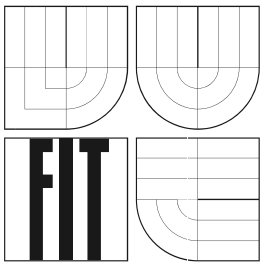
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ



Bezpečnost UNIXových systémů na úrovni jádra

Bakalářská práce

2005 **Jiří Hýsek**

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Kašpárka. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Brně dne 1.5. 2005 Jiří Hýsek

Abstrakt

Tato bakalářská práce se zabývá bezpečností UNIX-like systémů na úrovni jádra. Skládá ze dvou hlavních částí. První část, teoretická, popisuje bezpečnostní mechanismy, které jádra UNIX-like systémů obsahují ať již standardně nebo aplikací různých bezpečnostních záplat. Snažím se o obecnější pohled. Nezaměřuji se tedy pouze na Linux, na jeden z dnes nejrozšířenějších systémů, který z UNIXového návrhu vychází, ale také na BSD systémy, které jsou co se týče bezpečnosti, podle mého názoru, propracovanější. Z BSD systémů se často odovlávám na nejrozšířenější z nich – FreeBSD.

Druhá část je praktická, zabývá se technikami, které útočníci používají pro skrytí své činnosti a přítomnosti v systému, navrhuji a popisuji metody, jak jejich krycí mechanismy obejít. Většinu z popisovaných technik detekce jsem implementoval v automatickém nástroji. Jeho účinnost jsem otestoval na několika různých typech rootkitů – nástrojích, které útočníci využívají pro své krytí.

Klíčová slova

počítačová bezpečnost, jádro, detekce napadení, rootkit, přesměrování systémových volání, přesměrování VFS funkcí, capability, bezpečnostní úrovně, jail, chroot, ACL, MAC, Linux security module

Poděkování

Tímto bych chtěl poděkovat firmě Google, Inc., která mi pravidelně pomáhala řešit všechny problémy, které se během psaní práce vyskytly a v neposlední řadě také Ing. Kašpárkovi za svědomitou kontrolu textů a odpovědi na všechny mé dotazy.

Abstract

This bachelor thesis deals with security of UNIX-like systems at kernel layer. It contents two main parts. The first one introduces security mechanisms, which are in UNIX-like systems included by default or as a part of one of various security patches. I'm trying to see this problem more general. I'm not focusing on Linux only – one of the most popular UNIX-like system at present, but also on BSD systems, what are on security level, in my oppinion, more sofisticated. From family of BSD systems I often refer to the most widespread one – FreeBSD.

In second part are described techniques used by attacker to ensure his invisibility inside system. Also are discussed techniques how to bypass attackers' hidding tools (rootkits). The most of presented principles are implemented into automatic intrusion detection tool. It's accurancy have been tested on several types of kernel rootkits.

Key words

computer security, kernel, intrusion detection, rookit, system call hooking, VFS functions hooking, capability, secure level, jail, chroot, ACL, MAC, Linux security module

Obsah

Kapitola 1

Úvod

V posledních letech se počítačové bezpečnosti dostává stále větší pozornosti. Dříve nebyly počítačové systémy navrhovány s ohledem na důkladné zabezpečení dat. Tehdy se využívaly převážně k akademickým účelům. Dnes je však situace značně odlišná. Stále více a více se počítače, potažmo Internet, využívají k podnikatelským záměrům, v bankovnictví, používají je státní orgány a potřeba zabezpečit data před zcizením či zneužitím je v těchto oblastech extrémně vysoká.

O počítačové bezpečnosti toho bylo napsáno mnoho. Většina knih se však zabývá problematikou z pohledu správce systému — popisuje tedy bezpečnostní programy a jejich konfiguraci. Méně často se věnují nízkoúrovňové bezpečnosti, kde popisují chyby v programech, které útočníci využívají, a jak těmto útokům předcházet. O bezpečnosti na úrovni jádra většina literatury mlčí. Tato práce je tedy stručný úvod do této problematiky. Proto je teoretická část spíše přehledová. Snažil jsem se příliš často nezabíhat do zbytečných podrobností a implementačních detailů, popsat pouze princip s tím, že se zájemci mohou dočíst více v množství nabídnutých zdrojů.

V praktické části se věnuji jednom konkrétnímu problému, a to detekci napadení systému za běhu. Je to tedy součást real-time *forenzní analýzy*[1]. Zaměřuji se zde na popis technik, které útočníci používají pro krytí své činnosti a přítomnosti v systému a na návrh metod, jak toto krytí odhalit či obejít. Útočníci většinou používají automatické krycí nástroje – rootkity. Běžně dostupných rootkitů je spousta, existuje několik odlišných typů. Navržené techniky detekce přítomnosti krycích nástrojů jsem implementoval v automatickém nástroji. V poslední kapitole popisuji testy jeho účinnosti na několika různých typech rootkitů.

[1] Obecný úvod do forenzní analýzy můžete najít například v [8]

Kapitola 2

Standardní bezpečnostní prvky

V této kapitole si popíšeme a vysvětlíme princip bezpečnostních prvků, které můžeme najít jako standardní součást nějběžnějších UNIXových systémů. Téměř všechny popisované prvky jsou obsaženy v BSD systémech, v Linuxu řady 2.4 a nižší jsou některé dostupné pouze v podobě bezpečnostních patchů (grsecurity, LIDS, ...), v řadě 2.6 je situace již lepší.

2.1 Přístupová práva

Přístupová práva souborů jsou spíše záležitostí souborových systémů, ale protože různé bezpečnostní patche do jádra často přinášejí rozšířené možnosti přidělování práv, myslím, že není od věci vysvětlit princip, na kterém jsou práva založena už od počátku UNIXu.

Práva k souboru jsou dána zvlášť pro vlastníka souboru, pro skupinu, ke které soubor patří a pro ostatní. Práva určují, zda je možné soubor číst (r), zapisovat do něj (w) a spouštět jej (x). Pro adresáře má právo pro spouštění význam "vstup do adresáře".

Ve všech UNIX-like systémech se přístupová práva nastavují příkazem *chmod* a vypisují se příkazem *ls* s parametrem *-l*. Tento model přístupových práv se označuje jako DAC (discretionary access control).

2.2 Příznaky souborů

Systémy BSD mají oproti standardním UNIXovým právům další prostředky, jak určovat, co se soubory možné provádět je a co není. Jsou to příznaky nebo ACL (viz. 2.3). Přístupová práva souborů mají menší prioritu než příznaky, je-li tedy nastaven příznak zakazující něco, práva nejsou brána v potaz a daná činnost povolena není a to ani pro superuživatele. Pokud není nastaven žádný příznak, pak samozřejmě rozhodují pouze práva.

Příznaky mohou být systémové nebo uživatelské. Systémové příznaky může nastavovat pouze root a mohou být chráněny zvýšením bezpečnostní úrovněmi systému (viz. 2.4). Uživatelské příznaky mohou být nastavovány i vlastníkem souboru. Tyto příznaky je možné za běhu rušit bez ohledu na nastavenou bezpečnostní úroveň.

Pojďme se podrobněji podívat na jednotlivé příznaky. Ještě poznamenám, že jich existuje více, než popisuji — zaměřuji se pouze na příznaky sloužící ke zvýšení bezpečnosti. Máme tři základní. *appnd*, *chg* a *ulnk*. *appnd* povoluje pouze přidávání dat na konec souboru, *chg* znemožňuje modifikaci souboru a *ulnk* znemožňuje pouze mazání souboru (modifikace možná je). Každý z příznaků

existuje jak v systémové podobě, tak v uživatelské. Toto je rozlišeno prvním písmenem názvu. Jsou dostupné tedy následující příznaky:

- *sappnd* – do souboru s tímto příznakem je možno zapisovat pouze na konec souboru. Není možné data měnit ani soubor smazat. Může to být užitečné například pro nějaké logové soubory. Útočník pak nemůže smazat záznamy o jeho činnosti ani celý soubor s logy. Podle prvního písmene víme, že jde o systémový příznak, je tedy nastavitelný pouze superuživatelem a nelze rušit při vyšších bezpečnostních úrovních.
- *schg* – opět systémový příznak; určuje, že soubor není možné modifikovat ani mazat. Pokud tento příznak nastavíme například na vše v /bin a /sbin, máme jistotu, že útočník nepodvrhne systémové programy. To dříve útočníci (nejspíš se najdou takoví i dnes) běžně prováděli, aby skryli svou přítomnost nebo vytvořili zadní vrátka do systému.
- *sunlnk* – tento systémový příznak zajišťuje, že daný soubor není možné smazat. Jako prostředek k zabezpečení příliš užitečný není. Soubor lze modifikovat, tedy i úplně smazat jeho obsah. Slouží spíše jako ochrana před nechtěným smazáním souboru.
- *uappnd* – uživatelský příznak, může jej tedy nastavit vlastník souboru i root, oba ho mohou také za běhu zrušit, bez ohledu na nastavenou bezpečnostní úroveň. Z toho je zřejmé, že slouží opět spíše jako ochrana před náhodným smazáním nebo přepsáním, stejne jako všechny uživatelské příznaky.
- *uchg* — příznak znemožňující jakoukoli změnu souboru po dobu, po kterou je příznak nastaven. Opět ho může nastavit i kdykoli zrušit vlastník souboru a superuživatel.
- *uunlnk* – uživatelská varianta příznaku unlnk – znemožňuje smazání souboru.

Příznaky souborů se v systému FreeBSD nastavují příkazem *chflags*. Informace o nastavených příznacích získáme zadáním přepínače *-o* společně s *-l* příkazu *ls*.

Příznaky souborů jsem popisoval pro BSD systémy. Pro úplnost dodávám, že v Linuxu existuje podobný prostředek. Zde se jim říká atributy souborů a umožňují souborům přiřazovat tato omezení[1]:

- *append only* – do souboru je možné zapisovat pouze na konec
- *immutable* – soubor nelze modifikovat
- *secure deletion* – data souboru jsou před smazáním fyzicky přepsána nulami
- *undeletable* – pokud je sobor smazán, je zachován jeho obsah a je možné jej obnovit

Pro nastavení těchto atributů slouží příkaz *chattr*. Pro výpis souborů včetně přiřazených atributů použijeme příkaz *lsattr*.

