CS161 Fall 2025 — Introduction to Computer Security — Midterm

Name: _____

Student ID: _____

This exam is 110 minutes long. There are 8 questions of varying credit. (100 points total)

Question:	1	2	3	4	5	6	7	8	Total
Points:	0	15	18	13	13	12	19	10	100

For questions with **circular bubbles**, you may select only one choice.

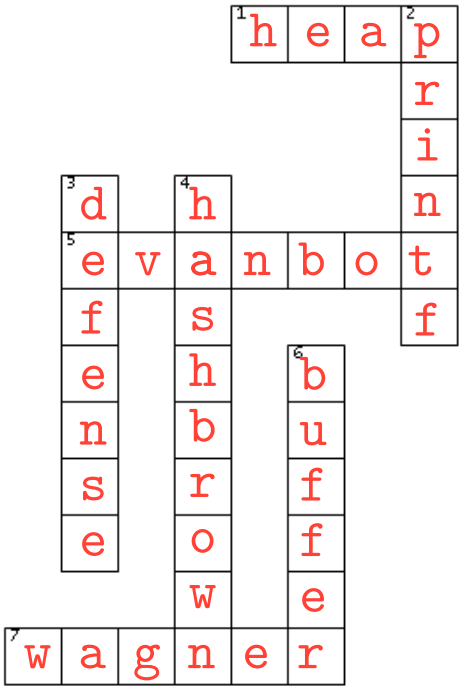
- ○ Unselected option (Completely unfilled)
- ◉ Don't do this (it will be graded as incorrect)
- ● Only one selected option (completely filled)

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

- ■ You can select
- ■ multiple squares (completely filled).
- ☑ (Don't do this)

Anything you write outside the answer boxes or you ~~cross out~~ will not be graded. If you write multiple answers, your answer is ambiguous, or the bubble/checkbox is not entirely filled in, we may grade the worst interpretation.

Pre-Exam Activity: Crossword (0 points):



ACROSS:
1. What goes up
5. Cutest mascot
7. Professor _____

DOWN:
2. Don't forget to format
3. _____ in depth
4. Tastiest one-way function
6. Always overflowing

Q1 *Honor Code* 📜 **(0 points)**

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.

Read the honor code above and sign your name: _____

Q2 *Potpourri* **(15 points)**

Q2.1 (1 point) True or False: The Trusted Computing Base (TCB) is another term for the data that we're trying to protect from an attack.

○ True ● False

Solution: The trusted computing base (TCB) is that portion of the system that must operate correctly in order for the security goals of the system to be assured. This is the foundation of trust, not the target of protection.

Q2.2 (1 point) A web app lets logged-in users access files by specifying a filename in the URL, but it does not restrict access, so any user can view all other users' files. Which security principle is violated?

○ Shannon's Maxim

○ Fail-safe defaults

○ Detect if you can't prevent

● Ensure complete mediation

○ None of the above

Solution: The security principle of **Ensure complete mediation** is violated because the web app fails to check for authorization on **every** access to a file, allowing logged-in users to view files they don't own.

We also accepted Shannon's Maxim. The course staff debated whether to accept this answer, and we ultimately decided to accept it because someone who knows how the web app is structured would know that they can access any file.

We did not accept "None of the above". While there could be an argument for least privilege in addition to "Ensure complete mediation" and/or "Shannon's Maxim", we thought those two were applicable enough that we weren't willing to accept "None of the above".

Q2.3 (1 point) True or False: When storing a word `0xDEADBEEF` in memory on a *big*-endian system, the bytes at increasing addresses are `\xEF`, `\xBE`, `\xAD`, `\xDE`.

○ True ● False

Solution: In a big-endian system, the most significant byte is stored at the lowest memory address.

Q2.4 (1 point) Which of the following registers stores the address of the top of the current stack frame?

- ○ `eax`
- ○ `eip`
- ● `ebp`
- ○ None of the above

Solution: The `ebp` (Base Pointer) register stores the base address of the current function's stack frame, acting as a stable reference point for accessing local variables and arguments.

Q2.5 (1 point) True or False: `gets` is a memory-safe function because it will always automatically append a null terminator at the end.

○ True ● False

Solution: `gets` has no bounds checking, making it memory unsafe and susceptible to overflow.

Q2.6 (1 point) True or False: Buffer overflows cannot be exploited if stack canaries are in use, because then the return address cannot be overwritten.

○ True ● False

Solution: Some buffer overflows can be exploited (`e.g., an overflow of a heap buffer, a format string vulnerability`). Even for an overflow of a stack-allocated buffer, the return address can still be overwritten; it might be detected, but it can still be overwritten.

