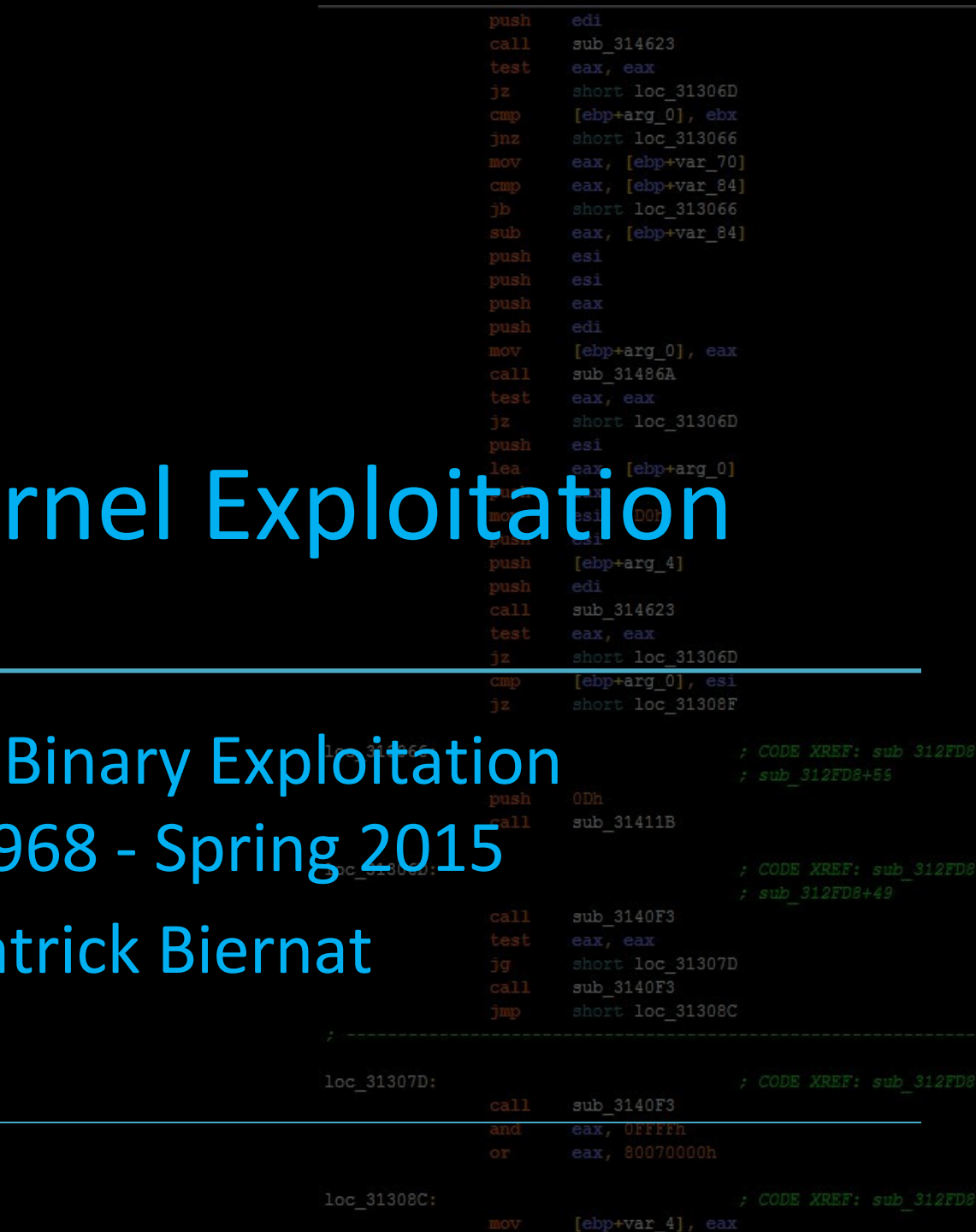
Linux Kernel Exploitation

Modern Binary Exploitation
CSCI 4968 - Spring 2015

Patrick Biernat

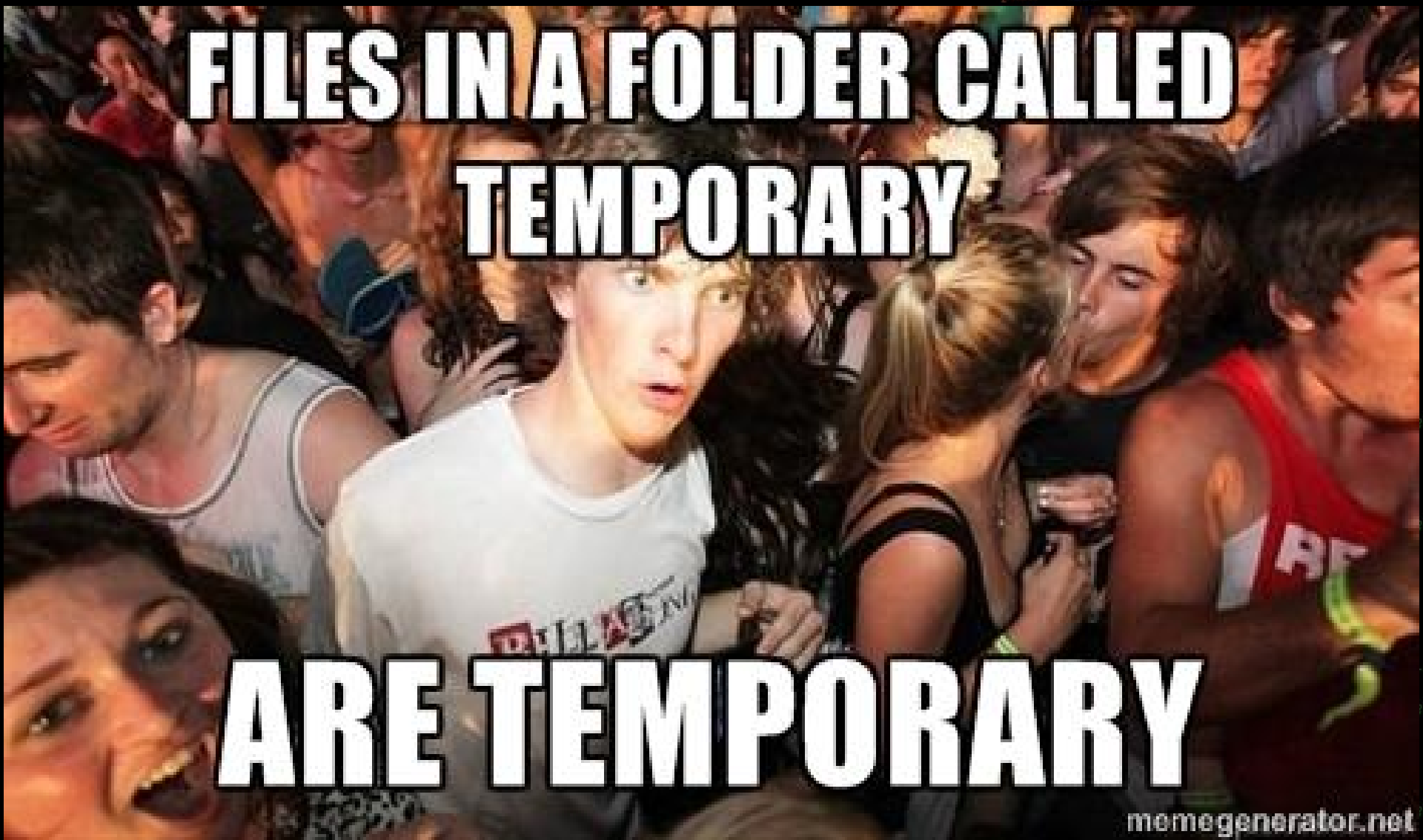


First order of Business

```
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], ebx
                jnz     short loc_313066
                mov     eax, [ebp+var_70]
                cmp     eax, [ebp+var_84]
                jb      short loc_313066
                sub     eax, [ebp+var_84]
                push    esi
                push    esi
                push    eax
                push    edi
                mov     [ebp+arg_0], eax
                call    sub_31486A
                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
                jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+55
                push    0Dh
                call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
                call    sub_3140F3
                test    eax, eax
                jg      short loc_31307D
                call    sub_3140F3
                jmp     short loc_31308C
; ---------------------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
                call    sub_3140F3
                and     eax, 0FFFFh
                or      eax, 80070000h

loc_31308C:                             ; CODE XREF: sub_312FD8
                mov     [ebp+var_4], eax
```

First order of Business

```
push    edi
call    sub_314623
test    eax, eax
jz      short loc_31306D
cmp     [ebp+arg_0], ebx
jnz     short loc_313066
mov     eax, [ebp+var_70]
cmp     eax, [ebp+var_84]
jb      short loc_313066
sub     eax, [ebp+var_84]
push    esi
```

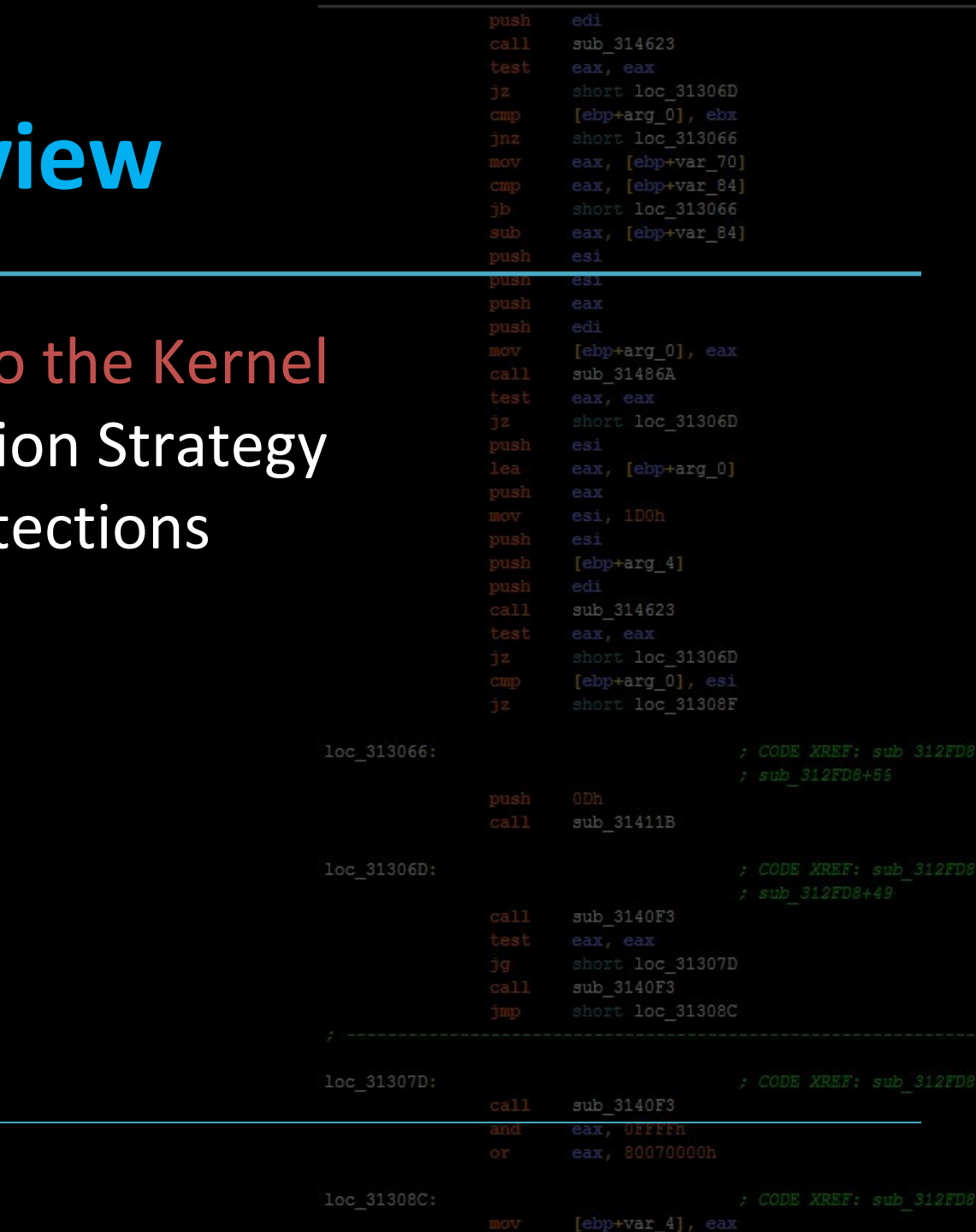


You probably feel like this

```
push    edi
call    sub_314623
test    eax, eax
jz      short loc_31306D
cmp     [ebp+arg_0], ebx
        rt loc_313066
        , [ebp+var_70]
        , [ebp+var_84]
jb      short loc_313066
sub     eax, [ebp+var_84]
push    esi
```

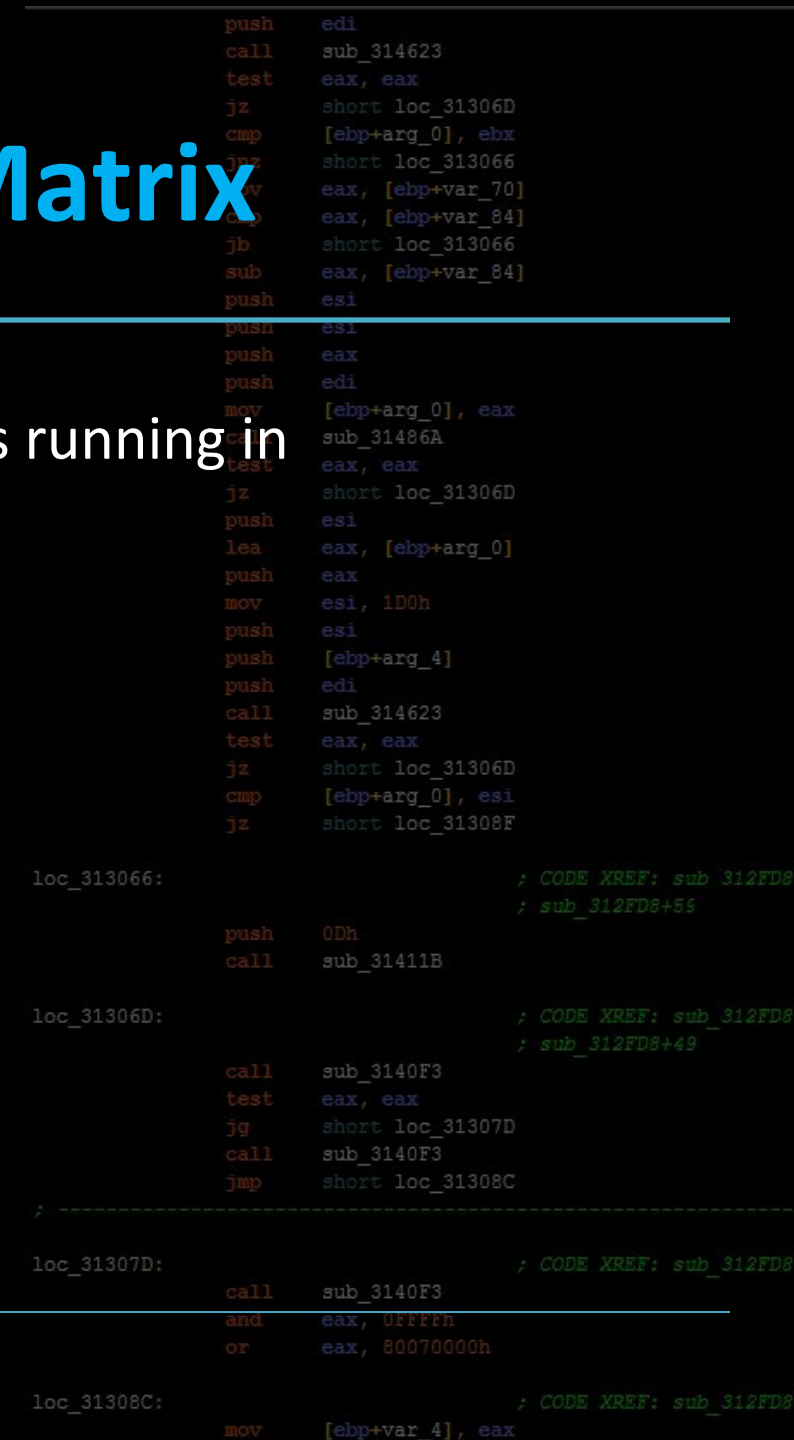
Lecture Overview

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



Jumping out of the Matrix

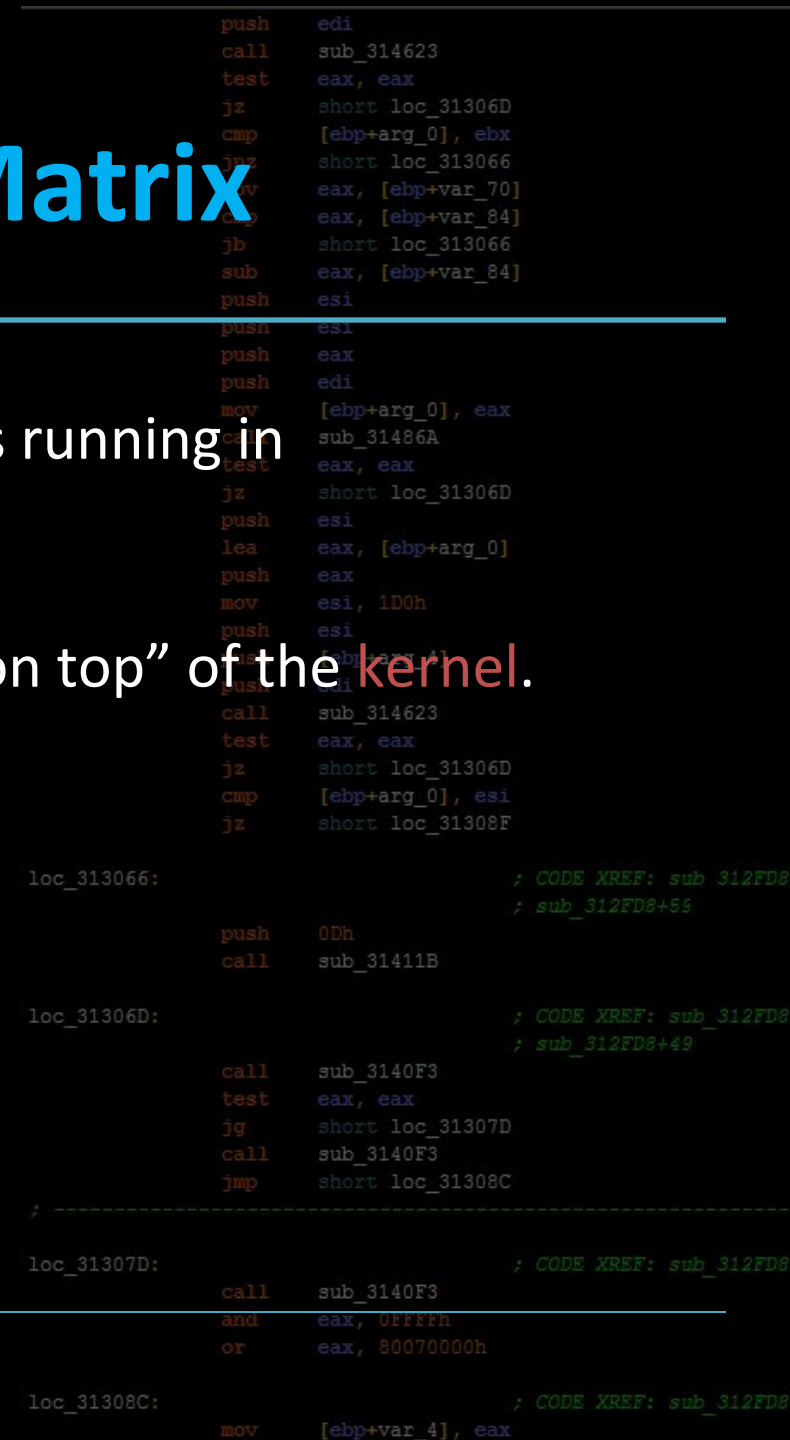
So far, we have been exploiting binaries running in userspace.



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.

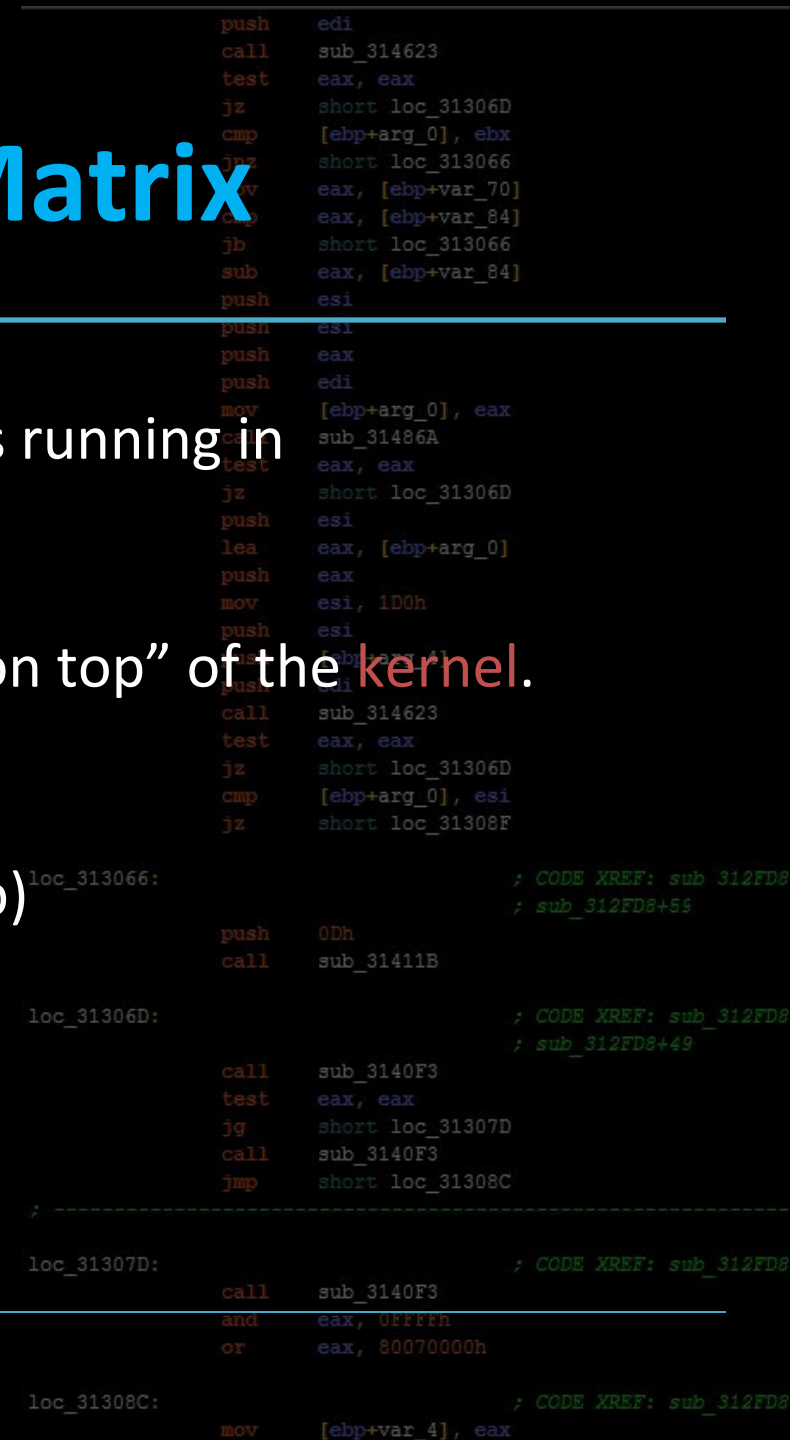


Jumping out of the Matrix

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



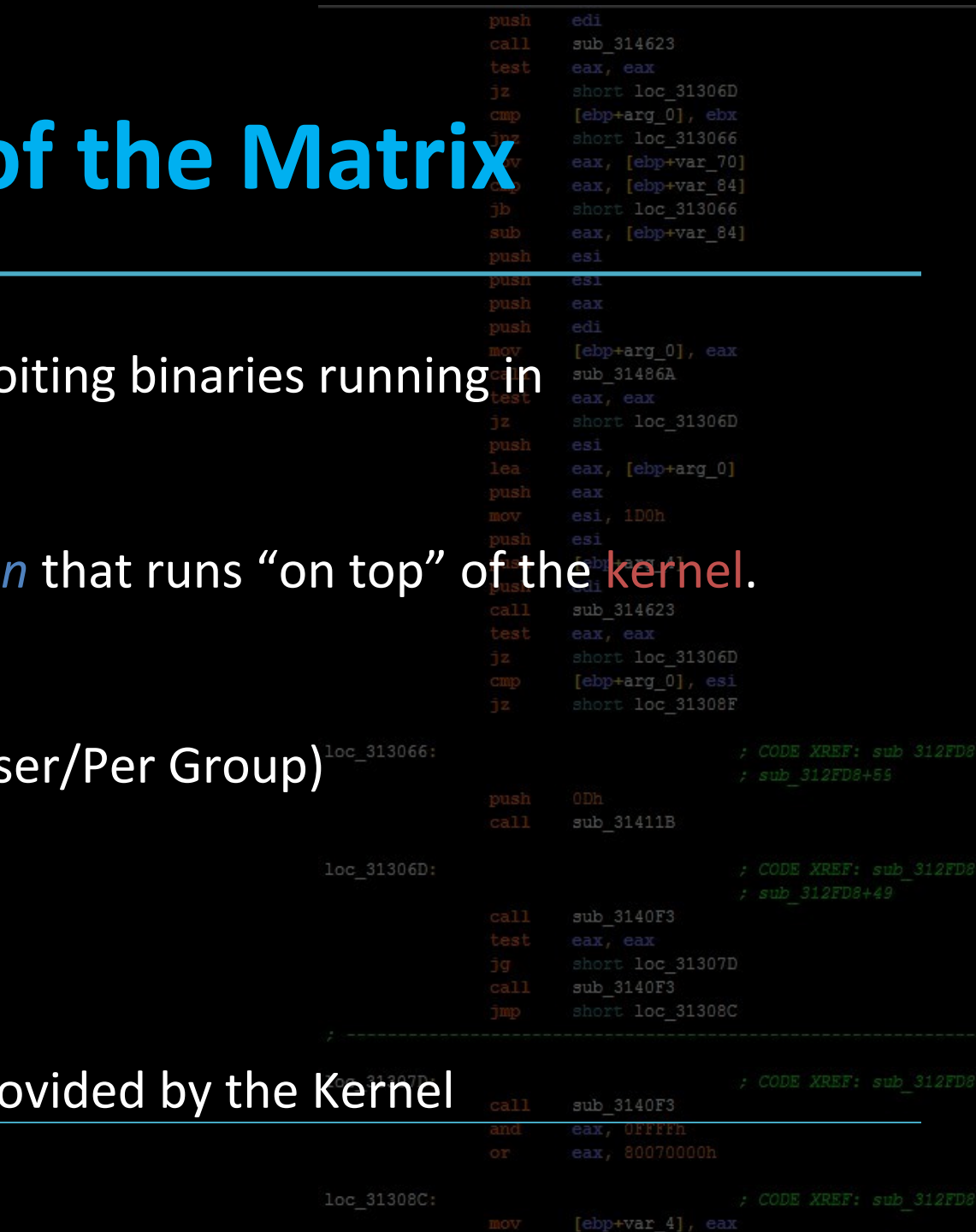
Jumping out of the Matrix

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

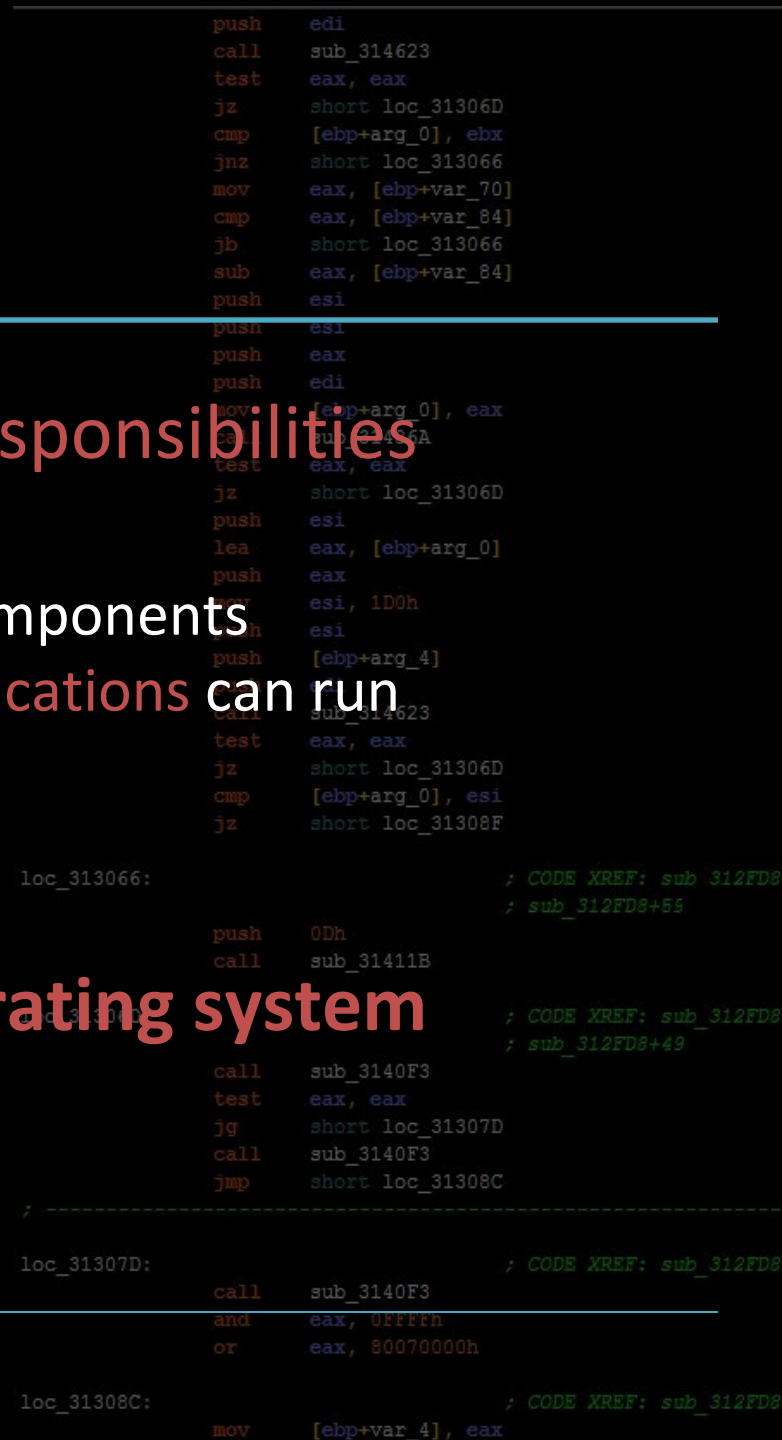
1. Interact with and control hardware components
2. Provide an Environment in which Applications can run