2.3 Securelevels – bezpečnostní úrovně systému

Jádra BSD-like systémů nabízejí prostředek, jakým je možné zavádět bezpečnostní omezení platná pro celý systém. Říkáme jim bezpečnostní úrovně (angl. securelevels). Bezpečnostní úroveň je možné za běhu zvýšit (vyšší = bezpečnejší, větší omezení), nikoli však snížit. Ke snížení bezpečnostní

[1] Uvádím zde pouze atributy související se zabezpečením. Kompletní seznam najdete v manuálových stránkých příkazu *chattr* (viz. [28]).

úrovně je nutno přejít do jednouživatelského režimu. V něm nejsou aktivní síťová rozhraní, je tedy třeba fyzický přístup k serveru. To efektivně zbavuje útočníka možnosti tato omezení snížit.

BSD má 5 bezpečnostních úrovní -1 až 3. Čím je číslo vyšší, tím se uplatňuje více omezení.

2.3.1 Securelevel -1 – "trvale nezabezpečený režim"

Pokud má securelevel hodnotu -1, nejsou uplatněna žádná omezení. Systém se tedy chová, jako by žádné bezpečnostní úrovně neznal.

2.3.2 Securelevel 0 – "nezabezpečený režim"

Tato hodnota se nikdy trvale nepoužívá. Je nastavena pouze při startu systému v případě, že je securelevel nastaven na hodnotu >=0. Po přechodu do víceuživatelského režimu se hodnota (v případě, že je nastavena na 0) automaticky přepne na hodnotu 1. Je to tedy totéž, jako bychom měli nastavenou hodnotu 1.

2.3.3 Securelevel 1 – "bezpečný režim"

Na této úrovni se již uplatňují tato omezení:

- Není možné přidávat a odebírat moduly do jádra [1],
- nelze měnit systémové příznaky souborů – to je velmi užitečná vlastnost. Pokud si nastavíte například na soubor /etc/passwd příznak schg a nastavíte tuto nebo vyšší bezpečnostní úroveň, můžete si být jistí, že nikdo, ani kdyby měl práva superuživatele, nemá možnost přidat dalšího uživatele. Ovšem ani administrátor ne. K tomu, aby přidal uživatele, musí přejít do jednouživatelského režimu, tudíž počítat s tím, že všechny služby budou během jeho činnosti nedostupné. Lépe je tedy tuto vlastnost využívat pro soubory, které se měnit za běhu naopak nemají, například jádro systému.
- je zakázáno zapisovat do souborů /dev/kmem[2] a /dev/mem,
- nelze spustit systém X Windows,
- nelze přímo zapisovat na připojené disky (tedy do příslušných souborů v /dev; prostřednictvím systémových volání pro práci se soubory zapisovat lze). Při bezpečnostní úrovni -1 je možné na disk díky speciálnímu souboru v /dev, který jej reprezentuje, zapisovat přímo voláním *write*. Je tedy možné zapsat na libovolnou část disku a tím obejít mechanismy omezující přístup k souborům.

2.3.4 Securelevel 2 – "vysoce bezpečný režim"

Ke všem omezením úrovně 1 přibudou ještě následující dvě:

- Nelze přímo zapisovat do připojených ani nepřipojených souborových systémů (opět tím rozmíme soubory v /dev, tentokrát se omezení týká i disků s nepřipojenými souborovými systémy), zamezí se tím možnosti odpojení disku a následného přepsání,
- systémový čas lze upravovat nanejvýš o 1 sekundu. Při pokusu o větší změnu, se uloží do logu varování.

[1] moduly do jádra se podrobněji budeme zabývat v 4.1
[2] o tomto souboru si více řekneme v 4.2

2.3.5 Securelevel 3 – "bezpečný síťový režim"

Na této úrovni opět zůstala všechna omezení úrovně předchozí, zakazuje však měnit pravidla paketového filtru *ipfw* nebo *ipf*.

2.3.6 Výhody

- globální omezení pro celý systém na úrovni jádra
- restrikce nelze za běhu rušit, pouze přidávat — útočník tudíž nemá možnost tato omezení odstranit, jedině, že by měl fyzický přístup k počítači, což je velmi nepravděpodobné.

2.3.7 Nevýhody

- obtížnější administrace systému — vysoká bezpečnostní úroveň omezuje nejen útočníka, ale samozřejmě i správce. Zvýšení bezpečnostní úrovně se provádí až po odladění konfigurace serveru. Poté se i pro banální úpravu konfiguračního souboru, který má například nastaven příznak znemožňující zápis, musí přejít do jednouživatelského režimu.

2.4 ACL (access control list)

Dostáváme k dalšímu prostředku pro omezování přístupu k souborům. Je jím takzvaný *access control list*, známý pod zkratkou ACL. Některé systémy jej nabízejí přímo (BSD, Linux 2.6.x), pro jiné (např. Linux v řadě $<= 2.4.x$) je zase součástí bezpečnostních záplat do jádra. Běžné UNIXové systémy implementují ACL podle specifikace POSIX 1003.1e (BSD, Linux, Solaris, ...).

V případě, že požadujeme nastavit práva pro několik uživatelů, je při použití standardních přístupových práv nutno vytvořit skupinu, nastavit práva pro ni a požadované uživatele do ni přidat. To může být ve složitějších případech velmi neelegantní, zdali vůbec možné. Navíc běžní uživatelé nemají možnost vytvářet nové skupiny, takže ani tuto náhražku použít nemohou. Situaci řeší ACL.

ACL umožňuje definovat pro každý jednotlivý soubor práva konkrétních uživatelů a skupin. Práva umožňuje definovat standardní, tedy čtení, zápis a spouštění. Je tedy snadné definovat například k jednomu souboru pro každého uživatele či skupinu jiná práva. ACL je v komplikovanějších situacích značně přehlednější a lépe udržovatelné řešení, než klasická práva pro vlastníka, skupinu a ostatní.

ACL se nastavují příkazem *setfacl* a vypisují pomocí *getfacl*.

2.5 MAC (mandatory access control)

V této kapitole se budeme věnovat MAC podle normy POSIX.6. MAC je obecně definováno jako způsob řízení přístupu k *objektům subjekty*. Jediným uvažovaným subjektem v MAC podle POSIX.6 je *proces* (dále v textu budeme místo subjekt používat proces). Objekty mohou být buď soubory (běžné soubory, adresáře, pojmenované roury apod.) nebo procesy. Každý objekt a proces má přiřazen MAC *identifikátor* (label).

MAC definuje omezení přístupu na základě porovnání identifikátoru objektu a procesu, který k němu přistupuje. Proces má k danému objektu přístup, pokud jeho identifikátor je na stejné nebo vyšší úrovni jako identifikátor objektu. Pokud process vytvoří nový objekt, musí mít tento objekt identifikátor na stejné nebo vyšší úrovni.

Identifikátory objektů a procesů včetně jejich nadřazenosti je možno definovat podle sebe. Pro čtení a nastavení MAC identifikátoru slouží příkazy *getfmac* (*getpmac* pro proces) a *setfmac* (*setpmac*).

2.6 Capabilities

Capability se dá přeložit jako *schopnost* nebo *způsobilost*[2]. Linuxové jádro rozpoznává, zda má daný proces na konkrétní akci právo, pomocí *capabilities*. Pokud je proces způsobilý provést danou akci, je akce povolena.

V Linuxu je možné nalézt všechny capability v hlavičkovém souboru *linux/capability.h*.

2.6.1 Capability procesů

Norma POSIX říká, že každý proces má tři množiny bitů – *inheritable*, *permitted* a *effective*. Každá capabilita je vyjádřena bitem v každé z těchto množin.

Pokud se proces pokusí provést privilegovanou operaci, systém ověří, zda má nastaven příslušný bit v množině *effective*.

Množina *permitted* říká, které z capabilit může mít proces nastaven. Nemůže mít tedy v množině *effective* nastaven bit, který není nastaven v množině *permitted*.

Množina *inheritable* určuje, které capability může zdědit proces tímto procesem vytvořený. Tedy množina *permitted* se vymaskuje s *inheritable* a výsledek bude množina *permitted* pro nový proces. Toto se dělá pouze při volání *exec*. Při *fork* nebo *clone* je množina *permitted* totožná s rodičovským procesem.

2.6.2 Capability souborů

Capability mají i spustitelné soubory. Mají také 3 množiny, ale jejich význam je jiný, proto se jmenují jinak. Jsou to *allowed, forced* a *effective*.

V množině *allowed* jsou nastaveny bity, které může nový proces zdědit od rodičovského procesu. Množina *permitted* nového procesu je to kombinace *permitted* a *inheritable* rodičovského a *allowed* souboru.

Množina *forced* má nastaveny bity, které jsou po spuštění souboru přidány do množiny *permitted*. Jsou to tedy capability, které proces dostane nezávisle na tom, zda je zdědí od rodičovského procesu. Je to tedy podobná vlastnost jako *setuid* bit.

Množine *effective* u souboru určuje, které bity z *permitted* se po spuštění přesunou do množiny *effective* nově vytvořeného procesu. Může se skládat buď ze samých jedniček nebo nul.

2.7 Chroot prostředí

chroot je systémové volání, které slouží ke změně kořenového adresáře (change root). Tím omezíme působnost procesu pouze na určitou část souborového systému. Požadovaný proces tedy uzavřeme do prostředí, které obsahuje pouze soubory, jež potřebuje k činnosti (může samozřejmě obsahovat jakékoli soubory, ale pokud neprovozujeme honeypot[3], nemá to příliš smysl). Pokud by program běžící v chrootu obsahoval bezpečnostní chybu, díky které by se útočník dostal do systému, nic kritického by se v podstatě nedělo. Útočník nemá šanci se dostat za hranice tohoto prostředí a napáchat nějaké škody. Tedy neměl by mít. Praxe je taková, že ve velké většině případů není

[2] český název není zažitý, nejspíš se ani nepoužívá, proto se budu držet anglického termínu capability

[3] uměle vytvořený systém sloužící ke sledování činnosti útočníků

problém ze z chrootu dostat. Existuje několik způsobů, jak opustit chrootem omezené prostředí (viz. např. [12]). Původně chroot nesloužil pro zajištění bezpečnosti, ale pouze pro oddělení anonymního ftp od zbytku systému. Operace, které ftp umožňuje byly chrootem kontrolovány, aby nemohlo dojít k úniku z prostředí. Časem se ukázalo, že neošetřuje rekurzivní volání *chroot*, použití ".." nebo volání *fchdir*. Tím chroot dnes v podstatě ztrácí smysl. Je to sice lepší než nic, ale dostatečně neplní účel, pro který jej administrátoři používají. Proto byla myšlenka chrootu dovedena dál a vznikl *jail*.

