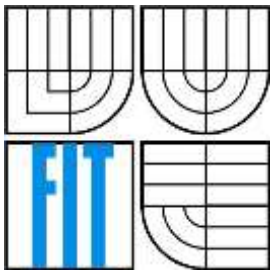
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



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

KLASIFIKACE ROOTKITŮ A JIMI POUŽÍVANÝCH TECHNIK

ROOTKITS CLASSIFICATION

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR
Bc. Radovan Plocek

VEDOUCÍ PRÁCE
SUPERVISOR
prof. Ing. Tomáš Hruška, CSc.

BRNO 2014

Abstrakt

Práce popisuje současné nejrozšířenější metody používané rootkity. Obsahuje základní principy, se kterými se vývoj rootkitů pojí, jako jsou používané registry, ochrana paměti a nativní API operačního systému Windows. Hlavním cílem této práce je poskytnout přehled technik, které rootkity využívají, jako jsou hákování, modifikace kódu a přímá modifikace systémových objektů a navrhnout způsob, jakým bude možné tyto techniky detekovat. Tento návrh bude posléze základem pro implementaci detektoru a odstraňovače rootkitů založených na zmíněných technikách.

Abstract

This paper describes information about current most widespread methods, which are used by rootkits. It contains basic information connected with development of rootkits, such as process registers, memory protection and native API of Windows operation system. The primary objective of this paper is to provide overview of techniques, such as hooking, code patching and direct kernel object modification, which are used by rootkits and present methods to detect them. These methods will be then implemented by detection and removal tools of rootkits based on these techniques.

Klíčová slova

Rootkit, Detekce rootkitů, Hákování, Modifikace kódu, DKOM

Keywords

Rootkit, Rootkits, Rootkits detection, Hooking, Code patching, DKOM

Citace

Plocek, Radovan: Klasifikace rootkitů a jimi používaných technik, diplomová práce, Brno, FIT VUT v Brně, 2014

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením prof. Ing. Tomáše Hrušky, CSc.
Další informace mi poskytl Ing. Vojtěch Vobr ze společnosti AVG.
Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

……
Bc. Radovan Plocek
28. 5. 2014

Obsah

Seznam obrázků

Seznam tabulek

1 Úvod

Počítačová bezpečnost je v moderní době čím dál tím více diskutovaným tématem. Zatímco počítačovým virům a červům se dostává velké pozornosti, existují i další potenciálně nebezpečné druhy softwaru, které nejsou tak dobře známé. Mezi ně patří právě rootkity, jejichž hlavním cílem je poskytnutí utajení a umožnění případné vzdálené kontroly nad systémem. Sami o sobě nepatří mezi škodlivý software, ale jsou spíše prostředkem, který je možno použít pro škodlivé účely. Lze je ale použít i pro bezpečnostní účely – jako součást softwaru pro detekci útoků, pro skrytí bezpečnostního softwaru v počítači či jako součást ochrany mobilních počítačových zařízení proti krádeži. V rámci této práce se ale budeme zabývat zejména případy, kdy je rootkitů použito pro škodlivé účely.

Abychom mohli omezit míru poškození, musíme znát způsoby, kterými rootkity dosahují utajení. Metod, kterými se mohou začlenit do struktur operačního systému, je větší množství, ale my si v této práci představíme ty nejdůležitější, kterými jsou hákování různých systémových tabulek, modifikace kódu systémových volání a přímá modifikace struktur jádra. Představíme si zároveň, jakým způsobem je možné tyto metody detekovat a případně odstranit. Z těchto metod si vybere pár nejzajímavějších, pro které následně vytvoříme detektor.

1.1 Vytčení cíle

Vývoj rootkitů je činnost, která vyžaduje velmi dobré pochopení fungování operačního systému a jeho jádra a schopnost orientovat se v mechanismech ochrany paměti. Z tohoto důvodu bude první část této práce zaměřena na stručné představení toho, jaké registry procesoru se používají k jakým účelům, a jak je implementována ochrana paměti pomocí segmentace a stránkování. Protože v celé této práci budeme používat 32bitovou operačního systému Windows 7, popíšeme si i způsob, jakým funguje jeho sada standartních funkcí nazývaná nativní API a jakým způsobem fungují systémové ovladače, které jsou nejběžnější formou zavedení kódu rootkitu do systému. Zároveň si v této části popíšeme obecné anti-forenzní techniky, které rootkity mohou používat pro dosažení svých cílů.

Druhá část této práce bude zaměřena již přímo na rootkity. Poznatky získané v první části si aplikujeme při popisu technik, které rootkity používají pro dosažení svých cílů, kterými mohou být blokování systémových volání, sledování systémových volání, filtrace výstupních volání či utajení přítomnosti některých procesů, systémových ovladačů či souborů. Těchto cílů je možno dosáhnout pomocí různých metod, v této práci

si ovšem popíšeme tři nejpoužívanější, kterými jsou hákování, modifikace kódu za běhu a přímá modifikace systémových objektů.

Hlavním cílem celé práce by mělo být vytvoření nástroje pro detekci a případné odstranění rootkitů. Proto si v poslední části této práce představíme způsoby detekce a odstranění zmíněných technik a na jejich základě realizujeme detektor.

1.2 Popis kapitol

Nejprve si v kapitole 2 probereme základy, na kterých jsou rootkity stavěny. Vzhledem k tématu této práce si nejprve vysvětlíme pojem rootkit a uvedeme si i jeho stručnou historii. Na to navážeme přehledem registrů používaných procesory rodiny IA-32 a představíme si techniku ochrany paměti pomocí segmentace a stránkování. Dále si v této kapitole popíšeme nativní API Windows a použití systémových ovladačů, které budeme potřebovat pro vývoj rootkitu i detektoru. Následně si představíme jednu z nejmodernějších metod prevence před rootkity v 64bitových systémech Windows. Na závěr této kapitoly si řekneme něco k forenzní analýze a anti-forenzním technikám, které mohou být využity u rootkitů.

Ve třetí kapitole se konečně dostaneme k nejrozšířenějším technikám, které rootkity používají. Začneme s hákováním systémových tabulek. Nejdříve si popíšeme techniku hákování v uživatelském módu (IAT) a následně si představíme techniky v režimu jádra, jako jsou hákování IDT, MSR, SSDT či IRP. Pokračovat budeme technikou modifikace kódu systémových volání za běhu systému. Na závěr této kapitoly si představíme jednu z nejmodernějších technik a to DKOM, což označuje přímou modifikaci systémových objektů.

Ve čtvrté kapitole si popíšeme několik existujících rootkitů a zejména si uvedeme, jaké techniky využívají a za jakým účelem. Mezi tyto rootkity patří Agony, FU rootkit a rodina rootkitů TDSS.

V páté kapitole si popíšeme způsoby, jakými lze detekovat a případně odstranit techniky používané rootkity, které byly popsány ve třetí kapitole.

V rámci páté kapitoly se podíváme na realizaci detektoru a odstraňovače rootkitů založených na technice hákování SSDT tabulky a přímé modifikaci systémových objektů.

1.3 Návaznost na semestrální projekt

Tato práce navazuje na semestrální projekt „Klasifikace rootkitů a jimi používaných technik“, ze kterého přebírá teoretický základ uvedený v kapitole 2 a techniky rootkitů popsané v kapitole 3. Tyto převzaté kapitoly jsou navíc rozšířeny o další informace.

2 Teoretický základ

Předtím, než se začneme zabývat přímo rootkity, metodami, které používají a možností jejich detekce a odstranění, vysvětlíme si několik základních principů, které se v rootkitech využívají. Jedná se zejména o používané registry, paměť a její ochranné mechanismy, prostředky pro vývoj v systému Windows a na závěr něco málo o forenzní analýze a anti-forenzních technikách

Tato kapitola je napsána na základě [1], [2], [3], [4]. Množství informací o vývoji bylo převzato i ze [7], kde jsou dostupné i zdrojové kódy existujících rootkitů.

2.1 Rootkit

Výraz rootkit pochází původně z unixových operačních systémů, kde se tímto pojmem označovaly škodlivé nástroje umožňující získání přístupu do systému s právy root uživatele. V současné době je ale jeho význam mírně posunutý. Pro jeho vysvětlení si nejprve uveďme dvě definice od expertů, kteří se tímto tématem zabývají do hloubky. Jsou jimi Greg Hoglund (autor [2]) a Mark Russinovich (spoluautor [3] a [4]):

„Rootkit je soubor programů a kódu, který umožňuje permanentní či konzistentní, nedetekovatelnou přítomnost na počítači.“

Greg Hoglund

„Software, který skrývá sebe či jiné objekty, jako jsou soubory, procesy a registry před standartním diagnostickým, administrativním a bezpečnostním softwarem.“

Mark Russinovich

Jak z těchto definic vyplývá, hlavním cílem je poskytnout autorovi rootkitu nedetekovatelný přístup k systémovým prostředkům počítače. Tyto prostředky může posléze využít za účelem kontroly či sledování.

Typickými úkoly rootkitů je:

- Blokování volání určitých aplikací (například bezpečnostní software)
- Nahrazení systémových volání vlastními funkcemi
- Sledování systémových volání
- Filtrování výstupních parametrů systémových volání
- Získání výpočetního výkonu pro svoje potřeby

Rootkity nejsou a priory škodlivým softwarem. Příkladem může být například firmwarový rootkit CompuTrace[1], který umožňuje sledování odcizeného mobilního zařízení a jeho navrácení. Avšak jejich škodlivé využití převažuje.

Vývoj rootkitů je velmi svázaný s cílovým operačním systémem a jeho datovými strukturami. Proto je velmi obtížné vytvořit multiplatformní rootkit, který by fungoval například na operačních systémech Windows 7 a Linux zároveň. Určité obecnosti můžeme dosáhnout v případě, že cílové systémy jsou ze stejné rodiny. Pro účely této práce se budeme, pokud nebude uvedeno jinak, zabývat rootkity pro operační systém Windows 7 v 32bitové verzi, který patří mezi jedny z nejrozšířenějších na trhu.

Historicky můžeme rootkity rozdělit na několik generací. Rootkity první generace byly velice primitivním programy umožňující vzdálený přístup k napadenému počítači, které kryly svoji přítomnost upravenou verzí systémových příkazů, jako jsou „*ls*“ v linuxových distribucích nebo „*dir*“ v operačních systémech Windows. Díky zlepšení kontroly integrity systému se však staly snadno detekovatelnými. Druhá generace rootkitů přesunula svoji přítomnost do režimu jádra, kde pomocí hákování (metoda rootkitů popsaná dále v této práci) upravovala existující komponenty operačního systému a vykonávání systémových volání. Rootkity třetí generace začaly používat techniky přímé modifikace objektů jádra (DKOM), která je opět popsaná dále v této práci. Mezi další generace patří rootkity, které se přesunuly mimo meze operačního systému a nacházející se například ve firmwaru zařízení či v některém z dalších módů procesoru. V této práci se budeme zabývat rootkity druhé a třetí generace.

2.2 Registry

V průběhu této práce se budeme setkávat s množstvím registrů, a proto si je zde stručně popíšeme. V této budeme používat zejména architekturu IA-32 od firmy Intel, což je 32bitová architektura s CISC instrukční sadou. Používané registry můžeme rozdělit do několika kategorií:

- Univerzální registry
- Indexové registry
- Segmentové registry
- Registr příznaků EFLAGS
- Kontrolní registry
- Registry tabulek deskriptorů
- Registr tabulky deskriptorů přerušení

Univerzální registry EAX (accumulator), EBX (base), ECX (counter), EDX (data) můžeme použít pro účely dočasných proměnných. Zároveň má každý z těchto registrů svoji vlastní specifickou funkci. K těmto registrům můžeme přistupovat i po částech.

[1] http://www.absolute.com/en/products/absolute-computrace

Mezi indexové registry patří ESI (source index), EDI (destination index), EBP (base pointer), ESP (stack pointer) a EIP (instruction pointer). Každý z těchto registrů má svoji specifickou funkci. ESI a EDI slouží jako ukazatel na zdroj a na cíl. EBP slouží k uložení vrcholu zásobníku při volání funkce. ESP je ukazatelem na vrchol zásobníku. Posledním registrem je EIP, který slouží jako ukazatel na právě prováděnou instrukci strojového jazyka. Tento registr je také zvláštní tím, že k jako jedinému z této skupiny nemůžeme přistupovat přímo.

Segmentové registry CS (code segment), DS (data segment), ES (extra segment), FS, GS a SS (stack segment) obsahují selektory segmentu, které se používají při výpočtu skutečné adresy, jak si ukážeme v následující kapitole, která je zaměřená na paměť. Pro CS registr také platí, že nemůže být přímo upraven – pro jeho změnu slouží instrukce pro změnu kontroly programu (JMP, CALL, INT, SYSENTER a další).

Registr příznaků EFLAGS obsahuje jednobitové příznaky, které reflektují stav procesoru a úspěšnost prováděných instrukcí. Pro potřeby této práce nás budou zajímat zejména příznaky TF (bit 8, trap-flag) a IF (bit 9, interrupt enable flag)

Velmi důležitými registry je pět kontrolních registrů CR0, CR1, CR2, CR3 a CR4. S některými z nich se budeme setkávat v průběhu této práce velmi často, a proto si je popíšeme detailněji, konkrétně se jedná zejména o registry CR0 a CR3. V registru CR0 nás budou zajímat zejména příznaky WP (bit 16, write protection) a PG (bit 31, paging enable). V případě, že první zmíněný příznak WP je nastavený, kód v módu supervizora nemůže zapisovat do uživatelských stránek v paměti (více o tom bude popsáno v kapitole věnující se implementaci ochrany paměti založené na stránkování). Druhý zmíněný příznak PG zapíná a vypíná stránkování, které si popíšeme později. CR3 registr se používá v rámci stránkování, protože je v něm obsažena bázová adresa struktury potřebné pro výpočet fyzické adresy.

Registry tabulek deskriptorů obsahují informace, konkrétně bázovou adresu a velikost, o tabulkách deskriptorů GDTR obsahuje informace o globální tabulce deskriptorů, LDTR obsahuje informace o lokální tabulce deskriptorů. Tyto registry budou podrobněji popsány v kapitole zabývající se segmentací paměti.

Posledním zmíněným registrem je IDTR, což je registr tabulky deskriptorů přerušení[2].

