UNIVERZITA PARDUBICE
Fakulta elektrotechniky a informatiky

Bezpečnost jádra operačního systému

Čeněk Ráliš

Bakalářská práce
2013

Univerzita Pardubice
Fakulta elektrotechniky a informatiky
Akademický rok: **2012/2013**

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

(PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU)

Jméno a příjmení:	**Čeněk Ráliš**
Osobní číslo:	**I10187**
Studijní program:	**B2646 Informační technologie**
Studijní obor:	**Informační technologie**
Název tématu:	**Bezpečnost jádra operačního systému**
Zadávající katedra:	**Katedra informačních technologií**

Zásady pro vypracování:

První část práce bude zaměřena na možnosti zvýšení bezpečnosti systému při spouštění procesů jako je např. implementace nespustitelného zásobníku, zákaz zápisu jádra do uživatelského prostoru apod. Diskutovány budou výhody a nevýhody možných přístupů v implementaci ochran a také známé exploity systému související s tématem.
Druhá část práce se zaměří na bezpečnost procesů z pohledu programátora, tedy zejména na implementaci systémových knihoven, možnosti zneužití nulových ukazatelů apod.
V praktické části budou vypracovány ukázky kódů jádra či možných exploitů a programů, které byly diskutovány v předchozí části. Celý souhrn ukázek bude vypracován jako výukový přehled určený správcům operačních systémů či programátorům jádra a ovladačů systému.

Rozsah grafických prací:

Rozsah pracovní zprávy:

Forma zpracování bakalářské práce: **tištěná/elektronická**

Seznam odborné literatury:

***ERICKSON, Jon. Hacking: umění exploitace. 2. upravené a doplněné vydání. Překlad Jan Pokorný. Brno: Zoner Press, 2009. 544 s. ISBN 978-80-7413-022-9.**
***DOBŠÍČEK, Miroslav. Linux: bezpečnost a exploity. 1. vydání. České Budějovice: Kopp, 2004. 157 s. ISBN 80-723-2243-5.**
***PERLA, Enrico. A guide to kernel exploitation: attacking the core. Amsterdam: Elsevier, 2011, 442 s. ISBN 978-159-7494-861.**

Vedoucí bakalářské práce: **Mgr. Tomáš Hudec**
Katedra informačních technologií

Datum zadání bakalářské práce: **21. prosince 2012**
Termín odevzdání bakalářské práce: **10. května 2013**

L.S.

prof. Ing. Simeon Karamazov, Dr.
děkan

Ing. Lukáš Čegan, Ph.D.
vedoucí katedry

V Pardubicích dne 29. března 2013

Prohlášení autora

Prohlašuji, že jsem tuto práci vypracoval samostatně. Veškeré literární prameny a informace, které jsem v práci využil, jsou uvedeny v seznamu použité literatury.

Byl jsem seznámen s tím, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorský zákon, zejména se skutečností, že Univerzita Pardubice má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona, a s tím, že pokud dojde k užití této práce mnou nebo bude poskytnuta licence o užití jinému subjektu, je Univerzita Pardubice oprávněna ode mne požadovat přiměřený příspěvek na úhradu nákladů, které na vytvoření díla vynaložila, a to podle okolností až do jejich skutečné výše.

Souhlasím s prezenčním zpřístupněním své práce v Univerzitní knihovně.

V Pardubicích dne 28. 4. 2013 Čeněk Ráliš

Poděkování

Mé poděkování patří na prvním především vedoucímu práce, panu Mgr. Tomáši Hudcovi za trpělivé vedení a odborné rady. Díky profesionálnímu vedení bylo vypracování práce radostí, zdrojem inspirace a cenných zkušeností. Dále bych rád poděkoval všem, kteří mě během doby studia podněcovali k lepším výkonům a byli mi oporou. Děkuji.

Anotace

Práce se zabývá bezpečností jádra operačního systémů. Pro přehlednost je práce dělena do několika logických částí. Teoretická část poskytuje základní teoretický aparát spolu s technikami užívaných útočníkem. Praktická část obsahuje případové studie vybraných operačních systémů a odbornou výukovou příručku. Problematika je probírána na architektuře IA-32, s využitím operačních systémů Microsoft Windows a GNU/Linux.

Klíčová slova

bezpečnost, operační systém, jádro, Linux, exploit

Title

Security of Operating System Kernel

Annotation

Topic of this bachelor work is a Security of operating system kernel. Work is divided to logical sections for a good arrangement. Theoretical section presents elementary staff and techniques applied by attacker. Case studies of selected operating systems with practical guide aimed to IT professionals are included in practical section. All these and examples are demonstrated at IA-32 microprocessor architecture with GNU/Linux or Microsoft Windows operating systems.

Keywords

security, operating system, kernel, Linux, exploit

Obsah

Seznam zkratek

AMD	Advanced Micro Devices
ASCII	American Standard Code for Information Interchange
AMP	Asymmetric Multiprocessing
BIOS	Basic Input/output System
CVE	Common Vulnerabilities an Exposures
DoS	Denial of Service
FUSE	Filesystem in Userspace
GNU	GNU is Not Unix!
IA	Intel Architecture
IPC	Inter Process Communication
NOP	No Operation
NSA	National Security Agency
NULL	Nullus
NX	No execute
PAE	Physical Address Extension
POSIX	Portable Operating System Interface
PSE	Page Size Extension
SMP	Symmetric Multiprocessing

Slovník

Exploit	program zneužívající implementační chybu
Hardware	hmotné, technické vybavení (součásti) systému
Patch	oprava části programu postiženého chybou či zranitelností (záplata)
Shellcode	část programového kódu schopná samostatného běhu

Seznam obrázků

Seznam tabulek

Úvod

Tématem bakalářské práce je bezpečnost jádra operačního systému. Díky stále většímu vlivu počítačů na lidský život je otázka kybernetické bezpečnosti stále vážnějším problémem, na který by se nemnělo zapomínat. Jedním z faktorů ovlivňujících vážnost tématu je rostoucí dostupnost komerčních nabídek cílených útoků. Zavádění přísnějších bezpečnostních politik v uživatelském prostoru vede k stále většímu počtu útoků zaměřených přímo na jádro operačního systému. Probíraná tématika by mněla být zohledněna během administrace kritických systémů, návrhu bezpečné infrastruktury či programování aplikací využívajících užší spolupráci s operačním systémem.

Teoretická část práce předkládá základní teoretický aparát a techniky užívané útočníkem. Praktická část využívá dříve probraný teoretický výklad pro potřeby vypracování případových studií jednotlivých operačních systémů. V rámci případových studií budou předloženy ukázkové exploity. Součástí praktické části je programátorská příručka. Práce je vypracována na architektuře IA-32, zvolené pro své rozšíření a dostupnost.

Během přípravy práce jsem neintenzivněji využíval knihu, jejímž autorem je Enrico Perla, *A guide to kernel exploitation: attacking the core*, popisující základní techniky útoku na jádro operačního systému. Dále jsem využíval knihu, jejímž autorem je Jon Erickson, *Hacking: umění exploitace*.

Probíraná látka je natolik rozsáhlá, že každá část práce by vydala na samostatnou knihu, je tedy nutné na hloubku probíraných detailů hledět jako na ucelený přehled základních informací. Hlubšího prozkoumání tématu je možné dosáhnout studiem odborné literatury a periodik, navštěvováním konferencí a v neposlední řadě spoustou praktických zkušeností získaných vlastním bádáním.

Věřím, že tato práce napomůže k zamyšlení se nad otázkou bezpečnosti a současně upozorní na chyby odborníků. Závažným problémem je časté opomíjení jádra při zabezpečování systému, toto je převážně způsobeno téměř neotřesitelnou důvěrou vkládanou některými odborníky do této kritické části operačního systému. Praktiky popisované v této práci mohou být v závislosti na použití a legislativních úpravách nezákonné. Autor této práce v žádném případě nenabádá ke konání trestního, neetické, či amorálního jednání.

1 Mikroprocesory architektury IA-32

Detaily fungování systému nejsou nutné k psaní programů zneužívajících programátorské chyby, ovšem jejich znalost je vhodná. Většina útočníků tyto znalosti má a během své práce jich plně využívá.

Označení IA-32 zavedla společnost Intel pro všechny mikroprocesory kompatibilní s mikroprocesorem 80386. Architektura IA-32 je platforma určena pro osobní počítače a servery, jedná se o jednu z nejrozšířenějších platforem. Hlubší pojednání je možné nalézt například v oficiální dokumentaci [1], či knihách zaměřených na popis funkce hardware [2].

1.1 Rozšíření AMD64

Potřeba rozšíření adresovatelné paměti a navýšení výkonu (zvýšení objemu dat zpracovávaných v jednom okamžiku) vedla k návrhu rozšíření původní architektury. Společnost AMD uvedla v roce 2000 rozšíření označené AMD64, společnost Intel jej převzala pod vlastním označením EMT64T. Pro zachování zpětné kompatibility je rozšíření implementováno formou dalších pracovních módů mikroprocesoru.

Architektura	Mód procesoru	Šířka adresy (bit)
AMD64	64-bit	64
	Kompatibilní	16/32
IA-32	Chráněný	32
	Virtuální	16
	Reálný	16

Tabulka 1.1: Operační módy mikroprocesoru

1.2 Ochrana vykonávání dat

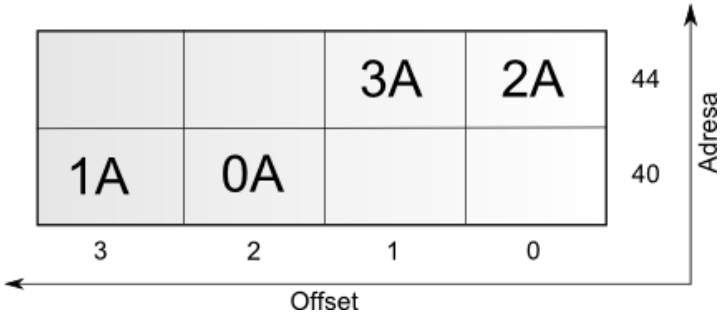
Útočník velmi často potřebuje vykonat data uložená v paměti jako programové instrukce, bit No-Execute (NX) má této technice zabránit. Ochranu je naneštěstí v závislosti na implementaci možné obejít (viz. případové studie), nikoliv však prolomit.

Doposud bylo možné kontrolovat pouze práva pro čtení, zápis a přístupová privilegia. Ochranný bit lze nalézt uvnitř záznamu stránkovací tabulky, konkrétně na **63. pozici**.

Díky umístění je možné funkčnost využít pouze za využití plně 64bitového módu, či rozšíření PAE (rozšíření adresovatelné fyzické paměti).

1.3 Reprezentace dat

Na architektuře IA-32 jsou data do paměti ukládána použitím notace **little endian**, kde jsou významnější bajty umístěny na vyšší paměťové adresy. Obrázek 1.1 znázorňuje uložení čísla 0x3A2A1A0A na adrese 0x42.

