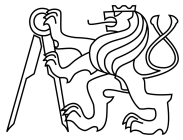
ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ



ZADÁNÍ DIPLOMOVÉ PRÁCE

Název:	Skrývání virtuálního prostředí před malwarem
Student:	Bc. Tomáš Hradský
Vedoucí:	Ing. Petr Kurtin
Studijní program:	Informatika
Studijní obor:	Počítačová bezpečnost
Katedra:	Katedra počítačových systémů
Platnost zadání:	Do konce zimního semestru 2018/19

Pokyny pro vypracování

Seznamte se s metodami virtualizace operačních systémů a jejich využití při detekci malwaru.

Popište způsoby, kterými malware detekuje virtuální prostředí a pro několik vybraných prostředí navrhněte protiopatření, která implementujte do vybraného volně dostupného virtualizačního nástroje (ideálně VirtualBox).

Zaměřte se pouze na tyto způsoby využívané malwarem, ne na detekci samotného malwaru.

Seznam odborné literatury

Dodá vedoucí práce.

prof. Ing. Róbert Lórencz, CSc.
vedoucí katedry

prof. Ing. Pavel Tvrdík, CSc.
děkan

V Praze dne 18. února 2017

České vysoké učení technické v Praze
Fakulta informačních technologií
Katedra počítačových systémů

Diplomová práce

Skrývání virtuálního prostředí před malwarem

Bc. Tomáš Hradský

Vedoucí práce: Ing. Petr Kurtin

8. května 2017

Poděkování

Děkuji svému vedoucímu Ing. Petru Kurtinovi za pomoc při řešení problémů a za mnohé cenné rady, bez nichž by tato práce nedosáhla patřičné kvality. Velké díky patří i Luboši Hnaníčkovi za jeho ochotu a obdivuhodnou schopnost předávat znalosti.

Prohlášení

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

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

V Praze dne 8. května 2017 .....

Odkaz na tuto práci

Hradský, Tomáš. *Skrývání virtuálního prostředí před malwarem.* Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2017.

Abstrakt

Cílem práce je představit počítačovou virtualizaci a její využití při analýze malwaru. Práce se zaměřuje na metody, kterými malware detekuje virtuální prostředí, na základě čehož modifikuje své chování a tím ztěžuje svou analýzu.

Pro uvedené metody jsou představena protiopatření, z nichž některá jsou implementována do volně dostupného virtualizačního nástroje *VirtualBox*.

Přínosem této práce je detailní popis počítačové virtualizace a její projevy uvnitř virtuálních počítačů. Práce zároveň představuje jednak teoretický základ pro zvětšení účinnosti analyzačních metod využívaných antivirovými firmami a poté i přímo využitelné praktické řešení.

Klíčová slova virtuální prostřední, malware, *VirtualBox*, VMX

Abstract

This thesis describes computer virtualization and its purpose in malware analysis. Most of the thesis is dedicated to methods implemented in malware by which it's able to detect virtual environment in order to modify its behavior and make its analysis more difficult.

Countermeasures are introduced and some of them are implemented into freely accessible virtualization tool *VirtualBox*.

Main contribution of this thesis is its detailed description of computer virtualization and its giveaways in virtualized computers. It also provides a solid theoretical basis and a practical example which antimalware industry might benefit from.

Keywords virtual environment, malware, *VirtualBox*, VMX

Obsah

Seznam obrázků

Úvod

Malware, neboli škodlivý software[1], se za několik málo desítek let přesunul z oblasti neškodných vtípků, které měly za účel především ukázat autorovu dovednost, do oblasti mnoha milionového obchodu[1]. A to jak pro tvůrce malwaru, tak pro společnosti, které se starají o ochranu firem a jednotlivců před jeho působením.

Jen pro představu, jedním z prvních virů, který infikoval osobní počítače byl Ping Pong virus[2]. Jeho cílem byl operační systém MS-DOS a projevoval se jako tečka odrážející se od hranic monitoru. Smyslem bylo nejspíše jen znepříjemnění uživatelova života.

Ping Pong virus se objevil v roce 1988, přesuneme-li se o necelých 30 let do současnosti, uvidíme, že nejčastějším malwarem dneška je takzvaný ransomware. Nejedná se o konkrétní program ale o celou skupinu programů. Ransomware je označení pro software, jehož cílem je znepřístupnění (nejčastěji zašifrování) vybraných souborů a požadovaní výkupného za jejich dešifrování.

Pro uvedenou skupinu malwaru je zajímavé, že autoři neprovádějí škodlivou činnost sami, nýbrž nabízejí své produkty společně s uživatelskou podporou. Tento business model se zažil pod názvem MaaS - *Malware as a Service*.

Je evidentní, že účelem moderního malwaru je finanční profit. Proto je také kladen velký důraz na ochranu jak osobních počítačů, tak velkých serverů a datových center. Jelikož 15 % veškerého nově vznikajícího softwaru je škodlivá[3, str. 17], musejí se antivirové firmy spoléhat stále více na automatické metody detekce.

Zde vstupuje do hry virtualizace. Na příkladu ransomwaru vidíme, že spuštění neznámého programu může mít devastační následky na uložené soubory, v horším případě na celý počítač. Při analýze je proto vhodné neznámé programy nejdříve spustit v kontrolovaném prostředí, jehož případné narušení lze snadno opravit. Pro tento účel se jeví ideální virtuální počítače, v nichž lze

[1] Z anglického Malicious software

neznámý vzorek bez obav pustit a na základě jeho chování rozhodnout, zda se jedná o malware.

Avšak tvůrci malwaru si jsou virtualizovaného prostředí vědomi, a proto do svých programů zahrnují mechanismus, který kontroluje, zda program běží ve virtuálním počítači a pokud ano, zdrží se jakékoli škodlivé činnosti, čímž se úspěšně vyhne chycení.

Ovšem pro autory virtualizačních nástrojů, jako je například *VMware* nebo *VirtualBox* není cílem zatajovat, že se jedná o virtuální počítač a tak nehledí zda svým snažením zanechávají indície, které může malware nalézt a rozpoznat tak virtuální prostředí.

Odstranění alespoň některých znaků virtuálního prostředí se věnuje tato práce. Soustředíme se pouze na virtualizaci procesorové architektury x86 nebo s ní kompatibilní - IA-32 a AMD64, jelikož se jedná o nejpoužívanější architekturu v osobních počítačích.

Práce je rozdělena na teoretickou a praktickou část. V první kapitole je popsána počítačová virtualizace, důvody vzniku a její výhody. Dále je popsáno několik dostupných virtualizačních nástrojů a jejich odlišnosti. V druhé kapitole je detailně rozepsána jedna z metod, jak vytvořit virtuální prostředí. V praktické části jsou popsány důvody, proč se skrýváním virtuálního prostředí zabývat. Následná pozornost je věnována *VirtualBoxu* a identifikaci míst, kterými se jím vytvořené virtuální prostředí liší od fyzických počítačů. Projevy *VirtualBoxu* jsou zkoumány z pohledu operačního systému, který je v jeho rámci spuštěn, dále z pohledu virtuálního hardwaru a následně je popsáno několik možností, jak rozpoznat metodu, kterou bylo virtuální prostředí vytvořeno. Pro každý uvedený projev *VirtualBoxu* je představen postup, jak jej skrýt.

Názvosloví: Může se zdát, že slova *malware* a *virus* jsou synonyma. Ve skutečnosti je malware souhrnné označení pro veškerý škodlivý/nechtěný software. Podle chování pak rozlišujeme konkrétní druhy malwaru. Virem pojmenováváme malware, jenž infikuje ostatní programy tím, že do nich zkopíruje svůj zdrojový kód - chování obdobné biologickým virům.

Kapitola 1

Cíl práce

Stěžejním cílem je jednak popis míst, kterými se virtuální počítače odlišují od fyzických a návrh, jak tato místa zamaskovat, což může mít za následek zvýšení počtu detekcí antivirovým softwarem.

Dalším z cílů práce je uvést čtenáře do tématiky virtualizace počítačů a jejich využití při analýze malwaru. Největší pozornost je věnována hardwarové podpoře virtualizace jako prostředku k vytváření virtuálního prostředí a virtualizační nástroji *VirtualBoxu*, který jej využívá.

Jelikož je *VirtualBox* volně šiřitelný produkt[4], budou některá navrhnutá řešení implementována přímo do jeho zdrojových kódů.

Část I

Úvod do virtualizace

Teoretická část

Kapitola 2

Počítačová virtualizace

Obecně lze virtualizaci chápat jako abstrakci fyzického celku do logického objektu. Na objasnění lze použít princip virtuální reality. Díky důmyslné vizuální projekci a simulovaní dalších smyslových vjemů lze v člověku vybudit dojem, že se nachází ve skutečném prostředí. Ve své podstatě má počítačová virtualizace naprosto stejný cíl - vytvořit v počítačových programech „dojem“, že běží na skutečném počítači. Dojem lze v přeneseném smyslu chápat jako schopnost fungovat beze změny. Stejně jako se člověk nemusí přizpůsobovat virtuální realitě a jednoduše interaguje s vytvořeným prostředím, které oplácí jeho podněty očekávaným způsobem, tak software nemusí obsahovat kód, který by byl nezbytný pro jeho funkci ve virtuálním počítači. A nejen to, virtualizace může poskytnout mnoho výhod, které fyzické medium nedokáže nabídnout.

V této kapitole se podíváme na historický rozvoj počítačové virtualizace a její významnost v moderním světě informačních technologií. Dále popíšeme několik dostupných virtualizačních nástrojů a jejich odlišnosti.

2.1 Pohled do minulosti

Princip počítačové virtualizace[1] není nijak nový. První výskyt virtuálních strojů[2] lze vidět již v šedesátých letech minulého století na sálových počítačích od IBM[5]. S rozšiřováním výpočetní techniky začaly firmy stále více spoléhat na počítače. Zatím však neexistoval žádný standard pro výměnu informací a tak bylo běžné, že každý výrobce počítačů dodával své produkty s proprietárním operačním systémem, a co více, platilo pravidlo jeden počítač, jedna aplikace.

[1] dále v textu již jen virtualizace

[2] *virtual machine*/virtuální stroj, dále jen VM

S každou další aplikací tak vznikala potřeba pořídit nový stroj, což ovšem mělo dalekosáhlé důsledky nejen na potřebné místo. Nový stroj vyžadoval odborníka zajišťující jeho údržbu, výdaje na provoz a chlazení, nehledě na to, že díky Moorovu zákonu[6] výpočetní technika rychle zastarává. Moorův zákon měl další neblahý vliv na politiku jednoúčelového serveru a sice že náročnost aplikace nestoupala tak rychle jako výkon počítačů, a tak docházelo k plýtvání výkonu. Haly se tedy plnily počítači, které dělaly méně a méně práce, dokud nepřišla revoluce ve formě virtualizace[1].

2.2 Formální specifikace

V roce 1974 byla zveřejněna práce *Formal requirements for virtualizable third generation architectures*[8], která formalizuje koncepty virtuálních strojů. Pro nás je zajímavá především proto, že zavádí pojem *Virtual Machine Monitor* (VMM), který je definován jako software, který má tři zásadní vlastnosti:

1. Věrnost vytvářeného prostředí
2. Efektivita
3. Správa zdrojů

První bod lze chápat ve smyslu uvedené analogie s virtuální realitou. Program v rámci VMM by měl běžet naprosto identicky s výjimkou různých časových závislostí a dostupností zdrojů, které mohou být způsobeny souběžným během více virtuální strojů.

Efektivita vyžaduje, aby statisticky významná podmnožina instrukcí virtuálního procesoru byla vykonávána přímo na fyzickém procesoru bez zásahu VMM. Tato podmínka zabraňuje, aby tradiční emulátory, simulátory a interprety byly považovány za VMM.

Správou zdrojů chápeme běžné počítačové prostředky jako paměť, disk, síť a další zařízení. Tento bod říká, že VMM má úplnou kontrolu nad uvedenými zdroji a žádný software běžící v jeho rámci nemůže využívat zdroje, které VMM nevyhradil pro jeho použití. Toto má velmi důležitý důsledek, jelikož zabraňuje, aby se dvě různé VM mohly bez vědomí VMM přímo ovlivňovat. Do uvedených zdrojů není zahrnut procesor, což reálně umožnuje současný běh VMM a dalšího nevirtualizovaného softwaru na fyzickém stroji.

Práce [8] také představuje koncept takzvané rekurzivní virtualizace[2]. Tedy možnost virtuálního stroje spustit instanci VMM a v ní další virtuální stroj, a tak dále až do vyčerpání prostředků.

VMM je někdy nazýván též „hypervisor“. Definice neklade požadavky na virtualizovaný software a teoreticky tak lze virtualizovat jakýkoliv program. Dále se v textu budeme zabývat pouze virtualizací celého operačního systému.

[1] celá tato podkapitola silně čerpá z knihy [7]

[2] angl. Nested (vnořená) Virtualization

2.3 Hypervisor

Rozlišujeme dva procesorové módy - *kernel* a *user*. Rozdíl spočívá především v dostupných instrukcích, kde *user* mód má k dispozici pouze podmnožinu z instrukční sady procesoru. Software v *user* módu tak nemá možnost přímého přístupu k hardwaru nebo paměti, čímž je docíleno určitého stupně ochrany kritických částí operačního, které běží v *kernel* módu.

Kernel mód byl historicky označován „supervisor“, tedy něco s více pravomocemi a dozorem nad *user* módem. Název hypervisor má nejspíše evokovat představu něčeho dozorujícího nad supervisorem.

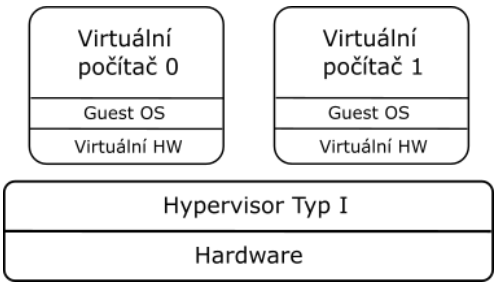
Role hypervisoru, který virtualizuje operační systém[1] je v zásadě jednoduchá. OS spuštěný ve VM, nazývaný *Guest* OS[2], musí mít přístup k hardwaru, který má k dispozici a nebo který si alespoň myslí, že má k dispozici. Pokud například *guest* požaduje čtení dat z virtuálního disku, hypervisor tento požadavek zachytí a sám tato data přečte z fyzického disku. Pozoruhodný je především fakt, že virtuální a fyzický disk mohou mít naprosto rozdílné kapacity, modely, výrobce, atd. To otevírá cestu například pro testování softwaru na různém hardwaru bez nutnosti pořizovat skutečný hardware.

Hypervisory se dělí do dvou skupin a na obě se detailně podíváme.

2.3.1 Typ I

Někdy též označován jako *bare-metal* díky svému logickému umístění přímo nad hardwarem. Viz obrázek 2.1.

Hypervisor typu I je tak efektivnější než typ II, jelikož nespoléhá na další mezivrstvu a k hardwaru přistupuje skutečně přímo. Zároveň jej lze považovat za bezpečnější, jelikož požadavky *guesta* jsou předávány přímo hardwaru a není tak příležitost, kdy by *guest* mohl narušit samotný hypervisor.



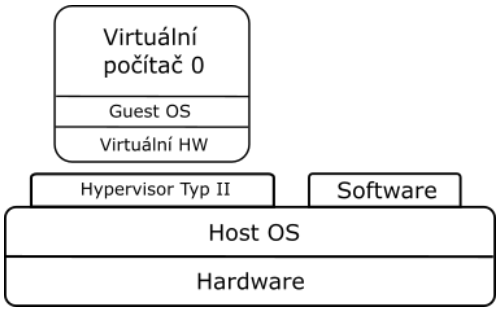
Obrázek 2.1: Hypervisor typu I

[1] dále jen OS
[2] česky Hostovaný (dále jen *guest*)

2.3.2 Typ II

Hypervisor toho typu je vytvořen programem, který je spuštěn na operačním systému. Tento OS bývá označován jako hostitelský[1].

Mezi ním a hardwarem je tak další vrstva, která se stará o správu systémových zdrojů sama. Viz obrázek 2.2. K našemu příkladu... *Guest* chce číst data z disku, který mu byl přiřazen. Hypervisor tento požadavek odchytí a místo aby přečetl data z disku sám (typ I), předá tento požadavek *hostu* a ten „obstará“ vlastní čtení. Díky této mezivrstvě je typ II pomalejší a zároveň i o něco méně stabilní, poněvadž vše co ovlivňuje stabilitu *hosta*, nepřímo ovlivňuje i hypervisor.



Obrázek 2.2: Hypervisor typu II

První hypervisory byly typu II díky své relativní jednoduchosti oproti typu I, jelikož veškeré nízko úrovňové operace byly nechány na starosti *hostu*.

2.4 Dostupné nástroje

Virtualizační nástroj je sada komponent, které umožňují spuštění VM. Nás však zajímá pouze jejich jádro a to je hypervisor. Zde popíšeme několik volně i komerčně dostupných nástrojů s významným podílem na trhu. Uvedené jsou především nuance v použitých hypervisorech. Celá tato podkapitola čerpá z [7, kap. *Comparing Today's Hypervisors*]

2.4.1 VMware

Historicky první komerční produkt, který umožňoval virtualizaci procesorové architektury x86 byl *Workstation 1.0* od společnosti *VMware*, který byl uveden na trh v roce 1999. Pravděpodobně díky svému prvenství, ale i konkurenčním výhodám zaujímá dodnes *VMware* okolo 70% trhu s x86 virtualizačními nástroji. Společnost v roce 2001 představila modely ESX a GSX, hypervisory typu I a II, z nichž GSX byl později přejmenován na *VMware Server* a dodnes

[1] *Host* OS (dále jen *host*)

je volně dostupný. Asi nejpopulárnějším produktem společnosti *VMware* je *Workstation Player*[9], jelikož se jedná o virtualizační nástroj pro osobní počítače a pro nekomerční a studijní účely je poskytován zdarma. Za jednu z konkurenčních výhod, která stojí za zmínku, je technologie *VMotion*, umožňující přesun běžícího VM z jednoho fyzického stroje na jiný bez přerušení.

