Today

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

x86-64 Linux Memory Layout

- Stack
  - Runtime stack (default 8MB soft limit)
  - E.g., local variables
- Heap
  - Dynamically allocated as needed
  - When call malloc(), calloc(), new()
- Data
  - Statically allocated data
  - E.g., global vars, static vars, string constants
- Text / Shared Libraries
  - Executable machine instructions
  - Read-only

x86-64 Example Addresses

- local
- p1
- p3
- p4
- p2
- big_array
- huge_array
- main()
- useless()

Memory Allocation Example

```c
char big_array[1L<<24]; /* 16 MB */
char huge_array[1L<<31]; /* 2 GB */
int global = 0;
int useless() { return 0; }
int main ()
{
  void *p1, *p2, *p3, *p4;
  int local = 0;
  p1 = malloc(LL << 28); /* 256 MB */
  p2 = malloc(LL << 8); /* 256 B */
  p3 = malloc(LL << 32); /* 4 GB */
  p4 = malloc(LL << 8); /* 256 B */
  /* Some print statements ... */
}
```
Recall: Memory Referencing Bug Example

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

double fun(int i) {
    volatile struct_t *s = &s;
    s->a[i] = 1073741824; /* Possibly out of bounds */
    return s->d;
}

fun(0) → 3.14
fun(1) → 3.14
fun(2) → 3.1399998664856
fun(3) → 2.00000061035156
fun(4) → 3.14
fun(6) → Segmentation fault

Result is system specific

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

double fun(int i) {
    volatile struct_t *s = &s;
    s->a[i] = 1073741824; /* Possibly out of bounds */
    return s->d;
}

fun(0) → 3.14
fun(1) → 3.14
fun(2) → 3.1399998664856
fun(3) → 2.00000061035156
fun(4) → 3.14
fun(6) → Segmentation fault

Such problems are a BIG deal

- Generally called a “buffer overflow”
  - when exceeding the memory size allocated for an array
- Why a big deal?
  - One of the most common technical causes of security vulnerabilities
- 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 old standard C 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;
  }
  ```

  * Bad design: 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 specification

Vulnerable Buffer Code

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

void call_echo() {
    echo();
}
```

btw, how big is big enough?

unix$ ./bufdemo-nsp
Type a string: 0123456789012345678901234567890123
012345678901234567890123
unix$ ./bufdemo-nsp
Type a string: 0123456789012345678901234
Segmentation Fault

Buffer Overflow Disassembly

echo:

```
00000000004006cf <echo>:
4006cf: 48 83 ec 18 sub $0x18,%rsp
4006d3: 48 89 a7 mov %esp,%rdi
4006d6: 48 89 a8 mov %rdi,%rdi
4006da: 48 3d fa ff ff callq 400520 <puts@plt>
4006e3: 48 83 c4 18 add $0x18,%rsp
4006e7: c3 retq
```

call_echo:

```
00000000004006e8 <call_echo>:
4006e8: 48 83 ec 18 sub $0x8,%rsp
4006e9: 48 00 00 00 00 mov $0x0,%eax
4006fd: 48 89 4f ff ff callq 4006f0 <echo>
4006f0: 48 34 c4 08 add $0x8,%rsp
4006f4: c3 retq
```

