Buffer Overflow
Buffer Overflow

Slides credit to Breno de Medeiros
Buffer Overflow Attacks

• What is a buffer?
  • A memory space in which data/code can be held
  • Buffer has finite capacity, often predefined size

• Buffer Overflows
  • User input data is too long
  • The program does not check the buffer boundary
  • Data overflows the boundary, overwrite adjacent data/code

• Buffer Overflow Attack
  • By carefully replacing the data/code in the buffer, attackers can take control of a process
char sample[10];
for(i=0;i<=9;i++)
sample[i] = 'A';
sample[10] = 'B';
Buffer Overflow

- Modern attacks on heap and JavaScript
  - Morris Worm exploited fingerd on VAX
- `gets`, `sprintf`, `strcat`, `strcpy`, `vssprintf` – common unsafe C routines

Non-malicious code causes failures (see P225 in P&P)
- in '69 Arpanet had a hardcoded limit of 347 nodes
- in '89 a 348-th node was added and overflow the table
Need to Refresh your “Memory”...
CPU
CPU state held in registers

- General Purpose
  - EAX, EBX, ECX, EDX, EDI, ESI
- Special Purpose:
  - EIP: Instruction Pointer
  - ESP: Stack Pointer
  - EBP: Frame/Base Pointer
CPU modifies state using instructions

- Executes assembly instructions
  - ADD, SUB, MULT, XOR, CMP, JMP, ...

- Syntax
  - tool dependent
  - AT&T
    - OP src dest
    - This lecture
  - Intel
    - OP dest src
    - What most people/tools use
Software state is registers + memory
ReCap: Process’s Memory Region

For static variables
- uninitialized variables
- initialized variables

Text region: code, read-only data

Direction of stack growth
- Stack: Last in, first out (LIFO)
- PUSH: add an element at top
- POP: remove element at top

for procedure call – jump and return

Store “control data”
- local variables
- parameters to functions
- Return values/addresses

Direction of heap growth
- Heap: dynamically allocated memory,
  e.g., using malloc.

High address: 0xFFFFFFFF

Low address: 0x00000000
//example.c:

void foo(int a, int b) {
    char buf1[10];
    return;
}

void main() {
    foo(3,6);
}

Data in a Stack

Direction of stack growth

Low address

local buffer variables
local variables
previous frame pointer
return address
function arguments
<previous stack frame>

High address
Stack Frame, Stack Pointers, Stack Overflow

- Stack frame
  - Consecutive stack space for each calling function
  - Each for a function that has not yet finished execution

- Stack pointer: memory location of the top of the stack
  - Stored in a register

- Stack overflow
  - Occurs when information is written into a variable on a stack
  - But the size of this information exceeds what was allocated
C Stack Frames

Grows toward lower address
Starts at end of address space
Two related registers
  %ESP - Stack Pointer (SP)
  %EBP - Frame Pointer (FP)
Stack Operation

push 0x0a
Stack Operation

push 0x0a
push 0x6c
Stack Operation

push 0x0a
push 0x6c
push 0xff
Stack Operation

push 0x0a
push 0x6c
push 0xff
pop r1 #0xff
Stack Operation

push 0x0a
push 0x6c
push 0xff
pop r1 #0xff
pop r2 #0x6c
Stack Operation

push 0x0a
push 0x6c
push 0xff
pop r1 #0xff
pop r2 #0x6c
push 0x88
C stack frames

The frame pointer (FP) allows for frame-relative addressing of local variables.
C stack frames
C stack frames
C stack frames

[Diagram showing C stack frames with layers for caller's FP, return address, function args, and local variables.]
C stack frames

- caller's FP
- return address
- function args
- Local variables

SP ➔ callee ➔ FP ➔ caller

VT
C stack frames

SP

Local variables

caller’s FP

return address

function args

Local variables

FP

} callee

} caller
example.c

```c
void foo(int a, int b) {
    char buf1[10];
    return;
}

void main() {
    foo(3,6);
}
```
main:

```
pushl %ebp
movl %esp, %ebp
subl $8, %esp
movl $6, 4(%esp)
movl $3, (%esp)
call foo
leave
ret
```
main:
    pushl  %ebp
    movl  %esp, %ebp
    subl  $8, %esp
    movl  $6, 4(%esp)
    movl  $3, (%esp)
    call   foo
    leave
    ret
example.s (x86)

main:

```
pushl %ebp
movl %esp, %ebp
subl $8, %esp
movl $6, 4(%esp)
movl $3, (%esp)
call foo
leave
ret
```
main:
  pushl  %ebp
  movl  %esp, %ebp
  subl  $8, %esp
  movl  $6, 4(%esp)
  movl  $3, (%esp)
  call   foo
  leave
  ret
example.s (x86)

main:
  pushl %ebp
  movl %esp, %ebp
  subl $8, %esp
  movl $6, 4(%esp)
  movl $3, (%esp)
  call foo
  leave
  ret
example.s (x86)

main:
    pushl  %ebp
    movl  %esp, %ebp
    subl  $8, %esp
    movl  $6, 4(%esp)
    movl  $3, (%esp)
    call   foo
    leave
    ret

return

3
6
prev FP
example.s (x86)

foo:

pushl %ebp
movl %esp, %ebp
subl $16, %esp
leave
ret
example.s (x86)

foo:

    pushl  %ebp
    movl  %esp, %ebp
    subl  $16, %esp
    leave
    ret
example.s (x86)

foo:
pushl %ebp
movl %esp, %ebp
subl $16, %esp
leave
ret
example.s (x86)

foo:
    pushl %ebp
    movl %esp, %ebp
    subl $16, %esp
    leave
    ret

mov %ebp, %esp
pop %ebp
example.s (x86)

foo:

pushl %ebp
movl %esp, %ebp
subl $16, %esp
leave
ret

movl %ebp, %esp
popl %ebp

main FP
return
3
6
prev FP
example.s (x86)

foo:

pushl %ebp
movl %esp, %ebp
subl $16, %esp
leave
ret

mov %ebp, %esp
pop %ebp
example.s (x86)

foo:

```
pushl  %ebp
movl  %esp, %ebp
subl  $16, %esp
leave
ret
```

```
mov  %ebp, %esp
pop  %ebp
```
main:
  pushl  %ebp
  movl  %esp, %ebp
  subl  $8, %esp
  movl  $6, 4(%esp)
  movl  $3, (%esp)
  call   foo
  leave
  ret

mov  %ebp, %esp
pop  %ebp
example.s (x86)

main:

pushl %ebp
movl %esp, %ebp
subl $8, %esp
movl $6, 4(%esp)
movl $3, (%esp)
call foo
leave
ret

mov %ebp, %esp
pop %ebp

prev FP
example.s (x86)

main:
    pushl %ebp
    movl %esp, %ebp
    subl $8, %esp
    movl $6, 4(%esp)
    movl $3, (%esp)
    call foo
leave
ret
    movl %ebp, %esp
    popl %ebp
An example of stack and stack frame

```
// ex.c
int bar( int j ) {
    int jj = j + j;
    return jj;
}

int foo( int i ) {
    int ii = i + i;
    int iii = bar( ii );
    int iiii = iii;
    return iiii;
}

int main() {
    int x = foo( 10 );
    printf( "the value of x = %d\n", x );
    return 0;
}
```

What stack looks like at “int iiii = iii;” in foo()
Stack OVerFlow

How does it work?
A Stack Overflow Example

```c
void foo(char *str) {
    char buffer[16];
    strcpy(buffer, str);
}

void main() {
    char buf[256];
    memset(buf, 'A', 255);
    buf[255] = '\x00';
    foo(buf);
}
```

str* is a large string, size 256
Buffer size is only 16
Stack overwritten by ‘A’ (0x414141…)

Return address is overwritten and becomes 0x41414141

You get segmentation fault
Worse, attacker can change flow of program
void foo(char *str) {
    char buffer[16];
    strcpy(buffer, str);
}

void main() {
    char buf[256];
    memset(buf, 'A', 255);
    buf[255] = '\x00';
    foo(buf);
}
void foo(char *str) {
    char buffer[16];
    strcpy(buffer, str);
}

void main() {
    char buf[256];
    memset(buf, 'A', 255);
    buf[255] = '\x00';
    foo(buf);
}
Buffer overflow example

```c
void foo(char *str) {
    char buffer[16];
    strcpy(buffer, str);
}

void main() {
    char buf[256];
    memset(buf, 'A', 255);
    buf[255] = '\0';
    foo(buf);
}
```
Buffer overflow example

```c
void foo(char *str) {
    char buffer[16];
    strcpy(buffer, str);
}

void main() {
    char buf[256];
    memset(buf, 'A', 255);
    buf[255] = '\0'x00';
    foo(buf);
}
```
void foo(char *str) {
    char buffer[16];
    strcpy(buffer, str);
}

void main() {
    char buf[256];
    memset(buf, 'A', 255);
    buf[255] = '\x00';
    foo(buf);
}
void foo(char *str) {
    char buffer[16];
    strcpy(buffer, str);
}

void main() {
    char buf[256];
    memset(buf, 'A', 255);
    buf[255] = '\x00';
    foo(buf);
}
void foo(char *str) {
    char buffer[16];
    strcpy(buffer, str);
}

void main() {
    char buf[256];
    memset(buf, 'A', 255);
    buf[255] = '\x00';
    foo(buf);
}
void foo(char *str) {
    char buffer[16];
    strcpy(buffer, str);
}
void main() {
    char buf[256];
    memset(buf, 'A', 255);
    buf[255] = '\x00';
    foo(buf);
}
void foo(char *str) {
    char buffer[16];
    strcpy(buffer, str);
}

void main() {
    char buf[256];
    memset(buf, 'A', 255);
    buf[255] = '\0';
    foo(buf);
}
Buffer overflow example

```c
void foo(char *str) {
    char buffer[16];
    strcpy(buffer, str);
}
void main() {
    char buf[256];
    memset(buf, 'A', 255);
    buf[255] = '\x00';
    foo(buf);
}
```

```assembly
mov %ebp, %esp
pop %ebp
ret
```
void foo(char *str) {
    char buffer[16];
    strcpy(buffer, str);
}

void main() {
    char buf[256];
    memset(buf, 'A', 255);
    buf[255] = '\x00';
    foo(buf);
}

Buffer overflow example
void foo(char *str) {
    char buffer[16];
    strcpy(buffer, str);
}

void main() {
    char buf[256];
    memset(buf, 'A', 255);
    buf[255] = '\0';
    foo(buf);
}
void foo(char *str) {
    char buffer[16];
    strcpy(buffer, str);
}

void main() {
    char buf[256];
    memset(buf, 'A', 255);
    buf[255] = '\0';
    foo(buf);
}
Buffer overflow example

%eip = 0x41414141

What gets executed?

prev FP
Buffer overflow FTW

- Success! Program crashed!

- So what? What else can an attacker do?


**Stack overflow attacks**

- **Goals of attackers**
  - Take control of the flow of the target program
  - Execute malicious code, control the system (e.g., open shell as root)

- **Typical attack method**
  - Overwrite the “return address” of the current function call
  - Once the current function finishes, execution of attacker’s code with the privilege of the original process
Stack Smashing

- Attacker gives a long string with malicious code
- The string length being much larger than the space allocated
- Overflow into the stack and overwrites the return address
- The return address now points to the beginning of the malicious code
What is the challenge?

- Guess the location of the return address in the stack?
  - Return address need to point to the entry point of the malicious code
  - The entry point address is hard to guess?
  - **NOP sledding**: increase the size of the target

The exact address of malicious code (0xFFFFFD00A) may be hard to guess

But as long as you guessed the address of any of the NOPs, you eventually reach the malicious code
Simple buffer overflow attack

```c
void main() {
    char buffer[100];
    printf("Enter name: ");
    gets(buffer);
    printf("Hello, %s!\n", buffer);
}
```

```
python -c "print \n'\x90'*102 + '\xeb\xfe' + '\x04\xd0\xff\xff'" \n./a.out
```

