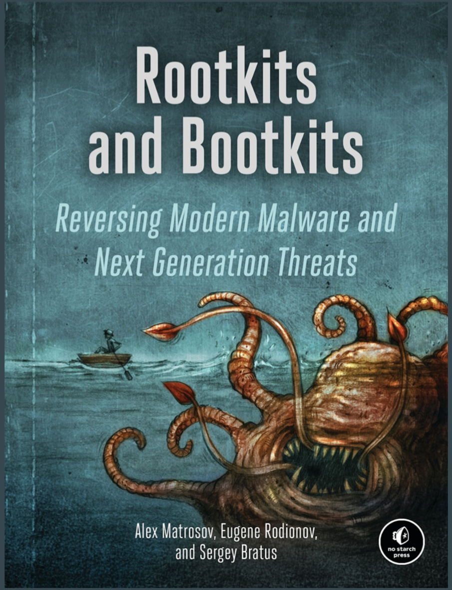
Construindo Bootkits: Ideias para GRUB2 com Linux

Who am I

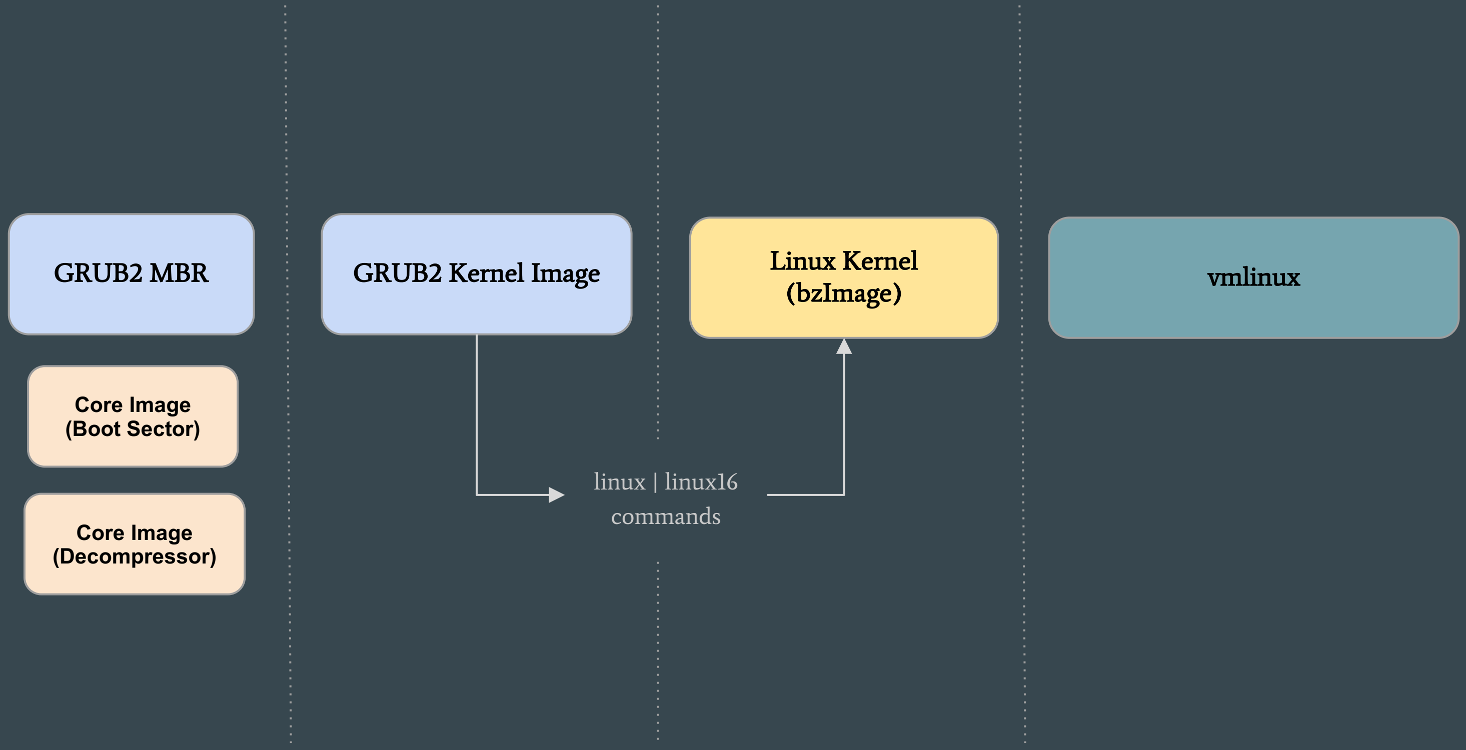
- Security Consultant at PRIDE Security
- ....

