Machine-Level Programming V: Memory Layout and Buffer Overflows

CSci 2021: Machine Architecture and Organization
Lecture #14, February 20th, 2015
Your instructor: Stephen McCamant

Based on slides originally by:
Randy Bryant, Dave O'Hallaron, Antonia Zhai

Today

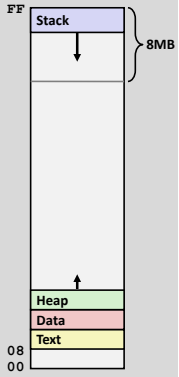
- **Memory Layout**
- **Buffer Overflow**
  - Vulnerability
  - Protection

IA32 Linux Memory Layout

- **Stack**
  - Runtime stack (Ubuntu sets 8MB soft limit)
  - E.g., local variables
- **Heap**
  - Dynamically allocated storage
  - When call malloc(), calloc(), new()
- **Data**
  - Statically allocated data
  - E.g., arrays & strings declared in code
- **Text**
  - Executable machine instructions
  - Read-only

Upper 2 hex digits
= 8 bits of address

not drawn to scale

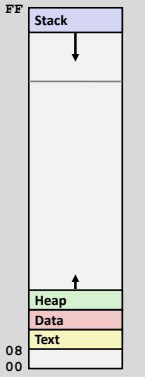


Memory Allocation Example

```
char big_array[1<<24];  /*  16 MB */
char huge_array[1<<28]; /* 256 MB */

int beyond;
char *p1, *p2, *p3, *p4;

int useless() {  return 0; }

int main()
{
 p1 = malloc(1 <<28);  /* 256 MB */
 p2 = malloc(1 << 8);  /* 256 B  */
 p3 = malloc(1 <<28);  /* 256 MB */
 p4 = malloc(1 << 8);  /* 256 B  */
 /* Some print statements ... */
}
```

Where does everything go?

not drawn to scale



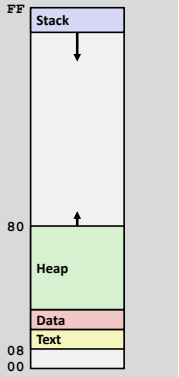
IA32 Example Addresses

address range ~2^{32}

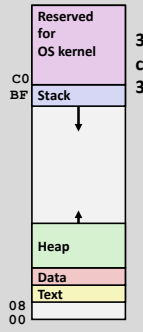
`$esp`	`0xffffbcd0`
`p3`	`0x65586008`
`p1`	`0x55585008`
`p4`	`0x1904a110`
`p2`	`0x1904a008`
`&p2`	`0x18049760`
`&beyond`	`0x08049744`
`big_array`	`0x18049780`
`huge_array`	`0x08049760`
`main()`	`0x080483c6`
`useless()`	`0x08049744`

`malloc()` is dynamically linked
address determined at runtime

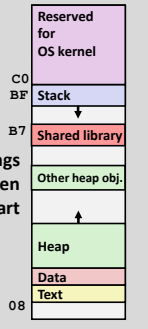
not drawn to scale



IA32 Layout Variations



3GB/1GB split:
common with
32-bit kernel



Other things
placed between
stack and heap start

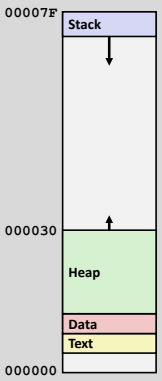
x86-64 Example Addresses

not drawn to scale

address range~2^{47} , rest reserved

`$rsp`	`0x00007fffffff8d1f8`
`p3`	`0x00002aaabaadd010`
`p1`	`0x00002aaaaaadc010`
`p4`	`0x0000000011501120`
`p2`	`0x0000000011501010`
`&p2`	`0x0000000010500a60`
`&beyond`	`0x0000000000500a44`
`big_array`	`0x0000000010500a80`
`huge_array`	`0x0000000000500a50`
`main()`	`0x0000000000400510`
`useless()`	`0x0000000000400500`

`malloc()` is dynamically linked
address determined at runtime



7

Today

- Memory Layout
- **Buffer Overflow**
  - Vulnerability
  - Protection

8

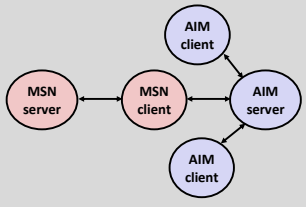
Internet Worm and IM War

- **November, 1988**
  - Internet Worm attacks thousands of Internet hosts.
  - How did it happen?

