CSCI2467: Systems Programming Concepts

Slideset 12: Stack attacks and defenses

Source: CS:APP section 3.10, Bryant & O'Hallaron

Course Instructors:

Matthew Toups
Caitlin Boyce

Course Assistants:

Saroj Duwal
David McDonald

Spring 2020

THE UNIVERSITY of NEW ORLEANS

Our last lab assignment

- Lab4 (attacklab) is out

  Due: Monday, April 20, 11:59pm

- like before, you will defeat "phases" of an unknown, unique binary program
- unlike before, no "explosion" penalty

Attacklab handin!

Yes you must hand in to Autolab

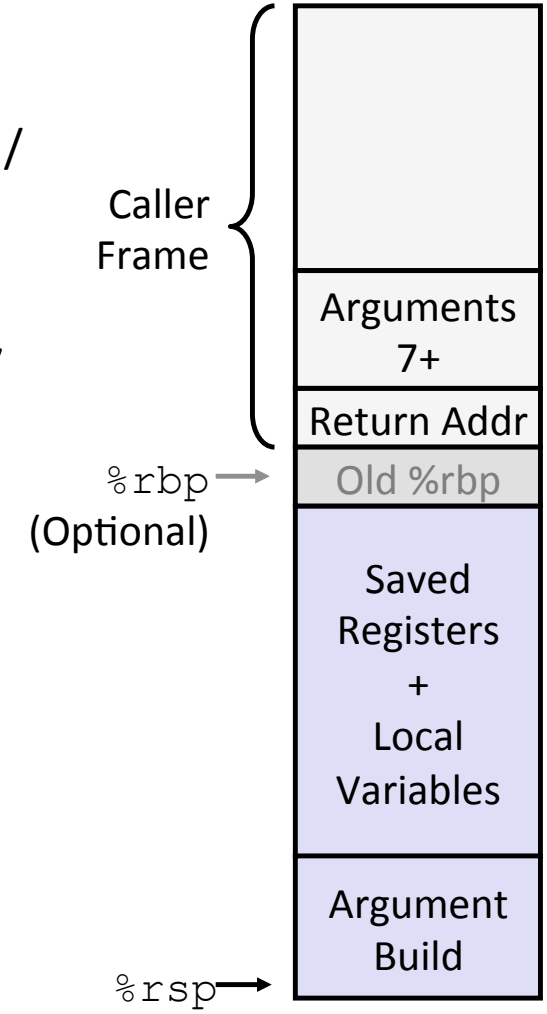
- You must hand in a commented, plain-text version of your solutions (eg. `phase1.txt`)
- Use autolab submit button
- After you submit, your score will show up on scoreboard
- Please submit your new solution every time you solve a phase
- Comments must explain how and why your solution works!
- Convince us you know! Otherwise we will not award points

Today

0. Class updates
1. Memory layout
2. Buffer Overflow
   - Vulnerability
   - History
   - Protection
     - Bug-free code?
     - ASLR and NX
     - Stack canaries
   - Arms race!
     - Return-oriented programming (ROP)
3. Attacklab (lab4)

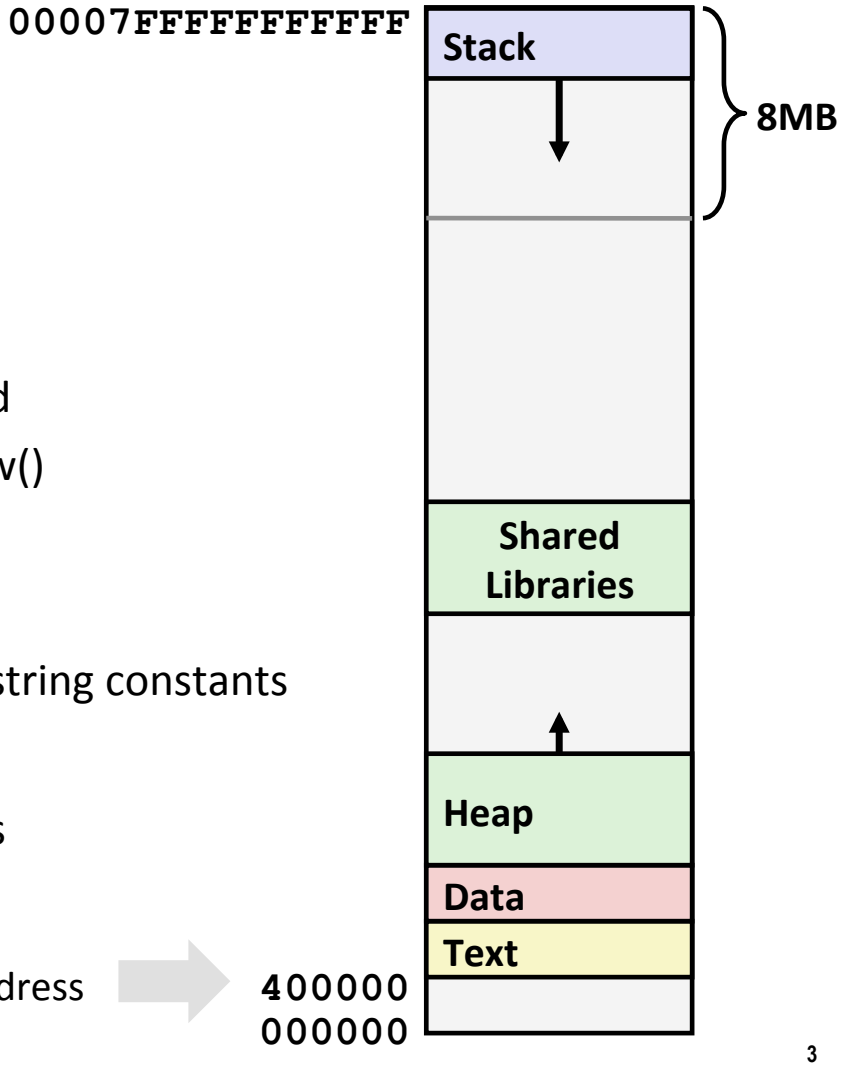
x86-64 Procedure Summary

- **Important Points**
  - Stack is the right data structure for procedure call / return
    - If P calls Q, then Q returns before P
- **Recursion (& mutual recursion) handled by normal calling conventions**
  - Can safely store values in local stack frame and in callee-saved registers
  - Put function arguments at top of stack
  - Result return in `%rax`
- Pointers are addresses of values
  - On stack or global



x86-64 Linux Memory Layout

not drawn to scale

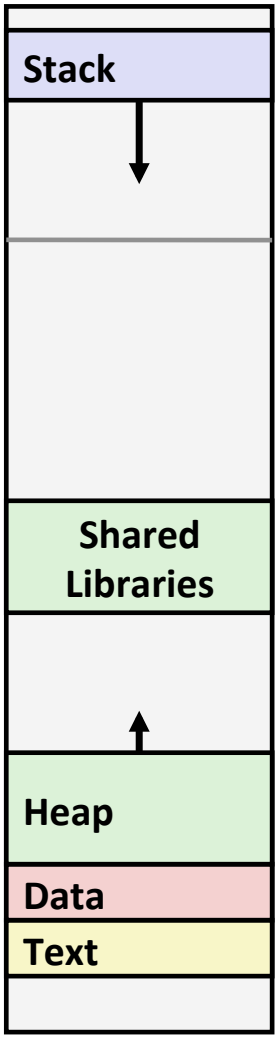


- **Stack**
  - Runtime stack (8MB limit)
  - E. g., local variables
- **Heap**
  - Dynamically allocated as needed
  - When call malloc(), calloc(), new()
- **Data**
  - Statically allocated data
  - E.g., global vars, `static` vars, string constants
- **Text / Shared Libraries**
  - Executable machine instructions
  - Read-only

