Linux Kernel Exploitation

Where no user has gone before

Overview

- Background
- The Vulnerabilities
- The Plans
- The Exploits
- Lessons
- Questions

Background

A kernel is:

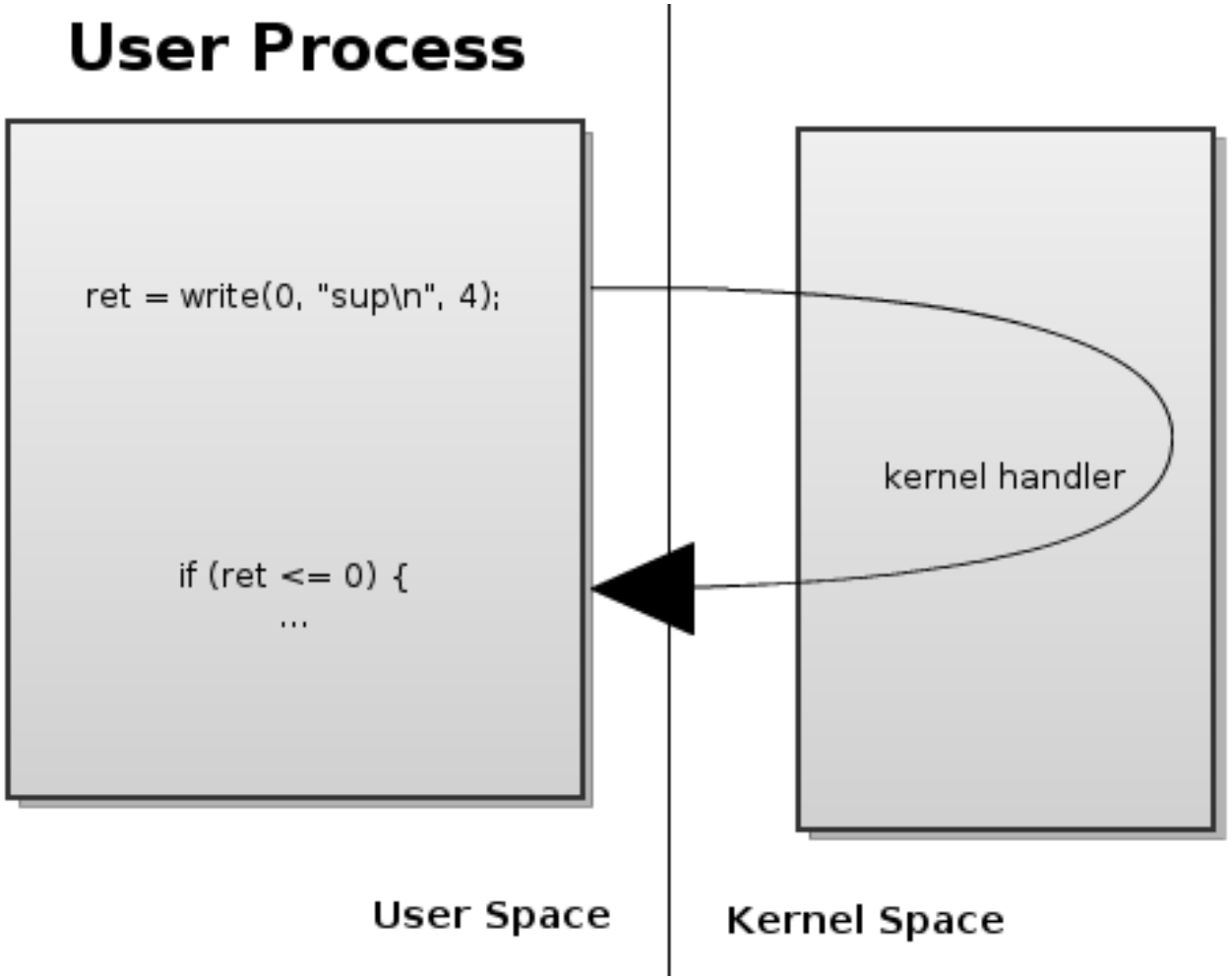
- the main OS program to run after boot
- a giant C program
- handles IO
  - interface between software and hardware
- the lowest level of your OS that programs interact with

