Peyrin Kao
Fall 2023

CS 161 Computer Security

Exam Prep 1

Q1 *Security Principles* (0 points)

Select the best answer to each question.

Q1.1 A company requires that employees change their work machines' passwords every 30 days, but many employees find memorizing a new password every month difficult, so they either write it down or make small changes to existing passwords. Which security principle does the company's policy violate?

- ○ Defense in depth
- ○ Ensure complete mediation
- ○ Consider human factors
- ○ Fail-safe defaults

Q1.2 In the midst of a PG&E power outage, Carol downloads a simple mobile flashlight app. As soon as she clicks a button to turn on the flashlight, the app requests permissions to access her phone's geolocation, address book, and microphone. Which security principle does this violate?

- ○ Security is economics
- ○ Least privilege
- ○ Separation of responsibility
- ○ Design in security from the start

Q1.3 A private high school has 100 students, who each pay $10,000 in tuition each year. The principal hires a CS 161 alum as a consultant, who discovers that the "My Finances" section of the website, which controls students' tuition, is vulnerable to a brute force attack. The consultant estimates an attacker could rent enough compute power with $20 million to break the system, but tells the principal not to worry because of *which security principle?*

- ○ Security is economics
- ○ Design in security from the start
- ○ Least privilege
- ○ Consider human factors

Q1.4 The consultant notices that a single admin password provides access to all of the school's funds and advises the principal that this is dangerous. What principle would the consultant argue the school is violating?

- ○ Don't rely on security through obscurity
- ○ Design security in from the start
- ○ Separation of responsibility
- ○ Fail-safe defaults

Q1.5 Course staff at Stanford's CS155 accidentally released their project with solutions in it! In order to conceal what happened, they quickly re-released the project and didn't mention what had happened in the hope that no one would notice. This is an example of not following which security principle?

- ○ Security is economics
- ○ Know your threat model
- ○ Don't rely on security through obscurity
- ○ Least privilege
- ○ Separation of responsibility
- ○ None of these

Q2 *x86 Potpourri* **(0 points)**

Here are the 11 steps for x86 calling convention for reference:

1. Push arguments onto the stack.
2. Push the old eip (rip) on the stack.
3. Move eip.
4. Push the old ebp (sfp) on the stack. (`push %ebp`)
5. Move ebp down. (`mov %esp, %ebp`)
6. Move esp down.
7. Execute the function.
8. Move esp up. (`mov %ebp, %esp`)
9. Restore the old ebp (sfp). (`pop %ebp`)
10. Restore the old eip (rip). (`pop %eip`)
11. Remove arguments from the stack.

Q2.1 In normal (non-malicious) programs, the EBP is *always* greater than or equal to the ESP.

○ True ○ False

Q2.2 Arguments are pushed onto the stack in the same order they are listed in the function signature.

○ True ○ False

Q2.3 A function always knows ahead of time how much stack space it needs to allocate.

○ True ○ False

Q2.4 Step 10 ("Restore the old eip (rip).") is often done via the `ret` instruction.

○ True ○ False

Q2.5 In GDB, you run `x/wx &arr` and see this output:

```
0xfffff62a: 0xfffff70c
```

True or False: `0xfffff62a` is the address of `arr` and `0xfffff70c` is the value stored at `arr`.

○ True ○ False

Q2.6 Which steps of the x86 calling convention are executed by the *caller*?

Q2.7 Which steps of the x86 calling convention are executed by the *callee*?

Q2.8 Which steps of the x86 calling convention are considered the "function prologue"?

Q2.9 Which steps of the x86 calling convention are considered the "function epilogue"?

Q2.10 What does the `nop` instruction do?

Q2.11

```
    RIP of main
    pop %eip
    SFP of bar
```

EvanBot has edited his program stack to look like the above. They reason that when `bar` returns, "`pop %eip`" will be popped into the EIP, which is then executed to pop "RIP of main" into the EIP. Note that the value "`pop %eip`" on the stack represents the actual value, not a variable name or pointer.

Is this correct? Explain why or why not.

Q3 *Terminated* **(0 points)**

Consider the following C code excerpt.

```
 1 typedef struct {
 2     char first[16];
 3     char second[16];
 4 } message;
 5
 6 void main() {
 7     message msg;
 8
 9     fgets(msg.first, 17, stdin);
10
11     for (int i = 0; i < 16; i++) {
12         msg.second[i] = msg.first[i];
13     }
14
15     printf("%s\n", msg);
16     fflush(stdout);
17 }
```

Q3.1 Fill in the following stack diagram, assuming that the program is paused at **Line 9**.

Stack

Q3.2 Now, draw arrows on the stack diagram denoting where the ESP and EBP would point if the code were executed until a breakpoint set on **line 14**.

You run GDB once, and discover that the address of the RIP of `main` is `0xffffcd84`.

Q3.3 What is the address of `msg.first`?

Q3.4 Here is the `fgets` documentation for reference:

```
char *fgets(char *s, int size, FILE *stream);

      fgets() reads in at most one less than size  characters from stream and
      stores them into the buffer pointed to by s. Reading stops after an EOF
      or a newline.  If a newline is read, it is stored into  the  buffer.  A
      terminating null byte ('\0') is stored after the last character  in the
      buffer.
```

Evanbot passes in "hello" to the `fgets` call and sees the program print "hello". He expected it to print "hellohello" since the first half was copied into the second half. Why is this not the case?

Q3.5 Evanbot passes in "hellohellohello!" (16 bytes) to the `fgets` call and sees the program print "hellohellohello!hellohellohello!oaNWActYKJjflv5wI …" (not real output). The program seems to have correctly copied the message, but EvanBot wonders why there seems to be garbage output at the end. Why is this the case, and how can they fix their program?