CS161 Spring 2026 — Introduction to Computer Security — Discussion 1

Q1 *Security Principles* (10 points)

We discussed the following security principles in lecture (*or in the textbook*):

A. *Know Your Threat Model:* Know your attacker and their resources; the security assumptions originally made may no longer be valid

B. *Consider Human Factors:* Security systems must be usable by ordinary people

C. *Security is Economics:* Security is a cost-benefit analysis, since adding security usually costs more money

D. *Detect If You Can't Prevent:* If one cannot prevent an attack, one should be able to at least detect when an attack happens

E. *Defense in Depth:* Layer multiple defenses together

F. *Least Privilege:* Minimize how much privilege you give each program and system component

G. *Separation of Responsibility:* Split up privilege, so no one person or program has complete power

H. *Ensure complete mediation:* Make sure to check *every* access to *every* object

I. *Consider Shannon's Maxim:* Do not rely on security through obscurity

J. *Use fail-safe defaults:* If security mechanisms fail or crash, they should default to secure behavior

K. *Design in security from the start:* Retrofitting security to an existing application after it has been developed is a difficult proposition

Identify the principle(s) relevant to each of the following scenarios.

Note that there may be more than one principle that applies in some of these scenarios.

Q1.1 (1 point) New cars often come with a valet key. This key is intended to be used by valet drivers who park your car for you. The key opens the door and turns on the ignition, but it does not open the trunk or the glove compartment.

> **Solution: Least Privilege.** They do not need to access your trunk or your glove box, so you don't give them access to do so.

Q1.2 (1 point) Many homeowners leave a house key under the floor mat in front of their door.

> **Solution: Shannon's Maxim.** The security of your home depends on the belief that most criminals don't know where your key is. With a modicum of effort, criminals could find your key and open the lock.

Q1.3 (1 point) It is not worth it to use a $400,000 bike lock to protect a $100 bike.

> **Solution: Security is Economics.** It is more expensive to buy $400 bike lock than to simply buy a new bike to replace it.

Q1.4 (1 point) Social security numbers were not originally designed as a secret identifier. Nowadays, they are often easily obtainable or guessable.

> **Solution: Design security from the start.** Social security numbers were not designed to be authenticators, so security was not designed from the start. The number is based on a geographic region, a sequential group number, and a sequential serial number. They have since been repurposed as authenticators.

Q1.5 (1 point) Warranties on cell phones do not cover accidental damage, which includes liquid damage. However, many consumers who accidentally damage their phones with liquid will wait for it to dry and then claim that "it broke by itself." To combat this threat, many companies have begun to include on the product a small sticker that turns red (and stays red) when it gets wet.

> **Solution:** There are probably two most relevant factors. In order of relevance:
>
> **Detect if you can't prevent.** It's prudent to try to add ways to detect something when creating the phone since something like water damage is impossible to prevent.
>
> **Consider Human Factors.** People will always try to lie and you must account for that when creating a system.

Q1.6 (1 point) Even if you use a password on your laptop lock screen, there is software that lets a skilled attacker with specialized equipment bypass it.

> **Solution: Know Your Threat Model.** Most petty thieves do not have access to this software.

Q1.7 (1 point) Shamir's secret sharing scheme allows us to split a "secret" between multiple people so that all of them have to collaborate in order to recover the secret.

> **Solution: Separation of Responsibility.** Everyone is required to come together to produce the secret, preventing one person from using the secret alone.

Q1.8 (1 point) Banks often make you answer your security questions over the phone. Answers to these questions are "low entropy," meaning that they are easy to guess. Some security-conscious people instead use a random password as the answer to the security question.[1] However, attackers can sometimes convince the phone representative by claiming "I just put in some nonsense for that question."

> **Solution: Consider Human Factors.** The phone rep is inclined to believe the attacker is not malicious (social engineering).

Q1.9 (1 point) Often times at bars, an employee will wait outside the only entrance to the bar, enforcing that people who want to enter the bar form a single-file line. Then, the employee checks each individual's ID to verify if they are 21 before allowing them entry into the bar.

