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

Memory Allocation Example

```c
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 ... */
}
```
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
  • Address 0x8048454 should be read 0x08048454

Breakpointing Example

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

Main
• Address 0x804856f should be read 0x0804856f

Stack
• Address 0xbffffc78

Example Addresses

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$esp</td>
<td>0xbffffc78</td>
</tr>
<tr>
<td>p3</td>
<td>0x500b5008</td>
</tr>
<tr>
<td>p1</td>
<td>0x400b4008</td>
</tr>
<tr>
<td>Final malloc</td>
<td>0x40006240</td>
</tr>
<tr>
<td>p4</td>
<td>0x190a640</td>
</tr>
<tr>
<td>p2</td>
<td>0x190a538</td>
</tr>
<tr>
<td>beyond</td>
<td>0x190a524</td>
</tr>
<tr>
<td>big_array</td>
<td>0x180a520</td>
</tr>
<tr>
<td>huge_array</td>
<td>0x080a510</td>
</tr>
<tr>
<td>main()</td>
<td>0x0804856f</td>
</tr>
<tr>
<td>useless()</td>
<td>0x08048560</td>
</tr>
<tr>
<td>Initial malloc</td>
<td>0x08048454</td>
</tr>
</tbody>
</table>

C operators

<table>
<thead>
<tr>
<th>Operators</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>() [] -&gt; .</td>
<td>left to right</td>
</tr>
<tr>
<td>! ~ ++ -- + * &amp; (type) sizeof</td>
<td>right to left</td>
</tr>
<tr>
<td>* / %</td>
<td>left to right</td>
</tr>
<tr>
<td>+ -</td>
<td>left to right</td>
</tr>
<tr>
<td>&lt;&lt; &gt;&gt;</td>
<td>left to right</td>
</tr>
<tr>
<td>&lt; &lt;= &gt; &gt;=</td>
<td>left to right</td>
</tr>
<tr>
<td>== !=</td>
<td>left to right</td>
</tr>
<tr>
<td>&amp;</td>
<td>left to right</td>
</tr>
<tr>
<td>^</td>
<td>left to right</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>left to right</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>?: = + = *= /= %= &amp;= ^= !=</td>
<td>right to left</td>
</tr>
<tr>
<td>&lt;= &gt;=</td>
<td>right to left</td>
</tr>
</tbody>
</table>

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 an array[13] of pointer to 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  

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

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

```c
#include <stdio.h>

int main() {
    printf("Type a string:");
    char buf[4];  /* Way too small! */
    gets(buf);
    puts(buf);
    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

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

804864d: mov 0xffffffff ($ebp),%ebx # Return Point

Buffer Overflow Stack Example #1

Before Call to gets

Input = "123"

No Problem

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
...
Malicious Use of Buffer Overflow

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.
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!
Floating Point Code Example

Compute Inner Product of Two Vectors

- Single precision arithmetic
- Scientific computing and signal processing workhorse

```c
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;
}
```

Inner Product Stack Trace

Initialization

1. fldz

![Stack Trace Image]

Iteration 0

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

![Stack Trace Image]

Iteration 1

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

![Stack Trace Image]

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