Masarykova univerzita
Fakulta informatiky

Bezpečnost v prostředí operačního systému UNIX

diplomová práce

Vlastislav Dohnal

Duben 2000

Prohlašuji, že tato práce je mým původním autorským dílem, které jsem vypracoval samostatně. Všechny zdroje, prameny a literaturu, které jsem při vypracování používal nebo z nich čerpal, v práci řádně cituji s uvedením úplného odkazu na příslušný zdroj.

Na tomto místě bych chtěl poděkovat vedoucímu své diplomové práce Mgr. Michaelu Mrákovi za pomoc, odborné vedení a trpělivost při vytváření tohoto dokumentu. Dále bych chtěl poděkovat všem svým spolubydlícím za morální podporu.

Shrnutí

Tato diplomová práce se zabývá problém bezpečnosti operačního systému UNIX. Je zde popsáno několik možných rozšíření základního bezpečnostního modelu tohoto operačního systému a zhodnocení jejich přínosu.

V praktické části je implementován vlastní program pro detekci zranitelných míst. Tento program umožňuje psát a spouštět vlastní testy detekující bezpečnostní slabiny systému.

Klíčová slova

bezpečnost, bezpečnostní model UNIXu, rozšíření modelu bezpečnosti, metody detekce narušení bezpečnosti, přístupová práva, ACL, capabilities

Obsah

Kapitola 1

Úvod

Počítače se v současnosti stávají stále důležitější součástí našeho života a začínají se používat v řadě různých oborů. S tím souvisí stále zvyšující se množství informací, které počítače uchovávají a zpracovávají. V dnešní době se informace stává cennější než vlastní počítače a jejich software, to láká různé lidi k získání takových informací. Někteří patří do skupiny zvědavců, které pouze zajímá co a kde je, další patří mezi sběrače, kteří se domnívají, že se jim jakákoli informace může hodit. Další skupinou jsou vandalové, kterým jde pouze o zničení informace. Poslední skupinou jsou profesionálové, ti si informaci zjistí a bez zjevných stop o návštěvě systému mizí – většinou informace nehledají pro sebe a neničí je. Z těchto důvodů se zabýváme otázkou bezpečnosti informačních systémů.

Bezpečnosti je velice široký pojem, který lze ho definovat mnoha způsoby. Tato diplomová práce se snaží čtenáři osvětlit problematiku bezpečnosti v prostředí operačního systému UNIX. Jejím cílem není dát podrobný návod, jak zabezpečit operační systém, ale ukázat problémy, s jakými se může uživatel a hlavně správce setkat. Součástí této práce je implementace systému, který má usnadnit lokalizaci problematický míst, které jsou zneužitelné k průniku do systému. Následuje stručný obsah jednotlivých kapitol.

Počítačová bezpečnost není pouze ochrana informací, ale také ochrana hardwaru, softwaru počítače,... Následující kapitola vysvětluje co je to bezpečnost, čím se zabývá a jaká jsou různá nebezpečí, na která musíme brát ohled.

Ve třetí kapitole se čtenář seznámí se základním bezpečnostním modelem operačního systému UNIX. Také jsou zde diskutovány možné vlivy na bezpečnost celého systému.

Možná rozšíření modelu popisuje čtvrtá kapitola. Tato rozšíření nejsou zcela jistě všechna. Tato kapitola má pouze ukázat, jakým směrem je možné jít v oblasti zabezpečení.

V páté kapitole jsou zmíněny standardní metody detekce průniků do systému, které operační systém UNIX správci i uživatelům poskytuje.

Poslední, šestá kapitola obsahuje popis systému, který byl vytvořen pro snadnější a pohodlnější analýzu bezpečnosti systému. Systém poskytuje pouze prostředí pro psaní vlastních testů bezpečnosti. Hlavním cílem programu je dát administrátorovi systému nástroj, s jehož pomocí snadno a rychle otestuje systém na bezpečnostní slabiny a který zobrazí zprávu o nalezených problémech.

Kapitola 2

Bezpečnost

Co je to bezpečnost

Pojmem *bezpečnost* rozumíme ochranu všeho co souvisí s počítačem – hardware a software počítače, tiskárny, komunikační média, disky,... Překvapující může být, že bezpečnost se zabývá také ochranou budov. Nejdůležitějším cílem bezpečnosti je ochrana informací, které jsou v počítači uloženy [17].

Bezpečnost je dána zajištěním tří aspektů: *důvěrnosti, integrity a autentičnosti, dostupnosti.*

- důvěrnost (tajemství) – přístup k informacím mají pouze autorizované subjekty
- integrita – systém musí udržovat uložené informace konzistentní a neporušené, dále nesmí povolit změnu informace bez autorizace
- autentičnost – lze ověřit původ informace zjištěním, kdo ji vytvořil nebo poslal
- dostupnost – zachování dostupnosti služeb a informací pro autorizované subjekty; dojde-li k výpadku, musí být systém schopen rychlé a úplné obnovy

Bezpečnostní politika je souhrn pravidel zajišťujících *bezpečnost*, tj. určuje způsob správy, distribuce a uchování informace. Žádný počítač není dokonale zabezpečený. Bezpečnostní politika pouze snižuje pravděpodobnost, že útok na systém bude úspěšný, nebo zajišťuje, že hacker bude potřebovat více prostředků na prolomení bezpečnostních bariér.

Zranitelné místo

Zranitelné místo je místo v systému využitelné k útoku. Rozlišujeme několik typů zranitelných míst:

- fyzické – do budov, kanceláří se lze vloupat a zničit zařízení nebo zcizit disky, pásky, výstupy tiskáren,... Obranou jsou poplašné zařízení, bezpečnostní služby, biometrické snímače,...

- přírodní – oheň, povodeň, zemětřesení, blesk, výpadek proudu – tyto pohromy mohou zničit vybavení, data – není proti nim žádná obrana. Problémy způsobuje i obyčejný prach.
- hardware a software – porucha ochrany paměti, chyby v návrhu a implementaci software,... Nesprávná montáž hardware, nesprávné propojení komponent, chybná nebo neúplná instalace software, toto všechno může vést ke vzniku zranitelných míst.
- média – krádeže nebo zničení disků, pásek, tiskových výstupů.
- vyzařování – elektronická zařízení vyzařují elektromagnetické záření, zachycené záření může být analyzováno.
- lidský faktor – nejvážnější zranitelné místo ze všech možných, zaměstnanci jsou uplatitelní (prozradí hesla, otevřou dveře,...).

Hrozba

Hrozbou rozumíme možnost využití *zranitelného místa* k útoku. Rozlišujeme tři kategorie: *přírodní, úmyslné a neúmyslné.*

- přírodní (fyzické) – povodeň, výpadek proudu, oheň, zemětřesení,... – prevence je obtížná, ale lze je poměrně rychle objevit (hlásiče požáru,...). Následky škod a následné znovu obnovení provozu řeší tzv. *havarijní plán.*
- neúmyslné – nebezpečí vzniklé neznalostí, nedbalostí nebo opomenutím neškoleného správce nebo uživatele. Mnohem více informací je zničeno neznalostí než zlomyslností.
- úmyslné
  - vnější útočníci – hackeři, špioni, teroristé, konkurence, kriminální živly,...
  - vnitřní útočníci – 80% všech útoků je vedeno vlastními zaměstnanci (např. propuštěný zaměstnanec)
  - kombinace obou typů – vůbec nejnebezpečnější

Protiopatření

Obranou proti útokům jsou *protiopatření.* Protiopatřeními odstraňujeme zranitelná místa, snižujeme riziko jejich využití nebo minimalizujeme náklady na napravení škod vzniklých zneužitím zranitelného místa.

Kapitola 3

Bezpečnostní model Unixu

UNIX je víceúlohový, víceuživatelský operační systém [3]. To znamená, že na jednom počítači může pracovat současně více uživatelů a že lze současně spustit více úloh. Proto je jednou z jeho základních funkcí zabránit ve vzájemném ovlivňování uživatelů a spuštěných úloh.
„UNIX nebyl od počátku navrhován jako bezpečný. Byl navržen s nezbytnými rysy, které umožňují bezpečnost zajistit.“, Dennis Richie [8].

3.1 Účty a hesla

Každý uživatel je vůči systému identifikován svým uživatelským jménem. Znalost hesla uživatele autentizuje. Heslo představuje sdílené tajemství mezi počítačem a uživatelem. Uživatelské jméno, heslo a další informace tvoří společně *účet*.

UNIX má všechny údaje o účtech uloženy v souboru `/etc/passwd`. Každý řádek souboru obsahuje několik základních údajů (uživatelské jméno, heslo, skutečné jméno, domovský adresář, UID, GID,...).

Příklad `/etc/passwd`:

```
root:fi2SdG83IBrs5:0:0:Administrátor:/root:/bin/bash
daemon:*:1:1::/tmp:
novak:i4awJRX2qHMWf:505:100:Josef Novák:/home/novak:/bin/bash
```

První dva účty v příkladu jsou systémové, třetí je uživatelský. Na účet se jménem `daemon` se není možné přihlásit, protože má místo hesla hvězdičku (hvězdička neodpovídá žádnému zakódovanému heslu). Popis jednotlivých položek obsahuje tabulka 3.1.

Hesla jsou zakódována funkcí `crypt(3)`. Algoritmus této funkce je založen na upraveném symetrickém šifrovacím algoritmu DES doplněným o sůl.

Soubor s účty musí být čitelný komukoli, protože jeho obsah využívá mnoho programů (např. finger, ls, ps,...) při své činnosti. Takovou činností může být například UID na jméno uživatele. Proto si kdokoli tento soubor může přečíst a pokusit se hesla dešifrovat. Abychom tomuto typu útoku zabránili, zavádíme „stínový soubor hesel“, který se jmenuje

Položka	Popis
`novak`	uživatelské jméno
`i4awJRX2qHMWf`	zakódované heslo uživatele
`505`	UID
`100`	GID
`Josef Novák`	celé jméno uživatele
`/home/novak`	domovský adresář uživatele
`/bin/bash`	uživatelův login shell[1]

Tabulka 3.1: Význam položek `/etc/passwd`

`/etc/shadow`. Zakódovaná hesla jsou z `/etc/passwd` přesunuta do souboru `/etc/shadow`, v souboru `/etc/passwd` jsou zakódovaná hesla nahrazena znakem `x`.

Příklad:

```
/etc/passwd: novak:x:505:100:Josef Novák:/home/novak:/bin/bash
/etc/shadow: novak:i4awJRX2qHMWf:10881:0:99999:7:::
```

Typy útoků na uživatelská hesla:

- opakované pokusy o přihlášení – vetřelec se snaží přihlásit tím, že zkouší různá obvyklá hesla. Tyto pokusy o proniknutí se snadno odhalí pravidelným sledováním systémových logů. Problémem může být, že všechny programy nemusí provádět logování opakovaných neúspěšných pokusů o přihlášení.
- dešifrování hesel – získání souboru s hesly a jejich následné dešifrování. Tomu zabraňuje `/etc/shadow`, který je čitelný pouze superuživateli a vybraným programům, které mají nastaveno příslušné oprávnění.
- odposlech – odezírání hesel z klávesnice, trojské koně, odposlouchávání sítě.

Abychom co nejvíce ztížili prozrazení svého hesla, definujme několik pravidel pro volbu správného hesla:

- nejméně 6 znaků
- kombinovat velká a malá písmena
- používat speciální znaky (`,.-/;!`) a číslice
- nepoužívat znaky národní abecedy
- nepoužívat žádné známé nebo snadno zjistitelné údaje (jméno manželky, rodné číslo, číslo občanského průkazu, pasu,...)

[1] Login shell je shell, který je spuštěn při přihlášení uživatele jako první. Ukončením login shellu se uživatel odhlásí. Pokud položka shell v `/etc/passwd` je prázdná, pak se implicitně jako login shell použije Bourne shell `/bin/sh`.

- není vhodné používat slova ze slovníku (i obráceně zapsaná) – útok slovníkovou metodou, např. `crack(1)`
- ale HLAVNĚ by mělo být zapamatovatelné

Správce by měl kontrolovat uživatelská hesla a uživatele se slabými hesly vyzvat k jejich změně. Dalším problémem porušující bezpečnost je sdílení hesel mezi uživateli, tomu lze zabránit vypracováním lokálních administrativních a personálních pravidel v souladu s konkrétní bezpečnostní politikou.