Memory Allocation Example

not drawn to scale

```
char big_array[1L<<24];  /* 16 MB */
char huge_array[1L<<31]; /*  2 GB */

int global = 0;

int useless() { return 0; }

int main ()
{
    void *p1, *p2, *p3, *p4;
    int local = 0;
    p1 = malloc(1L << 28); /* 256 MB */
    p2 = malloc(1L << 8);  /* 256  B */
    p3 = malloc(1L << 32); /*   4 GB */
    p4 = malloc(1L << 8);  /* 256  B */
 /* Some print statements ... */
}
```



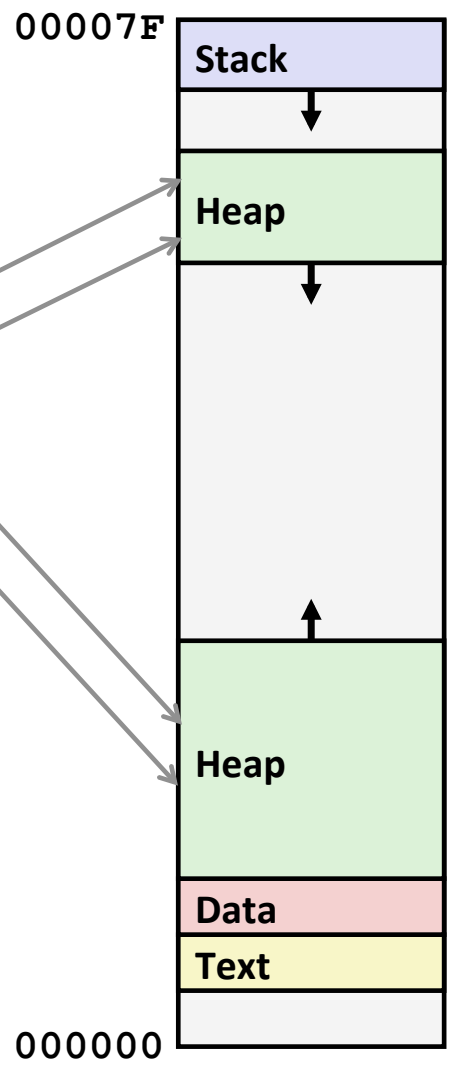
Where does everything go?

x86-64 Example Addresses

not drawn to scale

address range ~2^{47}

local	0x00007ffe4d3be87c
p1	0x00007f7262a1e010
p3	0x00007f7162a1d010
p4	0x000000008359d120
p2	0x000000008359d010
big_array	0x0000000080601060
huge_array	0x0000000000601060
main()	0x000000000040060c
useless()	0x0000000000400590



Memory Referencing Bug: an example

```
typedef struct {
  int a[2];
  double d;
} struct_t;

double fun(int i) {
  volatile struct_t s;
  s.d = 3.14;
  s.a[i] = 1073741824; /* Possibly out of bounds */
  return s.d;
}
```

See:

/home/CSCI2467/labs/misc/structmem/structmem.c

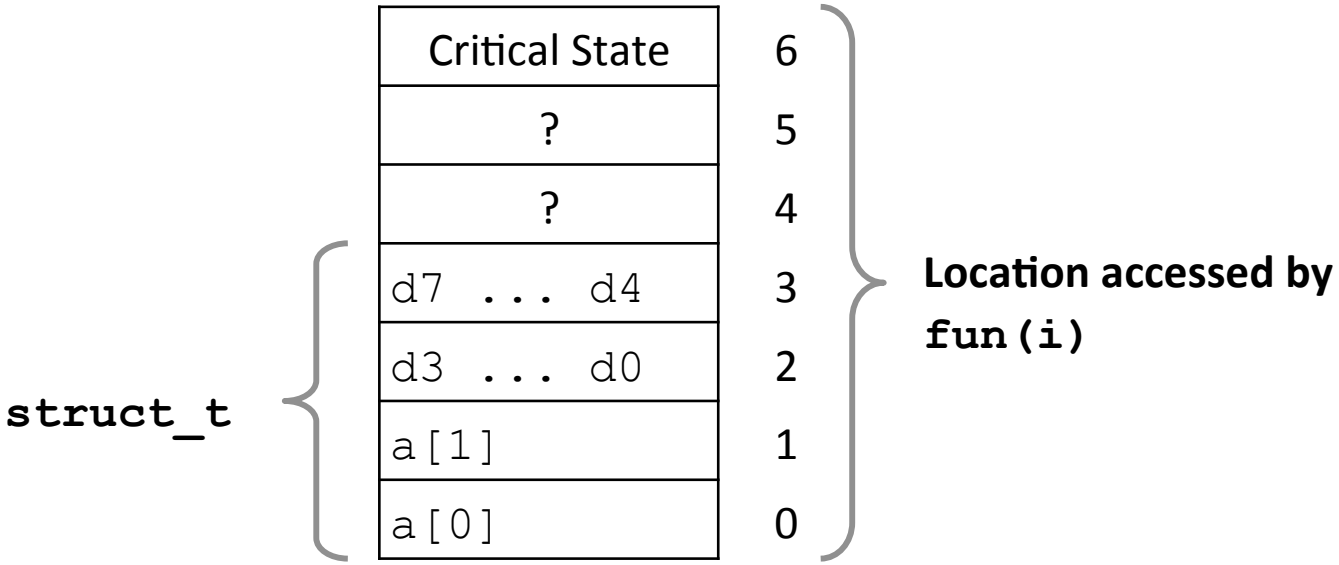
```
typedef struct {
  int a[2];
  double d;
} struct_t;

double fun(int i) {
  volatile struct_t s;
  s.d = 3.14;
```

Memory Referencing Bug: an example

```
typedef struct {
  int a[2];
  double d;
} struct_t;
```

```
fun(0)  →  3.14
fun(1)  →  3.14
fun(2)  →  3.1399998664856
fun(3)  →  2.0000061035156
fun(4)  →  3.14
fun(6)  →  Segmentation fault
```