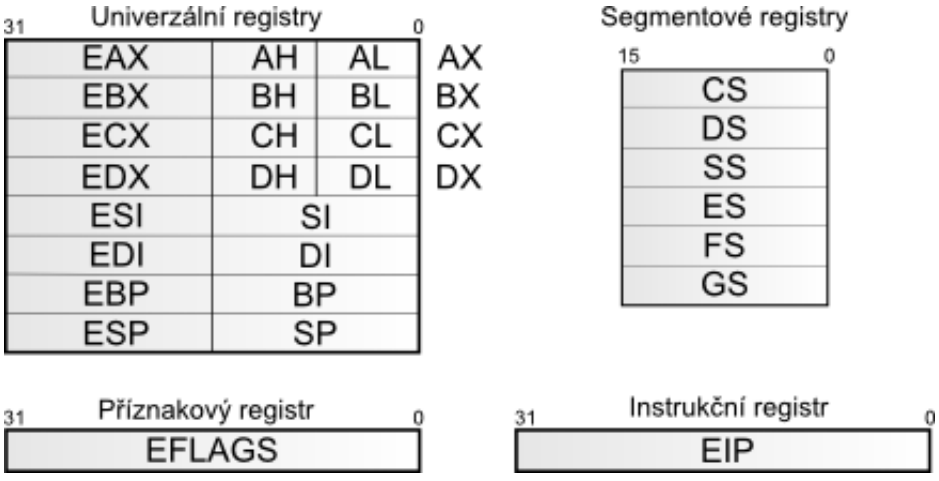


Obrázek 1.1: Little-endian – uložení hodnoty v paměti

Protikladná notace se nezývá **big endian** a je síťovým standardem. Některé architektury umožňují použití obou notací, poté jsou označovány termínem **bi-endian**. Příkladem takové architektury je např. IA-64, MIPS, atd.

1.4 Základní registry

Programátor má na architektuře IA-32 k dispozici 16 základních registrů. Obrázek 1.2 ukazuje základní registry spolu s jejich označením v 16 a 32bitovém módu. Vybrané univerzální registry poskytují 8bitový přístup k dolní polovině registru. Kromě těchto registrů jsou k dispozici další specializované registry, kterými se zde nebudeme zabývat.



Obrázek 1.2: Základní registry architektury IA-32

1.4.1 Univerzální registry

Univerzální registry mají roli operandů během vykonávání aritmetických, logických a paměťových výpočtů. Dále slouží jako paměťové ukazatele. Každý z osmi registrů má vlastní specifický účel:

- **EAX** – operandy a výsledky aritmetickologických operací,
- **EBX** – ukazatel vrcholu datové oblasti,
- **ECX** – čítač (cykly, práce s textem),
- **EDX** – vstupně/výstupní ukazatel,
- **ESI** – ukazatel na zdrojová data kopírovacích operací (např. práce s řetězci),
- **EDI** – ukazatel na cílová data kopírovacích operací,
- **ESP** – ukazatel dna zásobníku (stack),
- **EBP** – ukazatel vrcholu zásobníku.

1.4.2 Segmentové registry

Informace uchovávané segmentovým registrem, stejně jako způsob jeho využití závisí na použitém typu organizace paměti. Pro přístup ke konkrétnímu segmentu paměti je nutné, aby byl uložen uvnitř jednoho segmentového registru.

Při použití přímé adresace (**bez segmentace**) paměti ukazují všechny registry na počátek lineárního adresového prostoru (adresa 0), jednotlivé segmenty se tedy překrývají. Segmentový registr obsahuje 16 bitů širokou adresu segmentu s nulovou hodnotou. Tento režim práce s pamětí je zastaralý.

Obrázek 1.3: Struktura segmentového registru

Zapnutí **segmentace** rozdělí lineární adresový prostor na několik segmentů. Každý segmentový registr má na starost jeden segment paměti. Registr CS ukazuje na segment obsahující programový kód, DS, ES, FS a GS na datový segment, SS obstarává zásobník. Obrázek 1.3 popisuje vnitřní uspořádání selektoru segmentu. Bit **TI** rozhoduje, zda se jedná o globální (0), či lokální (1) tabulku deskriptorů. **RTL** obsahuje požadovanou úroveň oprávnění. Index odkazuje na konkrétní záznam uvnitř globální (GDT), nebo

lokální (LDT) tabulky deskriptorů. Segmentace je na architektuře IA-32 povinnou součástí adresace.

1.4.3 Příznakový registr

Příznakový registr obsahuje systémové, stavové a kontrolní příznaky. Bity 1, 3, 5, 12 a 22 až 31 jsou rezervovány a neměly by se používat. Po provedení inicializace, či restartování mikroprocesoru se registr nachází ve výchozím nastavení o hodnotě **0x00000002** (nastaven pouze rezervovaný bit).

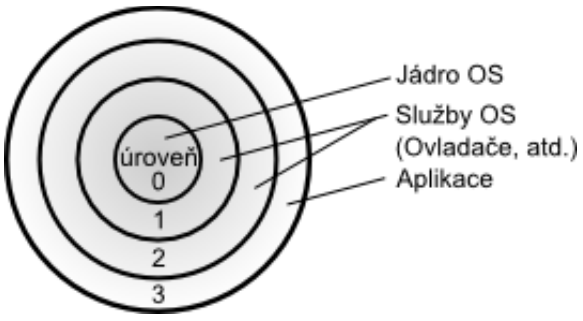
Mezi nejzajímavější hodnoty registru patří příznaky přetečení, nulového výsledku, zapnutého přerušení, virtuálního přerušení a virtuálního operačního módu.

1.4.4 Instrukční registr

Registr obsahuje offset, uvnitř aktuálního kódového segmentu, ukazující na následující programovou instrukci. Během načtení instrukce je obsah registru automaticky inkrementován. Obsah registru je pro programátora neměnný, jediný způsob jak změnit jeho hodnotu je pomocí skokových instrukcí, volání rutin (CALL), návratu (RET) a návratu z obsluhy přerušení (IRET).

1.5 Úroveň oprávnění

Mezi základní vlastnosti poskytované operačním systém patří izolace jednotlivých částí operačního systému. Přesně k tomu slouží úrovně oprávnění. Architektura podporuje čtyři úrovně oprávnění, kde vyšší číslo znamení méně práv. Prakticky se většinou využívá pouze třetí a nulová úroveň oprávnění. Podpora virtualizace zavádí další úroveň oprávnění označenou číslem -1.



Obrázek 1.4: Úrovně oprávnění (Rings)

Pravidla pro přístup mezi úrovněmi oprávnění říká, že není možné vykonávat kód z nižších úrovní oprávnění a přistupovat k datům na vyšší úrovni, nežli je aktuální úroveň.

Jak již bylo řečeno, segmentace je povinnou součástí adresace paměti, takže je vždy prováděn přístup pomocí segmentových registrů. Segmentové registry CS a SS obsahují přidělenou úroveň oprávnění (CPL), což ovlivňuje data a zásobník. Během přístupu do paměti je větší z hodnot (aktuální a požadované úrovně oprávnění) zvolena jako efektivní úroveň oprávnění (EPL). Efektivní úroveň oprávnění je následně porovnána s hodnotou v deskriptoru segmentu, podle čehož je vyhodnocena možnost přístupu.

1.6 Přístupová práva

Dalším ochranným mechanizmem implementovaným architekturou IA-32 jsou přístupová práva, která je možné využít při použití stránkování paměti. Tabulka stránek obsahuje dva bity zodpovědné za tuto funkčnost. První z nich určuje, zda je daná stránka systémová, druhý kontroluje práva pro čtení a zápis. Označení pouze pro čtení probíhá vynulováním příslušného bitu.

Je-li stránka pomocí ochranného (U/S) bitu označena jako systémová, mohou k ní přistupovat procesy s úrovní oprávnění nižší než 3. V opačném případě mohou přistupovat pouze k uživatelským stránkám (bit nastaven na logickou jedničku).

1.7 Organizace paměti

Programátorovi je umožněno používat tři základní přístupy pro práci s pamětí, některé z nich umožňují využívat výhod stránkování a virtuální paměti. Je nutné rozlišovat typy adres:

- **Fyzická** – adresa uvnitř fyzického média (operační paměť), většinou s ní pracuje pouze řadič paměti,
- **Logická** – pracuje s ní programoví kód.

1.7.1 Přímé mapování paměti

K paměti proces přistupuje jako k celistvému prostoru, který nazýváme **lineární adresový prostor**. Veškerá data procesu (kód, zásobník, data) jsou obsažena v tomto paměťovém prostoru. Prostor je adresovatelný po jednotlivých bajtech (**lineární adresa**). Rozsah adresovatelné paměti je 2^{32} bajtů (4GB, vyjma plně 64bitového módu).

1.7.2 Segmentace paměti

Paměť se procesu jeví jako skupina nezávislých paměťových úseků, nazývaných **segmenty**. Segmentace je většinou využívána takovým způsobem, aby byly části programu

obsahující kód, data a zásobník umístěny v různých segmentech. Logická adresa je rozdělena na index (selektor segmentu) a offset (posun) v rámci daného segmentu. Program je schopen adresovat 16 383 segmentů různých velikostí a typů. Každý segment může být široký až 2^{32} bajtů (4GB). Segmentace je povinnou součást adresace paměti.

1.7.3 Reálný paměťový model

Jedná se model paměti zachovaný pro zpětnou kompatibilitu s mikroprocesory řady 8086. Jedná se o specifickou implementaci segmentování paměti, kde je programům a operačnímu systému poskytnut lineární adresový prostor skládající se z posloupnosti segmentů o velikosti 64KB. Maximální adresovatelná paměť je 2^{20} bajtů (1MB).

1.7.4 Stránkování paměti

Stránkování paměti je možné využít s přímým mapováním paměti, případně spolu se segmentací. Při použití rychlého mapování (bez stránkování) odpovídá logická adresa adrese fyzické. Adresa tedy není překládána. Zapnutím stránkování je lineární adresový prostor rozdělen na stránky, které jsou mapovány do virtuální paměti. Poté jsou dle potřeby jednotlivé stránky mapovány do fyzické paměti. Architektura IA-32 navíc umožňuje využít další možné vlastnosti:

- **rozšíření adresovatelné fyzické paměti** (PAE) – umožňuje využít více než 4GB fyzické paměti,
- **rozšíření velkosti stránek** (PSE) – mapování lineárního adresového prostoru do fyzické paměti s využitím stránek o velikosti 4MB,
- **kombinace předchozích** (PAE + PSE) – možnost využít stránky o velikosti 2MB.

2 Operační systém

Co je operační systém? Odpověď na tuto otázku není tak jednoduchá, jak by se na první pohled zdálo, záleží na zaměření publika a taktéž na úhlu pohledu na systém. Nejčastější odpověď popisuje operační systém jako základní programové vybavení počítače. Tato odpověď vychází z uživatelského pohledu na problematiku. V této práci budeme na operační systém nahlížet očima programátora, tedy jako na systém poskytující abstrakci hardware, správu zdrojů, služby a potřebná aplikační rozhraní. Tento celek tvoří základní běhové prostředí pro uživatelské aplikace. Bližší informace je možné získat v publikacích věnovaných jednotlivým operačním systémům [3] [4], případně v knihách o operačních systémech obecně [5] [6].

2.1 Základní datové struktury

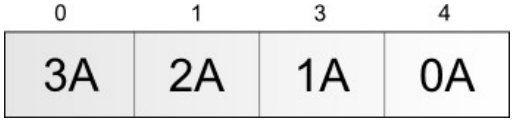
Operační systém potřebuje po celou dobu svého běhu uchovat obrovské množství informací různého druhu. Mezi takové informace patří například informace o volné fyzické paměti, běžících procesech, apod. Každý typ informaci je nutné uchovávat různými způsoby, pro maximální efektivitu zpracování. Například informace o volných blocích paměti je vhodné rychle procházet pro nalezení vhodné bloku.

Kromě základních struktur zde probraných existuje spousta dalších a dále nespočet jejich možných kombinací. Příkladem pokročilejšího typu datové struktury mohou být stromy, tabulky, apod.

