CS 161 Fall 2025 — Introduction to Computer Security — Final

Name: _____

Student ID: _____

This exam is 170 minutes long. There are 11 questions of varying credit. (100 points total)

Question:	1	2	3	4	5	6	7	8	9	10	11	Total
Points:	0	8	8	8	9	7	9	16	11	12	12	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: Caesar Cipher (0 points):

EvanBot wishes to share the following message, but doesn't want it to fall into the wrong hands. Decipher away!

Hint: Shift +3

J	→	
R	→	
R	→	
G	→	
O	→	
X	→	
F	→	
N	→	

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* 🍲 **(8 points)**

Q2.1 (0.5 points) True or False: The principle of "Least Privilege" dictates that a program or user should only be granted the specific permissions required to perform their intended task and nothing more.

● True ○ False

Solution: This is true. The principle of Least Privilege explicitly states that you should consider what permissions an entity needs to do its job correctly. Granting unnecessary permissions is dangerous because a malicious or compromised program could exploit those extra privileges against you.

Q2.2 (0.5 points) True or False: Using GDB, if you want to inspect the **contents** of the buffer `buf` in the code to the right, you can break at line 7 and then run the command `x/16x buf`.

○ True ● False

Solution: False. `buf` is a local variable inside `func`. At line 6 (inside `vulnerable`), `buf` is out of scope and likely does not even exist on the stack yet. You would need to break inside `func` (e.g., line 3) to inspect `buf`.

C code:

```
1  void func(){
2    char buf[16];
3    ...
4  }
5
6  void vulnerable(){
7    ...
8    func();
9  }
```

You run the code in GDB, break at line 7, and run `info frame`. You receive the output to the right.

GDB Output:

```
(gdb) info frame
...
Saved registers:
ebp at 0xffffdc58,
eip at 0xffffdc5c
```

Q2.3 (0.5 points) True or False: The RIP of `vulnerable()` is at the address `0xffffdc5c`.

● True ○ False

Solution: The GDB output indicates that `eip` was saved at address `0xffffdc5c`; the RIP is the saved value of `eip`.

Q2.4 (0.5 points) True or False: The value of the SFP of `func()` is `0xffffdc58`.

● True ○ False

Solution: The GDB output indicates that `ebp` was saved at address `0xffffdc58`, i.e., that the SFP of `vulnerable` is stored at address `0xffffdc58`. What about the SFP of `func`? The SFP of `func` is stored at a different address (lower on the stack), and it points to the SFP of its caller, i.e., it points to the SFP of `vulnerable`—so the value of the SFP of `func` must be `0xffffdc58`.

Q2.5 (0.5 points) True or False: Format string vulnerabilities allow an attacker to read data from the stack, but they cannot be exploited to write to memory or execute shellcode.

○ True ● False

Solution: When an attacker controls the format string, they can use the `%n` identifier to write to specific memory addresses. By overwriting targets like the RIP, this vulnerability can be exploited to execute arbitrary shellcode.

Q2.6 (0.5 points) True or False: Bear Systems modifies ASLR: instead of randomly generating the memory offset when the program starts, they randomly generate the memory offset when the program is compiled and hardcode this offset into the binary. Compared to standard randomized ASLR, Bear Systems' modification improves security against memory safety exploits.

○ True ● False

Solution: Hardcoding the offset into the executable ensures it is the same for every user and every execution. An attacker can simply analyze their own copy of the binary to recover the offset and successfully exploit any other user, effectively defeating the purpose of ASLR.

Q2.7 (0.5 points) True or False: The HMAC algorithm is specifically designed to be secure against length extension attacks (where an attacker can compute $H(M \parallel M')$ for some M' given only $H(M)$, the length of M, and M').

● True ○ False

Solution: HMAC uses a nested hashing structure involving an inner and outer pad. This design prevents length extension attacks that affect hashes like SHA256.

Q2.8 (0.5 points) True or False: In AES-CBC mode, encryption cannot be parallelized because the encryption of block C_i depends on the ciphertext of the previous block C_{i-1}, but decryption *can* be parallelized.

● True ○ False

Solution: CBC decryption uses the formula $P_i = D_K(C_i) \oplus C_{i-1}$. Since all ciphertext blocks C are available immediately upon receipt, each plaintext block can be computed independently.

Q2.9 (0.5 points) True or False: If an attacker intercepts a Diffie-Hellman key exchange where Alice sends $A = g^a \bmod p$ and Bob sends $B = g^b \bmod p$, the attacker can easily compute the shared secret S if they can solve the Discrete Logarithm Problem.

● True ○ False

Solution: The security of Diffie-Hellman relies on the hardness of the Discrete Logarithm Problem. If an attacker can solve for a given A (or b given B), they can compute the shared secret $S = g^{ab} \bmod p$.

Q2.10 (0.5 points) True or False: El Gamal encryption is deterministic; if you encrypt the same message M twice with the same public key, you will always generate the exact same ciphertext (C_1, C_2).

○ True ● False

Solution: El Gamal encryption is randomized. It uses a random nonce r for every encryption, resulting in different ciphertexts (g^r, $M \cdot g^{ab}$) for the same message. This is necessary for IND-CPA security.

Q2.11 (0.5 points) True or False: The "Same-Origin Policy" allows a script loaded by `http://example.com` to read the properties of a document from `https://example.com` because they share the same domain name and the protocol doesn't need to match.

○ True ● False

Solution: The Same-Origin Policy defines an origin by the tuple (Protocol, Domain, Port). `http` and `https` are different protocols, so these are treated as different origins.

Q2.12 (0.5 points) True or False: If a website sets a cookie with `Domain=.example.com`, that cookie will be sent by the browser in requests to both `www.example.com` and `secure.example.com`.

● True ○ False

Solution: Cookie domains match parent domains. A cookie set for `.example.com` is valid for any subdomain of `example.com`.

Q2.13 (0.5 points) True or False: In a TCP handshake, if the Initial Sequence Number (ISN) is generated randomly with a cryptographically secure pseudorandom number generator, this will prevent off-path attackers from easily injecting packets into the connection by guessing the sequence number.

● True ○ False

Solution: Off-path attackers cannot see the traffic and must guess the sequence numbers. Using random ISNs maximizes the difficulty of this guess (1 in 2^{32}).

Q2.14 (0.5 points) True or False: DNSSEC with NSEC prevents "zone walking" (enumerating all valid domain names in a zone) by using NSEC records that return a cryptographic hash of the queried domain name rather than the name itself.

○ True ● False

Solution: NSEC records return the next valid domain name in cleartext, enabling zone walking. NSEC3 makes zone walking harder by hashing domain names, but NSEC does nothing to prevent zone walking and in fact makes zone walking easy.

Q2.15 (0.5 points) True or False: The BGP (Border Gateway Protocol) includes built-in cryptographic verification to ensure that an Autonomous System (AS) actually owns the IP prefixes it advertises, preventing prefix hijacking attacks by default.

○ True ● False

Solution: BGP operates largely on trust and does not have built-in cryptographic mechanisms to verify prefix ownership, making it vulnerable to route hijacking.

Q2.16 (0.5 points) True or False: The Kaminsky DNS attack allows an attacker to poison the DNS cache of a recursive resolver by flooding it with spoofed responses for non-existent subdomains (e.g., `1.google.com`, `2.google.com`), aiming to overwrite the authority records for the target domain (e.g., `google.com`).

● True ○ False

Solution: The Kaminsky attack bypasses the TTL wait time by querying random non-existent subdomains. The goal is to get a spoofed response accepted which includes malicious authority (NS) records for the target zone.

Q3 *Memory Safety: Tomayto* 🍅 **(8 points)**

Consider the following vulnerable C code:

```
1  void tomayto(int count, char *input) {
2      char buffer[32];
3      if (count > 32) { return; }
4      memcpy(buffer, input, count);
5  }
6
7  void main() {
8      int user_int = 0;
9      char user_string[40];
10     fread(user_string, 4, 11, stdin);
11     tomayto(user_int, user_str);
12 }
```

Stack at Line 5

RIP of `main`
SFP of `main`
(1)
(2)
(3)
(4)
RIP of `tomayto`
SFP of `tomayto`
`buffer`

- All memory safety defenses are disabled.
- You run GDB, set a breakpoint at line 4, run and find that `buffer` starts at address `0x50ffd100`.
- You run GDB, set a breakpoint at line 4, run and find that the RIP of `tomayto` has the value `0x08048999`.
- Your goal is to execute the 32-byte long `SHELLCODE`.

(0.5 points each) What goes in the blanks in the stack diagram above?