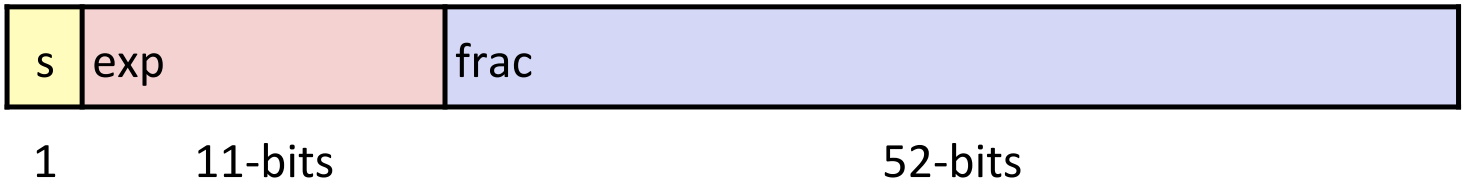
Explanation:



Recall floats

- Single precision: 32 bits

- Double precision: 64 bits



Buffer overflows: big security implications

- What we just saw is generally called a *buffer overflow*
- Why a big deal?
- For decades, has been the #1 technical cause of security vulnerabilities

  (#1 overall cause is social engineering)
- Most common form:
- unchecked lengths on string inputs
- particularly for bounded character arrays on the stack

  (sometimes referred to as "stack smashing")

String library code

- **Implementation of Unix function `gets()`**

```
/* Get string from stdin */
char *gets(char *dest)
{
    int c = getchar();
    char *p = dest;
    while (c != EOF && c != '\n') {
        *p++ = c;
        c = getchar();
    }
    *p = '\0';
    return dest;
}
```

  - No way to specify limit on number of characters to read
- **Similar problems with other library functions**
  - **`strcpy`, `strcat`**: Copy strings of arbitrary length
  - **`scanf`, `fscanf`, `sscanf`,** when given **`%s`** conversion specification

gets has a very bad reputation

```
GETS(3)                    Linux Programmer's Manual                    GETS(3)

NAME
       gets - get a string from standard input (DEPRECATED)

SYNOPSIS
       #include <stdio.h>

       char *gets(char *s);

DESCRIPTION
       Never use this function.

       gets()  reads  a  line from stdin into the buffer pointed to by s until
       either a terminating newline or EOF, which it replaces with a null byte
       ('\0').  No check for buffer overrun is performed (see BUGS below).

RETURN VALUE
       gets()  returns  s  on  success,  and NULL on error or when end of file
       occurs while no characters have been read.  However, given the lack  of
       buffer  overrun  checking, there can be no guarantees that the function
       will even return.

ATTRIBUTES
       For  an  explanation  of  the  terms  used  in  this  section,  see
       attributes(7).

       +-----------+-----------+---------+
       | Interface | Attribute | Value   |
       +-----------+-----------+---------+
```

Vulerable buffer code

```
/* Echo Line */
void echo()
{
    char buf[4];  /* Way too small! */
    gets(buf);
    puts(buf);
}
```

btw, how big is big enough?

```
void call_echo() {
    echo();
}
```

```
unix>./bufdemo-nsp
Type a string:012345678901234567890123
012345678901234567890123
```

```
unix>./bufdemo-nsp
Type a string:0123456789012345678901234
Segmentation Fault
```

Buffer overflow disassembly

echo:

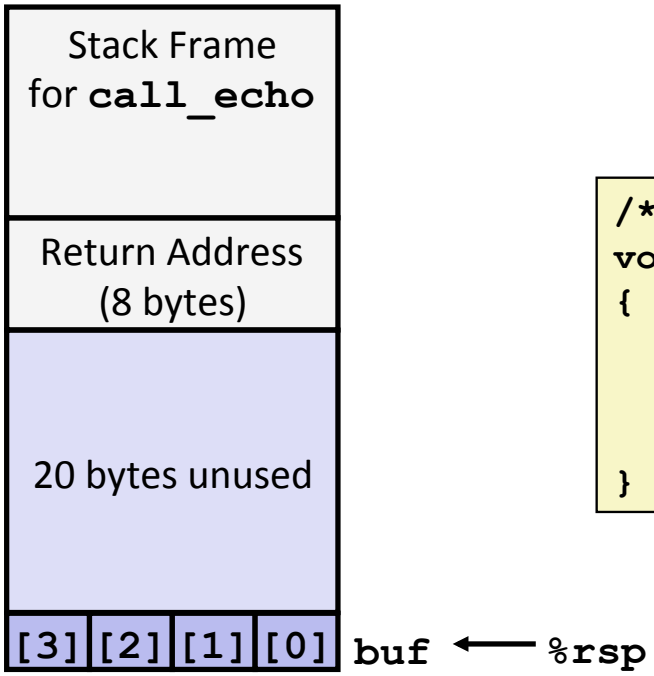
```
00000000004006cf <echo>:
 4006cf:  48 83 ec 18           sub    $0x18,%rsp
 4006d3:  48 89 e7              mov    %rsp,%rdi
 4006d6:  e8 a5 ff ff ff        callq  400680 <gets>
 4006db:  48 89 e7              mov    %rsp,%rdi
 4006de:  e8 3d fe ff ff        callq  400520 <puts@plt>
 4006e3:  48 83 c4 18           add    $0x18,%rsp
 4006e7:  c3                    retq
```

call_echo:

```
 4006e8:   48 83 ec 08          sub    $0x8,%rsp
  4006ec:  b8 00 00 00 00       mov    $0x0,%eax
  4006f1:  e8 d9 ff ff ff       callq  4006cf <echo>
  4006f6:  48 83 c4 08          add    $0x8,%rsp
  4006fa:  c3                   retq
```

Buffer overflow stack

Before call to gets

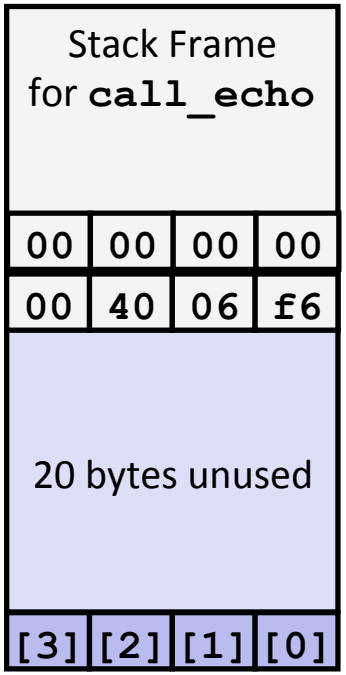



```
/* Echo Line */
void echo()
{
    char buf[4];  /* Way too small! */
    gets(buf);
    puts(buf);
}
```

```
echo:
  subq  $24, %rsp
  movq  %rsp, %rdi
  call  gets
  . . .
```

