Undermining the Linux Kernel:
Malicious Code Injection via /dev/mem

Anthony Lineberry
anthony.lineberry@gmail.com

SCALE 7x 2009

Introduction

- Security Researcher for Flexilis
- Experience in reverse engineering/exploit development
- Active in Open Source kernel development (non-mainstream kernel)

Overview

- What is a rootkit?
- Why is protection difficult?
- Current protection mechanisms/bypasses
- Injection via /dev/mem
- Fun things to do once you're in
- Proposed solutions

Part I

Rootkit?

What is a rootkit?

- Way to maintain access (regain “root” after successful exploitation)
- Hide files, processes, etc
- Control activity
  - File I/O
  - Network
- Keystroke Logger

Types of rootkits

- User-Land (Ring 3)
  - Trojaned Binaries (oldest trick in the book)
    - Binary patching
    - Source code modification
  - Process Injection/Thread Injection
    - PTRACE_ATTACH, SIGNAL injection
  - Does not affect stability of system

Types of rootkits

- Kernel-Land (Ring 0)
  - Kernel Modules/Drivers
  - Hot Patching memory directly! (we'll get to that ;)

Part II

Why are rootkits hard to defend against?

flexilis

Why so hard?

- Most modern rootkits live in the kernel
- Kernel is God
  - Impractical to check *EVERYTHING* inside kernel
    - Speed hits
  - Built in security can be circumvented by more kernel code (if an attacker can get code in, game over)

Part III

Current Rootkit Defense

flexilis

Current Defense

- Checking Tables in kernel (sys_call_table, IDT, etc)
  - Compares tables against known good
  - Can be bypassed by creating duplicate table to use rather than modifying the main table

Current Defense

- Hashes/Code Signing
  - In kernel
    - Hash critical sections of code
    - Require signed kernel modules
  - In userland
    - Hashes of system binaries
      - Tripwire, etc
    - Signed binaries
    - File System Integrity

Current Defense

- Non-Modularity
  - Main suggested end all way to stop kernel space rootkits (obviously this is a fail)
  - /dev/kmem was previously used in a similar fashion, but read/write access has since been closed off in kernel mainline

Part IV

Code Injection via /dev/mem

What is /dev/mem?

- /dev/mem
  - Driver interface to physically addressable memory.
  - lseek() to offset in “file” = offset in physical mem
    - EG: Offset 0x100000 = Physical Address 0x100000
  - Reads/Writes like a regular character device
- Who needs this?
  - X Server (Video Memory & Control Registers)
  - DOSEmu

Hijacking the kernel

Kernel addressing is virtual. How do we translate to physical addresses?