Previous Work

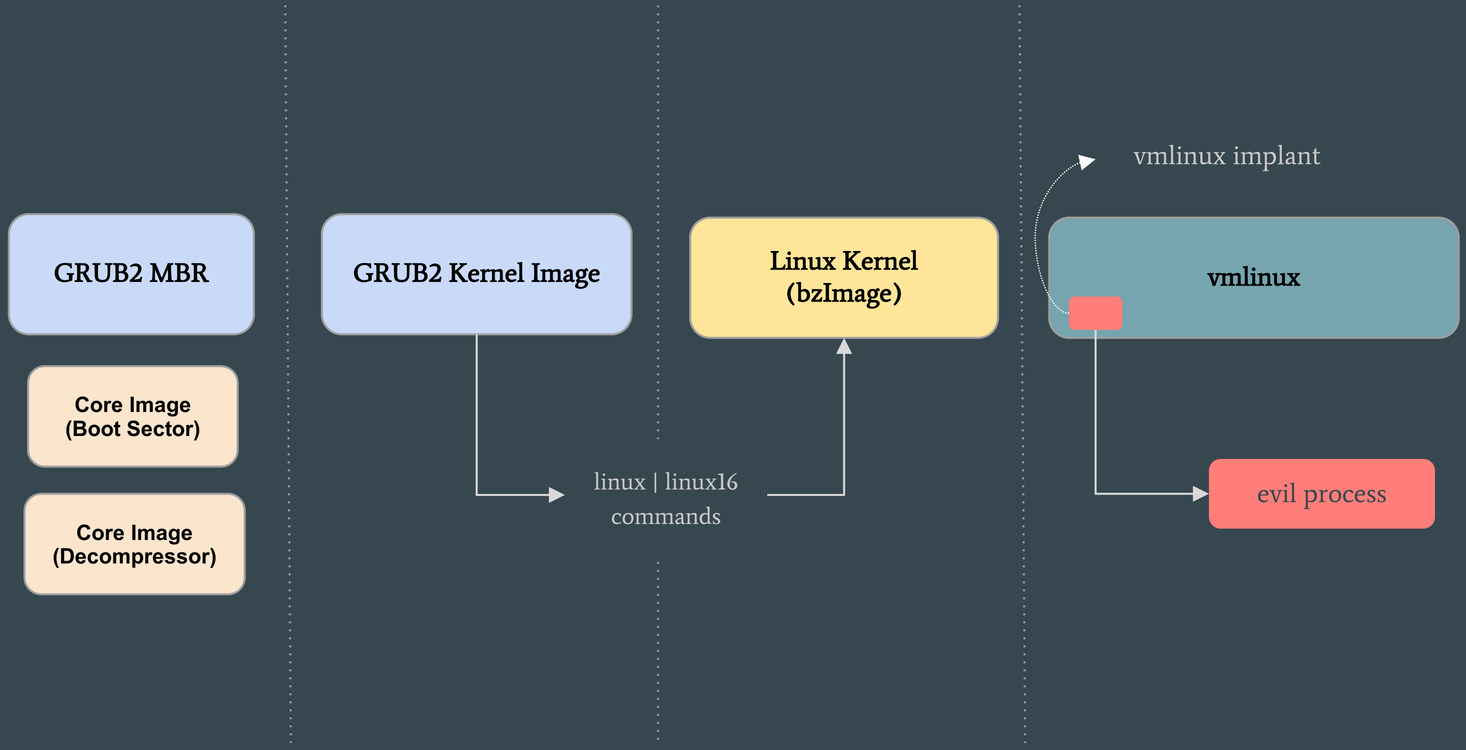


Matrosov, Alex, Eugene Rodionov, and Sergey Bratus. Rootkits and bootkits: reversing modern malware and next generation threats. No Starch Press, 2019.

