Return-Oriented Programming

And review of last time.
Memory Defenses
Address Space Layout Randomization

Randomizes the position of stack, heap, program, libraries
Upshot: Even if you can inject code into the stack, you won’t be able to find it.

Note that the text segment (binary code for program) isn’t randomized here.
Detour: Position Independent / Relocatable Code

- `.text` segment holds binary representation of program’s code
- All globbed together, each function one after other
- **Within** the text segment, the position of functions **not** changed
- E.g., if `foo` is at `bar+0x300`, it will **always** be at `bar+0x300`

Program depends on offsets **within** text segment
Detour: Position Independent / Relocatable Code

- .text segment holds binary representation of program’s code
- All globbed together, each function one after other
- **Within** the text segment, the position of functions **not** changed
- E.g., if foo is at `bar+0x300`, it will **always** be at `bar+0x300`

Program depends on offsets within text segment

However, **base address** of text could be randomized
- Code must be compiled with a flag `-f PIE`
  - (Position-Independent Execution)

Q: Why wouldn’t code be compiled with PIE?
A: Can be **faster** to run code that knows its base address
Shows you the **memory maps** for the **current process**

cat /proc/self/maps
Exercise

Find text, static app data, and app global variables
micinski@micinski:~$ cat /proc/self/maps
00400000-0040c000 r-xp 00000000 08:01 1704116 /bin/cat
00600000-0060c000 r--p 00000000 08:01 1704116 /bin/cat
0060c000-0060d000 rw-p 00000000 08:01 1704116 /bin/cat
00d37000-00d58000 rw-p 00000000 00:00 0 [heap]
7fb458920000-7fb458bf8000 r--p 00000000 08:01 2635826 /usr/lib/locale/locale-archive
7fb458bf8000-7fb458db8000 r-xp 00000000 08:01 25562894 /lib/x86_64-linux-gnu/libc-2.23.so
7fb458db8000-7fb458fb8000 ---p 001c0000 08:01 25562894 /lib/x86_64-linux-gnu/libc-2.23.so
7fb458fb8000-7fb458fbc000 r--p 001c0000 08:01 25562894 /lib/x86_64-linux-gnu/libc-2.23.so
7fb458fbc000-7fb458fbe000 rw-p 001c4000 08:01 25562894 /lib/x86_64-linux-gnu/libc-2.23.so
7fb458fbe000-7fb458fc2000 rw-p 00000000 00:00 0
7fb458fc2000-7fb458fe8000 r-xp 00000000 08:01 25562855 /lib/x86_64-linux-gnu/ld-2.23.so
7fb45919f000-7fb4591c4000 rw-p 00000000 00:00 0
7fb4591e5000-7fb4591e7000 rw-p 00000000 00:00 0
7fb4591e7000-7fb4591e8000 r--p 00025000 08:01 25562855 /lib/x86_64-linux-gnu/ld-2.23.so
7fb4591e8000-7fb4591e9000 rw-p 00026000 08:01 25562855 /lib/x86_64-linux-gnu/ld-2.23.so
7fb4591e9000-7fb4591ea000 rw-p 00000000 00:00 0
7fff36194000-7fff361b5000 rw-p 00000000 00:00 0 [stack]
7fff361f8000-7fff361fa000 r--p 00000000 00:00 0 [vvar]
7fff361fa000-7fff361fc000 r-xp 00000000 00:00 0 [vdso]
ffffffffffffff600000-ffffffffffff601000 r-xp 00000000 00:00 0 [vsyscall]
Defeating ASLR

Two main methods: **brute force** and **derandomization**

Just try a bunch of different addresses and hope for the best

(Doesn’t work so well in a 64-bit address space..)
Defeating ASLR

Two main methods: **brute force** and **derandomization**

Get program to **leak** the value of a pointer to you
void insecure(char *str) {
    char buffer[100];
    if (str[3] == 'H') {
        send("&x", &buffer); // Assume this goes back to user
    }
    strcpy(buffer,str);
}
void insecure(char *str) {
    char buffer[100];
    if (str[3] == 'H') {
        send("&x", &buffer); // Assume this goes back to user
    }
    strcpy(buffer,str);
}

This example is obviously fake
However, much more common is error logs
(If you can convince an app to throw an error to you that contains pointer, you win!)
PS4 Kernel dumped in 11 days via error logs attacker can control!
Careful: learning address of stack doesn’t tell you where text segment is
Non-executable (stack / heap)

W^X is a simple concept: don’t let the programmer execute parts of memory that they can also write

Simple and Effective Defense!