Q3.1 Blank (1): Ⓐ `count` Ⓑ `user_string` ● `user_int` Ⓓ `input`

Q3.2 Blank (2): Ⓐ `count` ● `user_string` Ⓒ `user_int` Ⓓ `input`

Q3.3 Blank (3): Ⓐ `count` Ⓑ `user_string` Ⓒ `user_int` ● `input`

Q3.4 Blank (4): ● `count` Ⓑ `user_string` Ⓒ `user_int` Ⓓ `input`

Solution:

Address	Content	
`0x50ffd160`	[4]	RIP of `main`
`0x50ffd15c`	[4]	SFP of `main`
`0x50ffd158`	[4]	`user_int`
`0x50ffd130`	[40]	`user_string`
`0x50ffd12c`	[4]	`input`
`0x50ffd128`	[4]	`count`
`0x50ffd124`	[4]	RIP of `tomayto`
`0x50ffd120`	[4]	SFP of `tomayto`
`0x50ffd100`	[32]	`buffer`

Q3.5 (1 point) Which of the following memory safety vulnerabilities are present in the above code?

- [A] Format String Vulnerability
- [■] Stack Buffer Overflow
- [E] Heap Overflow
- [■] Signed/Unsigned
- [D] ret2ret
- [F] None of the above

Solution: The variable `count` is a signed integer (`int`). The check `count > 32` effectively checks if `count` is positive and greater than 32.

However, `memcpy` takes a `size_t` for the length, which is `unsigned`. If we pass a negative number (e.g., `-1`), the check `-1 > 32` is False, so we proceed.

Then, `memcpy` interprets `-1` as `0xFFFFFFFF` (a huge unsigned number), causing a massive copy that overflows `buffer`.

Q3.6 (4 points) Provide an input to `fread` on Line 10 that would cause the program to execute shellcode. If a part of the input can be any non-zero value, use `'A' * n` to represent `n` bytes of garbage.

Don't worry about segfaults that could possibly occur during the `memcpy` (all memory is mapped in). If you weren't worried about that, please ignore this remark.

```
SHELLCODE + 'A'*4 + '\x00\xd1\xff\x50' +
'\x00\x00\x00\x80'
```

Solution: We use the first 40 bytes to change the value of `user_string` to:

1. `SHELLCODE` to fill up `buffer`
2. 4 bytes of garbage for the SFP of `tomayto`
3. The address of `buffer` to overwrite the RIP of `tomayto`

Next, we use the final 4 bytes of the `fread` call to overwrite the value of `user_int` to a negative value so that we can pass the size check on line 4, until it is cast as a large unsigned integer to `memcpy` on line 7. We'll accept any negative number here.

You could also use `\x30\xd1\xff\x50` (the address of `user_input`) instead of `\x00\xd1\xff\x50` (the address of `buffer`), as after the `memcpy` both will contain a copy of SHELLCODE.

Q3.7 (1 point) Which changes need to be made to make this code memory-safe?

- [■] Line 1: Change `int count` to `size_t count;`
- [B] Line 3: Change `count > 32` to `count >= 32`
- [C] Line 4: Add `input[31] = '\0';` before the `memcpy` on line 5
- [D] Line 8: Change `int user_int = 0;` to `int user_int = -1;`

Solution: Changing `count` to an unsigned type (`size_t` or `unsigned int`) ensures that the check `if (count > 32)` correctly handles large numbers. If the attacker passes a "negative" bit pattern (e.g., `0xFFFFFFFF`), the comparison `count > 32` will interpret both values as unsigned integers, see `0xFFFFFFFF` as a huge number, it will be greater than 32, and the function will return safely.

If `count` is exactly 32, then it's fine to pass 32 to `memcpy`; it will not write past the end of `buffer`.

`memcpy` is not interrupted by null bytes. There is no need for the data to be null-terminated.

Q4 *Memory Safety: I've played these games before...* 🍅 **(8 points)**

Consider the following vulnerable C code:

```
1  void tomahto() {
2      char cage[12];
3      fgets(cage, 12, stdin);
4      printf(cage);
5      gets(cage);
6  }
7
8  void main() {
9      tomahto();
10 }
```

Stack at Line 6

RIP of `main`
SFP of `main`
(1)
RIP of `tomahto`
(2)
(3)
(4)

- Stack canaries are enabled. All other memory safety mitigations are disabled.
- You run GDB, set a breakpoint at line 5, run and find that `cage` starts at address `0xffffd100` and that there is a copy of `SHELLCODE` at address `0xffffd204`.
- Through trial and error, you discover that the stack canary for `tomahto` is the 4th value `printf` reads from the stack when it looks for arguments (i.e., it is offset 4 words away from the stack pointer `printf` uses).

(0.25 points each) What goes in the blanks in the stack diagram above?

Q4.1 Blank (1): Ⓐ `cage` Ⓑ SFP of `tomahto` Ⓒ canary of `tomahto` ● canary of `main`

Q4.2 Blank (2): Ⓐ `cage` ● SFP of `tomahto` Ⓒ canary of `tomahto` Ⓓ canary of `main`

Q4.3 Blank (3): Ⓐ `cage` Ⓑ SFP of `tomahto` ● canary of `tomahto` Ⓓ canary of `main`

Q4.4 Blank (4): ● `cage` Ⓑ SFP of `tomahto` Ⓒ canary of `tomahto` Ⓓ canary of `main`

Solution: The stack diagram:

`0xffffd120`	`[4]`	RIP of `main`
`0xffffd11c`	`[4]`	SFP of `main`
`0xffffd118`	`[4]`	canary of `main`
`0xffffd114`	`[4]`	RIP of `tomahto`
`0xffffd110`	`[4]`	SFP of `tomahto`
`0xffffd10c`	`[4]`	canary of `tomahto`
`0xffffd100`	`[12]`	`cage`

The canary is always placed between the local buffers and the SFP to detect overflows before they corrupt the return address.

Q4.5 (1 point) Which of the following memory safety vulnerabilities are present in the above code?

■ Format String Vulnerability ■ Stack Buffer Overflow Ⓔ Heap Overflow

Ⓑ Signed/Unsigned Ⓓ ret2ret Ⓕ None of the above

Solution: Line 4 (`printf(cage)`): The user controls the first argument to `printf`. This is a Format String vulnerability that allows reading from the stack (leaking data).

Line 5 (`gets(cage)`): This allows us to write as many bytes into `cage` as we'd like, which is only 12 bytes long. This allows overwriting the stack (Buffer Overflow).

Q4.6 (2 points) Which of these inputs to `fgets` on Line 3 will always leak the value of the stack canary in the `tomahto` stack frame? Select all that apply.

Note: Stack canaries are four random bytes and do not contain a null byte.

■ `'%x' * 4`
■ `('%c' * 3) + '%x'`
Ⓔ `'%n' * 4`
Ⓑ `'%x' * 3`
Ⓓ `'%x' + ('%s' * 3)`
Ⓕ None of the above

Solution: We need `printf` to look up the stack to find the canary of `tomahto`. From our stack diagram, we know that `printf` will start looking for arguments at address `0xffffd100`, and the canary of `tomahto` is the 4th word on the stack relative to `printf`'s arguments.

1. `'%x' * 4`: This prints the first 4 values on the stack in hex (the 12 bytes of cage, and the canary). The 4th value printed is the canary.
2. `('%c' * 3) + '%x'`: The three `%c` specifiers consume the first 3 arguments on the stack (the 12 bytes of cage), printing them as characters. The following `%x` consumes the 4th argument (the canary), printing the canary in pointer format (hex).

Incorrect options: `%n` would attempt to write to the memory address, modifying the canary instead of leaking it. `%s` would attempt to treat the canary value as an address and dereference it, likely crashing the program, but in any case not printing the value of the canary. `'%x' + ('%s' * 3)` treats the 4th argument—the canary—as an address (`%s`) and tries to print memory starting at that address; but the canary is not an address and printing memory at that address probably won't help deduce the value of the canary itself.

In the next part, provide an input to `gets` on line 5 that would cause the program to execute `SHELLCODE`, assuming the correct input has been provided to `fgets` on line 3. You may use `CANARY` to refer to the correct 4-byte string value of the stack canary, as leaked by `printf`.

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

Q4.7 (4 points) Input to `gets` on line 5:

```
'A' * 12 + CANARY + 'B' * 4 + '\x04\xd2\xff\xff'
```

Solution: We are performing a stack buffer overflow. The memory layout is:
`[ 12 bytes cage ] + [ 4 bytes Canary ] + [ 4 bytes SFP ] + [ 4 bytes RIP ]`

1. Fill the buffer: We write 12 bytes of garbage (`'A' * 12`) to fill `cage`.
2. Restore the canary: We must overwrite the canary location with its original value (`CANARY`) so that the function `tomahto` does not crash when it checks the canary before returning.
3. Overwrite SFP: We write 4 bytes of garbage (`'B' * 4`) to write over the SFP of `tomahto`.
4. Overwrite RIP: We write the address of our `SHELLCODE` (`0xffffd204`) to replace the RIP.