2.1.1 Pole

Pole je základní datová struktura, vyznačující se konstantní dobou přístupu k libovolnému prvku. Jednotlivé prvky pole musí být shodného datového typu, můžeme tedy říci, že pole je daného typu. Samotné prvky se nacházejí v souvislém bloku paměti. Potřebujeme-li strukturu, jejíž velikost se bude často měnit, je vhodné uvažovat o použití seznamu.

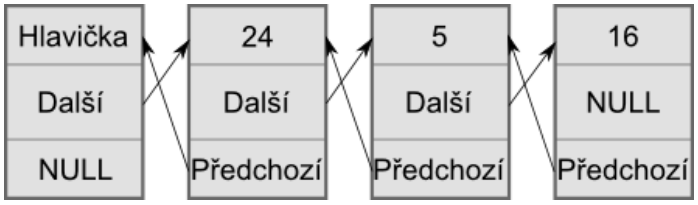
Zvykem bývá číslovat jednotlivé prvky od nuly, jedná se o frekventovanou oblast programátorských chyb. Takové chyby mají velice destruktivní dopad na bezpečnost.



Obrázek 2.1: Konkrétní ukázka pole

2.1.2 Zřetězený seznam

Díky postupné alokaci paměti jednotlivých prvků seznamu nepotřebuje tato struktura alokovat celistvý paměťový prostor. Tato výhoda je vykoupena nekonstantní přístupovou dobou k jednotlivým prvkům. Existuje **jednosměrně** a **obousměrně** zřetězený seznam. První z nich je označován také jako **lineární**, odkazuje-li poslední prvek opět na první, jedná se o **cyklický** seznam.



Obrázek 2.2: Příklad obousměrně zřetězeného seznamu

Každý prvek struktury obsahuje odkaz na následující prvek (v případě obousměrného zřetězení také na předchozí), konec seznamu je signalizován pomocí nulové adresy na začátku (případně počátku) seznamu. Někdy je seznam implementován tak, že místo ukládání adresy prvního prvku vytvoříme první prvek označený jako hlavička, který následně odkazuje na první prvek.

2.2 Procesy a vlákna

Proces je velice frekventovaný pojem, v tuto chvíli je vhodné jej podrobněji vysvětlit. Nejjednodušeji řečeno je proces konkrétní instancí aplikace uvnitř operační paměti a vykonávající dané programové instrukce. V širším záběru jde o sadu běžících vláken, přidělených systémových požadavků spolu s dalšími informacemi o procesu.

2.2.1 Tabulka procesů

Jedná se o datovou strukturu udržovanou operačním systémem, obsahující základní informace o jednotlivých procesech. Obsahuje následující informace:

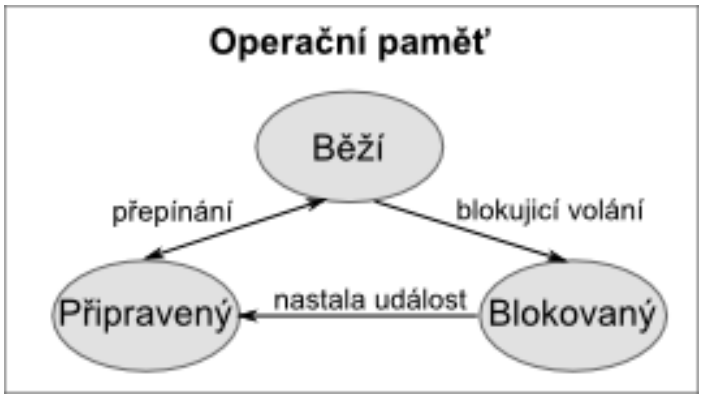
- **paměťový prostor** – ukazatele na jednotlivé paměťové segmenty (kód, data, zásobník, halda),
- **kontext procesu** – podrobné informace nutné k přerušení běhu procesu a jeho následujícímu pokračování (registry mikroprocesoru),
- **systémové informace** – identifikátor procesu, práva, přidělené prostředky (otevřené soubory).

Vlákno je zjednodušenou formou procesu, neobsahuje tedy veškeré informace výše popsané, pouze jejich podmnožinu. Konkrétně disponuje vlastními stavovými informacemi, kontext a zásobník, ostatní sdílí prostřednictvím procesu s ostatními vlákny. Vlastní zásobník je nutný, aby volání metod jednoho vlákna neovlivňovalo vlákna ostatní. Podobně toto platí i pro stavové informace a kontext.

2.2.2 Stavy procesů

Základní schopností moderního operačního systému je možnost současného běhu několika procesů. Alespoň zdánlivě, není-li možné využít více procesorů. Velmi úzce s tím souvisí možnost práce s virtuální pamětí, tedy možnost obhospodařovat paměťový prostor větší než rozsah operační paměti. Obě tyto funkcionality přinášejí potřebu definování několika stavů, v nichž se aktuálně proces nachází. Běžně jsou označovány dvě možné množiny, a to základní a rozšířené stavy procesu. Rozšířené stavy se prakticky nevyužívají, díky segmentaci paměti není nutné odkládat na disk celé procesy.

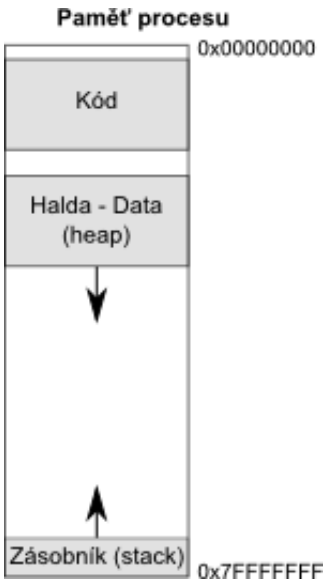
- **Základní stavy:**
    - *běžící* – vykonáván,
    - *připravený* – pozastaven jádrem systému,
    - *blokovaný* – čeká na vnější událost.



Obrázek 2.3: Rozšířený stavový model procesu

2.2.3 Paměť procesu

Virtuální adresový prostor procesu je rozdělen do několika částí, na počátku prostoru adres nalezneme programový kód aplikace, dále data ukládaná na haldu a na dně se nachází zásobník. Halda roste směrem nahoru, tedy od nižších paměťových adres k vyšším. Zásobník roste nahoru od nejvyšší adresy přiděleného prostoru, tedy k nižším adresám směrem k haldě.



Obrázek 2.4: Příklad uspořádání paměti procesu

2.3 Jádro operačního systému

Jádro tvoří hlavní programovou jednotku operačního systému. Dle zvolené architektury disponuje určitým rozsahem funkčnosti nezbytné pro práci sytému jako celku. Klasické vrstvené zobrazení operačního systému prezentuje jádro jako mezivrstvu vytvářející základní abstrakci a komunikační rozhraní mezi uživatelskými aplikacemi spolu s knihovnami a samotným hardware, případně příslušnými ovladači.

Existuje několik možných architektur jádra, nejčastěji se můžeme setkat s monolitickým jádrem. Dále existují **mikro** a **hybridní** jádra. **Mikrojádro** obstarává pouze základní funkce a komunikaci mezi jednotlivými částmi operačního systému, ostatní funkce jsou implementovány v jednotlivých modulech pracujících mimi privilegovaný prostor. (server). Uživatelské procesy (klienti) využívají služeb modulů jádra. Nízký výkon mikrojádra je způsoben notnou režií komunikace mezi moduly a jádrem samým. Hybridní jádro kombinuje rychlost monolitického jádra s elegancí a bezpečností mikrojádra.

Monolitické jádro se vnějšímu pozorovateli jeví jako kompaktní celek poskytující veškerou funkcionalitu. Architektura umožňuje vnitřní rozdělení na úrovně odpovědnosti, tedy konkrétně na hlavní program, obslužné a užitkové rutiny. Návrh takového jádra je nejjednodušší a výsledek má předpoklady k rychlému běhu. Nevýhodou tohoto návrhu je velká paměťová náročnost celku

2.3.1 Mapování paměti jádra

Paměťový prostor jádra operačního systému je možné do virtuálního paměťového prostoru zobrazit několika způsoby:

- **Nad uživatelský prostor** – virtuální paměťový prostor procesu je rozdělen na dvě části. Jedna je určena procesu samotnému a zbytek jádru. Jádro je nejčastěji umístěno v posledním gigabytu paměti. Stránkovací tabulky všech procesů obsahují nakopírované hodnoty stránkovací tabulky jádra. Toto rozložení je pro útočníka vhodnější.
- **Oddělený paměťový prostor** – jádro disponuje vlastním virtuálním prostorem, má tedy k dispozici kompletní rozsah adres. Dle podpory poskytovanou ze strany hardware má tato metoda vyšší předpoklady v oblasti bezpečnosti.

3 Techniky užívané útočníkem

Některým odborníkům může přijít diskuze o útočníkových metodách zbytečná či nebezpečná. Kvalitní obrana ovšem vyžaduje, aby administrátor/programátor stojící na pomyslné barikádě znal útočníkovy zbraně pro sestavení vhodného způsobu reakce na jeho postupy. Podrobnější informace je možné nalézt v odborné literatuře zabývající se tématikou využití programových chyb [7] [8] [9].

Kapitola se zabývá především problematikou tvorby exploitu (viz. slovník). Důvod je víceméně jasný, jedná se o denní chléb hackera. Většina látky je probírána v rámci uživatelského prostoru, případné odlišnosti od prostoru jádra budou zmíněny. Veškeré ukázky byly ozkoušeny na systému Microsoft Windows 7 x64 a distribuci GNU/Linux Arch Linux i686 bez zapnuté optimalizace. Zapnutá optimalizace vyžaduje drobné obměny v kódu ukázek.

3.1 Fyzické napadení systému

Může se zdát, že fyzické zabezpečení chráněných prostředků výpočetní techniky sem nepatří, opak je ovšem pravdou. Má-li útočník fyzický přístup k vytypovanému systému, může nad ním získat naprostou kontrolu, získat cenná data, nebo způsobit výpadek poskytovaných služeb (DoS).

Vhodnou prevencí proti fyzickému napadení je používání bezpečných hesel, šifrování dat, zabezpečení proti zavedení útočníkova systému pomocí vyměnitelných médií, případně hesla pro BIOS (nemá-li útočník možnost obnovit tovární nastavení). Nutno podotknout, že ani šifrování není vždy neprolomitelnou barierou (např. ColdBoot útok).

3.2 Lokální napadení systému

Exploit je termín označující program zneužívající chyb programátora obsažených v cílové aplikaci. Většinou se útočí na chybu vedoucí k porušením obsahu paměti. Cílem těchto technik je donucení programu k vykonávání činností autorem nezamýšleným způsobem. Výsledkem takovéhoto snažení je převzetí kontroly nad činností programu, které může vést k ovládnutí celého systému.

3.2.1 Poškození obsahu paměti

Standardně neexistuje žádný prostředek kontrolující velikost dat ukládaných do již alokované proměnné. Situaci, kdy dojde k zápisu mimo meze dané struktury (pole) nazý-

váme termínem **přetečení zásobníku** (buffer). Během zápisu mimo meze dojde k přepsání paměťových pozic nacházejících se za postiženou strukturou.

Zásobníkem rozumíme část paměti programu, kde jsou uchovávány staticky alokované proměnné. V užším měřítku je možné zásobník chápat jako datovou strukturu pole. Oba případy ovšem představují souvislou paměťovou oblast s danými hranicemi.