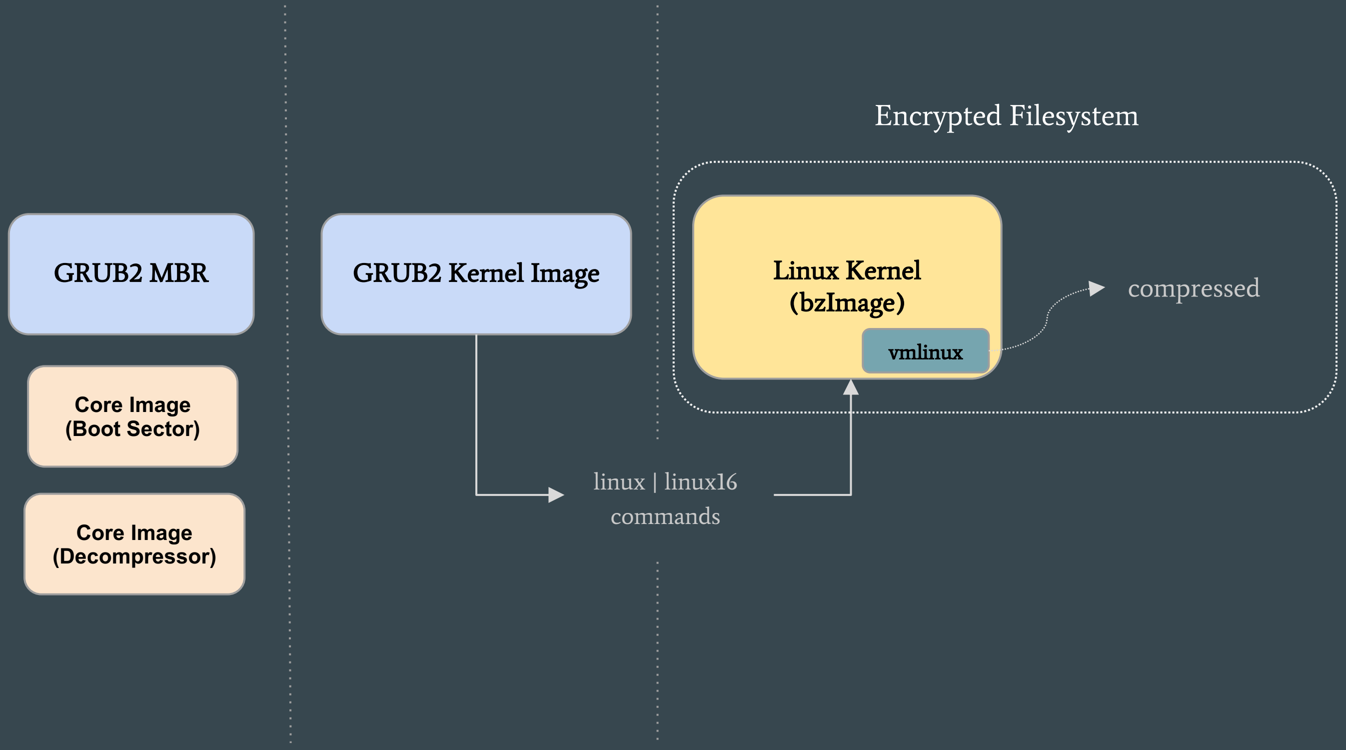
Startup Overview



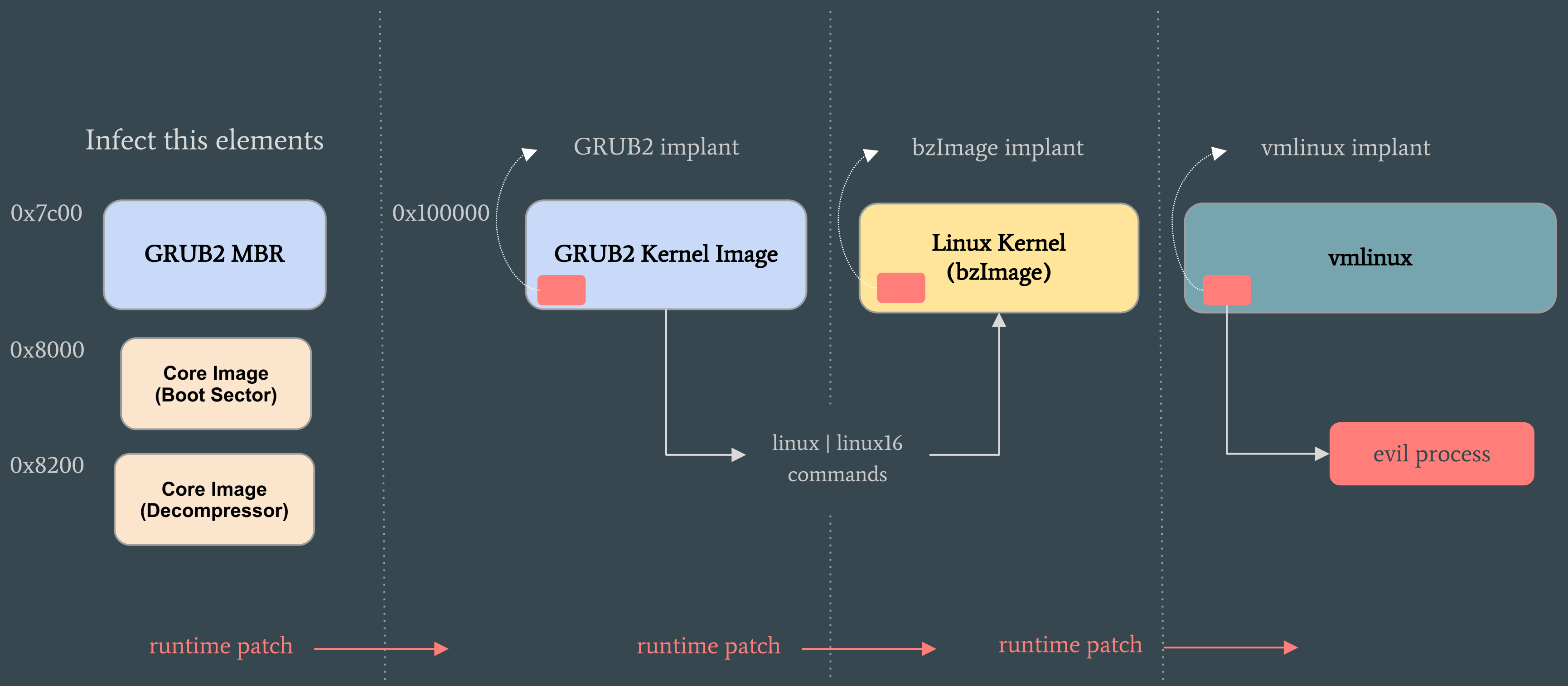
Startup Overview



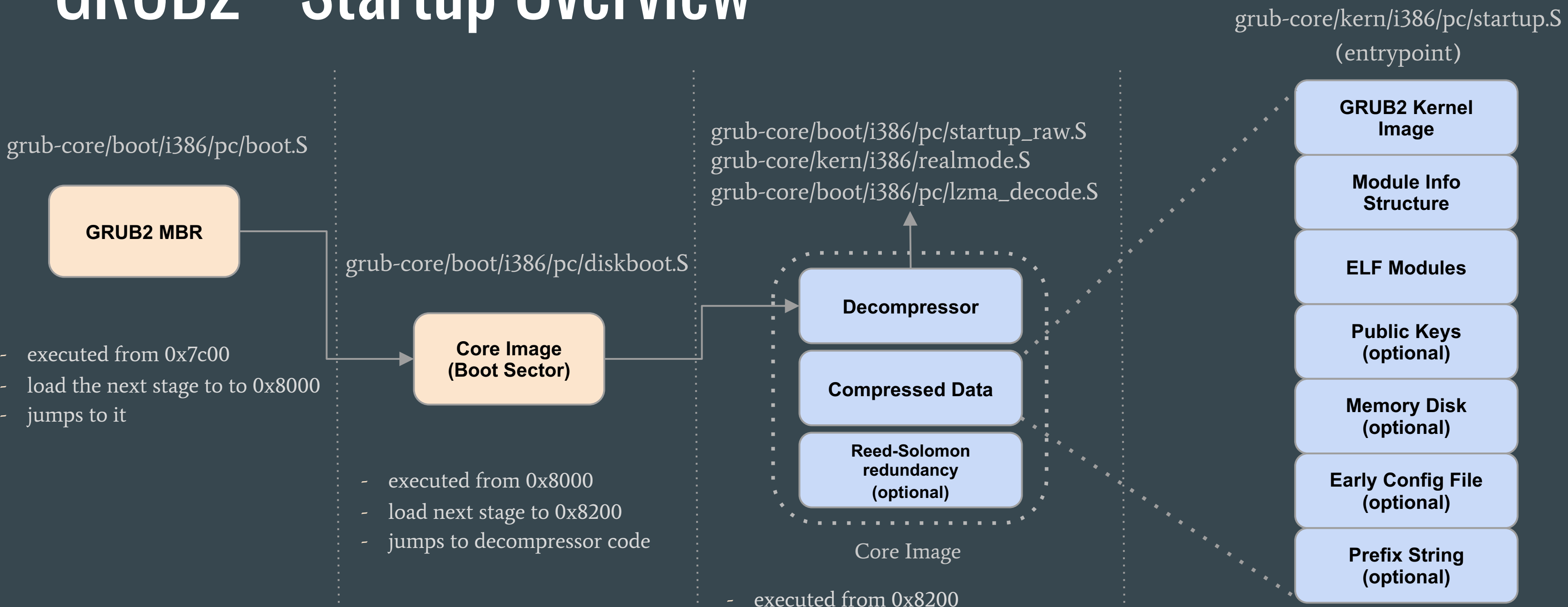
Startup Overview



Startup Overview



GRUB2 - Startup Overview



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

- executed from 0x8000
- load next stage to 0x8200
- jumps to decompressor code

- executed from 0x8200
- switch processor to protected mode
- decompress grub2 kernel to 0x100000
- jumps to uncompressed grub2 kernel

- executed from 0x100000
- copy kernel image to 0x9000
- module info and its later are not copied
- clear bss section
- call grub_main()