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

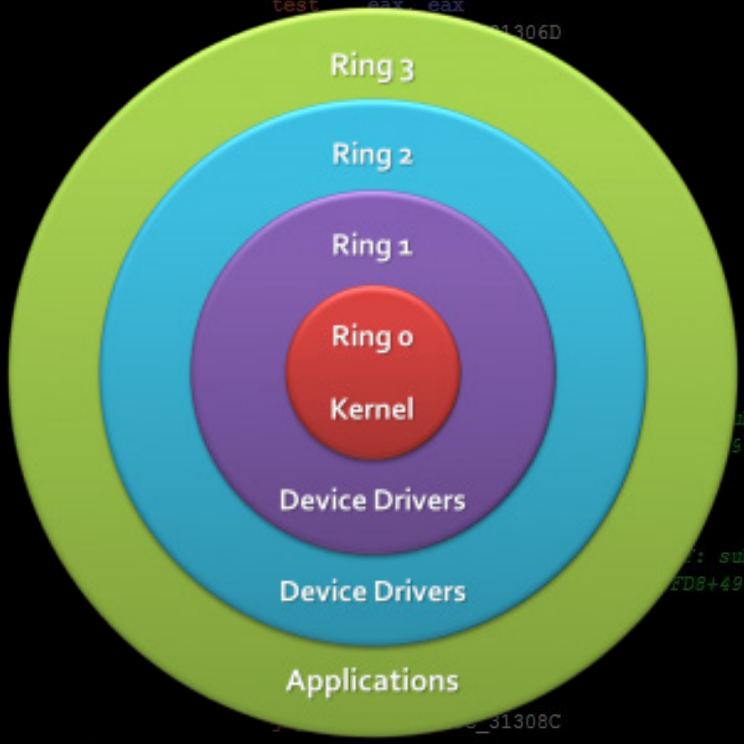


What's a Kernel? - Ring Model

Hardware Enforced Model

0: Privileged, Kernelspace

3: Restricted, Userspace



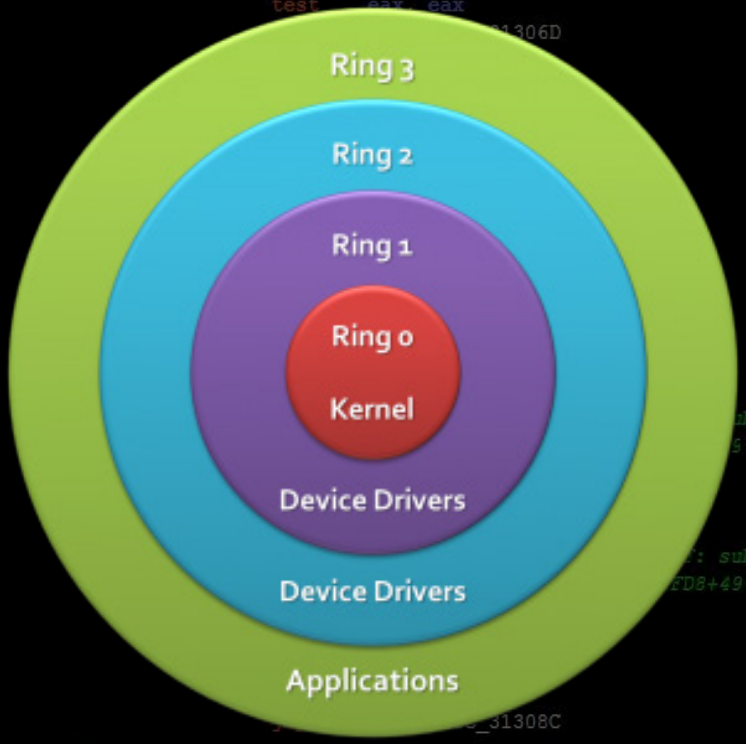
What's a Kernel? - Ring Model

Hardware Enforced Model

0: Privileged, Kernelspace

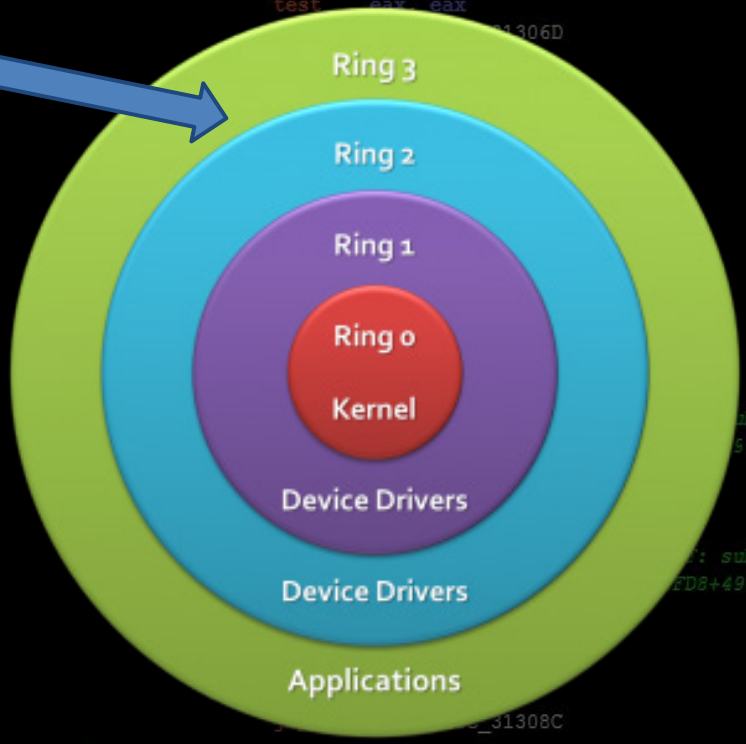
3: Restricted, Userspace

Ring 1 and Ring 2 are not utilized by most popular/modern Operating Systems (Linux / Windows / OSX)



What's a Kernel? - Ring Model

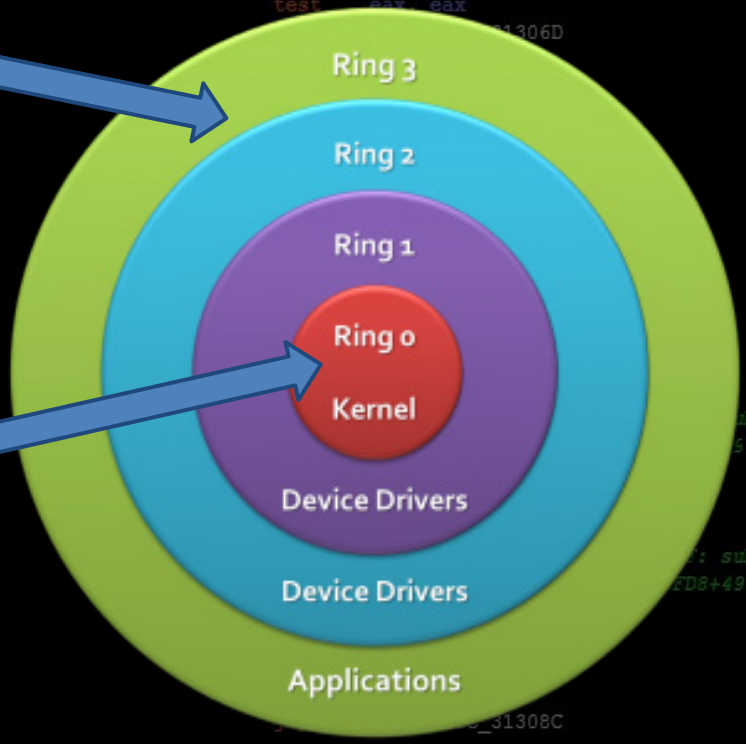
We've Been Here



What's a Kernel? - Ring Model

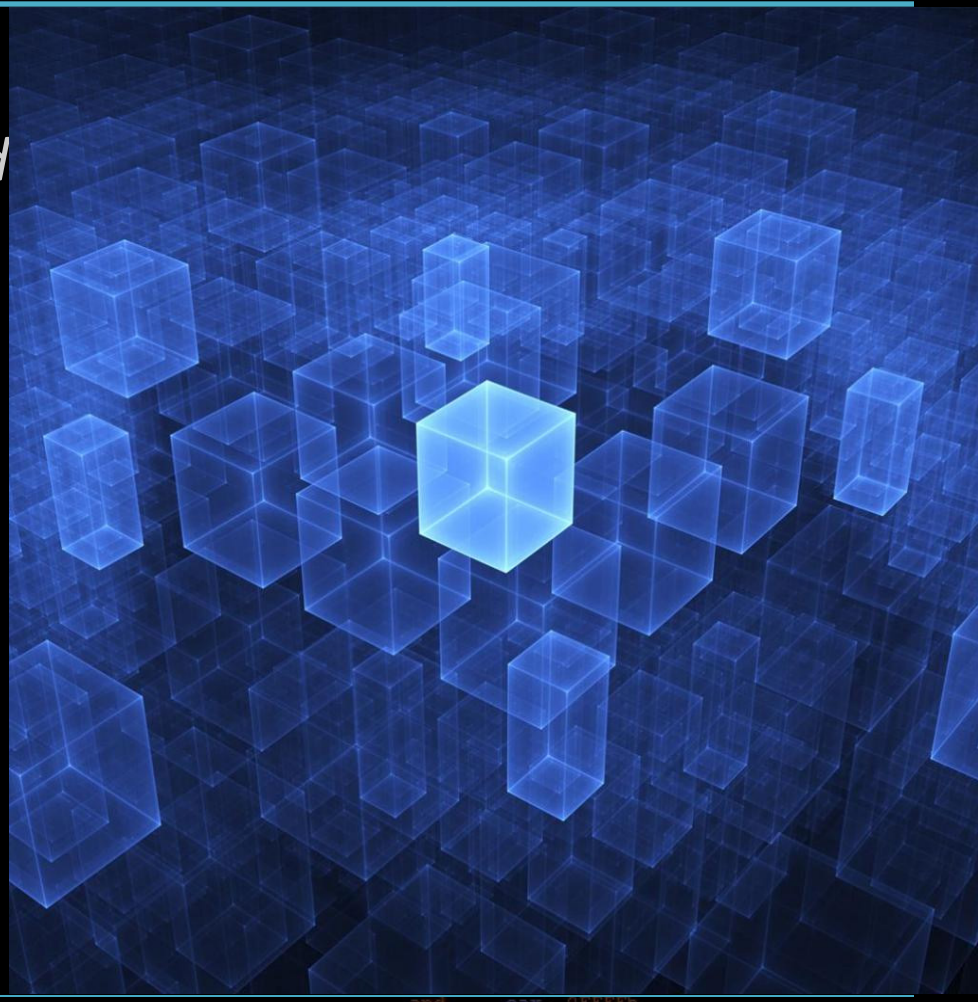
We've Been Here

We're Going Here



Obligatory Matrix Analogy

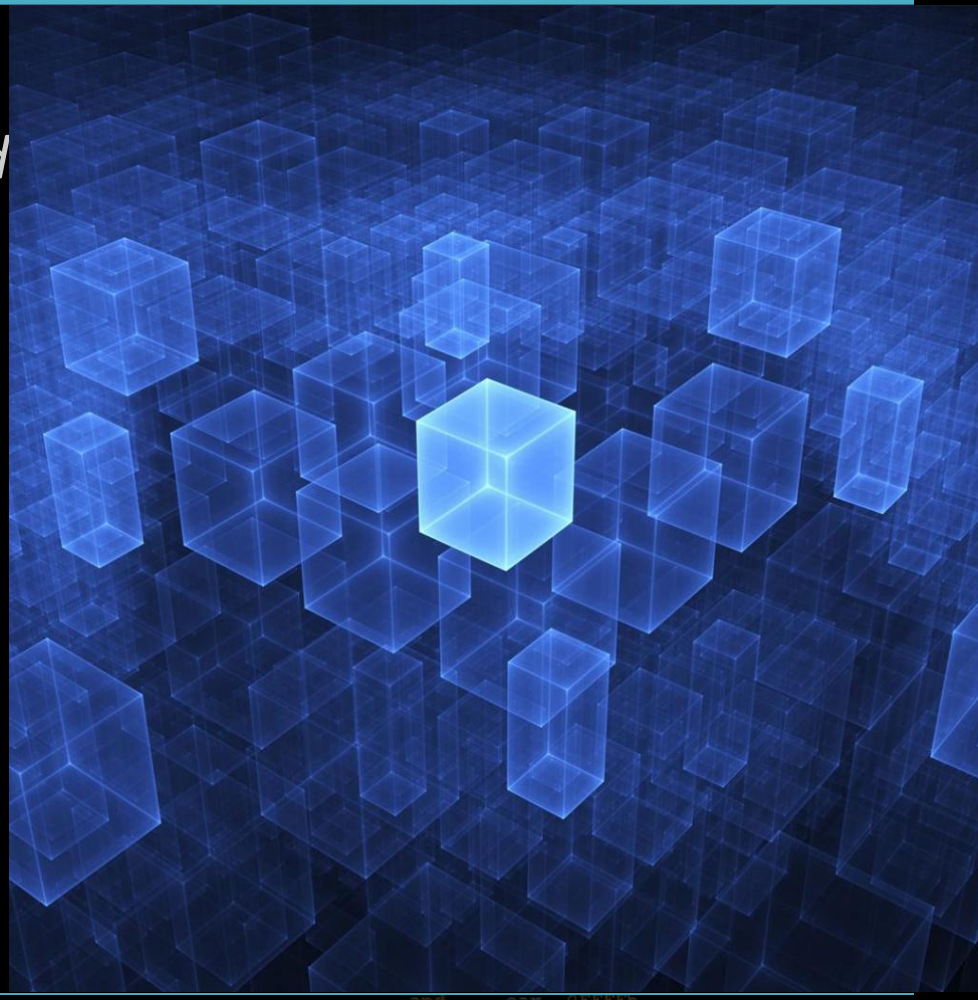
“The Matrix is the world that has been pulled over your eyes to blind you from the truth.” - Morpheus



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

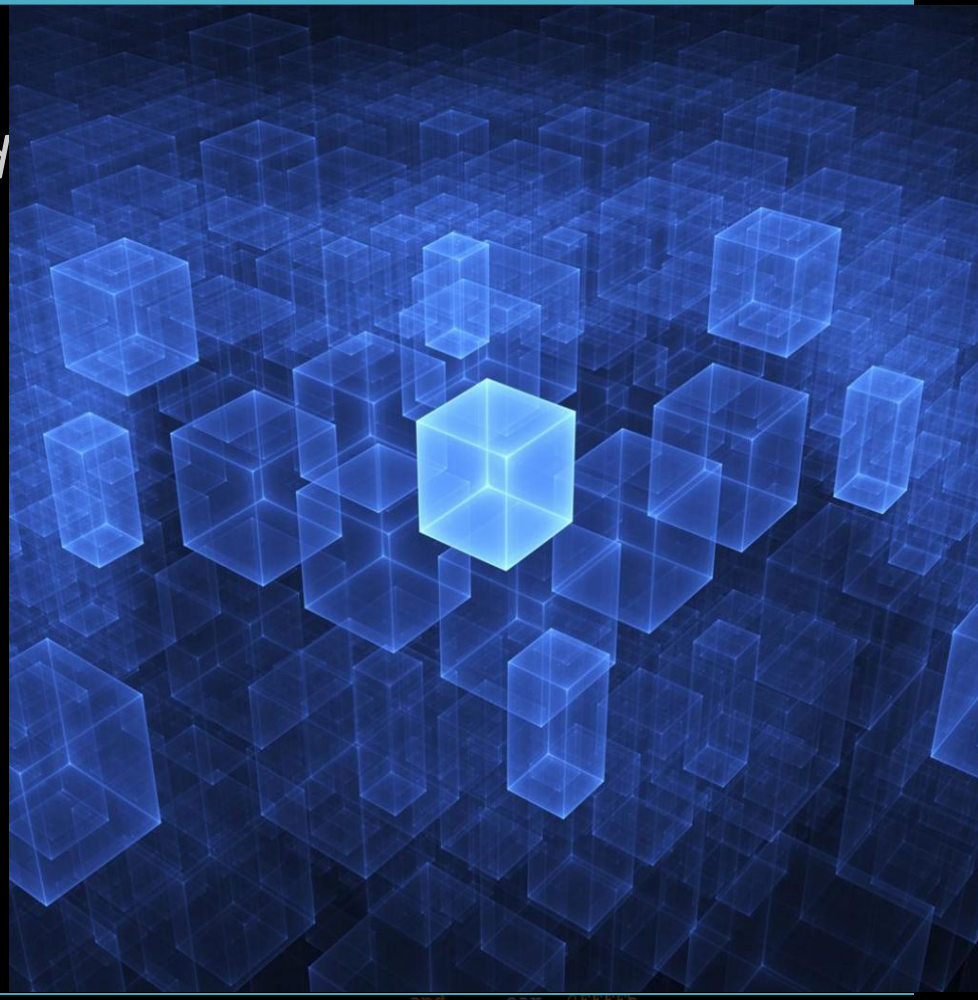


Obligatory Matrix Analogy

“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



Kernel Pwning in Popular Culture

"Jailbreaking" or "rooting" devices often depends on finding and leveraging Kernel bugs

Kernel Pwning in Popular Culture

"Jailbreaking" or "rooting" devices often depends on finding and leveraging Kernel bugs

Remember JailbreakMe?

Kernel Pwning in Popular Culture

"Jailbreaking" or "rooting" devices often depends on finding and leveraging Kernel bugs

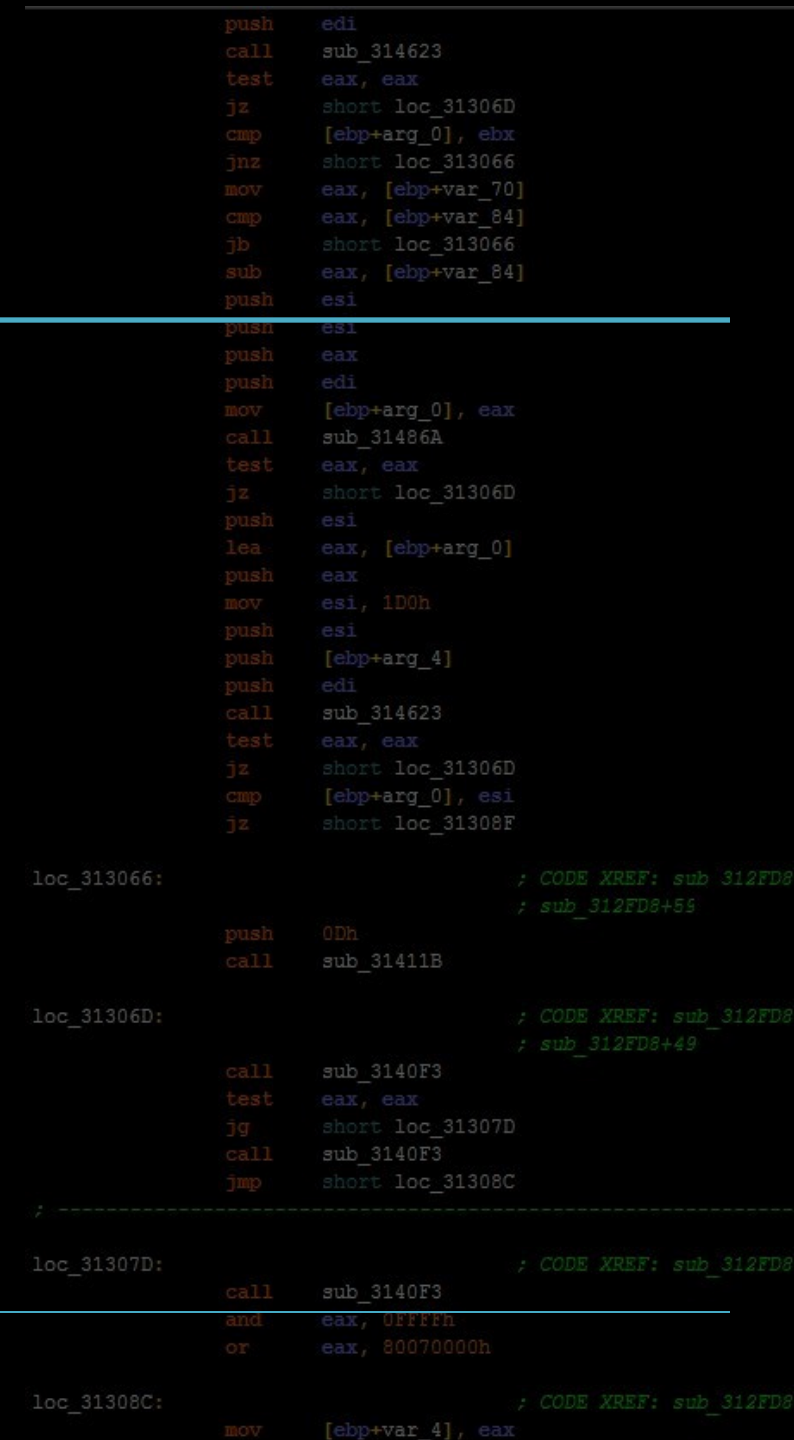
Remember JailbreakMe?

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

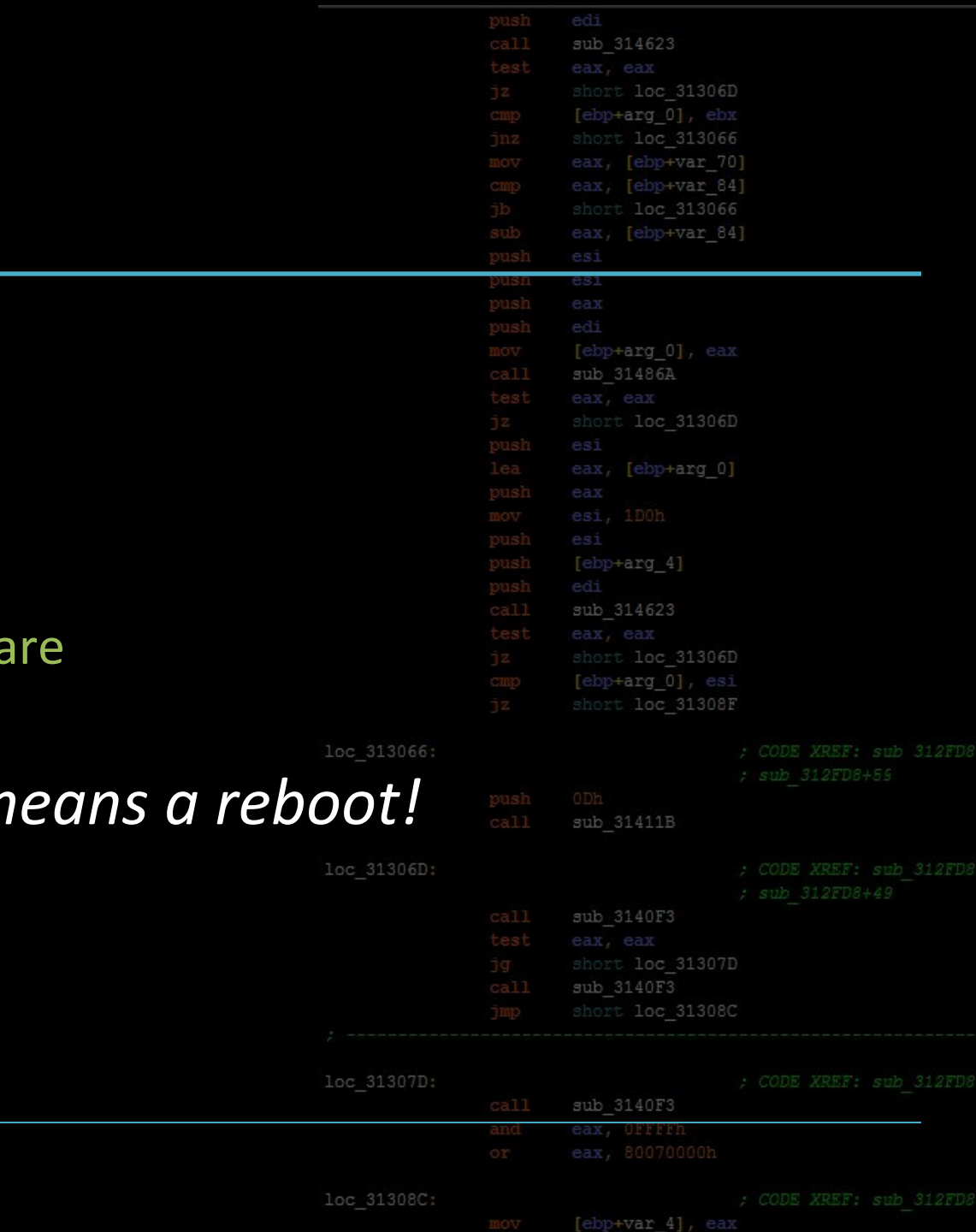


Kernel Basics

Your Kernel is:

Managing your Processes
Managing your Memory
Coordinating your Hardware

A crash oftentimes means a reboot!



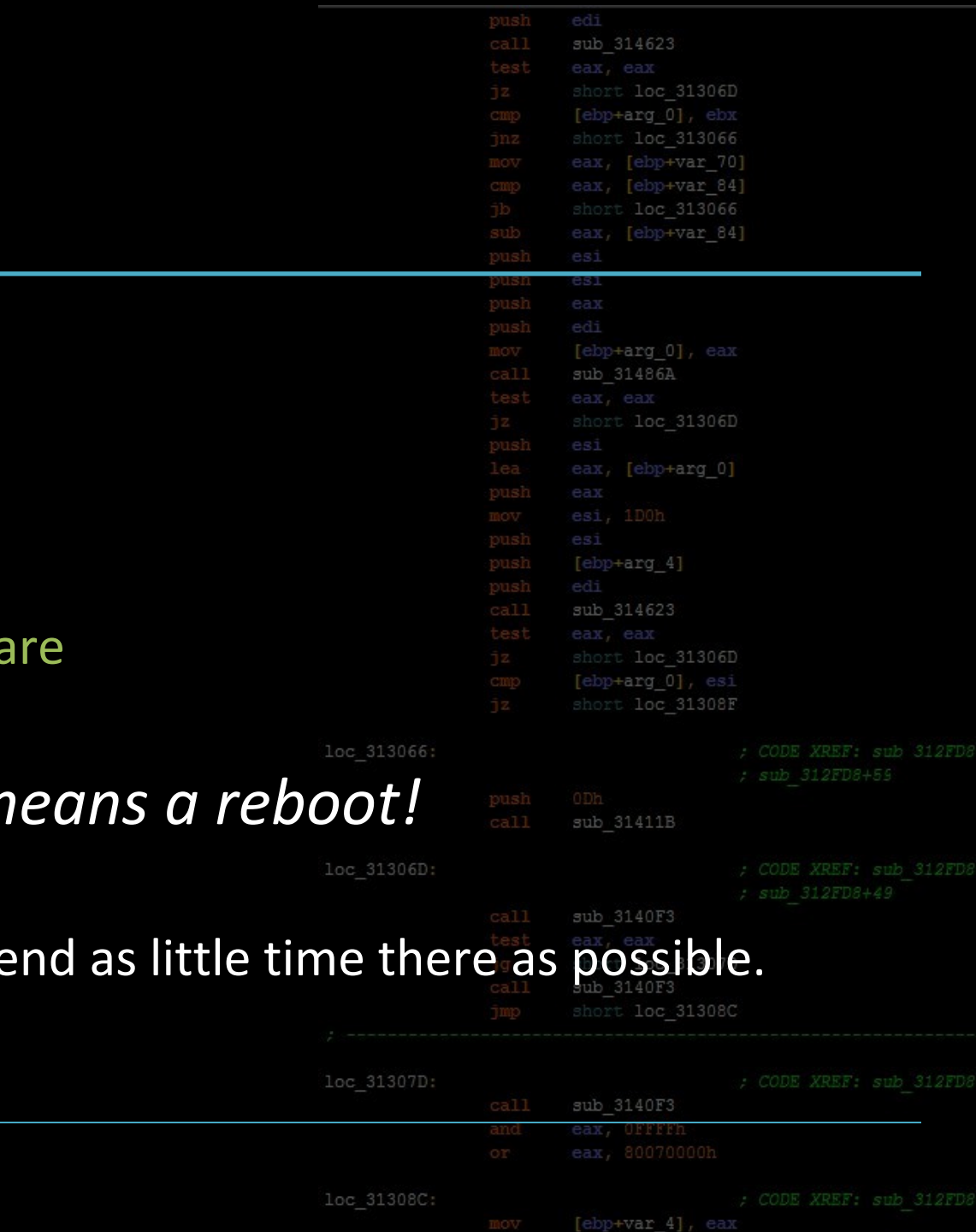
Kernel Basics

Your Kernel is:

Managing your Processes
Managing your Memory
Coordinating your Hardware

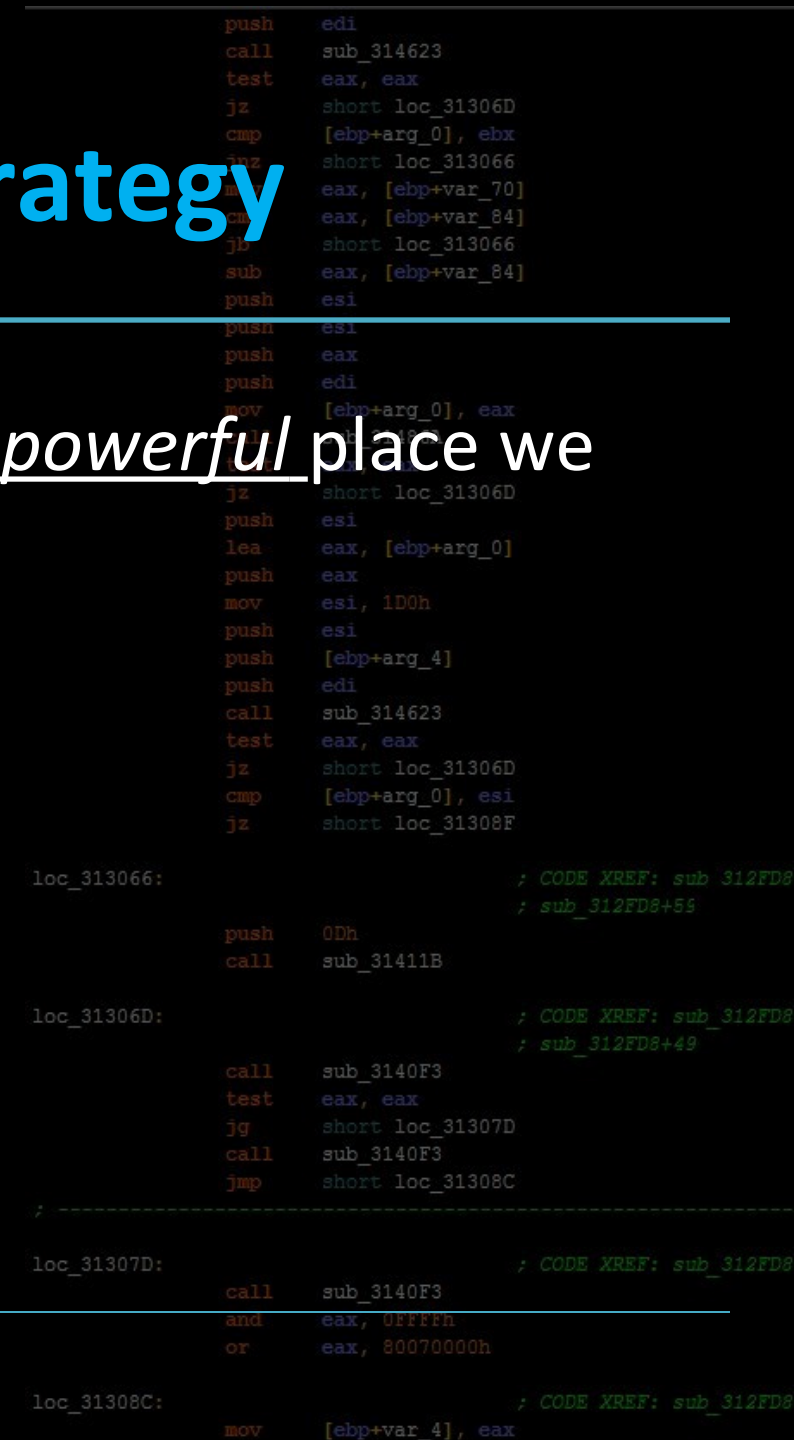
A crash oftentimes means a reboot!

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



Basic Exploitation Strategy

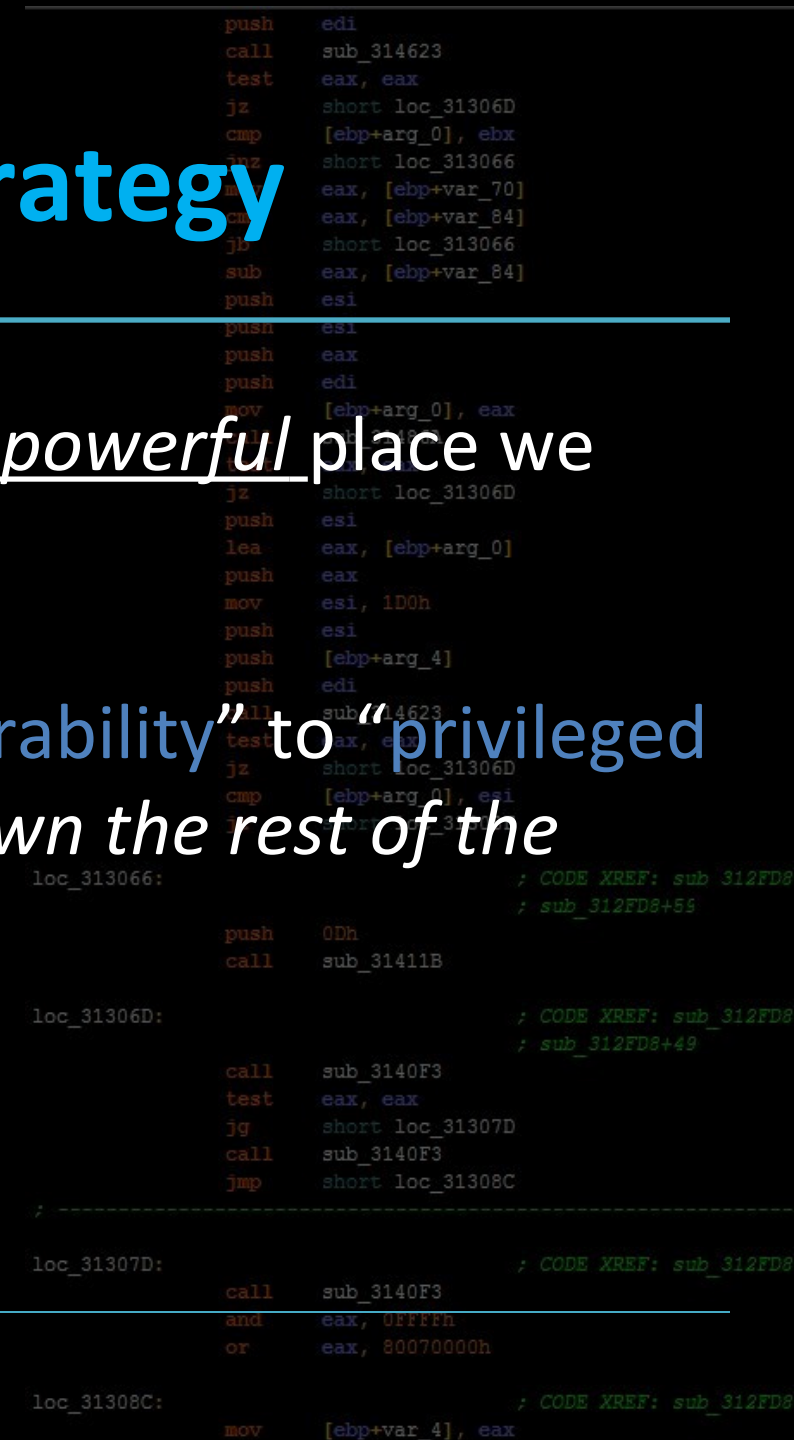
The Kernel is typically *the most powerful* place we can find bugs



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?*

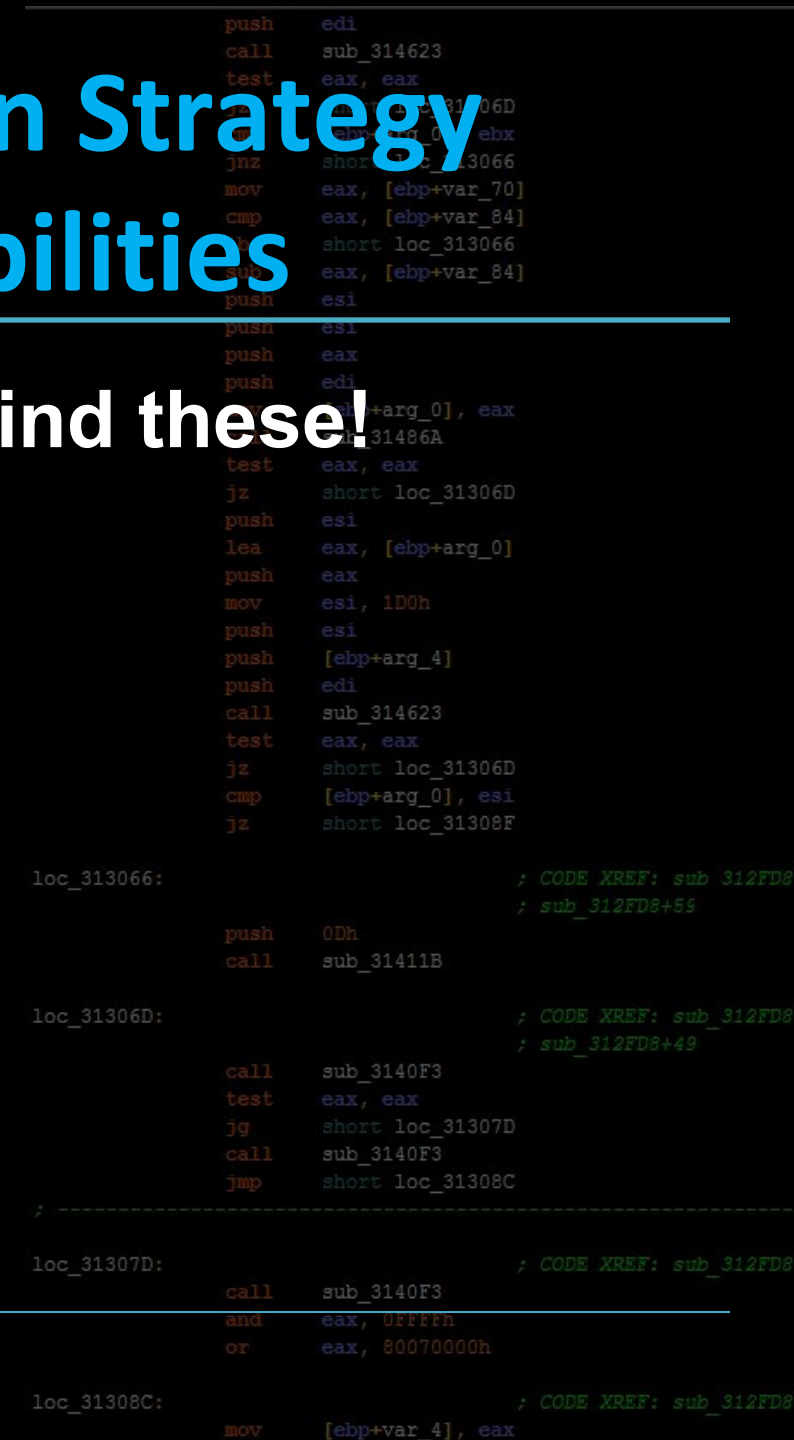


Basic Exploitation Strategy
The Big Picture

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

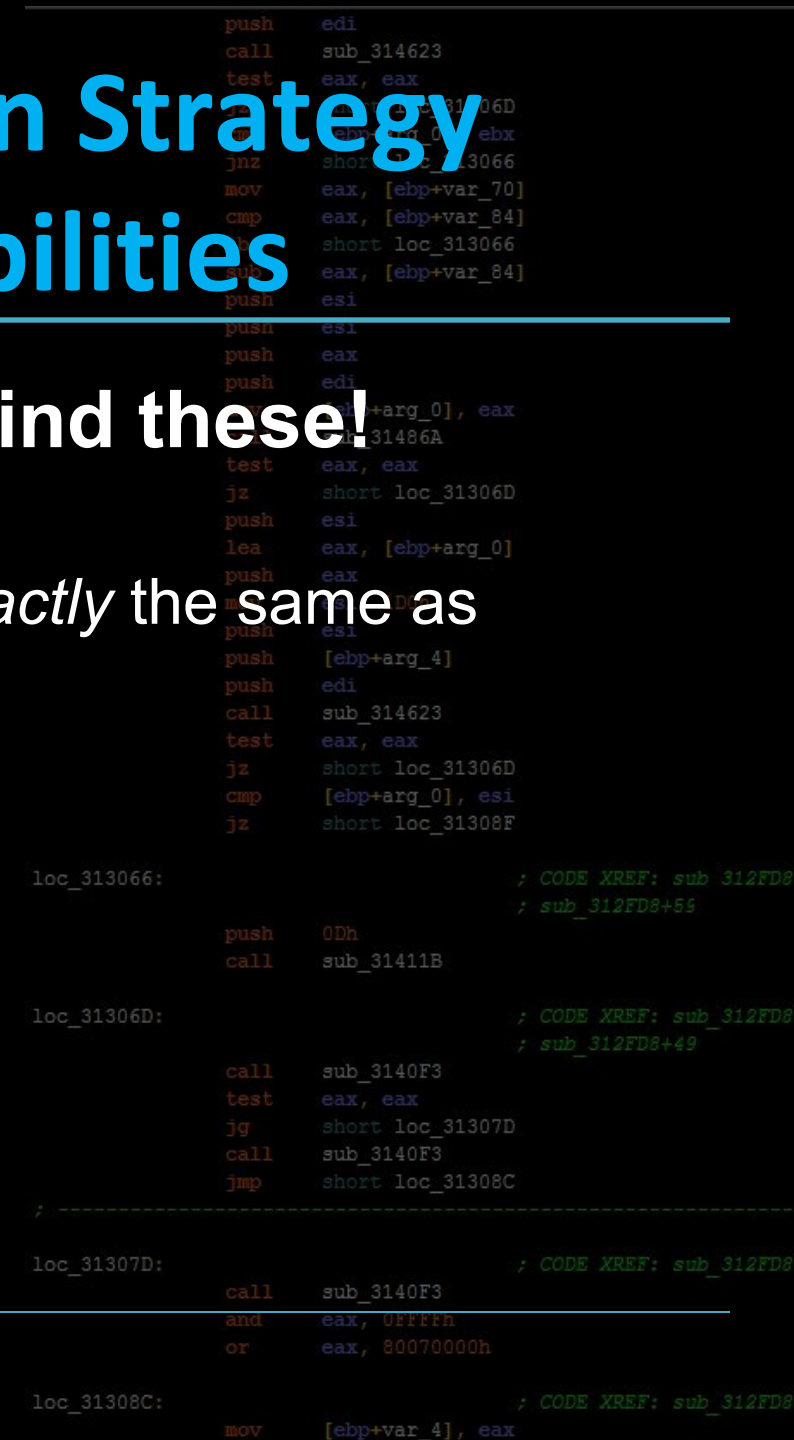
You already know how to find these!



Basic Exploitation Strategy
The Vulnerabilities

You already know how to find these!

Kernel vulnerabilities are almost *exactly* the same as userland vulnerabilities.



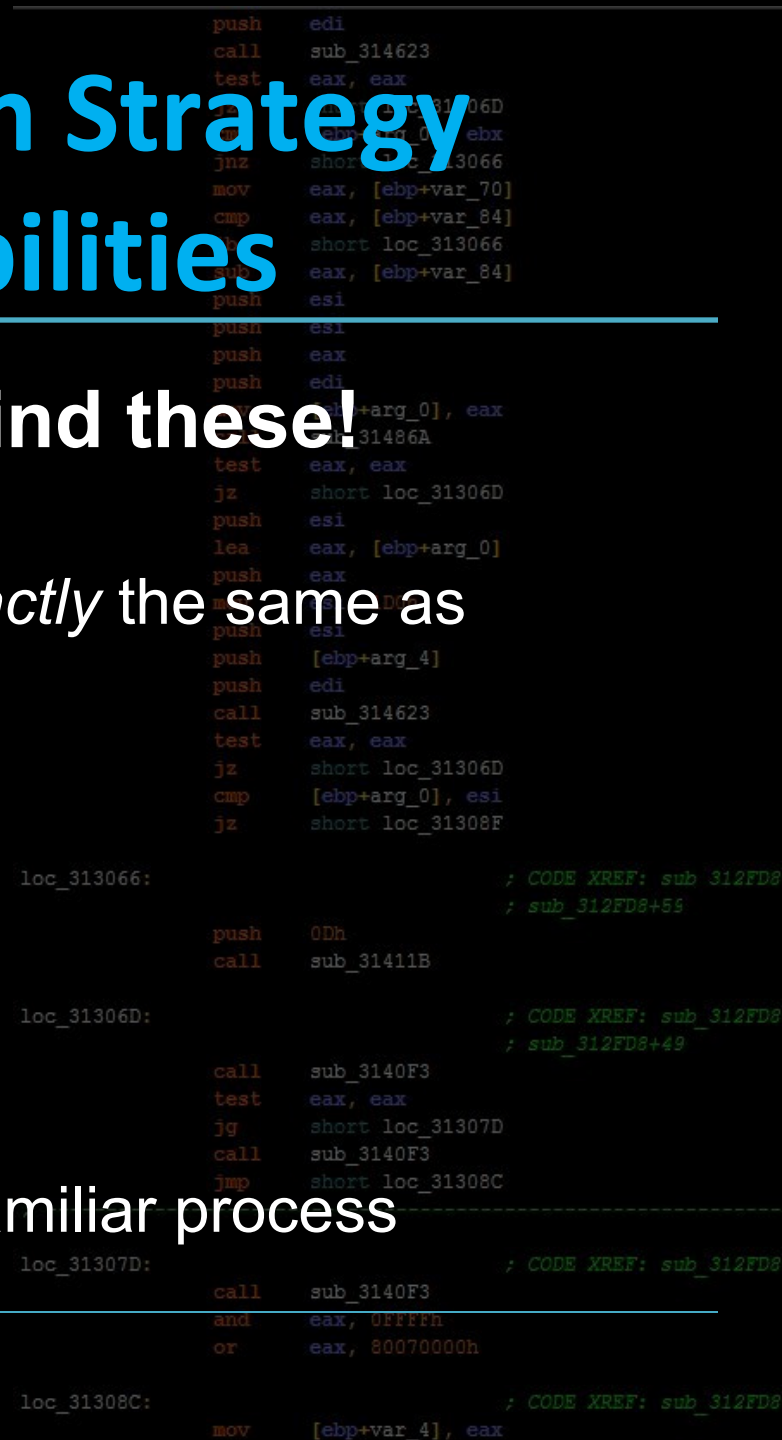
Basic Exploitation Strategy
The Vulnerabilities

You already know how to find these!

Kernel vulnerabilities are almost *exactly* the same as userland vulnerabilities.

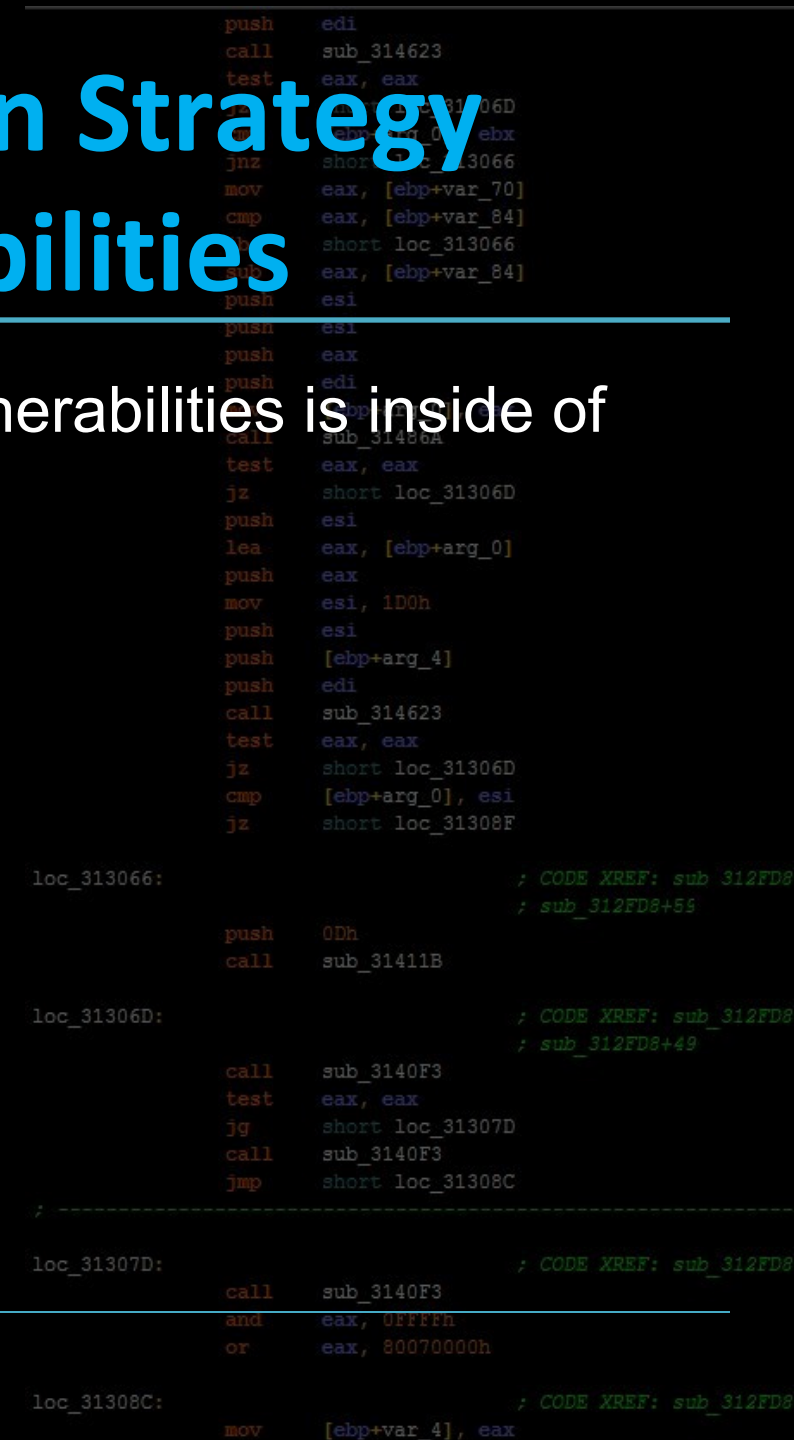
1. Buffer Overflows
2. Signedness issues
3. Partial Overwrites
4. Use-After-Free

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).

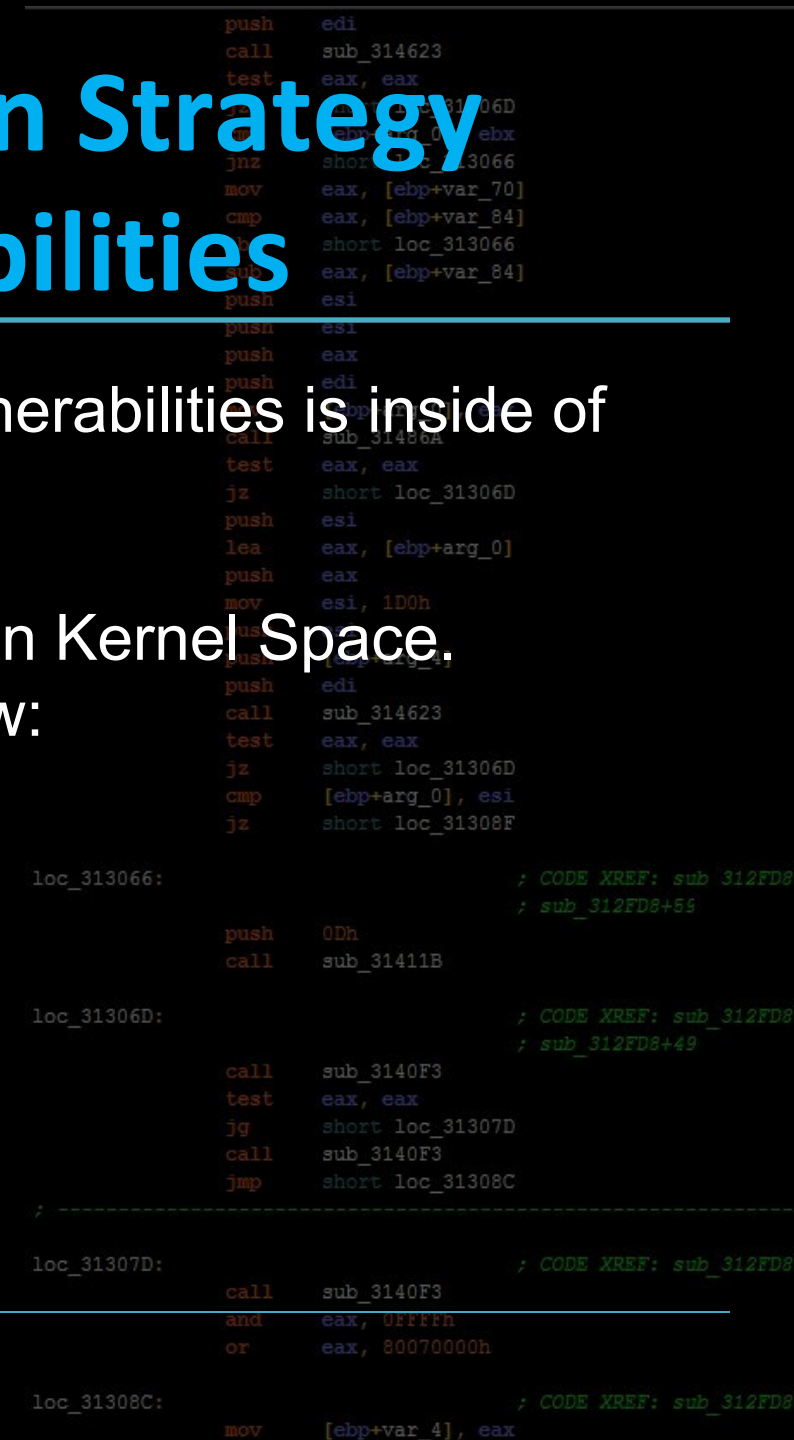


Basic Exploitation Strategy
The Vulnerabilities

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

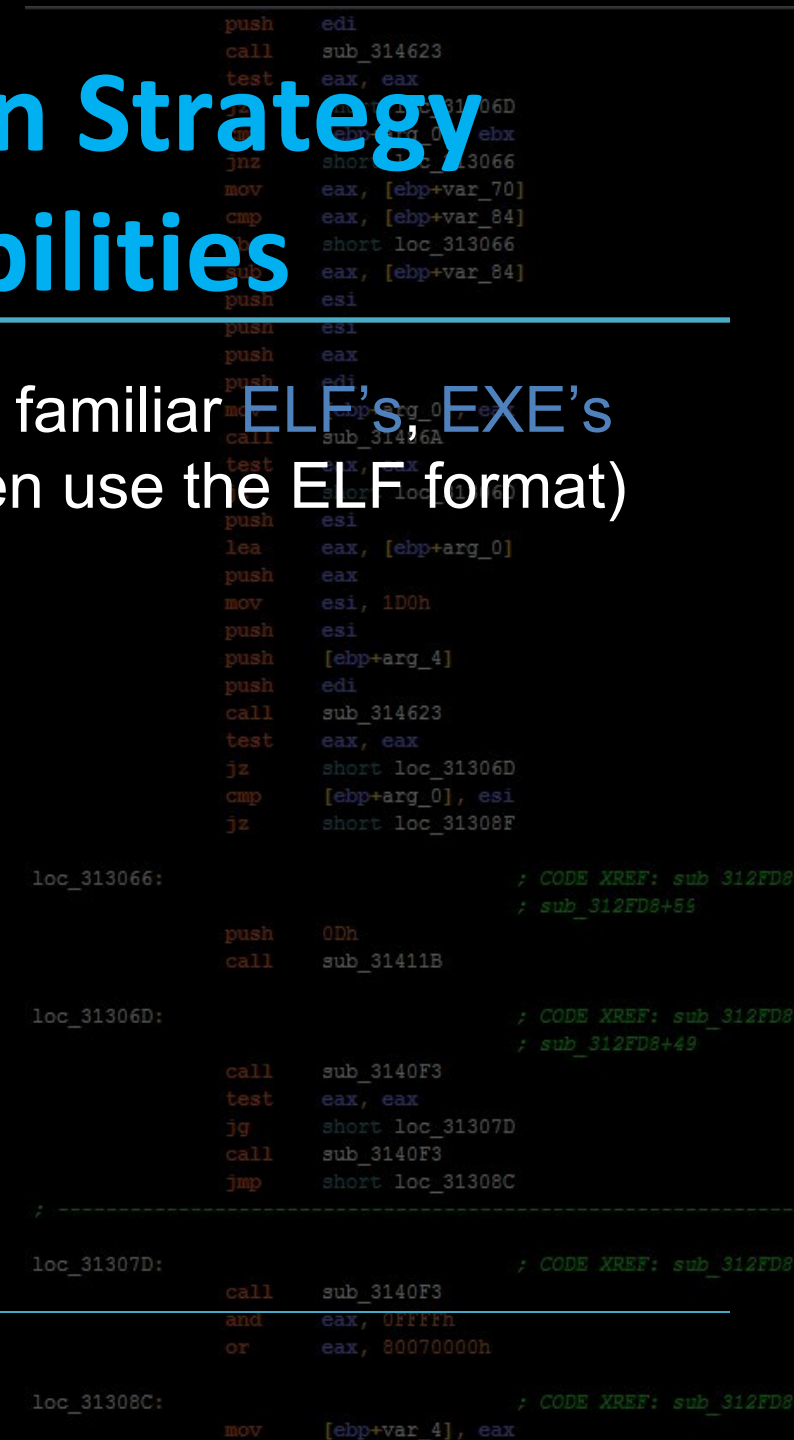
LKMs are like executables that run in Kernel Space.
A few common uses are listed below:

> Device Drivers
> Filesystem Drivers
> Networking Drivers
> Executable Interpreters
> Kernel Extensions
> (rootkits :P)



Basic Exploitation Strategy
The Vulnerabilities

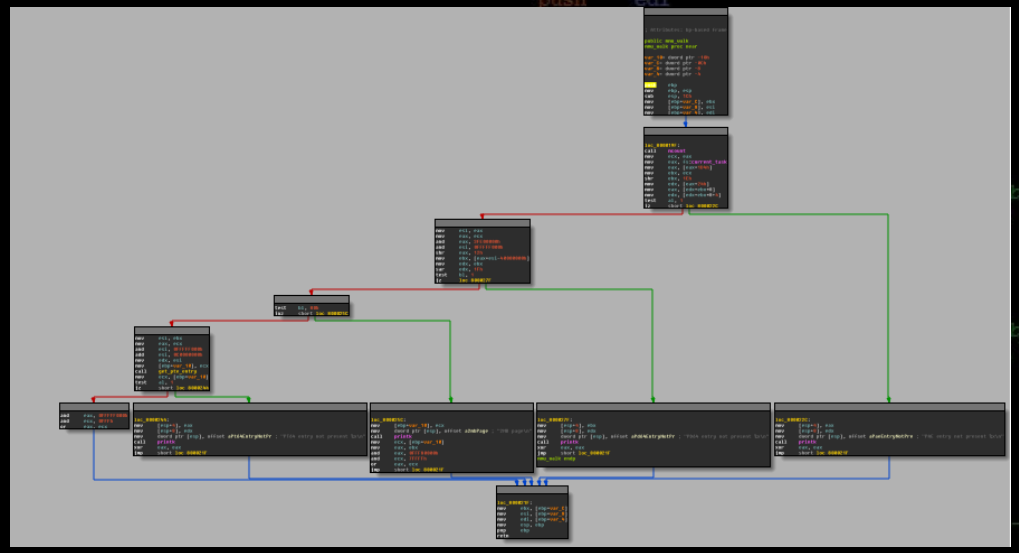
LKMs are just binary blobs like your familiar ELF's, EXE's and MACH-O's. (On Linux, they even use the ELF format)



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!

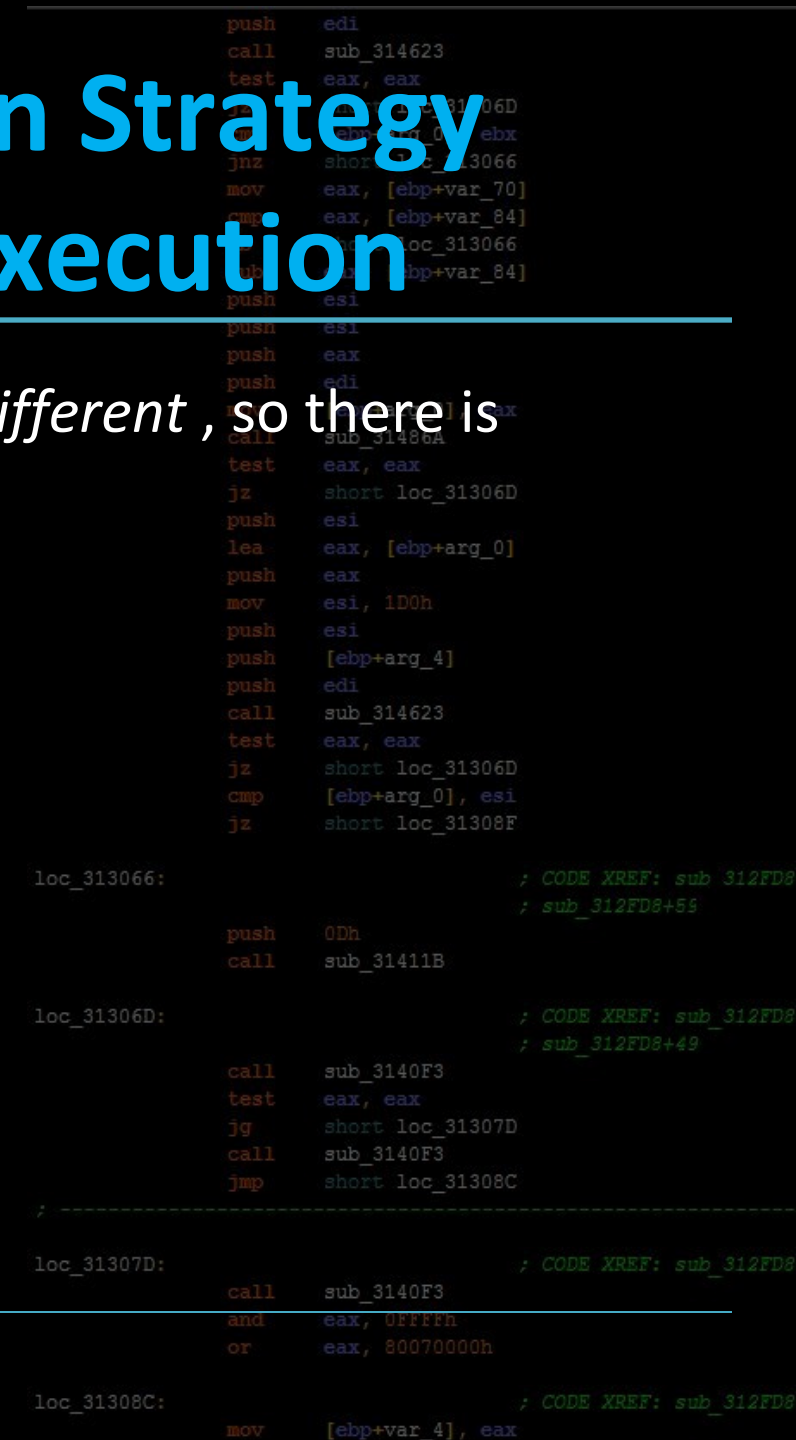
The same basic exploitation techniques apply to Kernelspace (After all, it's just x86 code!)

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

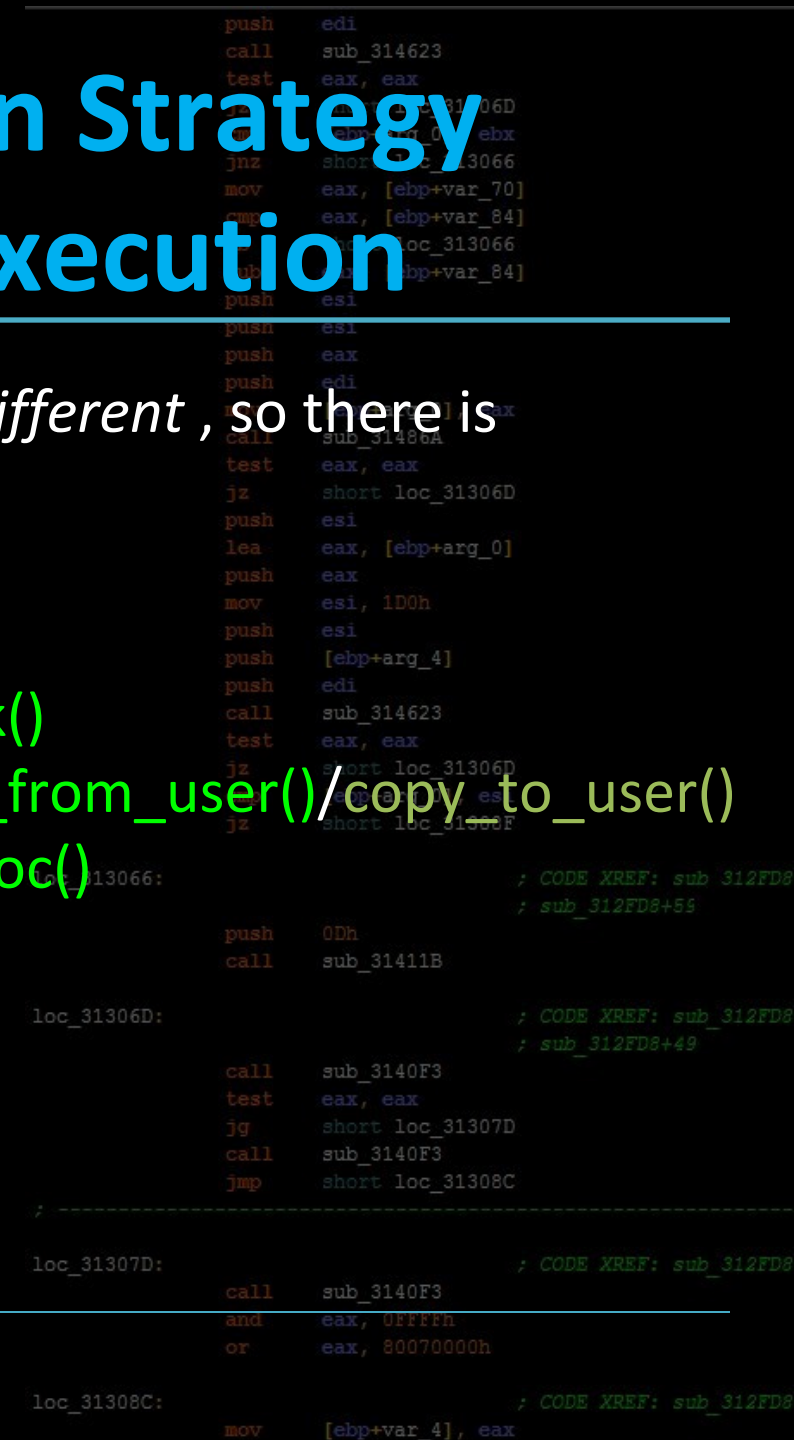
Common Library calls are sometimes *different* , so there is a slight learning curve involved.



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()

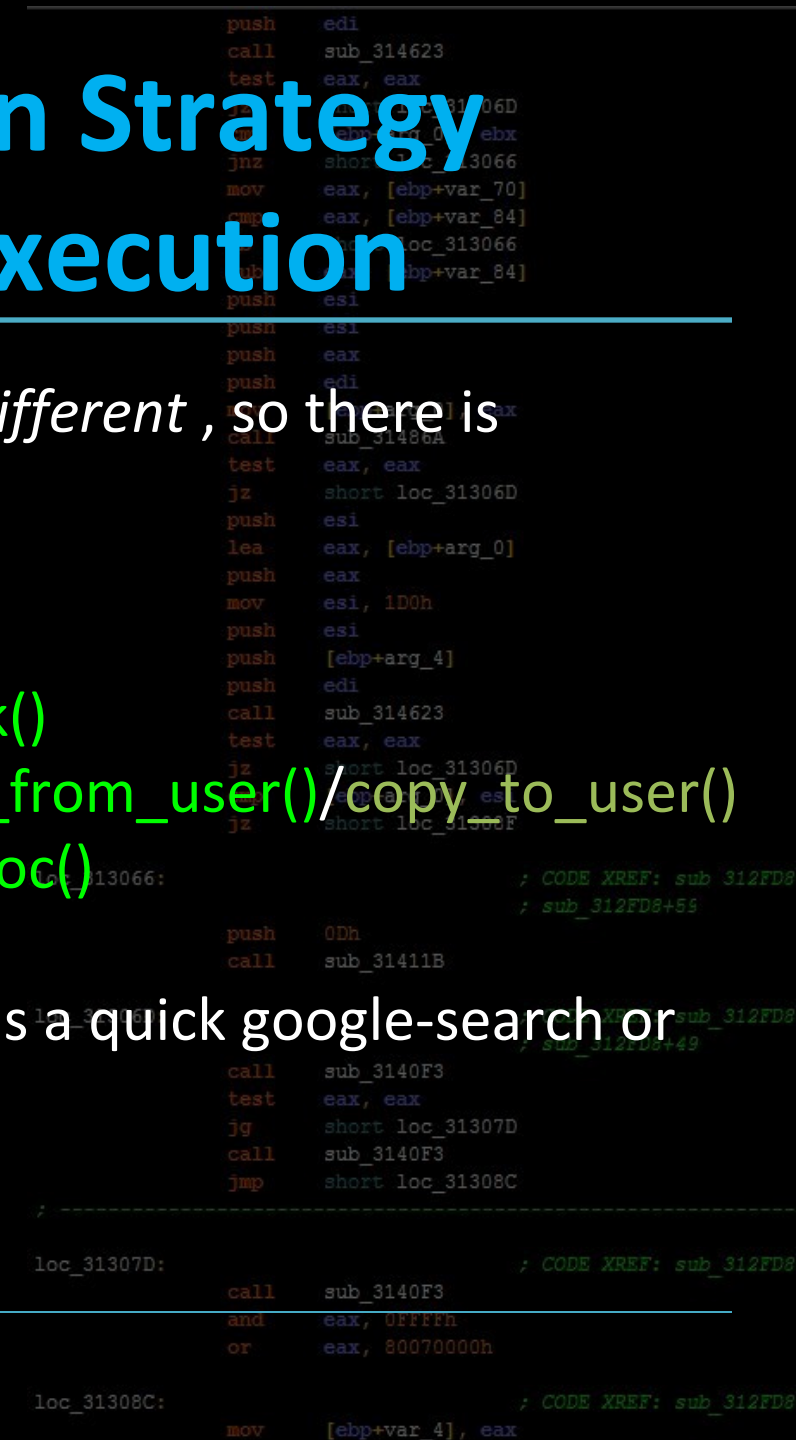


Basic Exploitation Strategy
Gaining Code Execution

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

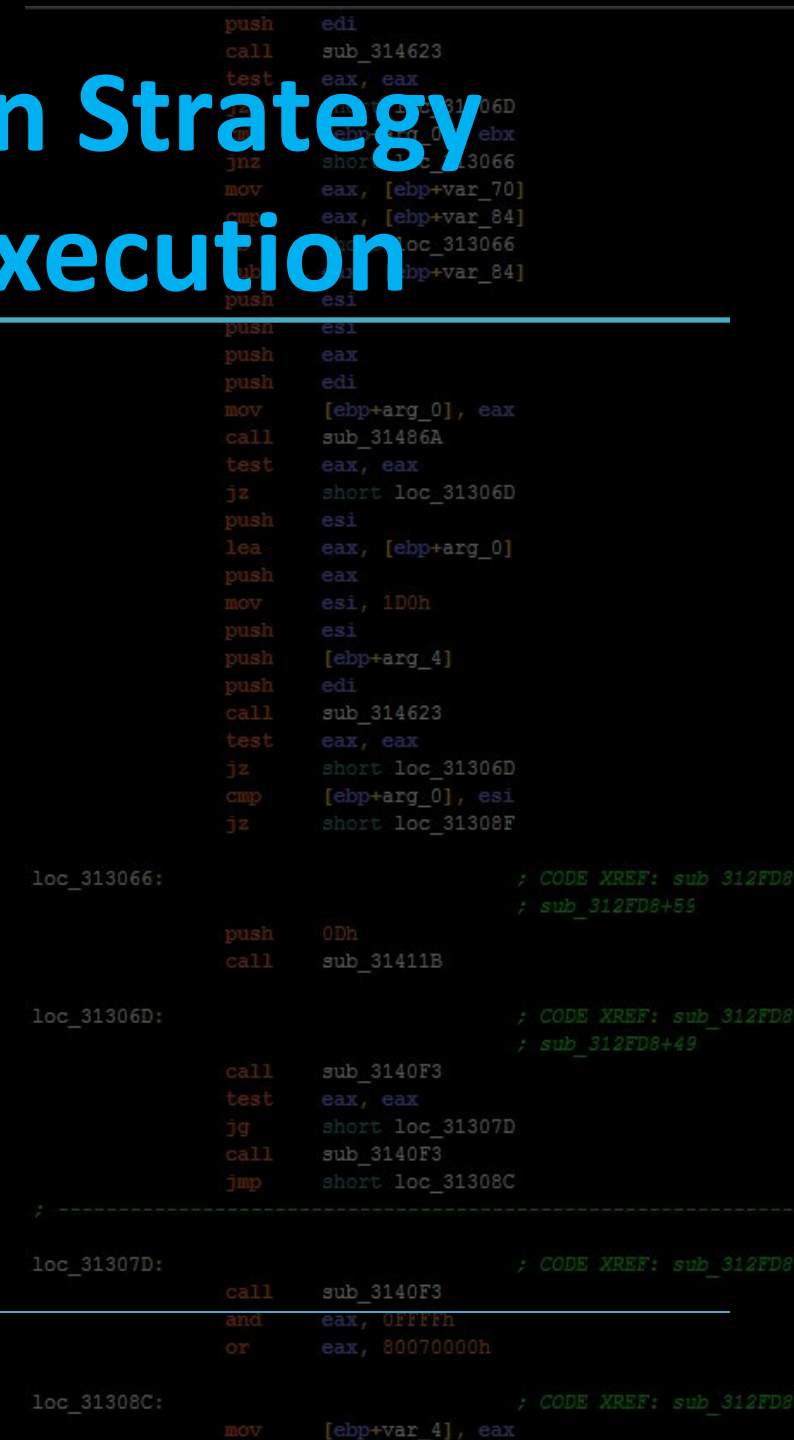
printf()	--->	printk()
memcpy()	--->	copy_from_user()/copy_to_user()
malloc()	--->	kmalloc()

Typically, whatever you want to know is a quick google-search or man page away.



Basic Exploitation Strategy
Gaining Code Execution

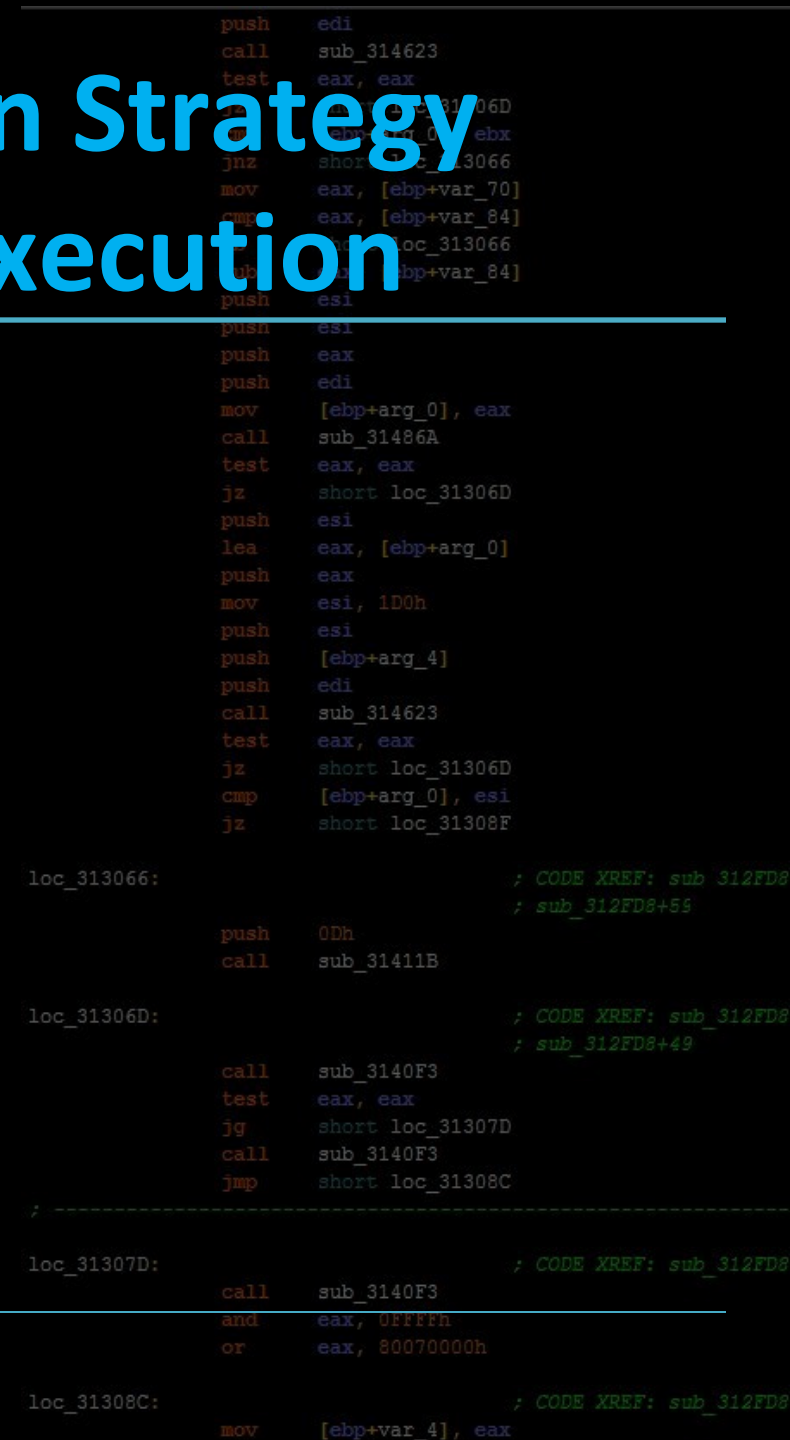
Debugging kernel code can be difficult



Basic Exploitation Strategy
Gaining Code Execution

Debugging kernel code can be difficult

We can't just run the kernel in gdb

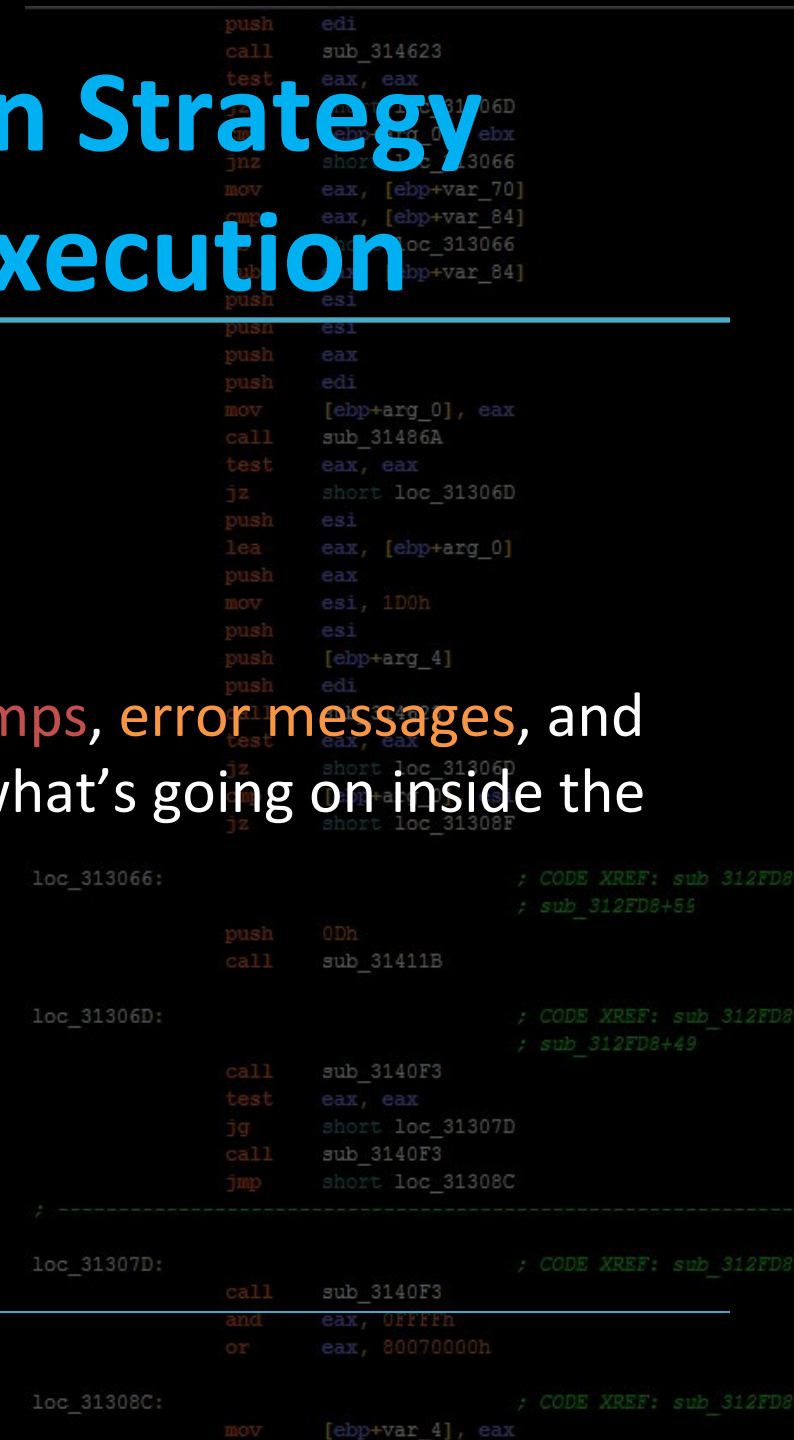


Basic Exploitation Strategy
Gaining Code Execution

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.

```
PID: 0      TASK: ffff81121ff987b0  CPU: 2   COMMAND: "swapper"
 #0 [ffff81011fff3b80] crash_kexec at ffffffff800b1287
 #1 [ffff81011fff3c40] __die at ffffffff80065137
 #2 [ffff81011fff3c80] do_page_fault at ffffffff8006741e
 #3 [ffff81011fff3d70] error_exit at ffffffff8005ddf9
    [exception RIP: uhci_scan_schedule+162]
    RIP: ffffffff880218ee  RSP: ffff81011fff3e20  RFLAGS: 00010007
    RAX: 0000002019105000  RBX: 0000002019105000  RCX: ffff81121ff8cb68
    RDX: 0000000000000000  RSI: 0000000000000000  RDI: ffff81091fe27950
    RBP: ffff81011fff3ed0   R8: 0000000000000000   R9: ffff81012b4f7df8
    R10: 0000000000000001  R11: 00000000af482de4  R12: ffff81091fe27950
    R13: 0000000000000286  R14: ffff81091fe27800  R15: ffffffff80200367
    ORIG_RAX: ffffffffffffffff  CS: 0010  SS: 0018
 #4 [ffff81011fff3e98] uhci_hub_status_data at ffffffff880232da [uhci_hcd]
 #5 [ffff81011fff3ec8] usb_hcd_poll_rh_status at ffffffff80200275
 #6 [ffff81011fff3f08] run_timer_softirq at ffffffff8009a819
 #7 [ffff81011fff3f58] __do_softirq at ffffffff800125a9
 #8 [ffff81011fff3f88] call_softirq at ffffffff8005e30c
 #9 [ffff81011fff3fa0] do_softirq at ffffffff8006d630
#10 [ffff81011fff3fb0] apic_timer_interrupt at ffffffff8005dc9e
--- <IRQ stack> ---
#11 [ffff81011ffefdf8] apic_timer_interrupt at ffffffff8005dc9e
    [exception RIP: acpi_safe_halt+37]
    RIP: ffffffff801a62ab  RSP: ffff81011ffefea0  RFLAGS: 00000246
    RAX: 0000000000000000  RBX: ffff81121ff1f8a8  RCX: 0000000000000000
    RDX: 0000000000000000  RSI: 0000000000000001  RDI: 0000000000000001
    RBP: ffff81011ffefee8   R8: ffff81011ffee000   R9: 000000000000003f
    R10: ffff81091fdc4008  R11: 00000000af482de4  R12: ffff81118d6700c0
    R13: 0000000000402040  R14: 0000000000000000  R15: ffff81118d6700c0
    ORIG_RAX: fffffffffffffff10  CS: 0010  SS: 0018
#12 [ffff81011ffefea0] acpi_processor_idle_simple at ffffffff801a6b29
```

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

```
PID: 0      TASK: ffff81121ff987b0  CPU: 2   COMMAND: "swapper"
 #0 [ffff81011fff3b80] crash_kexec at ffffffff800b1287
 #1 [ffff81011fff3c40] __die at ffffffff80065137
 #2 [ffff81011fff3c80] do_page_fault at ffffffff8006741e
 #3 [ffff81011fff3d70] error_exit at ffffffff8005ddf9
    [exception RIP: uhci_scan_schedule+162]
    RIP: ffffffff880218ee  RSP: ffff81011fff3e20  RFLAGS: 00010007
    RAX: 0000002019105000  RBX: 0000002019105000  RCX: ffff81121ff8cb68
    RDX: 0000000000000000  RSI: 0000000000000000  RDI: ffff81091fe27950
    RBP: ffff81011fff3ed0   R8: 0000000000000000   R9: ffff81012b4f7df8
    R10: 0000000000000001  R11: 00000000af482de4  R12: ffff81091fe27950
    R13: 0000000000000286  R14: ffff81091fe27800  R15: ffffffff80200367
    ORIG_RAX: ffffffffffffffff  CS: 0010  SS: 0018
 #4 [ffff81011fff3e98] uhci_hub_status_data at ffffffff880232da [uhci_hcd]
 #5 [ffff81011fff3ec8] usb_hcd_poll_rh_status at ffffffff80200275
 #6 [ffff81011fff3f08] run_timer_softirq at ffffffff8009a819
 #7 [ffff81011fff3f58] __do_softirq at ffffffff800125a9
 #8 [ffff81011fff3f88] call_softirq at ffffffff8005e30c
 #9 [ffff81011fff3fa0] do_softirq at ffffffff8006d630
#10 [ffff81011fff3fb0] apic_timer_interrupt at ffffffff8005dc9e
--- <IRQ stack> ---
#11 [ffff81011ffefdf8] apic_timer_interrupt at ffffffff8005dc9e
    [exception RIP: acpi_safe_halt+37]
    RIP: ffffffff801a62ab  RSP: ffff81011ffefea0  RFLAGS: 00000246
    RAX: 0000000000000000  RBX: ffff81121ff1f8a8  RCX: 0000000000000000
    RDX: 0000000000000000  RSI: 0000000000000001  RDI: 0000000000000001
    RBP: ffff81011ffefee8   R8: ffff81011ffee000   R9: 000000000000003f
    R10: ffff81091fdc4008  R11: 00000000af482de4  R12: ffff81118d6700c0
    R13: 0000000000402040  R14: 0000000000000000  R15: ffff81118d6700c0
    ORIG_RAX: fffffffffffffff10  CS: 0010  SS: 0018
#12 [ffff81011ffefea0] acpi_processor_idle_simple at ffffffff801a6b29
```

Basic Exploitation Strategy
Gaining Code Execution

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

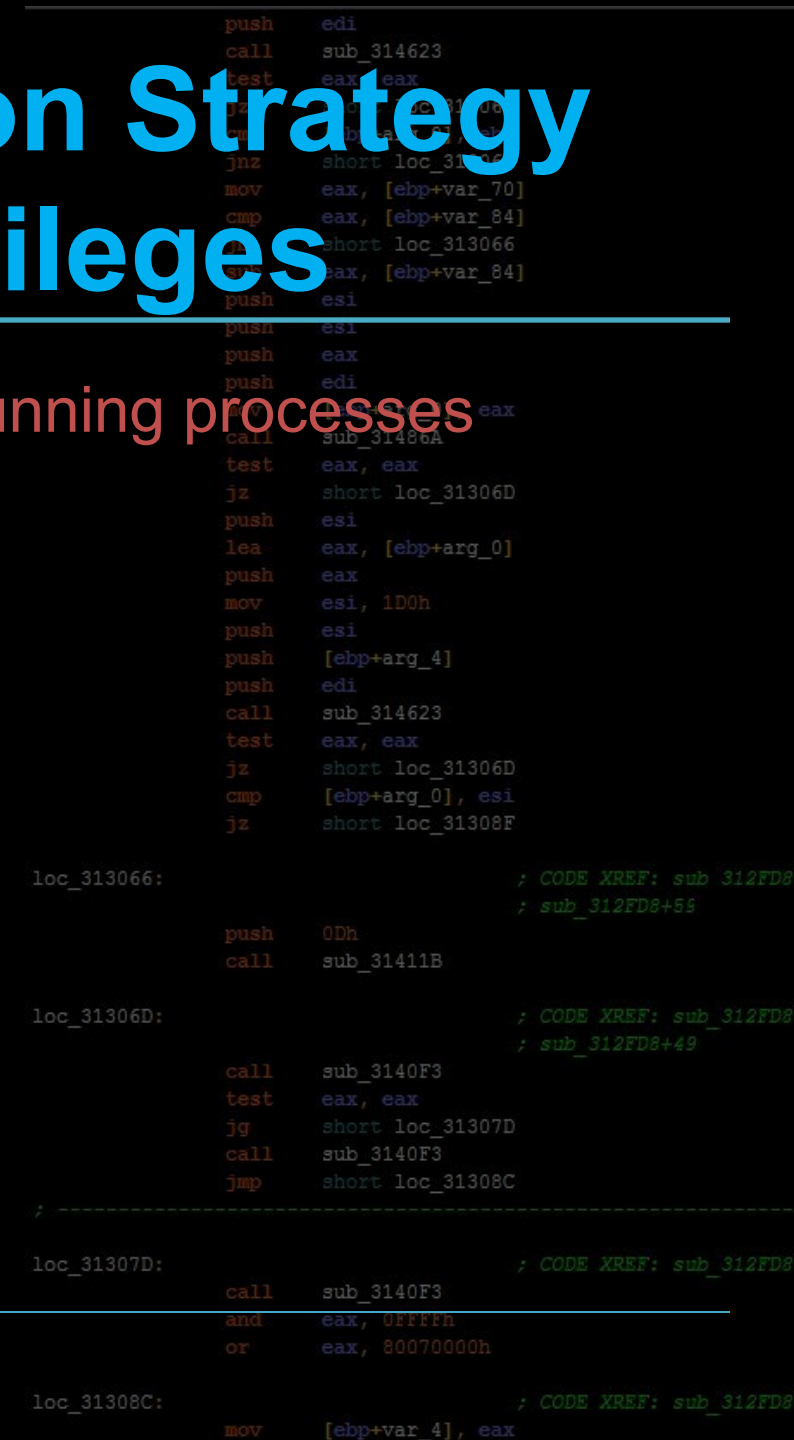
Call Trace
Register Dump
Stack Dump

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

```
PID: 0      TASK: ffff81121ff987b0  CPU: 2   COMMAND: "swapper"
 #0 [ffff81011fff3b80] crash_kexec at ffffffff800b1287
 #1 [ffff81011fff3c40] __die at ffffffff80065137
 #2 [ffff81011fff3c80] do_page_fault at ffffffff8006741e
 #3 [ffff81011fff3d70] error_exit at ffffffff8005ddf9
    [exception RIP: uhci_scan_schedule+162]
    RIP: ffffffff880218ee  RSP: ffff81011fff3e20  RFLAGS: 00010007
    RAX: 0000002019105000  RBX: 0000002019105000  RCX: ffff81121ff8cb68
    RDX: 0000000000000000  RSI: 0000000000000000  RDI: ffff81091fe27950
    RBP: ffff81011fff3ed0   R8: 0000000000000000   R9: ffff81012b4f7df8
    R10: 0000000000000001  R11: 00000000af482de4  R12: ffff81091fe27950
    R13: 0000000000000286  R14: ffff81091fe27800  R15: ffffffff80200367
    ORIG_RAX: ffffffffffffffff  CS: 0010  SS: 0018
 #4 [ffff81011fff3e98] uhci_hub_status_data at ffffffff880232da [uhci_hcd]
 #5 [ffff81011fff3ec8] usb_hcd_poll_rh_status at ffffffff80200275
 #6 [ffff81011fff3f08] run_timer_softirq at ffffffff8009a819
 #7 [ffff81011fff3f58] __do_softirq at ffffffff800125a9
 #8 [ffff81011fff3f88] call_softirq at ffffffff8005e30c
 #9 [ffff81011fff3fa0] do_softirq at ffffffff8006d630
#10 [ffff81011fff3fb0] apic_timer_interrupt at ffffffff8005dc9e
--- <IRQ stack> ---
#11 [ffff81011ffefdf8] apic_timer_interrupt at ffffffff8005dc9e
    [exception RIP: acpi_safe_halt+37]
    RIP: ffffffff801a62ab  RSP: ffff81011ffefea0  RFLAGS: 00000246
    RAX: 0000000000000000  RBX: ffff81121ff1f8a8  RCX: 0000000000000000
    RDX: 0000000000000000  RSI: 0000000000000001  RDI: 0000000000000001
    RBP: ffff81011ffefee8   R8: ffff81011ffee000   R9: 000000000000003f
    R10: ffff81091fdc4008  R11: 00000000af482de4  R12: ffff81118d6700c0
    R13: 0000000000402040  R14: 0000000000000000  R15: ffff81118d6700c0
    ORIG_RAX: fffffffffffffff10  CS: 0010  SS: 0018
#12 [ffff81011ffefea0] acpi_processor_idle_simple at ffffffff801a6b29
```

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

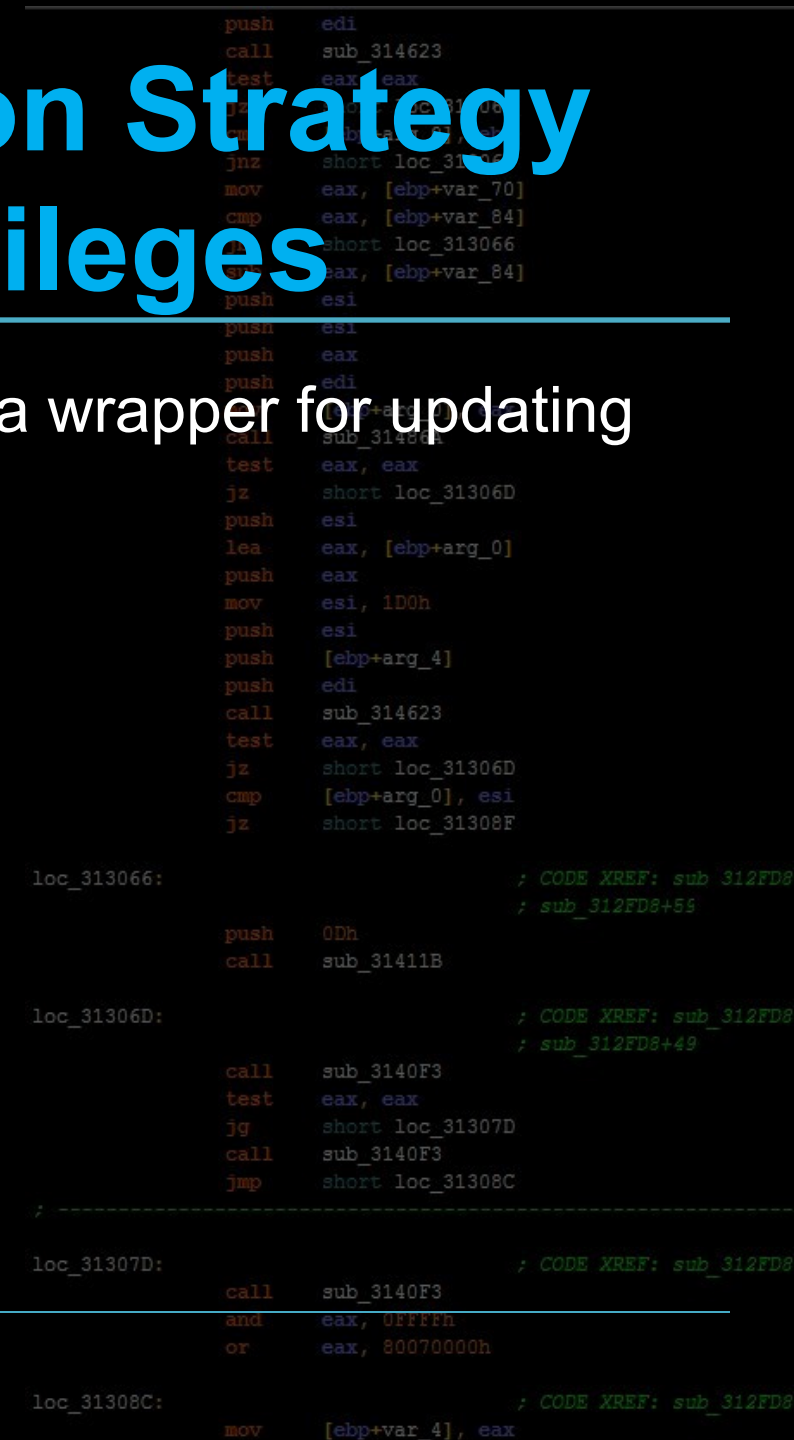
Therefore: The Kernel keeps track of permissions

```
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

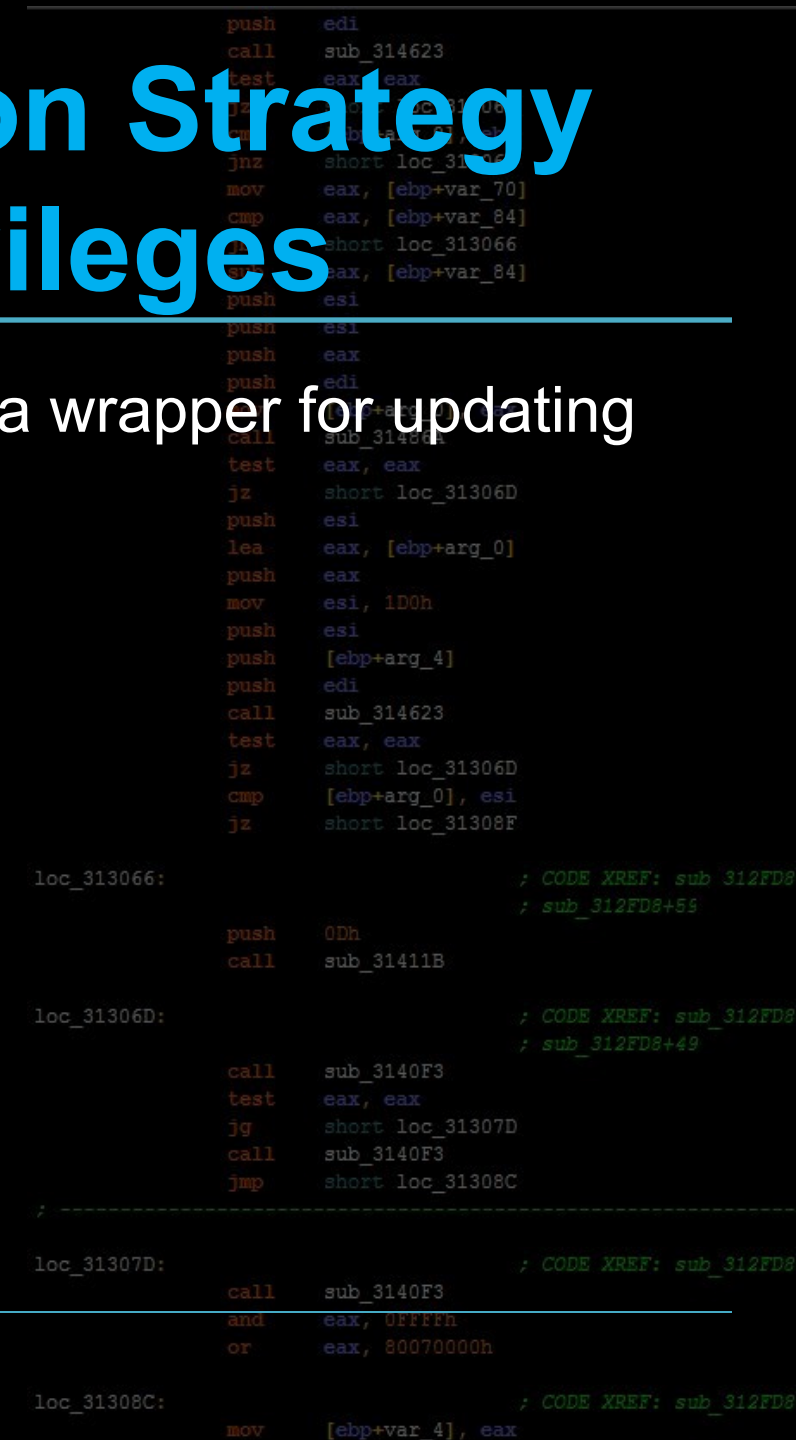
Conveniently, the Linux Kernel has a wrapper for updating process credentials!



Basic Exploitation Strategy Elevate Privileges

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

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

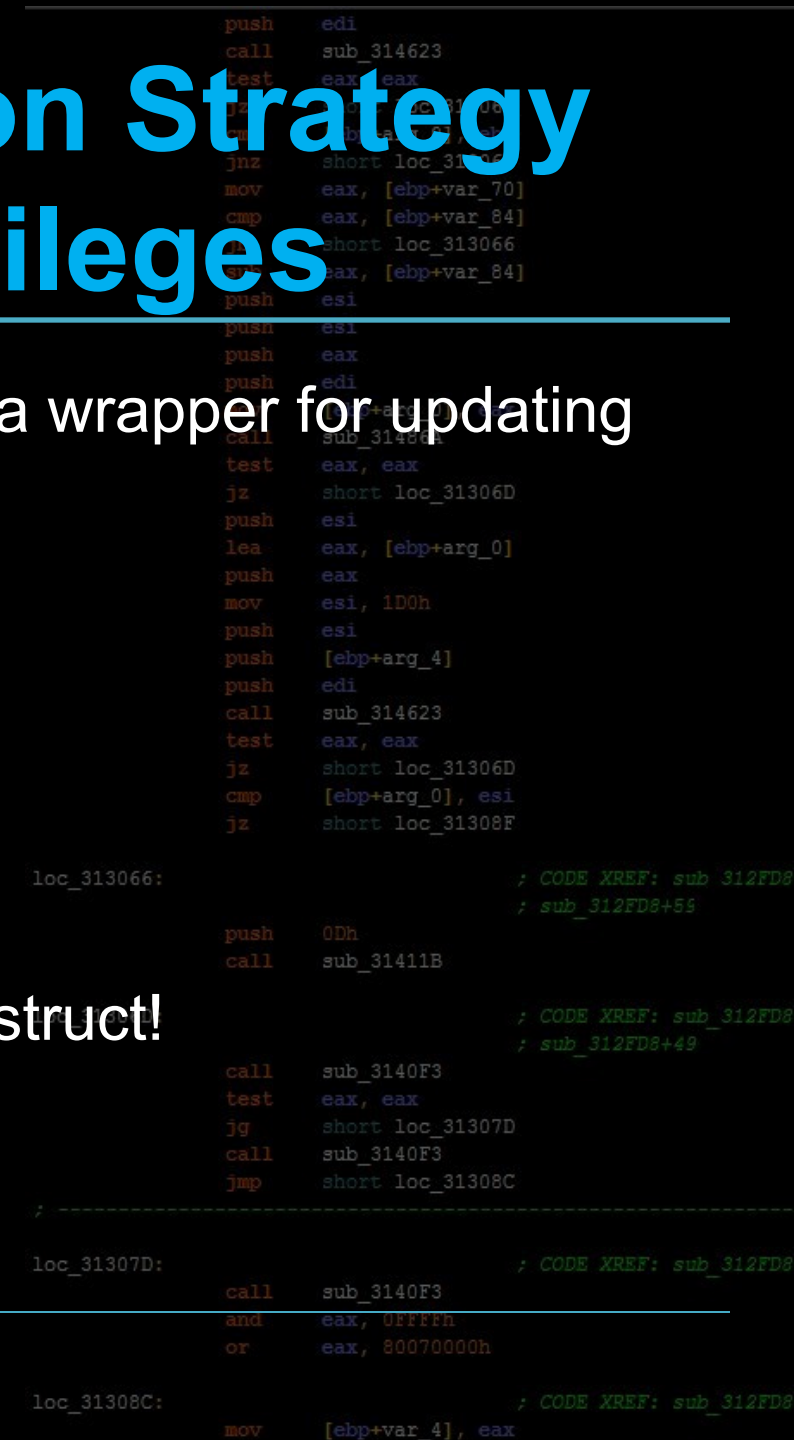


Basic Exploitation Strategy
Elevate Privileges

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

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

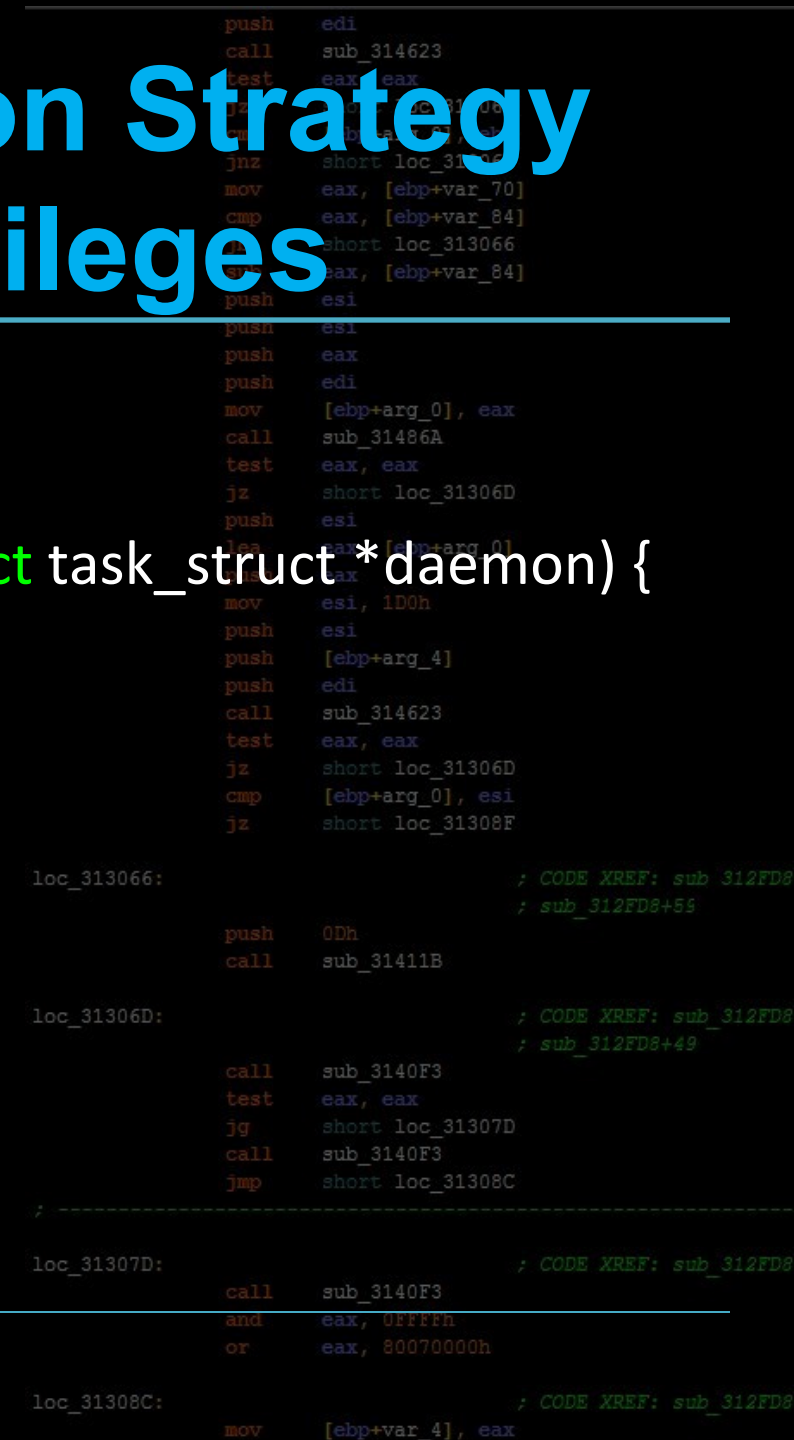
We just need to create a valid cred struct!



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!

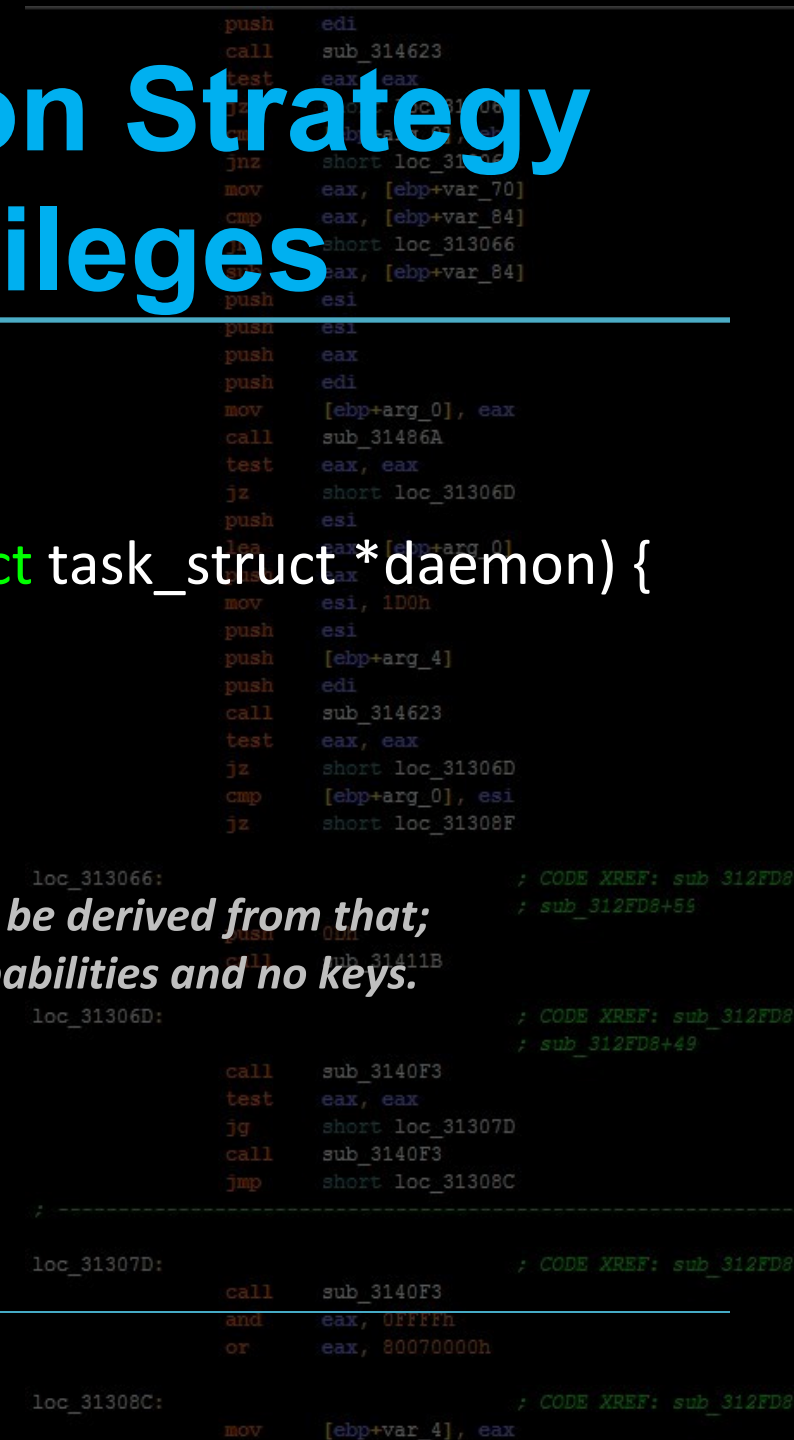
```
struct cred *prepare_kernel_cred(struct task_struct *daemon) {
        ...
}
```

“

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.

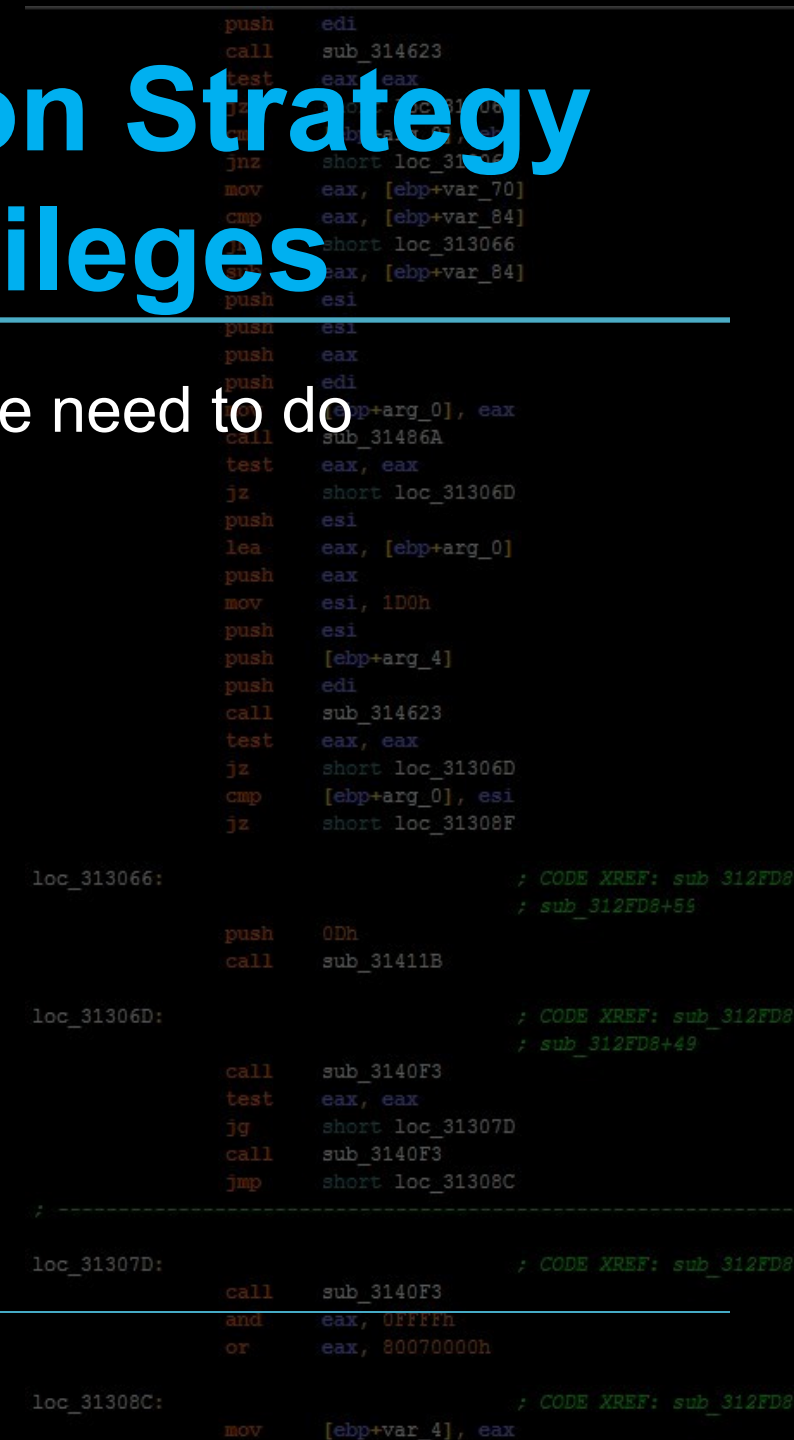
“

- *source/kernel/cred.c*



Basic Exploitation Strategy Elevate Privileges

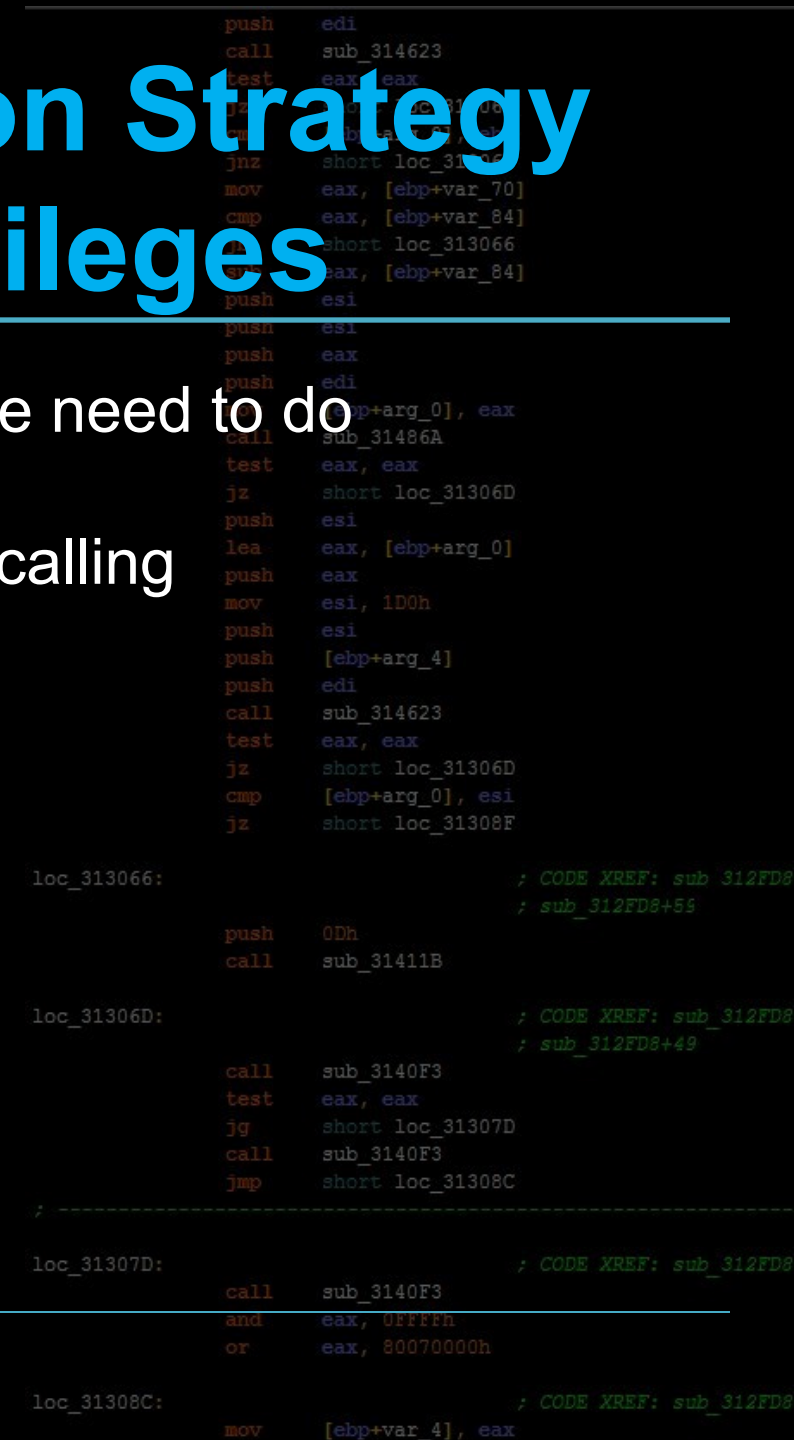
Great! Now we can map out what we need to do



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_cred(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!

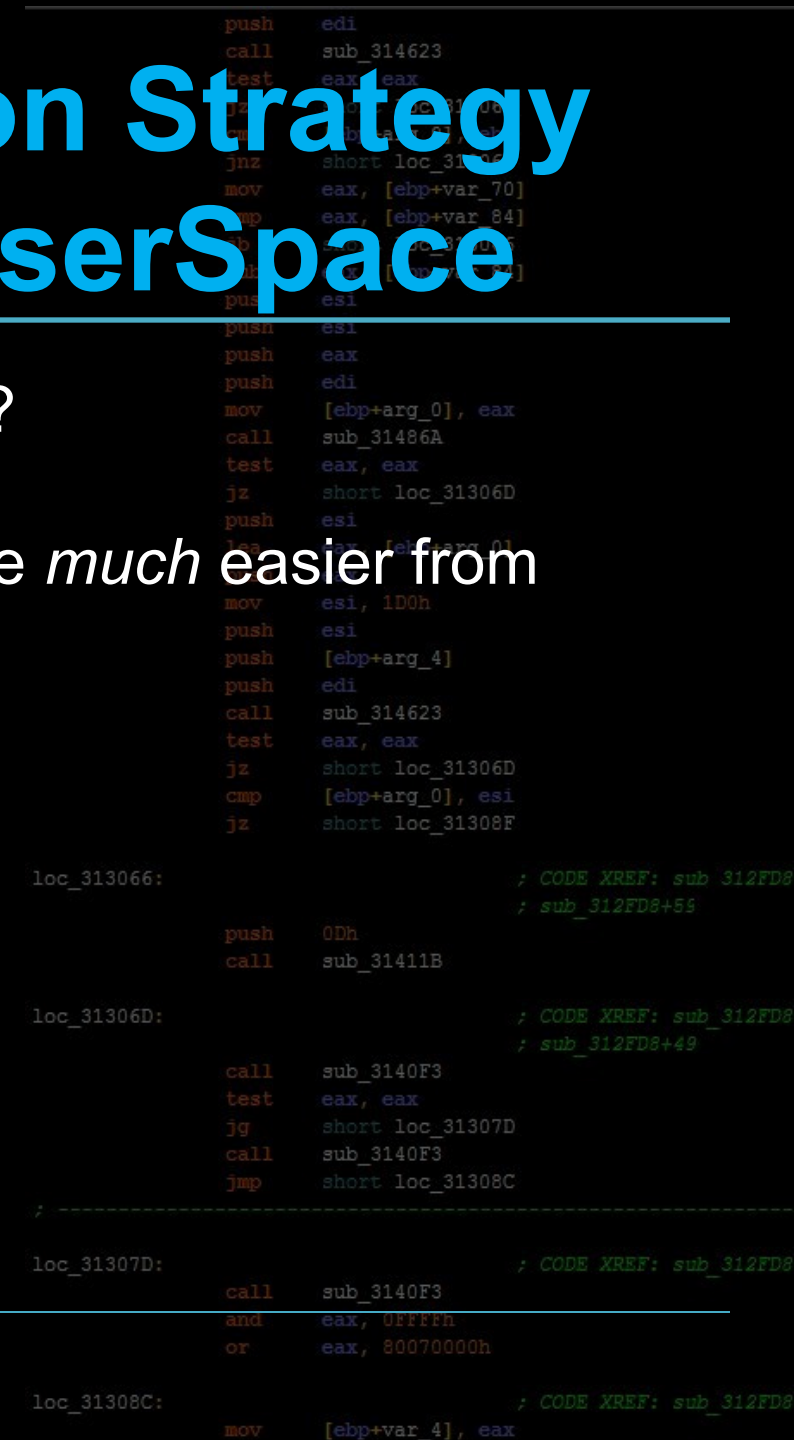
Basic Exploitation Strategy Returning To UserSpace

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.



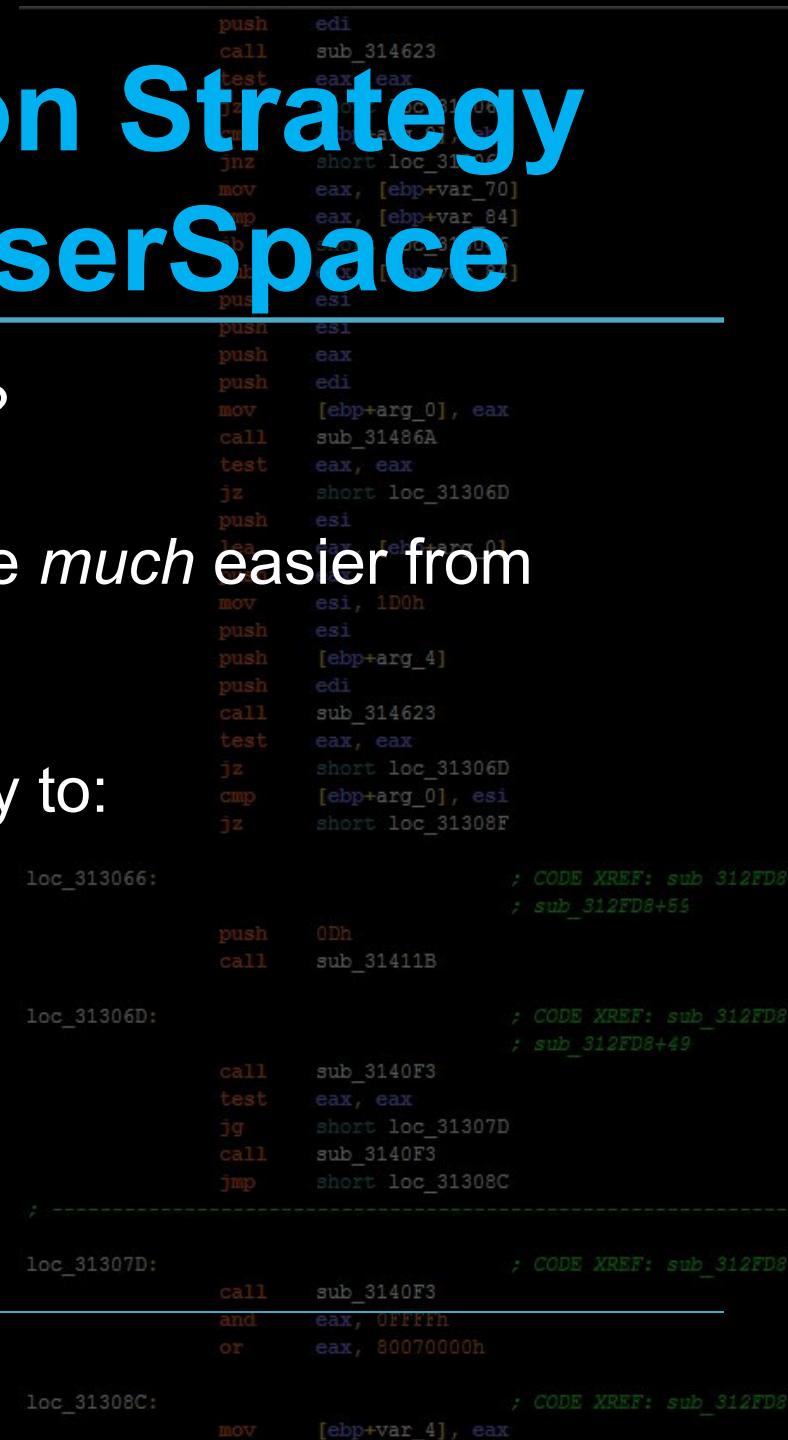
Basic Exploitation Strategy Returning To UserSpace

Why bother returning to Userspace?

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

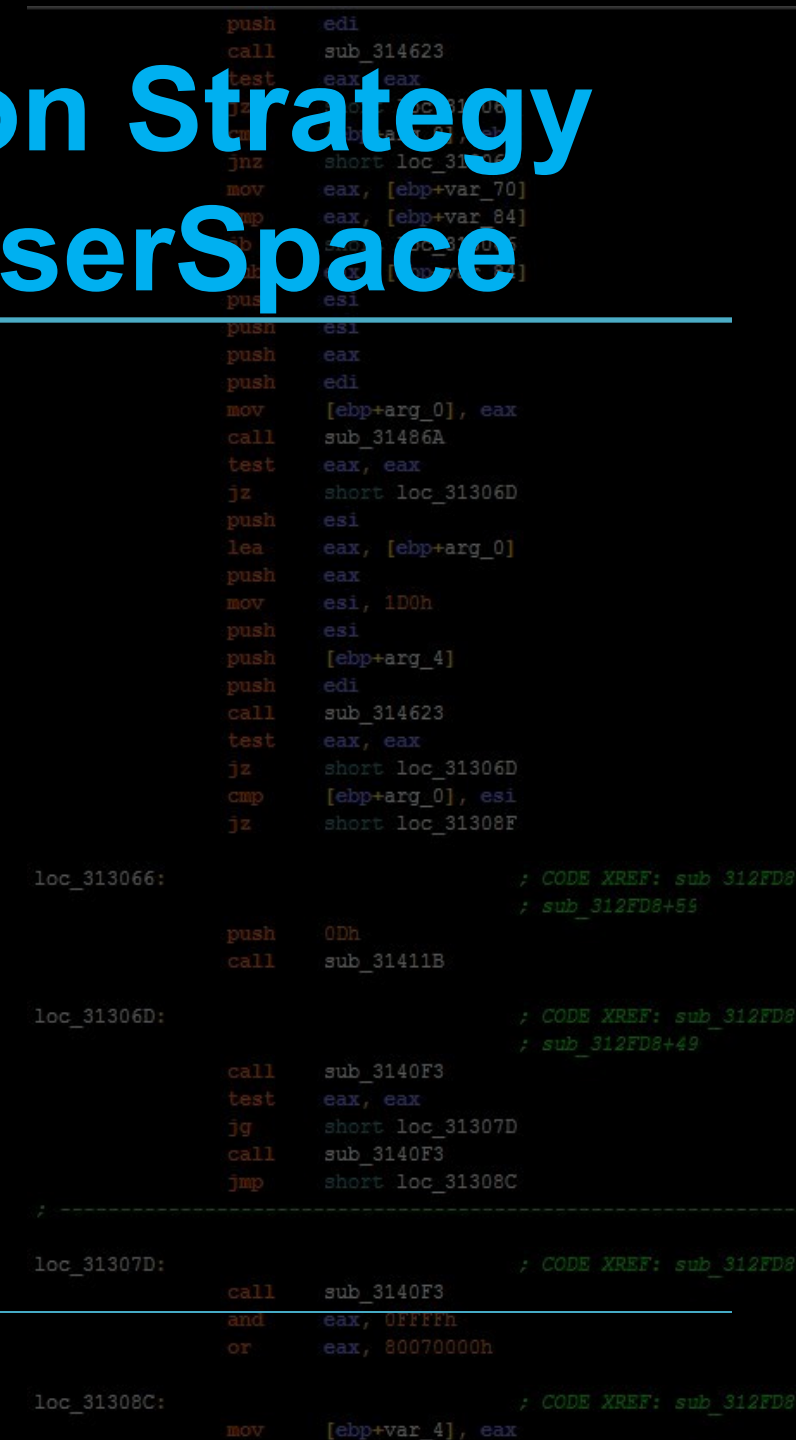
In KernelSpace, there's no easy way to:

> Modify the filesystem
> Create a new process
> Create network connections



Basic Exploitation Strategy Returning To UserSpace

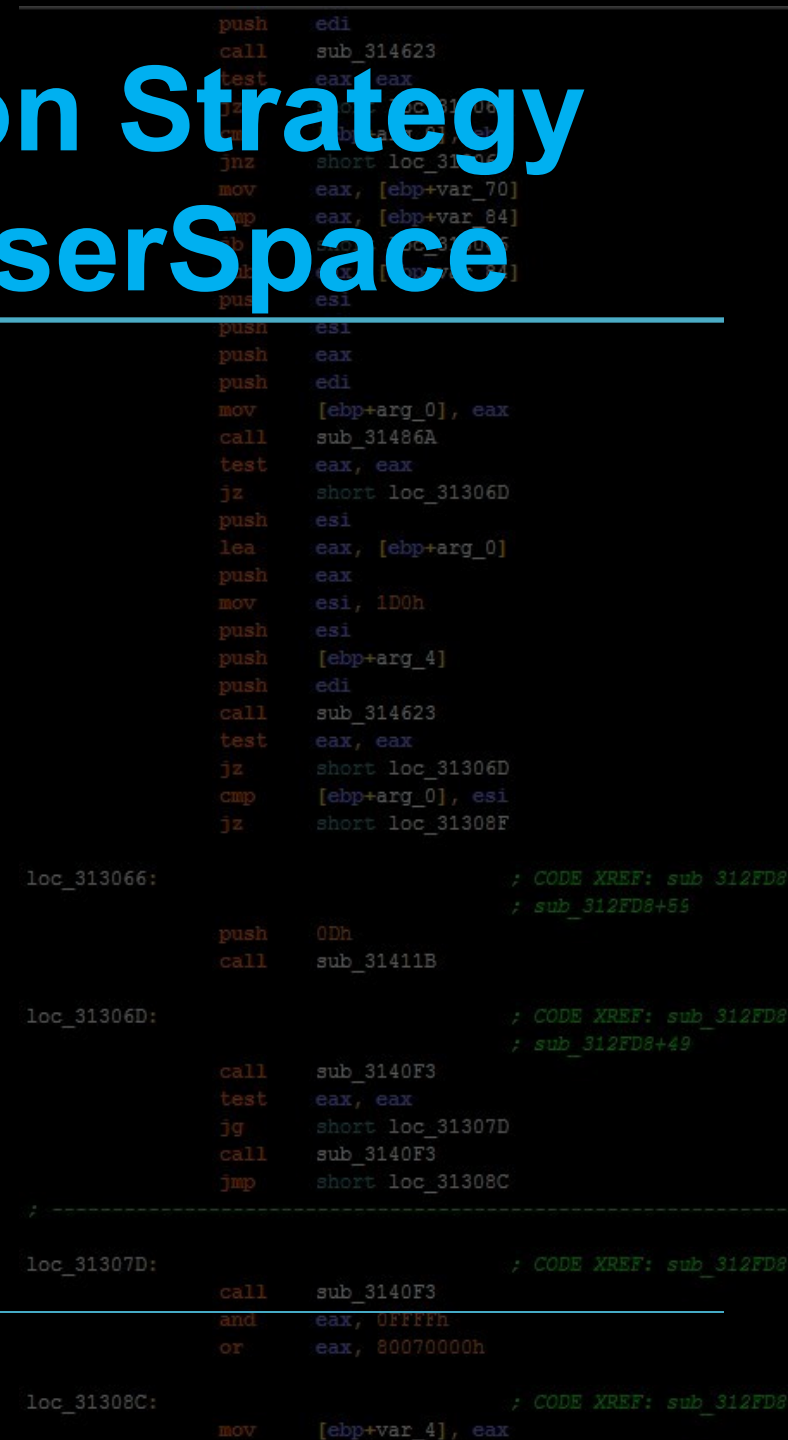
How does the kernel do it?



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

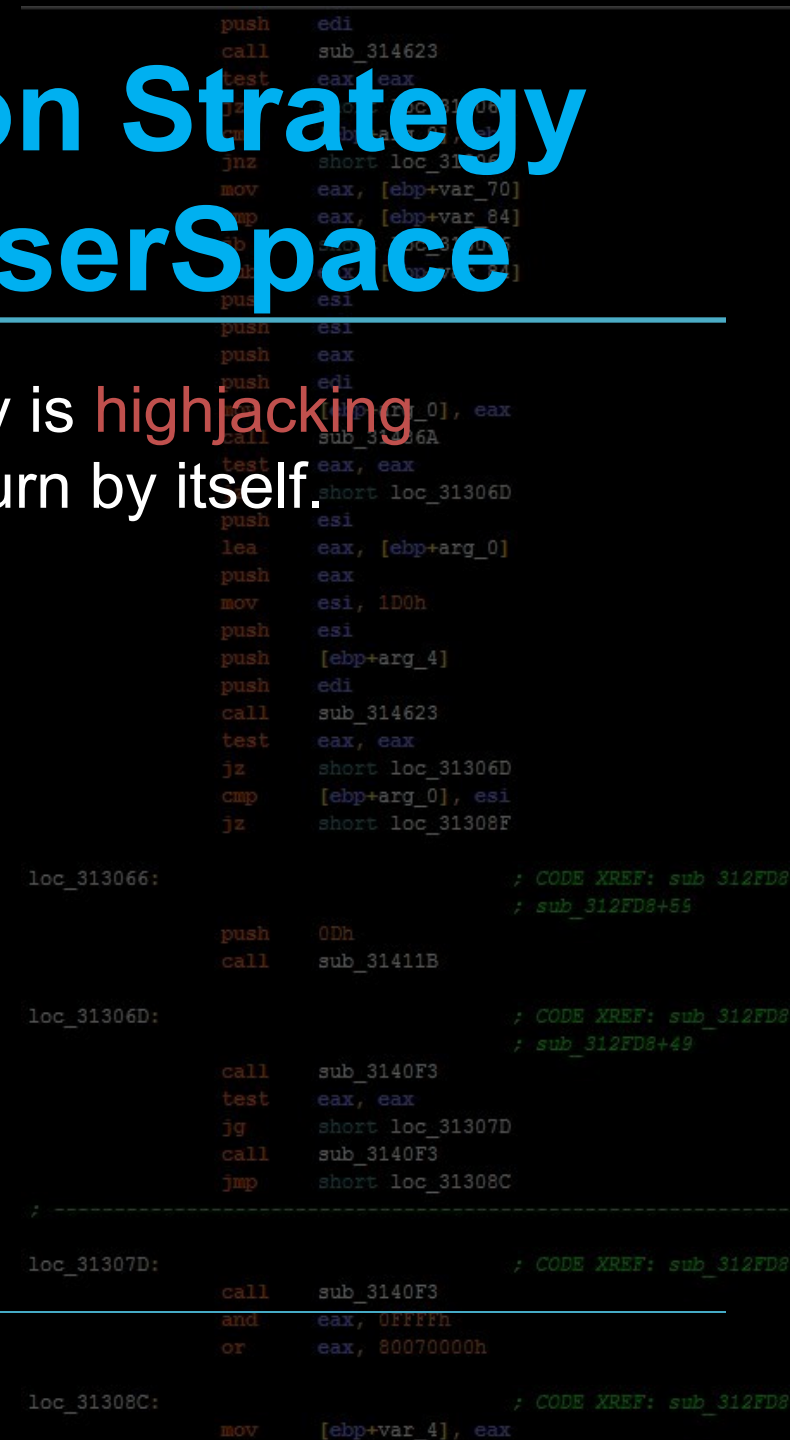
How does the kernel do it?