2.4.2 Xen Project

Xen je původně výzkumný projekt, který vznikl na *University of Cambridge* jako open-source. Ačkoliv produkt *Citrix XenServer* má podíl na trhu necelých 5 %, díky volně dostupným zdrojovým kódům je samotný hypervisor zahrnut v mnoho produktech jako například *Amazon Web Services*[10]. *Xen* je hypervisor typu I a za zmínku stojí díky odlišnému designu. V jeho rámci vždy běží alespoň jeden *guest*, označovaný *Domain0*, který má vyšší privilegia než ostatní, jelikož má přímý přístup k hardwaru. Veškeré požadavky neprivilegovaných *guestů* (*Domain1*, ...) jsou předávány hypervisorem ke zpracování do *Domain0* a jejich výsledek je opět přes hypervisor vrácen příslušnému *guestovi*. Tato okružní cesta má bohužel za následek lehký výkonnostní dopad.

2.4.3 Microsoft Hyper-V

Hyper-V[11] je hypervisor typu I a poprvé se objevil v roce 2008 jako volitelná součást OS Windows *Server 2008*. Navzdory svému relativně pozdnímu vstupu na trh zaujímá *Microsoft* přibližně 20 % trhu. *Hyper-V* zavádí nový pojem *partition* jakožto logickou jednotku izolace, ve které běží jednotlivé *guest* OS. Podobně jako *Xen*, musí i *Hyper-V* obsahovat jednu speciální *partition* zvanou *parent* ve které musí běžet některá z podporovaných verzí Windows *Server* OS.

2.4.4 Oracle VM VirtualBox

VirtualBox je volně dostupný nástroj distribuovaný pod GPL licencí[12], který obsahuje hypervisor typu II. Původně byl vytvořen v roce 2007 německou společností *Innotek*, kterou v roce 2008 koupila společnost *Sun Microsystems* a tu o dva roky později koupila společnost *Oracle*. *VirtualBox* zde zmiňujeme jen ve stručnosti, jelikož se mu budeme podrobně věnovat v praktické části.

2.5 Výhody virtualizace

Shrňme nyní důvody proč je počítačová virtualizace tak důležitá v moderním světě informačních technologií.

2.5.1 Konsolidace

Jeden z důvodů, který je zřejmý z předchozího textu, avšak doteď nebyl pojmenován, je konsolidace. Jak bylo popsáno v podkapitole 2.1, jedním z nejpalčivějších problémů raketově se rozvíjejících informačních technologií byla spotřeba místa a energie fyzických serverů. Virtualizace umožňuje z těchto strojů udělat virtuální a spustit je na jediném fyzickém stroji.

O tomto přesunu mluvíme jako o konsolidaci a o počtu virtualizovaných počítačů mluvíme jako o konsolidačním poměru[1] - například 5 : 1. Mohlo by se zdát, že počítač na kterém běží 5 VM by měl mít 5x výkonnější hardware než jednotlivé VM. Jelikož počítače pravděpodobně nevyužívaly 100 % svého dostupného výkonu, docházelo tak k plýtvání zdroji. Pro ilustraci předpokládejme, že všech 5 počítačů využívalo procesor v průměru z dvaceti procent. Hostujícímu stroji tak stačí mít jeden stejný procesor, který bude 100 % využívám. Z administrátorského hlediska je vhodné připravit rezervu pro výkonnostní špičky. Procesor hosta by v našem příkladu měl být o něco výkonnější než původní konsolidované. O kolik přesně už záleží na povaze provozovaných aplikací.

2.5.2 Vyvažování zátěže

Další výhoda spočívá ve vyvažování pracovní zátěže. Jak jsme již uvedli, představa VM o dostupném hardwaru může být zcela odlišná od fyzických prostředků to nejen co se výrobce a typu týče, ale i počtu. Pokud například VM má k dispozici pouze jeden procesor (z jejího pohledu), ale na *hostu* jich je fyzicky více, může hypervisor plánovat běh VM na nejméně vytíženém fyzickém procesoru.

A nejen to. Pokud virtuální stroj delší dobu využívá svůj virtuální procesor na 100 %, může hypervisor tomuto stroji přidělit další. Je ovšem na *guest* OS, zda je schopen přidaný procesor za běhu rozeznat a využít. Například Linux nebo Windows od verze *Server 2008* toto dokáží. V opačném případě by bylo potřeba VM restartovat, což nemusí být žádoucí. Tato schopnost se nazývá *hot-add* a moderní operační systémy podporují přidávání procesorů i paměti za běhu. Pokud potřeba zvýšeného výkonu pomine, bylo by vhodné přidané prostředky opět vrátit hypervisoru k obsluhování ostatních VM. Odebrání za běhu však není možné a virtuální stroj je potřeba restartovat. Toto však není nedostatek hypervisorů, nýbrž operačních systémů, které *hot-remove* nepodporují.

[1] angl. *Consolidation Ratio*

2.5.3 Správa paměti

Dále se podíváme na správu paměti. Podobně jako procesor je i paměť zřídkakdy využívána zplna. Na rozdíl od procesoru může být paměť sdílena více virtuálními stroji najednou. Procesor je samozřejmě sdílen také, ale v jeden okamžik je využíván pouze jednou VM. Hypervisor tak může virtuálním strojům poskytnout více virtuální paměti než má fyzicky k dispozici. Například pokud budeme mít opět konsolidační poměr 5 : 1 a původní stroje měly k dispozici 8 GiB paměti, které využívaly v průměru z dvaceti procent, je teoreticky možné mít na hostujícím stroji také 8 GiB paměti, přestože virtuální paměť bude v součtu 40 GiB. Této technice se říká anglicky *memory overcommitment*. Opět platí, že každý rozumný administrátor připraví fyzické paměti více pro neočekávané události.

2.5.4 Sdílení paměti

U paměti ještě zůstaneme a podíváme se na techniku zvanou *page sharing*. Běžné operační systémy rozdělují paměť na takzvané stránky[13]. Je velmi pravděpodobné, že hypervisor bude obsluhovat více stejných verzí *guest* OS nebo že v nich budou spuštěné stejné programy. V takovém případě budou stejné i některé části jejich pamětí. Hypervisor tak může stránky se stejným obsahem převést na jednu sdílenou a tím ušetřit ještě více paměti. Stránku musí označit jako sdílenou pro případ, že některá VM bude potřebovat do stránky zapsat, čímž by změnila data jiné VM, která stránku sdílí. Tím by došlo k porušení bodu 2 formální specifikace. Hypervisor tedy při pokusu o zápis do sdílené stránky musí stránku nejdříve zkopírovat na volnou pozici. Jedná se o techniku *copy-on-write*.

2.5.5 Správa disku

Podobnou techniku jako *memory overcommitment* může hypervisor použít i v případě datového uložiště. Technika se nazývá anglicky *thin provisioning*, ale má jeden nedostatek. Problém nastává ve chvíli, kdy VM skutečně využívají své virtuální disky naplno. Hypervisoru tak dojde místo kam data fyzicky uložit. Všimněme si, že v případě paměti toto nečinní problém. I v běžných operačních systémech se využívá takzvaného odkládacího souboru[1]. Pokud systému dojde operační paměť, může některé stránky paměti, typicky nejméně používané, přesunout na disk a v případně potřeby opět nahrát do paměti. Přestože celý proces výrazně degraduje výkonnost celého systému, je to přijatelné řešení adaptované každým moderním operačním systémem. Pokud ovšem dojde místo na disku, není už kam přebývající data uložit. Je opět na zodpovědnosti administrátora posoudit povahu běžících aplikací a určit, zda je *thin provisioning* vhodné použít.

[1] angl. *Page* nebo též *Swap file*

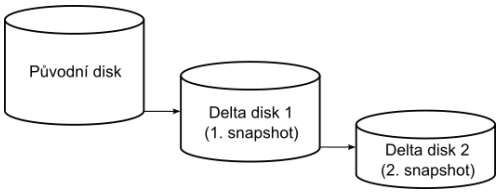
2.5.6 Klonování

Příprava fyzického počítače a prostředí vhodného pro nasazení produkční aplikace může zabrat teoreticky dny. Virtualizace tento proces sice nezkracuje, ale umožňuje jej udělat pouze jednou. Pokud firma poskytuje například VPN službu, která vyžaduje spoustu serverů na různých místech světa, může si potřebné aplikace připravit na virtuálním počítači a ten jen naklonovat na všechny servery. Klonování není nic jiného než kopírování, jelikož VM je v podstatě jen sada souborů v konkrétním formátu, například OVF - *Open Virtualization Format*[14]. Jedná se o otevřený formát, který podporují všechny výše uvedené virtualizační nástroje. To otevírá možnost přípravy virtuálního stroje například pomocí *VMware*, i když produkční prostředí používá *Xen*. OVF specifikuje i formát ve kterém lze přenášet všechny soubory virtuálního stroje v jednom souboru - OVA, *Open Virtual Appliance*. Jedná se jednoduše o *TAR* archiv[15] OVF souborů.

2.5.7 Snímek

Jak bylo uvedeno, VM je sada souborů. Za běhu virtuálního stroje se tyto soubory mění, ukládají a mažou se data virtuálního disku, mění se hardwarová konfigurace a tak dále. Snímek[1] je metoda, jak zachovat konkrétní stav VM a později se k němu vrátit. VM ukládá veškerá data na virtuální disk. Jedná se o soubor, nejčastěji ve formátu VMDK - *Virtual Machine Disk*[16], kterému rozumí běžné virtualizační nástroje.

V momentu vytvoření snímku se vytvoří nový virtuální disk a veškeré změny se od tohoto okamžiku ukládají do nového VMDK souboru. V případě obnovení snímku se místo originálního virtuálního disku použije tento nový, někdy nazývaný *delta* nebo *child* disk. Jelikož hypervisor ukládá na delta disk pouze změněná data, šetří tak místo na fyzickém disku. Ovšem nastávají komplikace, když VM vyžaduje data, ležící na původním disku. Hypervisor musí původní disk nejdříve načíst a data na něm nalézt. To se nemusí zdát jako problém, ale musíme si uvědomit, že je možné vytvářet snímky snímků a mít tak *delta* disk, který vznikl z jiného *delta* disku. Viz obrázek 2.3



Obrázek 2.3: Hierarchie snímků

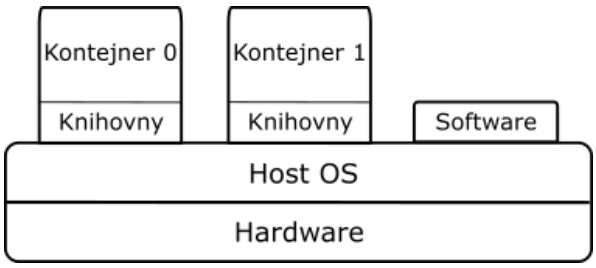
[1] angl. Snapshot

Hypervisor tak musí celou tuto spojovou strukturu projít zpětně až k původnímu disku a to může představovat výkonnostní problémy. Snímky proto nejsou náhradou standardních záloh, ale měly by sloužit pouze jako dočasný bod ke kterému se lze vrátit například pro potřeby testování.

2.5.8 Kontejnery

Doteď jsme mluvili o virtualizaci operačních systému na úrovni hardwaru a budeme se jí věnovat i nadále. Uděláme však malou odbočku do světa kontejnerů jelikož stojí za zmínku díky jejich stále rostoucí popularitě.

Zaprvé, kontejnery nevirtualizují hardware. Kontejnerové programy vidí a využívají stejné zdroje jako jeho hostující OS a pro komunikaci s hardwarem využívají přímo tento OS. Typicky se jedná jednu aplikaci a všechny její datové závislosti, například knihovny. Neobsahují už samotný operační systém ani virtuální hardware jako je to v případě VM. Z toho vyplývá, že *host* OS může obsahovat kontejnery připravené pouze pro daný OS. Viz obrázek 2.4



Obrázek 2.4: Kontejnery

Účel kontejnerů je oddělit jednotlivé aplikace. Stejně jako v případě virtuálních strojů, nemají aplikace v různých kontejnerech ponětí o ostatních a nemohou se tak ovlivňovat.

Výhoda kontejnerů spočívá v jejich jednoduchosti a snadném nasazení. Vytvořit celou VM kvůli jedné aplikaci znamená věnovat čas úkonům, které se samotnou aplikací nesouvisí, nemluvě zbytečném množství dat.

V této kapitole jsme představili počítačovou virtualizaci od jejích počátků až do dnešního rozmachu. Virtualizace zjednodušila spoustu zdlouhavých úkonů a výrazně snížila náklady firem na pořízení výpočetní techniky potřebné pro jejich činnost. Zároveň umožnila revoluci ve formě *cloud computing*[47], které se bohužel nemáme prostor se věnovat. S rozšiřováním virtualizace se začali přizpůsobovat i výrobci procesorů, kteří do svých výrobků přidali nativní podporu virtualizace, čímž ještě více zefektivnili běh virtuálních strojů a zjednodušili architekturu hypervisorů. Jakými způsoby tohoto dosahují bude náplní další kapitoly.

Kapitola 3

Hardwarová podpora virtualizace

Předchozí kapitolu jsme věnovali obecnému popisu virtualizace a jejím přínosům. Ještě než přistoupíme k praktické části práce musíme popsat, jak výrobci procesorů implementují podporu virtualizace, jelikož mnoho pojmů zde vysvětlených budeme využívat v praktické části.

Největší výrobce procesorů s architekturou x86 - Intel uvedl hardwarovou podporu virtualizace[1] v roce 2005 pod označením VT-x[17]. Intel používá spíše označení VMX[2] a i my se ho budeme držet.

Druhý největší výrobce - AMD představil svou implementaci o rok později pod názvem SVM[3][18]. Obě implementace jsou velmi podobné, ale jelikož AMD vlastní pouze 20% podíl trhu s x86 procesory[19], budeme se věnovat výhradně VMX. Zároveň nás také jednotlivé nuance mezi VMX a SVM nebudou v implementační části zatěžovat, protože nás od nich *VirtualBox* odstíní.

V této kapitole budeme často odkazovat na technickou specifikaci poskytovanou Intelem [20]. Jelikož specifikace je velmi rozsáhlá, budeme uvádět konkrétní kapitoly, odkud jsme čerpali. Čtenář tak může specifikaci použít pro získaní detailnějšího vhledu, jelikož internetové články na téma VMX bývají velmi strohé.

Kapitola též obsahuje několik ukázek kódu v jazyce C, ve kterých jsou použity konstrukce specifické pro OS Windows. Jmenovitě například hlavičkový soubor intrin.h[4], který umožňuje volat procesorové instrukce jako funkce.

[1] angl. *Hardware-assisted Virtualization*
[2] angl. *Virtual Machine Extensions*
[3] angl. *Secure Virtual Machine*
[4] angl. *Compiler Intrinsics*

3.1 Aktivace VMX

Ačkoliv převážná většina nových procesorů obsahuje VMX, musí každý software, který jej chce využívat nejprve zjistit, zda procesor skutečně VMX podporuje. Zjištění podpory se provádí pomocí instrukce CPUID[1]. Ta se využívá pro zjištění detailů o procesoru a přijímá argumenty v registrech EAX a ECX. Výsledek instrukce je pak uložen v EAX, EBX, ECX a EDX. Pokud je zavolána s argumenty EAX=1, ECX=0, bude bit na páté pozici v registru ECX určovat, zde je VMX podporováno - 1 znamená ano. Viz zdrojový kód 3.1

```
#include <basetsd.h>
#include <intrin.h>

BOOL VMXSupported (VOID)        // user mód
{
    INT32 CPUInfo[4];           // 0=EAX, 1=EBX, 2=ECX, 3=EDX

    __cpuid(CPUInfo, 1);        // EAX=1, ECX=0(výchozí)

    return (CPUInfo[2] >> 5) & 1;
}
```

Zdrojový kód 3.1: Zjištění podpory VMX

Pokud je VMX podporováno, je potřeba jej aktivovat nastavením VMXE bitu (13. pozice) registru CR4[2]. Viz zdrojový kód 3.2

```
VOID VMXEnable (VOID)           // pouze kernel mód
{
    UINT64 cr4 = __readcr4();
    cr4 |= (1 << 13);
    __writecr4(cr4);
}
```

Zdrojový kód 3.2: Aktivace VMX

Toto umožní provedení instrukce VMXON, která by jinak způsobila výjimku typu neplatný instrukční kód - #UD[3]. Viz podkapitolu 3.2. Úplný výpis podmínek potřebných pro úspěšné provedení instrukce VMXON lze najít v kapitole *VMXON—Enter VMX Operation* technické specifikace.

[1] kapitola *Processor identification and feature determination*[20]
[2] *Control Register*
[3] UnDefined Opcode, kapitola *Interrupts and exceptions*[20]

VMXON vyžaduje jeden operand a tím je adresa paměti, vyhrazená pro potřeby VMX. Tento paměťový prostor se nazývá VMXON *Region* a jeho adresa VMXON *Pointer*. Adresa musí být zarovnaná na 4 kiB (bity 0 až 11 musí být 0) a velikost regionu je závislá na modelu procesoru, nikdy však nesmí zabírat více než stránku pamětu - typicky 4 kiB. Vyžadovanou velikost lze zjistit přečtením bitů 44:32 IA32_VMX_BASIC MSR registru[1]. Dále je potřeba na začátek regionu uložit identifikátor revize VMCS, který je definován bity 31:0 MSR registru IA32_VMX_BASIC. Viz podkapitolu 3.3. Více v sekci *VMXON Region* technické specifikace.

```
#include <Ntddk.h>                  // pouze kernel mód
#define MSR_IA32_VMX_BASIC    0x480

...

UINT64 VMXBasic = __readmsr(MSR_IA32_VMX_BASIC);
UINT16 VMXRegionSize = (VMXBasic >> 32) & 0xfff;
UINT32 VMCSRevisionIdentifier = VMXBasic & 0xffffffff;

UINT8* VMXRegion =
  (UINT8*)MmAllocateContiguousMemory(VMXRegionSize, ...);

(UINT32*)VMCSRegion)[0] = VMCSRevisionIdentifier;

__vmx_on(VMXRegion);

...

__vmx_off();
MmFreeContiguousMemory(VMXRegion);

...
```

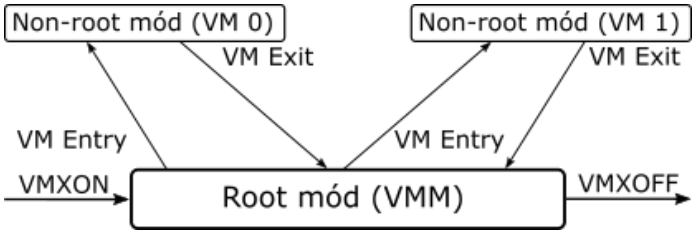
Zdrojový kód 3.3: Příprava VMXON *Region*

3.2 VMX operace

Vykonáním instrukce VMXON vstupuje procesor do takzvané VMX operace. Chování procesoru zůstává stejné, jediným rozdílem je zpřístupnění sady nových instrukcí. Viz přílohu B. Opuštění VMX operace se provádí instukcí VMXOFF. Rozlišujeme dva typy VMX operace - *root* a *non-root*. Ve výchozím stavu se procesor nachází v *root* módu a obecně platí, že hypervisor běží právě v *root* módu a jím spravované VM v *non-root* módu. Vstup do *non-root* nazýváme VM *Entry* a výstup zpět do *root* VM *Exit*[2]. Viz obrázek 3.1.