GRUB2 - MBR

```
vmdev@pc:~$ sudo hexdump -C -n512 /dev/sda
00000000  eb 63 90 10 8e d0 bc 00  b0 b8 00 00 8e d8 8e c0  |.c..............|
00000010  fb be 00 7c bf 00 06 b9  00 02 f3 a4 ea 21 06 00  |...|.........!..|
00000020  00 be be 07 38 04 75 0b  83 c6 10 81 fe fe 07 75  |....8.u........u|
00000030  f3 eb 16 b4 02 b0 01 bb  00 7c b2 80 8a 74 01 8b  |.........|...t..|
00000040  4c 02 cd 13 ea 00 7c 00  00 eb fe 00 00 00 00 00  |L.....|.........|
00000050  00 00 00 00 00 00 00 00  00 00 00 80 01 00 00 00  |................|
00000060  00 00 00 00 ff fa 90 90  f6 c2 80 74 05 f6 c2 70  |...........t...p|
00000070  74 02 b2 80 ea 79 7c 00  00 31 c0 8e d8 8e d0 bc  |t....y|..1......|
00000080  00 20 fb a0 64 7c 3c ff  74 02 88 c2 52 bb 17 04  |. ..d|<.t...R...|
00000090  f6 07 03 74 06 be 88 7d  e8 17 01 be 05 7c b4 41  |...t...}.....|.A|
000000a0  bb aa 55 cd 13 5a 52 72  3d 81 fb 55 aa 75 37 83  |..U..ZRr=..U.u7.|
000000b0  e1 01 74 32 31 c0 89 44  04 40 88 44 ff 89 44 02  |..t21..D.@.D..D.|
000000c0  c7 04 10 00 66 8b 1e 5c  7c 66 89 5c 08 66 8b 1e  |....f..\|f.\.f..|
000000d0  60 7c 66 89 5c 0c c7 44  06 00 70 b4 42 cd 13 72  |`|f.\..D..p.B..r|
000000e0  05 bb 00 70 eb 76 b4 08  cd 13 73 0d 5a 84 d2 0f  |...p.v....s.Z...|
000000f0  83 d0 00 be 93 7d e9 82  00 66 0f b6 c6 88 64 ff  |.....}...f....d.|
00000100  40 66 89 44 04 0f b6 d1  c1 e2 02 88 e8 88 f4 40  |@f.D...........@|
00000110  89 44 08 0f b6 c2 c0 e8  02 66 89 04 66 a1 60 7c  |.D.......f..f.`||
00000120  66 09 c0 75 4e 66 a1 5c  7c 66 31 d2 66 f7 34 88  |f..uNf.\|f1.f.4.|
00000130  d1 31 d2 66 f7 74 04 3b  44 08 7d 37 fe c1 88 c5  |.1.f.t.;D.}7....|
00000140  30 c0 c1 e8 02 08 c1 88  d0 5a 88 c6 bb 00 70 8e  |0........Z....p.|
00000150  c3 31 db b8 01 02 cd 13  72 1e 8c c3 60 1e b9 00  |.1......r...`...|
00000160  01 8e db 31 f6 bf 00 80  8e c6 fc f3 a5 1f 61 ff  |...1..........a.|
00000170  26 5a 7c be 8e 7d eb 03  be 9d 7d e8 34 00 be a2  |&Z|..}....}.4...|
00000180  7d e8 2e 00 cd 18 eb fe  47 52 55 42 20 00 47 65  |}.......GRUB .Ge|
00000190  6f 6d 00 48 61 72 64 20  44 69 73 6b 00 52 65 61  |om.Hard Disk.Rea|
000001a0  64 00 20 45 72 72 6f 72  0d 0a 00 bb 01 00 b4 0e  |d. Error........|
000001b0  cd 10 ac 3c 00 75 f4 c3  d6 12 9c 7d 00 00 80 20  |...<.u.....}... |
000001c0  21 00 83 fe ff ff 00 08  00 00 00 00 40 03 00 fe  |!...........@...|
000001d0  ff ff 05 fe ff ff fe 0f  40 03 02 e8 7f 00 00 00  |........@.......|
000001e0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
000001f0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 55 aa  |..............U.|
00000200
```

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)

```
vmdev@pc:~$ sudo hexdump -C -n512 -s512 /dev/sda
00000200  52 e8 28 01 74 08 56 be  33 81 e8 4c 01 5e bf f4  |R.(.t.V.3..L.^..|
00000210  81 66 8b 2d 83 7d 08 00  0f 84 e9 00 80 7c ff 00  |.f.-.}.......|..|
00000220  74 46 66 8b 1d 66 8b 4d  04 66 31 c0 b0 7f 39 45  |tFf..f.M.f1...9E|
00000230  08 7f 03 8b 45 08 29 45  08 66 01 05 66 83 55 04  |....E.)E.f..f.U.|
00000240  00 c7 04 10 00 89 44 02  66 89 5c 08 66 89 4c 0c  |......D.f.\.f.L.|
00000250  c7 44 06 00 70 50 c7 44  04 00 00 b4 42 cd 13 0f  |.D..pP.D....B...|
00000260  82 bb 00 bb 00 70 eb 68  66 8b 45 04 66 09 c0 0f  |.....p.hf.E.f...|
00000270  85 a3 00 66 8b 05 66 31  d2 66 f7 34 88 54 0a 66  |...f..f1.f.4.T.f|
00000280  31 d2 66 f7 74 04 88 54  0b 89 44 0c 3b 44 08 0f  |1.f.t..T..D.;D..|
00000290  8d 83 00 8b 04 2a 44 0a  39 45 08 7f 03 8b 45 08  |.....*D.9E....E.|
000002a0  29 45 08 66 01 05 66 83  55 04 00 8a 54 0d c0 e2  |)E.f..f.U...T...|
000002b0  06 8a 4c 0a fe c1 08 d1  8a 6c 0c 5a 52 8a 74 0b  |..L......l.ZR.t.|
000002c0  50 bb 00 70 8e c3 31 db  b4 02 cd 13 72 50 8c c3  |P..p..1.....rP..|
000002d0  8e 45 0a 58 c1 e0 05 01  45 0a 60 1e c1 e0 03 89  |.E.X....E.`.....|
000002e0  c1 31 ff 31 f6 8e db fc  f3 a5 1f e8 3e 00 74 06  |.1.1........>.t.|
000002f0  be 3b 81 e8 63 00 61 83  7d 08 00 0f 85 1d ff 83  |.;..c.a.}.......|
00000300  ef 0c e9 0f ff e8 24 00  74 06 be 3d 81 e8 49 00  |......$.t..=..I.|
00000310  5a ea 00 82 00 00 be 40  81 e8 3d 00 eb 06 be 45  |Z......@..=....E|
00000320  81 e8 35 00 be 4a 81 e8  2f 00 eb fe bb 17 04 f6  |..5..J../.......|
00000330  07 03 c3 6c 6f 61 64 69  6e 67 00 2e 00 0d 0a 00  |...loading......|
00000340  47 65 6f 6d 00 52 65 61  64 00 20 45 72 72 6f 72  |Geom.Read. Error|
00000350  00 bb 01 00 b4 0e cd 10  46 8a 04 3c 00 75 f2 c3  |........F..<.u..|
00000360  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
000003f0  00 00 00 00 02 00 00 00  00 00 00 00 65 00 20 08  |............e. .|
00000400
```

