Linux Kernel Exploitation

Modern Binary Exploitation
CSCI 4968 - Spring 2015
Patrick Biernat
First order of Business
First order of Business

FILES IN A FOLDER CALLED

TEMPORARY

ARE TEMPORARY
You probably feel like this

TELLS YOU TO PUT YOUR STUFF IN /TMP

MBE TA'S

DELETES /TMP
Lecture Overview

1. An Introduction to the Kernel
2. General Exploitation Strategy
3. Kernel-Space Protections
4. Example
5. Conclusion
Jumping out of the Matrix

So far, we have been exploiting binaries running in userspace.
Jumping out of the Matrix

So far, we have been exploiting binaries running in userspace.

Userspace is an abstraction that runs “on top” of the kernel.
Jumping out of the Matrix

So far, we have been exploiting binaries running in **userspace**.

Userspace is an *abstraction* that runs “on top” of the kernel.

1. Filesystem I/O
2. Privilege Levels (Per User/Per Group)
3. Syscalls
4. Processes
5. And so much more
Jumping out of the Matrix

So far, we have been exploiting binaries running in **userspace**.

Userspace is an **abstraction** that runs “on top” of the **kernel**.

1. Filesystem I/O
2. Privilege Levels (Per User/Per Group)
3. Syscalls
4. Processes
5. And so much more

These are all “services” provided by the Kernel
What’s a Kernel?

Low Level code with two major responsibilities:

1. Interact with and control hardware components
2. Provide an Environment in which Applications can run
What’s a Kernel?

Low Level code with two major responsibilities:

1. Interact with and control hardware components
2. Provide an Environment in which Applications can run

The Kernel is the core of the operating system
What’s a Kernel? - Ring Model

Hardware Enforced Model

0: Privileged, Kernelspace

3: Restricted, Userspace
What’s a Kernel? - Ring Model

Hardware Enforced Model

0: Privileged, Kernelspace

3: Restricted, Userspace

Ring 1 and Ring 2 are not utilized by most popular/modern Operating Systems (Linux / Windows / OSX)
What’s a Kernel? - Ring Model

We’ve Been Here

Ring 3

Ring 2

Ring 1

Ring 0

Kernel

Device Drivers

Device Drivers

Applications
What’s a Kernel? - Ring Model

We’ve Been Here

We’re Going Here
Obligatory Matrix Analogy

“The Matrix is the world that has been pulled over your eyes to blind you from the truth.” - Morpheus
Obligatory Matrix Analogy

“The Matrix is the world that has been pulled over your eyes to blind you from the truth.” - Morpheus

The kernel provides the “matrix” your programs run in
Obligatory Matrix Analogy

“The Matrix is the world that has been pulled over your eyes to blind you from the truth.” - Morpheus

The kernel provides the “matrix” your programs run in

Break out of the Matrix, and you pwn the entire system
“Jailbreaking” or “rooting” devices often depends on finding and leveraging Kernel bugs.
“Jailbreaking” or “rooting” devices often depends on finding and leveraging Kernel bugs

Remember JailbreakMe?
“Jailbreaking” or “rooting” devices often depends on finding and leveraging Kernel bugs

Remember JailbreakMe?

It used a remote code execution primitive inside Safari to trigger a kernel-level exploit to bypass Apple’s code-signing protection
Kernel Basics

Your Kernel is:

Managing your Processes
Managing your Memory
Coordinating your Hardware
Kernel Basics

Your Kernel is:

Managing your Processes
Managing your Memory
Coordinating your Hardware

A crash oftentimes means a reboot!
Kernel Basics

Your Kernel is:

Managing your Processes
Managing your Memory
Coordinating your Hardware

A crash oftentimes means a reboot!

In general, we want to spend as little time there as possible.
Basic Exploitation Strategy

The Kernel is typically the most powerful place we can find bugs
Basic Exploitation Strategy

The Kernel is typically **the most powerful** place we can find bugs.

