Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

Využitie virtualizácie na zlepšenie detekcie škodlivého software.

Diplomová práca

2018
Martin Ivančík

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

Využitie virtualizácie na zlepšenie detekcie škodlivého software.

Diplomová práca

Študijný program:	Informatika
Študijný odbor:	2508 Informatika
Školiace pracovisko:	Katedra informatiky
Školiteľ:	RNDr. Jaroslav Janáček PhD.
Konzultant:	Mgr. Peter Košinár

Bratislava, 2018
Martin Ivančík

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

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

Názov: Využitie virtualizácie na zlepšenie detekcie škodlivého softvéru
Using Virtualization for Improved Detection of Malicious Software

Anotácia:

Cieľ: Cieľom práce bude preskúmať možnosti využitia virtualizačných technológií moderných procesorov na detekciu podozrivého správania škodlivého softvéru fungujúceho až na úrovni jadra operačného systému. Tento typ škodlivého softvéru môže pri svojej činnosti využívať zásahy do aktivít v rámci operačného systému, ktoré nie sú efektívne odhaliteľné prostriedkami pracujúcimi na rovnakej úrovni oprávnení - medzi sledovaniahodné aktivity patrí napríklad sieťová komunikácia produkovaná ovládačom v rámci jadra operačného systému (ktorá môže byť úplne neviditeľná pre nástroje na sledovanie sieťovej prevádzky).
Výsledkom práce bude implementácia monitorovacieho / vyhodnocovacieho nástroja, ktorý umožní odhaliť podozrivé správania, či už na základe zistenia aktuálneho stavu alebo pomocou dlhodobého pozorovania aktivít v systéme. Dá sa predpokladať, že systém je v čase aktivácie nástroja už napadnutý škodlivým softvérom.
Získané poznatky a implementované nástroje možno vyhodnotiť na vzorkách reálneho škodlivého softvéru. Primárne uplatnenie výsledkov sa očakáva v prostredí operačných systémov rodiny Microsoft Windows; pre túto platformu existuje dostatočne veľa škodlivého softvéru, ktorý by takýmto spôsobom mohol byť detegovateľný.

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

Dátum zadania: 13.12.2016

Dátum schválenia: 13.12.2016

prof. RNDr. Rastislav Kráľovič, PhD.
garant študijného programu

....

študent vedúci práce

Poďakovanie:

Chcel by som sa poďakovať predovšetkým rodičom a súrodencom za podporu počas celého štúdia. Ďalej ďakujem konzultantovi Petrovi Košinárovi za poskytnuté rady a pomoc a v neposlednom rade školiteľovi Jaroslavovi Janáčkovi za ochotu viesť túto diplomovú prácu.

Abstrakt

V moderných operačných systémoch je použitý koncept privilegovanosti vykonávaného programového kódu. Operačný systém a ovládače bežia s vyššími právami ako bežné aplikácie. S rozvojom škodlivého kódu sa začal objavovať malvér určený pre beh v jadre operačného systému, s vyššími právami. Škodlivý kód tak môže zasahovať do systému a bezpečnostné prvky, ktoré teraz bežia na rovnakej úrovni ako malvér, nie sú schopné zabezpečiť účinnú ochranu systému. Príkladom môže byť škodlivá aplikácia, ktorá modifikuje štruktúry operačného systému za účelom skrývania sieťovej komunikácie.

V práci sa venujeme predstaveniu technológií pre virtualizáciu, ich použitiu pre beh programového kódu na privilegovanejšej úrovni ako operačný systém, alebo škodlivý ovládač v infikovanom systéme a následne navrhujeme a implementujeme nový spôsob detekcie skrytej sieťovej komunikácie, využitím virtualizačných technológií.

Kľúčové slová: virtualizácia, hypervisor, rootkit, hook, NDIS, sieťová komunikácia, TCP, ovládač

Abstract

Modern operating systems use concept of different privileges for executable code. Operating system and device drivers are executed under higher privileges than ordinary applications. With the growth of malware, there was tendency to target the kernel of operating system for the sake of execution under higher privileges. Privileged malicious code may modify crucial internal structures of operating system and security mechanisms which are executed under same privileges as malicious code are not able to provide sufficient protection for system anymore. One example is malicious application modifying system in order to hide network communication.

In our thesis we present technologies for virtualization, its usage for executing code on more privileged layer than operating system or malicious code in infected system. We then design and implement new way of detection of hidden network communication using virtualization technologies.

Keywords: virtualization, hypervisor, rootkit, hook, NDIS, network communication, TCP, driver.

Contents

Úvod **1**

1 Škodlivý kód **4**

1.1 Rootkit 4
1.2 Rootkit a jeho techniky 5
1.2.1 Skrývanie procesu 5
1.2.2 Skrývanie súborov a komunikácie 7
1.3 Patch Guard 16

2 Virtualizácia **18**

2.1 Stručný popis 18
2.2 VT-x 22
2.3 Virtual-machine control data structure 23
2.4 Virtualizácia a stránkovanie 25
2.4.1 Skrývanie pamäte 29

3 Sieťová komunikácia **32**

3.1 Network Driver Interface Specification 32
3.2 Štruktúra paketu 35

4 Detekcia skrytej sieťovej komunikácie **39**

4.1 Malvér Pitou 39
4.2 Detekcia 40
4.2.1 Funkcia SendNetBufferListsHandler 41
4.2.2 Odchytávanie odoslaných paketov 46
4.2.3 TCP Spojenia 47
4.3 Implementácia a testovanie 51
4.4 Výsledky 53

5 Záver **55**

List of Figures

1 Privilégia v operačnom systéme [14]. 2

1.1 Injektovanie DLL knižnice [2] 6
1.2 Druhy hookovania [1] 7
1.3 Druhy hookovania [2] 8
1.4 Inline hook [11] 9
1.5 User a Kernel mód [11] 9
1.6 Tok vykonávania kódu v systéme MS Windows [15] 10
1.7 SSDT Hook [11] 12
1.8 IRP Hook [10] 14
1.9 Filter Driver [18] 15
1.10 DKOM [13] 15

2.1 Virtualizácia [25]. 19
2.2 Druhy virtualizácie [34] 20
2.3 Interakcia medzi VMM a Guestom [24]. 23
2.4 Virtualizácia [24]. 24
2.5 Virtuálna pamäť [32]. 26
2.6 Virtualizácia [32]. 27
2.7 Shadow Page table [12]. 28
2.8 Shadow Page table [23]. 29

3.1 Štruktúra NDIS [28] 32
3.2 Štruktúry NDIS ovládačov[29] 34
3.3 Štruktúra paketov v NDIS [19] 35
3.4 Štruktúra NET_BUFFER_LIST [30] 36
3.5 Dáta v štruktúre NET_BUFFER [19] 36
3.6 Získanie dát zo štruktúry NET_BUFFER 37

4.1 Súkromný TCPIP/IP stack [20] 40
4.2 Získanie IP adresy predvolenej brány 42
4.3 Registrácia protokolu 44

4.4 Časť štruktúry _NDIS_PROTOCOL_BLOCK 44
4.5 Štruktúra _NDIS_MINIPORT_DRIVER_CHARACTERISTICS . . . 45
4.6 Získanie adresy Send Handlera 46
4.7 Postupnosť volaní v module tcpip.sys 47
4.8 Funkcie volané pri enumerovaní spojení [27] 48
4.9 Štruktúra PartitionTable [31] 49
4.10 Overenie existencie záznamu v tabuľke TCP spojení pre danú cieľovú IP adresu a port 50
4.11 Ukončovanie TCP spojenia [17] 51
4.12 Program kontrolujúci odosielané pakety 52
4.13 Odchytené pakety v Guest OS [17] 54
4.14 Odchytené pakety v Host OS 54
4.15 Výpis hypervisora zobrazený v programe DebugView 54

Úvod

Jednou z kľúčových podmienok kladených na informačné systémy v dnešnej dobe je jeho bezpečnosť. Táto vlastnosť je podmienená nielen bezpečnosťou aplikácií bežiacich v systéme, ale predovšetkým bezpečnosťou a stabilitou samotného operačného systému, tvoriaceho základ informačného systému.

V minulosti, keď operačný systém a bežné aplikácie mali rovnaké privilégia, vznikalo veľa rôznych problémov. Obyčajná aplikácia mohla čítať a zapisovať do ľubovoľnej časti pamäte, čo v prípade zle naprogramovanej aplikácie mohlo spôsobiť prepísanie časti operačného systému a následne pád celého systému. Na druhej strane táto skutočnosť umožňovala autorom aplikácií ľubovoľne modifikovať časti operačného systému, čo sa dalo využiť na prispôsobenie systému a tvorbu zaujímavých softvérových komponentov.

Neskôr sa však ukázalo, že táto vlastnosť môže byť zneužitá aj zámerne škodlivými aplikáciami. Modifikovať a prepisovať ľubovoľné časti programového kódu v pamäti môže mať fatálne následky. Výsledkom môže byť napríklad únik citlivých informácií zo systému alebo získanie úplnej kontroly nad systémom zo strany útočníka, čo v konečnom dôsledku znamenalo, že takýto koncept fungovania systému nebol správny z hľadiska bezpečnosti. S postupným rozvojom softvéru a informačných technológií preto došlo k potrebe oddeliť od seba aplikačné programy a operačný systém a zabezpečiť ich ochranu.

Vznikol koncept virtuálnej pamäte, podľa ktorého každý proces disponuje svojím vlastným izolovaným adresným priestorom, oddeleným od adresných priestorov ostatných aplikácií. Pri pokuse o prístup k pamäti mimo svojho adresného priestoru sa k činnosti dostane operačný systém, ktorý vzniknutú situáciu ošetrí, napríklad ukončením aplikácie, ktorá udalosť spôsobila. Ďalej vznikli stupne privilegovanosti, určujúce oprávnenia jednotlivých programov, bežiacich v systéme. Boli definované štyri úrovne oprávnenia pre programový kód, ktoré sú znázornené na obrázku 1. Úroveň označená menším číslo predstavuje vyššie práva ako úroveň s väčším číslom. V nadväznosti na vznik týchto úrovní bola pridaná podpora pre prácu s týmito úrovňami do procesorov, pridaním nových inštrukcií, umožňujúcim prepínať procesor medzi jednotlivými úrovňami.

Moderné operačné systémy však využívajú iba dve úrovne a to Ring 3 označovanú

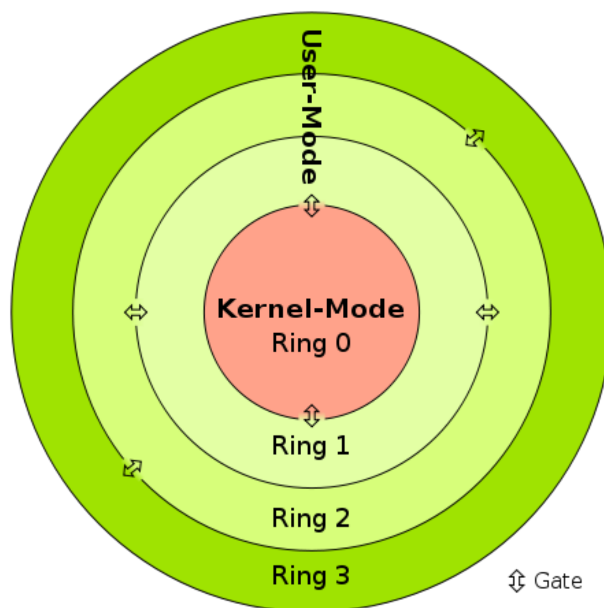


Figure 1: Privilégia v operačnom systéme [14].

ako užívateľský mód a Ring 0 alebo mód jadra operačného systému. Operačný systém beží na úrovni označovanej ako Ring 0 a ostatné aplikácie na úrovni Ring 3. Aplikačné programy bežiace na úrovni Ring 3 majú obmedzené možnosti. Konkrétna aplikácia (program) je ohraničená svojím vlastným adresným priestorom a môže vykonať len bežné neprivilegované inštrukcie. Programový kód bežiaci na úrovni Ring 0 je privilegovaný, to znamená, že môže vykonať ľubovoľnú inštrukciu procesora a má prístup k celej fyzickej pamäti. Operačný systém ako správca (supervisor) má väčšie privilégia, aby mohol riadiť beh systému a zabezpečiť ochranu seba i ostatných aplikácií.

Rozdielnosť privilégií využívajú aj programy určené na kontrolu a zabezpečenie systému. Typickým príkladom je antivírový program, ktorý potrebuje pre svoj beh vyššie oprávnenia ako bežné programy, a preto beží na rovnakej úrovni ako operačný systém. Tým pádom má antivírus prístup k adresnému priestoru všetkých bežiacich programov a môže kontrolovať ich vykonávanie, chrániť ich, či detegovať a zastaviť škodlivú činnosť týchto programov. Dôležitým predpokladom všetkých aplikácií, bežiacich ako privilegované, je ich stabilita.

Ako sme spomenuli na začiatku, pád privilegovanej aplikácie môže mať dosah na celý systém, na rozdiel od aplikácie, bežiacej v užívateľskom móde, ktorej pád nemusí pre systém predstavovať veľký problém.

S postupným rozvojom škodlivého kódu (malvéru) vznikali rozličné techniky ako získať kontrolu nad napadnutým systémom a cieľom autorov malvéru vždy bolo, aby táto kontrola bola maximálna. V praxi to znamenalo získať čo najvyššie možné oprávnenia. So vznikom rozdelenia privilégií bolo časom možné vidieť častejšie sa vyskytujúci malvér určený pre beh na úrovni jadra operačného systému.

Dôležitá vlastnosť, ktorú beh na úrovni jadra OS priniesol, bola možnosť skryť svoju prítomnosť v systéme modifikáciou niektorých častí OS, čo znamená zhoršenie možnosti detekcie škodlivého softvéru. Ďalej je potrebné si uvedomiť fakt, že ak programový kód malvéru beží na rovnakej úrovni ako operačný systém, či antivírový program, ich činnosť z hľadiska ochrany bezpečnosti môže byť nielen oslabená ale aj úplne znemožnená, pretože malvér môže priamo modifikovať časti bezpečnostného softvéru.

Ak by však detekcia škodlivého kódu bola napriek tomu možná, môže nastať iný problém. Systém je infikovaný malvérom skôr, ako je načítaný antivírový program, ktorého spustenie by mohlo byť znemožnené práve už bežiacim privilegovaným kódom škodlivého softvéru.

Vzniká teda otázka, ako riešiť spomenuté problémy. Riešenie by mohli ponúknuť moderné procesory, ktoré podporujú hardvérovú virtualizáciu. V tejto práci sa budeme venovať popisu virtualizačných techník, spôsobu ich využitia pre beh kódu na úrovni nižšej ako Ring 0, teda s ešte väčšími privilégiami ako operačný systém, alebo škodlivý kód v infikovanom systéme a popíšeme spôsoby zlepšenia detekcie škodlivého kódu v už infikovanom systéme, hoci by kód bežal na úrovni jadra operačného systému. Pozornosť budeme venovať prevažne detekcii skrytej sieťovej komunikácie.

Chapter 1

Škodlivý kód

V tejto kapitole sa zameriame na stručné predstavenie malvéru pracujúceho v jadre operačného systému a techník, ktoré využíva pre svoju činnosť.

1.1 Rootkit

Slovo rootkit predstavuje pomenovanie pre malvér operujúci v jadre OS a je zložené z dvoch častí: *root* (koreň) je pomenovanie, ktoré pochádza z unixových operačných systémov a znamená administrátorský prístup. Druhé slovo *kit* znamená skupina alebo sada (nástrojov). V minulosti, keď malvér pracoval na nízkej úrovni, pre svoju činnosť potreboval viacero rôznych modulov, teda skupinu nástrojov. Každý modul implementoval určitú konkrétnu funkcionalitu. Tento pojem sa zachoval dodnes aj pre softvér, ktorý pozostáva iba z jedného modulu, ale obsahuje komplexnú funkcionalitu.

Rootkity delíme podľa typu na nasledovné:

- **User-mode (Ring 3)**: Hoci sme v úvode a v definícii rootkitov spomenuli, že rootkity majú v systéme maximálne práva, existuje však výnimka. Kategória malvéru, ktorý pracuje na úrovni bežných užívateľských aplikácií a snaží sa skrývať svoju prítomnosť v systéme sa zvykne označovať ako užívateľský rootkit.
- **Kernel-mode (Ring 0)**: Rootkit s plnými právami pracujúci na úrovni jadra operačného systému.
- **Bootkit**: Malvér infikujúci MBR (Master Boot Record). MBR je záznam, ktorý sa nachádza na začiatku disku a obsahuje zavádzajúci program operačného systému. Pri prepísaní zavádzajúceho programu škodlivým kódom je malvér spustený ešte pred operačným systémom a týmto spôsobom je možné vyradiť z fukčnosti bezpečnostné mechanizmy OS a kontrolovať tak beh systému.

- **Hypervisor (Ring -1)**: Softvér bežiaci na nižšej úrovni ako operačný systém. Do tejto kategórie možno zaradiť softvér určený na virtualizáciu, ktorej sa budeme venovať v samostatnej kapitole.
- **Firmware malware (Ring -2)**: Operačný systém komunikuje s periférnymi zariadeniami prostredníctvom API, ktoré mu poskytuje mikrokód nahratý v pamäti konkrétneho zariadenia. Škodlivý softvér modifikujúci mikroprogram periférnych zariadení sa nazýva firmware malvér.

1.2 Rootkit a jeho techniky

1.2.1 Skrývanie procesu

S rozvojom škodlivého kódu sa vyvíjali rozličné spôsoby skrývania prítomnosti v systéme. Najčastejší spôsob ako sa malvér bežiaci v užívateľskom priestore pokúša skryť, je vloženie svojho programového kódu do iného procesu. Detekcia škodlivého kódu, ktorý beží v kontexte legitímneho procesu je oveľa náročnejšia.

Ako sme spomenuli v úvode, so vznikom konceptu virtuálnej pamäte a izolovaného adresného priestoru, sa zabránilo možnosti priameho zápisu do pamäte inej aplikácie. Existujú však API volania operačného systému a udalosti, prostredníctvom ktorých je možné tento zápis dosiahnuť. Teraz popíšeme niekoľko najrozšírenejších.

- **DLL Injection**: Na obrázku 1.1 je znázornený spôsob vloženia kódu do iného procesu využitím Windows API funkcií. Malvér uloží na disk škodlivú DLL knižnicu. Volaním funkcie *VirtualAllocEx()*, alokuje pamäť v konkrétnom procese a následne do tejto pamäte zapíše volaním *WriteProcessMemory()* cestu k tejto knižnici. Nakoniec spustí nové vlákno v procese volaním *CreateRemoteThread()*.

  Táto funkcia okrem iných berie ako argument smerník na funkciu, ktorá sa má vykonať po vytvorení nového vlákna a umožňuje špecifikovať jeden argument pre túto funkciu. V danom prípade malvér špecifikuje ako adresu funkciu *LoadLibrary()* a argument smerník na cestu, ktorú v predošlom kroku zapísal do alokovanej pamäte, výsledkom čoho je volanie funkcie *LoadLibrary()* s cestou ku škodlivej knižnici.

- **Code Injection**: Prebieha rovnako ako pri DLL Injection, ale do procesu sa zapíše priamo celý programový kód. Výhodou je napríklad to, že na disk nie je potrebné ukladať DLL knižnicu, ktorá by mohla byť detegovaná napríklad skenerom súborov.

Figure 1.1: Injektovanie DLL knižnice [2]

```
int _tmain(int argc, _TCHAR* argv[])
{
        char* buffer = "D:\\dllinject.dll";
        int procID = 4444; // Assuming we know the process ID.

        // Get handle to process using all access permissions
        HANDLE process = OpenProcess(PROCESS_ALL_ACCESS, FALSE, procID);

        //Get address of the LoadLibrary function.
        LPVOID addr = GetProcAddress(
        GetModuleHandle(L"kernel32.dll"), "LoadLibraryA");

        // Allocate new memory region inside the process's address space.
        LPVOID arg = VirtualAllocEx(process, NULL, strlen(buffer),
        MEM_RESERVE | MEM_COMMIT, PAGE_READWRITE);


        // Write the argument to LoadLibraryA to the
        //   process's newly allocated memory region.
        int n = WriteProcessMemory(
        process, arg, buffer, strlen(buffer), NULL);

        // Inject our DLL into the process's address space.
        HANDLE threadID = CreateRemoteThread(
        process, NULL, 0, (LPTHREAD_START_ROUTINE)addr, arg, NULL, NULL);
        // Close the handle to the process
        CloseHandle(process);

        return 0;
}
```

- **Thread execution Hijacking**: V predošlých spôsoboch spustenia cudzieho kódu v legitímnom procese vždy vzniklo nové vlákno, vykonávajúce vložený kód. Vznik nového vlákna je udalosť, ktorá je pozorovateľná a môže vzbudiť podozrenie z narušenia systému útočníkom. Ďalší spôsob ako vykonať kód, je pozastaviť existujúce bežiace vlákno legitímneho procesu a modifikovať tzv. context štruktúru, ktorá obsahuje údaje potrebné pre vlákno počas jeho existencie a zmeniť smerník v tejto štruktúre, ktorý ukazuje na adresu nasledujúcej vykonávanej inštrukcie na adresu, kam bol vložený kód. Následne stačí vlákno spustiť a nový kód beží v rámci pôvodného vlákna.

