Construindo Bootkits: Ideias para GRUB2 com Linux
Who am I

- Security Consultant at PRIDE Security
- ....
Previous Work

Startup Overview

- GRUB2 MBR
- GRUB2 Kernel Image
- Linux Kernel (bzImage)
- vmlinux

Core Image (Boot Sector)
Core Image (Decompressor)
Startup Overview

- GRUB2 MBR
- GRUB2 Kernel Image
- Linux Kernel (bzImage)
- vmlinux

Core Image (Boot Sector)
Core Image (Decompressor)

linux | linux16 commands

vmlinux implant

evil process
Startup Overview

- GRUB2 MBR
- Core Image (Boot Sector)
- Core Image (Decompressor)
- GRUB2 Kernel Image
- Linux Kernel (bzImage)
  - vmlinux
  - compressed
- Encrypted Filesystem
  - linux | linux16 commands
Startup Overview

**GRUB2 MBR**

**Core Image (Boot Sector)**

**Core Image (Decompressor)**

**GRUB2 Kernel Image**

**Linux Kernel (bzImage)**

**vmlinux**

**Infect this elements**

**GRUB2 implant**

**bzImage implant**

**vmlinux implant**

**runtime patch**

**runtime patch**

**runtime patch**

**0x7c00**

**0x8000**

**0x8200**

**0x100000**

**Infect this elements**
GRUB2 - Startup Overview

- **GRUB2 MBR**
  - executed from 0x7c00
  - load the next stage to 0x8000
  - jumps to it

- **Core Image (Boot Sector)**
  - executed from 0x8000
  - load next stage to 0x8200
  - jumps to decompressor code

- **Decompressor**
  - executed from 0x8200
  - switch processor to protected mode
  - decompress grub2 kernel to 0x10000
  - jumps to uncompressed grub2 kernel

- **Compressed Data**
  - executed from 0x10000
  - copy kernel image to 0x9000
  - module info and its later are not copied
  - clear bss section
  - call grub_main()

- **Reed-Solomon redundancy (optional)**

- **GRUB2 Kernel Image**
- **Module Info Structure**
- **ELF Modules**
- **Public Keys (optional)**
- **Memory Disk (optional)**
- **Early Config File (optional)**
- **Prefix String (optional)**

grub-core/xxx/xxx/xxx/xxx (entrypoint)
GRUB2 - MBR

kernel_address at 0x5a

kernel_sector at 0x5c

kernel_sector_high at 0x60
GRUB2 - Core Image (Boot Sector)

- implemented by boot/i386/pc/diskboot.S
- loads all sectors of the core image (decompressor and compressed data) to 0x8200
  - uses a table present at the bottom of the sector
  - each entry of the table has the following format:
    ```
    struct _load_entry {
      u32 sector_low;
      u32 sector_high;
      u16 num_of_sectors;
      u16 segment;
    };
    ```
  - we can find a small code cave between the last instruction and the start of the table (~144 bytes)
- jmps to decompressor code
GRUB2 - Core Image (Boot Sector)

The loop goes from bottom to up and stops when it finds `num_of_sectors == 0`

We can add another entries here
We can add code too

```
{ .sector_low = 0x2,
  .sector_high = 0x0,
  .num_of_sectors = 0x65,
  .segment = 0x0820
};
```
GRUB2 - Core Image (Decompressor)

- implemented by different files
  - the main file is grub-core/boot/i386/pc/startup_raw.S
    - includes grub-core/kern/i386/realmode.S
    - includes grub-core/boot/i386/pc/lzma_decode.S

- switch processor to protected mode, ensure a20 line enable
  - uses the function real_to_prot defined in grub-core/kern/i386/realmode.S

- decompress GRUB2 kernel image to 0x100000 jumps to uncompressed kernel
  - two function pointers are passed as argument:
    - prot_to_real, real_to_prot
    - all transitions real mode <-> protected mode are made using these functions
GRUB2 - Core Image (Decompressor)

- some important notes:
  - GRUB2 does not define any interruption handler for protected mode