But, how do we go from “vulnerability” to “privileged execution” without bringing down the rest of the system?
Basic Exploitation Strategy

The Big Picture

1. Find **vulnerability** in kernel code
2. Manipulate it to gain **code execution**
3. Elevate our process’s **privilege level**
4. **Survive** the “trip” back to userland
5. Enjoy our **root** privileges
Basic Exploitation Strategy

The Vulnerabilities

You already know how to find these!
You already know how to find these!

Kernel vulnerabilities are almost exactly the same as userland vulnerabilities.
Basic Exploitation Strategy
The Vulnerabilities

You already know how to find these!

Kernel vulnerabilities are almost exactly the same as userland vulnerabilities.

1. Buffer Overflows
2. Signedness issues
3. Partial Overwrites
4. Use-After-Free

By now, finding these should be a familiar process
Basic Exploitation Strategy
The Vulnerabilities

The most common place to find vulnerabilities is inside of Loadable Kernel Modules (LKMs).
The most common place to find vulnerabilities is inside of Loadable Kernel Modules (LKM).

LKM are like executables that run in Kernel Space. A few common uses are listed below:

> Device Drivers
> Filesystem Drivers
> Networking Drivers
> Executable Interpreters
> Kernel Extensions
> (rootkits :P)
Basic Exploitation Strategy
The Vulnerabilities

LKMs are just binary blobs like your familiar ELF’s, EXE’s and MACH-O’s. (On Linux, they even use the ELF format)
Basic Exploitation Strategy
The Vulnerabilities

LKMs are just binary blobs like your familiar ELF’s, EXE’s, and MACH-O’s. (On Linux, they even use the ELF format)

You can drop them into IDA and reverse-engineer them like you’re used to already.
Basic Exploitation Strategy

The Vulnerabilities

There’s a few useful commands that deal with LKMs on Linux.
Basic Exploitation Strategy

The Vulnerabilities

There’s a few useful commands that deal with LKMs on Linux.

**insmod** ---> Insert a module into the running kernel

**rmmod** ---> Remove a module from the running kernel

**lsmod** ---> List currently loaded modules
Basic Exploitation Strategy
The Vulnerabilities

There's a few useful commands that deal with LKMs on Linux.

insmod ---> Insert a module into the running kernel
rmmod ---> Remove a module from the running kernel
lsmod ---> List currently loaded modules

A general familiarity with these is helpful
Basic Exploitation Strategy
Gaining Code Execution

You already know how to do this too!
Basic Exploitation Strategy
Gaining Code Execution

You already know how to do this too!

The same basic exploitation techniques apply to Kernelspace
(After all, it’s just x86 code!)
Basic Exploitation Strategy
Gaining Code Execution

You already know how to do this too!

The same basic exploitation techniques apply to Kernelspace (After all, it’s just x86 code!)

Shellcoding, ROP, Pointer Overwrites, Type Confusion, etc can all be used to execute code in Kernel Land.
Basic Exploitation Strategy
Gaining Code Execution

You already know how to do this too!

The same basic exploitation techniques apply to Kernelspace (After all, it’s just x86 code!)

Shellcoding, ROP, Pointer Overwrites, Type Confusion, etc can all be used to execute code in Kernel Land.

Typically, you won’t have to deal with ASLR!
Basic Exploitation Strategy
Gaining Code Execution

Common Library calls are sometimes different, so there is a slight learning curve involved.
Basic Exploitation Strategy
Gaining Code Execution

Common Library calls are sometimes *different*, so there is a slight learning curve involved.