Coordinate w/ CPU
Defeating $\text{NX} / W^X$:

- Return-to-libc
- Return-oriented-programming
Return-to-libc

NX: If we try to execute shellcode here, program will **crash**!
Return-to-libc

But, can still point return addr at something in .text
E.g., system, exit, etc..

But, arguments must be set up for function already
Stack Canaries

Idea: use a **known value** that—if it gets smashed over—alerts you to presence
“Normal” execution

- `%rsp+X+0x10` → Stuff from foo...
- `%rsp+X+0x8` → Return addr
- `%rsp+X` → Saved `%rbp`
- `%rsp+0x3E8` → buffer[999]
- `%rsp` → buffer[0]
Canary Insertion

Compiler Inserts
This Canary
(Upon function entry)

Before exiting, **check**
canary to ensure same

Stuff from foo…

Return addr

Canary Value

Saved %rbp

... buffer[999]

... buffer[0]
**Exercise:** Compile with and *without* `-fno-stack-protector`
Defeating Canaries

Can still “skip past” canary occasionally

If attacks “owns” x, can set to skip canary

```c
void foo(char *p, int x) {
    char buffer[100];
    strcpy(buffer+x,p);
}
```
Defeating Canaries

Even if stack overflows can’t happen, heap overflows can…

```c
struct closure {
    int x;
    int y;
    void (*f)(int);
    char str[8];
}

closure *x = malloc(sizeof(closure));
strcpy(x->str, owned_string));
x->f(42);
```
Exercise: Describe w/ partner how you would break this program

```c
struct closure {
    int x;
    int y;
    char str[100];
    void (*f)(int);
};

int main(int argc, char **argv) {
    closure *x = malloc(sizeof(closure));
    strcpy(x->str, argv[1]);
    x->f(42);
}
```
In practice, many of these defenses are employed, and they really do pretty well.

However, the thinking here builds intuition for things we still see today...
Return-Oriented-Programming

Way of “scavenging” through the program’s binary code to trick it into doing what you want
Say I wanted to do the following:

• Set %rax to 0
• Execute the “syscall” instruction
Say I wanted to do the following:

- Set `%rdi` to 1 (arg for exit)
- Set `%rax` to 60 (exit)
- Execute the “syscall” instruction

If I have NX turned on, I can’t just **inject** this into the program:

```assembly
movq $1, %rdi
movq $60, %rax
syscall
```
What might I do instead?
What might I do instead?

I could try to see the program already has a function that does this already and use that.

(I.e., return-to-libc)
What might I do instead?

What if I can’t find a whole function that does this?
Normally… Function starts here and continues until (either) `ret`.

```
0xF000: pushq %rbp
0xF002: movq %rsp, %rbp
0xF004: subq $12, %rsp
0xF007: mov %eax, -4(%rbp)
0xF009: mov %eax, -8(%rbp)
0xF00b: mov %eax, -12(%rbp)
0xF00e: add %eax, %eax
0xF010: compl %eax, %eax
0xF013: jmpg 0xF01d
0xF015: addq $12, %rsp
0xF018: leave
0xF019: mov $60, %rax
0xF01b: syscall
0xF01c: ret
0xF01d: addq $12, %rsp
0xF01f: leave
0xF020: ret
```
Normally… Function starts here and continues until (either) `ret`

```
0xF000: pushq %rbp
0xF002: movq %rsp, %rbp
0xF004: subq $12, %rsp
0xF007: mov %eax, -4(%rbp)
0xF009: mov %eax, -8(%rbp)
0xF00b: mov %eax, -12(%rbp)
0xF00e: add %eax, %eax
0xF010: compl %eax, %eax
0xF013: jmpg 0xF01d
0xF015: addq $12, %rsp
0xF018: leave
0xF019: mov $60, %rax
0xF01b: syscall
0xF01c: ret
0xF01d: addq $12, %rsp
0xF01f: leave
0xF020: ret
```

But nothing stops me from jumping right `here`!
So I could look through binary and find all places with `ret` and jump to any number of bytes before that.