```
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

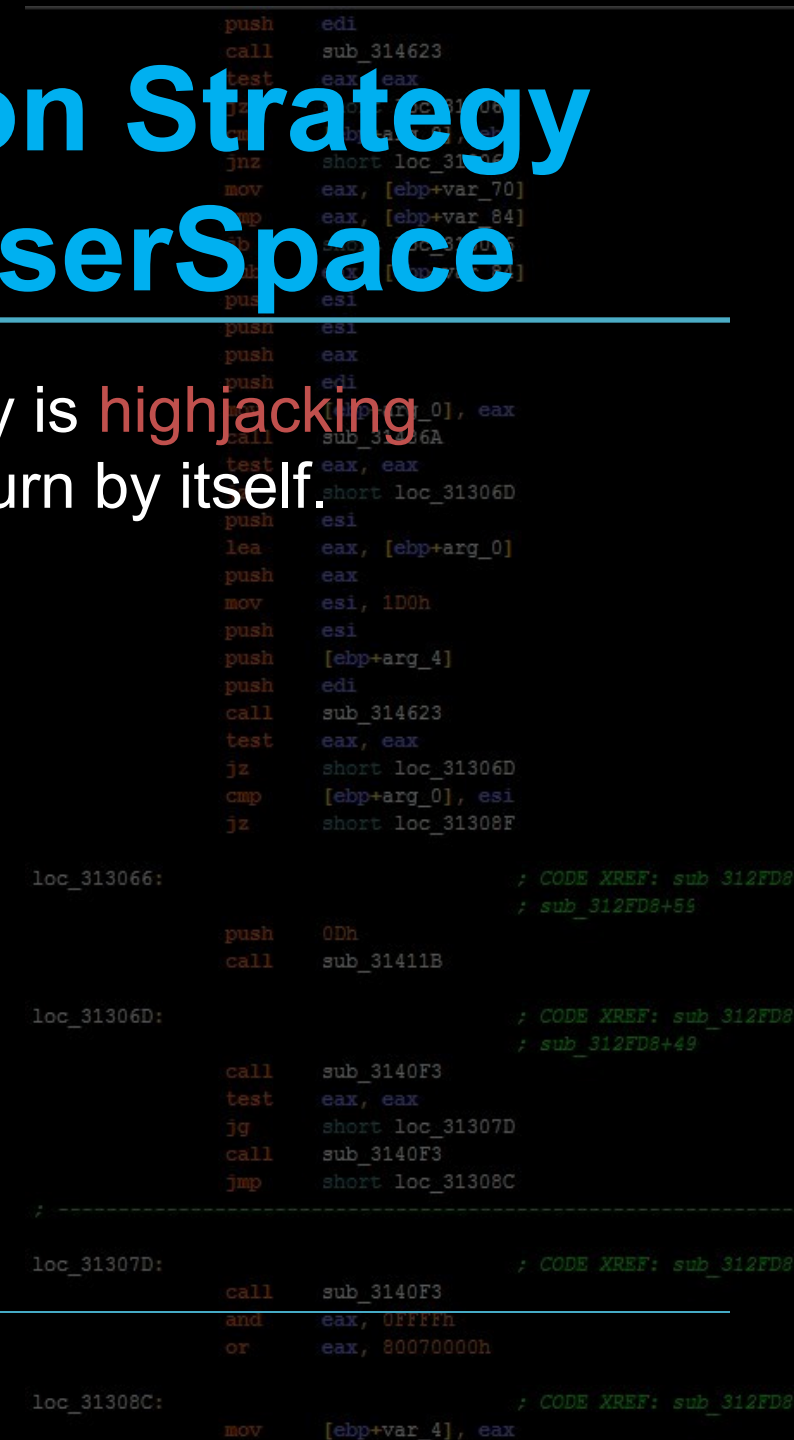
For exploitation, the easiest strategy is highjacking execution, and letting the kernel return by itself.



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



Basic Exploitation Strategy Returning To UserSpace

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

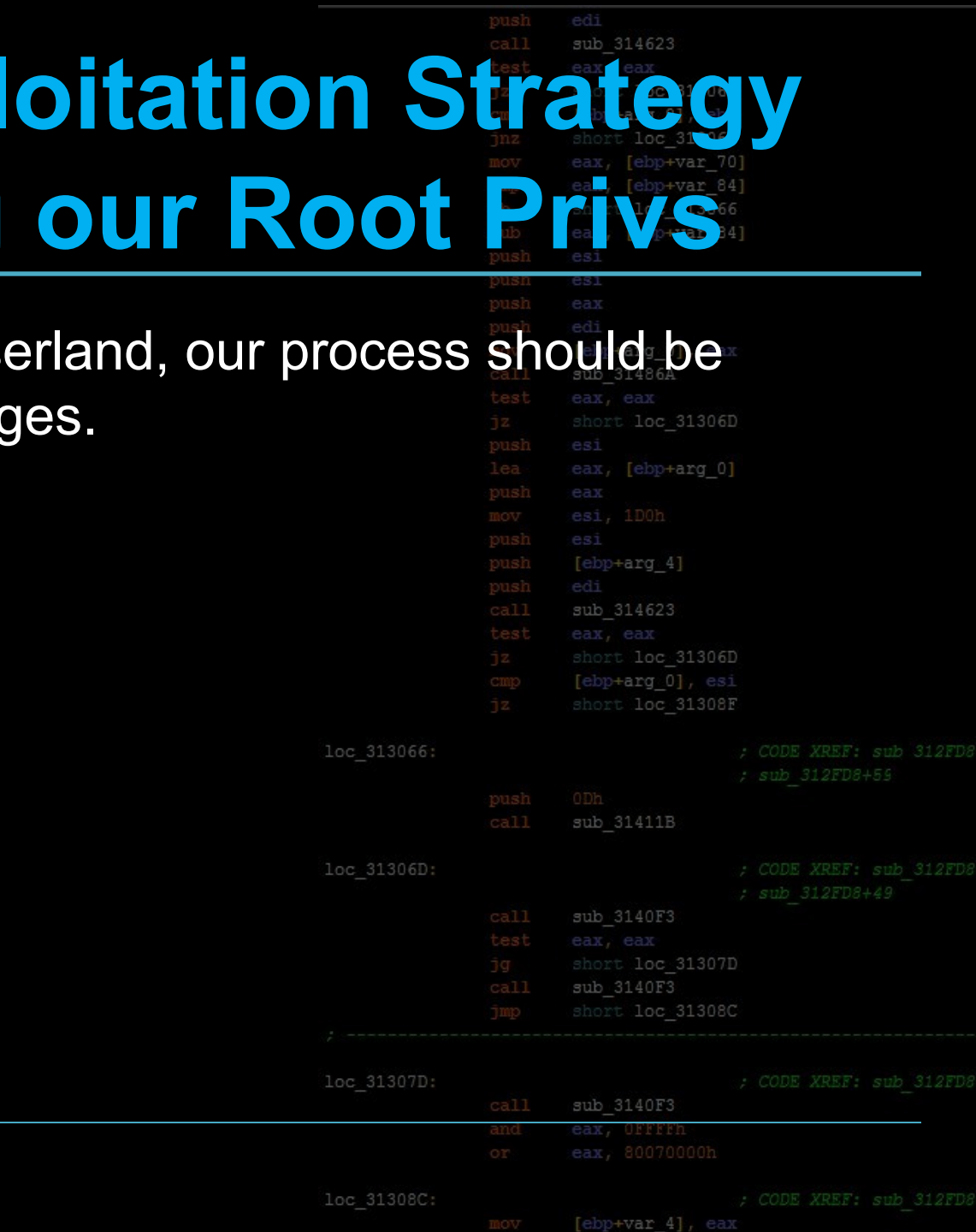
> Function Pointer Overwrites
> Syscall Table Highjacking
> Use-After-Free

You need to be very careful about destroying Kernel state.

A segfault probably means a reboot!

Basic Exploitation Strategy
Enjoying our Root Privs

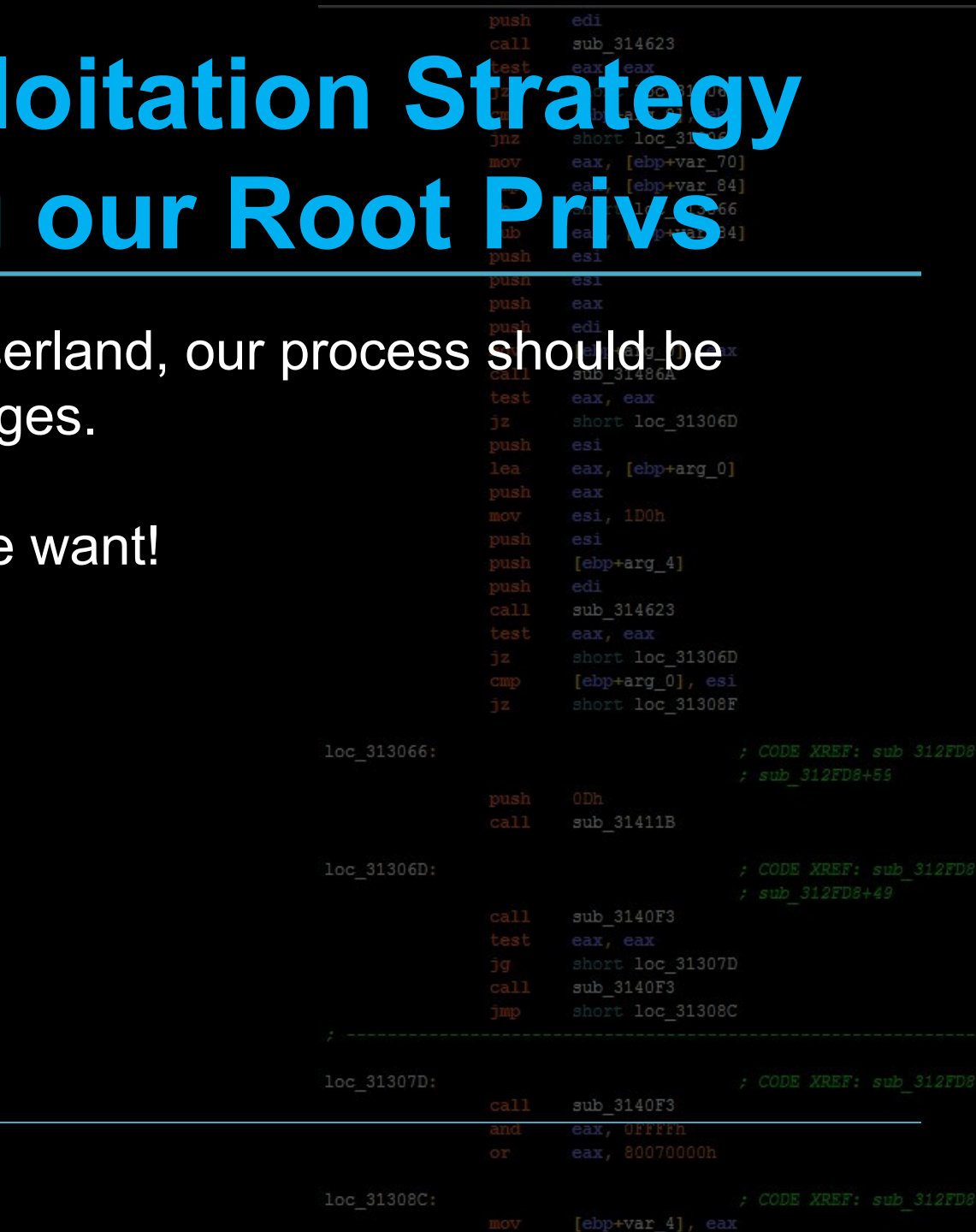
If we make it back to userland, our process should be running with root privileges.



Basic Exploitation Strategy
Enjoying our Root Privs

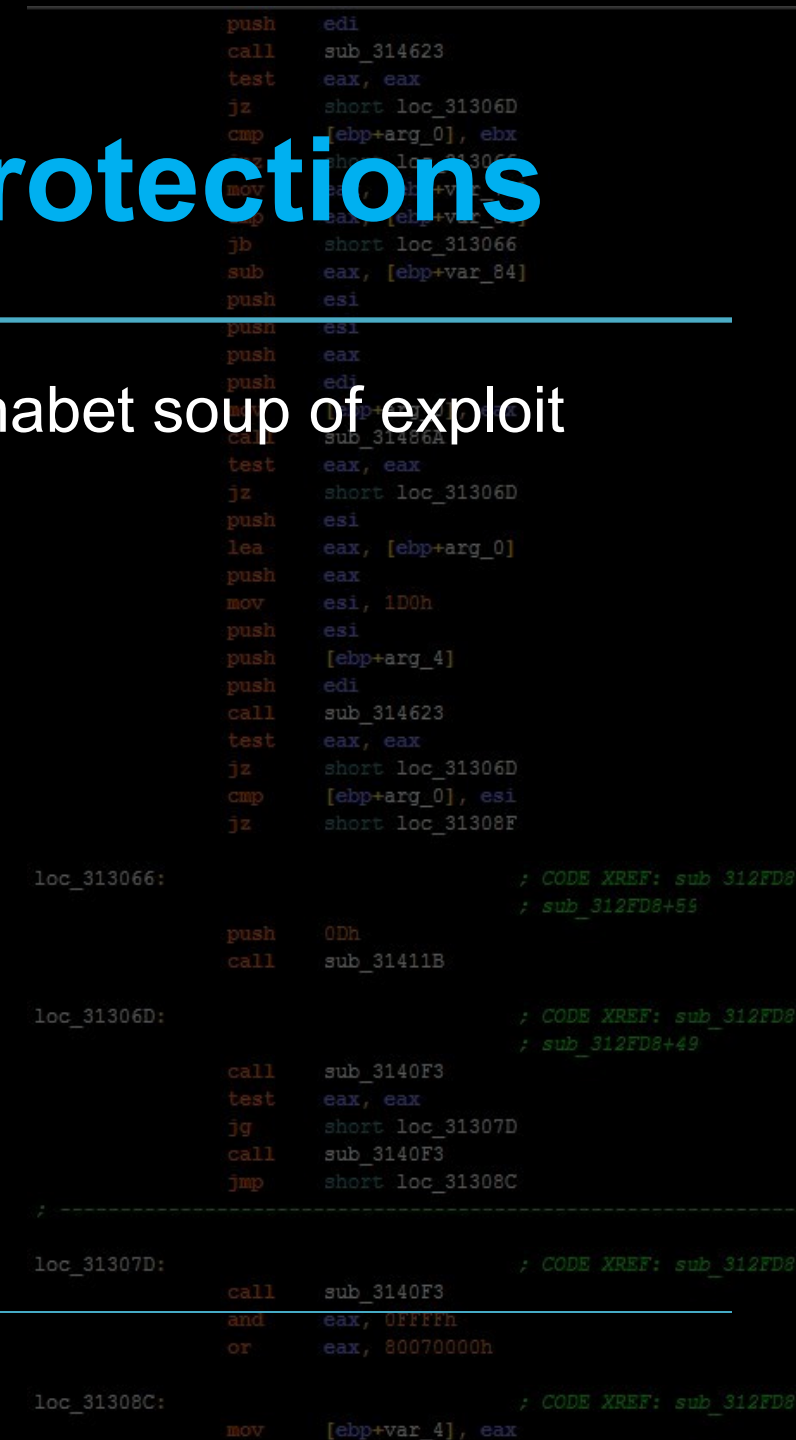
If we make it back to userland, our process should be running with root privileges.

We can do whatever we want!



Kernel Space Protections

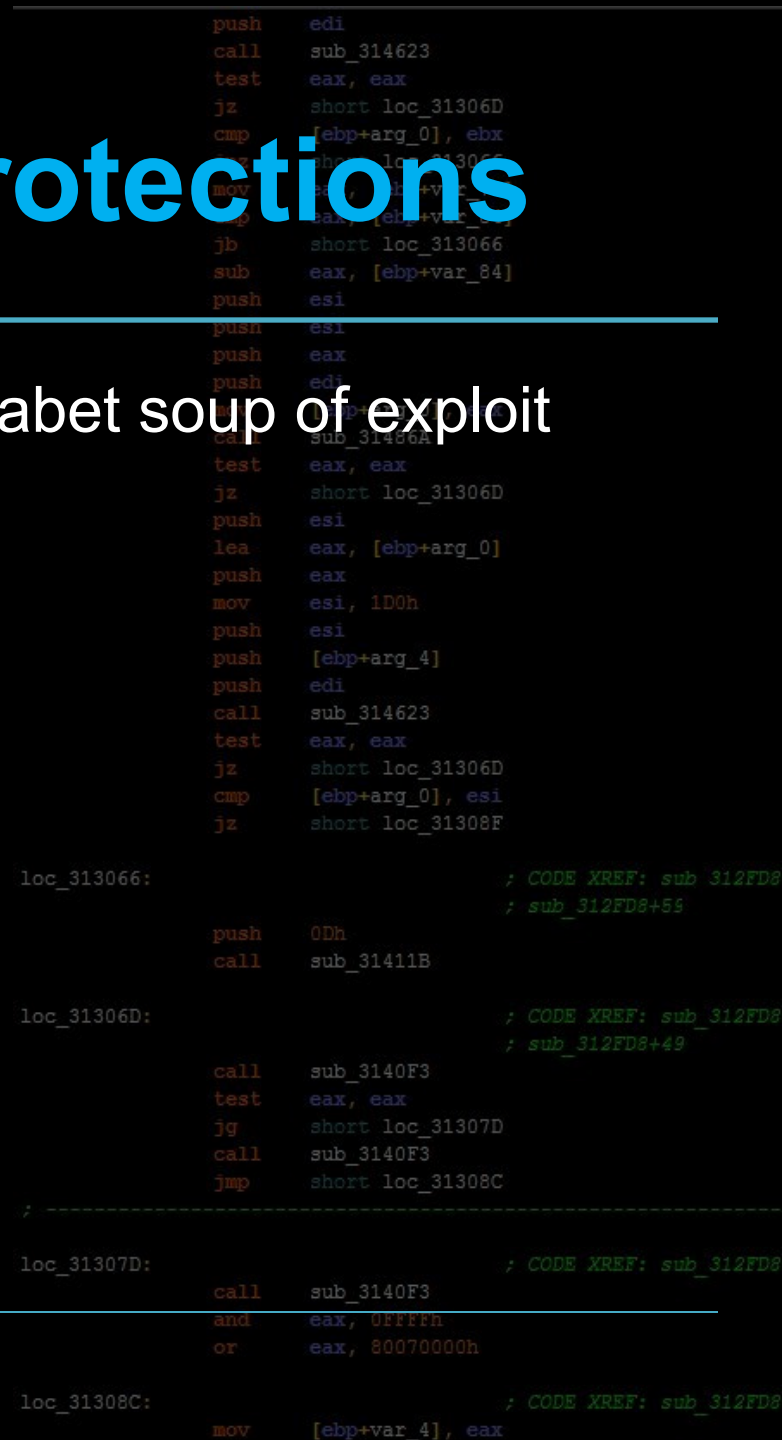
By now, you're familiar with the alphabet soup of exploit mitigations



Kernel Space Protections

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

DEP
ASLR
PIE
Canaries
RELRO
etc...

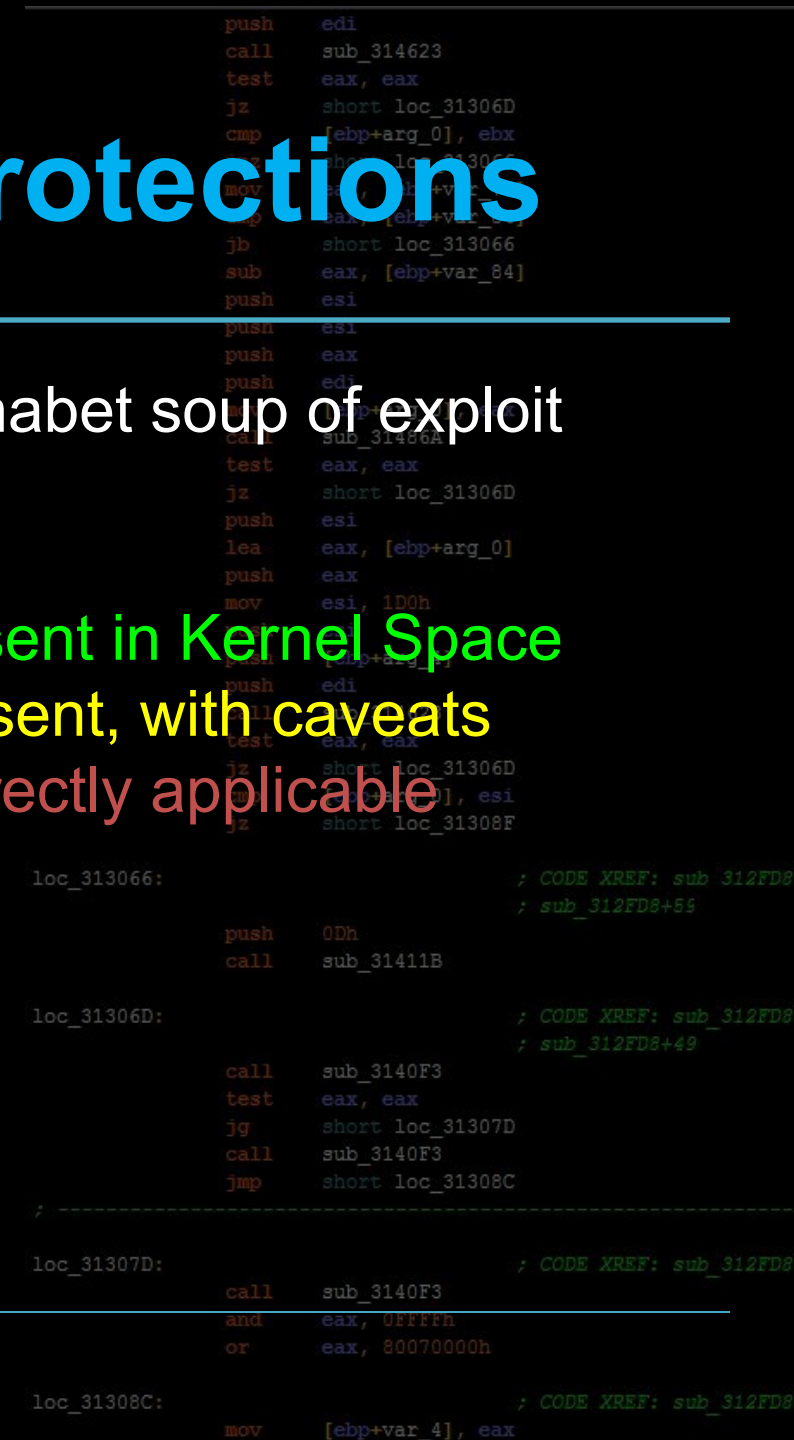


Kernel Space Protections

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

DEP
ASLR
PIE
Canaries
RELRO
etc...

Green: Present in Kernel Space
Yellow: Present, with caveats
Red: Not directly applicable



Kernel Space Protections

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

DEP
ASLR
PIE
Canaries
RELRO
etc...

Green: Present in Kernel Space
Yellow: Present, with caveats
Red: Not directly applicable

There's a whole new alphabet soup for Kernel Mitigations!

Kernel Space Protections

Some new words in our soup

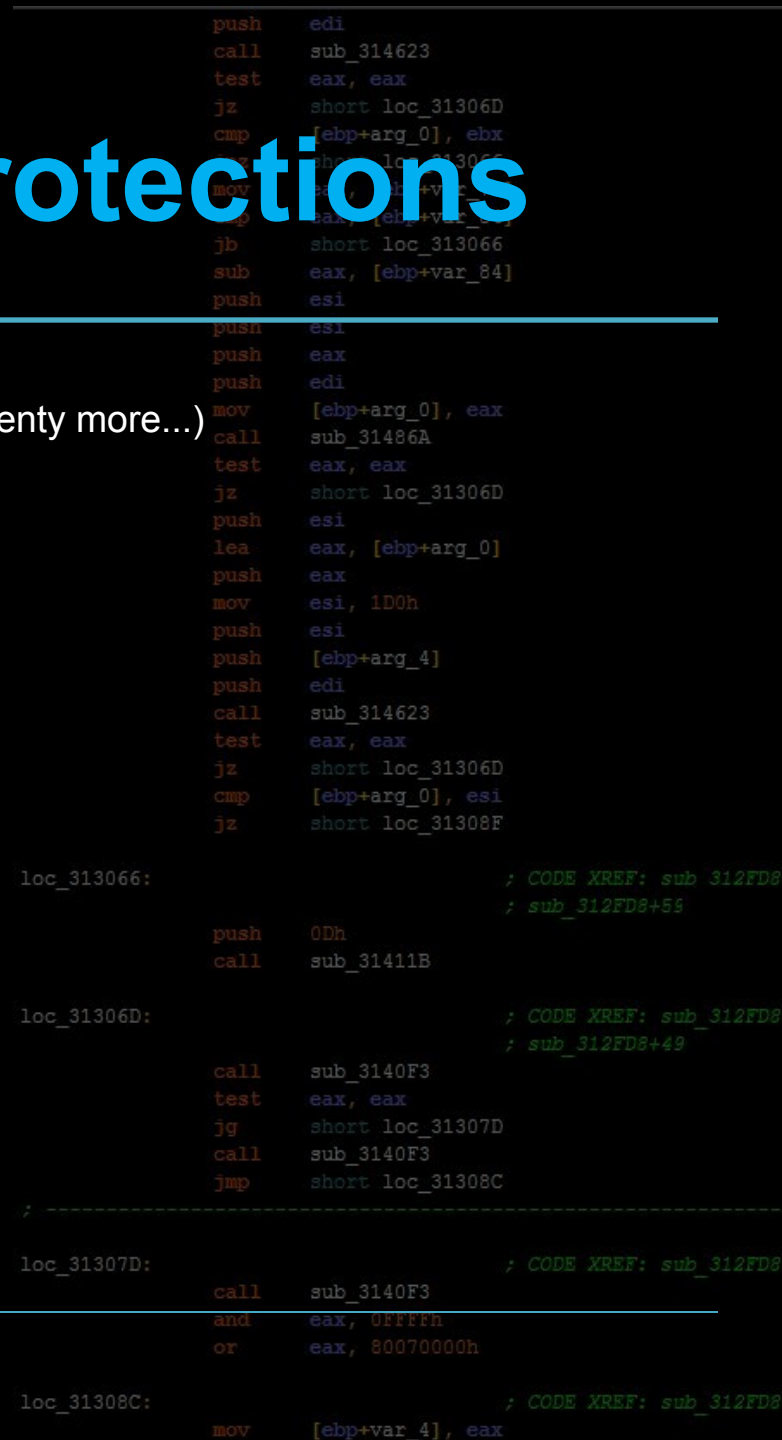
MMAP_MIN_ADDR
KALLSYMS
RANDSTACK
STACKLEAK
SMEP / SMAP

Kernel Space Protections

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

MMAP_MIN_ADDR
KALLSYMS
RANDSTACK
STACKLEAK
SMEP / SMAP

Most of these will be off for the labs!

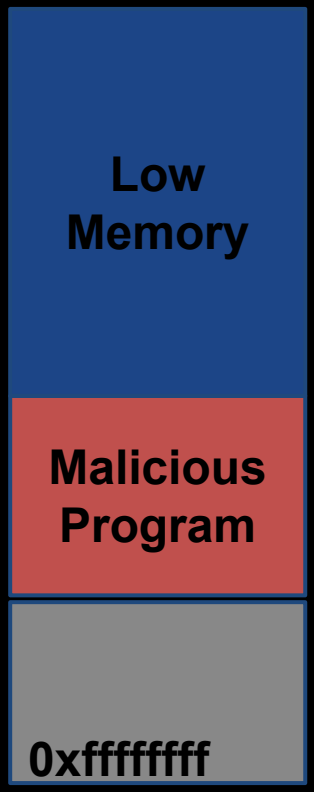


Kernel Space Protections mmap_min_addr

This makes exploiting NULL pointer dereferences harder.

Kernel Space Protections mmap_min_addr

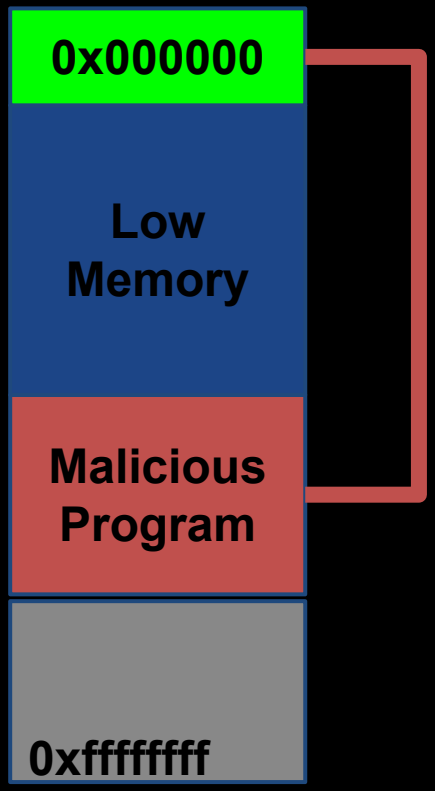
This makes exploiting NULL pointer dereferences harder.




```
push    edi
call    sub_314623
test    eax, eax
jnz     short loc_313066
mov     eax, [ebp+var_70]
cmp     eax, [ebp+var_84]
jz      short loc_313066
mov     eax, [ebp+var_84]
push    esi
push    esi
push    eax
push    edi
call    sub_31486A
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
jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+55
push    0Dh
call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
call    sub_3140F3
test    eax, eax
jg      short loc_31307D
call    sub_3140F3
jmp     short loc_31308C
; ---------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
call    sub_3140F3
and     eax, 0FFFFh
or      eax, 80070000h

loc_31308C:                             ; CODE XREF: sub_312FD8
mov     [ebp+var_4], eax
```

Kernel Space Protections mmap_min_addr

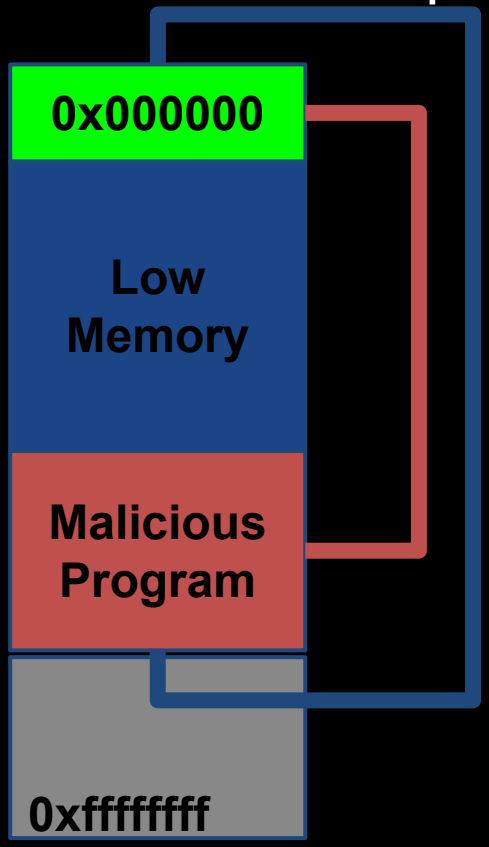
This makes exploiting NULL pointer dereferences harder.



Program does mmap(0,....)

Kernel Space Protections mmap_min_addr

This makes exploiting NULL pointer dereferences harder.

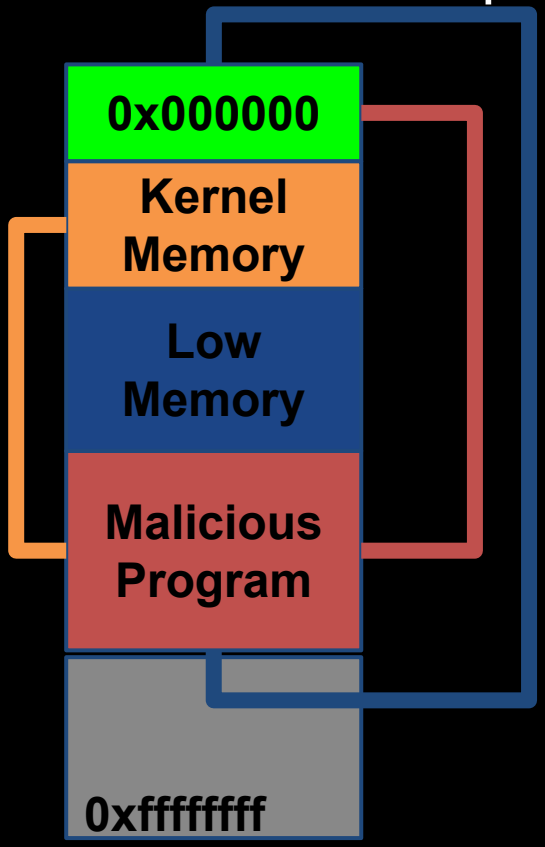


Program does mmap(0,....)

Program writes malicious Code

Kernel Space Protections mmap_min_addr

This makes exploiting NULL pointer dereferences harder.



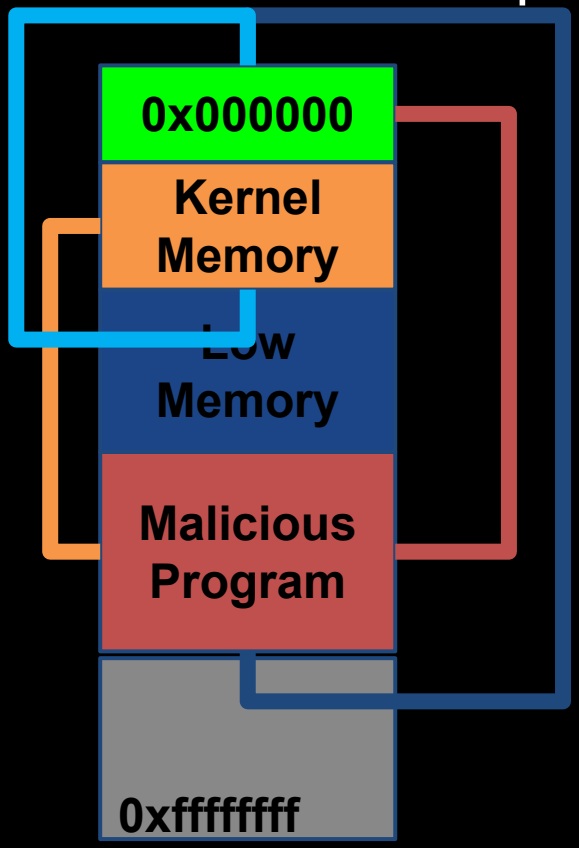
Program does mmap(0,....)

Program writes malicious Code

Program triggers Kernel Bug

Kernel Space Protections mmap_min_addr

This makes exploiting NULL pointer dereferences harder.



Program does mmap(0,....)

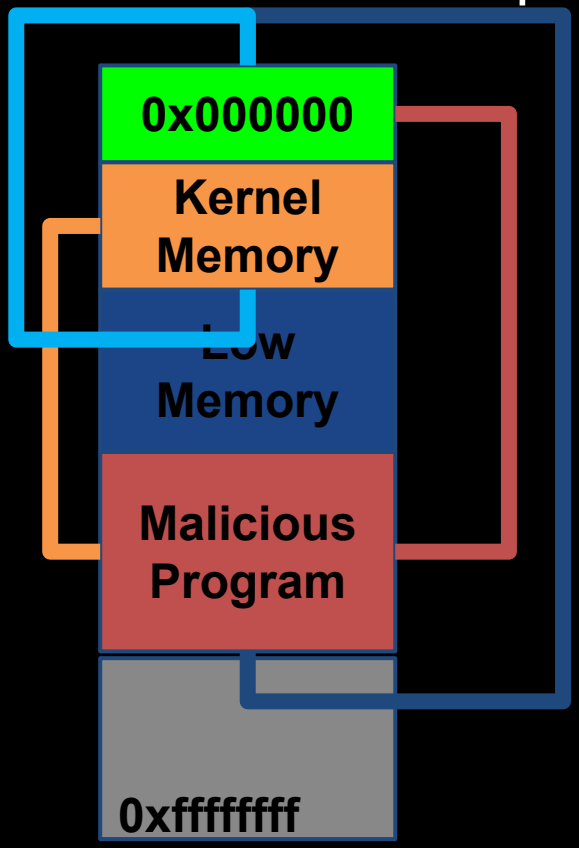
Program writes malicious Code

Program triggers Kernel Bug

Kernel starts executing malicious Code

Kernel Space Protections mmap_min_addr

This makes exploiting NULL pointer dereferences harder.



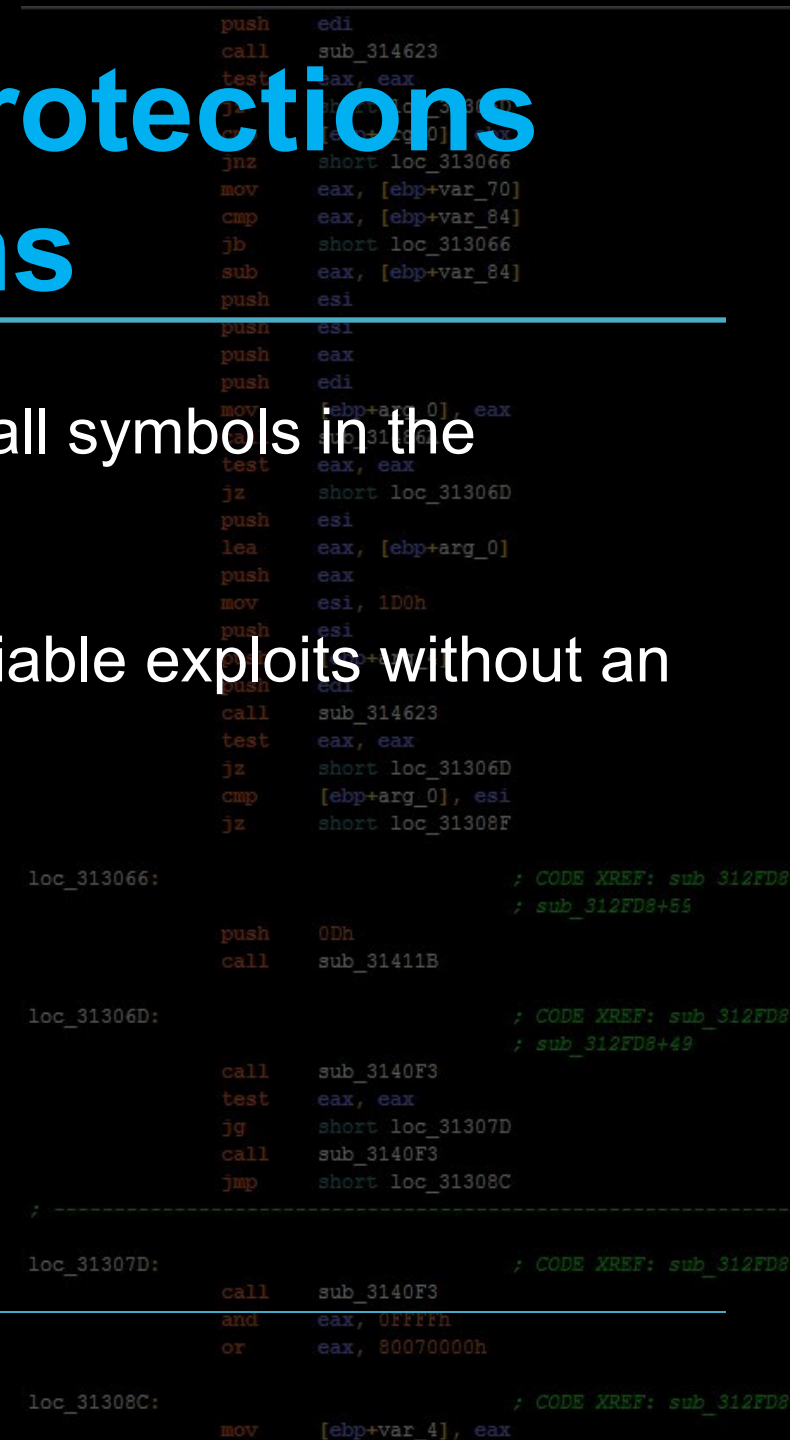
mmap_min_addr disallows programs from allocating low memory.

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!



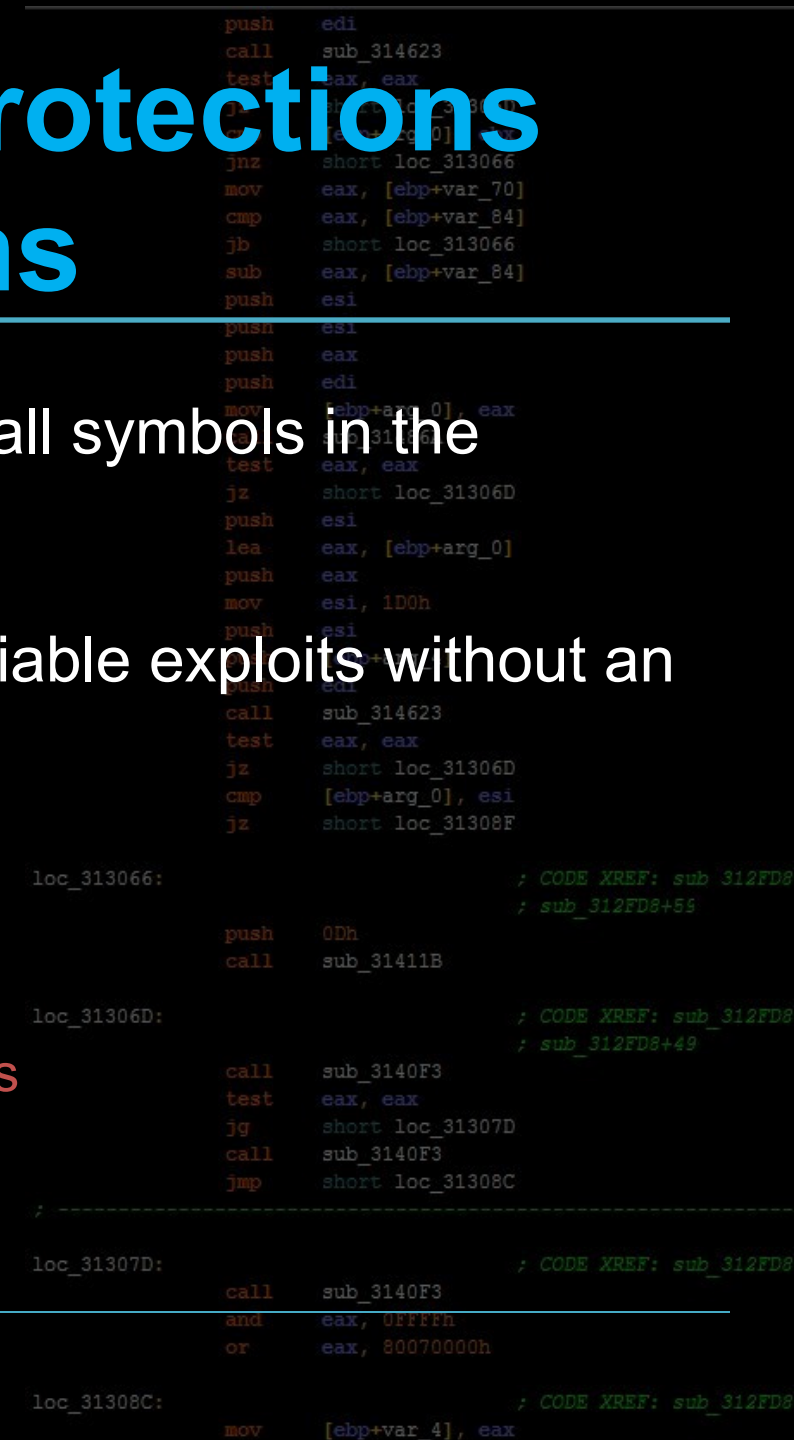
Kernel Space Protections kallsyms

/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

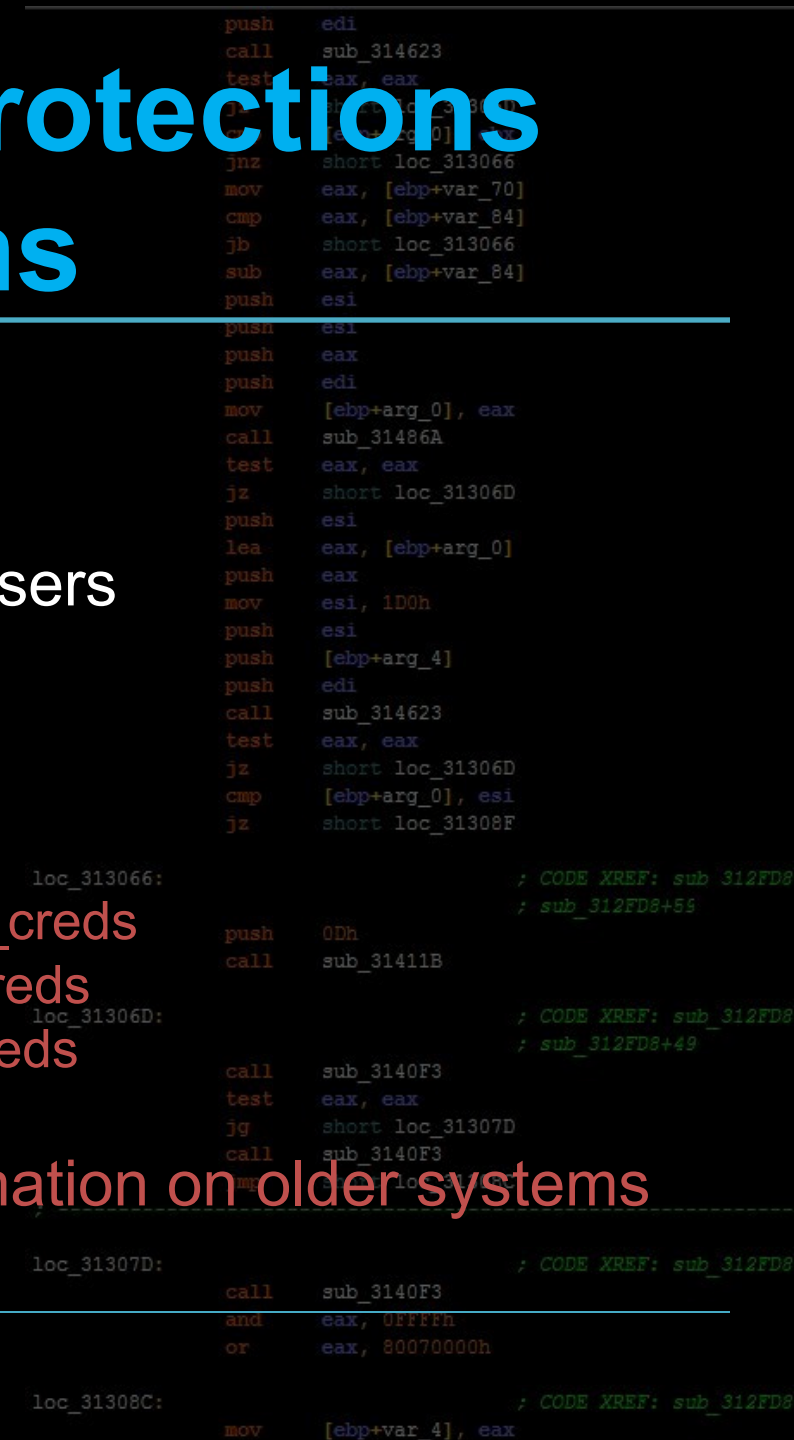
kallsyms used to be world-readable.

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

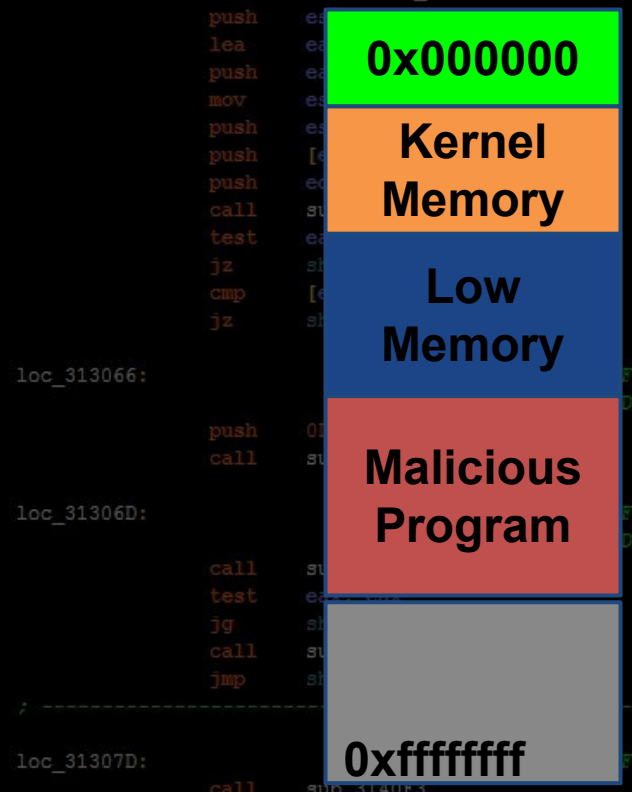
Introduced in Intel IvyBridge

SMAP: Supervisor Mode Access Protection

Introduced in Intel Haswell

Kernel Space Protections
SMEP / SMAP

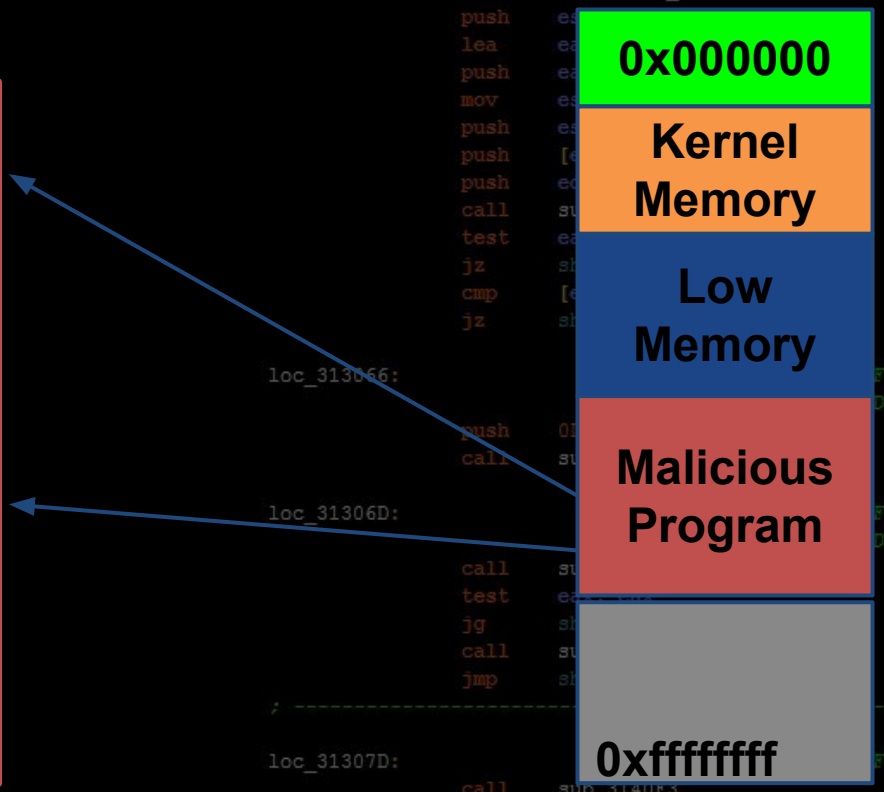
Common Exploitation Technique: Supply your own “get root” code.



Kernel Space Protections
SMEP / SMAP

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

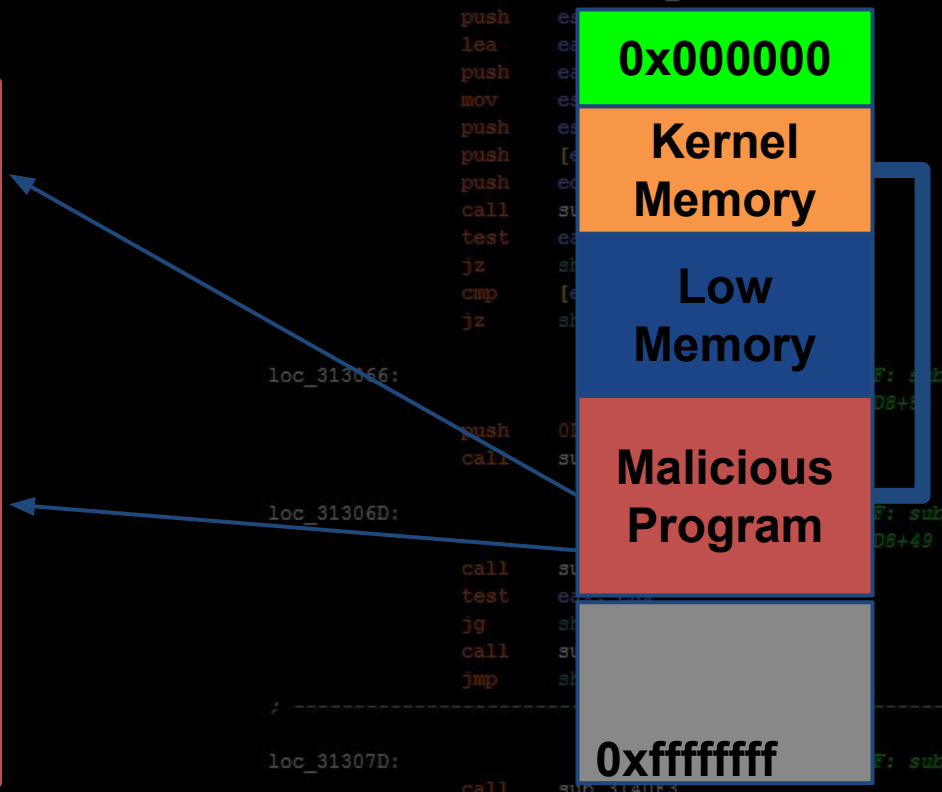
```
void get_r00t() {
        commit_creds(prepare_kernel_cred(0));
}

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.

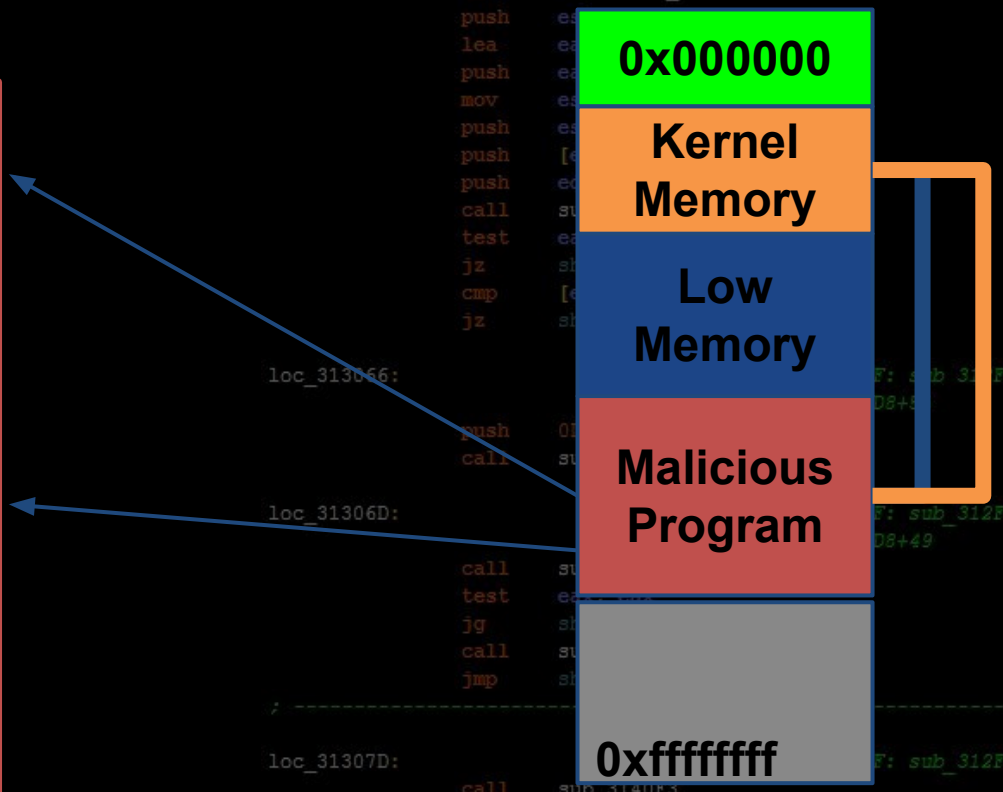
```
void get_r00t() {
        commit_creds(prepare_kernel_cred(0));
}

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.