2.3 Paměť

V současných operačních systémech se nepřistupuje přímo k fyzické paměti počítače, ale používá se paměť virtuální, která je na tu fyzickou mapována. Díky tomuto je dosaženo zvýšení bezpečnosti oddělením běžících procesů, umožnění snadnější sdílení paměti procesy či vyšší míru abstrakce. Velikost virtuálního adresového prostory závisí

[2] Anglicky „Interrupt Descriptor Table Register“

na systému. 32bitové systémy využívají prostoru o velikosti čtyř gigabytů (2^32B). Naproti tomu 64bitové systémy mohou teoreticky využít až prostor o velikost šestnácti exabytů (2^32B), avšak ve skutečnosti využívají pouze část. Celkový rozsah paměti označujeme jako lineární adresový prostor.

Virtuální adresový prostor bývá rozdělen do dvou částí – uživatelského a systémového prostoru. Uživatelský prostor je vždy specifický pro daný běžící proces, čímž je zaručeno, že procesy nemají přístup k paměti jiných procesů. Zároveň nemohou přímo přistupovat k paměti ze systémové části. Na rozdíl od vícečetných uživatelských prostorů, systémový prostor je společný pro všechny běžící procesy.

Existuje několik způsobů ochrany paměti, které jsou založeny na mechanismech segmentace a stránkování, které si popíšeme dále v dalších podkapitolách. Na závěr si ukážeme, jak je těchto mechanismů použito pro implementaci ochrany paměti. Pokud nebude uvedeno jinak, budeme mluvit o 32bitových systémech.

2.3.1 Segmentace

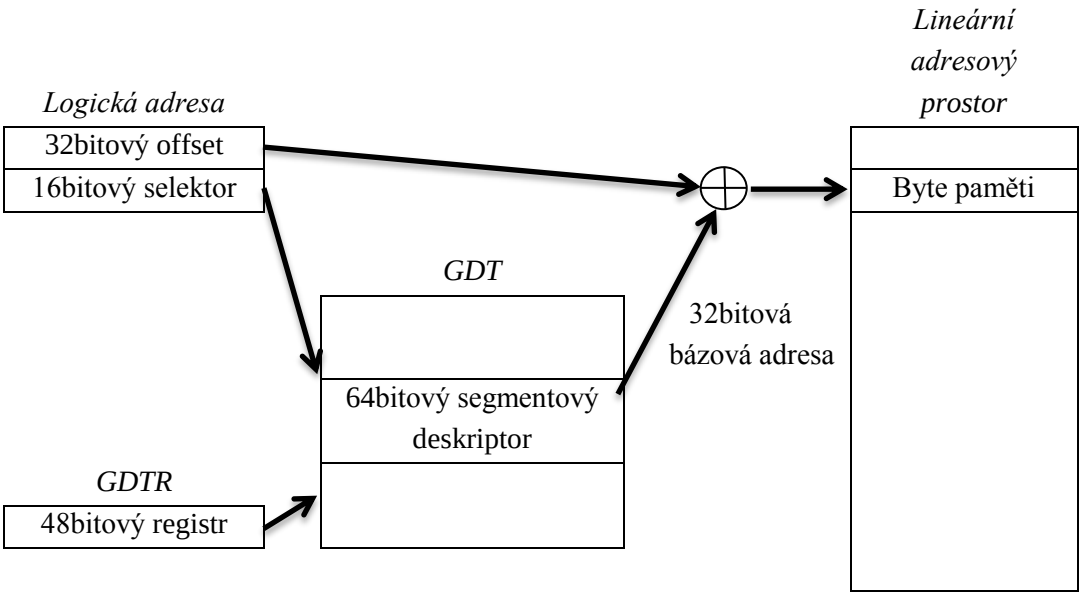
Segmentace je povinná metoda ochrany paměti všech IA-32 procesorů v chráněném režimu. Jak název této metody napovídá, paměť je rozdělena do několika segmentů.

Jakmile chceme přistoupit někam do paměti, začínáme s osmibytovou logickou adresou. Abychom získali adresu do lineárního adresového prostoru, musíme ji nejdříve vypočítat právě z této logické adresy skládající se ze dvou částí. První částí je 32bitový offset a druhou částí je 16bitový selektor použitý pro přístup do tabulky deskriptorů. Těchto šestnáct bitů je rozděleno na tři sekce. První část, zabírající bity 0 a 1, určuje úroveň oprávnění segmentu. Bit 2 udává typ tabulky selektorů. Zbylých třináct bitů obsahuje ukazatel do tabulky deskriptorů. V této tabulce deskriptorů získáme 32bitovou bázovou lineární adresu, kterou sečteme s offsetem, čímž získáme finální adresu. Toto je zobrazeno na schématu níže (viz Obrázek 1 - Schéma segmentace).

Jak již bylo naznačeno při popisu selektoru, existují dva typy tabulek deskriptorů – globální a lokální. Globální tabulka deskriptorů[3] je povinná a existuje v systému právě jedna. Informace o ní jsou uloženy v 48bitovém registru GDTR, který obsahuje její velikost (nižších šestnáct bitů) a bázovou adresu (zbylých třicet dva bitů). Naproti tomu lokální tabulka deskriptorů je pouze volitelná a informace o ní jsou uloženy v 16bitovém registru LGDT. Pro účely této práce budeme dále pracovat pouze s Globální tabulkou deskriptorů.

Se zmíněným registrem GDTR jsou spojeny dvě speciální instrukce LGDT a SGDT. První instrukce nahraje hodnotu do registru GDTR, instrukce SGDT naopak hodnotu z registru uloží.

[3] Anglicky „Global Descriptor Table“



Obrázek 1 - Schéma segmentace

Z hlediska ochrany paměti je velmi zajímavý 64bitový segmentový deskriptor uložený v tabulce deskriptorů. Struktura, popisující tento záznam obsahuje velké množství polí, ale pro naše účely je důležitých pouze následujících pět - *bázová adresa* (bity 31 až 16), *limit segmentu* (bity 19 až 16 a 15 až 0), *typ* (bity 11 až 8), *příznak S* (bit 12) a *DPL*[4] (bity 14 až 13). S bázovou adresou jsme se již setkali, ostatní si popíšeme až později v sekci zaměřené na implementaci ochrany paměti.

2.3.2 Stránkování

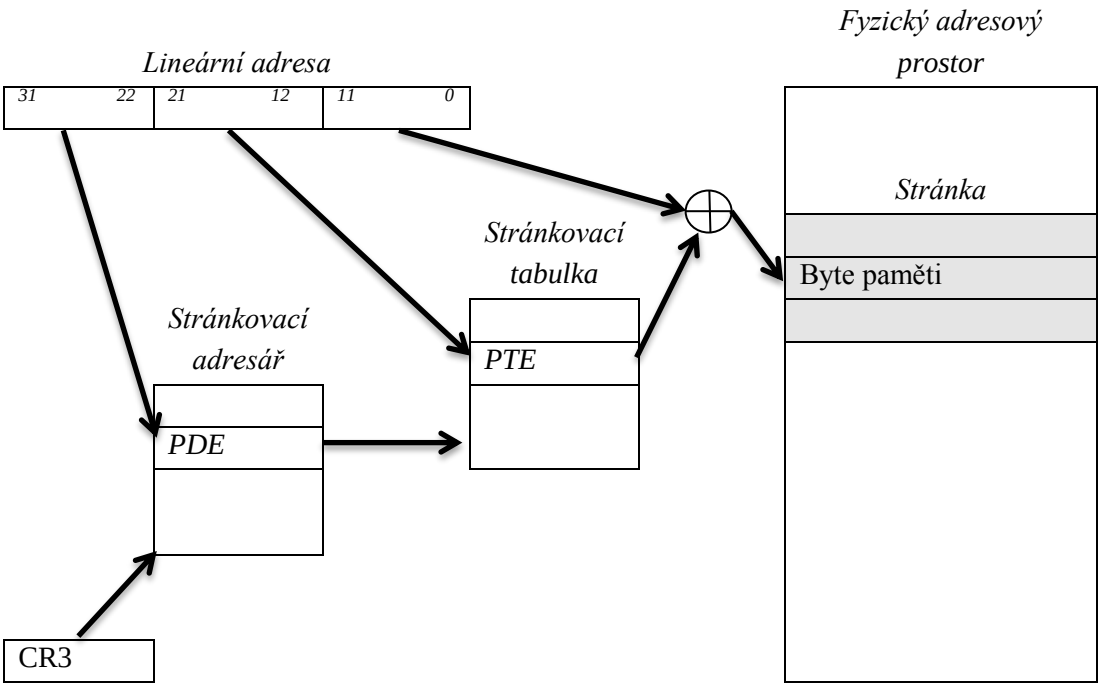
V předchozí části jsme pracovali s případem, kdy lineární adresový prostor je přímo mapovaný na fyzickou paměť. To má ale jednu velkou nevýhodu – jsme omezeni právě velikostí fyzické paměti. V případě 32bitových systému bychom byli schopni adresovat pouze 4GB. Abychom toto omezení obešli a zároveň ještě zlepšili ochranu paměti, můžeme využít stránkování.

Při využití této metody je lineární adresový prostor rozdělen na části o pevně dané velikosti (mohou nabývat velikosti 4KB, 2MB či 4MB), které se nazývají stránky. Tyto stránky mohou být uloženy jak ve fyzické paměti, tak na disku. Ve chvíli, kdy nějaký proces chce přistoupit ke stránce, která je uložena na disku, vygeneruje se systémové přerušení typu „výpadek stránky“. Operační systém na ně zareaguje tím, že danou stránku nahraje zpět do paměti, aby byla přístupná.

Nevýhodou této metody je složitější adresování, které si nyní popíšeme. Stránkování rozšiřuje segmentaci paměti o další adresaci a výpočty. Základem je lineární

[4] Zkratka z anglického „Descriptor Privilege Level“

adresa získaná pomocí segmentace. Tato adresa je rozdělena na tři části o velikosti 10, 10 a 12 bitů. Prvních deset bitů (bity 31 až 22) obsahuje ukazatel na záznam ve stránkovacím adresáři, což je tabulka, jejíž fyzická adresa je uložena v kontrolním registru CR3. Tento záznam obsahuje bázovou fyzickou adresu další tabulky, která se nazývá tabulka stránek. Právě do této tabulky ukazuje index obsažený v dalších deseti bitech lineární adresy (bity 21 až 11). Záznam v této tabulce již obsahuje fyzickou adresu hledané stránky, proto ji sečteme s offsetem, který se nachází v posledních dvanácti bitech (bity 11 až 0) lineární adresy, čímž získáme adresu do fyzického adresového prostoru. Celý postup výpočtu je zobrazen opět na schématu níže (viz Obrázek 2 - Schéma stránkování):



Obrázek 2 - Schéma stránkování

Pomocí popsaného modelu můžeme na 32bitových systémech adresovat stále maximálně pouze 4GB fyzické paměti. Abychom mohli adresovat více, můžeme použít PAE[5] mechanismus. Informace, zda je tato varianta stránkování zapnuta se nachází v registru CR4. Rozdíl oproti klasické variantě spočívá v tom, že lineární adresa je rozdělena na čtyři části po 2, 9, 9 a 12 bitech a využívá navíc tabulku ukazatelů na stránkovací adresář. Zatímco v předchozí variantě registr CR3 obsahoval bázovou fyzickou adresu stránkovacího adresáře, nyní obsahuje adresu zmíněné nové tabulky. Teprve záznam v ní obsahuje adresu stránkovacího adresáře. Pomocí PAE mechanismu jsme schopni namapovat 32bitovou lineární adresu na 52bitovou fyzickou adresu.

[5] Z anglického „Paging with Address Extension“

2.3.3 Implementace ochrany paměti

Techniky ochrany paměti jsou implementovány za použití segmentace a stránkování, které jsme si popsali v předchozích částích této kapitoly.

Ochrana paměti na základě segmentace je založená na následujících čtyřech kontrolách:

- Kontrola limitu
- Kontrola typu segmentu
- Kontrola úrovně přístupu
- Kontrola omezených instrukcí

Pro první zmíněnou kontrolu se používá pole *Limit segmentu*, o kterém jsme se již zmínili. Zaručuje, že nepřistupujeme k paměti, která neexistuje. Zároveň procesor využívá velikosti tabulky deskriptorů uvedené v registru GDTR, za účelem kontroly, že nepřistupujeme k záznamům mimo tabulku.

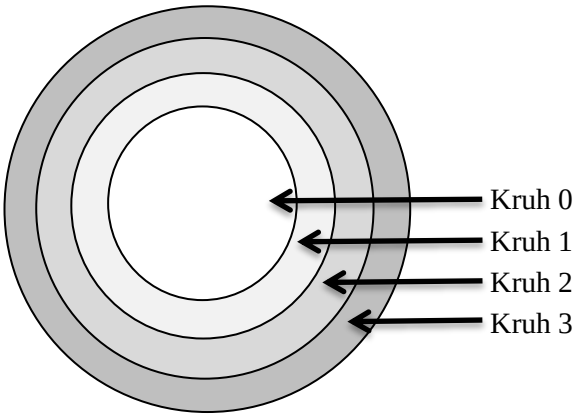
Kontrola typu segmentu využívá polí *typ* a *příznak S* opět ze segmentového deskriptoru uloženého v tabulce deskriptorů. Tyto dvě pole se používají pro určení typu daného deskriptoru. *Příznak S* označuje třídu deskriptorů (1 – deskriptor kódových a datových segmentů; 0 – deskriptor systémových segmentů) a *typ* popisuje konkrétní typ v dané třídě a jeho další parametry. Cílem této kontroly je zaručit, že se program nesnaží přistoupit k tomuto segmentu nějakým nevyhovujícím způsobem. Například, že žádná instrukce se nesnaží zapisovat do kódového segmentu.

V současných systémech se používá systém přístupu obsahující čtyři úrovně přístupu, kde úroveň 0 má nejvyšší privilegia a úroveň 3 má privilegia nejnižší. Těmto úrovním se také říká kruhy[6] a obvykle se zobrazují ve formě soustředných kružnic (viz Obrázek 3 - Model úrovní přístupu). V kruhu 0 běží procesy jádra operačního systému, v kruzích 1 a 2 obvykle běží služby operačního systému a v kruhu 3 běží uživatelské aplikace. Cílem této kontroly je jednoduše řečeno zamezit, aby proces z vnějšího kruhu zasahoval do segmentů, které existují v některém z vnitřních kruhů.