2.8 Jail

Jail poskytuje bezpečné prostředí *virtuálního stroje*. Procesy uvnitř mohou manipulovat se soubory v prostředí, ovlivňovat procesy v tomtéž prostředí a používat síťové služby. Nemohou však vidět ani manipulovat se soubory, procesy nebo síťovými spojeními mimo prostřejí jailu, ve kterém běží. Mohou s nimi komunikovat pouze prostřednictví síťových spojení.

Každé virtuální prostředí jailu je svázáno s jednou IP adresou. Procesy v něm nemohou používat jinou lokální IP adresu pro odchozí ani příchozí spojení. Takže například pokus procesu poslouchat na určitém portu na všech dostupných adresách se nezdaří — proces bude posouchat pouze na adrese, kterou má daný jail přiřazenu. Stejně tak i jiná volání budou omezena. Například pokus o vytvoření zařízení skončí s chybou.

Hlavní přínos jailu je tedy v tom, že je od počátku navržen pro zajištění bezpečnosti, ne pouze pro oddělení služby od zbytku systému, jako je tomu u chrootu.

2.9 Linux security modules (LSM)

LSM je framework umožňující modulům kontrolovat různé akce v jádře. Akcemi myslíme přístupy k interním objektům. LSM framework přidává do struktur jádra (viz. tabulka 1) ukazatele na funkce, které se provedou během vykonání operace (po všech běžných kontrolách oprávnění a ošetření chyb) těsně před vykonáním samotné akce (např. před přístupem k inodu). Modul má možnost tyto funkce implementovat, a tím si určit, zda daný proces má pro tuto akci oprávnění. Z umístění volání rozhodovací funkce (hooku) vyplývá, že pokud jádro akci povolí, může jí funkce zakázat, ale pokud jí nepovolí jádro, k provedení funkce již nedojde. Je tedy primárně *restriktivní*, nemá možnost povolit operaci, kterou jádro kvůli běžným právům nepovolí. LSM však umožňuje i *permisivní* (povolující) hooky. Je to při kontrole *capabilit* (viz. 2.6), typicky spojené s kontrolou standardních přístupových práv (DAC, viz. 2.1).

Pomocí LSM je tak možné implementovat bezpečnostní omezení známá z jiných systémů, jako např. Jail nebo bezpečnostní úrovně.

struktura jádra	objekt
task_struct	proces
linux_binprm	program
super_block	souborový systém
inode	soubor, roura nebo socket
file	otevřený soubor
sk_buff	síťový buffer (paket)
net_device	síťové zařízení
kern_ipc_perm	semafor, segment sdílené paměti nebo fronta zpráv
msg_msg	zpráva

tab. 1: struktury jádra modifikované LSM a odpovídající objekty

Kapitola 3

Bezpečnostní patche do jádra

Standardní jádro některých systémů (např. Linux řady 2.4 a nižší) neobsahuje prvky umožňující zabezpečení systému na dostatečné úrovni. Proto vznikají záplaty jádra od třetích stran, které se tyto problémy snaží řešit. Většinou jsou ve formě patchů, které se musí na jádro aplikovat a to znovu zkompilovat a nainstalovat. Jinou možností je použít modul do jádra (viz. 4.1.2) nebo v případě Linuxu 2.6 již zmiňované *Linux security modules* (viz. 2.9). Domovské stránky několika nejznámějších projektů jsou [30, 31, 32, 29]

3.1 Co přináší?

3.1.1 Propracovanější modely přístupových práv

Nejčastěji patche vylepšují možnosti omezení přístupu k souborům, procesům, zařízením, prostředkům pro meziprocesovou komunikaci nebo dalším prostředkům OS. Řeší také problém s neomezenými právy superuživatele — v případě, že útočník získá tato práva, má právo dělat *"vše"*.

Nejznámější rozšiřující model je **ACL**, ten se však do dnešních systémů implementuje standardně, takže již přestává být součástí bezpečnostních patchů. Ty často přinášejí tzv. **RBAC** model práv. Tedy "role based access control" – kontrola přístupu založená na *rolích*. Toto je známý princip například z databází. Nadefinují se určité role (např. správce databáze, běžný uživatel, webmaster,..) a jim se přidělí určitá práva. Uživatel potom "hraje" některou z definovaných rolí.

Situace hlavně v Linuxu řady 2.4 a nižších je v této oblasti velmi špatná. Standardně je k dispozici pouze model přístupových práv známý již od vzniku UNIXu. V jádrech řady 2.6 je již podpora ACL a MAC. V BSD systémech je ACL dostupné již dlouhou dobu[1].

3.1.2 Ochrana procesů a souborů

Kromě omezení přístupu právy umožňují bezpečnostní patche často *chránit* soubory nebo procesy.

Chráněný soubor není možné smazat ani jinak modifikovat, a to ani superuživatelem. Soubory je možné také skrýt — nebude viditelný pro žádné uživatele, včetně superuživatele.

Chráněný proces zase není možné ukončit, ani zasláním signálů, které za běžné situace proces nemůže ignorovat. I procesy je možné skrýt. Útočník pak nemá způsob[2], jak zjistit, že takový proces v systému běží.

[1] FreeBSD obsahuje ACL podle normy POSIX.1e již od verze 4.0, která byla vydána v březnu roku 2000, tedy více než 5 let zpět. MAC byl představen ve verzi 5.0 z ledna 2003.

[2] na teoretické úrovni. v 4.6 si řekneme, jak odhalit i skrytý process

3.1.3 Nespustitelné stránky paměti

Snad nejběžněji zneužívanou chybou v programech je přetečení bufferu. Programátor si nekontroluje, zda se uživatelem zadaná data vejdou do připraveného pole. To je uloženo na zásobníku, stejně jako adresa, odkud má program pokračovat po návratu z funkce. Toho útočník může zneužít tím, že nechá program přepsat část zásobníku, kde návratová adresa leží. Program ji při ukončení funkce vyzvedne a pokračuje prováděním kódu na této adrese. Pokud útočník do zadaných dat vloží kód a přepíše návratovou adresu tak, aby ukazovala na začátek jeho kódu, může program nechat provést, co si bude přát, a to s právy běžícího programu.

Bezpečnostní patche implementjí příznak paměťových stránek, který určuje, zda je datová nebo programová. Tedy je-li spustitelná nebo ne. Zásobník je datová oblast, bude tedy zakázáno vykonávat kód v něm uložený. Při pokusu o spuštění kódu z nespustitelné stránky, se vyvolá vyjímka a operační systém program, který ji způsobil, ukončí.

Tato ochrana však lze obejít. Byť velmi obtížně. Využívá se technika *return-into-libc*, kdy se kód nechá vykonat standardní knihovní funkcí *execve*. Podrobnosti můžete naléz např. v [18].

3.1.4 Adress space layout randomization (ASLR)

V běžných systémech mají procesy namapovány jednotlivé části paměti (datový a programový ELF segment, segmenty libc, zásobník,..) v předem známém pořadí. Podívat se na ně můžeme vypsánám souboru */proc/<PID>/maps*. Exploity, které využívají techniku *return-into-libc* spoléhají na znalost adres zásobníku a adres funkcí standardní knihovny C. Randomizací adres jednotlivých segmentů výrazně ztížíme útočníkovi práci. Nejde tedy o úplnou prevenci před těmito typy útoků, pouze jejich znepříjemnění. Útočník může adresy uhádnout, nebo použít hrubou sílu k nalezení těchto adres, což ale prodlouží dobu potřebnou k průniku. Navíc to ještě znepříjemňuje již tak dost komplikovanou techniku *retrun-into-libc*, většinu útočníků pravděpodobně odradí. O technice obcházení ASLR se můžete dočíst více v TODO.

Tuto ochranu (stejně jako předchozí popisované) nabízí například patch pro Linux *grsecurity* (PaX), nebo standardně systém OpenBSD, který je však zaměřen na bezpečnost, obsahuje tedy snad všechna popisovaná bezpečnostní vylepšení.

3.1.5 Real-time intrusion detection

Některé bezpečnostní záplaty obsahují také mechanismy pro odhalení potenciálního pokusu o útok či jeho předvoje. Tím mám na mysli především detektor skenování otevřených portů nebo ochrana před DoS útokem buď zevnitř (proces alokuje příliš paměti, vytváří spoustu dceřiných procesů apod.), nebo zvenší (zahlcování síťových služeb).

Při zaznamenání takové události mohou daný proces nebo síťové spojení ukončit a zaznamenat varování včetně IP adresy, či informovat superuživatele apod.

3.1.6 Ostatní vlastnosti

Bezpečnostní záplaty jsou často velké projekty a nabízejí spoustu různých vylepšení nejrůznějších částí systému. Kromě výše popisovaných to může být například důkladné zabezpečení chrootu, různé logování spousty činností pro provádění bezpečnostních auditů či usnadnění identifikace útočníka, randomizace zdrojových TCP portů nebo identifikačnách čísel procesů či jiná zabezpečení slabých míst nebo zamlžení informací využitelných k získání informací o systému nebo dokonce k jeho napadení.

Kapitola 4

Průnik a jeho detekce

Průnikem budeme v dalším textu rozumět změnu jádra, která umožní útočníkovi zvýšit práva, skrývat svou přítomnost a činnost v systému nebo ho jiným způsobem podporovat v jeho neoprávněné činnosti. Existuje více způsobů, jak změnit část jádra, a samozřejmě i spousta možností, jaké změny provést.

V této kapitole se zaměřím na objasnění principů provedení změn jádra a navrhnu možné techniky jejich odhalení. Změnami jádra se dále rozumí pouze takové, které vypovídají o přítomnosti rootkitu nebo přítomnosti "vetřelce" v systému. Může to být například změna záznamů v tabulce systémových volání, neshoda seznamu modulů získaným výpisem standardního programu pro tento účel se seznamem modulů nalezených v /dev/kmem a další.

Popisované principy platí obecně, ale vysvětlení se bude týkat Linuxu (projeví se to v názvech souborů, programů, struktur v jádře a případně dalších detailů).