Q2.7 (3 points) An attacker is exploiting a buffer overflow in a program with ASLR and non-executable pages enabled. Which of the following attacks is most feasible?

- ○ Inject shellcode onto the stack and jump to it.
- ● Find some other vulnerability that reveals the address of a `libc` function, then overwrite the return address with the `libc` function address (return-to-`libc`).
- ○ Overwrite the address of the saved frame pointer (SFP) on the stack.
- ○ Overwrite the least significant byte of the saved frame pointer (SFP) on the stack with a null byte, so when the current function returns, the caller uses a fake stack frame and jumps to shellcode stored in the fake RIP (off-by-one).

Solution:

- Non-executable pages enabled prevents execution of injected shellcode, so option 1 is blocked.
- ASLR randomizes the location of the stack, so it's hard for an attacker to know what address to write over the SFP, so option 3 won't work.
- ASLR randomizes the location of the code segment, so it's hard for an attacker to know what address to put in the fake RIP, so option 4 is hard to make work. The attacker has to hope that there is already a pointer/address stored somewhere on the stack, that points to malicious code that when executed has a harmful effect. That's unlikely to arise in most scenarios, so option 4 is typically not an option.

Q2.8 (1 point) True or False: In hybrid encryption, the public key is used to encrypt the message, and the symmetric key is used only to sign.

○ True ● False

Solution: In hybrid encryption, the public key encrypts a randomly chosen symmetric session key, and the session key encrypts the bulk message data. Signing is a separate operation and is not the purpose of the symmetric key.

Q2.9 (1 point) True or False: A major drawback of a Trusted Directory is that it is not scalable.

● True ○ False

Solution: This is explained in Lecture 12 slide 10. Trusted directories suffer from a scaling issue since no server can handle the amount of data required to serve information to the entire internet. We resolve this by using a hierarchical system that allows most of the traffic to be distributed to the servers that actually share the content.

Q2.10 (4 points) Suppose Alice and Bob share a secret key K used with a secure MAC algorithm. Alice sends (M, T), where $T = \text{MAC}(K, M)$, and a man-in-the-middle attacker intercepts it.

The attacker wants to replace Alice's message with $(M \parallel 1, T')$, with T' chosen so that Bob will accept this modified message as valid. Is it feasible for an attacker to do this, without knowing the key K?

○ Yes, because the attacker knows M and anyone can compute the hash of $M \parallel 1$

○ Yes, because MACs provide confidentiality, not integrity

○ Yes, because MACs are deterministic

● No, because MACs are unforgeable

○ No, because MACs are collision resistant

○ No, because MACs are deterministic

○ None of the above

Solution: The key property of a secure MAC is that, without the secret key, it should be computationally infeasible for an attacker to create a valid tag on a new message, even one closely related to a known authenticated message.

Q3 *Memory Safety: Betelgeuse* **(18 points)**

For each C code snippet below, answer the two subsequent questions regarding the vulnerability present and the best way to mitigate it.

```
1 void get_user_greeting() {
2     char user_name[32];
3     printf("Please enter your name:\n");
4     gets(user_name);
5     printf("Hello, %s!\n", user_name);
6 }
```

Q3.1 (2 points) Which of the following memory safety vulnerabilities are present in the code snippet above? Select all that apply.

- ■ Stack Smashing / Buffer Overflow
- ☐ Format String Vulnerability
- ☐ Integer Conversion Vulnerability
- ☐ Time-of-Check/Time-of-Use
- ☐ Off-by-One Vulnerability
- ○ None of the above

Solution:

- Replace `gets(user_name);` with `fgets(user_name, 32, stdin);`: This code-level fix prevents the overflow by limiting the input to the buffer's size.
- `Enable Stack Canaries`: This compiler-level mitigation detects the overflow by placing a sacrificial value (canary) on the stack, which, if overwritten, causes the program to abort.

Q3.2 (4 points) Which of the following changes would effectively mitigate the vulnerabilities identified in the previous subpart? Select all that apply.

- ■ Replace `gets(user_name);` with `fgets(user_name, 32, stdin);`.
- ☐ Insert `if (strlen(user_name) >= 32) return;` after line 4 and before line 5.
- ■ Enable Stack Canaries.
- ☐ Replace `char user_name[32];` with `char username[16];`.
- ○ None of the above

Solution:

- Replace `gets(user_name);` with `fgets(user_name, 32, stdin);`: This code-level fix prevents the overflow by limiting the input to the buffer's size.
- `Enable Stack Canaries`: This compiler-level mitigation detects the overflow by placing a sacrificial value (canary) on the stack, which, if overwritten, causes the program to abort.

```
1 void log_status(char *status) {
2     char msg[24];
3     fread(msg, 1, 24, stdin);
4     printf(status);
5     printf(msg);
6 }
```

Q3.3 (2 points) Which of the following memory safety vulnerabilities are present in the code snippet above? Select all that apply.

