Lecture #28 – Buffer Overflow Attacks, The Memory Hierarchy
Good news, everyone!

Changes due to (surprise) winter break next week:

• Assignment 5 will be out Dec 11
  Dec 21 (due Dec 25 Jan 4)
• Assignment 6 will be out Dec 25
  Jan 4 (due Jan 8 Jan 18)
• The lecture topics and related labs shifted by a week
  • Lab 10 canceled
Recap

- Structures and Alignment
- Floating Point
- Memory Layout
- Buffer Overflow
Recap: 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

Explanation:

```
<table>
<thead>
<tr>
<th>Critical State</th>
<th>Location accessed by fun(i)</th>
</tr>
</thead>
<tbody>
<tr>
<td>?</td>
<td>a[0]</td>
</tr>
<tr>
<td>?</td>
<td>a[1]</td>
</tr>
<tr>
<td>d7 ... d4</td>
<td>d3 ... d0</td>
</tr>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>6</td>
</tr>
</tbody>
</table>
```

Recap: 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

```c
void P(){
    Q();
    ...
    return address A
}

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

Stack after call to `gets()`

- P stack frame
- Q stack frame
- Data written by `gets()`
- Pad
- Exploit code
- B
Plan for Today

• Buffer overflow attacks and what to do about them
• Storage technologies and trends
• Locality of reference
• Caching in the memory hierarchy

Disclaimer: Slides for this lecture were borrowed from
—Randal E. Bryant and David R. O'Hallaroni’s CMU 15-213 class
Lecture Plan

• Buffer overflow attacks and what to do about them
• Storage technologies and trends
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 Assignment 5
  • 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@linuxpool.ku.edu.tr`
  • 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 😐)
    • see June 1989 article in Comm. of the ACM
  • the young author of the worm was prosecuted...
  • and CERT was formed... homed at CMU
Example 2: IM War

July 1999:

- Microsoft launches MSN Messenger (instant messaging system).
- Messenger clients can access popular AOL Instant Messaging Service (AIM) servers.
IM War (cont.)

August 1999:

• Mysteriously, Messenger clients can no longer access AIM servers

• Microsoft and AOL begin the IM war:
  • AOL changes server to disallow Messenger clients
  • Microsoft makes changes to clients to defeat AOL changes
  • At least 13 such skirmishes

• What was really happening?
  • AOL had discovered a buffer overflow bug in their own AIM clients
  • They exploited it to detect and block Microsoft: the exploit code returned a 4-byte signature (the bytes at some location in the AIM client) to server
  • When Microsoft changed code to match signature, AOL changed signature location
It was later determined that this email originated from within Microsoft!

Date: Wed, 11 Aug 1999 11:30:57 -0700 (PDT)
From: Phil Bucking <philbucking@yahoo.com>
Subject: AOL exploiting buffer overrun bug in their own software!
To: rms@pharlap.com

Mr. Smith,

I am writing you because I have discovered something that I think you might find interesting because you are an Internet security expert with experience in this area. I have also tried to contact AOL but received no response.

I am a developer who has been working on a revolutionary new instant messaging client that should be released later this year.

... It appears that the AIM client has a buffer overrun bug. By itself this might not be the end of the world, as MS surely has had its share. But AOL is now *exploiting their own buffer overrun bug* to help in its efforts to block MS Instant Messenger.

... Since you have significant credibility with the press I hope that you can use this information to help inform people that behind AOL's friendly exterior they are nefariously compromising peoples' security.

Sincerely,
Phil Bucking
Founder, Bucking Consulting
philbucking@yahoo.com
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
OK, what to do about buffer overflow attacks

• Avoid overflow vulnerabilities
• Employ system-level protections
• Have compiler use “stack canaries”