Buffer overflow stack: example

Before call to gets




```
void echo()
{
    char buf[4];
    gets(buf);
    . . .
}
```

```
echo:
  subq  $24, %rsp
  movq  %rsp, %rdi
  call  gets
  . . .
```

call_echo:

```
  . . .
  4006f1:  callq  4006cf <echo>
  4006f6:  add    $0x8,%rsp
  . . .
```

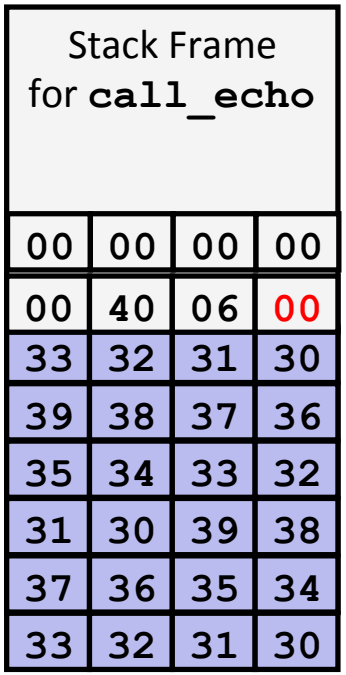
After call to gets


```
void echo()
{
    char buf[4];
    gets(buf);
```

```
echo:
  subq  $24, %rsp
  movq  %rsp, %rdi
  call  gets
```

Buffer overflow stack: example

After call to gets



register_tm_clones:

```
. . .
400600:  mov    %rsp,%rbp
400603:  mov    %rax,%rdx
400606:  shr    $0x3f,%rdx
40060a:  add    %rdx,%rax
40060d:  sar    %rax
400610:  jne    400614
400612:  pop    %rbp
400613:  retq
```

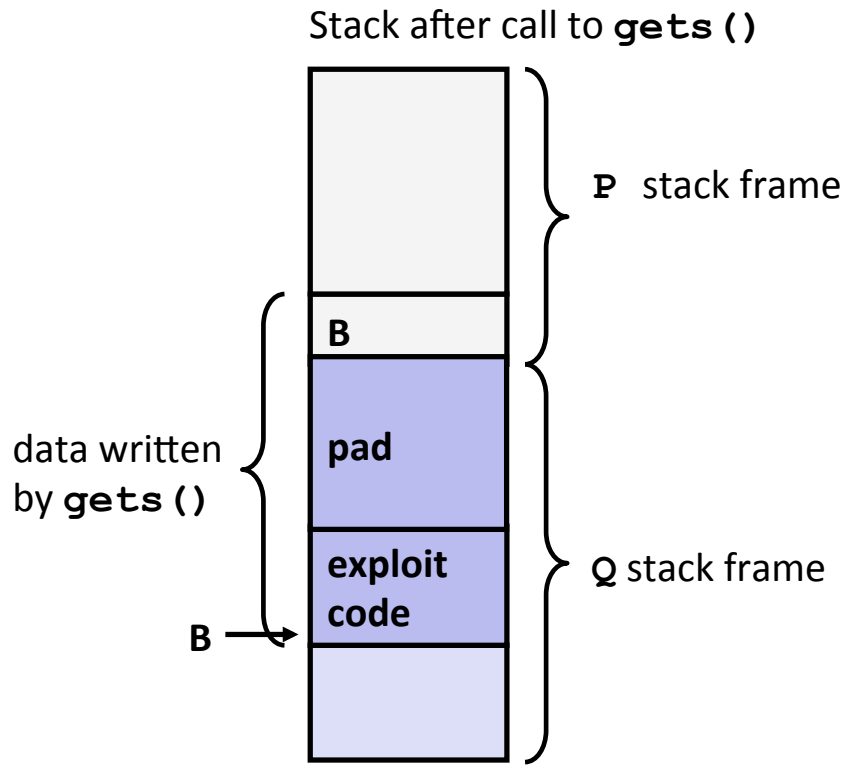
"Returns" to unrelated code
Lots of things happen, without modifying critical state
Eventually executes `retq` back to `main`

Code Injection Attacks

```
void P(){
  Q();
  ...
}
```

return address A

```
int Q() {
  char buf[64];
  gets(buf);
  ...
  return ...;
}
```



- **Input string contains byte representation of executable code**
- **Overwrite return address A with address of buffer B**
- **When Q executes `ret`, will jump to exploit code**

Exploits based on buffer overflows

- ***Buffer overflow bugs can allow remote machines to execute arbitrary code on victim machines***
- **Distressingly common in real progams**
  - Programmers keep making the same mistakes ☹
  - Recent measures make these attacks much more difficult
- **Examples across the decades**
  - Original “Internet worm” (1988)
  - “IM wars” (1999)
  - Twilight hack on Wii (2000s)
  - … and many, many more
- **You will learn some of the tricks in attacklab**
  - Hopefully to convince you to never leave such holes in your programs!!

Example: "Morris worm" (1988)

- First Internet-based malware
  (spread via network using buffer overflow exploits)

- Used several vulnerabilities to spread
  - `gets()` called in some UNIX services (fingerd)
    these services take input from users
    ... and with certain crafted input, would execute root shell
- Once the worm takes over a system, it scans the network for other computers to attack
  - Morris worm took over an estimated 6000 computers
    (10% of the internet at the time!)
  - Described in June 1989 article in *Communications of the ACM*

"Morris worm" gets widely studied (1989)