- ☐ Stack Smashing / Buffer Overflow
- ■ Format String Vulnerability
- ☐ Integer Conversion Vulnerability
- ☐ Off-by-One Vulnerability
- ☐ Time-of-Check/Time-of-Use
- ○ None of the above

Solution: This is a **Format String Vulnerability**. Passing user-controlled data (`status` and `msg`) as the first argument to `printf` allows an attacker to use format specifiers (like `%x` or `%n`) to read from or write to memory.

Q3.4 (4 points) Which of the following changes would effectively mitigate the vulnerabilities identified in the previous subpart? Select all that apply.

- ☐ Replace `fread(msg, 1, 24, stdin)` with `fgets(msg, 24, stdin)`.
- ■ Replace `printf(status)` with `printf("%s", status)` and replace `printf(msg)` with `printf("%s", msg)`.
- ☐ Allocate `status` and `msg` on the heap instead of the stack.
- ☐ Enable Address Space Layout Randomization.
- ○ None of the above

Solution: The correct fix is to replace `printf(status)` with `printf("%s", status)` and replace `printf(msg)` with `printf("%s", msg)`. This provides a static format string (`"%s"`) and treats the user data as a simple string to be printed, not as format commands to be interpreted.

```
1 void process_message(char *msg, int len) {
2     char buffer[128];
3
4     if (len > 128) {
5         printf("Error: Message length exceeds buffer size.\n");
6         return;
7     }
8     memcpy(buffer, msg, len);
9 }
```

Q3.5 (2 points) Which of the following memory safety vulnerabilities are present in the code snippet above? Select all that apply.

- ■ Stack Smashing / Buffer Overflow
- ☐ Heap Overflow
- ■ Signed/Unsigned Integer Vulnerability
- ☐ Time-of-check/Time-of-Use
- ☐ Format String Vulnerability
- ◯ None of the above

Solution: This code has a Signed/Unsigned Integer Vulnerability leading to a Stack Smashing / Buffer Overflow.

A negative `int` for `len` will pass the `len > 128` check. When this negative `len` is passed to `memcpy`, it's converted to a very large **unsigned** `size_t`, causing `memcpy` to copy far more than 128 bytes and overflow the stack buffer.

Q3.6 (4 points) Which of the following changes would effectively mitigate the vulnerabilities identified in the previous subpart? Select all that apply.

- ☐ Use `strncpy(buffer, msg, len);` instead of `memcpy(buffer, msg, len);`
- ■ Add a check to ensure `len` is not negative before the size comparison.
- ■ Replace `int len` with `size_t len`
- ◯ None of the above

Solution:

- Add a check to ensure `len` is not negative before the size comparison: Explicitly checking `if (len < 0 or len > 128)` prevents the negative value from reaching `memcpy`.
- `Replace int len with size_t len`: This is a more robust fix, as `size_t` is unsigned, making a negative length impossible. The check `if (len > 128)` is then sufficient.

Q4 *Cryptography: One Question After Another* **(13 points)**

Q4.1 (3 points) Is a deterministic encryption scheme (where the same plaintext always produces the same ciphertext for a given key) considered IND-CPA secure?

- ○ Yes, because IND-CPA security is guaranteed as long as the underlying block cipher is a one-way function.
- ○ Yes, because the security relies on the secrecy of the key, not on randomizing the ciphertext output.
- ● No, because an attacker can detect if the same message is sent twice and thereby win the IND-CPA game.
- ○ No, because determinism makes the scheme vulnerable to brute-force attacks on the key space.

Solution: IND-CPA security requires that an attacker can't distinguish between the encryptions of two messages they choose, m_0 and m_1. If the scheme is deterministic, an attacker can submit m_0 and m_1 to the challenger to get back a challenge ciphertext C_b. The attacker then simply asks their own encryption oracle to encrypt m_0, which deterministically produces C_0. If $C_b = C_0$, the attacker knows the message was m_0, winning the game with 100% certainty.

Q4.2 (3 points) In a message encrypted with CBC mode, ciphertext block C_5 is corrupted, but C_6 and C_7 are received intact. Will the corresponding plaintext block P_7 be decrypted correctly?

- ● Yes, because the decryption of P_7 does not depend on the corrupted C_5.
- ○ Yes, because P_7 is only dependent on C_7 and the original IV.
- ○ No, because the "chaining" in CBC mode means an error in one block corrupts all subsequent plaintext blocks.
- ○ No, because decrypting P_7 requires a correctly decrypted P_6, which is impossible since C_5 was corrupted.