we can add another entries here

we can add code too

the loop goes from bottom to up and stops when it finds *num_of_sectors == 0*

```
{
    .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/i386/realmode.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

GRUB2 - Core Image (Decompressor)

```
vmdev@pc:~$ sudo hexdump -C -n512 -s1024 /dev/sda
00000400  ea 1c 82 00 00 00 00 00  6c 58 00 00 20 ad 00 00  |........lX.. ...|
00000410  54 66 00 00 81 07 00 00  ff ff ff 00 fa 31 c0 8e  |Tf...........1..|
00000420  d8 8e d0 8e c0 66 bd f0  1f 00 00 66 89 ec fb 88  |.....f.....f....|
00000430  16 1b 82 cd 13 66 e8 97  00 00 00 fc e8 8b 06 00  |.....f..........|
00000440  00 8b 15 08 82 00 00 81  c2 bf 03 00 00 8b 0d 10  |................|
00000450  82 00 00 8d 05 81 89 00  00 fc e8 3b 03 00 00 e9  |...........;....|
00000460  7a 07 00 00 f0 ff 07 00  eb 16 8d b4 26 00 00 00  |z...........&...|
00000470  00 8d b4 26 00 00 00 00  8d b4 26 00 00 00 00 90  |...&......&.....|
00000480  00 00 00 00 00 00 00 00  ff ff 00 00 00 9a cf 00  |................|
00000490  ff ff 00 00 00 92 cf 00  ff ff 00 00 00 9e 00 00  |................|
000004a0  ff ff 00 00 00 92 00 00  eb 16 8d b4 26 00 00 00  |............&...|
000004b0  00 8d b4 26 00 00 00 00  8d b4 26 00 00 00 00 90  |...&......&.....|
000004c0  27 00 80 82 00 00 00 04  00 00 00 00 00 00 00 00  |'...............|
000004d0  00 00 fa 31 c0 8e d8 66  0f 01 16 c0 82 0f 20 c0  |...1...f...... .|
000004e0  66 83 c8 01 0f 22 c0 66  ea ef 82 00 00 08 00 66  |f....".f.......f|
000004f0  b8 10 00 8e d8 8e c0 8e  e0 8e e8 8e d0 8b 04 24  |...............$|
00000500  a3 f0 1f 00 00 a1 64 82  00 00 89 c4 89 c5 a1 f0  |......d.........|
00000510  1f 00 00 89 04 24 31 c0  0f 01 0d c6 82 00 00 0f  |.....$1.........|
00000520  01 1d cc 82 00 00 c3 0f  01 15 c0 82 00 00 0f 01  |................|
00000530  0d cc 82 00 00 0f 01 1d  c6 82 00 00 89 e0 a3 64  |...............d|
