CMPSC 447
Midterm Review

Trent Jaeger
Systems and Internet Infrastructure Security (SIIS) Lab
Computer Science and Engineering Department
Pennsylvania State University
Quiz 3

• #1 - The MITRE ATT&CK framework describes the tactic of "execution" as a tactic to enable adversaries to run adversary-controlled code on a system.
  ➔ True/False
ATT&CK Tactics

- Initial Access
- Execution
- Persistence
- Privilege Escalation
- Defense Evasion
- Credential Access
- Discovery
- Lateral Movement
- Collection
- Command and Control
- Exfiltration
- Impact
- Reconnaissance
- Resource Development
ATT&CK Tactics in Action

- **Initial Access, Discovery, and Credential access**
  - Gain and learn about (via secrets) an environment
  - What was that for Stuxnet?

- **Execution**
  - “Execution of adversary-controlled code”
  - How Stuxnet?

- **Collection and Exfiltration**
  - Steal data from the domain
  - Did Stuxnet do that?
ATT&CK Tactics in Action

• Persistence and Defense Evasion
  ‣ “to persist in the target environment” “undetected”
  ‣ How did Stuxnet do that?

• Privilege Escalation and Lateral Movement
  ‣ Gain more permissions in the environment and control more components of same privilege
  ‣ How for Stuxnet?

• Command and Control
  ‣ Method to obtain commands for malware
  ‣ Did Stuxnet do that?
Quiz 3

• #2 - A type error can violate memory safety by allowing an adversary to cause the program to treat data values as pointer values.
  ▸ True/False
Memory Safety

• What are the requirements for memory safety for all three categories

  ‣ **Spatial safety**: All reads and writes using a pointer to a memory region must be within that memory region

  • Strings additionally require a null-terminator

  ‣ **Temporal safety**: All reads and writes using a pointer must be to a live (not deallocated) memory region that is assigned to the pointer

  ‣ **Type memory safety**: Semantics of all field references at the same offset must be of the same type (weaker: cannot be both data and pointer)

• **Type safety**: Only pointers of one type for the memory region
Type Errors

- **Type errors** are possible when pointers of multiple types are used to access the same region
  - tl *p, t2 *q; // declare pointers
  - p = (tl *) malloc(sizeof(tl)); // allocate object and define p
  - p->field = value; // use pointer for tl
  - q = (t2 *)p; // type cast and define q
  - q->X(); // use pointer for t2

- Semantics of "p->field" may be different than "q->X"
  - Pointer vs. data
  - Data of multiple types (formats)
Type Errors

• **Downcasts** – Cast to a larger type; causes overflow
  
  ‣ `t1 *p, t2 *q;`  // declare pointers
  ‣ `p = (t1 *) malloc(sizeof (t1));`  // allocate t1 object, define p
  ‣ `p->field = value;`  // suppose this is an int field
  ‣ `q = (t2 *)p;`  // downcast, t2 is a larger type
  ‣ `q->extra= value2;`  // overflow memory of object

• E.g., t2 is a child type of t1
  
  ‣ So, the size of type t2 is greater than the size of type t1
  ‣ “extra” field is added to the type t1 to create type t2
Quiz 3

• A temporal error caused by a use-after-free vulnerability can be mitigated by which methods (may be multiple correct answers).

  ‣ Freeing the pointer memory along with the memory region
  ‣ Nullifying the pointer value when the assigned memory region is freed
  ‣ Deallocating memory on function returns
  ‣ Never freeing memory
  ‣ Only allocating memory regions in type-specific memory pools
• What does ”p” reference upon use?
  ‣ char *p; // declare pointer
  ‣ len = snprintf(p, size, "%s", original_value); // use pointer
  ‣ p = (char *) malloc(size); // define pointer to object
  ‣ free(p); // deallocate object

• Called “use before initialization” (UBI)
  ‣ Allows an adversary to use reference value defined at the location used to declare “p” (not an assignment)
  ‣ Could be anywhere
Use After Free

• What does ”p” reference upon use?
  - char *p; // declare pointer
  - p = (char *) malloc(size); // define pointer to object
  - free(p); // deallocate object – release memory for reuse
  - len = sprintf(p, size, "%s", original_value); // use pointer

• Called “use after free” (UAF)
  - Allows an adversary to use reference to memory region that may be allocated a different object
  - Could be anywhere
Zeroing Pointers

- Yes! Set every pointer value to zero on deallocation
  - Zero pointers on deallocation from the heap
    - `free(p), p = 0;`
  - Trickier on the stack
    - In theory, no stack reference should outlive its assignment
    - But, hard to guarantee since deallocation is implicit

- Also, the cost of zeroing on deallocation can be worse
  - Since not done at all normally
Temporal Defense Alternatives