Q5 *Cryptography: Secret Santa* 🎅 **(9 points)**

Annabella and Fred want to establish a secure communication channel. They require a protocol that supports asynchronous communication (Annabella can send a message even if Fred is offline, i.e., even if Fred is not connected to the Internet at that moment), mutual authentication, and forward secrecy.

Q5.1 (2 points) Why is a basic, unauthenticated Diffie-Hellman exchange vulnerable to Man-in-the-Middle (MITM) attacks?

Ⓐ The discrete logarithm problem is easier to solve when values are intercepted.

Ⓑ Diffie-Hellman keys are too short to resist brute-force attacks.

● The public keys exchanged are not cryptographically bound to the users' identities.

Ⓓ Servers cannot store Diffie-Hellman public values.

Solution: In basic DH (g^a, g^f), the values are just random integers. Annabella has no way of verifying that the value g^f actually came from Fred; an attacker (Mallory) can intercept Annabella's g^a, send her own g^m, and establish a key with Annabella while pretending to be Fred.

Q5.2 (1 point) Annabella and Fred decide to simply publish static (long-term) Diffie-Hellman (DH) public keys to a single, central, trusted server. They use these same keys to derive a shared secret for every message they ever send.

Why does this approach fail to provide forward secrecy?

Ⓐ Static DH outputs are deterministic and therefore predictable by random guessing.

Ⓑ The server must regenerate the group parameters for each session.

● If a long-term private key is stolen later, the attacker can decrypt all past recorded traffic.

Ⓓ Public keys expire too quickly to be useful.

Solution: Forward Secrecy means that if long-term keys are leaked or revealed to the attacker, this does not compromise past session keys. If g^a and g^f are static (fixed forever), the shared secret g^{af} is always the same. If an attacker records the ciphertext and steals a years later, they can compute g^{af} from g^f (which was sent unencrypted over the network and could have been recorded by the attacker) and a (which the attacker now knows) and decrypt everything.

The Protocol

To solve these issues, Annabella and Fred adopt a new scheme:

1. Fred uploads keys: Fred generates the following and uploads the **public** parts to a central trusted server:

- Identity Key (IK_F): Long-term static key pair.
- Signed Pre-Key (SPK_F): A medium-term key pair signed by IK_F.
- One-Time Pre-Keys ($\text{OPK}_F[i]$): A batch of key pairs intended to be used once and deleted. Not signed.

2. Annabella fetches keys: If Annabella wants to message Fred, she fetches Fred's IK_F, SPK_F, and a single $\text{OPK}_F[i]$ from the server; verifies the

signature, and generates a fresh Ephemeral Key pair (EK_A).

3. Key derivation: Annabella computes the shared secret SK by combining four Diffie-Hellman (DH) calculations:

1. $K_1 = \text{DH}(\text{IK}_A, \text{SPK}_F)$
   (Binds Annabella's Identity to Fred's Signed Key)
2. $K_2 = \text{DH}(\text{EK}_A, \text{IK}_F)$
   (Binds Session to Fred's Identity)
3. $K_3 = \text{DH}(\text{EK}_A, \text{SPK}_F)$
   (Binds Session to Fred's Signed Key)
4. $K_4 = \text{DH}(\text{EK}_A, \text{OPK}_F[i])$
   (Provides Strong Forward Secrecy)

$$\text{SK} = \mathsf{H}(K_1 \parallel K_2 \parallel K_3 \parallel K_4)$$

Q5.3 (2 points) The calculation includes the term $\text{DH}(\text{IK}_A, \text{SPK}_F)$. Why does this term specifically provide assurance to Annabella that the recipient is actually Fred?

- (●) Because Annabella verified the signature on SPK_F, she knows only the holder of Fred's private Identity Key could have authorized it.
- (B) Because Annabella's Identity Key (IK_A) is included, Fred automatically knows who sent the message.
- (C) Because DH values are universally unique, no one else could generate this specific integer.
- (D) Because the server performs a Zero-Knowledge Proof to validate the Pre-Key before storage.

Solution: The SPK_F is signed by Fred's long-term identity key IK_F. When Annabella verifies this signature, she knows SPK_F belongs to Fred. Therefore, any resulting shared secret derived from SPK_F can only be computed by the person holding the private key for SPK_F (which is Fred).

Q5.4 (1 point) Which architectural feature specifically enables asynchronous communication (Annabella sending a message while Fred is offline)?

- (A) The use of symmetric Key Derivation Functions (KDF).
- (B) The requirement for Annabella to sign her own messages.
- (C) The inclusion of One-Time Pre-Keys for forward secrecy.
- (●) The use of a server to store Fred's pre-published public keys.

Solution: Asynchrony requires that Annabella can generate a full shared secret without an interactive handshake with Fred. By fetching pre-published keys ($\text{IK}, \text{SPK}, \text{OPK}$) from the server, she can complete the DH math alone.

Q5.5 (1 point) Why couldn't Annabella and Fred just use a standard, ephemeral Diffie-Hellman handshake to achieve asynchronous messaging?

Ⓐ Fred's computer cannot generate DH keys when it is not connected to the Internet.

● Ephemeral DH requires both parties to be online simultaneously to exchange values.

Ⓒ Servers are technically incapable of storing DH integers.

Ⓓ Standard DH is too computationally expensive for mobile devices.

Solution: Standard ephemeral DH is interactive: Annabella sends g^a, Fred receives it and sends g^f. If Fred is offline, he cannot receive g^a nor generate/send g^f, preventing the handshake from completing.

Q5.6 (1 point) Consider a simplified protocol that **only** computes $\text{SK} = \text{DH}(\text{EK}_A, \text{IK}_F)$.

Which security properties are **missing** from this specific exchange? Select all that apply.

☐ A Confidentiality against passive eavesdroppers.

■ Authentication of Annabella (to Fred).

☐ C Authentication of Fred (to Annabella).

■ Forward Secrecy.

Ⓔ None of the above

Solution:

1. Forward Secrecy is missing: IK_F is static. If stolen, past messages can be decrypted.
2. Authentication of Annabella is missing: EK_A is random and anonymous. Fred knows someone sent a message, but not who. Anyone could have chosen their own EK, computed a SK, encrypted their message, and claimed to be Annabella.
3. Authentication of Fred is present: Annabella uses IK_F, so she knows she is encrypting for Fred, and only Fred will be able to compute SK and decrypt.
4. Confidentiality is present: A passive observer sees only public keys and cannot compute the secret, thanks to the use of Diffie-Hellman.

Q5.7 (1 point) Fred modifies the scheme to publish only an identity key IK_F and a batch of one-time pre-keys OPK_F, but not SPK_F (no signed pre-key is published). The secret key is computed as $\text{SK} = \mathsf{H}(K_2 \parallel K_4)$.

Select all the security guarantees/properties this modified scheme provides.

- [A] Authentication of Annabella (Fred can verify he is speaking with Annabella)
- [■] Authentication of Fred (Annabella can verify she is speaking with Fred)
- [■] Forward secrecy (as long as unused one-time pre-keys remain)
- [■] Asynchronous communication (only while unused one-time pre-keys remain)
- [E] None of the above

Solution: As long as unused OPK_F from Fred remain, the protocol provides forward secrecy and asynchronous communication.

Fred's identity key IK_F is still used to sign his public one-time pre-keys OPK_F, so Annabella knows she is deriving a key with Fred and not an attacker.

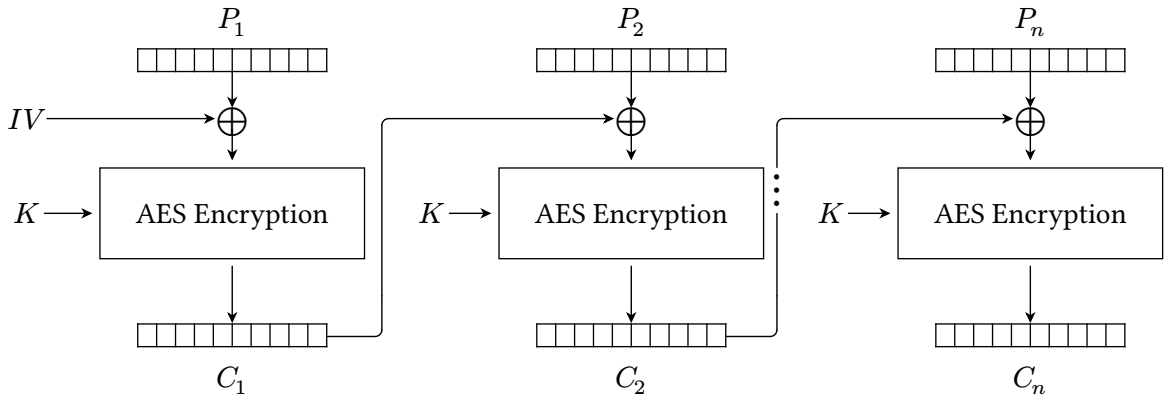
SK does not depend on Annabella's private key from IK_A. Annabella does not have to prove her identity by using some secret only she knows; the computation of SK could be done by anyone. So there is no guarantee that Fred is talking to Annabella, as opposed to an imposter falsely claiming to be Annabella.

