CS 105
Tour of Black Holes of Computing

Machine-Level Programming V: Miscellaneous Topics

Topics

- Linux Memory Layout
- Buffer Overflow
- C operators and declarations

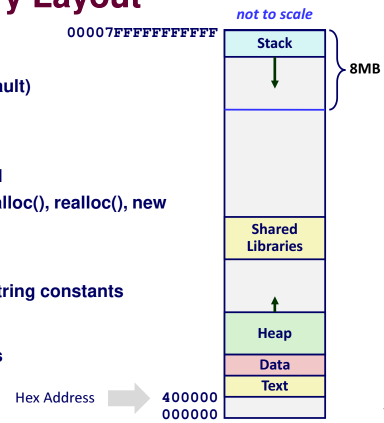
x86-64 Linux Memory Layout

Stack
- Runtime stack (8MB limit by default)
- E.g., local variables

Heap
- Dynamically allocated as needed
- When programs call malloc(), calloc(), realloc(), new

Data
- Statically allocated data
- E.g., global vars, `static` vars, string constants

Text / Shared Libraries
- Executable machine instructions
- Read-only



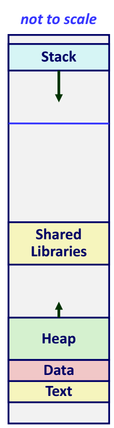
Memory Allocation Example

```
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 ... */
}
```

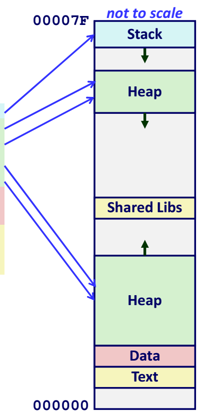


Where does everything go?

x86-64 Example Addresses

address range ~2^{47}

local	0x00007ffe4d3be87c
p1	0x00007f7262a1e010
p3	0x00007f7162a1d010
p4	0x000000008359d120
p2	0x000000008359d010
big_array	0x0000000080601060
huge_array	0x0000000000601060
global	0x0000000000400a28
main()	0x000000000040060c
useless()	0x0000000000400590



Note: very much **not** to scale!

Memory-Referencing Bug Example

```
typedef struct {       /* An "anonymous" structure */
  int a[2];
  double d;
} struct_t;            /* "typedef" gives it a type name */

double fun(int i)
{
  volatile struct_t s;
  s.d = 3.14;
  s.a[i] = 1073741824; /* 2**30, possibly out of bounds */
  return s.d;
}
```

```
fun(0)  ⇒   3.14
fun(1)  ⇒   3.14
fun(2)  ⇒   3.1399998664856
fun(3)  ⇒   2.00000061035156
fun(4)  ⇒   3.14
fun(6)  ⇒   Segmentation fault
```

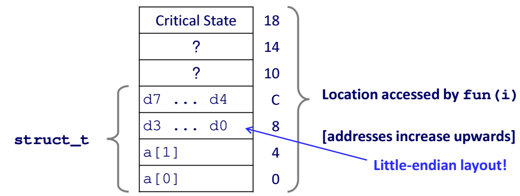
- Result is system-specific

Memory-Referencing Bug Example

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

```
fun(0)  ⇒   3.14
fun(1)  ⇒   3.14
fun(2)  ⇒   3.1399998664856
fun(3)  ⇒   2.00000061035156
fun(4)  ⇒   3.14
fun(6)  ⇒   Segmentation fault
```

Explanation:



Such problems are a BIG deal

Generally called a "buffer overflow"

- When exceeding the memory size allocated for an array

Why a big deal?

- It's the #1 technical cause of security vulnerabilities
  - #1 overall cause is social engineering / user ignorance

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

Vulnerable 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
Type a string:01234567890123456789012
01234567890123456789012
```

```
unix>./bufdemo
Type a string:012345678901234567890123
Segmentation Fault
```


Buffer Overflow Disassembly

echo:

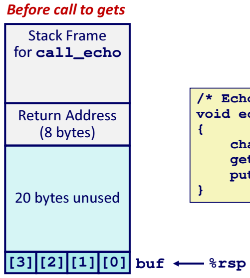
```
00000000004006cf <echo>:
 4006cf:  48 83 ec 18          sub    $0x18,%rsp
 4006d3:  48 89 e7             mov    %rsp,%rdi
 4006d6:  e8 a5 ff ff ff       callq  400680 <gets>
 4006db:  48 89 e7             mov    %rsp,%rdi
 4006de:  e8 3d fe ff ff       callq  400520 <puts@plt>
 4006e3:  48 83 c4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

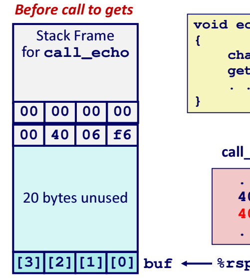



```
/* Echo Line */
void echo()
{
    char buf[4];   /* Way too small! */
    gets(buf);
    puts(buf);
}
```

```
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
  . . .
```

Buffer Overflow Example #1

After call to gets

Stack Frame for `call_echo`				
00	00	00	00	
00	40	06	f6	
00	32	31	30	
39	38	37	36	
35	34	33	32	
31	30	39	38	
37	36	35	34	
33	32	31	30	buf ⟵ %rsp

```
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
 . . .
```

```
unix>./bufdemo
Type a string:01234567890123456789012
01234567890123456789012
```