- **Hypothesis**: memory is so cheap and abundant, we just do not need to deallocate
  - Will be some cases where this is not going to work
  - But, for others, why risk attack?
- **Hypothesis**: garbage collection
  - Too expensive for C
- **Hypothesis**: temporal safety like Rust’s “safe” objects
  - Harder to program with lifetimes and ownerships
- **Hypothesis**: use type-specific allocation
  - All objects and fields are aligned
Quiz 3

• What are the differences between strncpy and snprintf with respect to safe string processing?
  ‣ Only strncpy ensures a null terminator is added to the end of the string
  ‣ Only snprintf ensures a null terminator is added to the end of the string
  ‣ Only snprintf returns an integer for the amount of data that would have been written to detect truncation
  ‣ Only snprintf/strncpy does bounds checking
  ‣ Only strncpy returns a pointer to the resultant buffer memory region to detect truncation
Traditional Solution – That Works!

- `int snprintf(char *S, size_t N, const char *FORMAT, ...);`
  - Writes output to buffer S up to N chars (**bounds check**)
  - Always writes '\0' at end if N>=1 (**terminate**)
  - Returns “length that would have been written” or negative if error (**reports truncation or error**)

- Thus, achieves goals of correct bounds checking
  - Enforces bounds, ensures correct C string, and reports truncation or error
    - `len = snprintf(buf, buflen, "%s", original_value);`
    - if (len < 0 || len >= buflen) … // handle error/truncation
Bounds Checking

- For each byte in the operation:
  - If oversized option (1) – stop processing input
    - Reject and try again, or even halt program (may make DoS)
  - If oversized option (2) – truncate data
    - Common approach, but has issues:
      - Terminates text “in the middle” at place of attacker’s choosing
      - Way better to truncate than to allow easy buffer overflow attack
      - But, should report when truncation occurs for the programmer to handle the possible impacts
Quiz 3

• What kind of memory error flaw does the following code demonstrate?

```c
int a;
unsigned int b;
a = adv_input;
if (a < MAX_VALUE) {   // MAX_VALUE is a constant
    b = (unsigned int)a;
    read(fd, buf, b);      // assume fd and buf are initialized
}
```

- **Integer overflow** / Downcast error / Special error / Use-after-initialization / Recast error
Integer Overflows

- Key question
  - What is an integer?
  - In a computer system?

- There are several different computer representations for integers
  - **Size** – number of bytes used to represent
  - **Signedness** – range of values integers can take
Quiz 3

• `safe_strcpy(dest, src)` is a secure string copy function. What properties should that function ensure and how could you implement that function to ensure those properties given the limitations in the arguments available?

  ‣ **Idea**: Automatic memory resizing
Automatic Resizing

- For each byte in the operation:
- If oversized – **Auto-resize** – move string to a new memory region, if necessary
  - This is what most languages do automatically
    - other than C
    - Must deal with “too large” data
- By default, handling auto-resize manually in C can create issues
  - More `code changes/complexity` in existing C code
    - But, available APIs support options to handle this for you
  - **Dynamic allocation** is manual in C, so adds new risks
    - Temporal errors
Quiz 3

- Produce the stack layout to use the following return-oriented programming (ROP) gadgets to move a value at 0xffcd to 0x0804.

G1: push %ebx; ret
G2: push %ecx; ret
G3: pop %ebx; ret
G4: pop %ecx; ret
G5: mov %ecx, (%ebx); ret // store value in %ecx to memory location (%ebx)
G6: mov %ebx, (%ecx); ret // store value in %ebx to memory location (%ecx)
G7: mov (%ecx), %ebx; ret // load value in %ebx from memory location (%ecx)
G8: mov (%ebx), %ecx; ret // load value in %ecx to memory location (%ebx)

- G4 | 0xffcd | G7 | G4 | 0x0804 | G6
RDP Example

- Use ESP as program counter
  - E.g., Store 5 at address 0x8048000 (without introducing new code)

<table>
<thead>
<tr>
<th>Code</th>
<th>Stack</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>G1: pop %eax ret</td>
<td>G1</td>
<td>0x8048000 =</td>
</tr>
<tr>
<td>G2: pop %ebx ret</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>G3: movl %eax, (%ebx)</td>
<td>G2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0x8048000</td>
<td></td>
</tr>
<tr>
<td></td>
<td>G3</td>
<td></td>
</tr>
<tr>
<td></td>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

Registers
- %eax =
- %ebx =
Code Reuse in General

- Code reuse attacks can be employed more generally to enable adversaries to execute existing code under their control
- Termed "return-oriented attacks" (ROP)

**Example**
- Use ESP as program counter
- E.g., Store 5 at address 0x8048000 (without introducing new code)

```
G1:  pop %eax
     ret

G2:  pop %ebx
     ret

G3:  movl %eax, (%ebx)
     ret
```

<table>
<thead>
<tr>
<th>Code</th>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>G1</td>
<td>G1</td>
</tr>
<tr>
<td></td>
<td>5</td>
</tr>
<tr>
<td>G2</td>
<td>G2</td>
</tr>
<tr>
<td></td>
<td>0x8048000</td>
</tr>
<tr>
<td>G3</td>
<td>G3</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

**Registers**
- %eax = 5
- %ebx =

**Memory**
- 0x8048000 =
Code Reuse in General

- Code reuse attacks can be employed more generally to enable adversaries to execute existing code under their control
  - Termed "return-oriented attacks"

**ROP Example**

- Use ESP as program counter
  - E.g., Store 5 at address 0x8048000 (without introducing new code)

```
G1:  pop %eax
    ret

G2:  pop %ebx
    ret

G3:  movl %eax, (%ebx)
    ret
```

```
%eax = 5
%ebx =
```

```
Code

G1:  pop %eax
    ret

G2:  pop %ebx
    ret

G3:  movl %eax, (%ebx)
    ret

Stack

G1
5
G2
0x8048000
G3
...

Return Address

buf

Memory

0x8048000 =
```
Code Reuse in General

- Code reuse attacks can be employed more generally to enable adversaries to execute existing code under their control
  - Termed "return-oriented attacks"

**Example**

- Use ESP as program counter:
  - E.g., Store 5 at address 0x8048000 (without introducing new code)

```
G1: pop %eax
    ret
G2: pop %ebx
    ret
G3: movl %eax, (%ebx)
    ret
```

### Registers

- \%eax = 5
- \%ebx = 0x8048000

### Memory

- 0x8048000 = buf

### Stack

- Return Address
- G1
- 5
- G2
- 0x8048000
- G3
- ...

### Code

- \(5\)
- \(0x8048000\)
Code Reuse in General

- Code reuse attacks can be employed more generally to enable adversaries to execute existing code under their control
  - Termed "return-oriented attacks"

**ROP Example**

- Use ESP as program counter
  - E.g., Store 5 at address 0x8048000 (without introducing new code)

```
%eax = 5
%ebx = 0x8048000
```
Code Reuse in General

- Code reuse attacks can be employed more generally to enable adversaries to execute existing code under their control
  - Termed “return-oriented attacks” (ROP)

**Example**

- Use ESP as program counter
  - E.g., Store 5 at address 0x8048000 (without introducing new code)

```asm
%eax = 5
%ebx = 0x8048000
```

![Diagram](image)
Code Reuse in General

- Code reuse attacks can be employed more generally to enable adversaries to execute existing code under their control
  - Termined “return-oriented attacks” (ROP)

**Example**

- Use ESP as program counter
  - E.g., Store 5 at address 0x8048000 (without introducing new code)

```asm
G1:  pop %eax  
    ret

G2:  pop %ebx  
    ret

G3:  movl %eax, (%ebx)  
    ret
```

```
%eax = 5  
%ebx = 0x8048000
```

```
Stack

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>G1</td>
<td>5</td>
</tr>
<tr>
<td>G2</td>
<td></td>
</tr>
<tr>
<td>G3</td>
<td>0x8048000</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

Registers
```

```
Memory

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x8048000</td>
<td>5</td>
</tr>
</tbody>
</table>
```

Return Address

buf

Finding Gadgets

- Snippets of code ending in “ret” are called gadgets.
- How do we build a complete exploit from available code?
  - Must find the gadgets that are available in that code.
- How do you think one finds all the gadgets in a code region?
  - From each byte offset in the code region, see what sequence of instructions are encoded until a ”ret” is reached.
  - Find “a, b, c, ret” – where a, b, and c are other instructions.
Previous Quizzes (#2)

- How many filler bytes are necessary to reach the field (*fn) in the following structure if there is a buffer overflow for writing to the field "buffer" (assume 32-bit binary and 4-byte ints)?

```c
struct X {
    int index;
    char buffer[12];
    char other[8];
    int answer;
    int (*fn)(int y);
};
```

- **24 bytes**
Hijack Control Flow

- Let’s create a payload to hijack control by overwriting the return address
  - To print a string from the binary

- To create the payload
  - Insert filler to reach the return address
  - Add the new return address (printf@plt) at 0x10a0
    - **Note**: changed the from the prior figure where printf@plt at 0x1080
  - And the reference to a string at 0x342
    - “__libc_start_main”
Hijack Control Flow

- Create the payload
  - Actually, code is loaded at an offset

- So, need to account for the offset in the payload
  - Add the new return address (printf@plt) at offset 0x1080 → 0x56555000 + 0x10a0 = 0x565560a0
    - Little endian \xa0\x60\x55\x56
  - And the reference to the format string at offset 0x342 → 0x56555000 + 0x342 = 0x56555342
    - Little endian \x42\x53\x55\x56 or “BSUV” in ascii
Previous Quizzes (#1)

• Specify a payload to a buffer overflow vulnerability for writing a buffer of size 10 that overwrites the return address that is eight bytes above the buffer with the address 0x080432f0.

  ▶ Payload

  • Fill buffer (10 bytes)

  • Fill rest of space to the return address (8 bytes)

  • Set the return address to 0x800432f0
In a 32-bit program, suppose the heap metadata structure only contains the fields "bk" (for referencing the previous block) and "fd" (for referencing the next block) in that order.

And the metadata is updated using the follow code ("chunk2" is an instance of the heap metadata struct):

- chunk2→bk→fd = chunk2→fd;

If you want to write "0xffff" at address "0x4d78," you need to write the chunk2→fd to be 0xffff and chunk2→bk to be 0x4d74.
• The Heap Memory Layout often includes metadata
  ‣ Depends on the heap allocator
  ‣ Often placed between objects to store information needed to manage allocation state – e.g., sizes and status
Heap Overflows

- Heap allocators maintain a doubly-linked list of allocated and free chunks
- `malloc()` and `free()` modify this list
• Suppose user2 has a symbolic link 'linkfile' in '/home/user2' to '/'. If a program running as root opens the file '/home/user2/linkfile/etc/foo.txt', which pathname elements does the program have to check for confused deputy attacks to detect/prevent attacks?

› /home/user2/linkfile
› /home/user2

• Pathname elements modifiable by someone other than root
Common Threat (1)

- What is the threat that enables link traversal and file squatting attacks?
  - Common to both

- In both cases, the adversary has write permission to a directory that a victim uses in name resolution
  - Could be any directory used in resolution, not just the last one
  - Enables the adversary to plant links and/or files
• What is the threat that enables directory traversal attacks?

• In this case, the victim uses adversary input to construct file names
  ‣ Any parts of file names
Integrity (and Secrecy) Threat

- Confused Deputy
  - Process is tricked into performing an operation on an adversary’s behalf that the adversary could not perform on their own
  - Write to (read from) a privileged file
Previous Quizzes (#1)

- Which code is guaranteed to produce a C string in the buffer defined by 'char buffer[20];'?
  - None of the answers supplied are correct
  - How would you do that now?
    - E.g., `strcpy(buffer, src, 20);`
    - Check others
Traditional Solution – That Works!

- int `snprintf`(char *S, size_t N, const char *FORMAT, ...);
  - Writes output to buffer S up to N chars (bounds check)
  - Always writes ‘\0’ at end if N>=1 (terminate)
  - Returns “length that would have been written” or negative if error (reports truncation or error)
- Thus, achieves goals of correct bounds checking
  - Enforces bounds, ensures correct C string, and reports truncation or error
    - len = snprintf(buf, buflen, "%s", original_value);
    - if (len < 0 || len >= buflen) … // handle error/truncation
Previous Quizzes (#2)

• What properties do we expect from all secure string copy operations? Select one or more correct answers.
  ‣ Null-terminated
  ‣ Within memory bounds
  ‣ Truncation reported
Previous Quizzes (#1)

- Why is it possible to execute code injected on the stack? Choose the best answer.
  - Because the page permissions of the stack memory region (all pages) include execute permission
Injecting Shell Code

- How do you invoke “execve” using injected code?

Figure 6.5: Buffer overflow of stack-based local variable.
Code Reuse in General

- Code reuse attacks can be employed more generally to enable adversaries to execute existing code under their control.
  - Termed "return-oriented attacks".

Return-Oriented Programming

Bad code versus bad behavior

- "Bad" behavior
- "Good" behavior
- Attacker code
- Application code

Problem: this implication is false!
Previous Quizzes (#2)

- What is one way (procedure) that the Stuxnet worm achieved tactic of "lateral movement"?
  - Infected any USB device inserted

- Compare Stuxnet behaviors to MITRE ATT&CK tactics
Stuxnet: Tactics

- Stuxnet tactics
  - Zero-day exploits (initial access)
  - Windows rootkit (persistence)
  - PLC rootkit (execution)
  - Antivirus evasion (defense evasion)
  - Peer-to-Peer updates (command and control)
  - Signed driver with a valid certificate (credentials)
- And more
  - Go through Stuxnet and map actions to tactics
Take Away

- Reviewed for midterm from the quiz questions and their answers
- Scope of exam includes these questions
  - And a little more
    - More about type and temporal attacks
    - Including more context about what we discussed, so go back to the related slide decks in the original
- Think about variants of these questions to give yourself a broader understanding
- Good luck!