- **Process Hollowing**: Názov odvodený z anglického *hollow out*(vydlabať) spočíva vo vytvorení nového legitímneho procesu, ktorý je však v stave pozastavený. Malvér nahradí pamäť tohoto procesu vlastným kódom a následne ho spustí.

- **Modifikácia registrov**: V databáze Registry systému Microsoft Windows existujú položky obsahujúce cesty k DLL knižniciam, ktoré majú byť automaticky načítané do novovzniknutých procesov za určitých okolností. Ak sa malvéru po-

darí uložiť na disk škodlivú knižnicu a zapísať cestu k tejto knižnici do registrového kľúča, škodlivý kód sa vykoná pri vzniku nového procesu. Ako príklad možno uviesť nasledovné registrové kľúče:

```
HKLM\Software\Microsoft\WindowsNT\CurrentVersion\Windows\Appinit_Dlls
HKLM\System\CurrentControlSet\Control\SessionManager\AppCertDlls
```

Knižnice uvedené v kľúči AppInit_Dlls sú načítané do každého nového procesu používajúceho knižnicu User32.dll, ktorá patrí medzi najčastejšie používane knižnice v systéme Windows. Knižnice z kľúča AppCertDlls sú načítané v prípade, že aplikácia používa nejakú z nasledujúcich API funkcií: *CreateProcess()* , *CreateProcessAsUser()* *CreateProcessWithLogonW()*, *CreateProcessWithTokenW()*, *WinExec()*.

1.2.2 Skrývanie súborov a komunikácie

Hoci je možné skryť vykonávanie kódu za iný, legitímne bežiaci proces, pre úplnú neviditeľnosť v systéme to nestačí. Indikátory kompromitácie môžu byť napríklad súbory uložené na disku, informácie zapísané v databáze Registry, alebo sieťová komunikácia.

Pre vytvorenie ilúzie, že daná entita alebo udalosť nie je prítomná v systéme, je potrebné modifikovať časti programového kódu aplikácie a operačného systému za účelom zmeny správania. Takéto techniky sa nazývajú hookovanie. Na obrázku 1.2 sú znázornené druhy hookovania, ktoré teraz popíšeme.

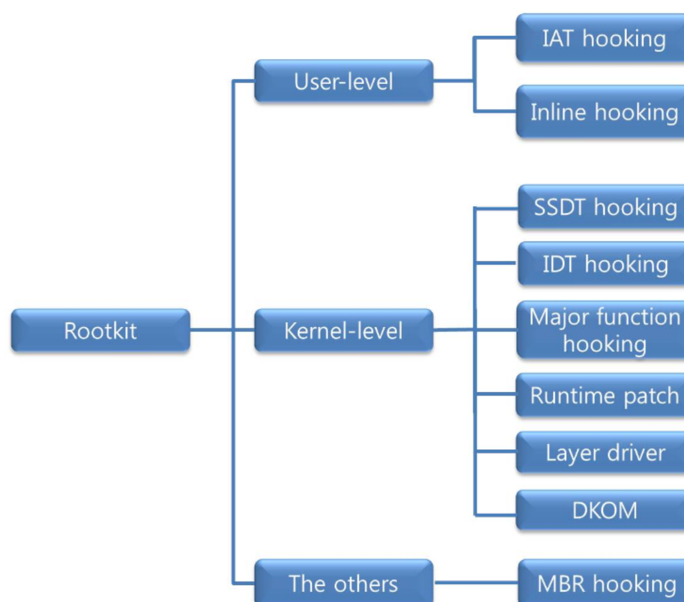


Figure 1.2: Druhy hookovania [1]

User mode Hooking

V užívateľskom móde sú využívané dve metódy modifikácie existujúceho kódu. Prvá sa nazýva **IAT (import address table) hooking**. Každý proces, ktorý volá funkcie z externých knižníc obsahuje v pamäti dátovú štruktúru nazývanú tabuľka importov, obsahujúcu zoznam smerníkov na funkcie, ktoré sú používané z konkrétnej externej knižnice. Volanie funkcie z programového kódu je realizované prostredníctvom týchto smerníkov. Nahradením smerníka v tejto tabuľke možno vynútiť volania inej funkcie a tým zmeniť správanie aplikácie.

Použitie tejto metódy je znázornené na obrázku 1.3. Po modifikácii tabuľky importov sa pri volaní funkcie *CreateFile()* najprv vykoná škodlivý kód, ktorý môže napríklad zmeniť argumenty funkcie, zavolať ešte inú funkciu a až potom sa riadenie odovzdá pôvodnej funkcii. Výsledkom takejto vhodnej úpravy kódu môže byť napríklad neschopnosť otvoriť súbor „tajny.txt“ prostredníctvom štandardného správcu súborov, ak táto modifikácia nastala v procese „explorer.exe“.

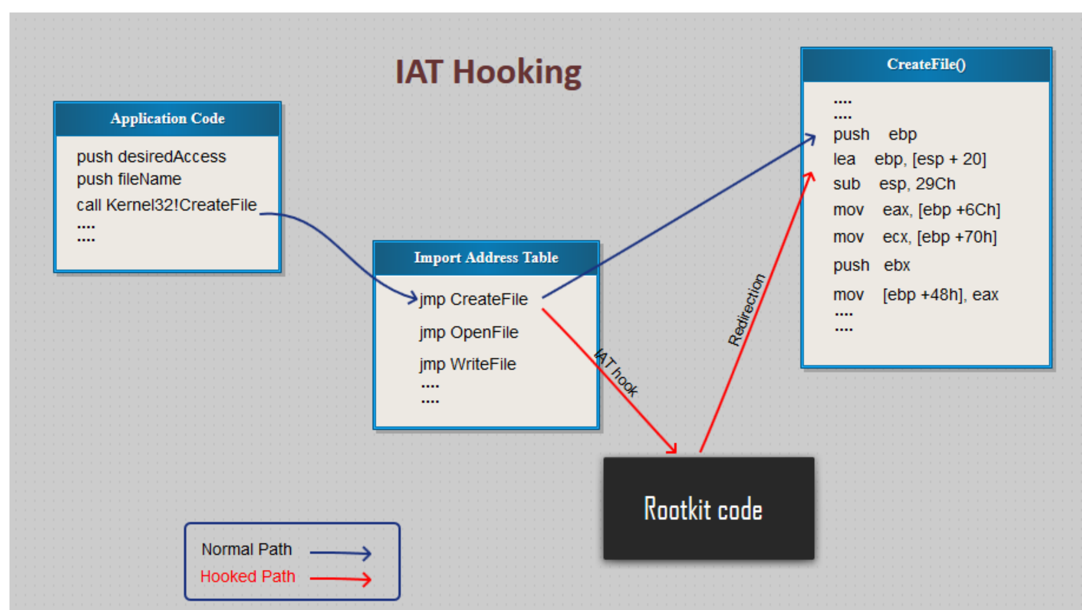


Figure 1.3: Druhy hookovania [2]

Druhý spôsob modifikácie funkcie sa nazýva **Inline Hooking** a spočíva v priamej modifikácii kódu funkcie. Princíp je znázornený na obrázku 1.4.

Pri inline hook metóde je prepísaný začiatok API funkcie inštrukciou skoku na adresu, kde sa nachádza škodlivý kód alebo iná obslužná funkcia. Pred modifikáciou funkcie je však potrebné uložiť začiatok funkcie, ktorý bude prepísaný, aby po vykonaní škodlivého kódu bolo možné pokračovať vo vykonávaní pôvodnej funkcie, vykonaním uložených inštrukcií a potom skokom naspäť do pôvodnej funkcie na prvú nemodifikovanú inštrukciu.

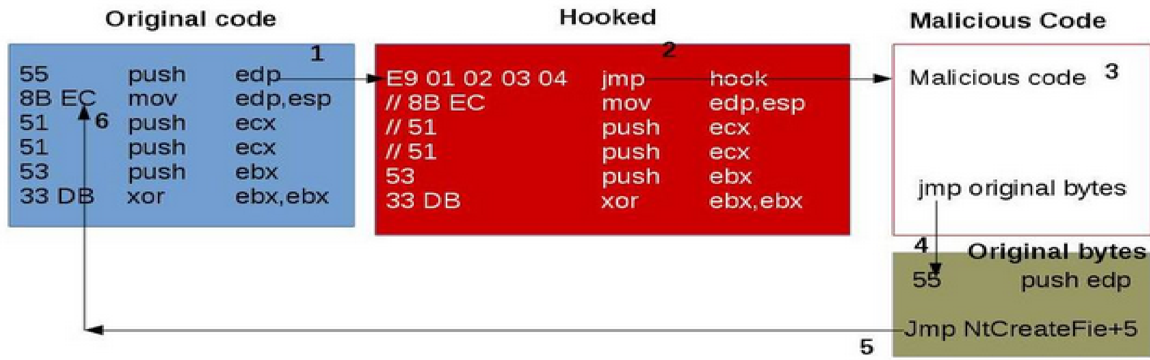


Figure 1.4: Inline hook [11]

Kernel mode Hooking

Modifikácia funkcií v užívateľskom priestore má dopad len na konkrétny proces, v ktorom bola modifikácia vykonaná, na rozdiel od modifikácie funkcií a štruktúr, ktoré sa nachádzajú v jadre operačného systému. V kernel móde existuje iba jeden adresný priestor spoločný pre všetok programový kód. To znamená, že v pamäti existuje práve jedna inštancia načítaného modulu a v prípade, že bude modifikovaný programový kód takéhoto modulu, ovplyvnený bude celý systém.

Na obrázku 1.5 je znázornená architektúra systému MS Windows. Volania API funkcií, ktoré vyžadujú asistenciu operačného systému používajú funkcie v knižnici ntdll.dll. Metódy v tejto knižnici najprv vhodne predspracujú argumenty funkcie a následne odovzdajú riadenie systémovej funkcii.

Volať priamo obslužné funkcie systémových ovládačov je možné prostredníctvom funkcie *DeviceIoControl()*.

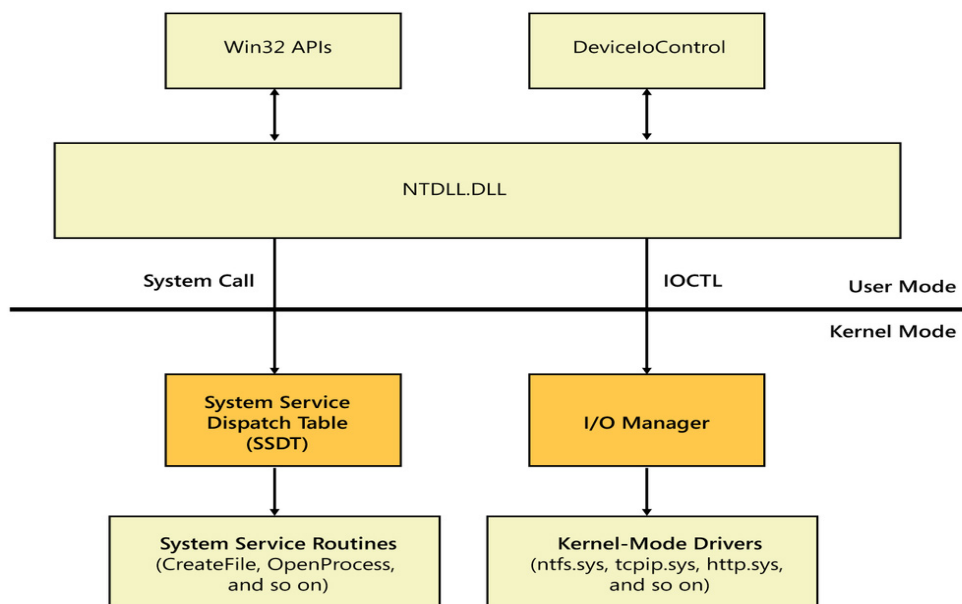


Figure 1.5: User a Kernel mód [11]

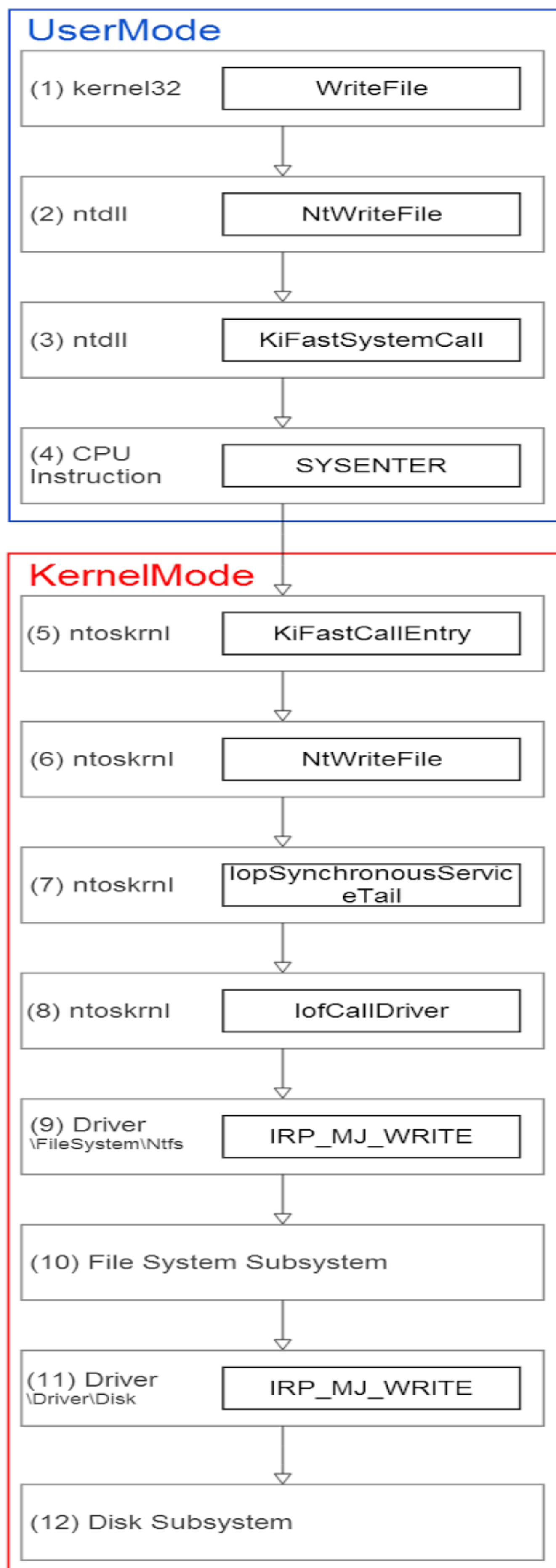


Figure 1.6: Tok vykonávania kódu v systéme MS Windows [15]

Detailný popis toku riadenia pri volaní Windows API funkcií, ktoré využívajú asistenciu operačného systému sa nachádza na obrázku 1.6. Volanie funkcie pre zápis do súboru *WriteFile()* bude pokračovať volaním funkcie *NtWriteFile()* v knižnici ntdll, odkiaľ sa riadenie presunie do funkcie *KiFastSystemCall()* a vykonaním inštrukcie SYSENTER sa procesor prepne do kernel módu.

V kernel móde sa vykoná funkcia *KiFastCallEntry()*, ktorá zavolá systémovú funkciu *NtWriteFile()*. Systémová funkcia ďalej využíva služby ovládačov. Využívanie služieb, ktoré poskytujú systémové ovládače, neprebieha priamo volaním funkcií s argumentami, ale prostredníctvom dátovej štruktúry IRP (I/O Request Packet). Každý IRP má svoj typ a obsahuje dáta (pôvodné argumenty) a ďalšie príkazy s ktorými ovládače pracujú.

Ovládače môžu využívať služby ďalších ovládačov a navzájom si preposielať IRP. Ovládače tak tvoria takzvanú reťaz. Posledný ovládač v reťazi zapíše dáta do súboru a výsledok vráti naspäť predchádzajúcim ovládačom a funkciám, ktoré boli volané.

- **SSDT (System Service Dispatch Table) Hook**:

  SSDT je tabuľka smerníkov na funkcie, ktoré implementujú služby poskytované operačným systémom Microsoft Windows. Funkcie sú implementované v jednom z modulov: *ntoskrnl.exe*, *ntkrnlmp.exe*, *ntkrnlpa.exe* alebo *ntkrpamp.exe*, v závislosti od verzie OS. Funkcia *KiFastCallEntry()* z predchádzajúceho príkladu používa tabuľku SSDT na nájdenie konkrétnej systémovej funkcie.

  Modifikácia smerníka v tejto tabuľke, rovnako ako pri IAT hook je znázornená na obrázku 1.7. V danom prípade sa namiesto pôvodnej funkcie *NtOpenProcess()*, ktorá vráti handle na proces, zavolá podvrhnutá funkcia, ktorá najprv skontroluje, na aký proces sa pokúša volajúca aplikácia získať handle a rozhodne, či vráti handle na tento proces alebo chybu ACCES_DENIED.

  Takouto modifikáciou možno implementovať napríklad ochranu proti ukončeniu alebo modifikácii procesu. Proces možno ukončiť volaním funkcie *ZwTerminateProcess()*, ktorá ako argument berie handle na konkrétny proces. Bez získania handle, proces nie je možné ukončiť, či zapísať do jeho adresného priestoru, keďže väčšina API funkcií pracuje s objektami prostredníctvom handle.

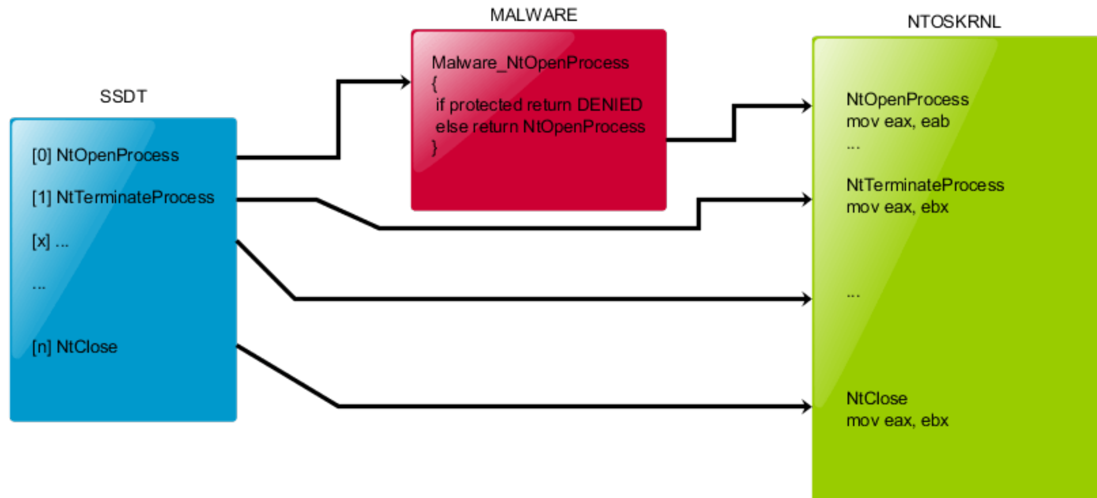


Figure 1.7: SSDT Hook [11]

- **IDT (Interrupt descriptor table) Hook**

  IDT je tabuľka prerušení, v ktorej sa nachádzajú smerníky na funkcie spracúvajúce konkrétne prerušenia. Prerušenia nastávajú pri určitých udalostiach, ako sú chyby alebo výnimky. Na starších systémoch sa systémové volania realizovali prostredníctvom prerušenia s kódom 0x2e, na rozdiel od novších, používajúcich inštrukciu *sysenter*. Nahradením smerníka v tejto tabuľke možno takéto volanie zachytiť a modifikovať.

- **Major Function Hooking**

  Súčasťou každého načítaného ovládača v pamäti je štruktúra _DRIVER_OBJECT. Táto štruktúra obsahuje tabuľku smerníkov na funkcie (nazývané IRP Major Functions). Keď ovládač dostane IRP od iného ovládača, zavolá sa funkcia z tejto tabuľky na základe typu prijatého IRP. V tabuľke 1.1 sú uvedené niektoré typy IRP, ktorých spracovanie musí každý nainštalovaný ovládač povinne implementovať.

Table 1.1: Niektoré typy IRP

Typ IRP	Kód
IRP_MJ_CREATE	0x00
IRP_MJ_CLOSE	0x02
IRP_MJ_READ	0x03
IRP_MJ_WRITE	0x04
IRP_MJ_DEVICE_CONTROL	0x0e
IRP_MJ_SCSI	0x0f
IRP_MJ_SHUTDOWN	0x10

Pre úplnosť ešte uvedieme, že volanie Major funkcií sa nikdy nevráti. Na získanie výsledku spracovania IRP ďalších ovládačov v reťazi sa používajú takzvané IO-Completion Routines, ktoré je možné špecifikovať v Major funkciách. Ak nasledujúci ovládač v reťazi skončí spracovanie IRP, volaním *IOCompletionRoutine()* odovzdá riadenie funkcii, ktorú špecifikoval predchádzajúci ovládač v reťazi. Takýmto spôsobom sa výsledok vstupno-výstupnej operácie posiela naspäť až k prvému ovládaču v reťazi.