`printf()`  -->  `printk()`
`memcpy()`  -->  `copy_from_user()` / `copy_to_user()`
`malloc()`  -->  `kmalloc()`
Basic Exploitation Strategy
Gaining Code Execution

Common Library calls are sometimes *different*, so there is a slight learning curve involved.

printf()  ---\>  printk()
memcpy()  ---\>  copy_from_user() / copy_to_user()
malloc()  ---\>  kmalloc()

Typically, whatever you want to know is a quick google-search or man page away.
Basic Exploitation Strategy
Gaining Code Execution

Debugging kernel code can be difficult
Basic Exploitation Strategy
Gaining Code Execution

Debugging kernel code can be difficult

We can’t just run the kernel in gdb
Basic Exploitation Strategy
Gaining Code Execution

Debugging kernel code can be difficult

We can’t just run the kernel in `gdb`

You will often have to rely on `stack dumps`, `error messages`, and other “black box” techniques to infer what’s going on inside the kernel.
Basic Exploitation Strategy
Gaining Code Execution

This is an example of what you might see if you get a crash in the kernel.
Basic Exploitation Strategy
Gaining Code Execution

This is an example of what you might see if you get a crash in the kernel.

Call Trace
Register Dump
Stack Dump
Basic Exploitation Strategy
Gaining Code Execution

This is an example of what you might see if you get a crash in the kernel.

Call Trace
Register Dump
Stack Dump

You might be able to see this with `dmesg` if the crash is not fatal.
Basic Exploitation Strategy

Elevate Privileges

Remember: The Kernel manages running processes
Basic Exploitation Strategy

Elevate Privileges

Remember: The Kernel manages running processes

Therefore: The Kernel keeps track of permissions
Basic Exploitation Strategy

Elevate Privileges

Remember: The Kernel manages running processes

Therefore: The Kernel keeps track of permissions

```c
struct task_struct {
    ...
    /* process credentials */
    const struct cred ___rcu *real_cred;
    const struct cred ___rcu *cred;
    char comm[TASK_COMM_LEN];
    ...
};
```

linux/include/linux/sched.h
Basic Exploitation Strategy

Elevate Privileges

Conveniently, the Linux Kernel has a wrapper for updating process credentials!
Basic Exploitation Strategy
Elevate Privileges

Conveniently, the Linux Kernel has a wrapper for updating process credentials!

```c
int commit_creds(struct cred *new) {
    ...
}
```
Basic Exploitation Strategy

Elevate Privileges

Conveniently, the Linux Kernel has a wrapper for updating process credentials!

```c
int commit_creds(struct cred *new) {
    ...
}
```

We just need to create a valid cred struct!
Basic Exploitation Strategy
Elevate Privileges

The kernel is helpful again!

```
struct cred *prepare_kernel_cred(struct task_struct *daemon) {
    ...
}
```
Basic Exploitation Strategy
Elevate Privileges

The kernel is helpful again!

```c
struct cred *prepare_kernel_cred(struct task_struct *daemon) {
    ...
}
```

"If @daemon is supplied, then the security data will be derived from that; otherwise they'll be set to 0 and no groups, full capabilities and no keys."

- source/kernel/cred.c
Basic Exploitation Strategy
Elevate Privileges

Great! Now we can map out what we need to do.
Basic Exploitation Strategy

Elevate Privileges

Great! Now we can map out what we need to do.

1. Create a "root" "struct creds" by calling prepare_kernel_creds( NULL );
2. Call commit_creds(root cred *);
Basic Exploitation Strategy
Elevate Privileges

Great! Now we can map out what we need to do.

1. Create a “root” “struct creds” by calling `prepare_kernel_cred(NULL);`
2. Call `commit_creds(root cred *);`
3. Enjoy our new root privileges!
Basic Exploitation Strategy
Returning To UserSpace

Why bother returning to Userspace?
Basic Exploitation Strategy
Returning To UserSpace

Why bother returning to Userspace?

Most useful things we want to do are much easier from userland.
Basic Exploitation Strategy
Returning To UserSpace

Why bother returning to UserSpace?

Most useful things we want to do are *much* easier from userland.