Solution: In CBC mode, the decryption formula for a plaintext block P_i is $P_i = D_k(C_i) \oplus C_{i-1}$.

- To decrypt P_7, the receiver computes $P_7 = D_k(C_7) \oplus C_6$. Since both C_7 and C_6 were received intact, P_7 is decrypted correctly.
- The corrupted C_5 only affects the decryption of P_5 (which uses $D_k(C_5)$) and P_6 (which uses C_5 for the XOR).

Q4.3 (4 points) Alice attempts to provide message integrity by sending a message M concatenated with its hash, as $M \| H(M)$, where H is a secure cryptographic hash function. Does this scheme protect against an active adversary who can modify the message in transit?

- ◯ Yes, because the one-way property of the hash prevents an attacker from creating a new message M' that hashes to the same value as the original hash $H(M)$.
- ◯ Yes, because Bob's verification step, where he re-computes the hash of the received M, would detect any tampering.
- ◯ No, because this scheme fails to provide confidentiality, and integrity is not possible without it.
- ● No, because the hash is unkeyed, allowing an attacker to modify the message and simply recompute a valid hash for the new message.

Solution: An active adversary can intercept the message $M \| H(M)$, create a new malicious message M', compute the new hash $H(M')$ (since the hash function H is public), and send $M' \| H(M')$ to the recipient. The recipient's check will pass, as the hash $H(M')$ correctly matches the malicious message M'. This scheme lacks authentication; a **Message Authentication Code (MAC)**, which uses a shared secret key, is required for integrity.

Q4.4 (3 points) Alice and Bob use the Diffie–Hellman key exchange to derive a shared secret g^{ab}. After each session, they erase both their private exponents (a, b) and the derived value g^{ab} before starting a new one. Does this provide forward secrecy if an attacker later learns the values of g, a and b?

- ◯ Yes, because forward secrecy concerns the ability to predict future session keys after the compromise, not before.
- ● Yes, because a compromise of future secrets would not reveal past session keys, as they were derived from discarded ephemeral secrets.
- ◯ No, because the attacker can perform a man-in-the-middle attack during the initial exchange, which retroactively breaks forward secrecy.
- ◯ No, because if the public parameters g and p are ever revealed, all past sessions become insecure.

Solution: This scenario describes Ephemeral Diffie-Hellman. The core principle of forward secrecy is that the compromise of secrets today (e.g., a future session's keys) does not allow an attacker to decrypt past communication. Since the private exponents a and b (and the resulting shared secret) are securely erased after each session, an attacker who compromises the system later and learns new exponents a' and b' has no way to recover the old, discarded keys.

Q5 *Cryptography: Slack DMs* **(13 points)**

Alice wishes to send Bob a message m over an insecure channel, and is deliberating over what scheme $F(m)$ to employ. A secure scheme should provide:

- **Confidentiality.** From the value $F(m)$ alone, no adversary should be able to efficiently recover the value of message m. Assume that dictionary attacks on m are not possible (there are too many possible values of m for the attacker to enumerate all of them).
- **Integrity.** It should be computationally infeasible to find two distinct messages $m \neq m'$ such that $F(m) = F(m')$ (a collision).

For each scheme $F(m)$ below, determine whether it provides **confidentiality**, **integrity**, **both**, or **neither**. In all questions, $\|$ denotes concatenation, and H denotes a secure cryptographic hash function.

Q5.1 (4 points) Alice sends:

$$F(m) = H(m)$$

○ Confidentiality only ○ Integrity only ○ Neither ● Both

Solution:

- Confidentiality: A secure cryptographic hash function H is preimage resistant (one-way). This means given the output $H(m)$, it is computationally infeasible to find the original input m. This directly matches the definition of confidentiality provided.
- Integrity: A secure cryptographic hash function H is collision resistant. This means it is computationally infeasible to find two distinct messages $m \neq m'$ such that $H(m) = H(m')$. This directly matches the definition of integrity provided.

Q5.2 (3 points) Let p be a publicly known, large prime number. Alice sends:

$$F(m) = 1^m \bmod p$$

● Confidentiality only ○ Integrity only ○ Neither ○ Both

Solution:

- Confidentiality: For any value of m, $F(m) = 1 \bmod p$. On top of that, because of the discrete log problem, it is computationally infeasible to find the value of m
- Because of this same reasoning, we can easily find $m \neq m'$ such that $F(m) = F(m')$ as $1^m \bmod p = 1^{m'} \bmod p = 1 \bmod p$.