- the function real_to_prot also sets idtr.base = 0 and idtr.size = 0
  - using the values defined by protidt which is defined as (check grub-core/kern/i386realmode.S):

```
protidt:
    .word 0
    .long 0
```

- we can set another value for protidt (which implies to define some entries for IDT)
- hardware breakpoints might be useful
In the current version of GRUB2, these values are always in the first sector of the decompressor.
GRUB2 - Minimal changes to inject a payload loader

Patch the pointer in the offset 0x5a to jump to LOADER (0x8000 + offset)

LOADER: small piece of code injected into the cave
- reserve memory (e.g.: decreasing "Memory Size" at Bios Data Area)
- load all payloads on memory (int 13)
- execute the first

Patch the variable "protidt" to point to a custom IDT (Interrupt Descriptor Table)
there are some fixed addresses to use, e.g.: anything in the range between 0x7e00 - 0x8000

Payload #1: grub2
Payload #2: bzImage
Payload #3: vmlinux
Payload #4: userspace shellcode

One nice place to put the payloads is the free sectors before the first partition
GRUB2 - Minimal changes to inject a payload loader

GRUB2 MBR

LOADER

Core Image (Boot Sector)

Core Image (Decompressor)

GRUB2 payload #1

Set IDT entries on memory
Set a hardware breakpoint for execution on 0x100000

0x100000

GRUB2 Kernel Image

GRUB2 payload #2

#DB (fault)
GRUB2 - Uncompressed Kernel Image (overview)

The first task is to copy itself from 0x100000 to 0x9000.
Then, the startup code clears the bss section and calls the grub_main function.
Parsing this code we can find the size of the uncompressed kernel.
Every exported symbol of grub2 kernel has an entry in a symbol table.
Each entry of the table has the following format:

```c
struct symtab {
    const char *name;
    void       *addr;
    int        isfunc;
};
```

Finding this table on memory we can find the address of some interesting symbols, e.g.: grub_register_command_prio, grub_file_open, grub_file_read, grub_file_seek, grub_file_close.
GRUB2 - Commands

- Some important functions (both in kernel and modules) are implemented as commands, e.g.: `insmod`, `set`, `unset`, `ls`, `normal`, `linux`, `linux16`, `initrd`, `initrd16`, `ntldr`

- All commands are registered using the function `grub_register_command_prio` which is exported by the kernel, soon has an entry in the symbol table

- Controlling the calls to `grub_register_command_prio` we can find the address of all commands at runtime
- However, some command registrations might have a different meaning, e.g:
  - the module "normal.mod" implements an approach to load all the necessary commands on-demand

    grub_register_extcmd_prio(
    name,
    grub_dyncmd_dispatcher,
    GRUB_COMMAND_FLAG_BLOCKS |
    GRUB_COMMAND_FLAG_EXTCMD |
    GRUB_COMMAND_FLAG_DYNCMD,
    0, N_("module isn't loaded"), 0, prio);

    grub_register_command_prio(
    name,
    grub_extcmd_dispatch,
    0,
    N_("module isn't loaded"),
    prio);

    for all command in the file command.lst

    this ends by registering  the command with a common dispatch function
    the command function will be loaded and registered in the first use

- if we're hooking every call to grub_register_command_prio, we need a way to filter that behaviour
  - a simple way is just to check if the fourth argument is "module isn't loaded"
GRUB2 implant (Controlling Commands)

**GRUB2 MBR**
- **loader**: GRUB2 payload #1
  - Set IDT entries on memory
  - Set a hardware breakpoint for execution on 0x100000

**GRUB2 Kernel Image**
- **GRUB2 payload #1**
- **GRUB2 payload #2**
- **#DB (fault)**
  - Find symtable on memory
  - Find the address of the functions: grub_register_command_prio, grub_file_open, grub_file_read, grub_file_close
  - Hook all of them

**GRUB2 payload #1**
- **hook command function**
  - if command name == “linux” || command name == “linux16”
  - Control file operations to infect the bzImage