Poslední zmíněná kontrola ověřuje, že se program nesnaží volat instrukce, které jsou omezeny pouze pro některou z nižších úrovní přístupu, zpravidla pro kruhu 0. Příkladem takovéto instrukce může být LGDT, o které jsme se již zmínili.

V případě, že jakákoli z těchto kontrol selže, systém vygeneruje „general protection“ výjimku, která se často označuje jako #GP. Všechny tyto kontroly jsou prováděny paralelně s výpočtem adresy, takže nedochází k snížení výkonosti systému.

[6] Anglicky „Rings“



Obrázek 3 - Model úrovní přístupu

Nyní si popíšeme, jakým způsobem lze použít stránkování pro zvýšení ochrany paměti. Ještě předtím si ovšem připomeňme, že stránkování je volitelné a ochrana na něm založená se dá jednoduše vypnout. Toho lze dosáhnout použitím kontrolního registru CR0 a vynulováním jeho WP příznaku při současném nastavení R/W a U/S příznaků v záznamech ve stránkovacím adresáři a stránkovacích tabulkách. Tímto se všechny stránky stanou zapisovatelnými a jejich úroveň přístupu se změní na uživatelskou (kruh 3).

Nicméně pokud je stránkování zapnuté, provádí se následující dvě kontroly:

- Kontrola módu stránky
- Kontrola typu stránky

Kontrola módu stránky využívá již zmíněného příznaku U/S, který je obsažen v záznamu stránkovacího adresáře i stránkovací tabulky. Pokud je nastaven na 1, označuje uživatelský mód, pokud je 0, označuje mód supervizora. V tomto módu může program přistupovat ke všem stránkám v paměti s výjimkou uživatelských stránek pouze pro čtení při nastaveném WP příznaku v registru CR0. Pokud program běží v uživatelském módu, může přistupovat pouze k dalším uživatelským stránkám a zapisovat do nich pouze když je příznak R/W nastaven. Nikdy nemůže přistupovat ke stránkám v režimu supervizora.

V případě, že se použijí metody ochrany paměti založené na segmentaci a na stránkování zároveň, ty založené na segmentaci se provedou jako první. Při neúspěchu některé z kontrol založených na stránkování se vygeneruje „page-fault“ výjimka, často označovaná jako #PF.

Bill Blunden v [1] uvádí, že v současných systémech je ochrana paměti pomocí segmentace minimalizována a spoléhá se stránkování. Této minimalizace je dosaženo vytvořením globální tabulky deskriptorů pouze se čtyřmi deskriptory segmentů - dvěma dvojicemi deskriptorů kódového a datového segmentu, kde jedna dvojice má nastaveno DPL na 0 (Kruh 0) a druhá dvojice na 3 (Kruh 3). Všechny deskriptory mají nastavenou bázovou adresu na 0x00000000 a limit velikosti segmentu na 0xFFFFFFFF. Díky tomu všechny deskriptory začínají na stejné pozici a zabírají celý adresový prostor, což odpovídá situaci bez segmentace.

2.4 Nativní API Windows

Operační systém Windows poskytuje uživatelským aplikacím přístup ke svým základním službám pomocí sady funkcí, která se nazývá „Nativní Windows API“. Nevýhodou při používání této sady funkcí je, že ne všechny jsou podrobně popsány a některé nejsou popsány vůbec. Tyto funkce jsou podpořeny speciálními strukturami, o kterých se nyní zmíníme podrobněji, protože jsou častým cílem jedné z metod používaných rootkity.

Základní strukturou, která slouží ke spojení s režimem jádra je IDT[7], což je tabulka přerušení, která se používají při použití instrukce INT (například 0x21). Windows při svém startu zkontrolují typ procesory na počítači a podle toho upraví systémová volání. Například, pokud je procesor starší než Pentium II, pro systémová volání se použije instrukce INT 0x2E vedoucí na záznam ukazující na funkci `KiSystemService`, která funguje jako rozcestník služeb systému. V novějších typech procesorů se tento způsob systémových volání nahradil voláním instrukce SYSENTER a IDT se používá spíše pro zpracování signálu generovaných hardwarem.

Jak jsme se již zmínili, v současné době je používanější instrukce SYSENTER. Při zavolání se nejprve naplní tři strojově specifické registry (MSR[8]) IA32_SYSENTER_CS, IAS32_SYSENTER_ESP a IA32_SYSENTER_EIP a z těchto registrů se zjistí, jednak kam se má skočit a také jaká je pozice zásobníku v režimu jádra. Konkrétně se skončí na funkci `KiFastCallEntry`, jejíž adresa bude uložena v registru IA32_SYSENTER_EIP.

Před zavoláním výše zmíněných instrukcí se vždy do registru EAX uloží číslo systémové služby a teprve posléze se instrukce zavolá. To v případě obou instrukcí vede k zavolání rozcestníku k jednotlivým funkcím jádra operačního systému. Uložené 32bitové číslo systémové služby se použije k nalezení záznamu v jedné ze čtyř tabulek deskriptorů služeb. Která tabulka se má zavolat se zjistí na základě bitů 12 a 13. Konkrétně se používají dvě tabulky. První je `KeServiceDescriptorTable` (zmíněné bity 12 a 13 obsahují hodnotu 0x00), která je exportována pomocí ntoskrnl.exe a druhá je `KeServiceDescriptorTableShadow` (bity obsahují hodnotu 0x01), která je viditelná pouze zevnitř spustitelného souboru. Tyto tabulky dále obsahují vnořené struktury nazývané tabulky systémových služeb:

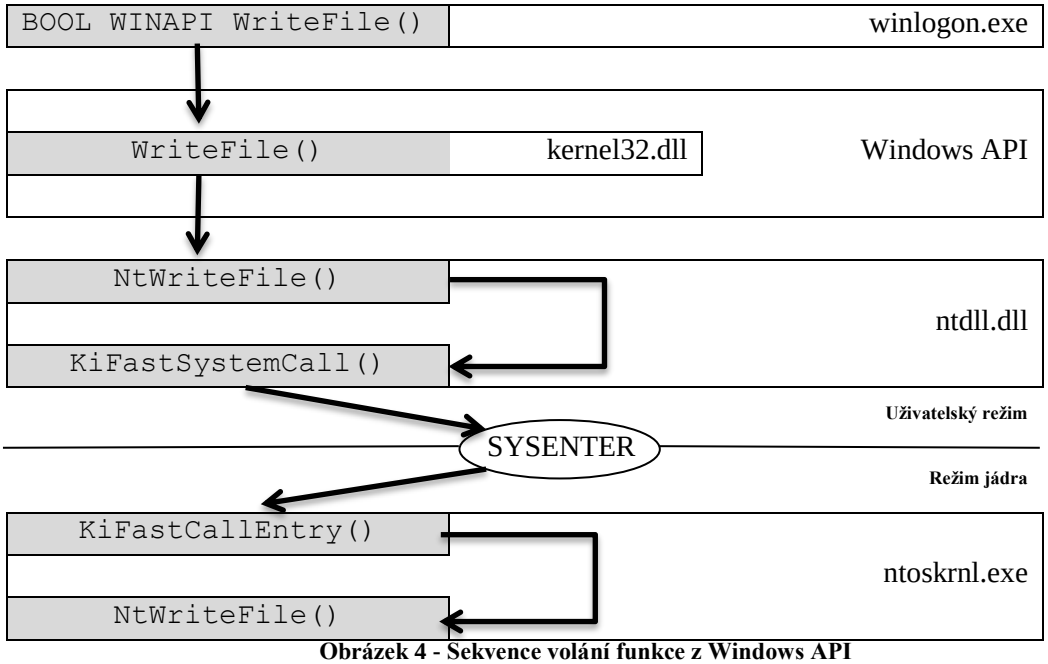
```
typedef struct _SYSTEM_SERVICE_TABLE
{
      PDWORD serviceTable;
      PDWORD field2;
```

[7] Z anglického „Interrupt Dispatch Table“
[8] Z anglického „Machine-Specific Registers“

```
        DWORD nEntries;
        PBYTE argumentTable;
}
```

Proměnná *serviceTable* je ukazatel na první položku pole lineárních adres funkcí režimu jádra. Toto pole se jinak nazývá SSDT[9]. Druhý element struktury se obvykle nepoužívá. *nEntries* udává počet položek v SSDT a *argumentTable* je ukazatel do pole bytů, které obsahuje množství bytů alokovaných pro parametry jednotlivých funkcí v SSDT.

Na závěr této sekce se ukážeme, jak konkrétně probíhá zavolání funkce ze systémového prostoru pomocí instrukce SYSENTER. Následující příklad je převzat z knihy [1] a je zobrazen níže (viz Obrázek 4 - Sekvence volání funkce z Windows API).



Obrázek 4 - Sekvence volání funkce z Windows API

Mějme aplikaci Winlogon.exe běžící v uživatelském prostoru. Ta v rámci svého kódu volá funkci `WriteFile()`, která je implementována v kernel32.dll. Tato implementace volá funkci `NtWriteFile()` z ntdll.dll. V tuto chvíli jsme stále ještě v uživatelském prostoru. Jakmile ale poslední zmíněná funkce zavolá bránu `KiFastSystemCall()` v ntdll.dll, provede se instrukce SYSENTER a vykonávání se přesune do systémového prostoru do již zmíněné funkce `KiFastCallEntry()`, která zavolá příslušnou systémovou implementaci funkce `NtWriteFile()`.

[9] Z anglického „System Service Dispatch Table“

2.5 Systémové ovladače

Základním účelem systémových ovladačů je poskytnout přístup k hardwaru počítače. Nicméně kromě tohoto účelu se dají využít i pro přístup do vnitřní části operačního systému. Toho je hojně využíváno útočníky, kteří mohou díky rootkitům založeným na systémových ovladačích, napadený počítač kontrolovat a získávat z něj data. Využívá jich ale i bezpečnostní software, který díky nim může lépe kontrolovat prostředí a snáze odhalit škodlivý kód. Za účelem vývoje systémových ovladačů poskytuje Microsoft vývojové sadu WDK[10].

Příklad minimálního systémového ovladače je uveden na konci této práce v příloze (viz Příloha č. 2). Funkce `DriverEntry()` odpovídá funkci `main()` známé z uživatelských aplikací. V rámci této funkce se bude vyskytovat vlastní implementace systémového ovladače.

Systémový ovladač je v řetězci zpracování umístěn mezi správce vstupu a výstupu[11], se kterým komunikuje pomocí příkazů vstupně/výstupního řízení IOCTL[12] prostřednictvím vstupně/výstupních paketů IRP[13], a s hardwarem na druhé straně. S tím obvykle komunikuje pomocí hal.dll, což je knihovna poskytující abstraktní vrstvu pro hardware. Příloha č. 1 obsahuje strukturu `DRIVER_OBJECT`, ve které nás v tuto chvíli bude zajímat poslední člen, `MajorFunction`, který slouží k obsluze zmíněných paketů IRP, kterými jsou například:

- IRP_MJ_CLEANUP
- IPR_MJ_DEVICE_CONTROL
- IRP_MJ_QUERY_INFORMATION
- IRP_MJ_READ
- IRP_MJ_WRITE
- IRP_MJ_CREATE

Pro každou z těchto funkcí, které budeme chtít v používat, vytvoříme uvnitř našeho ovladače funkci, která ji bude obsluhovat a ve funkci `DriverEntry()` ji přiřadíme do pole `MajorFunction`.

```
NTSTATUS ReadFunction(IN PDEVICE_OBJECT DeviceObject, IN PIRP Irp)
{
   DbgPrint("Read Function called");
   return STATUS_SUCCESS;
}
```

[10] Zkratka z anglického „Windows Driver Kit“
[11] Anglicky „I/O Manager“
[12] Zkratka z anglického „I/O Control“
[13] Zkratka z anglického „I/O Request Packets“

```
DriverObject->MajorFunction[IRP_MJ_READ] = ReadFunction;
```

Posledním krokem, který musíme provést, aby ovladač mohl komunikovat s uživatelskou aplikací, je vytvoření manipulátoru pro tento ovladač a zaregistrovat jej.

Ve chvíli, kdy máme ovladač vytvořený, musíme jej ještě nahrát do systémového prostoru. Toho lze dosáhnout mnoha způsoby, zmiňme si stručně alespoň následující dva:

- Použitím správce řízení služeb[14]
- Zneužitím zranitelnosti systému

První zmíněná technika je nejstabilnější a zároveň nejjednodušší pro použití. Její nevýhodou je, že zanechává množství artefaktů, které je možné detekovat, proto se úplně nehodí pro reálnou implementaci rootkitů, ale spíše pouze pro vývoj. Ze všeho nejdříve ustanovíme spojení s řízením služeb pomocí funkce `OpenSCManager()`. Následně vytvoříme záznam v registrech, otevřeme námi vytvořenou službu a nastartujeme ji. Posloupnost těchto operací v kódu je zobrazena níže (u funkce `CreateService`() byly uvedeny jen první dva parametry, je jich ale více):

```
SC_HANDLE sh = OpenSCManager(NULL, NULL, SC_MANAGER_ALL_ACCESS);
SC_HANDLE rh = CreateService(sh, DriverName, ...);
rh = OpenService(sh, DriverName, SERVICE_ALL_ACCESS);
StartService(rh, 0, NULL);
```

Oproti předchozí variantě je zneužití zranitelnosti systému přímo považováno za bezpečnostní útok. Je mnohem složitější na provedení, ale zároveň snižuje možnost detekce. Využívá například zranitelností založených na přetečení zásobníku za účelem nahrání a spuštění shell-kódu. Celá problematika shell-kódů je velmi rozsáhlá a v této práci se jí nebudeme zabývat.