Dále následuje představení principu přetečení zásobníku pomocí ukázkového programu preteceni_zasobnik.c. Řetězec vkládaný do pole znaků přepisuje obsah celočíselné proměnné umístěné za ním. Pole je plněno zadaným textem s délkou větší nežli rozsah pole. Výsledek takovéto činnosti je dán několika faktory. Zaprvé záleží na rozdílu velikosti pole a vkládaného řetězce, dále na obsazené paměti nacházející se v paměti za polem. Při přesání příliš rozsáhlé paměťové oblasti dojde k pádu programu, zapříčiněného porušením paměti (SEGFAULT). V příkladu je vkládaný text součástí zdrojového kódu, běžněji je chybu možné nalézt při použití uživatelského vstupu, případně při práci s textovými řetězci. Uživatelský vstup je obzvláště nebezpečný, a to především díky minimální možnosti kontrolovat chování uživatele.

preteceni_zasobnik.c

```
#include <stdio.h>
#include <string.h>

int main(void) {
   char buffer[1];
   int canary = 0;

   printf("[PRED] kanarek: 0x%X\n", canary);
   strcpy(buffer, "Ahoj");
   printf("[PO] kanarek: 0x%X\n", canary);

   return 0;
}
```

Obrázek 3.1: Výstup programu – přetečení zásobníku

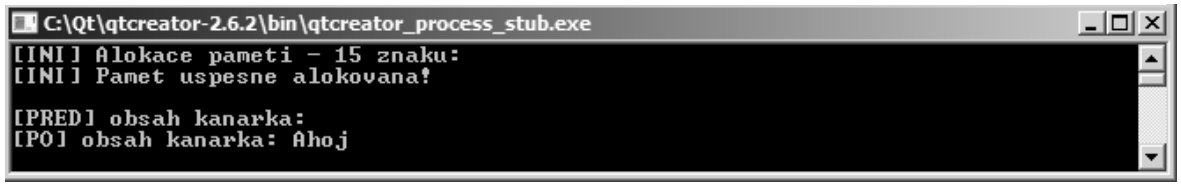
Na technice přetečení zásobníku je založeno velké množství zranitelností, některé metody používané útočníkem mohou mít jiné označení, přesto velice často vycházejí z této základní techniky. Případně je možné využít základní princip pro změny obsahu paměti ve prospěch útočníka.

3.2.2 Poškození obsahu dynamické paměti

Běžně napsaný program obsahuje veškeré potřebné proměnné již v době kompilace, nemá možnost pružné reakce na vnější podněty. Takovou pružnost zavádí využití dynamicky alokované paměti, kdy program obsazuje (alokuje) paměť až tehdy, potřebuje-li jí. Paměťová oblast obsahující paměť alokovanou tímto způsobem je běžně označována jako halda, v anglické terminologii „heap“. Předchozí metodiku je možné rozšířit pro použití na haldě, či v jiné paměťové oblasti.

preteceni_halda.c

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#define TEXT "AhojAhojAhojAhojAhojAhojAhojAhojAhojAhojAhojAhoj"
"AhojAhojAhoj"

int main(void) {
  const long kapacita = 10;
  char *buffer, *canary;

  printf("[INI] Alokace pameti - %2.0f znaku:\n", kapacita * 1.5);
  buffer = (char*) calloc(kapacita + 1, sizeof(char));
  canary = (char*) calloc(kapacita * 0.5 + 1, sizeof(char));

  if (buffer == NULL || canary == NULL) {
    printf("[ERR] Neuspesna alokace pameti!\n");
    return 1;
  }

  printf("[INI] Pamet uspesne alokovana!\n\n");
  printf("[PRED] obsah kanarka: %s\n", canary);
  strcpy(buffer, TEXT);
  printf("[PO] obsah kanarka: %s\n", canary);

