Lecture 26: Security

Lecture Participation Poll #26

Log onto pollev.com/cse374
Or
Text CSE374 to 22333

CSE 374: Intermediate Programming Concepts and Tools
Administrivia

- HW 5 (final HW) posted
- Final review assignment will release last week of quarter
- End of quarter due date Wednesday December 16th @ 9pm
Human to Computer Roadmap

C:

```c
    car *c = malloc(sizeof(car));
c->miles = 100;
c->gals = 17;
    float mpg = get_mpg(c);
    free(c);
```

Java:

```java
    Car c = new Car();
c.setMiles(100);
c.setGals(17);
    float mpg = c.getMPG();
```

Assembly language:
```
get_mpg:
    pushq  %rbp
    movq  %rsp, %rbp
    ...
    popq  %rbp
    ret
```

Machine code:
```
0111010000011000
100011010000100000000010
1000100111000010
110000011111110100001111
```

Computer system:

OS:
Windows 10
OS X Yosemite
Assembly Instruction Basics

Assembly instructions fall into one of 3 categories:

- **Transfer data** between memory and register
  - Load data from memory into register
    - \%reg = Mem\[address\]
  - Store register data into memory
    - Mem\[address\] = %reg

- **Perform arithmetic** operation on register or memory data
  - \(c = a + b; \quad z = x << y; \quad i = h \& g;\)

- **Control flow**: what instruction to execute next
  - Unconditional jumps to/from procedures
  - Conditional branches

Items in Assembly fall into one of 3 operand categories:

- **Immediate**: Constant integer data
  - Examples: $0x400, $-533
  - Like C literal, but prefixed with ‘$’
  - Encoded with 1, 2, 4, or 8 bytes

- **Register**: 1 of 16 integer registers
  - Examples: \%rax, \%r13

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>1st argument (x)</td>
</tr>
<tr>
<td>%rsi</td>
<td>2nd argument (y)</td>
</tr>
<tr>
<td>%rax</td>
<td>return value</td>
</tr>
</tbody>
</table>

- **Memory**: Consecutive bytes of memory at a computed address
  - Simplest example: (\%rax)
Example: Moving Data

- **General form:** `mov_ source, destination`
  - Missing letter (_) specifies size of operands
  - Lots of these in typical code

**Examples:**

- **`movb src, dst`**
  - Move 1-byte “byte”

- **`movw src, dst`**
  - Move 2-byte “word”

- **`movl src, dst`**
  - Move 4-byte “long word”

- **`movq src, dst`**
  - Move 8-byte “quad word”

---

<table>
<thead>
<tr>
<th>Source</th>
<th>Dest</th>
<th>Src, Dest</th>
<th>C Analog</th>
</tr>
</thead>
<tbody>
<tr>
<td>Imm</td>
<td>Reg</td>
<td><code>movq $0x4, %rax</code></td>
<td><code>rax = 4;</code></td>
</tr>
<tr>
<td>Mem</td>
<td></td>
<td><code>movq $-147, (%rax)</code></td>
<td><code>*rax = -147;</code></td>
</tr>
<tr>
<td>Reg</td>
<td>Reg</td>
<td><code>movq %rax, %rdx</code></td>
<td><code>rdx = rax;</code></td>
</tr>
<tr>
<td>Mem</td>
<td>Reg</td>
<td><code>movq %rax, (%rdx)</code></td>
<td><code>*rdx = rax;</code></td>
</tr>
<tr>
<td>Mem</td>
<td>Reg</td>
<td><code>movq (%rax), %rdx</code></td>
<td><code>rdx = *rax;</code></td>
</tr>
</tbody>
</table>
Example: Arithmetic Operations

```c
long simple_arith(long x, long y) {
    long t1 = x + y;
    long t2 = t1 * 3;
    return t2;
}
```