4.1 Pojmy

4.1.1 Úrovně oprávnění

V chráněném režimu, v němž pracují snad všechny moderní operační systémy, existuje více úrovní oprávnění. Intelovská architektura obsahuje 4 urovně 0 - 3. V dokumentaci Intelu jsou znázorňované pomocí 4 soustředných kruhů, úrovním se tedy říká ring 0 až ring 3. Unix-like systémy využívají pouze dvě. Úroveň 0 pro kernel a 3 pro vše ostatní. Paměťovým prostorům s těmito úrovněmi oprávnění říkáme *prostor jádra* (kernelspace) a *uživatelský prostor* (userspace). Programy v uživatelském prostoru nemohou vykonavát privilegované instrukce (to lze pouze s úrovní 0), provádět I/O instrukce (out, in, ..) a samozřejmě nemají přístup k paměti jádra. Platí, že proces nemá přímý přístup do paměťového prostoru s jinou úrovní oprávnění.

4.1.2 Interrupt descriptor table (IDT)

Programy často potřebují provádět činnosti, na které je potřeba vyšší úroveň oprávnění (např. zápis nebo čtení z disku), ale proces nesmí předat řízení do paměťového segmentu s jinou úrovní oprávnění, než má sám. Pouze přes tzv. brány (gates). Brány mohou být buď *přerušení*, *obsluhy vyjímek* (traps) nebo *brány procesů* (*task gates*) Brány mají takzvané deskriptory uloženy v poli o 256 položkách (všechny nemusí být využity). Této tabulce se říká "interrupt descriptor table", zkráceně IDT. Adresa IDT je uložena ve speciálním registru IDTR (IDT register). Registr obsahuje

bázi IDT (4 bajty), tedy adresu, kde IDT začíná, a *limit* (2 bajty), který určuje délku IDT. Registr lze naplňit pouze privilegovanou instrukcí *LIDT* a přečíst instrukcí *SIDT*.

Každý deskriptor má velikost 8bajtů a obsahuje například offset obslužné rutiny, informaci o úrovni oprávnění a pod. Podrobné vysvětlení problematiky je nad rámec této práce, zájemce odkáži na dokumentaci Intelu [24].

Jednou z těchto bran je přerušení *system call* (systémové volání;uloženo na pozici 128 – běžně vyjadřované hexadecimálně – 80h), pomocí kterého programy provádějí běžné operace jako zápis, čtení a podobně.

4.1.3 Tabulka systémových volání

Tabulka systémových volání v Linuxu je pole ukazatelů na funkce, které jsou volány obslužnou rutinou přerušení 80h. Například, když program v uživatelském prostoru zavolá funkci write, provede se to, že se do registru EAX uloží číslo tohoto volání (je to pevně daná pozice v tabulce systémových volání) a vyvolá se přerušení 80h. Obslužná funkce tohoto přerušení spustí funkci, jejíž adresu najde na EAXté pozici v tabulce systémových volání.

V BSD systémech tabulka neobsahuje pouze ukazatele na funkce, uspořádání tabulky systémových volání je odlišné. Každý záznam v tabulce obsahuje počet argumentů systémového volání a ukazatel na funkci, která má stejný prototyp (počet a typ parametrů a návratovou hodnotu) pro všechna systémová volání. Ovšem až na to, že všechny procesy nemusí používat stejnou tabulku systémových volání jako je tomu v Linuxu, je princip stejný.

4.1.4 Moduly do jádra

Modul do jádra[1] je kus kódu, který se po připojení stane součástí jádra a provádí se tedy s úrovní oprávnění ring 0. S tímto oprávněním může vykonávat privilegované instrukce a přímo číst a zapisovat do paměti jádra. Může tedy za běhu měnit kód jádra. Moduly se používají jako ovladače zařízení nebo pro přidávání různých funkčností jádra. Více podrobností o modulech a jejich programování najdete v [16] nebo [17] .

4.1.5 Virtuální paměť jádra – soubor /dev/kmem

Ve speciálním souboru */dev/kmem* je namapována virtuální paměť jádra. Pozice v tomto souboru souhlasí se skutečnou adresou v paměti jádra. Superuživatel má právo do tohoto souboru zapisovat, získává tím tedy možnost za běhu měnit jádro, stejně jako by použil modul. Z uživatelského prostoru se však nemá dostupné adresy struktur a funkcí v kernelu, musí je tedy nějakým způsobem zjistit.

Zapisovat do tohoto souboru běžně není potřeba, proto možnost zápisu představuje jistou bezpečnostní slabinu. Získá-li útočník superuživatelská práva, má možnost za běhu upravit celý systém podle sebe. Zápis zakážeme např. zvýšením bezpečnostní úrovně (jedná-li se o systém BSD) nebo patchem do jádra. Změna přístupových práv k souboru, či samotné nastavení příznaku nic neřeší. Má-li útočník práva roota, může tato opatření zrušit.

4.2 Techniky změny jádra za běhu

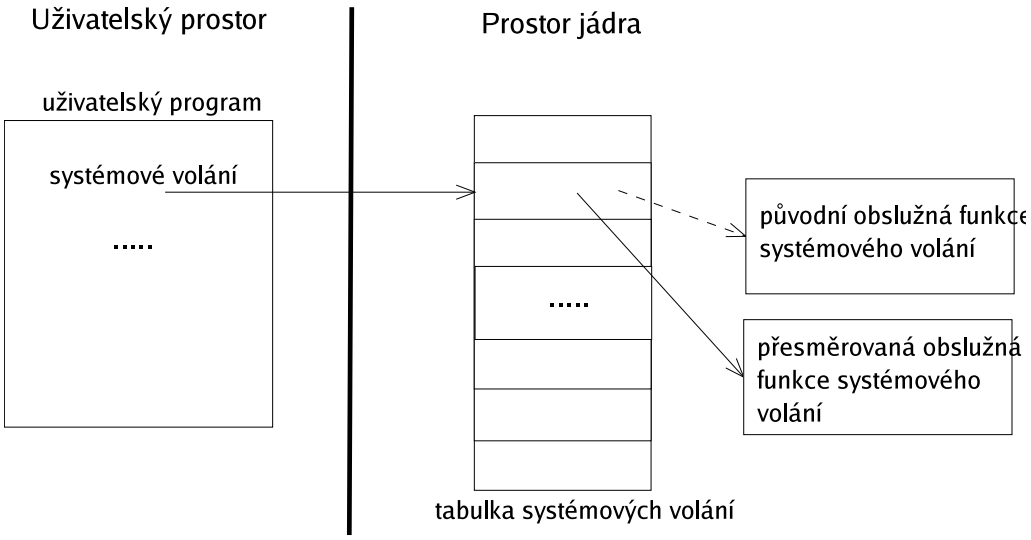
4.2.1 Hookování systémových volání

Hookování systémových volání je, dá se říci, zaběhnutý termín pro činnost, kterou se dosáhne toho, že jádro používá jinou, upravenou, obslužnou funkci daného systémového volání. Je to vlastně

[1] V Linuxu se označují jako LKM – loadable kernel module a např. v BSD KLD – dynamic kernel linker facility

přesměrování řízení z původní funkce na naší.

Nejjednodušší způsob, jak změnit systémové volání je přepsat jeho adresu v tabulce systémových volání na adresu jiné upravené funkce.



Obr. 1: Nejpoužívanější technika přesměrování (hookování) systémových volání

Paranoidnější útočník může vytvořit svojí upravenou kopii tabulky systémových volání a přinutit obslužnou rutinu přerušení 80h, aby používala tuto kopii namísto původní. Tím dosáhne toho, že původní tabulka zůstane nezměněna. Další a nejméně používaný způsob je přepsání začátku funkce daného systémového volání instrukcí skoku na vlastní funkci. Tabulka systémových volání zůstane neporušena, stejně tak obslužná rutina přerušení 80h.

Není to jedinná možnost, jak přesměrovat systémové volání. Rootkit může přepsat začátek obslužné funkce skokem na funkci vlastní. Pokud by v ní potřeboval volat tu původní, stačí její začátek opravit (nejdříve si těch prvních několik bajtů zálohoval), provést ji a po zkončení jí zase vrátit do stavu, kdy okamžitě předává řízení upravené funkci.

Upravenou funkci je samozřejmě potřeba nejdříve dostat do paměťového prostoru se stejnou úrovní oprávnění jako má jádro. To se nejčastěji dosahuje použitím modulu. Další způsob je přímý zápis do souboru */dev/kmem*. Útočník si musí zajistit možnost, jak z uživatelského prostoru alokovat paměť v prostoru jádra. To provede například tak, že nahradí adresu některého systémového volání za adresu funkce *kmalloc* a toto volání pro alokaci paměti využije. Do ní (tedy na daný offset do souboru */dev/kmem*) zapíše novou funkci a pak přepíše další ukazatel v tabulce systémových volání na adresu této funkce.

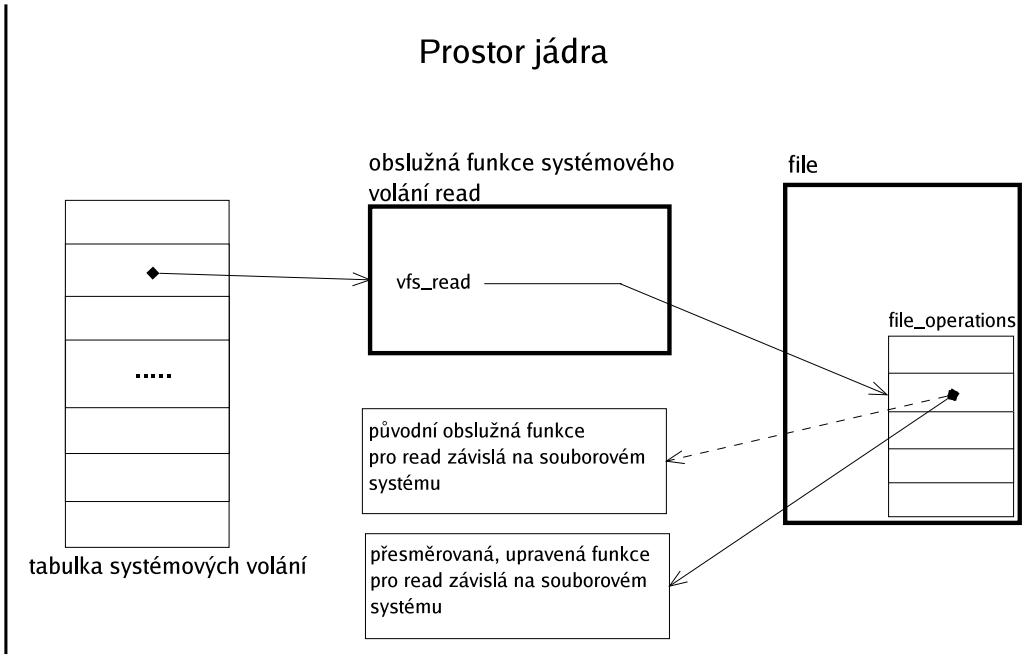
Podrobnější informace o této problematice najdete například v [?], [19] nebo [21].