  return 0;
}
```



Obrázek 3.2: Výstup programu – přetečení haldy

Výše zobrazený příklad přetečení dynamicky alokovaných dat pracuje na totožném principu jako přetečení zásobníku. Obrázek 3.2 zobrazuje výstup tohoto příkladu.

3.2.3 Ukazatele

Ukazatel je proměnná obsahující symbolický odkaz na odlišné paměťové umístnění, cílem takového odkazu může být umístění proměnných, struktur, či funkcí. Předchozími

technikami je možné stejně jako data, změnit také obsah ukazatelů a ovlivnit tak například tok programu. Z předchozího vyplývá, že poškození obsahu ukazatele je většinou důsledkem přetečení paměti. Neinicializovaný ukazatel obsahuje nespecifikovaná (náhodná) data, je-li deklarován jako statický, obsahuje nulovou (NULL) hodnotu. Kromě nekorektního ukazatele je tedy možné zneužít ukazatel obsahující nulovou, případně s neinicializovanou hodnotou. Tato metodika je založena na použití nevalidního ukazatele.

Operační systém využívající jádro umístěné nad uživatelským prostorem může nevalidovaný ukazatel vést do uživatelského prostoru, naneštěstí je většinou moderních operačních systémů prováděna kontrola ukazatelů na schodu s nulovou hodnotou, případně je provedena kontrola povoleného rozsahu adres.

Obrana proti většině technik pozměnění obsahu paměti je velice jednoduchá. Používá se metodika tzv. „kanárků“, název je odvozen od kanárků používaných horníky, jde o hodnotu uloženou v paměti za sledovaným objektem a její známá hodnota kontrolována po operacích spojených s kontrolovaným objektem. Staticky definovanou hodnotu může útočník odhalit a učinit protiopatření, je tedy bezpečnější použít náhodné hodnoty.

3.2.4 Formátovací řetězce

Některé funkce poskytují možnost formátovat výstupní text pomocí speciálních sekvencí znaků označených jako formátovací řetězce. Jedná se o výkonný nástroj práce s textem, či interakci s uživatelem. Stinnou stránkou je možnost zneužití některých speciálních formátovacích řetězců k vypisování, či změně obsahu paměti.

Parametr	Vstup	Výstup
%d, %i	Celé číslo	Dekadická hodnota
%s	Odkaz na řetězec	Řetězec
%p	Ukazatel	Adresa ukazatele
%n	Ukazatel na celé číslo	Nic není tisknuto, ukládá počet doposud zapsaných znaků

Tabulka 3.1: Seznam důležitých formátovacích řetězců

Obsahuje-li uživatelem zadávaný text formátovací znaky, které nebyly programem filtrovány a je použit přímý tisk bez využití formátovacího řetězce „%s“, může se aplikace začít chovat neočekávaně. Jak uvádí Jon Erickson [7], pomocí formátovacího řetězce

„%s“ je útočník schopen číst obsah libovolné paměťové adresy, podobně lze pomocí řetězce „%n“ zapisovat na libovolnou paměťovou adresu. Jelikož se formátovací řetězec nachází na vrcholu zásobníku, je možné požadovanou adresu vložit přímo do vkládaného řetězce.

3.2.5 Shellcode

Označení shellcode (viz. slovník) vychází z primárního účelu této techniky, tedy snahy o zpřístupnění příkazového procesoru privilegovaného uživatele. Také se velice často označuje jako op-code, což je odvozené od využívání operačních kódů (strojový kód) mikroprocesoru. Například instrukci „**int 80h**“ lze zapsat řetězcem ve tvaru „**\xcd\x80**“.

Nyní je vhodné uvést jakým způsobem vytvořený shellcode využít. Obecně je shellcode uložen do paměti a následně je program donucen k vykonávání jeho kódu. Zavedení do paměti je možné například podle výše popsaných technik narušení obsahu paměti, či umístěním do procesu vlastněného procesu. Dále je potřeba donutit proces vykonat kód představovaný shellcode, toho je nejčastěji přepsáním ukazatele na adresu odkazující na shellcode samotný, případně jeho „přistávací“ oblasti obsahující NOP instrukce.

Shellcode je samostatný program přebírající řízení napadené aplikace. Nejčastěji se jedná o program napsaný v assembleru. Samotná reprezentace je uchovávána buď uvnitř binárního souboru, nebo textovém řetězci. Existuje několik základních typů shellcode dle Dobšíčka a Ballnera [8]:

- **Generický** – základní typ shellcode, umožňuje pouze spustit příkazový procesor,
- **Alfanumerický** – op-code je možné zapsat pomocí tisknutelných znaků ASCII, obrana je stižená díky podobě běžného textu, hlubší pojednání nabízí magazín Phrack [10],
- **Schopný běhu na více operačních systémech**,
- **Schopný běhu na více platformách**,
- **Polymorfní** – Jeden vstup generuje odlišné binární podoby, což ztěžuje detekci.

Bajt s nulovou hodnotou má v řetězci speciální význam, konkrétně signalizuje konec řetězce. Většina funkcí skončí během nalezení prvního bytu null, některé funkce při známé délce řetězce naopak bajt null přeskakují a dolní na konec řetězce ukončovací

znak sami. Při takové úpravě by se pracně vytvořený shellcode stal nefunkčním, existuje tedy několik technik k odstranění těchto hodnot z posloupnosti instrukcí.

První metoda používá instrukcí pracujících s menším datovým rozsahem, není tedy nutné doplnit rozsah nevýznamnými nulovými hodnotami. Podobně je možné využít menší části registru, například pouze spodních 8 bitů, v takovém případě je ovšem nutné použitý registr nulovat.

Dále je možné použít dvojkový doplněk, tedy využití skoků zpět (záporná čísla). Bohužel dvojkový doplněk obsahuje velice často bajty s hodnotou 0xFF, které je možné snadno detekovat. Další sekvence, která se velice často do shellcode vkládá je instrukce se strojovým kódem 0x90, označená jako NOP (no operation), tedy instrukce ovlivňující pouze čítač instrukcí. Také existuje možnost nahrazení jinou instrukcí, která nemění obsah registrů.

3.3 Vzdálené napadení systému

Napadení vzdáleného systému může být velice podobné zneužití systému, k němuž útočník vlastní přístup. Cílem vzdáleného útoku nejčastěji bývá získání přístupu do systému, případně znemožnění využití služeb tímto systémem poskytovaných (DoS).

3.3.1 Únos spojení

Technika v angličtině označovaná jako „Man in the middle“ označuje techniku, během níž se útočník stane prostředníkem mezi komunikujícími stranami. Jedná se o techniku s vysokým stupněm úspěšnosti a nízkým stupněm o pravděpodobnosti odhalení během probíhajícího útoku. Na této technice je základem velkého množství útoků.

3.3.2 Syn-flood útok

Útok spočívá v zahlcení systému požadavky SYN o navázání spojení. Záludnost toho útoku spočívá v malém nárůstu datového toku. Útočník se snaží navázat v krátkém časovém úseku maximální počet nových spojení, která ovšem nejsou nikdy dokončena. Systém sice většinou čeká na navázání jistý časový interval. Ovšem velké množství dotazů během krátké doby může systém zahltit. Základní ideou je zneužití normy vyžadující připravení systému na následující komunikaci, jsou tedy vyčleněny systémové prostředky. Cílem je vyčlenit takové množství systémových prostředků, aby systém havarovat.

Ačkoliv proti většině útoků typu odepření služeb je účinná obrana obtížná, je možné se proti zahlcení pakety SYN celkem účinně bránit, nedojde-li provozem k zahlcení linky. Obrana je možná například pomocí syn-cookies, omezení množství navázaných spojení, případně vystavění systémových prostředků po potvrzení komunikace.

3.3.3 Ping smrti

Jedná se o útok zneužívající důsledné dodržování normy a spoléhání na dodržování všemi účastníky komunikace. Norma protokolu ICMP specifikuje maximální velikost datové sekce paketu (64kB). Během útoku je vytvořen paket s datovou částí větší než normou stanovené maximum, což má za následek nepředvídatelné chování systému. Dnes je většina systémů proti tomuto útoku imunní, naneštěstí implementace některých nových protokolů trpí velmi podobnými nedostatky.

3.3.4 Slza

Jedná se o útok typu odmítnutí služeb, který je velice podobný pingu smrti. Využívá chyby ve zpětném sestavování paketu IP. Paket typu „slza“ zasílá paket obsahující překrývající se offsety. Takové offsety mohou být některými implementacemi neočekávány, což vede k neočekávanému chování, nejčastěji pádu systému. [7]

3.4 Útok na jádro systému

Techniky napadení jádra operačního systému jsou v podstatě totožné s technikami popsanými dříve. Rozdílem v privilegovaném prostoru jádra je jeho komplexnost a nutnost běhu systému jako takového. Pokud útočník neúspěšnými pokusy neustále restartuje nějakou aplikaci, nemusí si nikdo ničeho všimnout, provede-li ovšem útok na jádro neopatrně, znemožní tím běh celého systému. Je nutné tedy pracovat mnohem pečlivěji a opatrněji.

Během diskuzí o bezpečnosti je důležité si uvědomit, že žádné zabezpečení nemůže být dokonalé, a také, že operační systém včetně jeho jádra je pouze sofistikovaný programový kód tvořený lidmi. Komplexnost jádra operačního systému je výhodná pro útočníka, jelikož komplexní systémy se mnohem hůře udržují. Většina ochranných mechanismů implementovaná uvnitř jádra chrání pouze uživatelský prostor, nikoliv jádro samotné. Další faktor, který podkopává snahy o zabezpečení jádra je vysoká výkonová daň za dostatečně kvalitní ochranné prvky. Uživatelé budou ve většině případů preferovat vyšší výkon před bezpečností.

4 Chyby programátora

Každá zranitelnost, kterou útočníci využívají je v podstatě chyba v návrhu systému, případně chyba způsobená špatně napsaným kódem aplikace. Jelikož veškeré fáze vývoje aplikací jsou v lidské režii, a jak známo lidé nejsou neomylní, není možné se vyhnout všem chybám. Není v ničích silách promyslet naprosto veškeré možné scénáře, které se mohou vyskytnout.

Následující kapitola se snaží popsat možné obrany proti dříve předneseným technikám útoků na chyby v programu. Hlavním cílem je zlepšit povědomí programátorů o možných dopadech jednoduchých chyb na bezpečnost, či funkčnost výsledného produktu. Dále představuje techniky vedoucích k zlepšení stylu programování, dodatečné kontrole zdrojových souborů.

Jazyk C je nízkoúrovňový nástroj s minimální kontrolou práce programátora, tento přístup programátora nesvazuje, takováto svoboda je vykoupena obrovskou měrou zodpovědnosti. Programátor pracující s jazykem C si musí být vědom, co dělá a za jakým účelem. V opačném případě může tento mocný nástroj používat nevhodným způsobem, což velice často vede k chybám popsaným v této kapitole.

Jazyk C je označován jako systémový jazyk, protože je v něm možné programovat jádro operačního systému i všechny ostatní komponenty operačního systému. Aby bylo možné takto citlivé operace vykonávat s minimálním dopadem na rychlost celého systému, nechává jazyk většinu kontrolních úkonů přímo na programátorovi. Mezi takové kontroly patří například hlídání hranic polí, validita ukazatelů a podobné. Zároveň se jedná o oblast, ve které mají programátoři tendenci nejvíce chybovat.

4.1 Numerické chyby

Často se vyskytujícími a velice těžko odhalitelnými chybami jsou numerické chyby. Nejčastěji je možné je nalézt při práci s polem, či textovými řetězci. Primární oblastí výskytu takové chyby jsou podmínky a cykly související s výše zmíněným. Jedná se o chybu, kdy jsou hranice pole či platnosti podmínky nevhodně zapsány prostřednictvím daného jazyka.

Následně může aplikace fungovat naprosto správným způsobem, dokud nedojde na onu podmínku. Taková chyba se velice špatně hledá a může mít nedozírné bezpečnostní

následky. Podobné chyby se nevyhnuli ani aplikacím zakládajících se na vysoké bezpečnosti, jako je OpenSSH či OpenBSD.

4.2 Potenciálně nebezpečné knihovní funkce

Každý řádek kódu je při nesprávném použití potenciálně nebezpečný. Knihovní funkce se z programátorova pohledu chovají jako černé skříňky, což eventuelní nebezpečí může umocnit. Dalším problémem je rozdílná implementace a chování na různých platformách, z toho důvodu je důležité seznámit se s chováním používaných funkcí prostřednictvím dokumentace.

4.2.1 Práce s řetězci

Textový řetězec je ze své podstaty polem znaků, což samo o sobě přináší úskalí spojená s přetékáním paměti. Frekventovanost problému umocňuje nula indikující konec řetězce. Velice často je řetězec přepsán do posledního znaku, koncová nula je tedy přepsána. Během další práce s takto upraveným řetězcem dojde ke čtení až k následujícímu nulovému bajtu v paměti.

Následuje seznam některých funkcí, jejich potenciálního nebezpečí a bezpečnějších alternativ. Nutné je upozornit, že ani navržené náhradní metody nemusejí být za všech okolností bezpečnější. Bezpečnost vždy závisí na konkrétní situaci, programátor tedy musí přesně vědět, co dělá. Z těchto důvodů není následující seznam vyčerpávající a udává pouze nejčastější problémy. [11]

Funkce	Možná náhrada	Popis
gets()	fgets()	Vhodné s dynamickou alokací paměti.
strcpy()	strncpy(), strlcpy()	strncpy() nevkládá na konec řetězce nulu! strlcpy() dostupná pouze na BSD systémech.
strcat(), strcmp()	strncat(), strncmp()	
printf(),sprintf(), fprintf()	snprintf(), vsprintf()	S uživatelem zadaným vstupem pracovat pomocí řetězce „%s“.

Tabulka 4.1: Potenciálně nebezpečné funkce pracující s textovými řetězci

4.2.2 Spouštění systémových příkazů

Spouštění příkazů systému z aplikace prostřednictvím funkce **system()** je potenciálně nebezpečné, obzvláště, je-li vykonávaný příkaz zadávaný uživatelem. Má-li aplikace

snažící se vykonat systémový příkaz vysokou úroveň oprávnění, má k dispozici téměř neomezené pole působnosti. Program disponující takovouto vlastností může útočník zneužít k libovolné škodlivé činnosti. Takovéto činnosti je nejlepší se vyhnout, v opačném případě je vhodné kontrolovat vkládaný příkaz, zadávat plnou cestu, spouštět prostřednictvím takové aplikace, která se bezprostředně po spuštění zbaví svých práv.

4.3 Dočasné soubory

Dočasné soubory by nikdy neměli mít pevně zadané jméno, takový soubor je snadné napadnout například vytvořením odkazu na jiný soubor. Vhodným řešením tohoto problému je generování takového souboru pomocí funkce **FILE *tmpfile()**, naneštěstí díky chybné implementaci v systému UNIX V není doporučeno používat tuto metodu na systémech založených na tomto systému. Na systémech založených na UNIX V je doporučeno používat funkci **mkstemp()** k vygenerování názvu souboru. [8]

4.4 Chyby souběhu

Chyby souběhu, označované také jako **časově závislé chyby** (race conditions), patří mezi nejhůře odhalitelné chyby. Záludnost tkví především v nepředvídatelném projevu následků chyb, chyba se může projevit pouze za určitých podmínek a program může ve většině případů pracovat zcela správně. Zjednodušeně můžeme říci, že chyba se projeví v závislosti na vytížení systému a přidělených prostředcích. Některé chyby tohoto druhu se mohou projevovat pouze na systémech osazených současně více mikroprocesory (jádry mikroprocesoru), takové systémy označujeme termínem multiprocesorové (SMP/AMP).