For the next two subparts, Alice wishes to send a two-part message composed of m_0 and m_1 where $m_0 \neq m_1$. For each scheme $F(m)$ below, determine whether it provides **integrity**.

Note:
- $\mathsf{len}(m_0 \parallel m_1)$ returns the length of $m_0 \parallel m_1$,
- $\oplus$ denotes the XOR function,
- $(m_0, m_1) = (m_0', m_1')$ is true if and only if both $m_0 = m_0'$ and $m_1 = m_1'$.

Q5.3 (3 points) Alice sends:

$$F(m_1) = H(m_0 \parallel m_1 \parallel \mathsf{len}(m_0 \parallel m_1))$$

Does this scheme provide integrity?
In other words: Is it computationally infeasible to find two distinct pairs $(m_0, m_1) \neq (m_0', m_1')$ such that $F(m_0 \parallel m_1 \parallel \mathsf{len}(m_0 \parallel m_1)) = F(m_0' \parallel m_1' \parallel \mathsf{len}(m_0' \parallel m_1'))$.

○ Yes ● No

Solution: No this not provide integrity. If $m = (m_0, m_1) = (\text{Hello}, \text{World})$, then we can set $m' = (m_0', m_1') = (\text{HelloW}, \text{orld})$, such that $F(m) = F(m')$

Q5.4 (3 points) Alice sends:

$$F(m) = H(m_0 \oplus m_1)$$

Does this scheme provide integrity?
In other words: Is it computationally infeasible to find two distinct pairs $(m_0, m_1) \neq (m_0', m_1')$ such that $F(m_0 \oplus m_1) = F(m_0' \oplus m_1')$.

○ Yes ● No

Solution: No this not provide integrity. If $m = (m_0, m_1)$ then we can set $m_0' = m_1$ and $m_1' = m_0$, leaving us with $m' = (m_1, m_0)$ such that $F(m) = F(m')$

Q6 *Cryptography: Is This Random?* **(12 points)**

A pseudorandom number generator (PRNG) uses an initial **seed** of truly random bits to generate a long sequence of outputs. The seed serves as the starting point, or **initial state**, denoted s_0. From any given state s_i, the PRNG produces an output r_{i+1} and computes the next internal state s_{i+1}.

A secure PRNG should have the following properties:

- **Deterministic** – Given the same seed, a PRNG must always produce the same sequence of outputs. This ensures reproducibility (anyone who knows the seed and algorithm can regenerate the exact sequence).
- **Pseudorandom** – The output sequence is *computationally indistinguishable* from true randomness for any efficient adversary who does not know the seed or internal state, i.e., an adversary cannot tell the difference between output from the PRNG vs truly random bits. In practice, this means past outputs should not help an attacker predict the next output.
- **Rollback Resistant** – Even if an attacker learns the *current* internal state, they cannot reconstruct *earlier* outputs or states.

For each of the three functions below, select **all** the properties that the function satisfies. If a function has none of these properties, select "None of the above." H denotes SHA256 (which is a secure cryptographic hash function).

Q6.1 (4 points) Select all of the characteristics that this PRNG satisfies:

$$r_{i+1} = H(s_i \parallel 0), \quad s_{i+1} = H(s_i \parallel 1)$$

■ Deterministic ■ Pseudorandom ■ Rollback Resistant ○ None of the above

Solution:
- **Deterministic**: With the same seed, each step computes the same pair since both r_{i+1} and s_{i+1} are $H(s_i)$.
- **Pseudorandom**: For a cryptographic H, $H(s_i)$ is computationally indistinguishable from random to anyone who doesn't know the internal state.
- **Rollback Resistant**: From $s_{i+1} = H(s_i)$ (and r_{i+1}), computing s_i would require inverting H (preimage), which is infeasible.

Q6.2 (4 points) Select all of the characteristics that this PRNG satisfies:

$$r_{i+1} = H(i), \quad s_{i+1} = i + 1$$

■ Deterministic ☐ Pseudorandom ☐ Rollback Resistant ○ None of the above

Solution:
- **Deterministic**: The index i evolves deterministically and always yields the same $r_{i+1} = H(i)$ and $s_{i+1} = i + 1$.
- **Not Pseudorandom**: The sequence is predictable: it's just $H(0), H(1), H(2), \ldots$, so an adversary can reproduce/verify outputs without any secret.
- **Not Rollback Resistant**: Knowing the current state $s_i = i$ lets an adversary recompute all prior outputs $r_j = H(j)$ for $j < i$; past output is not protected.