4.2.2 Hookování funkcí jádra na úrovni *VFS*

VFS (virtual filesystem) je jakási vrstva abstrakce pro práci s různými souborovými systémy. Pro operace se soubory, které leží v různých soborových systémech je třeba provádět různý kód. Proto je tu VFS, který přináší jednotné rozhraní a o rozdíly mezi jednotlivými souborovými systémy

se postará za nás. Programátor pak jen používá stejné funkce například pro vypsání adresářové struktury a už ho nemusí zajímat, na jakém se nachází souborovém systému.

V praxi to vypadá tak, že každý soubor (resp. struktura nesoucí informace o něm, je-li otevřen) obsahuje sadu ukazatelů na operace s ním. Deklaraci této struktury (*struct file_ operations*) najdete v hlavičkovém souboru *linux/fs.h*. Takže pokud zavoláme například *read*, spustí se ta funkce, na kterou ukazuje ukazatel *read* ve struktuře *file_ operations* náležící souboru, z nějž se čte.



Obr. 2: Přesměrování funkce na úrovni VFS

Pokud tedy útočník tento ukazatel změní na vlastní funkci, má kontrolu nad čtenými daty (v případě *read*) a tabulka systémových volání zůstala nedotčena. Tuto techniku využívá jen malé množství dostupných rootkitů (vím pouze o jednom) o to však je nebezpečnější.

4.3 Kernelové rootkity

Existuje spousta automatických nástrojů, které skrývají útočníkovu přítomnost a aktivitu. Říká se jim *rootkity*. Určitá skupina rootkitů mění systémová volání popisovanými způsoby. Ve zbytku práce se budeme věnovat technikám, jak odhalit, že jádro je "napadeno" rootkitem. Nejznámější rootkity, které mění záznamy v tabulce systémových volání a používají modul do jádra jsou např. *Adore*, *Knark* nebo *KIS*. A snad jedinným známým zástupcem rootkitů, které zapisují přímo do */dev/kmem* je český *SucKit*, který navíc nemění adresy v tabulce přímo, ale používá vlastní upravenou kopii.

4.4 IDS, IPS

IDS (intrusion detection system) jsou systémy, které monitorují činnost systému a rozpoznávají pokusy o průnik. Často se kombinují se schopností předejít průniku, pak se těmto systémům říká IPS (intrusion prevention system).

Tyto systémy monitorují provoz a zaznamenávají podezřelé chování. Mohou například sledovat provoz sítě a rozpoznat skenování portů, dané spojení pak blokovat. Sledovat např. počet volání *fork* pro každý proces a v případě překročení určitých hranic vyhodnotí, že může jít o DoS útok a provinělý proces ukončí. Nebo rozpoznávat jiné "zvláštní" chování. Například pokud se uživatel rok přihlašuje v době od 9 do 17 hodin a najednou se několikrát přihlásí ve 2 hodiny ráno, je to minimálně podezřelé chování hodné varování administrátora.

Aby takový systém správně fungoval — nehlásil příliš planých poplachů a nepropouštěl příliš skutečných útoků bez povšimntí — je třeba správně nastavit meze, po jejichž překročení provede nějakou akci (varování, zamezení útoku, apod.). V této oblasti se začínají uplatňovat evoluční optimalizační algoritmy, pomocí nichž lze s vyžitím velkého množství různých příkladů útoku a legitimního chování najít optimální meze pro určení vážnosti situace. Více informací najdete v [9, 10].

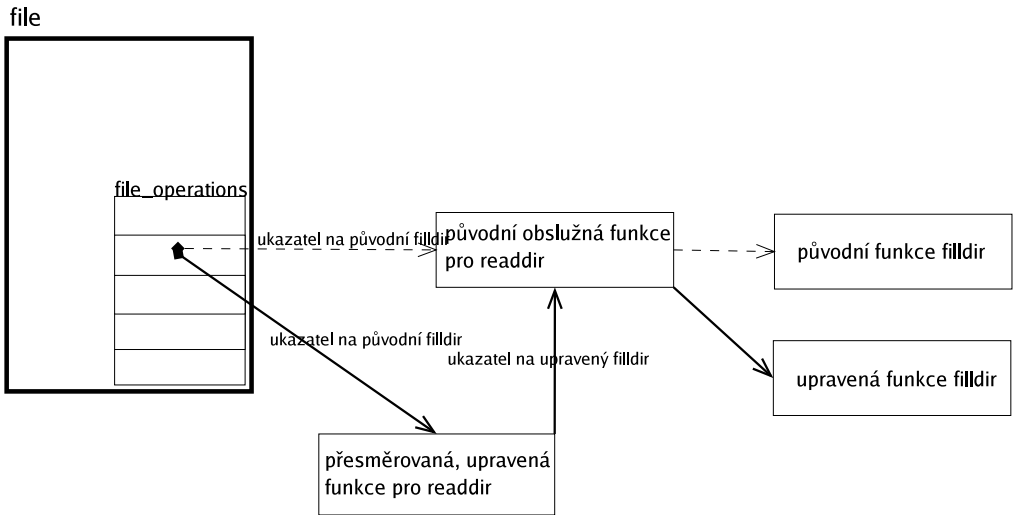
4.5 Skrývání souborů

4.5.1 Princip

Soubory se běžně skrývají přesměrováním systémového volání pro získání adresářové struktury – *getdents* případně *getdents64*. Tato funkce seznam soborů uloží do paměťi ve formě struktur *linux_dirent64*, které jsou uloženy jedna za druhou. Rootkit nahradí tuto funkci funkcí svou. Ta nejdříve zavolá původní systémové volání, pak projde postupně přečtené struktury, ověří, zda-li některá z nich neodpovídá souboru, který je potřeba skrýt, a pokud ano, vhodným způsobem ji z dat odstraní. Detailní popis této techniky najdete v [6].

Méně používaná o to však nebezpečnější technika je skrývání souborů změnou ukazatele na funkci *readdir* přímo ve struktuře obsahjící funkce pro práci s daným souborem (viz. 4.2.2). Tady je ovšem situace trochu komplikovanější. Tato funkce *readdir* má tři parametry. Důležitý je třetí parametr *filldir*, což je ukazatel na funkci. Tuto funkci používá *readdir* pro naplnění potřebých struktur. *filldir* je volána pro každý soubor ve čteném adresáři zvlášť. Důležité je to, že pokud *filldir* neudělá nic a zároveň vrátí hodnotu, která značí, že vše proběhlo v pořádku, nebude o daném souboru dále ani zmíňka. Skrytí souboru se tedy snadno provede tak, že donutíme *readdir*, aby používal upravenou funkci *filldir*. Ve ní se pouze zkontroluje název soboru a pokud je ho potřeba skrýt, vrátí se 0, pokud ne, provede se původní *filldir*.

Jak ovšem donutím *readdir* používat vlastní *filldir*? Jednoduše přepíšeme ukazatel na *readdir* ve *file_operations* na vlastní funkci a v ní pouze uložíme původní *filldir* (který dostaneme jako 3. parametr) a zavoláme původní *readdir* se upraveným *filldir* jako 3. parametr. Situace je znázorněna na obrázku 3.



Obr. 3.: Přesměrování funkcí VFS pro skrytí souboru

Čárkované šipky ukazují původní situaci, plné šipky ukazují situaci po přesměrování funkcí. Podrobnější vysvětlení této techniky najdete v [6, 15, 14, 20].

4.5.2 Detekce

Potenciální výskyt skrytých souborů poznáme podle změněného systémového volání *getdents64*. Pokud je však skrývání zařízeno jinak, např. přesměrováním příslušné VFS funkce, je situace horší. Ano, i zde bychom mohli po nainstalování systému uložit všechny ukazatele na VFS funkce pro práci se soubory na všech v systému používaných souborových systémech a pak jen prověřit, zda s touto zálohou souhlasí. Nejsem si však jistý, zda by tato snaha k něčemu byla. Je to jednak velmi neobecné řešení, které se stará o nepatrný zlomek dnešních rootkitů. Největší problém je ale v tom, že útočník nemusí změnit přímo ukazatel ve struktuře *file_ operations*. "Po cestě" od systémového volání až po provedení příslušné akce, je více funkcí. A v každé z nich je možné přečtená data zcenzurovat. Nemůžeme vědět, kde jaký ukazatel či skok na funkci útočník přepíše a tím si zajistí možnost měnit přečtená data.

Je tu však stále způsob, jak odhalit, že něco bylo změněno. Jde o techniku prezentovanou v elektronickém magazínu *Phrack* (viz. [23]). Nemusí být vždy stoprocentní, mění záznam v IDT, je náročná na programování, zato však odhalí jak změny systémových volání tak i změny hlouběji v kernelu — je tedy mnohem obecnější. Je založena na počítání instrukcí, které se provedou při provedení daného systémového volání. Samozřejmě i zde potřebujeme vědět, kolik které volání s danými parametry provádí instrukcí v době, kdy není útočníkem změněno. Navíc funkce obsahují spoustu podmínek, takže i v případě originální funkce se počet může měnit. Ale přibude-li např. 1000 instrukcí, je téměř jisté, že není něco v pořádku. Z principu této techniky vyplývá, že může pouze hlásit nesrovnalosti. Nedokáže určit, co je změněno a samozřejmě už vůbec není možné na jejím základě dát systém zpět do pořádku.

Program *chkrootkit* dokáže najít skryté podadresáře (ne soubory). Navíc nedokáže říct, jak se jmenuje, pouze kolik jich v daném adresáři je. Technika je založena na prostém principu – program si pomocí funkce *lstat* zjistí počet pevných odkazů, které v adresáři jsou a poté pomocí funkce *readdir* prochází obsah adresáře a počítá jeho podadresáře. Rodičovský adresář obsahuje o dva více pevných odkazů než je počet jeho podadresářů (obsahje totiž také odkaz na nadřazený adresář a na sebe). Pokud je podadresářů méně, znamená to, že jsou některé skryty.