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>1\textsuperscript{st} argument (x)</td>
</tr>
<tr>
<td>%rsi</td>
<td>2\textsuperscript{nd} argument (y)</td>
</tr>
<tr>
<td>%rax</td>
<td>return value</td>
</tr>
</tbody>
</table>

```assembly
simple_arith:
addq  %rdi, %rsi
imulq $3, %rsi
movq  %rsi, %rax
ret
```

```assembly
y += x;
y *= 3;
long r = y;
return r;
```
Example: swap()

```c
void swap(long *xp, long *yp) {
    long t0 = *xp;
    long t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```

```plaintext
swap:
    movq (%rdi), %rax
    movq (%rsi), %rdx
    movq %rdx, (%rdi)
    movq %rax, (%rsi)
    ret
```
Example: swap()

### Registers

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>0x120</td>
</tr>
<tr>
<td>%rsi</td>
<td>0x100</td>
</tr>
<tr>
<td>%rax</td>
<td>123</td>
</tr>
<tr>
<td>%rdx</td>
<td>456</td>
</tr>
</tbody>
</table>

### Memory

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x120</td>
<td>123</td>
</tr>
<tr>
<td>0x108</td>
<td>456</td>
</tr>
</tbody>
</table>

### Word Address

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x120</td>
<td></td>
</tr>
<tr>
<td>0x118</td>
<td></td>
</tr>
<tr>
<td>0x110</td>
<td></td>
</tr>
<tr>
<td>0x108</td>
<td></td>
</tr>
<tr>
<td>0x100</td>
<td></td>
</tr>
</tbody>
</table>

### Code:

```assembly
swap:
    movq (%rdi), %rax  # t0 = *xp
    movq (%rsi), %rdx  # t1 = *yp
    movq %rdx, (%rdi)  # *xp = t1
    movq %rax, (%rsi)  # *yp = t0
    ret
```
Example: swap()

**Registers**

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>0x120</td>
</tr>
<tr>
<td>%rsi</td>
<td>0x100</td>
</tr>
<tr>
<td>%rax</td>
<td>123</td>
</tr>
<tr>
<td>%rdx</td>
<td>456</td>
</tr>
</tbody>
</table>

**Memory**

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x120</td>
<td>456</td>
</tr>
<tr>
<td>0x110</td>
<td>123</td>
</tr>
</tbody>
</table>

**Word Address**

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x120</td>
<td></td>
</tr>
<tr>
<td>0x118</td>
<td></td>
</tr>
<tr>
<td>0x110</td>
<td></td>
</tr>
<tr>
<td>0x108</td>
<td></td>
</tr>
<tr>
<td>0x100</td>
<td></td>
</tr>
</tbody>
</table>

**swap:**

```assembly
movq (%rdi), %rax  # t0 = *xp
movq (%rsi), %rdx  # t1 = *yp
movq %rdx, (%rdi)  # *xp = t1
movq %rax, (%rsi)  # *yp = t0
ret
```
Where does everything go?

```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(1L << 28); /* 256 MB */
    p2 = malloc(1L << 8); /* 256 B */
    p3 = malloc(1L << 32); /* 4 GB */
    p4 = malloc(1L << 8); /* 256 B */
    /* Some print statements ... */
}
```
Function Pointers & Frames

- Coded instructions are translated into numerical values stored in memory and fed into the processor for execution.

- Function pointer – address of a function stored in memory, pointing to the start of the block of memory storing the set of instructions expressed by the function.

- Stack frames – section of the stack that is set aside for each function call.
  - Frame pushed onto the stack when the function is called and popped off when the function returns.
  - Each frame contains: arguments, return address, pointer to last frame, local variables.
Procedure Call Overview

- Coordinating between function memory frames
  - Callee must know where to find arguments
  - Callee must know where to find return address
  - Caller must know where to find return value

- Caller and Callee run on the same CPU, so they use the same registers

- Calling convention - convention of where to leave/find things
  - caller saves contents of %rax before triggering callee that returns value (to prevent lose due to overwrite)
  - callee places return value into %rax
  - for values greater than 8 bytes, return pointer
What is a Buffer?

- A buffer is an array used to temporarily store data
  - You’ve probably seen “video buffering…”
  - Functions that accept user input set aside memory for incoming data
  - Specify size of buffer before you know size of user input

