

# Machine Programming

## Advanced Topics

CSC 235 - Computer Organization

# References

- Slides adapted from CMU

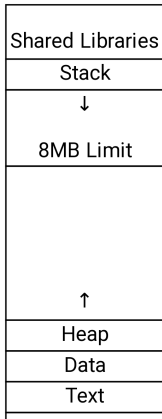
# Outline

- Memory Layout
- Buffer Overflow
  - Vulnerability
  - Protection
- Unions

# x86-64 Linux Memory Layout

- Stack
  - Runtime stack (8MB limit)
  - For example, local variables
- Heap
  - Dynamically allocated as needed
  - `malloc`, `calloc`, `new`
- Data
  - Statically allocated data
  - For example, global variables, `static` variables, string constants
- Text / Shared Libraries
  - Executable machine instructions
  - Read-only

# x86-64 Linux Memory Layout



# Memory Allocation Example

```
char big_array[1L<<24]; // 16 MB
char huge_array[1L<<31]; // 2 GB

int global = 0;

int useless() { return 0; }

int main() {
    void *phuge1, *psmall2, *phuge3, *psmall4;
    int local = 0;
    phuge1 = malloc(1L << 28); // 256 MB
    psmall2 = malloc(1L << 8); // 256 B
    phuge3 = malloc(1L << 32); // 4 GB
    psmall4 = malloc(1L << 8); // 256 B
    // some print statements
}
```

# x86-64 Example Addresses

Symbol	Example Address	Location
local	0x00007ffe4d3be87c	stack
phuge1	0x00007f7262a1e010	heap
phuge3	0x00007f7162a1d010	heap
psmall4	0x000000008359d120	heap
psmall2	0x000000008359d010	heap
big_array	0x0000000080601060	data
huge_array	0x0000000000601060	data
main()	0x000000000040060c	text
useless()	0x0000000000400590	text

- Note: there is a “gap” in the heap memory. More on this later.

# Runaway Stack Example

```
int recurse(int x) {
    int a[1<<15]; // 128 KiB
    printf("x = %d.    a at %p\n", x, a);
    a[0] = (1<<14)-1;
    a[a[0]] = x-1;
    if (a[a[0]] == 0) {
        return -1;
    }
    return recurse(a[a[0]]) - 1;
}
```

- Functions store local data on stack in stack frame.
- Recursive functions cause deep nesting of stack frames.



# Memory Referencing Bug Example

```
typedef struct {
    int a[2];
    double d;
} struct_t;

double fun(int i) {
    volatile struct_t s;
    s.d = 3.14;
    s.a[i] = 1073741824; // possibly out of bounds
    return s.d;
}
```

# Memory Referencing Bug Example

- Example inputs/outputs

- fun(0) -> 3.1400000000
- fun(1) -> 3.1400000000
- fun(2) -> 3.1399998665
- fun(3) -> 2.0000006104
- fun(6) -> Stack smashing detected
- fun(8) -> Segmentation fault

- Results are system specific

# Such Problems are a BIG deal

- Generally called a “buffer overflow”
  - when exceeding the memory size allocated for an array
- Why a big deal?
  - It is the number 1 technical cause of security vulnerabilities
  - What is the number 1 overall cause?
    - social engineering / user ignorance
- Most common form
  - Unchecked lengths on string inputs
  - Particularly for bounded character arrays on the stack
    - sometimes referred to as stack smashing

# String Library Code

- Implementation of Linux function `gets()`

```
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 library functions
  - `strcpy`, `strcat`: copy strings of arbitrary length
  - `scanf`, `fscanf`, `sscanf`: when given `%s` conversion specifier

# Vulnerable Buffer Code

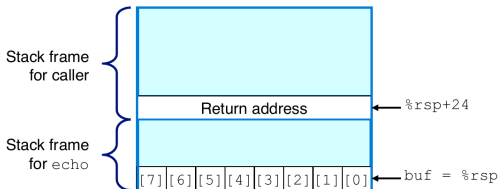
```
/* Echo Line */  
void echo() {  
    char buf[8]; // way too small  
    gets(buf);  
    puts(buf);  
}  
  
void call_echo() {  
    echo();  
}
```

# Buffer Overflow Disassembly

echo:

```
subq $24, %rsp    # allocate 24 bytes on stack
movq %rsp, %rdi   # compute buf as %rsp
call gets         # call gets
movq %rsp, %rdi   # compute buf as %rsp
call puts        # call puts
addq $24, %rsp    # deallocate stack space
ret
```

# Buffer Overflow Stack Example



---

Characters typed	Additional Corrupted State
0 - 7	None
9 - 23	Unused stack space
24 - 31	Return address
32+	Saved state in caller

---

# Stack Smashing Attacks

- Overwrite the normal return address A with address of some other code S
- When Q executes ret, will jump to other code.