Speciální účty

Speciálním uživatelem v UNIXu je uživatel, který má UID 0. Dle konvencí má tento účet jméno *root*. Programy běžící pod uživatelem root mají vypnuta téměř všechna bezpečnostní omezení, proto tento účet není určen k běžné práci, ale pouze ke správě a provozu systému.

Právě z důvodu neomezeného přístupu k prostředkům je tento účet vystaven mnoha útokům.

3.2 Přístupová práva a soubory

UNIX má unifikovaný přístup ke všem prostředkům přes souborový systém. K procesům, síťovým spojením, vstupně/výstupním zařízením lze přistupovat přes souborový systém [6].

Soubory jsou organizovány do stromové struktury, která se nazývá adresářová struktura. Každý soubor má přiřazeno své jméno, svého vlastníka, skupinu, přístupová práva a další atributy. Jméno je uloženo v adresáři a ostatní atributy ve struktuře, která se nazývá *i-uzel* (angl. i-node).

Jméno souboru může obsahovat libovolné znaky kromě `/` a `\0` (null). Mezi nimi jsou i řídící znaky, což nám přináší zajímavé bezpečnostní dopady, ke kterým se vrátíme později.

Přístupová práva

Přístupová práva k objektu (souboru, adresáři) se kontrolují při jeho otevírání, pak již kontrolována nejsou. Přístupová práva jsou organizována do tří samostatných trojic.

Příklad:
```
-rw-r--r--   1 novak    users          26 Jun 23  1999 report.txt
drwxr-x---   2 novak    users        1024 Mar 19 12:13 tmp
brw-rw----   1 root     disk       3,   8 May  5  1998 hda8
lrwxrwxrwx   1 root     root           9 Mar  7 11:16 X
^^^^^^^^^^ typ souboru a bity přístupových práv
```

Hodnota	Význam
`-`	normální soubor
`d`	adresář
`c`	znakové zařízení (terminál, tiskárna, streamer)
`b`	blokové zařízení (disky)
`l`	symbolický odkaz
`s`	soket
`=` nebo `p`	roura (angl. pipe)

Tabulka 3.2: Typy souboru

První znak vyjadřuje typ souboru, tabulka 3.2. Za ním následují již zmíněné tři trojice:

1. trojice určuje přístupová práva pro vlastníka souboru
2. skupina práv pro skupinu vlastnící soubor
3. trojice jsou přístupová práva pro ostatní, tj. pro ty, kteří nejsou vlastníkem ani nepatří do skupiny vlastnící soubor

Významy jednotlivých práv v každé trojici uvádí tabulka 3.3 pro soubor a tabulka 3.4 pro adresář. Přístupová práva speciálních souborů mají stejný význam jako u běžných souborů. U symbolických odkazů se přístupová práva nenastavují, často jsou `lrwxrwxrwx`.

Hodnota	Význam
`r`	READ – soubor lze číst. Lze použít funkce `open(2)` a `read(2)`.
`w`	WRITE – do souboru lze zapisovat, přepisovat, mazat. Možno použít funkce `open(2)`, `write(2)` a `truncate(2)`.
`x`	EXECUTE – má význam pouze pro programy. Znamená, že soubor lze spouštět buď zadáním jeho jména na příkazovém řádku nebo pomocí funkce `exec(3)`. Pokud má soubor pouze právo spouštění, nelze ho číst. Toto je nutné si uvědomit, protože pokud je soubor skript (např. shellu), nepůjde spustit.

Tabulka 3.3: Přístupová práva souboru

Jméno souboru

Jak již bylo řečeno, jméno souboru může obsahovat libovolné znaky kromě lomítka a znaku konce řetězce. Délka jména je omezená, v UNIXu je to 255 znaků.

Hodnota	Význam
`r`	READ – lze vypsat soubory a podadresáře, které adresář obsahuje.
`w`	WRITE – v adresáři je možné soubory a podadresáře vytvářet, mazat a přejmenovávat.
`x`	EXECUTE – do adresáře lze přistoupit (nastavit ho jako aktuální). Toto právo je nutné pro otevírání souborů, které jsou v adresáři.

Tabulka 3.4: Přístupová práva adresáře

Shell interpretuje některé znaky jako speciální [5] – středník `;` je oddělovač příkazů, příkaz uzavřený ve zpětných apostrofech ‘ je nahrazen svým výstupem, atd. Toho lze s úspěchem využít při útoku, stačí by takové jméno bylo použito jako argument příkazu. Touto chybou jsou nejvíce ohroženy příkazy pro prohledávání souborových systémů, takovými příkazy jsou například `find` a `xargs`.[2]

3.3 Speciální soubory

Speciálními soubory rozumíme soubory *znakových zařízení*, *blokových zařízení*, *rour*, *soketů* a *symbolických linků*. Dále se budeme zabývat problematikou znakových a blokových zařízení.

Znaková a bloková zařízení systém rozlišuje podle dvou čísel: *hlavního* a *vedlejšího*. Hlavní číslo určuje typ zařízení, vedlejší číslo jednoznačně označuje zařízení se stejným hlavním číslem. Např. všechny oddíly pevného disku budou mít stejné hlavní číslo, ale různá vedlejší. Jádro převádí všechny operace se souborem zařízení na vstupně/výstupní operace daného zařízení.

Znaková zařízení, jinak také nazývaná *přímá* (angl. raw), poskytují přímý přístup k zařízení. Přístup k těmto zařízením musí odpovídat jejich fyzickému typu, tj. terminál pracuje s jedním znakem, ale disk pracuje s blokem (např. 512 bytů). Aplikace znakového zařízení při přístupu k souborovému systému je příkladem nevhodného použití, protože častým případem u souborových systémů je modifikace pouze několika bytů souboru. To by při každém přístupu k souboru vedlo k přečtení a zápisu celého bloku, což není efektivní. Proto UNIX poskytuje bloková zařízení, která umožňují přístup ke znakovým zařízením přes vyrovnávací paměti. Zápis vyrovnávacích pamětí na fyzické zařízení se provádí periodicky každých 30s nebo 60s. Obvykle tuto činnost vykonává některý démon, např. *sync*, *fsflush* nebo *kflushd*.

Soubory zařízení jsou obvykle umístěny v adresáři `/dev`. To ovšem neznamená, že musí být umístěny pouze v tomto adresáři. Chybným nastavením přístupových práv u zařízení můžeme vytvořit bezpečnostní díru. Např. pokud umožníme komukoli zapi-

[2] dnešní verze těchto příkazů již tuto chybu neobsahují

sovat do `/dev/kmem`[3], pak může snadno celý systém zhroutit příkazem `cat /dev/random >/dev/kmem`. Proto je nutné správně nastavit a pravidelně kontrolovat přístupová práva všech zařízení. Také je důležité dát pozor na nastavení parametrů při připojování souborových systémů, viz. 3.5 (část Připojování souborových systémů).

3.4 Sticky bit

Každému souboru a adresáři lze nastavit také tzv. *sticky-bit*. Dříve byl u souborů využíván pro ponechání kódu programu v paměti. To bylo kvůli rychlejšímu spouštění často používaných programů. Tento bit měl význam na strojích s malou operační pamětí (řádově KB), dnes se již nepoužívá. U souborů mohl *sticky-bit* nastavovat pouze superuživatel.

U adresářů se stále používá a jeho význam je následující: máme-li adresář s nastavenými právy `rwxrwxr-t`[4], pak soubory v tomto adresáři může mazat a přejmenovávat pouze vlastník adresáře, vlastník souboru a superuživatel. Tuto vlastnost lze s výhodou využít u adresáře `/tmp` – uživatelé si pak navzájem nemohou mazat soubory.

3.5 SUID a SGID bit

Každý proces má přiřazeny dvě UID – efektivní UID (EUID) a reálné UID. Za normální situace má program po spuštění obě UID stejná a jsou rovna UID uživatele, který program spustil. Všechny kontroly bezpečnosti[5] se provádí vůči efektivnímu UID.

Někdy i běžný uživatel potřebuje provádět administrativní operace[6], na které nemá oprávnění, proto se zavádí tzv. *Set-UID bit* a *Set-GID bit*.

SUID bit slouží ke změně efektivního UID procesu. Program s nastaveným SUID bitem má po spuštění nastaveno efektivní UID na vlastníka souboru, reálné se nemění (je nastaveno na UID uživatele, který program spustil). Tím je programu propůjčena identita jiného uživatele a jeho přístupová práva. Příkladem takového programu je `/usr/bin/passwd`[7]. SGID bit má stejný význam jako SUID, ale týká se skupin.

U adresářů nemá SUID bit žádný význam. Nastavením SGID bitu u adresáře dosáhneme toho, že soubory a podadresáře vytvořené v adresáři budou mít nastavenu skupinu stejnou jako má adresář, podadresáře navíc budou mít nastaven SGID bit.

SUID bit, popř. SGID bit, je ve výpisu příkazu `ls` označen znakem `s` na místě `x` v právech vlastníka, popř. skupiny.

Bezpečnostní rizika

SUID bity přináší možné bezpečnostní problémy. Například libovolný uživatel se může stát superuživatelem, když spustí SUID kopii shellu, která patří uživateli root. Ovšem zkopírovaným programem nemusí být nutně jenom shell. Tomuto nebezpečí lze zabránit

[3] paměť jádra

[4] znakem `t`, na pozici `x` v přístupových právech pro ostatní, se označuje *sticky-bit*

[5] přístupová práva, ...

[6] např. změna hesla, shellu atd.

[7] tento program slouží ke změně hesla uživatele

pravidelným prohledáváním souborového systému na SUID programy. Prohledávání musí provádět root, protože útočník může kopii programu ukrýt do adresáře, kam má přístup pouze on.

Chyby v programech

Nejčastěji jsou bezpečnostní incidenty způsobeny chybami v SUID programech nebo špatným nastavením přístupových práv.

Příkladem může být program `write`, který umožňuje psát jinému uživateli na terminál. UNIX zápis na terminál ostatním uživatelům zakazuje pomocí přístupových práv. Právo zápisu má skupina `tty`. Proto je `write` SGID. Z historických důvodů umožňuje `write` spouštět příkazy v shellu. Předtím než se shell spustí, provede se příkaz `setgid(getgid())`. Tedy spuštěný shell má stejné oprávnění jako uživatel, který `write` spustil. Problém nastává ve chvíli, kdy má `write` nastavený (nesprávně) SUID bit. Potom je shell spuštěn sice se skupinou uživatele, ale s EUID rovným `0` (root).

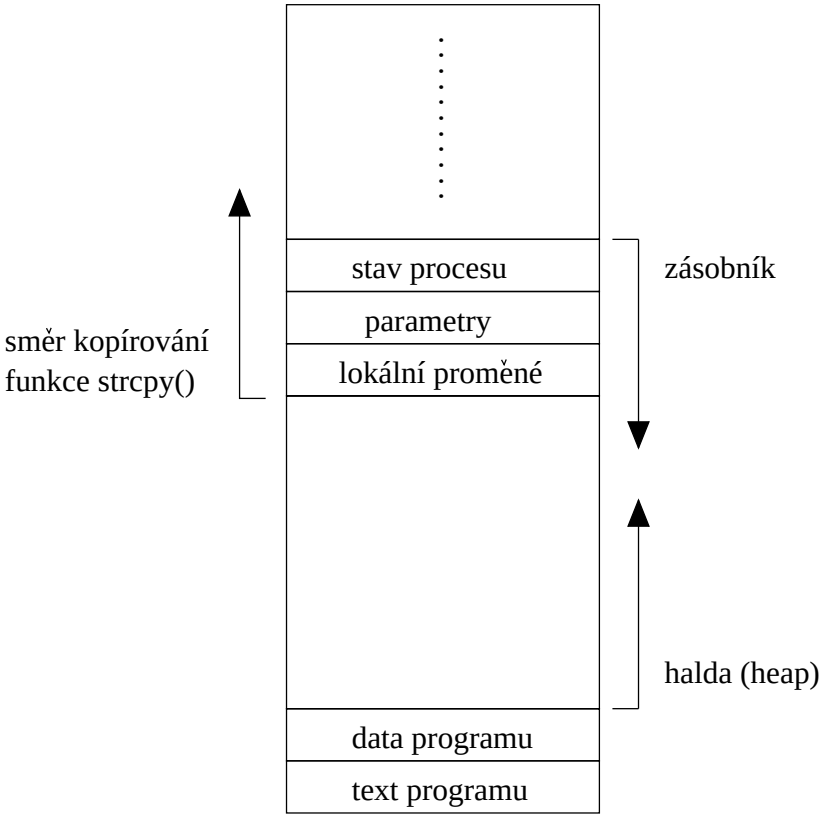
Nejznámější a asi nejčastější chybou v SUID programech je přetečení bufferu [16]. Každý program používá zásobník. Zásobník je struktura, která se používá při volání funkce pro ukládání stavu procesu, ukládání předávaných parametrů a pro ukládání lokálních proměnných (situaci demonstruje obrázek 3.1).