Background

System Calls

- One way for processes to interact with the kernel
- Calling process passes off execution to the kernel completely

Background

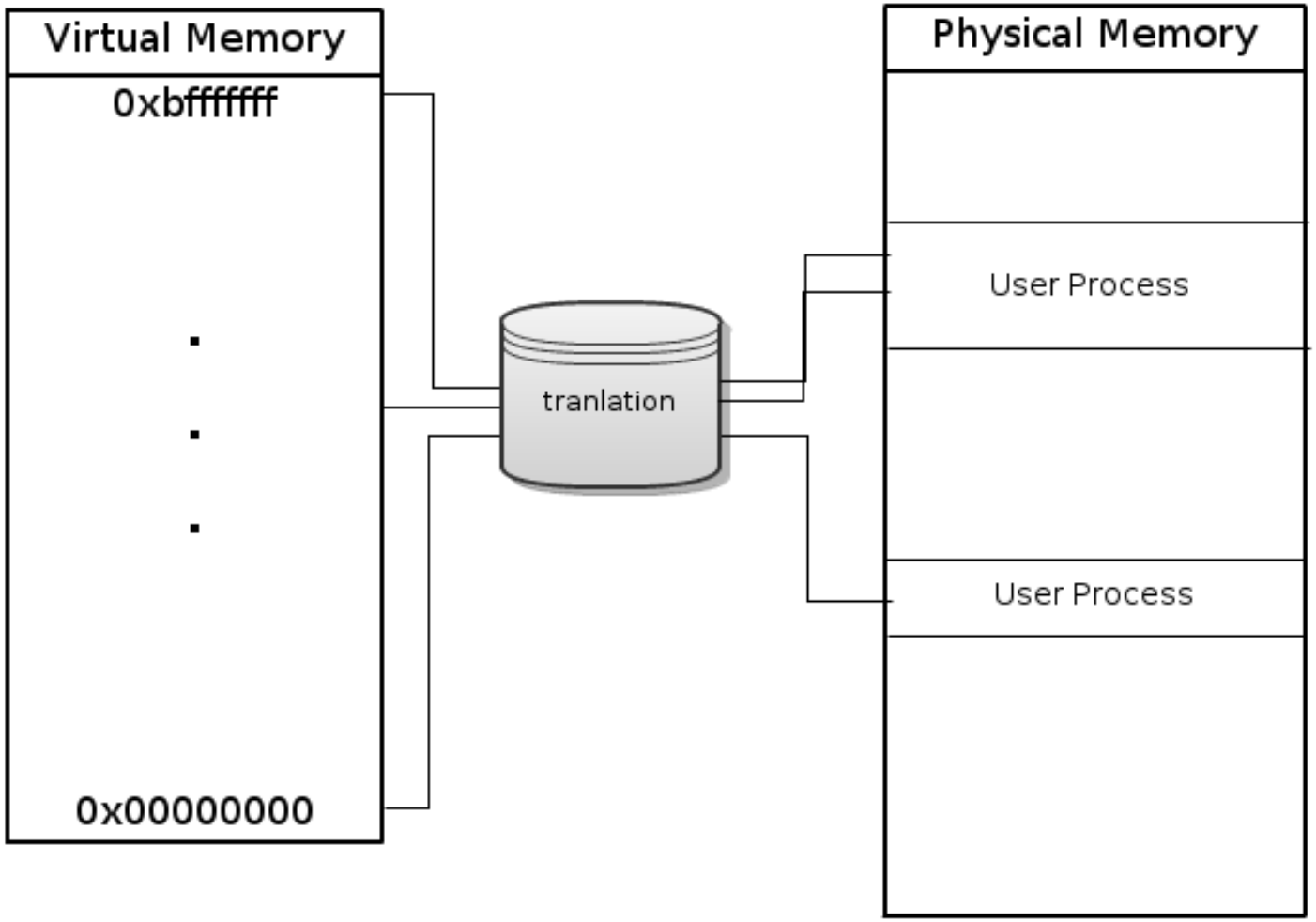

User Process
ret = write(0, "sup\n", 4);
if (ret <= 0) {
...
kernel handler
User Space
Kernel Space