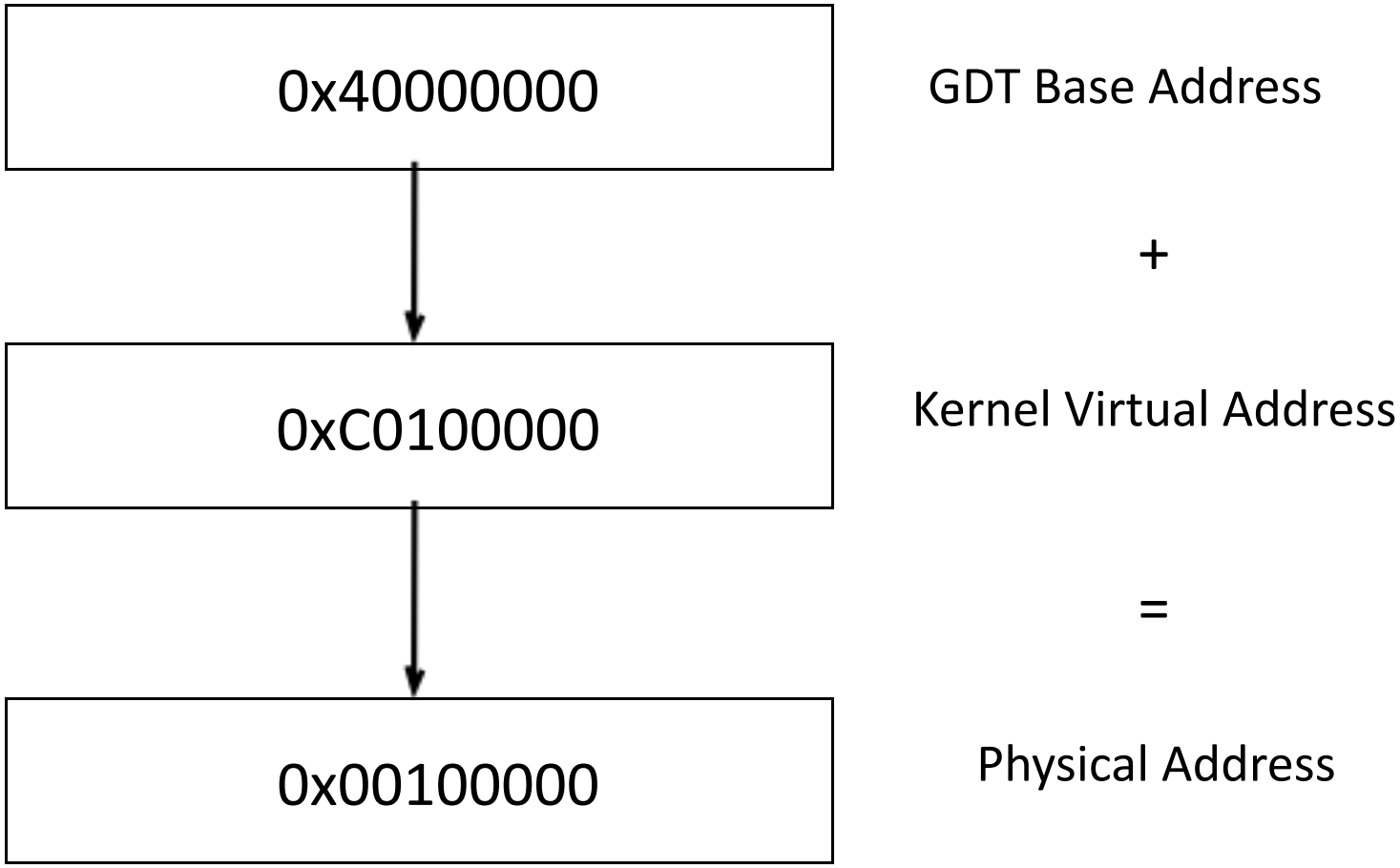
Address Translation

- Find a Page Table Directory (stored in cr3 register)
  - Finding one is harder done than said
  - We can cheat because of how the kernel is mapped

Address Translation

- Higher half GDT loading concept applies
- Bootloader trick to use Virtual Addresses along with GDT in unprotected mode to resolve physical addresses.
  - Kernel usually loaded at 0x100000 (1MB) in physical memory
  - Mapped to 0xC0100000 (3GB+1MB) Virtually

Address Translation



Address Translation

- Obviously over thinking that…
- No need to wrap around 32bit address, just subtract.

  0xC0100000 – 0xC0000000 = 0x100000

Hijacking the kernel

```
#define KERN_START 0xC0000000
int read_virt(unsigned long addr, void *buf, unsigned int len)
{
   if(addr < KERN_START)
       return -1;
   /* addr is now physical address */
   addr -= KERN_START;
   lseek(memfd, addr, SEEK_START);

   return read(memfd, buf, len);
}
```

Useful structures

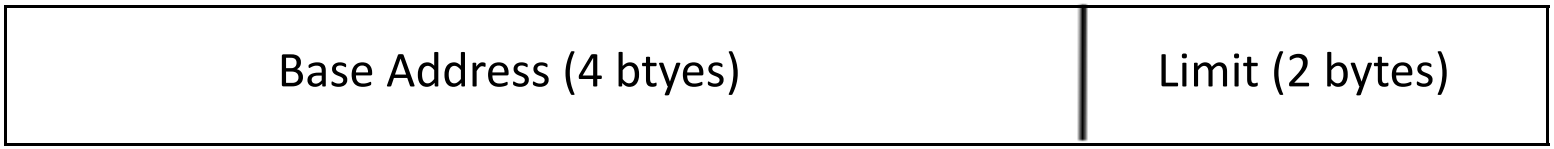
- Determine offset to important structures
  - IDT
  - sys_call_table
  - kmalloc()
- Where are they?

IDT

- Interrupt Descriptor Table (IDT)
  - Table of interrupt handlers/call gates
  - 0x80'th handler entry = Syscall Interrupt
- IDTR holds structure with address of IDT
  - Get/Set IDTR with LIDT/SIDT assembly instructions
  - Unlike LIDT instruction, SIDT is not protected and can be executed from user space to get IDT address.