THE INTERNET WORM

Crisis and Aftermath

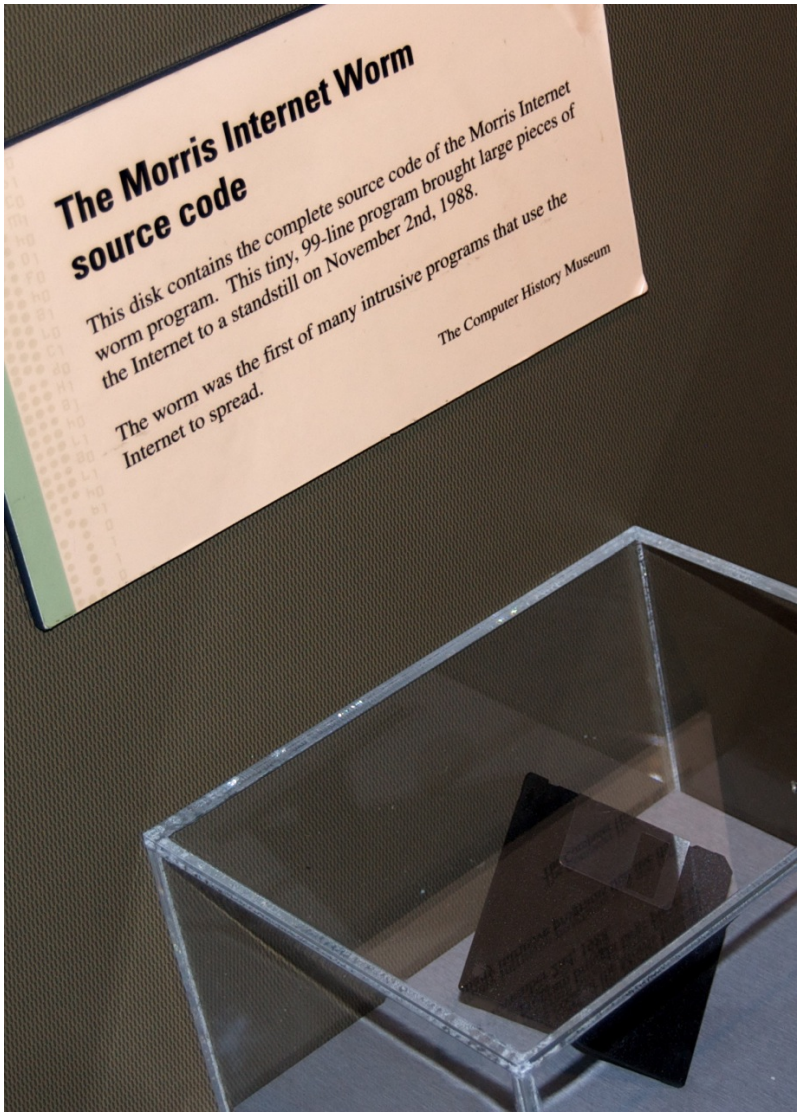
Last November the Internet was infected with a worm program that eventually spread to thousands of machines, disrupting normal activities and Internet connectivity for many days. The following article examines just how this worm operated.

Eugene H. Spafford

On the evening of November 2, 1988 the Internet came under attack from within. Sometime after 5 p.m.,[1] a program was executed on one or more hosts connected to the Internet. That program collected host, network, and user information, then used that information to break into other machines using flaws present in those systems' software. After breaking in, the program would replicate itself and the replica would attempt to infect other systems in the same manner.

Although the program would only infect Sun Microsystems' Sun 3 systems and VAX® computers running variants of 4 BSD UNIX,® the program spread quickly, as did the confusion and consternation of system administrators and users. Initial speculation was that the program was somehow tampering with system resources in a way that could not be readily detected—that while a cure was being sought, system files were being altered or information destroyed. By 5 a.m. Thursday morning, less than 12 hours after the program was first discovered on the network, the Computer Systems Research Group at Berkeley had developed an interim set of steps to halt its spread. This included a preliminary patch to the *sendmail* mail agent. The suggestions were published in mailing lists and on the Usenet, although their spread was hampered by systems disconnecting from the Internet to attempt a "quarantine."

“Morris worm” goes down in history



Morris worm consequences

Computer Intruder Is Put on Probation And Fined $10,000

By JOHN MARKOFF, Special to The New York Times
Published: May 5, 1990

SYRACUSE, May 4— Saying the punishment of prison did not fit the crime, a Federal judge today placed a 24-year-old computer science student on three years' probation for intentionally disrupting a nationwide computer network. The student, Robert Tappan Morris, was also fined $10,000 and ordered to perform 400 hours of community service.

The sentencing of Mr. Morris had been awaited with great interest by computer security experts and those who try to evade them.

The case, which began when Mr. Morris wrote a program that copied itself wildly in thousands of separate machines in November 1988, has become a symbol of the vulnerabilities of the computer networks that serve as the nation's highways in the age of instant information.

Legal experts said the Government's decision to prosecute Mr. Morris, after an eight-month debate in the Justice Department, sent a strong message that tampering with computers, even when not intentionally destructive, was not acceptable. When Mr. Morris was found guilty last January, he became the first person convicted by a jury under the Federal Computer Fraud and Abuse Act of 1986.

worm vs. virus

- Worm – a program that:
  - can run by itself
  - can propagate a full working version of itself to other computers
- Virus – code that:
  - adds itself to other programs
  - does not run independently
- Both are designed to spread on their own

1990s: the word gets out

- 1995: "How to write buffer overflows" published
- 1996: First *complete*, public explanation of buffer overflows: "Smashing The Stack For Fun And Profit"

.oO Phrack 49 Oo.

Volume Seven, Issue Forty-Nine

File 14 of 16

BugTraq, r0Ot, and Underground.Org
bring you

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
Smashing The Stack For Fun And Profit
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

by Aleph One
aleph1@underground.org

`smash the stack` [C programming] n. On many C implementations it is possible to corrupt the execution stack by writing past the end of an array declared auto in a routine. Code that does this is said to smash the stack, and can cause return from the routine to jump to a random address. This can produce some of

Later worms

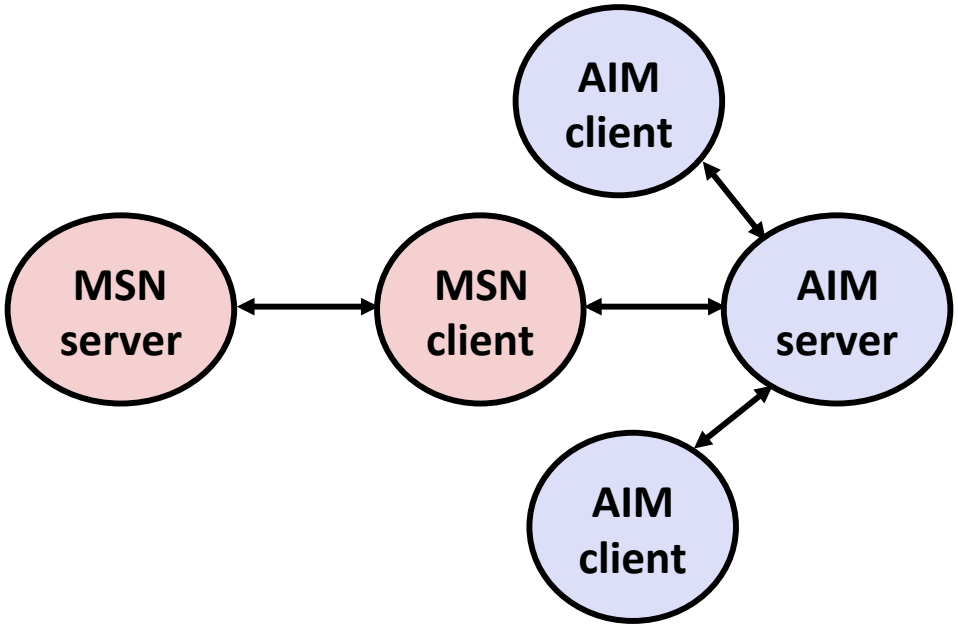
Buffer overflow a common tactic for malware

- 2001: Code Red worm
  - buffer overflow in Microsoft IIS
- 2003: SQL Slammer
  - buffer overflow MS SQL server
  - hit 75,000 victims within 10 minutes
  - 376 bytes

Worms are not the only (in)famous uses of buffer overflows...

