Module:
Return Oriented Programming

Introduction

Yan Shoshitaishvili
Arizona State University

Recap: The "No-eXecute" bit

Modern architectures support memory permissions:

- **PROT_READ** allows the process to read memory
- **PROT_WRITE** allows the process to write memory
- **PROT_EXEC** allows the process to execute memory

Intuition: *normally*, all code is located in .text segments of the loaded ELF files. There is no need to execute code located on the stack or in the heap.

By default in modern systems, the stack and the heap are *not* executable.

In the absence of Code *Injection*, we turn to Code *Reuse*.

Blast from the past: Return-to-libc

How can we deal with a non-executable stack?

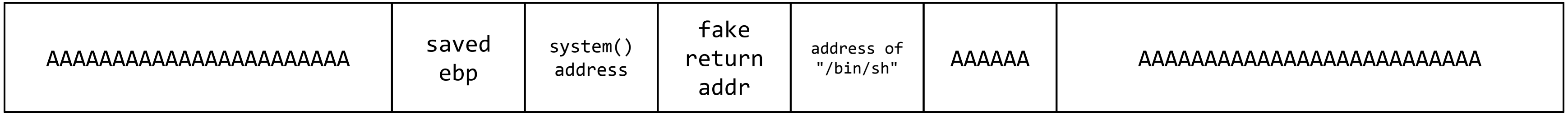
In the old times (32-bit x86), arguments were passed on the stack. During a stack-based buffer overflow, we could overwrite the return address *and* the arguments.

vuln buffer	saved ebp	return address	vuln arg	vuln arg	vuln arg	caller stack frame

Blast from the past: Return-to-libc

How can we deal with a non-executable stack?

In the old times (32-bit x86), arguments were passed on the stack. During a stack-based buffer overflow, we could overwrite the return address *and* the arguments.



AAAAAAAAAAAAAAAAAAAAAAA	saved ebp	system() address	fake return addr	address of "/bin/sh"	AAAAAA	AAAAAAAAAAAAAAAAAAAAAAAAAA

When vuln() returns, it will call system("/bin/sh").

Blast from the past: Return-to-libc

Discovered in 1997 by Solar Designer.

Why is this a blast from the past?

Modern architectures don't take arguments on the stack...

Game over?

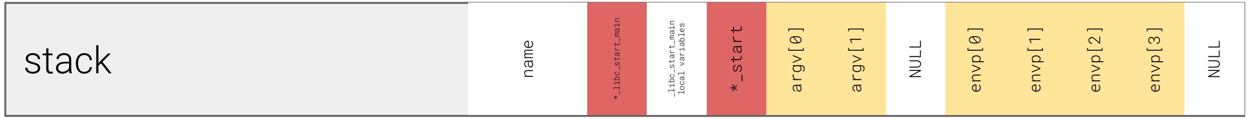
Code reuse in AMD64

All is not lost!

To begin with, recall the memory errors module:

```
int main() {
    char name[16];
    read(0, name, 128);
}
int win() {
    sendfile(1, open("/flag", 0), 0, 1024);
}
```

We can jump to functions in the code!



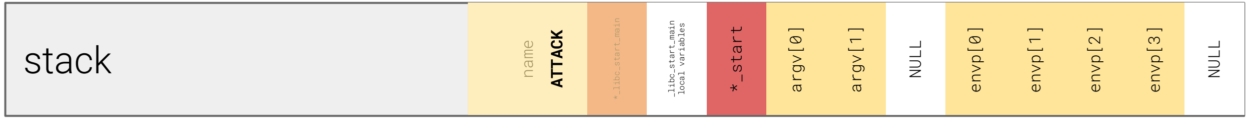
Code reuse in AMD64

All is not lost!

To begin with, recall the memory errors module:

```c
int main() {
    char name[16];
    read(0, name, 128);
}
int win() {
    sendfile(1, open("/flag", 0), 0, 1024);
}
```

We can jump to functions in the code!



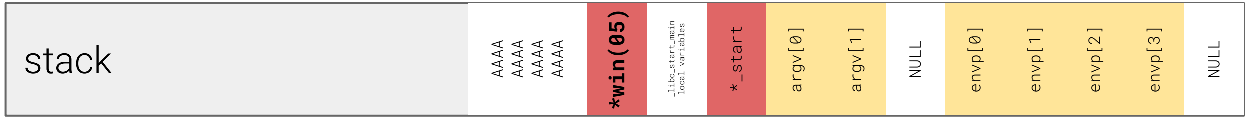
Code reuse in AMD64

All is not lost!

To begin with, recall the memory errors module:

```
01 int main() {
02    char name[16];
03    read(0, name, 128);
04 }
05 int win() {
06    sendfile(1, open("/flag", 0), 0, 1024);
07 }
```

We can jump to functions in the code!

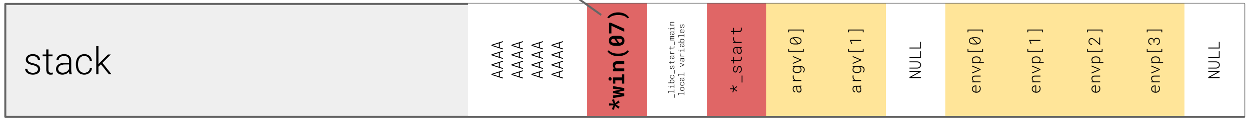


Code reuse in AMD64

Going further: recall later levels of babymem:

```
01 int main() {
02     char name[16];
03     read(0, name, 128);
04 }
05 int win(int tricky) {
06     if (tricky != 1337) return;
07     sendfile(1, open("/flag", 0), 0, 1024);
08 }
```

We can jump into the middle of functions in the code!



Code reuse in AMD64

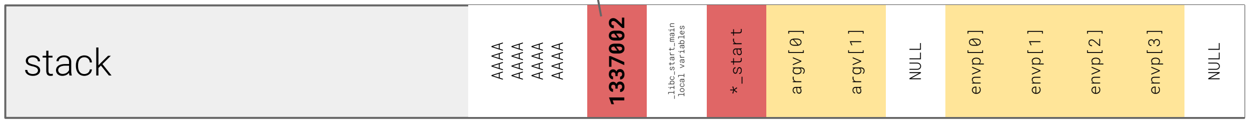
Keep going! Recall jitspraying in toddler1!

```
0x1337000        49 bc 31 c0 b0 3c 0f 05 90 90        mov r15, 0x9090050f3cb0c031
```

If you jump to 0x1337002, you will execute:

```
0x1337002        31 c0          xor eax eax
0x1337003        b0 3c          mov al, 60
0x1337004        0f 05          syscall
0x1337005        90             nop
0x1337006        90             nop
```

We can jump into the middle of functions in the code!



Return Oriented Programming

The generalization of Return-to-libc is Return Oriented Programming.

These capabilities, when an attacker is able to overwrite return addresses on the stack, are extremely powerful.

Now, you will master them!