**NOP**: instruction that says “move to the next one”

“Malicious” Code

**JUMP**

**Return to stack**
**Simple attack payload**

Valid return targets thanks to NOP sled

```plaintext
buffer 0xFFFFFD000

return addr 0xFFFFFD004
```

```
nop
nop
nop
  ...
  jmp -2
  0xFFFFFD004
```
void main() {
    char buffer[100];
    printf("Enter name: ");
    gets(buffer);
    printf("Hello, %s!\n", buffer);
}
Buffer overflow example

```c
void main() {
    char buffer[100];
    printf("Enter name: ");
    gets(buffer);
    printf("Hello, %s!\n", buffer);
}
```
Buffer overflow example

```c
void main() {
    char buffer[100];
    printf("Enter name: ");
    gets(buffer);
    printf("Hello, %s!\n", buffer);
}
```
void main() {
    char buffer[100];
    printf("Enter name: ");
    gets(buffer);
    printf("Hello, %s!\n", buffer);
}

buffer (100 bytes)
prev FP
return
Buffer overflow example

```c
void main() {
    char buffer[100];
    printf("Enter name: ");
    gets(buffer);
    printf("Hello, %s!
", buffer);
}
```
Buffer overflow example

void main() {
    char buffer[100];
    printf("Enter name: ");
    gets(buffer);
    printf("Hello, %s!\n", buffer);
}
Buffer overflow example

```c
void main() {
    char buffer[100];
    printf("Enter name: ");
    gets(buffer);
    printf("Hello, %s!\n", buffer);
}
```
Buffer overflow example

```c
void main() {
    char buffer[100];
    printf("Enter name: ");
    gets(buffer);
    printf("Hello, %s!\n", buffer);
}
```
Buffer overflow example

```c
void main() {
    char buffer[100];
    printf("Enter name: ");
    gets(buffer);
    printf("Hello, %s!\n", buffer);
}
```
void main() {
    char buffer[100];
    printf("Enter name: ");
    gets(buffer);
    printf("Hello, %s!\n", buffer);
}

Buffer overflow example
Buffer overflow example

```c
void main() {
    char buffer[100];
    printf("Enter name: ");
    gets(buffer);
    printf("Hello, %s!
", buffer);
}
```
Buffer overflow example

```c
void main() {
    char buffer[100];
    printf("Enter name: ");
    gets(buffer);
    printf("Hello, %s!
", buffer);
}
```
Buffer overflow example

```c
void main() {
    char buffer[100];
    printf("Enter name: ");
    gets(buffer);
    printf("Hello, %s!\n", buffer);
}
```

```assembly
mov %ebp, %esp
pop %ebp
ret
```

`main's FP`

`printf_args`

NOP Sled

NOP NOP JMP

0xFFFFFFFFD004
void main() {
    char buffer[100];
    printf("Enter name: ");
    gets(buffer);
    printf("Hello, %s\n", buffer);
}
Buffer overflow example

```c
void main() {
    char buffer[100];
    printf("Enter name: ");
    gets(buffer);
    printf("Hello, %s!
", buffer);
}
```
void main() {
    char buffer[100];
    printf("Enter name: ");
    gets(buffer);
    printf("Hello, %s!
", buffer);
}

main's FP
return
printf_args
NOP Sled
NOP NOP JMP

0xFEEB9090

0xFFFFFD004
Buffer overflow example

```c
void main() {
    char buffer[100];
    printf("Enter name: ");
    gets(buffer);
    printf("Hello, %s!\n", buffer);
}
```

```
mov %ebp, %esp
pop %ebp
ret
```

```
main's FP
printf_args
0x00FEEB9090
```

```
NOP Sled
NOP NOP JMP
0xFFFFFD004
```

```
0xFFFFFD004
```

```
0xFFFFFD004
```
Buffer overflow example

```c
void main() {
    char buffer[100];
    printf("Enter name: ");
    gets(buffer);
    printf("Hello, %s!\n", buffer);
}
```
Buffer overflows

