naprostý opak. IDS mylně předpokládá, že určitý paket již nepatří ke stejnému datovému toku a ten se díky tomu vyhne kontrole.



Obrázek 3.2: Evasion

Pro lepší porozumění využiji obrázek 3.2. Útočník použil útok reprezentovaný řetězcem UTOK. Zde však zaslal přímo tento řetězec do cílové sítě. IDS paket s daty U nepřiřadil

aktuálnímu toku, tudíž analyzoval pouze řetězec TOK, který prošel. Cílová stanice však obdrží plný řetězec UTOK.

Jestliže insertion útok byl způsoben nedůsledností IDS, evasion zase využívá jeho přehnané důslednosti. Obecně je pro útočníka výhodnější, když IDS pakety s útokem vůbec neobdrží, než když je jen špatně dekóduje.

Tuto metodu lze opět použít při rozdílných časech vypršení platnosti dat pro IDS a systém. Zatímco u předchozí techniky byl časovač IDS větší než stanice, zde je to naopak. Datům na IDS již vypršela platnost a tak nově příchozí data jsou zařazena do nového proudu, ale stanice je zařadí ještě do předchozího.[11]

3.2.3 Fragmentace IP paketů

Fragmentování bylo vytvořeno za účelem spolehlivého přenesení dat na různých linkách podporujících různé velikosti rámců. Některé detekční systémy k rozlišení, zdali probíhá útok, nebo ne kontrolují délku paketu. Tohoto lze využít pro nucené rozdělení paketů na více menších paketů než je nutné. V tomto případě, i když pakety dojdou ve správném pořadí, není útok detekovaný. Zpětným sestavením toku by se dal tento způsob útoku odhalit, avšak tato operace je příliš náročná na zdroje, tudíž není realizovaná všemi detekčními systémy. Navíc se může stát, a protokol to dovoluje, že pakety nepřijdou ve správném pořadí.[16]

S fragmentováním souvisí i problém s překrývajícími daty. Obvykle když dojde k rozdělení paketu na menší, tak každý menší paket nese určitou část dat, ale útočník může vytvořit speciální paket, jenž bude přepisovat už došlá data extrahována z předešlých částí. Tato situace nastává obvykle v okamžiku různých velikostí fragmentů. Zpracování těchto dat se liší systém od systému, tudíž IDS musí data interpretovat stejně jako systém, který chrání.[8]

3.3 Úprava dat aplikační vrstvy

Metody v předchozí sekci se zabývaly přidáváním šumu do síťové komunikace na nižších vrstvách. V této části uvedu způsoby úpravy dat v různých aplikačních závislostech. V souvislosti s OSI modelem se nacházíme v aplikační, relační a prezenční vrstvě.

3.3.1 Protocol rounds

Vícero protokolů z důvodu úspory síťového prostoru nadbytečnou režií dovoluje sestavit více aplikačních sezení skrze jedno již vytvořené připojení. Na druhou stranu z výkonnostních důvodů mnoho senzorů kontroluje pouze prvotní ustavení připojení, nově vznikající spojení již tedy procházejí bez kontroly. Toho lze využít v případě útoku sestavením korektního úvodního spojení, až poté spustit samotný útok.[16]

3.3.2 Striktní vyhodnocování protokolů

Můžeme nalézt aplikace, které neimplementují vyhodnocování protokolu tak striktně oproti IDS. Například u protokolu HTTP útočník může vytvořit proud dat porušující specifikace, jelikož většina webových serverů ignoruje drobné chyby a nesprávný formát dat a data vyhodnotí.

Naproti tomu NIDS implementuje striktní vyhodnocování protokolů a ten tato data nezpracuje a zahodí jako nesprávná, jelikož mylně předpokládá zahození dat i webovým

serverem. Takovýto proud lze vytvořit vkládáním bílých znaků, CR znaků, vkládáním nevhodných dat do numericky zpracovávaných polí a další. Podobné techniky je možno využít i proti IMAP nebo FTP.[16]

3.3.3 Zranitelnost při porovnávání řetězců

Útoky na zranitelnost při porovnávání řetězců jsou nejjednodušší pro pochopení a vytvoření. Detekční metody na základě otisků skoro výhradně spoléhají na porovnávání řetězců. Vytvoření řetězce pro překonání špatně napsané detekční analýzy je triviální. I většina behaviorálně založených analyzátorů má velkou závislost na porovnávání řetězců.

Jednoduchost této techniky předvedu na následujícím příkladu. Budu požadovat přístup k souboru `/etc/passwd`. Tento soubor je textová databáze obsahující informace o uživatelích, kteří se můžou přihlásit do systému, příslušnost ke skupinám a dalších uživatelských identitách operačního systému. Typické oprávnění umožňuje číst jej všemi uživateli, zapisovat však může pouze superuživatel, nebo za pomoci několika speciálně privilegovaných příkazů i jiný uživatel.

Pro detekci přístupu k tomuto souboru používá Snort tuto signaturu:

```
alert tcp $EXTERNAL_NET any -> $HTTP_SERVERS 80 (msg:"WEB-MISC
/etc/passwd"; flags: A+; content:"/etc/passwd"; nocase;
classtype:attempted-recon; sid:1122; rev:1;)
```

Signatura se snaží nalézt přesný řetězec `"/etc/passwd"`. Tato jednoduchost dovoluje upravit řetězec za pomoci tzv. no-ops.

No-ops znamenají systémová volání, která nemají žádný efekt, nebo je jejich efekt irelevantní vůči požadovanému cíli. V našem kontextu budou tyto operace znamenat např. změnu adresáře.

Výsledný příkaz pak bude mít tuto podobu:

```
/etc/ rc.d/../passwd
```

Pečlivě vytvořenou databázi signatur, jež jsou dostatečně obecné, je možno tuto úpravu detekovat. Od této jednoduché techniky se dostáváme k pokročilejší, která je mnohem složitější na detekci. Opět se pokusíme získat souboru `/etc/passwd`, tentokrát ale použijeme interaktivní sezení skrze telnet. Existence signatur hledající běžné názvy programů používaných k útokům nebo názvy souborů není tak neobvyklá. Obejít nastalou situaci lze za použití příkazu určitého řádkového interpretu. Příklad zde uvedený využije interpretu jazyka Perl pro zamaskování příkazu `cat /etc/passwd`.

```
badguy@host$ perl ?e
?$foo=pack(?C11?,47,101,116,99,47,112,97,115,115,119,100);
@bam=‘/bin/cat/ $foo‘; print?@bam\n?;?
```

Neexistuje způsob, který lze použit pro běžné porovnávání řetězců, jenž odhalí daný útok. Obrana je tedy mnohem složitější, jelikož IDS musí rozumět interpretu a znát jeho výstup.[15]

3.3.4 Polymorfní Shellcode

Polymorfní shellcode je maskovací technika limitovaná svým použitím pouze na útoky využívající přetečení bufferu. Rozdíl oproti klasickému útoku je patrný z obrázku 3.3. Základní útok nahraje do bufferu pouze kód, který vykoná útočníkem požadovanou operaci, dále pak přepíše EIP registr adresou náležející první instrukci škodlivého kódu, popřípadě přepsaní kódu náležející pro SEH, ale zde již čistě záleží na principu funkce daného útoku.

Obrázek 3.3: Polymorfní útok

Konkrétně si útok můžeme představit jako sekvenci [NNN][SSS][RRR], kdy N znamená NOOP instrukce, S je kód vykonávající požadovanou operaci a R je návratová adresa, například hodnoty, které se nahrají do EIP.

Po zakódování má útok podobu [RnRn][D][CsCsCs][RRR]. Na začátku jsou opět NOOP instrukce, ale byla zde sekvence klasických NOP instrukcí (na platformě IA32 kód 0x90) nahrazená alternativními instrukcemi se stejným důsledkem pro vykonávaný kód. Pro platformu IA32 existuje až 55 různých náhrad. Dále byly přidány instrukce pro dekódování škodlivého kódu následovány zakódovaným kódem. Kódovaní musí být symetrické k instrukcím dekodéru. Návratová adresa se nemění, tudíž pro spuštění kódu lze použit stejné principy jako pro základní útok.

Zakódování je nejčastěji prováděno za pomoci XOR operací, kdy se náhodně vygeneruje klíč a následně se provede XOR všech instrukcí s tímto klíčem. Dekódovací hlavička poté obsahuje smyčku procházející paměť o předem určené délce a za pomoci dříve vygenerovaného klíče, zpětně dekóduje dané instrukce.

3.3.5 Další způsoby maskování

Závěrem se ještě zmíním o dvou možnostech vyhýbaní se detekci. Prvním z nich je použití alternativního kódování, druhý je šifrované spojení.

Alternativní kódováni - z výkonnostních důvodů, nebo pro zabezpečení integrity dat dovoluje mnoho aplikací použití více způsobů kódování dat, příkladem může být BASE-64.

Některé NIDS při normalizování HTTP provozu jsou zranitelné vůči URL-kódovaným řetězcům.[16]

Šifrované spojení - pokud se útočníkovi podaří vytvořit šifrované spojení s cílovou stanicí, stává se jeho následující komunikace nekontrolovatelnou. Toto vychází z principu NIDS a šifrování. NIDS pouze odchytává paket proudící v síti, a jelikož šifrování zabraňuje získání dat z odchycených paketů, tak pak ani NIDS nezná význam odchycených dat.