```c
void echo() {
    char buf[8];
    gets(buf);
    puts(buf);
}
```
Unix buffer overflow vulnerability

- C does not check array bounds, no way to specify limit on number of characters to read into a function
  - arrays in C/C++ don’t store their length
  - Many Unix/Linux/C functions don’t check argument sizes
    - strcpy: copies string of arbitrary length to a destination
    - scanf, fscanf, sscanf,
- Allows overflowing (writing past the end) of buffers (arrays)
  - Buffer Overflow - Writing past the end of an array
- Provides opportunities for malicious programs
  - Stack grows “backwards” in memory
  - Data and instructions both stored in the same memory
  - surprisingly easy to exploit, programmers often leave code open to attacks

Implementation of Unix gets()

```c
/* Get string from stdin */
char* gets(char* dest) {
    int c = getchar();
    char* p = dest;
    while (c != EOF && c != '
') {
        *p++ = c;
        c = getchar();
    }
    *p = '\0';
    return dest;
}
```
Buffer Overflow

- Stack grows down towards lower addresses
- Buffer grows up towards higher addresses
- If we write past the end of the array, we overwrite data on the stack!

Enter input: hello
-> no overflow

Enter input: helloabcdef
-> overflow!
What happens when there is an overflow?

- Buffer overflows on the stack can overwrite “interesting” data
  - Attackers just choose the right inputs

- Simplest form (sometimes called “stack smashing”)
  - Unchecked length on string input into bounded array causes overwriting of stack data
  - Try to change the return address of the current procedure

- Why is this a big deal?
  - It was the #1 technical cause of security vulnerabilities
    - #1 overall cause is social engineering / user ignorance

Enter input: helloabcdef

We’ve lost our way!
Lost address of function pointer telling us which instruction to return to
Malicious Buffer Overflow – Code Injection

- Buffer overflow bugs can allow attackers to execute arbitrary code on victim machines
  - Distressingly common in real programs
- Input string contains byte representation of executable code
- Overwrite return address A with address of buffer B
- When bar() executes ret, will jump to exploit code

```c
void foo() {
    bar();
    A:... return address A
}
```

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

Stack after call to `gets()`

- foo stack frame
- bar stack frame

buf starts here → B

Change return to last frame

- Skip the line "x = 1;" in the main function by modifying function's return address.
  - Identify where the return address is in relation to the local variable buffer1
  - Figure out how many bytes the actual compiled C instruction "x=1;" takes, so that we can increment by that many bytes

- Use GDB
  - break function
    - break right at beginning of function execution
  - x buffer1
    - prints the location of buffer1
  - info frame
    - "rip" will hold the location of the return address
  - print <rip-location> - <buffer1-location>
    - prints the number of bytes between buffer1 and rip
  - disassemble main
    - shows the machine code and how many bytes each instruction takes up.
    - We identify the line that calls function, then see that the next // instruction moves 1 into x. That instruction takes 7 bytes, so we
      - have now found the second number!

```c
void bufferplay (int a, int b, int c) {
    char buffer1[5];
    uintptr_t ret; //holds an address
    //calculate the address of the return pointer
    ret = (uintptr_t) buffer1 + 0; //change to be address of return
    //treat that number like a pointer,
    //and change the value in it
    *((uintptr_t*)ret) += 0; //change to add how much to advance
}
int main(int argc, char** argv) {
    int x;
    x = 0;
    printf("before: %d\n",x);
    bufferplay (1,2,3);
    x = 1; // want to skip this line
    printf("after: %d\n",x);
    return 0;
}
```
Trigger malicious program

```c
int bar(char *arg, char *out) {
    strcpy(out, arg);
    return 0;
}
void foo(char *argv[]) {
    char buf[256];
    bar(argv[1], buf);
}
int main(int argc, char *argv[]) {
    if (argc != 2) {
        fprintf(stderr, "target1: argc != 2\n");
        exit(1);
    }
    foo(argv);
    return 0;
}
```

Attacker Program