- Not just for the return address
  - Function pointers
  - Arbitrary data
  - C++: exceptions
  - C++: objects
  - Heap/free list
- Any code pointer!
Another Example

gcc -fno-stack-protector buffover2.c -o buffover2

#include <stdio.h>
int main()
{
    while(1) foo();
}

int foo()
{
    unsigned int yy = 0;
    char buffer[5]; char ch; int i = 0;
    printf("Say something: ");
    while ((ch = getchar()) != '\n') buffer[i++] = ch;
    buffer[i] = '\0';
    printf("You said: %s\n", buffer);
    printf("The variable yy: %d\n", yy);
    return 0;
}
A Variation Of Buffer Overflow

- **Web applications**: overflow when passing parameters to a routine


- Web developer may just allocate 20 bytes for param1.
- How does the program handle long phone number, e.g., 1000 digits?
- Additional data overflows to the next region in the memory…
An Attack against SSH Communications (via an integer overflow bug)

```c
void do_authentication(char *user, ...) {
    int auth = 0;
    ...
    while (!auth) {
        /* Get a packet from the client */
        type = packet_read();
        switch (type) {
            ...
            case SSH_CMSG_AUTH_PASSWORD:
                if (auth_password(user, password))
                    auth = 1;
            case ...
                if (auth) break;
        }
    /* Perform session preparation. */
    do_authenticated(...);
}
```
A Countermeasure: Reducing Data Lifetime for Security

Original SSHD

do_authentication()
{
    int auth = 0;
    while (!auth) {
        type = packet_read();
        switch (type) {
        case CMSG_AUTH_PASSWORD:
            if (auth_password(passwd))
                auth = 1;
        case ...
        }
        if (auth) break;
    }
    do_authenticated(pw);
}

Modified SSHD

do_authentication()
{
    int auth = 0;
    while (!auth) {
        type = packet_read();
        auth = 0;
        switch (type) {
        case CMSG_AUTH_PASSWORD:
            if (auth_password(passwd))
                auth = 1;
        case ...
        }
        if (auth) break;
    }
    do_authenticated(pw);
}
Heap Overflow

Very similar to stack overflow
Heap Overflow

- Heap
  - Dynamically allocated memory space in run time
  - (Stack is statically allocated during compile time)

- Heap buffer overflow
  - User input data is much bigger than allocated space
  - Corrupt/overwrite internal data structures (e.g. linked list pointers)
  - There is not return address to overwrite
  - (Stack overflow often changes “control data” such as return address)

- Examples
  - iOS jailbreaking uses heap overflows to gain arbitrary code execution
  - Drive-by download: download malware without user knowledge
Defense

Buffer overflow, stack overflow
Defense against stack buffer overflow

- Canaries: a known value on the stack just before the return address
  - Check the canary when function is to return
  - Stack guard by Crispin Cowan (a gcc extension)

- Non-executable stacks
  - Malicious code in the stack cannot be executed

- Address randomization
  - Harder to guess the location of code/return address

- Compiler boundary checking
  - In Java
Stack canaries

# on function call:
canary = secret

buffers

canary

main FP

return
Stack canaries

# vulnerability:
strcpy(buffer, str)
Stack canaries

# on return:

if canary != secret:
    exception
ret
No eXecute (aka W^X aka DEP aka...)

- Attacker cannot execute code on the stack
- Mark pages as EITHER (never both)
  - Read/write (stack/heap)
  - Executable (.text/code segments)
- Challenges
  - Self-modifying code
  - JIT compilation
- Requires hardware support (MMU/MPU)
### Address Space Layout Randomization

1GB

- **Kernel space**
  - User code CANNOT read from nor write to these addresses, doing so results in a Segmentation Fault
  - 0xc0000000 == TASK_SIZE
  - Random stack offset
  - RLIMIT_STACK (e.g., 8MB)
  - Random mmap offset

3GB

- **Stack (grows down)**
- **Memory Mapping Segment**
  - File mappings (including dynamic libraries) and anonymous mappings. Example: /lib/libc.so
- **Heap**
  - Program break
  - brk
  - start_brk
  - Random brk offset
  - end_data
  - start_data
  - end_code
  - 0x00000000

- **BSS segment**
  - Uninitialized static variables, filled with zeros. Example: static char *userName;

- **Data segment**
  - Static variables initialized by the programmer. Example: static char *gonzo = "God’s own prototype";

- **Text segment (ELF)**
  - Stores the binary image of the process (e.g., /bin/gonzo)