1. Avoid Overflow Vulnerabilities in Code (!)

/* 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
  • Use fgets to read the string
  • Or use %ns where n is a suitable integer
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

Local: 0x7ffe4d3be87c  0x7fff75a4f9fc  0x7ffeadb7c80c  0x7ffeaea2fdac  0x7ffcd452017c
2. System-Level Protections can help

Nonexecutable 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

Any attempt to execute this code will fail
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 the default (disabled earlier)
Demo: Stack Canaries

bufdemo.c
Protected Buffer Disassembly

echo:

40072f: sub $0x18,%rsp
400733: mov %fs:0x28,%rax
40073c: mov %rax,0x8(%rsp)
400741: xor %eax,%eax
400743: mov %rsp,%rdi
400746: callq 4006e0 <gets>
40074b: mov %rsp,%rdi
40074e: callq 400570 <puts@plt>
400753: mov 0x8(%rsp),%rax
400758: xor %fs:0x28,%rax
400761: je 400768 <echo+0x39>
400763: callq 400580 <__stack_chk_fail@plt>
400768: add $0x18,%rsp
40076c: retq
Setting Up Canary

Before call to gets

Stack Frame for call_echo

Return Address (8 bytes)

Canary (8 bytes)

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

/* Echo Line */
void echo()
{
    char buf[4]; /* 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
    ...

buf ← %rsp
Checking Canary

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

Return Address (8 bytes)

Stack Frame for call_echo

Canary (8 bytes)

Input: 0123456

eecho:
    ...  
    movq 8(%rsp), %rax  # Retrieve from stack
    xorq %fs:40, %rax  # Compare to canary
    je .L6            # If same, OK
    call __stack_chk_fail  # FAIL
    .L6:                ...  
    buf ← %rsp

After call to gets

00  36  35  34
33  32  31  30
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 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
long ab_plus_c
    (long a, long b, long c) {
    return a*b + c;
}
```

```
00000000004004d0 <ab_plus_c>:
4004d0:  48 0f af fe  imul %rsi,%rdi
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

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

\[ <\text{setval}>: \]
\[
\begin{align*}
4004d9: & \quad \text{c7 07 d4 48 89 c7} \quad \text{movl} \quad \text{\$0xc78948d4,}\%\text{rdi} \\
4004df: & \quad \text{c3} \quad \text{retq}
\end{align*}
\]

- \( \text{rdi} \leftarrow \text{rax} \)
- Gadget address = \( \text{0x4004dc} \)
- Repurpose byte codes
ROP Execution

• Trigger with `ret` instruction
  • Will start executing Gadget 1
• Final `ret` in each gadget will start next one
COMP201 Topic 7: How does the memory system is organized as a hierarchy of different storage devices with unique capacities?
Lecture Plan

• Buffer overflow attacks and what to do about them
• Storage technologies and trends
Random-Access Memory (RAM)

- Key features
  - RAM is traditionally packaged as a chip.
  - Basic storage unit is normally a **cell** (one bit per cell).
  - Multiple RAM chips form a memory.

- RAM comes in two varieties:
  - SRAM (Static RAM)
  - DRAM (Dynamic RAM)
## SRAM vs DRAM Summary

<table>
<thead>
<tr>
<th></th>
<th>Trans. per bit</th>
<th>Access time</th>
<th>Needs refresh?</th>
<th>Need EDC?</th>
<th>Cost</th>
<th>Applications</th>
</tr>
</thead>
<tbody>
<tr>
<td>SRAM</td>
<td>4 or 6</td>
<td>1X</td>
<td>No</td>
<td>Maybe</td>
<td>100X</td>
<td>Cache memories</td>
</tr>
<tr>
<td>DRAM</td>
<td>1</td>
<td>10X</td>
<td>Yes</td>
<td>Yes</td>
<td>1X</td>
<td>Main memories, frame buffers</td>
</tr>
</tbody>
</table>
Nonvolatile Memories

• DRAM and SRAM are volatile memories
  • Lose information if powered off.

• Nonvolatile memories retain value even if powered off
  • Read-only memory (ROM): programmed during production
  • Programmable ROM (PROM): can be programmed once
  • Eraseable PROM (EPROM): can be bulk erased (UV, X-Ray)
  • Electrically eraseable PROM (EEPROM): electronic erase capability
  • Flash memory: EEPROMs with partial (block-level) erase capability
    • Wears out after about 100,000 erasings

• Uses for Nonvolatile Memories
  • Firmware programs stored in a ROM (BIOS, controllers for disks, network cards, graphics accelerators, security subsystems,...)
  • Solid state disks (replace rotating disks in thumb drives, smart phones, mp3 players, tablets, laptops,...)
  • Disk caches
Traditional Bus Structure Connecting CPU and Memory

- A bus is a collection of parallel wires that carry address, data, and control signals.
- Buses are typically shared by multiple devices.
Memory Read Transaction (1)

- CPU places address $A$ on the memory bus.