9

Internet Worm and IM War

- **November, 1988**
  - Internet Worm attacks thousands of Internet hosts.
  - How did it happen?
- **July, 1999**
  - Microsoft launches MSN Messenger (instant messaging system).
  - Messenger clients can access popular AOL Instant Messaging Service (AIM) servers



10

Internet Worm and 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 the final round in the war?
- **The Internet Worm and AOL/Microsoft War were both based on *stack buffer overflow* exploits!**
  - many library functions do not check argument sizes.
  - allows target buffers to overflow.

11

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

12

Vulnerable Buffer Code

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

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

```
unix>./bufdemo
Type a string:1234567
1234567
```

```
unix>./bufdemo
Type a string:12345678
Segmentation Fault
```

```
unix>./bufdemo
Type a string:123456789ABC
Segmentation Fault
```

13

Buffer Overflow Disassembly

echo:

```
80485c5:  55                  push   %ebp
80485c6:  89 e5               mov    %esp,%ebp
80485c8:  53                  push   %ebx
80485c9:  83 ec 14            sub    $0x14,%esp
80485cc:  8d 5d f8            lea 0xfffffff8(%ebp),%ebx
80485cf:  89 1c 24            mov    %ebx,(%esp)
80485d2:  e8 9e ff ff ff      call   8048575 <gets>
80485d7:  89 1c 24            mov    %ebx,(%esp)
80485da:  e8 05 fe ff ff      call   80483e4 <puts@plt>
80485df:  83 c4 14            add    $0x14,%esp
80485e2:  5b                  pop    %ebx
80485e3:  5d                  pop    %ebp
80485e4:  c3                  ret
```

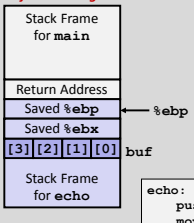
call_echo:

```
80485eb:  e8 d5 ff ff ff      call   80485c5 <echo>
80485f0:  c9                  leave
80485f1:  c3                  ret
```

14

Buffer Overflow Stack

Before call to gets




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

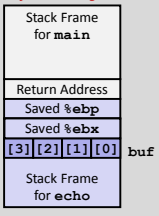
```
echo:
    pushl %ebp              # Save %ebp on stack
    movl  %esp, %ebp
    pushl %ebx              # Save %ebx
    subl  $20, %esp         # Allocate stack space
    leal  -8(%ebp),%ebx     # Compute buf as %ebp-8
    movl  %ebx, (%esp)      # Push buf on stack
    call  gets              # Call gets
    . . .
```

15

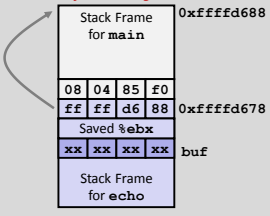
Buffer Overflow Stack Example

```
unix> gdb bufdemo
(gdb) break echo
Breakpoint 1 at 0x80485c9
(gdb) run
Breakpoint 1, 0x80485c9 in echo ()
(gdb) print /x $ebp
$1 = 0xffffd678
(gdb) print /x *(unsigned *)$ebp
$2 = 0xffffd688
(gdb) print /x *((unsigned *)$ebp + 1)
$3 = 0x80485f0
```

Before call to gets



Before call to gets

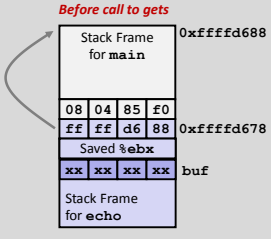



```
80485eb:  e8 d5 ff ff ff    call   80485c5 <echo>
80485f0:  c9                leave
```

