Introduction to Computer Security

Q1 Security Principles

CS 161

Spring 2025

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?

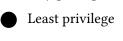
O Defense in depth	O Ensure complete mediation
Consider human factors	\bigcirc Fail-safe defaults

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

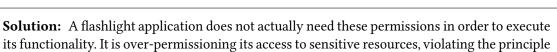
Q1.2 In the midst of a PG&E power outage, Carol downloads a simple mobile flashlight app. As soon as she clicks a button to turn on the flashlight, the app requests permissions to access her phone's geolocation, address book, and microphone. Which security principle does this violate?

Security is economics

of least privilege.



Separation of responsibility



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*?



Loost privilago

Design in security from the start

Consider human factors

Design in security from the start

Least privilege

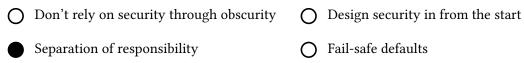
Security is economics

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.

(0 points)

Exam Prep 1

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?



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?

0	Security is economics	0	Know your threat model
	Don't rely on security through obscurity	0	Least privilege
0	Separation of responsibility	0	None of these

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

Q2 x86 Potpourri (Extended)

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

	• True	O False		
		vely, the EBP represents the top of the stack frame and the ESP repre- current stack frame. You can also follow the calling convention to see re concretely.		
Q2.2	Arguments are pushed	nto the stack in the same order they are listed in the function signature		
	O True	False		
	Solution: Arguments lowest offset from the	are pushed in reverse order, since we want the first argument to be the EBP.		
Q2.3 A function always knows ahead of time how much stack space it needs to allocate.				
	True	O False		
	Solution: This corresponds to Step 6 in the calling convention.			
Q2.4	Step 10 ("Restore the old eip (rip).") is often done via the ret instruction.			
	True	O False		
	Solution: ret is equ	ralent to pop %eip.		
Q2.5	In GDB, you run x/wx	arr and see this output:		
	0xfffff62a: 0x:	ffff70c		
	True or False: 0xfffff62a is the address of arr and 0xfffff70c is the value stored at arr.			
	• True	O False		
	Solution: Left side is	address, right side is values.		
Q2.6	Which steps of the x86 calling convention are executed by the <i>caller</i> ?			
	Solution: Steps 1, 2, 3, and 11 take place in the caller function.			

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

Solution: Steps 4-10 take place in the callee function.

```
Q2.8 What does the nop instruction do?
```

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

Q2.9 Consider the following C code and some of its assembly:

```
void foo(int bar) {
    // Implementation not shown
}
void main() {
    int bar = 0;
    foo(bar);
}
1 0x08001008: ______
2 0x0800100c: call foo
3 0x08001010: ______
```

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.

```
1 0x08001008: push $0
2 0x0800100c: call foo
3 0x08001010: add $4, %esp
```

- Q2.10 EvanBot manages to set the value of the SFP of foo to 0x00000000 before foo returns. What is most likely to happen next?
 - O The program will crash immediately, before returning from **foo**.
 - O The program will crash when attempting to return from **foo**.
 - The program will crash when attempting to return from main.
 - O 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

RIP of main pop %eip SFP of foo

EvanBot has edited his program stack to look like the above. 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.

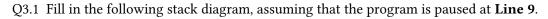
Solution: This will not work because EIP holds an address to an instruction, not the instruction itself. We would need to have the address of ret instead of ret itself.

(0 points)

Q3 Terminated

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
       fgets (msg. first, 17, stdin);
9
10
       for (int \ i = 0; \ i < 16; \ i++) {
11
12
            msg.second[i] = msg.first[i];
13
       }
14
15
       printf ("\%s \n", msg);
       fflush(stdout);
16
17
  }
```



Stack

1	
1	

Solution: Stack diagram:

RIP of main SFP of main msg.second msg.first 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**.

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

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?

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.

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?

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

Q3.5 Evanbot passes in "hellohellohello!" (16 bytes) to the fgets call and sees the program print "hellohellohellohellohellohello!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.