Memory Referencing Bug Example

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

fun(0) → 3.14
fun(1) → 3.14
fun(2) → 3.1399998664856
fun(3) → 2.00000061035156
fun(4) → 3.14
fun(6) → Segmentation fault
```
Buffer Overflow Stack Example #3

After call to gets

Stack Frame for call_echo

00 00 00 00
00 40 00 00
39 38 37 36
35 34 33 32
31 30 39 38
37 36 35 34
33 32 31 30
buf ← trap

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

bufdemo

call_echo:

4006f1: callq 4006cf <echo>
4006f6: add $0x8,trap
...

buf ← trap

register_tm_clones:

400600: moveq %rax,%rdx
400603: mov %esp,%rbp
400606: shr $0x3f, %rdx
40060a: add %rdx,%rax
40060d: sar %rax
400610: jne 400614
400612: pop %rbp
400613: retq

"Returns" to unrelated code
Lots of things happen, without modifying critical state
Eventually executes setq back to main

Overflown buffer, corrupted return pointer, but program seems to work!
**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

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

Exploits Based on Buffer Overflows

- **Buffer overflow bugs can allow remote machines to execute arbitrary code on victim machines**
- Distressingly common in real programs
  - Programmers keep making the same mistakes 😓
  - Recent measures make these attacks much more difficult
- **Examples across the decades**
  - Original “Internet worm” (1988)
  - “IM wars” (1999)
  - Twilight hack on Wii (2000s)
  - ... and many, many more
- **You will learn some of the tricks in attacklab**
  - Hopefully to convince you to never leave such holes in your programs!!

**Example: the original Internet worm (1988)**

- Exploited a few vulnerabilities to spread
  - 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.
- Once on a machine, scanned for other machines to attack
  - invaded ~6000 computers in hours (10% of the Internet 😊)
  - saw June 1989 article in Comm. of the ACM
- the young author of the worm was prosecuted
- and CERT was formed

**Discussion Break: Unknown Addresses?**

- Basic attack requires attacker to know address B of buffer
- Is an attack still possible if B is variable?
- E.g. what if attacker only knows B +/- 30?

- Some possible attack strategies:
  - Try attack repeatedly
  - “NOP sled”: (0x90 is one-byte no-operation in x86)

**Aside: Worms and Viruses**

- **Worm**: A program that
  - Can run by itself
  - Can propagate a fully working version of itself to other computers
- **Virus**: Code that
  - Adds itself to other programs
  - Does not run independently
- Both are (usually) designed to spread among computers and to wreak havoc

**Announcements: Cheating**

- All assignments and labs in 2021 are individual: no group work
  - OK to discuss concepts, but answers must be your own
  - Similarly, don’t look for answers online
- Minimum penalty is 0 on assignment and report to OSCAI
- We found suspicious similarity on both HW1 and Data Lab
  - Using both staff vigilance and automated analysis
- Please do not cheat!
Announcements: Schedule

- I have left-over assignment 1s available after class
- Homework assignment 2 due at beginning of class on Wednesday
- Bomb lab due 11:55pm Friday night
- Midterm in class next Monday
  - Friday lecture is review

OK, what to do about buffer overflow attacks

- Avoid overflow vulnerabilities
- Employ system-level protections
- Have compiler use “stack canaries”
- Let’s talk about each...

1. Avoid Overflow Vulnerabilities in Code (!)

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

- For example, use library routines that limit string lengths
  - fgets instead of gets
  - strncpy instead of strcpy
  - Don’t use scanf with %s conversion specification

1. Avoid Overflow Vulnerabilities in Code (!)

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

OK, what to do about buffer overflow attacks

- Avoid overflow vulnerabilities
- Employ system-level protections
- Have compiler use “stack canaries”
- Let’s talk about each...

2. System-Level Protections can help

- Randomized stack offsets
  - At start of program, allocate random amount of space on stack
  - Shifts stack addresses for entire program
  - Makes it difficult for hacker to predict beginning of inserted code
  - E.g.: 5 executions of memory allocation code
  - Stack repositioned each time program executes

2. System-Level Protections can help

- Randomized stack offsets
  - At start of program, allocate random amount of space on stack
  - Shifts stack addresses for entire program
  - Makes it difficult for hacker to predict beginning of inserted code
  - E.g.: 5 executions of memory allocation code
  - Stack repositioned each time program executes

3. Stack Canaries can help

- Idea
  - Place special value (“canary”) on stack just beyond buffer
  - Check for corruption before exiting function
- GCC Implementation
  - -fstack-protector
  - Now commonly enabled by default

3. Stack Canaries can help

- Idea
  - Place special value (“canary”) on stack just beyond buffer
  - Check for corruption before exiting function
- GCC Implementation
  - -fstack-protector
  - Now commonly enabled by default
Protected Buffer Disassembly

```
sub $0x18,%rsp
mov %fs:0x28, %rax
mov %rax,0x8(%rsp)
ox %eax, %eax
mov %rsp, %rdi
callq 4006e0 <gets>
mov %rsp, %rdi
callq 400570 <puts@plt>
mov 0x8(%rsp), %rax
xor %fs:0x28, %rax
add $0x18,%rsp
retq
```

Setting Up Canary

```
sub $0x18,%rsp
mov %fs:0x28, %rax
mov %rax,0x8(%rsp)
ox %eax, %eax
mov %rsp, %rdi
callq 4006e0 <gets>
mov %rsp, %rdi
callq 400570 <puts@plt>
mov 0x8(%rsp), %rax
xor %fs:0x28, %rax
je 400768 <echo+0x39>
callq 400580 <__stack_chk_fail@plt>
add $0x18,%rsp
retq
```

Checking Canary