Chyby samotné spočívají v situaci, kdy se několik procesů (vláken) snaží přistupovat ke sdíleným prostředkům vyžadujících výlučný přístup. Zamezení chybám tohoto druhu je možné pomocí prostředků mezi procesové komunikace (IPC), nejčastěji se setkáme s takzvaným „zamykáním“, kdy je po nutnou dobu zajištěn výlučný přístup jedinému procesu (případně většímu množství procesů).

Využívá-li útočník chyb souběhu, nemůže většinou vkládat vlastní kód či měnit tok běhu aplikace, může ovšem získat přístup k alokovaným prostředkům napadeného sys-

tému. Ve většině případů mají být takové prostředky chráněny a přístup z vnější je nevhodný až nebezpečný.

4.5 Odstranění chyb

4.5.1 Statická kontrola

Mezi základní typ statické kontroly je možné započítat kontrolu prováděnou překladačem v průběhu kompilace. Během kompilace je vhodné zapnout pravidla striktní kontroly, tato vlastnost je závislá na použitém kompilátoru. Dále je možné využít aplikace dodatečně kontrolující zdrojové kódy na výskyt častých chyb. Konkrétní ukázky jsou obsaženy v rámci případových studií.

4.5.2 Dynamická kontrola

Dynamickou kontrolou rozumějme kontrolu prováděnou během vykonávání programového kódu. Stinnou stránkou dynamické kontroly je možné znemožnit kompilaci, potažmo běh některých specifických aplikací a snížení výkonnosti systému. Dynamickou kontrolu je možné aplikovat upravením kódu několika částí systému:

- kompilátor,
- standardních knihovny,
- jádro operačního systému.

Potřebné úpravy je možné provést na více úrovních systému současně, taková úprava ovšem degraduje výkon systému ve větší míře, nežli malé množství vhodně vybraných aktualizací. Aplikací konkrétních úprav je možné získat výhody jako kontroly mezí polí, automatické vkládání kontrolních hodnot do operační paměti a podobně.

5 Případová studie – Linux

Název „Linux“ označuje pouze jádro, na kterém je možné založit vlastní operační systém. Takzvané „distribuce“ staví kolem samotného jádra zbytek operačního systému, nejčastěji je k tomuto účelu využito nástrojů rodiny GNU, takový operační systém se poté nazývá GNU/Linux. Jedná se o jádro založené na filozofii systému UNIX.

5.1 Historie

V devadesátých letech cítil finský student Linus Torvalds potřebu operačního systému unixového typu, který by byl volně dostupný a schopný běhu na architektuře IA-32. Původní návrh vycházel z operačního systému Minix, jehož autorem je profesor Andrew S. Tanenbaum. [12]

Během třetího června 1991 odeslal Linus Torvalds do diskusní skupiny „comp.os.minix“ první zmínku o plánovaném projektu, k němuž potřeboval definici standardu POSIX. Následně byla 17. září téže roku publikována první verze (0.01) jádra nazvaného „Linux“. Následně se projekt svobodného jádra stával oblíbenějším a k jeho vývoji se přidávalo větší množství vývojářů.

5.2 Implementace

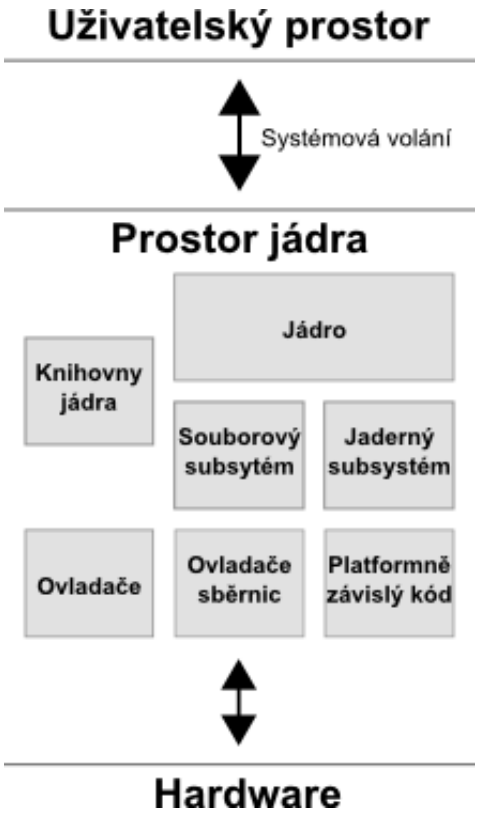
Znalost implementace jádra operačního systému je vhodná během vývoje úzce spolupracujících aplikací, ovladačů, modulů jádra či programů zneužívajících možné nedostatky. Implementační detaily byly převzaty z publikace zabývající se popisem jádra Linuxu [4].

5.2.1 Postupný vývoj implementace

Během doby vývoje Linuxu doznávala architektura vývoj, který umožnil jednodušší implementaci nových ovladačů zařízení a přenositelnosti na jiné architektury. Jedním z primárních cílů inovace je zvýšení modularity jádra pro jeho větší bezpečnost a jednodušší implementaci nových funkcionalit. Se zvyšováním modularity přímo souvisí snahy o přesun architektury blíže mikrojádru a zvýšení úrovně abstrakce. Jedním z hlavních cílů vývojářů je nerozbíjet změnami uživatelské aplikace, takovýto postoj ovšem občas brzdí vývoj jádra samotného, případně jsou vývojáři nuceni problém obcházet, což může vést k chybám.

5.2.2 Architektura jádra

Architekturou jádra rozumíme vnitřní stavbu jádra, komunikaci mezi jednotlivými částmi jádra a jakým způsobem komunikuje s okolím. Linux je v podstatě jádro monolitického typu, jedná se tedy o kompaktní celek spadající do souvislé paměťové oblasti. I když jádro není mikrojádrem, obsahuje řadu rozhraní, které umožňují implementovat podobné vlastnosti. Díky tomu je možné provozovat důležitou funkcionalitu v uživatelském prostoru, například souborové systémy prostřednictvím FUSE.



Obrázek 5.1: Architektura Linuxu

5.2.3 Moduly jádra

Modulární návrh umožňuje minimalizovat velikost jádra zavedeného do operační paměti při zachování rozsáhlé funkčnosti a rozsahu podporovaného hardware. Získání jádra specializovaného pro použití na konkrétním stroji je možné zahrnutím nezbytných modulů přímo do jádra.

Informace o každém modulu ukládá jádro do datové struktury nazvané „module“. Datová struktura je implementována formou obousměrně zřetězeného seznamu, každý prvek tedy odkazuje na následující a předchozí prvek v seznamu modulů. Dále jsou ukládány informace o stavu modulu (běžící, načítaný, uvolňovaný), unikátní název modulu a další specifické informace. Obsah struktury se mění v závislosti na konfiguraci a verzi jádra.

Modularita implementovaná Linuxem se vyznačuje několika přednostmi:

- **jednotné rozhraní**,
- **integrace rozhraní modulů** – ostatní moduly mohou pracovat s rozhraním jiného modulu, jako by bylo součástí jádra,
- **možnost integrace modulů** – během kompilace je možné modul integrovat do jádra, nebo jej používat jako připojitelný modul,
- **minimální režie práce s moduly**,
- **automatické zavádění modulů**.

5.2.4 Mapování paměti jádra

Na architektuře IA-32 využívá Linux mapování jádra v posledním gigabytu paměti, v případě 64bitového systému je rozložení paměti odlišné. Díky velkému množství dostupné virtuální paměti je možné vytvořit složitější uspořádání paměti. K vhodnému využití paměti je stránkovací tabulka implementována pomocí čtyř úrovní (je možné používat tříúrovňový model), na 32bitových systémech je používán pouze dvouúrovňový model stránkování.

Velikost	Obsah
128 TB	uživatelský prostor
64 TB	zobrazení veškeré fyzické paměti
32 TB	alokované stránky a vstupně/výstupní paměti (vmalloc/iorema)
1 TB	virtuální paměť
512 MB	programový kód jádra
1536 MB	moduly jádra

Tabulka 5.1: Linux – virtuální adresní prostor architektury AMD64

5.2.5 Správa paměti

Správa paměti je stěžejní vlastností operačního systému, zajišťující dostatek paměťového prostoru všem částem systému i přes omezené fyzicky dostupné zdroje. Chování správy paměti je možné během kompilace přizpůsobit potřebám systému, na kterém bude nasazeno. Některé vlastnosti je možné přizpůsobit během běhu systému pomocí souborů obsažených v adresáři **/proc/sys/vm**.

Ideální systém umožňující nastavení oprávnění nejmenší adresovatelné paměťové jednotce a osvobozen od veškerých omezení paměťových rozsahů ze strany hardware umožňuje použít libovolnou část paměti k libovolnému účelu. Takový idealizovaný systém ovšem neexistuje a v praxi se setkáváme s omezením ze strany hardware, kterému je nutné se podřídit. Pro takové účely dochází k rozdělení

Zóna	Velikost [MB]	Popis
ZONE_DMA	16 MB	DMA zóna při praci přes sběrnici ISA (adresa šířky 20 bitů)
ZONE_NORMAL	880 MB	běžné použití
ZONE_HIGHMEM	zbytek paměti nad 896 MB (je-li osazeno více fyzické paměti)	horní paměť

Tabulka 5.2: IA-32 – zóny paměti

Stránka je úsek paměti používaný během paměti spravované stránkováním, zároveň se jedná o nejmenší přidělitelný úsek paměti, je sice možné alokovat menší množství paměti, vnitřně ovšem tyto metody pracují s celými stránkami. Uvnitř jádra se o alokaci paměti stará tzv. „**zónový alokátor**“. Označení zónový vychází z faktu, že pro přidělení paměti v rámci jednotlivých zón paměti je používán odlišný postup. Jedním z použitých algoritmů je takzvaný „**buddy algoritmus**“:

1. Pokus o přidělení bloku požadované velikosti, je-li nalezen, je přidělen a postup je ukončen.
2. Po neúspěchu následuje pokus o přidělení bloku dvojnásobné velikosti, pokud není opět nenalezen, je druhý krok znovu opakován.
3. Nalezený blok je rozdělen tak, aby jedna část odpovídala požadované velikost, zbytek je možné použít během dalšího přidělování.

Některé funkce operačního systému nemohou využít služeb blokujícího volání během požadavku o přidělení potřebné paměti, v takovém případě je využito mechanismu rezervovaných stránek. Takovým případem mohou být například události obsluhované během přerušení. Výchozí velikost paměti určené pro potřeby rezervovaných stránek je rovna druhé odmocnině množství přímo adresovatelné paměti.

Ne vždy je možné alokovat paměť v souvislých blocích, v takovém případě je nutné využít nesouvislého přidělování paměti. Proces jako takový spočívá v alokaci několika

částí pamětí, které jsou následně mapovány do adresního prostoru zdánlivě souvisle. Takový proces je pro kód pracující s takovou pamětí naprosto transparentní. Tento mechanizmus je využíván převážně při velké fragmentaci paměti, případně při nutnosti přidělení rozsáhlého paměťového prostoru.

Velice často se systém může dostat do stavu, kdy nastane nedostatek volné paměti. Mohou nastat dvě situace, zaprvé není možné problém řešit jiným způsobem, nežli pracovat na úkor některých procesů, zadruhé a častěji je situaci možné řešit nekonfliktním způsobem. Jako nejlepší možné řešení se jeví předcházení takovýmto stavům vhodnými metodami prevence. Mezi hlavní metody řešení nedostatku paměti patří odkládání do odkládací oblasti, použití reverzního mapování k nalezení sdílené paměti, zavedení mechanizmů uvolnění paměti a v kritických případech zavedení mechanizmu násilného ukončení procesů (OOM killer).

5.2.6 Procesy a vlákna

Linuxová implementace procesů a vláken pohled na problematiku komplikuje. Díky minimálním rozdílům mezi vlákny a procesy je v podstatě možné tvrdit, že jsou tyto dva pojmy zaměnitelné. Z tohoto důvodu zavádí odborné publikace pojem úloha, některé publikace spolu se zdrojovými kódy jádra bohužel pojem úloha a proces zaměňují. Proces jako takový je v podstatě sada (skupina) vláken a vlákno samotné vlastní velkou část znaků procesu.

Všechny úlohy (procesy a vlákna) jsou popsány pomocí datové struktury „**task_struct**“ a obsahují informace o stavu, příznaky, vztahy mezi úlohami a další datové struktury definující konkrétní úlohu. Příznaky obsahují informace informující, zda je konkrétní úloha právě spouštěna, ukončována, alokuje paměť, využívá randomizace adresního prostoru a podobně. Implementace procesů a vláken obsahuje následující stavy obsažené v proměnné „**state**“ popisné struktury:

- **běžící úloha** – úloha je vykonávána či odložená,
- **přerušitelně pozastavená úloha** – úloha je uspána s možností přerušení a čeká na splnění podmínky, případně je možné jí probudit pomocí signálu, přerušení či uvolněním požadovaných prostředků,
- **nepřerušitelně pozastavená úloha** – obdobně jako předchozí, výjimku tvoří probuzení signálem, které je zakázáno,

- **pozastavená úloha** – úloha je zastavena, většinou pozastavena pomocí signálu,
- **úloha sledovaná** – úloha je zastavena pomocí bodu přerušení činnosti (break-point) během procesu ladění (debugging),
- **mrtvá úloha** – vykonávání bylo ukončeno, dojde k poslednímu přidělení procesorového času (naplánování) k odstranění referencí na úlohu.

5.2.7 Úroveň ladění jádra

Během vývoje jednotlivých částí jádra je díky jeho komplexnosti velmi obtížné provádět některé ladící úkony, nejen k tomuto účelu disponuje jádro možností informovat vývojáře o akutních stavech. Pro větší přehlednost během produkčního nasazení je možné tyto informace podávat v několika úrovních akutnosti. Je-li jádro vykonáváno v běžném režimu, nebude zobrazovat například ladící zprávy určené vývojářům. Existují sice ladící nástroje specializované pro použití při vývoji Linuxu, nejedná se ovšem vždy o nejefektivnější řešení.

- **KERN_EMERGE** – systém je v nepoužitelném stavu,
- **KERN_ALERT** – je nutné neprodleně vykonat protiopatření,
- **KERN_CRIT** – kritický stav,
- **KERN_ERR** – chybový stav,
- **KERN_WARNINR** – varování,
- **KERN_NOTICE** – jádro se nachází v normální kondici, jedná se ovšem o významný stav (zprávu),
- **KERN_INFO** – informativní stav,
- **KERN_DEBUG** – zprávy režimu ladění

5.3 Překlenutí ochrany vykonávání dat

Na architektuře IA-32 bez 64bitového rozšíření je možné ochranu proti vykonávání dat implementovanou pomocí PaX obejít pomocí techniky zvané „**návrat do libc**“ [13], případně **rozšířenou variantu** [14] této techniky. Základní varianta většinou spočívá v náhradě skoku do funkce zapsané uvnitř dat za skok do existující funkce. V takovém případě je ochrana proti vykonávání dat bezmocná. Rozšířená verze přidává možnost

volání funkce s požadovanými parametry (občas nemusí jít o problém) a možnost předávat parametry obsahující nulové hodnoty (bajt NULL).

Architektura AMD64 neumožňuje použití výše popsaných technik díky nutnosti předávat parametry během volání funkce prostřednictvím registrů, technika návratu ovšem vyžaduje vkládání parametrů na zásobník. Pokusíme-li se vložit parametry funkce **system()** do registru RDI, funkce spadne nebo se pokusí vykonat nesmyslný (náhodný) příkaz. [15]

V rámci publikace zabývající se možností obejití implementace ochrany NX bitu používané 64bitovými mikroprocesory Intel a kompatibilními byla vyvinuta metoda nazvaná „**vypůjčení částí kódu**“ (borrowed code chunks). Stejně jako techniky používané v rámci 32bitových systémů, je i tato technika založena na principu přetečení zásobníku. Na tomto místě je vhodné si uvědomit, že jakmile má útočník možnost zapisovat libovolný obsah na jakékoliv paměťové adresy, může tedy převést útok založený na haldě či formátovacích řetězcích na útok postavený nad zásobníkem, jelikož zásobník spadá do oblasti útočníkovi dostupných adres.

Výše popsané techniky jsou způsobem jak ochranu proti vykonávání dat obejít, nikoliv prolomit. Ochrana implementována programově i v hardware je dostatečně kvalitní a znesnadňuje útočníkovi práci. Samotné techniky spadají mezi obtížněji implementovatelné, ovšem ne nemožné.

Autor materiálů zabývajícího se prolomením na architektuře AMD64 ve své práci [15] demonstruje exploit napojený na síťové porty, pomocí kterých může útočník pracovat s lokálním příkazovým procesorem (shell). Exploit je napsán takovým způsobem, aby pracoval pouze s útočníkem lokálně připojeným, především z důvodu ztížení zneužití příkladu. Konkrétně tento exploit spouští funkci „**system()**“ s parametry udávajícími lokální vstupní a výstupní síťový port. Příkaz je následujícího tvaru „**system(¨sh </dev/tcp/127.0.0.1/3128>/dev/tcp/127.0.0.1/8080¨)**“.

5.4 Vývoj modulů jádra a ovladačů

Motivací pro tvorbu jaderných modulů může být například tvorba ovladačů zařízení, souborového systému, plánovače blokových operací či procesů, bezpečnostních mechanismů a v neposlední řadě také škodlivé moduly.

prvnimodul.c