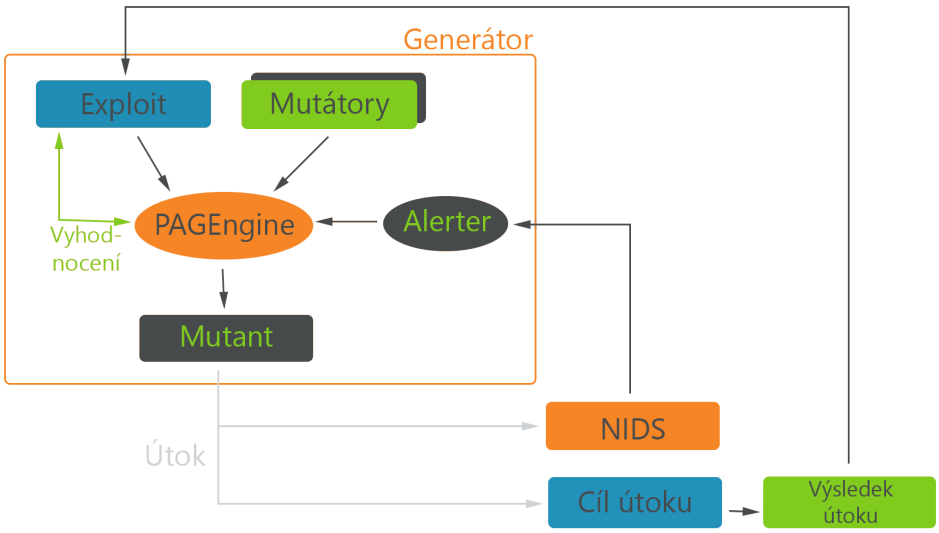
Kapitola 4

Návrh aplikace

Jak už bylo zmíněno v úvodu, cílem této práce bylo vytvořit generátor polymorfních síťových útoků, což znamená, že tato práce obsahuje i naprogramovanou aplikaci, která provádí zasílání velkého množství odlišně působících útoků, s cílem vyvolat stejnou situaci na cílovém stroji.

4.1 Model generátoru

Při návrhu modelu generátoru jsem se inspiroval návrhem použitým autory v práci Testing Networkbased Intrusion Detection Signatures Using Mutant Exploits[16]. Výsledný návrh lze vidět na obrázku 4.1.



Obrázek 4.1: Model enginu

Už od samotného počátku byla aplikace tvořena jako určitý nástroj sloužící k testování NIDS uživateli na různých službách, tudíž musí umožňovat i použití různých druhů exploitů, pokud možno i uživatelsky definovaných. Tento přístup se odrazil i v samotném návrhu modelu generátoru.

Jak je z výše uvedeného návrhu patrné, generátor sám o sobě nedokáže určit, zdali je námi použitý útok funkční nebo dokonce zdali se útok vyhne detekci. Aby mohl toto rozhodnout, potřebuje, funkční správně nastavené NIDS a cílový stroj, který interpretuje daný škodlivý kód.

Jádro generátoru tvoří PAGEngine, tento objekt je zodpovědný za použití jednotlivých mutátorů k úpravě signatury daného otisku, dále je zde Alerter, který kontroluje, zdali byl daný útok detekovaný.

Za nedílnou součást generátoru lze považovat i jednotlivé exploity a mutátory. Bližšímu popisu jednotlivých komponent se budu věnovat v následující kapitole.

4.2 Použité technologie

Jako programovací jazyk byl při návrhu, pro svoji rychlost tvorby i rychlost výsledné aplikace, vybrán jazyk C# a prostředí .NET. Toto prostředí rovněž umožňuje vytvořit grafické uživatelské rozhraní na slušné úrovni.

S použitím .NET a jazyka C# souvisí i návrhový model MVVM. Tento model je podobný klasickému MVC, jelikož se taky snaží o oddělení jádra aplikace od uživatelského rozhraní a zavádí zde, na první pohled zbytečný, objekt rozhraní. V prvotní fázi vývoje může docházet a dochází k prodlužování zdrojových kódů a času implementace, ale tato negativa se odráží v následné jednodušší údržbě, kdy při změně nějaké části buďto v uživatelském rozhraní nebo v jádře stačí změnit pouze referenci v rozhraní, není tak potřeba procházet stovky řádků kódu a postupně opravovat daná volání.

Dříve zmíněná zkratka MVVM znamená Model View ViewModel. Tento návrhový vzor pochází z dílny microsoftu. Model byl určen pro událostmi-řízené programování ve Windows presentaion foundation (WPF).

Model v MVVM lze chápat stejně jako v MVC, je to vlastní jádro aplikace, datová část obsahující definice jednotlivých objektů a operací s daty. View označuje uživatelské rozhraní vytvořené za pomocí jazyka xaml, jedná se opět o podobnou komponentu jako v MVC. Odlišností však je ViewModel. Existuje zde podobnost s rozhraním u MVC, taky zajišťuje komunikaci uživatelského rozhraní s modelem, případně modelu s uživatelským rozhraním, ale zde je odlišností přítomnost tzv. binderu. Binder umožňuje napojit proměnnou z uživatelského prostředí např. výpis nějaké hodnoty, nebo povolení tlačítka, na určitou veřejnou proměnnou ViewModelu. Kdy při změně hodnoty proměnné se tato změna projeví i v ViewModelu i v uživatelském rozhraní. Dále binder umožňuje vytvoření příkazu, kdy dochází na základě povolovací funkce příkazu k povolení jeho spuštění, a následného provedení.

Použití .NET frameworku by se mohlo jevit jako určité omezení pouze na Windows platformu, ale existují i porty pro unix. Naopak výhodou může být fakt, že po pouhém upravení uživatelského rozhraní lze použit generátor i jako webovou aplikaci. Ani by zde neměl nastat větší problém při úpravě do mobilní verze pro Windows Phone. Za další nespornou výhodo lze vidět použití vývojového prostředí Visual Studio a jeho debuggeru.

Kapitola 5

Implementace

V následujících podkapitolách bych se už chtěl věnovat podrobnějšímu popisu jednotlivých komponent. Pro vývoj aplikace jsem zvolil metodiku inkrementálního přístupu. Jednotlivé iterace pak probíhaly cestou vytvoření dané komponenty v uživatelském rozhraní, úpravy jejich grafických vlastnosti, aby zapadala do koncepce grafického návrhu a následné implementace důležitých vlastností komponenty ve ViewModelu, případně pak vytvoření datového modelu, jenž je nutný pro komponentu. Následovaly testy nově přidané funkcionality. Úspěšné testy znamenaly ukončení dané iterace.

Přestože jsem nejprve navrhoval grafické uživatelské rozhraní, jeho popis si nechám až na závěr kapitoly.

5.1 Načítání pluginů

Jako prvnímu bych se chtěl věnovat načítání jednotlivých exploitů a mutátorů. Jak již bylo dříve zmíněno, aplikace umožňuje zařadit nové útoky. K postupu, kdy uživatel může rozšířit stávající funkcionalitu, se nejlépe hodí zkompilovat daný útok a zařadit ho ve formě DLL knihovny.

Samotné začlenění dané knihovny pak probíhá po startu aplikace, kdy se rekurzivně prohlédnou složky `Exploits` pro exploity a `Evaders` pro mutárory, z prohledávání se vyřazují podsložky s názvem `libs`. Následně se provede klasické načtení každé nalezené DLL knihovny a pro všechny její třídy se provede kontrola, jestli se jedná o veřejnou třídu a zdali implementuje požadované rozhraní. Pokud je podmínka splněna, provede se instanciace nalezeného objektu a uložení jeho instance do seznamu exploitu, nebo mutátorů.

Z toho přístupu vyplývá omezení, kdy pro každý nově přidaný nebo odebraný plugin musíme provést restart aplikace. Další důsledek je, že lze jen v omezené míře kontrolovat, zdali nedojde k načtení nějakého pluginu dvakrát. Tuto kontrolu lze provádět jen na základě kontroly názvu souboru. Pokud dojde k opakovanému načtení stejného sestavení, dochází u dříve načtených a vytvořených objektů ke ztrátě reference na XAML soubory. Tuto situaci ale řeší jádro aplikace.

5.2 Rozhraní Exploitu

Aby bylo možné používat uživatelsky definovaný exploit, musí mít implementováno rozhraní IExploit. Toto rozhraní obsahuje celkově deset metod, které můžeme rozdělit do tří skupin.

První skupinu tvoří metody samotného exploitu:

- `Exploit SetExploit(IPAddress IPv4, IPAddress IPv6 )` - metoda, která vrací objekt `Exploit`. Tato metoda může být zavolána kdykoliv a oba její parametry mohou být nedefinované.
- `object SetView()` - metoda, která vrací komponentu `UserControl`. Tato metoda se volá poté, co uživatel spustil útok ale před spuštěním samotného útoku. Díky této metodě lze zajistit nastavení všech potřebných vlastností, které může uživatel exploitu nastavit.
- `bool IsOk()` - metoda, která vrací, jestli je exploit správně nastaven.