Q6.3 (4 points) Select all of the characteristics that this PRNG satisfies:

$$r_{i+1} = H(s_i), \quad s_{i+1} = H(r_{i+1})$$

■ Deterministic ☐ Pseudorandom ■ Rollback Resistant ◯ None of the above

Solution:

- **Deterministic**: With the same seed, H is deterministic, so the (r_{i+1}, s_{i+1}) sequence repeats exactly.
- **Not Pseudorandom**: An adversary who observes r_{i+1} can compute s_{i+1} and then compute r_{i+2} (since the hash function is public and anyone can compute it), so an attacker who observes one output can predict all future outputs.
- **Rollback Resistant**: Since $s_{i+1} = H(H(s_i))$ is a one-way mapping in s_i, recovering s_i from s_{i+1} is computationally infeasible.

Q7 *Memory Safety: EvanBond, Double-O \x90* **(19 points)**

Consider the following vulnerable C code:

```
1  void q(int spectre) {
2      char goldfinger[8];
3      fread(goldfinger, 13, 1, stdin);
4      for (int i = 0; i < 8; i++) {
5          goldfinger[i] = 0x90;
6      }
7  }
8
9  void m() {
10     q(007);
11 }
```

Stack at Line 2

RIP of `m`
(1)
(2)
(3)
SFP of `q`
(4)

Assumptions:

- All memory safety defenses are disabled.
- You run GDB and break before executing line 3. You find that the RIP of `q` has the value `0xffffd6c0`.
- You also find that `goldfinger` is located at address `0xffffd620`.
- Recall that the byte `0x90` is also known as the NOP (no-operation) instruction.
- Your goal is to execute the 4-byte-long SHELLCODE.

Q7.1 (2 points) What values go in blanks (1) through (4) in the stack diagram above?

○ (1) SFP of `m`	(2) RIP of `q`	(3) `spectre`	(4) `goldfinger`
● (1) SFP of `m`	(2) `spectre`	(3) RIP of `q`	(4) `goldfinger`
○ (1) RIP of `q`	(2) `spectre`	(3) SFP of `m`	(4) `goldfinger`
○ (1) RIP of `q`	(2) `goldfinger`	(3) `spectre`	(4) SFP of `m`

Solution: The stack diagram:

`0xffffd638`	`[4]`	RIP of `m`
`0xffffd634`	`[4]`	SFP of `m`
`0xffffd630`	`[4]`	`spectre`
`0xffffd62c`	`[4]`	RIP of `q`
`0xffffd628`	`[4]`	SFP of `q`
`0xffffd620`	`[8]`	`goldfinger`

Q7.2 (4 points) Which of these values does the exploit have to overwrite to execute `SHELLCODE`? Select all that apply.

- [x] `goldfinger`
- [x] Least significant byte of the RIP of `q`
- [x] SFP of `q`
- [ ] SFP of `m`
- [ ] `spectre`
- ◯ None of the above

Q7.3 (4 points) Provide an input to the `fread` on Line 3.

If a part of the input can be any non-zero value, use `'A' * n` to represent `n` bytes of garbage.

```
'A' * 8 + SHELLCODE + '\x20'
```

Solution: `fread` gives us 13 bytes to write, with which we will need to:

- overwrite `goldfinger` with 8 bytes of grab to reach the SFP of `q`
- Then we place our 4-byte `SHELLCODE`
- Finally we overwrite the LSB of the RIP of `q` so that we can point it somewhere into `goldfinger` or directly to the `SHELLCODE`. Any byte choice between `\'x20'` and `\'x28'` would have the RIP of `q` pointing either into `goldfinger` (which recall is a no-op sled thanks to the for loop beginning on line 4) or directly at `SHELLCODE`, which was placed at the address of the SFP of `q` (`0xffffd628`).

We cannot place `SHELLCODE` into `goldfinger`, as that same for loop would corrupt the bytes of `SHELLCODE`, leaving us with no choice but to place it right above `goldfinger`.

Q7.4 (2 points) Which memory safety defenses would cause the correct exploit from Q7.3 (without modifications) to fail? Consider each choice independently.

- [x] Stack canaries
- [x] Non-executable pages
- ◯ None of the above

Solution: Stack canaries would foil this exploit in two ways:

1. They would add 4 bytes onto the stack in between `goldfinger` and the SFP of `q`, meaning that we can no longer fully overwrite the SFP or overwrite the LSB of the RIP.
2. The intended exploit would also overwrite the value of the canary, causing the program to abort

Seeing as we are writing our exploit onto the stack, setting the stack to non-executable (through non-executable pages) would also result in us being unable to execute the instructions of `SHELLCODE`.