Background

Virtual Memory

- memory is chopped into pages
- each page maps from virtual addresses to physical addresses
- isolates processes from one another
- allows each process access to the full virtual address space
- allows for easy IPC, and shared libraries

Background


Virtual Memory
0xbfffffff
0x00000000
tranlation
Physical Memory
User Process
User Process

Background

On x86 Linux, the kernel is

- mapped into every process (0xc0000000-0xffffffff)
- has hundreds of ioctls, syscalls
- does not have KASLR
- has stack canaries
- doesn't allow certain addresses to be allocated to users (MMAP_MIN_ADDR)

The Vulnerability

```
if (copy_from_user(&fp, ubuf, count))

…

    fp();
```

The Vulnerability

$ cat poc.summary

```
FILE *f = open_file
("/proc/trivial/do_not_read");
/* Set */
fwrite(0x41414141, sizeof(void *), 1, f);
fflush(f);
fread(&p, sizeof(p), 1, f);/* Trigger  */
fclose(f);
```

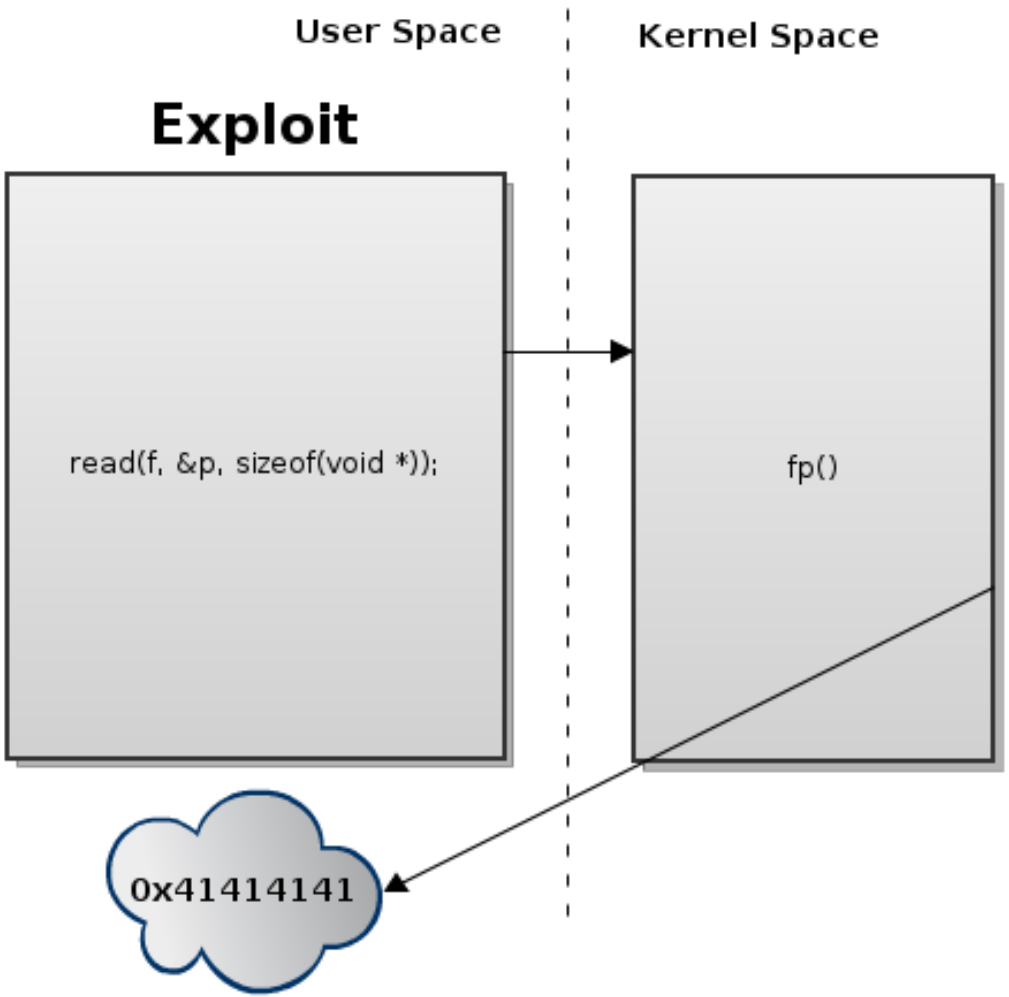
$ ./poc

Killed

The Vulnerability

BUG: unable to handle kernel paging request at 41414141
IP: [<41414141>] 0x41414140
*pdpt = 000000002e38e001 *pde = 0000000000000000
Oops: 0010 [#4] SMP
EIP: 0060:[<41414141>] EFLAGS: 00010206
CPU: 0
EIP is at 0x41414141

The Vulnerability



The Plan

Since we control EIP in kernel mode, we must:

- figure out what we want to do
- figure out how to do so without panicking the kernel

The Plan

What we want to do:

The Plan

What we want to do:

- give ourselves root permissions

The Plan

What we want to do:

- give ourselves root permissions
- run a program of our choosing with our new permissions

The Plan

Permissions

The Plan

Permissions

- how does Linux store permissions?

The Plan

Permissions

- how does Linux store permissions?
- how does Linux check permissions?