Další skupinu tvoří metody používané k ověření, zdali byl útok úspěšný. Jedná se o tyto tři metody:

- `bool SuccessInit()` - metoda, která se volá před každým novým zasláním útoku, pokud bude návratová hodnota false, dojde k ukončení útoků.
- `bool Success()` - metoda používaná k ověření, zdali byl útok úspěšný.
- `void SuccessFinalize()` - metoda, která je zavolána, jakmile jsou dokončeny všechny útoky.

Poslední skupinu tvoří funkce informativního charakteru:

- `bool string Name()` - vrací jméno daného exploitu, které bude zobrazeno. Toto jméno je čistě informativního charakteru a tak nemusí dodržovat žádné konvence.
- `string Hint()` - krátký textový popis daného exploitu.
- `UInt16 StdPort()` - jak už název odpovídá, tato funkce vrací standartní port, na kterém cílová služba běží, např. pro HTTP server se jedná o port 80. Tato hodnota je použita pokud uživatel nespecifikoval v parametrech útoku jinačí port.

Toto rozhraní dále využívá třídu `Exploit`. Tato třída byla vytvořena pro snazší reprezentaci škodlivého kódu. Každý objekt se skládá ze dvou položek a to z hlavičky a těla. Hlavička nese jméno `cred` a jedná se o pole řetězců `ExStrings`. Tělo pak může být buď objektem `BodyShellcode` se jménem `shell`, nebo opět pole `ExStrings` s označením `strings`. Jelikož objekt obsahuje oba atributy pro tělo, je nutné, aby jeden obsahoval hodnotu `null`. Tato hodnota je automaticky nastavena jazykem C#. Pokud by uživatel nastavil nenulově oba atributy, útok skončí s chybou. Položka `cred` může být `null`.

Třída `ExStrings` byla zavedena pro odlišení řetězců určených k úpravě, od řetězců, které upraveny být nesmí. Tohoto chování je docíleno vytvořením dvou podtříd `BaseString` a `MutaString`, pro uložení hodnoty se zde používá běžný řetězec jazyka C#, tato proměna je veřejná. Dále obsahuje objekt ještě parametr `IsExtended`, který určuje, zdali další řetězec v poli je rozšířením aktuálního a musí být odeslán zároveň. Toto chování však zaručuje síťová služba, a tak v případě použití uživatelského síťového mutátoru nemusí být zajištěna. U `MutaString` existuje i možnost specifikovat délku, jenž nesmí výsledek přesáhnout. `ExStrings` neobsahuje veřejný konstruktor a tak musí uživatel volat přímo konkrétní potomky.

Naproti tomu třída `BodyShellcode` je trochu složitější a tak si dovolím její parametry vypsat v seznamu. Výsledný zasílaný kód se skládá z konkatenace bytových polí v uvedeném pořadí.

- `public Byte[] junks` - jedná se o pole "odpadních"bytů, které mohou být použity k určitému zaplnění bufferu.
- `public int nop` - hodnota, určující kolik bytů pole shellcode znamená NOOP instrukce.
- `public Byte[] shellcode` - samotný škodlivý kód s NOOP instrukcemi na začátku.
- `public int afterNop` - hodnota, kolik NOP instrukcí se má vygenerovat a přiřadit za shellcode. Toto musí zajistit daný mutátor.
- `public Byte[] afterJunks` - opět "odpadní"byty sloužící k zaplnění bufferu.
- `public Byte[] retAdress` - hodnota, kterou se přepíše EIP
- `public Byte[] banned` - pole určující, které byty se nesmí objevit ve výsledných polích. S tímto polem jako jediným se neprovádí konkatenace.

5.3 Rozhraní mutátorů

Tak jako exploity i každý mutátor musí implementovat speciální rozhraní. Na rozdíl od exploitu však zde existují dvě rozhraní. Toto dělení je zapříčiněno situací, kdy lze použít mutátor i na síťovou vrstvu, která potřebuje speciální rozhraní pro mutovací funkci, dále to umožňuje kontrolu, aby na síťové vrstvě mohl být použit jen mutátor, který je pro ní určen. Obě tato rozhraní jsou vytvořena na základě dědičnosti od rozhraní `EvadeInterface`, které teď podrobněji popíši.

Rozhraní `EvadeInterface` obsahuje pár podobných metod jako již dříve popsané rozhraní `IExploit`:

- `object SetView()` - viz 5.2
- `string Name()` - viz 5.2
- `string Hint()` - viz 5.2
- `bool IsOk()` - viz 5.2
- `void MutantSuccess(SuccesStatus status)` - vrací úroveň, na které daný mutátor pracuje. Pro řazení exploitu jsem vymyslel čtyři vrstvy, na kterých mohou pracovat. Tyto vrstvy vycházejí z OSI modelu a jsou to: transportní, relační, prezentační a aplikační. Jednotlivé mutátory jsou volány stejně jako v případě OSI modelu. Dělení na vrstvy se využívá pro možnost kombinace jednotlivých mutátorů. Z každé vrstvy lze použít v jednu chvíli jen jeden mutátor. Mutátory na transportní síťové vrstvě musí implementovat rozhraní `INetEvade`, ostatní vrstvy pak `IExploitEvade`.

Rozhraní `IExploit` poskytuje navíc jen metodu `Exploit Evade(Exploit exp, int attackCount)`, která zajistí patřičnou úpravu exploitu který vrátí pro další použití.

Naproti tomu `INetEvade` rozšiřuje základní rozhraní trochu komplexněji a to konkrétně o tyto tři metody:

- `void Evade(Exploit exp, IPAddress IPv6, IPAddress IPv4, UInt16 port)` - provádí mutace dat při odesílání na cílovou stanici. Jako mutaci si můžeme představit např. fragmentace dat. Povinností je zajistit doručení dat na cílovou stanici specifikovanou parametry.
- `void Finalizer()` - metoda, která se zavolá po skončení všech útoků.
- `IPWorks IPSupport()` - určuje verzi IP vrstvy na které evader pracuje. IPWorks může nabývat třech hodnot, pokud je podporovaná IPv4, IPv6 nebo obě adresy.

5.4 Generátor polymorfních síťových útoků

Aplikace po spuštění zobrazí jedno hlavní okno, které lze vidět na obr. 5.1. Návrh okna se snaží zapadnout do ModernUI koncepce od Microsoftu, a tak je rámeček okna podobný s programy jako je např. Visual Studio nebo Microsoft Word. Pro dosažení výsledku okna, které kolem sebe nezobrazuje standartní Windows rámeček, byla použita volně dostupná knihovna `MahApps.Metro`[6].

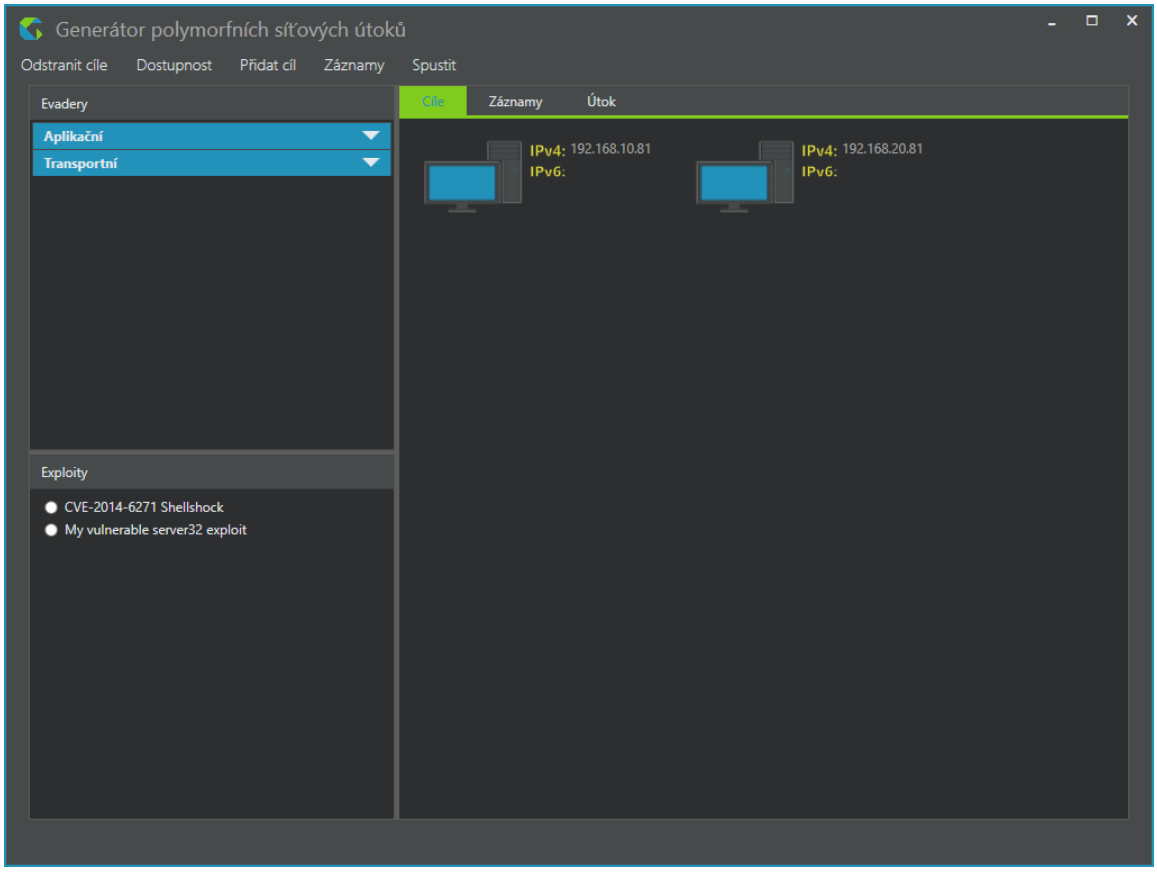
Knihovna `MahApps.Metro` sama o sobě přináší ModernUI šablonu vzhledu okna. Žel tato knihovna je velmi špatně dokumentovaná a z dostupné dokumentace vyplynulo, že jestli chci použit vlastní styl okna, budu tuto knihovnu moci využít jen pro zajištění správných rozměrů okna a možnost zvětšovat okno tažením za mnou definovaný rámeček. Tuto funkcionalitu sice obsahuje již základní okno poskytované WPF, avšak je potřeba zároveň použít základní styl okna. Z knihovny dále využívám nekonečnou animaci při zaneprázdnění aplikace. A poslední věcí je hlavní menu, kdy došlo jen k upravení barev.