IDTR

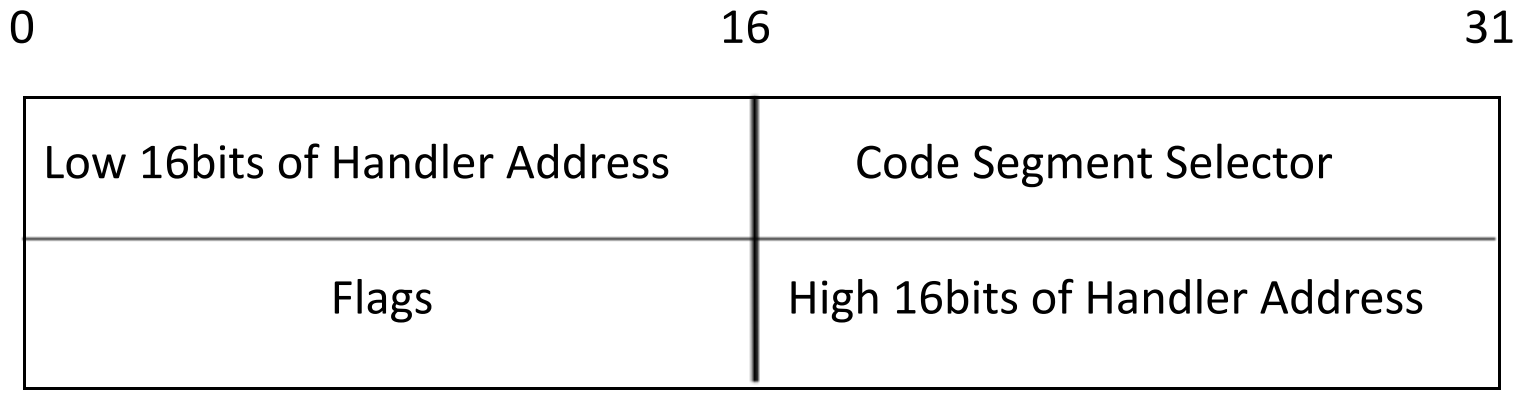
IDTR Structure



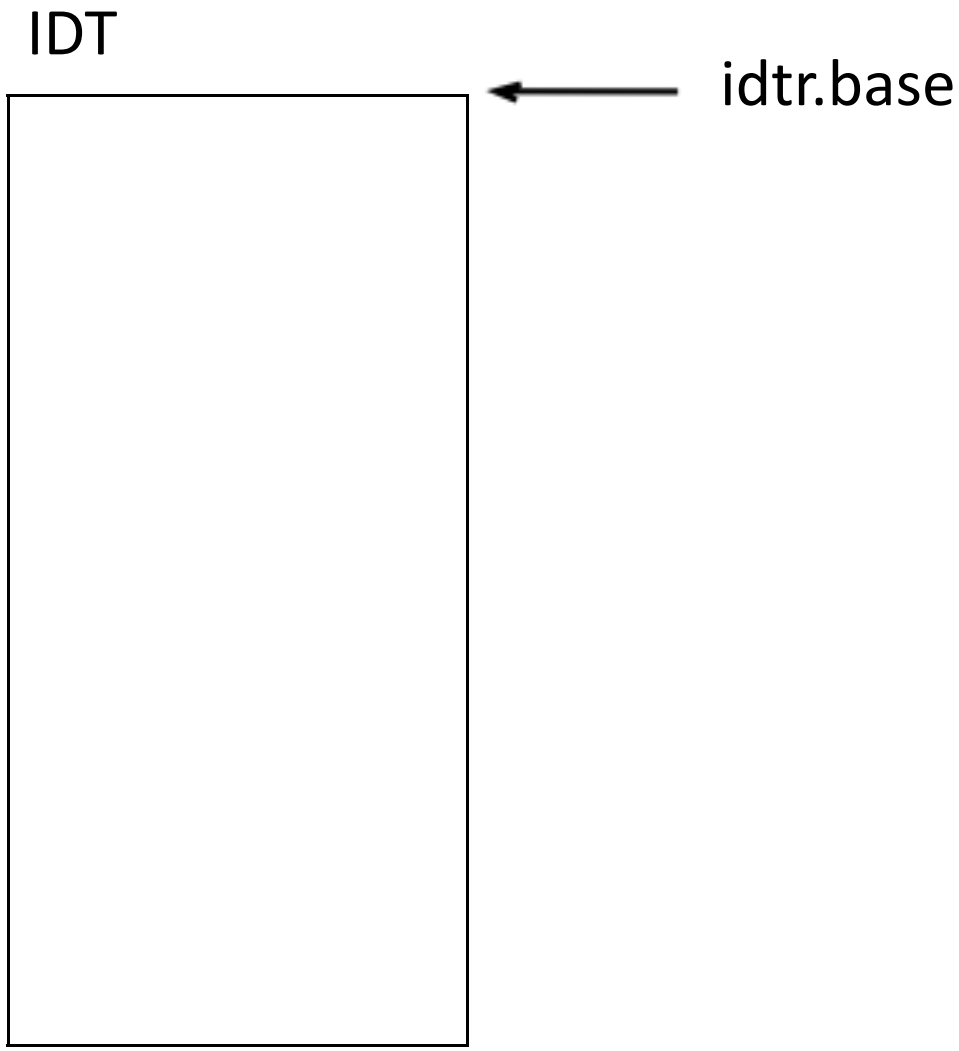

```
struct {
 uint32_t base;
 uint16_t limit;
} idtr;

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

IDT Entry

IDT Entry (8 bytes)

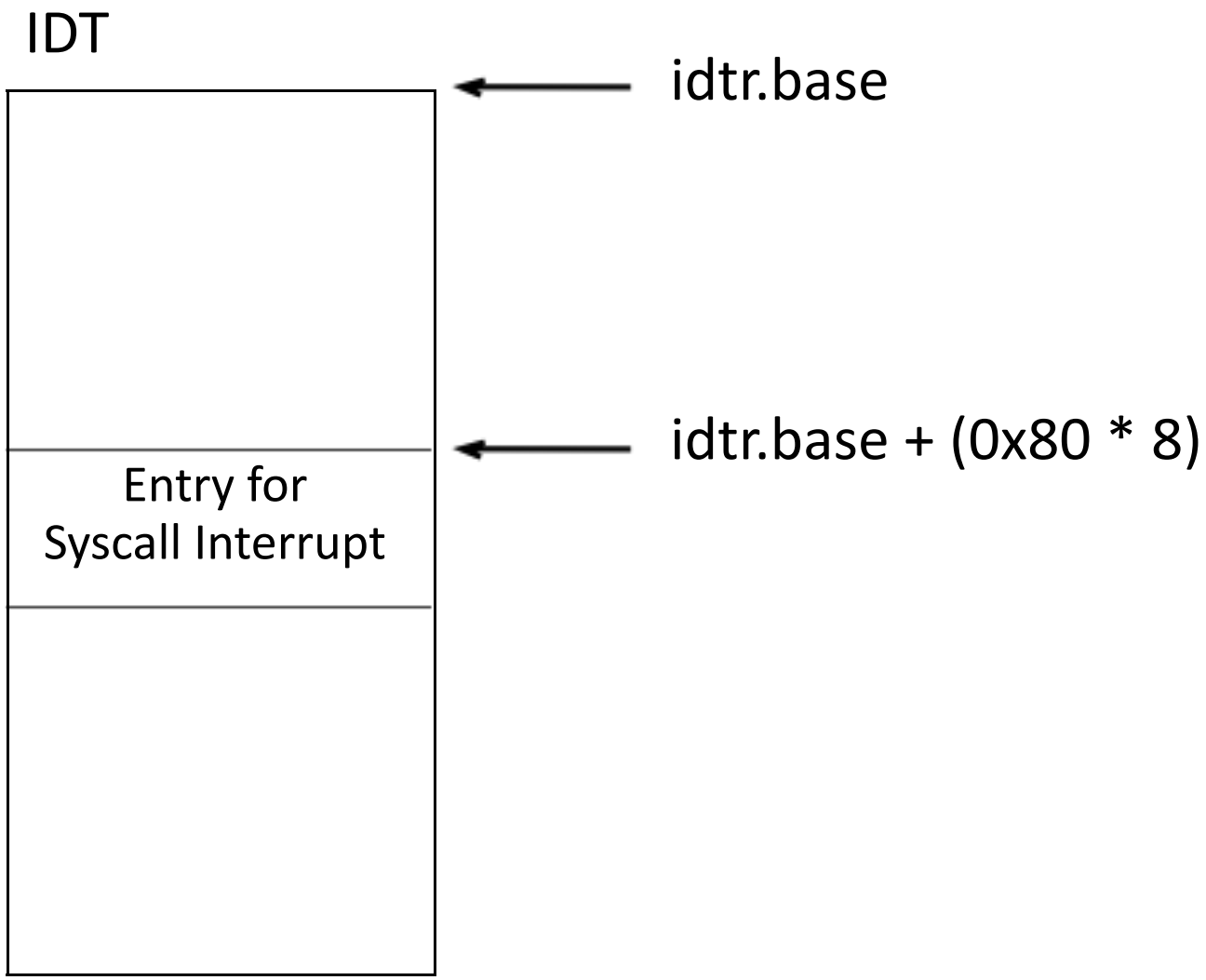


IDT

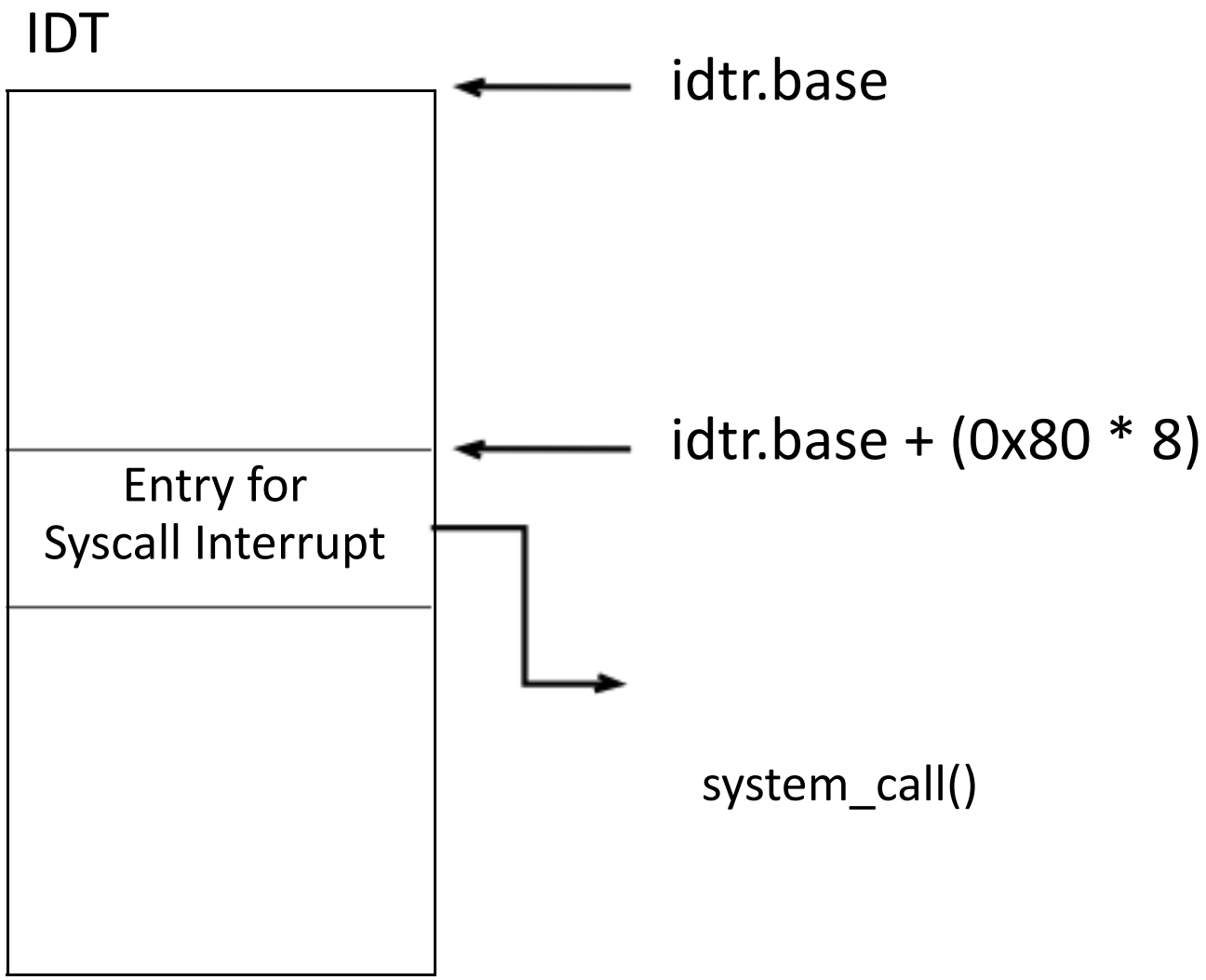


flexilis

IDT



IDT


IDT
idtr.base
idtr.base + (0x80 * 8)
Entry for
Syscall Interrupt
system_call()

System Calls

- system_call() – Main entry point for system calls
- sys_call_table – Array of function pointers
  - sys_read(), sys_write(), etc

System Calls

- Syscall Number stored in EAX register