00000540  82 00 00 8b 04 24 a3 f0  1f 00 00 b8 f0 1f 00 00  |.....$..........|
00000550  89 c4 89 c5 66 b8 20 00  8e d8 8e c0 8e e0 8e e8  |....f. .........|
00000560  8e d0 ea 69 83 00 00 18  00 0f 20 c0 66 83 e0 fe  |...i...... .f...|
00000570  0f 22 c0 66 ea 7b 83 00  00 00 00 66 31 c0 8e d8  |.".f.{.....f1...|
00000580  8e c0 8e e0 8e e8 8e d0  fb 66 c3 55 89 e5 57 56  |.........f.U..WV|
00000590  53 89 c6 89 cf 31 db 31  c0 85 d2 78 29 0f b6 0c  |S....1.1...x)...|
000005a0  16 84 c9 74 0e 0f b6 89  00 02 10 00 32 84 0b 00  |...t........2...|
000005b0  00 10 00 01 fb 81 fb fe  00 00 00 7e 06 81 eb ff  |...........~....|
000005c0  00 00 00 4a eb d3 5b 5e  5f 5d c3 55 89 e5 84 d2  |...J..[^_].U....|
000005d0  74 21 84 c0 74 1d 0f b6  c0 0f b6 88 00 02 10 00  |t!..t...........|
000005e0  0f b6 d2 0f b6 82 00 02  10 00 8a 84 01 00 00 10  |................|
000005f0  00 eb 02 31 c0 5d c3 55  89 e5 57 56 53 83 ec 24  |...1.].U..WVS..$|
00000600
```

gdt entries

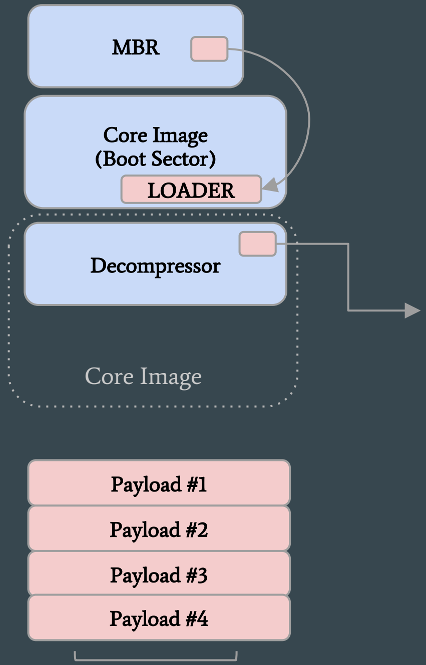
realidt

protidt

gdtdesc

In the current version of GRUB2, this values are always in the first sector of the decompressor

GRUB2 - Minimal changes to inject a payload loader



One nice place to put the payloads is the free sectors before the first partition

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

LOADER: small piece of code injected into the cave

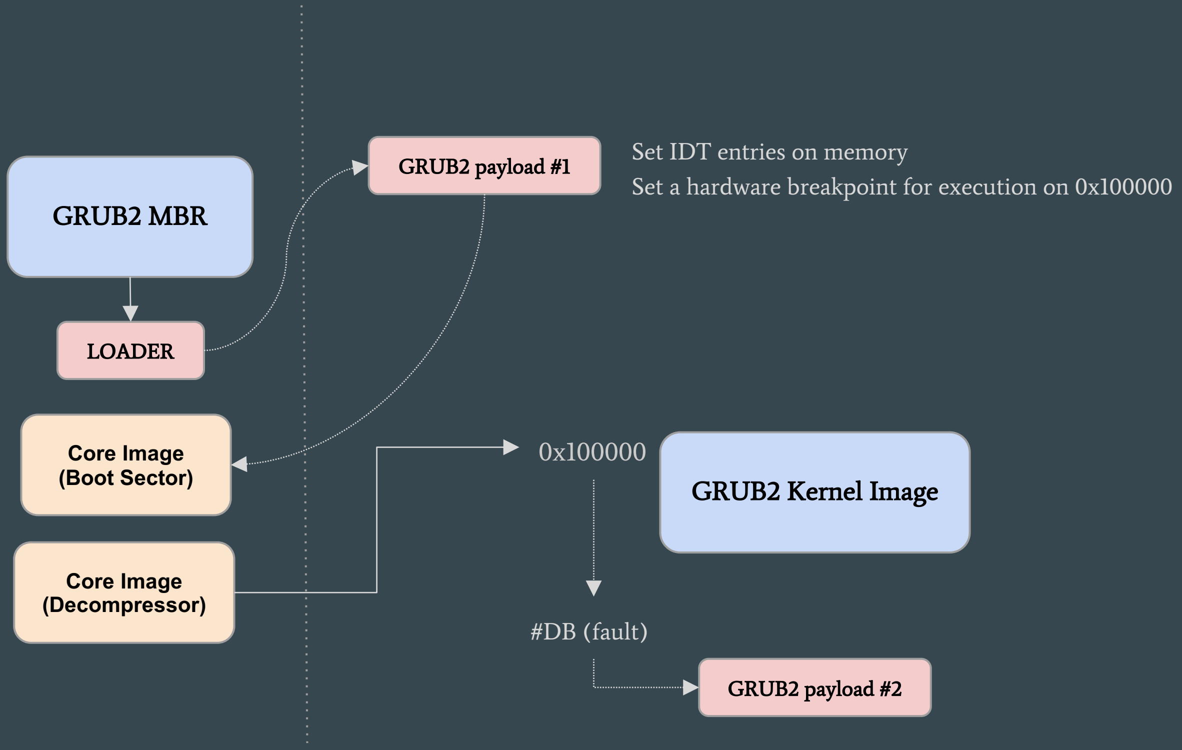
- reserve memory (e.g.: decreasing "Memory Size" at Bios Data Area)
- load all payloads on memory (int 13)
- execute the first

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

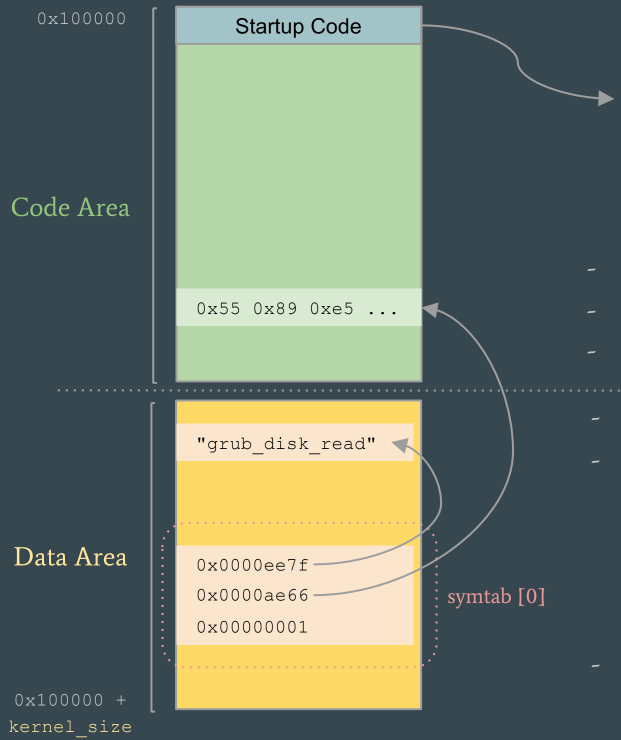
Payload #1: grub2
Payload #2: bzImage
Payload #3: vmlinux
Payload #4: userspace shellcode

GRUB2 - Minimal changes to inject a payload loader



GRUB2 - Uncompressed Kernel Image (overview)

GRUB2 Kernel Image



grub-core/kern/i386/pc/startup.S