In KernelSpace, there’s no easy way to:

> Modify the filesystem
> Create a new process
> Create network connections
Basic Exploitation Strategy

Returning To UserSpace

How does the kernel do it?
Basic Exploitation Strategy

Returning To UserSpace

How does the kernel do it?

push $SS_USER_VALUE
push $USERLAND_STACK
push $USERLAND_EFLAGS
push $CS_USER_VALUE
push $USERLAND_FUNCTION_ADDRESS

swapgs
iretq
Basic Exploitation Strategy
Returning To UserSpace

How does the kernel do it?

push $SS_USER_VALUE
push $USERLAND_STACK
push $USERLAND_EFLAGS
push $CS_USER_VALUE
push $USERLAND_FUNCTION_ADDRESS
swapgs
iretq

This *will usually* get you out of “Kernel Mode” safely.
Basic Exploitation Strategy
Returning To UserSpace

For exploitation, the easiest strategy is highjacking execution, and letting the kernel return by itself.
Basic Exploitation Strategy
Returning To UserSpace

For exploitation, the easiest strategy is **highjacking** execution, and letting the kernel return by itself.

> Function Pointer Overwrites
> Syscall Table Highjacking
> Use-After-Free
For exploitation, the easiest strategy is highjacking execution, and letting the kernel return by itself.

- Function Pointer Overwrites
- Syscall Table Highjacking
- Use-After-Free

You need to be very careful about destroying Kernel state.

A segfault probably means a reboot!
Basic Exploitation Strategy

Enjoying our Root Privs

If we make it back to userland, our process should be running with root privileges.
Basic Exploitation Strategy

Enjoying our Root Privs

If we make it back to userland, our process should be running with root privileges.

We can do whatever we want!
Kernel Space Protections

By now, you’re familiar with the alphabet soup of exploit mitigations
Kernel Space Protections

By now, you’re familiar with the alphabet soup of exploit mitigations:

- DEP
- ASLR
- PIE
- Canaries
- RELRO
- etc...

```assembly
push   edi
call   sub_319623
test   eax, eax
jz     short loc_31306D
cmp    [ebp+arg_0], ebx
jz     short loc_31306D
jnz    short loc_31306D
mov    eax, edi
push   esi
call   sub_314b0c
test   eax, eax
jz     short loc_31306D
push   esi
lea    eax, [ebp+arg_0]
push   eax
mov    esi, 1D0h
push   esi
push   [ebp+arg_4]
push   edi
call   sub_314623
test   eax, eax
jz     short loc_31306D
cmp    [ebp+arg_0], esi
jnz    short loc_31308F

loc_313066:           ; CODE XREF: sub_312FD0       ; sub_312FD0+51
push    0Eh
call    sub_314113

loc_31306D:           ; CODE XREF: sub_312FD0       ; sub_312FD0+49
call    sub_3140F3
test    eax, eax
jnz     short loc_31307D
call    sub_3140F3
jmp     short loc_31308C

loc_31307D:           ; CODE XREF: sub_312FD0
and     eax, 0FFFFFFh
or      eax, 80070000h

loc_31308C:           ; CODE XREF: sub_312FD0
mov     [ebp+var_4], eax
```
Kernel Space Protections

By now, you’re familiar with the alphabet soup of exploit mitigations:

- **DEP Green**: Present in Kernel Space
- **ASLR Yellow**: Present, with caveats
- **PIE Red**: Not directly applicable
- **Canaries**
- **RELRO**
- etc...

```
push    edi
call    sub_314623
test    eax, eax
jz      short loc_31306D
cmp     [ebp+arg_0], ebx
jz      short loc_31306D
cmp     eax, [ebp+var_48]
je      short loc_313066
sub     eax, [ebp+var_84]
```

```
push    esi
push    eax
push    edx
call    sub_31460c
test    eax, eax
jz      short loc_31306D
push    esi
lea     eax, [ebp+arg_0]
push    eax
mov     esi, 1D0h
call    sub_31460B
push    edi
call    sub_31411B
```

```
loc_313066:  ; CODE XREF: sub_312FD0+51
            ; sub_312FD0+55
    push    0Dh
    call    sub_31411B
loc_31306D:  ; CODE XREF: sub_312FD0+57
            ; sub_312FD0+49
    call    sub_3140F3
    test    eax, eax
    jmp     short loc_31307D
    call    sub_3140F3
    jmp     short loc_31308C
```

```
loc_31307D:  ; CODE XREF: sub_312FD0+57
    call    sub_3140F3
    add     eax, 0FEDh
    or      eax, 000070000h
loc_31308C:  ; CODE XREF: sub_312FD0+57
    mov     [ebp+var_4], eax
```
Kernel Space Protections

By now, you’re familiar with the alphabet soup of exploit mitigations:

- **DEP** Green: Present in Kernel Space
- **ASLR** Yellow: Present, with caveats
- **PIE** Red: Not directly applicable
- **Canaries**
- **RELRO**
- etc...

There’s a whole new alphabet soup for Kernel Mitigations!
Kernel Space Protections

Some new words in our soup

**MMAP_MIN_ADDR**
**KALLSYMS**
**RANDSTACK**
**STACKLEAK**
**SMEP / SMAP**
Kernel Space Protections

Some new words in our soup  (There’s plenty more...)

- MMMAP_MIN_ADDR
- KALLSYMS
- RANDSTACK
- STACKLEAK
- SMEP / SMAP

Most of these will be off for the labs!
Kernel Space Protections

mmap_min_addr

This makes exploiting NULL pointer dereferences harder.
This makes exploiting NULL pointer dereferences harder.
Kernel Space Protections

This makes exploiting NULL pointer dereferences harder.

Program does mmap(0, ...)
Kernel Space Protections

mmap_min_addr

This makes exploiting NULL pointer dereferences harder.

Program does mmap(0,....)

Program writes malicious Code
Kernel Space Protections

This makes exploiting NULL pointer dereferences harder.

Program does mmap(0,...)

Program writes malicious Code

Program triggers Kernel Bug
This makes exploiting NULL pointer dereferences harder.

Program does `mmap(0,...)`

Program writes malicious Code

Program triggers Kernel Bug

Kernel starts executing malicious Code
Kernel Space Protections

`mmap_min_addr`

This makes exploiting NULL pointer dereferences harder.

`mmap_min_addr` disallows programs from allocating low memory.

Makes it much more difficult to exploit a simple NULL pointer dereference in the kernel.
Kernel Space Protections

Kallsyms

/proc/kallsyms gives the address of all symbols in the kernel.

We need this information to write reliable exploits without an info-leak!
Kernel Space Protections
kallsyms

/proc/kallsyms gives the address of all symbols in the kernel.

We need this information to write reliable exploits without an info-leak!

$: cat /proc/kallsyms | grep commit_creds

ffffffff810908c0 T commit_creds
ffffffff81b01390 R __ksymtab_commit_creds
ffffffff81b1cf38 r __kcrctab_commit_creds
ffffffff81b2c33b r __kstrtab_commit_creds
Kernel Space Protections

kallsyms

kallsyms used to be world-readable.

Now, it returns 0’s for unprivileged users

$: cat /proc/kallsyms | grep commit_creds

0000000000000000 T commit_creds
0000000000000000 R __ksymtab_commit_creds
0000000000000000 r __kcrctab_commit_creds
0000000000000000 r __kstrtab_commit_creds

Can still be a useful source of information on older systems
**Kernel Space Protections**

**SMEP / SMAP**

**SMEP: Supervisor Mode Execution Protection**

Introduced in Intel IvyBridge

**SMAP: Supervisor Mode Access Protection**

Introduced in Intel Haswell
Kernel Space Protections
SMEP / SMAP

Common Exploitation Technique: Supply your own “get root” code.
Kernel Space Protections
SMEP / SMAP

Common Exploitation Technique: Supply your own “get root” code.

```c
void get_root() {
    commit_creds(prepare_kernel_cred(0));
}

int main(int argc, char * argv) {
    ...
    trigger_fp_overwrite(&get_root);
    ...
    //trigger fp use
    trigger_vuln_fp();
    // Kernel Executes get_root
    ...
    // Now we have root
    system("/bin/sh");
}```
Kernel Space Protections
SMEP / SMAP

Common Exploitation Technique: Supply your own “get root” code.

```c
void get_r00t() {
    commit_creds(prepare_kernel_cred(0));
}

int main(int argc, char * argv) {
    ...
    trigger_fp_overwrite(&get_r00t);
    ...
    //trigger fp use
    trigger_vuln_fp();
    // Kernel Executes get_r00t()
    ...
    // Now we have root
    system("/bin/sh");
}
Kernel Space Protections
SMEP / SMAP

Common Exploitation Technique: Supply your own “get root” code.

```c
void get_r00t() {
    commit_creds(prepare_kernel_cred(0));
}

int main(int argc, char * argv) {
    ...
    trigger_fp_overwrite(&get_r00t);
    ...
    //trigger fp use
    trigger_vuln_fp();
    // Kernel Executes get_r00t()
    ...
    // Now we have root
    system("/bin/sh");
    }
```
Kernel Space Protections
SMEP / SMAP

Common Exploitation Technique: Supply your own “get root” code.

```c
void get_root() {
    commit_creds(prepare_kernel_cred(0));
}

int main(int argc, char * argv) {
    ...
    trigger_fp_overwrite(&get_root);
    ...
    //trigger fp use
    trigger_vuln_fp();
    // Kernel Executes get_root()
    ...
    // Now we have root
    system("/bin/sh");
}```
Kernel Space Protections
SMEP / SMAP

Common Exploitation Technique: Supply your own “get root” code.

```c
void get_r00t() {
    commit_creds(prepare_kernel_credential(0));
}

int main(int argc, char * argv) {
    ...
    trigger_fp_overwrite(&get_r00t);
    ...
    //trigger fp use
    trigger_vuln_fp();
    // Kernel Executes get_r00t()
    ...
    // Now we have root
    system("/bin/sh");
}
```
Kernel Space Protections
SMEP / SMAP

**SMEP** prevents this type of attack by triggering a *page fault* if the processor tries to execute memory that has the "user" bit set while in "ring 0".
SMEP prevents this type of attack by triggering a page fault if the processor tries to execute memory that has the “user” bit set while in “ring 0”.

SMAP works similarly, but for data access in general.
Kernel Space Protections

**SMEP / SMAP**

**SMEP** prevents this type of attack by triggering a page fault if the processor tries to execute memory that has the “user” bit set while in “ring 0”.

**SMAP** works similarly, but for data access in general.

This doesn’t *prevent* vulnerabilities, but it adds considerable work to developing a working exploit.
Kernel Space Protections

SMEP / SMAP

SMEP prevents this type of attack by triggering a page fault if the processor tries to execute memory that has the “user” bit set while in “ring 0”.

SMAP works similarly, but for data access in general. This doesn’t prevent vulnerabilities, but it adds considerable work to developing a working exploit.

We need to use ROP, or somehow get executable code into kernel memory.
Example

We’ll walk through a short example of a backdoored LKM to get a feel for dealing with the kernel.
Conclusion

Kernel Exploitation is *weird*, but *extremely powerful*
Conclusion

Kernel Exploitation is *weird*, but extremely powerful.

As userland exploit-dev becomes more challenging and more expensive, kernelspace is becoming a more attractive target.
Kernel Exploitation is **weird**, but *extremely powerful*

As userland exploit-dev becomes more challenging and more expensive, kernelspace is becoming a more attractive target.

A single bug can be used to bypass sandboxes, and gain root privileges, which may otherwise be impossible
Conclusion

*The* book on Kernel Exploitation: