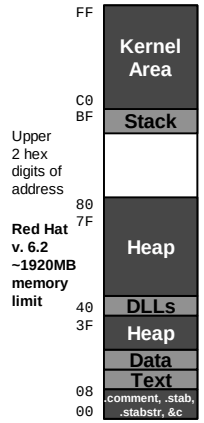
15-213

Machine Language V: Miscellaneous Topics
February 12, 2002

Topics

- Linux Memory Layout
- Understanding Pointers
- Buffer Overflow
- Floating Point Code
- Reading: 3.12 – 3.16
- Problems: 3.24 and 3.39

Linux Memory Layout



Stack
- Runtime stack (8MB limit)

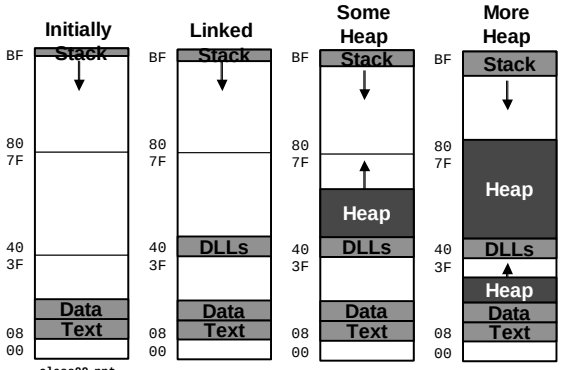
Heap
- Dynamically allocated storage
- When call `malloc`, `calloc`, `new`

DLLs
- Dynamically Linked Libraries
- Library routines (e.g., `printf`, `malloc`)
- Linked into object code when first executed

Data
- Statically allocated data
- E.g., arrays & strings declared in code

Text
- Executable machine instructions
- Read-only

Linux Memory Allocation



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 ... */
}
```

Finding the Stack and Heap

```
(gdb) break main
(gdb) run
  Breakpoint 1, 0x80483fc in main ()
(gdb) print $esp
  $3 = (void *) 0xbffffb00
```

Main

- Address `0x80483fc` should be read `0x080483fc`

Stack

- Address `0xbffff00`

Loading of A Dynamic Library

```
(gdb) disassemble 0x400734e0
  No function contains specified address.
(gdb) print * (char *) 0x400734e0
  Cannot access memory at address 0x400734e0
(gdb) break malloc
  Breakpoint 1, main () at q.c:11
  11       p1 = malloc(1 <<28);  /* 256 MB */
  Dump of assembler code for function __libc_malloc:
  0x400734e0 <__libc_malloc>:     push   %ebp
  0x400734e1 <__libc_malloc+1>:   mov    %esp,%ebp
  0x400734e3 <__libc_malloc+3>:   sub    $0x8,%esp
  …
```

Main

- Address `0x80483fc` should be read `0x080483fc`

Stack

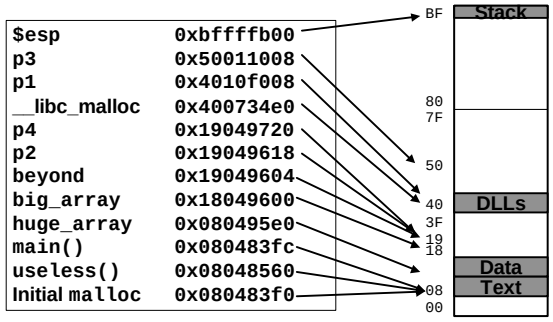
- Address `0xbffffb00`

malloc()

- `Implemented by __libc_malloc`
- `Linked into memory at runtime – not present before program starts`

Example Addresses

`$esp`	`0xbffffb00`
`p3`	`0x50011008`
`p1`	`0x4010f008`
`__libc_malloc`	`0x400734e0`
`p4`	`0x19049720`
`p2`	`0x19049618`
`beyond`	`0x19049604`
`big_array`	`0x18049600`
`huge_array`	`0x080495e0`
`main()`	`0x080483fc`
`useless()`	`0x08048560`
Initial `malloc`	`0x080483f0`



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

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 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

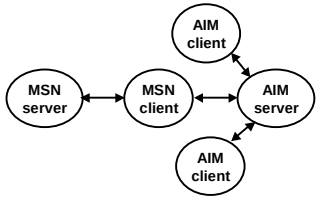
Internet Worm and IM War

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



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.
- How did it happen?

The Internet Worm and AOL/Microsoft War were both based on *stack buffer overflow* exploits!

- many Unix functions, such as `gets()` and `strcpy()`, do not check argument sizes.
- allows target buffers to overflow.

Vulnerable Buffer Code

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