[1] *Model Specific Register*

[2] Též „výstup do hypervisoru“



Obrázek 3.1: Módy VMX operace[20]

3.3 Virtual Machine Control Structure

Virtuální stroje jsou reprezentovány VMCS strukturou - Virtual Machine Control Structure. Hypervisor správou této struktury kontroluje chování VM v *non-root* módu. Přesněji řečeno tato struktura definuje průběh a podmínky VM *Entry* a VM *Exit*.

3.3.1 Organizace VMCS

Hypervisor musí pro VMCS nejprve připravit paměť. Ta se nazývá VMCS *Region* a ukazatel na ní VMCS *Pointer*. Pro tuto paměť a její ukazatel platí stejné podmínky jako pro VMXON *Region* a *Pointer* a zároveň musí první 4 byty obsahovat stejné číslo revize jako VMXON *Region*. Jakékoliv operace nad VMCS selžou pokud tyto revize nebudou shodné.

Následující 4 byty obsahují VMX Abort indikátor. V případě chyby při VM *Exit* je na tuto pozici uložen chybový kód a je doporučeno tuto oblast před VM *Entry* vynulovat.

Zbývající prostor vyplňují VMCS data a jejich formát je závislý na modelu procesoru. To však nevadí, jelikož VMCS by nikdy nemělo být čteno přímo, ale pouze za pomoci instrukce VMREAD. VMCS data jsou rozdělena do šesti skupin.

3.3.1.1 Guest-state area

Položky z této části popisují procesorový stav VM a jsou načteny při VM *Entry* a automaticky zapsány při VM *Exit*. Hypervisor musí tuto část vhodně nastavit před prvním VM *Entry*. Obsaženy jsou zde například kontrolní a segmentové registry, *stack* a *instruction pointer* registr a další. Mimo registrové hodnoty jsou zde i části popisující nastavení přerušení, stránkovací informace atd. Úplný seznam lze nalézt v části *Guest-state area* technické specifikace.

Za zmínku stojí položka VMX-*preemption timer*, což je časovač, který, pokud je použit, určuje nejzazší možný VM *Exit*. Hodnota tak definuje maximální možnou dobu strávenou v *non-root* módu.

Další významná položka je už zmíněný *instruction pointer* - RIP[1]. Jeho hodnota říká na jaké instrukci začne *non-root* mód. Ukázka nastavení:

```
#define GUEST_RIP 0x0000681e        // pouze kernel mód

...
___vmx_vmwrite(GUEST_RIP, (uintptr_t)VMEntry);
...
___vmx_vmlaunch();                  // viz 3.3.2
...

// Vstupní bod VM (po zavolání ___vmx_vmlaunch())
VOID VmEntry (VOID) {...}
```

Zdrojový kód 3.4: Nastavení vstupního bodu VM

Kompletní výpis VMCS položek a jejich kódování lze nálezt v sekci *Field encoding in VMCS* technické specifikace.

3.3.1.2 Host-state area

Tato sekce popisuje procesorový stav hypervisoru, který bude obnoven při VM *Exit*. Oproti *guest-state area* obsahuje pouze registrové hodnoty. Viz sekci *Host-state area* technické specifikace. Důležitá položka je opět RIP, jeho hodnota udává, odkud bude hypervisor pokračovat při VM *Exit*. Nejčastěji se bude jednat o funkci, která zjistí důvod VM *Exitu* a podle něj se zachová, tzn. zavolá pomocné rutiny, vhodně nastaví *guest-state area* a opět provede VM *Entry*. Viz 3.3.1.6.

3.3.1.3 VM-Execution control fields

Položkami v této sekci lze detailně řídit průběh *non-root* módu. Popisuje jak se má procesor zachovat v případě konkrétních akcí jako například přerušení, výjimky, čtení a zápisu do registru, vykonání určité instrukce a tak dále. Respektive zda tyto akce mají způsobit VM *Exit*.

Například položka *RDTSC exiting*. Pokud je nastavena, vykonání instrukce RDTSC[2] způsobí VM *Exit*. Standardně tato instrukce vrací v registrech EDX a EAX počet cyklů vykonaných procesorem od spuštění a lze ji použít k měření času. Pokud hypervisor zjistí, že důvodem VM *Exit* byla instrukce RDTSC, viz 3.3.1.6, může do registrů EDX a EAX uložit jím vybranou hodnotu a tím podvrhnout čas z pohledu VM. Tyto registry bohužel nejsou součástí guest-state area a je tak potřeba je nastavit přímo[3] před VM *Entry*. O důvodu

[1] Na 32 bitové architektuře označován EIP

[2] *Read Time Stamp Counter*

[3] Windows poskytují funkce `RtlCaptureContext` a `RtlRestoreContext`

podvrhování času více v podkapitole 8.4. Lze říci, že čím více akcí způsobí VM *Exit*, tím větší má hypervisor kontrolu nad VM.

Uveďme, že existují i akce, které způsobí nepodmíněné výstupy do hypervisoru. Tedy takové, které nastanou vždy a nelze je ignorovat nastavením v této sekci.

3.3.1.4 VM-Exit control fields

Zde hypervisor nastavuje chování při VM Exit. Jmenovitě například zda uložit *preemption timer* nebo MSR registry. Více v sekci *VM Exit control fields* technické specifikace.

3.3.1.5 VM-Entry control fields

Podobně jako v předchozí sekci lze zde nastavit, jaké akce provést při VM *Entry*. Zajímavá je především možnost *event injection*. Díky ní lze vyvolat simulované přerušení nebo výjimku. Více v sekci *VM Entry control fields* technické specifikace.

3.3.1.6 VM-Exit information fields

Nejdůležitější část této sekce je *Exit Reason* - důvod VM *Exitu*. Na základě této informace se může hypervisor rozhodnout, jak se dále zachovat a případně obsloužit potřeby VM. Vypisů důvodů VM *Exit* a jejich hodnoty lze nalézt v sekci *VMX Basic exit reasons* technické specifikace. Náčrt funkce obsluhující VM Exit:

```
#define VM_EXIT_REASON      0x00004402
#define EXIT_REASON_RDTSC 16

VOID VMExitHandler (VOID)       // pouze kernel mód
{
    UINT16 ExitReason;
    __vmx_vmread(VM_EXIT_REASON, &ExitReason);

    switch(ExitReason)
    {
        ...
        case EXIT_REASON_RDTSC:
            HandleRDTSCReason(); break;
        ...
    }
    ...
    __vmx_vmresume();              // viz 3.3.2
}
```

Zdrojový kód 3.5: Obsluha VM *Exit*

3.3.2 VMX instrukce a životní cyklus VMCS

Spravuje-li hypervisor více VM, znamená to, že se musí starat o více struktur VMCS. V předchozím textu si lze všimnout instrukcí jako například VMREAD, které operují nad VMCS, avšak nedostávají explicitně parametr o jakou VMCS se jedná. VMCS se může nacházet různých v stavech, popsaných těmito atributy:

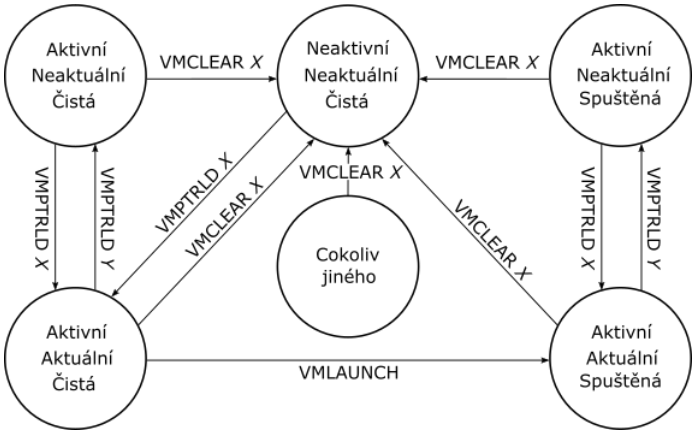
- Aktivní (Active/Inactive)
- Aktuální (Current/Not current)
- Spuštěná (Launched/Clear)

Více VMCS může být aktivních, ale vždy je pouze jedna aktuální. Instrukce VMREAD a VMWRITE používané pro čtení, resp. zápis do VMCS a instrukce VMLAUNCH a VMRESUME, které se používají pro VM *Entry*, manipulují nad **aktuální** VMCS. Instrukce selžou, pokud žádná VMCS není aktuální, což je signalizováno nastavením CARRY bitu v příznakovém registru FLAGS.

Celý životní cyklus VMCS lépe vysvětlí obrázek 3.2. Při prvním vstupu do *root* módu vykonáním instrukce VMXON, není žádná VMCS aktivní ani aktuální. Poté co hypervisor dokončí přípravu, vykoná instrukci VMPTRLD *X*, kde *X* je VMCS Pointer, čímž se tato VMCS stává aktivní a aktuální. Pokud jiná VMCS byla v tu chvíli aktuální, přechází do stavu aktivní a neaktuální. Stav ostatních VMCS se jinak nemění. Opačnou instrukcí VMCLEAR *X* přechází VMCS do stavu neaktivní a neaktuální bez ohledu na to v jakém stavu se nacházela. Pokud tato VMCS byla aktuální, pak se žádná jiná nestane aktuální. Vykonáním instrukce VMLAUNCH se aktuální VMCS dostane do stavu spuštěná - nastane VM *Entry* a v tomto stavu zůstává do vykonání VMCLEAR, tedy i po VM *Exit*. Nad spuštěnou VMCS se doporučuje používat VMRESUME z důvodů menších režijních nákladů - není potřeba měnit stav VMCS. Vykonání VMRESUME nad nespuštěnou VMCS selže, což je signalizováno nastavením ZERO bitu v příznakovém registru FLAGS.

Veškeré výše uvedené VMX instrukce, společně s pár dalšími[1], způsobí nepodmíněný VM *Exit*, jsou-li vykonány v *non-root* módu. I ten nejjednodušší hypervisor tak musí být schopen obstarat minimálně tyto VM *Exit*y. Jedna z nich, která stojí za zmínění je VMCALL, která nemá žádný další efekt kromě vyvolání VM *Exit*. Podpůrný software hypervisoru běžící ve VM může tuto instrukci využít pro komunikování s hypervisorem. Rozhraní této komunikace není definováno ve VMX a je tudíž volitelné. Lze například využít registry pro předávání příkazů a dat. Hypervisor tak může obsluhovat potřeby *guesta*, které nejsou definované v technické specifikaci. Viz zdrojový kód 3.6.

[1] kapitola *Instructions That Cause VM Exits Unconditionally*[20]



Obrázek 3.2: Životní cyklus VMCS[20]

```
INT32 VmCall (INT32 cmd)        // non-root mód
{
    SetEcx(cmd);                // pseudofunkce
    ___vmx_vmcall();
    return ReadEcx();           // pseudofunkce
}

...

VOID VMExitHandler (VOID)       // root mód
{
    ...                         // viz zdrojový kód 3.5
    case EXIT_REASON_VMCALL:
        switch(ReadEcx())
        {...}
    ...

    // Zjisti délku instrukce, která způsobila VM Exit
    // a přičti ji k instruction pointeru VM.
    // Po VM Entry tak bude VM pokračovat přesně kde skončila
    // - zde konkrétně za ___vmx_vmcall();
    INT32 ExitInstrLen = ___vmx_vmread(VM_EXIT_INSTRUCTION_LEN);
    INT64 GuestRIP = ___vmx_vmread(GUEST_RIP);

    GuestRIP += ExitInstrLen;

    ___vmx_vmwrite(GUEST_RIP, GuestRIP);
    ___vmx_vmresume();
}
```

Zdrojový kód 3.6: Obsluha VMCALL a následný VM *Entry*

Ukázali jsme několik základních částí, které tvoří srdce každého hypervisoru využívajícího podporu VMX. Spuštění kompletně virtualizovaného operačního systému je stále kilometry daleko. Pro představu - napsání vlastního BIOSu, správy paměti, ovladačů a tak dále. Není v zájmu a pravděpodobně ani v silách autora tohoto dosáhnout, a proto pro naše cíle využijeme již existující a volně dostupnou implementaci - *VirtualBox*.

Část II

Stopy virtualizace a jejich zakrytí

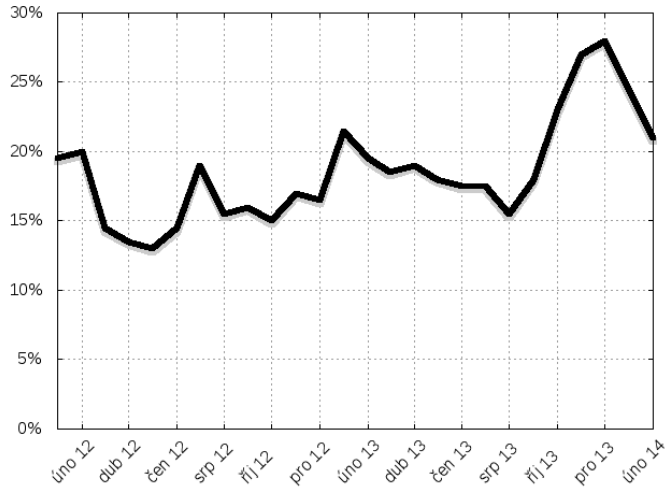
Praktická část

Kapitola 4

Malware a virtualizace

Nyní již máme pevné teoretické základy z oblasti virtualizace a můžeme přistoupit k jejímu spojení se světem malwaru. Antivirové společnosti vedou boj, ve kterém jsou odsouzeny být neustále o krok pozadu za svými protivníky. Mají-li se dostat do vedení, musí vědět, co jejich oponent vymyslí ještě dříve, než to ví on sám a to se zdá nemožné. Ačkoliv je popis různých forem malwaru fascinující oblast, není náplní naší práce. Zde se zaměříme pouze na jednu ze zbraní opačné fronty a pokusíme se jí zlepšit. Řeč je samozřejmě o virtualizaci a její role je nejspíše patrná. Denně na světě vznikají tisíce a tisíce nových programů a byť se k analýze dostane jen zlomek, není v lidských silách každý z nich ručně rozebrat a označit za neškodný či nebezpečný. Většinu práce tak přebraly stroje, které celý průchod analýzou automatizují. Jednou ze zastávek je statická analýza kdy je neznámý vzorek prohledán na výskyt specifických řetězců. Tvůrci malwaru si tohoto jsou samozřejmě vědomi a tak své výtvory vybavují protiopatřeními. Na vzorku tak nelze poznat jak se zachová, dokud se skutečně nespustí. A to je příležitost pro virtuální stroje, které hrají roli pokusného pacienta, jež můžeme beztrestně infikovat neznámým vzorkem a sledovat jak pacient dopadne. Pokud nepřežije, je vzorek přinejmenším velmi podezřelý. My však můžeme ze skříně vytáhnout dalšího pacienta ve formě snímku. Malware si je bohužel vědom i tohoto a tak se snaží různými způsoby rozpoznat, zda se právě nacházejí v nastrčeném pacientovi a pokud pojme dostatečné podezření, zdrží se škodlivých aktivit. Díky tomu může proplout celou analýzou nerozpoznán a na konci být prohlášen za „čistý“.

Bohužel neexistují žádné aktuální statistiky, které by uváděly kolik procent malwaru se pokouší o detekci VM. Asi se není čemu divit, pro vytvoření takovéto statistiky by bylo potřeba velkého množství vzorků a každý spustit jak ve virtuálním počítači tak na fyzickém, najít rozdíl ve výsledcích a určit zda skutečně vznikl detekcí virtuálního prostředí. To se jeví jako velmi pracná a nudná činnost. Druhá metoda, která se nabízí je přidat do automatické analýzy detekci známých metod pro rozpoznání VM. To znamená sledovat veškerou činnost vzorku a pokud udělá něco o čem se ví, že má rozdílný výsle-



Obrázek 4.1: Poměr malwaru detekující *VMWare*[21]

dek na fyzickém a virtuální počítači, tak činnost zaznamenat. Detekce těchto činností však nikdy nemůže být stoprocentní, stejně jako detekce malwaru samotného nemůže být stoprocentní. Vyvstává tedy otázka, jak spolehlivé by byly výsledky této metody.

Je vidět, že vytvoření této statistiky by bylo velmi náročné a proto se jí pravděpodobně nikdo nezabývá a ani my ji nebudeme pro účely této práce vytvářet. Poslední kdo se podobnou statistikou zabýval byla společnost Symantec v roce 2014[21]. Viz obrázek 4.1, který ukazuje, že průměrně jeden z pěti vzorků malwaru detekuje *VMWare* a ukončí proces. Dá se očekávat, že toto číslo bude o něco menší. Díky výhodám virtualizace je stále více důležitých systémů a infrastruktur postaveno na virtuálních strojích a tvůrci malwaru se nechtějí ošidit o takto zajímavou oběť.

Předpokládejme, bez hlubšího opodstatnění, že opatrného malwaru bude kupříkladu 5 procent. Pokud uvážíme, že úspěšnost detekce například antiviru *Avast* je přes 90 procent[22], pak je jakýkoliv přírůstek k této míře úspěšnosti velmi cenný a úprava virtuálních strojů se jeví jako velmi slibná oblast.

Tuto část práce dále rozdělíme na tři kapitoly. V první popíšeme Windows a jak jsou poznamenány virtualizačním nástrojem ve kterém běží. Druhou věnujeme rozdílům mezi skutečným a virtuálním hardwarem a ve třetí se pokusíme identifikovat nesrovnalosti v samotném jsoucnu virtuálního prostředí.

