CMPSC 497: Midterm Review

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

• Format
  ‣ True/False
    • 8 questions – 16 pts
  ‣ Short answer – word/phrase to sentence or two
    • 8 questions – 36 pts
  ‣ Question – conceptual (why?) and constructions (how?)
    • 6 questions – 48 pts
    • Conceptual – can be a bit open-ended
    • Constructions – like questions 11-14 from homework, but fewer parts
  ‣ Exams can be long-ish – 3+ minutes per question
Homework – Question #1

• Know the definitions of
  ‣ vulnerability, implicit information flow, memory error, no-op sled, canary, buffer overflow, buffer overread, use-after-free vulnerability, name resolution attack, confused deputy, soundness in static analysis, path constraints in symbolic execution, fuzz testing
Question #2

• What is a format string vulnerability? What format specifier enables writing to memory? What enables reading from memory?
Format String Vulnerabilities

• Who uses `printf` in their programs?

```c
printf ("This class is %s\n", string);
```

  ‣ In some cases, `printf` can be exploited

• `Printf` takes a format string and an arbitrary number of subsequent arguments

  ‣ Format string determines what to print
    • Including a set of format parameters
  
  ‣ Arguments supply input for format parameters
    • Which may be values (e.g., %d) or references (e.g., %s)

• An argument for each format parameter
Format String Vulnerabilities

• Who uses `printf` in their programs?
  ‣ In some cases, `printf` can be exploited

• As usual, arguments are retrieved from the stack
  ‣ What happens when the following is done?

    ```c
    printf("%s%s%s%s");
    ```

• Traditionally, compilers do not check for a match between arguments and format string – do now…
  ‣ So, `printf` would print “strings” using next four values on stack as string addresses – whatever they are
Format String Vulnerabilities

• Who uses `printf` in their programs?
  ‣ In some cases, `printf` can be exploited

• As usual, arguments are retrieved from the stack
  ‣ What happens when the following is done?

    ```c
    printf(arg);
    ```

• An interesting format parameter type – `%n`
  ‣ “%n” in a format string tells the `printf` to write the number of bytes written via the format string processing up to that point to an address specified by the argument
Printf and the Stack

• Suppose format string generates an adversary-controlled number of bytes
• Suppose adversary controls Arg1-Arg3 on stack
• Adversary can control number of bytes generated by format string with Arg1 and Arg2
• Adversary can direct where to write that number (of bytes) using %n with address at Arg3
Question #3

• What is the difference between a code-reuse attack and a code-injection attack?
Injecting Code

void function (int a, int b) {
    char buffer[12];
    gets(buffer);
    return;
}

void main() {
    int x;
    x = 0;
    function(1,2);
    x = 1;
    printf("%d\n",x);
}

The injected code can do anything. E.g., download and install a worm.
Code Injection

• Attacker creates a malicious argument—a specially crafted string that contains malicious code and a pointer to that code

• When the function returns, control is transferred to the malicious code
  ‣ Injected code runs with the permission of the vulnerable program when the function returns.
  ‣ Programs running as root or other elevated privileges are normally targeted
    • Programs with the setuid bit on
Injecting Shell Code

- This brings up a shell
- Adversary can execute any command in the shell
- The shell has the same privilege as the process
- Usually a process with the root privilege is attacked

Stack frame for main:

```
call execve ("/bin/sh")
ret 1
2
```
Question #4

- Why is a disclosure vulnerability harmful for canary defenses?
• Packet overflows overwrite the authenticated value
• **Big limitation**: Disclosure attacks
  ‣ By performing a buffer “overread”
  ‣ Example is the famous Heartbleed attack against SSL
  ‣ Why is this a problem for Stackguard?

```c
char packet[10];
...
// suppose len is adversary controlled
strncpy(buf, packet, len);
send(fd, buf, len);
```

<table>
<thead>
<tr>
<th>packet</th>
</tr>
</thead>
<tbody>
<tr>
<td>old ebp</td>
</tr>
<tr>
<td>canary</td>
</tr>
<tr>
<td>ret addr</td>
</tr>
<tr>
<td>arg</td>
</tr>
<tr>
<td>previous</td>
</tr>
<tr>
<td>stack frame</td>
</tr>
</tbody>
</table>
Question #5

• What is address space layout randomization? How does it prevent code reuse attacks?
What is address space layout randomization? How does it prevent code reuse attacks?

