Q1 *Security Principles* (10 points)

Select the best answer to each question.

Q1.1 (2 points) 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

Solution: Here is an article that discusses why password rotation should be phased out in practice, if you're interested in reading more.

Q1.2 (2 points) 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

Solution: A flashlight application does not actually need these permissions in order to execute its functionality. It is over-permissioning its access to sensitive resources, violating the principle of least privilege.

Q1.3 (2 points) 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

Solution: The website handles $1 million per year; not large enough that an attacker would have an incentive to spend $20 million to steal it.

Q1.4 (2 points) 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 in security from the start
- ● Separation of responsibility
- ○ Fail-safe defaults

Q1.5 (2 points) 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 the above

Solution: Uhh, can you guess where we got the idea for this question? Hint: It wasn't Stanford...

Q2 *x86 Potpourri (Extended)* **(11 points)**

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

● True ○ False

Q2.2 (1 point) Arguments are pushed onto the stack in the same order they are listed in the function signature.

○ True ● False

Solution: Arguments are pushed in reverse order.

Q2.3 (1 point) A function always knows ahead of time how much stack space it needs to allocate.

● True ○ False

Solution: This corresponds to Step 6 of the calling convention.

Q2.4 (1 point) Step 10 ("Restore the old eip (rip).") is often done via the `ret` instruction.

● True ○ False

Solution: `ret` is equivalent to `pop %eip`.

Q2.5 (1 point) 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

Solution: The left side of a GDB output corresponds to the address, and the right side corresponds to the value at the address.

Q2.6 (1 point) Which steps of the x86 calling convention are executed by the *caller*?

Steps 1, 2, 3, and 11.

Q2.7 (1 point) Which steps of the x86 calling convention are executed by the *callee*?

Steps 4-10.

Q2.8 (1 point) What does the `nop` instruction do?

Solution: `nop` does nothing and moves the EIP to the next instruction.

Q2.9 (1 point) Consider the following C code and some of its assembly:

```
void foo(int bar) {
    // Implementation not shown
}

void main() {
  int bar = 0;
  foo(bar);
}
```

```
x08001008: ____________________
x0800100c: call foo
x08001010: ____________________
```

Fill in the blanks for the instructions surrounding `call foo` in the assembly for `main`.

Solution: The first line will be pushing the arguments (in this case, a single 0, represented as the immediate $0).

The last line will be Step 11 in the calling convention, moving the ESP back up past the arguments pushed onto the stack.

```
0x08001008: push $0
0x0800100c: call foo
0x08001010: add $4, %esp
```

Q2.10 (1 point) EvanBot manages to set the value of the SFP of `foo` to `0x00000000` before `foo` returns. What is most likely to happen next?

- ○ The program will crash immediately, before returning from `foo`.
- ○ The program will crash when attempting to return from `foo`.
- ● The program will crash when attempting to return from `main`.
- ○ The program will finish executing without crashing.

Solution: When returning from `foo`, EBP will be set to null, but is otherwise not used (note that no arguments are accessed in `main`). When `main` returns, ESP is set to EBP and then popped, which will cause a segmentation fault crash due to trying to read from a null pointer.

Q2.11 (1 point) EvanBot has edited their program stack to look like the following.

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

They reason that when `foo` 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.

○ Correct ● Incorrect

Solution: This will not work because EIP holds an address to an instruction, not the instruction itself. We would need to have the address of ret instead of ret itself.

Q3 *Terminated* (5 points)

Consider the following C code excerpt.

```c
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 (1 point) Fill in the following stack diagram, assuming that the program is paused at **Line 9**.

[4] `RIP of main`
[4] `SFP of main`
[16] `msg.second`
[16] `msg.first`

Q3.2 (1 point) 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.

Solution:

ESP points to msg.first, EBP points to main's SFP.

	[4] RIP of main
EBP →	[4] SFP of main
	[16] msg.second
ESP →	[16] msg.first

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

Q3.3 (1 point) What is the address of `msg.first`?

0xffffcd60

Solution:

SFP + msg.second + msg.first

= 4 bytes + 16 bytes + 16 bytes

= 36 bytes away

So, the address of msg.first is `0xffffcd84` − decimal 36 = `0xffffcd60`.

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.
```

Q3.4 (1 point) 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?

Solution:

`fgets` puts a null terminator at the end, which stops the **`printf`** after the first string.

Q3.5 (1 point) 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?

Solution:

fgets puts a null terminator at the end, which stops the printf after the first string. However, the limit given is 17 instead of 16, which means the entire first buffer is filled with non-null characters. This buffer is then copied to the one above it on the stack, erasing the null terminator, and letting printf keep going up the stack past the end of the normal buffer.