Každou kapitolu následně rozčleníme na sekce, kde popíšeme jednotlivé prvky, jak je malware může zneužít a jak mu v tom můžeme zabránit. Veškerý náš výzkum a pokusy se točí kolem *VirtualBoxu*. Všechna protiopatření se však snažíme navrhovat dostatečně obecně, aby mohla být aplikována na jiné nástroje.

Kapitola 5

Použité nástroje

Popišme nejdříve s jakými nástroje jsme přišli do styku

5.1 Visual Studio Code

Jelikož je *VirtualBox* multiplatformní projekt, neexistuje jednoduchý způsob jak celý kód naimportovat do *Visual Studia*[1], které je dostupné pouze pro Windows.

Microsoft pro editaci kódu nabízí odlehčenou alternativu *Visual Studio Code*, která obsahuje vše pro nás podstatné - vyhledání deklarací a definic, našeptávání, zvýraznění syntaxe a tak dále. *Visual Studio Code*[23] je volně dostupný program s otevřeným zdrojovým kódem distribuovaným pod MIT licencí[24].

5.2 Windows Sysinternals

Sysinternals je sada nástrojů diagnostiku, správu a monitorování interních mechanismů Windows. My je použijeme pro ověření některých našich změn. Sysinternals jsou dostupné volně ke stažení[25].

5.2.1 WinObj

Jednou z vnitřností je *Object Manager*[26], který se stará o správu objektů v jádře Windows. Objekt může reprezentovat například soubor, knihovnu nebo zařízení. Díky WinObj uvidíme objekty, které může malware zneužít pro identifikaci *VirtualBoxu*.

[1] Ve zdrojém kódu není Solution (.sln) soubor

5.2.2 ProcExp

Process Explorer nabízí detailnější pohled na procesy než vestavěný *Task Manager* (taskmgr.exe). Zobrazuje mimo jiné hierarchickou strukturu procesů, počet vláken a pro nás zajímavé objekty, které konkrétní proces používá.

5.2.3 DbgView

Pro snazší diagnostiku našich změn v hypervisoru budeme používat funkce jako `DbgPrint` nebo `KdPrint`. Jejich výstup je určen pro kernelový debugger, což by znamenalo mít dva fyzické počítače - na jednom mít spuštěný *VirtualBox* a na druhém sledovat vypisované hlášky. *Debug View* nám umožní vidět výstupy z těchto funkcí přímo na počítači, kde budeme testovat naše změny.

5.3 WinDbg

Bez kernelového debuggeru se tak úplně neobejdeme, jelikož některé naše změny ve VM nastanou už při startu VM, kdy nemáme k dispozici DbgView.

WinDbg je debugger obsažený v balíčku Windows *Driver Kit*[27], ale lze jej stáhnout i samostatně. My se jím „vzdáleně“[28] připojíme k virtuálnímu počítači a budeme sledovat dění ve VM během celého jeho běhu [1].

Jen pro upřesnění - pomocí DbgView sledujeme námi vypisované hlášky v hypervisoru a díky WinDbg máme úplnou kontrolu nad VM.

5.4 VirtualKD

VirtualKD[29] je protikus WinDbg, který musí být nainstalován ve VM, kde vytvoří pojmenovanou rouru[2][30] ke které se WinDbg připojí. VirtualKD je licencován pod LGLP[31] licencí.

[1] zobrazování hlášek je potřeba umožnit příkazem `eb nt!kd_default_mask 0x0F`

[2] *angl. Named Pipe*

Kapitola 6

Windows

Vžijme se teď role malwaru, který se ocitl v potenciálním virtuálním stroji s OS Windows, kde veškeré jeho nekalé akce jsou sledovány a mohou způsobit jeho zapsání na seznam hledaných. Víme, že *VirtualBox* je nežádoucí prostředí, které je ovšem na spoustě místech zakořeněné ve Windows a tak se nejprve vydáme tato místa zkontrolovat.

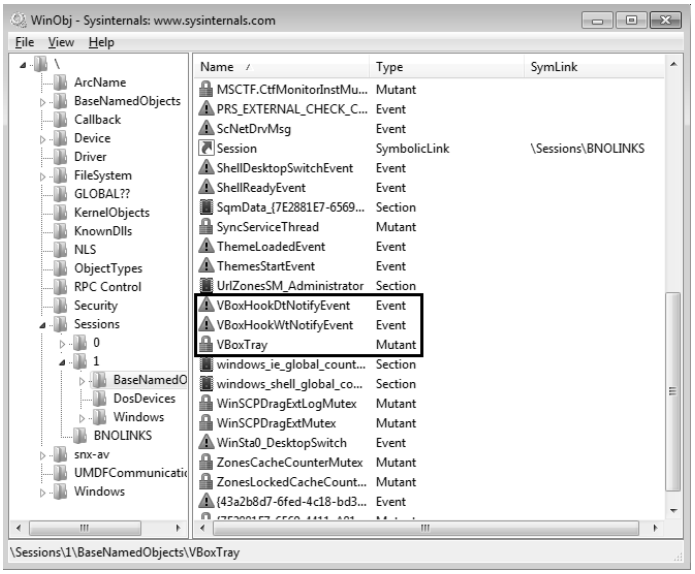
Spoustu kritických míst vytváří *Guest Additions*[32], které zároveň velice zpříjemňují práci s VM díky lepší integraci myši, plynulejšímu vykreslování grafiky, vytvoření sdílených složek s *host* OS a tak dále. Naše práce předpokládá nainstalované *Guest Additions* a věnuje se i skrývání jimi vytvořených kompromitujících míst.

6.1 Pojmenované objekty

VirtualBox pro své vnitřní potřeby vytváří několik pojmenovaných objektů[33]. Jelikož ty jsou vytvářeny vždy s pevně daným jménem, typicky ve tvaru „VBox*“, může malware díky existenci takového objektu s jistotou určit, že je spuštěn ve *VirtualBoxu*. Viz obrázek 6.1. Popišme nejdříve několik důležitých pojmů.

6.1.1 Handle

Windows implementují pseudo-objektově orientovaný přístup k datovým strukturám. Objekty v jádře OS, například soubor nebo vlákno, mají své atributy a Windows nabízejí funkce, kterými lze s objetky manipulovat, nedovolují však k nim přistupovat přímo. Objekty jsou tak reprezentovány pomocí takzvaného *handle*. Ten lze zjednodušeně chápat jako *pointer*, Windows však při jeho použití kontrolují některé atributy jako například oprávnění nebo zda byl vytvořen pro zápis pokud je požadována modifikace objektu, atd.



Obrázek 6.1: WinObj, původní pojmenované objekty

Pro snazší meziprocesové sdílení *handlů* k synchronizačním objektům jako jsou mutexy[34], semafory[35] nebo eventy[36], umožňují Windows tyto objekty pojmenovat. Pokud proces vytvoří mutex pomocí funkce `CreateMutex`, může specifikovat jeho název[1]. Jiný proces pak může použít tento název ve funkci `OpenMutex` a tím získat *handle* ke stejnému objektu. Jako malware můžeme použít stejnou funkci a pokud uspěje[2] víme, že se nacházíme ve *VirtualBoxu*.

6.1.2 Jmenný prostor objektů

Při vytváření objektů lze určit namespace ve kterém budou objekty dostupné. Namespace je určen prefixem před jménem a určuje z jakých uživatelských session [3] budou objetky dostupné. Prefix *Global* znamená, že objekt bude dostupný ze všech session. Prefix *Local* omezí dostupnost objektu pouze na konkrétní session. Pokud prefix není uveden, použije se implicitně *Local*. V různých session tak můžou existovat objekty se stejným názvem, v jádru OS to však budou rozdílné objekty.

6.1.3 Skrývání

Nabízí se několik metod jak pojmenované objekty zamaskovat. Naivní metoda spočívá v odstranění předpony „VBox“ nebo vymyšlení jiného, ovšem stále

[1] jedná se o volitelný parametr
[2] tzn. nevrátí NULL
[3] zjednodušeně „pro jaké uživatele“

napevno zakódovaného jména. Ta bude úspěšná pouze do doby než se malware dozví toto nové jméno. To může být teoreticky velmi dlouho a v případě kompromitace by stačilo objekty znovu přejmenovat. Samotný problém však tento způsob neřeší, pouze odsouvá.

Jako rozumná alternativa se jeví generovat jména dynamicky, ideálně při každém vytvoření objektu. To samo o sobě není problém, vyvstává však otázka, jak vygenerovaná jména distribuovat mezi součásti, které jména potřebují znát. Může se jednat o rozdílné procesy a pro některé objekty je potřeba znát jméno v *user* i *kernel* módu. Tyto části nesdílí nebo ani nemohou sdílet paměťový prostor a je tedy potřeba vymyslet jinou metodu distribuce jmen. Pro tyto účely využijeme hypervisoru, který má nad VM úplnou kontrolu.

Než tuto podkapitolu dokončíme musíme popsat způsob komunikace mezi VM a VMM.

6.2 CPUID komunikace

V části 3.3.2 jsme uvedli, že pomocí instrukce VMCALL se může *guest* dovolat pozornosti hypervisoru. VMCALL je přímo určen pro výměnu dat mezi VM a hypervisorem, jelikož způsobuje nepodmíněný VM Exit. Rozhodli jsme se ji však nepoužít z důvodů uvedených v podkapitole 8.3.

Pro řešení tohoto problému jsme zvolili použití instrukce CPUID. Ta stejně jako VMCALL způsobuje nepodmíněný VM *Exit*, ale na rozdíl od VMCALL je CPUID běžná instrukce dostupná i v nevirtualizovaném prostředí.

Hypervisor tedy při VM *Exitu* způsobeném CPUID zkontroluje zda registr EAX obsahuje předem domluvenou konstantu:

CPUID_COMM_MAGIC = 0x41564153 ('AVAS')[1]

Její hodnota nesmí kolidovat s žádnou běžně používanou. Tato konstanta znamená požadavek na komunikaci a v závislosti na hodnotě v registru ECX hypervisor vhodně nastaví výstupní hodnoty. Fakt, že komunikace má možnost použít jen čtyři 32bitové registry znamená, že návratová hodnota může být nejvíce 16bytová.

6.2.1 Generování náhodných řetězců

Vytváření náhodných řetězců je ponecháno na VMM, ne na procesu, který jej vyžaduje. K tomu potřebujeme generátor náhodných čísel, ovšem kód hypervisoru je stejný pro všechny operační systémy, které *VirtualBox* podporuje a není tak možné použít funkce dostupné pouze ve Windows, např. `RtlRandom`. Zároveň není implementovaný proprietární generátor náhodných čísel a tak jsme si museli napsal svůj.

[1] velikostí jsme omezeni na 32bitový registr - 4 znaky

Zvolili jsme lineární kongruenční generátor, který sice není kryptograficky bezpečný, věříme však že pro naše účely bude dostatečný.

$$X_{n+1} = (aX_n + c)\ mod\ m$$

Kde:

- X_0: *seed*
- a: násobič = 1103515245
- c: inkrement = 12345
- m: modul = $2^{31} - 1$

Hodnoty jsou stejné jako ve standardní knihovně jazyka C. *Seed* je inicializován při startu VMM na hodnotu *Times Stamp Counteru*.

Náhodné řetězce generujeme ze staticky alokovaného pole malých, velkých písmen a číslic 0 - 9.

6.2.2 Výstupní komunikace

Jednotlivé objekty, které lze od VMM požadovat mají své identifikátory, které jsou parametrem komunikace. Pokud je objekt vytvářen nebo je potřeba jeho jméno obnovit, je nutné nastavit *regenerate* bit - nejvyšší bit parametru. V opačném případě bude vráceno již vygenerované jméno nebo prázdný řetězec.

Před každé použití skrývaného objektu je potřeba vložit volání funkce `CPUIDCommGetName`[1], která provede dotaz na VMM a naplní připravené pole[2] vrácenými znaky. Viz zdrojový kód 6.1.

6.2.3 Vstupní komunikace

Při předávání většího objemu dat do VMM jsme omezeni na jediný 32bitový registr, který můžeme použít. Musíme tedy data posílat postupně a zároveň být schopni určit, kterému objektu patří. Registr rozdělujeme na následující části:

- SEND_MAGIC = 91 — bity 31-24
- ID objektu — bity 23-16
- Pozice posílaného bytu — bity 15-8
- Jeden byte informace — bity 7-0

Viz zdrojový kód 6.2

[1] alternativně `CPUIDCommGetWName` která vrátí jméno ve znacích typu wide-char
[2] předpokládá se maximální možná velikost 16 bytů

```
void CPUIDCommGetName(uint32_t uiName, // Identifikátor objektu
                      char *pszOut,    // Výstupní pole
                      bool bRegenerate)
{
    uint32_t CPUInfo[4];

    if (bRegenerate)
        uiName |= CPUID_COMM_REGENERATE_BIT; // 1 << 31

    asm volatile
    (
        "cpuid" : "=a" (CPUInfo[0]),
                  "=b" (CPUInfo[1]),
                  "=c" (CPUInfo[2]),
                  "=d" (CPUInfo[3])
                : "a" (CPUID_COMM_MAGIC),
                  "c" (uiName)
    );

    // Vlastní implementace strcpy
    _strcpy(pszOut, (char *)CPUInfo);
}
```

Zdrojový kód 6.1: Dotaz na přejmenovaný objekt

```
void CPUIDCommSendDataToVMM(uint8_t uiName, char *szData)
{
    uint8_t i = 0;
    uint32_t send;
    uint32_t CPUInfo[4];

    do
    {
        send = (((CPUID_COMM_SEND_MAGIC & 0xFF) << 24)
              | ((uiName                & 0xFF) << 16)
              | ((i                     & 0xFF) << 8)
              | (szData[i]              & 0xFF));

        // Pouze Windows, můžeme použít intrin.h
        __cpuidex((int *)CPUInfo, CPUID_COMM_MAGIC, send);
    }
    while (szData[i++]); // Pošle i ukončovací nulu
}
```

Zdrojový kód 6.2: Předání dat VMM

Máme tedy možnost předávat i získávat data od hypervisoru a to jak v *user* tak *kernel* módu. Jak uvidíme, budeme této možnosti hojně využívat.

Přítomnost samotného komunikačního kanálu ovšem představuje další možnost, jak odhalit virtuální prostředí. Abychom zamezili odhalení, museli bychom validovat iniciátora CPUID komunikace a odpovídat pouze ověřeným žadatelům. Tento sebeobranný mechanizmus momentálně není implementován. Pokud bychom zvolili cestu digitálních podpisů, jakožto ověřovacího způsobu, museli bychom implementovat validační mechanizmus. To jsme z důvodu odhadované časové náročnosti odložili.

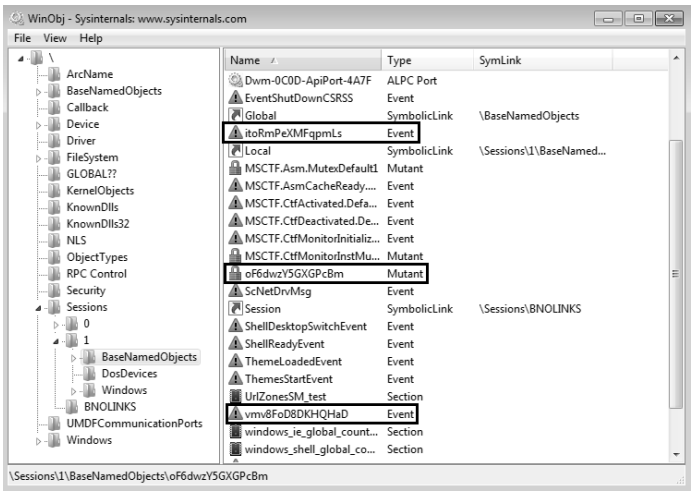
6.3 Pojmenované objekty - dokončení

Pro generování a distribuci jmen objektů použijeme nastíněnou CPUID komunikaci. Každý výskyt pevně definovaných jmen nahradíme polem znaků, které předtím necháme naplnit funkcí `CPUIDCommGetName`.

Díky tomuto řešení budou pojmenované objekty vytvořené *VirtualBox*em mít pokaždé jiné jméno a malware zjišťující přítomnost původních jmen tak neuspěje.

```
VBoxTray: Window name vI92TixmDa7aVCh
Windows version 6.1
VBoxTray: Before calling CreateEvent(DT) Local\itoRmPeXMFqpmLs
VBoxTray: Before calling CreateEvent(WT) Local\vmv8FoD8DKHQHaD
Starting services ...
```

Obrázek 6.2: WinDbg, přejmenované objekty



Obrázek 6.3: WinObj, přejmenované objekty

Tato funkcionalita je zcela implementována. Viz obrázky 6.2 a 6.3. VBoxTray mutex není vidět ve WinDbg, jelikož ve chvíli jeho vytvoření ještě není inicializován *logger*.

6.4 Procesy

Mezi procesy běžícími ve virtualizovaných Windows lze vidět dva s předponou „VBox“ - VBoxTray a VBoxService - oba jsou součástí *Guest Additions*. VBoxService je proces hostující službu a k těm budeme přistupovat rozdílně v podkapitole 6.7. Zaměřme se teď na VBoxTray nebo spíše na procesy obecně.

Malware může použít několik způsobů jak proces patřící *VirtualBoxu* objevit. Jedním z nich je vylistování všech běžících procesů funkcí `EnumProcesses` a vyhledání kompromitujícího jména.

Druhým je nalezení souboru na disku. Oba procesy *VirtualBoxu* mají své binární soubory uloženy mezi systémovými[1]. Malware se tedy může pokusit otevřít soubor např. funkcí `CreateFile`.

Procesy mohou být spouštěny automaticky při startu systému a VBoxTray skutečně spouštěn je. Windows takto spouštějí programy, které jsou uvedené v registrech[37] pod klíčem:

HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Run

Všechny registrové hodnoty mají svůj název a hodnota. Hodnota v tomto klíči je cesta k souboru a název je typicky pouze jméno souboru bez přípony, například:

Název	Hodnota
VBoxTray	C:\Windows\System32\VBoxTray.exe

Malware může například zkusit přečíst hodnotu VBoxTray v uvedeném klíči - funkce `RegQueryValue` nebo vylistovat všechny hodnoty a hledat výskyt „VBox“ jak v názvu, tak v datech - funkce `RegEnumValue`.