Q7.5 (3 points) Which **values** of the RIP of `q` would cause the correct exploit from Q7.3 (without modifications) to fail? Select all that apply.

- ☐ `0xffffd640`
- ☐ `0xffffd610`
- ■ `0xffffdbd0`
- ☐ `0xffffd630`
- ☐ `0xffffd600`
- ■ `0xffffdbc0`
- ☐ `0xffffd620`
- ■ `0xffffdbe0`
- ○ None of the above

Solution: Seeing as we can only overwrite the LSB of the RIP of `q`, we need the three most-significant bytes of the value the RIP to match the address of `goldfinger` (which is `0xffffd620`). Thus any value of the RIP of `q` which is not in the form `0xffffd6__` would cause the program to fail.

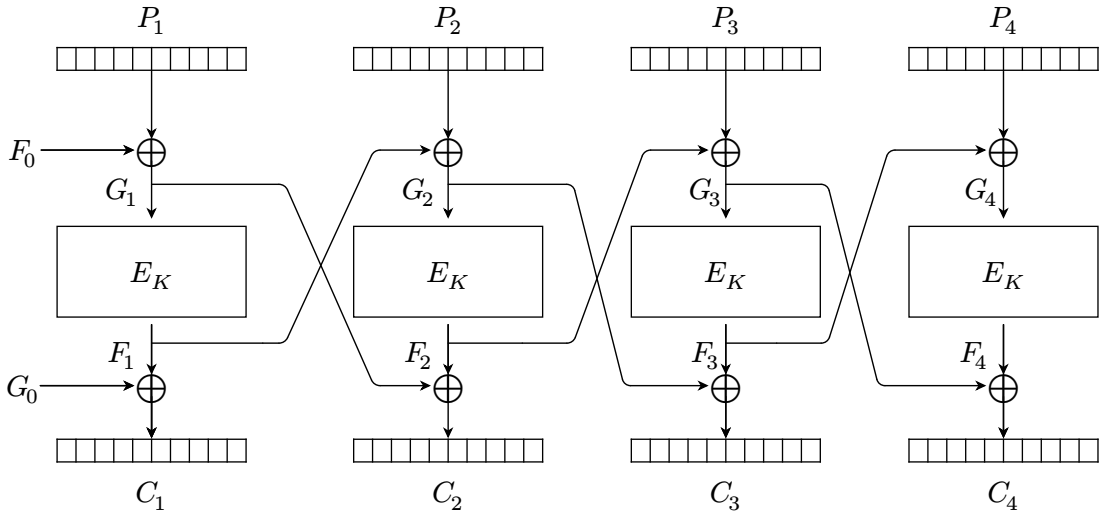
Q7.6 (4 points) When does the correct exploit from Q7.3 start executing instructions in `SHELLCODE`?

- ○ When the function `fread` on line 3 returns
- ○ When the `for` loop on line 4 completes
- ● When the function `q` on line 1 returns
- ○ When the function `m` on line 8 returns

Solution: The exploit works by overflowing `goldfinger` to overwrite RIP of `q` on the stack. This corrupted return address now points to the location of the goldfinger buffer (which the for loop conveniently turns into a NOP sled) leading to the instructions of `SHELLCODE` (which was placed just above `goldfinger`) to be executed.

Q8 *Cryptography: The Mumbling-Jumbling Block Cipher* **(10 points)**

EvanBot creates a new block cipher mode of operation. The encryption formula requires using two IVs: $G_0 = IV_0$ and $F_0 = IV_1$. Given a message $M = (P_1, P_2, P_3)$, the sender appends one more block $P_4 = 0$, which is used for integrity checking, then encryption proceeds as shown below:



$$G_i = P_i \oplus F_{i-1} \qquad F_i = E_K(G_i) \qquad C_i = F_i \oplus G_{i-1}$$

In this entire question, assume that all IVs are independently randomly generated.

Q8.1 (3 points) What formulas should we use to decrypt the ciphertexts?

○	$F_i = C_{i-1} \oplus G_{i-1}$	$G_i = D_K(F_i)$	$P_i = G_i \oplus F_{i+1}$
○	$F_i = C_{i+1} \oplus G_{i-1}$	$G_i = D_K(F_{i+1})$	$P_i = G_{i+1} \oplus F_i$
●	$F_i = C_i \oplus G_{i-1}$	$G_i = D_K(F_i)$	$P_i = G_i \oplus F_{i-1}$

Solution: Algebraically reverse the encryption formulas provided above to solve for P_i.