The Plan

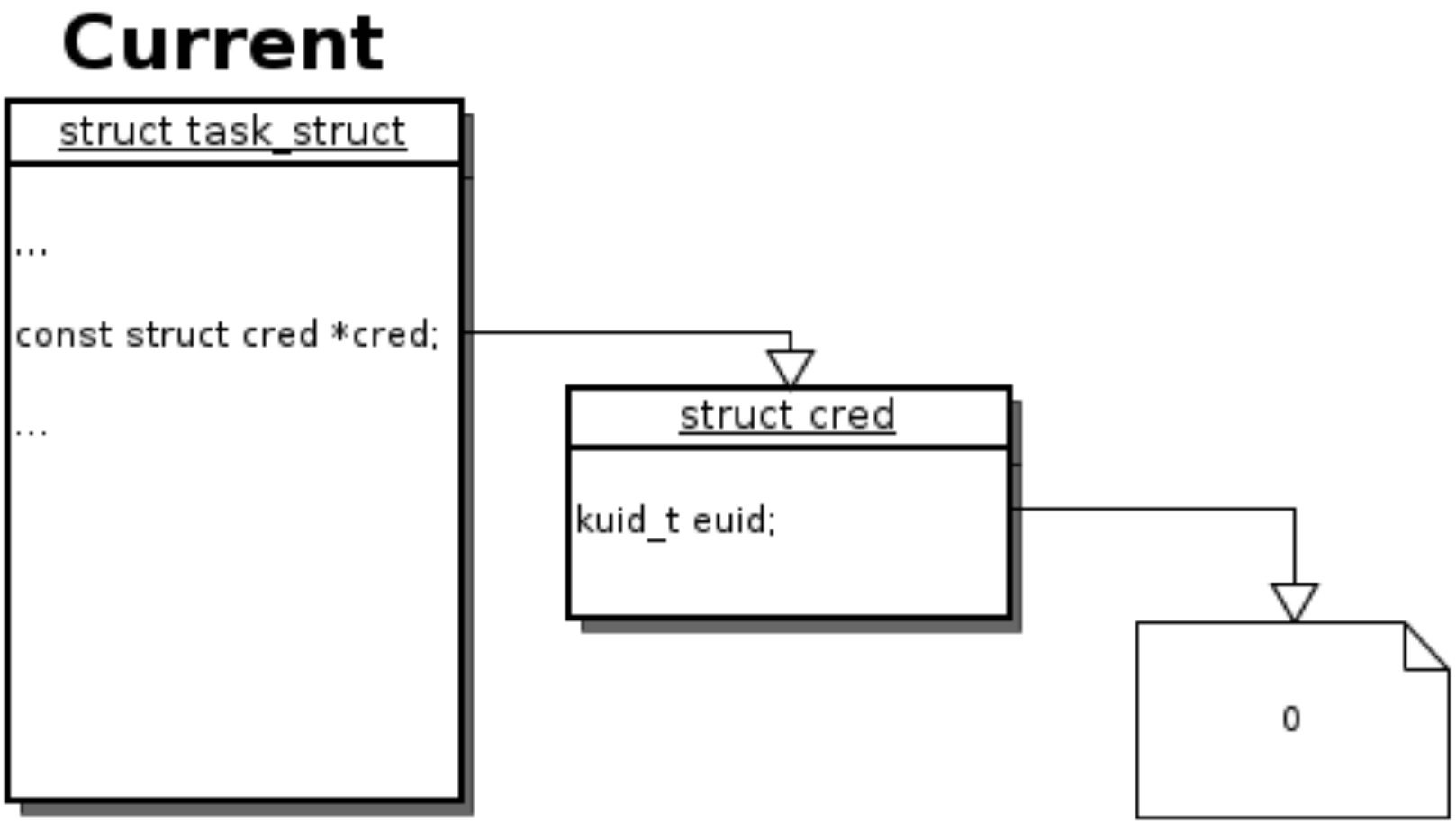
Permissions

- how does Linux store permissions?
- how does Linux check permissions?
- how does Linux give permissions?

The Plan

- each process is stored as type **`struct task_struct`**
- the currently executing process is referenced by the macro **`current`**
- one of the members of **`current`** is **`*cred`**, which is of type **`struct cred`**
- one of the members of **`struct cred`** is **`euid`** (of type **`kuid_t`**, equivalent to **`unsigned int`**)
- **`current->cred->euid`**

The Plan


Current
struct task_struct
...
const struct cred *cred;
...
struct cred
kuid_t euid;
0

The Plan

In light of this our payload should:

- escalate our privileges by setting **`current->cred->euid`** to **`0`**

The Plan

But how do we locate **`current`** in memory?

The Plan

But how do we locate **`current`** in memory?

We don't!* Instead, let's use some of the kernel's infrastructure to help us out.

*But we could if we wanted.

The Plan

We can use:

- `prepare_kernel_cred`
- `commit_creds`

To elevate the current process we do:

- `commit_creds(prepare_kernel_cred(0));`

The Plan

To locate kernel functions, all we have to do is ask the kernel nicely.

$ grep **prepare_kernel_cred** /proc/kallsyms

c046504b T prepare_kernel_cred

$ grep **commit_creds** /proc/kallsyms

c0464d9b T commit_creds

Whoops!

The Plan

Once we have root, we simply:

```
execl("/bin/sh", "sh", "-i", NULL);