Ako sme už skôr spomenuli, komunikácia aplikácií z User-módu s ovládačmi je možná prostredníctvom volania *DeviceIOControl()*, ktorým je možné poslať ovládaču kontrolný kód a ďalšie dáta. Pri volaní *DeviceIOControl()* systém vytvorí štruktúru IRP typu IRP_MJ_DEVICE_CONTROL, naplní ju potrebnými údajmi a odovzdá konkrétnemu ovládaču.

Ovládač tcpip.sys v systéme Windows implementuje sieťový TCP/IP stack. Modifikovaním funkcie spracúvajúcej IRP typu IRP_MJ_DEVICE_CONTROL je možné skryť sieťovú komunikáciu.

Malvér nahradí smerník v tabuľke smerníkom ukazujúcim na svoju funkciu, ktorá bude fungovať nasledovne. Ak aplikácia poslala IRP s kontrolným kódom IOCTL_TCP_QUERY_INFORMATION_EX (informácie o TCP spojeniach), nastaví IOCompletionRoutine() na svoju podvrhnutú funkciu, ktorá bude zavolaná na konci spracovania prijatého IRP. V tejto funkcii sa odstránia záznamy o TCP spojeniach, ktoré majú byť skryté a výsledok sa odošle ďalej. V opačnom prípade sa zavolá pôvodná funkcia. Modifikácia IRP handlera je znázornená na obrázku 1.8.

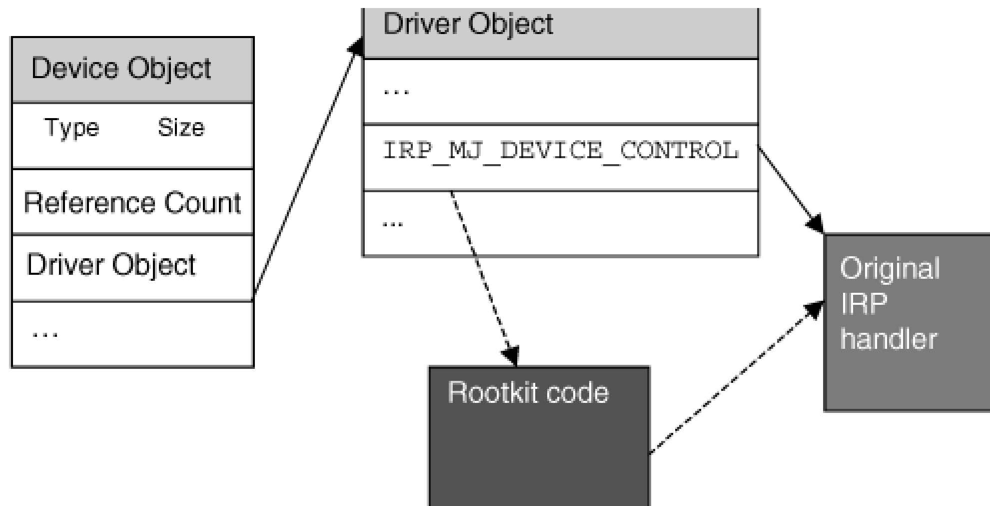


Figure 1.8: IRP Hook [10]

- **Runtime patch**: Táto metóda zodpovedá priamej modifikácii programového kódu, rovnako ako pri Inline hook v užívateľskom móde.

- **Layer Driver**: Spomenuli sme, že ovládače si navzájom preposielajú IRP a tvoria reťaz. Operačný systém umožňuje niektorým ovládačom, aby boli zaradené, prípadne odstránené z tejto reťaze. Tieto ovládače rôznym spôsobom spracúvajú IRP, môžu ho modifikovať, prípadne nepreposlať ďalej. Na obrázku 1.9 je znázornená reťaz ovládačov.

  Zaradením škodlivého ovládače do takejto reťaze sa malvér opäť dostáva k IRP, ktoré môže ľubovoľne modifikovať a dôsledok je rovnaký, ako sme popísali v predchádzajúcich prípadoch.

- **DKOM Direct kernel object manipulation**

  Príkladom môže byť skrývanie bežiaceho procesu modifikáciou interných systémových štruktúr. Windows si o každom procese udržiava štruktúru _EPROCESS, ktorá obsahuje informácie ako napríklad názov procesu, cestu k spustiteľnému súboru, stav a iné. Tieto štruktúry tvoria obojsmerný spájaný zoznam, ktorý je znázornený na obrázku 1.10. Vylúčením prvku z tohoto zoznamu možno docieliť, že konkrétny proces nebude viditeľný. Hoci sme takto modifikovali internú štruktúru, beh procesu tým neovplyvníme, pretože plánovanie úloh neprebieha prostredníctvom štruktúr _EPROCESS, ale prostredníctvom vlákien [3].

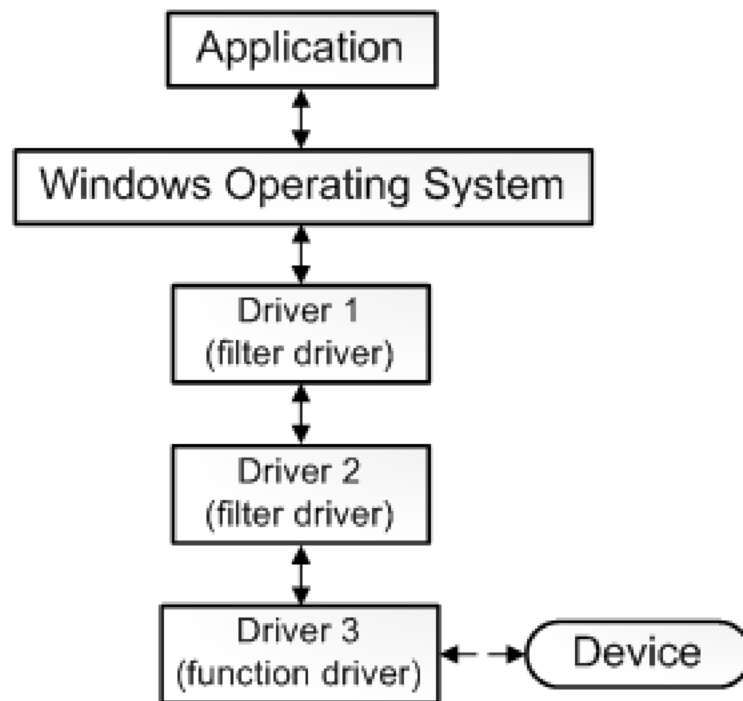


Figure 1.9: Filter Driver [18]

Figure 1.10: DKOM [13]

Modifikácia MBR

Master Boot Record je pomenovanie pre prvý sektor na disku. Pri zapnutí počítača a inicializácii zariadení BIOS odovzdá riadenie programu, ktorý sa nachádza v MBR. Úlohou tohto programu je spustiť načítanie operačného systému. Ak sa malvéru podarí prepísať kód zodpovedný za načítanie OS, môže zabezpečiť, že OS bude spustený až potom, čo bol vykonaný škodlivý kód, ktorý môže pred štartom systému ľubovoľne modifikovať disk, či iné časti OS.

V definícii slova rootkit sme spomenuli, že takýto druh malvéru pozostáva z viacerých druhov komponentov. Malvér infikujúci MBR spravidla zabezpečí iba jednu časť potrebnú pre svoju funkcionalitu. Typickým príkladom je vyradenie z činnosti ochranu pred načítaním nepodpísaného ovládača do jadra systému. Ďalší komponent (systémový ovládač) už priamo implementuje škodlivú činnosť, prípadne poskytuje služby ďalej škodlivej aplikácii, ktorá môže bežať aj v užívateľskom móde.

Uvedieme ešte jeden neformálny príklad. Za účelom skrývania súboru malvér zapísal súbor na koniec disku, ale priamo, mimo súborového systému. Súbor tak nebol v systéme viditeľný, ale dáta sa na disku nachádzali. Keďže malvér štartoval ešte pred operačným systémom, vytvoril ilúziu, že disk je trochu menší než aká bola realita. V prípade, že malvér potreboval prečítať svoje dáta z disku, vedel kde sa nachádzajú a akým spôsobom k nim pristupovať.

Firmware malvér

Tendencia posúvať sa stále na nižšiu úroveň išla až tak ďaleko, že vznikol škodlivý kód určený priamo pre periférne zariadenia a ich mikrokód. Takýmto príkladom môže byť dielo skupiny Equation Group, ktorá vytvorila kód modifikujúci firmware diskov, ktorý umožňoval vytvorenie skrytých partícií a tak zabezpečil neviditeľnosť súborov.

1.3 Patch Guard

V predchádzajúcej časti sme popísali techniky, ktorými malvér môže získať kontrolu nad systémom. Väčšinou išlo o modifikáciu pôvodného programového kódu, alebo o zmenu smerníka v nejakej dátovej štruktúre, ktorý ukazoval na funkciu, ktorá sa mala pri určitej udalosti vykonať. S príchodom týchto techník vznikali otázky, ako sa brániť pred modifikáciami kódu v jadre OS.

Patch Guard alebo Kernel Patch Protection (KPP) je technológia, ktorú zaviedol Microsoft v roku 2005 do 64 bitoých verzií OS. Úlohou je kontrolovať integritu kľúčových štruktúr operačného systému, medzi ktoré patria:

- System Service Descriptor Table (SSDT)

- Interrupt Descriptor Table (IDT)
- Global Descriptor Table (GDT)
- Windows Kernel moduly
- NDIS vrstva
- HAL vrstva

KPP pravidelne počíta a porovnáva kontrolné sumy týchto štruktúr a kódu. Pri zistení modifikácie, operačný systém spadne s chybou
CRITICAL_STRUCTURE_CORRUPTION.

Zavedenie KPP znamenalo problém pre niektoré softvérové spoločnosti, ktorých produkty využívali k svojej funkčnosti modifikáciu štruktúr OS, hoci s dobrým úmyslom. Keďže KPP prinieslo problém pre tvorcov malvéru, začali sa objavovať techniky ako obísť KPP, ktorá žiaľ funguje na princípe „security through obscurity". Reverzným inžinierstvom sa podarilo zistiť princíp a funkcionalitu KPP, a tak výsledkom týchto činností bolo, že za funkcionalitou KPP je obfuskovaný kód s množstvom anti-debug a anti-reverznými technikami. Hoci najnovšia verzia tejto ochrany je naozaj pokročilá, časom sa vždy objaví malvér, ktorému sa táto ochrana podarí prekonať. Na druhej strane treba povedať, že modifikácia častí OS už nie je taká jednoduchá, ako to bolo v minulosti.

Chapter 2

Virtualizácia

Trendom moderných informačných systémov je rozsiahle využívanie virtualizačných technológii. V tejto kapitole predstavíme technológiu virtualizácie operačných systémov, popíšeme princípy akými funguje a objasníme výhody použitia týchto technológií.

2.1 Stručný popis

Virtualizácia je technika, pri ktorej operačný systém, alebo špeciálny softvér vytvorí virtuálny hardvér, ktorý fyzicky neexistuje, ale javí sa ako reálny so všetkými vlastnosťami reálneho hardvéru. Virtualizácia umožňuje beh viacerých nezávislých operačných systémov, zdieľajúcich spoločný hardvér (vo virtuálnych počítačoch) bez vzájomného ovplyvnenia sa. Operačnému systému, ktorý beží takýmto spôsobom, je vytvorená ilúzia, že disponuje prostriedkami ako sú procesor, pamäť, vstupno-výstupné zariadenia, disky, či sieť a tieto prostriedky môže využívať rovnako, akoby pracoval priamo so skutočným hardvérom. V skutočnosti však operačné systémy zdieľajú reálne prostriedky a musia sa o ne deliť. Operačné systémy sú navzájom izolované a „nevedia“ o sebe. To znamená, že sa nemôže stať, že by jeden operačný systém zasahoval do činnosti iného systému a mohol tak manipulovať s jeho pamäťou, alebo inými prostriedkami. Takto fungujúci operačný systém označujeme ako virtualizovaný.

Na obrázku 2.1 je znázornený princíp virtualizácie. Spomenuli sme, že virtualizované operačné systémy sa musia deliť o prostriedky, ktoré používajú. O prideľovanie jednotlivých prostriedkov pre OS sa stará hypervisor, alebo VMM (Virtual Machine Monitor). VMM je softvér, ktorý beží pod virtualizovanými operačnými systémami a vytvára takzvanú medzivrstvu medzi skutočným hardvérom a virtuálnymi počítačmi ktoré sú označované ako Guest. [1].

Existujú dva typy VMM. V prvom prípade beží VMM priamo na hardvéri a hov-

[1] V nasledujúcom texte budeme používať pojem hypervisor a Virtual Machine Monitor (VMM), ktoré budeme zamieňať. Myslíme tým však to isté - softvér, ktorý riadi virtualizáciu.

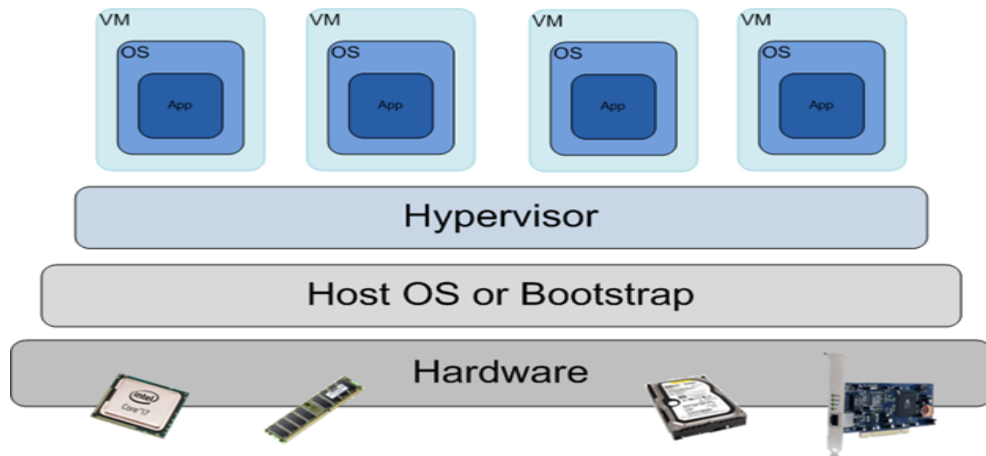


Figure 2.1: Virtualizácia [25].

oríme o hypervisore typu 1, ktorého príkladom môže byť Vmware ESXi produkt. Hypervisor typu 2 beží pod operačným systémom ako obyčajná aplikácia, a tento operačný systém sa nazýva Host OS. Príkladom tohto typu hypervisora je Vmware Workstation alebo Virtualbox.

Keďže s reálnym hardvérom pracuje priamo iba hypervisor, ktorý riadi celú virtualizáciu, virtualizovaný operačný systém používajúci iba virtuálny hardvér musí byť nejakým spôsobom obmedzený. Vykonanie vstupno-výstupných a iných privilegovaných inštrukcií virtualizovaným OS musí byť odchytené a spracované hypervisorom. V opačnom prípade by Guest OS mohol napríklad manipulovať s pamäťou iného Guest OS.

Počas behu nevirtualizovaného OS sú všetky inštrukcie programového kódu vykonávané priamo procesorom, čo je znázornené na obrázku 2.2a. Privilegované inštrukcie pri virtualizácii, však musia byť spracované odlišne a spôsob spracovania závisí od druhu použitého typu virtualizácie. V nasledujúcom texte popíšeme druhy virtualizácie:

- Virtualizácia s asistenciou operačného systému / Paravirtualizácia:

  Pri paravirtualizácii je Guest OS modifikovaný tak, že vykonanie privilegovanej inštrukcie v kóde je nahradené tzv. hypervolaním. Situáciu znázorňuje obrázok 2.2b. Prostredníctvom tohto volania komunikuje Guest OS s hypervisorom, ktorý potom požiadavku spracuje a vykonanie privilegovanej inštrukcie odsimuluje. Hypervisor ponúka rozhranie aj pre ďalšie kritické operácie v rámci jadra OS ako správa pamäte, spracovanie prerušení alebo časové operácie. Nevýhoda tohto prístupu k virtualizácii spočíva v tom, že je nutné, aby každý operačný systém určený pre beh vo virtuálnom stroji bol modifikovaný. Ďalším problémom je, že OS si je „vedomý“ tejto modifikácie, a teda aj toho, že beží vo virtuálnom prostredí, čo však často nie je žiadúce.

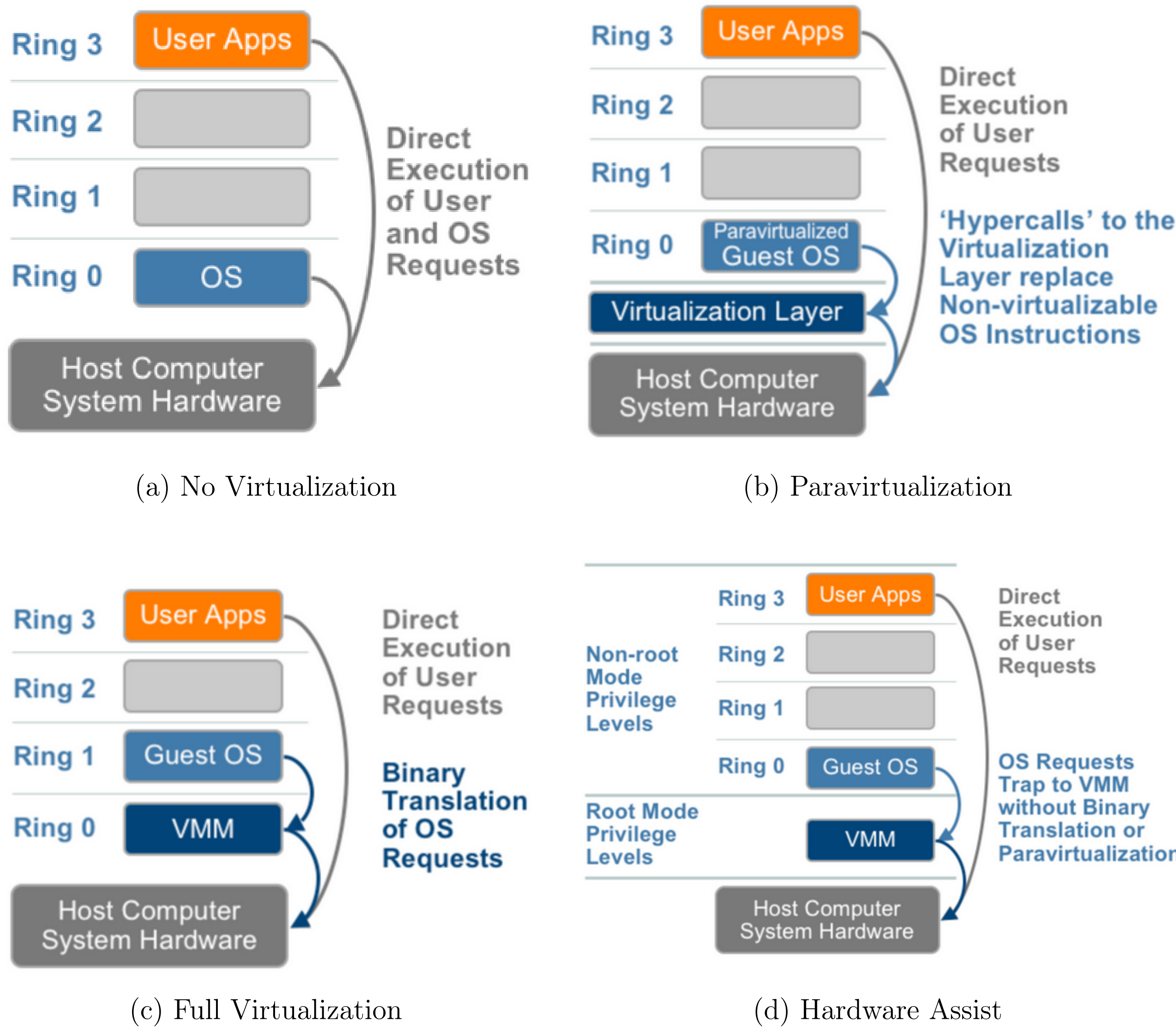


Figure 2.2: Druhy virtualizácie [34]

- Plná virtualizácia:

  Na obrázku 2.2c je znázornený princíp plnej virtualizácie. Kód z úžívateľskej časti (Ring 3) je vykonávaný priamo procesorom, čo má za následok rýchlosť pri behu virtuálneho stroja. Guest OS a všetok privilegovaný kód (Ring 0), však nebeží na tejto úrovni, ale na úrovni Ring 1. Na úrovni Ring 0 beží hypervisor. Privilegované inštrukcie sú odchytené hypervisorom, ktorý danú inštrukciu odemuluje, nahradí inou sekvenciou inštrukcií. Hovoríme o binárnom preklade.

  Výhodou je, že operačný systém, ktorý beží vo virtuálnom stroji nie je potrebné modifikovať a OS navyše „nevie“ o tom, že beží vo virtuálnom prostredí. Nevýhoda tohoto prístupu, však spočíva v tom, že programový kód je potrebné počas behu prekladať, čo vedie k spomaleniu vykonávania oproti nevirtualizovanému behu.