Move the base address of a memory segment based on a secret, random value

Normally, 0x8000000 – move to 0x8ab0000
  - Why not 0x800ab00?

Impacts code reuse – data locations to place new stack (writable) and locations of code are moved

Limits – not all code is moved, disclosure...
Question #6

• Why does W xor X defense prevent code injection attacks?
Question #6

• Why does W xor X defense prevent code injection attacks?
  ‣ Cannot write code into data section and execute
    • Cannot run “Call execve” by putting that instruction in writable memory
  ‣ How to circumvent?
Question #6

• Why does W xor X defense prevent code injection attacks?
  ‣ Cannot write code into data section and execute
    • Cannot run “Call execve” by putting that instruction in writable memory
  ‣ How to circumvent?
    • Build stack for ROP attack
    • Pointer to execve gadget
  ‣ Even with ASLR on?
Question #6

• Why does $W \text{ xor } X$ defense prevent code injection attacks?
  ‣ Cannot write code into data section and execute
    • Cannot run “Call execve” by putting that instruction in writable memory
  ‣ How to circumvent?
    • Build stack for ROP attack
    • Pointer to execve gadget
  ‣ Even with ASLR on
    • Use execve PLT entry in executable
Question #7

- Describe the steps necessary for you to exploit a buffer overflow to disable W xor X defenses? Assume you can overwrite the return address.
Question #7

• Describe the steps necessary for you to exploit a buffer overflow to disable W xor X defenses? Assume you can overwrite the return address.
  ‣ Overwrite return address with gadgets to create desired ROP stack
  ‣ Desired ROP stack can disable W xor X using mprotect
  ‣ If there is a PLT entry for mprotect, invoke that with the arguments necessary to change perms so data can be writable and executable
  ‣ Arguments
Question #8

• What are the requirements for copying a string safely?
Many C functions don’t check bounds (examples)

- `gets(3)` – reads input without checking. Don’t use it!
- `strcpy(3)` – `strcpy(dest, src)` copies from `src` to `dest`
  - If `src` longer than `dest` buffer, keeps writing!
- `strcat(3)` – `strcat(dest, src)` appends `src` to `dest`
  - If `src + data in dest` longer than `dest` buffer, keeps writing!
- `scanf()` family of input functions – many dangerous options
  - `scanf(3), fscanf(3), sscanf(3), vscanf(3), vsscanf(3), vfscanf(3)`
  - Many options don’t control max length (e.g., bare “%s”)
- Many other dangerous functions, e.g.:
  - `realpath(3), getopt(3), getpass(3)`
  - `streadd(3), strecpy(3), and strtrns(3)`
- It’s not just functions; ordinary loops can overflow
Two code solution alternatives: Bounds-checking & auto-resize

- Bounds-checking to stop overwrite; then if oversized:
  - Stop processing input
    - Reject and try again, or even halt program (turns into DoS)
  - Truncate data. Common approach, but has issues:
    - Terminates text “in the middle” at place of attacker’s choosing
    - Can strip off critical data, escapes, etc. at the end
    - Can break in the middle of multi-byte character
      - UTF-8 character can take many bytes
      - UTF-16 usually 2 bytes/character, but not if it’s outside BMP
    - Some routines truncate & return indicator so you can stop processing input
    - Way better to truncate than to allow easy buffer overflow attack

- Auto-resize – move string if necessary
  - This is what most languages do automatically (other than C)
  - Must deal with “too large” data
  - C: Requires more code changes/complexity in existing code
  - C/C++: Dynamic allocation manual, so new risks (double-free)
Question #9

- What is soundness in static analysis?
Correctness

- **Soundness:**
  - Predicted results must apply to every system execution
    - Overapproximate the effect of every program statement
    - Absolutely mandatory for trustworthiness of analysis results!

- **Completeness:**
  - Behavior of every system execution caught by analysis
  - Prove any true statement in program is really true
    - Usually not guaranteed due to approximation
  - Degree of completeness determines quality of analysis

- **Correctness**: Soundness $\wedge$ Completeness (rare)
Soundness

• **Soundness:**
  ‣ All executions are represented
  ‣ **Implication 1:** no false negatives, as static analysis model represents all executions possible

• However, unlikely that model is a correct representation of the program semantics
  ‣ **Implication 2:** Sound model is not complete
  ‣ **Implication 3:** A sound static analysis will produce some false positives
  ‣ The number of false positives determines the quality of the analysis
Question #10

• What should you do after every malloc? Every free? Every function call?
  ‣ Malloc
  ‣ Free
  ‣ Function call
Question #11

- Low integrity variables based on sockfd

```c
2: int connect_limit = MAXCONN; int *size, *type;
3: char buf[MAXLEN];
4: size = &buf[8]; type = &buf[12];
5: ...

/* code in that function */
6: while(connect_limit--) {
7:   readData(sockfd, buf);
8:   if(*type == NONE ) break;
9:   if(*type == STREAM)
10:      *size = *(srv->cur_max);
11:   else {
12:      srv->typ = *type;
13:      srv->total += *size;
14:   }
15:   err = processData(buf, &result);
16:   if (!err) sendData(sockfd, result);
```

(a) If sockfd is low integrity identify the variables that are low integrity based on the information flow (i.e., assume no memory errors)?

(b) How would you use fuzz testing to detect errors in this program caused by reading data from sockfd?

(c) Develop a static analysis sufficient to identify a buffer overflow vulnerability in this code?

(d) If variables are placed on the stack in the order they are declared in the code what variables could be overwritten by the buffer overflow?

(e) Describe one exploit that an adversary could perform given that buffer overflow?

(f) Harden the code to prevent the buffer overflow.
Question #11

- Low integrity variables based on sockfd
  - Explicit flows
    - buf from readData
    - result from processData
    - err from processData
    - Also, type and size since point to data in buf
  - Implicit flows
    - Conditionals based on type are low integrity
    - *size and srv->typ and srv->total

```c
int connect_limit = MAXCONN; int *size, *type;
char buf[MAXLEN];
size = &buf[8]; type = &buf[12];
... /* code in that function */
while(connect_limit--) {
    readData(sockfd, buf);
    if(*type == NONE ) break;
    if(*type == STREAM)
        *size = *(srv->cur_max);
    else {
        srv->typ = *type;
        srv->total += *size;
    }
    err = processData(buf, &result);
    if (!err) sendData(sockfd, result);
}
```
Question #11

- Use fuzz testing to detect errors in this code?
- Which variable to fuzz?

```c
2:  int connect_limit = MAXCONN; int *size, *type;
3:  char buf[MAXLEN];
4:  size = &buf[8]; type = &buf[12];
5:  ...

/* code in that function */
6:  while(connect_limit--) {
7:    readData(sockfd, buf);
8:    if(*type == NONE ) break;
9:    if(*type == STREAM)
10:       *size = *(srv->cur_max);
11:    else {
12:       srv->typ = *type;
13:       srv->total += *size;
14:    }
15:    err = processData(buf, &result);
16:    if (!err) sendData(sockfd, result);
```
Question #11

- Use fuzz testing to detect errors in this code?
- Which variable to fuzz?
- Data supplied to sockfd

```c
2: int connect_limit = MAXCONN; int *size, *type;
3: char buf[MAXLEN];
4: size = &buf[8]; type = &buf[12];
5: ...

/* code in that function */
6: while(connect_limit--) { 
7:   readData(sockfd, buf);
8:   if(*type == NONE ) break;
9:   if(*type == STREAM) 
10:      *size = *(srv->cur_max); 
11:   else {
12:     srv->typ = *type; 
13:     srv->total += *size;
14:   }
15:   err = processData(buf, &result);
16:   if (!err) sendData(sockfd, result);
```
Question #11

• Static analysis to identify buffer overflow?

```c
2: int connect_limit = MAXCONN; int *size, *type;
3: char buf[MAXLEN];
4: size = &buf[8]; type = &buf[12];
5: ...

/* code in that function */
6: while(connect_limit--) {
7:   readData(sockfd, buf);
8:   if(*type == NONE ) break;
9:   if(*type == STREAM)
10:      *size = *(srv->cur_max);
11:   else {
12:      srv->typ = *type;
13:      srv->total += *size;
14:   }
15:   err = processData(buf, &result);
16:   if (!err) sendData(sockfd, result);
```
Question #11

- Static analysis to identify buffer overflow?
- Type-based
  - `readData` is not bounded, but `buf` is bounded
  - Abstract interpretation on size of buffer produced by `readData`
Question #11

- Variable overflowed?
  - All
- Exploit?
  - Stack attacks
- Harden?
  - See string copy reqs

```c
2: int connect_limit = MAXCONN; int *size, *type;
3: char buf[MAXLEN];
4: size = &buf[8]; type = &buf[12];
5: ...

/* code in that function */
6: while(connect_limit--) {
7:   readData(sockfd, buf);
8:   if(*type == NONE ) break;
9:   if(*type == STREAM)
10:     *size = *(srv->cur_max);
11:   else {
12:     srv->typ = *type;
13:     srv->total += *size;
14:   }
15: err = processData(buf, &result);
16: if (!err) sendData(sockfd, result);
```
Question #12

Gadgets

G1: push %eax; ret
G2: pop %eax; ret
G3: push %ebx; ret
G4: pop %ebx; ret
G5: mov %eax, (%ebx); ret // store value in %eax to memory location (%ebx)
G6: pop %esp; ret
G7: jmp %ebx; ret
G8: add (%ebx), %eax; ret // add value at memory location (%ebx) to %eax
G9: ret
G10: pop %esp; ret

• (a) Change the value of a function pointer at address 0xfabcd to reference code at 0x80488
  ▶ Build the Stack:
Question #12

Gadgets

G1: push %eax; ret
G2: pop %eax; ret
G3: push %ebx; ret
G4: pop %ebx; ret
G5: mov %eax, (%ebx); ret // store value in %eax to memory location (%ebx)
G6: pop %esp; ret
G7: jmp %ebx; ret
G8: add (%ebx), %eax; ret // add value at memory location (%ebx) to %eax
G9: ret
G10: pop %esp; ret

• (a) Change the value of a function pointer at address 0xfabcd to reference code at 0x80488

  ▶ Stack:

  • 0x80488 into register – pop eax; 0x80488
  • 0xfabcd into register – pop ebx; 0xfabcd
  • Move 0x80488 to 0xfabcd – mov eax, (ebx)
Question #12

Gadgets

G1: push %eax; ret
G2: pop %eax; ret
G3: push %ebx; ret
G4: pop %ebx; ret
G5: mov %eax, (%ebx); ret // store value in %eax to memory location (%ebx)
G6: pop %esp; ret
G7: jmp %ebx; ret
G8: add (%ebx), %eax; ret // add value at memory location (%ebx) to %eax
G9: ret
G10: pop %esp; ret

• (b) Add 0x10 to the value at address 0xf0020
  ‣ Build the Stack:
Question #12

Gadgets

G1: push %eax; ret
G2: pop %eax; ret
G3: push %ebx; ret
G4: pop %ebx; ret
G5: mov %eax, (%ebx); ret // store value in %eax to memory location (%ebx)
G6: pop %esp; ret
G7: jmp %ebx; ret
G8: add (%ebx), %eax; ret // add value at memory location (%ebx) to %eax
G9: ret
G10: pop %esp; ret

(b) Add 0x10 to the value at address 0xf0020

Stack:

- 0x10 into register – pop eax; 0x10
- 0xf0020 into register – pop ebx, 0xf0020
- Add 0x10 to the value at 0xf0020 into eax – add (ebx), eax
- Move value from eax to 0xf0020 – mov eax, (ebx)
Question #12

Gadgets

G1: push %eax; ret
G2: pop %eax; ret
G3: push %ebx; ret
G4: pop %ebx; ret
G5: mov %eax, (%ebx); ret // store value in %eax to memory location (%ebx)
G6: pop %esp; ret
G7: jmp %ebx; ret
G8: add (%ebx), %eax; ret // add value at memory location (%ebx) to %eax
G9: ret
G10: pop %esp; ret

• (c) Create an infinite loop that runs instructions starting at 0xf8800 repeatedly.

  ▶ Build Stack:
Question #12

Gadgets

G1: push %eax; ret
G2: pop %eax; ret
G3: push %ebx; ret
G4: pop %ebx; ret
G5: mov %eax, (%ebx); ret // store value in %eax to memory location (%ebx)
G6: pop %esp; ret
G7: jmp %ebx; ret
G8: add (%ebx), %eax; ret // add value at memory location (%ebx) to %eax
G9: ret
G10: pop %esp; ret

• (c) Create an infinite loop that runs instructions starting at 0xf8800 repeatedly.

  ▶ Build Stack:

  • (1) Load gadget G10 at 0xf8800
  • (2) Load value of 0xf8800 after that
  • Memory write gadgets for performing (1) and (2)
Question #13

• Harden the following statements as much as is feasible ...
  ‣ fgets
  ‣ sscanf(string, “%s”, …)
  ‣ strcpy
  ‣ snprintf(char *out, “%s”, …)
  ‣ strtok(char *str, …)
Question #13

- Harden the following statements as much as is feasible ...

  - `fgets`
  - `sscanf(string, "\%s", ...)`
  - `strcpy`
  - `sscanf(string, "\%ms", ...)`
  - `strtok(char *str, ...)`

- `getline`
- `sscanf(string, "\%ms", )`
- `strlcpy`
- `strlcpy or snprintf w/ \"\%.*s\"
- `strtok_s`
Question #14

• Attacks on name resolution

• Files readable by adversary supplying html_page?
  ‣ What does the strcat prevent?
  ‣ What does the check prevent?
  ‣ What does strip prevent?

// file_path is "/home/" at this point
1:  bytes = recv(socket, html_page, LIMIT, FLAGS);
2:  strcat(file_path, html_page, LIMIT);
3:  check_stat(html_page, NOT_SYMLINK);
4:  fd = open(html_page, RDONLY);
5:  bytes = read(fd, outbuf, LIMIT);
6:  send(socket, outbuf, LIMIT, FLAGS);
Question #14

- Attacks on name resolution

- Files readable by adversary supplying html_page?
  - What does the strcat prevent?
  - What does the check prevent?
Question #14

• Attacks on name resolution

```c
// file_path is "/home/" at this point
1: bytes = recv(socket, html_page, LIMIT, FLAGS);
2: strcat(file_path, html_page, LIMIT);
3: check_stat(html_page, NOT_SYMLINK);
4: fd = open(html_page, RDONLY);
5: bytes = read(fd, outbuf, LIMIT);
6: send(socket, outbuf, LIMIT, FLAGS);
```

• Files readable by adversary supplying html_page?

  🌟 What does the strcat prevent? **Nothing – “../”**
  🌟 What does the check prevent? **Nothing – TOCTTOU**
  🌟 What does strip “../” prevent? **Nothing unless canonicalized**
Question #14

• Attacks on name resolution

• What attacks are possible if adversary can write to a directory used in name resolution?

// file_path is "/home/" at this point
1: bytes = recv(socket, html_page, LIMIT, FLAGS);
2: strncat(file_path, html_page, LIMIT);
3: check_stat(html_page, NOT_SYMLINK);
4: fd = open(html_page, RDONLY);
5: bytes = read(fd, outbuf, LIMIT);
6: send(socket, outbuf, LIMIT, FLAGS);
Question #14

- Attacks on name resolution

- What attacks are possible if adversary can write to a directory used in name resolution?
  - Link traversal – to another directory
  - Squatting – plant a file

- Runtime testing? Create a test case directed at these attacks

```c
// file_path is "/home/" at this point
1: bytes = recv(socket, html_page, LIMIT, FLAGS);
2: strcat(file_path, html_page, LIMIT);
3: check_stat(html_page, NOT_SYMLINK);
4: fd = open(html_page, RDONLY);
5: bytes = read(fd, outbuf, LIMIT);
6: send(socket, outbuf, LIMIT, FLAGS);
```
Midterm

• Format
  ‣ True/False
    • 8 questions – 16 pts
  ‣ Short answer – word/phrase to sentence or two
    • 8 questions – 36 pts
  ‣ Question – conceptual (why?) and constructions (how?)
    • 6 questions – 48 pts
    • Conceptual – can be a bit open-ended
    • Constructions – like questions 11-14 from homework, but fewer parts
  ‣ Exams can be long-ish – 3+ minutes per question