```

The Exploit

- locate **`prepare_kernel_cred`**, and **`commit_creds`**
- map our payload to the address we set fp to
- trigger the vulnerability
- open a shell

The Exploit

```
$ ./sploit
[+] prepare_kernel_cred: 0xc046504b
[+] commit_creds: 0xc0464d9b
[+] mapped 0x31337000
[+] enjoy the shell :)
#
```

Lesson

- lots more leg work than userland exploits
- most exploits are one-shot, meaning if we mess up the machine panics
- a lot more tedious to debug
- makes you feel like a champ

Intermission

The Vulnerability

```
memcpy(ubuf, buf + uoff, MAX_LENGTH);

...

if (copy_from_user(buf, ubuf, ucount))
```

The Vulnerability

$ cat poc1.summary

lseek(fd, 16, SEEK_SET);
read(fd, buf, MAX);
print_hex(buf, MAX);

$ ./poc1

```
65 2e 20 42 65 73 74 20  6f 66 20 6c 75 63 6b 21  e. Best of luck!
0a 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
a2 5a 55 96 00 80 ec f7  40 00 00 00 00 00 00 00  .ZU.....@.......
00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
```

The Vulnerability

$ cat poc2.summary

fd = open("/proc/csaw", O_RDWR);

write(fd, buf, TOO_MUCH);

$ ./poc2

[kernel panic]

The Vulnerability

An attacker has these tools at his disposal:

- information disclosure in the form of an arbitrary read of kernel memory
  - let's hope they don't read the canary!
- kernel stack based buffer overflow that is unconstrained in both size and characters
  - let's hope they don't do that!