Okno aplikace se snaží zachovat klasickou funkcionalitu, na kterou jsou uživatelé zvyklí, jako je přemístění tažením za titulek nebo maximalizací. Jediná odlišnost nastává v okamžiku, kdy uživatel chce obnovit okno z maximalizace tažením za titulek. Zde se okno zmenší již při kliknutí a ne až po stlačení tlačítka myši a jejího pohybu.

Problém mnoha aplikací můžeme vidět v nekonsistentním návrhu, nebo nevhodně zvolené kombinací barev. Tuto situaci jsem řešil vhodně zvoleným výběrem barev. Jako základ jsem pro aplikaci zvolil tmavé téma. Vycházel jsem nejen z osobních preferencí, ale také ze zkušenosti s kreativním programy od společnosti Adobe nebo programem Microsoft Visual Studio a dále i ze skutečnosti, že mnoho programátorů preferuje tmavé barvy vývojových prostředí.

I když na schématu 4.1 je uvedena třída `PAGEngine` jako jádro generátoru, jedná se pouze o jádro modelu. Faktickým jádrem aplikace je třída `MainWindowViewModel`, která se stará o správu hlavního okna (obr. 5.1), ale i volání přidružených objektů. Aplikační okno můžeme rozdělit na čtyři oblasti. První z nich je hlavní menu. Hlavní menu je plně v režii objektu `MainWindowViewModel`, který se stará o jeho vykreslení i vykonání všech požadovaných příkazů. Další dvě oblasti jsou si podobné, jedná se o `Evadery` a `Exploity`, zde `MainWindowViewModel` pouze zajišťuje vyhrazení místa a pro samotné zobrazení se už volají přidružené view modely. Poslední oblast je v částečné režii. `MainWindowViewModel` zajišťuje vykreslení záložek a jejich přepínání, avšak samotný obsah opět zajišťují přidružené modely. Rozměry jednotlivých oblastí, kromě hlavního menu, umožňují jejich změnu tažením za dělící části. Jediným omezením je minimální velikost daných oblastí.

Spodní část okna je vyhrazena pro zobrazení aktuálně důležitých informací. Zobrazuje se zde buď nekonečná animace, nebo indikátor průběhu a taky řetězec popisující právě prováděnou činnost.



Obrázek 5.1: Hlavní okno aplikace

Přidání nového cíle lze provést kliknutím na odpovídající položku v menu. Následně se otevře dialogové okno pro zadání potřebných údajů. O toto okno se opět stará jeho vlastní view model. U přidávání nového cíle existuje možnost ověření jeho dostupnosti za pomocí odeslání `ICMP` paketu `ECHO` a následného obdržení `ICMP` paketu `ECHO REPLY`. Nevýhodou tohoto testu je, že nemůže stoprocentně vyvrátit nedostupnost cílové stanice, jelikož některá nastavení firewallu z bezpečnostních důvodů blokují tyto ICMP pakety. Spuštění testu dostupnosti lze provést i na již přidané cíle.

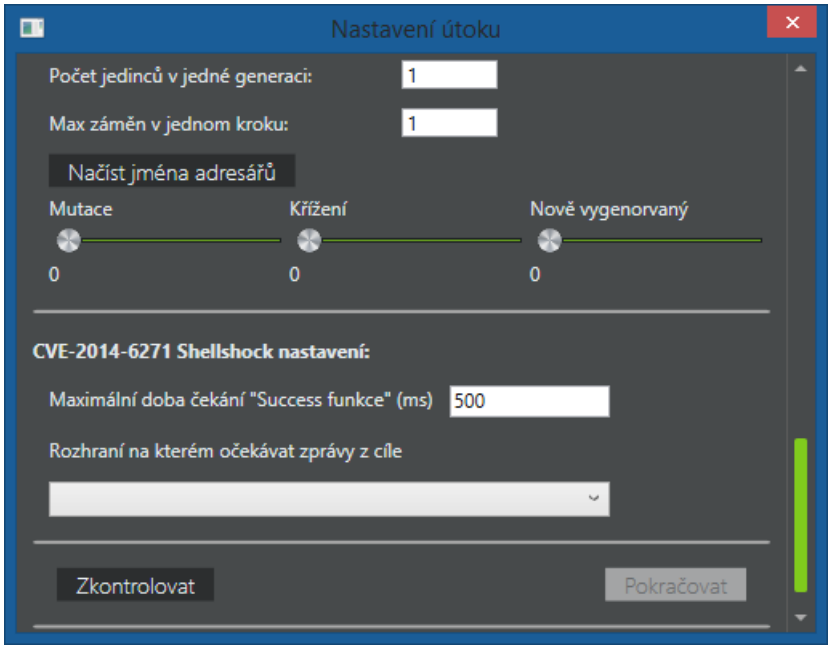
Ke spuštění útoku je zapotřebí vybraného alespoň jednoho cíle a nutnost vybrat exploit. O nesplnění podmínek je uživatel informován chybovým hlášením. Jestliže jsou obě podmínky splněny, vytvoří se nová instance `EngineViewModel` (dále jen `EVM`). `EVM` obdrží od hlavního modelu celkem tři `Listy` obsahující všechny exploity, evadery a cíle. Tato data zpracuje, nastaví třídní proměnnou na referenci na vybraný exploit a z evaderů sestaví čtyřprvkové pole, kdy každý index znamená jednu úroveň mutátorů. Možnost vybrat z jedné vrstvy pouze jeden mutátor je zajištěna GUI vrstvou, ale jestliže by přeci jen nastala situace, kdy by jich bylo vybráno více, použije se první v seznamu evaderů pro danou vrstvu. Pokud uživatel nechce na určité vrstvě použít žádný mutátor, nastaví se příslušná hodnota v poli na `null`. Dále pak dojde k určení vybraného exploitu. Z cílů se pouze zjistí počet označených, z čehož pak vyplývá nutnost dodatečných nastavení.

Pro všechny vybrané exploity a mutátory se zavolá funkce `SetView()`, získané objekty se použijí jako parametr pro instanciaci třídy `HideUC`, jež se následně uloží do struktury `List`. Jak už název napovídá, třída `HideUC` se stará pouze o skrytí objektu `UserControl`,

který se uloží do jejího jediného atributu. Nutnost zavedení nové třídy vyplynula z chování výpisu struktury `List` ve WPF, které neumožňuje dodatečné nastavení vzhledu.

Problém nastíněný v kapitole 5.2 o ztrátě reference na XAML soubor, je řešen vytvářením nové instance všech evaderů před přidáním do pole. Také u exploitu se provede nová instanciace. Toto řešení zároveň zajistí nastavení jejich defaultních hodnot při spuštění nového útoku.

Následně se řízení opět vrací `MainWindowViewModel`, jenž si získá z `EVM List` objektů `HideUC`, tento list pak předá nově vytvořenému objektu `ShowMultipleViews`, který má na starosti jen zobrazení tohoto Listu. Příklad můžeme vidět na obr. 5.2.



Obrázek 5.2: Nastavení útoku

Jakmile uživatel vše nastavil, potvrdil pokračování, které je podmíněno vrácením hodnoty `true` funkcí `IsOk()` pro všechny použité mutátory a exploit, provede řídící objekt `MainWindowViewModel` nastavení portu u všech vybraných cílů na defaultní hodnotu, případně na uživatelem specifikovanou hodnotou, dále pak vytvoření defaultní obsluhy sítě jestliže nebyl vybrán mutátor pracující na síťové vrstvě. Poslední činností, co provede, je vytvoření instance `Alerteru`, `PAGEenginu` jemuž v konstruktoru předá odkaz na objekt výpisu a cíle a spuštění enginu. Tímto úloha objektu `MainWindowViewModel` v útoku končí.

