Buffer overflows

- Buffer overflows are possible because C does not check array boundaries
- Buffer overflows are dangerous because buffers for user input are often stored on the stack

Topics:
- 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

Buffer Overflow
IA32/Linux Stack Frame

- **Caller’s Stack Frame**
  - Arguments for this call
  - Return address
    - Pushed by `call` instruction

- **Current /Callee Stack Frame**
  - Old frame pointer (for caller)
  - Saved register context (when reusing registers)
  - Local variables (if can’t be kept in registers)
  - “Argument build” area (if callee needs to call another function - parameters for function about to be called)
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 ... */
}
```

Where does everything go?
IA32 Example Addresses

address range $\sim 2^{32}$

$\text{esp}$: 0xffffbcd0
$p3$: 0x65586008
$p1$: 0x55585008
$p4$: 0x1904a110
$p2$: 0x1904a008
\&$p2$: 0x18049760
\text{beyond}: 0x08049744
\text{big_array}: 0x18049780
\text{huge_array}: 0x08049760
\text{main}(): 0x080483c6
\text{useless}(): 0x08049744
\text{final malloc}(): 0x006be166

\text{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?

- 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?
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 to a dest
  - `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

```assembly
080484f0  <echo>:
  80484f0:  55               push   %ebp
  80484f1:  89 e5             mov    %esp,%ebp
  80484f2:  53               push   %ebx
  80484f3:  8d 5d f8         lea    0xffffffff8(%ebp),%ebx
  80484f6:  83 ec 14         sub    $0x14,%esp
  80484fa:  89 1c 24         mov    %ebx,(%esp)
  80484fd:  e8 ae ff ff ff   call   80484b0 <gets>
  8048502:  89 1c 24         mov    %ebx,(%esp)
  8048505:  e8 8a fe ff ff   call   8048394 <puts@plt>
  804850a:  83 c4 14         add    $0x14,%esp
  804850d:  5b               pop    %ebx
  804850e:  c9               leave
  804850f:  c3               ret

  80485f2:  e8 f9 fe ff ff   call   80484f0 <echo>
  80485f7:  8b 5d fc         mov    0xfffffffffffc(%ebp),%ebx
  80485fa:  c9               leave
  80485fb:  31 c0            xor    %eax,%eax
  80485fd:  c3               ret
```
Buffer Overflow Stack

Before call to gets

Stack Frame for `main`

<table>
<thead>
<tr>
<th>Return Address</th>
<th>Old <code>%ebp</code></th>
<th>Saved <code>%ebx</code></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>[3] [2] [1] [0]</td>
<td>buf</td>
</tr>
</tbody>
</table>

/* 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
- Old `%ebp`
- Saved `%ebx`
- `[3][2][1][0]`

buf

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

**Before call to gets**

Stack Frame for `main`

- 0xfffff658
- 0xfffff638
- Saved `%ebx`
- `xx xx xx xx`

buf

0xfffff630
Buffer Overflow Example #1: “1234567”

- Overflow buf, and corrupt saved %ebx, but seems like it is not problem, why not?
- What happens if input has one more byte?
Buffer Overflow Example #2: “12345678”

Before call to gets

Stack Frame for main

0xfffffc658

f7 85 04 08
58 c6 ff ff

Saved %ebx

xx xx xx xx

buf

0xfffffc638

Input “12345678”

Stack Frame for main

0xfffffc658

f7 85 04 08
58 c6 ff 00

buf

0xfffffc638

38 37 36 35

34 33 32 31

Old %ebp corrupted!

. . .

804850a: 83 c4 14 add $0x14,%esp # deallocate space
804850d: 5b pop %ebx # restore %ebx
804850e: c9 leave # movl %ebp, %esp; popl %ebp # Return
804850f: c3 ret
Buffer Overflow Example #3: “123456789ABC”

Before call to gets

Stack Frame for main

Saved %ebx

buf

0xfffffc630

0xfffffc638

0xfffffc658

Input “123456789ABC”

Stack Frame for main

0xfffffc630

0xfffffc638

0xfffffc658

We can change the return address? Hmmm...

080485f2: call 80484f0 <echo>

080485f7: mov 0xffffffffc(%ebp),%ebx # Return Point
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

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