2.6 Kernel patch protection

Tato podkapitola je vypracována na základě [7]. Kernel patch protection je technologie ochrany jádra 64bitových operačních systémů Windows, známá též jako PatchGuard. Poprvé byla představena ve Windows Server 2003 a Windows XP. Tato technologie zajišťuje, že nedochází k neoprávněné úpravě jádra systému pomocí nedokumentovaných způsobů prostřednictvím systémových ovladačů. V případě takové neoprávněné modifikace dojde k vypnutí systému. Díky tomu je zajištěna integrita systému a tudíž zvýšení jeho bezpečnosti a stability. Mezi touto technologií blokované akce patří modifikace tabulek systémových služeb (například dříve zmíněná tabulka `KeServiceDescriptor`), modifikace tabulky deskriptorů přerušení IDT a tabulky

[14] Anglicky „Service Control Manager“

globální deskriptorů GDT, používání zásobníků v režimu jádra, které nejsou alokovány jádrem, a modifikace kódu v jádře.

V určitých případech je ale potřebné takovouto modifikace provést. Za tímto účelem společnost Microsoft zavedla systém podepisování ovladačů. Tyto ovladače procházejí testováním a schvalovacím procesem přímo v Microsoftu a posléze jsou podepsány, což umožní jejich instalaci.

Vzhledem k tomu, jak výhodná tato technologie je pro ochranu systému, vyvstává otázka, proč je přítomna pouze v 64bitových verzích Windows a není obsažena i v 32bitových verzích. Důvodem je kompatibilita s existujícím softwarem. Velké množství 32bitových programů totiž používalo modifikaci jádra pro svoje účely. Pokud by tedy tato technologie byla zavedena, nastaly by velké komplikace s nefunkčností existujícího softwaru. V době, kdy byla tato technologie představena, však ještě neexistovalo mnoho programů pro 64bitové systémy a tudíž bylo bezpečné ji pro tyto systémy implementovat.

2.7 Forenzní analýza

Forenzní analýza je investigativní postup, který je používán za účelem vyšetřování. V souvislosti s rootkity se budeme bavit o forenzní analýze digitálních dat.

S touto analýzou souvisí výskyt dvou druhů chyb – falešně pozitivní a falešně negativní chyby. Prvním termínem (ve statistice se používá označení chyba typu I) označujeme výsledek, kdy data označíme jako škodlivá či podezřelá, ačkoli jsou v pořádku, což může být při práci detektoru rootkitů nebezpečné. Druhý termín (ve statistice chyba typu II) naproti tomu znamená, že jsme něco přehlédli a škodlivá data jsme nerozpoznali.

2.7.1 Anti-forenzní techniky

Při implementaci detektoru se budeme setkávat s rozdílnými technikami, jak rootkity ovlivňují data na počítači. Cílem těchto obecných anti-forenzních technik je dosažení stavu, kdy analýza je příliš nákladná a zdlouhavá a tudíž se v ní již nevyplatí pokračovat.

Obecně tohoto můžeme dosáhnout třemi způsoby. Prvním z nich je snaha o minimalizaci počtu detekovatelných artefaktů, čehož docílíme eliminací datových zdrojů či jejich zničením. Ve chvíli, kdy nějaké artefakty musíme ponechat, přichází na řadu druhý způsob, při kterém se snažíme, aby tyto artefakty byly dobře utajené a případně pomocí různých transformací těžce čitelné. Třetím způsobem je vytvoření falešných artefaktů, které mohou průběh analýzy ovlivnit a odvést pozornost od vlastního

škodlivého kódu. Tyto techniky jsou navíc velmi často kombinovány, aby se dosáhlo co největší účinnosti.

2.7.1.1 Poškození dat

V průběhu bezpečnostního útoku je mnohdy potřeba vytvoření datových artefaktů, které by mohly být pomocí forenzní analýzy snadno zjistitelné. Proto je velmi často používanou technikou právě ničení dat - ve chvíli, kdy daný artefakt již není potřeba, se jej útočník jednoduše zbaví.

V extrémním případě spadá do této anti-forenzní strategie i přímo poškození souborového systému. Ovšem vzhledem k tomu, že cílem rootkitů je být nenápadný a co nejdéle se vyhýbat detekci, je tato možnost vysoce nepravděpodobná.

2.7.1.2 Utajení dat

Utajení dat je strategie, pomocí které se snižuje pravděpodobnost nálezu artefaktů používaných rootkitem. Existují dvě varianty – pasivní a aktivní.

V rámci pasivního utajení jsou data umístěna v místech, která nemusí být na první pohled viditelná, a tudíž je náročnější zde něco najít. Jedná se například o „System Volume Information“, což je systémový adresář, do kterého má přístup pouze systém a který uchovává například data pro případné obnovení systému.

Aktivním utajením rozumíme techniku, pomocí které rootkit po svém spuštění pozmění operační systém tak, aby nebyl viditelný. Toho je dosaženo například pomocí skrytí procesu či vlákna.

2.7.1.3 Transformace dat

Transformací dat souhrnně označujeme techniky, pomocí kterých měníme podobu uložených dat tak, aby bylo náročnější rozpoznat jejich význam a funkci. Řadíme sem například steganografii, kdy schováváme důležitou informaci mezi jinými daty, či různé druhy šifrování dat.

Častou metodou, kterou rootkity používají je tzv. „Obrnění“, což je forma datové transformace umožňující obejít detekci známých vzorů. Toho je docíleno tím, že rootkit je zaveden jako zašifrovaný spustitelný soubor, který se rozšifruje až za svého běhu.

2.7.1.4 Eliminace datových zdrojů

V rámci předchozích dvou strategií jsme si uvedli, jak skrýt a zamaskovat artefakty, které jsou součástí bezpečnostního útoku. Na opačný konec spektra anti-forenzních strategií patří eliminace datových zdrojů. Než aby se útočník pokoušel tyto

stopy skrýt, je účinnější množství takovýchto artefaktů snížit. Toho je dosaženo použitím netriviálním metod pro vytvoření rootkitu.

2.7.1.5 Tvorba falešných dat

Pro případ, že by mělo dojít k detekci rootkitu, je možné vytvořit falešná data, která zvýší množství falešně pozitivních nálezů a pokusí se odvrátit pozornost směrem od rootkitu. Nevýhodou ovšem je, že vysoký počet takovýchto nálezů sám o sobě napovídá, že je něco v nepořádku.

3 Techniky rootkitů

Technik, které rootkity používají je velké množství, ale mezi ty nejrozšířenější patří hákování, modifikace kódu za běhu programu a přímá modifikace systémových objektů, pro kterou se používá zkratka DKOM[15]. Tyto techniky si v následujících podkapitolách popíšeme. Tato kapitola je napsána na základě [1], [2].

3.1 Hákování

Začneme jednou z jednodušších metod, pomocí kterých můžeme rootkit implementovat – hákováním. Cílem této metody je nahrazení adresy v některé ze systémových tabulek adresou naší funkce rootkitu. Tyto tabulky můžeme nalézt jak v uživatelském režimu, tak v režimu jádra.

Nejdůležitější tabulkou volání v uživatelském režimu je takzvaná IAT. To je zkratka z anglického „Import Address Table“, tj. „Tabulka importovaných adres“. Naprostá většina běžících aplikací má jednu a více těchto tabulek, které obsahují adresy funkcí importovaných z dynamicky linkovaných knihoven použitých danou aplikací.

Nevýhodou hákování v uživatelském režimu ale je jeho snadné odhalení. Z tohoto důvodu je výhodnější zahákovat nějakou ze struktur nacházejících se v systémovém prostoru. Zároveň zde máme větší počet možností volby. Patří mezi ně například následující struktury, o kterých jsme se již zmínili dříve:

- IDT
- MSR
- SSDT
- IRP

U prvních tří musíme počítat s faktem, že jsou specifické pro každý procesor v počítači, což znamená, že počítače s větším počtem procesorů budou mít vícečetné instance těchto tabulek. Proto pokud budeme některé z těchto struktur upravovat, musíme je upravit pro všechny procesory.

Postup hákování je z obecného hlediska u všech těchto tabulek stejný a skládá se ze čtyř kroků:

1. Identifikace cílové tabulky.
2. Uložení existujícího záznamu v tabulce.
3. Nahrazení tohoto záznamu adresou naší funkce.
4. Obnovení původního stavu.

[15] Zkratka anglického „Direct Kernel Object Modification“

Poslední krok možná vypadá z pohledu rootkitů nelogicky, ale během vývoje nám to velmi pomůže a zároveň nám to poskytne vyšší míru stability v reálném prostředí. Jakmile máme takto zahákovanou některou z uvedených tabulek, umožňuje nám to kontrolovat všechna volání původního záznamu a přesměrovat je kam potřebujeme. Tímto způsobem můžeme například monitorovat vstupní parametry zahákovaného volání, blokovat volání určitých aplikací či filtrovat výstupy.

Dále v této kapitole si popíšeme postup hákování zmíněných tabulek.

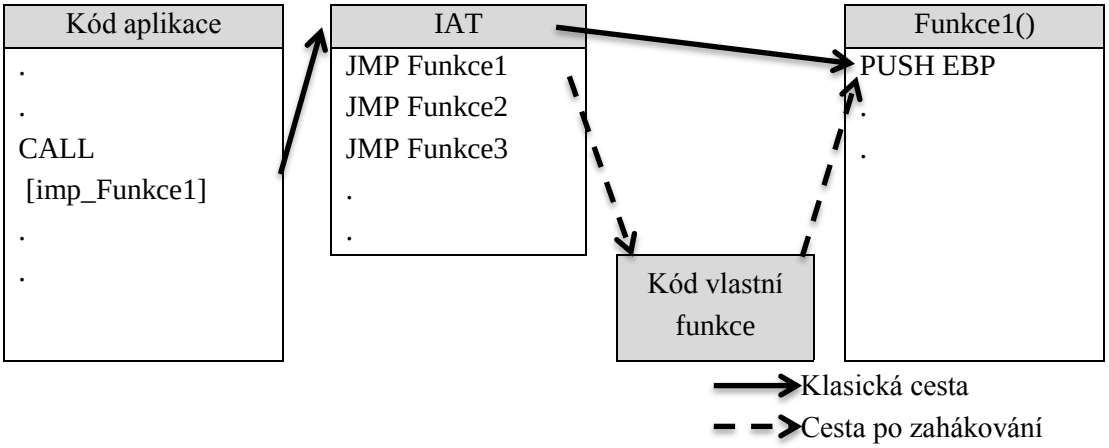
3.1.1 IAT

Pokud uživatelská aplikace chce používat funkce vyskytující se v nějaké dynamicky linkované knihovně DLL, musí tuto knihovnu nejdříve importovat a znát adresy jejích funkcí. Toho je dosaženo právě pomocí IAT. Každá importovaná DLL knihovna je obsažena v aplikaci ve struktuře jménem IMAGE_IMPORT_DESCRIPTOR, která obsahuje jméno importované knihovny a dva ukazatele na pole struktur `IMAGE_IMPORT_BY_NAME`, které obsahují jména importovaných funkcí. Jakmile je aplikace spuštěna, operační systém projde všechny importované knihovny a seznam jejich funkcí a zjistí jejich skutečné adresy. Těmito skutečnými adresami přepíše odpovídající záznamy v druhém poli `IMAGE_IMPORT_BY_NAME`.

Postup hákování IAT tabulky se skládá ze tří kroků. Nedříve je nutné získat přístup do adresového prostoru daného procesu, který obsahuje funkci, kterou chceme zahákovat. Nejjednodušším způsobem, jak toho dosáhnout, je pomocí některé z metod vsunutí DLL knihovny jako jsou využití hodnoty registru `AppInit_DLL`, volání API funkce `SetWindowsHookEx()` či použití vzdálených vláken.

Jakmile má rootkit přístup do adresového prostoru aplikace, projde všechny záznamy v IAT tabulce a najde konkrétní funkci, kterou chce zahákovat. Její adresu si uloží a posléze ji v IAT tabulce přepíše adresou vlastní funkce. Ta v rámci svého kódu může zavolat původní funkci, případně místo ní použít vlastní implementaci. Toho lze využít jak pro blokování volání, tak pro filtraci výsledků původní funkce. (viz Obrázek 5 - Sekvence volání zahákované funkce).

Jak již bylo dříve zmíněno, nevýhodou tohoto typu zahákování v uživatelském prostoru je, že je snadno odhalitelné, zejména pokud je detekční software umístěn v systémovém prostoru. Z tohoto důvodu se při hákování dává přednost strukturám v systémovém prostoru.



Obrázek 5 - Sekvence volání zahákované funkce

3.1.2 IDT

S IDT tabulkou jsme se již setkali v kapitole o nativním API systému Windows, když jsme si uváděli použití instrukce INT pro vstup do systémového prostoru skrze funkci `KiSystemService()` na adrese 0x2e v IDT. Tato tabulka sice obsahuje více deskriptorů přerušení, ale právě systémové přerušení je z nich nejzajímavější a proto se nejvíce rootkitů založených na této technice zaměřuje právě na ně.

První věcí, kterou musíme při této metodě hákování vyřešit, je umístění IDT tabulky. Toho lze snadno dosáhnout pomocí registru IDTR, o kterém jsme se zmínili již v kapitole popisující registry procesorů rodiny IA-32. Tento registr uchovává adresu tabulky přerušení. Abychom informace z něj dostali, zavoláme instrukci SIDT a jako parametr ji předáme ukazatel na alokovanou paměť o velikost 6B, protože registr IDTR má sám velikost 6B. Jak již bylo ale naznačeno dříve, každý z procesorů má svůj vlastní registr IDTR. Kvůli tomu musíme tento proces opakovat pro všechny procesory, jinak by výsledná funkce nebyla konzistentní.

Dále už provedeme vlastní nahrazení adresy, uložené v IDT tabulce naší vlastní adresou. V rámci naší vlastní obsluhy přerušení můžeme provést detekci či blokaci volání přerušení a poté případně zavolat původní funkci pro obsluhu přerušení. Oproti předchozí variantě hákování, se do této funkce ovšem vykonávání nevrací a proto nemůžeme provést filtraci výstupních parametrů.

Jak jsme si již dříve uvedli, tato instrukce se již tolik nepoužívá, což má důsledek i pro rootkity založené na technice hákování IDT – zahákovat tuto tabulku můžeme, ale nebude to mít až takový efekt, protože se naše funkce nebude volat tak často.