**Load operation:** `movq A, %rax`

![Diagram showing the memory read transaction process](image)
Memory Read Transaction (2)

• Main memory reads \( A \) from the memory bus, retrieves word \( x \), and places it on the bus.

Load operation: \texttt{movq} \( A, \%rax \)
Memory Read Transaction (3)

- CPU read word $x$ from the bus and copies it into register %rax.

Load operation: `movq A, %rax`
Memory Write Transaction (1)

- CPU places address A on bus. Main memory reads it and waits for the corresponding data word to arrive.

Store operation: `movq %rax, A`
 Memory Write Transaction (2)

• CPU places data word y on the bus.

Store operation: movq %rax, A
Memory Write Transaction (3)

- Main memory reads data word $y$ from the bus and stores it at address $A$. 

\[
\text{Store operation: } \text{movq} \ %rax, \ A
\]
What’s Inside A Disk Drive?

- Spindle
- Arm
- Actuator
- Platters
- Electronics (including a processor and memory!)
- SCSI connector

Image courtesy of Seagate Technology
Disk Geometry

- Disks consist of platters, each with two surfaces.
- Each surface consists of concentric rings called tracks.
- Each track consists of sectors separated by gaps.
Disk Geometry (Multiple-Platter View)

- Aligned tracks form a cylinder.
Disk Capacity

• **Capacity**: maximum number of bits that can be stored.
  - Vendors express capacity in units of gigabytes (GB), where 1 GB = 109 bytes.

• Capacity is determined by these technology factors:
  - **Recording density** (bits/in): number of bits that can be squeezed into a 1-inch segment of a track.
  - **Track density** (tracks/in): number of tracks that can be squeezed into a 1-inch radial segment.
  - **Areal density** (bits/in²): product of recording and track density.
Recording zones

• Modern disks partition tracks into disjoint subsets called **recording zones**
  • Each track in a zone has the same number of sectors, determined by the circumference of innermost track.
  • Each zone has a different number of sectors/track, outer zones have more sectors/track than inner zones.
  • So we use average number of sectors/track when computing capacity.
Computing Disk Capacity

Capacity = (# bytes/sector) × (avg. # sectors/track) ×

(# tracks/surface) × (# surfaces/platter) ×

(# platters/disk)

Example:

• 512 bytes/sector
• 300 sectors/track (on average)
• 20,000 tracks/surface
• 2 surfaces/platter
• 5 platters/disk

Capacity = 512 × 300 × 20000 × 2 × 5
= 30,720,000,000
= 30.72 GB
Disk Operation (Single-Platter View)

The disk surface spins at a fixed rotational rate.

The read/write head is attached to the end of the arm and flies over the disk surface on a thin cushion of air.

By moving radially, the arm can position the read/write head over any track.
Disk Operation (Multi-Platter View)

Read/write heads move in unison from cylinder to cylinder.
Disk Structure - top view of single platter

Surface organized into tracks

Tracks divided into sectors
Disk Access

Head in position above a track
Disk Access

Rotation is counter-clockwise
Disk Access – Read

About to read blue sector
Disk Access – Read

After BLUE read

After reading blue sector
Disk Access – Read

After BLUE read

Red request scheduled next
Disk Access – Seek

After **BLUE** read  Seek for **RED**

Seek to red’s track
Disk Access – Rotational Latency

Wait for red sector to rotate around
Disk Access – Read

After BLUE read  Seek for RED  Rotational latency  After RED read

Complete read of red
Disk Access – Service Time Components

- After **BLUE** read
- Seek for **RED**
- Rotational latency
- After **RED** read

Time components:
- Data transfer
- Seek
- Rotational latency
- Data transfer
Disk Access Time

• Average time to access some target sector approximated by:
  • $T_{\text{access}} = T_{\text{avg seek}} + T_{\text{avg rotation}} + T_{\text{avg transfer}}$

• Seek time (Tavg seek)
  • Time to position heads over cylinder containing target sector.
  • Typical $T_{\text{avg seek}}$ is 3—9 ms

• Rotational latency (Tavg rotation)
  • Time waiting for first bit of target sector to pass under r/w head.
  • $T_{\text{avg rotation}} = \frac{1}{2} \times \frac{1}{\text{RPMs}} \times 60 \text{ sec/1 min}$
  • Typical $T_{\text{avg rotation}} = 7200 \text{ RPMs}$

• Transfer time (Tavg transfer)
  • Time to read the bits in the target sector.