- Hardvérová virtualizácia:

  Virtualizáciu možno dosiahnuť softvérovou alebo hardvérovou implementáciou. Nevýhoda softvérovej virtualizácie spočíva v jej rýchlosti. Keďže vykonávanie niektorých inštrukcií musí byť simulované, čo zaberá nejaký čas, výrobcovia procesorov sa preto rozhodli vložiť podporu virtualizácie priamo do procesorov a vznikla tak podpora pre hardvérovú virtualizáciu. Technológia, ktorú priniesla spoločnosť AMD sa nazýva AMD-V a v prípade Intelu to je Intel VT-x. Akcelerácia je umožnená pridaním nových inštrukcií určených pre virtualizáciu a špeciálneho stavu pre procesor, z ktorého je virtualizácia riadená. Ilustrácia hardvérovej virtualizácie je na obrázku 2.2d a bližšie ju popíšeme v časti 2.2.

Na záver tejto časti uvedieme ešte niekoľko výhod použitia virtualizácie v praxi:

- Izolovanosť OS:

  Je dôležitá z hľadiska bezpečnosti. Pád jedného OS neovplyvní ostatné systémy. Tie navyše môžu byť úplne odlišné, teda naraz môžu byť virtualizované systémy ako Windows, Linux, či MacOS.

- Jednoduchá inštalácia a záloha:

  Virtualizovaný operačný systém je zložený z niekoľkých konfiguračných a dátových súborov. Tieto súbory je možné jednoducho skopírovať, a v prípade poškodenia OS rýchlo obnoviť. Navyše nová inštalácia systému zo zálohy je oveľa rýchlejšia ako inštalovanie OS od začiatku spolu s aplikačnými programami a aktualizáciami. Takto rýchlo obnoviteľný systém je využiteľný pre testovanie rozličného softvéru až po spúšťanie škodlivého kódu pre jeho analýzu. Ďalšou vlastnosťou je jednoduchá migrácia na iný hardvér, v prípade poruchy hostiteľského počítača.

- Lepšie využitie hardvéru:

  Keďže niekoľko rôznych OS môže bežať na jednom kuse hardvéru, znižujú sa náklady na správu fyzických zariadení ako sú chladenie, elektrická energia (rozdeľovanie výkonu medzi rôznymi OS), fyzická ochrana.

2.2 VT-x

VT-x je technológia od spoločnosti Intel umožňujúca hardvérovú podporu pre virtualizáciu. Rozšírenie, ktoré Intel pridal do procesorov, sa nazýva Virtual Machine Extensions (VMX) a pridáva nové inštrukcie uvedené v tabuľke 2.1.

Table 2.1: VMX Inštrukcie

Inštrukcia	**Akcia**
VMPTRLD	Load VMCS pointer
VMPTRST	Store Pointer to Virtual-Machine Control Structure
VMCLEAR	Set Launch State of Referenced VMCS to "Clear"
VMREAD	Read a component from a VMCS
VMWRITE	Write component to a VMCS
VMCALL	Call VMM for service
VMFUNC	Software in VMX non-root may invoke a VM function without VM exit
VMLAUNCH	Launches a virtual machine managed by the VMCS
VMRESUME	Resumes a virtual machine managed by the VMCS
VMXOFF	Causes the processor to leave VMX operation
VMXON	Causes a logical processorto enter VMX root operation
INVEPT	Invalidates entries in the TLBs and paging-structure caches
INVVPID	Invalidates entries in the TLBs and paging-structure caches

Procesor pri zapnutej virtualizácií môže pracovať v dvoch módoch. Prvý je neprivilegovaný, non-root mód, v ktorom beží Guest softvér. Operačný systém bežiaci v tomto móde sa nazýva Guest OS. Procesor bežiaci v non-root móde je obmedzený a vykonávanie niektorých inštrukcií nie je povolené v záujme podpory virtualizácie. Guest OS je vytvorená ilúzia, že má prístup ku všetkému hardvéru, aj k privilegovaným inštrukciam a vrámci non-root módu beží v ring 0. Aplikačné programy bežia v režime ring 3. Pri zapnutej hardvérovej virtualizácii má teda operačný systém vytvorenú ilúziu, že beží priamo na reálnom hardvéri bez nutnosti modifikácie tohto OS.

Druhý mód je privilegovaný, root mód, v ktorom beží hypervisor. Hypervisor bežiaci v root móde má plný prístup k procesoru, a tým pádom môže riadiť beh virtualizovaných systémov. Ako sme v úvode spomenuli, to, že sme schopní spúšťať kód

s väčšími privilégiami ako operačný systém nám neskôr umožní kontrolovať beh OS a aplikácií takým spôsobom, ktorí by nebol možný v prípade použitia systémového ovládača v jadre OS.

Prechod do non-root módu sa nazýva VM-ENTRY a prepnutie stavu procesora z non-root do root módu sa nazýva VM-EXIT. Keďže procesor je v non-root móde obmedzený, niektoré udalosti a vykonanie privilegovaných inštrukcií spôsobí VM-EXIT, kedy procesor prejde do root módu a ku slovu sa dostane VMM. Počas behu procesora, pri zapnutej virtualizácii, dochádza teda k striedaniu režimov root a non-root.

Udalosti počas virtualizácie znázorňuje obrázok 2.3. Procesor zapne režim virtualizácie a prejde do VMX módu (root) vykonaním inštrukcie VMXON. VMM potom môže odovzdať riadenie Guest OS vykonaním inštrukcie VMLAUNCH (prvé spustenie Guest) alebo VMRESUME (vrátenie riadenia po VM-EXIT). Prostredníctvom VM-EXIT získa VMM riadenie naspäť, a na základe udalosti, ktorá vyvolala VM-EXIT vykoná kroky pre jej spracovanie, vráti riadenie opäť Guest OS, alebo ukončí režim virtualizácie procesora inštrukciou VMXOFF a procesor prejde do pôvodného režimu vykonávania.

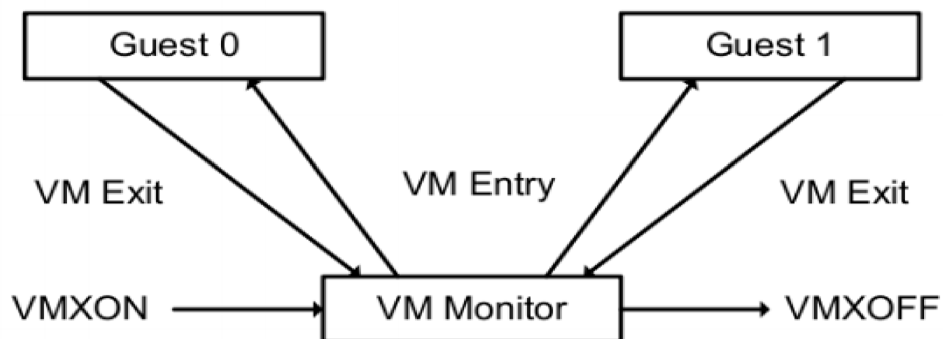


Figure 2.3: Interakcia medzi VMM a Guestom [24].

2.3 Virtual-machine control data structure

Procesor počas VMX operácií používa štruktúru Virtual machine control data structure (VMCS), v ktorej odkladá dáta potrebné pre beh virtualizovaných OS. Táto štruktúra riadi správanie procesora počas virtualizácie a jednotlivé prechody medzi VMX operáciami. VMM môže používať viacero VMCS štruktúr pre virtuálny stroj s viacerými logickými procesormi (pre každý procesor jedna štruktúra). Štruktúra VMCS sa nachádza v pamäti, ktorá sa nazýva VMCS región. Prístup k tejto pamäti je možný prostredníctvom 64-bitového smerníka (VMCS smerník).

VMCS štruktúra sa môže nachádzať vo viacerých stavoch. Celkový stav je určený kombináciou troch skupín menších stavov, uvedených v tabuľke 2.2.

Table 2.2: VMX Stavy

Skupina	Stav A	Stav B
1	Active	Inactive
2	Current	Not Current
3	Launched	Clear

Štruktúra môže byť v stave *Active* alebo *Inactive* a najviac jedna zo štruktúr v stave *Active* môže byť *Current*, ostatné sú *Not Current*. Inštrukcie VMLAUNCH, VMREAD, VMRESUME a VMWRITE operujú iba na VMCS štruktúre v stave *Active* [24]. Ďalej môže byť štruktúra v stave *Clear* alebo *Launched*.

Prechody medzi jednotlivými stavmi sú znázornené na obrázku 2.4.

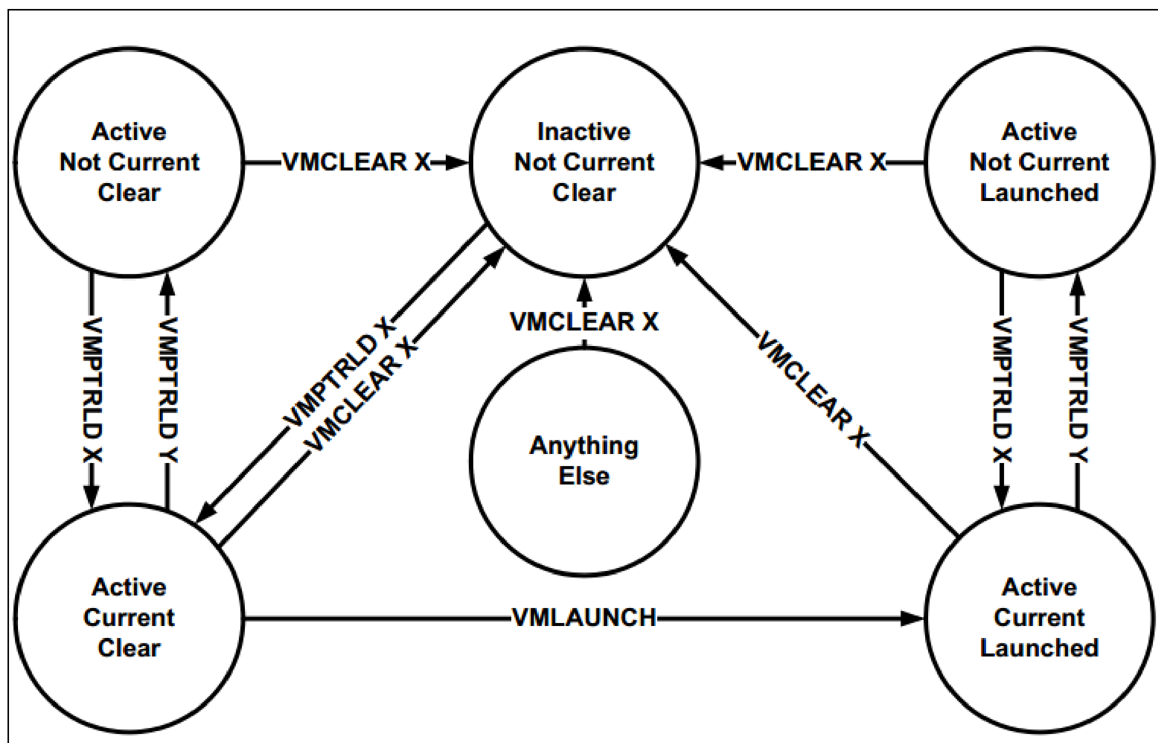


Figure 2.4: Virtualizácia [24].

Inštrukcia VMCLEAR zmení stav VMCS na *Inactive, Not Current, Clear*. Ak bola štruktúra VMCS *Current* na logickom procesore, procesor nebude mať po vykonaní inštrukcie žiadnu štruktúru v stave *Current*.

Po vykonaní inštrukcie VMPTRLD bude VMCS v stave *Active* a *Current* na danom logickom procesore, ale pôvodná štruktúra, ktorá bola v stave *Current* zmení stav na *Not Current*.

Vykonaním VMLAUNCH prejde štruktúra zo stavu *Active, Current, Clear* do stavu *Active, Current, Launched*, čo zodpovedá prvému spusteniu VM.

VMCS štruktúra obsahuje dáta, ktoré možno rozdeliť do týchto skupín:

- Guest-state area: Stav procesora Guest OS je uložený do Guest-state area počas VM-EXIT a obnovený keď nastane VM Entry.
- Host-state area: Procesor obnoví svoj stav z Host-state pri VM-EXIT.
- VM-EXIT control fields: Tu sa nastavuje správanie procesora v non-root móde a určuje sa, ktoré udalosti vyvolajú VM Exit.
- VM-ENTRY control fields: Dáta určujú, kedy nastane VM Entry.
- VM-EXIT information fields: Dáta obsahujú informácie o príčinách, pri ktorých došlo k VM Exit udalosti.

Hypervisor teda prostredníctvom VMCS určuje, pri ktorých udalostiach sa procesor prepne z non-root módu do root a na základe informácií o príčinách prepnutia stavu vykoná ďalšie kroky.

V tabuľke 2.3 sú uvedené niektoré udalosti, ktoré spôsobia VM Exit.

Table 2.3: Udalosti spôsobujúce VM Exit [24].

CPUID	Guest sa pokúsil vykonať inštrukciu CPUID.
HLT	Guest sa pokúsil vykonať inštrukciu HLT.
RDTSC	Guest sa pokúsil vykonať inštrukciu RDTSC.
Prístup ku kontrolným registrom	Guest prístupil k registrom CR0 CR3 CR4 CR8 prostredníctvom inštrukcií CLTS, LMSE, MOV CR.
I/O Inštrukcia	Guest sa pokúsil vykonať vstupno-výstupnú inštrukciu.
Prístup ku GDTR a IDTR	Guest sa pokúsil pristúpiť k tabuľke prerušení alebo globálnej tabuľke popisovačov.
EPT violation	**Prístup na fyzickú pamäť Guest OS bol zakázaný konfiguráciou EPT.**

2.4 Virtualizácia a stránkovanie

V moderných operačných systémoch bol zavedený koncept virtuálnej pamäte. Fyzická pamäť je logicky rozdelená na časti rovnakej veľkosti (väčšinou 4KB), ktoré sa nazývajú stránky. Operačný systém programu pri spustení pridelí časť pamäte. Táto pamäť sa procesu javí ako jeden súvislý úsek, a nazýva sa virtuálna pamäť. Adresa vo virtuálnej pamäti však nemusí zodpovedať tej istej adrese vo fyzickej pamäti, a reálne

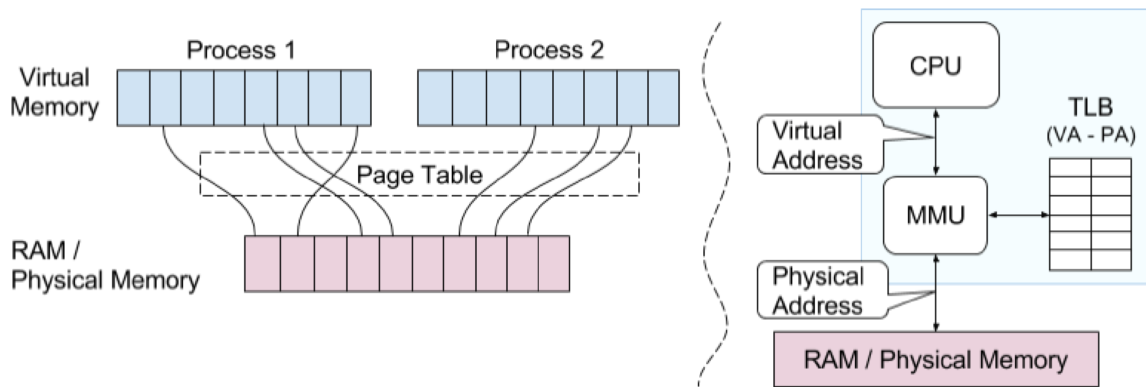


Figure 2.5: Virtuálna pamäť [32].

môže byť táto pamäť zložená z úsekov, ktoré sa môžu nachádzať v rôznych častiach fyzickej pamäte. Príklad je uvedený na obrázku 2.5.

Operačný systém si pre každý proces udržiava štruktúru nazývanú tabuľka stránok (Page Table), na základe ktorej je možné priradiť stránku virtuálnej pamäte k stránke fyzickej. Pri prístupe programu do pamäte je teda potrebné urobiť premapovanie. Procesory obsahujú jednotku pre správu pamäte (MMU - Memory Management Unit), ktorá robí preklad adries z virtuálnych na fyzické použitím tabuľky stránok. Pre zefektívnenie práce s pamäťou sa získané mapovania z tabuľky stránok ukladajú to cache, nazývanej TLB (Tralnslation Lookaside Buffer). Pri mapovaní virtuálnej adresy (VA) na fyzickú (PA) sa procesor pozrie najprv do TLB, či už tam náhodou záznam nie je uložený. Ak áno, použije sa tento záznam, inak sa záznam hľadá v tabuľke stránok.

Počas virtualizácie chceme zabezpečiť, aby operačný systém bežiaci vo virtuálnom prostredí nemal priamy prístup k MMU a nemohol voľne pristupovať k fyzickej pamäti, ale aby správu pamäte mal na starosti hypervisor. Privilegované inštrukcie, ktoré narábajú s MMU sú teda odchytené a k slovu sa dostane hypervisor, ktorý musí situáciu ošetriť. Mapovanie pamäte pri virtualizácii je uvedené na obrázku 2.6.

Guest OS používa svoju tabuľku stránok pre mapovanie virtuálnych adries na fyzické. O týchto adresách sa Guest domnieva, že zodpovedajú skutočne fyzickým adresám v pamäti, ale hypervisor má svoju vlastnú tabuľku stránok a robí ďalšie premapovanie z Guest fyzickej adresy na reálnu (Host fyzickú) adresu.

Aby tento koncept fungoval efektívne, virtualizovanie virtuálnej pamäte prebieha nasledovne. Tabuľka stránok, ktorú používa Guest OS je určená iba na čítanie. Tabuľka stránok, ktorú používa hypervisor sa nazýva zatienená tabuľka stránok (shadow page table) a táto tabuľka obsahuje kópiu tabuľky, ktorú má Guest a navyše mapovanie na reálnu fyzickú adresu v pamäti. Situácia je znázornená na obrázku 2.7.

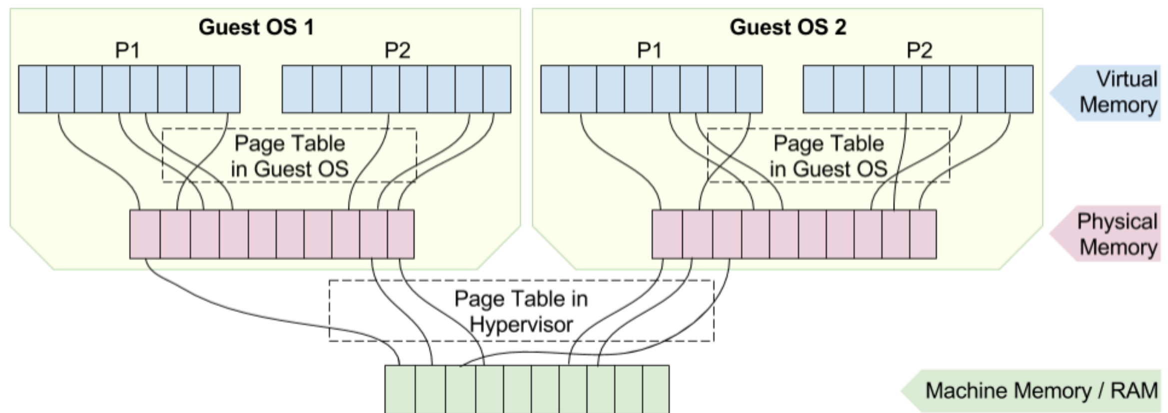


Figure 2.6: Virtualizácia [32].

Keďže zatienená tabuľka stránok je rovnaká ako tabuľka, ktorú má Guest (až na ďalšie mapovanie) pri prístupe do pamäte, pri ktorom nie je potrebné modifikovať tabuľku stránok, sa použije rovno Shadow page table.

Ak sa Guest pokúsi modifikovať svoju tabuľku stránok, napríklad pri udalosti *Page Fault*, táto udalosť je privilegovaná a spôsobí VM Exit, čím sa riadenie odovzdá hypervisoru. Hypervisor urobí potrebnú modifikáciu vo svojej tabuľke stránok a následne v tabuľke stránok Guest OS, aby udržal tabuľky synchronizované a následne mu vráti riadenie.

Pre lepšiu predstavu príklad rozpíšeme detailnejšie:

1. Program pristúpi na adresu 0xdeadbeef a zodpovedajúca pamäťová stránka nie je namapovaná.

2. Nastala chyba *Page Fault* a operačný systém sa teda pokúsi modifikovať tabuľku stránok a namapovať stránku do adresného priestoru daného procesu.

3. Akákoľvek modifikácia tabuľky stránok však spôsobí udalosť VM-EXIT a riadenie sa odovzdá hypervisoru.

4. Hypervisor urobí modifikáciu, ktorú sa Guest pokúsil urobiť najprv vo svojej tabuľke, a potom v tabuľke, ktorú má Guest a odovzdá mu riadenie naspäť.

5. Guest OS pokračuje vo vykonávaní ďalej a obe tabuľky stránok sú naďalej synchronizované.

Keďže *Page Fault* je udalosť, ktorá vzniká v systéme pomerne často a hypervisor musí túto udalosť ošetriť vždy keď nastane, dochádza k častému striedaniu udalostí VM-EXIT a VM-EMTRY, čo vedie k rapídnemu spomaleniu behu OS.