IM wars

- **July, 1999**
  - Microsoft launches MSN Messenger (instant messaging system).
  - Messenger clients can access popular AOL Instant Messaging Service (AIM) servers



IM war hits NY Times page A1[1]

July 24, 1999

In Cyberspace, Rivals Skirmish Over Messaging

By SAUL HANSELL

America Online closed its on-line service yesterday to new software from two of its fiercest rivals, Microsoft and Yahoo, that had been designed to tap into one of America Online's most popular features: instant messages.

On Thursday, both the Microsoft Corporation and Yahoo introduced

IM wars: behind the scenes

- **August 1999**
  - Mysteriously, Messenger clients can no longer access AIM servers
  - Microsoft and AOL begin the IM war:
    - AOL changes server to disallow Messenger clients
    - Microsoft makes changes to clients to defeat AOL changes
    - At least 13 such skirmishes
  - What was really happening?
    - AOL had discovered a buffer overflow bug in their own AIM clients
    - They exploited it to detect and block Microsoft: the exploit code returned a 4-byte signature (the bytes at some location in the AIM client) to server
    - When Microsoft changed code to match signature, AOL changed signature location

IM wars

From: Phil Bucking <philbucking@yahoo.com>
Subject: AOL exploiting buffer overrun bug in their own software!
To: rms@pharlap.com

Mr. Smith,

I am writing you because I have discovered something that I think you might find interesting because you are an Internet security expert with experience in this area. I have also tried to contact AOL but received no response.

I am a developer who has been working on a revolutionary new instant messaging client that should be released later this year.
...
It appears that the AIM client has a buffer overrun bug. By itself this might not be the end of the world, as MS surely has had its share. But AOL is now *exploiting their own buffer overrun bug* to help in its efforts to block MS Instant Messenger.
....
Since you have significant credibility with the press I hope that you can use this information to help inform people that behind AOL's friendly exterior they are nefariously compromising peoples' security.

It appears that the AIM client has a buffer overrun bug. By itself this might not be the end of the world, as MS surely has had its share. But AOL is now *exploiting their own buffer overrun bug* to help in its efforts to block MS Instant Messenger.

Twilight hack (2008)[2]

- First method of running "homebrew" apps on Nintendo Wii
- Using a special saved-game file with a custom name for Epona (Link's horse) containg exploit code (code injection)

[2] http://www.wiibrew.org/wiki/Twilight_Hack

iPhone jailbreaks (2007-present)

- Apple locks down iPods / iPhones / etc
- Why? Restrict apps to iTunes store, lock devices to carriers, prevent malware
- Device owners try to circumvent "jail"
- Why? Run other apps, evade censorship, customize OS, unlock device, etc

iPhone jailbreaks[3]

- How? Many techniques
- Buffer overruns and integer overflows are common
- Apple's response? Issue OS update to prevent jailbreak (patch vulnerabilities)
- So new vulnerabilities are found, new jailbreak released

Apple plugs critical iPhone jailbreak holes

Reuters Staff

4 MIN READ

Apple today patched the two vulnerabilities used to jailbreak Apple's newest iOS 4 operating system, bugs that security researchers warned could be used to hijack iPhones , iPod Touches or iPads .

The patches came just 10 days after a group published a site that automatically exploited and then jailbroke any iOS 4 device that used the mobile Safari browser to surf to jailbreakme.com

Also last week, other researchers confirmed that the first exploit of the pair leveraged a flaw in Safari's parsing of fonts in PDF documents to compromise the browser. A second vulnerability was exploited to break out of the isolating "sandbox" and gain full, or "root,"

0. Class updates
1. Memory layout
2. Buffer Overflow
   - Vulnerability
   - History
   - Protection
     - Bug-free code?
     - ASLR and NX
     - Stack canaries
   - Arms race!
     - Return-oriented programming (ROP)
3. Attacklab (lab4)

Defense 1: avoid overflow vulerabilities when writing programs

```
/* Echo Line */
void echo()
{
    char buf[4];  /* Way too small! */
    fgets(buf, 4, stdin);
    puts(buf);
}
```

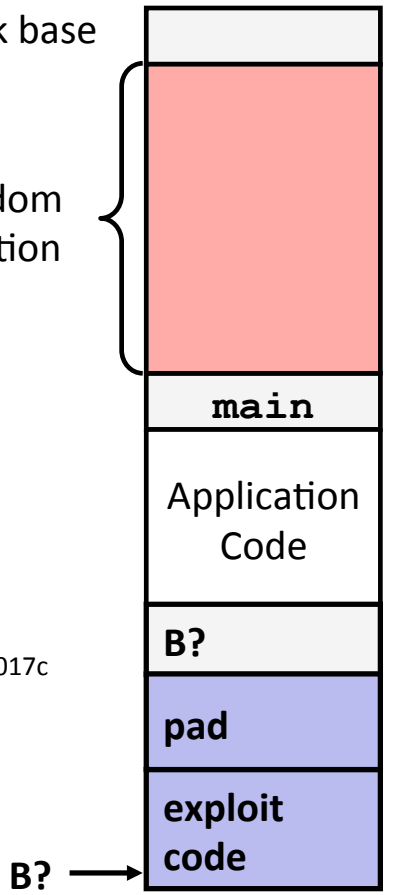
- **For example, use library routines that limit string lengths**
  - **`fgets`** instead of **`gets`**
  - **`strncpy`** instead of **`strcpy`**
  - Don't use **`scanf`** with **`%s`** conversion specification
    - Use **`fgets`** to read the string
    - Or use **`%ns`** where **`n`** is a suitable integer

Defense 2: System-level protections
Randomized stack offsets

- **Randomized stack offsets**
  - At start of program, allocate random amount of space on stack
  - Shifts stack addresses for entire program
  - Makes it difficult for hacker to predict beginning of inserted code
  - E.g.: 5 executions of memory allocation code