Because both Annabella's ephemeral key and Fred's one-time pre-key are deleted after use, past session keys remain secure even if long-term keys are compromised, providing forward secrecy. An attacker who steals the private keys corresponding to IK_F or IK_A is still not able to compute SK for past communications.

However, each asynchronous session consumes a one-time pre-key. Once Fred runs out of these keys, Annabella has no authenticated public key left to start a session while Fred is offline, which is why the signed pre-key exists in the original protocol. Therefore, asynchronous communication is only guaranteed while there exists unused OPKs.

Q6 *Cryptography: Double Dipping* 🧁 **(7 points)**

Recall the implementation of AES-CBC encryption:



- Alice uses AES-CBC encryption to encrypt the plaintext $P = P_1 \parallel P_2 \parallel P_3$. She sends the corresponding ciphertext $C = C_1 \parallel C_2 \parallel C_3$ to Bob.
- Alice and Bob use a key K_1.

Q6.1 (2 points) Under AES-CBC, which of the following are the correct value for C_3? Select all that apply.

- [A] $C_3 = E_{K_1}(C_2)$
- [B] $C_3 = E_{K_1}(C_2 \oplus P_3)$
- [C] $C_3 = E_{K_1}(P_3 \oplus E_{K_1}\left(P_2 \oplus E_{K_1}(P_1 \oplus IV)\right)$
- [D] $C_3 = E_{K_1}(P_1 \oplus P_2 \oplus P_3 \oplus IV)$
- [E] $C_3 = E_{K_1}(P_1 \oplus P_2 \oplus P_3)$
- [F] $C_3 = E_{K_1}(C_2 \oplus P_3) \oplus IV$

Solution: $C_i = E_{K_1}(C_{i-1} \oplus P_i)$, so option B is correct. Also, since $C_3 = E_{K_1}(C_2 \oplus P_3)$ and $C_2 = E_{K_1}(C_1 \oplus P_2)$, we obtain $C_3 = E_{K_1}\left(P_3 \oplus E_{K_1}(C_1 \oplus P_2)\right)$, and then plugging in $C_1 = E_{K_1}(P_1 \oplus IV)$, we obtain option C.

Q6.2 (1.5 points) Mallory wants to manipulate the message. She flips the **first bit** of the IV. She leaves all ciphertext blocks (C_1, C_2, C_3) unchanged.

What happens to the decrypted plaintext P'?

- [A] The first bit of P_1' is flipped; all other blocks are correct.
- [B] The whole block P_1' is garbled (randomized); all other blocks are correct.
- [C] The first bit of P_1' is flipped, and P_2' is completely garbled.
- [D] The decryption fails completely due to padding errors.

Solution: In CBC decryption for the first block, $P_1 = D_{K_1}(C_1) \oplus IV$. If we flip a bit in the IV, that error propagates directly through the XOR into P_1 at the exact same position. Since IV is not used for P_2 or P_3, the rest of the message is intact.

Q6.3 (1.5 points) Alternatively, suppose Mallory flips the **last bit** of ciphertext block C_1. She leaves IV, C_2, C_3 unchanged.

What is the specific effect on the decrypted plaintext blocks P_1' and P_2'?

Ⓐ Both P_1' and P_2' have their last bits flipped.

Ⓑ P_1' has its last bit flipped; P_2' is completely garbled.

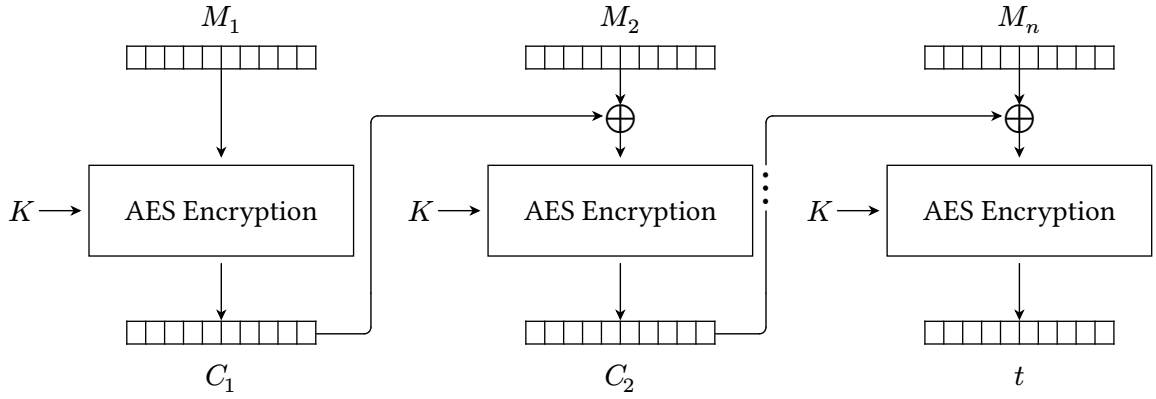
● P_1' is completely garbled; P_2' has its last bit flipped.

Ⓓ P_1' is completely garbled; P_2' is unaffected.

Solution:

1. Effect on P_1: Bob computes $P_1 = D_{K_1}(C_1) \oplus IV$. Since C_1 goes into the AES decryption function, changing even one bit randomizes the output entirely (Avalanche Effect). P_1 is garbage.
2. Effect on P_2: Bob computes $P_2 = D_{K_1}(C_2) \oplus C_1$. Here, C_1 is just XORed against the result. Therefore, the specific bit flip in C_1 carries over directly to flip the same bit in P_2.

Consider the CBC-MAC scheme, which takes an input message $M = (M_1, M_2, ..., M_n)$ and key K, and outputs a tag t. The same key is used for all CBC-MAC computations in this question.



Suppose for the next two subparts:

- Recall that Alice encrypted P with K_1 and sent it to Bob. Bob will also decrypt with K_1.
- Assume that Alice used $IV = 0$ when encrypting P.
- Before Bob can decrypt, Mallory (an on-path attacker) tampers with the first 2 ciphertext blocks so that Bob receives $C' = C'_1 \parallel C'_2 \parallel C_3$. Note that C_3 and IV remain unchanged.
- Alice also computes a CBC-MAC tag t on the plaintext P, using key K_2, and sends it to Bob.
- Bob decides to compute the CBC-MAC tag t' on his tampered P' (the decryption of the tampered C') to verify the integrity of the plaintext.
- Alice and Bob use K_2 as the key for CBC-MAC, and K_1 for AES-CBC encryption.

Q6.4 (1 point) Bob decrypts C' to get P'. Which blocks of P' will be guaranteed to be the same as the corresponding blocks from P? Select all that apply.

A P'_1

B P'_2

C P'_3

● None of the above

Solution: P_1 and P_2 are different because they were tampered by Eve. Although C_3 is left untampered, the previous block has been altered ($C'_2 \neq C_2$). Because CBC decryption requires XORing with the previous ciphertext block to recover the plaintext ($P'_3 = C'_2 \oplus E_K^{-1}(C_3)$), this change to C'_2 results in a corrupted plaintext P'_3.

Q6.5 (1 point) Is it possible for $t' = t$ (i.e., the MAC remains valid despite the tampering)?

● Yes, if $K_1 = K_2$.

Ⓑ No, because C_2' has been tampered with.

Ⓒ No, because the plaintext P' is different from P.

Ⓓ No, because the attacker does not know the key K_2.

Solution: If $K_1 = K_2$, the MAC calculation on the decrypted plaintext P' is mathematically identical to re-encrypting P' via CBC mode with $IV = 0$ and keeping the last block.

1. MAC Input: The MAC calculation for the final block (using P_3') uses the previous ciphertext block C_2' as its input state. The input to the final E_K function is $P_3' \oplus C_2'$.
2. Decryption Formula: Bob decrypted the final block C_3 using:
$$P_3' = D_{K(C_3)} \oplus C_2'$$
3. Substitution and Cancellation: When Bob calculates the final MAC tag t', he substitutes the decryption result into the MAC formula:
$$t' = E_K(P_3' \oplus C_2')$$
$$t' = E_K\Big(\big(D_{K(C_3)} \oplus C_2'\big) \oplus C_2'\Big)$$

The C_2' terms cancel out because $A \oplus A = 0$:

$$t' = E_K(D_K(C_3))$$
$$t' = C_3$$

Since the original tag $t = E_{k_1}(M_3 \oplus C_2) = C_3$, if $K_1 = K_2$, then $t' = t$ and the forgery succeeds.

Q7 *Networking: The Fate of Streamify* 🎶 **(9 points)**

Nazar is live-streaming video to Zoir over Streamify, their new streaming service. To send the video, Nazar's computer sends UDP packets to Zoir's computer.

Suppose there are no security provisions: Zoir's computer accepts the UDP packet if and only if it has the correct UDP destination port. In UDP, port numbers are 16 bits.

Q7.1 (2 points) What is the probability that a single spoofed UDP packet from an off-path attacker is accepted, if it uses the correct destination IP address and if the destination port is chosen randomly?

Ⓐ 1 Ⓑ $1/2^8$ ● $1/2^{16}$ Ⓓ $1/2^{24}$ Ⓔ $1/2^{32}$ Ⓕ 0

Solution: The 16-bit UDP port must be guessed correctly.

Q7.2 (2 points) Now suppose an **on-path** attacker observes a few packets sent by Nazar, and then wants to send a new spoofed packet to Zoir. What is the best probability that the on-path attacker can have their spoofed UDP packet be accepted by Zoir?

● 1 Ⓑ $1/2^8$ Ⓒ $1/2^{16}$ Ⓓ $1/2^{24}$ Ⓔ $1/2^{32}$ Ⓕ 0

Solution: An on-path attacker can observe the UDP destination port number in packets from Nazar to Zoir, then copy that port number into the spoofed packet. No guessing is needed.

Nazar and Zoir want stronger security, so they add a 16-bit integrity tag to each packet: $t = \mathsf{SHA256\text{-}HMAC}(K,\ \texttt{packet})$, where `packet` is the rest of the packet, and K is an 8-bit secret key shared securely between Nazar and Zoir. A packet is accepted if and only if has both a correct 16-bit destination port and correct 16-bit integrity tag t.

Q7.3 (2 points) An **off-path** attacker wants to send a spoofed packet to Zoir. What is the best probability that the off-path attacker can have their spoofed UDP packet be accepted by Zoir, in this new design? Assume the attacker doesn't know the UDP port and has to guess it randomly.

Ⓐ 1 Ⓑ $1/2^8$ Ⓒ $1/2^{16}$ ● $1/2^{24}$ Ⓔ $1/2^{32}$ Ⓕ 0

Solution: The attacker can guess a random 16-bit UDP port number and guess a random 8-bit key, put that destination port number in their spoofed packet, use the key to compute the integrity tag, and put the integrity tag in their spoofed packet. If both guesses were correct, their packet will be accepted. There is a $1/2^{16}$ chance the UDP port is guessed correctly and a $1/2^8$ chance the key is guessed correctly, so the total probability is $1/2^{24}$.

This is better than guessing the 16-bit port number and 16-bit integrity tag, as that would have a success probability of $1/2^{32}$.

Q7.4 (3 points) Now suppose an **on-path** attacker observes a few packets sent by Nazar, and then wants to send a new spoofed packet to Zoir. What is the best probability that the on-path attacker can have their spoofed UDP packet be accepted by Zoir, in this new design? Assume the attacker can do any reasonable amount of computation before constructing the spoofed packet.

● 1 Ⓑ $1/2^8$ Ⓒ $1/2^{16}$ Ⓓ $1/2^{24}$ Ⓔ $1/2^{32}$ Ⓕ 0

Solution: The attacker can try all 2^8 possibilities for `K`, and check which is compatible with the observed packets from Nazar (i.e., which possibility for `K` yields the correct tag for all of those packets). That will almost surely be the correct `K`. Then the attacker can use the correct `K` to compute an integrity tag for the spoofed packet to Zoir. So, with a little computation, the attacker can infer `K`, and their spoofed packet will be accepted with near-certainty.

Q8 *Web Security: Super Query League* ⚽ **(16 points)**

EvanBot has created a fantasy football app called FantasyLeague. Each user is able to name their team, draft players, and manage lineups. The app relies on the following SQL tables:

Account	
`username`	`string`
`password`	`string`
`team_name`	`string`

Team	
`team_name`	`string`
`draft_order`	`uint32_t`
`completed`	`boolean`

Player	
`team_name`	`string`
`player_name`	`string`
`in_lineup`	`boolean`

When users create an account, they select a `team_name` that is used to represent their team. In each round of drafting, the `draft_order` determines the order in which teams get to select players. Once a team has run out of people to draft, `completed` is set to `true` and they can start playing. Each week, they set the `in_lineup` variable for some of the players on their team, and score based on the people who are in their lineup.

Q8.1 (3 points) To search for players, the app executes the following query:

```
SELECT player_name, team_name FROM Player WHERE player_name = '$search';
```

Which of the following payloads, when injected into the `$search` parameter, will allow the attacker to learn the `username` and `password` of every user in the `Account` table, if the attacker can see the results of the query?

Select all that apply.

- [A] `' OR 1=1; --`
- [■] `' UNION SELECT username, password FROM Account --`
- [C] `' UNION SELECT username, password, team_name FROM Account --`
- [■] `' UNION SELECT password, username FROM Account --`
- [E] `' UNION SELECT * FROM Account --`
- [F] `' UNION SELECT player_name, team_name FROM Player --`

Solution: The goal of this question is to extract data from a different table (`Account`) than the one the query was originally designed for (`Player`). To do this, we use the `UNION` operator. The `UNION` operator allows you to combine the results of two different `SELECT` statements into a single result set.

The injected `SELECT` statement must have the exact same number of columns as the original `SELECT` statement.

Now, turning to the options

- `' OR 1=1; --` Incorrect. This is a tautology attack. It returns all rows from the `Player` table, but it fails to retrieve data from the `Account` table.
- `' UNION SELECT username, password FROM Account --` Correct. This selects exactly 2 columns (`username`, `password`). Since 2 matches the original query's 2, the database accepts it.
- `' UNION SELECT username, password, team_name FROM Account --` Incorrect. This explicitly selects 3 columns, causing a syntax error because the column count does not match the original query.
- `' UNION SELECT password, username FROM Account --` Correct. This also selects exactly 2 columns. The database accepts it regardless of the logical order of data (it simply puts the password in the first slot and username in the second).
- `' UNION SELECT * FROM Account --` Incorrect. This selects all 3 columns from `Account`. Since 3 does not equal 2, this causes a syntax error.
- `' UNION SELECT player_name, team_name FROM Player --` Incorrect. This returns data from the `Account` table, but it doesn't reveal the username or password, so it doesn't meet the requirements of the question.

Q8.2 (3 points) Jonah's jealousy knows no bounds. He decides to ban his rival, **Fred**, from the platform entirely. To login, the app executes the following query:

```
SELECT * FROM Account WHERE username = '$username';
```

When injected into the **$username** parameter of the login query, which of the following payloads will delete the row **Fred** from the **Account** table, and only delete that row?

Select all that apply.

Note: **DELETE FROM table WHERE condition** removes rows from **table** that satisfy **condition**.

- [A] `'; DELETE FROM Account WHERE username = 'Fred';`
- [■] `'; DELETE FROM Account WHERE username = 'Fred'; --`
- [■] `'; DELETE FROM Account WHERE username = 'Fred' AND '1' = '1`
- [D] `'; DELETE FROM Account WHERE username = 'Fred' AND '1' = '1';`
- [■] `'; DELETE FROM Account WHERE username = 'Fred' AND '1' = '1'; --`
- [F] `'; DELETE FROM Player WHERE player_name = 'Fred';`
- [G] `'; DELETE FROM Player WHERE player_name = 'Fred'; --`

Solution: The goal is to remove a specific user from the **Account** table. Like the previous question, this requires a Stacked Query injection using a semicolon (**;**) to execute a second command after the initial login query. To be correct, the payload must:

1. Close the original string (**'**).
2. End the statement (**;**).
3. Execute a **DELETE** on the correct table (**Account**).
4. Comment out the rest of the query (**--** or **#**).

Now, turning to the options:

- `'; DELETE FROM Account WHERE username = 'Fred';` Incorrect. The resulting query will end with a mismatched single-quote, which will cause a syntax error.
- `'; DELETE FROM Account WHERE username = 'Fred'; --` Correct. Standard stacked query syntax targeting the correct table.
- `'; DELETE FROM Account WHERE username = 'Fred' AND '1' = '1` Correct. A `';` will be appended to the end from the original query, so the last quote will be closed and all quotes will match, and the condition `'1' = '1'` will be a tautology.
- `'; DELETE FROM Account WHERE username = 'Fred' AND '1' = '1';` Incorrect. The original query will append `';`, yielding mismatched quotes.
- `'; DELETE FROM Account WHERE username = 'Fred' AND '1' = '1'; --` Correct. The `--` ensures that the quote appended after this is ignored.
- `'; DELETE FROM Player WHERE player_name = 'Fred';` Incorrect. This will cause a syntax error after `';` is appended, yielding a mismatched quote. Also, it deletes from the wrong table.
- `'; DELETE FROM Player WHERE player_name = 'Fred'; --` Incorrect. This deletes data from the **Player** table. The question specifically asked to delete the row from the **Account** table.