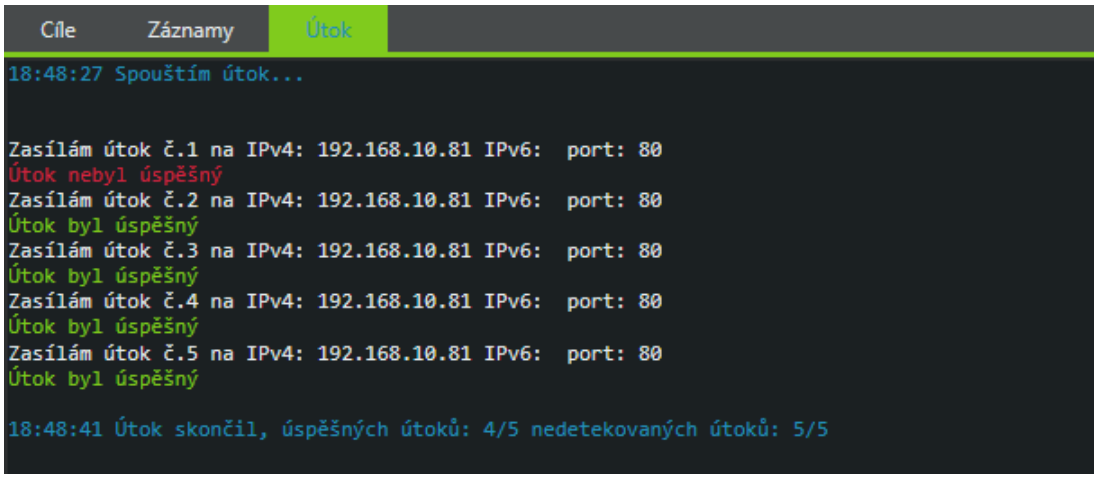
`PAGEngine` obdrží v parametrech referenci na `EVM`, cíle, `Alerter`, množství útoku a v neposlední řadě referenci na funkci pro aktualizování hodnoty průběhu. Aby proměnná mohla být použitá v GUI, musí mít definovaný `getter` a `setter`, ale takovou proměnnou nelze předávat referenci, tudíž bylo nutné vytvořit funkci na její aktualizaci, jenž musí umět pracovat v situaci zavolání z jiného než GUI vlákna. Samotný útok pak probíhá v novém vlákně a tlačítko spuštění útoku není povoleno, dokud útok neskončí.

`PAGEngine` může pracovat ve třech režimech, které se liší jen drobností. První režim se používá, pouze pokud byl vybraný jen jeden cíl a zasílá nový exploit, dokud se neuskuteční požadovaný počet útoků. Druhý zašle všechny útoky nejdříve na první cíl, a pak na další. Zároveň umožňuje zasílat pokaždé stejnou sadu útoků. Třetí režim pak zasílá útoky na cíle postupně, zde se hodí upozornit, že pokaždé zasílá nový exploit.

Postup útoku je již pak u všech třech režimů stejný, nejdříve se provede mutace `Exploitu` na prvních třech vrstvách, následně se zkontroluje, zdali byl specifikovaný mutátor i pro transportní vrstvu, pokud ano použije se pro odeslání dat on, v opačném případě se použije základní síťová služba. Před každým útokem se pracuje s novým `Exploitem`. Zde bych si dovolil upřesnit a upozornit, že se jedná o objekt `Exploit` vrácený exploitem. Získání toho objektu z exploitu se provede pouze jednou, následně se vytváří hluboká kopie toho objektu.

Po odeslání útoků se volá funkce exploitu pro zkontrolování, zda byl útok úspěšný, a dále pokud bylo specifikováno použití `Alerteru`, zavolá se pro zjištění detekce. Klíčovou myšlenkou zjišťování detekce je, že jak Snort tak i Suricata, podporuje zasílání varovných zpráv přes `syslog` protokol a tak lze předpokládat dostupnost této funkce i v ostatních NIDS. Před započetím útoku se inicializuje UDP klient na portu 514, jenž je specifikován v RFC. Následně se kontroluje s maximální čekací dobou nastavenou uživatelem, jestli byl obdržen `syslog` paket. Jakmile časovač vyprší, je útok považovaný za nedetekovaný.

Na závěr se pouze vypíšou informace pro uživatele. Tento výpis má podobu klasického konzolového výpisu s barevným odlišením viz obr. 5.3. Výpis se skládá ze struktury `List`, jenž obsahuje objekty `CLOstring`. Tento objekt obsahuje dvě informace: řetězec k vypsání a WPF štětec označující barvu textu ve výpisu. Po vypsání se provede zápis aktuálního exploitu pro uložení. K ukládání se využívá XML soubor, kdy se zapisuje pořadí útoku, název exploitu, funkčnost, vyhnutí se detekci, IPV4 adresa cíle, IPv6 adresa cíle, port, čas, zpráva od NIDS a samotné tělo exploitu. Pokud ještě nebyly odeslány veškeré útoky, pokračuje automaticky dalším. Uživatel v nastavení útoku může zvolit, aby se dalším útokem pokračovalo až po jeho interakci. Toto zastavování lze v průběhu útoku vypnout, nikoliv však znova zapnout.



Obrázek 5.3: Nastavení útoku

K uložení záznamu do souboru dojde až po odeslání všech útoků do souboru pojmenovaného časem zahájení útoku do složky `Records`. V případě vzniklé výjimky v průběhu útoků se vypíše nastalá chyba a dojde k přerušení útoku. Za takovou chybu lze považovat použití nějakého mutátoru na exploit nevhodného typu. I přes vzniklou chybu dojde k uložení záznamů.

Generátor byl implementován v nejmenší verzi .NET, který zajistil všechny požadované funkcionality a tak je určen pro verzi 4.0.

5.5 Implementované exploity a mutátory

Součástí aplikace bylo i vytvoření polymorfních síťových útoků, jejichž cílem je úspěšné spuštění a vyhnutí se detekci. K dosažení tohoto cíle bylo nutné implementovat pár exploitů a mutátorů, jejichž popisu se budu věnovat v následujících podkapitolách.

5.5.1 CVE-2014-6271 Shellshock

Jako první zde uvedu tzv. shellshock útok na apache server. Shellshock je bezpečnostní chyba v unixovém Bash shellu, kdy při speciálně sestaveném řetězci začne bash vykonávat příkazy z řetězce. Toho chování lze dosáhnout, pokud se v řetězci nalézají příkazy pro definici funkce `() { : ;};`.

Problém v případě apache serveru souvisí se zpracováním dat z hlavičky z HTTP požadavku. Běžně se stává, že web server pro lepší jednodušší zkoumání převádí jednotlivé položky hlavičky na vnitřní proměnné. Samotný převod by nebyl problém do té doby, dokud by tyto proměnné zůstávaly v samotném serveru. Problém nastává, jakmile jsou tyto proměnné předány bashi.

Můj exploit zasílá na daný server klasický GET HTTP požadavek pro získání stránky. Hlavička obsahuje standartní informace, jako kódování nebo jazyk. Pro útok využívám upravené pole User-Agent, kde předávám magický řetězec pro spuštění útoku `() { : ;};`, dále se pak snažím o spuštění programu `/bin/client.out`, následuje IP adresa očekávající odpověď a číslo portu. `Client` je jednoduchý program, který zašle na parametry specifikovanou IP adresu a číslo portu UDP paket obsahující řetězec "pwned via shellshock". Exploit tedy před samotným odesláním musí mít běžící službu na dané IP adrese a portu, která tento paket přijme. Na základě obdržené zprávy pak exploit označí tento útok za fungující.

5.5.2 MyVulnServerExploit

Druhým a posledním implementovaným exploitem je `MyVulnServerExploit`. Jedná se o útok na přetečení bufferu, cílící na speciálně vytvořený program server32. Server32 je TCP server, který na standartní výpis vypisuje řetězec přijatý přes síť. Zmíněné přetečení bufferu nastává ve výpisu, kdy zásobník pro přijímání dat je mnohem větší, než následné pole, do kterého se řetězec kopíruje. Pro kopírování se používá klasická C funkce `strcpy`, která kopíruje zásobník tak dlouho, dokud nenarazí na nulový znak `\0` znamenající konec řetězce.

V nejpoužívanějších Little Endian architekturách roste halda z vyšších adres na nižší. V případě volání programu se uloží na haldu potřebné registry a adresa, kterou se bude pokračovat při návratu z programu. V případě deklarování proměnné se tato proměnná alokuje způsobem, že se od ukazatele na vrchol haldy odečte velikost dané proměnné v bytech. Adresa proměnné je dána aktuální hodnotou ukazatele na zásobník a ukládání hodnoty pak probíhá se vzrůstajícími adresami, kdy při dostatečně velkém množství dat dokážeme přesáhnout buffer a přepsat všechny dříve uložené hodnoty. Obecně tato situace končí neoprávněným přístupem do paměti, avšak v okamžiku, kdy nahrajeme přesný počet dat s konkrétní hodnotou, jsme schopni získat kontrolu nad programem.

Aby bylo možné tento typ útoku uskutečnit, bylo nutné při překladu programu server32 vypnout ochranu haldy, dále pak pro jednoduchost byla na cílové stanici vypnuta randomizace adres ASLR a samotný program byl přeložen do 32 bitové verze.