Overflowed buffer, but did not corrupt state

Buffer Overflow Example #2

After call to gets

Stack Frame for `call_echo`				
00	00	00	00	
00	40	00	34	
33	32	31	30	
39	38	37	36	
35	34	33	32	
31	30	39	38	
37	36	35	34	
33	32	31	30	buf ⟵ %rsp

```
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
 . . .
```

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

Overflowed buffer and corrupted return pointer

Buffer Overflow Example #3

After call to gets

Stack Frame for `call_echo`				
00	00	00	00	
00	40	06	00	
33	32	31	30	
39	38	37	36	
35	34	33	32	
31	30	39	38	
37	36	35	34	
33	32	31	30	buf ⟵ %rsp

```
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
 . . .
```

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

Overflowed buffer, corrupted return pointer, but program seems to work!

Buffer Overflow Example #3 Explained

After call to gets

Stack Frame for `call_echo`				
00	00	00	00	
00	40	06	00	
33	32	31	30	
39	38	37	36	
35	34	33	32	
31	30	39	38	
37	36	35	34	
33	32	31	30	buf ⟵ %rsp

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`
Basis of "return-oriented programming" (ROP)

Exploits Based on 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 lab 4
- Hopefully to convince you to never leave such holes in your programs!!

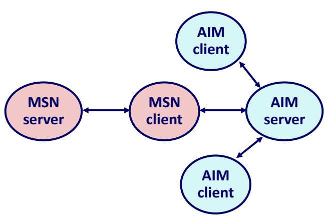
Example: Original Internet Worm (1988)

Exploited a few vulnerabilities to spread
- Early versions of the finger server (fingerd) used `gets()` to read the argument sent by the client:
  - `finger geoff@cs.hmc.edu`
- Worm attacked fingerd server by sending phony argument:
  - `finger "exploit-code padding new-return-address"`
  - exploit code: executed a root shell on the victim machine with a direct TCP connection to the attacker.

Once on a machine, scanned for other machines to attack
- invaded ~6000 computers in hours (10% of the Internet ☺)
  - see June 1989 article in *Comm. of the ACM*
- the young author of the worm was prosecuted…
- and CERT was formed… still homed at CMU

Example 2: IM War

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



IM War (cont.)

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

```
Date: Wed, 11 Aug 1999 11:30:57 -0700 (PDT)
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.

Sincerely,
Phil Bucking
Founder, Bucking Consulting
philbucking@yahoo.com
```

It was later determined that this email originated from within Microsoft!

Aside: Worms and Viruses

Worm: A program that

- Can run by itself
- Can propagate a fully working version of itself to other computers

Virus: Code that

- Adds itself to other programs
- Does not run independently

Both are (usually) designed to spread among computers and to wreak havoc

OK, What to Do About Buffer Overflow Attacks?

Avoid overflow vulnerabilities

Employ system-level protections

Have compiler use “stack canaries”

Lets talk about each…

1. Avoid Overflow Vulnerabilities in Code (!)

```
/* 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

2. System-Level Protections Can Help

Randomized stack offsets

- At start of program, allocate random amount of space on stack
- Makes it hard 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

Nonexecutable code segments

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

3. Stack Canaries can help

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-protected
Type a string:0123456
0123456
```

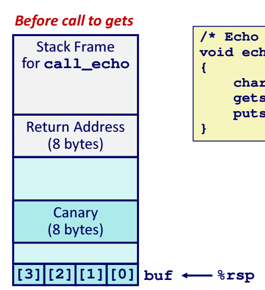
```
unix>./bufdemo-protected
Type a string:01234567
*** stack smashing detected ***
```


Protected Buffer Disassembly

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
```

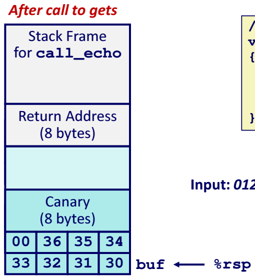

Setting Up Canary




```
/* 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
   . . .
```

Checking Canary




```
/* 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:        . . .
```


C Operators

```
Operators                                    Associativity
()  []  ->  .                                left to right
!  ~  ++  --  +  -  *  & (type) sizeof       right to left
*  /  %                                      left to right
+  -                                         left to right
<<  >>                                       left to right
<  <=  >  >=                                 left to right
==  !=                                       left to right
&                                            left to right
^                                            left to right
|                                            left to right
&&                                           left to right
||                                           left to right
?:                                           right to left
= += -= *= /= %= &= ^= != <<= >>=            right to left
,                                            left to right
```

Note: Unary +, –, and * have higher precedence than binary forms

See `~geoff/c_precedence` on Wilkes and Knuth

C Pointer Declarations

`int *p`	p is a pointer to int
`int *p[13]`	p is an array[13] of pointer to int
`int *(p[13])`	p is an array[13] of pointer to int
`int **p`	p is a pointer to a pointer to an int
`int (*p)[13]`	p is a pointer to an array[13] of int
`int *f()`	f is a function (unknown arguments) returning a pointer to int
`int (*f)()`	f is a pointer to a function returning int
`int (*(*f())[13])()`	f is a function returning ptr to an array[13] of pointers to functions returning int
`int (*(*x[3])())[5]`	x is an array[3] of pointers to functions returning pointers to array[5] of ints