```
call ptr 0x????????(eax,4)
```

  - 0x???????? Is the address of sys_call_table
  - Opcode for instruction:

    FF 14 85 ?? ?? ?? ??

  - Read in memory at system_call(), search for byte sequence “\xFF\x14\x85”. Next 4 following bytes are address of sys_call_table!

Hijacking the kernel

- Now we can:
  - Find IDT
  - Find system_call() handler function
  - Use simple heuristic to find address of sys_call_table
- What now?
  - Overwrite system calls with our own code!

Hijacking the kernel

- Where do we put our code?
  - Need to allocate space in the kernel
- We can locate __kmalloc() inside the kernel and use that.

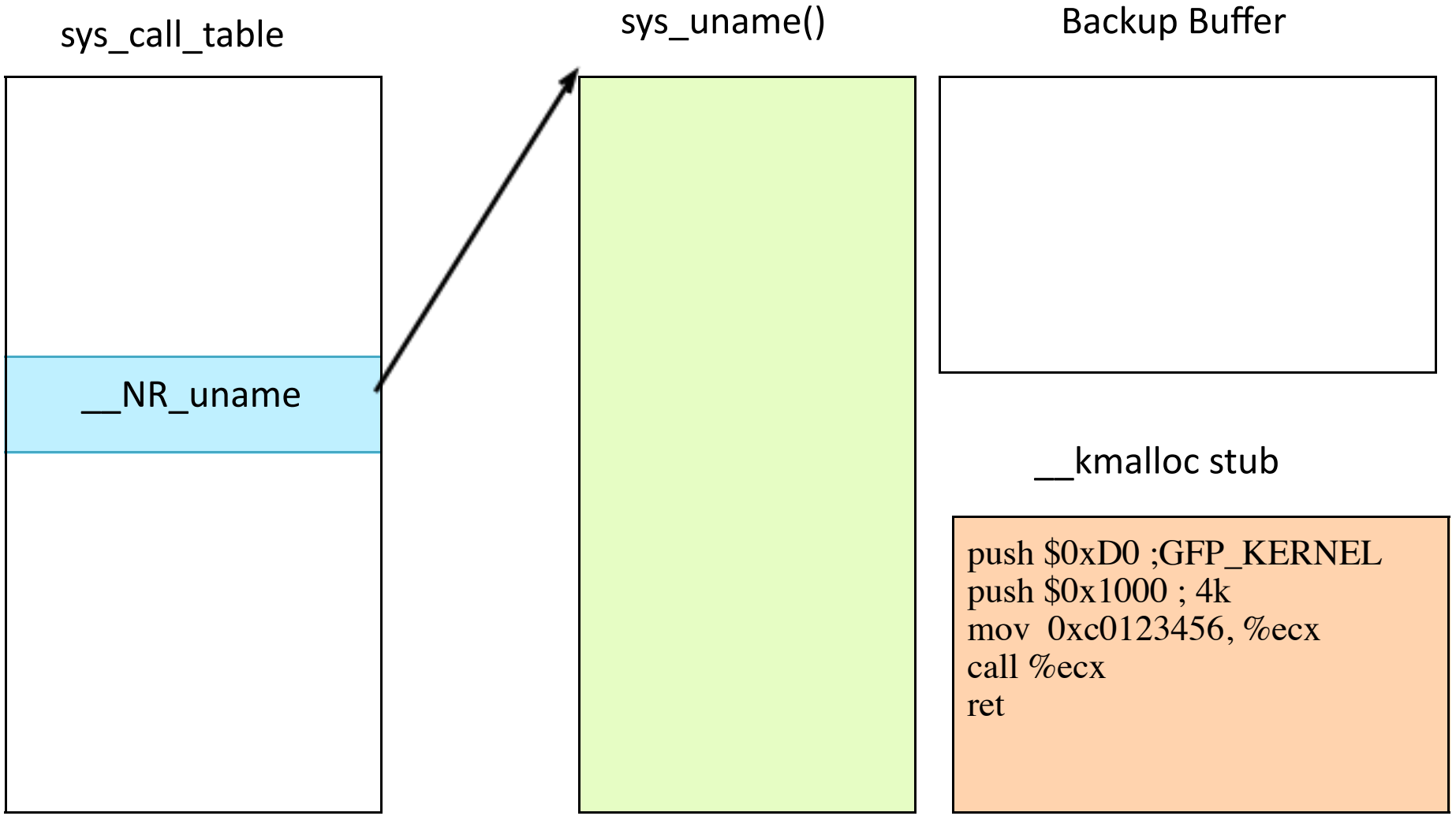
Hijacking the kernel

- Finding __kmalloc()
  - Use heuristics
    - push GFP_KERNEL
    - push SIZE
    - call __kmalloc
  - Find kernel symbol table
    - Search for “\0__kmalloc\0” in memory