Q8.3 (2 points) The login page has a "Check Username" feature to see if a user exists. It is vulnerable to SQL injection but does not display any database content. It simply prints "User Found" if the query returns any rows, and "User Not Found" otherwise.

```
SELECT * FROM Account WHERE username = '$user';
```

You suspect the `admin` user has a password starting with the letter 's'. Which payload(s) will let you infer whether your suspicion is correct, based on whether the program prints "User Found" or not in response to your payload?

Select all that apply.

Note: `substring(str, start, length)` extracts `length` characters from `str` beginning at index `start`.

A `admin' --`

B `admin' OR '1'='1' --`

■ `admin' AND substring(password, 1, 1) = 's' --`

D `admin' AND '0'='1' UNION SELECT * FROM Account WHERE substring(password, 1, 1) = 's' --`

Solution: The goal is to leak data by making the "User Found" message appear conditionally. We need a query that returns a row **only** if our guess is correct.

- `admin' --` Incorrect. Always finds the admin user. The program prints "User Found" regardless of the password.
- `admin' OR '1'='1' --` Incorrect. Always finds the admin user. The program prints "User Found" regardless of the password.
- `admin' AND substring(password, 1, 1) = 's' --` Correct. The program will print "User Found" if and only if `admin`'s password starts with 's'.
  - If password starts with 's', the expression evaluates to `True AND True` for the `admin` user, so the SQL query returns one row for the `admin` user, so the program prints "User Found".
  - If password does not start with 's', the expression evaluates to `True AND False` for the `admin` user, so the SQL query nothing, so the program prints "User Not Found".
- `... UNION SELECT ...` Incorrect. The `UNION` finds all users whose password starts with 's'. The program prints "User Found" if any user has a password starting with 's' (regardless of whether it is the `admin` user or not), or "User Not Found" if no user has a password starting with 's', so this does not let you infer whether `admin`'s password starts with 's'. Most likely, if there are many users, at least one of them will have a password starting with 's', so most likely the program will print "User Found" no matter what the `admin`'s password might be.

Q8.4 (3 points) You can ask the app to release a player from their team. The app will execute the following SQL command.

```
DELETE FROM Player
 WHERE player_name = '$player'
   AND team_name = '$my_team';
```

You are logged in as `guest` and your team name is `GuestTeam`. Assume `$my_team` is set to `GuestTeam` and cannot be modified. You control `$player` but not `$my_team`.

Provide a SQL payload for the `$player` input that will delete every player belonging to the rival team `Winners`, without deleting players from other teams.

Solution: `Scrub' and '0'='1'; DELETE FROM Player WHERE team_name = 'Winners'; --`

Solution: The goal is to execute an arbitrary `DELETE` command against a specific target (`Winners`), bypassing the team name restriction enforced by the application.

Modifying the existing query is a little tricky because the application appends `AND team_name = 'GuestTeam'` to the end.

The cleanest solution is a Stacked Query injection:

1. Finish the first query: `Scrub' AND '0'='1'` closes the string, and `;` terminates the statement. This lets us start a new SQL command, without deleting any players from other teams. It would also work to provide a player name that is unlikely to appear in any team.
2. Start a new query: `DELETE FROM Player WHERE team_name = 'Winners';` is a completely new command. Since we are starting fresh, we are not bound by the `AND team_name = 'GuestTeam'` constraint of the previous query.
3. Cleanup: `--` comments out the dangling `AND team_name = 'GuestTeam'` that the application appends, preventing a syntax error.

Q8.5 (3 points) On signup, the application executes the following SQL query as a prepared statement. The user-provided inputs `user` and `team` are bound securely to the `?` placeholders:

```
INSERT INTO Account (username, password, team_name)
VALUES (?, 'dummy_pass', ?);
```

Later, when a user views their team page, the app executes the following SQL query (notice how there is no prepared statement here):

```
SELECT * FROM Team WHERE team_name = '$my_team';
```

Assume `$my_team` is retrieved securely from the `Account` table and matches the `team` provided at signup.

Provide a stored-injection payload for the `team` parameter during signup that will drop the entire `Team` table when the user visits their team page later.

Solution: `x'; DROP TABLE Team; --`

Solution: A suitable payload for `$team` is:

```
x'; DROP TABLE Team; --
```

1. At signup, this entire string is stored as the `team_name` for the new account, and stored in the `Account` table. Since the `INSERT` is done using a prepared statement, no injection happens yet.
2. Later, the team page builds a query by concatenating the team name provided earlier: `SELECT * FROM Team WHERE team_name = '$my_team'`;
3. With our payload in `$my_team`, the resulting SQL is: `SELECT * FROM Team WHERE team_name = 'x'; DROP TABLE Team; --'`;
4. The first statement just selects rows with `team_name = 'x'`. The second statement `DROP TABLE Team;` executes and deletes the entire `Team` table.
5. The `--` comments out the trailing ‘ from the original query so there is no syntax error.

The key idea: the malicious string is stored during signup and only turns into executable SQL when the app later injects it into a new query.

Q8.6 (2 points) In 10 words or fewer, describe one other serious security flaw with the signup design in the prior part. Do **not** mention SQL injection; find some other flaw.

There is a default password / Security through obscurity

Solution: In particular, the prepared statement creates an account with the user-supplied username and team name, and uses `dummy_pass` as the initial password. Hopefully, the application later updates the password to whatever password the user selected. However, this provides a brief time window when the account exists with the password `dummy_pass`. During that time, anyone in the world could log into the user's account, using the password `dummy_pass`.

Q9 *Network Security: Caffè Strada* ☕ (11 points)

Consider a local area network (LAN) at a coffee shop.
- Alice is a legitimate user trying to connect to the internet and communicate with Bob.
- Mallory is a malicious attacker on the same LAN.
- The network uses standard ARP for address resolution and DHCP for configuration.

Q9.1 (1 point) Mallory is on the same local network as Alice. She wants to intercept Alice's traffic destined for the Internet by providing false information about the Gateway Router.

If Mallory uses **ARP spoofing**, what information must she substitute in her malicious replies?

Ⓐ Substitute the IP address normally provided by the gateway with Mallory's IP address

● Substitute the MAC address normally provided by the gateway with Mallory's MAC address

Solution: The router/gateway is the system that relays the user's traffic to the Internet. ARP resolves IP addresses to MAC addresses. To spoof the gateway via ARP, Mallory claims that the Gateway's IP maps to **her** MAC address.

Q9.2 (1 point) Mallory is on the same local network as Alice. She wants to intercept Alice's traffic destined for the Internet by providing false information about the Gateway Router.

If Mallory uses **DHCP spoofing**, what information must she substitute in her malicious replies?

● Substitute the IP address normally provided by the gateway with Mallory's IP address

Ⓑ Substitute the MAC address normally provided by the gateway with Mallory's MAC address

Solution: The router/gateway is the system that relays the user's traffic to the Internet. DHCP provides network configuration, including the IP of the default gateway. To spoof the gateway via DHCP, Mallory sends a configuration claiming that the gateway is at **her** IP address.

Q9.3 (2 points) Alice restarts her computer and rejoins the network. She broadcasts a `Client Discover` message to get a network configuration via DHCP.

Mallory sends a malicious `DHCP Offer` to Alice. Which fields in this offer would Mallory alter to position herself as a Man-in-the-Middle? Select all that apply.

■ Gateway Router IP
[B] Alice's IP Address
[C] Alice's public key
[D] WiFi WPA2 pre-shared key
Ⓔ None of the above

Solution: By spoofing the Gateway Router, Mallory ensures that when Alice sends messages to the internet, she sends them to Mallory first.

Q9.4 (1 point) The coffee shop owner wants to improve security. Which of the following defenses would effectively mitigate these vulnerabilities? Select all that apply.

- [A] Implementing WPA2-PSK will prevent Mallory from spoofing ARP packets even if she knows the WiFi password.
- [B] DHCP attacks are trivial to fix by having the router sign all DHCP offers with a hardcoded certificate.
- ● None of the above

Solution: DHCP spoofing is hard to fix because when a user first connects, they don't know who to trust (no root of trust), so they generally have to accept the first valid-looking offer. Signing doesn't help because the user doesn't know what public key to use to verify signatures. WPA2-PSK doesn't help either, because if Mallory knows the Wifi password, Mallory can compute the cryptographic keys needed to eavesdrop on all Wifi packets and send spoofed Wifi packets.

EvanBot and CodaBot are working on a project at a coffee shop. Mallory is connected to the same local network (LAN) and wants to intercept their traffic.

Mallory is an on-path attacker who can guarantee that her packets arrive before any legitimate packets, 100% of the time.