4.6 Skrývání procesů

4.6.1 Princip

Princip je stejný jako v případě skrývání souborů. Program *ps* pro vypisování procesů totiž spoléhá na informace, které zjistí z */proc*. Proces se tedy skryje pouhým skrytím adresáře jména podle PID procesu v */proc*.

Navíc každý proces obsahuje ukazatel na další proces. Jsou tedy uspořádány v seznamu. Pokud je z něj vypojen, nebude mít ani záznam v */proc*.

4.6.2 Detekce

4.6.2.1 proces uložený v seznamu

Je-li proces skryt ukrytím soboru v */proc*, je stále uložen v linárním seznamu společně s ostatními. Ten je navíc cyklický, takže se z libovolného procesu postupně můžeme dostat na všechny ostatní. Adresa prvního procesu (je tím myšlen proces *swapper* s PID 0) je vždy stejná, jeho adresu exportuje jádro jako symbol *init_task_union*. Můžeme si tedy z */dev/kmem* postupně přečíst a vypsat všechny procesy v tomto seznamu. Pokud bychom použili modul do jádra, můžeme pro průchod všemi procesy použít makro *for_each_process* (nebo *for_each_task*, podle verze jádra). Toto makro samozřejmě nepoužívá systémová volání — čte struktry *task_struct* ze seznamu v paměti. "Vidí" tedy i skryté procesy, pokud jsou stále v seznamu.

4.6.2.2 proces vypojený ze seznamu

Jak proces najdeme, pokud byl ze seznamu vypojen? V takovém případě nezbývá než prohledat paměť jádra a pokusit se najít všechny procesy. Pameť můžeme prohledávat přímo (pomocí modulu) nebo přes soubor */dev/kmem*.

Jak je ale mezi všemi daty poznáme? Jádro má informace o každém procesu uložené ve struktuře *task_struct*. Najdeme ji ve zdrojových kódech jádra ve hlavičkovém souboru *linux/sched.h*. Mohli bychom se do jádra podívat na způsob alokování těchto struktur, zjistili bychom, že každá tato struktura je uložena na začátku paměťové stránky. Z toho vyplývá, že není třeba prohledávat celou paměť, bylo by to velmi neefektivní.

Když se podíváme na tuto struktru *task_struct*, uvědomíme si, jakých hodnot budou nabývat a toho využijeme při hledání. Například první položka *state* určuje stav procesu, je to čtyřbajtový long, ale příliš hodnot nabývat nemůže. -1 pro *unrunnable*, 0 pro *runnable* a číslo >0 pro zastavené procesy. Bežící proces tedy bude mít první 4 bajty ve struktře nulové.

Velikost paměťové stránky na intelovské architektuře je 4096 bajtů, stačí tedy číst každý 4096tý long (4B) v paměti jádra a kontrolovat, zda je nulový. Pokud ano, načteme ze stejné adresy celou strukturu a ověříme, zda souhlasí i některé další položky. Můžeme například kontrolovat příznaky procesu *flag*. Struktura také obsahje spoustu ukazatelů. Jde o strukturu jádra — ukazatele mohou ukazovat do paměti jádra (adresy větší 0xc0000000) nebo mohou mít hodnotu *null*. Spousta z ukazatelů ani hodnotu *null* běžně nikdy nemají.

Čím víc položek zkontrolujeme, tím máme větší šanci že, vyhledáme pouze to, co chceme, ale zvyšujeme riziko, že rootkit proces skryje změnou položky, kterou sice kontrolujeme, ale pro pro běh daného procesu nemá zásadní vliv. Přesto se může stát, že proniknou jiná data, než tyto struktry. V tomto případě jde již často správně rozpoznat podle názvu procesu, zda jde skutečně o proces nebo o irelevantní data. I to by se dalo kontrolovat programově, ovšem většinou to je zbytečné.

4.7 Skrývání modulů jádra

4.7.1 Princip

Program pro výpis připojených modulů v Linuxu (lsmod) provádí pouze úpravu a výpis souboru */proc/modules*. Starší LKM rootkity toho využívaly a zajišťovaly své krytí tím, že pozměnily systémové volání *read*, takovým způsobem, že při čtení z tohoto souboru se odfiltruje záznam něm. Čtoucí program informace o něm tedy ani nedostane.

Jiný způsob využívá faktu, že moduly jsou uloženy v paměti v cyklickém lineárním seznamu. Obsah tohoto seznamu jádro vypisuje do souboru */proc/modules*. Pokud se však modul z tohoto seznamu vypojí, nebude o něm nikde ani zmíňka. Toto je velmi snadná, účinná a tudíž často používaná technika.

4.7.2 Detekce

4.7.2.1 Starší technika

Pokud rootkit přesměrje *read* a upravuje data přečtená z */proc/modules*, využívá toho, že data byla čtena řádek po řádku. Funkce *read* tedy měla k dispozici celý řádek a bylo v ní snadné zjistit, zda jde o řádek s daným modulem. Tento způsob krytí se tedy velmi snadno obejde tím, že data budeme číst po jednotlivých bajtech např. příkazem

```
dd if=/proc/modules bs=1
```

Výpis stačí porovnat s výpisem příkazu *"cat /proc/modules"* a nesouhlasí-li, je hned jasné, že je něco v nepořádku. Rootkit se ale dokáže skrýt tak, že v souboru */proc/modules* uveden ani nebude. Pak nám tato technika nebude nic platná.

4.7.2.2 Pokročilá technika

I když je modul vypojen ze seznamu, v paměti být stále musí. Můžeme jej tedy, stejně jako skrytý proces, najít v */dev/kmem*. Informace o modulech nesou struktury *module*, jejíž definici najdeme v hlavičkovém souboru *linux/module.h*. Vidíme, že hned jako první položkou je *size_of_struct*, což je proměnná typu unsigned long a obsahuje velikost struktury. Ta je samozřejmě pro všechny moduly stejná (*sizeof(struct module)*). Stejným způsobem jako při hledání procesů prohledáváme paměť a ověřujeme, zda položky ve struktuře souhlasí. Můžeme kontrolovat různé flagy, zda obsahují korektní hodnoty, nebo třeba název — jde o ukazatel na pole znaků, takže pokud místo struktury *module* načteme nějaká jiná data, máme slušnou šanci, že čtení názvu skončí chybou *Bad address* — nízké adresy v *kmem* nejsou, proto při pokusu o čtení dat z nich vrátí read chybu.

Tato technika je vcelku rychlá a velmi účinná proti *dnešním* technikám skrývání modulů. Ovšem pokud si modul změní údaj o velikosti, je rázem "nezjistitelný". Je problematické najít takovou kombinaci ověřovaných prvků struktury, aby byly nalezeny všechny moduly a pouze moduly a přitom bylo nemožné tyto proměnné měnit. Můžeme se však tomuto stavu alespoň přiblížit.

4.8 Skrývání dalších informací

4.8.1 Princip

Kromě souborů a procesů chce útočník skrýt i jiné věci. Například informaci o tom, že je zrovna přihlášen (z výpisu *w* a *who*), jeho aktivní síťové spojení (z výpisu *netstat*) a podobně. Většinou

jde o cenzurování informací, které systém čte z různých souborů (např. z */var/run/utmp* nebo */proc/net/tcp*).

Nejčastěji se to dělá přesměrováním čtecí funkce, tedy systémového volání *read*. Je možné také přesměrovat i jinou funkci hlouběji v kernelu. Funkce se změní často tak, že je volána ta původní a v přečteném bufferu se vyhledá a odstraní nebo změní požadovaná informace.

Tímto způsobem je možné skrývat informace o přihlášených uživatelích, síťových spojeních, připojených modulech, nebo třeba i virtuální vymazání uživatele ze systému — pro všechny procesy kromě *sshd* nebo *login* skrýt řádek z */etc/passwd* a */etc/shadow*.

4.8.2 Detekce

Pokud známe princip skrytí přihlášeného uživatele, je jeho objevení velmi snadné. Programy jako *who* čtou informace ze souboru */var/run/utmp*. Tento obsahuje struktury *utmp*[2] a je pochopitelné, že *who* čte najednou celou tuto strukturu. Funkce *read* ji tedy uloží do určené paměti. Změněný *read* tuto paměť prohledá a pokud zjistí, že jde o strukturu s informacemi o uživateli, který je skryt, pak buffer vynuluje a vrátí 0 (návratová hodnota znamená počet přečtených bajtů). Snadnou obranou je nečíst soubor po celých strukturách, ale například po bajtech. *read* tak bude vždy mít k dispozici pouze 1B, v němž toho příliš nepozná a nemá tedy co skrýt.

Stejný princip platí u všech technikách, které používají jednoduché vyhledávání dat v přečteném bufferu.

Pokud se navíc cenzurování paměti s přečtenými daty provádí přes hooknuté systémové volání, objeví ji i kontrola integrity systémových volání.

4.9 Kontrola integrity IDT

4.9.1 Nebezpečí útoku

Zadní vrátka se dají udělat téměř všude. V [22] je popisován způsob, jak měnit obslužné funkce vyjímek a přerušení ke svému prospěchu. Adresy funkcí pro obsluhu vyjímek jsou uloženy v IDT. Je možnost také přesměrovat obsluhu přerušení 80h – systémových volání.

4.9.2 Detekce

K detekování takových změn je snadné kontrolovat záznamy v IDT s dříve uloženou zálohou. To nemusí být vždy výhodné, správce systém musí myslet do budoucna a po nainstalování tuto zálohu vytvořit. Ovšem žádný způsob kontroly integrity IDT, který není založen na porovnávání zálohy (obsahu nebo kontrolních součtů) a aktuálního stavu, nejspíš ani neexistje.

Správce by měl mít takovou zálohu uloženou na místě, kde jí útočník nemůže změnit (nejlépe CD nebo jiné externí záznamové médim).

4.10 Kontrola integrity tabulky systémových volání

4.10.1 Nebezpečí útoku

Drtivá většina kernelových rootkitů mění záznamy v tabulce systémových volání a tím zajišťjí, že systém provádí upravené funkce. Přepsáním adresy v této tabulce má útočník možnost kontrolovat například, které soubory či procesy budou viditelné a podobně. Systémová volání používají všechny programy v uživatelském prostoru, takže jediná změna ovlivní celý systém.

[2] viz. man utmp

4.10.2 Detekce

Stejně jako u IDT se kontroluje obsah tabulky se zálohou pořízenou po instalaci systému. Je to jednoduché a spolehlivé, pokud však máme zálohu. Je také samozřejmě možné využít techniky založené na počítání instrukcí, popisované v 4.5.2.

4.11 Prevence změny jádra za běhu

Nejsnazší preventivní opatření je používat monolitické jádro bez podpory modulů. Tím samozřejmě přijdeme o všechny výhody, které modulární jádro přináší. Navíc tím nevyřešímě vše, protože je stále možné zapisovat do */dev/kmem*. Proto toto opatření předem zavrhneme. Položme si otázku, zda potřebujeme na běžícím serveru přidávat za běhu další ovladače. Nejspíš nepotřebujeme. Při konfiguraci si připojíme vše potřebné a další přidávání je tedy nežádoucí.

U BSD systémů to lze elegantně řešit zvýšením bezpečnostní úrovně alespoň na 1. Zabijeme tím dvě i více much jednou ranou. Existují techniky, jak přidávat moduly do jádra zápisem do souboru */dev/kmem*[3]. Jak jsme se dozvěděli v 2.3.3, je už od securelevelu 1 zakázáno do tohoto souboru zapisovat a to kýmkoli, včetně superuživatele. Navíc je při nastavení této úrovně zakázáno přidávat moduly. Prakticky tím zamezíme jakékoli možnosti změny jádra za běhu — všechny moderní jaderné rootkity si tedy vylámou zuby.

Linux lze zabezpečit použitím bezpečnostních patchů, jako např. *grsecurity*. Ty ovšem musí být správně nakonfigurované — pro zamezení těchto útoků je vhodné zakázat zápis do souboru */dev/kmem* a zakázat připojování nových modulů do jádra. Existují také security moduly, které implementují bezpečnostní úrovně podobné BSD systémům.

4.12 Možnost odstranění nesrovnalostí

Skryté informace sice můžeme odhalit, ale moc s tím běžnými prostředky neuděláme. Ostatně proto byly tyto techniky navrženy. Skrytý modul nelze odpojit pomocí *rmmod*, skrytý soubor nelze smazat a proces ukončit. Leda bychom si vytvořili speciální nástroje, které se nespoléhají na prostředky dostupné v uživatelském prostoru.

Jednodušší cesta je zrušit jejich krytí. Pokud rootkit využívá LKM skryté vypojením ze seznamu modulů, můžeme se pokusit jej do tohoto seznamu opět zapojit. Pak byl modul viditelný pro nástroje jako *lsmod* a *rmmod*, mohli bychom jej tedy odstranit z jádra. Tím bychom zařídíli zrušení všech přesměrovaných funkcí a slušně vychovaný modul před odpojením vrátí systém do původního stavu. Pokud by to neudělal, systém by se zhroutil hned potom, co bychom se snažili využít funkci, kterou modul přesměroval. Po svém odpojení totiž jádro uvolní veškerou paměť, kterou modul využíval, tedy všetně všech upravených funkcí. Kdyby nevrátil ukazatele na původní funkce, došlo by při pokusu o jejich provedení k předání řízení do nealokované paměti, což samozřejmě jádro nemá rádo a okamžitě by se zastavilo.

Pokud rootkit nevyužívá modul, můžeme dát do pořádku ukazatele, které změnil. Máme-li zálohované adresy v tabulce systémových volání (jako že máme, protože detekční nástroj je využívá pro porovnání s aktuálními adresami), není problém změněné záznamy zpět přepsat na původní hodnotu. Pokud si rootkit svou činnost zajišťoval pouze přesměrovanými systémovými voláními, kompletně ho tímto vyřadíme z činnosti. Bude v systému pořád, jen už nebude mít na jeho funkci žádný vliv.

[3] viz. např. [19]

Kapitola 5

Automatická kontrola integrity systému

Nedílnou součástí této práce je program[1], který je navržen pro automatickou kontrolu integrity systému. Implementuje popisované techniky detekce nejrůznějších úprav jádra, které rootkity v drtivé většině používají. Testy tedy nejsou cíleně psány pro odhalení konkrétních rootkitů, ale obecněji, aby objevily co nejvíce známých i neznámých rootkitů, backdoorů a nejrůznějších utilit.

5.1 Návrh

Aby program byl obecný a bylo snadné ho používat v různých operačních systémech, je navržen modulárně. Skládá se z jádra a modulů. Jádro programu je psáno s ohledem na přenositelnost mezi POSIXovými systémy, moduly jsou dynamicky linkovatelné knihovny.

Jádro předává při spuštění konkrétního testu informace o operačním systému, verzi, v případě BSD i hodnotu bezpečnostní úrovně, režimu činnosti apod. Modul si tak může ověřit, zda program běží na systému, pro který je napsán. Každý modul implementuje *test* určitých příznaků napadení systému, jako např. kontrolu integrity tabulky systémových volání. Testy jsou řazeny do kategorií, přičemž každý z nich může být ve více kategoriích. Jádro pak může spustit testy pouze z určité kategorie (např. *kernel, backdoor, linux, BSD* a pod.)

Jádro loguje výsledky do jednotných logů, moduly jádru informace předávají pomocí fronty zpráv. Používají se tři logovací soubory. Jeden pro poruchy integrity systému, další pro varování a poslední pro informace.

Testy jsou ve fázi "proof-of-concept" (praktická ukázka toho, že popisované principy fungují), jsou tedy navrženy pro konkrétní systém, a to Linux řady 2.4. Protože hlavní program poskytuje informace o OS, na kterém běží, není problém podporu dalších systémů dodělat.

5.2 Popis testů

Testy jsou navrženy tak, aby nepotřebovaly podporu modulů jádra (až na hledání skrytých procesů, to je řešeno s použitím LKM kvůli přenositelnosti alespoň v rámci jedné řady jádra). Princip jejich činnosti je podrobně popsán výše v kapitolách 4.5 – 4.8.

[1] můžete jej najít na přiloženém CD v adresáři *intrusion_detector/*

- **hledání skrytých modulů** – vyhledává veškeré moduly přímo v souboru */dev/kmem* a výsledky porovnává s výstupem programu *lsmod*.
- **hledání skrytých procesů** – protože se struktura *task_ struct* často v různých verzích jádra liší, vyžil jsem pro výpis procesů LKM, které pomocí makra *for_ each_ process* vypíše seznam procesů s jejich PID do zvláštního souboru v */proc*. Test tento seznam porovná s výpisem programu *ps*.
- **kontrola integrity IDT** – porovná adresy obslužných funkcí a DPL jednotlivých přerušení s dříve uloženou zálohou. Je ji tedy potřeba vytvořit po nainstalování systému. Změněné záznamy je schopen obnovit podle původní zálohy.
- **kontrola adresy tabulky systémových volání** – prověří, zda obslužná rutina přerušení 80h používá správnou tabulku systémových volání. Opět kontroluje s dříve uloženou zálohou. Adresu umí vrátit do původního stavu.
- **kontrola integrity tabulky systémových volání** – porovnává adresy všech systémových volání s dříve uloženou zálohou. Změněné adresy lze nahradit za původní.
- **kontrola zda jsou viditelní všichni přihlášení uživatelé** – porovnává informace získané pomocí programu *who* a skutečné informace v souboru */var/run/utmp* čtené po jednotlivých bajtech.

5.3 Testy účinnosti detekčního programu

Nainstaloval jsem si postupně několik dostupných kernelových rootkitů a testoval, zda a podle jakých příznaků je detekční nástroj najde. Rootkitů jsem nevolil velké množství, protože spousta z nich pracuje na stejném principu. Proto jsem zvolil několik odlišných typů. Všem rootkitům jsem nechal skrýt proces a sebe samého.

To by však dostatečně detektor neotestovalo — předpokládám, že nalezne všechny rootkity, podle přesměrovaných systémových volání, skrytého procesu či modulu v jádře. Proto jsem otestoval několik experimentálních programů, které využívají důmyslnější techniky ukrývání, než běžné rootkity nebo některé techniky, které testované rootkity neobsahovaly (např. skrytí přihlášeného uživatele).

5.3.1 Stručný popis testovaných rootkitů

- **Adore** - velmi starý rozšířený LKM rootkit. Skrývá svůj modul vypojením ze seznamu, patchuje systémová volání a to přímo záměnnou ukazatele v tabulce systémových volání. Jeho detekce by neměla dělat nejmenší problémy. Další rootkity stejného typu jsou například *knark*, *KIS*, *heroin* a další.
- **SucKit** - jediný z testovaných rootkitů, který zapisuje přímo do */dev/kmem* namísto použití modulu do jádra. Také patchuje systémová volání, ovšem používá vlastní tabulku systémových volání, kterou donutí obslužnou funkci přerušení 80h používat. S touto možností detekční nástroj počítá, měl by být nalezen podle přesměrovaných systémových volání. Pro své přežití po restartu počítače používá ošklivý trik — nahradí program *init* sám sebou, původní init uloží jinam a po spuštění sebe sama, teprve spustí init. SucKit je tedy prvním zaváděným procesem v systému po restartu. Po restartu by kromě změněné adresy tabulky systémových volání měl být spolehlivě odhalen podle skrytého procesu *init{koncovka skrytých procesů}*.

- **Adore NG** - z testovaných rootkitů nejnovější a poměrně pokročilý. Již nepřesměrovává systémová volání — mění funkce na úrovni VFS. Jeho odhalení může dělat problémy, pokud není rootkit aktivní (tedy pokud nic nedělá). Jinak by měl být odhalen podle skrytých procesů a neviditelného modulu v jádře.
- **Další testovací nástroje** - vyzkoušel jsem také rootkit, který zapisuje přímo do */dev/kmem*, ale mění systémová volání přímým zápisem do jejich kódu – na začátku obslužné funkce skočí na vlastní funkci. Protože nepoužívá LKM, bude jej možné odhalit pouze podle následků jeho činnosti, např. podle skrytých procesů. Tento nástroj je pouze "proof-of-concept", takže nedělá nic jiného, než skrytí sama sebe. V dalším textu se na tento program budu odkazovat jako na program *Test 1.*
  Další nástroj ukrývá přihlášeného uživatele přesměrováním VFS funkce pro čtení ze souboru. Používá ovšem modul do jádra, takže by měl být odhalen, pokud ho skryjeme (sám neobsahje prostředky pro své ukrytí). Samozřejmě pak detekční nástroj musí zaznamenat skryté uživatele. Tento program bude označován jako *Test 2.*

5.3.2 Nalezené stopy po rootkitech

Postupně jsem vybrané rootkity zkompiloval a nainstaloval do systému. Každý rootkit jsem skryl (SucKIT se ukryje automaticky, k Adore je přiložen zvláštní modul) a nechal jím ukrýt běžící shell. Potom jsem spustil detekční program a sledoval, jaké změny v systému zaznamenal.