    - Find reference to address of above sequence then subtract 4 bytes from location

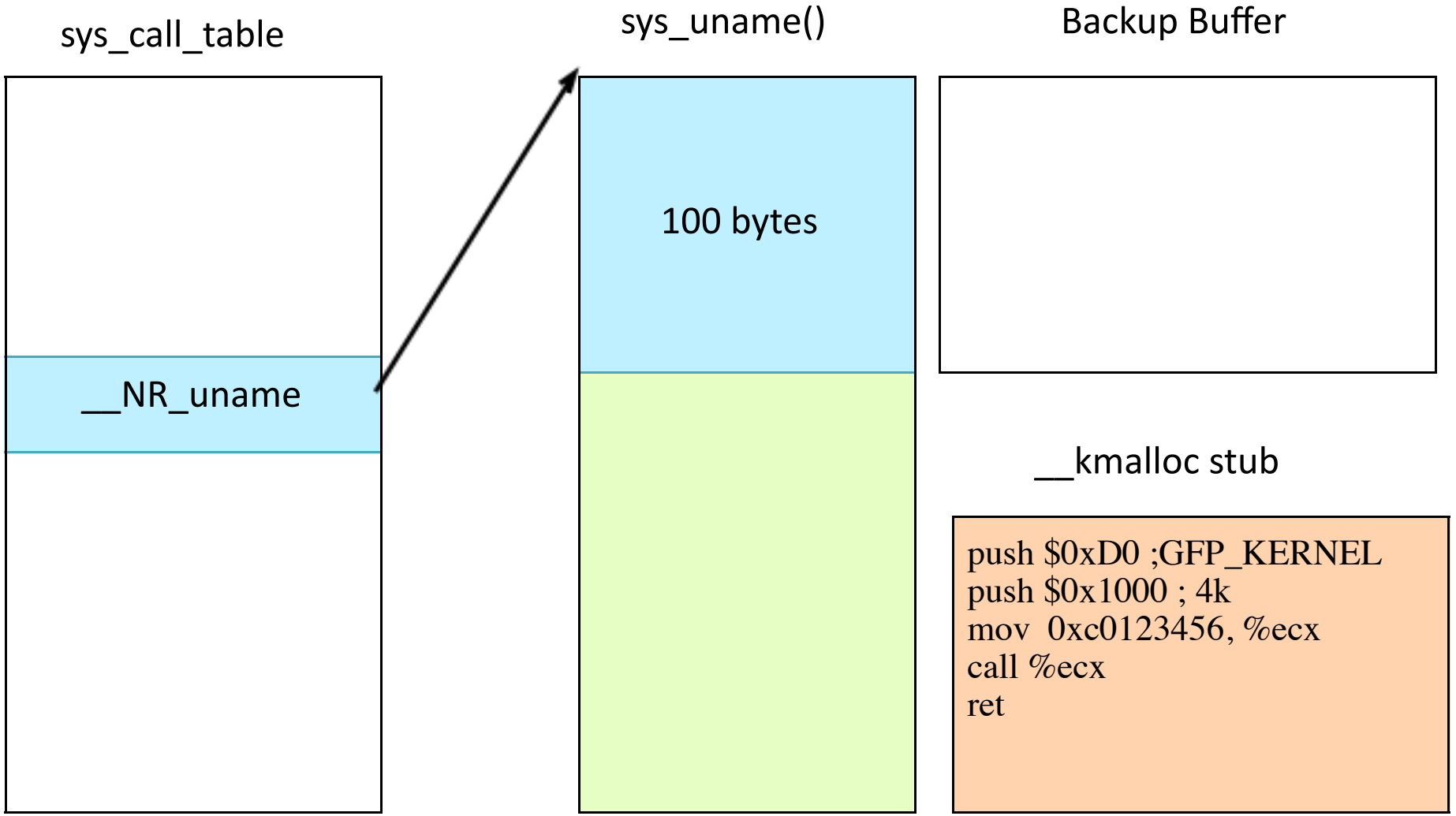
Hijacking the kernel

- How can we allocate kernel memory from userspace?
  - Overwrite a system call with code to call __kmalloc()

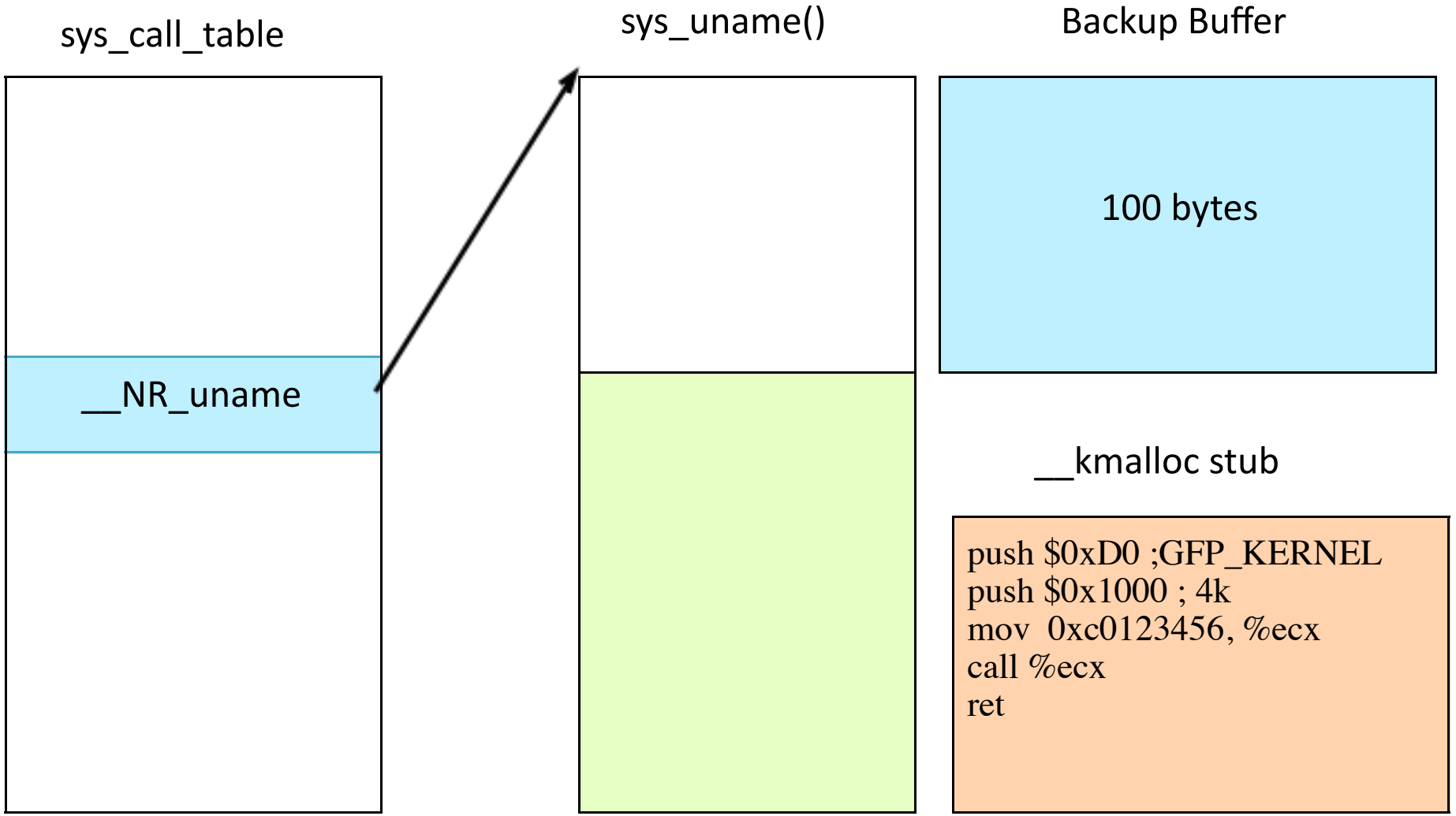
Function Clobbering



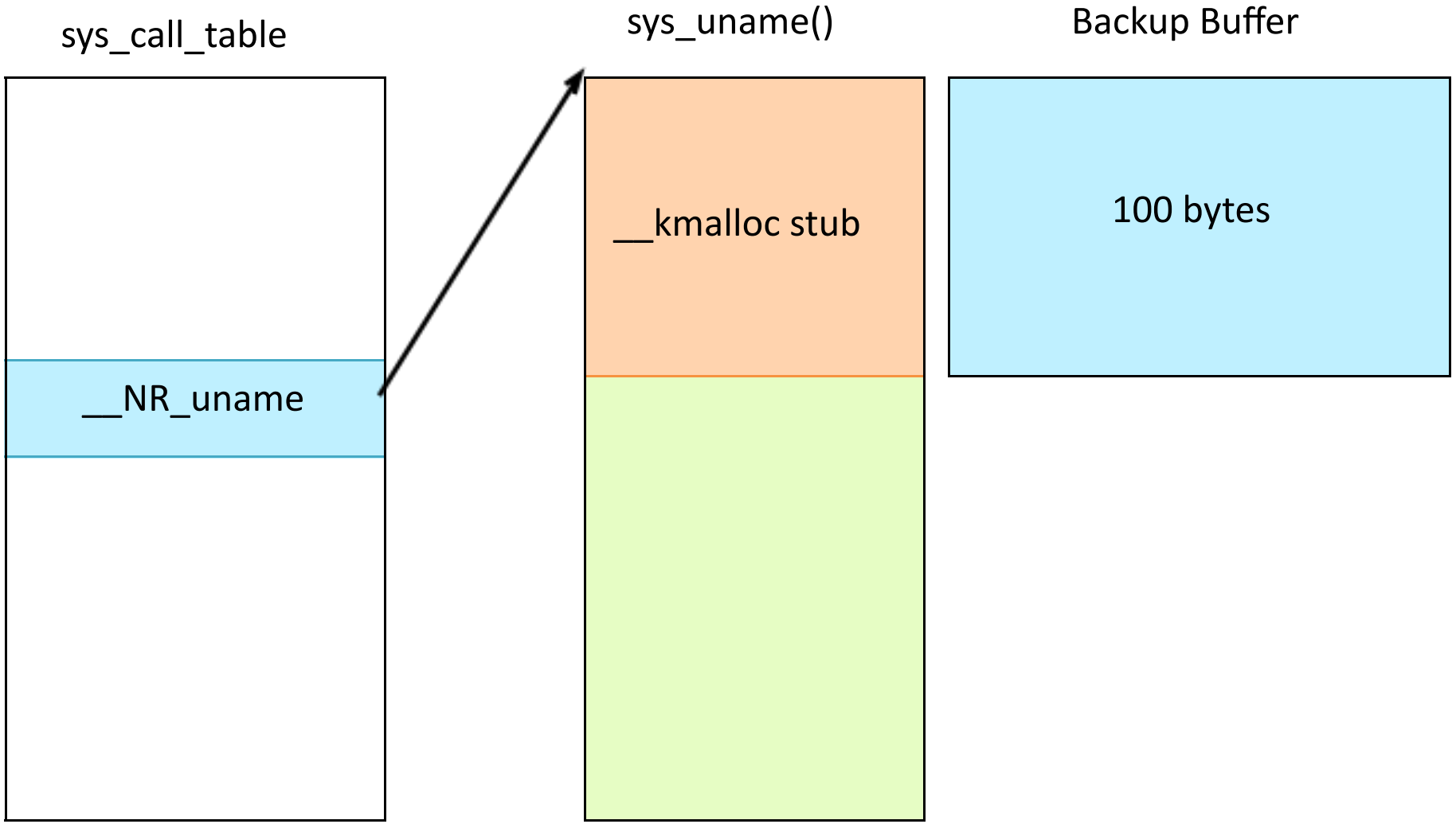
Function Clobbering



Function Clobbering



Function Clobbering



Hijacking the kernel

- Call sys_uname()