Ďalším problémom je, že každý proces má svoju vlastnú tabuľku stránok a pri udalosti Context Switch (zmena procesu, ktorý bude aktuálne bežať) je potrebné, aby

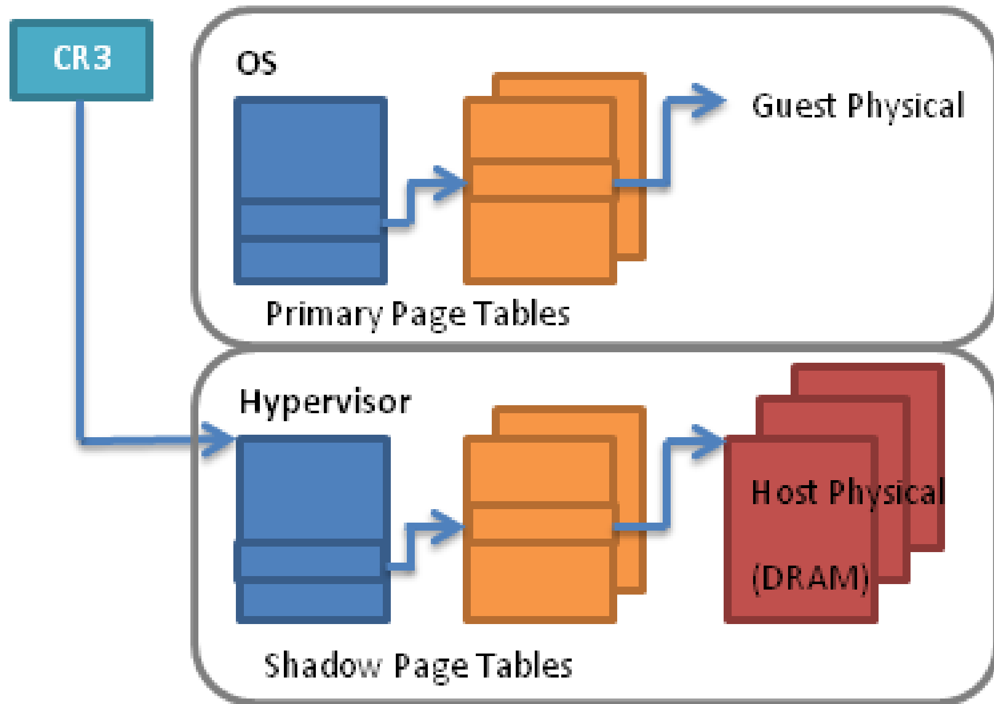


Figure 2.7: Shadow Page table [12].

sa zmenila aj Shadow page table pre tento proces. Tým pádom sa znova musí k činnosti dostať hypervisor a urobiť potrebné zmeny.

Do procesorov tak bola pridaná hardvérová podpora priameho prekladu Guest fyzickej pamäte na reálnu fyzickú pamäť. Riadenie premapovania má na starosti hypervisor, ale samotný preklad sa už deje bez nutnosti zásahu hypervisora, čo vedie k urýchleniu procesu pri prístupe do pamäte. Toto rozšírenie pre preklad virtuálnych fyzických adries sa nazýva rozšírená tabuľka stránok (Extended Page Table - EPT) a ilustrácia je na obrázku 2.8.

Procesor môže naraz pracovať s dvoma tabuľkami stránok. TLB ukladá mapovanie virtuálnej adresy v Guest OS priamo na reálnu fyzickú adresu (MA - Machine Address). Pri modifikácii tabuľky stránok hypervisor vôbec nemusí zasahovať do správy pamäte, čo značným spôsobom urýchľuje beh virtualizovaného systému oproti predošlému spôsobu virtualizácie.

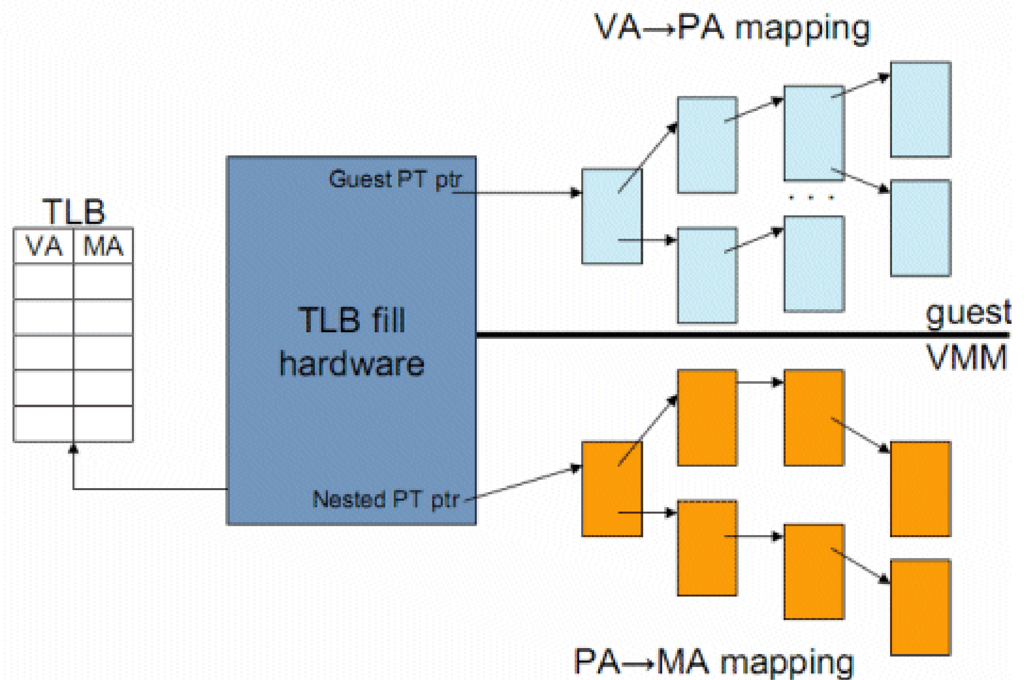


Figure 2.8: Shadow Page table [23].

2.4.1 Skrývanie pamäte

Detekcia škodlivého softvéru bežiaceho v jadre OS je náročná, pretože program beží s vysokými právami, má prístup ku kľúčovým štruktúram OS a môže vykonávať privilegované inštrukcie. Modifikáciou niektorých funkcií dokáže zabezpečiť svoju neviditeľnosť vrámci OS a znefunkčniť bezpečnostné mechanizmy samotného OS. Preto, ak chceme zabezpečiť detekciu škodlivého kódu s takýmito privilégiami, ktorý je schopný aj aktívne sa brániť, je potrebné vedieť pristupovať k pamäti jadra OS a upravovať ju takým spôsobom, aby si OS spolu s ostatným privilegovaným kódom nebol žiadnych činností vedomý.

Ako sme už raz spomenuli, niektoré inštrukcie a udalosti môžu vyvolať VM-EXIT, čo spôsobí, že sa k činnosti dostane hypervisor, ktorý situáciu vhodným spôsobom ošetrí.

Počas prieskumu oblasti virtualizácie sme sa dostali k implementácii[33] s názvom DdiMon, ktorá využíva virtualizáciu a vhodnú konfiguráciu EPT za účelom vytvorenia neviditeľných hookov v systéme Windows. Neviditeľný v tomto prípade znamená, že OS ani iná aplikácia bežiaca na úrovni OS, nie je schopná detegovať modifikáciu kódu štandardným spôsobom (čítanie pamäťovej stránky a následná kontrola integrity).

Túto implementáciu sme sa rozhodli využiť v našej práci za účelom detekcie sieťovej komunikácie, ktorá môže byť generovaná ovládačom pracujúcim v jadre operačného

systému. Tento ovládač môže modifikovať interné štruktúry OS takým spôsobom, že táto komunikácia v systéme je neviditeľná. Bližšie sa tejto téme budeme venovať v ďalších kapitolách. Na záver tejto časti ešte popíšeme spôsob akým DdiMon pracuje.

Hook konkrétnej funkcie, ktorá sa nachádza na nejakej pamäťovej stránke je implementovaný nasledovne. DdiMon vytvorí ďalšie dve pamäťové stránky s rovnakým obsahom (stránka na vykonávanie sa mierne líši, čo popíšeme neskôr) ako bola pôvodná, avšak jednu určenú iba na čítanie a zápis. Druhá stránka bude určená iba na vykonávanie kódu, obsahujúca modifikáciu (hook nejakej funkcie). Pôvodná pamäťová stránka sa nebude používať a hypervisor vhodným spôsobom „podhodí“ vždy jednu z dvoch novovytvorených pamäťových stránok. Pri vykonávaní kódu sa teda použije modifikovaná pamäťová stránka a pri pokuse o čítanie/zápis hypervisor prostredníctvom zmeny konfigurácie EPT vráti stránku bez modifikácie. Teraz uvedieme ešte konkrétny príklad, aby sme mohli popísať aj hlbšie detaily implementácie.

Ak bola napríklad hooknutá funkcia na adrese 0x1234, hypervisor vytvorí dve pamäťové stránky s rovnakým obsahom, ale odlišnými právami, viď. tabuľka 2.4.

Table 2.4: Pamäťové stránky pri hooknutí funkcie [33]

	Pokus o prístup na adresu	Reálna adresa, kam sa pristúpilo
Hypervisor	0x1234	0x1234 (všetky práva)
Guest	0x1234	0xa234 (iba vykonávanie kódu)
Guest	0x1234	0xb234 (iba čítanie/zápis)

- **Východiskový stav**

  DdiMon nastaví konfiguráciu v EPT tak, aby sa stránka s adresami 0x1000-0x1fff mapovala na stránku 0xa000 a zakáže čítanie a zápis na túto stránku.

- **Čítanie / Zápis**

  Pri pokuse o čítanie alebo zápis vznikne chyba EPT Violation a VM-EXIT. Hypervisor modifikuje EPT položku tak, aby sa použila stránka na adrese 0xb000, ktorá má povolený zápis a čítanie. Zároveň hypervisor nastaví MTF (Monitor Trap Flag). MTF spôsobí VM-EXIT v momente ako Guest vykoná inštrukciu.

  Guest teraz môže ľubovoľne čítať a zapisovať do pamäte, pracujúc so stránkou na to určenou. Ak sa Guest pokúsi vykonať kód na tejto stránke, vznikne udalosť MTF VM-EXIT a riadenie sa opäť predá hypervisoru, ktorý zruší MTF Flag a zmení mapovanie stránky späť na pôvodnú 0xa000, ktorá je určená na vykonávanie kódu.

- **Vykonávanie kódu**

Hook je implementovaný takým spôsobom, že prvá inštrukcia vo funkcii je prepísaná na inštrukciu int3 (prerušenie). Pri prerušení vznikne VM-EXIT, hypervisor sa pozrie do tabuľky, v ktorej na základe adresy, kde vzniklo prerušenie, nájde príslušný hook handler. Ten sa vykoná a riadenie sa vráti pôvodnej funkcii.

Chapter 3

Sieťová komunikácia

3.1 Network Driver Interface Specification

Network Driver InterfaceSpecification (NDIS) je špecifikácia, ktorá popisuje, akým spôsobom komunikujú sieťové karty s ovládačmi a operačným systémom Windows. NDIS definuje štandardné rozhrania pre komunikáciu medzi vrstvami ovládačov, ako sú ovládače pre protokoly, filter ovládače, ovládače sieťových kariet a vytvára abstrakciu nad hardvérom. NDIS spravuje stav a parametre jednotlivých sieťových ovládačov, ktoré zahŕňajú smerníky na funkcie, handle a ďalšie systémové premenné. Windows implementuje túto špecifikáciu v module ndis.sys, nazývanom aj NDIS knižnica.

Štruktúra NDIS je ilustrovaná na obrázku 3.1.

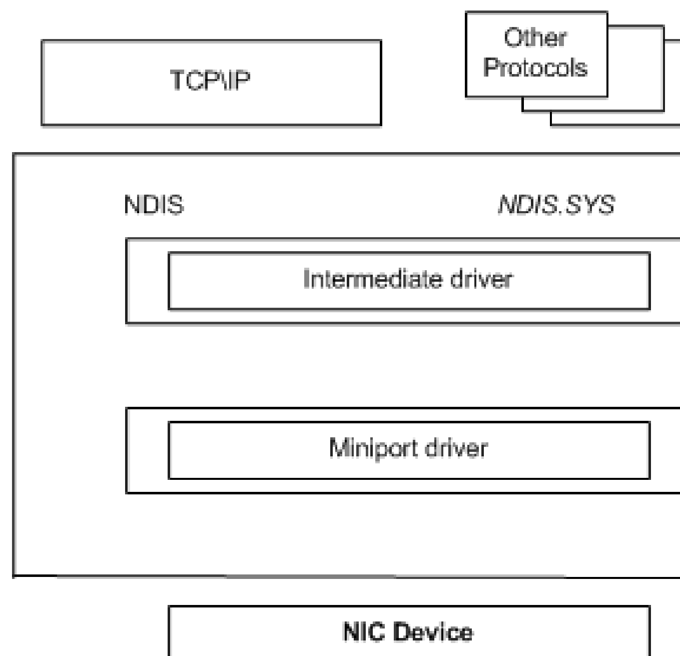


Figure 3.1: Štruktúra NDIS [28]

NDIS rozlišuje nasledovné typy sieťových ovládačov:

- **Protocol (Transport) drivers**

  Ovládače, ktoré sa nachádzajú v hierarchii sieťových ovládačov najvyššie. Príkladom je protokol TCP/IP. Protokol ovládač alokuje dátovú štruktúru pre paket, nakopíruje dáta od aplikácie a odošle prostredníctvom *Ndis*Xxx** funkcií ďalšiemu ovládaču na nižšej úrovni. Protokoly tohoto typu zároveň exportujú funkcie *ProtocolXxx*, ktoré používa NDIS, keď preposiela dáta od ovládača z nižšej úrovne.

- **Intermediate drivers**

  Tieto ovládače sa nachádzajú v hierarchii medzi Protocol a Miniport ovládačmi a musia poskytovať API, ktoré používa NDIS na komunikáciu v mene ďalších ovládačov v hierarchii. NDIS odovzdá dáta od Transport ovládača volaním *MiniportXxx* a od Miniport ovládača volaním príslušnej *ProtocolXxx* funkcie. Ilustrácia je na obrázku 3.2a.

  Intermediate ovládač môže exportovať jeden alebo viac virtuálnych adaptérov, ktoré sa javia Protokol ovládačom ako fyzická sieťová karta a ku ktorým sa môže Protokol ovládač priviazať.

- **Filter drivers**

  Do hierarchie sieťových ovládačov je možne zaradiť ešte takzvané Filter ovládače nachádzajúce sa vo vrstvách medzi Miniport adaptérmi a Viazaniami protokolov, čo je znázornené na obrázku 3.2b [1].

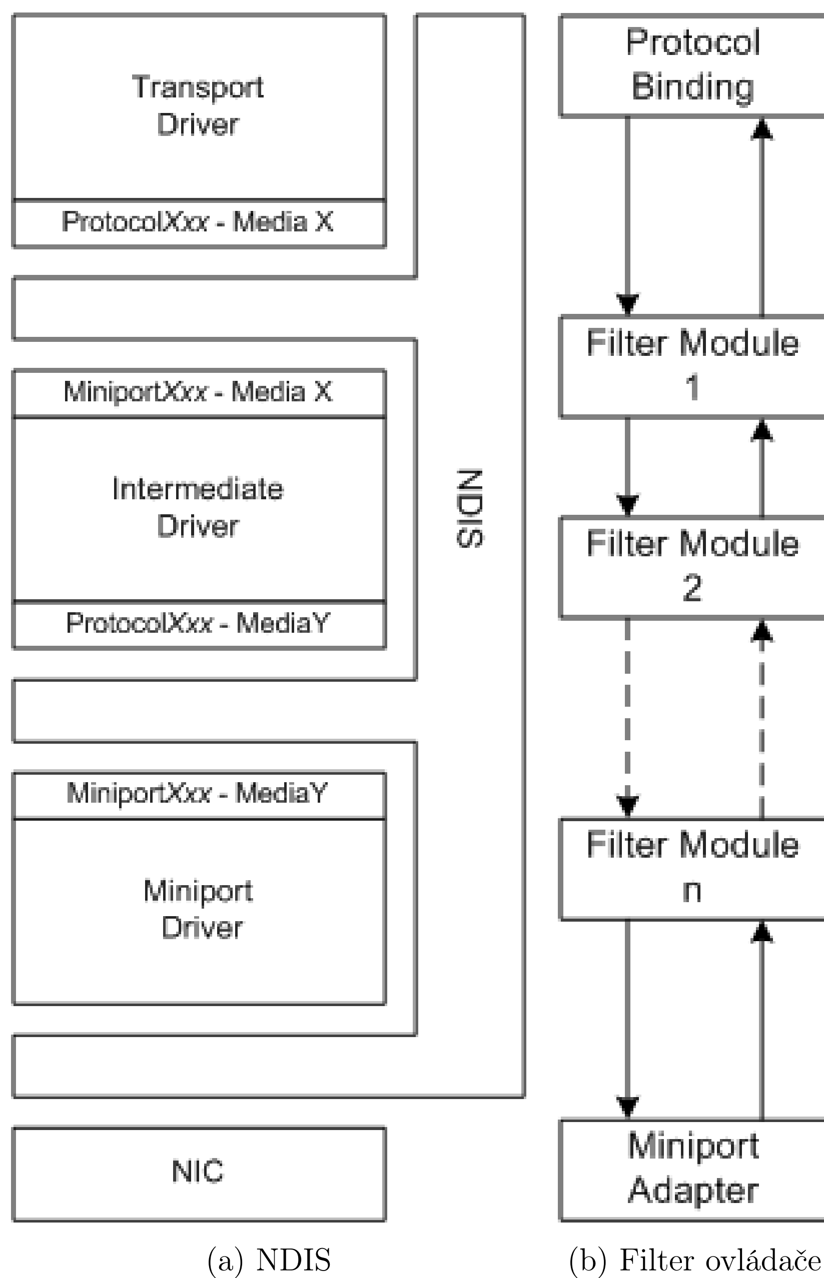
- **Miniport drivers**

  Úlohou Miniport ovládačov je priama správa sieťovej karty (NIC - Network Interface Card).

  Ak má Transport driver paket na odoslanie, zavolá funkciu *Ndis*Xxx** exportovanú NDIS knižnicou, ktorá odovzdá paket miniport ovládaču volaním vhodnej funkcie *MiniportXxx*, ktorú tento ovládač exportuje. Miniport ovládač pošle paket sieťovej karte znova prostredníctvom NDIS volaním príslušnej *Ndis*Xxx** funkcie.

  Keď sieťová karta príjme paket, vyvolá hardvérové prerušenie. Prerušenie je spracované NDIS knižnicou, ktorá volaním *MiniportXxx* funkcie notifikuje príslušný Miniport ovládač a odovzdá mu paket. Miniport ovládač paket po spracovaní odošle ďalšiemu ovládaču v hierarchii volaním *Ndis*Xxx** funkcie.

[1] Ďalšie detaily nebudeme uvádzať ani podrobnejšie vysvetľovať. Týmto textom chceme zdôrazniť, že ovládače tvoria hierarchiu (takzvanú reťaz, ktorú sme spomínali v časti popisujúcej modifikáciu interných štruktúr OS) a vzájomná komunikácia medzi ovládačmi prebieha vždy prostredníctvom NDIS



(a) NDIS (b) Filter ovládače

Figure 3.2: Štruktúry NDIS ovládačov[29]

3.2 Štruktúra paketu

Na obrázku 3.3 sú znázornené dátové štruktúry, ktoré knižnica NDIS používa na reprezentáciu paketov a sieťovej komunikácie. Sieťový paket (rámec) je reprezentovaný štruktúrou NET_BUFFER.

Pakety, ktoré sú súčasťou rovnakého prúdu dát tvoria spájaný zoznam, na ktorého začiatok ukazuje smerník *FirstNetBuffer* v dátovej štruktúre NET_BUFFER_LIST_DATA, ktorá sa nachádza na začiatku štruktúry NET_BUFFER_LIST. Detailná ilustrácia štruktúry NET_BUFFER_LIST je na obrázku 3.4.

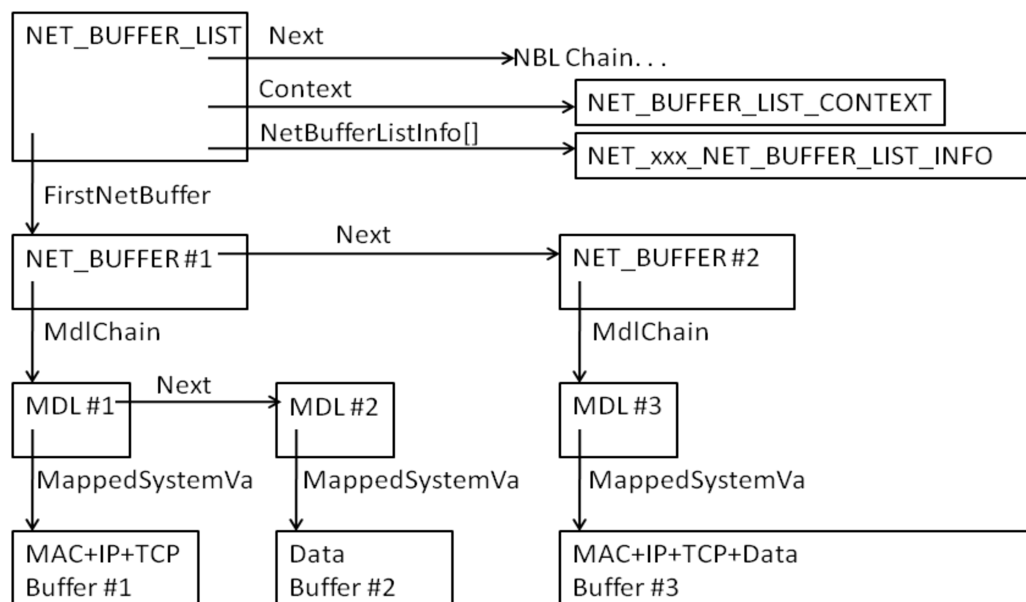


Figure 3.3: Štruktúra paketov v NDIS [19]

Dáta tvoriace paket sa však nenachádzajú súvislo za sebou v pamäti, ale sú reprezentované spájaným zoznamom štruktúr MDL. Memory Descriptor List (MDL) je štruktúra, ktorá reprezentuje jeden súvislý úsek virtuálnej pamäte (buffer) a obsahuje mapovanie na fyzické stránky. Popis reprezentácie dát v štruktúre NET_BUFFER je uvedený na obrázku 3.5.