```
int main()
{
  printf("Type a string:");
  echo();
  return 0;
}
```

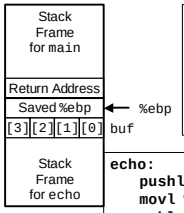
Buffer Overflow Executions

```
unix>./bufdemo
Type a string:123
123
```

```
unix>./bufdemo
Type a string:12345
Segmentation Fault
```

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


Buffer Overflow Stack



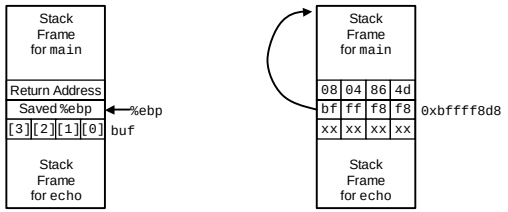

```
/* Echo Line */
void echo()
{
    char buf[4];  /* Way too small! */
    gets(buf);
    puts(buf);
}
```

```
echo:
   pushl %ebp          # Save %ebp on stack
   movl %esp,%ebp
   subl $20,%esp       # Allocate space on stack
   pushl %ebx          # Save %ebx
   addl $-12,%esp      # Allocate space on stack
   leal -4(%ebp),%ebx  # Compute buf as %ebp-4
   pushl %ebx          # Push buf on stack
   call gets           # Call gets
   . . .
```


Buffer Overflow Stack Example

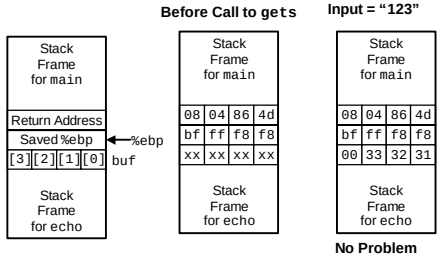
```
unix> gdb bufdemo
(gdb) break echo
Breakpoint 1 at 0x8048583
(gdb) run
Breakpoint 1, 0x8048583 in echo ()
(gdb) print /x *(unsigned *)$ebp
$1 = 0xbffff8f8
(gdb) print /x *((unsigned *)$ebp + 1)
$3 = 0x804864d
```

Before Call to gets

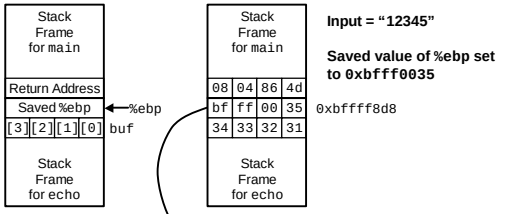



```
8048648: call 804857c <echo>
804864d: mov  0xffffffe8(%ebp),%ebx # Return Point
```


Buffer Overflow Stack Example #1



Buffer Overflow Stack Example #2



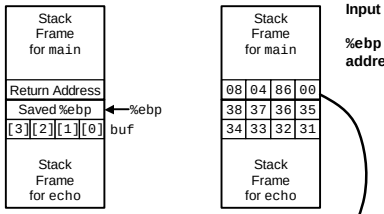
Input = "12345"

Saved value of %ebp set to 0xbfff0035

echo code:

```
8048592: push   %ebx
8048593: call   80483e4 <_init+0x50>
8048598: mov    0xffffffe8(%ebp),%ebx
804859b: mov    %ebp,%esp
804859d: pop    %ebp    # %ebp gets set to invalid value
804859e: ret
```


Buffer Overflow Stack Example



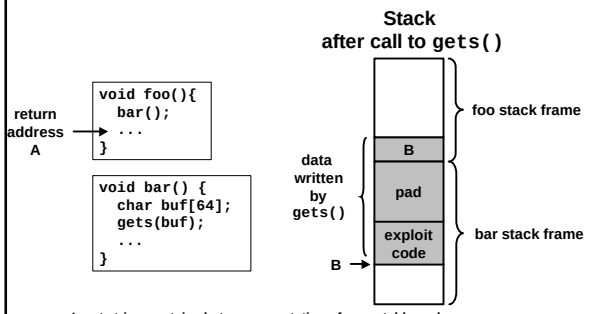
Input = "12345678"

%ebp and return address corrupted

Function good_echo

```
80485fc: 80 3c 33 0a    cmpb   $0xa,(%ebx,%esi,1)
8048600: 75 ae          jne    80485b0 <good_echo+0x10>
8048602: a1 b8 97 04 08 mov    0x80497b8,%eax
```


Malicious Use of Buffer Overflow



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

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.

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.

```
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 itsshare.
But AOL is now *exploiting their own buffer overrun bug* to helpin
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!

IA32 Floating Point

History

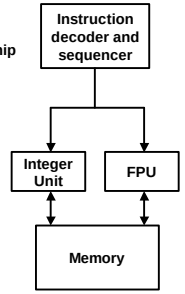
- **8086: first computer to implement IEEE FP**
  - separate 8087 FPU (floating point unit)
- **486: merged FPU and Integer Unit onto one chip**

Summary

- **Hardware to add, multiply, and divide**
- **Floating point data registers**
- **Various control & status registers**

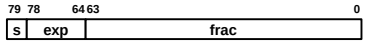
Floating Point Formats

- **single precision (C `float`): 32 bits**
- **double precision (C `double`): 64 bits**
- **extended precision (C `long double`): 80 bits**



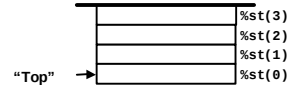
FPU Data Register Stack

FPU register format (extended precision)



FPU registers

- **8 registers**
- **Logically forms shallow stack**
- **Top called `%st(0)`**
- **When push too many, bottom values disappear**



stack grows down

FPU instructions

Large number of floating point instructions and formats

- **~50 basic instruction types**
- **load, store, add, multiply**
- **sin, cos, tan, arctan, and log!**

Sample instructions:

Instruction	Effect	Description
`fldz`	push 0.0	Load zero
`flds Addr`	push M[Addr]	Load single precision real
`fmuls Addr`	`%st(0) <- %st(0)*M[Addr]`	Multiply
`faddp`	`%st(1) <- %st(0)+%st(1); pop`	Add and pop

Floating Point Code Example

Compute Inner Product of Two Vectors

- Single precision arithmetic
- Scientific computing and signal processing workhorse

```
float ipf (float x[],
           float y[],
           int n)
{
  int i;
  float result = 0.0;

  for (i = 0; i < n; i++) {
    result += x[i] * y[i];
  }
  return result;
}
```

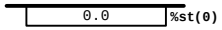
```
   pushl %ebp              # setup
   movl %esp,%ebp
   pushl %ebx

   movl 8(%ebp),%ebx       # %ebx=&x
   movl 12(%ebp),%ecx      # %ecx=&y
   movl 16(%ebp),%edx      # %edx=n
   fldz                    # push +0.0
   xorl %eax,%eax          # i=0
   cmpl %edx,%eax          # if i>=n done
   jge .L3
.L5:
   flds (%ebx,%eax,4)      # push x[i]
   fmuls (%ecx,%eax,4)     # st(0)*=y[i]
   faddp                   # st(1)+=st(0); pop
   incl %eax               # i++
   cmpl %edx,%eax          # if i<n repeat
   jl .L5
.L3:
   movl -4(%ebp),%ebx      # finish
   movl %ebp, %esp
   popl %ebp
   ret                     # st(0) = result
```


Inner Product Stack Trace

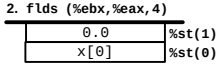
Initialization

1. `fldz`

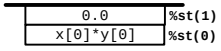


Iteration 0

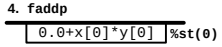
2. `flds (%ebx,%eax,4)`



3. `fmuls (%ecx,%eax,4)`

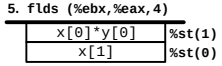


4. `faddp`

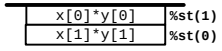


Iteration 1

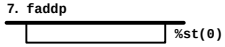
5. `flds (%ebx,%eax,4)`



6. `fmuls (%ecx,%eax,4)`



7. `faddp`



x[0]*y[0]+x[1]*y[1]

Final Observations

Memory Layout

- OS/machine dependent (including kernel version)
- Basic partitioning: stack/data/text/heap/DLL found in most machines

Type Declarations in C

- Notation obscure, but very systematic

Working with Strange Code

- Important to analyze nonstandard cases
  - E.g., what happens when stack corrupted due to buffer overflow
- Helps to step through with GDB

IA32 Floating Point

- Strange "shallow stack" architecture