**GRUB2 payload #2**
- **hook command function**
  - Control file operations to infect the bzImage
Linux Kernel bzImage (x86_64)

arch/x86/boot/header.S

+0x00

+0x200

16-bit entrypoint

64-bit entrypoint (startup_64 in arch/x86/boot/compressed/head_64.S)

32-bit entrypoint (startup_32 in arch/x86/boot/compressed/head_64.S)
- The first task is to parse the code in memory
  - find the point in decompressor code where the kernel is about to be called
  - patch there, to get execution right before the vmlinux entrypoint
Berni Zafar

Linux Kernel bzImage (x86_64)

```
arch/x86/boot/compressed/head_64.S

leaq relocated(%rbx), %rax
jmp *%rax

relocated:
...

/*
 * Do the extraction, and jump to the new kernel..
 */
pushq %rsi
movq %rsi, %rdi
lea boot_heap(%rip), %rsi
lea input_data(%rip), %rdx
movl $z_input_len, %ecx
movq %rbp, %r8
movq $z_output_len, %r9
call extract_kernel /* returns kernel location in
 %rax */
popq %rsi

/*
 * Jump to the decompressed kernel.
 */
jmp *%rax
```
Linux Kernel bzImage (x86_64)

16-bit entrypoint

32-bit entrypoint

64-bit entrypoint

decompression code

Setup Code

arch/x86/boot/compressed/head_64.S

/*
 * Jump to the relocated address.
 */
leaq relocated(%rbx), %rax
jmp *%rax

relocated:
...
subq %rdi, %rcx
shrq $3, %rcx
rep stosq
pushq %rsi
movq %rsi, %rdi
leaq boot_heap(%rip), %rsi
leaq input_data(%rip), %rdx
movl $z_input_len, %ecx
movq %rbp, %r8
movq $z_output_len, %r9
call extract_kernel /* returns kernel location in
%rax */
popq %rsi

/*
 * Jump to the decompressed kernel.
 */
jmp *%rax

Be careful: This code is slightly different for the kernels v3, v4, v5
Linux Kernel bzImage (x86_64)

Linux >= 3 seems to have a indirect jump after the vmlinuz decompression

```
leaq relocated(%rbx), %rax
jmp *%rax
```

```
subq %rdi, %rcx
shrq $3, %rcx
rep stosq
pushq %rsi
movq %rsi, %rdi
leaq boot_heap(%rip), %rsi
leaq input_data(%rip), %rdx
movl $z_input_len, %ecx
movq %rbp, %r8
movq $z_output_len, %r9
call extract_kernel /* returns kernel location in %rax */
popq %rsi
```

```
/*
 * Jump to the decompressed kernel.
 */
jmp *%rax
```
Payload #2 - Linux Kernel implant

- after decompression...
  - the execution calls startup_64 defined in linux/arch/x86/kernel/head_64.S
  - kernel are using an 1:1 mapping between physical and virtual address spaces (identity pages)
  - the code are running with just one processor (no race conditions)

- vmlinux_implant_start()
  - resolve the virtual address where the kernel will execute
    - get from the switch: identity mapping -> full virtual address mapping
  - find systall table (pattern matching)
  - hook some not implemented syscalls (userspace interface)
Payload #2 - Linux Kernel implant

kernel implant + Systemd Stub + Userspace Shellcode

- btk_manager_loop
  - systemd stay in a loop calling sys_epoll_wait
  - we have the chance to access Systemd context
  - filter process by pid
  - allocate memory, copy systemd stub to process memory

- restore_parent_process
  - unmap memory
  - restore original regs
  - returns to original flow

- init_evil_process
  - copy shellcode to user memory
Payload #2 - Linux Kernel implant

- bootkit manager: hook in sys_epoll_wait
  - wait for init process (systemd): just ignore a number of calls
  - if there is no user space implant running, spawn one
  - be careful with hibernation

- spawning evil process
  - allocate memory (rxw), for now, I use sys_mmap (yeah, inside the kernel)
  - [https://lwn.net/Articles/751052](https://lwn.net/Articles/751052) (different internal syscall calling convention)
  - inject a stub into process memory
  - set new return address on kernel stack
Demo
Questions