3.1.3 MSR

Nevýhoda zmíněná na konci předchozí podkapitoly o hákování IDT může být snadno vyřešena hákováním strojově specifických registrů MSR, které se používají při použití instrukce SYSENTER. Jak již bylo zmíněno dříve, tyto registry jsou tři:

- IA32_SYSENTER_CS
- IA32_SYSENTER_ESP
- IA32_SYSENTER_EIP

Při zavolání instrukce SYSENTER, se obsah registru IA32_SYSENTER_CS nahraje do registru CS, obsah registru IA32_SYSENTER_EIP se nahraje do registru EIP, obsah IA32_SYSENTER_CS + 8 se nahraje do registru SS a obsah registru IA32_SYSENTER_ESP se nahraje do registru ESP. Poté se úroveň provádění přepne do kruhu 0 a začne se provádět kód na adrese specifikované pomocí CS:EIP. Vzhledem k tomu, že se v rámci Windows všechny segmenty mapují do celého paměťového prostoru, stačí nám pro potřeby hákování změnit registr IA32_SYSENTER_EIP.

Při hákování MSR si tedy nejprve načteme obsah zmíněné registru, který obsahuje adresu funkce `KiFastCallEntry()` a uložíme si jej na později. Dále změníme offset adresy obsažené v tomto registru tak, aby ukazoval na naši funkci, ve které kromě našeho vlastního kódu můžeme zavolat původní funkci. Stejně jako u předchozí varianty zde platí, že nemůžeme filtrovat výstupní parametry a že zahákování musíme provést na všech procesorech počítače, abychom se vyhnuli nestabilnímu chování.

3.1.4 SSDT

SSDT jsme si již popsali v části o nativním API Windows, připomeňme si jen, že se jedná o tabulku obsahující adresy všech funkcí dostupných v režimu jádra, které jsou exportovány pomocí ntoskrnl.exe. Tyto adresy budou cílem hákování SSDT.

Než ale budeme moci tyto adresy začít měnit, musíme obejít jedno bezpečnostní opatření – SSDT sídlí v paměti pouze pro čtení, takže budeme muset nejdříve tuto ochranu vypnout. Hlavním prvkem této ochrany je WP bit (bit 17) v kontrolním registru CR0. Pokud je nastavený, ochrana závisí na příznaku R/W ve stránkovacích adresářích a tabulkách. Pokud je ovšem vynulovaný, přístup pro čtení i zápis je povolen. Proto jediné, co potřebujeme udělat je tento bit vynulovat, což lze udělat velmi jednoduše pomocí základních instrukcí assembleru – do nějaké univerzálního registru nahrajeme pomocí instrukce MOV registr CR0, provedeme bitový součin tohoto univerzálního registru a operandu 0xFFFEFFFF (0 pouze na sedmnáctém místě) a výsledek nahrajeme opět pomocí instrukce MOV zpátky do CR0.

Jakmile je ochrana proti zápisu deaktivovaná, najdeme adresu konkrétního systémového volání, které chceme zahákovat. Na rozdíl od předchozích metod hákování

systémových struktur, SSDT tabulka je společná pro všechny procesory a proto potřebujeme při zápisu výhradní přístup. Toho docílíme zavoláním funkce `InterlockedExchange(target,newAddress)`, kde *target* je pozice funkce v SSDT tabulce a *newAddress* je adresa naší funkce. Jakmile máme tabulku zmodifikovanou, můžeme opět povolit ochranu proti zápisu.

Výhodou oproti předchozím metodám hákování v systémovém prostoru je, že v tomto případě máme možnost filtrovat výstupní parametry původní funkce, kterou zavoláme uvnitř naší funkce.

3.1.5 IRP

Poslední metodou hákování, kterou si popíšeme, je hákovaní tabulky vstupně/výstupních packetů IRP, o kterých jsme si řekli v kapitole 2.5. Ve chvíli, kdy je systémový ovladač nainstalován, je vytvořena tabulka funkcí pro obsluhu konkrétních packetů IRP, které slouží například pro zápis (IRP_MJ_WRITE), čtení (IRP_MJ_READ) a různé druhy dotazů (IRP_MJ_QUERY_INFORMATION). Tato tabulka se nachází v poli *MajorFunction* ve struktuře DRIVER_OBJECT (viz Příloha č. 1), jejíž adresa je předána hlavní funkci ovladače `DriverEntry()` (viz Příloha č. 2).

Při hákování IRP potřebujeme nejdříve získat přístup k objektu zařízení. Toho docílíme zavoláním funkce `IoGetDeviceObjectPointer()`, které specifikujeme jméno zařízení. Tento objekt obsahuje hledanou strukturu DRIVER_OBJECT, uvnitř které máme přes pole *MajorFunction* přístup k adresám obsluhy všech IRP packetů. Odtud tedy získáme adresu obsluhy packetů, kterou chceme zahákovat, a na závěr pomocí funkce `InterlockedExchange()` ji vyměníme za adresu naší vlastní funkce.

U této techniky se opět setkáváme s nemožností filtrovat výstupní parametry původní obsluhy packetů, kterou jsme zavolali uvnitř naší nové funkce, protože vykonávání se do ní nevrací.

3.2 Modifikace kódu

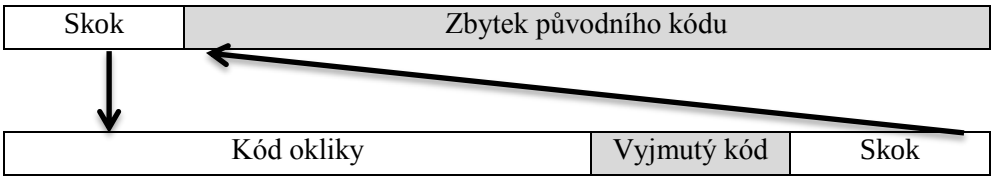
Zatímco v předchozí kapitole o hákování jsme modifikovali tabulky volání, v této části, jak její název napovídá, budeme modifikovat přímo kód existujících souborů a systémových volání. Tyto modifikace můžeme rozdělit dvěma způsoby – podle toho, kdy provádíme modifikaci a podle toho, kde provádíme modifikaci.

Podle prvního způsobu můžeme rozdělit modifikaci na modifikaci binárních souborů typu .exe, .dll a .sys, a na modifikaci za běhu. První způsob je velmi častý u bootkitů a provádí se před nahráním souboru do paměti. Druhý způsob je zajímavější,

protože modifikace probíhá až ve chvíli, kdy soubory již existují v paměti. Ve zbytku této kapitoly se budeme zaměřovat právě na modifikaci za běhu.

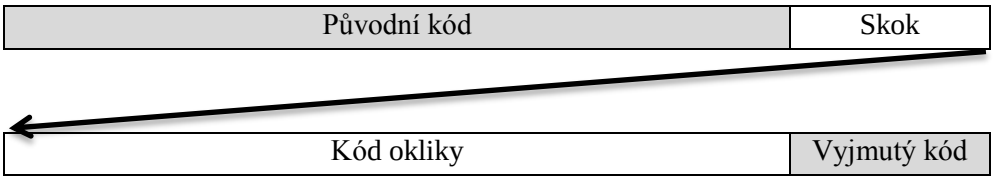
Druhý způsob modifikace záleží, kde se nachází námi modifikovaný kód. Jednou z možností je, že v paměti, kterou proces používá, pouze nahradíme část instrukcí jinými instrukcemi. Nevýhodou tohoto způsobu ale je, že jsme velmi limitování velikostí, protože námi upravený kód nesmí být delší než původní kód. Z tohoto důvodu existuje druhá možnost, kdy provádíme modifikaci pomocí tzv. okliky[16]. V rámci této metody vložíme do původního kódu příkaz skoku, kterým skočíme někam mimo, kde se nalézá naše vlastní funkce. Díky tomu už nejsme limitování množství instrukcí, které musíme použít. Protože v rámci rootkitu není cílem znemožnit fungování původního kódu, instrukce, které by se vložením našich instrukcí skoku přepsaly, vyjmeme a vložíme na konec naší vzdálené funkce.

Existují dvě varianty oklik, mezi kterými se rozhodujeme na základě účelu, jaký má kód okliky mít. Pokud potřebujeme volání funkce sledovat či blokovat, použijeme startovní okliku, kdy umístíme příkaz skoku někam na začátek původní funkce. V tomto případě se bude vzdálený kód skládat z našeho kódu okliky následovaného vyjmutým kódem a skokem zpět na první nepřepsanou instrukci původního kódu ().



Obrázek 6 - Startovní oklika

Pokud je naším cílem filtrovat výstupní parametry funkce, použijeme koncovou okliku, při které je příkaz skoku ke konci původního kódu před instrukci návratu. Vzdálený kód se pak skládá opět z našeho kódu okliky a za ním umístěného vyjmutého kódu. Zde již příkaz skoku zpět vkládat nemusíme, protože vyjmutý kód bude s největší pravděpodobností obsahovat příkaz návratu (viz Obrázek 7 - Koncová oklika). Je potřeba počítat i s variantou, že původní kód má více příkazů návratu, pak musíme koncovou okliku podle toho přizpůsobit. Nejúčinnější variantou implementace okliky je kombinace obou zmíněných variant, protože můžeme volání jak sledovat či blokovat, tak filtrovat výstupní parametry.



Obrázek 7 - Koncová oklika

[16] Anglicky „Detour patching“

3.3 DKOM

V předchozích dvou kapitolách 3.1 a 3.2 jsme si uvedli, jakým způsobem můžeme změnit relativně statická data (pomocí hákování) či kód (modifikace kódu). V jádře se ale vyskytují struktury, které jsou dynamičtější a velmi často se mění, což z nich činní obtížnější cíl, na druhou stranu poskytují vyšší míru skrytí. Tyto objekty budeme modifikovat právě pomocí v této kapitole zmíněné technice – přímou modifikací systémových objektů.

Tato technika nám neumožňuje dosáhnout všech cílů rootkitů, protože modifikovat budeme objekty, které slouží zejména k evidenci. Z tohoto důvodu bývá hlavním cílem této techniky skrytí procesů, systémových ovladačů či portů, pomocí kterých proces komunikuje po síti. Dalšími nevýhodami je, že tyto objekty jsou velmi specifické pro konkrétní verze operačního systému a že jsou obvykle málo zdokumentované.

Konkrétně se v této kapitole zaměříme na modifikaci struktury EPROCESS a DRIVER_SECTION.

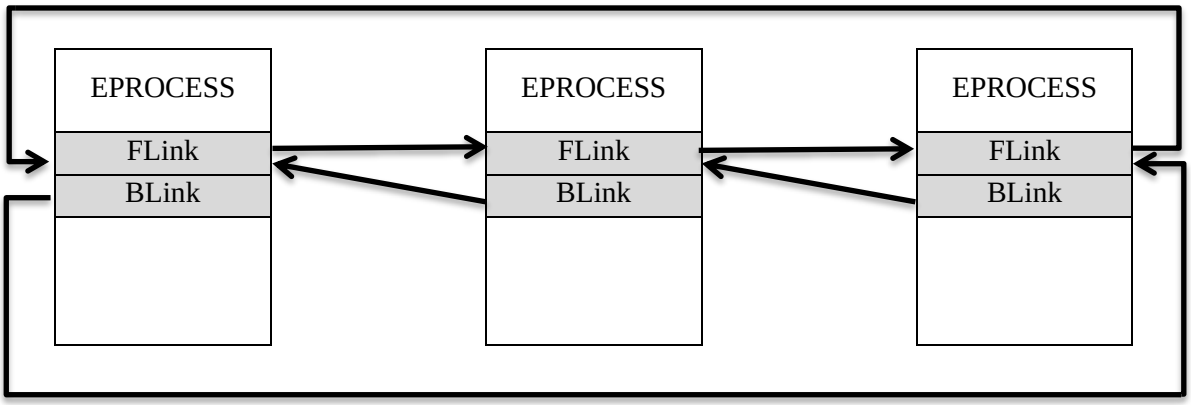
3.3.1 EPROCESS

EPROCESS je systémový objekt, který vnitřně reprezentuje proces. Abychom získali proces běžící v právě vykonávaném vlákně, zavoláme funkci režimu jádra `PsGetCurrentProcess()`, která nám vrátí ukazatel právě na strukturu EPROCESS. Tato struktura má velké množství proměnných, pro nás zajímavé jsou ale v tuto chvíli hlavně následující tři:

- UniqueProcessId
- ActiveProcessLinks
- ImageFileName

Pomocí ukazatele v *UniqueProcessId* se můžeme dostat k jedinečnému identifikačnímu číslu PID úlohy spojené s daným procesem. Pro popis procesu slouží i *ImageFileName*, což je pole šestnácti ASCII znaků, ve kterém je uloženo jméno (či prvních 16 bytů jména) binárního souboru, pomocí kterého se proces spustil.

Poslední proměnná, kterou jsme si nezmínili je proměnná *ActiveProcessLinks* typu LIST_ENTRY. Ta se vztahuje ke způsobu, jakým si Windows uchovává informaci o aktivních procesech. Všechny aktivní procesy jsou uspořádány do obousměrně zřetězeného cyklického seznamu právě pomocí dopředných a zpětných ukazatelů umístěných v této struktuře (viz Obrázek 8 - Uspořádání aktivních procesů). Jak dopředný ukazatel *FLink*, tak zpětný ukazatel *BLink* míří na první byt struktury LIST_ENTRY následující, resp. předchozí, struktury EPROCESS.



Obrázek 8 - Uspořádání aktivních procesů

Díky tomuto uspořádání můžeme relativně snadno skrýt existující proces tak, že upravíme ukazatele objektů EPROCESS sousedící s EPROCESS objektem, který chceme skrýt. Zároveň nastavíme oba ukazatele skrytého procesu tak, aby ukazovaly na strukturu LIST_ENTRY v tom samém procesu, čímž poskytneme operačnímu systému validní data a zamezíme potenciálnímu pádu systému.

3.3.2 DRIVER_SECTION

Druhým systémovým objektem, který můžeme používat pro DKOM metodu, je DRIVER_SECTION, který je operačním systémem používán pro sledování nahraných ovladačů. Tento objekt se nachází uvnitř struktury DRIVER_OBJECT (viz Příloha č.), se kterou jsme se již setkali a víme, jak ji získat. Dvacet bytů od začátku této struktury se nalézá právě ukazatel na DRIVER_SECTION objekt. Prvním proměnnou v tomto objektu je opět struktura LIST_ENTRY, pomocí které můžeme procházet všechny nahrané ovladače. Najdeme ten správný ovladač, který chceme skrýt a opět upravíme ukazatele sousedních ovladačů.