```
#include <linux/module.h>
#include <linux/init.h>

MODULE_AUTHOR("RALIS Cenek <rc@domena.cz>");
MODULE_DESCRIPION("Prvni modul");
MODULE_LICENSE("GPL");

static int _init initialization_module(void) {
    printk(KERN_INFO "prvni_modul: modul inicializovan\n");
    return 0;
}

static void _exit clean_module(void) {
    printk(KERN_INFO "prvni_modul: modul odebran\n");
}

module_init(initialization_module);
module_exit(clean_module);
```

Základní modul musí implementovat inicializační a čistící rutiny, které jsou předány příslušným makrům. Dobrým zvykem je označování rutin klíčovým slovem „static“, ačkoliv toto není u jader řady 2.6 a vyšších explicitně vyžadováno. Dalším dobrým zvykem je uvedení základních informací o modulu, jakými je jednoduchý popisek účelu modulu, autor a kontakt, licence, pod níž je modul uvolněn a případně také verze modulu. Pokud chceme modul zkompilovat a vyzkoušet, je nutné vytvořit „Makefile“, ten je následně možné vykonat pouhým zadáním příkazu „make“. Následné načtení modulu provedeme příkazem „insmode prvnimodul.ko“.

Makefile

```
# Kořenoví adresář zdrojového balíku cíleného jádra
# Tento příkaz nastaví aktuálně používané jádro
KDIR = /lib/modules/`uname -r`/build