```c
int main(void) {
    char *args[3];
    char *env[1];
    args[0] = "/tmp/target";
    args[2] = NULL;
    env[0] = NULL;

    args[1] = (char*) malloc(sizeof(char)*265);
    memset(args[1], 0x90, 264);
    // Null-terminate the string.
    args[1][264] = '\0';

    // Add in the attack code to the front of the argument.
    memcpy(args[1], shellcode, strlen(shellcode));
    *(uintptr_t*)(args[1] + 264) = 0x7fffffffb90;
    // call the victim program.
    execve("/tmp/target", args, env);
}
```
Hack – Internet Worm

▪ Original “Internet worm” (1988)

▪ Exploited vulnerability in `gets()` method used in Finger protocol
  - Worm attacked fingerd server with phony argument
    - `finger "exploit-code padding new-return-addr"`
    - Exploit code: executed a root shell on the victim machine with a direct connection to the attacker

▪ Worm spread from machine to machine automatically
  - denial of service attack – flood machine with so many requests it is overloaded and unavailable to its intended users
  - took down 6000 machines, took days to get machine back online
  - government estimated damage $100,000 to $10,000,000

▪ Written by Robert Morris while a grad student at Cornell, but launched it from the MIT computer system
  - meant to be an intellectual experiment, but made it too damaging by accident
  - Now a professor at MIT, first person convicted under the ’86 Computer Fraud and Abuse Act
Hack – Heartbleed

- Buffer over-read in Open-Source Security Library
  - when program reads beyond end of intended data from a buffer and reads

- Maliciously designed input – “Heartbeat” packet sent out
  - Specifies length of message and server echoes it back
  - Library just “trusted” this length
  - Allowed attackers to read contents of memory anywhere they wanted

- Est. 17% of internet affected
Protect Your Code!

- **Employ system-level protections**
  - Code on the Stack is not executable
  - Randomized Stack offsets

- **Avoid overflow vulnerabilities**
  - Use library routines that limit string lengths
  - Use a language that makes them impossible

- **Have compiler use “stack canaries”**
  - place special value (“canary”) on stack just beyond buffer
System Level Protections

- Non-executable code segments
  - In traditional x86, can mark region of memory as either “read-only” or “writeable”
    - Can execute anything readable
  - x86-64 added explicit “execute” permission
- Stack marked as non-executable
  - Do NOT execute code in Stack, Static Data, or Heap regions
  - Hardware support needed
System Level Protections

▪ Many embedded devices do not have feature to mark code as “non-executable”
  - Cars
  - Smart homes
  - Pacemakers

▪ Randomized stack offsets
  - At start of program, allocate random amount of space on stack
  - Shifts stack addresses for entire program
    - Addresses will vary from one run to another
  - Makes it difficult for hacker to predict beginning of inserted code
Avoid Overflow Vulnerabilities

- Use library routines that limit string lengths
  - fgets instead of gets (2\textsuperscript{nd} 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

- Or… don’t use C – use a language that does array index bounds check
  - Buffer overflow is impossible in Java
    - ArrayIndexOutOfBoundsException
  - Rust language was designed with security in mind
    - Panics on index out of bounds, plus more protections

```c
/* Echo Line */
void echo()
{
    char buf[8]; /* Way too small! */
    fgets(buf, 8, stdin);
    puts(buf);
}
```
Stack Canaries

▪ Basic Idea: place special value ("canary") on stack just beyond buffer
  - Secret value that is randomized before main()
  - Placed between buffer and return address
  - Check for corruption before exiting function

▪ GCC implementation
  - -fstack-protector