4 Příklady rootkitů

V této kapitole si představíme tři rootkity, které jsou zajímavé z pohledu použitých technik. První rootkit, Agony, představuje rootkity založené na metodě hákování SSDT. Druhý rootkit, FU, představuje rootkity založené na přímé modifikaci systémových objektů. Poslední zde zmíněný rootkit, respektive rodina rootkitů, TDSS ukazuje současný stav technologie rootkitů, kde se kombinuje velké množství metod a pokročilých technik.

4.1 Agony

Tato kapitola popisuje rootkit agony dostupný z [8]. Je to open-source proof-of-concept rootkit používající metodu hákování tabulky systémových deskriptorů služeb SSDT. Skládá se z vlastního systémového ovladače a aplikace běžící v uživatelském módu, která rootkit nahraje a umožňuje jej pomocí příkazů ovládat.

Tento rootkit umožňuje skrývat procesy, soubory, TCP a UDP porty a záznamy v registrech. Toho je docíleno zahákováním různých systémových služeb, jak je znázorněno v tabulce níže (Tabulka 1 - Přehled funkcí rootkitu Agony):

Zahákované systémové volání	**Skrývá**
`ZwQuerySystemInformation`	Procesy
`ZwQueryDirectoryFile`	Soubory a složky
`ZwEnumerateKey`	Klíče registrů
`ZeEnumerateValueKey`	Hodnoty v registrech
`ZwDeviceIoControlFile`	TCP a UDP porty

Tabulka 1 - Přehled funkcí rootkitu Agony

4.2 FU rootkit

FU je jedním z prvních rootkitů, které jsou postaveny na technice přímé modifikace systémových souborů. I přestože byl vydaný již v roce 2004, je stále zajímavý, protože množství jiných rootkitů z něj vychází a také byl použit jako součást velkého množství škodlivých programů, což z něj činní jeden z nejvíce využívaných rootkitů. Zrojový kód tohoto rootkitu je dostupný v [9].

Používá obě varianty přímé modifikace systémových objektů, které jsme zmínili třetí kapitole. Pomocí úpravy ukazatelů v dvousměrně vázaných seznamech `PsActiveProcessHead` a `PsLoadedModuleList` dokáže skrýt jak běžící procesy, tak i nahrané systémové ovladače. Navíc umožňuje upravit bezpečnostní kontext pomocí modifikace procesového tokenu. Na takto kompromitovaném systému je možné

upravovat přístupová práva kteréhokoli běžícího procesu, případně zmást bezpečnostní software.

Skládá se ze dvou souborů. Jedním z nich je aplikace fu.exe běžící v uživatelském režimu a umožňuje pomocí IOCTL zasílat parametry do systémového ovladače, který je reprezentován druhým souborem, msdirectx.sys.

4.3 TDSS

Tato kapitola vychází z informací dostupných v [10] a [11]. Poprvé se varianta TDSS rootkitu (používáme pro ni označení TDSS.TDL-1) objevila již v roce 2008, kdy byla detekována laboratoří Kaspersky pod jménem Rootkit.Win32.Clbd.a. Tento název odkazoval na jméno systémového ovladače, který poskytoval hlavní funkcionalitu tohoto rootkitu – clbdriver.sys a jemu odpovídající dynamicky linkovanou knihovnu clbdll.dll. Tato první generace TDSS rootkitu se označuje jako TDSS.TDL-1. Mezi funkce tohoto rootkitu patřilo například skrývání registrových klíčů, skrývání souborů, vkládání škodlivého kódu do systémových procesů a skrývání TCP portů. Použité metody byly relativně jednoduché a současnými detektory jsou snadno odhalitelné.

Z důvodů snadné detekce předchozí varianty TDSS rootkitu vznikla verze TDL-2, která se poprvé objevila v roce 2009. Této varianty se objevilo více verzí s vícerem různými funkcemi. Aby se ztížila analýza a možnost detekce, hlavní funkce rootkitu byly zamaskovány (například přidáním náhodných slov ze hry Hamlet od W. Shakespeara do souboru s rootkitem) a zašifrovány. Účely rootkitu zůstaly víceméně stejné jako u předchozí verze, ale bylo jich dosaženo jinými prostředky. Velmi podstatnou schopností, která byla do této verze přidána, bylo zotavení se z pokusu o odstranění. Za tímto účelem bylo použito několik vláken, která kontrolovala, zda jsou všechna zahákování rootkitu a všechny záznamy v systémových záznamech přítomny a v případě, že tomuto tak nebylo, byly znovu obnoveny.

Další verzí je TDL-3 ve které byly vylepšeny možnosti sebeobrany rootkitu, opraveny chyby předchozí verze a implementovány nové postupy jako reakce na vývoj tehdejších technologií detekce. Aby se dosáhlo nahrání rootkitu co nejdříve, byl infikován některý ze systémových ovladačů, které splňovaly požadované kritérium nahrání téměř ihned po startu operačního systému, například ovladač pro MiniPort/Port. Jakmile se infikovaný ovladač nahrál, proběhlo nahrání vlastního škodlivého kódu rootkitu do paměti. Zajímavou technikou je použití vlastního šifrovaného souborového systému pro uložení konfiguračních dat a dalších potřebných souborů.

Zatím nejnovější verzí tohoto rootkitu je TDSS.TDL-4, jejíž ranou podobu si popíšeme podrobněji. Tato verze je nově schopna infikovat i 64bitové operační systémy a obejít technologii PatchGuard a kontrolu integrity systému Windows. Stejně jako u předchozí varianty je použito vlastního zašifrovaného souborového systému pomocí algoritmu RC4, který je umístěn v posledním sektoru disku. Oproti předchozí

verzi se ovšem změnila metoda infikace, protože TDL-4 infikuje přímo hlavní bootovací záznam MBR[17]. Díky tomuto dojde k nahrání ještě před operačním systémem. Funkcionality rootkitu v MBR slouží k nalezení a nahrání komponenty ldr16 do paměti a předání jí řízení. Tato komponenta zahákuje přerušení 13h, které slouží pro diskový vstup a výstup. Navíc přehraje infikovaný MBR původní verzí, která byla zálohována ještě před infikováním. Poté je předáno řízení právě původnímu bootovacímu záznamu. Pro další nahrávání potřebuje rootkit nalézt pomocí zmíněného zahákování přerušení dynamicky linkovanou knihovnu kdcom.dll a nahradit ji vlastní komponentou ldr32 (pro 32bitové systémy) či ldr64 (pro 64bitové systémy). Ldr16 komponenta má však i další funkci. Kromě výše zmíněného ještě upraví konfigurační data pro bootování BCD[18]. BCD projde, nalezne záznam `BcdLibraryBoolean_EmsEnabled` a nahradí jej záznamem `BcdOsLoaderBoolean_WinPEMode`, čímž aktivuje WinPE mód operačního systému ve kterém neprobíhá žádná kontrola integrity kódu. Po úspěšném nahrání škodlivé komponenty se může tento mód opět deaktivovat, protože rootkit už je pevně usídlen v systému. Nahrazení kdcom.dll vlastní verzí má i výhodu v tom, že znemožní používání systémového debuggeru. V úplném závěru je do paměti nahrána komponenta rootkitu zodpovědná za skrývání infikovaných struktur a souborů. Při každém pokusu o přístup do sektoru disku, který obsahuje škodlivý kód rootkitu, jsou vrácena falešná data. Navíc je použito kontrolní vlákno, které v případě potenciální dezinfekce některé z částí rootkitu provede opětovné infikování.

Rootkity z rodiny TDSS se velmi rychle vyvíjí v reakci na nejnovější detekční postupy současných anti-virových produktů. Jejich analýza je náročná i z důvodu, že dokáží detekovat, že běží na virtuálním stroji.

[17] Z anglického „Master Boot Record“
[18] Z anglického „Boot Configuration Data“

5 Detekce a odstranění

5.1 Hákování

Hákování je metoda, která je docela snadno zjistitelná, protože vložená hodnota neodpovídá adresovému rozsahu, který by měla dodržovat. Tento rozsah je závislý na použité metodě. Vzhledem k tomu, že při hákování IAT tabulky měníme adresu, která původně ukazovala dovnitř knihovny, adresou která ukazuje mimo knihovnu, můžeme tento fakt použít pro detekci. Obdobně to platí i pro hákování IDT, MSR a SSDT, u kterých platí, že původní adresa ukazovala dovnitř modulu ntoskrnl.exe, a pro hákování IRP, kde všechny adresy ukazovaly dovnitř paměťového rozsahu daného ovladače. Přehledné shrnutí je uvedeno v následující tabulce:

IAT	Záznam leží mimo rozsah paměti daného DLL souboru
IDT	Adresa obsluhy 0x2e leží mimo modul ntoskrnl.exe
MSR	Obsah registru IA32_SYSENTER_EIP leží mimo modul ntoskrnl.exe
SSDT	Ukazatel na systémovou službu leží modul ntoskrnl.exe
IRP	Adresa obsluhy IRP packetů leží mimo rozsah paměti systémového ovladače

Tabulka 2 - Nesprávné umístění zahákovaných struktur

Pro detekci hákování tedy musíme v rámci detektoru nejprve zjistit, jaká je bázová adresa daného modulu a jeho velikost, pomocí čehož zjistíme povolený rozsah. Posléze si projdeme všechny adresy v příslušných strukturách a porovnáme je s tímto zjištěným rozsahem. V případě kontroly IDT a MSR nastává drobná komplikace, protože tyto struktury jsou specifické pro každý procesor. Má-li tedy počítač více procesorů, musíme provést kontrolu na každém z nich, jinak by výsledky nemusely být směrodatné.

V rámci detekce hákování systémových struktur použijeme systémové volání `ZwQuerySystemInformation()` pro získání seznamu všech modulů, které se aktuálně nachází v paměti. Tyto informace jsou pro každý modul uloženy ve struktuře, která obsahuje mimo jiné název modulu, jeho bázovou adresu a velikost, což je přesně to, co potřebujeme. Projdeme tedy seznam, najdeme správný modul a spočítáme rozsah.

Při detekci hákování v uživatelském prostoru nejprve získáme seznam všech dynamicky linkovaných knihoven, které aplikace používá a spočítáme jejich rozsahy. Na závěr si projdeme všechny záznamy v IAT tabulkách v těchto knihovnách a zkontrolujeme, zda všechny spadají do správného paměťového rozsahu.

Komplikace mohou nastat, pokud jsou zahákována systémová volání či API volání použitá pro získání platných rozsahů. V takovém případě se nemůžeme na získané informace spolehnout a musíme se uchýlit k manuálnímu procházení různých systémových struktur, abychom získali potřebné informace jinou cestou. Toto řešení je

mnohem komplikovanější a zvyšuje pravděpodobnost, že uděláme při programování chybu, která uvede operační systém do nestabilního stavu.

5.2 Modifikace kódu

Při detekci modifikace kódu máme dvě možnosti. První možnost vychází přímo ze způsobu, jakým je implementována oklika, druhá je naopak založena na tom, že zdrojový kód systémových volání je relativně statický.

Nejdříve si popíšeme první techniku detekce, pomocí které se dá odhalit modifikace kódu za běhu pomocí okliky. Vzhledem k tomu, že při této metodě se používá některá z instrukcí pro předání řízení programu, můžeme se pokusit tyto instrukce detekovat. V případě startovní okliky se bude pravděpodobně nacházet ze začátku zdrojového kódu, v případě koncové okliky se bude pravděpodobně nacházet někde před instrukcí návratu z funkce. Vytvořením seznamu možných případů a implementací správné heuristiky můžeme při analýze zdrojového kódu rozhodnout, zda případná instrukce předání řízení na dané místo patří, či tam byla uměle vložena. Nevýhodou této metody je, že se dá relativně snadno obejít umístěním skoku do kódu okliky hlouběji v původním kódu. Úprava heuristiky za účelem kontroly delšího úseku ale může zvýšit počet falešně pozitivních nálezů. Na co ale v tomto případě nesmíme zapomenout je fakt, že oklika nemusí být do kódu umístěna s nekalými účely ale například jako součást aktualizace operačního systému. Detekce a odstranění takovéhoto použití by uvedlo systém do nestabilního stavu. Abychom této metody detekce mohli využít, musíme být schopni rozpoznat původ a účel okliky, což je velmi netriviální problém.

Z tohoto důvodu se jeví jako výhodnější druhá technika detekce, která je založena na relativně statické podstatě kódu systémových volání a kontrolních součtech. Při prvním spuštění detektoru se pro vytipovaná systémová volání vytvoří pomocí kontrolního součtu specifická signatura, která je pak periodicky kontrolována a v případě odlišnosti se objeví výstraha. Pokud bychom navíc původní stav systémového volání uložili, můžeme jej v případě výstrahy obnovit a tím rootkit používající tuto metodu zneutralizovat.

5.3 DKOM

Detekce procesů a systémových ovladačů skrytých pomocí metody přímé modifikace systémových objektů je možná pomocí křížové kontroly. Pokud nám různé metody pro zjištění všech běžících procesů/ovladačů vrátí různé výsledky, je to známkou toho, že některá z těchto metod byla napadnutá. Tyto metody můžeme rozdělit na metody vyšší úrovně, tj. ty které spoléhají na funkce nacházející se v nativním API Windows,

a na metody nižší úrovně, které oficiální API obcházejí a informace získávají manuálně procházením systémových struktur.

Základní metodou vyšší úrovně je použití snímku operačního systému pomocí funkce `CreateToolhelp32Snapshot()` a procházení jeho procesů pomocí API funkcí `Process32First()` a `Process32Next()`. Tímto způsobem získáme seznam procesů pouze pomocí standartních systémových volání. Tento seznam pak bude porovnán se seznamy získanými pomocí metod nízké úrovně.