6.4.1 Skrývání

Každý proces zná své jméno. V případě konzolové aplikace je toto jméno dostupné v parametrech příkazové řádky. V případě grafické aplikace, jako je VBoxTray, lze použít např. funkci `GetModuleFileName`. Proces tedy při spuštění zjistí své současné jméno a sdělí jej hypervisoru přes CPUID kanál. Hypervisor tedy zná aktuální jméno přejmenovaného procesu a může jej sdělit ostatním komponentám budou-li ho potřebovat.

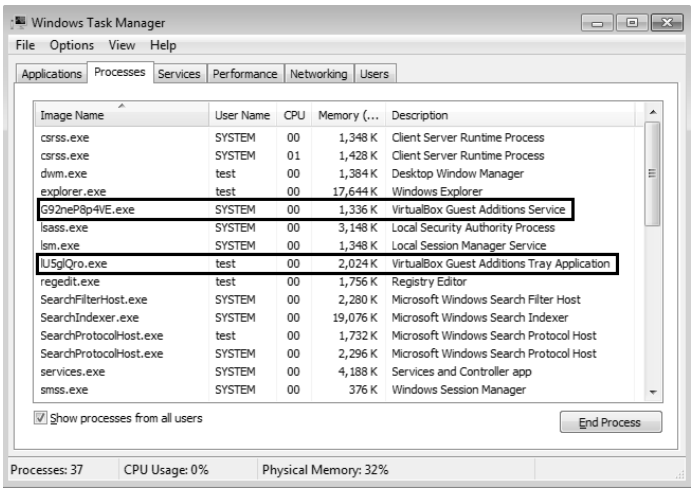
Skrytí kompromitujícího procesu nám velmi usnadňuje fakt, že je spouštěn operačním systémem při startu. Pokud by nebyl, mohli bychom jej mezi

[1] C:\Windows\System32

automaticky spouštěné přidat. Hodnota v uvedeném registrovém klíči může specifikovat parametry, které budou procesu předány a ten upravit, aby při použití specifického parametru pouze sdělil své jméno hypervisoru a ukončil se.

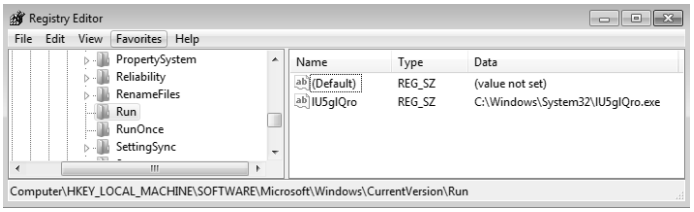
Při prvním spuštění VM budou mít procesy svá původní jména. Název procesu ani jeho binárního souboru nelze měnit pokud je již spuštěn. Přejmenování tedy musí proběhnout po jeho ukončení. Proces, který má být přejmenován se před ukončením zeptá hypervisoru na nové jméno. Pokud je mezi automaticky spouštěnými, přepíše registrovou hodnotu - název i hodnotu - na nové jméno. Nemůže ovšem přejmenovat svůj binární soubor jelikož je stále spuštěn. Windows nabízejí možnost zaregistrovat soubor k přejmenování při dalším spuštění systému[1] a poskytují k tomu funkci `MoveFileEx`, která může být použita s příznakem `MOVEFILE_DELAY_UNTIL_REBOOT`. Pokud by se jednalo o proces, který je spouštěn vícekrát během jednoho spuštění VM, muselo by se zajistit, aby funkce `MoveFileEx` byla zavolána až při jeho posledním spuštění. V podkapitole 6.7 popíšeme jak je toto řešeno. Po restartu tedy Windows soubor samy přejmenují a následně spustí, přičemž je sděleno hypervisoru současné jméno. Malware detekující konkrétní procesy, jejich soubory či uvedený registrový klíč tak nyní selže. Je však potřeba provést alespoň jeden restart virtuálního počítače. Ideálně při přípravě snímku před zahájením analýzy.

Tato funkcionalita je zcela implementována. Viz obrázky 6.4 a 6.5. V ukázce je vidět ponechaný popis souborů, který sloužil pro snazší kontrolu změn. Z finální verze jsou tyto řetězce odstraněny úplně - jejich dynamické generování by nepřineslo žádný užitek.



Obrázek 6.4: TaskMgr, přejmenované procesy

[1] *Pending File Rename Operations*



Obrázek 6.5: Registry, přejmenované procesy

6.5 Ovladače

Ovladač lze sáhodlouze definovat, spokojme se ale s popisem, že se jedná o kus kódu, který je vykonáván v kernel módu a zajišťuje funkcionalitu jako například čtení z disku nebo síťovou komunikaci. Virtualizační nástroje instalují do VM několik ovladačů. V zásadě je ovladač, vedle konzolové a grafické aplikace, další formou spustitelného souboru, liší se jen způsob jeho spuštění, respektive zavedení do systému. Všechny ovladače jsou v systému zaregistrovány pod klíčem:

HKLM\SYSTEM\CurrentControlSet\Services

Podle funkcionality, kterou ovladač poskytuje, pak může být uveden i na jiných místech. Například VBoxSF (*Shared Folders*) je zaregistrován jako síťový poskytovatel pod klíčem:

HKLM\SYSTEM\CurrentControlSet\Control\NetworkProvider

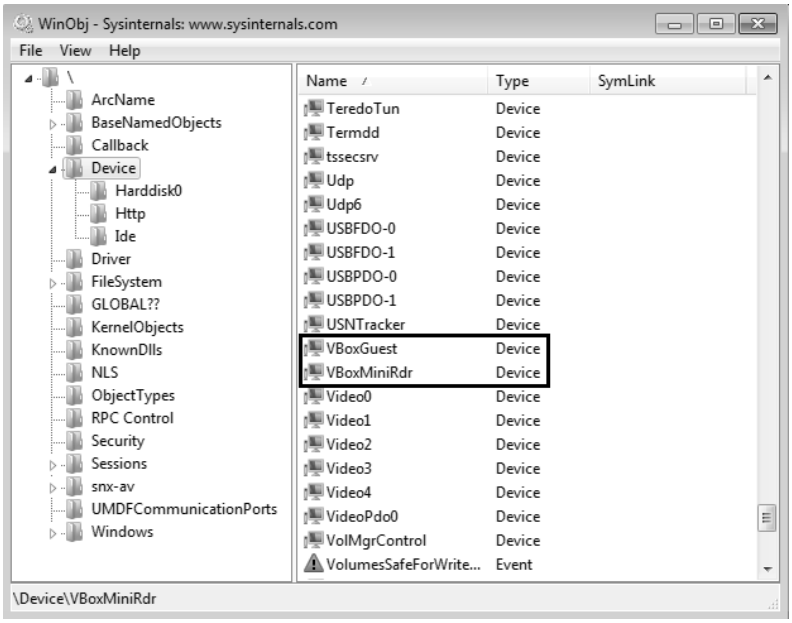
Další způsob, kterým ovladače prozrazují přítomnost, jsou *device* objekty. Ovladač může při zavedení nebo kdykoliv během svého běhu vytvořit více *device* objektů pomocí funkce `IoCreateDevice`. Tímto objektem dáva ovladač okolnímu světu možnost s ním komunikovat. Pomocí nástroje WinObj lze *device* objekty vidět v *Device*. Viz obrázek 6.6

V tomto místě je však dostupný pouze ostatním ovladačům. Aby mohl komunikovat i s „obyčejnými“ programy v user módu, musí ovladač vytvořit link a umístit ho do adresáře \Global?? - funkce `IoCreateSymbolicLink`. Viz obrázek 6.7.

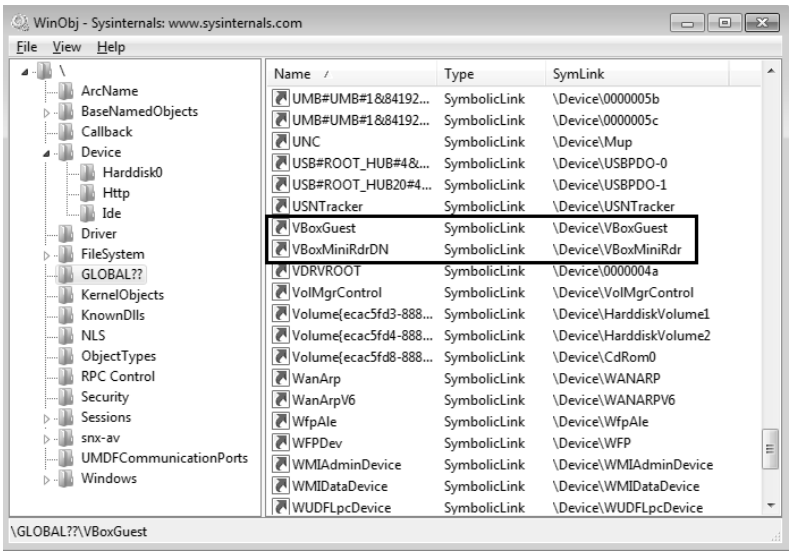
Na obrázku je vidět, že *VBoxMiniRdrDN* je ve skutečnosti symbolický link na *device* objekt *VBoxMiniRdr*. Pokud tedy v *user* módu použijeme například funkci `CreateFile` na „soubor“ *VBoxMiniRdrDN*, zpracuje se toto volání v ovladači, který vytvořil *device* objekt *VBoxMiniRdr*.

Název *device* objekt může vytvářet dojem, že musí být spojený s nějakým fyzickým zařízením. V případě disku tomu tak je, nicméně nemusí. Ovladač pouze vytvoří *device* objekt a oznámí operačnímu systému jím podporované hlavní funkce[1][38], které slouží systému k identifikaci obslužných rutin.

[1]angl. *Major functions*



Obrázek 6.6: WinObj, *Device* objekty



Obrázek 6.7: WinObj, symbolické linky na *Device* objekty

Jestli ve skutečnosti komunikuje s nějakým hardwarem již není podstatné. Například jedna z hlavních funkcí je `IRP_MJ_CREATE`, která je vyvolána z *user* módu funkcí `CreateFile`. Pokud k ní ovladač přiřadil obslužbou rutinu, skončí volání `CreateFile`, nad jím vytvořeným *device* objektem, právě v této rutině.

Ovladač může také nabídnout i specifickou funkcionalitu, kterou nelze vyjádřit pomocí `CreateFile`, `WriteFile` a podobně, a proto existuje hlavní funkce `IRP_MJ_DEVICE_IO_CONTROL`, kterou lze z *user* módu vyvolat funkcí `DeviceIoControl`. Jedním z jejich parametrů je *control code*[1], označovaný zkráceně *IOCTL*, který určuje, jakou specifickou funkcionalitu po ovladači požadujeme. Pokud chce program použít tímto způsobem *device* objekt, musí znát *IOCTL*y, které je jeho ovladač schopen takzvaně dispečovat.

Malwaru se tedy naskytuje několik možností, jak objevit přítomnost konkrétního ovladače. Jak bylo uvedeno, jedná s o kód vykonávaný v *kernel* módu. Díky tomu neběží v samostatném procesu, který by mohl malware detekovat. Vždy však může vyhledat binární soubor ovladače na disku, který bývá uložen mezi systémovými soubory. Stejně jako u služeb lze vyhledat záznam o ovladači v registrech. Navíc jsou zde *device* objekty, které se může malware pokusit otevřít. Další způsob je zneužití *IOCTL*ů. Paranoidní malware si může vylistovat všechny *device* objekty a zkoušet, zda je jejich ovladač schopen odbavit *IOCTL*y, které používá *VirtualBox*.

6.5.1 Skrývání

Princip je podobný skrývání služeb a procesů. Naše snaha je dynamicky generovat názvy, které jsou jinak statické a tudíž detekovatelné. Oproti službám zná ovladač název pod kterým je veden v registrech. Vstupní bod ovladače, typicky pojmenován `DriverEntry`, přijímá jako jeden z parametrů `RegistryPath`, což je právě cesta k jeho záznamu v registrech, kterou může použít k načtení nebo uložení své konfigurace. Při zavedení se přes CPUID kanál oznámí hypervisoru současné jméno ovladače - potažmo jeho služby. Pokud ovladač vytváří nějaké *device* objekty, zeptá se nejdříve hypervisoru jak objekt pojmenovat, stejně jako v případě pojmenovaných objektů. Co se týče *IOCTL*ů, tak v místech, kde jsou používány, tj. vně ovladače, se nejprve přičte náhodný ofset vygenerovaný hypervisorem. Uvnitř ovladače, v odbavovací metodě proběhne dotaz na tento ofset a ten je následně odečten od *IOCTLu*, který zrovna dorazil. Ovladač odmítne *IOCTL* odbavit, pokud by malware zkusil použít původní hodnotu, jelikož po odečtení ofsetu získá neznámý *IOCTL*.

Při vypínání systému chceme přejmenovat binární soubor ovladače a název služby na všech místech v registrech. Ovladač vypínání systému pozná podle toho, že je vyvolána hlavní funkce `IRP_MJ_SHUTDOWN`. V tu chvíli nechá vygenerovat nové jméno, kterým přepíše své současné v registrech. Přejmenování binárního souboru nelze provést odloženě jako v případě služeb a procesů.

[1] 32bitové číslo

```
VBoxGuest-win: vgdrvNtAddDevice: \DosDevices\5yF0TaLCpQNud0f
vgdrvHeartbeatInit: Setting up heartbeat to trigger every 2000 milliseconds
vgdrvNtInit: Device is ready!
RegisterDriverForMorph: Current name of 10: VBoxSF
VBOXSF: DriverEntry: Before Calling RxRegisterMinirdr \Device\qjq1ujkpE1gZIhs
VBOXSF: DriverEntry: Before Calling IoCreateSymbolicLink \??\xYr0365iJW1S7Kx
```

Obrázek 6.8: WinDbg, VBoxSF před prvním restartem (ID VBoxSF je 10)

```
VBoxGuest-win: vgdrvNtAddDevice: \DosDevices\5STSp4BqpCt8NgN
vgdrvHeartbeatInit: Setting up heartbeat to trigger every 2000 milliseconds
vgdrvNtInit: Device is ready!
RegisterDriverForMorph: Current name of 10: zsjGRK
VBOXSF: DriverEntry: Before Calling RxRegisterMinirdr \Device\qVyDEXQ5Y54VaF8
VBOXSF: DriverEntry: Before Calling IoCreateSymbolicLink \??\jiHWpgLOJofY1md
```

Obrázek 6.9: WinDbg, VBoxSF po restartu

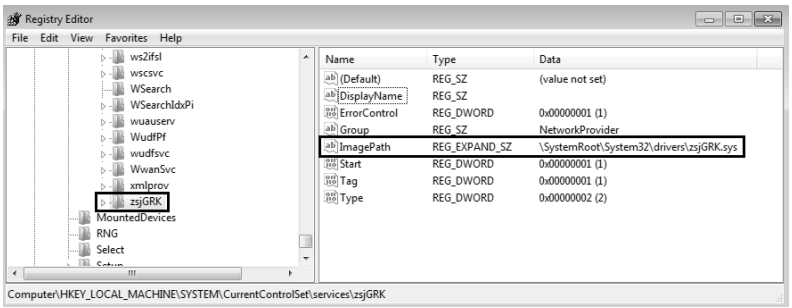
```
VBOXNP: DLL loaded, IOCTL Offset: 12963
VBOXNP: vbsfIOCTL: Modifying IoctlCode: 1311144 + 12963 = 1324107
VBOXSF: MRxDevFcbXXXControlFile: Modifying IoctlCode: 1324107 - 12963 = 1311144
VBOXSF: MRxDevFcbXXXControlFile: Modifying IoctlCode: 1324107 - 12963 = 1311144
VBOXSF: IOCTL: Before Calling InitializeObjectAttributes \??\LoBAPAngZIjONS1
```

Obrázek 6.10: WinDbg, *IOCTL* ofset, NP-*user* mód, SF-*kernel* mód

Ovladače mohou být zaváděny ve velmi rané fázi startu OS, ještě před provedením odložených přejmenování. V takovém případě by se nepovedlo ovladač zavést, protože v registrech by již byla uvedena cesta k novému jménu avšak soubor samotný by měl stále staré jméno. Ovladače jsou naštěstí kompletně načteny v paměti a jejich binární soubor tak není chráněn proti přepsání. Díky tomu může sám ovladač přejmenovat svůj binární soubor. V podkapitole 6.7 však ukážeme, že jsme zvolili jinou metodu pro přejmenování ovladačů.

Stejně jako u procesů jsme docílili úplného zamaskovaní přítomnosti kompromitujících ovladačů i jimi vytvářených objektů. Jediný předpoklad je alespoň jeden restart systému.

Funkcionalita je zcela implementována pouze pro jeden ovladač - VBoxSF, čímž jsme ověřili funkčnost našeho řešení. Viz obrázky 6.8, 6.9, 6.10 a 6.11



Obrázek 6.11: Registry, VBoxSF po restartu

6.6 Dynamicky linkované knihovny

Mezi spustitelným souborem a DLL[1] je z hlediska systému velmi nepatrný rozdíl - oba soubory jsou ve formátu PE[2]. Účel DLL je poskytovat kód, který je společný pro více programů a nemá tak cenu, aby jeho kopie byla v každém z nich. Pokud program potřebuje funkci poskytovanou nějakou knihovnou, může si ji za běhu načíst funkcí `LoadLibrary` a hledanou funkci pak získat pomocí `GetProcAddress`.

Jeden z rozdílů mezi EXE a DLL je, že DLL nejde spustit sama o sobě. Windows nástroj *rundll.exe* sice nabízí možnost „spustit“ DLL, neudělá ale prakticky nic jiného, než že knihovnu načte, což vyvolá funkci `DllMain` pokud je v knihovně implementována.

Virtualizační nástroje mohou do VM nahrát několik knihoven pro podporu své činnosti. Tím se vytváří další vektor pro detekci virtuálního prostředí. Nejjednodušší možnost je opět objevit samotné soubory, které bývají v systémové složce. Dále mohou být knihovny vedeny na různých místech v registrech v závislosti na funkcionalitě kterou poskytují, podobně jako ovladače. Na rozdíl od ovladačů neexistuje v registrech žádné centrální místo, kde by byly knihovny vedeny.