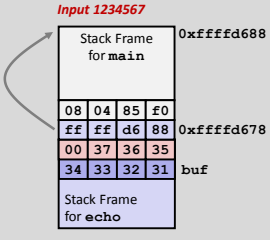
16

Buffer Overflow Example #1

Before call to gets



Input 1234567

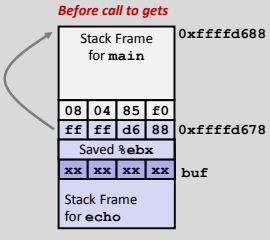


Overflow buf, and corrupt %ebx, but no crash

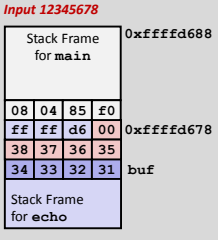
17

Buffer Overflow Example #2

Before call to gets



Input 12345678

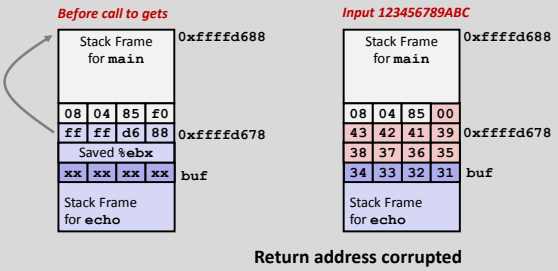


Frame pointer corrupted

```
. . .
80485eb:   e8 d5 ff ff ff   call   80485c5 <echo>
80485f0:   c9               leave    # Set %ebp to corrupted value
80485f1:   c3               ret
```

18

Buffer Overflow Example #3

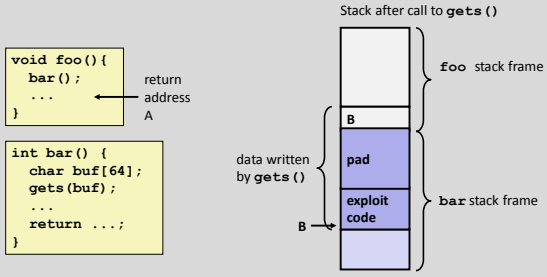


Return address corrupted

```
80485eb:    e8 d5 ff ff ff    call   80485c5 <echo>
80485f0:    c9                leave  # Desired return point
```

19

Malicious Use of Buffer Overflow



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

20

Discussion Break: Unknown Addresses?

- Basic attack requires attacker to know address B of buffer
- Is an attack still possible if B is variable?
- E.g. what if attacker only knows B +/- 30?

- Some possible attack strategies:
  - Try attack repeatedly
  - "NOP sled": (0x90 is one-byte no-operation in IA32)

NOP NOP NOP NOP NOP NOP NOP NOP NOP NOP NOP NOP Exploit Code

21

Exploits Based on Buffer Overflows

- *Buffer overflow bugs allow remote machines to execute arbitrary code on victim machines*
- **Internet worm**
  - Early versions of the finger server (fingerd) used `gets()` to read the argument sent by the client:
    - `finger droh@cs.cmu.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.

22

Exploits Based on Buffer Overflows

- *Buffer overflow bugs allow remote machines to execute arbitrary code on victim machines*
- **IM War**
  - AOL exploited existing buffer overflow bug in AIM clients
  - exploit code: returned 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.

23

```
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
```

This email originated from within Microsoft; the employee was "disciplined"

24

Avoiding Overflow Vulnerability

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

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

System-Level Protections

- **Randomized stack offsets**
  - At start of program, allocate random amount of space on stack
  - Makes it difficult for hacker to predict beginning of inserted code
  - Modern version: address space layout randomization "ASLR"
- **Nonexecutable data segments**
  - In traditional x86, can mark region of memory as either "read-only" or "writeable"
    - Can execute anything readable
  - More recent processors added explicit way to disable "execute" permission, e.g. for stack

```
unix> gdb bufdemo
(gdb) break echo

(gdb) run
(gdb) print /x $ebp
$1 = 0xffffc638

(gdb) run
(gdb) print /x $ebp
$2 = 0xffffbb08

(gdb) run
(gdb) print /x $ebp
$3 = 0xffffc6a8
```