> **Solution: Ensure Complete Mediation.** There is a single access point through which everyone who wishes to enter the bar must be verified to be 21 before obtaining access.

[1]Q: What is your dog's maiden name? A: "`60ba6b1c881c6b87`"

Q1.10 (1 point) Some electric vehicles come equipped with a secure park mode which records footage of any break-ins to the vehicle and alerts the vehicle owner of the incident.

Solution: Detect if you can't prevent. The vehicle owner learns about the intrusion into their vehicle even if they were not able to prevent it.

Q2 *Stack Diagram Practice* **(3 points)**

For your reference, reproduced below are the 11 steps of x86 calling convention:

1. Push arguments onto the stack.
2. Push the old eip (rip) onto the stack
3. Move EIP

   Execution changes to the callee here.
4. Push the old ebp (sfp) onto 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.

Consider the following function.

```
int swap(int* num1, int* num2, int arr_local[]) {
    int temp = *num1;
    *num1 = *num2;
    arr_local[0] = *num1;
    *num2 = temp;
    arr_local[1] = *num2;
    return 0;
}

int main(void) {
    int x = 61;
    int y = 1;
    int arr[2];
    swap(&x, &y, arr);
    return 0;
}
```

Q2.1 (1 point) Draw the stack diagram if the code were executed until a breakpoint set on line 4. Assume normal (non-malicious) program execution. You do not need to write the values on the stack, only the names. When drawing the stack diagram, assume that each row in your diagram doesn't have to represent 4 bytes in memory. The bottom of the page represents the lower addresses.

[4] `RIP of main`
[4] `SFP of main`
[4] `x`
[4] `y`
[8] `arr`
[4] `int* arr_local`
[4] `int* num2`
[4] `int* num1`
[4] `RIP of swap`
[4] `SFP of swap`
[4] `temp`

Q2.2 (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 4. Label these `ESP4` and `EBP4` respectively.

Solution: The `ESP` should point to `temp` and the `EBP` should point to `SFP of swap`.

	`[4] RIP of main`
	`[4] SFP of main`
	`[4] x`
	`[4] y`
	`[4] x`
	`[8] arr`
	`[4] int* arr_local`
	`[4] int* num2`
	`[4] int* num1`
	`[4] RIP of swap`
EBP4 $\rightarrow$	`[4] SFP of swap`
ESP4 $\rightarrow$	`[4] temp`

Q2.3 (1 point) The return instruction executes steps 8-10 of the calling convention. Draw arrows on the stack diagram denoting where the ESP and EBP would point for each of these steps. Label these `ESP8-10` and `EBP8-10` respectively.

Solution:

1. `ESP8` and `EBP8` point to `SFP of swap`.
2. `ESP9` points to `RIP of swap` and `EBP9` points to `SFP of main`.
3. `ESP10` points to `int* num1` and `EBP10` points to `SFP of main`.
   Note that `EIP` now points to line 15.

	`[4] RIP of main`
EBP9, EBP10 $\rightarrow$	`[4] SFP of main`
	`[4] x`
	`[4] y`
	`[4] x`
	`[8] arr`
	`[4] int* arr_local`
	`[4] int* num2`
ESP10 $\rightarrow$	`[4] int* num1`
ESP9 $\rightarrow$	`[4] RIP of swap`
ESP8, EBP8 $\rightarrow$	`[4] SFP of swap`
	`[4] temp`

Q3 *x86 Potpourri* (8 points)

Q3.1 (1 point) In normal (non-malicious) programs, the EBP is *always* greater than or equal to the ESP.

[✓ True] False

Q3.2 (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.

Q3.3 (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.

Q3.4 (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`.

Q3.5 (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.

Q3.6 (1 point) Which steps of the x86 calling convention are executed by the *caller*?

Steps 1, 2, 3, and 11.

Q3.7 (1 point) Which steps of the x86 calling convention are considered the "function epilogue"?

Steps 8-10.

Q3.8 (1 point) What does the `nop` instruction do?

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