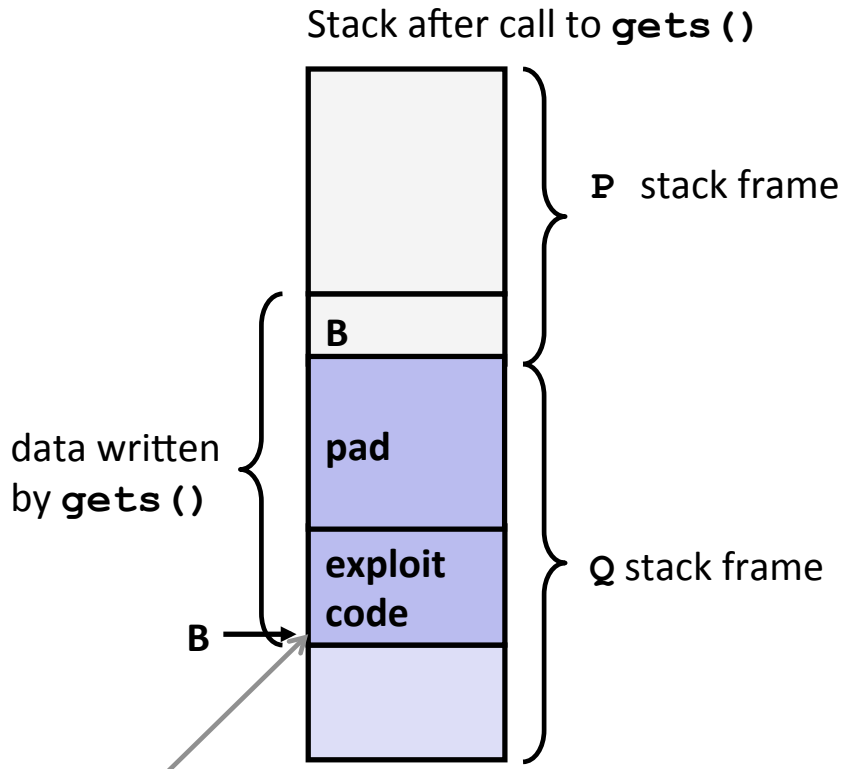
local	0x7ffe4d3be87c	0x7fff75a4f9fc	0x7ffeadb7c80c	0x7ffeaea2fdac	0x7ffcd452017c

    - Stack repositioned each time program executes



Defense 2: System-level protections
Non-executable stack

- **Nonexecutable code segments**
  - In traditional x86, can mark region of memory as either "read-only" or "writeable"
    - Can execute anything readable
  - X86-64 added explicit "execute" permission
  - Stack marked as non-executable



Any attempt to execute this code will fail

Canary in a coal mine

Source: U.S. State Department/Doug Thompson

Defense 3: Stack Canaries

- **Idea**
  - Place special value ("canary") on stack just beyond buffer
  - Check for corruption before exiting function
- **GCC Implementation**
  - `-fstack-protector`
  - Now the default (disabled earlier)

```
unix>./bufdemo-sp
Type a string:0123456
0123456
```

```
unix>./bufdemo-sp
Type a string:01234567
*** stack smashing detected ***
```

Defense 3: Canary example

Disassembly shows additional instructions in every function

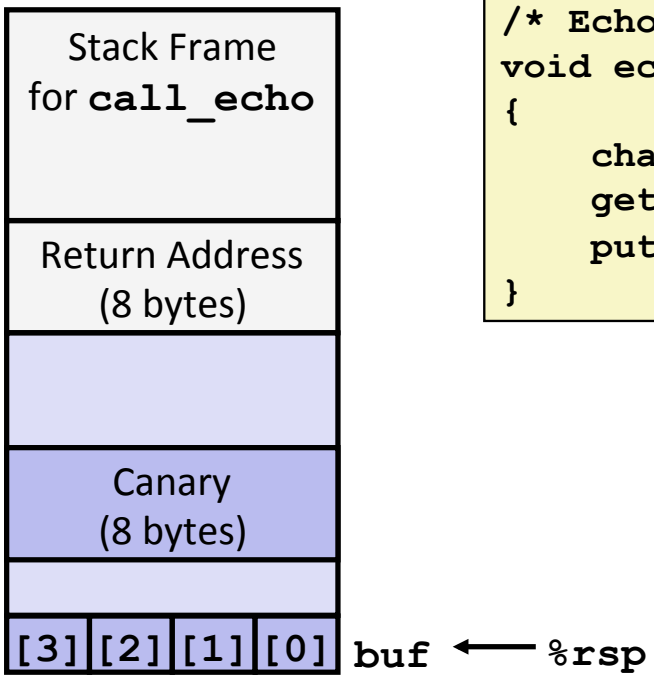
echo:

```
40072f:  sub    $0x18,%rsp
400733:  mov    %fs:0x28,%rax
40073c:  mov    %rax,0x8(%rsp)
400741:  xor    %eax,%eax
400743:  mov    %rsp,%rdi
400746:  callq  4006e0 <gets>
40074b:  mov    %rsp,%rdi
40074e:  callq  400570 <puts@plt>
400753:  mov    0x8(%rsp),%rax
400758:  xor    %fs:0x28,%rax
400761:  je     400768 <echo+0x39>
400763:  callq  400580 <__stack_chk_fail@plt>
400768:  add    $0x18,%rsp
40076c:  retq
```

Defense 3: Setting up Canary

Placing canary value onto stack

Before call to gets



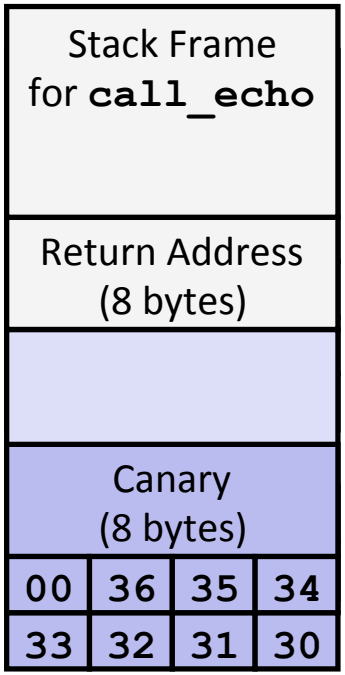

```
/* Echo Line */
void echo()
{
    char buf[4];  /* Way too small! */
    gets(buf);
    puts(buf);
}
```

```
echo:
    . . .
    movq    %fs:40, %rax  # Get canary
    movq    %rax, 8(%rsp) # Place on stack
    xorl    %eax, %eax    # Erase canary
    . . .
```

Defense 3: Checking Canary

Compare value on stack to original immediately before returning

After call to gets




```
/* Echo Line */
void echo()
{
    char buf[4];  /* Way too small! */
    gets(buf);
    puts(buf);
}
```

Input: *0123456*

```
echo:
    . . .
    movq    8(%rsp), %rax     # Retrieve from stack
    xorq    %fs:40, %rax      # Compare to canary
    je      .L6               # If same, OK
    call    __stack_chk_fail  # FAIL
.L6:        . . .
```

Defense 4: Safer programming languages

- Some languages offer *memory safety*
  - usually means no pointers or explicit allocate/free
- Another important feature: *type safety*
  - strict enforcement of data types
  - type-checking prevents many common errors
  - but usually means no casting, disables other low-level operations
- Many classes of errors are preventable but not all!
  - errors in program logic are always possible

- Class updates
- 1 Memory layout
- 2 Buffer Overflow
  - Vulnerability
  - History
  - Protection
    - Bug-free code?
    - ASLR and NX
    - Stack canaries
  - Arms race!
    - Return-oriented programming (ROP)
- 3 Attacklab (lab4)