```
unsigned long kernel_buf;
__asm__("mov $122, %%eax \n"
        "int $0x80       \n"
        "mov %%eax, %0     " :
        "=r"(kernel_buf));
```

- Address of buffer allocated in kernel space returned by syscall in EAX register

Part V

Fun things to do inside the kernel

Hijacking the kernel

- Recap:
  - read/write anywhere in memory with /dev/mem
  - sys_call_table
  - Kernel allocation capabilities
  - Time to have fun!

Hijacking the kernel

- What can we do?
  - Use our kernel buffers we allocated to store raw executable code.
  - Overwrite function pointers in kernel with address of our allocated buffers
    - sys_call_table entries, page fault handler code
  - Setup code to use Debug registers to “hook” system call table

Hijacking the kernel

- What can we do with our injected code?
  - Anything most other rootkits can do.
    - Hide files, processes, etc
    - Control network activity
- Limitations
  - All injected code must usually be handwritten assembly
  - Some structures/functions can be difficult to locate in memory

Part V

Solutions/Mitigation

Solutions

- Why does a legitimate user process need access to read anything from above 16k in physical memory?
  - Modify mem driver to disallow lseeks past 16k
  - SELinux has created a patch to address this problem (RHEL and Fedora kernels are safe)
- Would like to see this limition applied to main kernel source

Questions?