```bash
unix> ./buf
Enter string: 12345678
12345678

unix> ./buf
Enter string: 123456789
*** stack smashing detected ***
```
What is Concurrency?

- Running multiple processes simultaneously
  - running separate programs simultaneously
  - running two different ‘threads’ in one program

- Each ‘process’ is one ‘thread’

- Parallelism refers to running things simultaneously on separate resources (e.g., separate CPUs)
- Concurrency refers to running multiple threads on a shared resource
- Sequential programming demands finishing one sequence before starting the next one
- Previously, performance improvements could only be made by improving hardware
  - Moore’s Law

- Allows processes to run ‘in the background’
  - Responsiveness – allow GUI to respond while computation happens
  - CPU utilization – allow CPU to compute while waiting (waiting for data, for input)
  - Isolation – keep threads separate so errors in one don’t affect the others
Concurrentiy

- C and Java support parallelism similarly
  - one pile of code, globals, heap
  - multiple "stack + program counter’s" – called threads
  - threads are run or pre-empted by a scheduler
  - threads all share the same memory
  - Various synchronization mechanisms control when threads run
    - don’t run until I’m done with this

- C: the POSIX Threads (pthreads) library)
  - #include <pthread.h>
  - pass -lpthread to gcc (when linking)
  - pthread_create takes a function pointer and arguments, run as a separate thread

- Java: built into the language
  - subclass java.lang.Thread, and override the run method
  - create a Thread object and call its start method
  - any object can "be synchronized on" (later today)
Pthread functions

▪ `pthread_t` thread ID;
  - the threadID keeps track of to which thread we are referring

▪ `int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start routing) (void*), void *arg);`
  - note – pthread_create takes two generic (untyped) pointers
  - interprets the first as a function pointer and the second as an argument pointer

▪ `int pthread_join(pthread_t thread, void **value_ptr);`
  - puts calling thread ‘on hold’ until ‘thread’ completes – useful for waiting to thread to exit

https://pubs.opengroup.org/onlinepubs/7908799/xsh/pthread.h.html
Memory Consideration

• if one thread did nothing of interest to any other thread, why bother running?

• threads must communicate and coordinate
  - use results from other threads, and coordinate access to shared resources

• simplest ways to not mess each other up:
  - don’t access same memory (complete isolation)
  - don’t write to shared memory (write isolation)

• next simplest
  - one thread doesn’t run until/unless another is done
Parallel Processing

- common pattern for expensive computations (such as data processing)

1. split up the work, give each piece to a thread (fork)
2. wait until all are done, then combine answers (join)
   - to avoid bottlenecks, each thread should have about the same about of work

- performance will always be less than perfect speedup

- what about when all threads need access to the same mutable memory?
multiple threads with one memory

- often you have a bunch of threads running at once and they might need the same mutable (writable) memory at the same time but probably not
  - want to be correct, but not sacrifice parallelism

- example: bunch of threads processing bank transactions
data races
Questions
Protected Buffer Disassembly (buf)

```
400607:  sub    $0x18,%rsp
40060b:  mov    %fs:0x28,%rax
400614:  mov    %rax,0x8(%rsp)
400619:  xor    %eax,%eax
          ... call printf ...
400625:  mov    %rsp,%rdi
400628:  callq  400510 <gets@plt>
40062d:  mov    %rsp,%rdi
400630:  callq  4004d0 <puts@plt>
400635:  mov    0x8(%rsp),%rax
40063a:  xor    %fs:0x28,%rax
400643:  jne    40064a <echo+0x43>
400645:  add    $0x18,%rsp
400649:  retq
40064a:  callq  4004f0
          <__stack_chk_fail@plt>
```
Setting up Canary

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

**Before call to gets**

- Stack frame for `call_echo`
- Return address (8 bytes)
- Canary (8 bytes)

```
[3] [2] [1] [0]
```

```
buf ← %rsp
```

- `movq %fs:40, %rax` # Get canary
- `movq %rax, 8(%rsp)` # Place on stack
- `xorl %eax, %eax` # Erase canary
```
```
Checking Canary

After call to gets

Stack frame for call_echo

Return address (8 bytes)

Canary (8 bytes)

00 37 36 35
34 33 32 31

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

echo:
    ...
    movq %fs:40, %rax    # Get canary
    movq %rax, 8(%rsp)   # Place on stack
    xorl %eax, %eax      # Erase canary
    ...
.L4: call __stack_chk_fail

Input: 1234567