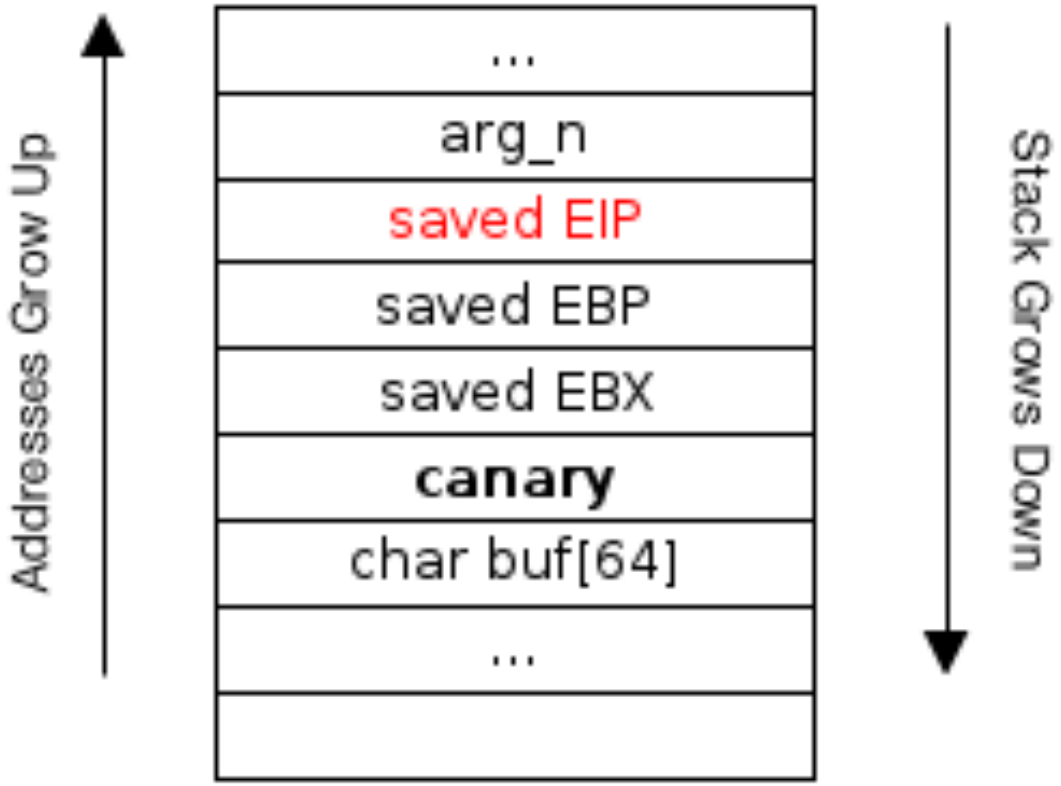
The Plan

Since we:

- can read the canary
- have a text-book case of stack buffer overflow

We can, basically, follow the steps in the previous exploit. Only passing control back to userland will require some rethinking.

The Plan


Addresses Grow Up
...
arg_n
saved EIP
saved EBP
saved EBX
canary
char buf[64]
...
Stack Grows Down

The Plan

Since we clobber the stack, we have two options:

1. Repair the stack (simulate execution of the intended epilogue)
2. Eschew good practice and build our own boat to userland

The Plan

So...

How do we get to userland?

The Plan

iret:

- like ret but needier

The Plan

iret:

