Buffer overflows

- 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

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

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 ... */
}
```
### IA32 Example Addresses

**address range ~2^{32}**

- $esp 0xffffbcd0
- p3 0x65586008
- p1 0x55585008
- p4 0x1904a110
- p2 0x1904a008
- 6p2 0x18049760
- beyond 0x08049744
- big_array 0x18049780
- huge_array 0x08049760
- main() 0x080483c6
- useless() 0x08049744
- malloc() 0x006be166

*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?

### Buffer Overflow in a nutshell

- Many classic Unix/Linux/C functions do not check argument sizes.
- C does not check array bounds.
- Allows overwriting (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

---

**Stack buffer overflow exploits!**
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?

- 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);
}
```

```c
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>:
  80484f0:  55  push %ebp
  80484f1:  89 e5  mov %esp,%ebp
  80484f3:  53  push %ebx
  80484f4:  8d 5d f8  lea 0xfffffff8(%ebp),%ebx
  80484f7:  83 ec 14  sub $0x14,%esp
  80484fa:  89 1c 24  mov %ebx,(%esp)
  8048502:  e8 8a ff ff ff  call 80484b0 <gets>
  8048505:  89 1c 24  mov %ebx,(%esp)
  8048508:  e8 ae ff ff ff  call 8048394 <puts@plt>
  804850b:  83 c4 14  add $0x14,%esp
  804850e:  5b  pop %ebx
  8048510:  c9  leave
  8048512:  c3  ret

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

```assembly
echo:
pushl %ebp  # Save %ebp on stack
movl %esp, %ebp  # Save %ebp
leal -8(%ebp),%ebx  # Compute buf as %ebp-8
subl $20, %esp  # Allocate stack space
movl %ebx, (%esp)  # Push buf addr on stack
leal -4(%ebp),%esp  # Call gets
```

**Buffer Overflow Stack Example**

```assembly
call 80484f0 <echo>
mov 0xfffffffc(%ebp),%ebx  # Return Point
```

**Buffer Overflow Example #1**

Before call to gets

Before call to gets

Before call to gets

Before call to gets

Input “1234567”

Before call to gets

Before call to gets

Before call to gets

Before call to gets

Before call to gets

Before call to gets

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

Before call to gets

Before call to gets

Before call to gets

Input “12345678”

Before call to gets

Before call to gets

Before call to gets

Before call to gets

Frame pointer corrupted

---
Buffer Overflow Example #3

Before call to gets

Stack Frame for main

Input “123456789ABC”

Stack Frame for main

Return address corrupted

Input $\text{080485f2: call 80484f0 <echo>}$

$\text{080485f7: mov 0xffffffff(%ebp),%ebx # Return Point}$

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 bar() executes 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:
    - $\text{finger droh@cs.cmu.edu}$
  - Worm attacked fingerd server by sending phony argument:
    - $\text{finger "exploit-code padding new-return-address"}$
    - exploit code: executed a root shell on the victim machine with a direct TCP connection to the attacker

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 (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
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

---

![Memory Segmentation Diagram](image-url)