  • $T_{\text{avg transfer}} = \frac{1}{\text{RPM}} \times \frac{1}{(\text{avg # sectors/track})} \times 60 \text{ secs/1 min.}$
Disk Access Time Example

• Given:
  • Rotational rate = 7,200 RPM
  • Average seek time = 9 ms.
  • Avg # sectors/track = 400.

• Derived:
  • Tavg rotation = \( \frac{1}{2} \times \left( \frac{60 \text{ secs}}{7200 \text{ RPM}} \right) \times 1000 \text{ ms/sec} = 4 \text{ ms.} \)
  • Tavg transfer = \( \frac{60}{7200} \text{ RPM} \times \frac{1}{400} \text{ secs/track} \times 1000 \text{ ms/sec} = 0.02 \text{ ms} \)
  • Taccess = 9 ms + 4 ms + 0.02 ms

• Important points:
  • Access time dominated by seek time and rotational latency.
  • First bit in a sector is the most expensive, the rest are free.
  • SRAM access time is about 4 ns/doubleword, DRAM about 60 ns
    • Disk is about 40,000 times slower than SRAM,
    • 2,500 times slower then DRAM.
Logical Disk Blocks

• Modern disks present a simpler abstract view of the complex sector geometry:
  • The set of available sectors is modeled as a sequence of b-sized logical blocks (0, 1, 2, …)

• Mapping between logical blocks and actual (physical) sectors
  • Maintained by hardware/firmware device called disk controller.
  • Converts requests for logical blocks into (surface, track, sector) triples.

• Allows controller to set aside spare cylinders for each zone.
  • Accounts for the difference in “formatted capacity” and “maximum capacity”.
I/O Bus

Expansion slots for other devices such as network adapters.
Reading a Disk Sector (1)

CPU initiates a disk read by writing a command, logical block number, and destination memory address to a port (address) associated with disk controller.
Reading a Disk Sector (2)

Disk controller reads the sector and performs a direct memory access (DMA) transfer into main memory.
When the DMA transfer completes, the disk controller notifies the CPU with an interrupt (i.e., asserts a special “interrupt” pin on the CPU)
Solid State Disks (SSDs)

- Pages: 512KB to 4KB, Blocks: 32 to 128 pages
- Data read/written in units of pages.
- Page can be written only after its block has been erased
- A block wears out after about 100,000 repeated writes.
SSD Performance Characteristics

• Sequential access faster than random access
  • Common theme in the memory hierarchy
• Random writes are somewhat slower
  • Erasing a block takes a long time (~1 ms)
  • Modifying a block page requires all other pages to be copied to new block
• In earlier SSDs, the read/write gap was much larger.

<table>
<thead>
<tr>
<th></th>
<th>Sequential read</th>
<th>Sequential write</th>
<th>Random read</th>
<th>Random write</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequential read throughput</td>
<td>550 MB/s</td>
<td>470 MB/s</td>
<td>365 MB/s</td>
<td>303 MB/s</td>
</tr>
<tr>
<td>Random read throughput</td>
<td>303 MB/s</td>
<td>470 MB/s</td>
<td>365 MB/s</td>
<td>303 MB/s</td>
</tr>
<tr>
<td>Avg seq read time</td>
<td>50 us</td>
<td>60 us</td>
<td>Avg seq write</td>
<td>60 us</td>
</tr>
<tr>
<td>Avg seq write time</td>
<td></td>
<td></td>
<td>time</td>
<td></td>
</tr>
</tbody>
</table>

Source: Intel SSD 730 product specification.
SSD Tradeoffs vs Rotating Disks

• **Advantages**
  • No moving parts → faster, less power, more rugged

• **Disadvantages**
  • Have the potential to wear out
    • Mitigated by “wear leveling logic” in flash translation layer
    • E.g. Intel SSD 730 guarantees 128 petabyte (128 x 1015 bytes) of writes before they wear out
  • In 2015, about 30 times more expensive per byte

• **Applications**
  • MP3 players, smart phones, laptops
  • Beginning to appear in desktops and servers
The CPU-Memory Gap

• The gap widens between DRAM, disk, and CPU speeds.
Locality to the Rescue!

• The key to bridging this CPU-Memory gap is a fundamental property of computer programs known as locality
Recap

• Buffer overflow attacks and what to do about them
• Storage technologies and trends

Next: Locality of reference, caching in the memory hierarchy