```
void P() {
    Q();
    ... // <- return address A
}
void Q() {
    char buf[64];
    gets(buf);
    ...
    return ...;
}
void S() {
    // something unexpected
}
```



# Crafting Smashing String

- Target C code

```
void smash() {  
    printf("I've been smashed!\n");  
    exit(0);  
}
```

- Target Disassembly

```
00000000004006c8 <smash>:  
    4006c8:      48 83 ec 08
```

- Goal: craft a string that overwrites the return address with the address of the target function.

# Performing Stack Smash

- Put hex string in file `smash-hex.txt`
- Use `hexify` program to convert hex digits to characters
  - Some of them are non-printing
- Provide as input to a vulnerable program.

# Code Injection Attacks

- Input string contains byte representation of executable code
- Overwrite return address A with address of buffer B
- When Q executes ret, will jump to exploit code

```
void P() {  
    Q();  
    ... // <- return address A  
}
```

```
int Q() {  
    char buf[64];  
    gets(buf);  
    ...  
    return ...;  
}
```

# Preventing Buffer Overflow Attacks

- Avoid overflow vulnerabilities
- Employ system-level protections
- Have compiler use “stack canaries”

# 1. Avoid Overflow Vulnerabilities in Code

```
void echo() {  
    char buf[4];  
    fgets(buf, 4, stdin);  
    puts(buf);  
}
```

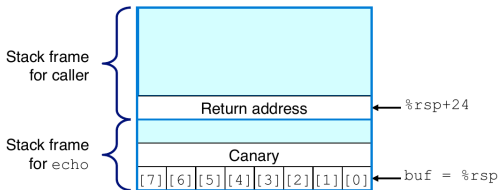
- For example, use library routines that limit string lengths
  - fgets instead of gets
  - strncpy instead of strcpy
  - Do not use scanf with %s conversion specifier
    - Use fgets to read the string
    - or use %ns where n is a suitable integer

# System-Level Protections Can Help

- Randomize stack offsets
  - At start of program, allocate random amount of space on stack
  - Shift stack addresses for entire program
  - Makes it difficult for a hacker to predict the beginning of inserted code
- Non-executable code segments
  - In tradition x86 can mark region of memory as “read-only” or “writable”
    - Can execute anything readable
  - x86-64 added explicit “execute” permission
  - Stack marked as non-executable

# Stack Canaries Can Help

- Idea
  - Place a special value (“canary”) on stack just beyond buffer
  - Check for corruption before exiting function
- GCC Implementation
  - Flag `-fstack-protector`
- Example



# Return-Oriented Programming Attacks

- Challenge (for hackers)
  - Stack randomization makes it hard to predict buffer location
  - Marking stack non-executable makes it hard to insert binary code
- Alternative Strategy
  - Use existing code
    - For example, library code from `stdlib`
  - String together fragments to achieve overall desired outcome
  - Does not overcome stack canaries
- Construct program from *gadgets*
  - Sequence of instructions ending in `ret`
    - Encoded by single byte `0xc3`
  - Code positions fixed from run to run
  - Code is executable



# Gadget Example #1

- C example code

```
long ab_plus_c(long a, long b, long c) {  
    return a*b + c;  
}
```

- Disassembly

```
0000000000000000 <ab_plus_c>:  
0:   f3 0f 1e fa           endbr64  
4:   48 0f af fe           imul   %rsi,%rdi  
8:   48 8d 04 17          lea    (%rdi,%rdx,1),%rax  
c:   c3                   retq
```

- $rax \leftarrow rdi + rdx$
- Gadget address = 0x8

- Use tail end of existing functions

# Gadget Example #2

- C example code

```
void setval(unsigned *p) {  
    *p = 3347663060u;  
}
```

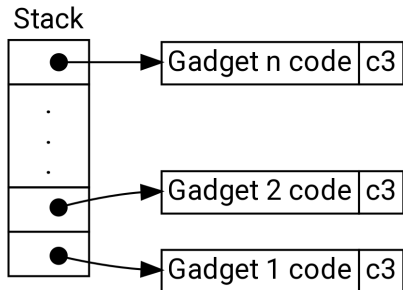
- Disassembly

```
000000000000000d <setval>:  
    d:   f3 0f 1e fa           endbr64  
11:   c7 07 d4 48 89 c7       movl    $0xc78948d4, (%rdi)  
17:   c3                       retq
```

- Bytes 48 89 c7 encodes `movq %rax, %rdi%`
  - `rdi ← rax`
  - Gadget address = 0x15
- Repurpose byte codes

# Return-Oriented Programming Execution

- Trigger with `ret` instruction
  - Will start executing Gadget 1
- Final `ret` in each gadget will start next one
  - `ret`: pop address from stack and jump to that address



# Union Allocation

- Allocate according to largest element
- Can only use one field at a time
- Union example: 8 bytes total storage

```
union U1 {  
    char c;    // 1 byte  
    int i[2]; // 8 bytes  
    double v; // 8 bytes  
} *up;
```

- Struct example: 24 bytes total

```
struct S1 {  
    char c;    // 1 byte + 3 bytes padding  
    int i[2]; // 8 bytes + 4 bytes padding  
    double v; // 8 bytes  
} *up;
```

# Using Union to Access Bit Patterns

```
typedef union {
    float f;
    unsigned u;
} bit_float_t;

float bit2float(unsigned u) {
    bit_float_t arg;
    arg.u = u;
    return arg.f;
}

unsigned float2bit(float f) {
    bit_float_t arg;
    arg.f = f;
    return arg.u;
}
```

# Byte Ordering Revisited

- Idea
  - Short/long/quad words stored in memory as 2/4/8 consecutive bytes
  - Which byte is most (least) significant?
  - Can cause problems when exchanging binary data between machines
- Big Endian
  - Most significant byte has lowest address
  - Sparc, Internet
- Little Endian
  - Least significant byte has lowest address
  - x86, ARM
- Bi Endian
  - Can be configured either way
  - ARM

# Summary of Compound Types in C

- Arrays
  - Contiguous allocation of memory
  - Aligned to satisfy every element's alignment requirement
  - Pointer to first element
  - No bounds checking
- Structures
  - Allocate bytes in order declared
  - Pad in middle and at end to satisfy alignment
- Unions
  - Overlay declarations
  - Way to circumvent type system

# Summary

- Memory Layout
- Buffer Overflow
  - Vulnerability
  - Protection
  - Code Injection Attack
  - Return-Oriented Programming
- Unions