Q9.5 (2 points) EvanBot connects to the network for the first time and broadcasts a DHCP `Client Discover` message.

To trick EvanBot into accepting her configuration, how many packets is Mallory required to send that contain a falsified Source IP or Source MAC address (i.e., pretending to be the legitimate server in Source IP address or Source MAC address of the packet header)?

● 0 Ⓑ 1 Ⓒ 2 Ⓓ More than 2

Solution: 0 packets with falsified Source IP/MAC address.

Because DHCP clients typically accept the first valid offer they receive, regardless who it is from, and Mallory is guaranteed to be faster, she does not need to forge the packet headers to look like the legitimate server. She can simply reply as herself (a rogue server) with her own IP/MAC address, and EvanBot will accept the offer because it arrived first.

Q9.6 (2 points) CodaBot is already connected to the network. He wants to send a packet to the real Gateway Router (IP `10.1.6.1`), but his ARP cache is empty.

CodaBot broadcasts: "Who has IP `10.1.6.1`?".

How many packets does Mallory need to send to poison CodaBot's ARP cache and force him to send his traffic to her instead of the real router?

Ⓐ 0 ● 1 Ⓒ 2 Ⓓ More than 2

Solution: 1 packet.

Mallory only needs to send one malicious ARP response: "My IP is `10.1.6.1` and my MAC address is `[Mallory's MAC address]`". Because of the race condition vulnerability in ARP, CodaBot will accept this single response if it arrives before the real router's response.

Q9.7 (2 points) What kind of attackers can execute a DHCP spoofing attack? Select all that apply.

■ Man-in-the-middle

Ⓒ Off-path

■ On-path

Ⓓ None of the above

Solution: To successfully spoof a DHCP response, the attacker must be able to see the client's `DHCP Discover` or `Request` broadcast to know when to respond (and often to learn transaction IDs, so they can include the transaction ID in their response).

- On-path attackers can see packets and inject spoofed packets, so they can perform this attack.
- Man-in-the-middle attackers can see, modify, and drop packets, so they can also perform this attack (and are often the result of it).
- Off-path attackers are defined as unable to see packets, so they cannot execute this attack because they cannot see the initial request. Also, DHCP packets are not forwarded outside the local area network, so an off-path attacker (on a different LAN) cannot get a malicious DHCP packet to the victim.

Q10 *Web Security: Criss Cross Apple Sauce* 🍏 **(12 points)**

Q10.1 (2 points) Alice visits a search engine that displays her search terms back to her on the results page. For example, if she searches for "cats", the page displays: `You searched for: cats`.

Mallory notices that the website does not sanitize this output. She creates a link containing a malicious JavaScript payload in the search query and tricks Alice into clicking it. When Alice clicks the link, the script executes in her browser.

What specific type of attack is this?

Ⓐ Stored XSS

● Reflected XSS

Ⓒ Cross-Site Request Forgery (CSRF)

Ⓓ SQL Injection

Solution: Reflected XSS. The malicious script is not stored on the server's database; it is "reflected" off the web server immediately as part of the response to the specific request (the search query).

Q10.2 (1 point) Mallory creates a malicious website called `freemoney.com`. On this page, she places a "Claim Prize" button.

Directly on top of this button, she layers a transparent (invisible) `<iframe>` that loads Alice's bank account settings page, specifically positioning the "Delete Ac=count" button directly over her "Claim Prize" button.

Alice visits `freemoney.com` and clicks "Claim Prize," but she unknowingly clicks the "Delete Account" button on the invisible bank page.

What specific type of attack is this?

Ⓐ Phishing

Ⓑ Cross-Site Request Forgery (CSRF)

● Clickjacking (UI Redressing)

Ⓓ Man-in-the-Middle (MITM)

Solution: Clickjacking (UI Redressing). The attacker uses transparent layers to trick the user into clicking on a UI element (the bank button) that is different from what they perceive they are clicking (the prize button).

Q10.3 (2 points) A web developer tries to prevent XSS by writing a filter that simply removes the text `<script>` and `</script>` from all user input.

Which of the following payloads would successfully bypass this specific filter and execute JavaScript?

Ⓐ `<script>alert(1)</script>`

● `<img src=x onerror="alert(1)">`

Ⓒ `<bold>alert(1)</bold>`

Ⓓ `<a href="http://evil.com">Click me</a>`

Solution: `<img src=x onerror="alert(1)">`. Since the developer only filters the specific `<script>` tag, the attacker can use other HTML tags that support event handlers (like `onerror` on an image) to execute JavaScript.

Q10.4 (2 points) Mallory creates a malicious website containing a hidden HTML form. When Alice visits Mallory's site, a script automatically submits this form to `bank.com` to transfer money.

Why does `bank.com` accept this request and transfer the money, even though Alice did not intend to click the button?

Ⓐ Mallory guessed Alice's password and included it in the form.

● Alice's browser automatically attached her `bank.com` session cookies to the request.

Ⓒ Mallory intercepted Alice's Wi-Fi traffic and modified her packets.

Ⓓ The bank's server allows any request that comes from the same IP address as Mallory.

Solution: Alice's browser automatically attached her `bank.com` session cookies to the request. The core vulnerability of CSRF is that browsers automatically include ambient credentials (cookies) with cross-site requests, so the server cannot distinguish between a user clicking a button and a script clicking it for them.

Q10.5 (1 point) A server defends against CSRF by checking the `Referer` header. It rejects requests where the Referer is `evil.com`. However, to protect user privacy, the server accepts requests where the `Referer` header is missing (blank).

How can Mallory bypass this defense?

Ⓐ By encrypting the Referer header so the server cannot read it.

Ⓑ By spoofing the IP address to look like it came from the server.

● By configuring her website (e.g., using `<meta name="referrer" content="no-referrer">`) to tell the browser not to send a Referer header.

Ⓓ It is impossible to bypass; browsers always send the Referer header.

Solution: By configuring her website to tell the browser not to send a Referer header. If the server "fails open" (accepts requests with no header), the attacker simply needs to suppress the header to bypass the check.

Q10.6 (2 points) Alice wants to implement a strong defense against XSS that restricts where scripts can be loaded from. She configures her server to send a specific HTTP header that tells the browser: "Only load scripts from `https://mysite.com`. Block all other scripts."

What is the name of this defense?

Ⓐ Same-Origin Policy (SOP)

Ⓒ Cross-Site Request Forgery (CSRF) Token

● Content Security Policy (CSP)

Ⓓ Referer Validation

Solution: Content Security Policy (CSP). CSP allows site administrators to declare approved sources of content that browsers are allowed to load, effectively mitigating XSS by blocking malicious inline scripts or scripts from unauthorized domains.

Q10.7 (2 points) The most robust defense against CSRF is the use of a CSRF Token (a random, secret value included in forms).

Why does this prevent Mallory from successfully forging a request from `evil.com`?

Ⓐ The token encrypts the session cookie so Mallory cannot use it.

● Mallory cannot read the token from the legitimate site due to the Same-Origin Policy, so she cannot include the correct token in her forged request.

Ⓒ The token verifies that the IP address of the request matches the IP address of the user.

Ⓓ The token prevents the browser from sending cookies to the server.

Solution: Mallory cannot read the token from the legitimate site due to the Same-Origin Policy. While Mallory can force the browser to **send** a request (which includes cookies), she cannot **read** the response or the content of the legitimate page to find the secret token. Without that token, the server rejects the forged request.

Q11 *Memory Safety: Jonah Dreams of Sheep* (12 points)

Consider the following vulnerable C code:

```
1  void sleep() {
2    char dream[24];
3    fread(dream, 24, 1, stdin);
4    count_sheep(dream);
5  }
6
7  void count_sheep(char* input) {
8    size_t num_sheep;
9    fread(&num_sheep, sizeof(size_t), 1, stdin);
10
11   if (num_sheep > 64) {
12     exit(1);
13   }
14   printf(input);
15 }
```

Stack at Line 9

RIP of `sleep`
(1)
(2)
(3)
RIP of `count_sheep`
SFP of `count_sheep`
`num_sheep`

Assumptions:

- All memory safety defenses are disabled.
- There is a copy of `SHELLCODE` at address `0xdeadbeef`.
- There is no compiler padding.
- RIP of `sleep` is located at address `0xffffd634`.

Q11.1 (0.5 points) What goes in blank (1) in the stack diagram above?

Ⓐ `dream` Ⓑ `input` Ⓒ `num_sheep` Ⓓ RIP of `sleep` ● SFP of `sleep`

Q11.2 (0.5 points) What goes in blank (2) in the stack diagram above?

● `dream` Ⓑ `input` Ⓒ `num_sheep` Ⓓ RIP of `sleep` Ⓔ SFP of `sleep`

Q11.3 (0.5 points) What goes in blank (3) in the stack diagram above?