Z metod nižší úrovně se pro získání seznamu procesů dá využít metody procházení zřetězeného seznamu uvnitř *ActiveProcessLinks* v struktuře EPROCESS, což je základní metoda modifikace systémových objektů, která je popsána v příslušné kapitole této práce. Struktura EPROCESS nám nicméně poslouží i pro další dvě metody. Nachází se v ní proměnná *ObjectTable* typu HANDLE_TABLE, která obsahuje reference na všechny objekty, se kterými proces pracuje. Uvnitř této struktury je opět zřetězený seznam, pomocí kterého můžeme opět projít všechny existující objekty tohoto typu a zjistit, kterým procesům jsou přidruženy, čímž získáme druhý seznam. Poslední metodou, kterou si zde popíšeme je přímo procházení běžících vláken. Abychom toho dosáhli, musíme se nejprve dostat ke struktuře ETHREAD, která popisuje dané vlákno. Toho docílíme použitím proměnné *ThreadListHead* nacházející se opět v EPROCESS. Tato proměnná neukazuje přímo na začátek hledané struktury, ale na proměnnou *ThreadListEntry*. Odečtením offsetu této proměnné se však dostaneme na začátek celé struktury a můžeme ji procházet. Bude nás zajímat hlavně identifikační číslo vlákna a jemu nadřazeného procesu a opět zřetězený seznam, pomocí kterého budeme moci projít všechna vlákna, čímž bychom měli získat opět další seznam procesů.

Jakmile získáme pomocí výše zmíněných metod seznamy běžících procesů, porovnáme je. Pokud se liší, znamená to, že proběhla přímá modifikace systémových objektů.

Detekce modifikace objektu DRIVER_SECTION bude probíhat podobným způsobem – získáme seznamy nahraných ovladačů pomocí různých metod a porovnáme je.

6 Realizace detektoru

6.1 Použitý software

Windows 7 32bit

32bitové operační systémy jsou stále ještě mezi uživateli běžně rozšířené, a proto byl Windows 7 ve svojí 32bitové verzi zvolen jako systém, pro který bude detektor rootkitů implementován.

Oracle VM VirtualBox 4.2.10

Pro vývoj a testování detektoru rootkitů bude použit software Oracle VM VirtualBox, který umožňuje virtualizaci operačního systému. Zároveň umožňuje vytváření snímků pro rychlou obnovu systému v případech, kdy se vinou chyby ovladače dostane systém do nestabilního stavu.

WDK 7.1.0

Windows Driver Kit je soubor nástrojů určený pro podporu vývoje a ladění systémových ovladačů. Jeho součástí je i nástroj Checked Build Environment, který umožňuje sestavit systémový ovladač.

Code::Blocks 13.12

Protože Code::Blocks má lepší podporu vývoje v jazyce C, bylo rozhodnuto, že pro vývoj bude použit právě tento nástroj.

Sysinternals Suite

Soubor mnoha freeware diagnostických nástrojů, které nám umožní monitorovat stav systému před i po nahrání systémového ovladače našeho detektoru. Původně se jednalo o produkt společnosti Winternals Software LP, později ale byla společnost převzata přímo Microsoftem.

VirtualKD 2.8

Aby bylo možné debuggovat kód v režimu jádra, bude použit VirtualKD debugger, který je obdobou Kernel Debuggeru od Microsoftu optimalizovanou pro použití s virtualizačním softwarem.

6.2 Návrh aplikace

Detektor rootkitů se bude skládat ze dvou částí – z uživatelské aplikace, která slouží k ovládání systémového ovladače a případnému výpisu některých zpráv pro uživatele a vlastního systémového ovladače, který bude vykonávat detekci a případné odstranění použitých technik rootkitů. Z uživatelské aplikace bude ovladač řízen pomocí IOCTL příkazů. Na základě vstupních parametrů se zavolá příslušná funkce v systémovém ovladači a provede se požadovaná akce.

Pro jednoduchost aplikace bude výpis detekovaných problémů probíhat pomocí debuggovacích hlášek, které se budou vypisovat do konzole VirtualKD debuggeru. Na základě těchto výpisů bude moci uživatel identifikovat potenciální problémy a zadáním správných parametrů a jména systémového volání odstranit zahákování záznamu v SSDT tabulce, či specifikací identifikátoru procesu ukončit skrytý proces.

Uživatelská aplikace i systémový ovladač budou vytvořeny v jazyce C za použití funkcí z nativního API Windows.

6.2.1 Uživatelská aplikace

Uživatelská aplikace se skládá kvůli zlepšení čitelnosti z několika modulů. Hlavním souborem je antirootkit.c obsahující spouštěcí funkci main(). Při spuštění aplikace z konzole, se ze všeho nejdříve provede zpracování parametrů pomocí funkcí implementovaných v souboru utils.c. Výsledné nastavení je uloženo do struktury typu `Seetings`, obsahující informaci o požadované akci, jméno systémového volání v případě požadavku na odhákování SSDT záznamu a číslo procesu, který by měl být ukončen v případě odstranění skrytého procesu. V tuto chvíli podporované akce jsou:

- LoadDriver
- UnloadDriver
- DetectSSDTHook
- DetectDKOMProcessHiding
- RemoveSSDTHook
- RemoveDKOMProcessHiding

Po zpracování parametrů proběhne otevření našeho ovladač pomocí následujícího volání:

```
CreateFile("\\\\.\\antirootkit",
           GENERIC_WRITE|GENERIC_READ,
           0,
           NULL,
           OPEN_EXISTING,
           FILE_ATTRIBUTE_NORMAL,
           NULL);
```

Následně proběhne požadované akce. Během zpracování některých akcí proběhne v uživatelském režimu i další aktivita. Konkrétně se jedná o detekci skrytých procesů pomocí metody přímé modifikace systémových objektů a o odstranění SSDT zahákování.

V prvním případě je zavolána funkce scanProcessesBySnapShot() deklarovány v souboru processes.h a implementována v processes.c. Tato funkce provede jednu z variant vysokoúrovňové skenování procesů a vypíše běžící procesy a jejich počet do konzole. Fungování této techniky je popsané v kapitole 6.3.2.

V druhém případě je potřeba nejdříve vypočítat původní adresu systémového volání, které chceme odhákovat. Toho je docíleno zavoláním funkcí getSSDTIndexByZwSystemCall() a getNtFunctionAdressBySSDTIndex(), které jsou deklarovány v souboru hooks.h a implementovány v souboru hooks.c. První funkce nalezne index daného volání v SSDT tabulce, druhá ji vypočítá její adresu.

Jakmile jsou všechny případné informace pro detekci či odstranění technik získány, zavolá se požadovaná obsluha ze systémového ovladače. Příklad, jak vypadá toto volání pro akci odstranění SSDT hákování, je níže:

```
DeviceIoControl(hDevice,
            IOCTL_REMOVE_SSDT,
            inputBuffer,strlen(inputBuffer),
            outputBuffer,sizeof(outputBuffer),
            &numberOfBytes,
            NULL);
```

Požadovaná akce je specifikována jedním z maker, které jsou definovány v hlavičkovém souboru defines.h. Makro pro odstranění SSDT hákování vypadá následovně:

```
#define SIOCTL_TYPE 40000
#define IOCTL_REMOVE_SSDT\
    CTL_CODE( SIOCTL_TYPE, \
              0x802, \
              METHOD_BUFFERED, \
              FILE_READ_DATA|FILE_WRITE_DATA)
```

6.2.2 Systémový ovladač

Implementace systémového ovladače se opět skládá z více souborů. Pro potřeby překladu pomocí prostředí Checked Build Environment z WDK je vytvořen soubor SOURCES, který obsahuje základní nastavení pro překlad a seznam všech zdrojů, které jsou součástí systémového ovladače. Obsah tohoto souboru vypadá následovně:

```
TARGETNAME=ANTIROOTKIT
TARGETPATH=OBJ
TARGETTYPE=DRIVER
SOURCES=antirootkit.c \
        hooks.c \
        processes.c \
        majorfunctions.c
```

Antirootkit.c je soubor obsahující hlavní funkce systémového ovladače DriverEntry(), sloužící pro spuštění činnosti ovladače, a DriverUnload() sloužící k zastavení ovladače. Na začátku DriverEntry() vytvoříme zařízení pomocí následující funkce IoCreateDevice():

```
IoCreateDevice(DriverObject,
           0,
           &deviceNameUnicodeString,
           FILE_DEVICE_ANTIROOTKIT,
           0,
           TRUE,
           &pDeviceObject);
```

Pokud se zařízení vytvoří korektně, vytvoříme pro ně i symbolický odkaz, aby mohlo být snadno voláno z uživatelské aplikace. Toho docílíme funkcí IoCreateSymbolicLink():

```
IoCreateSymbolicLink(&deviceLinkUnicodeString,
                     &deviceNameUnicodeString);
```

Na závěr DriverEntry() funkce už jen specifikujeme funkci pro zastavení ovladače a funkce pro obsluhu konkrétních IRP žádostí, které jsou implementovány v soubotu majorfunctions.c. Z těchto funkcí nás bude nejvíce zajímat funkce obsluhující IRP_MJ_DEVICE_CONTROL:

```
DriverObject->DriverUnload = DriverUnload;
DriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL] =
                     IoControlFunction;
```

Zmíněná obsluha nás bude nejvíc zajímat proto, že právě ona bude na základě specifikovaného IOCTL příkazů rozhodovat, která činnost se má provést. Za tímto

účelem nejdříve získáme lokaci IRP zásobníku a ze struktury, umístěné na dané adrese získáme kód IOCTL příkazu:

```
IrpStackLocation = IoGetCurrentIrpStackLocation(Irp);
CtlCode = IrpStackLocation->
            Parameters.DeviceIoControl.IoControlCode;
```

Na základě tohoto kódu proběhne rozhodnutí o požadované akci. Funkce pro detekci a odstranění hákování SSDT jsou umístěny v souboru hooks.c a funkce pro detekci a odstranění skrytých souborů pomocí techniky přímé modifikace systémových objektů jsou umístěné v souboru processes.c.

6.3 Detekce a odstranění

V rámci detektoru je v současné době implementována detekce dvou technik používaných rootkity, a to hákování SSDT tabulky a skrývání procesů pomocí přímé modifikace systémových objektů. Případná implementace detekce dalších metod se dá snadno přidat vytvořením nového IOCTL příkazu a implementací jeho obsluhy.

6.3.1 SSDT Hákování

Detekce zahákování SSDT je vcelku přímočará. Pomocí *for* cyklu projdeme pole systémových volání v tabulce volání `KiServiceTable` ve struktuře `KeServiceDescriptorTable`. Pro každé systémové volání zkontrolujeme, zda jeho adresa leží ve správném rozsahu a zda neukazuje někam ven. V případě, že v tomto rozsahu neleží, je to znamení toho, že bylo dané volání zahákováno a tudíž bude zaznamenáno.

Pokud chceme dané systémové volání odhákovat, musíme nejdříve zjistit původní adresu systémového volání. To provedeme v uživatelské části aplikace. Nejprve získáme informace o modulu ntoskrnl.exe, v jehož rozsahu adres by se systémové volání mělo nacházet a vypočítáme právě tento adresový rozsah. Nyní získáme referenční adresy některého ze systémových volání, například `ZwCreateFile`. V ntoskrnl.exe zjistíme adresu exportované funkce `NtCreateFile` a z SSDT tabulky získáme adresu `ZwCreateFile`. U těchto získaných adres vypočítáme jejich offset. V tuhle chvíli se vrátíme k systémovému volání, které chceme odhákovat. Zjistíme adresu exportované funkce v ntoskrnl.exe, která odpovídá danému systémovému volání a tu předáme pomocí IOCTL příkazu systémovému ovladači našeho detektoru. Na závěr nahradíme touto získanou adresou špatnou adresu v SSDT tabulce. Nesmíme přitom zapomenout, že vzhledem k umístění SSDT tabulky v paměti pouze pro čtení a jejím sdílení více procesory, musíme nejprve dočasně povolit zápis do dané části paměti a zajistit výlučný

přístup k dané adrese. Prvního docílíme deaktivací WP bitu, což je metoda, která byla popsána již dříve v této práci. Druhého docílíme pomocí použití funkce `InterLockedExchange()` pro záměnu adres.

6.3.2 Skryté procesy

Jak již bylo řečeno v kapitole věnované detekci techniky přímé modifikace systémových objektů, pro úspěšnou identifikaci skrytých procesů musíme získat seznamy procesů z různých zdrojů. V rámci tohoto detektoru použijeme čtyři metody:

- Vytvoření snímku systému a procházení procesů v něm
- Skenování procesů hrubou silou
- Skenování vláken hrubou silou
- Procházení tabulek obsluhy procesů

První metoda se řadí mezi vysokoúrovňové, protože je implementována přímo v uživatelské aplikaci a spoléhá tudíž pouze na volání existujících funkcí. Využívá snímku systému, který získáme zavoláním funkce `CreateToolhelp32Snapshot()`. Tím získáme přístup k obsluze snímku a pomocí funkcí `Process32First()` a `Process32Next()` můžeme projít všechny procesy a vytvořit z nich seznam. Zjednodušený kód tohoto algoritmu je uveden níže:

```
PROCESSENTRY32 procEntry;

Snapshot = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);
Process32First(snapshot, &procEntry);

do
{
  printf("PID %04d: %s\n", procEntry.th32ProcessID, procEntry.sxExeFile)
}
while(Process32Next(snapshot, &proceEntry))
```

Druhá metoda, skenování procesů hrubou silou, už je implementována v systémovém ovladači. Používá standartní volání API, a tudíž se stále ještě jedná o vysokoúrovňové procházení. V rámci této metody se pokusíme otevřít obsluhu procesů s identifikátorem procesu 0x0000 až 0x4E1C pomocí funkce `OpenProcess()`. Protože PID procesů jsou dělitelné čtyřmi, nemusíme procházet všechny možnosti, ale pouze právě ty, které splňují podmínku dělitelnosti čtyřmi. Pokud se otevření podaří, znamená to, že proces s daným PID existuje. Získáme ukazatel na strukturu EPROCESS, která jej reprezentuje, a z ní získáme informace, které přidáme do seznamu.