Veľkosť paketu udáva položka *DataLength*. Smerník *CurrentMdl* ukazuje na prvý MDL obsahujúci dáta a položka *CurrentMdlOffset* obsahuje offset na začiatok dát vrámci buffera. Dáta v bufferi, ktorý je reprezentovaný prvým MDL, totiž nemusia začínať hneď na začiatku. Na tento buffer ukazuje smerník *MappedSystemVa* v štruktúre MDL, viď obrázok 3.3.

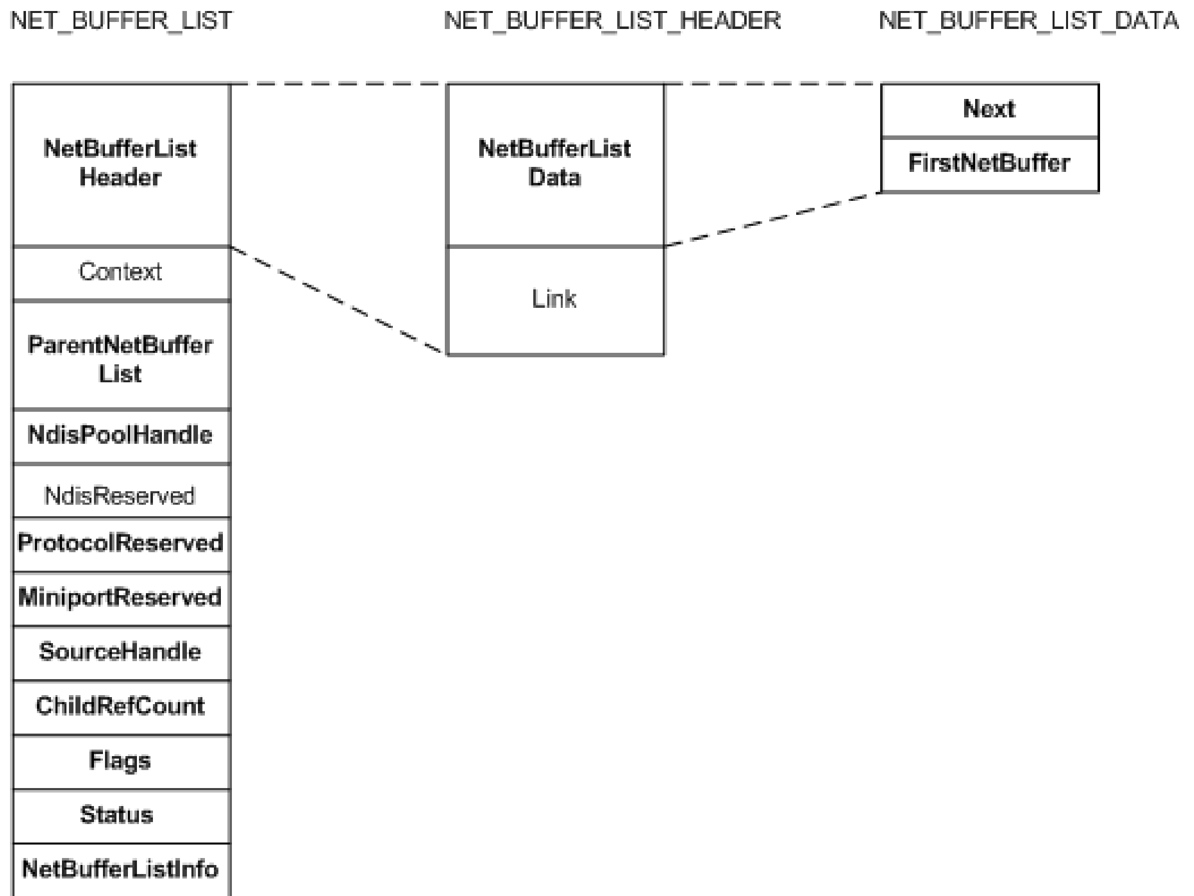


Figure 3.4: Štruktúra NET_BUFFER_LIST [30]

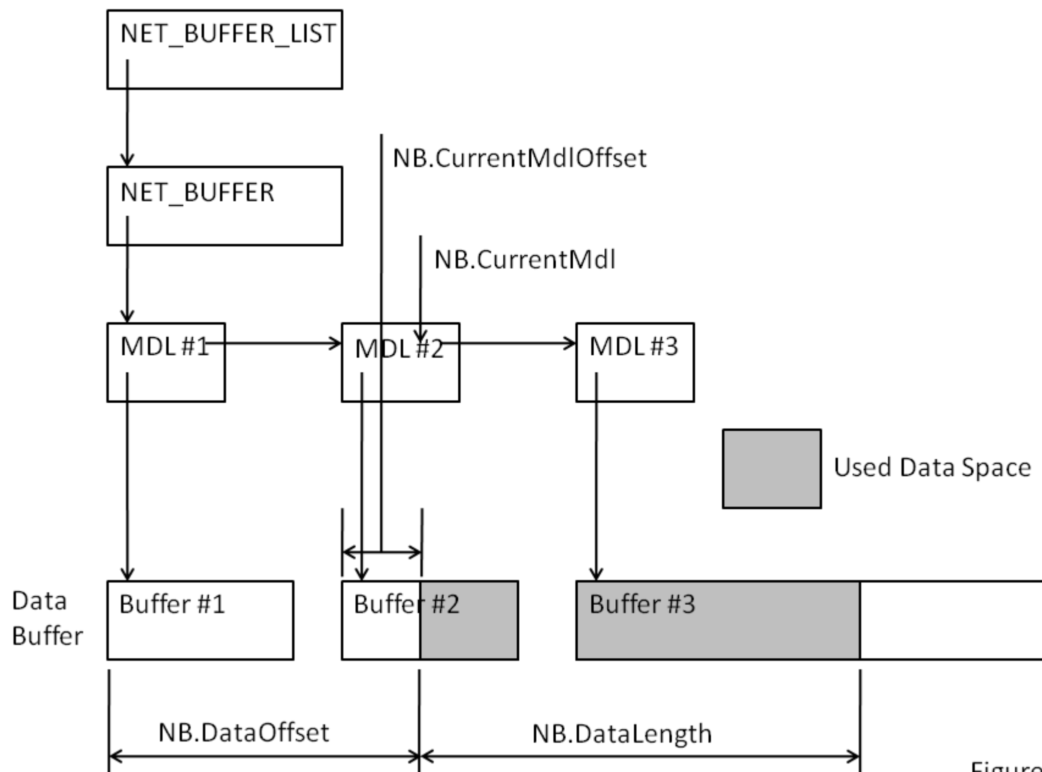


Figure 3.5: Dáta v štruktúre NET_BUFFER [19]

Na obrázku 3.6 je uvedený program funkcie, ktorá ako prvý argument berie smerník na štruktúru NET_BUFFER, reprezentujúcu paket. Funkcia alokuje pamäť pre dáta paketu, iterovaním cez jednotlivé štruktúry MDL prekopíruje tieto dáta tak, aby sa nachádzali súvislo v alokovanej pamäti a vráti smerník na tieto dáta. Druhý argument je výstupný a volajúci ním špecifikuje premennú, kam sa má uložiť veľkosť nakopírovaných dát z paketu.

Následne je možné paket spracovať pretypovaním smerníka a použiť vhodné dátové štruktúry, reprezentujúce jednotlivé hlavičky protokolov (ethernetový rámec, IP,TCP alebo UDP hlavička).

V programe sú pri práci so štruktúrami využívané makrá, popísané v tabuľke 3.1.

Figure 3.6: Získanie dát zo štruktúry NET_BUFFER

```
// Allocates memory for Ethernet Frame and copies there data from all MDLs
static PVOID GetFrameData(PNET_BUFFER pNetBuffer, ULONG * size) {

if (pNetBuffer == nullptr || size == nullptr) return NULL;

ULONG frameSize = NET_BUFFER_DATA_LENGTH(pNetBuffer);
PMDL pMdl = NET_BUFFER_CURRENT_MDL(pNetBuffer);
*(size) = frameSize;
if (frameSize == 0 || !pMdl) return NULL;
ULONG remainingData = frameSize;
ULONG offset = 0;

// Allocate memory for Frame
PVOID data = ExAllocatePoolWithTag(NonPagedPool, frameSize, 'data');
if (data == nullptr) return NULL;

// This is first MDL, handle it specially
PVOID pMdlData = (PVOID)(static_cast<char *>
(MmGetMdlVirtualAddress(pMdl)) + NET_BUFFER_CURRENT_MDL_OFFSET(pNetBuffer));
ULONG mdlDataSize = min(remainingData, MmGetMdlByteCount(pMdl)
 - NET_BUFFER_CURRENT_MDL_OFFSET(pNetBuffer));


// Iterate over MDLs
while (remainingData > 0)
 {
        memcpy((void *)(static_cast<char *>(data) + offset),
                pMdlData, mdlDataSize);
        offset += mdlDataSize;
        remainingData -= mdlDataSize;
        pMdl = pMdl->Next;
        if (!pMdl) break;
        pMdlData = MmGetMdlVirtualAddress(pMdl);
        mdlDataSize = min(remainingData, MmGetMdlByteCount(pMdl));
 }
return data;
}
```

Table 3.1: Použité makrá

NET_BUFER_DATA_LENGTH	vráti veľkosť dát paketu reprezentovaného štruktúrou NET_BUFFER
NET_BUFER_CURRENT_MDL	vráti smerník na prvý MDL
NET_BUFER_CURRENT_MDL_OFFSET	vráti offset na dáta v prvom MDL
MmGetMdlVirtualAddress	pre MDL vráti smerník na buffer, ktorý MDL reprezentuje
MmGetMdlByteCount	pre MDL vráti veľkosť buffera, ktorý MDL reprezentuje

Chapter 4

Detekcia skrytej sieťovej komunikácie

4.1 Malvér Pitou

Ústrednou motiváciou pri tejto práci bola existencia bootkitu s názvom Pitou. Ide o pokročilý, viac-komponentový rootkit, ktorý sa objavil prvýkrát v roku 2014 a jeho úlohou bolo rozosielať spam, prostredníctvom ovládača pracujúceho v jadre OS. Využíval pokročilé techniky skrývania, ktorými zabezpečil, že sieťová komunikácia ktorú generoval bola v systéme „neviditeľná“. Teraz stručne popíšeme spôsob akým fungoval, detaily sú bližšie rozvedené v článku od F-Secure [22], ktoré v našom popise nebudeme popisovať.

Kód malvéru, určený pre beh v Ring 3, je zodpovedný za získanie administrátorských privilégií v systéme a za nainštalovanie bootkitu. Úlohou bootkitu, okrem zabezpečenia spustenia malvéru pri každom štarte počítača, je aj odstavenie KMCS (Kernel Mode Code Signing) Policy - ochrany proti načítaniu ovládača bez podpisu alebo s neplatným podpisom. Ovládač okrem iného hooknutím niektorých IRP handlerov zabezpečil skrývanie modifikácie MBR, registrových kľúčov a súborov.

Ovládač pracujúci v jadre OS najprv čítaním registrových kľúčov zistí, aké sieťové adaptéry sú nainštalované v systéme, a potom na základe predvolenej sieťovej brány určí, prostredníctvom ktorého ovládača bude komunikovať. Následne do pamäte načíta kópiu modulu tcpip.sys (implementácia TCP/IP), situáciu ilustruje obrázok 4.1.

Potom prostredníctvom NDIS získa adresy nízkoúrovňových funkcií, Send a Receive handlerov, ktoré hookne. Malvér používa teda vlastný TCP/IP stack a posiela pakety priamo funkcii ovládača sieťovej karty pre odoslanie paketu. Pri prijatí paketu, malvér odchytí paket a overí na základe portu, či je paket určený pre neho. Ak áno, paket pošle na svoj vlastný stack a systému vytvorí ilúziu, že žiaden paket nebol prijatý, v opačnom prípade pošle paket normálnym spôsobom na spracovanie pôvodným tcpip modulom.

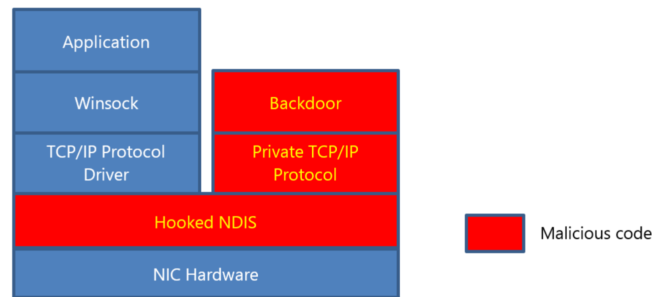


Figure 4.1: Súkromný TCPIP/IP stack [20]

4.2 Detekcia

Spôsob, akým malvér pracoval a ktorý sme popísali v predchádzajúcej časti, nás inšpirovalo vytvoriť ovládač využívajúci virtualizáciu za účelom skrývania hookov, ktorý by bol schopný detegovať skrytú sieťovú komunikáciu v systéme. V našej implementácii využívame projekt DdiMon, ktorý poskytuje API pre hooknutie funkcie neviditeľným spôsobom, využívajúc virtualizáciu, ktorý sme popísali v predchádzajúcej časti.

Schopnosť využívať neviditeľné hooknutia funkcií má niekoľko výhod. Na získanie informácií zo systému je možné využiť hooknutia systémových funkcií, ktoré by za bežných okolností nebolo možné modifikovať, keďže systém sa chráni technológiu KPP a naším cieľom je, aby táto ochrana pracovala stále a zvyšovala tak bezpečnosť systému. Keďže hook je neviditeľný, škodlivá aplikácia, ktorá by sa snažila zistiť prítomnosť ochranného mechanizmu v systéme nie je schopná takúto modifikáciu detegovať. Problém nenastane ani vtedy, ak by sme sa pokúšali inštalovať hook na funkciu, ktorá už bola malvérom hooknutá skôr.

Idea detekčnej techniky, ktorú chceme prezentovať v tejto práci spočíva v nasledovnom. Hooknutím funkcie, ktorú NDIS používa na odosielanie paketov zo systému, na základe argumentov tejto funkcie (detaily popíšeme neskôr) môžeme odchytávať pakety, ktoré boli odoslané sieťovej karte. Zameriame sa na TCP spojenia, pretože systém si udržiava tabuľku všetkých TCP spojení.

Pri odosielaní paketu teda stačí skontrolovať, či existuje záznam o spojení, ktorému patrí daný paket. Ak sa záznam nenájde, môžeme vyhlásiť, že zo systému odchádzajú pakety, o ktorých chýba záznam v štandardných štruktúrach, a preto je tento paket pravdepodobne pokusom o skrytú sieťovú komunikáciu.

Princíp, na akom je založený spôsob našej detekcie, je odlišný od štandardných, ktoré pre detekciu využívajú nejaký vzor, voči ktorému porovnávajú testovanú vzorku. My sa snažíme o detekciu „nadbytočnosti“ vzorky v systéme.

4.2.1 Funkcia SendNetBufferListsHandler

V tejto časti popíšeme spôsob získania adresy funkcie, ktorú NDIS používa pre odosielanie paketov zo systému. Ako sme už spomenuli, zdrojom inšpirácie pre našu implementáciu je malvér Pitou, ktorý získal adresu tejto funkcie a odosielal pakety priamo volaním tejto funkcie, vynechaním štandardných mechanizmov, určených na komunikáciu v systéme Windows. Počas implementácie monitorovacieho sme museli riešiť niekoľko problémov.

Zdrojový kód malvéru nie je verejne známy a samotný ovládač implementujúci túto funkcionalitu bol chránený pred metódami reverzného inžinierstva, aplikovaním techník ako obfuskácia a virtualizácia [1]. V analýze [26] malvéru Srizbi, ktorý je považovaný za predchádzajúcu verziu Pitou, bol popísaný spôsob získania tejto adresy funkcie, avšak pre úplnosť bolo potrebné skombinovať informácie z tejto analýzy, informácie z analýzy reverzným inžinierstvom malvéru Pitou a štruktúr operačného systému Windows.

Výsledkom je nasledovný postup:

1. **Získanie IP adresy predvolenej sieťovej brány** [2]

   Pre získanie IP adresy predvolenej brány používame Windows API funkciu *GetIpForwardTable2()*. Volanie tejto funkcie vráti v druhom argumente smerník na tabuľku obsahujúcu zoznam štruktúr _MIB_IPFORWARD_ROW2. Táto štruktúra obsahuje informácie o smerovacích záznamoch v systéme. Iterovaním cez tieto záznamy získame adresu sieťovej brány. Smerovací záznam pre predvolenú bránu má položku DestinationPrefix v štruktúre _MIB_IPFORWARD_ROW2 nulovú. Keďže v systéme môže existovať viac predvolených brán, vyberieme záznam s najmenšou cenou nasledovne. Celková cena pri použití daného smerovacieho záznamu je súčtom ceny sieťového rozhrania a ceny smerovacieho záznamu.

   Položka InterfaceLuid v štruktúre _MIB_IPFORWARD_ROW2 obsahuje identifikátor sieťového rozhrania, ktorému zodpovedá konkrétny smerovací záznam. Volaním funkcie *GetIpInterfaceEntry* možno získať štruktúru s informáciami o sieťovom rozhraní, ktorá zahŕňa cenu daného rozhrania. Detailne je opisovaný spôsob získania IP adresy znázornený na obrázku 4.2.

[1] Tu sa virtualizácia chápe v inom slova zmysle ako spomíname v práci. Pri virtualizácii spustiteľného súboru sa program prekompiluje a výsledkom je program s rovnakou funkcionalitou ako pôvodný, avšak kód zahŕňa vykonávanie virtuálnych inštrukcií vo virtuálnom softvérovom počítači. Inštrukcie sú počas behu dekódované a výsledok vykonania je ekvivalentný až na to, že pri pokuse o dekompiláciu je program veľmi neprehľadný a efekt samostatných virtuálnych inštrukcií je neznámy.

[2] V našej práci sa venujeme konceptu detekcie skrytej sieťovej komunikácie, preto sa zameriavame len na monitorovanie sieťového rozhrania, prostredníctvom ktorého sa komunikuje s predvolenou bránou. Implementáciu je však možné rozšíriť a aplikovať detekciu na všetky sieťové rozhrania.

Figure 4.2: Získanie IP adresy predvolenej brány

```
static PSTR GetBestInterfaceIP()
{
MIB_IPFORWARD_TABLE2* table = NULL;
MIB_IPFORWARD_ROW2* BestGatewayRow = NULL;
ULONG BestGatewayMetric = ULONG_MAX;
NETIO_STATUS status;
PSTR ipv4Gateway = NULL;

GetIpForwardTable2(AF_INET, &table);

for (ULONG i = 0; i < table->NumEntries; i++) {
     MIB_IPFORWARD_ROW2* row = table->Table + i;
     IP_ADDRESS_PREFIX* prefix = &row->DestinationPrefix;
     SOCKADDR_INET* destAddr = &prefix->Prefix;

      // Take Default route
     if (destAddr->Ipv4.sin_addr.S_un.S_addr == 0 &&
         prefix->PrefixLength == 0) {

              MIB_IPINTERFACE_ROW interfaceRow;
              memset(&interfaceRow, 0, sizeof(MIB_IPINTERFACE_ROW));
              interfaceRow.InterfaceLuid = row->InterfaceLuid;
              interfaceRow.Family = AF_INET;

              // Get Interface IPINTERFACE_ROW structure
              // to obtain Cost of interface
              status = GetIpInterfaceEntry(&interfaceRow);
               if (NETIO_SUCCESS(status)) {
                          // Found better Route
                          if ( (interfaceRow.Metric + row->Metric)
                           < BestGatewayMetric ) {
                                 BestGatewayRow = row;
                      }
              }
     }
}


if (BestGatewayRow) {
          ipv4Gateway = (PSTR)ExAllocatePool(NonPagedPool, 20);
          RtlIpv4AddressToString(&BestGatewayRow->NextHop.Ipv4.sin_addr, ipv4G
          FreeMibTable(table);
          return ipv4Gateway;
    }
return NULL;
}
```

2. **Získanie GUID rozhrania s predvolenou bránou**

   V predchádzajúcom kroku sme získali IP adresu predvolenej brány sieťového rozhrania s najmenšou celkovou cenou.