Stručný popis instalace jednotlivých rootkitů, způsobu, jakým testy byly prováděny a způsobu, jak nainstalované rootkity odstranit, najdete na přiloženém CD v souboru *rootkits/README.*

Seznam příznaků, které byly objeveny u jednotlivých rootkitů:

Adore

- skrytý modul *'adore'*
- skrytý proces
- 15 změněných záznamů v tabulce systémových volání, mezi nimi getdents64, fork, clone, write, což jsou typické cíle těchto typů rootkitů

SucKIT

- skrytý proces *'sk'*
- skrytý proces – tedy libovolný proces, který jsme pomocí rootkitu schovali
- 25 změněných systémových volání – nechává sice původní tabulku systémových volání v klidu a úpravy provádí ve vlastní tabulce, detekční nástroj však kontroluje vždy tabulku, která je právě používaná obslužnou funkcí přerušení 80h. Takže na něj tento trik neplatí.

Adore NG

- skrytý modul *'adore_ng'*
- skrytý proces a jeho potomci

Test 1

- skrytý proces

Test 2

- skrytí uživatelé

5.3.3 Shrnutí testů

Byly nalezeny všechny testované rootkity. To však zejména díky tomu, že detekční program byl navržen se znalostí technik, které používají. Pokud by naopak autor rootkitu znal techniky, které používám pro jejich detekci, nemusel by být velký problém najít způsob, jak je obejít. Například rootkit AdoreNG spolehlivě najdeme podle skrytého modulu v jádře. Ve 4.7.2.2 vidíme, že u každého potenciálního modulu, který v paměti najdeme, kontrolujeme první položku struktury, zda obsahuje číslo rovno *sizeof(struct module)*. Pokud si tedy rootkit ve vlastní struktře tento údaj přepíše na jiné číslo, bude rázem pro nás neviditelný.

Tento nástroj je tedy vhodný pro detekce rootkitů, které nejsou speciálně napsány tak, aby byly "imunní" vůči jeho testům. Což jsou dnes prakticky všechny, které jsou volně dostupné[2].

[2] mám samozřejmě na mysli pouze kernelové rootkity

Kapitola 6

Závěr

Teoretická část shrnuje nejčastěji využívané mechanismy pro zajištění bezpečnosti. Spoustu dalších informací můžete najít v poskytnutých zdrojích či na domovských stránkách projeků zabývajících se zvýšením bezpečnosti na úrovni jádra systému.

V praktické části práce jsem navrhnul sadu testů, které spolehlivě rozpoznají změny v jádře prováděné drtivou většinou dnešních kernelových rootkitů, backdoorů či jiných nástrojů, které podporují útočníky v jejich neoprávněné činnosti. Tyto testy jsou implementovány v detekčním nástroji a jejich účinnost jsme si pak mohli ověřit na několika typech odlišných rootkitů.

Hledal jsem programy podobného zaměření a našel jsem jich pouze několik. Snad jen jeden je víceméně stejného typu (*kstat*) — hledá obecné příznaky napadení a tím dokáže odhalit i nové nebo neznámé rootkity. Většina ostatních obsahují spíše řadu testů zaměřených pro nalezení konkrétních rootkitů ze své databáze, podle jejich implicitních příznaků.

Toto byla pouze jedna z mnoha oblastí bezpečnosti jádra. Pokračování v diplomové práci by se mohlo zaměřit na jiné, ne však zcela vzdálené, téma – systém pro realtimovou detekci a zamezení napadení, který využívá moderní postupy, jako například optimalizaci své činnosti využitím evolučních algoritmů, či využití LSM frameworku k implementaci některých využitelných bezpečnostních prvků, které v Linuxu zatím chybí.

Literatura

[1] Carnahan, L.: Security in open systems. Dokument dostupný na URL http://csrc.nist.gov/publications/nistpubs/800-7/ (květen 2005)

[2] Lucas, M.: Síťový operační systém FreeBSD. 2003, Computer Press 2003.

[3] Adress space layout randomization. Dokument dostupný na URL http://pax.grsecurity.net/docs/aslr.txt (květen 2005)

[4] FreeBSD handbook. Dokument dostupný na http://www.freebsd.org/doc/en_US.ISO8859-1/books/handbook/ (květen 2005)

[5] LSM Framework. Dokument dostupný na URL http://lsm.immunix.org/docs/overview/framework.html (duben 2005)

[6] Hýsek, J.: Linuxový rootkit. magazín Hakin9. 2004/2, 3. Dostupný na URL http://trace.dump.cz/papers.php (květen 2005)

[7] Hýsek, J.: Detekce kernelových rootkitů. el. časopis Prielom 22. Dokument dostupný na URL http://hysteria.sk/prielom/22/#2 (duben 2005)

[8] Kadlec, J.: Forenzní analýza digitálních dat [semestrální práce], Universita Hradec Králové. Dokument dostupný na URL http://jose.dump.cz/files/digital_forensics.pdf (duben 2005)

[9] Kadlec, J.: Detekce průniku užitím inteligentního systému pro podporu rozhodování. [semestrální práce], Univerzita Hradec Králové
Dokument dostupný na URL http://jose.dump.cz/files/IDS_DSS.pdf (duben 2005)

[10] Pluskal, T.: Intrusion detection system based on process behavior rating. [diplomová práce], Karlova univerzita v Praze.
Dokument dostupný na URL http://plusik.pohoda.cz/thesis/thesis.pdf (duben 2005)

[11] POSIX Security Interfaces and Mechanisms. Dokument dostupný na URL http://csrc.nist.gov/publications/nistpubs/800-7/node17.html (duben 2005)

[12] How to break out of a chroot() jail. Dokument dostupný na URL http://www.bpfh.net/simes/computing/chroot-break.html (duben 2005)

[13] Linux Capabilities FAQ 0.2. Dokument dostupný na URL http://ftp.kernel.org/pub/linux/libs/security/linux-privs/kernel-2.4/capfaq-0.2.txt

[14] Hýsek, J.: Vetřelec v systému Linux [*slajdy*]. Dokument dostupný na URL http://trace.dump.cz/papers/intruder.pdf (duben 2005)

[15] Hýsek, J.: Rootkity založené na hookování VFS. Dokument dostupný na URL http://trace.dump.cz/papers/vfs_hooking (duben 2005)

[16] Salzman, J. P, Pomerantz, O.: The Linux Kernel Module Programming Guide. Dokument dostupný na URL http://www.faqs.org/docs/kernel/ (květen 2005)

[17] Reiter, A.: Dynamic Kernel Linker (KLD) Facility Programming Tutorial. Dokument dostupný na URL http://www.daemonnews.org/200010/blueprints.html (květen 2005)

[18] sd: Obcházení security patchů. el časopis Prielom 19. Dokument dostupný na URL http://hysteria.sk/prielom/19/#3 (květen 2005)

[19] sd: Patchování linuxového /dev/kmem. El. časopis Prielom 17. Dokument dostupný na URL http://hysteria.sk/prielom/17/#2 (květen 2005)

[20] palmers: Advances in kernel hacking. El. časopis Phrack. Dokument dostupný na URL http://www.phrack.org/show.php?p=58&a=6 (květen 2005)

[21] pragmatic: (nearly) Complete Linux Loadable Kernel Modules. Dokument dostupný na URL http://reactor-core.org/linux-kernel-hacking.html (květen 2005)

[22] kad: Handling Interrupt Descriptor Table for fun and profit. El. časopis Phrack. Dokument dostupný na URL http://www.phrack.org/show.php?p=59&a=4 (květen 2005)

[23] Rutkowski, J.K.: Execution path analysis: finding kernel rootkits. El. časopis Phrack. Dokument dostupný na URL http://www.phrack.org/show.php?p=59&a=10 (květen 2005).

[24] Intel Architecture, Software Developer′s manual volume 3: System programming. 1999. Dokument dostupný na URL http://www.intel.com/design/pentiumii/manuals/24319202.pdf (květen 2005)

[25] Durden, T.: Bypassing PaX ASLR protection. El. časopis Phrack. Dokument dostupný na http://www.phrack.org/show.php?p=59&a=9 (květen 2005)

[26] Manuálová stránka *securelevel*. Dostupná na URL http://mirbsd.bsdadvocacy.org/cman/man7/securelevel.htm (květen 2005)

[27] Manuálová stránka *chflags*. Dostupná na URL http://mirbsd.bsdadvocacy.org/man1/chflags.htm (květen 2005)

[28] Manálová stránka *chattr*. Dostupná na URL http://linux.com.hk/PenguinWeb/manpage.jsp?section=1&name=chattr (květen 2005)

[29] Domovská stránka projektu *grsecurity* – http://www.grsecurity.net

[30] Domovská stránka projektu *LIDS* – http://www.lids.org

[31] Domovská stránka projektu *RSBAC* – http://www.rsbac.org

[32] Domovská stránka projektu *Medusa* – http://medusa.terminus.sk

[33] Domovská stránka projektu *chkrootkit* – http://www.chkrootkit.org

[34] Zdrojový kód programu *kstat* – http://www.s0ftpj.org/tools/kstat24_v1.1-2.tgz

[35] Zdrojový kód rootkitu *SucKIT* – //http://packetstormsecurity.nl/UNIX/penetration/rootkits/sk-1.3a.tar.gz

[36] Zdrojový kód rootkitu *Adore* – http://packetstormsecurity.org/groups/teso/adore-0.42.tgz

[37] Zdrojový kód rootkitu *KIS* – http://packetstormsecurity.org/UNIX/penetration/rootkits/kis-0.9.tar.gz

[38] Zdrojový kód rootkitu *Knark* – http://packetstormsecurity.org/UNIX/penetration/rootkits/knark-2.4.3.tgz

[39] Zdrojový kód rootkitu *Adore NG* – http://packetstormsecurity.org/groups/teso/adore-ng-0.41.tgz

[40] Elektronický časopis *phrack* – http://www.phrack.org

[41] Elektronický časopis *prielom* – http://hysteria.sk/prielom