```
0x100000:    mov   %ecx,0x41(%esi)
0x100006:    mov   %edi,0x45(%esi)
0x10000c:    mov   %eax,0x164(%esi)
0x100012:    mov   $0x6cec,%ecx
0x100017:    mov   $0x9000,%edi
0x10001c:    rep movsb %ds:(%esi),%es:(%edi)
0x10001e:    mov   $0x9025,%esi
0x100023:    jmp   *%esi
0x100025:
```

- 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:

```
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

GRUB2 - Commands

- 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-core/normal/dyncmd.c

```
read_command_list(...)
```

for all command in the file command.lst

grub-core/commands/extcmd.c

```
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);
```

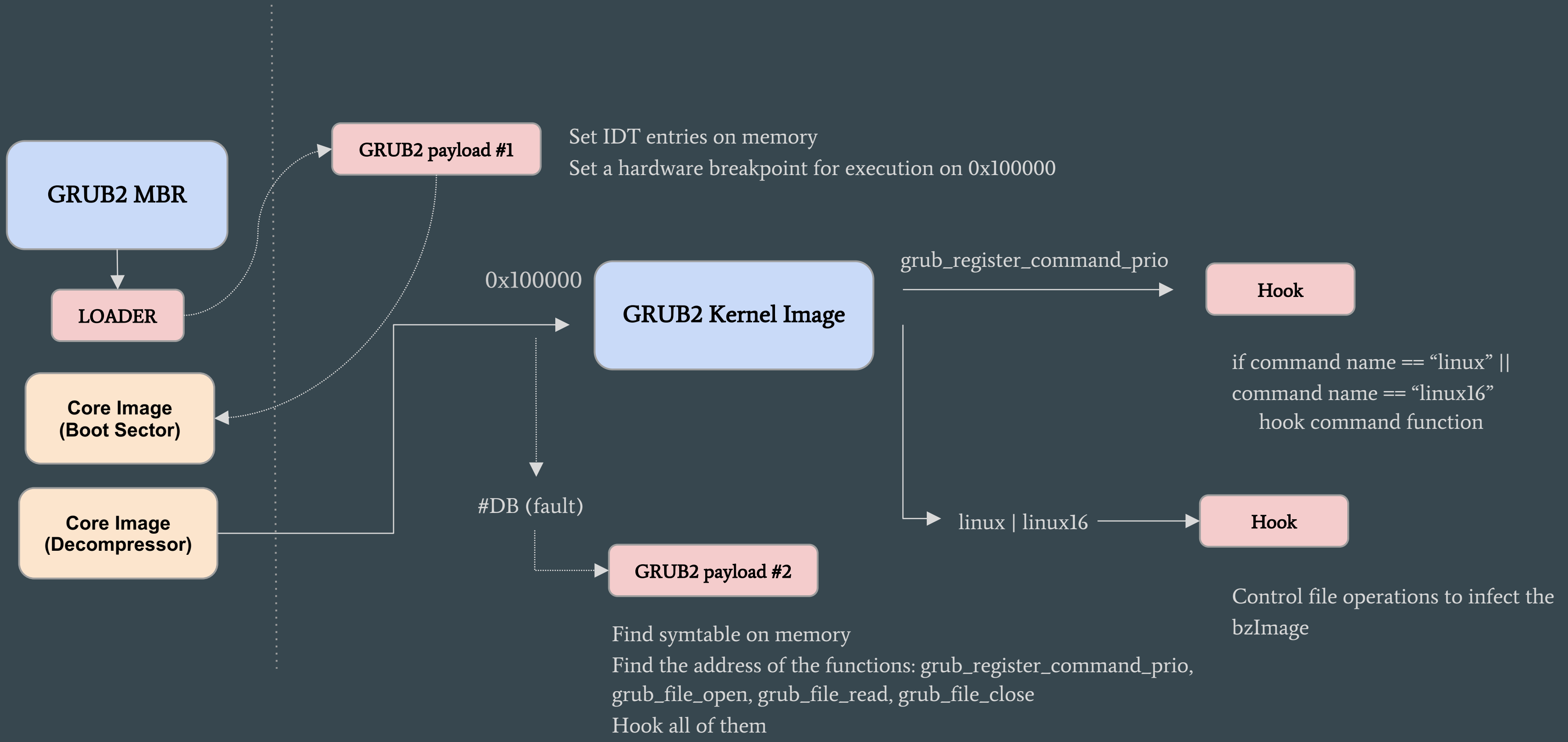
grub-core/kern/command.c

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

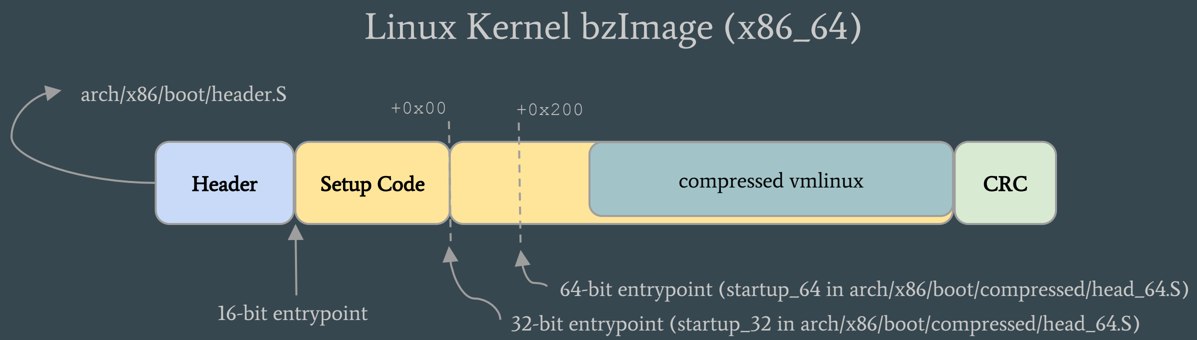
this ends by registering the command with a common dispatch function
the command function will be loaded and registered in the first use

- 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`"

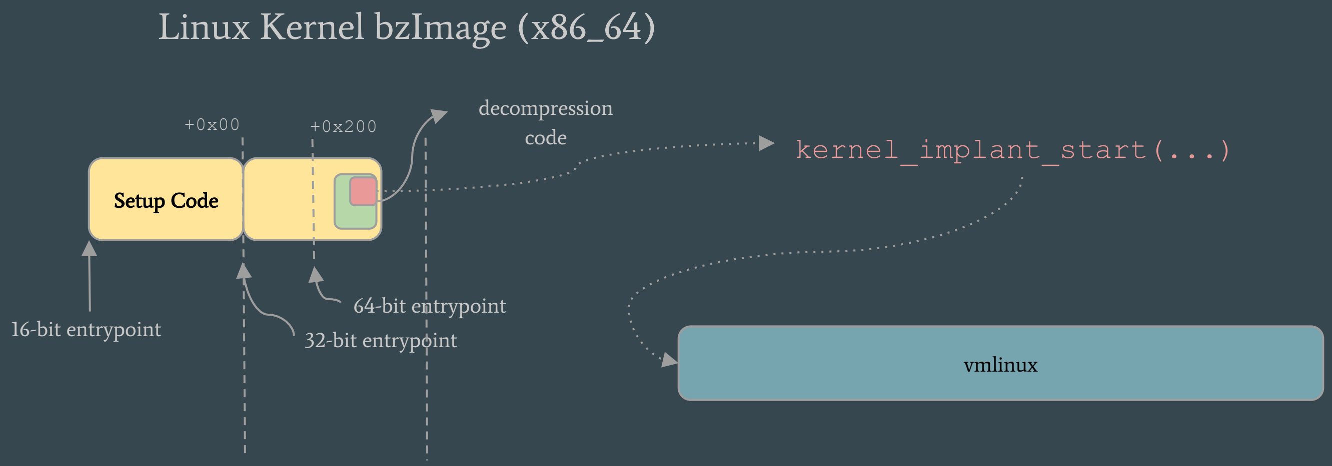
GRUB2 implant (Controlling Commands)



Linux Kernel bzImage (x86_64)

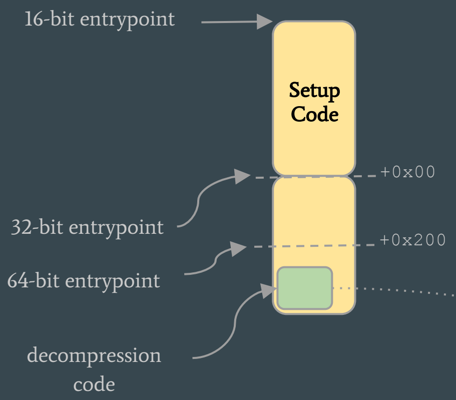


Linux Kernel bzImage (x86_64)



- 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

Linux Kernel bzImage (x86_64)

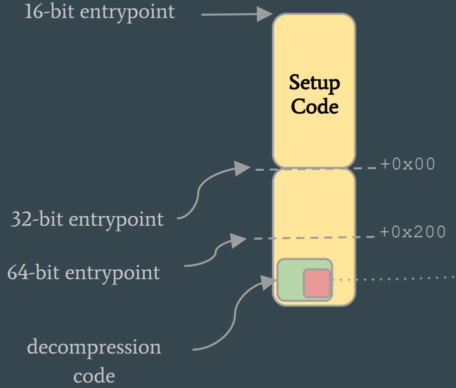


arch/x86/boot/compressed/head_64.S

```
/*
 * Jump to the relocated address.
 */

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

```
relocated:
...

/*
 * Do the extraction, and jump to the new kernel..
 */
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
```

Linux Kernel bzImage (x86_64)



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

```
/*
 * Jump to the relocated address.
 */

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

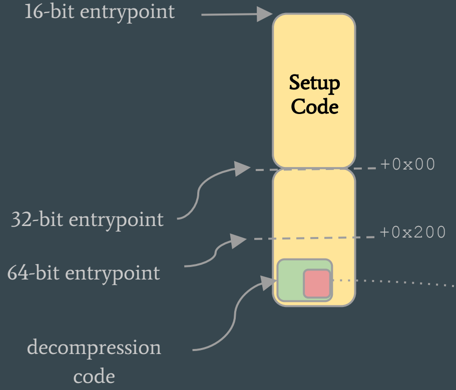
inline hook

```
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
```

`kernel_implant_start(...)`

Be careful: This code is slightly different for the kernels v3, v4, v5

Linux Kernel bzImage (x86_64)



arch/x86/boot/compressed/head_64.S

```
/*
 * Jump to the relocated address.
 */

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

inline hook

```
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
```

vmlinux_implant_start(...)

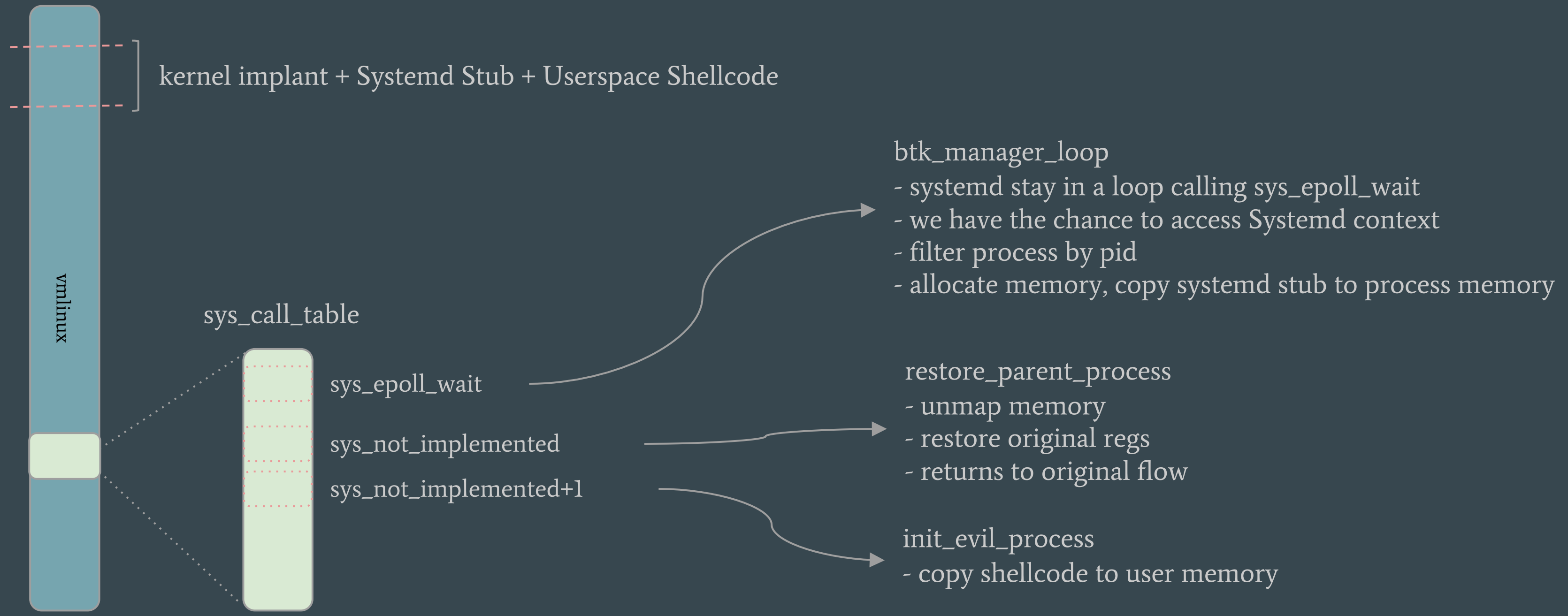
Linux >= 3 seems to have a indirect jump after the vmlinuz decompression

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



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 (different internal syscall calling convention)
  - inject a stub into process memory
  - set new return address on kernel stack

Demo

Questions