Sever32 pro uložení dat používá pole o velikosti 1024 bytů. EIP se nachází až 1036 bytů od začátku pole. Tato velikost byla určena zkoumáním registru v procesoru za pomoci ladícího programu GDB, zasláním určitého vzoru dat. Do EIP je nutné nahrát určitou adresu

z bufferu, na které se nachází požadovaný kód k provedení. V případě nevypnutého ASLR by se tato adresa při každém spuštění měnila a bylo by nutné najít v různých systémových knihovnách, které nejsou ASLR ovlivněny, adresu instrukce jmp EIP.

Do bufferu nahrávám 500 NOOP instrukcí. NOOP instrukce se vyžívají pro bezpečný skok na vykonávaný kód, jelikož se obvykle jedná o 1 bytové instrukce, které neprovádějí žádnou činnost s vedlejšími efekty na tok programu, lze je použít pro přibližný skok. Po NOOP instrukcích následuje už škodlivý kód.

```
0xEB,0x1F,0x5E,0x89,0x76,0x08,0x31,0xC0,0x88,0x46,0x07,0x89,0x46,0x0C,
0xB0,0x0B,0x89,0xF3,0x8D,0x4E,0x08,0x8D,0x56,0x0C,0xCD,0x80,0x31,0xDB,
0x89,0xD8,0x40,0xCD,0x80,0xE8,0xDC,0xFF,0xFF,0xFF,0x2f,0x62,0x69,0x6e,
0x2f,0x68,0x63
```

Uvedený škodlivý kód jsou hexadecimální kódy strojových instrukcí pro spuštění systémové služby `execve`, která spustí mnou požadovaný program `/bin/hc`. Tento program má uloženou adresu cíle a hodnotu portu, na který odešle UDP paket s řetězcem "pwned via bu". Po úspěšném ukončení spuštěného programu, dojde k ukončení celého serveru pomocí příkazu `exit`. Na cílovém stroji tedy kromě ukončeno serveru nezbydou žádné stopy po útoku.

Na shellcode navazuje 491 "odpadních"bytů pro zaplnění bufferu a adresa skoku. Adresu skoku lze zjistit zkoumáním spuštěného programu v GDB, nebo si ji rovnou vypsat samotným bufferem. Jediná věc, na kterou si je třeba dát pozor, je že v GDB se adresa může mírně lišit. Tato nuance má za následek funkční útok, pokud je server spuštěný v GDB a však neoprávněný přístup do paměti, pokud je spuštěný z příkazové řádky. V opačném směru, při použití adresy získané běžným spuštěním, jsem se s neoprávněným přístupem do paměti nesetkal. Ale bezpečnější než používat přímo adresu začátku bufferu, by bylo použít určitou adresu z bufferu, která ukazuje na náš NOOP sled.

5.5.3 NetworkEvader

Nyní se už dostáváme k mutátorům. Začnu tím nejjednodušším a to je `NetworkEvader`. Jak je již z názvu patrné tento mutátor pracuje na transportní vrstvě a je tedy zodpovědný za odeslání dat.

Úprava podpisu útoku probíhá odesíláním různě velkých paketů. Tato úprava se děje už při přístupu k rozhraní, nikoliv až TCP fragmentací. Uživatel může navolit velikost paketu, jakou chce, nebo maximální velikost, kdy aktuální velikost bude náhodně vygenerována.

Provádění fragmentace nad daty může být nevhodné u aplikací, které nepoužívají žádný přenosový protokol, na druhou stranu je zde zcela stejný přístup jak k TCP tak k UDP tokům.

Jazyk C# obsahuje objekty pro práci se síťovým rozhraním, kdy lze pracovat na dvou úrovních: za pomocí `TCP klientu`, nebo `socketu`, což je jen obálka pro `winsock`. Já jsem si vybral `socket`, jelikož se jedná o přístup k síti na nižší úrovni. V rámci optimalizace je u každého připojení zapnutý Nagle algoritmus, který se snaží více menších odeslání seskupit do jednoho většího a šetřit tím linku menší TCP režií. Toto chování je z pohledu mutace nevhodné tudíž ho bylo nutné vypnout, dále je třeba u každého odeslání přesně specifikovat množství dat k odeslání a nastavit vlajku SocketFlags.Partial, jinak nedojde k odeslání paketu malých velikostí, což u blokujícího přístupu vede k zamrznutí aplikace.

Mutátor umožňuje pauzy mezi jednotlivými odeslanými daty, ale pouze jen u exploitů s těly tvořenými řetězci, kde nejsou všechny řetězce označeny jako rozšířené.

5.5.4 NixStringMutator

Další mutátor se již zaměřuje na aplikační vrstvu. Jeho použití je omezené na exploity, jejichž tělo je tvořeno řetězci a cílová aplikace musí běžet na unixovém systému.

Mutátor pracuje s daty v textovém režimu, kdy umožňuje přidávat tyto vlastnosti:

- Seberefence - do řetězce se přidá `/./`, jenž znamená přístup k aktuálnímu adresáři. Pro zápis cesty to nic neznamená, ale naruší to porovnávání na shodu řetězců.
- Adresářový průchod - do řetězce se přidává `/dir/../`, kde `dir` znamená konkrétní jméno adresáře. Z pohledu cesty to znamená pouze zanoření se o jednu úroveň a ihned opětovné vynoření. Jména adresářů může uživatel specifikovat souborem, ve kterém jsou jednotlivá jména oddělena mezerou. Zároveň lze použít náhodně generována jména adresářů a mutátor obsahuje i jména základních unix root adresářů.
- Komentáře - do řetězce jsou náhodně vložené komentáře buď typu `$(: text_komentáře )`, nebo `‘# text_komentáře‘`, tyto komentáře jsou pak při zpracovávání bashem ignorovány. Pro generování slov komentáře jsem využil generátor náhodných slov od Luke Sampson[9].
- `%xx` kódování - provede se nahrazení náhodného znaku jeho hexadecimálním ascii kódem s procentem na začátku, např. písmeno A se nahradí hodnotou %41.
- Unix aliasy - přidá příkaz na vytvoření přezdívky pro určitý podřetězec, který pak na jeho originálním místě nahradí kódem, jenž při následném zpracování provede zpětnou náhradu přezdívky za originální text.
- Proměnné prostředí - podobná funkce s Unix aliasy, akorát pro náhrady využívá proměnné prostředí.

Při použití je nutno ještě specifikovat maximální počet mutací v jednom kroku a dále pak maximální počet jedinců v jedné generaci. Zvlášť druhá podmínka je důležitá, jelikož díky ní lze použit genetické algoritmy pro vytváření nových exploitů. Během prvního zavolání mutátor vytvoří nově vygenerovanou sadu jedinců. Při každém dalším volání postupně zasílá jednotlivé jedince až do okamžiku, kdy zaslal všechny. Poté se provede se sadou dle specifikovaných vah mutace, křížení, nebo nově vygenerování na základě původní hodnoty exploitu. Tady lze vidět, že při nevhodně zvoleném počtu jedinců nemusí být využit potenciál genetických algoritmů. Při genetických operacích je přihlédnuto k úspěšnosti předchozího útoku. Konkrétní mutační funkce je vybraná na základě náhodně vygenerovaného desetinného čísla, u kterého se zjišťuje interval, do kterého spadá. Dolní hranice intervalu je nula. Do horní hranice se taky nejprve nahraje nula a postupně se k ní přičítá normalizovaná váha pravděpodobnosti použití kontrolované funkce.

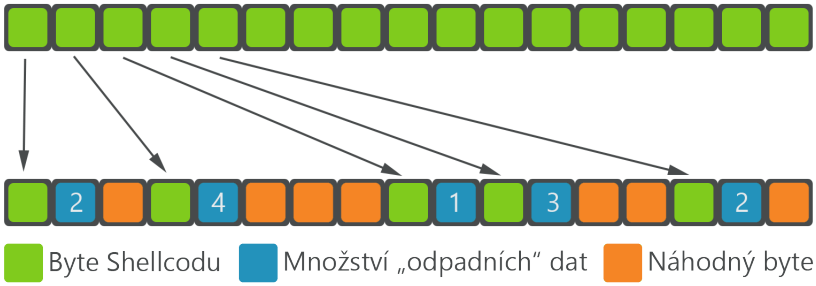
Křížení pracuje na principu, kdy se vygeneruje náhodně křížící bod. Poté se hledá na základě fitness funkce další vhodný jedinec a provede se vzájemná záměna, kdy pokud byl první řetězec A1A2 a druhý B1B2, jako výsledné řetězce se použijí A1B2 a B1A2. Jestliže není nalezen další řetězec vhodný ke křížení provede se mutace.

Fitness funkce rozlišuje pouze dva stavy a to jestli útok byl funkční, nebo ne. Na základě toho pak rozhodne, jestli se může daný exploit použít, nebo ne. Fitness funkce se používá pouze u křížení, kdy pokud není možné použít daný exploit, tak se sice nepoužije ke křížení, ale provede se jeho nahrazení původním řetězcem, na který se použije mutace.