   Následne enumerujeme podkľúče v databáze Registry uložené v kľúči `\Registry\Machine\SYSTEM\CurrentControlSet\Services\Tcpip\Parameters\Interfaces`. Podkľúče sú pomenované ako GUID[3] zodpovedajúce jednotlivým sieťovým rozhraniam a obsahujú parametre a vlastnosti rozhraní. Na základe rovnosti položky *DhcpDefaultGateway* v podkľúči so získanou IP adresou, určíme GUID rozhrania, ktoré budeme monitorovať.

3. **Získanie prístupu k interným štruktúram NDIS**

   V tomto kroku využívame Windows API funkciu *NdisRegisterProtocolDriver()*. Na obrázku 4.3 je uvedený príklad programu registrovania protokolu s názvom „KIWI“. Protocol driver volá túto funkciu na registráciu svojich *ProtocolXxx* funkcií, ktoré bude NDIS používať.

   Povinným argumentom pre funkciu je smerník na štruktúru NDIS_PROTOCOL_DRIVER_CHARACTERISTICS, ktorá obsahuje informácie o novom registrovanom protokole, napríklad meno, verziu alebo smerníky na spomínané funkcie. V našom prípade jednotlivé *ProtocolXxx* funkcie budú mať prázdne telo, pretože reálne nemáme záujem registrovať nový protokol, iba získať prístup k niektorým interným štruktúram. Druhým vyžadovaným argumentom je smerník na premennú (*protocolHandle*), kam sa uloží handle, ktoré zodpovedá registrovanému protokolu a využíva sa pri volaní ďalších *NdisXxx* funkcií.

   Teraz sa dostávame k časti, kedy prestaneme používať štandardné postupy a odporúčania od Microsoftu a budeme pracovať s oficiálne nedokumentovanými štruktúrami. Spomenuté handle je v skutočnosti smerník na štruktúru _NDIS_PROTOCOL_BLOCK, ktorá obsahuje informácie o registrovanom protokole.

[3] GUID - Globally Unique Identifier - je 128 bitové číslo určené na identifikáciu objektu v operačnom systéme a zapisuje sa ako kombinácia n-tíc hexadecimálnych číslic v kučeravých zátvorkách. N-tice sú oddelené pomlčkami v nasledujúcich intervaloch: 8-4-4-4-12. Príklad {123e4567-e89b-12d3-a456-426655440000}

Figure 4.3: Registrácia protokolu

```
NDIS_PROTOCOL_DRIVER_CHARACTERISTICS characteristics;
NDIS_HANDLE protocolHandle;
// Initialize required fields of structure.
NdisInitUnicodeString(&name, L"KIWI");
characteristics.Header.Type = NDIS_OBJECT_TYPE_DEFAULT;
characteristics.Header.Size = sizeof(NDIS_PROTOCOL_DRIVER_CHARACTERISTICS);
characteristics.Header.Revision = NDIS_PROTOCOL_DRIVER_CHARACTERISTICS_REVIS
characteristics.MajorNdisVersion = 6;
characteristics.MinorNdisVersion = 0;
characteristics.MajorDriverVersion = 1;
characteristics.MinorDriverVersion = 1;
characteristics.Name = name;
// Specify mandatory ProtocolXXX Handlers
characteristics.BindAdapterHandlerEx = PtBindAdapter;
characteristics.CloseAdapterCompleteHandlerEx = PtCloseAdapterComplete;
characteristics.NetPnPEventHandler = PtPNPHandler;
characteristics.OidRequestCompleteHandler = PtRequestComplete;
characteristics.OpenAdapterCompleteHandlerEx = PtOpenAdapterComplete;
characteristics.ReceiveNetBufferListsHandler = PtReceiveNBL;
characteristics.SendNetBufferListsCompleteHandler = PtSendNBLComplete;
characteristics.SetOptionsHandler = PtSetOptions;
characteristics.StatusHandlerEx = PtStatus;
characteristics.UnbindAdapterHandlerEx = PtUnbindAdapter;
// Register protocol and get acces
// to list of protocol and miniport driver structures
NdisRegisterProtocolDriver(NULL, &characteristics, &protocolHandle);
```

4. **Získanie adresy Send Handlera**

   Na obrázku 4.4 je zobrazená časť štruktúry _NDIS_PROTOCOL_BLOCK. Dôležité položky v štruktúre sú zobrazené červenou farbou. Položka *Name* obsahuje meno protokolu. Registrované protokoly v systéme tvoria spájaný zoznam, a prostredníctvom smerníka *NextProtocol* možno prechádzať cez jednotlivé štruktúry.

```
struct _NDIS_PROTOCOL_BLOCK {
        _NDIS_OBJECT_HEADER Header;
        void* ProtocolDriverContext;
        _NDIS_PROTOCOL_BLOCK* NextProtocol;
        _NDIS_OPEN_BLOCK* OpenQueue;
        _REFERENCE ref;
        UCHAR MajorNdisVersion;
        UCHAR MinorNdisVersion;
        UCHAR MajorDriverVersion;
        UCHAR MinorDriverVersion;
        UINT32 Reserved;
        UINT32 Flags;
        UNICODE_STRING Name;
// Shortened, other Parts are not important
};
```

Figure 4.4: Časť štruktúry _NDIS_PROTOCOL_BLOCK

V našej implementácii prechádzame zoznamom, až kým sa nedostaneme ku štruktúre pre protokol s názvom „tcpip“. Následne cez položku *OpenQueue* získame prístup ku štruktúre _NDIS_OPEN_BLOCK.

Táto štruktúra obsahuje informácie o viazaniach medzi protokolmi a Miniport ovládačmi. Položka *BindDeviceName* obsahuje názov Miniport ovládača a položka *NextGlobalOpen* je smerník na ďalšie viazanie protokolu a Miniport ovládača. Tieto štruktúry teda tvoria tiež spájaný zoznam. Následne prechádzame zoznamom viazaní, až kým sa nedostaneme k viazaniu protokolu s miniport ovládačom, ktorého názov obsahuje GUID získané v predošlom kroku.

Po získaní viazania obsahujúceho Miniport ovládač zodpovedajúci sieťovému rozhraniu, ktoré chceme monitorovať použijeme položku *MiniportHandle*, ktorá predstavuje smerník na štruktúru _NDIS_MINIPORT_BLOCK, obsahujúcu informácie o konkrétnom miniport ovládači a cez položku *DriverHandle* ku štruktúre _NDIS_M_DRIVER_BLOCK, súčasťou ktorej je štruktúra popísaná na obrázku 4.5. Položka *SendNetBufferListsHandler* je smerník na funkciu, ktorú hľadáme. Štruktúra obsahuje ešte ďalšie smerníky na funkcie, ktoré NDIS používa.

```
ndis!_NDIS_MINIPORT_DRIVER_CHARACTERISTICS
+0x000 Header                : _NDIS_OBJECT_HEADER
+0x004 MajorNdisVersion      : UChar
+0x005 MinorNdisVersion      : UChar
+0x006 MajorDriverVersion    : UChar
+0x007 MinorDriverVersion    : UChar
+0x008 Flags                 : Uint4B
+0x010 SetOptionsHandler     : Ptr64    int
+0x018 InitializeHandlerEx   : Ptr64    int
+0x020 HaltHandlerEx         : Ptr64    void
+0x028 UnloadHandler         : Ptr64    void
+0x030 PauseHandler          : Ptr64    int
+0x038 RestartHandler        : Ptr64    int
+0x040 OidRequestHandler     : Ptr64    int
+0x048 SendNetBufferListsHandler : Ptr64 void
+0x050 ReturnNetBufferListsHandler : Ptr64    void
+0x058 CancelSendHandler     : Ptr64    void
+0x060 CheckForHangHandlerEx : Ptr64    unsigned char
+0x068 ResetHandlerEx        : Ptr64    int
+0x070 DevicePnPEventNotifyHandler : Ptr64    void
+0x078 ShutdownHandlerEx     : Ptr64    void
+0x080 CancelOidRequestHandler : Ptr64    void
+0x088 DirectOidRequestHandler : Ptr64    int
+0x090 CancelDirectOidRequestHandler : Ptr64    void
+0x098 SynchronousOidRequestHandler : Ptr64    int
```

Figure 4.5: Štruktúra _NDIS_MINIPORT_DRIVER_CHARACTERISTICS

Pre lepšie pochopenie ešte uvádzame ilustráciu procesu prechádzania cez dátové štruktúry, ktorým sme získali adresu hľadanej funkcie na obrázku 4.6.

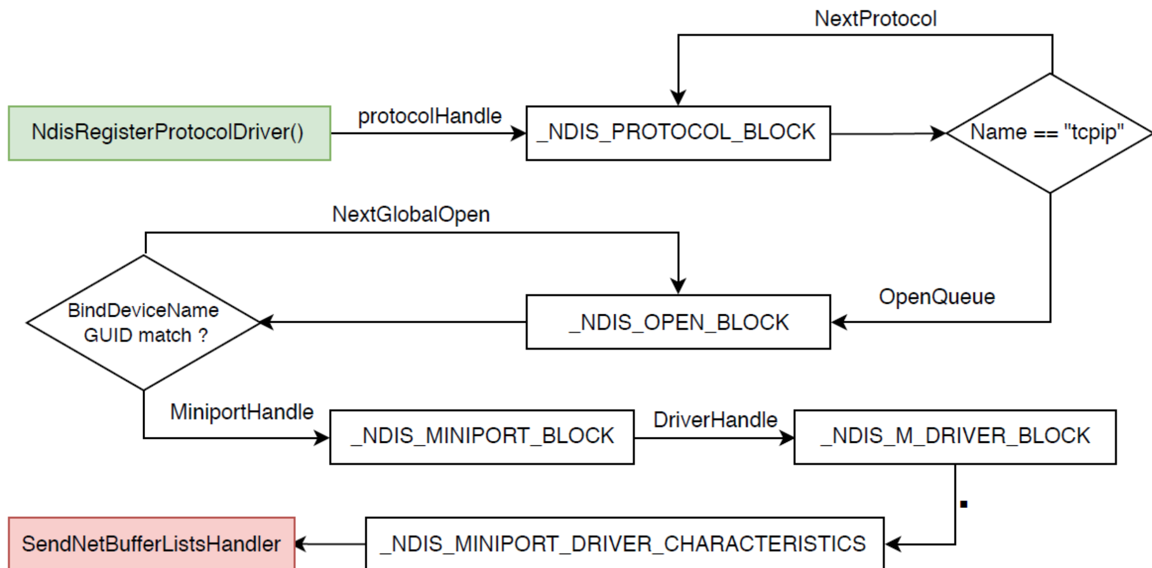


Figure 4.6: Získanie adresy Send Handlera

4.2.2 Odchytávanie odoslaných paketov

Hlavička funkcie send handlera vyzerá nasledovne.

```
VOID SendNetBufferLists(
    NDIS_HANDLE NdisBindingHandle,
    PNET_BUFFER_LIST NetBufferLists,
    NDIS_PORT_NUMBER PortNumber,
    ULONG SendFlags) {
```

Z argumentov je pre nás zaujímavý druhý, smerník na štruktúru _NET_BUFFER_LIST. Spracovanie odosielaného paketu v štruktúre realizujeme spôsobom, ktorý sme popísali v časti 3.2.

Hooknutím tejto funkcie sa pri odosielaní paketu zo systému najprv zavolá naša funkcia, ktorá skontroluje, či paket nie je pokusom o skrytú sieťovú komunikáciu v systéme. V prípade detegovania takejto komunikácie informáciu o zistení tejto skutočnosti vypíšeme pomocou funkcie *DbgPrint()* pre výpis debugovacích hlášok systémových ovládačov. Debugovacie výpisy možno sledovať prostredníctvom programu Dbgview. Podrobnejšie túto časť popíšeme neskôr.

4.2.3 TCP Spojenia

Pri odosielaní sieťového paketu chceme overiť, že systém má záznam o spojení, ktorého súčasťou je odosielaný paket. Na získanie zoznamu TCP spojení existuje Windows API funkcia *GetTcpTable2()*, ktorá vráti tabuľku všetkých spojení. Funkcia je určená pre aplikácie pracujúce User móde. Riešenie sa zdá byť jednoduché, detekčná aplikácia by pozostávala z dvoch samostatných komponentov. Ovládač by odchytával pakety a prostredníctvom aplikácie by overoval, či má systém o danom spojení záznam.

Treba sa však zamyslieť nad nevýhodami tohoto postupu. Volať pri odoslaní každého jedného paketu funkcie programu, ktorý beží v Ring-3 je drahá operácia, pretože prepínanie procesoru medzi jednotlivými úrovňami vyžaduje čas a keď vezmeme do úvahy systém s veľkou sieťovou komunikáciou, dochádzalo by k značnému spomaleniu behu celého systému.

Ďalším problémom odhliadnuc od rýchlosti je, že štandardný model pre komunikáciu ovládača a aplikácie pracujúcej v Ring-3 je princíp žiadosť-odpoveď. Aplikácia žiada ovládač o službu volaním, napríklad už spomenutej funkcie *DeviceIoControl()* a ovládač vráti odpoveď príslušnou Major funkciou. V našom prípade potrebujeme opačný smer, volanie obslužných funkcií v aplikácii bežiacej v Ring-3. Existuje však podpora aj pre takýto neštandardný model komunikácie a nazýva sa Inverted Call Model. Pri pokuse o použitie tohto modelu, hypervisor nefungoval a dochádzalo k jeho uvoľneniu (unload) z pamäte. Vzhľadom na tieto problémy sme sa rozhodli, že detekčná aplikácia bude pozostávať iba zo samotného systémového ovládača.

Windows však neposkytuje podobnú API funkciu na získanie informácií o spojeniach pre ovládače, čo znamená, že tieto informácie sme museli získať iným, neštandardným spôsobom.

Na obrázku 4.8 je znázornená postupnosť volaní funkcií programom netstat.exe, ktorý vzhľadom na argumenty vypíše informácie o sieťovej komunikácii v systéme, napríklad aj tabuľku TCP spojení. Program volá nedokumentovanú funkciu *InternalGetTcpTable2()*, ktorá však interne používa už spomenutú funkciu *GetTcpTable2()*. Riadenie sa postupne dostane až do funkcie *TcpEnumerateAllConnections()* v module tcpip.sys a práve toto je funkcia, ktorá by nás mohla zaujímať. Funkcia je samozrejme nedokumentovaná a nie je ani exportovaná modulom tcpip.sys. Analyzovaním sme zistili, že táto funkcia je len obal pre volanie funkcie *TcpEnumerateConnectionType()*, ktorá následne ešte volá funkciu *TcpEnumerateConnections()*. Postupnosť volaní je ilustrovaná na obrázku 4.7.

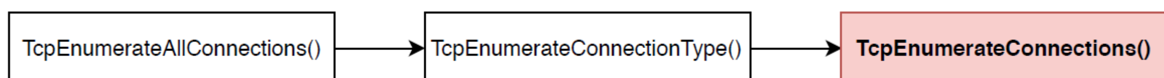


Figure 4.7: Postupnosť volaní v module tcpip.sys

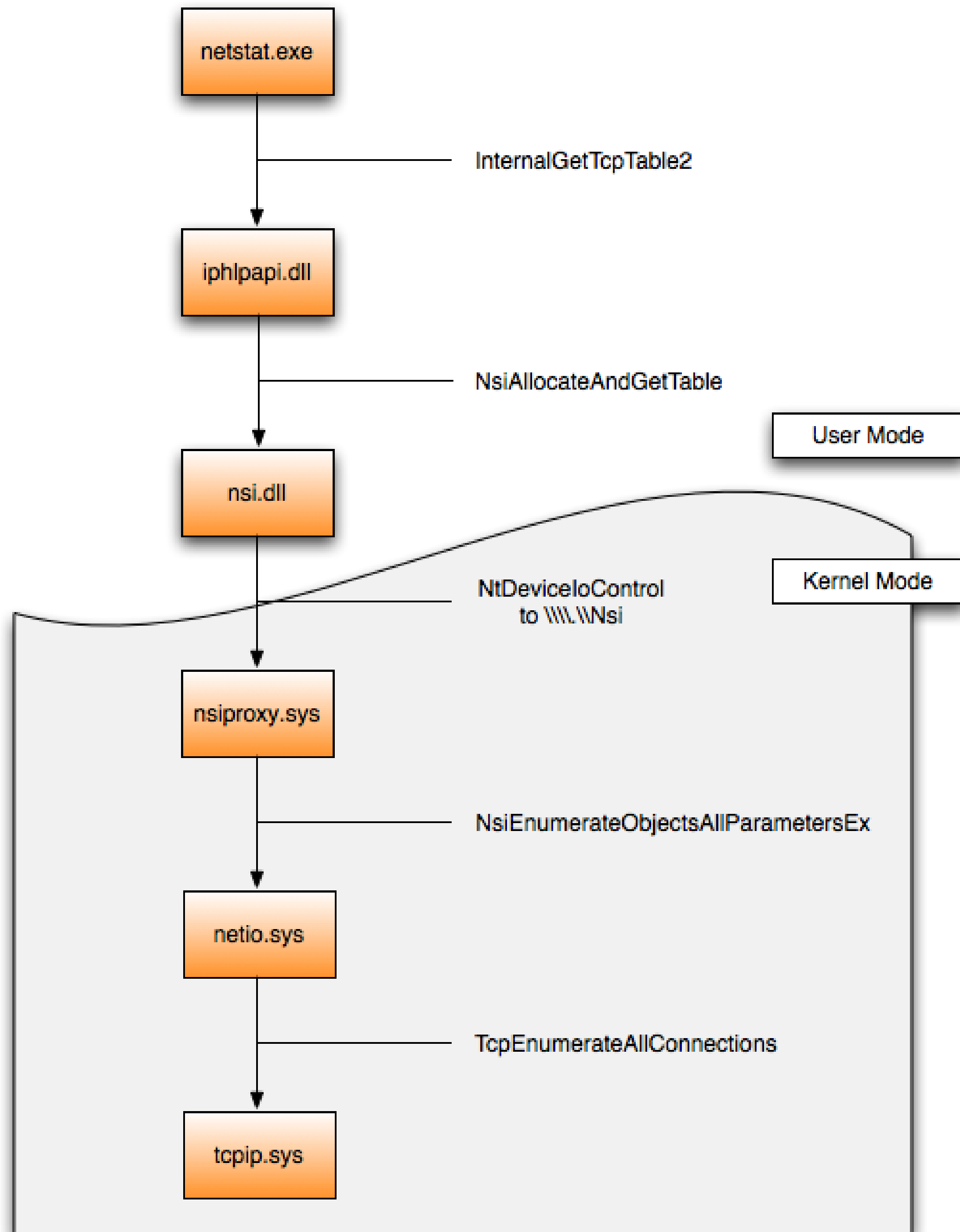


Figure 4.8: Funkcie volané pri enumerovaní spojení [27]

Funkcia *TcpEnumerateConnections()* používa dve globálne premenné pri získavaní informácií o sieťových spojeniach v systéme. Prvá premenná je smerník na dátovú štruktúru *PartitionTable* a druhá s názvom *PartitionCount* uchováva počet štruktúr *PartitionEntry* v štruktúre *PartitionTable*. Ilustrácia spomenutých štruktúr je na obrázku 4.9, ktoré sú bližšie popísané v článku [31].

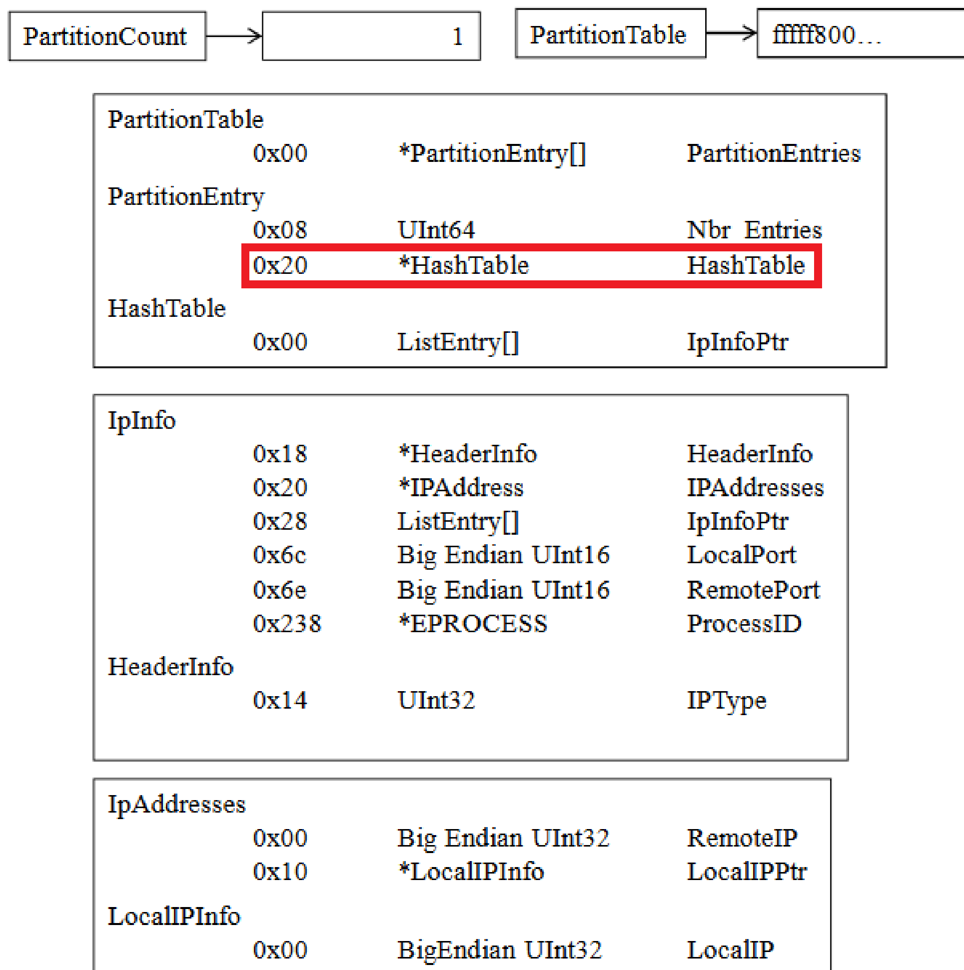


Figure 4.9: Štruktúra PartitionTable [31]

Štruktúra *PartitionEntry* obsahuje smerník na hashovaciu tabuľku, v ktorých systém Windows uchováva informácie o spojeniach. Predpoklad, ktorý využívame v našej implementácii je, že pozícia globálných premenných v rámci modulu tcpip.sys je konštantná. Pre rôzne verzie operačných systémov a architektúr sa však offsety líšia. Prístup k spomenutým globálnym premenným dostaneme nasledovne. Najprv získame adresu modulu tcpip.sys v pamäti, a indexovaním do pamäte tohto modulu získame danú premennú.

Analýzou funkcie *TcpEnumerateConnections()* sme vytvorili funkciu znázornenú na obrázku 4.10, ktorá obdobným spôsobom prechádza cez tieto štruktúry a overí, či pre dané argumenty (cieľová IP adresa a port) v systéme existuje záznam o spojení s takýmito parametrami. Položky znázornené červenou farbou sú offsety, ktoré sme získali analýzou spomenutých štruktúr.

Figure 4.10: Overenie existencie záznamu v tabuľke TCP spojení pre danú cieľovú IP adresu a port

```
bool CheckTcpipConnection(UCHAR* tcpipBase,IN_ADDR* dstIP,USHORT port) {

  RTL_DYNAMIC_HASH_TABLE_ENUMERATOR enumerator;
  KLOCK_QUEUE_HANDLE LockHandle;
  bool found = false;
  // Get addresses of global variables in tcpip.sys based on fixed offsets
  UCHAR ** partitionTablePtr = (UCHAR **)(tcpipBase + PARTITION_TABLE_PTR_OFFSET);
  ULONG * partitionCount = (ULONG*)(tcpipBase + PARTITION_COUNT_OFFSET);
  TCPIP_PARTITION_TABLE* partitionTable;

  // Iterate through partitions
  for (int i = 0; i < *partitionCount; i++) {
        if (found) break;
        partitionTable = (TCPIP_PARTITION_TABLE*)
        (((*partitionTablePtr) + NEXT_PARTITION_OFFSET*i));
        KeAcquireInStackQueuedSpinLock(partitionTable->SpinLock, &LockHandle);
        RtlInitWeakEnumerationHashTable(partitionTable->hashTable, &enumerator);

        // Enumerate entries in hash table belonging to a partition
        while (1) {
                UCHAR* hEntry = (UCHAR*)RtlWeaklyEnumerateEntryHashTable(
                partitionTable->hashTable, &enumerator);
                if (!hEntry) break;
                TABLE_ENTRY* tableEntry =
                (TABLE_ENTRY*)(hEntry - HASH_TABLE_ENTRY_OFFSET);
                IN_ADDR ip = *tableEntry->ipAddrStruct->ipAddr;

                // Check entry match IP and port
                if (ip.S_un.S_addr == dstIP->S_un.S_addr &&
                        tableEntry->remotePort == port) {
                        found = true;
                        break;
                    }
            }
        // End Enumeration and free lock
        RtlEndWeakEnumerationHashTable(partitionTable->hashTable, &enumerator);
        KeReleaseInStackQueuedSpinLock(&LockHandle);
  }
  return found;
}
```

4.3 Implementácia a testovanie

Odchytávanie odosielaných paketov funguje na operačných systémoch Windows 7 a 10. Podporované sú architektúry x86 aj x64. Vzhľadom na časové podmienky, detekcia skrytej sieťovej komunikácie je implementovaná iba pre OS Windows 7. Rozdiel medzi Windows 10 je v spomenutých offsetoch, ktoré by nemal byť problém zistiť a potrebnú funkcionalitu doimplementovať. V práci išlo najmä o poukázanie na nový spôsob a pohľad na detekciu.

Pri testovaní sme narazili na problémy, keď bola detegovaná aj komunikácia, ktorá nebola pokusom o skrytú sieťovú komunikáciu. Problém nastával pri ukončovaní alebo zrušení TCP spojenia lokálnou stranou. Analýzou správania systému sme prišli k nasledovným záverom. Pri zrušení spojenia, systém odstráni informáciu o spojení so spomínaných dátových štruktúr a až potom odošle TCP paket s flagmi RST ACK. Tým pádom sme detegovali tento paket ako nadbytočný, keďže o ňom neexistoval v systéme záznam. Tento problém sme vyriešili ignorovaním paketov s nastavenými flagmi RST a ACK.

Pri ukončení spojenia, v čase odosielania posledného TCP paketu s flagom ACK už záznam o spojení v systéme tiež neexistuje a znova nastáva prípad popísaný vyššie. Ukončovanie TCP spojenia je znázornené na obrázku 4.11.

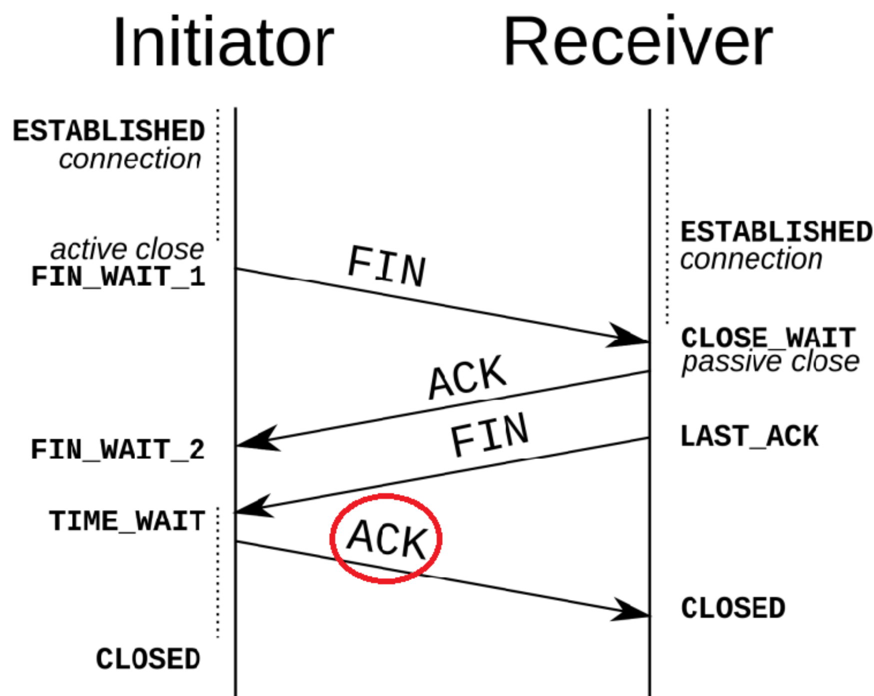


Figure 4.11: Ukončovanie TCP spojenia [17]

Tento problém sme vyriešili tak, že informácie o spojeniach (IP adresa a Port) si ukladáme do hashovacej tabuľky. Ak sa odosiela paket, o ktorom existuje v systéme záznam, uložíme informáciu do tabuľky. Ak v systéme odosielaný paket nemá záznam, pozrieme sa do našej tabuľky, či spojenie existovalo. Ak áno, namiesto chybného hlásenia iba odstránime záznam o spojení z tabuľky, v opačnom prípade ide o nevyžiadanú komunikáciu. Program, ktorý implementuje popísanú funkcionalitu je uvedený na obrázku 4.12.

```
if (ipPacket->protocol == TCP_PROTOCOL) {
    PTCP tcpHeader = (PTCP) ipPacket->data;
    struct ip_port Connection;
    Connection.ip = ipPacket->dst_ip.S_un.S_addr;
    Connection.port = tcpHeader->destination_port;
    // Check system has record about this connection
    if(!CheckTcpipConnection(tcpipBase,&ipPacket->dst_ip,tcpHeader->destination_port)){
        // Check if hypervisor has seen this connection
        if (activeIPs.find(Connection) == activeIPs.end()) {
            UCHAR flags = tcpHeader->flags;
            // If not TCP termination (RST ACK), Log Detected packet
            if (flags != 0x14) {
                    HYPERPLATFORM_LOG_ERROR
                    ("Detected: DST IP: %d.%d.%d.%d:%d SRC IP: %d.%d.%d.%d:%d ",
                            ipPacket->dst_ip.S_un.S_un_b.s_b1,
                            ipPacket->dst_ip.S_un.S_un_b.s_b2,
                            ipPacket->dst_ip.S_un.S_un_b.s_b3,
                            ipPacket->dst_ip.S_un.S_un_b.s_b4,
                            RtlUshortByteSwap(tcpHeader->destination_port),
                            ipPacket->src_ip.S_un.S_un_b.s_b1,
                            ipPacket->src_ip.S_un.S_un_b.s_b2,
                            ipPacket->src_ip.S_un.S_un_b.s_b3,
                            ipPacket->src_ip.S_un.S_un_b.s_b4,
                            RtlUshortByteSwap(tcpHeader->source_port));
            }
        }
        else {
            // Connection is terminating, remove from hash table
            activeIPs.erase(Connection);
        }
    }
    else {
        // Save this connection to hash table
        if (activeIPs.find(Connection)==activeIPs.end()) {
            activeIPs.insert(Connection);
        }
    }
}
```

Figure 4.12: Program kontrolujúci odosielané pakety

4.4 Výsledky

Aplikáciu, ktorú sme vytvorili za účelom detekcie skrytej sieťovej komunikácie sme testovali pod OS Windows 7 x64 vo virtuálnom prostredí použitím softvéru VMWare s bežiacim malvérom Pitou. Softvér VMWare obsahuje podporu pre vnorenú virtualizáciu (Nested Virtualization), a teda umožňuje beh hypervisora pod iným hypervisorom, vďaka čomu sme mohli pri vývoji a ladení hypervisora pracovať vo virtuálnom stroji a prípadné chyby v implementácii spôsobili iba pád virtuálneho stroja. Viac informácií je v článku od VMWare [16]. Pre inštalovanie a spúšťanie ovládačov sme použili aplikáciu „OSR Driver Loader“[6].

Na obrázku 4.13 je zobrazených posledných päť odosielaných rámcov zo systému odchytených nástrojom Wireshark, bežiacim vo virtuálnom stroji. Podobne sú na obrázku 4.14 znázornené odchytené rámce, avšak tentoraz bežal monitorovací nástroj mimo virtuálneho prostredia, na hostiteľskom počítači. Na tomto obrázku je možné vidieť komunikáciu smerujúcu na IP adresu 195.154.237.14. Keďže sieťová komunikácia prebiehala na nízkej úrovni, bežné mechanizmy na monitorovanie komunikácie nezaznamenali žiadnu komunikáciu, viď. ilustrácia 4.13. Na obrázku 4.15 je zobrazený výpis informačných hlášok v programe DbgView od nášho detekčného modulu, ktorému sa úspešne podarilo identifikovať skrytú sieťovú komunikáciu.

621 60.936389	192.168.34.144	23.42.27.27	GET /MFEwTzBNMEswSTAJBgUrDgMCGgUABBTA%2FgJ4
622 60.936683	23.42.27.27	192.168.34.144	80 → 49176 [ACK] Seq=1 Ack=325 Win=64240 Le
623 61.082292	23.42.27.27	192.168.34.144	Response
624 61.176683	23.42.27.27	192.168.34.144	[TCP Retransmission] 80 → 49176 [PSH, ACK]
625 61.176742	192.168.34.144	23.42.27.27	49176 → 80 [ACK] Seq=325 Ack=203 Win=64038

Figure 4.13: Odchytené pakety v Guest OS [17]

604 58.593248	192.168.34.144	23.42.27.27	GET /MFEwTzBNMEswSTAJBgUrDgMCGgUABBTA%2FgJ4%2FJkY
605 58.593437	23.42.27.27	192.168.34.144	80→49176 [ACK] Seq=1051343332 Ack=1577757171 Win=
606 58.739344	23.42.27.27	192.168.34.144	Response
607 58.834043	23.42.27.27	192.168.34.144	[TCP Retransmission] 80→49176 [PSH, ACK] Seq=1051
608 58.834302	192.168.34.144	23.42.27.27	49176→80 [ACK] Seq=1577757171 Ack=1051343534 Win=
638 62.506471	192.168.34.144	195.154.237.14	10001→7384 [SYN] Seq=6511 Win=8192 Len=0 MSS=1460
641 62.536878	195.154.237.14	192.168.34.144	7384→10001 [SYN, ACK] Seq=2142974421 Ack=6512 Win
642 62.540988	192.168.34.144	195.154.237.14	10001→7384 [ACK] Seq=6512 Ack=2142974422 Win=8192
643 62.572503	192.168.34.144	195.154.237.14	10001→7384 [PSH, ACK] Seq=6512 Ack=2142974422 Win
644 62.572636	195.154.237.14	192.168.34.144	7384→10001 [ACK] Seq=2142974422 Ack=6640 Win=6424
645 62.605858	195.154.237.14	192.168.34.144	7384→10001 [FIN, PSH, ACK] Seq=2142974422 Ack=664
646 62.608716	192.168.34.144	195.154.237.14	10001→7384 [ACK] Seq=6640 Ack=2142974423 Win=8191

Figure 4.14: Odchytené pakety v Host OS

```
Detected: DST IP Address: 195.154.237.14:7384 SRC IP Address: 192.168.34.144:10001
SeqN:6511 AckN:0
SYN
Detected: DST IP Address: 195.154.237.14:7384 SRC IP Address: 192.168.34.144:10001
SeqN:6512 AckN:2142974422
ACK
Detected: DST IP Address: 195.154.237.14:7384 SRC IP Address: 192.168.34.144:10001
SeqN:6512 AckN:2142974422
PSH
ACK
Detected: DST IP Address: 195.154.237.14:7384 SRC IP Address: 192.168.34.144:10001
SeqN:6640 AckN:2142974423
ACK
```

Figure 4.15: Výpis hypervisora zobrazený v programe DebugView

Chapter 5

Záver

V práci sme popísali princípy a techniky, ktoré využíva škodlivý kód na ovládnutie systému a zabezpečenie svojej ochrany a neviditeľnosti pred obrannými mechanizmami OS. Ďalej sme predstavili a popísali techniky určené na virtualizáciu, ich výhody, nevýhody a použitie na skrývanie pamäte pred OS, ktoré nám zabezpečili lepšie možnosti monitorovania udalostí v systéme.

Analyzovali sme škodlivý kód, využívajúci modifikácie v jadre OS za účelom skrytej sieťovej komunikácie, skúmali sme vnútorné štruktúry operačného systému Microsoft Windows a využitím existujúcej implementácie DdiMon používajúcej virtualizáciu na skrytú modifikáciu funkcií sme vytvorili nástroj umožňujúci detekciu takejto komunikácie, ktorý sme úspešne otestovali.

Do budúcnosti vidíme možné rozšírenie práce v získavaní informácií o existujúcich spojeniach v systéme, použitím nedokumentovaných funkcií, ktoré ležia na vyššej úrovni ako interné štruktúry OS, ktoré používame. Vzdialenosť medzi úrovňou, kde odchytávame pakety a úrovňou, kde zisťujeme informácie o spojeniach, ktoré má systém, by sa tak zväčšila a viac škodlivého softvéru pracujúceho medzi týmito úrovňami by mohlo byť úspešne odhaleného.

Bibliography

[1] A Brief Survey on Rootkit Techniques in Malicious Codes. Získané 10.04.2018 z https://pdfs.semanticscholar.org/9a1a/913dbc2f8d3fbf5175e5fae7a8e4a304cb24.pdf.

[2] Code Injection and API Hooking Techniques. Získané 10.04.2018 z http://nagareshwar.securityxploded.com/2014/03/20/code-injection-and-api-hooking-techniques/.

[3] Direct Kernel Object Manipulation. Získané 13.04.2018 z https://www.blackhat.com/presentations/win-usa-04/bh-win-04-butler.pdf.

[4] Dll injection. Získané 29.03.2018 z https://www.endgame.com/blog/technical-blog/ten-process-injection-techniques-technical-survey-common-and-

[5] Dll injection. Získané 05.04.2018 z http://blog.opensecurityresearch.com/2013/01/windows-dll-injection-basics.html.

[6] Driver Loader . https://www.osronline.com/article.cfm?article=157.

[7] Hook types. Získané 05.04.2018 z http://blog.opensecurityresearch.com/2013/01/windows-dll-injection-basics.html.

[8] intel vt x based full virtualization. Získané 03.01.2017 z http://www.cubrid.org/files/attach/images/220547/462/683/intel_vt_x_based_full_virtualization.png.

[9] KernelMode Rootkits: Part 1, SSDT hooks. Získané 10.04.2018 z https://www.adlice.com/kernelmode-rootkits-part-1-ssdt-hooks/.

[10] KernelMode Rootkits: Part 2, IRP hooks. Získané 13.04.2018 z https://www.adlice.com/kernelmode-rootkits-part-2-irp-hooks/.

[11] Layer 7 (application layer) – What is the information security key factors? Získané 10.04.2018 z http://www.antihackingonline.com/application-development/layer-7-application-layer-what-is-the-information-security-key-factors/.

[12] Operating System Inside - General. Získané 17.03.2017 z http://osinside.net/osinside/osinside.htm.

[13] Plugin Post: Robust Process Scanner . Získané 13.04.2018 z https://moyix.blogspot.sk/2010/07/plugin-post-robust-process-scanner.html.

[14] Protection rings. Získané 29.03.2018 z https://en.wikipedia.org/wiki/Protection_ring#/media/File:Priv_rings.svg.

[15] Ring3 / Ring0 Rootkit Hook Detection 1/2. Získané 11.04.2018 z https://www.malwaretech.com/2013/09/ring3-ring0-rootkit-hook-detection-12.html.

[16] Running Nested VMs. https://communities.vmware.com/docs/DOC-8970.

[17] Volatility's New Netscan Module. Získané 26.4.2018 z https://www.researchgate.net/figure/TCP-Connection-Termination_fig1_320625383.

[18] What are device drivers? Získané 13.04.2018 z https://www.quora.com/What-are-device-drivers.

[19] Codemachine. NDIS 6 Net Buffer Lists and Net Buffers. Získané 12.02.2018 z http://www.codemachine.com/article_ndis6nbls.html.

[20] DataProtectionCenter. The "hidden" backdoor – VirTool:WinNT/Exforel.A. Získané 23.4.2018 z http://dataprotectioncenter.com/general/the-hidden-backdoor-virtoolwinntexforel-a/.

[21] F-Secure. Bootkits are not dead. Pitou is back! Získané 13.04.2018 z https://www.tgsoft.it/english/news_archivio_eng.asp?id=884.

[22] F-Secure. The "silent" resurrection of the notorious Srizbi kernel spambot. https://www.f-secure.com/documents/996508/1030745/pitou_whitepaper.pdf.

[23] Johan De Gelas. Hardware Virtualization: the Nuts and Bolts. Získané 17.03.2017 z https://www.anandtech.com/show/2480/10.

[24] Intel. Intel® 64 and IA-32 Architectures Software Developer's Manual.

[25] Intel. The Advantages of Using Virtualization Technology in the Enterprise. Získané 03.01.2017 z https://software.intel.com/sites/default/files/m/d/4/1/d/8/aspose.words.demos.001.png.

[26] Kimmo Kasslin. Spam from the kernel. Získané 14.04.2018 z https://www.virusbulletin.com/virusbulletin/2007/11/spam-kernel.

[27] Michael Hale Ligh. Volatility's New Netscan Module. Získané 26.4.2018 z https://mnin.blogspot.sk/2011/03/volatilitys-new-netscan-module.html.

[28] Microsoft. Microsoft Developer Network. https://msdn.microsoft.com/en-us/.

[29] Microsoft. NDIS driver types. https://docs.microsoft.com/en-us/windows-hardware/drivers/network/ndis-drivers.

[30] Microsoft. Net Buffer List Structure. Získané 12.11.2016 z https://docs.microsoft.com/en-us/windows-hardware/drivers/network/net-buffer-list-structure.

[31] James S. Okolica* and Gilbert L. Peterson. Windows Driver Memory Analysis: A Reverse Engineering Methodology.

[32] DIPAK K. SINGH. Era of Hardware Assisted Native Hypervisors. Získané 19.04.2018 z http://dipak123.info/htmlslideshow/native_vm.html#/.

[33] Satoshi Tanda. DdiMon. https://github.com/tandasat/DdiMon.

[34] Vmware. Understanding Full Virtualization, Paravirtualization, and Hardware Assist. Získané 10.02.2017 z https://www.vmware.com/techpapers/2007/understanding-full-virtualization-paravirtualizat-1008.html.