Response: return-oriented programming (ROP) attacks

- **Challenge (for hackers)**
  - Stack randomization makes it hard to predict buffer location
  - Marking stack nonexecutable makes it hard to insert binary code
- **Alternative Strategy**
  - Use existing code
    - E.g., library code from stdlib
  - String together fragments to achieve overall desired outcome
  - *Does not overcome stack canaries*
- **Construct program from *gadgets***
  - Sequence of instructions ending in **`ret`**
    - Encoded by single byte **`0xc3`**
  - Code positions fixed from run to run
  - Code is executable

ROP: gadget example #1

Use tail end of existing functions

```
long ab_plus_c
   (long a, long b, long c)
{
    return a*b + c;
}
```

```
00000000004004d0 <ab_plus_c>:
  4004d0:  48 0f af fe  imul %rsi,%rdi
  4004d4:  48 8d 04 17  lea (%rdi,%rdx,1),%rax
  4004d8:  c3           retq
```

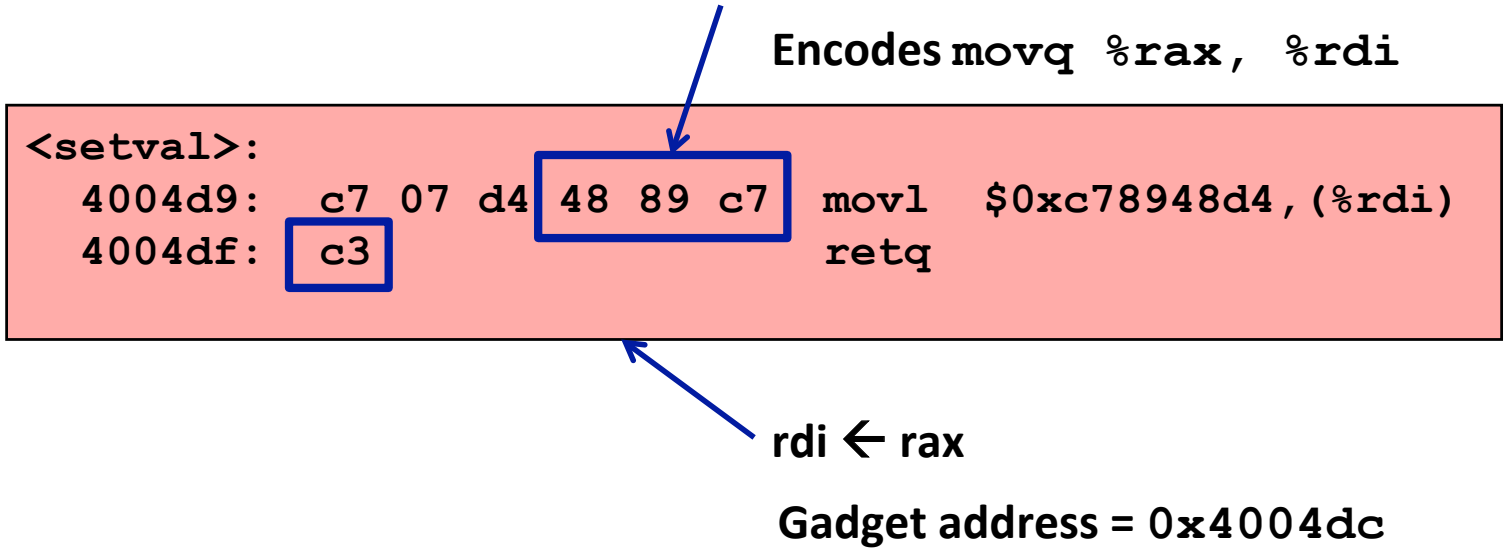
rax ← rdi + rdx

Gadget address = 0x4004d4

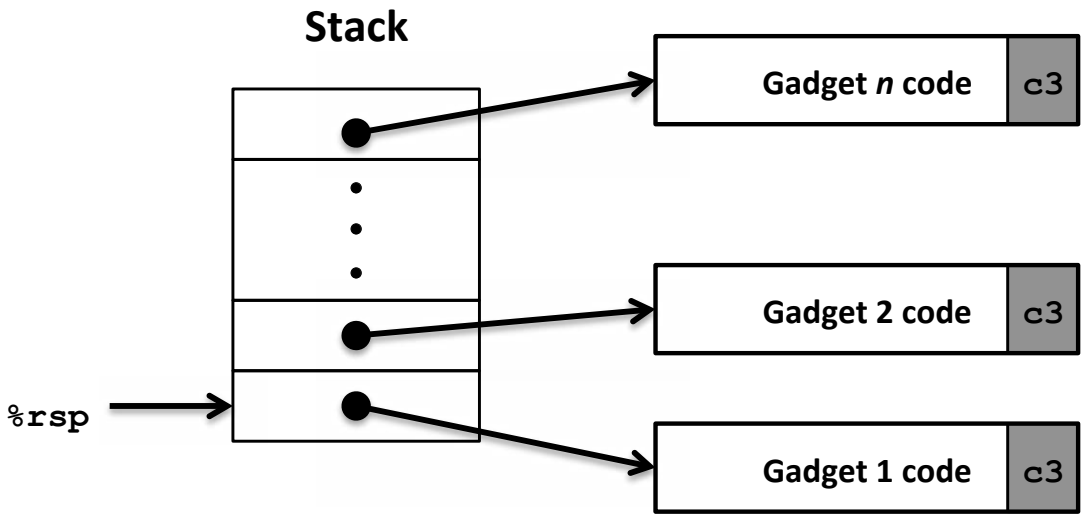
ROP: gadget example #2

Repurpose opcode bytes within longer instruction

```
void setval(unsigned *p) {
    *p = 3347663060u;
}
```



ROP execution



- **Trigger with `ret` instruction**
  - Will start executing Gadget 1
- **Final `ret` in each gadget will start next one**

Return-oriented programming
. . . can defeat some defenses but not all

Key observations:

- randomized stack offsets can make code injection difficult or impossible
- non-executable stack can defeat code injection attacks
- ROP can defeat both of these defenses because it does not depend on code injection, instead re-uses existing code
- stack canaries can prevent both code injection attacks and ROP attacks, because they prevent overwriting the return address
- . . . but stack canaries are not invincible either!
- Want to know a way to defeat stack canaries? Check out this excellent lecture by Prof. James Mickens (via MIT OCW, graduate level Computer Security course)

- Class updates
- 1 Memory layout
- 2 Buffer Overflow
  - Vulnerability
  - History
  - Protection
    - Bug-free code?
    - ASLR and NX
    - Stack canaries
  - Arms race!
    - Return-oriented programming (ROP)
- 3 Attacklab (lab4)

Attacklab

- First 3 phases of lab: code-injection attacks on `ctarget`
- Next 2 phases: return-oriented programming attacks on `rtarget`
- 50 points possible (5 phases)