5.5.5 Beaker

`Beaker` je označení pro poslední mnou implementovaný mutátor. Tento mutátor je určen pro aplikační vrstvu a pracuje s exploity, jejichž tělo je tvořeno objektem `BodyShellcode`. Při návrhu mutátoru jsem využil princip i rutinu strojových instrukcí pro zpětný překlad kódu do původní podoby ze stránky Fun Over IP[12]. Použití mutátoru je omezeno pouze na platformu IA32.

Princip mutátoru je znázorněn na obrázku 5.4. Pro každý byte shellcodu se náhodně vygeneruje číslo od 1 do 4 určující počet vložených bytů. Dále se vygeneruje o 1 menší počet odpadních bytů (jeden byte už zabírá samotné určení délky). Generuji pouze byty, jejichž hodnota se vyskytuje v běžném textu.



Obrázek 5.4: Princip kódování shellcodu

Výsledný zakódovaný kód je ovšem delší než původní, a proto se ubere z NOOP instrukci na začátku. Jestliže není dostatek místa v NOOP instrukcích, mutace se neprovede a v exploitu zůstane zachován původní kód.

Dále mutátor umožňuje nahrazení NOOP instrukce jejími alternativami, kdy uživatel opět může zvolit jen instrukce, jejichž kódy znamenají nějaký běžný znak řetězce. Tato úprava se snaží vyhnout detekci na pravidlo NOOP sledu, zaměřující se na větší výskyt bytů s hodnotou `0x90`. Kromě toho, může uživatel zvolit úplnou eliminaci NOOP instrukcí na začátku, kdy v případě, že je zvoleno i kódování, se nejdřív škodlivý kód zakóduje, následně se počet zbylých NOOP instrukcí na začátku odejme a za shellcode se přidá stejný počet náhodných řetězcových znaků.

Kapitola 6

Testování aplikace

Testování aplikace bylo prováděno na lokálním počítači s virtualizovanými operačními systémy. Samotná aplikace běžela na hostitelském počítači s operačním systémem Windows 8.1 Pro, pro pokročilejší informace o síti byl používán program Wireshark. Adresa toho počítače byla v obou NIDS nastavena jako externí. Oba exploity útočily na stejný systém a tím byl virtualizovaný Linux Mint 13 Cinnamon 64-bit bez aktualizací s nainstalovaným webovým serverem apache2. K virtualizaci jsem použil Hyper-V.

6.1 NIDS Snort

Počátečním testovaným NIDS byl `Snort` ve verzi 2.9.7.2 a k tomu pravidla k dispozici pro registrované uživatele aktuální k 17. 4. 2015. Samotný `Snort` byl nainstalovaný ve virtualizovaném počítači s operačním systémem Windows 8.1 Enterprise. Zasílání zpráv na `Syslog` se povedlo nastavením patřičné adresy cílového počítače v `Syslog` sekci v konfiguračním souboru.

Prvním zvoleným útokem byl MyVulnServer exploit a úplně na začátku došlo k testu s neupraveným útokem, zda jsou všechna pravidla správně nakonfigurovaná. Snort v tom útoku správně rozeznal Nop sled.

Po ujištění se o funkčnosti celého testovacího prostředí, jsem pokračoval testy útoku s úpravami. Jelikož `Snort` neobsahoval signaturu pro mnou spouštěný proces, rozhodl jsem se nejdříve obejít detekci Nop sledu za pomoci `Beakeru`. Při použití jakékoliv možnosti pro kódování byly útoky nedetekovány, jediné omezení bylo s použitím nebezpečných instrukcí, kdy občas nebyl úspěšný samotný útok. Toto chování odpovídalo očekávání, protože všechna pravidla `Snortu` očekávají použití stejné, nebo dvou opakujících se instrukcí při záměně za standartní. Při použití náhodně vkládaných ekvivalentů, kdy mnohé z nich odpovídají svou hodnotou znakům běžně se vyskytujícím v řetězci, není možné toto odhalit.

Následně jsem přidal tyto dvě signatury pro odchycení epxloitu:

```
alert ip $EXTERNAL_NET any -> $HOME_NET any (msg:"VulnServer exploit
cont"; content:"/bin/hc";  nocase; depth:1000; sid:1000004)

alert ip $EXTERNAL_NET any -> $HOME_NET any (msg:"VulnServer exploit
cont"; content:"bin/hc";  nocase; depth:1000; sid:1000005)
```

Testovací útok opět potvrdil funkčnost pravidla. Při použití kódování shellcodu znova z modulu `Beaker` byly všechny útoky nedetekované. Po těchto testech jsem přidal tyto signatury spolupracující se Snort dekodéry:

```
alert ip $EXTERNAL_NET any -> $HOME_NET any (msg:"VulnServer exploit
asn1"; content:"bin/hc"; asn1:bitstring_overflow; nocase; depth:1000;
sid:1000006)

alert ip $EXTERNAL_NET any -> $HOME_NET any (msg:"VulnServer exploit
asn1"; content:"bin/hc"; gid: 116; nocase; depth:1000; sid:1000007)

alert ip $EXTERNAL_NET any -> $HOME_NET any (msg:"VulnServer exploit
asn1"; content:"bin/hc"; gid: 127; nocase; depth:1000; sid:1000008)
```

Avšak ani zde nebyl Snort úspěšný s detekcí útoků. Jediné možné řešení, které vyplynulo, bylo přidat signaturu `alert ip $EXTERNAL_NET any->$HOME_NET 9988(msg:"VulnServer exploit buffer check"; dsize:>1024; sid:1000009)`, která kontroluje samotnou délku délku zasílaných na server. Tímto byl Snort schopný 100% detekovat útok a obejití tohoto pravidla není možné.

typ útoků	počet útoků	počet funkčních	počet nedetekovaných	počet úspěšných
test NOOP	1	1	0	0
nahrazení NOOP ASCII ekvivalenty, použití i nebezpečných instrukcí	42	40	42	40
nahrazení NOOP	20	20	20	20
test signatury	1	1	0	0
zakódované tělo	21	21	21	21
pokročilá pravidla	21	21	21	21
pravidlo velikosti dat	1	1	0	0

Tabulka 6.1: Útoky za použití MyVulnServer

Jiná situace byla u druhého testovaného exploitu. Opět jsem začal čistým útokem, který byl odhalen. Následně jsem použil modul `Network evader`, avšak i při použití maximální velikosti paketu `20 B` nebo i `10 B` byl útok pokaždé detekovaný.

Situace se opakovala i při použití `Unix string mutator`. Z toho modulu byly použity nejdříve úpravy s přidáním komentářů, u kterých jsem nejvíce věřil v obejití detekce, ale při testování s 300 útoky se nenašel jediný exploit, který by byl funkční a nedetekovaný. `Snort` dokonce odhalil i plno nefunkčních útoků.

K vyhnutí se detekci ani nepomohly další úpravy. Při zatržených všech možnostech v mutátoru a generování 700 útoků se situace opakovala jako i při použití pouze komentářů.

typ útoků	počet útoků	počet funkčních	počet nedetekovaných	počet úspěšných
test detekce	21	21	0	0
paket max 20 B	21	21	0	0
přidání komentářů	200	113	9	0
všechny NixString úpravy	700	50	49	0
bez náhodných adresářů, dělení dat	700	369	25	0

Tabulka 6.2: Útoky za použití Shellshock

6.2 NIDS Suricata

NIDS `Suricata` běžel ve virtualizovaném prostředí Ubuntu 14. Verze NIDS byla 2.0.7 s pravidly stažených pomoci programu `wget` z adresy `http://rules.emergingthreats.net/open/suricata/emerging.rules.tar.gz` dne 21.4.2015. Pro zasílání `Syslog` zpráv na vzdálený počítač bylo nutné nejprve zapnout v konfiguračním souboru `Syslog` logování a jelikož `Suricata` sama o sobě nepodporuje zasílání na vzdálený server, bylo nutno využít funkce samotného `rsyslog` a ve složce `rsyslog.d` vytvořit soubor pro odeslání všech zpráv z programu `Suricata`.

Stejně jako u předchozího NIDS i zde byl nejdříve testovaný `MyVulnServer` exploit. Při testovaní `Suricata` překvapila alarmem se zprávou `SURICATA STREAM CLOSEWAIT FIN out of window`. Pro další testování bylo nutné toto pravidlo vypnout. `Suricata` sama o sobě neobsahuje pravidlo vůči NOP sledu, tudíž bylo další testování tohoto pravidla zbytečné.