# Název modulu
obj-m := prvnimodul.o

# Cíle pro make
COMMAND = make -C $(KDIR) M=`pwd`
all:
    $(COMMAND)
install:
    $(COMMAND) modules_install
clean:
    $(COMMAND) clean
```

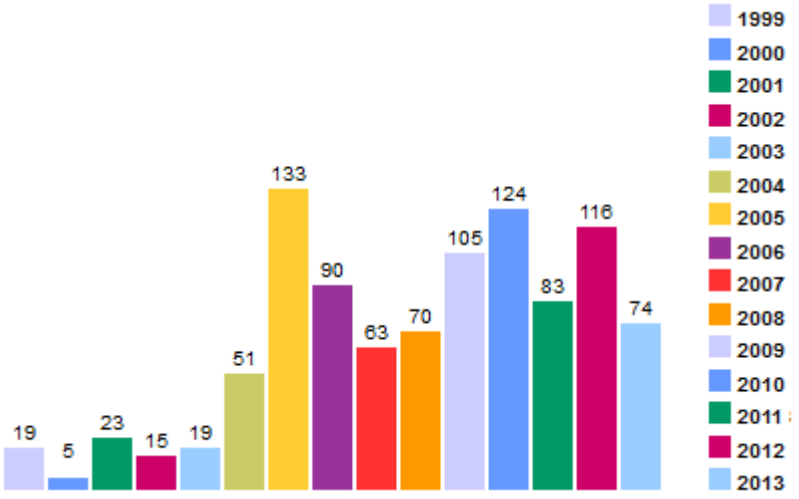
Tvorba modulů jádra, které cílí na průnik do systému, je po teoretické stránce velice oblíbeným tématem, problémem je nutnost privilegovaného oprávnění k práci s moduly jádra (načtení, apod.). Prakticky tedy nejsou nebezpečné jaderné moduly příliš časté.

Problematika vývoje ovladačů je natolik rozsáhlou oblastí, že není v možnostech práce takovéhoto rozsahu popsat více než naprosté základy. Bližší informace je možné nalézt v knize, jejímž autorem je Lukáš Jelínek [4], také tato práce z jeho knihy vychází. Dalším možným využitím modulů jádra je oprava nalezených chyb (zranitelností) bez nutnosti nového startu systému.

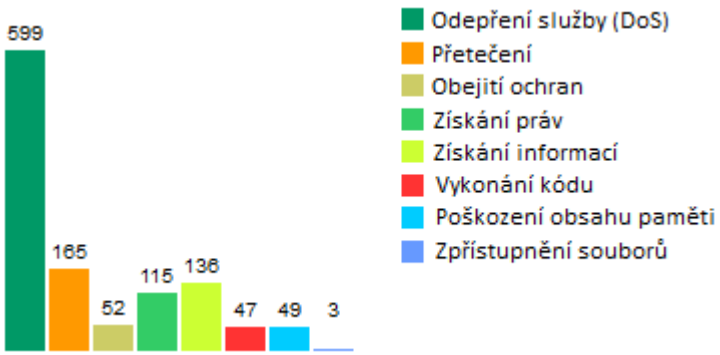
5.5 Známé zranitelnosti

Velice často se diskuze o bezpečnosti mine účinkem, nejsou-li představeny reálně existující zranitelnosti spolu s prezentací jejich možného zneužití. Jedná se o logický stav, jelikož vynakládat prostředky k zabezpečení domnělé či velice obtížně zneužitelné chyby je pro většinu obránců nevhodné z několika důvodů. Každá ochrana je aplikována na úkor uživatelského komfortu, výkonnosti systému nebo obojího. Taktéž je nevhodné zabezpečovat systém nákladným způsobem, není-li součástí citlivé infrastruktury, neobsahuje-li citlivá data nebo by náklady vynaložené na zabezpečení řádově přesáhly hodnotu takových dat. V konečném důsledku je možné pohlížet na bezpečnost jako na nekonečný proces hledání aktuálně optimálního kompromisu mezi zabezpečením a použitelností systému.

Obrázek 5.2 a Obrázek 5.3 ilustrují vývoj a zastoupení zranitelností obsažených v Linuxu. Statistiky začínají rokem 1999 z důvodu malého množství zranitelností nalezených před tímto obdobím a možnosti případného zkreslení dat.



Obrázek 5.2: Vývoj množství zranitelností v čase [16]



Obrázek 5.3: Četnost výskytu zranitelností dle typu [16]

Předchozí obrázky ilustrují vývoj a zastoupení zranitelností obsažených v Linuxu. Statistiky začínají rokem 1999 z důvodu malého množství zranitelností nalezených před tímto obdobím a možnosti případného zkreslení dat.

5.5.1 Chyba souběhu ptrace (CVE-2013-0871)

Chyba postihující jádro Linuxu 3. vývojové řady, konkrétně všechny verze v rozmezí 3.0 až 3.7.4, umožňuje lokálním uživatelům systému získat práva privilegovaného uživatele. Zranitelnost spočívá v možnosti poškození zásobníku jádra při použití **ptrace** s volbou „**PTRACE_SETREGS**“. [17]

Zneužití chyby spočívá ve vytvoření specifického scénáře, během kterého se původně podařilo chybu zneužít původně. Scénář spočívá v souběžném běhu několika vláken, kdy je jedno uspáváno a probouzeno a druhé se snaží modifikovat jeho registry. Po úspěšné modifikaci během spánku prvního vlákna je po jeho probuzení přečtena nesprávná hodnota ze zásobníku, což díky chybě umožní získání privilegovaného přístupu. Třetí vlákno je spuštěno po pozastavení druhého vlákna a je označeno jako vlákno vyžadující běh v reálném čase (RT), díky čemuž je schopné určit dobu, kdy první vlákno neběží. První a třetí vlákna se střídají na procesoru 0, ostatní vlákna na něm nesmějí být vykonávána. [18]

5.5.2 Chyba v systémovém volání vmsplice

Během roku 2008 byla zveřejněna zranitelnost obsažená v jádrech řady 2.6, konkrétně od jádra 2.6.1 po 2.6.24. Chyba se nacházela v systémovém volání **vmsplice()** a projevovala se během volání **mmap** s parametrem obsahujícím hodnotu NULL, útočník následně získal privilegovaný přístup k systému. Pro bezprostřední opravení nalezeného problému byla veřejně uvolněna oprava prostřednictvím modulu jádra. [19]

5.6 Obrana

Zabezpečení jádra operačního systému je velice problematickou záležitostí. Rozmanitost používaných verzí jádra a jejich rozdílné konfigurace činní zabezpečení problematičtější nežli u běžných aplikací. Například je nutné zároveň udržovat velké množství podporovaných verzí spolu s jejich značnou odlišností v důležitých aspektech. Kvalitní zabezpečení může rozbít funkcionalitu v neprivilegovaném prostoru, razantně snížit výkonnost a odezvu celého systému a v neposlední řadě má ve většině případů negativní dopad na komfort práce se systémem.

Kromě zabezpečení na úrovni programového kódu, ať už jaderném či nikoliv, je vhodným prvkem obrany sledování logů systému, prověření potenciálních bezpečnostních incidentů a zavedení propracované bezpečnostní politiky.

5.6.1 Úpravy jádra

Základním prvkem zabezpečujícím jádro operačního systému formou úpravy zdrojového kódu (patch) je oficiální bezpečnostní záplata. Většinou není vhodné aplikovat záplaty ihned po objevení či opravení dané hrozby, je vhodné ověřit funkčnost a stabilitu takového řešení, toto pravidlo je aplikovatelné na všechny zásahy do bezproblémově pracujícího systému. V roce 2007 byla například přidána zásadní oprava znemožňující využití útoku za pomoci ukazatele s hodnotou NULL kontrolou minimální adresy.

Velice častou chybou je opomíjení kontroly dat předávaných uživatelskými aplikacemi jádru, například pomocí systémových volání. Nejen, že taková data mohou obsahovat například shellcode, či jiný nebezpečný obsah, ovšem pouhá kontrola nepostačuje. Provede-li jádro kontrolu předávaných dat a následně s nimi pracuje, aplikace může naprosto bez problému měnit obsah přímo pod rukama pracujícího jádra, tato situace může nastat díky svobodě, kterou proces má uvnitř svého adresního prostoru. Možné řešení takovéhoto problému je kopírování dat do prostoru jádra. Při mapování jádra do samostatného paměťového prostoru, do nějž není procesu umožněn přístup, je možné využít výhod přístupu „**copy on write**“, kdy jsou modifikované bloky paměti kopírovány a ostatní aplikace stále vidí původní data.

Kromě obrany jádra před uživatelským prostorem je současně vhodné chránit tok dat opačným směrem. Například vadný modul jádra by neměl poškodit paměť procesu. Tento problém řeší oprava uvedená během října 2012, samotná oprava navazuje na implementaci ze strany společnosti Intel, který přidává ochranný bit „SMAP“

do registru CR4. Tato funkcionalita umožňuje snadnější nalezení chyb v jádře a navíc zabraňuje celé řadě exploitů, které se snaží donutit jádro, aby zapisovalo do uživatelského prostoru. [20]

Projekt „**grSecurity**“ je sadou úprav jádra Linuxu, poskytující obranu proti nejpopulárnějším typům exploitace a častým zranitelnostem. Na rozdíl od některých projektů zaměřených na bezpečnost není součástí hlavní větve jádra. Podporuje takové techniky, které v paměťovém prostoru náhodně umisťuje zásobník jádra i aplikací a taktéž rozložení adhesního prostoru, dále implementuje ochranu vykonávání dat spolu s kontrolou cesty k souborům.

Úprava nazvaná „**4G/4G**“ mění zobrazení paměti jádra takovým způsobem, že uživatelský prostor a prostor jádra využívají celý adresní prostor. Odezva systému klesne, ale na druhou stranu je možné přidělit každému procesu rozsáhlejší virtuální paměťový prostor.

Projekt „**SELinux**“vyvinutý americkým úřadem pro bezpečnost (NSA) implementuje povinné řízení přístupu (Mandatory Access Control). V konečném důsledku vynucuje jemnější rozlišení dostupných práv ke všem dostupným objektům. Je poskytována kompletní kontrola chování programů díky omezení přístupu do všech ostatních částí systému. Samotný program běží v kontrolovaném prostředí, které nazýváme „**sandbox**“. Útočník v takovém případě může využít pouze úkony poskytované takovým prostředím. To se týká také programů privilegovaného uživatele. Rozšíření je součástí hlavní větve jádra od verze 2.6.

Rozšíření „**AppArnor**“ vytváří bezpečnostní vrstvu mezi systémem a aplikacemi. Vrstva je implementována formou profilu určeného jednotlivým aplikacím. Takový seznam implikuje podobné kontrolované prostředí, jaké zaručuje projekt „**SELinux**“. Díky zabezpečení formou seznamu je výsledná konfigurace přehlednější a jednodušší nežli u předchozího rozšíření. Součástí hlavní větve jádra je od verze 2.6.36.

5.6.2 Úprava kompilátoru

Jelikož každý program musí projít kompilací, aby jej bylo možné vykonávat, je možné velkému množství chyb zabránit již během tohoto procesu. Existují převážně dvě možné varianty úpravy kompilátoru. První rozšíří kontrolní schopnosti kompilátoru, v zásadě tak probíhá statická kontrola zdrojových kódů. Druhá varianta modifikuje vý-

sledný program takovým způsobem, aby byla zajištěna například automatická kontrola mezí datových struktur.

Modifikovaný kompilátor nazvaný „**checker**“ rozšiřuje výsledné programy o vlastní implementaci knihovních funkcí pracujících s pamětí a navíc přidává kontrolu využití veškerých deklarovaných ukazatelů. Kompilátor „**cyclone**“ přidává kontrolu mezí polí, i když v tomto případě se jedná o zcela nový kompilátor.

5.6.3 Upravené knihovny

Podobná situace jako u kompilátorů je i na straně knihoven. Zde mezi nejznámější patří „**libsafe**“, obsahující novou implementaci převážné většiny problematických funkcí standardních knihoven. Její použití je velice jednoduché, pouze je pomocí direktivy „LD_PRELOAD“ zavedena do paměti. Díky vlastnosti zajišťující setrvání dříve načtených knihoven budou použity funkce obsažené v „libsafe“.

Závěr

Cílem bakalářské práce bylo seznámení s problematikou bezpečnosti, přednostně se zaměřením na bezpečnost jádra operačního systému. Bakalářská práce měla shrnovat základní principy fungování výpočetní techniky založené na architektuře mikroprocesoru IA-32, operačních systémů a seznámit s technikami nejčastěji používanými útočníky.

Vypracování práce bylo rozděleno do dvou rovin. Zaprvé předložit nutný teoretický základ, na kterém stavěla, a druhá část práce prakticky prezentovat dříve uvedené informace na konkrétních operačních systémech. Z důvodu udržení přehlednosti a probrání témat do potřebné hloubky byl nakonec vybrán jediný operační systém. Konkrétně operační systémy založené na jádře Linux.

Informace a technické detaily byly během psaní práce čerpány převážně z oficiálně dostupných dokumentů, odborných publikací a z elektronických médií zaměřených na bezpečnost. Malé množství kvalitních publikací dostupných v českém jazyce zapříčinilo převahu materiálů psaných v anglickém jazyce.

Mám-li zhodnotit míru odklonění od původního záměru, konstatují, že práce je v přesně takové míře, v jaké je možné se v daném časovém úseku udržet původní myšlenky v jasných obrysech. Některé části práce se od původního záměru odklonily více a některé části jsem rozpracoval nečekaně rozsáhle. Části, které bych zařadil do méně přesné kategorie, například popis možné ochrany a implementace napadení známých zranitelností. Do druhé kategorie bych naopak zařadil například popis některých implementačních detailů vybraného jádra operačního systému či techniky užívané útočníkem.

Možný budoucí vývoj práce bych viděl ve větším osobním přínosu k dané problematice, například vývoj zneužití známých zranitelností, k nimž není veřejně dostupný exploit. Dále bych rád práci více prohloubil o implementační detaily, známé zranitelnosti a případovou studii alespoň jednoho dalšího operačního systému. Jako vhodné vidím prodiskutovat více do hloubky možná rizika, jejich příčiny a možné nápravy.

Práce na vybraném tématu mě velice bavila a prohloubila mé odborné znalosti ohledně operačního systému. Dále jsem poznal, jakým způsobem je možné programovat moduly jádra, čemuž bych se velice rád dále věnoval více do hloubky. Samotná práce mi pomohla se zorientovat v otázkách bezpečnosti. Práce zaměnila způsob, jakým nahlížím na bezpečnost ve svém okolí. Rozhodně bych jí zhodnotil jako obrovský osobní

a profesní posun. Pěvně doufám, že práce napomůže proniknout do zdánlivě nedostupné problematiky.

Literatura

1. **Intel Corporation.** *Intel 64 and IA-32 Architectures Software Developer's Manual: Combined Volumes.* [Online] January 2013. [Cited: February 8, 2013.] http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-manual-325462.pdf. Order Number 325462-045US.

2. **MESSMER, Hans-Peter a DEMBOWSKI, Klaus.** *Velká kniha hardware: architektura, funkce, programování.* Brno : CP Books, 2005. ISBN 80-251-0416-8.

3. **DRÁB, Martin.** *Jádro systému Windows: kompletní průvodce programátora.* Brno : Computer Press, 2011. ISBN 978-80-251-2731-5.

4. **JELÍNEK, Lukáš.** *Jádro systému Linux: kompletní průvodce programátora.* Brno : Computer Press, 2008. ISBN 978-80-251-2084-2.

5. **TANENBAUM, Andrew.** *Modern operating systems.* 2st ed. Upper Saddle River : Prentice Hall, 1992. ISBN 0-13-595752-4.

6. **STALLING, William.** *Operating systems: Internals and Design Principles.* 5th ed. Upper Saddle River, N.J. : Pearson Prentice Hall, 2005. ISBN 0-13-127837-1.

7. **ERICKSON, Jon.** *Hacking: umění exploitace.* 2. vydání. Brno : Zoner Press, 2009. ISBN 978-80-7413-022-9.

8. **DOBŠÍČEK, Miroslav a BALLNER, Radim.** *Linux: bezpečnost a exploity.* České Budějovice : Koop, 2004. ISBN 80-7232-243-5.

9. **PERLA, Enrico and OLDANI, Massimiliano.** *A guide to kernel exploitation: attacking the core.* Amsterdam : Elsevier, 2011. ISBN 978-159-7494-861.

10. **Phrack inc.** Writing ia32 alphanumeric shellcodes. *The Phrack Magazine.* [Online] August 11, 2001. [Cited: April 6, 2013.] http://www.phrack.org/issues.html?issue=57&id=5#article.

11. **CERN Computer Security Team.** Common vulnerabilities guide for C programmers. *CERN Computer Security.* [Online] 2013. [Cited: April 20, 2013.] https://security.web.cern.ch/security/recommendations/en/codetools/c.shtml.

12. **Kolektiv autorů.** *Linux: dokumentační projekt.* [překl.] Lubomír PTÁČEK. 4. aktualizované vydání. Brno : Computer Press, 2007. ISBN 978-80-251-1525-1.

13. **Solar Designer.** Getting around non-executable stack (and fix). *SecLists.Org Security Mailing List Archive.* [Online] August 10, 1997. [Cited: April 22, 2013.] http://seclists.org/bugtraq/1997/Aug/63.

14. **Phrack inc.** The advanced return-into-lib(c) exploits. *The Phrack Magazine.* [Online] August 11, 2001. [Cited: April 22, 2013.] http://www.phrack.org/issues.html?issue=58&id=4#article.

15. **KRAHMER, Sebastian.** *x86-64 buffer overflow exploits and the borrowed code chuncks exploitation technique.* [Online] September 28, 2005. [Cited: April 22, 2013.] http://users.suse.com/~krahmer/no-nx.pdf.

16. **ÖZKAN, Serkan.** Linux Kernel: Vulnerability Statistics. *CVE Details: The ultimate security vulnerability datasource.* [Online] 2013. [Cited: April 21, 2013.] http://www.cvedetails.com/product/47/Linux-Linux-Kernel.html.

17. —. Vulnerability Details: CVE-2013-0871. *CVE Details: The ultimate security vulnerability datasource.* [Online] 2013. [Cited: April 21, 2013.] http://www.cvedetails.com/cve/CVE-2013-0871/.

18. **TINNES, Julien.** Linux kernel race condition with PTRACE_SETREGS (CVE-2013-0871). *SecLists.Org Security Mailing List Archive.* [Online] February 15, 2013. [Cited: April 22, 2013.] http://seclists.org/oss-sec/2013/q1/326.

19. **JOMBÍK, Ondrej.** Rýchla oprava lokálnej vmsplice() zraniteľnosti. *Platon technologies.* [Online] 12. únor 2008. [Citace: 28. duben 2013.] http://opensource.platon.sk/article.php?vmsplice-zranitelnost-rychla-oprava.

20. **CORBET, Jonathan.** Supervisor mode access prevention. *LWN.net.* [Online] September 26, 2012. [Cited: April 22, 2013.] https://lwn.net/Articles/517475/.