```
movq 8(%%rsp), %rax # Retrieve from stack
movq %fs:0x28, %rax # Compare to canary
je .L2 # If same, OK
call __stack_chk_fail # FAIL
```

Return-Oriented Programming Attacks

- **Challenge (for hackers)**
  - Stack randomization makes it hard to predict buffer location
  - Marking stack nonexecutable makes it hard to insert binary code
- **Alternative Strategy**
  - Use existing code
    - E.g., library code from stdlib
  - String together fragments to achieve overall desired outcome
    - Does not on its own 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

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

```
00000000004004d0 <ab_plus_c>
4004d0:  48 0f af fe
        imul %rdi,%rsi
4004d4:  48 8d 04 17
        lea (%rdi,%rdx,1),%rax
4004d8:  c3           retq
```

Use tail end of existing functions

Gadget address = 0x4004d4

Gadget Example #2

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

```
<setval>:
4004d9:  c7 07 d4 48 89
c7"movl$0xc78948d4,(%rdi)
4004dc: c7 retq
```

Repurpose instruction bytes

Gadget address = 0x4004dc
ROP Execution

- Trigger with `ret` instruction
  - Will start executing Gadget 1
- Final `ret` in each gadget will start next one

Union Allocation

- Allocate according to largest element
- Can only use one field at a time

Using Union to Access Bit Patterns

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

- Little Endian
  - Least significant byte has lowest address
  - Intel x86, ARM Android and IOS

- Bi Endian
  - Can be configured either way
  - ARM

Byte Ordering Example

- 32-bit
  - `char` 1 byte
  - `short` 2 bytes
  - `int` 4 bytes
  - `long` 8 bytes

- 64-bit
  - `char` 1 byte
  - `short` 2 bytes
  - `int` 4 bytes
  - `long` 8 bytes

Today

- Memory Layout
- Buffer Overflow
  - Vulnerability
  - Protection
- Unions
Byte Ordering Example (Cont).

```c
int j;
for (j = 0; j < 8; j++)
    dw.c[j] = 0xf0 + j;
printf("Characters 0-7 == [0x%x,0x%x,0x%x,0x%x,0x%x,0x%x,0x%x,0x%x]\n", dw.c[0], dw.c[1], dw.c[2], dw.c[3], dw.c[4], dw.c[5], dw.c[6], dw.c[7]);
printf("Shorts 0-3 == [0x%x,0x%x,0x%x,0x%x]\n", dw.s[0], dw.s[1], dw.s[2], dw.s[3]);
printf("Ints 0-1 == [0x%x,0x%x]\n", dw.i[0], dw.i[1]);
printf("Long 0 == [0x%lx]\n", dw.l[0]);
```

Byte Ordering on IA32

Little Endian

```
0    1    2    3    4    5    6    7
```

Output:

Characters 0-7 == [0xf0, 0xf1, 0xf2, 0xf3, 0xf4, 0xf5, 0xf6, 0xf7]
Shorts 0-3 == [0xf0f1, 0xf2f3, 0xf4f5, 0xf6f7]
Ints 0-1 == [0xf3f2f1f0, 0xf7f6f5f4]
Long 0 == [0xf3f2f1f0]

Byte Ordering on Sun

Big Endian

```
0    1    2    3    4    5    6    7
```

Output on Sun:

Characters 0-7 == [0xf0, 0xf1, 0xf2, 0xf3, 0xf4, 0xf5, 0xf6, 0xf7]
Shorts 0-3 == [0xf0f1f2, 0xf3f4f5, 0xf6f7f8]
Ints 0-1 == [0xf3f2f1f0, 0xf7f6f5f4]
Long 0 == [0xf3f2f1f0]

Byte Ordering on x86-64

Little Endian

```
0    1    2    3    4    5    6    7
```

Output on x86-64:

Characters 0-7 == [0xf0, 0xf1, 0xf2, 0xf3, 0xf4, 0xf5, 0xf6, 0xf7]
Shorts 0-3 == [0xf0f1f2f3, 0xf4f5f6, 0xf7f8f9]
Ints 0-1 == [0xf3f2f1f0, 0xf7f6f5f4]
Long 0 == [0xf3f2f1f0]

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