Observation: can execute sequences of code that weren’t technically in program to begin with
Observation: x86_64 instructions are **variable length**

Like words…

“the address”
Observation: x86_64 instructions are **variable length**

Like words…

“the address”
Observation: x86_64 instructions are \textit{variable length}

Like words…

“\textit{the address}”
Read starting at c7
Let’s say that I want to call D01F and **then** F019

```
...  0xD01F: pop %rdi
  0xD020: ret
...

...  0xF019: mov $60, %rax
  0xF01B: syscall
  0xF01C: ret
...
```
To “set up” the attack we put 0xD01F in saved RIP

```
... 0xD01F: pop %rdi
  0xD020: ret
...

... 0xF019: mov $60, %rax
  0xF01B: syscall
  0xF01C: ret
...```

Stuff from foo...

Return addr

Saved %rbp

Bar’s frame

buffer[999]

buffer[0]
To “set up” the attack we put 0xD01F in saved RIP

... 0xD01F: pop %rdi
  0xD020: ret
...

... 0xF019: mov $60, %rax
  0xF01B: syscall
  0xF01C: ret
...
Before **foo** returns, it pops all of this stuff from the stack

```assembly
... 0xD01F: pop %rdi
  0xD020: ret
...

... 0xF019: mov $60, %rax
  0xF01B: syscall
  0xF01C: ret
...```

**Bar’s frame**

- Stuff from foo...
- Saved %rbp
- buffer[999]
- buffer[0]
Now it goes here

Super Critical: **pops** 0xD01F from stack!

(Rather than it’s caller `foo`)
So now whatever’s on stack will be popped into %rdi

(Which is previously stuff in foo’s stack)
So if I want to put 1 in RDI, I put it **here**

(Which is previously stuff in foo's frame)
So if I want to put 1 in RDI, I put it **here**

(Which is previously stuff in **foo**’s frame)
Now, when the code hits this point, it's going to execute a return

Which will yet again go to whatever address is in %rsp
... 0xD01F: pop %rdi
0xD020: ret
...

... 0xF019: mov $60, %rax
0xF01B: syscall
0xF01C: ret
...

Critical observation: if %rsp is now 0xF019, we’ll get what we want
%rsp

0xF019

... 0xD01F: pop %rdi
    0xD020: ret
...

... 0xF019: mov $60, %rax
    0xF01B: syscall
    0xF01C: ret
...

Critical observation: if %rsp is now 0xF019, we’ll get what we want
• Set `%rdi` to 1 (arg for exit)
• Set `%rax` to 60 (exit)
• Execute the “syscall” instruction

...  
0xD01F: pop `%rdi`
0xD020: ret
...

...  
0xF019: mov $60, `%rax`
0xF01B: syscall
0xF01C: ret
...

Critical observation: if `%rsp` is **now** 0xF019, we’ll get what we want
**Observation:** We can **chain** multiple sequences (that all end in **ret**) by setting up the stack right
Exercise
write(1, “Hello, world!”, 13);

%rax = 1 %rdi = 1 %rsi = &”Hello, world”, %rdx = 13

0xC110: pop %rsi
0xC112: ret

0xD235: xchang %rdx, %rdi
0xD238: ret

0xB0FF: pop %rdx
0xB102: ret

0xCA2F: syscall

0x1029: pop %edx
0x102a: ret

0xF019: pop %eax
0xF01B: ret

0xB0FF: pop %rdx
0xB102: ret

Assume this is 128

buffer = 0x40000

Saved %rbp
...
buffer[999]
...
buffer[0]