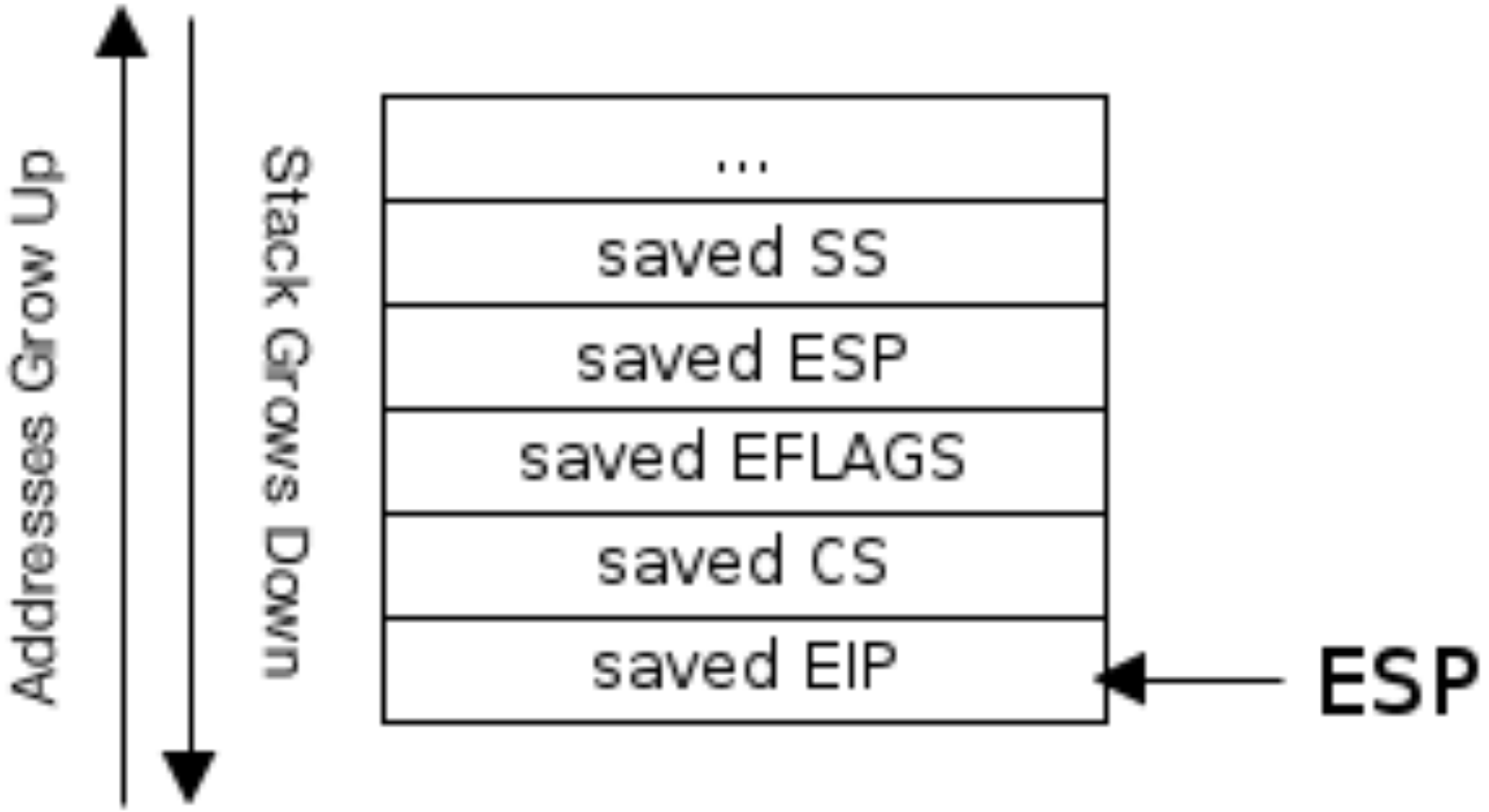
- like ret but needier
- pops 5 words from the stack

The Plan

iret:

- like ret but needier
- pops 5 words from the stack
- easy

The Plan


Addresses Grow Up
Stack Grows Down
...
saved SS
saved ESP
saved EFLAGS
saved CS
saved EIP
ESP

The Plan

Payload

- elevates our privileges
- returns to userland
- opens a shell

The Exploit

1. do all the setup from the previous exploit
2. read the canary
3. overflow the buffer, preserving canary
4. run payload

The Exploit

```
$ ./sploit
[+] using 00321000 for our kernel
payload's addr
[+] using 00960000 for our user
payload's addr
[+] found canary: 44697b18
[*] hold on to your butts
#
```

Lessons

- the kernel does a lot of stuff
  - proper voodoo must be performed
- predicting all of the effects of an exploit is hard
- virtual machines are really nice
- symbols are for those lacking neckbeards

Questions

What will the geopolitical landscape look like in the next hundred years?

Questions?