Při volání funkce je na zásobník uložen stav procesu, dále předávané parametry a jsou na něm vytvořeny lokální proměnné. Proces potom pokračuje ve výpočtu těla funkce. Při ukončení funkce jsou ze zásobníku odstraněny lokální proměnné a parametry funkce a je obnoven stav procesu. Proces poté pokračuje ve svém běhu za místem voláním funkce. Zásobník má tu vlastnost, že při ukládání hodnot roste od vyšších adres paměti směrem k nižším. Ale funkce pro kopírování bloku paměti[8] pracují obráceně, tj. paměť kopírují od nižších adres k vyšším.

Mějme příklad programu, který akceptuje jméno souboru jako svůj parametr. S tímto parametrem zavolá funkci `zalohuj()`, která na konec jména souboru přidá řetězec `".backup"`. Funkce `zalohuj()` toto implementuje pomocí lokální proměnné (umístěné na zásobníku), do které si předané jméno souboru zkopíruje funkcí `strcpy(3)` a za něj funkcí `strcat(3)` připojí řetězec `".backup"`. Protože funkce `zalohuj()` používá proměnnou na zásobníku, je této proměnné vyhrazeno na zásobníku místo pevné délky. Pokud programu dáme jako parametr jméno souboru, které je delší než místo vyhrazené pro tuto lokální proměnnou, tak voláním funkce `strcpy(3)`[9] přepíšeme paměť za lokální proměnnou (tedy na vyšších adresách). Ale touto operací také můžeme přepsat důležité hodnoty uložené na zásobníku. Většinou se stává, že program po této operaci skončí chybou a vytvořením souboru jádra. Když útočník zná dokonale chování tohoto programu, může mu dát jako parametr kód, kterým se přepíše zásobník. Po ukončení funkce dojde k obnovení stavu procesu, ale tento stav nebude stavem, který byl při volání funkce uložen. Tento stav je modifikován tak, že po ukončení funkce běh programu pokračuje v útočníkem předložené části kódu. Pak záleží pouze na vlastnostech tohoto kódu, co se stane.

[8] Takovou funkcí je například `strcpy(3)`.

[9] Funkce `strcpy(3)` kopíruje znak po znaku zdrojový řetězec do paměti určené cílovým řetězcem. Toto kopírování se provádí až do konce zdrojového řetězce – znaku `null`.



Obrázek 3.1: Adresový prostor procesu

Proto je tato chyba velice nebezpečná pro SUID programy. V těchto případech útočník programu předkládá například kód, který spouští shell (tento shell je spuštěn s oprávněními vlastníka programu).

Připojování souborových systémů

Příkaz `mount` se používá k připojování souborových systémů do stávajícího. Připojujeme například vyměnitelné jednotky, jednotky sdílené přes síť[10],... Pokud je uživatelům systému povoleno připojovat vyměnitelné jednotky (pružné disky, CD-ROM,...), je vhodné z důvodu bezpečnosti, aby z těchto jednotek nešly spouštět SUID programy nebo dokonce aby programy nešly spouštět vůbec nebo aby se ignorovaly speciální zařízení. Proto má `mount` volitelné parametry `nosuid`, `noexec` a `nodev`. Některé verze příkazu přidávají u vyměnitelných zařízení tyto parametry implicitně, pokud není spouštěn pod rootem.

Kdyby `mount` tyto volby nepodporoval nebo je správce systému nepoužíval, pak útočníkovi stačí na vlastním počítači vytvořit disketu s SUID root shellem, připojit ji v atakovaném systému a shell spustit.

3.6 Proměnná PATH

Proměnná PATH obsahuje dvojtečkou oddělený seznam adresářů, které jsou prohledávány, když je na příkazovém řádku zadán program bez cesty. Pokud je program na některé z cest nalezen, spustí se, jinak shell vrátí chybu. Obsahuje-li PATH i aktuální adresář (znak tečky . nebo dva znaky dvojtečky : za sebou), jsou programy vyhledávány i v něm. Uvádět aktuální adresář do seznamu prohledávaných cest není moc bezpečné – ukážeme si to na dvou příkladech.

Útočník provede následující příkazy:

```
$ cd /tmp
$ cat - <<EOF >ls
> #!/bin/bash
> cp /bin/bash /tmp/xxx
> chmod 4555 /tmp/xxx
> rm /tmp/ls
> /bin/ls $*
> EOF
$ chmod 0755 /tmp/ls
```

Tím vytvoří v adresáři `/tmp` shellový skript se jménem `ls`, který je spustitelný kýmkoli a simuluje činnost správného `ls`. Pokud nyní uživatel s tečkou na začátku proměnné PATH provede `cd /tmp` a vypíše si pomocí `ls` aktuální adresář, chytí se do nastražené pasti. Výpis se mu sice zobrazí, ale mezi tím se vytvořil SUID shell. Nejhorší situace nastane pokud napadeným uživatelem je neopatrný správce.

[10] NFS

Problémy může způsobit i tečka na konci proměnné PATH, i když ne takového rozsahu. K porušení bezpečnosti pak může dojít velice podobně – když je v názvu spouštěného programu překlep a útočník s tím počítá a vytvoří program stejného jména, pak může získat SUID shell jako v předchozím případě.

Obecně není příliš bezpečné přidávat do prohledávací cesty adresáře, do kterých může kdokoli zapisovat.

3.7 Programy a superuživatelská práva

Dnešní systémy poskytují mnoho služeb, např. internetové služby. Systém všechny tyto služby spouští při svém startu procesem *init*.[11] Proces init běží vždy s oprávněním superuživatele, proto i služby jsou spouštěny s těmito právy. Některé služby ovšem taková oprávnění pro vykonávání své činnosti nepotřebují, proto je důležité tyto služby nakonfigurovat tak, aby se ihned po svém startu těchto oprávnění vzdaly. Tím zamezíme možným průnikům do systému pomocí získání superuživatelských oprávnění.

Některé služby[12] při plnění svých úkolů nepotřebují pracovat s celým souborovým systémem, ale vystačí si s nějakým jeho podstromem. Proto UNIX poskytuje funkci `chroot(2)`, která provádí změnu kořenového adresáře do libovolného podadresáře, který je parametrem této funkce. Po zavolání `chroot(2)` nebude proces schopen otevřít jiné soubory než soubory, které jsou uloženy v nastaveném podstromu. Ovšem je nutné dát pozor na chování této funkce, která sice změní kořenový adresář, ale nezmění aktuální adresář (který nemusí být podadresářem nového kořenového adresáře). Pokud spouštíme program, který již má změněný kořenový adresář, pod uživatelem root, může se tento program ze svého omezení dostat novým voláním `chroot(2)` a následujícím provedením `cd ../../../../...` Tento mechanismus je vhodné používat u služeb, jejichž činnost lze omezit na určitý podstrom. Tímto zabráníme případnému útočníkovi v procházení celého souborového systému a vyhledávání zranitelných míst.

[11] Služby mohou být spouštěny i jiným způsobem, viz 5.5.

[12] Např. Anonymous FTP

Kapitola 4

Rozšíření modelu

Základní bezpečnostní model UNIXu neposkytuje příliš mnoho funkcí a mechanismů pro zajištění vyšší bezpečnosti celého systému. Proto v této části se zaměříme na některá možná rozšíření tohoto standardního modelu.

Pokud některý program potřebuje ke své činnosti číst soubor se zakódovanými hesly uživatelů, jsou mu přidělena práva superuživatele – to je zbytečné, protože tento program potřebuje pouze číst určitý soubor, ale už nepotřebuje do něj zapisovat atd. Tento problém řeší rozšíření nazvané *oprávnění* (angl. capabilities), které je popsané v části 4.1. Toto rozšíření rozděluje celá práva superuživatele na menší části. Pomocí takového dělení lze lépe a bezpečněji pokrýt funkce programů, které vyžadují některá oprávnění superuživatele.

Další slabinou UNIXového modelu jsou přístupová práva k souborům, UNIX standardně umožňuje nastavit přístupová práva pro vlastníka souboru, dále pro skupinu vlastnící soubor a pro ostatní. Přičemž práva jsou rozdělena na právo čtení, zápisu a spouštění. Ovšem v některých situacích nejsou takové možnosti dostatečné. Toto omezení ruší rozšíření *seznam řízení přístupu* (angl. Access Control List). Seznam řízení přístupu umožňuje nastavit přístupová práva k souboru na úrovni jednotlivých uživatelů, tj. např. dva uživatelé stejné skupiny mohou mít povolen přístup, ale nikdo jiný. Část 4.2 popisuje toto rozšíření.

Tato dvě vylepšení modelu se netýkají omezení práv samotného superuživatele. Možnosti, které poskytuje *ext2fs* (viz 4.3), omezují práva superuživatele i když ne úplně spolehlivě.

Poslední rozšíření, které bude v této kapitole popsáno, se týká šifrování souborů. Máme-li nějaký soubor zašifrovaný dostatečně robustní metodou, tak bez znalosti klíče nemáme způsob, jak zjistit jeho obsah. Proto kryptografie poskytuje ideální metodu pro utajení důležitých informací [9]. V části 4.4 této kapitoly je uvedena jedna možnost, jak šifrování souborů implementovat. Zavedení uvedené metody nevyžaduje žádné změny v operačním systému.

4.1 Oprávnění (capabilities)

Standardní „superuživatelský“ model je z hlediska bezpečnosti nedostatečný, proto jsou zaváděna další rozšíření. Jedním z možných rozšířeních jsou *úrovně privilegií*. Toto rozšíření definuje různé úrovně oprávnění, ve kterých může proces pracovat (každá vyšší úroveň poskytuje postupně vyšší oprávnění). Tímto rozšířením se nebudeme dále zabývat. Další možné rozšíření poskytuje ještě podrobnější dělení superuživatelských oprávnění a nazývá se tzv. *oprávnění* (angl. capabilities).

Oprávnění jsou definovány normou POSIX.6 [10]. Hlavní důvody pro zavedení nového standardu jsou:

- poskytnout přenositelné řešení pro možnost povolit procesu provádět jinak zakázané systémové služby
- implementovat mechanismus *nejnižšího oprávnění* (angl. least privilege) – povolit procesu pouze ty operace, které jsou nezbytně nutné pro jeho správnou funkci

Pro zajištění těchto podmínek je nutné zavést oprávnění pro procesy a pro soubory. Oprávnění u procesů odpovídají možnosti vykonávat privilegované operace. U souborů zajišťují možnost implementace SUID programů.

Oprávnění a procesy

Proces má přiřazeny tři bitové mapy (angl. sets of capabilities), jednotlivé bity mapy odpovídají jednotlivým oprávněním. Bitové mapy mají následující významy:

- *efektivní mapa* (angl. effective set) – obsahuje oprávnění, která proces může aktuálně používat. Pokud chce proces provést privilegovanou operaci, operační systém kontroluje efektivní mapu na přítomnost bitu, který odpovídá této operaci. Ve standardním modelu systém kontroluje EUID procesu na nulu.
- *přípustná mapa* (angl. permitted set) – oprávnění, která si proces může udělit, tj. nastavené bity efektivní mapy jsou podmnožinou bitů přípustných. Proces může nastavovat pouze ty bity efektivní mapy, které jsou obsaženy v mapě přípustné.
- *dědičná mapa* (angl. inheritable set) – určuje bity přípustné mapy, které budou děděny programem spuštěným pomocí `exec()`, tj. při `exec()` jsou ponechány nastaveny pouze ty bity přípustné mapy, které jsou obsaženy v dědičné mapě. Při volání funkcí `fork()` nebo `clone()` nejsou prováděny žádné změny v přípustné mapě.