6.6.1 Skrývání

U knihoven bohužel nejde aplikovat metodu, kterou jsme používali doteď. Zatím jsme se spoléhali na Windows, které spouštěly skrývané prvky při startu a ty samy oznámily své aktuální jméno a při ukončování se samy přejmenovaly.

Knihovna nemá jak hypervisoru oznámit své jméno - `DllMain` je vykonána až při načtení, pro které je nutné jméno již znát. Problém nastává i v přejmenování při vypínání systému. Během uvolňování z paměti[3] je opět zavolána `DllMain`, jen s jinými parametry. Knihovna tedy pozná, že je uvolňována, jen nedokáže rozhodnout zda z důvodu vypínání systému. Problém taky vyvolává fakt, že `DllMain` je vykonána při uvolňování z každého procesu a pokud by se knihovna zaregistrovala k odloženému přejmenování pokaždé, nastala by kolize, protože soubor by byl přejmenován na první zaregistrované nové jméno[4], zatímco hypervisoru by bylo oznámeno jméno z posledního uvolnění knihovny.

Přejmenování tedy za knihovnu musí udělat někdo jiný a zároveň potřebujeme zajistit perzistentní uložení současných jmen. V podkapitole 6.7 popíšeme jak docílit úspěšného přejmenování DLL. Předpokládejme prozatím, že hypervisor zná aktuální jména knihoven. *VirtualBox*, a pravděpodobně i ostatní nástroje, uchovávají nastavení jednotlivých VM v konfiguračních souborech a zároveň nabízí možnost jak do souboru zapsat. Pro *VirtualBox* to jsou XML

[1] angl. *Dynamic-link library*
[2] angl. *Portable Executable*
[3] angl. *Unload*
[4] `MoveFileEx` nepřepisuje záznamy o přejmenování stejných souborů

soubory .vbox - právě jeden pro jednu VM. Hypervisor po obržení nového jména jej uložit do tohoto souboru a při spuštění VM si je odsud načte.

Přejmenování knihoven vyžaduje modifikaci mnoho částí kódu. Tato funkcionalita momentálně není implementována z důvodů její odhadované časové náročnosti.

6.7 Služby

Služba[1] je pojem specifický pro Windows, i když podobné koncepty jsou i v jiných operačních systémech, například *Daemon* ve světě Linuxu. Windows dále rozlišují pojem *service application*, popisovaný v této podkapitole a *driver service*, popsaný v podkapitole 6.5. Dle definice je služba proces, který je spuštěn *Service Control Manager*em - SCM[39] a splňuje definované rozhraní. SCM oproti tomu je proces spuštěný při startu systému - *services.exe*. Pro naše účely je důležitý poznatek, že SCM spravuje seznam nainstalovaných služeb. Tento seznam je uložen v registrech a pro jeho lépe čitelné zobrazení či modifikaci lze použit nástroj *Services.msc*. Na *VirtualBoxu* můžeme vidět, že jedna ze služeb se jmenuje VBoxService. Jedná se tak o další místo, které může malware zneužít. Seznam služeb lze nalézt v registrovém klíči *Services*. Viz podkapitolu 6.5.

Pro malware zde exituje několik míst, která může využít. Za prvé je to samotný podklíč v registrech, který reprezentuje název služby. Malware může použít např. funkci `RegOpenKeyEx` a zkusit otevřít kompromitující záznam nebo `OpenService` a pokusit se získat handle k samotné službě.

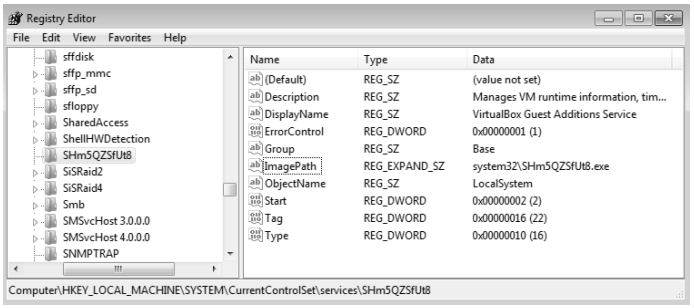
Jedna z registrových hodnot každé služby je `ImagePath` - cesta k binárnímu souboru. Pokud by byl malware o něco opatrnější a nespoléhal se na konkrétní jméno služby, může si vylistovat všechny nainstalované služby, tzn. podklíče Services, pomocí funkce `RegEnumKey` a pro každý z nich zkontrolovat, zda se v `ImagePath` nevyskytuje kompromitující řetězec.

Stejně jako `ImagePath` může malware využít i další registrové hodnoty, které mohou obsahovat závadné řetězce a to `DisplayName` a `Description`.

6.7.1 Skrývání

Nejprve musíme o něco detailněji popsat jak jsou vlastně služby v OS Windows implementovány. Bylo uvedeno, že služba musí splňovat určité rozhraní. To mimo jiné zahrnuje zavolání funkce `StartServiceCtrlDispatcher` do 30 sekund od spuštění procesu. Ta způsobí, že se vlákno procesu, které tuto funkci zavolalo, připojí k SCM. Služba by poté měla zaregistrovat takzvaný *control handler* - funkce `RegisterServiceCtrlHandler`, do kterého bude SCM posílat řídící kódy, na které by služba měla adekvátně reagovat.

[1] angl. *Service*



Obrázek 6.12: Registry, VBoxService po restartu

Další pro nás zajímavý detail je, že při instalaci služby lze zvolit, zda bude spouštěna při startu systému a dále pak zda bude spuštěna ve vlastním nebo sdíleném procesu. Sdílený proces se tradičně jmenuje *svchost.exe*. Pokud ovšem služba zvolí vlastní proces, jako je tomu v případě VBoxService, je malwaru vytvořena další příležitost pro detekci virtuálního prostředí. Viz podkapitolu 6.4.

Námi zvolený způsob skrývání probíhá následovně. Služba při startu sdělí hypervisoru svůj aktuální název stejně jako skrývaný proces. Zde může nastat problém, jelikož hypervisoru je ve skutečnosti sdělen název souboru a ten nemusí být shodný s názvem služby. Proces bohužel nemá jednoduchou možnost jak zjistit název služby kterou reprezentuje. V případě VBoxService je naštěstí název stejný jako jejího binárního souboru - *VBoxService.exe*.

Služby se na rozdíl od běžných procesů dozví, že nastává vypínání počítače, když obdrží řídící kód `SERVICE_CONTROL_SHUTDOWN`. V tu chvíli proběhne podobná procedura jako při ukončování skrývaného procesu. Vygeneruje se nové jméno a registruje se přejmenování binárního souboru. Přejmenuje se registrový klíč s konfigurací. Efektivně to znamená, že dojde k přejmenování služby a přepíší se `ImagePath`, `DisplayName` a `Description`.

Tato funkcionalita je zcela implementována. Opět jsme nechali v ukázce 6.12 popis pro snazší identifikování změn.

6.8 Centralizované skrývání aktivních objektů

Nyní se dostáváme k tolik avizované implementaci skrývání procesů, služeb, ovladačů a knihoven, které popisujeme souhnně jako aktivní objekty. Popsali jsme, jak by mohly jednotlivé prvky být samy zodpovědné za své skrývání. Bylo ovšem uvedeno několik nedostatků, kterým se níže popsaným způsobem vyhneme. Princip skrývání se nemění, jen je přesunut na jedno centralizované místo a tím je služba VBoxService.

Pokaždé když aktivní objekt oznámí hypervisoru své jméno, znamená to, že se tím „registruje“ k přejmenování. VBoxService při vypínání počítače

zavolá funkci MorphObjects. Ta se postupně zeptá hypervisoru na současná jména objektů pomocí jejich ID. Pokud jméno není prázdný řetězec, objekt se registroval k přejmenování. ID objektu také slouží jako index do pole dvojic funkcí, které provedou samotné změny. Viz zdrojový kód 6.3. První funkce je shodná pro všechny objekty stejného typu, například ovladač, a mění atributy, které objekty sdílí - registrové klíče, soubory, atd. Druhá funkce je specifická pro každý objekt a slouží k přejmenování atributů, které jsou pro daný objekt jedinečné.

```
static struct
{
    DWORD (*MorphRoutine)
          (LPWSTR pwszCurrentName, LPWSTR pwszNewName);
    DWORD (*MorphHelperRoutine)
          (LPWSTR pwszCurrentName, LPWSTR pwszNewName);
} g_CPUIDCommMorphObjectsStruct[] =
{
    ...
    { MorphDriver, VBoxSFMorphHelper }, // VBoxSF ID = 10
    ...
}
```

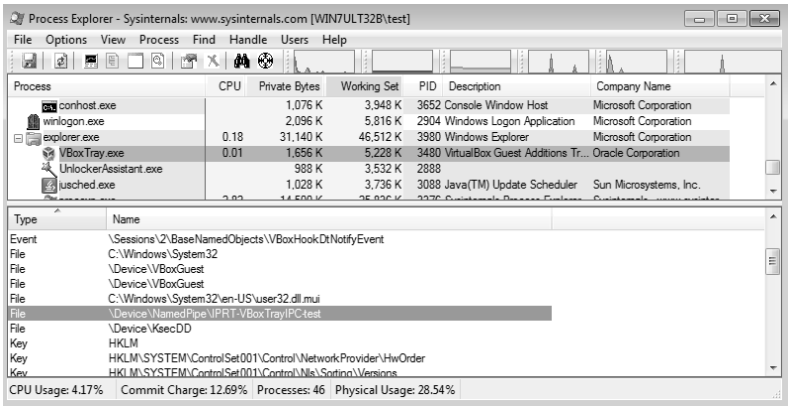
Zdrojový kód 6.3: Struktura přejmenovacích rutin

6.9 Pojmenované roury

V komplexním systému jako je virtualizační nástroj je potřeba sdílet data mezi jednotlivými komponenty. Jedním ze způsobů jak implementovat meziprocesovou komunikaci[1] jsou roury. Z pohledu operačního systému je pojmenovaná roura část paměti, do které mohou různé procesy zapisovat i z ní číst a lze k ní získat handle právě podle jména. Proces, který rouru vytvořil, funkce `CreateNamedPipe`, se označuje server a proces, který se k rouře připojuje se označuje klient. Server má možnost vytvořit rouru pro obousměrnou (duplexní) komunikaci. Klient může s rourou zacházet jako s běžným souborem - získat handle pomocí `CreateFile`, číst a zapisovat funkcemi `ReadFile`, `WriteFile` nebo `TransactNamedPipe` (čtení a zapis v jedné funkci). Případně může použít `CallNamedPipe`, což je pouze obálka nad `CreateFile`, `TransactNamedPipe` a `CloseHandle`. Další možnosti meziprocesové komunikace jsou například sockety, RPC[2] a file mapping[40].

VirtualBox využívá pojmenované roury pro sdílení informací mezi uživateli OS a VBoxService. Služby jsou totiž spouštěny v speciální session 0, která nepatří žádnému uživateli. V systému tak běží vždy jen jedna instance služby, i když je přihlášeno více uživatelů. Služba na druhou stranu nemá přístup

[1] angl. *Inter-process communication* - IPC
[2] *Remote Procedure Call*



Obrázek 6.13: ProcExp, původní pojmenovaná roura

k informacím o jednotlivých uživatelích, a tak zde vystupuje VBoxTray jako IPC server běžící v každé uživatelské session a VBoxService jako klient, který takto získá přístup k uživatelským session a může například uživateli oznámit, že je dostupná nová verze *Guest Additions*.

6.9.1 Skrývání

Roury jsme neuvedli v podkapitole 6.1 jelikož k nim systém přistupuje odlišně. Pro jejich zobrazení nelze použít WinObj, nýbrž ProcExp. Viz obrázek 6.13, kde je vidět, název roury ve tvaru:

IPRT-VBoxTRayIPC-<jméno uživatele>

Ke skrývání pojmenované roury budeme přistupovat stejně jako k pojmenovaným objektů. Roura má sice rozdílný název pro každého uživatele, nicméně prefix *IPRT-VBoxTrayIPC* je stále stejný. Vytvoření prefixu však nemůžeme nechat v místě vytvoření roury, jelikož by každé přihlášení uživatele prefix přegenerovalo. Jak jsme popsali, služby jsou systémem spouštěny pouze jednou a před přihlášením prvního uživatele necháme tedy vygenerování jména na nějaké službě - VBoxService. V místě vytváření roury pak jen proběhne dotaz na aktuální pojmenování.

Jelikož název roury není nikde v systému zaznamenán[1], nemá malware jinou možnost jak ji objevit, než se pokusit o připojení[2]. Název sice obsahuje jméno uživatele, která malware zná, nicméně už nemá možnost jak zjistit prefix a jeho pokus o připojení selže. Viz obrázky 6.14 a 6.15.

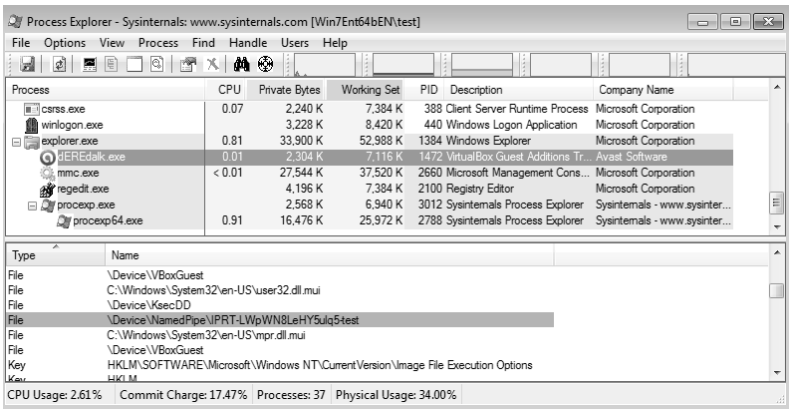
Tato funkcionalita je zcela implementována.

[1] proto jej také nelze zobrazit WinObj

[2] případně lze vylistovat všechny *handly* všech spuštěných procesů a pomocí `NtQueryObject` zjistit typ *handlů*

```
Starting service 'IPC' ...
VBoxTray: Pipe name LWpWN8LeHY5ulq5-
VBoxIPCInit: Local IPC server now running at "LWpWN8LeHY5ulq5-test"
Service 'IPC' started
```

Obrázek 6.14: WinDbg, přejmenovaná roura



Obrázek 6.15: ProcExp, přejmenovaná roura

6.10 Okna

Pojmem okno označujeme veškeré grafické prvky zobrazené na displeji počítače. Při vytváření okna pomocí funkce `CreateWindow` je možné udat jméno a třídu. Ta může být jedna z předdefinovaných, například BUTTON nebo SCROLLBAR, nebo lze použít vlastní třídu, která byla registrována funkcí `RegisterClass`. *VirtualBox* takto registruje třídu *VBoxTrayToolWndClass* a vytváří okno se jménem *VBoxTrayToolWnd*.

Pokud virtualizační nástroj vytváří okna se specifickým jménem nebo třídou, vzniká tak malwaru další příležitost jak rozhodnout o nechtěném prostředí. Může totiž prostým použitím funkce `FindWindow` zjistit přítomnost konkrétního okna.

6.10.1 Skrývání

Pro schování názvu oken použijeme již ověřený mechanizmus dynamického generování a distribuce přes CPUID kanál.

Tato funkcionalita je zcela implementována.

```
VBOXNP: DLL loaded.
VBoxTray: Window name 2NqBSlmFe5UxYnm
Windows version 6.1
VBoxTray: Before calling CreateEvent(DT) Local\vepAReLWD27yhWB
VBoxTray: Before calling CreateEvent(WT) Local\Uh2bahiHGZUZ4nm
Starting services ...
Starting service 'display' ...
Service 'display' started
```

Obrázek 6.16: WinDbg, přejmenované okno

6.11 Debug výstup

Poslední metoda, kterou v krátkosti zmíníme je sledování *debug*[1] hlášek. Používali jsme WinDbg a DebugView pro kontrolu našich změn a pro snazší nalezení hlášek jsme je označovali „VBox*:“. Viz například obrázek 6.16.

Několik hlášek bylo přidáno při implementaci výše uvedených mechanizmů, nicméně sám *VirtualBox* jich také pár vypisuje. Malware může teoreticky číst *buffer* ve kterém jsou schraňovány tyto hlášky a hledat závadné řetězce. Pokud bychom chtěli být důslední, museli bychom ze všech zpráv smazat prefix „VBox“ nebo ještě lépe po odladění veškerých změn hlášky odstranit úplně. Funkcionalita není implementována z důvodů snazší kontroly ostatních změn.

[1] česky Ladících

Kapitola 7

Hardware

Zaměřme se nyní na části virtuálních strojů, které se nachází „pod“ operačním systémem. Virtuální OS nesmí mít přístup k fyzickým zařízením jako je disk nebo síťová karta. Musí však být přesvědčen o tom, že zařízení dostupná jsou a musí být s nimi schopen komunikovat způsobem, kterým je zvyklý. To znamená, že virtualizační nástroje nesou na bedrech nutnost virtualizovat všechny komponenty počítače - od BIOSu po sběrnice a periferní zařízení jako síťová karta. Dá se téměř s jistou říci, že v každé komponentě existují místa, která prozrazují, že se vskutku jedná o virtuální počítač. Opět se nevyhneme popisu některých principů než se budeme schopni vrhnout na formy jejich zneužití a obrany.

7.1 Hardware ID

Než se opět ponoříme do útrob *VirtualBoxu*, musíme nejdříve popsat postup navázání komunikace mezi OS a hardwarem. Takzvaná periferní zařízení jsou k základní desce připojena přes PCI[1] sběrnici. Může jít o koncová zařízení jako grafická karta nebo o řadiče, ke kterým lze připojit další zařízení. Jedná se tedy o hierarchickou strukturu. Například USB paměti se připojují k USB řadiči, ne přímo na PCI sběrnici.

Všechna zařízení, byť připojená k řadiči, obsahují několik standardizovaných registrů, které poskytují operačnímu systému cenné informace. Nás zajímá hlavně *Vendor* ID - identifikátor výrobce a *Device* ID[2] - identifikátor zařízení.