Deriving Decryption:

1. Given that $C_i = F_i \oplus G_{i-1}$, isolate F_i to get $F_i = C_i \oplus G_{i-1}$.
2. Given that $F_i = E_K(G_i)$, decrypt $F_i \Rightarrow G_i = D_K(F_i)$
3. Finally, given that $G_i = P_i \oplus F_{i-1}$, isolate $P_i \Rightarrow P_i = G_i \oplus F_{i-1}$

These three derived formulas exactly match option **C**.

Q8.2 (3 points) Is this scheme IND-CPA-secure?

- ● Yes, no attacker can win the IND-CPA game with a probability greater than $\frac{1}{2}$.
- ○ Yes, because no attacker can forge the ciphertexts without being detected.
- ○ No, the last block of the plaintext, P_4, allows for attackers to distinguish between M_1 and M_2.
- ○ No, because the IVs are non-deterministic.

Solution: IND-CPA does not guarantee integrity/authenticity. IVs need to be random and non-deterministic for a scheme to be IND-CPA secure. The IND-CPA game allows for the attacker to choose the two messages and guarantees that for any two known messages, the ciphertexts are indistinguishable.

When receiving a ciphertext, the recipient decrypts it (using the correct method from Q8.1), then verifies integrity by checking that $P_4 = 0$. If $P_4 \neq 0$, the recipient ignores the ciphertext (i.e. it is invalid). Otherwise, the recipient accepts the decrypted message.

Q8.3 (4 points) Does this scheme provide authenticity?

- ○ Yes, this is exactly Encrypt-then-MAC with CBC mode as the encryption scheme.
- ○ Yes, an attacker who observes the encryption of either M_1 or M_2 cannot guess which was encrypted.
- ○ No, because an attacker can always recover the secret key k from a ciphertext and reconstruct a new ciphertext that way.
- ● No, if a message has a 0 block, an attacker could truncate the ciphertext after that without being detected.
- ○ No, if Alice encrypts $M = (P_1)$ with $P_1 = 0$ and the attacker observes the corresponding ciphertext $C = (C_1, C_2)$, then sending the ciphertext $C' = (C_1, C_2, C_1, C_2)$ will decrypt to the message $M' = (0, 0, 0)$ and the recipient will accept the decrypted message.

Solution: Suppose Alice encrypts the message $M = (P_1, P_2, P_3)$ such that P_2 happens to be zero. The mode of operation will append $P_4 = 0$, and encrypt to get the ciphertext $C = (C_1, C_2, C_3, C_4)$ with IVs, F_0 and G_0.

Now, suppose that Mallory observes this ciphertext and truncates it to get the truncated ciphertext $C' = (C_1, C_2)$ with IVs, F_0 and G_0.

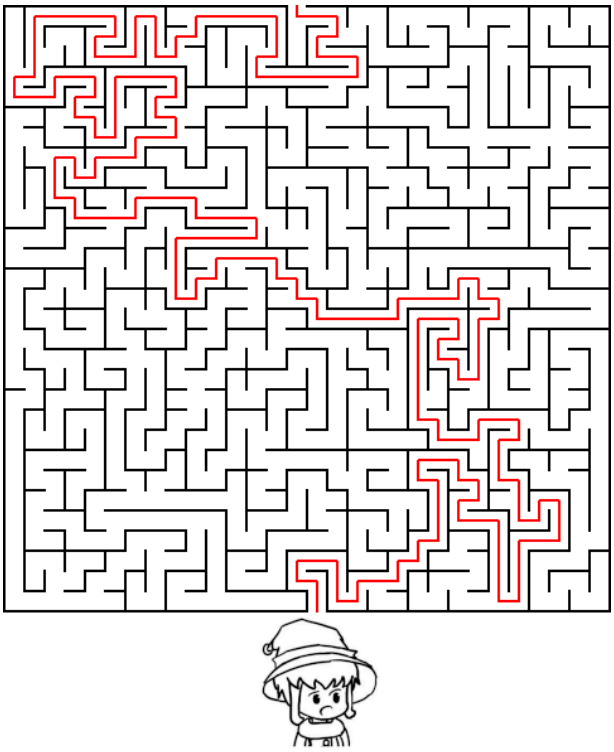
When Bob receives C' and decrypts it, he will obtain $P'_1 = P_1$ and $P'_2 = P_2 = 0$. Then, Bob will check that the last block of P is equal to zero...which it is. Therefore, Bob will accept this as a valid decryption, and receive the message $M' = (P_1)$.

In other words, Bob will think that Alice sent $M' = (P_1)$ when that's not actually what Alice sent.

(Question 8 continued...)

Post-Exam Activity: The Bot and the Maze

Help Evanbot get through the maze!



Comment Box

Congratulations for making it to the end of the exam! Feel free to leave any final thoughts, comments, feedback, or doodles here: