Labs

- **Lab 1**
  - Grading snags hopefully resolved?
- **Lab 2**
  - Due Thurs 11:59 PM
  - Questions?
- **Lab 3**
  - Assigned on Thurs

Last Time

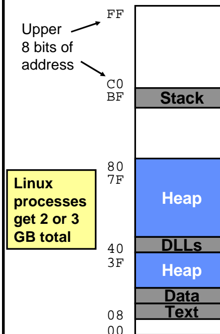
- **Arrays in C**
  - Contiguously allocated
  - No bounds checking
  - Closely resemble pointers
  - Indexing always reduces to pointer arithmetic
  - 2-D arrays are row-major
- **Structs in C**
  - Store heterogeneous data
  - Struct members (and structs) have alignment requirements
  - Alignment can have a strong effect on memory consumption
- **Unions in C**
  - Save memory by storing different types at different times
  - No language support for finding out which type is currently there
  - Can be used as a way to look at underlying data representations

Machine-Level Programming V: Miscellaneous Topics

- **Topics**
  - Linux memory layout
  - Type declarations in C
  - Buffer overflows

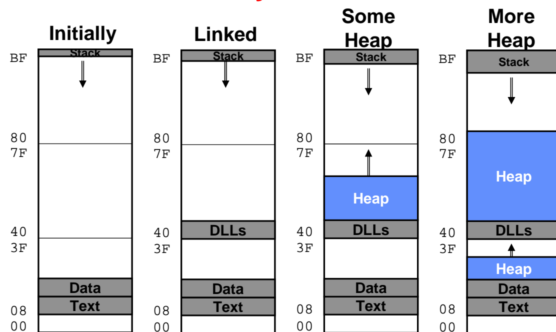
Linux / x86 Memory Layout

Upper 8 bits of address



- **Stack**
  - Runtime stack (8MB limit)
- **Heap**
  - Dynamically allocated storage
  - Use `malloc`, `calloc`, `new`
- **Shared libraries**
  - Linked in dynamically
  - 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

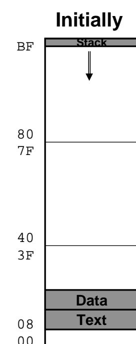
Linux Memory Allocation



Text and Stack Example

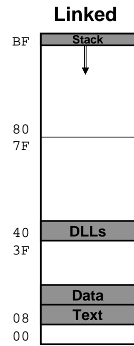
```
(gdb) break main
(gdb) run
  Breakpoint 1, 0x804856f in main ()
(gdb) print $esp
  $3 = (void *) 0xbffffc78
```

- **Main**
  - Address `0x0804856f`
- **Stack**
  - Address `0xbffffc78`



Dynamic Linking Example

```
(gdb) print malloc
  $1 = {<text variable, no debug info>}
    0x8048454 <malloc>
(gdb) run
  Program exited normally.
(gdb) print malloc
  $2 = {void *(unsigned int)}
    0x40006240 <malloc>
```



- **Initially**
  - Code in text segment that invokes dynamic linker
  - Address `0x8048454` should be read `0x08048454`
- **Final**
  - Code in DLL region

Memory Allocation Example

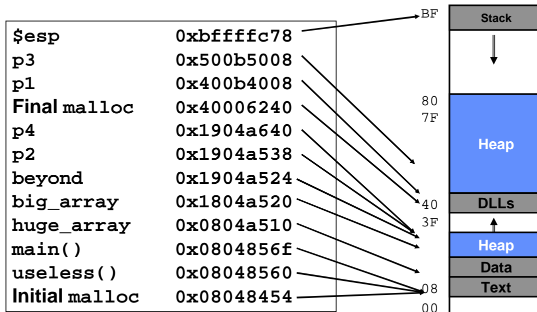
```
char big_array[1<<24];  /*  16 MB */
char huge_array[1<<28]; /* 256 MB */

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 ... */
}
```


Example Addresses



`$esp`	`0xbffffc78`
`p3`	`0x500b5008`
`p1`	`0x400b4008`
Final `malloc`	`0x40006240`
`p4`	`0x1904a640`
`p2`	`0x1904a538`
`beyond`	`0x1904a524`
`big_array`	`0x1804a520`
`huge_array`	`0x0804a510`
`main()`	`0x0804856f`
`useless()`	`0x08048560`
Initial `malloc`	`0x08048454`

C operators

- Operators appear within expressions
  - `+, -, *, /, ->,` etc.
- Operators have precedence ordering
  - This determines order of evaluation in an expression
  - For example, `->` binds tighter than `*`
  - `A + B->X * C`
- Operators have associativity
  - This determines order of evaluation in an expression for a group of operators that have the same precedence
  - `A + B + C`
  - `**~C`
- The point: The meaning of a complex expression must be well-defined

C operator precedence

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

Note on ++ and --

- They are side effecting
  - What happens when a single expression contains multiple ++ and/or -- of the same variable?
  - Anybody know what a “sequence point” is?
  - Don’t get burned by this: avoid ++ and – in complex expressions
- Why do they exist?
  - Borrowed from C’s indirect predecessor, B
  - Probably originally inspired by auto-increment and auto-decrement addressing modes

Reading C Type Declarations

```
int (*(*x)(char *,double))[9][20];
```

- Easy types:
  - `float i;`
  - `float *j;`
  - `int ***k;`
- Basic types:
  - `char, signed char, unsigned char, short, signed short, unsigned short, int, signed int, …`
- Derived types are pronounced:
  - `*` as "pointer to"
  - `[]` as "array of"
  - `()` as "function returning"

13

The "Right-Left" Rule

1. Find the identifier
2. Look on the right side of the identifier
   - Pronounce each symbol that you run into
   - Stop when you run out of symbols OR hit a right-paren ")"
3. Look at the symbols to the left of the identifier
   - If it's something like "int" say it
   - It's a symbol use the translation
   - Stop when you run out of symbols OR hit a left-paren "("
4. Repeat steps 2 and 3 until finished

`*` is "pointer to"
`[]` is "array of"
`()` is "function returning"

14

More Type Decl

- Parentheses are either
  1. "grouping" – they surround the identifier
  2. "function returning" – they are to the right of the identifier
- The point: The meaning of a complex type declaration must be well-defined

15

C pointer declarations

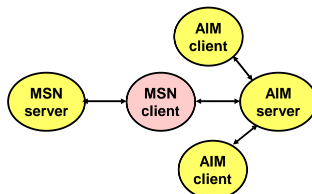
```
int *p;
int *p[13];
int *(p[13]);
int **p;
int (*p)[13];
int *f();
int *(*f())();
int (*f)();
int (*(*f())[13])();
int (*(*x[3])())[5];
int (*(*x)(char *,double))[9][20];
char *(*(**foo[8][])())[];
```

`*` is "pointer to"
`[]` is "array of"
`()` is "function returning"

16

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



17

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 part of the AOL/Microsoft War were both based on *stack buffer overflow* exploits!
    - many Unix functions do not check argument sizes
    - allows target buffers to overflow

18

String Library Code

- Implementation of Unix function `gets`
  - » No way to specify limit on number of characters to read

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

- Similar problems with other Unix functions
  - » `strcpy`: Copies string of arbitrary length
  - » `scanf`, `fscanf`, `sscanf`, when given `%s` conversion specification

19

Vulnerable Buffer Code

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

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

20

Buffer Overflow Executions

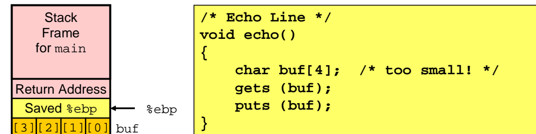
```
unix>./bufdemo
Type a string:123
123
```

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

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

21

Stack Buffer Overflow



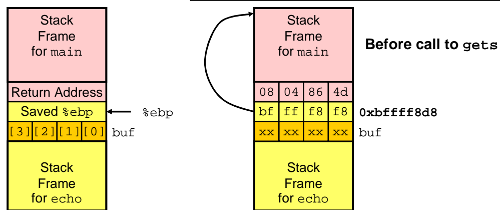
```
/* Echo Line */
void echo()
{
    char buf[4];  /* too small! */
    gets (buf);
    puts (buf);
}
```

```
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
   . . .
```

22

Stack Buffer Overflow Example

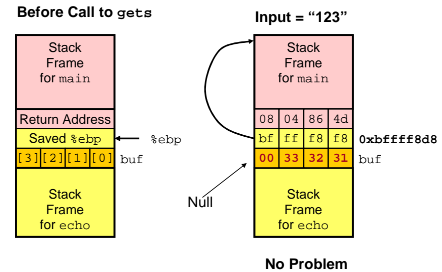
```
unix> gdb bufdemo
(gdb) break echo
Breakpoint 1 at 0x8048583
(gdb) run
Breakpoint 1, 0x8048583 in echo ()
(gdb) print /x *(unsigned *)$ebp
$1 = 0xbffff8f8
(gdb) print /x *((unsigned *)$ebp + 1)
$2 = 0x804864d
```




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

23

Buffer Overflow Example #1



No Problem

24

Buffer Overflow Stack Example #2

Input = “12345”

Stack Frame for `main`
Return Address
Saved `%ebp` ← `%ebp`
[3][2][1][0] `buf`
Stack Frame for `echo`

Stack Frame for `main`
08 04 86 4d
bf ff 00 35 — `0xbffff8d8`
34 33 32 31 — `buf`
Stack Frame for `echo`

Saved value of `%ebp` set to `0xbfff0035`

Bad news when later attempt to restore `%ebp`

`echo` code:

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

25

Buffer Overflow Stack Example #3

Input = “12345678”

Stack Frame for `main`
Return Address
Saved `%ebp` ← `%ebp`
[3][2][1][0] `buf`
Stack Frame for `echo`

Stack Frame for `main`
08 04 86 00
38 37 36 35 — `0xbffff8d8`
34 33 32 31 — `buf`
Stack Frame for `echo`

Invalid address

No longer pointing to desired return point

`%ebp` and return address corrupted

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

26

Malicious Use of Buffer Overflow

Stack after call to `gets()`

```
void foo(){
  bar();
  ...
}
```

return address A

```
void bar() {
  char buf[64];
  gets(buf);
  ...
}
```

foo stack frame
A
pad
exploit code
bar stack frame

data written by `gets()`

buf starts here

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

27

Exploits Based on Buffer Overflows

- **What if string arrives over the network instead of being typed on the command line?**
  - Buffer overflow bugs allow remote machines to execute arbitrary code on victim machines.
- **Morris Internet worm**
  - Early versions of the finger server daemon (fingerd) used `gets()` to read the argument sent by the client:
    - » *finger regehr@cs.utah.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

28

```
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!

29

Code Red Worm

- **History**
  - June 18, 2001. Microsoft announces buffer overflow vulnerability in IIS Internet server
  - July 19, 2001. over 250,000 machines infected by new virus in 9 hours
  - White house must change its IP address. Pentagon shut down public WWW servers for day
- **Web servers received strings of form**

```
GET
  /default.ida?NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN....NNNNNNNNNNNNNN
  NNNNNNNNNNNNNNNNNNNNNNNNNNNN%u9090%u6858%ucbd3%u7801%u9090%u6858%ucbd3%
  u7801%u9090%u6858%ucbd3%u7801%u9090%u9090%u8190%u00c3%u0003%u8b00%u531
  b%u53ff%u0078%u0000%u00=a
HTTP/1.0" 400 325 "-" "-"
```

30

Code Red Exploit Code

- Starts 100 threads running
- Spread self
  - » Generate random IP addresses & send attack string
  - » Between 1st & 19th of month
- Attack www.whitehouse.gov
  - » Send 98,304 packets; sleep for 4-1/2 hours; repeat
    - Denial of service attack
  - » Between 21st & 27th of month
- Deface server's home page
  - » After waiting 2 hours

gets() vs. fgets()

- From the man pages

char *gets (char *s);

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

DANGEROUS – no size limit

char *fgets (char *s, int size, FILE *stream);

fgets() reads in at most one less than size characters from stream and stores them into the buffer pointed to by s. Reading stops after an EOF or a newline. If a newline is read, it is stored into the buffer. A '\0' is stored after the last character in the buffer.

SAFER – but you still have to supply the right argument

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 `%s` conversion specification
    - » Use `fgets` to read the string
- Use a language that helps you out
  - Java, for example
  - However, JVMs often implemented in C and can still be vulnerable!

Summary

- Memory Layout
  - OS / machine / compiler dependent (including kernel and compiler version)
  - Basic partitioning: stack/data/text/heap/DLL found in most machines
- Operator precedence and type declarations in C
  - Obscure but systematic
  - All legal expressions and type decls should have well-defined interpretations
- Buffer overflow
  - Probably the dominant non-human security problem these days
  - Programming language can help
    - » Overflows (theoretically) impossible in Java
    - » C gives you no help at all