Při startu systému probíhá takzvaná enumerace zařízení. Operační systém se každé PCI patice „zeptá“, jestli je v ní něco zapojeno. Pokud ano, zařízení odpoví svým *Vendor* a *Device* ID. OS se tak dozví o jaké konkrétní zařízení se jedná a pokud má ovladač, který při své instalaci oznámil, že dokáže zařízení

[1] *Peripheral Component Interconnect*
[2] někdy *Product* ID

obsloužit, tak jej načte. Pokud ovladač není zrovna k dispozici, je uživateli oznámeno, že bylo připojeno neznámé zařízení a ovladač se pokusí stáhnout. V ideálním případě tedy stačí zařízení pouze připojit a OS se s ním sám skamarádí. Jedná se o standardní způsob, který se nazývá PnP - *Plug and Play*. Samotná výměna dat již pro nás není tak zajímavá.

Vendor a *Device* ID lze souhrně označit jako Hardware ID. Neoficiální seznam hardware ID PCI zařízení a společností, kterým patří lze nalézt v [41]. Stejný seznam pro USB zařízení lze nalézt v [42].

Virtualizační nástroje používají svá vlastní *Vendor* ID. Musí. Pokud by používaly nějaké známé hrozí, že operační systém načte originální ovladače a ten nebude schopen se zařízením správně komunikovat.

Malware může provést stejnou enumeraci zařízení jako OS při startu. Windows k tomu nabízí rozhraní `IPortableDeviceManager` a metodu `GetDevices`, která vytvoří seznam *Device* ID připojených zařízený. Další metody jako například `GetDeviceManufacturer` nebo `GetDeviceFriendlyName` slouží k získání dodatečných informací. Malware má tedy stejný přístup k infomacím o zařízeních jako OS a může se tak ujistit zda se nenachází na podezřelém místě. *VirtualBox* používá:

- *Vendor* ID: 0x80EE
- *Device* ID: 0xBEEF a 0xCAFE

7.1.1 Skrývání

Ačkoliv se nacházíme pod operačním systémem, stále je pod virtuálním hardwarem hypervisor. Z implementačního hlediska nám tak nic nebrání generovat hardware ID dynamicky a ovladače, které na ID spoléhají upravit tak, aby se nejdříve dotázali na současné označení. Jenže jak bylo uvedeno, hrozí, že bychom vygenerovali již obsazené označení a mohly by nastat problémy. Další problém by byl, jak tuto změnu správně oznámit operačnímu systému. Windows při instalaci ovladače postupují podle takzvaných INF souborů[43], které musí být dodány společně s ovladačem. Jedná se o konfigurační soubory, které mimo jiné obsahují hardware ID zařízení pro která jsou určena, například:

- PCI\VEN_80EE&DEV_BEEF

INF soubory jsou používány pouze při instalaci, kde všude jsou poté informace jako hardware ID uchovávány je interní záležitost Windows, která není nikde popsána. Pokud uvedený identifikátor necháme vyhledat v registrech zjistíme, že je na mnoho místech a přepisovat všechny dynamicky se zdá neschůdné.

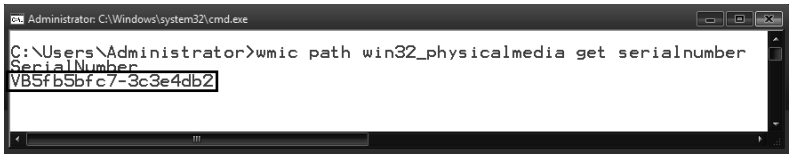
Přístup, který jsme zvolili je prosté přepsání hodnot 0x80EE, 0xBEEF a 0xCAFE na něco jiného a dostupného. Věříme, že problémy, které si tím ušetříme vyváží možnou omezenou efektivitu řešení.

Tato funkcionalita není implementována opět z důvodů její odhadované časové náročnosti.

7.2 DMI

Desktop Management Interface (DMI) je standard pro správu komponent počítače. Software tak může například zjistit verzi BIOSu, výrobce procesoru nebo základní desky. Windows pro tyto účely poskytují konzolové programy `wmic` nebo `systeminfo`. Viz obrázky 7.1 a 7.2.

Obrázek 7.1: SystemInfo, původní DMI řetězce



Obrázek 7.2: WMIC, původní sériové čísko disku

7.2.1 Skrývání

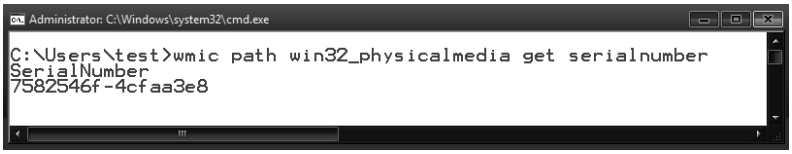
Využijeme výstup programu `systeminfo` pro identifikaci problémových míst. Z implementačního hlediska nám nic nebrání generovat tyto problémové názvy dynamicky, nicméně o přínosu této metody by se dalo s úspěchem pochybovat. Řekněme, že paranoidní malware může zvolit defensivní přístup a místo kompromitujících názvů jako *VirtualBox* bude hledat známá jména jako *Dell* a podobně.

Zvolíme tedy přístup, kdy necháme naše komponenty vystupovat pod jmény fyzicky existujících. Z hlediska implementace to pouze znamená najít a přepsat ve zdrojovém kódu všechny řetězce, které nám zobrazuje `systeminfo`.

Další možný způsob, který nám *VirtualBox* přímo nabízí, je změnění DMI dat přímo v konfiguraci jednotlivých virtuálních strojů. *VirtualBox* umožňuje nastavovat DMI data v konfiguračním souboru příslušné VM (.vbox soubor). Viz podkapitolu 6.6. Pokud tento soubor obsahuje takzvaná DMI extra data, použije je. Pokud ne, použije výchozí hodnoty, které vidíme na obrázcích 7.3 a 7.4.

Tato funkcionalita je zcela implementována.

Obrázek 7.3: SystemInfo, změněné řetězce



Obrázek 7.4: WMIC, změněné sériové číslo disku

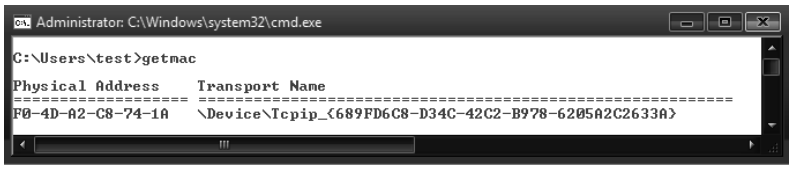
7.3 MAC adresa

Každá síťová karta má svůj identifikátor - MAC[1] adresu. Jedná se o 48bitové číslo, které musí být unikátní v rámci počítačové sítě pro funkční posílání paketů. Prvních 24 bitů je označováno OUI[2] a výrobci si je musí registrovat, pokud chtějí své karty uvést na trh. Zavazují se při tom, že všechny jimi vyrobené karty budou mít tento registrovaný OUI a že každá karta bude mít zbylých 24 bitů unikátních.

[1] Media Access Control

[2] Organizationally Unique Identifier

Obrázek 7.5: Původní MAC adresa



Obrázek 7.6: Změněná MAC adresa

VirtualBox, a pravděpodobně i ostatní nástroje, mají registrovaný OUI, který malwaru dává najevo, kde se nachází. Seznam OUI lze nálézt v [44]. *VirtualBox* používá 08-00-27, který je registrovaný na společnost PCS Systemtechnik GmbH. Nepodařilo se nám zjistit jakou souvislost má Oracle s touto společností, nicméně i ve zdrojových kódech je hodnota 08-00-27 pevně daná.

7.3.1 Skrývání

Zde máme zjednodušenou práci díky tomu, že virtuální síťová karta vytvářená *VirtualBox*em se tváří jako běžně dostupná fyzická karta od Intelu (*Vendor* ID 8086). Díky tomu nejsou v OS jiné stopy po virtuální síťové kartě kromě MAC adresy. Ve zdrojovém kódu tak pouze změníme OUI na ten registrovaný na Intel - F0-4D-A2.

Kapitola 8

Virtualizace

Nyní se oprostíme od projevů konkrétního virtualizačního nástroje a budeme se soustředit na samotnou podstatu virtuálního prostředí. V této práci jsme se zaměřili na hardwarovou podporu virtualizace jako na prostředek vytvoření virtuálního prostředí. Takto vytvořený svět není naprosto identickou kopií skutečného. Je sice natolik důvěryhodný, že dokáže přesvědčit operační systém, který do něj vložíme, stále však existují způsoby jak přetvářku prohlédnout.

Metody detekce, které představíme jsou pevně svázány s podstatou našeho virtuálního světa - podporou procesoru. Ta se neustále vyvíjí a stále více přibližuje virtuální prostředí skutečnému. Díky tomu je možné, že některé představené metody rozpoznání virtualizace již nebudou v dohledné době funkční, což je pro nás samozřejmě dobře.

8.1 Hypervisor present bit

Po vstupu do *root* módu (vzpomeňme na instrukci VMXON) je v procesoru nastaven bit oznamující přítomnost hypervisoru. Hodnotu lze získat instrukcí CPUID s parametry EAX=1 a ECX=0, výsledek bude umístěn na nejvyšším bitu registru ECX. Viz zdrojový kód 8.1:

```
BOOL HypervisorPresentBit(VOID)      // user mód
{
    INT32 CPUInfo[4] = { 0, 0, 0, 0 };
    __cpuid(CPUInfo, 1);

    return CPUInfo[2] >> 31;
}
```

Zdrojový kód 8.1: Zjištění přítomnosti hypervisoru

8.1.1 Skrývání

Díky zaručenému VM *Exit* máme pevně pod kontrolou návratové hodnoty. Na tom je ostatně založena celá naše CPUID komunikace. Do části starající se o emulaci CPUID stačí vložit kontrolu na parametry 1 a 0 a před VM *Entry* vynulovat poslední bit registru ECX.

Tato funkcionalita je zcela implementována.

8.2 Hypervisor vendor

U instrukce CPUID ještě zůstaneme. Hodnoty 0x40000000 až 0x4FFFFFFF se Intel v technické specifikaci[1] zavazuje neimplementovat. Tím jsou nepřímo nabídnuty pro softwarové použití. Virtualizační nástroje přijmuly hodnotu 0x40000000 jako výzvu k identifikaci a vrací pro ně specifický řetězec v registrech EBX, EDX a ECX (v tomto pořadí). Například *VirtualBox* se legitimuje jako „VBoxVBoxVBox“.

Ukázka jak získat tento řetězec:

```
VOID HypervisorType(PCSTR szHvProductName)      // user mód
{
    INT32 CPUInfo[4] = { 0, 0, 0, 0 };
    __cpuid(CPUInfo, 0x40000000);

    SecureZeroMemory((PVOID)szHvProductName, 13);

    ((INT32*)szHvProductName)[0] = CPUInfo[1];
    ((INT32*)szHvProductName)[1] = CPUInfo[3];
    ((INT32*)szHvProductName)[2] = CPUInfo[2];
}
```

Zdrojový kód 8.2: Zjištění výrobce hypervisoru

8.2.1 Skrývání

VirtualBox tuto hodnotu nastavuje stejně jako DMI data, buď se použije ta co je specifikována v konfiguračním souboru, nebo výchozí „VBoxVBoxVBox“. Pokud uvedenou funkci spustíme na fyzickém procesoru je nám vrácena nespecifikovaná hodnota. Pokusně jsme ověřili, že to jsou hodnoty 1 0 1[2]. Naše řešení je změnit výchozí hodnotu právě na 1 0 1. Ve zdrojovém kódu není vidět, že něco na této hodnotě záviselo.

Tato funkcionalita je zcela implementována.

[1] kapitola *Information Returned by CPUID Instruction*

[2] v jednotlivých 32bitových registrech

8.3 VMX instrukce

VMX instrukce rozdělíme na dvě části - ty které jsou určeny pro spuštění VMX operace správu VMCS a mohou být vykonány pouze v *root* módu a na ty které jsou určeny pro software v *non-root* módu.

8.3.1 Root mód

Prakticky neexistuje překážka, která by zamezovala ve virtuálním stroji vytvořit další. Hardwarová podpora obsahuje prvky, které usnadňují implementaci vnořené virtualizace. *VirtualBox* ji však v současné verzi neumožňuje a při pokusu o spuštění VMX operace nastane výjimka typu #UD, stejně tak jako všechny ostatní instrukce pro správu VMCS. Zároveň je *VirtualBox*em zamlčena samotná podpora VMX. Viz 3.1.

Jelikož je podpora VMX zcela běžná, může se malwaru zdát podezřelé, když nebude podporována.

8.3.1.1 Skrývání

První krok je prozradit dostupnost VMX. Tím zmateme malware, který dělá pouze tuto snadnou kontrolu, která je možná z *user* módu. Pokud by chtěl být důslednější a skutečně se pokusil VMX zapnout, musel by běžet v kernel módu. Mohli bychom teoreticky nastavit všechny VM *Exit*y způsobené VMX instrukcemi, aby neinjektovaly #UD výjimku a tím přelstít malware spoléhající na vyvolání #UD, zároveň bychom mohli uvést VM do nedefinovaného stavu a způsobit neočekávané chování.

Správné řešení by bylo umožnit vnořenou virtualizaci. To je ovšem výprava na kterou se v této práci nechystáme vydávat pro její odhadovanou náročnost a relativně nesouvisející cíl.

8.3.2 Non-root mód

Existují dvě instrukce určené pro spuštění v *non-root* módu, VMCALL a VMFUNC, kterými si může software vyžádat specifickou obsluhu. První jsme již zmiňovali, druhou popíšeme nyní. VMFUNC má jeden operand, který určuje, jaká funkce se má vykonat v *root* módu. Těchto funkcí je 64 a lze je jednotlivě povolit či zakázat nastavením ve VMCS. Viz kapitolu *VM Functions* technické specifikace. VMFUNC nemusí vždy vyvolat VM *Exit*, pouze pokud je zavolána na nepovolenou funkci. V případě, že operand je větší než 63 nastane #UD výjimka.

Mimo VMX operaci tyto instrukce způsobí #UD výjimku a malware může zneužít rozdílného chování vně VM prostým zavoláním VMCALL nebo vyvoláním všech VM funkcí. Pokud nenastane #UD je jasné, že se nachází ve VM.

8.3.2.1 Skrývání

Řešení nám může zkomplikovat sám virtualizační nástroj pokud je zavislý na funčnosti VMCALL nebo VMFUNC. Odstranění této závislosti by mohl být velmi komplikovaný, ne-li nemožný proces. V případě *VirtualBoxu* nejsou VM funkce naštěstí používány vůbec a VMCALL je používán pouze pokud je umožněna paravirtualizace. Paravirtualizace znamená, že VM bude schopna využít služeb hypervisoru. Lze ji nastavit přímo ve *VirtualBoxu*, nicméně, aby ji mohl software ve VM využít musí se o ní dozvědět a jedním z mechanismů je Hypervisor present bit[45].

Řešením je tedy zakázat všechny VM funkce a při příslušných VM *Exit*ech nainjektovat #UD výjimku.

Tato funckionalita je zcela implementována.

8.4 Časování

Poslední metoda, na kterou se podíváme, je měření času. Přestože hardwarová podpora velmi urychluje běh VM, některé operace musí zákonitě trvat déle než na fyzickém počítači. Řeč je především o VM *Entry* a VM *Exit*. Metoda, kterou malware může využít je opakované volání instrukcí o kterých ví, že zaručeně způsobí VM *Exit*, například CPUID. Pokud trvání takového cyklu překročí určitou hranici je počítač pravděpodobně virtuální.

```
QueryPerformanceFrequency: 2727536 Hz

CPUID                       100000x
QueryPerfomanceCounter:     21288          (Total)
Time Stamp Counter:         21794265       (Total)
Prumerna doba CPUID:        217            OK
```

Obrázek 8.1: Měření volání instrukce CPUID na fyzickém počítači

```
QueryPerformanceFrequency: 2727871 Hz

CPUID                       100000x
QueryPerfomanceCounter:     305940         (Total)
Time Stamp Counter:         313200993      (Total)
Prumerna doba CPUID:        3132            (VM)
```

Obrázek 8.2: Měření volání instrukce CPUID na *VirtualBoxu*

Čas, který takto ve VM naměříme, je ve skutečnosti součtem doby trvání VM *Exit*, CPUID a VM *Entry*. Je zřejmé, že rozdíl oproti jediné instrukci, která by se vykonala na fyzickém počítači, musí být značný. Viz obrázky 8.1 a 8.2 a zdrojový kód 8.3.

```
#pragma optimize("", off)

BOOL TimingAttack(CONST UINT uiThreshold)
{
    ULONG64 ul64Start, ul64End, ulAverage;
    UINT uiRunner = 0, uiCycles = 100000;

    // Vlákno bude spouštěno na konkrétním CPU
    SetProcessAffinityMask(GetCurrentProcess(), 1);

    // Maximální priorita pro minimalizaci předbíhání
    SetPriorityClass(GetCurrentProcess(),
                     REALTIME_PRIORITY_CLASS);

    SetThreadPriority(GetCurrentThread(),
                      THREAD_PRIORITY_TIME_CRITICAL);

    ul64Start = __rdtsc();          // Začátek měření
    while (uiRunner++ < uiCycles)
        __cpuid(out, 1);            // Vynucený VM Exit
    ul64End = __rdtsc();            // Konec měření

    ulAverage = (ul64End - ul64Start) / uiCycles;
    return uiThreshold < ulAverage;
}

#pragma optimize("", on)
```

Zdrojový kód 8.3: Měření CPUID

Nabízí se otázka, jaký zdroj času použít, jelikož mohou být různě přesné. V naší ukázce jsme použili Time-stamp counter, který je v každém logickém procesoru (jádru). Museli jsme proto omezit běh funkce pouze na jeden procesor. Další možnosti jsou například *Performance counter* implementovaný Windows[46] - funkce `QueryPerfomanceCounter`, APIC Timer nebo externí zdroj jako například síťové hodiny.

8.4.1 Skrývání

Potřebujeme vytvořil iluzi, kdy čas ve VM plyne pomaleji než na jejím hostitelském počítači. Podvádění času může být obecně velmi špatně řešitelné, protože bychom museli pokrýt všechny možné zdroje a pokaždé, když si je VM vyžádá, je vhodně upravit.

VirtualBox naštěstí k času přistupuje pro nás výhodným způsobem. Různé zdroje času[1] emuluje jedním čítačem, kterému je přičítána pouze doba strávená v *non-root* módu. Díky tomu je započítán pouze běh VM a ne čas strávený v hypervisoru. To bohužel neřeší celý problém jelikož samotné přechody také něco trvají. My však využijeme tohoto způsobu a při každém ukončování měření, tj. při VM *Exit*, odečteme vhodnou hodnotu. To by měla být právě doba, kterou trvá VM *Exit* a VM *Entry*. Efektivně tak způsobíme, že pokaždé když se VM zeptá kolik je, resp. jak dlouho běží, dostane čas od svého spuštění mínus čas strávený v hypervisoru mínus čas strávený VM přechody.

A to je přesně co chceme. Otázkou zůstává kde vzít tu *vhodnou* hodnotu. Postup, který jsme zvolili je následující: Necháme hypervisor i VM při startu změřit, jak dlouho trvá instrukce CPUID uvedeným způsobem. VM poté sdělí svůj čas přes CPUID kanál a hypervisor od něj odečte svůj. Od času VM přechodů a CPUID instrukce tak odečteme čas samotného CPUID a dostaneme čistý čas VM přechodů, což je naše hledaná hodnota. Viz obrázek 8.3. Obrázek 8.4 ukazuje výsledek měření na upraveném *VirtualBoxu*.

```
10.91522121  656606562500 - STORMINI: StorAHCI - LPM:
11.70820427  EMInterpretCpuId: CPUIDOffset: 1755
11.91580772  656616562500 - STORMINI: StorAHCI - LPM:
```

Obrázek 8.3: DbgView, rozdíl v CPUID měřeních VM/VMM

```
QueryPerformanceFrequency: 2727880 Hz

CPUID                          100000x
QueryPerfomanceCounter:        223370          (Total)
Time Stamp Counter:            51614833        (Total)
Prumerna doba CPUID:           516             OK
```

Obrázek 8.4: Měření CPUID na upraveném *VirtualBoxu*

[1] mimo externích

Závěr

Představili jsme počítačovou virtualizaci od jejích počátků až k dnešnímu využití antivirovými firmami. Zároveň jsme uvedli několik výhod virtualizace, které jsou důvodem její stále rostoucí popularity. Uvedli jsme rozdíly mezi hypervisory typu I a II a představili několik virtualizačních produktů a jejich rozdíly.

Po teoretickém úvodu jsme detailně popsali mechanizmus hardwarové podpory virtualizace, jako jeden z prostředků vytvoření virtuálního prostředí. Ukázali jsme několik činností, které musí zastat každý hypervisor využívající hardwarové podpory.

V praktické části jsme zkoumali jeden z volně dostupných virtualizačních nástrojů *VirtualBox* a hledali jsme místa, kterými se jím vytvářené virtuální počítače odlišují od fyzických. Naše zkoumání jsme rozdělili do třech částí.

V první jsme popsali operační systém Windows a stopy, které v něm *VirtualBox* zanechává. Nevyhnuli jsme se při tom popisu některých interních mechanizmů OS Windows. Základním principem skrývání odlišností bylo dynamické generování hodnot, např. názvů souborů, které jsou jinak statické a tudíž detekovatelné. Pro naše potřeby jsme vytvořili komunikační kanál mezi hypervisorem a VM s využitím instrukce CPUID. Byli jsme tak schopni snadno distribuovat generované hodnoty mezi jednotlivými komponentami virtuálního počítače, který závisí na jejich znalosti.

V druhé části jsme popsali virtuální hardware a jak rozpoznat, že jej vytváří virtualizační nástroj. Zde jsme upravili virtuální hardware tak, aby vystupoval jako skutečný. To obnášelo změnit výrobce BIOSu, model pevného disku a tak dále.

Poslední část byla věnována rozdílům v samotném virtuálním prostředí. Jmenovitě odlišné chování některých instrukcí, rozdílné procesorové hodnoty nebo různé časové závislosti mezi virtuálním a fyzickým počítačem. Podobně jako v druhé části jsme se snažili sjednotit vystupování virtuálního počítače s fyzickým.

Rozhodně si netroufáme tvrdit, že jsme nalezli všechny odlišnosti virtuálních strojů. Pro každou uvedou jsme však představili postup, jak ji zakrýt a většinu jsme implemetovali přímo do zdrojových kódů *VirtualBoxu*. Uvedené principy jsme se snažili konstruovat dostastečně obecně, aby mohly být použity i v jiných nástrojích.

Cílem veškerého našeho snažení bylo obelstít malware a donutit ho projevit se i ve virtuálních počítačích. Po doimplementování změn, které jsme nestihli v rámci této práce, je dalším krokem nasazení upraveného *VirtualBoxu* na servery společnosti AVAST Software s.r.o., na kterých probíhá analýza malwaru a změření učinnosti úprav.

Dalším místem, kde společnost AVAST Software s.r.o. využívá virtualizaci je přímo u koncových zákazníků ve formě takzvaného *sandboxu*. Jedná se o softwarovou virtualizaci, kterou jsme neměli příležitost v práci zmínit. Vzdálenějším výhledem do budoucna je tak studium softwarové virtualizace a její případné zlepšení.

Literatura

[1] Cybersecurity Ventures [online]: *Cybersecurity Market Report*
Cybersecurity Ventures: ©1999-2016 [vid. 30. 4. 2017].
Dostupné z: `http://cybersecurityventures.com/cybersecurity-market-report/`

[2] McAfee [online]: *Virus Profile: Ping Pong*
McAfee, LLC: ©2003-2017 [vid. 30. 4. 2017].
Dostupné z: `https://home.mcafee.com/virusinfo/virusprofile.aspx?key=938`

[3] Webroot [online]: *Webroot 2015: Threat brief*
Webroot Inc.: ©2004-2016 [vid. 30. 4. 2017].
Dostupné z: `www.webroot.com/shared/pdf/Webroot_2015_Threat_Brief.pdf`

[4] Oracle [online]: *VirtualBox*
Oracle: ©1995-2017 [vid. 30. 4. 2017].
Dostupné z: `https://www.virtualbox.org`

[5] VMWare [online]: *Virtualization Overview*
VMWare, Inc.: ©2017 [vid. 30. 4. 2017].
Dostupné z: `http://www.vmware.com/pdf/virtualization.pdf`

[6] Moore's Law [online]: *Moore's Law or how overall processing power for computers will double every two years*
MemeBridge: ©2017 [vid. 30. 4. 2017].
Dostupné z: `http://www.mooreslaw.org`

[7] Matthew Portnoy. *Virtualization Essentials, 2nd Edition.*
John Wiley & Sons Inc., 978-1-119-26772-0, 2016.

[8] Gerald J. Popek, Robert P. Goldberg. *Formal requirements for virtualizable third generation architectures.*
Magazine Communications of the ACM
Volume 17 Issue 7, 1974. Strany 412-421.

[9] VMWare [online]: *Workstation for Windows*
VMWare, Inc.: ©2017 [vid. 30. 4. 2017].
Dostupné z: `http://www.vmware.com/products/workstation.html`

[10] Xen Project [online]: *Amazon Web Services*
Xen Project, a Linux Foundation Collaborative Project: ©2013 [vid. 30. 4. 2017].
Dostupné z: `https://www.xenproject.org/project-members/141-amazon-web-services.html`

[11] Microsoft Developer Network [online]: *Hyper-V Architecture*
Microsoft: ©2017 [vid. 30. 4. 2017].
Dostupné z: `https://msdn.microsoft.com/en-us/library/cc768520(v=bts.10).aspx`

[12] GNU [online]: *GNU General Public License, version 2*
Free Software Foundation [vid. 30. 4. 2017].
Dostupné z: `https://www.gnu.org/licenses/old-licenses/gpl-2.0.en.html`

[13] Microsoft Developer Network [online]: *Managing Virtual Memory*
Microsoft: ©2017 [vid. 30. 4. 2017].
Dostupné z: `https://msdn.microsoft.com/en-us/library/ms810627.aspx`

[14] DTMF [online]: *Open Virtualization Format*
Distributed Management Task Force, Inc.: ©2017 [vid. 30. 4. 2017].
Dostupné z: `https://www.dmtf.org/standards/ovf`

[15] GNU [online]: *GNU Tar*
Free Software Foundation [vid. 30. 4. 2017].
Dostupné z: `https://www.gnu.org/software/tar/`

[16] VMWare [online]: *Virtual Disk Format*
VMWare, Inc.: ©2017 [vid. 30. 4. 2017].
Dostupné z: `https://www.vmware.com/support/developer/vddk/vmdk_50_technote.pdf`

[17] Intel [online]: *Enabling Intel Virtualization Technology Features and Benefits*
Intel Corporation: ©2010 [vid. 30. 4. 2017].
Dostupné z: http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/virtualization-enabling-intel-virtualization-technology-features-and-benefits-paper.pdf

[18] AMD [online]: *AMD64 Virtualization*
Advanced Micro Devices, Inc.: ©2010 [vid. 30. 4. 2017].
Dostupné z: http://developer.amd.com/wordpress/media/2012/10/9-David_Obrien-AMD_India_SVM_DO_v1.pdf

[19] CPU Benchmarks [online]: *AMD vs Intel Market Share*
PassMark Software: ©2017 [vid. 30. 4. 2017].
Dostupné z: https://www.cpubenchmark.net/market_share.html

[20] Intel [online]: *Intel 64 and IA-32 Architectures Software Developer's Manual*
Intel Corporation: ©2016 [vid. 30. 4. 2017].
Dostupné z: https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf

[21] Candid Wueest [online]: *Does malware still detect virtual machines?*
Symantec Corporation: ©2017 [vid. 30. 4. 2017].
Dostupné z: https://www.symantec.com/connect/blogs/does-malware-still-detect-virtual-machines

[22] AV-Comparatives [online]: *Real-World Protection Test*
AV-Comparatives: ©2016 [vid. 30. 4. 2017].
Dostupné z: http://www.av-comparatives.org/wp-content/uploads/2016/05/avc_factsheet2016_04.pdf

[23] Microsoft [online]: *Visual Studio Code*
Microsoft: ©2017 [vid. 30. 4. 2017].
Dostupné z: https://code.visualstudio.com

[24] Microsoft [online]: *MIT License*
Microsoft: ©2015 [vid. 30. 4. 2017].
Dostupné z: https://github.com/Microsoft/vscode/blob/master/LICENSE.txt

[25] Microsoft [online]: *Windows Sysinternals*
Microsoft: ©2017 [vid. 30. 4. 2017].
Dostupné z: https://technet.microsoft.com/en-us/sysinternals/default

[26] Microsoft Developer Network [online]: *Windows Kernel-Mode Object Manager*
Microsoft: ©2017 [vid. 30. 4. 2017].
Dostupné z: `https://msdn.microsoft.com/en-us/library/windows/hardware/ff565763(v=vs.85).aspx`

[27] Microsoft Developer Network [online]: *Windows Driver Kit*
Microsoft: ©2017 [vid. 30. 4. 2017].
Dostupné z: `https://msdn.microsoft.com/en-us/library/windows/hardware/ff557573(v=vs.85).aspx`

[28] Microsoft Developer Network [online]: *Remote Debugging*
Microsoft: ©2017 [vid. 30. 4. 2017].
Dostupné z: `https://msdn.microsoft.com/en-us/library/y7f5zaaa.aspx`

[29] Sysprogs [online]: *VirtualKD*
Sysprogs OÜ: ©2012-2017 [vid. 30. 4. 2017].
Dostupné z: `https://msdn.microsoft.com/en-us/library/y7f5zaaa.aspx`

[30] Microsoft Developer Network [online]: *Named Pipes*
Microsoft: ©2017 [vid. 30. 4. 2017].
Dostupné z: `https://msdn.microsoft.com/en-us/library/windows/desktop/aa365590(v=vs.85).aspx`

[31] GNU [online]: *GNU Lesser General Public License*
Free Software Foundation [vid. 30. 4. 2017].
Dostupné z: `https://www.gnu.org/licenses/lgpl-3.0.en.html`

[32] Oracle [online]: *Guest Additions*
Oracle: ©1995-2017 [vid. 30. 4. 2017].
Dostupné z: `https://www.virtualbox.org/manual/ch04.html`

[33] Microsoft Developer Network [online]: *Object Names*
Microsoft: ©2017 [vid. 30. 4. 2017].
Dostupné z: `https://msdn.microsoft.com/en-us/library/windows/desktop/ms684292(v=vs.85).aspx`

[34] Microsoft Developer Network [online]: *Mutex Objects*
Microsoft: ©2017 [vid. 30. 4. 2017].
Dostupné z: `https://msdn.microsoft.com/en-us/library/windows/desktop/ms684266(v=vs.85).aspx`

[35] Microsoft Developer Network [online]: *Semaphore Objects*
Microsoft: ©2017 [vid. 30. 4. 2017].
Dostupné z: `https://msdn.microsoft.com/en-us/library/windows/desktop/ms685129(v=vs.85).aspx`

[36] Microsoft Developer Network [online]: *Event Objects*
Microsoft: ©2017 [vid. 30. 4. 2017].
Dostupné z: `https://msdn.microsoft.com/en-us/library/windows/desktop/ms682655(v=vs.85).aspx`

[37] Microsoft Developer Network [online]: *Registry*
Microsoft: ©2017 [vid. 30. 4. 2017].
Dostupné z: `https://msdn.microsoft.com/en-us/library/windows/desktop/ms724871(v=vs.85).aspx`

[38] Microsoft Developer Network [online]: *IRP Major Function Codes*
Microsoft: ©2017 [vid. 30. 4. 2017].
Dostupné z: `https://msdn.microsoft.com/en-us/library/windows/hardware/ff550710(v=vs.85).aspx`

[39] Microsoft Developer Network [online]: *Service Control Manager*
Microsoft: ©2017 [vid. 30. 4. 2017].
Dostupné z: `https://msdn.microsoft.com/en-us/library/windows/desktop/ms685150(v=vs.85).aspx`

[40] Microsoft Developer Network [online]: *Interprocess Communications*
Microsoft: ©2017 [vid. 30. 4. 2017].
Dostupné z: `https://msdn.microsoft.com/en-us/library/windows/desktop/aa365574(v=vs.85).aspx`

[41] PCI Database [online]: *PCI Vendor and Device Lists*
2017 [vid. 30. 4. 2017].
Dostupné z: `http://pcidatabase.com`

[42] Stephen J. Gowdy [online]: *The USB ID Repository*
2017 [vid. 30. 4. 2017].
Dostupné z: `https://usb-ids.gowdy.us/read/UD/`

[43] Microsoft Developer Network [online]: *About INF Files*
Microsoft: ©2017 [vid. 30. 4. 2017].
Dostupné z: `https://msdn.microsoft.com/en-us/library/windows/desktop/aa376858(v=vs.85).aspx`

[44] IEEE Standards Association [online]: *List OUI*
IEEE: ©2017 [vid. 30. 4. 2017].
Dostupné z: `http://standards-oui.ieee.org/oui.txt`

[45] Microsoft Developer Network [online]: *Requirements for Implementing the Microsoft Hypervisor Interface*
Microsoft: ©2012 [vid. 30. 4. 2017].
Dostupné z: `https://github.com/Microsoft/Virtualization-Documentation/raw/master/tlfs/Requirements%20for%20Implementing%20the%20Microsoft%20Hypervisor%20Interface.pdf`

[46] Microsoft Developer Network [online]: *Performance Counters*
Microsoft: ©2017 [vid. 30. 4. 2017].
Dostupné z: `https://msdn.microsoft.com/en-us/library/windows/desktop/aa373083(v=vs.85).aspx`

[47] IBM Cloud [online]: *What is cloud computing?*
International Business Machines Corporation: ©2017 [vid. 30. 4. 2017].
Dostupné z: `https://www.ibm.com/cloud-computing/learn-more/what-is-cloud-computing/`

Příloha **A**

Seznam použitých zkratek

AMD Advanced Micro Devices

BIOS Basic Input Output System

DDOS Distributed denial of service

DOS Disk Operating System

IBM International Business Machiness

MaaS Malware as a Service

MS Microsoft

NS Namespace

OS Operating System/Operační Systém

OVA Open Virtual Appliance

OVF Open Virtualization Format

SVM Secure Virtual Machine

VM Virtual Machine

VMCS Virtual Machine Contol Structure

VMM Virtual Machine Monitor

VMDK Virtual Machine Disk

VMX Virtual Machine Extenstions

VPN Virtual Private Network

XML Extensible Markup Language

#UD UnDefined Opcode

Příloha B

VMX instrukce

Zkratka	**Popis**
VMPTRLD *X*	*X* je 64bitový ukazatel na VMCS strukturu. Ta se po vykonání stane aktivní a aktuální.
VMPTRST *X*	*X* je 64bitový ukazatel na adresu, kam bude po vykonání uložen ukazatel na aktuální VMCS strukturu nebo 0xFFFFFFFF_FFFFFFFF, pokud žádná VMCS není aktuální.
VMCLEAR *X*	*X* je 64bitový ukazatel na VMCS strukturu. Ta se po vykonání stane neaktivní, neaktuální a nespuštěná.
VMREAD *Y X*	Přečte položku *X* aktuální VMCS struktury a uloží ji na místo určené *Y*.
VMWRITE *Y X*	Uloží hodnotu *X* do položky *Y* aktuální VMCS struktury.
VMLAUNCH VMRESUME	Aktuální VMCS struktura se stane spuštěná - nastane VM *Entry*.
VMXON *X*	*X* je 64bitový ukazatel na VMXON *Region*. Procesor po vykonání vstoupí do *root* módu VMX operace.
VMXOFF	Procesor ukončí VMX operaci.
INVEPT INVVPID	Zneplatní stránkovací struktury - *translation lookaside buffer* a *page-structure cache*
VMCALL (*non-root* mód)	Způsobí VM *Exit*, pokud se procesor nachází v *non-root* módu VMX operace.
VMFUNC (*non-root* mód)	Vyvolá VM funkci v *root* módu VMX operace bez způsobení VM *Exit*. Identifikátor funkce je hodnota EAX registru.

Příloha C

Obsah přiloženého DVD

- `src`
  - `Thesis` ..... zdrojové kódy práce
  - `TimingAttack` ..... zdrojové kódy měření CPUID instrukce
  - `VBoxDetect` ..... zdrojové kódy detektoru *VirtualBoxu*
  - `VirtualBox` ..... zdrojové kódy *VirtualBoxu*
    - `README.txt` ..... popis jak sestavit *VirtualBox*
- `text`
  - `DP_Hradsky_Tomas_2017.pdf` ..... text práce ve formátu PDF