Stack Canaries

- **Idea**
  - Place special value ("canary") on stack just beyond buffer
  - Check for corruption before exiting function
- **GCC Implementation**
  - **-fstack-protector**
  - **-fstack-protector-all**

```
unix>./bufdemo-protected
Type a string:1234
1234
```

```
unix>./bufdemo-protected
Type a string:12345
*** stack smashing detected ***
```

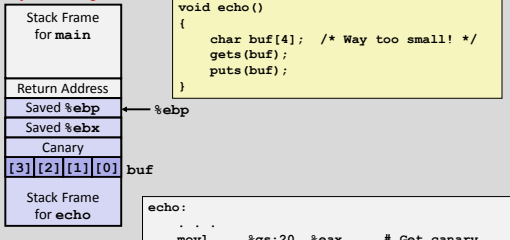

Protected Buffer Disassembly

echo:

```
804864d:   55                      push   %ebp
804864e:   89 e5                   mov    %esp,%ebp
8048650:   53                      push   %ebx
8048651:   83 ec 14                sub    $0x14,%esp
8048654:   65 a1 14 00 00 00       mov    %gs:0x14,%eax
804865a:   89 45 f8                mov    %eax,0xfffffff8(%ebp)
804865d:   31 c0                   xor    %eax,%eax
804865f:   8d 5d f4                lea    0xfffffff4(%ebp),%ebx
8048662:   89 1c 24                mov    %ebx,(%esp)
8048665:   e8 77 ff ff ff         call   80485e1 <gets>
804866a:   89 1c 24                mov    %ebx,(%esp)
804866d:   e8 ca fd ff ff         call   804843c <puts@plt>
8048672:   8b 45 f8                mov    0xfffffff8(%ebp),%eax
8048675:   65 33 05 14 00 00 00    xor    %gs:0x14,%eax
804867c:   74 05                   je     8048683 <echo+0x36>
804867e:   e8 a9 fd ff ff         call   804842c <FAIL>
8048683:   83 c4 14                add    $0x14,%esp
8048686:   5b                      pop    %ebx
8048687:   5d                      pop    %ebp
8048688:   c3                      ret
```


Setting Up Canary

Before call to gets

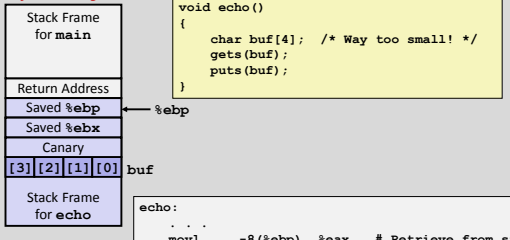



```
/* Echo Line */
void echo()
{
    char buf[4];  /* Way too small! */
    gets(buf);
    puts(buf);
}
```

```
echo:
    . . .
    movl    %gs:20, %eax    # Get canary
    movl    %eax, -8(%ebp)  # Put on stack
    xorl    %eax, %eax      # Erase canary
    . . .
```


Checking Canary

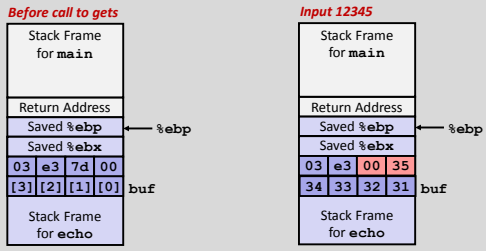
Before call to gets




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

```
echo:
    . . .
    movl    -8(%ebp), %eax   # Retrieve from stack
    xorl    %gs:20, %eax     # Compare with Canary
    je      .L24             # Same: skip ahead
    call    __stack_chk_fail # ERROR
.L24:
    . . .
```

Canary Example




```
(gdb) break echo
(gdb) run
(gdb) stepi 3
(gdb) print /x *((unsigned *) $ebp - 2)
$1 = 0x3e30035
```

(The canary always ends with a zero byte. Why is this a good idea?)

32

Today

- **Memory Layout**
- **Buffer Overflow**
  - Vulnerability
  - Protection

34