Oprávnění a soubory

Soubor má přiřazeny také tři bitové mapy, podobně jako proces. Oprávnění mají smysl jenom u spustitelných souborů (programů). Významy bitových map jsou částečně odlišné od svých významů u procesů, proto jsou také odlišně pojmenované:

- *povolená mapa* (angl. allowed set) – určuje oprávnění, která může spouštěný program přijmout od procesu, který ho spouští. Tzn. nová přípustná mapa procesu, který provedl `exec()`, je tvořena bity, které jsou obsaženy jak v dědičné mapě procesu tak i v povolené mapě programu.

- *nastavená mapa* (angl. forced set) – jsou oprávnění přidaná k přípustné mapě procesu (implementace SUID bitu).

- *efektivní mapa* (angl. effective set) – určuje jaké bity efektivní mapy procesu budou nastaveny podle bitů mapy přípustné. Jednoduše řečeno, určuje oprávnění, která budou procesu nastavena ihned po `exec()` bez toho, aby je proces musel explicitně v efektivní mapě nastavovat. Někdy je tato mapa implementována jedním bitem (buď se přípustná mapa zkopíruje do efektivní nebo ne).

Oprávnění umožňují nastavit procesu pouze ta práva, která potřebuje ke své činnosti. Efektivní mapa dovoluje procesu si dynamicky přidělovat nebo odebírat práva, ovšem proces si může nastavovat jenom ta práva, která jsou obsažena v přípustné mapě. Pokud proces volá jiný program pomocí `exec()`, pak dědičná mapa mu umožňuje změnit maximální možná práva, která spuštěný program bude moci využívat.

Programu lze pomocí nastavené mapy určit, která oprávnění bude moci využívat, když bude spuštěn. Pomocí povolené mapy lze programu některá práva odebrat, který by mu jinak byla volajícím procesem ponechána. Efektivní mapa u souboru pouze určuje, které bity přípustné mapy budou programu nastaveny v efektivní mapě ihned po svém spuštění.

Následující pravidla vyjadřují změny v jednotlivých mapách procesu během volání funkce `exec()`:

$$\begin{aligned} pI' &= pI \\ pP' &= fP \mid (fI \;\&\; pI) \\ pE' &= pP' \;\&\; fE \end{aligned}$$

kde fP je nastavená mapa programu, fI je povolená mapa programu,
fE je efektivní mapa programu, pP je přípustná mapa procesu,
pI je dědičná mapa procesu, pE je efektivní mapa procesu,
znak ' označuje stav po ukončení volání `exec()`.

Oprávnění u uživatelů lze implementovat nastavením daných uživatelských oprávnění login shellu, který bude tato práva dědit na další procesy. Takto lze například vytvořit uživatelský účet, který je určený pro zálohování.

Použití této techniky v systému vyžaduje změny v implementaci funkcí, které jsou před použitím chráněny pomocí bitů oprávnění.

4.2 Seznam řízení přístupu (ACL)

Potřeba nastavovat přístupová práva k objektu přesněji než umožňují *bity přístupových práv* (angl. permission bits) (viz 3.2) vedla k zavedení nového mechanismu. Tento mechanismus se nazývá *seznam řízení přístupu* (angl. Access Control List) (ACL) [17] a je specifikován normou POSIX.6 [10]. Tento standard byl vytvořen s ohledem na zpětnou kompatibilitu, proto ACL je možné používat současně s bity přístupových práv.

Seznam řízení přístupu [13] je objekt, který je asociován se souborem, a obsahuje seznam položek, které určují přístupová práva libovolného uživatele nebo skupiny. Tím je umožněno nastavovat přístupová práva na úrovni jednotlivých uživatelů.

Příklad:

```
OWNER: rwx
KAREL: r--
USERS: rw-
OTHER: ---
```

Tento demonstrační příklad ukazuje nastavení přístupových práv pro čtení a zápis skupině USERS a zároveň ukazuje omezení práv pouze na čtení uživateli KAREL (ačkoli náleží do skupiny USERS), ostatním přístup nepovoluje. V uvedeném příkladu je konflikt přístupových práv (viz uživatel KAREL), proto je definováno, že při kontrole přístupových práv se nejdříve hledá shoda podle uživatele, potom podle skupiny a nakonec jsou při neúspěchu použita přístupová práva pro ostatní.

Položka seznamu ACL se skládá ze tří částí:

- typ – určuje typ objektu, kterému jsou přiřazena přístupová práva obsažená v položce; možné hodnoty jsou: vlastník souboru, skupina vlastnící soubor, uživatel, skupina nebo ostatní
- hodnota typu – identifikátor objektu určeného typem, např. UID nebo GID; toto pole nemá význam pro typy: vlastník souboru, skupina vlastnící soubor a ostatní
- přístupová práva – přístupová práva objektu určeného pomocí polí typ a hodnota typu

ACL musí podporovat alespoň následující přístupová práva: právo pro čtení, zápis a spuštění. Některé implementace mohou poskytovat i další práva.[1]

Každý seznam ACL musí obsahovat položky pro vlastníka souboru, skupinu vlastnící soubor a pro ostatní. Zobrazení seznamu ACL se provádí příkazem `getacl` a nastavení pomocí `setacl`. Pokud jsou v systému podporovány bity přístupových práv i seznam řízení přístupu, pak změna v bitech provádí změnu v příslušných položkách ACL a naopak.

I přes možnost současné implementace bitů přístupových práv a seznamu řízení přístupu v systému je nutné provést změny některého software, např. pro zálohování.

[1] Např. `write-append` umožňuje zápis pouze za konec souboru, `write-change` povoluje změnu souboru, ale soubor nesmí měnit svoji velikost, právo `delete` dovoluje soubor smazat.

4.3 Souborový systém ext2fs

Souborový systém ext2fs implementuje, kromě jiných věcí, atributy souboru [7]. Možné atributy jsou:

- bezpečné mazání (angl. secure deletion) – soubor není pouze smazán, ale jeho data jsou přepsána náhodnými hodnotami
- neměnnost (angl. immutable file) – soubor s tímto atributem nastaveným nemůže být nikým modifikován (nikdo nemůže do souboru psát nebo ho smazat, soubor je povoleno pouze číst)
- přidávání (angl. append-only) – do souboru je možné psát, ale data jsou vždy přidávána na konec souboru, podobně jako u atributu neměnnost nelze soubor smazat nebo číst

Atribut bezpečné mazání nám zaručuje, že data souboru, který byl smazán, nebude nikdo schopen obnovit. Atribut neměnnost zabraňuje změnám v souboru, případný útočník nebude schopen do důležitých souborů nic zapsat a tak si nevytvoří zadní vrátka. Atribut přidávání zaručuje, že data souboru nebudou žádným způsobem změněna, mohou být pouze rozšířena. Toto zabraňuje útočníkovi v modifikaci systémových logů.

Restrikce aplikované na soubory se týkají všech uživatelů (i superuživatele). Pouze v jednouživatelském režimu[2] jsou povoleny změny těchto souborů. Ovšem pokud bude mít superuživatel přímý přístup pro zápis do paměti, lze všechny tyto ochrany obejít.

4.4 Kryptografický souborový systém (CFS)

Zajištění důvěrnosti šifrováním je v současnosti téměř nezbytné. Existuje mnoho metod, jak provádět šifrování. Mezi hlavní přístupy k šifrování souborů patří následující dva:

- *Uživatelská úroveň* – toto je prakticky nejjednodušší metoda. Uživatel provádí utajení vlastními silami s použitím různých nástrojů, např. UNIXového programu `crypt`. Tyto nástroje lze také často použít jako kryptografické filtry[3].

  Některé programy mají již šifrovací mechanismy v sobě integrovány. Pokud pracujeme nad stejnými daty ve více různých programech, musíme zajistit, aby používaly stejné algoritmy a stejné klíče pro šifrování. Toto zajistit nebývá snadné. Použití těchto nástrojů nám přináší také jiné problémy: chybou software nebo opomenutím uživatele může zůstat důležitý dokument buď nezašifrovaný, nebo otevřený text není po zašifrování smazán.

  Ačkoli tyto systémy mohou poskytovat automatické šifrování, je stále nutné po uživateli vyžadovat zadání šifrovacího klíče při spuštění aplikace nebo při otevření souboru.

[2] Operační systém UNIX má několik úrovní práce systému [18].
[3] Text na vstupu je šifrován a posílán na výstup.

- *Systémová úroveň* – touto úrovní rozumíme šifrování dat na straně souborového serveru, tj. na opačné straně než v případě použití šifrování na uživatelské úrovni. Po nastavení kryptografického klíče je šifrování úplně transparentní. Tím odbouráme všechny problémy vznikající na uživatelské úrovni.

  Největším problémem tohoto přístupu je absence zabezpečení komunikačních kanálů mezi klienty a serverem. Abychom toto zajistili, je nutné použít šifrování komunikačních kanálů.

Kryptografický souborový systém (CFS) [4] využívá dobrých vlastností obou přístupů. Poskytuje transparentní přístup k datům a šifrování provádí na straně klienta.

CFS je implementován jako speciální NFS server (`cfsd`), který běží na lokálním stroji a pouze z něj akceptuje spojení. CFS všechny požadavky na něj kladené přesměruje na souborový systém lokálního stroje (viz obrázek 4.1). V tomto přesměrování je také skryto automatické šifrování a dešifrování dat. Tím vytváří virtuální souborový systém.

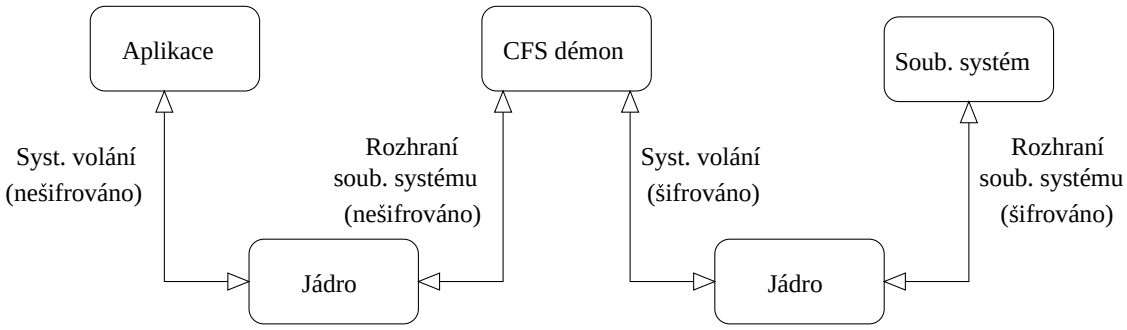
Zpřístupnění šifrovaných dat umožňují následující programy:

- `cattach` – provádí připojení šifrovaného adresáře do CFS. Jeho argumenty jsou jméno připojovaného adresáře, jméno adresáře, pod kterým budou dekódovaná data přístupná, a šifrovací klíč.
- `cdetach` – odpojuje dříve připojený adresář.
- `cmkdir` – slouží k vytvoření šifrovaného adresáře a k přiřazení šifrovacího klíče.

Po připojení adresáře se přistupuje ke chráněným souborům stejně jako k normálním (nezašifrovaným). Kontrola přístupu je prováděna na úrovni UID – přistupující uživatel musí mít stejné UID jako uživatel, který adresář připojil.

Protože data poskytovaná přes CFS jsou automaticky šifrována, popř. dešifrována dříve než jsou někam zapisována (nebo odesílána), popř. čtena, jsme zbaveni nutnosti používat šifrovaná spojení. Kódování je implementováno pomocí algoritmu DES. Kromě toho, že je šifrován obsah souborů, je šifrováno i jejich jméno. Jediné co zůstává v otevřené podobě je velikost souborů, jejich přístupová práva, časy přístupu a vytvoření a adresářová struktura.

Protože zakódované soubory jsou viditelné i bez použití CFS (pouze jejich obsah a jméno není možné zjistit), není nutné upravovat software pro zálohování. V tom je hlavní síla tohoto řešení.



Obrázek 4.1: Architektura CFS

Kapitola 5

Metody detekce a sledování

Čím více správce systému ví o činnostech systému, tím více bezpečný je schopen systém udělat [1].

V této kapitole se zmíníme o základních mechanismech sledování práce systému a uživatelů [2, 15].

UNIX má několik sledovacích mechanismů, které jsou implementovány pomocí logů. Soubory logů představují záznamy událostí, které se v minulosti v systému udály. Proto je lze využít pro sledování správné funkce bezpečnostních mechanismů systému nebo pro detekci průniku do systému. Dnešní verze UNIXu monitorují mnoho činností – přihlášení a odhlášení uživatelů, prováděné přenosy souborů po síti, použití elektronické pošty, pokusy uživatelů stát se superuživatelem apod.

Pro vysokou informační hodnotu jsou logy terčem častých útoků. Může jít o útočníkovy pokusy o zahlazení stop nebo o pokusy běžných uživatelů analyzovat logy, které jim jsou přístupné pro čtení. Z uvedeného vyplývá, že je důležité starat se o utajení důležitých logů a také o jejich nezměnitelnost.

Vhodným nastavením přístupových práv omezíme nepovolený přístup k logům, nezměnitelnost můžeme zajistit odesíláním logovaných zpráv po síti na jiný počítač, který obvykle bývá hůře napadnutelný.

Logy bývají umístěny v adresáři `/var/adm` a jeho podadresářích nebo ve `/var/log`.

5.1 Soubor `lastlog`

UNIX zaznamenává čas posledního přihlášení do souboru `/var/adm/lastlog`. Tento čas je také vypisován při příštím přihlášení. Např. služba *finger* využívá obsah souboru `lastlog`. Problémem je, že při novém přihlášení jsou informace o předchozím přihlášení přepisovány. Proto, když si uživatel podezřelého času nevšimne a obrazovku si smaže, už ho nikdy nezjistí. Tomu lze částečně zabránit pravidelným zálohováním tohoto souboru.[1] Některé systémy zaznamenávají také čas posledního neúspěšného přihlášení.

Soubor `lastlog` slouží k rychlému zjištění času posledního přihlášení. Kompletní databázi o přihlášeních uživatelů poskytuje soubor `wtmp`.

[1] např. každých 6 hodin apod.

5.2 Soubory `utmp` a `wtmp`

UNIX udržuje databázi aktuálně přihlášených uživatelů v souboru `/var/run/utmp`[2]. Tento soubor čte program `who` a vypisuje jeho obsah ve srozumitelné formě. Jeho rozměry jsou malé, obvykle řádově kilobajty.

Příklad:

```
$ who
vlasta   :0       Mar 28 20:05
petr     pts/0    Mar 29 13:36 (dione.ascs.muni.cz)
```

Druhý soubor `/var/adm/wtmp` obsahuje úplné záznamy o všech přihlášeních uživatelů. Tento soubor se stále zvětšuje, proto jsou staré záznamy mazány. Program zobrazující obsah `wtmp` je `last`.

Příklad:

```
$ last
vlasta   pts/0                        Wed Mar 29 11:36 - 11:37  (00:00)
vlasta   tty1                         Wed Mar 29 11:34 - 11:39  (00:05)
root     tty1                         Wed Mar 29 11:33 - 11:34  (00:01)
vlasta   tty1                         Wed Mar 29 11:32 - 11:33  (00:00)
vlasta   pts/1        ganymedes.ascs.m Wed Mar 29 11:32 - 11:32  (00:00)
...
```

Některé programy do těchto logů nezapisují (např. rsh), proto je někdy vhodnější zjišťovat uživatele, kteří systém využívají, pomocí programu `ps`. Dalším programem, který nemodifikuje tyto soubory, je `su`.

Důležité je správné nastavení přístupových práv k těmto souborům. Pokud běžný uživatel bude mít možnost zapisovat do těchto souborů, může nevhodným způsobem modifikovat chování některých programů, které zapisují na terminál (např. `wall` nebo `biff`).

5.3 Soubor `acct`

Kromě logování přihlášení a odhlášení umožňuje UNIX zaznamenávat i všechny provedené příkazy. Toho lze využít ke sledování činnosti uživatelů, ale i ke zjištění, co předcházelo průniku do systému – pokud ovšem nebyly soubory vymazány. Obsah souboru v čitelné podobě lze zobrazit pomocí `lastcomm` nebo `acctcomm`. Příkazy se bohužel logují bez parametrů a bez adresářů, ve kterých byly spuštěny.

Zaznamenávání příkazů je velice náročná činnost na místo – i na málo vytížených systémech narůstá soubor poměrně rychle. Proto není tento audit zapínán automaticky, ale musí ho administrátor zapnout ručně, provede to příkazem `accton <filename>`, kde `<filename>` je jméno souboru, do kterého se zápis provádí.

[2] případně `/etc/utmp`

Hodnota	Původce
`authpriv`	autorizační zprávy – programy vyžadující uživatelské jméno a heslo
`cron`	programy `cron` a `at`
`daemon`	jiní systémoví démoni
`kern`	jádro systému
`local0-local7`	rezervováno pro lokální použití
`lpr`	tiskový systém
`mail`	poštovní systém
`news`	USENET news subsystém
`syslog`	zprávy generované procesem `syslogd`
`user`	uživatelský proces
`uucp`	UUCP subsystém

Tabulka 5.1: Hodnoty původu zprávy

5.4 Syslog

Syslog – systémový log UNIXu je univerzální nástroj pro logování. Původně vznikl jako součást programu `sendmail`. Celý systém je založen na logovacím procesu, kterému se posílají všechny informace k zaznamenání.

Syslogu může zasílat zprávy libovolný proces. Zpráva se skládá ze čtyř částí:

- jméno programu
- původ zprávy
- priorita zprávy
- text zprávy

Hodnoty položky původ zprávy uvádí tabulka 5.1. Položka priorita může nabývat hodnot dle tabulky 5.2.

Konfigurace syslogu je uložena v souboru `/etc/syslog.conf`. Každý řádek se skládá ze dvou částí: *filtru zpráv* a *akce*. Tyto části musí být odděleny alespoň jedním tabulátorem. Akce určuje, co se bude dělat se zprávami vyhovující filtru zpráv. Filtr zpráv má následující strukturu:

```
<původce>.<priorita>[;<původce>.<priorita>[;...]]
```

Například filtru `user.debug` vyhovují zprávy od uživatelského procesu s prioritou `debug`. V položce `<původce>` a `<priorita>` lze použít symbol `*`, který vyhovuje libovolné hodnotě. Akce může být následující:

- zapsání do souboru nebo zařízení, př. `/var/log/messages`, `/dev/console`

Priorita	Popis
`emerg`	havarijní stav – zhroucení systému,...
`alert`	nutnost okamžité opravy vzniklé situace (porušení systémové databáze)
`crit`	kritická situace, hardwarová chyba
`err`	normální chyba
`warning`	varování
`notice`	oznámení, ale nemělo by být ignorováno
`info`	informativní zpráva
`debug`	zprávy generované při ladění programů
`none`	vybrané zprávy nebudou logovány, např. `*.info;user.none` všechny informativní zprávy budou ukládány, kromě zpráv od normálních uživatelských procesů

Tabulka 5.2: Hodnoty priority zprávy

- poslání zprávy uživateli, př. `root` nebo `root,adm`, tj. může zde být i seznam uživatelů; zpráva je zobrazena na všech terminálech, na kterých jsou daní uživatelé přihlášení
- poslání zprávy všem uživatelům, určuje symbol `*`
- předání zprávy programu, jméno programu je uvedené za symbolem `|`
- poslání zprávy syslogu na jiném počítači, počítač uvedený za `@`

Všechna data, která syslog zpracovává, jsou čtena z následujících tří zařízení:

- zařízení `/dev/log`, do kterého zapisují uživatelské programy a démoni
- zařízení `/dev/klog`, do tohoto zařízení zapisuje proces jádra
- síťový port 514/udp, na který posílají data syslogy jiných počítačů

Příklad `/etc/syslog.conf`:

```
kern.*                                  /dev/console
*.info;mail.none;authpriv.none          /var/log/messages
authpriv.*                              /var/log/secure
mail.*                                  /var/log/maillog
*.emerg                                 *
uucp,news.crit                          /var/log/spooler
```

Jak již bylo zmíněno, do syslogu může zapisovat jakýkoli proces. Proto se v systémovém logu mohou vyskytovat i falešné záznamy.

5.5 Inetd a TCP wrapper

Inetd (InterNET Daemon) je často nazýván jako superdémon a slouží ke spouštění internetových služeb [18, 11]. Služby spouštěné přes inetd neběží stále, ale jsou spouštěny až při příchodu spojení. Inetd se chová dvěma možnými způsoby:

- po spuštění služby čeká, až se služba ukončí, a pak znovu poslouchá na daném portu
- po spuštění služby začne ihned opět poslouchat na portu a při příchodu nového spojení spustí další kopii služby

Použitím inetd lze snížit nároky na prostředky systému. Superdémon se používá zvláště u služeb, které nejsou často používány. Konfigurační soubor démona se skládá z jednotlivých řádků. Na každém z nich je uvedeno jméno služby a jaký program má být spuštěn[3].

Příklad `/etc/inetd.conf`:

```
telnet  stream  tcp     nowait  root    /usr/sbin/tcpd  in.telnetd
ftp     stream  tcp     nowait  root    /usr/sbin/tcpd  in.ftpd -l -a
smtp    stream  tcp     nowait  root    /usr/bin/smtpd  smtpd
talk    dgram   udp     wait    root    /usr/sbin/tcpd  /usr/bin/kotalkd
finger  stream  tcp     nowait  root    /usr/sbin/tcpd  in.fingerd
```

Způsob, jakým jsou služby pomocí inetd spouštěny, nám dává možnost kontrolovat přístup ke službám. Pro tento účel byl vytvořen program *TCPwrapper*.

TCPwrapper [11] umožňuje zakazovat a povolovat přístup ke službám podle doménového jména nebo IP adresy klienta. Spuštění služby zaznamenává do systémového logu. Kontrola klienta je prováděna vzhledem k následujícím souborům: `/etc/hosts.allow` a `/etc/hosts.deny`. V příkladu `/etc/inetd.conf` vidíme použití TCPwrapperu u služeb telnet, ftp, talk a finger.

Soubory `hosts.allow` a `hosts.deny` mají stejný formát. Každý řádek je ve tvaru: `<seznam služeb>:<seznam klientů>[:<příkaz>]`

- `<seznam služeb>` je mezerami nebo čárkami oddělený seznam jmen služeb, na které je aplikováno omezení; místo jména služby lze použít hodnotu `ALL`, která vyhovuje libovolnému jménu
- `<seznam klientů>` je seznam počítačů, kterých se omezení týká; zde může být uveden seznam doménových jmen, IP adres; lze použít `ALL`, `EXCEPT` a další[4].
- `<příkaz>` je nepovinnou částí; příkaz je proveden, pokud podmínky uvedené v seznamu služeb a seznamu klientů jsou platné

TCPwrapper nejprve prochází soubor `/etc/hosts.allow`, pokud klient vyhoví podmínkám, je mu přístup povolen. Dále prochází `/etc/hosts.deny`, klient splňující uvedené podmínky je odmítnut. Pokud klient nevyhověl předchozím filtrům, je jeho požadavku vyhověno.

[3] Přesnější informace jsou uvedeny v manuálových stránkách `inetd(8)`.

[4] Podrobnější dokumentaci poskytují manuálové stránky `hosts_access(5)` nebo publikace [18].

5.6 Historie shellu

Další možností, jak zjistit uživatelem prováděné činnosti, je soubor historie shellu. Každý shell zaznamenává spouštěné příkazy do své historie. Soubor s historií je často uložen v domovském adresáři uživatele, jeho jméno je obvykle ve tvaru `.<jméno shellu>_history`, např. `.bash_history`.

Historie má omezenou délku, tzn. shell soubor s historií sám zkracuje, proto nemůže dosáhnout gigantických rozměrů. Do historie jsou ukládány všechny příkazy provedené v *interaktivním režimu* shellu. Př.: do historie bude zapsáno spuštění skriptu, ale již se tam neobjeví příkazy, které tvoří jeho tělo. Procházením historie příkazů lze kontrolovat činnosti prováděné uživateli.

Kapitola 6

Lokální scanner

Tato kapitola obsahuje popis instalace a implementace programu *lokální scanner*, který byl vytvořen v rámci praktické části této práce. Tento systém slouží k vyhledávání bezpečnostních problémů lokálního systému, tj. neprovádí žádné testování vzdálených strojů. Program byl vytvářen s důrazem na jeho rozšiřitelnost a přenositelnost na jiné platformy.

Celý systém je rozdělen na ovládací program a na testy. Každý test je samostatnou komponentou celého systému. Hlavním cílem vývoje bylo splnit snadnost tvorby vlastních testů. Proto byl zvolen systém pluginů[1], které komunikují s řídícím programem pomocí jednoduchého rozhraní. Součástí systému jsou ukázky možných pluginů.

Protože systém provádí testy, které se týkají bezpečnosti počítačového systému, byl kladen důraz na co nejmenší možnost zneužití. Základní zabezpečení programu:

- řídící prostředí (program) může spouštět pouze superuživatel
- přístupová práva k adresáři se systémem dovolují přístup pouze superuživateli
- binární podobu testů nelze spustit samostatně
- každé spuštění programu je zaznamenáváno do logu programu a do systémového logu

6.1 Instalace

Instalace systému je snadná. Je vhodné ji provádět pod uživatelem `root`, protože program lze zneužít k proniknutí do systému. Systém je distribuován ve formě zdrojových textů. Pro úspěšnou instalaci provedeme následující kroky:

- zkopírujeme balík s programem z přiložené diskety do cílového adresáře, kde bude program nainstalován. Pozn.: program si vytvoří vlastní adresářovou strukturu. Provedeme pomocí příkazu[2]:
  ```
  # cp /mnt/floppy/scanner.tar.gz ~
  ```

[1] Tento systém pluginů je také použit v programu Nessus, ze kterého bylo částečně čerpáno. Nessus [14] je komplexní systém na provádění testů vzdálených počítačů a sítí.

[2] Program budeme instalovat do domovského adresáře uživatele `root`, všechny uvedené příkazy jsou zapsány v tomto kontextu.

- rozbalíme zkopírovaný balík, např. pomocí:
  ```
  # cd ~
  # gzip -dc scanner.tar.gz | tar xv
  ```
- nyní je program rozbalen do vlastní adresářové struktury. Zdrojový balík můžeme smazat:
  ```
  # rm ~/scanner.tar.gz
  ```

Před vlastní kompilací je třeba spustit konfigurační skript, který připraví program k překladu. Přejdeme do adresáře s programem a spustíme konfigurační skript:

```
# cd ~/scanner
# ./configure
```

Tento skript zjišťuje, zda jsou nainstalovány programy `make`, překladač jazyka C a požadované knihovny. Pokud potřebné programy chybí, není možné pokračovat v instalaci. Dále položí několik dotazů:

- dotaz na úplnou cestu k nainstalovanému programu (včetně adresáře `scanner`), v názvu cesty nesmí být použity žádné proměnné prostředí nebo znak `~`. V našem případě jméno adresáře může být[3]: `/root/scanner`
- další otázkou je, zda zahrnout symboly pro ladění do výsledné binární podoby programu. Implicitní odpověď je ne.

Konfigurace před kompilací je dokončena. Provedeme překlad:

```
# make clean
# make all
```

Po úspěšném provedení překladu mají všechny soubory a adresáře z důvodu vyšší bezpečnosti zrušena všechna práva přístupu pro skupinu a pro ostatní. Nyní přejdeme ke konfiguraci parametrů programu.

6.2 Konfigurace

Konfigurace programu je uložena v souboru `scanner.conf`. Na příkladu konfiguračního souboru si vysvětlíme významy jednotlivých položek.

Příklad `scanner.conf`:

```
# Scanner's Default Setup
#
# Every line begining with a '#' is a comment
#
```

[3] pokud je adresář `/root` domovským adresářem superuživatele

```
# Plugins run with this UID
plugins_uid = 500

# Plugins directory name (relative or absolute path)
plugins_folder = plugins
# Default plugin's timeout, plugin will be killed after this time
# Maximum value is TIMEOUT_MAX seconds (defined in src/config.h)
plugins_timeout = 80

# Default number of plugins running concurently
# Maximum value is THREADS_MAX (defined in src/config.h)
max_threads = 20

# Log file name (optionaly with path)
logfile = scanner.log
# end.
```

Řádek začínající znakem `#` je komentář. Významy jednotlivých položek:

- `plugins_uid = 500` – tato položka nastavuje UID uživatele, pod kterým budou spouštěny všechny testy. Pro zjišťování bezpečnostních problémů je nutné mít práva běžného uživatele, provádět testy pod identitou superuživatele nemá žádný smysl.
- `plugins_folder = plugins` – cesta k adresáři s binárními soubory pluginů. Pokud cesta není absolutní, je vztažena vůči kořenovému adresáři programu.
- `plugins_timeout = 80` – čas v sekundách. Tuto dobu může plugin provádět testy, po uplynutí času bude plugin ukončen. Tento čas je implicitní dobou pro všechny pluginy – každý plugin má možnost si timeout změnit.[4]
- `max_threads = 20` – nejvyšší počet současně běžících testů.[5]
- `logfile = scanner.log` – název souboru pro záznam hlášení programu, může být i s cestou.

Po provedení konfigurace můžeme program spustit.

[4] Maximální možná hodnota timeoutu je uvedena ve zdrojovém textu souboru `src/config.h`, implicitní maximum je 15 minut.

[5] Horní mez počtu je uvedena ve zdrojovém textu `src/config.h` jako konstanta, implicitní maximum je 25.

6.3 Parametry a ovládání programu

Systém se spouští skriptem `run`. Podporované parametry:

`-d, -D` – zapíná výpis ladících informací do logovacího souboru

`-h, -H` – výpis stručné nápovědy

`-v, -V` – verze programu

– při neznámém parametru vypíše nápovědu a skončí

Program poskytuje uživateli příjemné a snadno ovladatelné rozhraní. Ovládací klávesy použité v celém programu:

Enter – tato klávesa provádí volbu zvýrazněného tlačítka, volbu tlačítka lze také provést pomocí *horkých kláves*, které jsou v názvu tlačítka zvýrazněny

Tab – slouží k přesunu mezi tlačítky,

↑ – přesun o jednu položku výše v zobrazeném seznamu,

↓ – přesun o položku níže v seznamu

→ – posun textu v okně vlevo, pouze když je vpravo nějaký nezobrazený text

← – posune text vpravo, pouze když je vlevo nezobrazený text

␣ – viz klávesa Enter

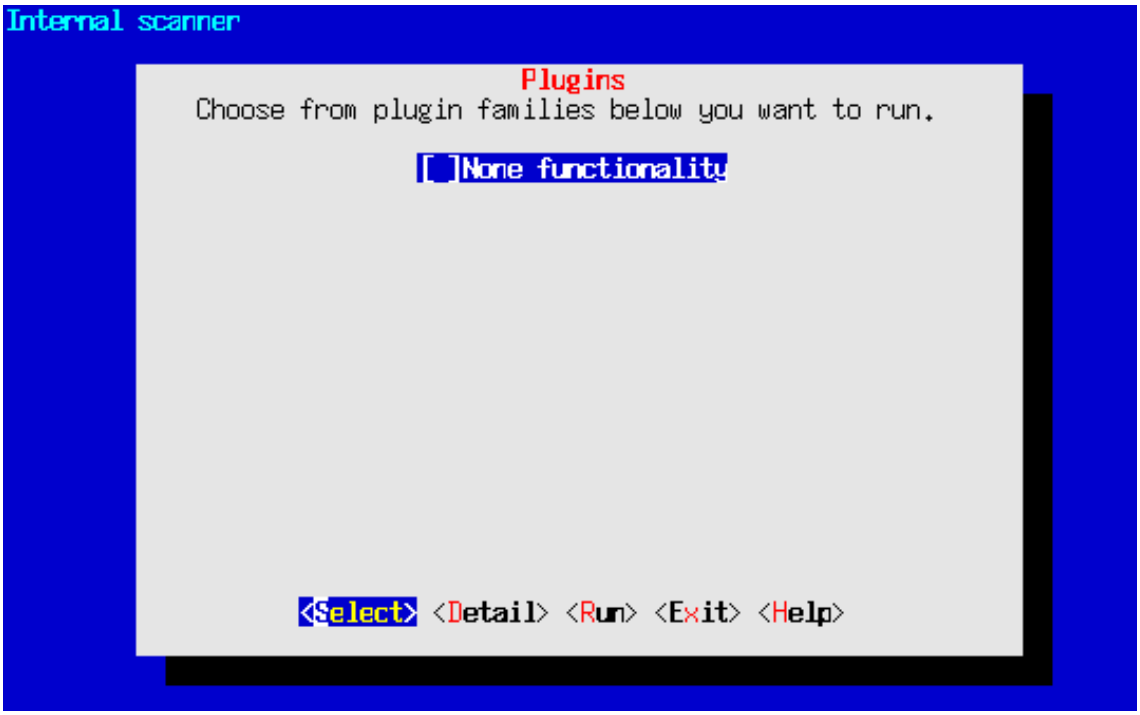
Po spuštění programu se zobrazí hlavní okno (obr. 6.1), kde jsou zobrazeny všechny rodiny pluginů. V tomto seznamu může uživatel volbou `<Select>` označovat celé skupiny (rodiny pluginů). Označení položky (skupiny pluginů) indikuje znak vlevo od názvu položky:

`X` (velké X) – vyjadřuje označení všech pluginů ve skupině

`x` – některé, ale ne všechny, pluginy jsou označeny

(nic) – ve skupině nejsou označeny žádné pluginy

Pokud uživatel zvolí volbu `<Detail>`, zobrazí se detailní okno (obr. 6.2) se seznamem pluginů, které přísluší vybrané skupině. Volbou `<Run>` se spustí testy (všechny označené pluginy). `<Exit>` slouží k ukončení programu. Poslední volbou `<Help>` zobrazíme nápovědu.



Obrázek 6.1: Hlavní obrazovka se seznamem rodin pluginů

Obrázek 6.2: Obrazovka s výběrem jednotlivých pluginů

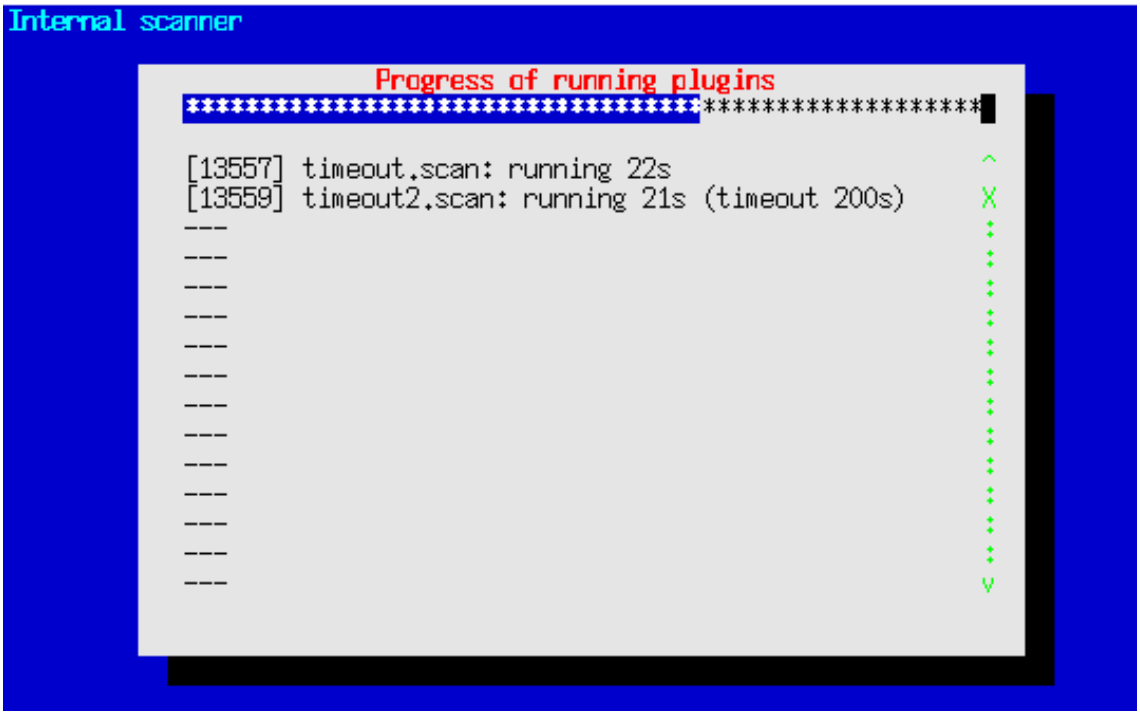
Detailní okno (obr. 6.2) umožňuje následující volby:

`<Select>` – označení/zrušení označení jednotlivých testů

`<Close>` – uzavření detailního okna

`<Help>` – zobrazení nápovědy o vybraném testu (jeho popis)

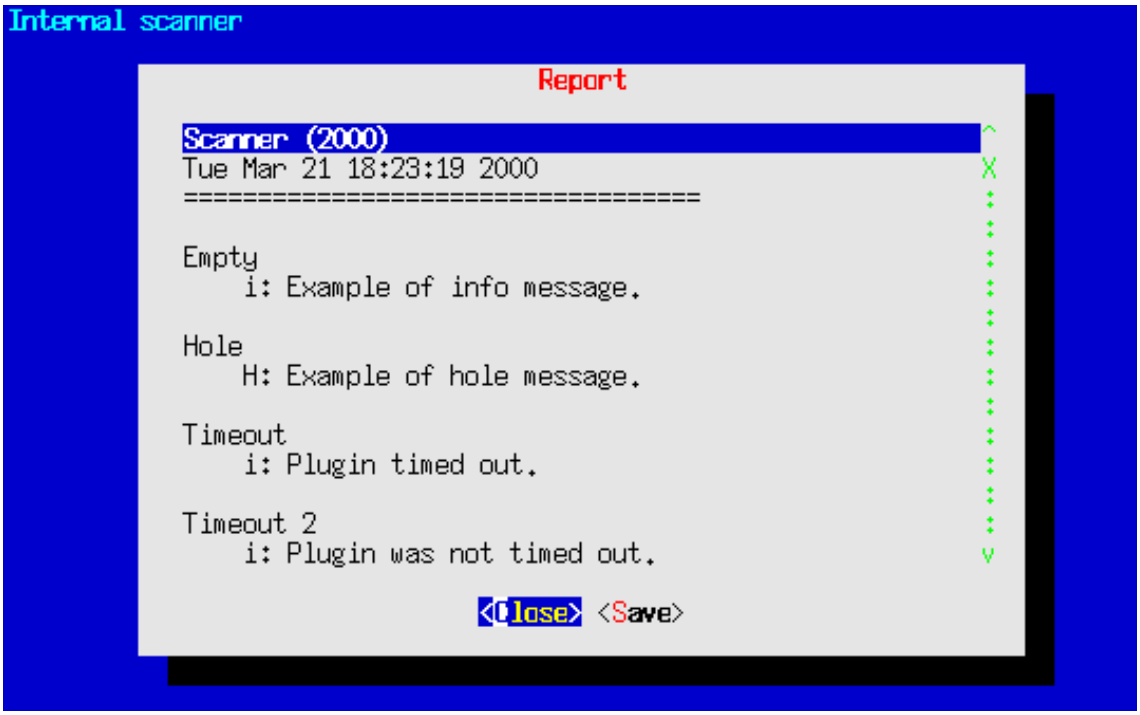
V titulku detailního okna je název rodiny pluginů, pro kterou bylo detailní okno otevřeno.



Obrázek 6.3: Obrazovka s průběhem testů

Průběh testů zobrazuje zvláštní okno (obr. 6.3). V zobrazeném seznamu vidíme právě běžící testy. Každá položka seznamu zobrazuje následující informace: PID pluginu, název testu, čas v sekundách, který už test běží, a případně hodnotu timeoutu, pokud je nastavena jinak než implicitně. V seznamu je možné se pohybovat pomocí šipek. V horní části okna je zobrazen celkový průběh všech testů. Aktualizace údajů se provádí desetkrát za sekundu. Již spuštěné testy nelze přerušit.

Po ukončení všech testů je zobrazeno okno s výsledky (obr. 6.4). Zde jsou zobrazeny pouze ty testy, které vytvořily nějaké hlášení. Výsledky je možno volbou `<Save>` uložit do textového souboru (v názvu souboru je akceptován znak ~). Po uzavření okna se zobrazí zpět hlavní okno programu.



Obrázek 6.4: Obrazovka s výsledky testů

6.4 Deinstalace

Instalace programu probíhá pouze do jednoho adresáře `scanner` a jeho podadresářů. K provedení deinstalace programu stačí smazat tento adresář včetně jeho podadresářů.

6.5 Popis implementace

Celý systém je napsán v programovacím jazyce C s využitím knihovny `ncurses` [12, 19]. Knihovna `ncurses` poskytuje rozhraní pro práci s terminálovým oknem a byla využita při psaní uživatelského rozhraní.

Z pohledu binárních souborů je systém rozdělen na tři části: *řídící program*, *sdílená knihovna* a jednotlivé *pluginy*. Tyto části jsou podrobněji popsány v následujících podkapitolách.

6.5.1 Architektura

Architektura systému je rozdělena na tři samostatné části: řídící část, část testů a část rozhraní. Všechny komponenty mezi sebou komunikují pomocí nepojmenovaných rour. Každá část je reprezentována jedním procesem (obr. 6.5).

Proces *řídící prostředí* obsluhuje uživatelské rozhraní a procesy *rozhraní*. Od procesů rozhraní čte data, která generují jednotlivé testy. Tato data (výsledky testů) jsou později

Obrázek 6.5: Procesy systému

použita pro generování souhrnné zprávy.

Každý test je spouštěn v samostatném procesu *plugin*. Systém omezuje dobu běhu každého testu timeoutem. Proto z důvodu snazší implementace timeoutů přidáváme procesy rozhraní. Každému procesu plugin odpovídá jeden proces rozhraní.

Proces rozhraní kontroluje dobu běhu pluginu a případně test ukončí. Další úlohou tohoto procesu je přeposílat data generovaná pluginem do procesu řídící prostředí.

Procesy byly použity z důvodu stability celého systému. Pokud test provede neplatnou instrukci, bude jeho proces ukončen. Ale to nebude mít vliv na ostatní testy ani na řídící prostředí, protože se jedná o samostatný proces.

6.5.2 API

Pro tvorbu vlastních testů je určeno aplikační rozhraní (API) poskytované systémem. Tabulka 6.1 uvádí seznam funkcí využitelných při tvorbě pluginů. Podrobný popis jednotlivých funkcí je uveden v příloze A.

6.5.3 Sdílená knihovna

Sdílená knihovna obsahuje programový kód všech funkcí, které jsou nutné pro správný běh pluginu a pro správnou spolupráci pluginu s řídícím programem. Zavedení sdílené dynamické knihovny bylo nutné z důvodu sdílení globálních proměnných a programového kódu mezi řídícím programem a pluginy.

6.5.4 Řídící program

Spuštěný řídící program je proces *řídící prostředí*. Program vytváří uživatelské rozhraní, které je popsáno v části 6.3.

Jeho dalším úkolem je spouštět pluginy a kontrolovat jejich běh. Při práci s pluginy program používá čtyři globální proměnné:

- `plugins_torun` – seznam pluginů, které mají být spuštěny
- `plugins_running` – seznam aktuálně běžících pluginů
- `plugins_done` – seznam pluginů, které již svoji činnost ukončily

Funkce	**Účel**
`emalloc`	alokace paměti s vynulováním
`efree`	uvolnění alokované paměti
`estrdup`	duplikace řetězce
`estrlen`	délka řetězce
`plug_set_family`	nastavuje rodinu pluginu
`plug_get_family`	vrací rodinu pluginu
`plug_set_timeout`	změna implicitního timeoutu na jinou hodnotu
`plug_set_name`	nastavení jména pluginu
`plug_get_name`	vrací jméno pluginu
`plug_set_summary`	nastavení krátké informace o pluginu
`plug_get_summary`	vrací krátký popis pluginu
`plug_set_description`	nastavení popisu činnosti pluginu
`plug_get_description`	vrací popis pluginu
`plug_set_category`	nastaví kategorii
`plug_set_copyright`	nastavuje autorská práva pluginu
`post_hole`	zašle hlášení o nalezeném bezpečnostním problému
`proto_post_hole`	obecnější verze `post_hole`
`post_info`	hlášení o možném problému (informativní charakter)
`proto_post_info`	obecnější verze `post_info`
`get_preference`	vrací položku z nastavení
`add_plugin_preference`	přidá položku do nastavení pluginu
`get_plugin_preference`	vrací položku ze seznamu nastavení pluginu

Tabulka 6.1: Přehled API funkcí

- `plugins_all` – seznam všech pluginů z adresáře `scanner/plugins`, které byly úspěšně inicializovány

Jestliže uživatel zvolí volbu `<Run>`, program všechny označené pluginy přidá do seznamu `plugins_torun`. Pokud nejsou označeny žádné testy, je uživatel upozorněn a je mu znovu nabídnut výběr testů.

Program cyklicky prochází seznam běžících pluginů tak dlouho, dokud není prázdný a dokud je co spouštět (`plugins_torun` je neprázdný). V jednotlivých průchodech provádí následující akce:

- kontrola stavu procesu plugin – pokud je ukončen, je ze seznamu `plugins_running` přesunut do `plugins_done`
- přečtení a uložení informací z roury spojující procesy řídící prostředí a rozhraní[6]
- pokud počet běžících testů je menší než maximální povolený, pak program spustí plugin a přesune ho ze seznamu `plugins_torun` do seznamu `plugins_runnning`
- aktualizace informací na obrazovce (obr. 6.3)

Po ukončení běhu zobrazí výsledky (obr. 6.4) a všechny pluginy odstraní ze seznamu `plugins_done`.

6.5.5 Plugin

Plugin je část systému, ve které jsou spouštěny vlastní testy. Z implementačního pohledu je to dynamická knihovna. Toto řešení bylo zvoleno z důvodu zabránění spouštění testů samostatně (bez programu) a z důvodu možnosti přidávat k systému nové testy, bez nutnosti znovu překládat celý systém. Každý plugin musí obsahovat následující dvě funkce:

- `plug_init` – sloužící k inicializaci pluginu
- `plug_run` – určenou ke spouštění testu

Plugin využívá dříve popsané API funkce (viz 6.5.2).

Psaní pluginu

Tato část obsahuje doporučení a rady pro psaní funkčních pluginů. Nejprve musíme definovat dvě funkce (`plug_init` a `plug_run`), které mají následující prototypy:

```
int plugin_init(struct arglist *desc);
int plugin_run(struct arglist * env);
```

[6] viz obrázek 6.5

Funkce `plug_init` je určena k inicializaci, tj. k nastavení jména testu, jeho popisu, jméno autora, timeoutu apod. Nic jiného není vhodné v této funkci provádět. Inicializace pluginu probíhá v procesu řídící prostředí, tedy všechna provedená nastavení budou uchována.

Funkce `plug_run` je určena k vlastní implementaci testu. Test běží již v samostatném procesu, proto nemá smysl provádět nastavení (nebudou uchována). Tato funkce je vždy volána po `plug_init`. Při psaní testu je nutné brát v úvahu, že testy mohou být spouštěny paralelně. Plugin komunikuje s procesem řídící prostředí pomocí funkcí `post_hole` a `post_info`.

Každý plugin má omezenou dobu svého běhu timeoutem. Po uplynutí této doby bude test ukončen.[7] Každé volání funkce `post_hole` nebo `post_info` nuluje dobu dosavadního běhu, tj. dává pluginu k dispozici dalších „timeout“ sekund běhu. Pokud je funkce `post_hole` nebo funkce `post_info` volána s prázdnou zprávou, není přidána do výsledného hlášení.

Příklad: (plugin, který se pokusí otevřít soubor `/etc/shadow` pro čtení)

```
#include <stdio.h>

#include "plug_interface.h"

#define NAME "Shadow"
#define DESC "Shadow plugin attempts to read /etc/shadow file."
#define COPYRIGHT "No copyright"
#define SUMM "Shadow plugin"
#define FAMILY "Examples"


int plugin_init(struct arglist *desc);
int plugin_init(struct arglist *desc)
{
  plug_set_name(desc, NAME);
  plug_set_description(desc, DESC);
  plug_set_summary(desc, SUMM);
  plug_set_copyright(desc, COPYRIGHT);
  plug_set_category(desc, ACT_GATHER_INFO);
  plug_set_family(desc, FAMILY);
  return(0);
}


int plugin_run(struct arglist * env);
int plugin_run(struct arglist * env)
{
```

[7] signálem SIGTERM

```
    FILE *f;

    if ( (f = fopen("/etc/shadow", "r")) != NULL ) {
           post_hole(env, "Ordinary user can read /etc/shadow file !!!");
           fclose(f);
     }
    return(0);
  }
```

Několik dalších ukázek je přiloženo k systému.

Pro využití automatického překladu vytvořeného pluginu a jeho automatické instalace[8] je nutné respektovat následující pokyny:

- vytvořit adresář `<jméno>`, který musí být podadresářem `scanner/src/plugins`
- použít soubory z adresáře `scanner/src/plugins/skeleton`
- modifikovat jméno pluginu v souboru `plugin.tmpl` na `<jméno>`
- přejmenovat soubor `newplug.c` na `<jméno>.c`

Po splnění těchto kroků bude plugin překládán a instalován automaticky při překladu celého systému.

[8] Binární soubory s pluginy se instalují do adresáře `scanner/plugins`.

Kapitola 7

Závěr

Cílem této diplomové práce bylo ukázat základní bezpečnostní model UNIXu, popsat jeho publikovaná rozšíření a popsat základní metody detekce vniknutí do systému, které UNIX poskytuje. Tato práce nedává úplný výčet možných bezpečnostních problémů, spíše se snaží upozornit a ukázat problémy stávajícího systému a jeho rozšíření. Postihnout všechny problémy není možné, již jen z důvodu velice rychlého rozvoje nových technologií a jejich nasazování do praxe – to všechno přináší nové, ještě neznámé, bezpečnostní problémy, kterými se musíme zabývat.

Hlavním cílem práce bylo implementovat systém, který poskytuje prostředí pro provádění testů, které upozorňují na možné bezpečnostní problémy. Důležitou částí implementace bylo vytvořit jednoduché rozhraní umožňující psát vlastní testy.

Literatura

[1] Arnold N. Derek: *UNIX security: a practical tutorial*,
McGraw-Hill, 1992, ISBN 0-07-002559-2

[2] Bace Rebecca: *Intrusion Detection*,
MTP, 1999, ISBN 1-578-70185-6

[3] Bach M. J.: Principy operačního systému UNIX,
SAS Praha, 1993

[4] Blaze Matt: *A Cryptographic File System for Unix*,
Proceedings of the First ACM Conference on Computer and Communications Security, Fairfax, VA, November 1993

[5] Brandejs Michal, Ing., CSc.: *UNIX-LINUX Praktický průvodce*,
Grada Publishing, 1996, ISBN 80-7169-170-4

[6] Burk Robin: *Unix Unleashed*,
Třetí vydání, Sams, 1998, ISBN 0-672-31411-8

[7] Card R., Ts'o T., Tweedie S.: *Design and Implementation of the Second Extended Filesystem*,
In Proceedings of the First Dutch International Symposium on Linux, ISBN 90-367-0385-9

[8] Garfinkel Simson, Spafford Gene: *Pratical UNIX & Internet Security*,
Druhé vydání, O'Reilly & Associates, Inc., 1996, ISBN 1-56592-148-8

[9] Gruska Jozef: *Foundations of Computing*,
International Thompson Computer Press, 1997

[10] IEEE: *POSIX Draft 17 – System Application Program Interface*,
POSIX.6, October 1997

[11] Mann Scott, Mitchell Ellen L.: *Linux System Security: The Administrator's Guide to Open Source Security Tools*,
Prentice Hall, 1999, ISBN 0-130-15807-0

[12] Manuálové stránky knihovny `ncurses(3)`

[13] National Institute of Standards and Technology: *Security in Open Systems*, NIST, 1994, `http://www-08.nist.gov/nistpubs/800-7/main.html`

[14] Nessus project: *Nessus*, `http://www.nessus.org/`

[15] Northcutt Stephen: *Network Intrusion Detection: An Analyst's Handbook*, New Riders, 1999, ISBN 0-735-70868-1

[16] Rohleder David: *Přetečení bufferu*, Linuxové noviny, březen 1998, `http://www.linux.cz/`

[17] Russell Deborah, Gangemi G. T. Sr.: *Computer Security Basics*, O'Reilly & Associates, Inc., 1991, ISBN 0-937175-71-4

[18] Satrapa Pavel, Randus Jiří A.: *Linux - Internet Server*, Neokortex, 1996, ISBN 80-902230-0-1

[19] Skočovský Luděk: *Principy a problémy operačního systému UNIX*, SCIENCE, 1993, ISBN 80-901475-0-X

[20] Stevens W. Richard: *Advanced Programming in the UNIX Environment*, Addison-Wesley, 1992, ISBN 0-201-56317-7

Příloha A

API funkce

Tato příloha obsahuje podrobnou dokumentaci k API funkcím. Seznam API funkcí, zde popisovaných, je uveden v tabulce 6.1.

emalloc

Syntaxe: `void * emalloc(size_t size)`

Popis: Alokuje paměť o velikosti `size` a vrací na ni ukazatel. V případě neúspěchu ukončí program funkcí `exit()`. Alokovanou paměť nuluje.

efree

Syntaxe: `void efree(void *ptr)`

Popis: Uvolňuje dříve alokovanou paměť a nastavuje hodnotu ukazatele na `null`. Ukazatel na alokovanou paměť je nutno předávat referencovaný, tj. `efree(&moje_pamet)`.

estrdup

Syntaxe: `char * estrdup(const char *str)`

Popis: Duplikuje řetězec předaný parametrem `str`. Vrací ukazatel na nově alokovanou paměť obsahující duplikovaný řetězec. Je-li `str` roven `null`, funkce vrací `null`. Pokud se nepovedlo naalokovat paměť pro kopii řetězce, funkce vrací `null`.

estrlen

Syntaxe: `size_t estrlen(const char *s, size_t n)`

Popis: Vrací délku řetězce předaného parametrem `s`. Pokud je proměnná `s` rovna `null`, funkce vrací 0. Je-li délka řetězce větší než `n`, funkce vrací `n`.

plug_set_family

Syntaxe: `void plug_set_family(struct arglist *desc, const char *family)`

Popis: Funkce nastavuje rodinu pluginu. Název rodiny je v parametru `family`, parametr `desc`[1]je seznam všech položek nastavení pro plugin. Název rodiny může být libovolný. Hodnotu proměnné `family` si funkce kopíruje.

[1]Hodnota tohoto parametru je předána pluginu v parametru funkce `plug_init` nebo `plug_run`.

plug_get_family

Syntaxe: `const char * plug_get_family(struct arglist *desc)`

Popis: Vrací hodnotu položky rodina ze seznamu `desc` nastavení pro plugin. Pokud není hodnota nastavena, funkce vrací `null`.

plug_set_timeout

Syntaxe: `void plug_set_timeout(struct arglist *desc, int timeout)`

Popis: Funkce nastavuje pluginu vlastní hodnotu timeoutu. Parametr `timeout` obsahuje novou hodnotu, `desc` seznam položek nastavení pro plugin.

plug_set_name

Syntaxe: `void plug_set_name(struct arglist *desc, const char *name)`

Popis: Nastavuje název pluginu. Nový název je v `name`. Funkce si obsah proměnné `name` zkopíruje. Parametr `desc` obsahuje seznam položek nastavení pro plugin.

plug_get_name

Syntaxe: `const char * plug_get_name(struct arglist *desc)`

Popis: Vrací nastavení položky name. Pokud není jméno nastaveno, vrací `null`. Seznam položek je v proměnné `desc`.

plug_set_summary

Syntaxe: `void plug_set_summary(desc, summary)`

Popis: Nastavuje krátký popis pluginu. Obsah `summary` si funkce kopíruje. Parametr `desc` je seznam položek nastavení pro plugin.

plug_get_summary

Syntaxe: `const char * plug_get_summary(struct arglist *desc)`

Popis: Vrací hodnotu položky summary – krátký popis pluginu. Pokud hodnota není nastavena, funkce vrací `null`.

plug_set_description

Syntaxe:
```
void plug_set_description(struct arglist *desc,
                          const char *description)
```

Popis: Nastavuje popis pluginu. Popis pluginu je zobrazen v okně nápovědy o pluginu. `desc` je seznam položek nastavení pro plugin, do kterého se hodnota `description` přidá. Funkce hodnotu proměnné `description` kopíruje.

plug_get_description

Syntaxe: `const char * plug_get_description(struct arglist *desc)`

Popis: Vrací popis pluginu. Pokud není nastaven, vrací `null`. Proměnná `desc` obsahuje seznam položek nastavení pro plugin.

plug_set_category

Syntaxe: `void plug_set_category(struct arglist *desc, int category)`

Popis: Funkce nastavuje kategorii pluginu. Možné kategorie jsou následující:

`ACT_SCANNER` – plugin prohledává, např. scan portů

`ACT_DENIAL` – plugin znepřístupňuje služby, stroj,...

`ACT_ATTACK` – plugin provádí útok na systém

`ACT_GATHER_INFO` – plugin pouze zjišťuje veřejně přístupné informace

`desc` je seznam položek nastavení pro plugin.

plug_set_copyright

Syntaxe:
```
void plug_set_copyright(struct arglist *desc,
                        const char *copyright)
```

Popis: Nastavení copyrightu pluginu. Proměnnou `copyright` funkce kopíruje. Proměnná `desc` je seznam položek nastavení pro plugin.

post_hole

Syntaxe: `void post_hole(struct arglist *desc, const char *action)`

Popis: Funkce provádí zápis hlášení o nalezeném bezpečnostním problému. Hlášení je obsaženo v parametru `action`. `desc` je seznam položek nastavení pro plugin.

proto_post_hole

Syntaxe: `void proto_post_hole(int fdesc, const char *action)`

Popis: Funkce zapisuje data o nalezeném bezpečnostním problému do deskriptoru `fdesc`. Zapisovaná data jsou vytvořena z obsahu proměnné `action`.

post_info

Syntaxe: `void post_info(struct arglist *desc, const char *action)`

Popis: Funkce provádí zápis informativního hlášení. Hlášení obsahuje text z parametru `action`. `desc` je seznam položek nastavení pro plugin.

proto_post_info

Syntaxe: `void proto_post_info(int fdesc, const char *action)`

Popis: Funkce zapisuje data zjištěná pluginem do deskriptoru `fdesc`. Tato data mají informativní charakter a jsou vytvořena z obsahu parametru `action`.

get_preference

Syntaxe: `char * get_preference(struct arglist *desc, const char *name)`

Popis: Funkce zpřístupňující hodnoty nastavení programu (přístup k souboru `scanner.conf`). Funkce vrací hodnotu položky se jménem `name`, pokud taková položka existuje. Jinak vrací `null`.

add_plugin_preference

Syntaxe:
```
void add_plugin_preference(struct arglist *desc,
     const char *name, const char *type, const char *defaul)
```

Popis: Funkce přidá položku do nastavení (parametr `desc`). Položka se skládá ze jména (parametr `name`), typu ukládané hodnoty (parametr `type`) a ukládané hodnoty (parametr `defaul`). Všechny hodnoty parametrů (kromě `desc`) jsou funkcí kopírovány.

get_plugin_preference

Syntaxe:
```
char * get_plugin_preference(struct arglist *desc,
                             const char *name)
```

Popis: Funkce vrací hodnotu položky z `desc` se jménem `name`. Pokud není položka se jménem `name` nalezena, vrací `null`.

Příloha B

Obsah přiložené diskety

Disketa přiložená k této diplomové práci obsahuje následující soubory a adresáře:

```
/
|- README
|- scanner.md5
|- scanner.tar.gz
|- text                (dir)
|   |- diplomka.md5
|   |- diplomka.ps.gz
|   |- dipl_src.tar.gz
|   |- zadani_dp.tex
|- zdroje              (dir)
```

V souboru `README` je stručný popis obsahu diskety. Adresář `text` obsahuje zdrojový text diplomové práce (`dipl_src.tar.gz`), přeloženou podobu v PostScriptu (`diplomka.ps.gz`) a zadání (`zadani_dp.tex`). MP5 součty těchto souborů jsou v souboru `diplomka.md5`. Některé zdroje, ze kterých jsem čerpal, jsou nakopírovány do adresáře `zdroje`. Soubor `scanner.tar.gz` obsahuje program popisovaný v kapitole 6. V souboru `scanner.md5` je MD5 součet programu:

```
3ff434fd0f2618dbd70d5131cf564475  scanner.tar.gz
```