`Suricata` umožnuje použití pravidla `Snortu`, avšak vyžaduje, na rozdíl od `Snortu` středník i za poslední hodnotou před koncovou závorkou. Díky tomu jsem mohl, s menšími úpravami ohledně volaných dekodérů, využít pravidla definovaná v předchozí podkapitole. Při testování s těmito pravidly, bylo dosaženou naprosto shodných výsledků jako u `Snortu`.

typ útoků	počet útoků	počet funkčních	počet nedetekovaných	počet úspěšných
test NOOP	1	1	1	1
test signatury	1	1	0	0
zakódované tělo a pokročilá pravidla	10	10	10	10
pravidlo velikosti dat	1	1	0	0

Tabulka 6.3: Útoky za použití MyVulnServer

Stejné výsledky se dostavily i při testování druhého exploitu. Zde jsem zkusil zasílat i pakety o maximální velikosti `5 B`, ale nevedlo to k jakémukoliv úspěchu. Odchycený datový tok při tomto testu, lze najít na přiloženém DVD, kde je záznam provedený programem `Wireshark`, filtrující pakety pouze pro daný `apache` server a `Syslog` zprávy od NIDS.

typ útoků	počet útoků	počet funkčních	počet nedetekovaných	počet úspěšných
test detekce	21	21	0	0
paket max 5 B	21	21	0	0
přidání komentářů	320	117	39	0
všechny NixString úpravy	700	14	15	0
bez náhodných adresářů dělení dat	700	259	71	0

Tabulka 6.4: Útoky za použití Shellshock

Kapitola 7

Závěr

Cílem této práce bylo zhotovit aplikaci pro generování polymorfních síťových útoků. Uživatelé tak mohou tuto aplikaci využít při testování samotných detekčních systémů, ale i při testování zabezpečení celkové sítě.

V rámci tvorby byl nejprve sestaven návrh architektury aplikace. Pro možnost rozšíření a lepší využití aplikace došlo k návrhu dvou rozhraní pro přidávání exploitů a mutátorů, kde každé tvoří jeden DLL soubor. Dále bylo navrženo grafické uživatelské rozhraní pro co možná nejlepší uživatelskou přívětivost dané aplikace. Také v rámci zadání byly vytvořeny dva exploity a tři mutátory používající různé maskovací techniky pro vyhnutí se detekci. Na závěr došlo k testování výsledné aplikace, exploitů a mutátorů na reálně nasazených detekčních systémech.

Výstupem práce je aplikace, schopná generovat velké množství útoků. Při testování bylo použito v rámci jednoho testu maximálně 700 útoků, avšak limit počtu útoků v aplikaci není. O všech útocích jsou prováděny záznamy v XML formátu s aktuální hodnotou exploitu. Uživatel si následně může tento záznam v aplikaci otevřít a podívat se na detail útoku. Pro konkrétní hodnoty samotných paketů je potřeba využít programy třetích stran. Dále díky použitým rozhraním může uživatel do aplikace snadno zařadit nový exploit nebo mutátor.

Výsledky testovaní se silně odvíjí od použitého exploitu nebo mutátoru. Při použití Buffer overflow útoku bylo jasně ukázáno, že i v dnešní době je často nedetekovatelný. Samotné systémy spoléhají na pravidla obsahující otisk aktuálně známého útoku případně použitých technik pro maskování instrukcí. Například Snort obsahuje pravidlo pro detekování kódování shellcodu za pomocí Shikata ga nai algoritmu. Z výsledku tudíž vyplývá, že než používat pravidla na konkrétní signatury útoku, je lepší použít pravidla na existující chyby, jak bylo dokázáno pravidlem hlídajícím maximální délku zaslaných dat. Naopak při použití klasických řetězcových útoků lze vidět velikou vyspělost současných detekčních systémů, které byly schopny odhalit veškeré útoky. Na druhou stranu při testování jsem se setkal s omezením na hostitelský systém. Jelikož oba NIDS potřebují pro svou funkci na platformě Windows wincap driver, který podporuje pouze specifikaci sítě NDIS 5 a ne verzi 6, která je ve Windows od verze Vista a díky tomu ho nelze použít na operačním systému Windows 10, jenž od sestavení 10041 nepodporuje verzi 5.

Aplikaci lze rozšířit přidáním nových mutátorů nebo exploitů. Dalším rozšířením by mohlo být zaznamenávání i samotných paketů zasílaných aplikací v rámci útoků, ale k tomuto by z principu funkce samotných mutátorů muselo být použito zaznamenávání všech dat na rozhraní, jak to dělá například Wireshark. Dalším rozšířením by mohla být automatická analýza záznamů útoků a hledání vzorů pro úspěšné útoky.

Literatura

[1] BACE, Rebecca Gurley. *Intrusion detection.* Indianapolis: Macmillan Technical Publishing, c2000, xix, 339 s. ISBN 15-787-0185-6.

[2] BACE, Rebecca; MELL, Peter. *NIST special publication on intrusion detection systems.* BOOZ-ALLEN AND HAMILTON INC MCLEAN VA, 2001. Dostupné z: http://permanent.access.gpo.gov/lps72073/sp800-31.pdf

[3] CISCO. *Snort.Org* [online]. 1998, 2015 [cit. 2015-04-27]. Dostupné z: https://www.snort.org/

[4] DEBAR, Hervé. An introduction to intrusion-detection systems. *Proceedings of Connect*, 2000, 2000. Dostupné z: http://www.pcporoje.com/filedata/947354.pdf

[5] FFIEC Information Technology Examination Handbook InfoBase. *FFIEC IT Examination Handbook InfoBase - Network Intrusion Detection Systems* [online]. neznámý [cit. 2015-12-13]. Dostupné z: http://ithandbook.ffiec.gov/it-booklets/information-security/security-monitoring/activity-monitoring/network-intrusion-detection-systems.aspx

[6] JENKINS, Paul. *MahApps.Metro* [online]. 2011, 2015 [cit. 2015-04-26]. Dostupné z: http://mahapps.com/

[7] LONVICK, Chris. *The BSD syslog protocol* [online]. 2001 [cit. 2015-01-05]. Dostupné z: https://tools.ietf.org/html/rfc3164

[8] PTACEK, Thomas H.; NEWSHAM, Timothy N. *Insertion, evasion, and denial of service: Eluding network intrusion detection.* SECURE NETWORKS INC CALGARY ALBERTA, 1998. Dostupné z: https://sparrow.ece.cmu.edu/group/731-s08/readings/ptacek-newsham.pdf

[9] SAMPSON, Luke. Random Word Generator. In: *CodePlex - Open Source Project Hosting* [online]. 2010 [cit. 2015-03-27]. Dostupné z: https://randword.codeplex.com/

[10] SCARFONE, Karen; MELL, Peter. Guide to intrusion detection and prevention systems (idps). *NIST special publication*, 2007, 800.2007: 94. Dostupné z: http://csrc.nist.gov/publications/nistpubs/800-94/SP800-94.pdf

[11] SIDDHARTH, Sumit. Evading NIDS. In: *All of Connect — Symantec Connect* [online]. 2005, 2010-01-02 [cit. 2015-01-01]. Dostupné z: http://www.symantec.com/connect/articles/evading-nids-revisited

[12] Simple shellcode obfuscation. In: *Fun over IP* [online]. 2011 [cit. 2015-04-26]. Dostupné z: https://funoverip.net/2011/09/simple-shellcode-obfuscation/

[13] SUNDARAM, Aurobindo. An introduction to intrusion detection. *Crossroads* [online]. 1996-04-01, vol. 2, issue 4, s. 3-7 [cit. 2015-01-01]. DOI: 10.1145/332159.332161. Dostupné z: http://portal.acm.org/citation.cfm?doid=332159.332161

[14] THE OPEN INFORMATION SECURITY FOUNDATION. *Suricata — Open Source IDS / IPS / NSM engine* [online]. 2009, 2015 [cit. 2015-04-27]. Dostupné z: http://suricata-ids.org/

[15] TIMM, Kevin. IDS Evasion Techniques and Tactics. In: *All of Connect — Symantec Connect* [online]. 2002, 2010-11-02 [cit. 2015-01-01]. Dostupné z: http://www.symantec.com/connect/articles/ids-evasion-techniques-and-tactics

[16] VIGNA, Giovanni, William ROBERTSON a Davide BALZAROTTI. Testing network-based intrusion detection signatures using mutant exploits. In: *Proceedings of the 11th ACM conference on Computer and communications security - CCS '04: Washington, DC, USA, October 25-29, 2004* [online]. New York, New York, USA: ACM Press, 2004, s. 21- [cit. 2015-04-27]. ISBN 1581139616ISSN 1-58113-961-6. DOI: 10.1145/1030083.1030088. Dostupné z: http://portal.acm.org/citation.cfm?doid=1030083.1030088

[17] ZALEWSKI, Michal. P0f v3. *P0f v3* [online]. neznámý [cit. 2015-01-05]. Dostupné z: http://lcamtuf.coredump.cx/p0f3/

[18] ZALEWSKI, Michal. *Silence on the wire: field guide to passive reconnaissance and indirect attacks*. Vyd. 1. San Francisco: No Starch Press, 2005, 281 s. ISBN 15-932-7046-1.

Příloha A

Obsah DVD

Obsah DVD je složen z následujících adresářů:

/Src
: Visual studio 2013 projekt aplikace, exploitů, mutátorů a jejich zdrojové kódy

/Bin
: Spustitelná verze aplikace

/Rec
: Záznamy z útoků provedených v rámci testování

/Doc
: Technická zpráva dokumentace

/Tex
: Zdrojové kódy technické zprávy

/Help
: Generovaná dokumentace

/Exploit necessities
: Soubory které je nutno nahrát na cílovou stanici pro otestování implementovaných exploitů.

Příloha B

Návod k instalaci

Instalace aplikace se provede nakopírováním obsahu složky `/Bin` do cílového adresáře.