Ⓐ `dream` ● `input` Ⓒ `num_sheep` Ⓓ RIP of `sleep` Ⓔ SFP of `sleep`

Q11.4 (1.5 points) Which of the following memory safety vulnerabilities are present in the above code?

☐ A Stack Buffer Overflow ☐ C Off-by-one Ⓔ None of the above

☐ B Heap Buffer Overflow ■ Format String Vulnerability

Warning: Q11.5 and Q11.6 are very hard. Consider attempting all other questions before spending too much time on these two parts.

Q11.5 (5 points) Provide an input to the `fread` call on Line 3 that will help us execute `SHELLCODE`.

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

For the purposes of this question, the `%*u` specifier reads one argument from the stack (call it `m`), treats it as an integer, and prints `m` bytes. It then proceeds to read a second argument off the stack, but does nothing with it.

Hint: When processed by printf, `%hhn` writes one byte to the address given by the corresponding argument; the value written is the number of characters printed so far.

```
'  \x0c\xd6\xff\xff  ' + '%*u' + ('%c' *  2  ) +
 '%hnn'  +  \xef\xbe\xad\xde  + ('A' *  5  )
```

Solution:

Let's first take a look at the stack diagram:

0xffffd634	[4] RIP of `sleep`	`arg12`
0xffffd630	[4] SFP of `sleep`	`arg11`
0xffffd618	[24] `dream`	`arg5` - `arg10`
0xffffd614	[4] `input`	`arg4`
0xffffd610	[4] RIP of `count_sheep`	`arg3`
0xffffd60c	[4] SFP of `count_sheep`	`arg2`
0xffffd608	[4] `num_sheep`	`arg1`

Our exploit resembles the method for exploiting off-by-one vulnerabilities. The main idea of our exploit is to modify an SFP pointer to cause the stack to become "misaligned", so that a location that already has `&SHELLCODE` stored in memory is interpreted by the CPU as a RIP.

Why so complicated? Why not just overwrite a RIP stored on the stack? Well, the format string vulnerability will allow us (the attacker) to write one byte of our choice somewhere on the stack. Overwriting just one byte of a RIP is not enough to replace it with the address of `SHELLCODE`. We don't have any way to write an entire word (4 bytes), because that would require a long format specifier (e.g., to write each of the bytes separately), and `dream` isn't big enough for that. So we need to get clever.

In more detail:

We note that the hint tells us to use a format specifier, and we take advantage of the unsanitized `printf` call on line 14 to execute a format string attack.

We will overwrite one byte of the SFP of `count_sheep`. We know that it contains the value `0xffffd630` (the address of SFP of `sleep`). The format string vulnerability allows us to manipulate the least significant byte of this value, with a `%hhn` format specifier. We can store the address of `SHELLCODE` somewhere in our `dream` buffer and, using the `%hhn` format specifier, rewrite the SFP of `count_sheep` to point 4 bytes below the address where `&SHELLCODE` is stored.

To do this, we need to make sure that `%hhn` writes to the address where the SFP of `count_sheep` is stored, i.e., to `0xffffd60c`. So, we write `0xffffd60c` into the start of `dream` (the position marked `arg5`, i.e., the 5th argument that `printf` reads from), and we'll construct the format string to so that the `%hhn` is the 5th specifier in the format string.

The first format specifier in the string will be `'%*u'`. This will read two words from the locations marked `arg1`, `arg2`, i.e., from `num_sheep` and the SFP of `count_sheep`, and write `num_sheep` many bytes. Since we can control the contents of `num_sheep` (by specifying an appropriate input to the `fread` on line 9), we control how many bytes `printf` writes. This will be useful in a moment.

Next, we need to consume the 3rd, and 4th argument to `printf` (`arg3`, `arg4`) and effectively ignore them. We use 2 `%c` format specifiers for this purpose, so that our argument pointer moves up to point to the start of `dream` (`arg5`).

Next, we add `'%hnn'`. This format specifier will read an address from `arg5` (the start of `dream`) and write one byte to that address. We've previously arranged that this address will be `0xffffd60c`, i.e., the address where the SFP of `count_sheep` is stored. So this will overwrite the first byte, i.e., the least significant byte, of the SFP of `count_sheep` (with a value that the attacker can control, by controlling the input to the `fread` on line 9).

Then, we add the address `\xef\xbe\xad\xde`. This 4-byte value needs to be stored somewhere on the stack, and here is a good spot. By overwriting the SFP of `count_sheep`, we'll cause the stack to be mis-aligned so that this location on the stack gets treated by the CPU as the RIP of `sleep`. `\xef\xbe\xad\xde` appears 15 bytes after the beginning of `dream`, i.e., at address `0xffffd627`. Therefore, we want to mis-align the stack so that address `0xffffd623` is treated as the SFP of `sleep` and address `0xffffd627` is treated as the RIP of `sleep`. We can do this by overwriting the SFP of `count_sheep` so it contains the address `0xffffd623`, i.e., overwriting its least significant byte with `0x23`.

To overwrite its least significant byte with `0x23`, we need to arrange that `printf` has printed `0x23` (i.e., 35 in decimal) bytes. Before the `'%hnn'` specifier, the format string prints `\x0c\xd6\xff\xff` (4 bytes), whatever is printed by the `'%*u'` (`num_sheep` bytes), then whatever is printed by `'%c' * 2` (2 bytes). Therefore, we want `num_sheep` + 6 to be 35, which requires `num_sheep` = 29 = `0x1D`.

When `count_sheep` returns, the prologue of `count_sheep` will execute `pop %ebp`, which will set `%ebp` to `0xffffd623` (since the location storing the SFP of `count_sheep` was modified to hold `0xffffd623`). When `sleep` returns, its prologue will first execute `mov %ebp, %esp`, which copies the value in `%ebp` (`0xffffd623`) into `%esp` (now `%esp` contains `0xffffd623`); then executes `pop %ebp`, which adds 4 to `%esp` (now `%esp` contains `0xffffd627`); then executes `ret`, which reads the pointer stored at `0xffffd627` and starts executing code there. Since we previously arranged that `&SHELLCODE` is stored at address `0xffffd627`, the `ret` in `sleep`'s prologue will start executing `SHELLCODE`.

There are multiple valid answers to this question. `\xef\xbe\xad\xde` can be moved up to 3 bytes later in the exploit string, as long as the answer to Q11.6 is increased by the same amount. It can also be moved earlier (but not into the first four bytes), with adjustments to the answer to Q11.6.

Q11.6 (2 points) Provide an input to the `fread` call on Line 9 that, together with your answer to the previous part, will execute `SHELLCODE`.

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

```
'\x1D\x00\x00\x00'
```

Q11.7 (1 point) Which memory safety defenses would cause the correct exploit (without modifications) to fail? Consider each choice independently.

■ ASLR ■ Non-Executable Pages Ⓒ None of the above

Solution: ASLR would prevent the exploit, because the RIP of `sleep` would be at a different address.

We had a bug in this question. We didn't clearly specify what memory region `SHELLCODE` is stored in. As a result, it's not possible to tell reliably whether non-executable pages would stop the exploit. If `SHELLCODE` is stored in the heap or stack, non-executable pages would stop the exploit; if for some reason `SHELLCODE` was found in the code region (e.g., it already happened to appear in the code!?), then non-executable pages would not stop the exploit. Therefore, we will ignore whether or not you marked the 2nd option, and only grade the 1st option.

Q11.8 (1 point) Would the same exploit work if the call to `fread` on line 3 was replaced with `fgets` (with the same parameters)?

● Yes, because we can still write in the correct exploit input into `dream`.

Ⓑ Yes, because `fgets` does the same thing as `fread` when given the same number of bytes to read.

Ⓒ No, because `fgets` will overwrite the last byte of the SFP of `sleep` with a null terminator.

Ⓓ No, because the amount of space in `dream` required to perform the exploit will no longer be sufficient.

Solution: The key difference between `fread` and `fgets` here is that given a buffer of `n` bytes, `fread` will read exactly `n` bytes into the buffer, while `fgets` will read `n-1` bytes into the buffer and append a null terminator. Since our exploit input is only 21 bytes long and does not contain any null bytes, it could be used with either `fread` or `fgets`.

(Question 11 continued...)

Post-Exam Activity: The Meaning of it All.

EvanBot, in search of a greater truth that has eluded them for far too long, has fallen deep into their own mind, seemingly trapped indefinitely in their pensive state. Help EvanBot escape by answering life's great mystery:

Select your favorite topic from the cryptography portion of the course.

- ● Diffie-Hellman
- Ⓑ Encrypt-then-MAC
- Ⓒ PRNGs
- Ⓓ El Gamal
- Ⓔ MAC-then-Encrypt (You wouldn't dare)
- Ⓕ None of the above

Showcase how this topic illuminates the meaning of life (you may use words, math, drawings, etc.):

Comment Box

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