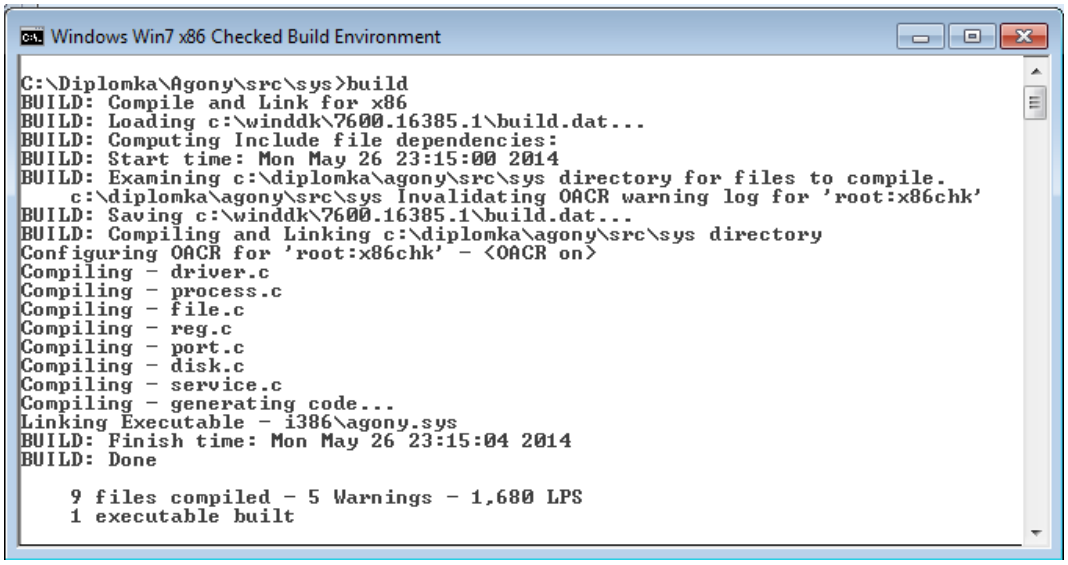
Třetí metoda je podobná předešlé, opět budeme procházet identifikátory hrubou silou, ale tentokrát to budou identifikátory vláken. Opět budeme procházet od 0x0000 do 0x4E1C všechny možné vlákna. V případě, že narazíme na existující vlákno, získáme

ukazatel ke struktuře ETHREAD, ze které získáme ukazatel na strukturu EPROCESS s potřebnými informacemi, které vložíme do seznamu procesů.

Poslední metodou, která bude implementována v detektoru, je procházení tabulek obsluhy, které jsou asociovány s procesy, která představuje metodu procházení procesů na nízké úrovni. Jak již bylo uvedeno v kapitole 5.3, projdeme dvousměrně vázaný seznam objektů *ObjectTable* typu HANDLE_TABLE a získáme z ní všechny existující procesy.

6.4 Testování

Pro otestování schopnosti detekce a odstranění SSDT hákování použijeme testovací rootkit Agony, představený v kapitole 4.1. Použijeme jej pro skrytí procesu cmd.exe, čehož je dosaženo pomocí zahákování `ZwQuerySystemInformation`. Nejprve tedy pomocí příkazu *build* v prostředí Checked Build Environment sestavíme systémový ovladač Agony, viz Obrázek 9 - Sestavení rootkitu Agony.



Obrázek 9 - Sestavení rootkitu Agony

Poté sestavíme uživatelskou aplikaci pro obsluhu ovladače a z příkazového řádku zavoláme následující příkaz:

```
agony -p cmd.exe
```

Tímto jsme skryli všechny procesy se jménem cmd.exe pomocí zahákování SSDT, což můžeme zkontrolovat spuštěním prohlížeče procesů. Nyní z příkazového řádku zavoláme uživatelskou aplikaci našeho detektoru a předáme jí jako parametry přepínače `-s` a `-d`, čímž specifikujeme, že chceme detekovat SSDT hákování:

```
antirootkit –s -d
```

V konzoli debuggeru VirtualKD zkontrolujeme výpis nalezených SSDT hákování, mezi kterými by se měl v tuto chvíli nalézat i `ZwQuerySystemInformation`. Proto opět zavoláme náš detektor, tentokrát ale specifikujeme jako parametry přepínače `–s` a `–r` a jméno systémového volání, které chceme odhákovat, čímž by se měl proces cmd.exe opět objevit v prohlížeči procesů:

```
antirootkit –s –r ZwQuerySystemInformation
```

Obdobným způsobem otestujeme možnosti detektoru pro odhalení skrytých procesů s pomocí nám již známého rootkitu FU. Rootkit sestavíme a s jeho pomocí skryjeme opět proces cmd.exe. Následně pomocí parametrů specifikujeme, že chceme provést detekci skrytých procesů:

```
antirootkit –p –d
```

V konzoli debuggeru VirtualKD porovnáme výpisy běžících procesů, které našel náš detektor, a najdeme skryté procesy, mezi kterými by měl být právě cmd.exe. Na závěr zavoláme opět náš detektor, tentokrát mu ale jako parametry předáme přepínače `–p` a `–r` a PID skrytého procesu, čímž by měl být tento proces bude ukončen:

```
antirootkit –p –r 12760
```

6.5 Zhodnocení

Metody detekce rootkitů, používajících metody hákování SSDT a skrývání procesů pomocí přímé modifikace systémových objektů, které byly použity v tomto detektoru, patří mezi metody základní. Jejich nevýhodou je, že se vesměs spoléhají na existující API volání, která mohou být již napadena a upravena. Pro jednoduché starší rootkity, jako jsou Agony či FU, které používají pouze jednu konkrétní metodu skrývání, je tento detektor relativně použitelný. Proti rootkitům, které jsou složitější a používají technik skrývání více, jako je například moderní TDSS.TDL4, je tento detektor neúčinný.

Aby se možnosti detekce rootkitů a jejich odstranění implementovaným detektorem zlepšily, bylo by potřeba implementovat detekci i dalších metod, a to nejen těch, které byly popsány v kapitole 5, ale i třeba detekci infekce hlavního bootovacího záznamu. Zkombinováním těchto technik by se schopnosti detektoru zvýšily.

Důležitým dalším vylepšením detektoru by byla jeho větší automatizace za účelem snížení nutnosti zapojení uživatele do procesu odstraňování rootkitu a úprava

komunikace s uživatelskou aplikací, aby nebylo nutné vypisovat zprávy pomocí debuggovacích hlášek v konzoli systémového debuggeru, ale přímo v uživatelské aplikaci.

V případě použití technik, které používají průběžnou kontrolu integrity systémových volání, by bylo nutné dosáhnout co nejdřívějšího nahrání systémového ovladače detektoru, abychom mohli získat nekompromitované údaje. Toho by se dalo docílit úpravou hlavního bootovacího záznamu.

7 Závěr

Hlavním nebezpečím rootkitů je jejich schopnost skrývat sebe či jiné entity, jako jsou procesy, systémové ovladače a soubory, a ovlivňovat systém, aniž by o tom uživatel věděl. Ve chvíli, kdy je přítomnost rootkitu odhalena, může být způsobená škoda již obrovská. Odstranění rootkitu bohužel bývá často velmi složité a nemusí být stoprocentní, protože moderní rootkity, jako je například TDSS.TDL.4, umějí obnovovat svoje části, které byly bezpečnostním softwarem zneutralizované. Nejlepším způsobem ochrany před rootkity je tedy prevence a to jak používáním nejaktuálnějších verzí programů a pravidelnou aktualizací operačního systému, tak dodržováním zásad bezpečného procházení internetu. I přes dodržování zmíněných pravidel se však může stát, že počítač bude napaden škodlivým softwarem, který využívá rootkitů, a proto je potřebné stále vyvíjet nové metody detekce těchto rootkitů. Vývoj, jak na straně rootkitů, tak na straně bezpečnostního softwaru, probíhá rapidním tempem a vznikají stále nové a nové postupy jako reakce na techniky druhé strany.

V této práci byly uvedeny nejzákladnější techniky, které rootkity používají za účelem skrývání. Některé z nich, jako jsou například hákování IDT, MSR či SSDT, byly v poslední době velmi ztížené novými ochrannými prostředky na úrovni operačního systému, jako jsou PatchGuard či kontrola integrity systému a podepisování systémových ovladačů. Stále však existuje možnost, jak obejít i tyto techniky. Jejich nevýhodou je, že jsou relativně snadno detekovatelné a proto je výhodné používat techniky, které nejsou tak nápadné. Mezi ně patří modifikace kódu za běhu programu a zejména přímá modifikace systémových struktur. Zatímco modifikace kódu se dá odhalit pomocí pravidelné kontroly integrity či porovnáním kódu v paměti oproti stejnému kódu na disku, přímá modifikace systémových objektů je odhalitelná hůře, zejména dojde-li k ní na co nejnižší úrovni. V současné době se navíc mezi vývojáři rootkitů projevuje patrná tendence přesouvat rootkity mimo meze operačního systému, například do firmwaru hardwaru. Takové rootkity jsou pro současný bezpečnostní software prakticky nezjistitelné a představují tudíž obrovskou výzvu.

Výstupem této práce je demonstrační detektor rootkitů založený na detekci technik hákování SSDT a skrývání procesů pomocí přímé modifikace systémových souborů. Detektory založené na detekci úzké množiny technik rootkitů však nejsou příliš účinné proti moderním rootkitům. Aby je byly možné detekovat, je potřeba implementovat detekci více metod a snažit se o nahrání do systému co nejdřív. Ten, kdo se dokáže do systému dostat dříve má totiž nespornou výhodu.

Literatura

[1] BLUNDEN, Bill. Rootkit arsenal: escape and evasion in the dark corners of the system. 2nd ed. Burlington, MA: Jones & Bartlett Learning, 2012, xxxi, 783 p. ISBN 14-496-2636-X.

[2] HOGLUND, Greg. Rootkits: subverting the windows kernel. Vyd. 1. Boston: Addison-Wesley Professional, 2005, 324 s. ISBN 03-212-9431-9.

[3] RUSSINOVICH, Mark E., David A. SOLOMON a Alex IONESCU. *Windows internals: Part 1*. 6th ed. Redmond, Wash.: Microsoft Press, 2012, v. <1> (xxii, 726 s.). ISBN 07-356-4873-5.

[4] RUSSINOVICH, Mark E., David A. SOLOMON a Alex IONESCU. *Windows internals: Part 2*. 6th ed. Redmond, Wash: Microsoft Press, 2012. ISBN 07-356-6587-7.

[5] SZOR, Peter. *Počítačové viry: analýza útoku a obrana*. Vyd. 1. Brno: Zoner Press, 2006, 608 s. ISBN 80-868-1504-8.

[6] *KernelMode.info: A forum for kernel-mode exploration.* [online]. [cit. 2014-05-27]. Dostupné z: http://www.kernelmode.info/

[7] Kernel patch protection: frequently asked questions. In: *Microsoft MSDN* [online]. January 22, 2007 [cit. 2014-05-27]. Dostupné z: http://msdn.microsoft.com/en-us/library/windows/hardware/dn613955%28v=vs.85%29.aspx

[8] Agony ring0 rootkit. In: *OPEN HACKING* [online]. 2012 [cit. 2014-05-27]. Dostupné z: http://openhacking.blogspot.cz/2012/01/malware-sourceagony-ring0-rootkit.html

[9] FU Ring0 Rootkit. In: *Secret-zone.net* [online]. 10-27-2012 [cit. 2014-05-27]. Dostupné z: http://www.secret-zone.net/f124/fu-ring0-rootkit-5118/#

[10] GOLOVANOV, Sergey a Vyacheslav RUSAKOV. TDSS. *SECURELIST* [online]. 2010 [cit. 2014-05-27]. Dostupné z: http://www.securelist.com/en/analysis/204792131/TDSS#1

[11] RUSAKOV, Vyacheslav. TDSS. TDL-4. *SECURELIST* [online]. 2011 [cit. 2014-05-27]. Dostupné z: https://www.securelist.com/en/analysis/204792157/TDSS_TDL_4

Seznam příloh

Příloha č. 1 – DRIVER_OBJECT

Tento kód je převzat z hlavičkového souboru wdm.h, dodávaného společně s Microsoft WDK.

```
typedef struct _DRIVER_OBJECT {
    CSHORT Type;
    CSHORT Size;
    //
    // The following links all of the devices created by a single driver
    // together on a list, and the Flags word provides an extensible flag
    // location for driver objects.
    //
    PDEVICE_OBJECT DeviceObject;
    ULONG Flags;
    //
    // The following section describes where the driver is loaded.  The count
    // field is used to count the number of times the driver has had its
    // registered reinitialization routine invoked.
    //
    PVOID DriverStart;
    ULONG DriverSize;
    PVOID DriverSection;
    PDRIVER_EXTENSION DriverExtension;
    //
    // The driver name field is used by the error log thread
    // determine the name of the driver that an I/O request is/was bound.
    //
    UNICODE_STRING DriverName;
    //
    // The following section is for registry support.  Thise is a pointer
    // to the path to the hardware information in the registry
    //
    PUNICODE_STRING HardwareDatabase;
    //
    // The following section contains the optional pointer to an array of
    // alternate entry points to a driver for "fast I/O" support.  Fast I/O
    // is performed by invoking the driver routine directly with separate
    // parameters, rather than using the standard IRP call mechanism.  Note
    // that these functions may only be used for synchronous I/O, and when
    // the file is cached.
    //
    PFAST_IO_DISPATCH FastIoDispatch;
    //
    // The following section describes the entry points to this particular
    // driver.  Note that the major function dispatch table must be the last
    // field in the object so that it remains extensible.
    //
    PDRIVER_INITIALIZE DriverInit;
    PDRIVER_STARTIO DriverStartIo;
    PDRIVER_UNLOAD DriverUnload;
    PDRIVER_DISPATCH MajorFunction[IRP_MJ_MAXIMUM_FUNCTION + 1];

} DRIVER_OBJECT;
typedef struct _DRIVER_OBJECT *PDRIVER_OBJECT;
```

Příloha č. 2 – Kostra ovladače

Zde je uvedena kostra systémového ovladače, která je uvedena v [1].

```
#include <ntddk.h>

NTSTATUS
STDCALL
DriverDispatch(IN PDEVICE_OBJECT DeviceObject,
               IN PIRP Irp)
{
    return STATUS_SUCCESS;
}

VOID
STDCALL
DriverUnload(IN PDRIVER_OBJECT DriverObject)
{
    DbgPrint("DriverUnload() !\n");
    return;
}

NTSTATUS
STDCALL
DriverEntry(IN PDRIVER_OBJECT DriverObject,
            IN PUNICODE_STRING RegistryPath)
{
    DbgPrint("DriverEntry() !\n");

    DriverObject->DriverUnload = DriverUnload;

    return STATUS_SUCCESS;
}
```

Příloha č. 3 – Zdrojové kódy

Příloha č. 4 – Manuál programu