Today

- Happy Monday!
- HW2 due, how is Lab 3 going?
- Today we’ll go over:
  - Address space layout
  - Input buffers on the stack
  - Overflowing buffers and injecting code
  - Defenses against buffer overflows
IA32 Linux Memory Layout

- **Stack**
  - Runtime stack (8MB limit)

- **Heap**
  - Dynamically allocated storage
  - Allocated by `malloc()`, `calloc()`, `new()`

- **Data**
  - Statically allocated data
    - Read-only: string literals
    - Read/write: global arrays and variables

- **Text**
  - Executable machine instructions
  - Read-only

Upper 2 hex digits = 8 bits of address
IA32/Linux Stack Frame

- **Current Stack Frame** ("Top" to Bottom)
  - "Argument build" area (parameters for function about to be called)
  - Local variables (if can’t be kept in registers)
  - Saved register context (when reusing registers)
  - Old frame pointer (for caller)

- **Caller’s Stack Frame**
  - Return address
    - How does call/ret change the stack?
  - Arguments for this call

![Diagram of stack frame]

- Frame pointer %ebp
- Stack pointer %esp
- Caller Frame
  - Arguments
  - Return Addr
  - Old %ebp
  - Saved Registers + Local Variables
  - Argument Build
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?
IA32 Example Addresses

address range $\sim 2^{32}$

$\texttt{esp}$ 0xffffffbcd0
$\texttt{p3}$ 0x65586008
$\texttt{p1}$ 0x55585008
$\texttt{p4}$ 0x1904a110
$\texttt{p2}$ 0x1904a008
$\texttt{&p2}$ 0x18049760
$\texttt{beyond}$ 0x08049744
$\texttt{big\_array}$ 0x18049780
$\texttt{huge\_array}$ 0x08049760
$\texttt{main()}$ 0x080483c6
$\texttt{useless()}$ 0x08049744
$\texttt{final\_malloc()}$ 0x006be166

$\texttt{malloc()}$ is dynamically linked; its address is determined at runtime.
Internet Worm

- These characteristics of the traditional IA32 Linux memory layout provide opportunities for malicious programs
  - Stack grows “backwards” in memory
  - Data and instructions both stored in the same memory

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

- These characteristics of the traditional IA32 Linux memory layout provide opportunities for malicious programs
  - Stack grows “backwards” in memory
  - Data and instructions both stored in the same memory

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

- Stack buffer overflow exploits!
Buffer Overflow in a nutshell

- Many classic Unix/Linux/C functions do not check argument sizes.
- C does not check array bounds.
- Allows overflowing (writing past the end of) buffers (arrays)
- Overflows of buffers on the stack overwrite interesting data.
- Attackers just choose the right inputs.
- Probably the most common type of security vulnerability
String Library Code

- Implementation of Unix function `gets()`

```c
/* 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;
}
```

- What could go wrong in this code?

- pointer to start of an array

- same as: `*p = c; p++;`
String Library Code

- Implementation of Unix function `gets()`

```c
/* 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 Unix functions
  - `strcpy`: Copies string of arbitrary length
  - `scanf`, `fscanf`, `sscanf`, when given `%s` conversion specification
Vulnerable Buffer Code

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

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

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

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

Unix>
```
./bufdemo
Type a string:123456789ABC
Segmentation Fault
```
Buffer Overflow Disassembly

080484f0 <echo>:

- 080484f0:  55  push %ebp
- 080484f1:  89 e5  mov %esp,%ebp
- 080484f3:  53  push %ebx
- 080484f4:  8d 5d f8  lea 0xffffffff8(%ebp),%ebx
- 080484f7:  83 ec 14  sub $0x14,%esp
- 080484fa:  89 1c 24  mov %ebx,(%esp)
- 080484fd:  e8 ae ff ff ff  call 80484b0 <gets>
- 08048502:  89 1c 24  mov %ebx,(%esp)
- 08048505:  e8 8a fe ff ff  call 8048394 <puts@plt>
- 0804850a:  83 c4 14  add $0x14,%esp
- 0804850d:  5b  pop %ebx
- 0804850e:  c9  leave
- 0804850f:  c3  ret

- 080485f2:  e8 f9 fe ff ff  call 80484f0 <echo>
- 080485f7:  8b 5d fc  mov 0xfffffffffc(%ebp),%ebx
- 080485fa:  c9  leave
- 080485fb:  31 c0  xor %eax,%eax
- 080485fd:  c3  ret
Buffer Overflow Stack

Before call to gets

Stack Frame for main

Return Address

Saved %ebp

Saved %ebx

buf

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

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

Before call to gets

Stack Frame for main
Return Address
Saved %ebp
Saved %ebx
buf

buf

80485f2: call 80484f0 <echo>
80485f7: mov 0xffffffffc(%ebp),%ebx # Return Point
Buffer Overflow Example #1

Overflow buf, and corrupt saved %ebx, but no problem, why? What happens if input has one more byte?
Buffer Overflow Example #2

Before call to `gets`

Stack Frame for `main`

Before call to `gets`

Stack Frame for `main`

Input “12345678”

Frame pointer corrupted

```
804850a: 83 c4 14 add $0x14, %esp  # deallocate space
804850d: 5b pop %ebx  # restore %ebx
804850e: c9 leave  # movl %ebp, %esp; popl %ebp
804850f: c3 ret  # Return
```
Buffer Overflow Example #3

Before call to gets:

Stack Frame for main

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xfffffc638</td>
<td>58 c6 ff ff</td>
</tr>
<tr>
<td>Saved %ebx</td>
<td>xx xx xx xx</td>
</tr>
<tr>
<td>buf</td>
<td>0xfffffc630</td>
</tr>
</tbody>
</table>

Input “123456789ABC”:

Stack Frame for main

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xfffffc638</td>
<td>43 42 41 39</td>
</tr>
<tr>
<td>Saved %ebx</td>
<td>38 37 36 35</td>
</tr>
<tr>
<td>buf</td>
<td>0xfffffc630</td>
</tr>
</tbody>
</table>

Return address corrupted

080485f2: call 80484f0 <echo>
080485f7: mov 0xfffffffff(%ebp),%ebx # Return Point

Hmmm, what can you do with it?
Malicious Use of Buffer Overflow

- Input string contains byte representation of executable code
- Overwrite return address A with address of buffer (need to know B)
- When \texttt{bar()} executes \texttt{ret}, will jump to exploit code (instead of A)
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

commandline facebook of the 80s!
Avoiding Overflow Vulnerability

- Use library routines that limit string lengths
  - `fgets` instead of `gets` (second argument to `fgets` sets limit)
  - `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

- Other ideas?
System-Level Protections

- Randomized stack offsets
  - At start of program, allocate random amount of space on stack
  - Makes it difficult for exploit to predict beginning of inserted code

- Use techniques to detect stack corruption

- Nonexecutable code segments
  - Only allow code to execute from “text” sections of memory
  - Do NOT execute code in stack, data, or heap regions
  - Hardware support needed

Diagram:
- Stack
- Heap
- Data
- Text

(not drawn to scale)