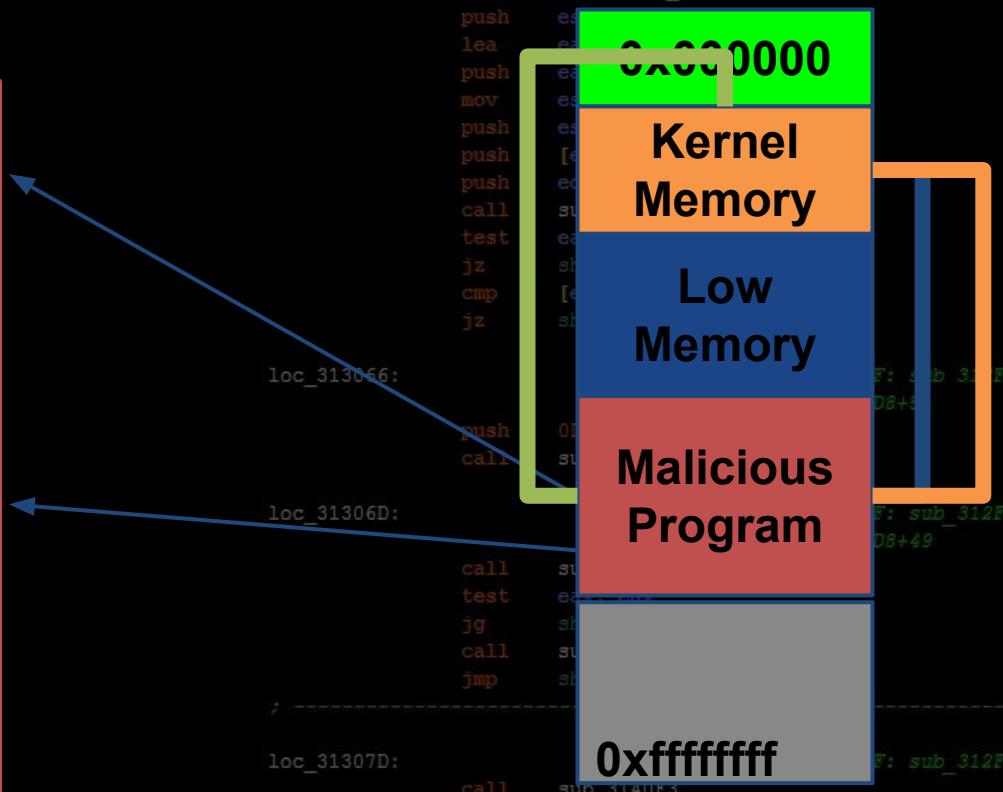
```
void get_r00t() {
        commit_creds(prepare_kernel_cred(0));
}

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.

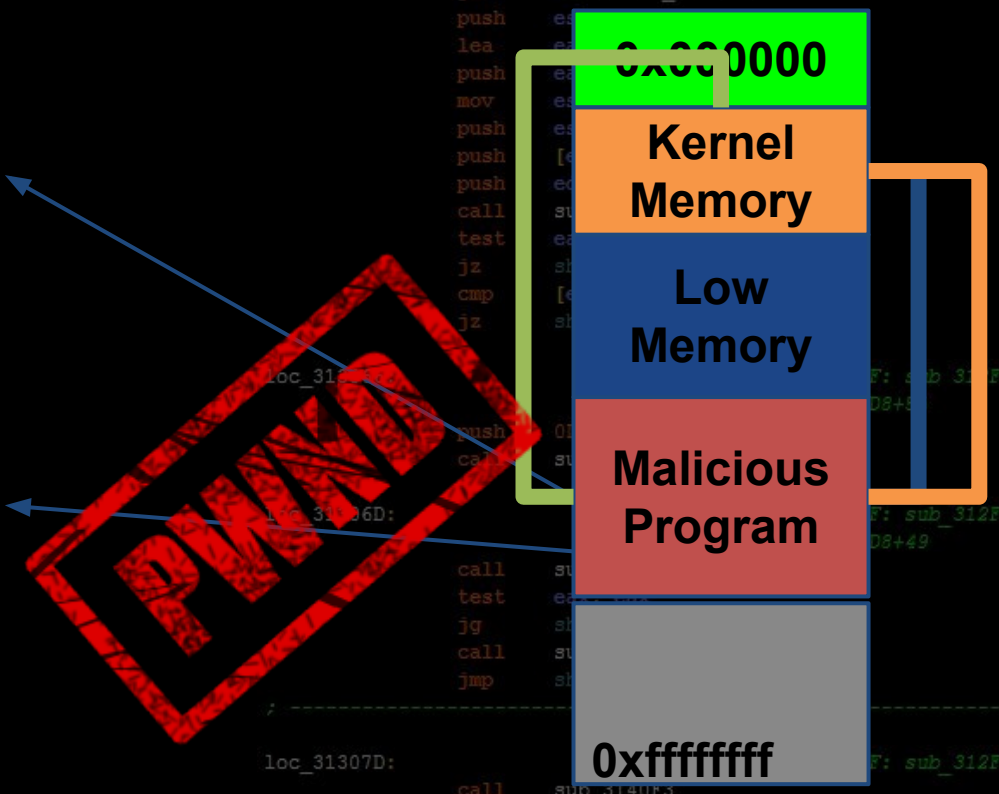
```
void get_r00t() {
        commit_creds(prepare_kernel_cred(0));
}

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

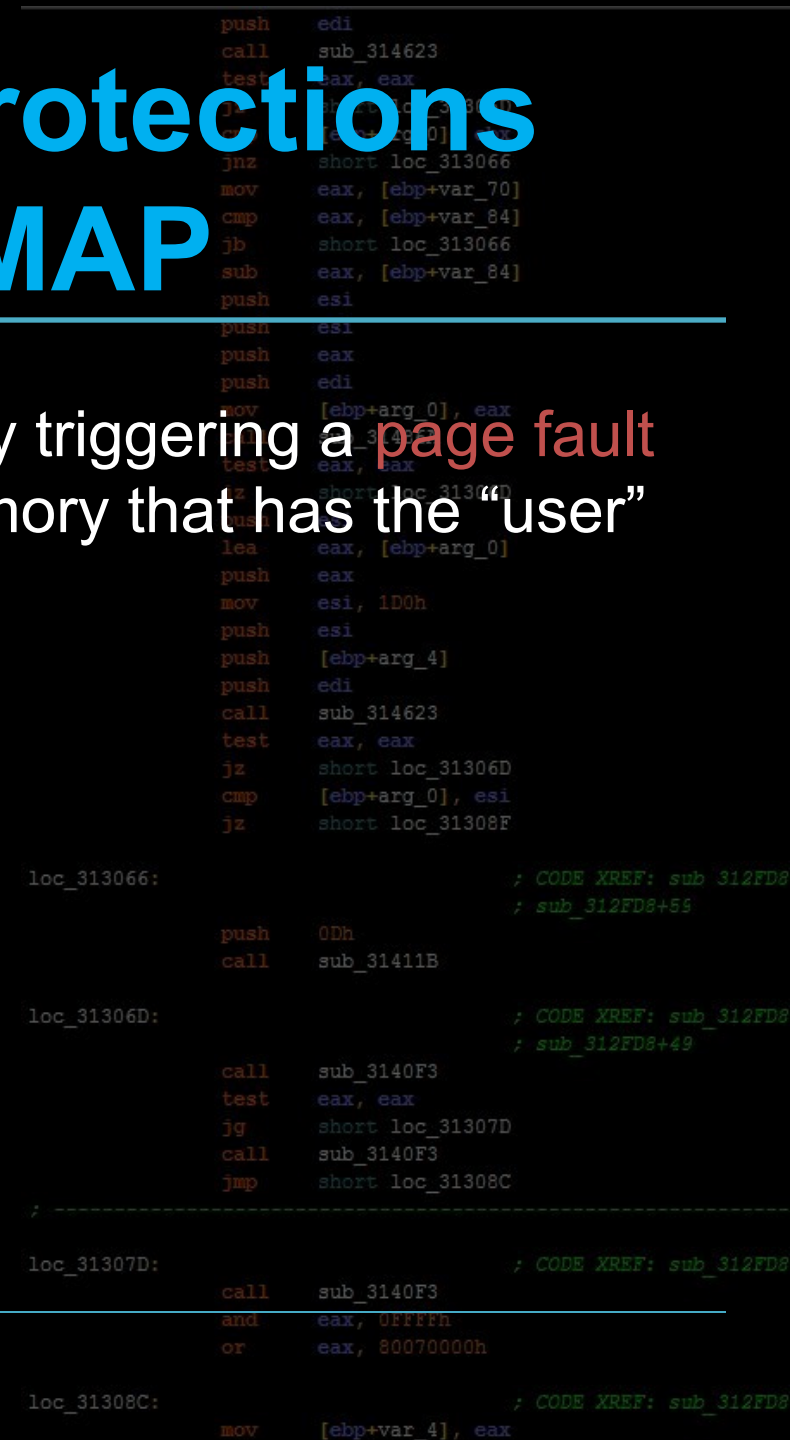
Common Exploitation Technique: Supply your own “get root” code.

```
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

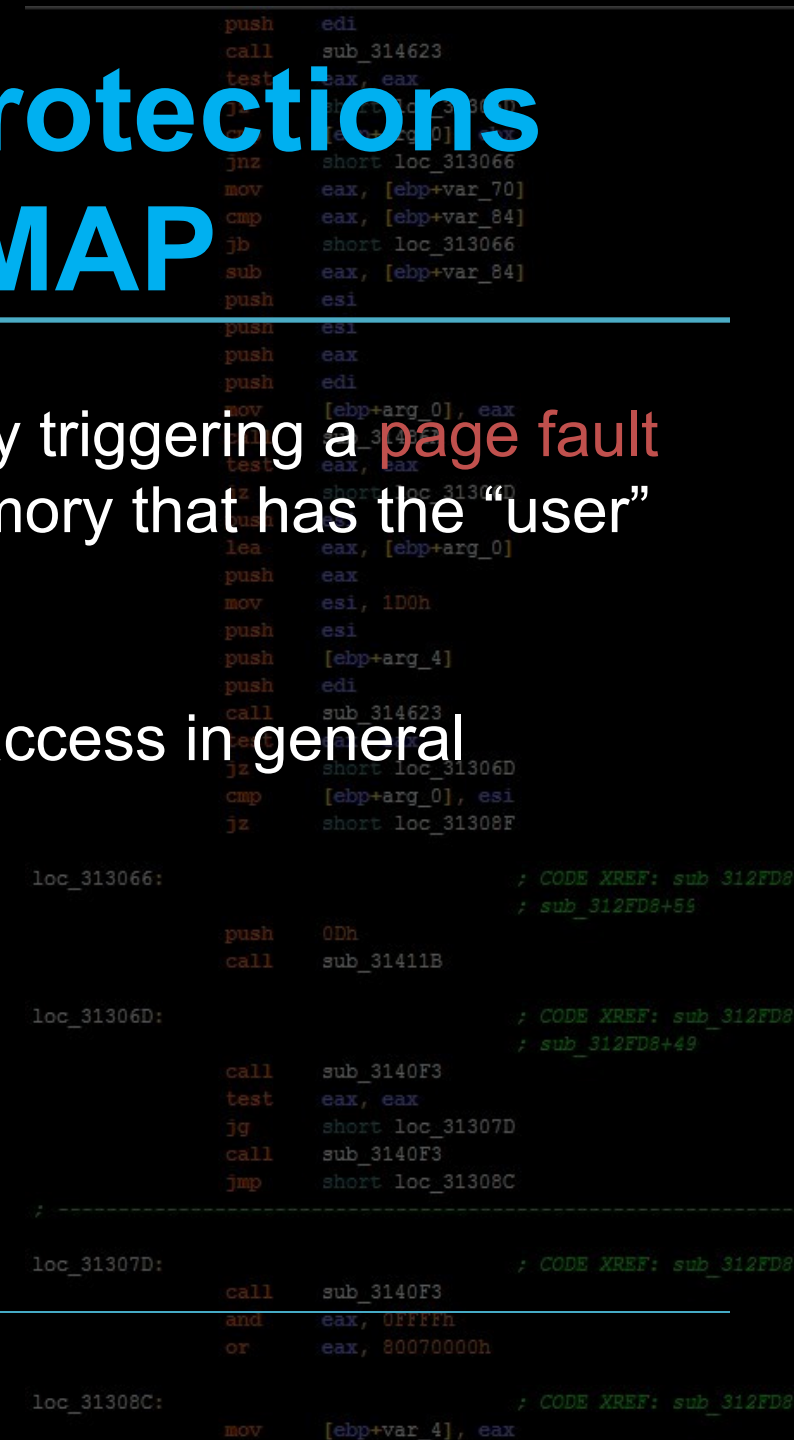
SMEP prevents this type of attack by triggering a page fault if the processor tries to execute memory that has the “user” bit set while in “ring 0”.



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

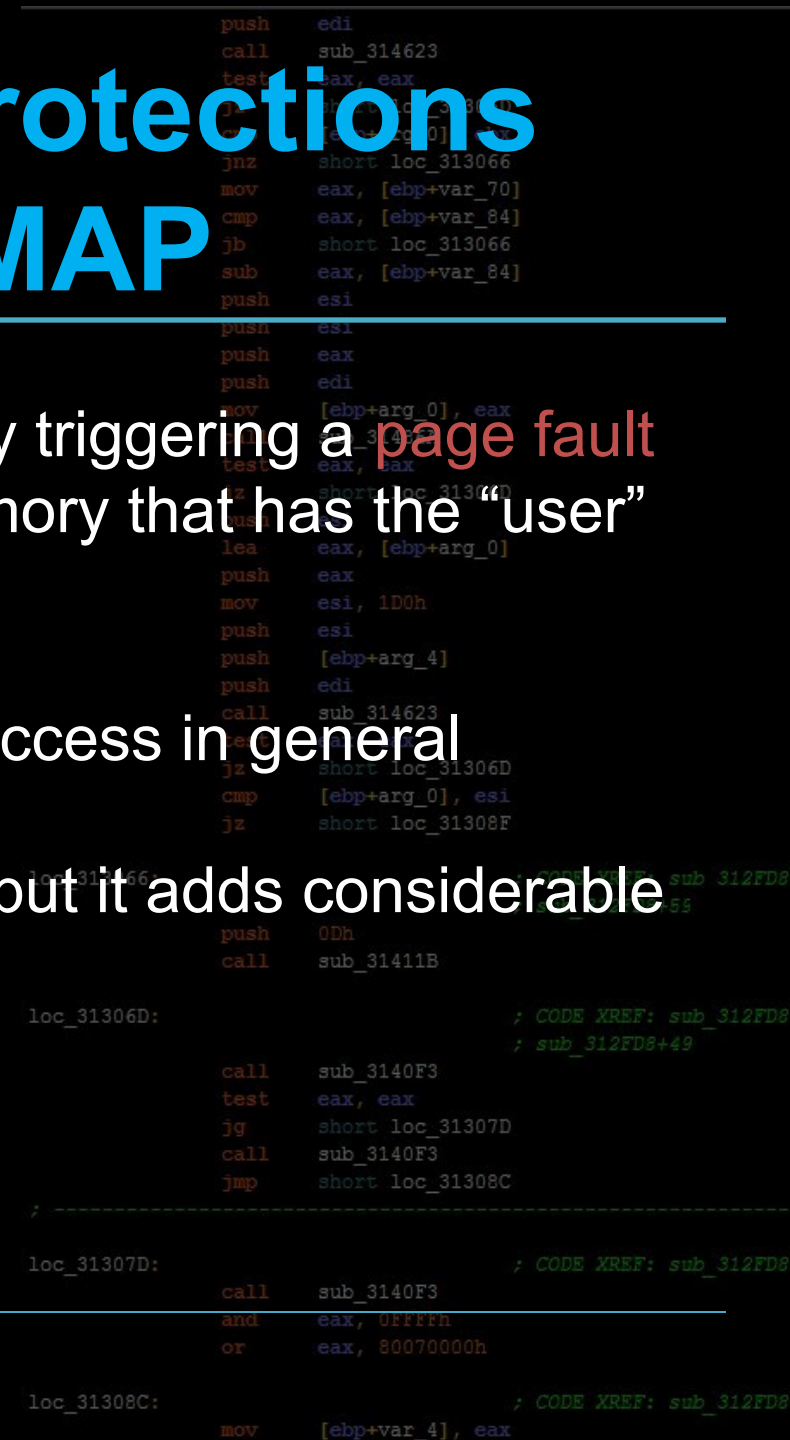


Kernel Space Protections
SMEP / SMAP

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



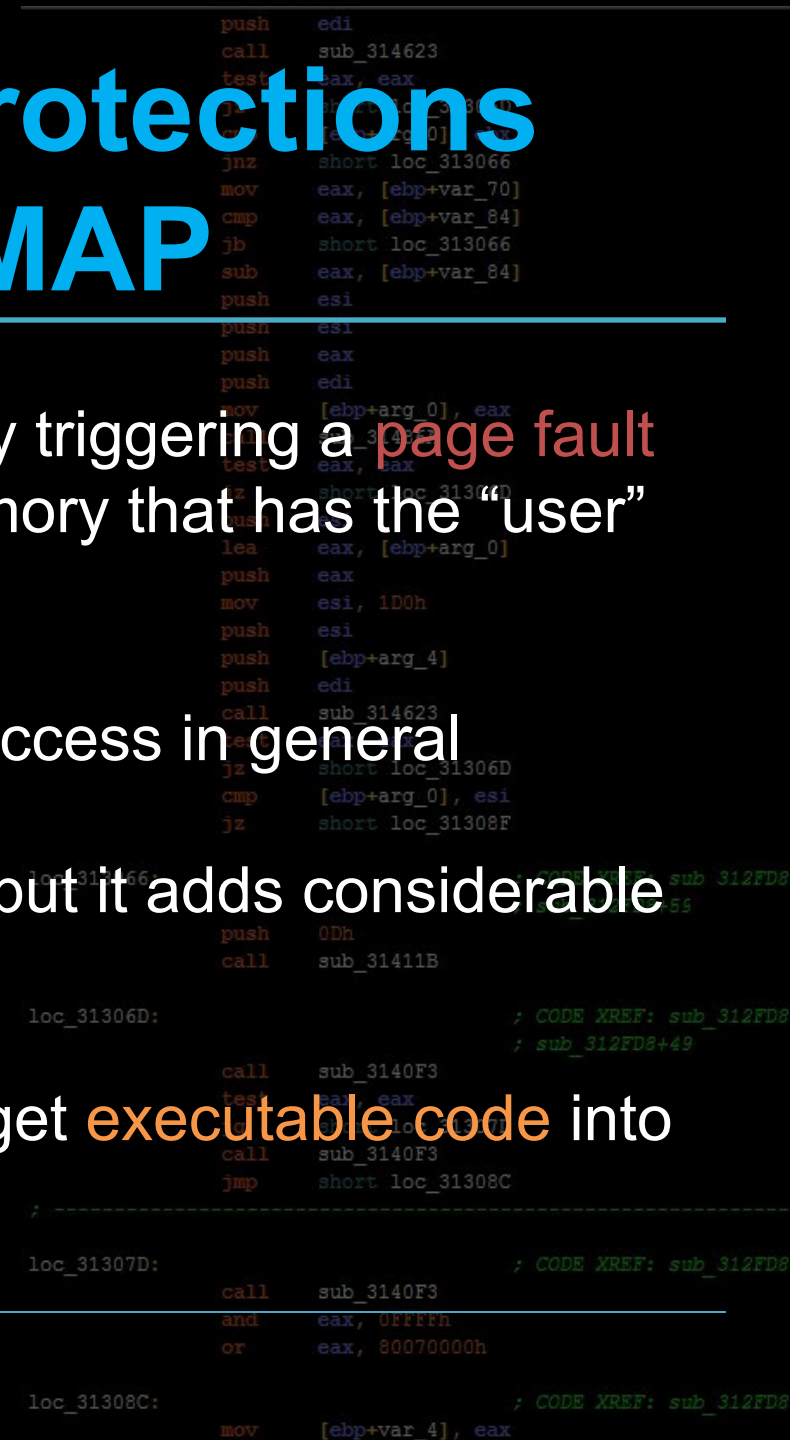
Kernel Space Protections
SMEP / SMAP

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

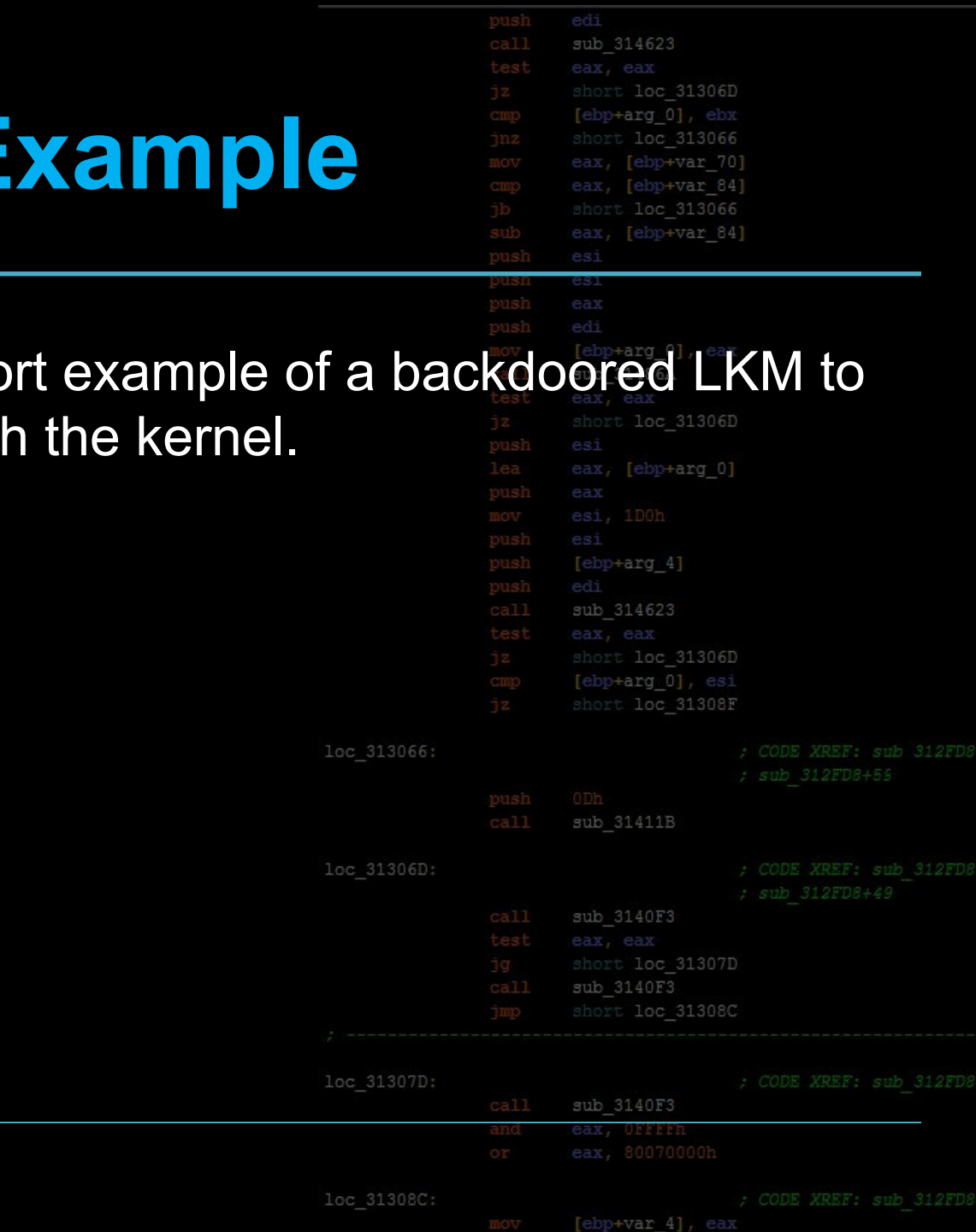
This doesn't *prevent* vulnerabilities, but it adds considerable work to developing a working exploit

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



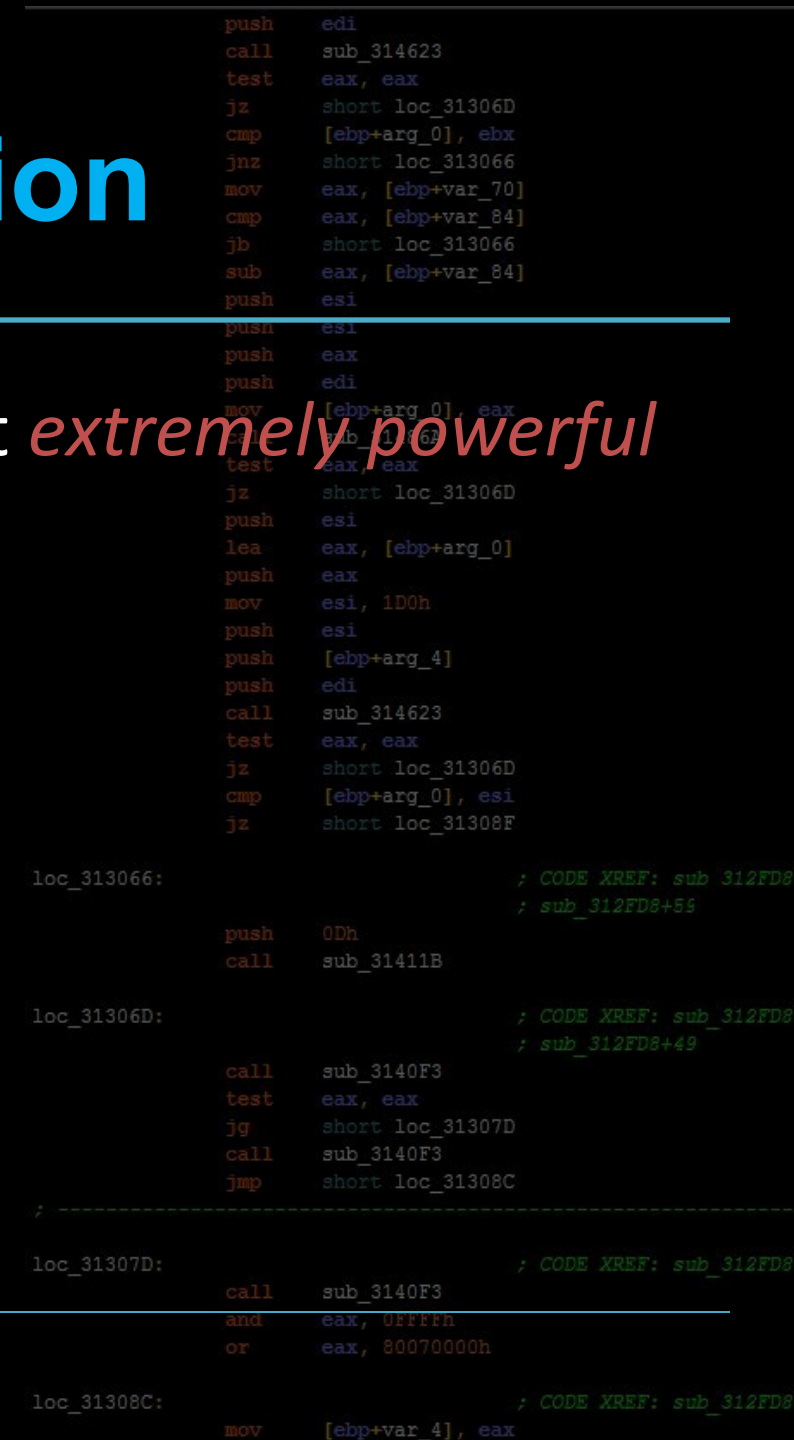
Example

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



Conclusion

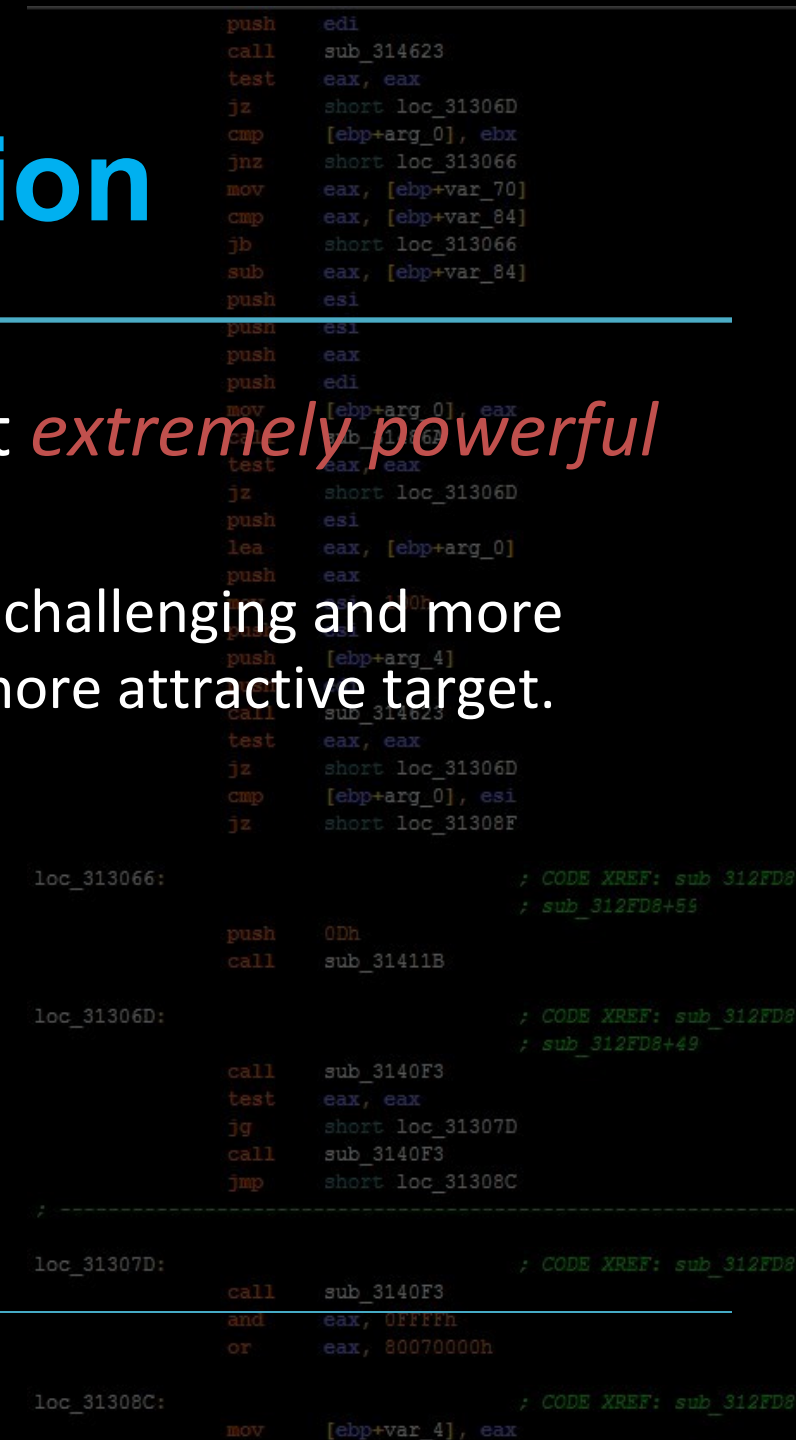
Kernel Exploitation is *weird*, but *extremely powerful*



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.

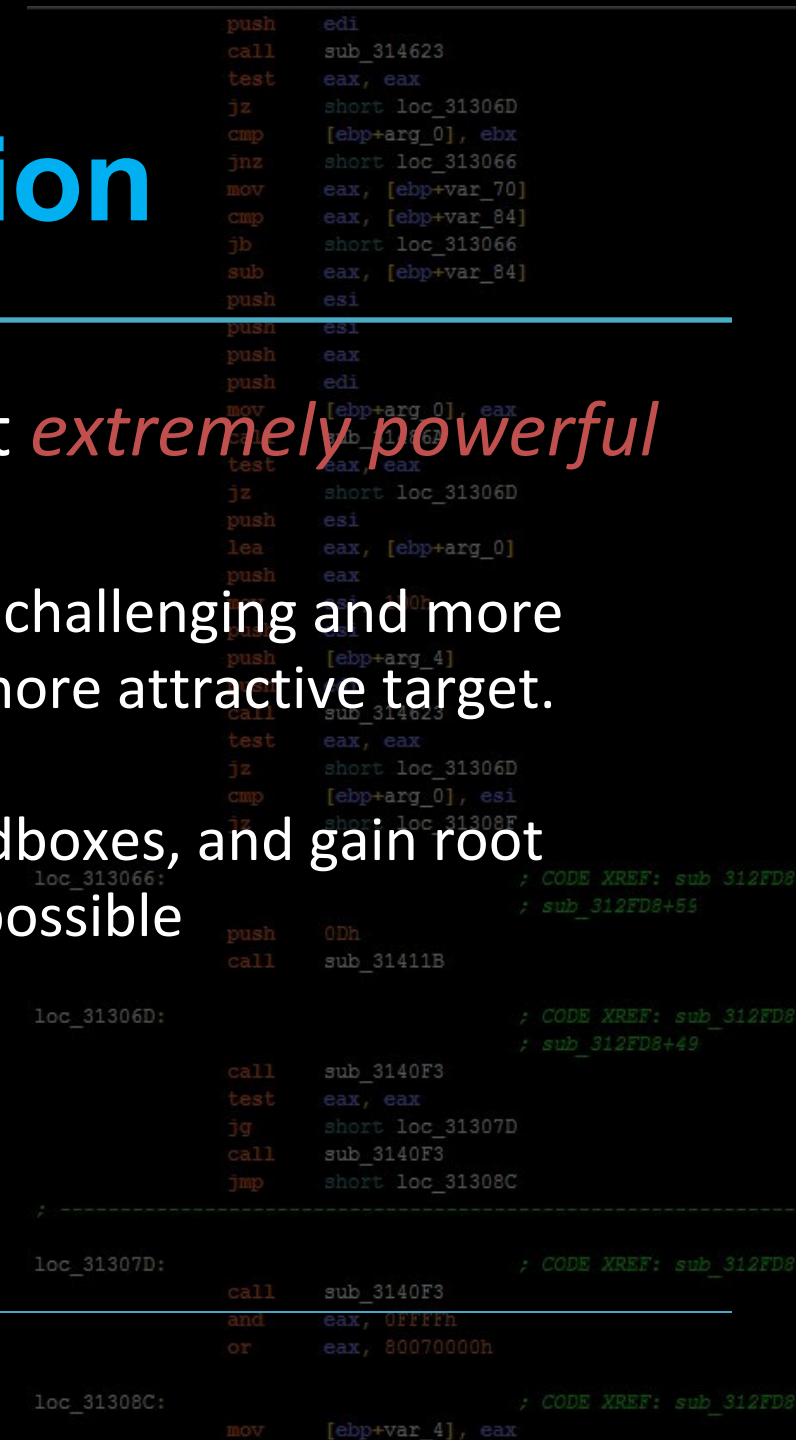


Conclusion

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

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

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



Conclusion

The book on Kernel Exploitation: