CS 161 Summer 2026 — Introduction to Computer Security — Midterm

Name: _____

Student ID: _____

This exam is 110 minutes long. There are 7 questions of varying credit. (100 points total)

Question:	1	2	3	4	5	6	7	Total
Points:	0	14	18	19	18	15	16	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 (0 points):
EvanBot's Excellent Word Search!

K R S G Z V S F A I
Z K T M M V J H C K
M Y E Y E D W L P J
E M M K M H I Q S O
O D F V O B O B X P
X H O B R I J R B I
P P A N Y O A D B M
T E D D S P Z Y K F
M F O Y A L H P K B
D C R N F P O R E O
Z P L K E G R I S K
V F U A T A E N X N
V Y Y I Y X P T Z S
H S O S M I A F V Y
W J E K I R B M K T
G O L C O Q C L Y O
A X S I U T C N Z I
U X H W A R A M Y T
V E P E Y R I N E D
X C V M B Y W T K Y
Z E Q A C E I B Y Q
I U Z V N N G G W V
X Y F Q Y B P C X L
P Q Y S G R O W B C
J F Q S R F Y T O T

Words
Security
Evanbot
Peyrin
Yokota
Memory
Safety
Printf

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*  **(14 points)**

Each question is worth 1 point.

Q2.1 True or False: A system that follows Kerckhoffs's principle stays secure even if the attacker learns every detail of its design, as long as the secret key is not leaked.

● True Ⓑ False

Solution: True. Kerckhoffs's principle (Shannon's maxim, "the enemy knows the system") says security should rest on the secrecy of the **key**, not the obscurity of the design. Designing under the assumption that the attacker knows the algorithm is the opposite of relying on security through obscurity.

Q2.2 True or False: A format-string vulnerability such as `printf(user_input)` allows the attacker to both read from and write to memory.

● True Ⓑ False

Solution: True. Specifiers like `%x` and `%s` read values off the stack (a read primitive), while `%n` writes the number of bytes printed so far to a pointed-to address (a write primitive). A single format-string bug yields both.

Q2.3 True or False: Writing exactly one byte past the end of a buffer never affects a program's control flow.

Ⓐ True ● False

Solution: False. A single byte can overwrite the least-significant byte of an adjacent saved pointer (e.g. the saved frame pointer), which can shift where the program later reads its return address and ultimately redirect control flow.

Q2.4 True or False: Marking the stack non-executable prevents an attacker from overwriting the saved return address.

Ⓐ True ● False

Solution: False. NX stops injected code **on the stack** from executing, but the overflow that overwrites the return address still happens. Attackers bypass NX by redirecting execution to code that already exists (e.g. return-to-libc / ROP).

Q2.5 True or False: A stack canary still stops a buffer-overflow attack, even when the attacker learns the value of the canary in the same execution of the program.

Ⓐ True ● False

Solution: False. A canary works only because its value is secret. If the attacker can leak the canary, they can place the correct value back into their overflow, pass the epilogue check, and still overwrite the RIP. An info leak collapses the defense.

Q2.6 True or False: Prepared statements prevent SQL injection because user-supplied values are never interpreted as SQL code by the database.

● True Ⓑ False

Solution: True. A prepared statement (`... WHERE name = ?`) sends the query template and the data to the database separately, so the substituted value is treated purely as data — never re-parsed as code. This is why it fixes the code-versus-data confusion that string-building and escaping fail to.

Q2.7 True or False: `https://cs161.org:443` and `https://cs161.org:8080` have the same origin.

Ⓐ True ● False

Solution: False. An origin is the triple (protocol, domain, port). The default port for `https` is 443, so the first URL is port 443 and the second is port 8080. A mismatch in any of the three — here, the port — makes them different origins.

Q2.8 True or False: Rate-limiting failed login attempts is vulnerable to a denial-of-service attack against a legitimate user.

● True Ⓑ False

Solution: True. An attacker can deliberately submit wrong guesses on a victim's account to trip the rate limit and lock the victim out — a denial of service. This is a known downside of rate-limiting as a defense.

Q2.9 True or False: The `HttpOnly` cookie attribute prevents a cookie from being sent over non-HTTPS connections.

Ⓐ True ● False

Solution: False. The name is misleading: `HttpOnly` stops **JavaScript** from reading the cookie via `document.cookie` (mitigating token theft via XSS). It says nothing about the transport — the `Secure` attribute is what restricts a cookie to HTTPS.

Q2.10 True or False: A `SameSite=Strict` cookie is enforced by the web server, which rejects any request that originates from a different site.

Ⓐ True ● False

Solution: False. `SameSite=Strict` is enforced by the **browser**: on a cross-site request the browser simply does not attach the cookie, so no request has to be rejected server-side. Checking **after** the request arrives is what the server-side Referer-header defense does instead.

Q2.11 True or False: A Merkle tree lets you prove that a particular block belongs to a large dataset while sending only a logarithmic number of hashes, not the whole dataset.

● True Ⓑ False

Solution: True. A membership proof is the sibling hashes along the path from the block's leaf up to the root — $O(\log n)$ hashes. The verifier recomputes the root from the block plus the proof and checks it against the trusted root hash.

Q2.12 True or False: Unlike passwords, a MAC provides authentication even when an attacker can read data on the insecure channel.

● True Ⓑ False

Solution: True. A MAC lets the recipient verify that a message was not modified and came from someone holding the key, but the message can still travel in the clear — a MAC is not encryption and provides no confidentiality on its own.

Q2.13 True or False: A digital signature is created using the signer's public key and verified using the signer's private key.

Ⓐ True ● False

Solution: False. It is the reverse: the signer signs with their **private** key, and anyone can verify using the corresponding **public** key. Only the private-key holder can produce a valid signature, which is what makes it unforgeable.

Q2.14 True or False: A certificate binds a public key to an identity.

● True Ⓑ False

Solution: True. A certificate is the CA's signature over "(identity, public key)". To verify that signature — and thus trust the binding — you need the CA's public key, which is why browsers ship with a set of trusted root CA public keys.

Q3 *Memory Safety: Length of the Law* ⚖️ **(18 points)**

Consider the following vulnerable C code:

```
1  void log_packet(uint8_t total, char *input) {
2      char buffer[64];
3      uint8_t len;
4
5      if (total > 64) {
6          return;
7      }
8      len = total - 4;
9      memcpy(buffer, input + 4, len);
10 }
```

Stack at Line 5

(1)
(2)
(3)
`len`

Assumptions:

- All memory-safety defenses are disabled.
- You run GDB once and find that `buffer` starts at address `0xffffd6c0`.
- The arguments to the function (`total` and `input`) are values that you provide.
- Your goal is to place and execute a 64-byte long `SHELLCODE`.

(1 point each) What values go in blanks (1) through (3) in the stack diagram above?

Q3.1 Blank (1): Ⓐ SFP of `log_packet` Ⓑ `buffer` ● RIP of `log_packet`

Q3.2 Blank (2): ● SFP of `log_packet` Ⓑ `buffer` Ⓒ RIP of `log_packet`

Q3.3 Blank (3): Ⓐ SFP of `log_packet` ● `buffer` Ⓒ RIP of `log_packet`

Q3.4 (2 points) Which of these vulnerabilities are present in the code? Select all that apply.

- ■ Integer overflow (positive or negative)
- ■ Buffer overflow
- ☐ C Off-by-one error
- ☐ D Format-string vulnerability
- ☐ E Use-after-free error
- Ⓕ None of the above

Solution: The subtraction `len = total - 4` underflows in unsigned 8-bit arithmetic, and the resulting oversized `len` lets `memcpy` write past the end of `buffer` — an integer underflow leading to a buffer overflow. There is no off-by-one, format string, or freed memory here.

In Q3.5–Q3.6, provide inputs 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.

Q3.5 (3 points) Provide a value for `total` (as an integer):

0, 1, 2, or 3

Q3.6 (4 points) Provide a value for `input` (in Python syntax, like in Project 1):

```
'A' * 4 + SHELLCODE + 'A' * 4 + '\xc0\xd6\xff\xff'
```

Solution: `total` must be small enough to underflow Line 9: any `total` in $\{0, 1, 2, 3\}$ passes the check yet gives `len = total - 4` $\in \{252, 253, 254, 255\}$. Take `total = 0`, so `len = 252` and `memcpy` copies 252 bytes from `input + 4`, giving `buffer[i] = input[4 + i]`:

- `input[0..3]` — the 4 header bytes, skipped by the copy (any value).
- `input[4..67]` — copied into `buffer[0..63]`; place the 64-byte `SHELLCODE` here.
- `input[68..71]` — lands on the SFP (any 4 bytes).
- `input[72..75]` — lands on the RIP; set to `0xffffd6c0` (the address of `buffer`), written little-endian as `'\xc0\xd6\xff\xff'`.

When `log_packet` returns, execution jumps to the start of `buffer` and runs `SHELLCODE`. This works because the stack is executable (NX off) and `buffer`'s address is fixed (ASLR off).

Q3.7 (4 points) Which changes to the code would cause the correct exploit (without modification) to fail? Consider each choice independently. Select all that apply.

■ Line 1: Change `uint8_t total` to
`int total`

B Line 2: Change `char buffer[64];` to
`char buffer[68];`

C Line 5: Change `if (total > 64)` to
`if (total < 4 || total > 64)`

■ Line 8: Change `len = total - 4;` to
`len = total - 8;`

E Line 9: Change `memcpy(buffer, input + 4, len);` to
`memcpy(buffer, input + 4, 64);`

F None of the above

Solution: Breaks (select these):

- **Line 2 (`buffer[64]` → `buffer[68]`):** the RIP shifts up one word to offset 72, so `input[72..75]` now lands on the SFP and the RIP is never set to `0xffffd6c0`.
- **Line 5 (add `total < 4`):** this is the real fix; it rejects `total = 0`, so the underflow never occurs.
- **Line 9 (`len` → `64`):** the copy is now exactly `buffer`'s size, so it fills `buffer` without overflowing — the RIP is untouched.

Still works (do not select):

- **Line 1 (`uint8_t total` → `int total`):** `len` is still a `uint8_t`, so `total = 0` gives `len = (uint8_t)(0 - 4) = 252` exactly as before. The type of `total` is irrelevant — the truncation when storing into `len` is what produces the underflow.
- **Line 8 (`- 4` → `- 8`):** `total = 0` now gives `len = 248`, still far past `buffer`, and `memcpy` still copies from `input + 4`, so the `buffer[i] = input[4 + i]` mapping and the RIP at offset 68 are unchanged.

Q3.8 (2 points) Which of these defenses would cause the correct exploit (without modification) to fail? Consider each defense independently. Select all that apply.

■ Non-executable pages

■ Stack canaries

■ ASLR

Ⓓ None of the above

Solution: All three break it independently:

- **NX:** `SHELLCODE` sits on the stack, which is no longer executable, so the jump to `BUFFER_ADDR` faults.
- **ASLR:** `BUFFER_ADDR` is randomized, so the hardcoded RIP is wrong.
- **Stack canaries:** reaching the RIP requires overwriting the canary inserted between `buffer` and the SFP; the corrupted canary is detected in the epilogue and the program aborts before returning.

Q4 *Memory Safety: Double Trouble*  **(19 points)**

Consider the following vulnerable C code:

```
1  struct packet {
2      uint32_t len;
3      char     payload[32];
4      void     (*handler)();
5  };
6
7  void process(char *net1, char *net2) {
8      struct packet q;
9      struct packet p;
10
11     p.len = 37;
12     q.len = 32;
13
14     memcpy(p.payload, net1, p.len);
15     memcpy(q.payload, net2, q.len);
16
17     q.handler();
18 }
```

Stack at Line 11

RIP of `process`
SFP of `process`
Stack Canary
(1)
`q.payload`
(2)
(3)
`p.payload`
`p.len`

Assumptions:

- Stack canaries are enabled and consist of four non-null bytes.
- All other memory safety defenses are disabled.
- You run GDB once and find that the RIP of `process` is located at `0xffffdc90`.
- The arguments to the function (`net1` and `net2`) are values that you provide.
- Your goal is to place and execute a 32-byte long `SHELLCODE`.

C syntax hints:

- `void (*handler)()` is a pointer to a function that has 0 arguments and 0 return values.
- You can call the name of a function pointer in the same way as the name of a function.

(1 point each) What goes in blanks (1) through (3) in the stack diagram above?

Q4.1 Blank (1): Ⓐ `q.len` ● `q.handler` Ⓒ `p.handler`

Q4.2 Blank (2): ● `q.len` Ⓑ `q.handler` Ⓒ `p.handler`

Q4.3 Blank (3): Ⓐ `q.len` Ⓑ `q.handler` ● `p.handler`

Solution:

Stack math:

RIP of `process`	`0xffffdc90`
SFP of `process`	`0xffffdc8c`
Stack Canary	`0xffffdc88`
`q.handler`	`0xffffdc84`
`q.payload`	`0xffffdc64`
`q.len`	`0xffffdc60`
`p.handler`	`0xffffdc5c`
`p.payload`	`0xffffdc3c`
`p.len`	`0xffffdc38`

Q4.4 (2 points) Which of these vulnerabilities are present in the code? Select all that apply.

- ■ Buffer overflow
- [B] Signed/unsigned vulnerability
- [C] Format-string vulnerability
- [D] Use-after-free
- Ⓔ None of the above

Solution: The `memcpy` on Line 14 allows writing 37 bytes, so an over-long `net1` writes past the 32-byte `p->payload` and into the adjacent struct fields — a buffer overflow.

`len` is an unsigned `uint32_t` that is never converted to a signed type, `net1`/`net2` are never used as format strings, and no memory is freed, so none of the other classes apply.

In Q4.5–Q4.6, provide inputs 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.

Q4.5 (4 points) Provide a value for `net1`:

```
'A' * 36 + '\x24'
```

Solution: The 36 bytes of garbage overwrite all of `p.payload` and `p.handler` directly above it.

Then, the 37th byte affects the LSB of `q.len`. The value of `q.len` is currently 32 = `0x00000020`, and we can overwrite its value to become 36 = `0x00000024`.

This will allow the next subpart to write 4 bytes past the end of `q.payload`.

Q4.6 (4 points) Provide a value for `net2`:

```
SHELLCODE + '\x64\xdc\xff\xff'
```

Solution: After Q4.5, `q.len = 36`, so `memcpy(q.payload, net2, 36)` copies 36 attacker-controlled bytes into `q.payload` (32 bytes) and the 4 bytes immediately after it — which are `q.handler`. The mapping is `q.payload[i] = net2[i]`:

- `net2[0..31]` — the 32-byte `SHELLCODE`, placed in `q->payload`, which starts at `0xffffdc64`.
- `net2[32..35]` — overwrites `q->handler`; set it to `0xffffdc64` (the address of `q->payload`), written little-endian as `'\x64\xdc\xff\xff'`.

When Line 17 calls `q.handler()`, execution jumps to the start of `q.payload` and runs `SHELLCODE`.

Alt solution: place shellcode into p.payload in net1, then call q.handler() with `0xffffdc3c`

- `net1 = SHELLCODE + 'A'*4 + '\24'`
- `net2 = 'A' * 32 + '\x3c\xdc\xff\xff'`

Q4.7 (4 points) Which changes to the code would cause the correct exploit (without modification) to fail? Consider each choice independently. Select all that apply.

■ Line 3: Change `char payload[32];` to
`char payload[64];`

☒ Line 12: Change `q.len = 32;` to
`q.len = 8;`

■ Line 14: Change `memcpy(p.payload, net1, p.len);` to
`strlcpy(p.payload, net1, p.len);`

■ Swap Lines 14 and 15.

Ⓔ None of the above

Solution: Breaks the exploit (select these):
- **Line 14 (`strcpy` → `strncpy(..., 32)`):** the first copy is now bounded to 32 bytes, so it can no longer reach `q->len`; `q->len` stays 32 and the second copy never reaches `q->handler`.
- **Line 3 (`payload[32]` → `payload[64]`):** every offset changes — `q.len` is now 68 bytes past `p.payload`, so `net1` no longer reaches it, and `q.payload`/`q.handler` move as well.
- **Swap Lines 14/15:** the `memcpy` runs while `q.len` is still 32, copying only 32 bytes (no overflow into `q.handler`); the later `strcpy` corrupts `q->len` too late to matter, so `q.handler` still points at `dispatch`.

Does **not** break it (do not select):
- **Line 12 (`q.len = 32` → `q.len = 8`):** irrelevant — the `strcpy` from Q4.5 overwrites `q->len` with 36 before the `memcpy` reads it.

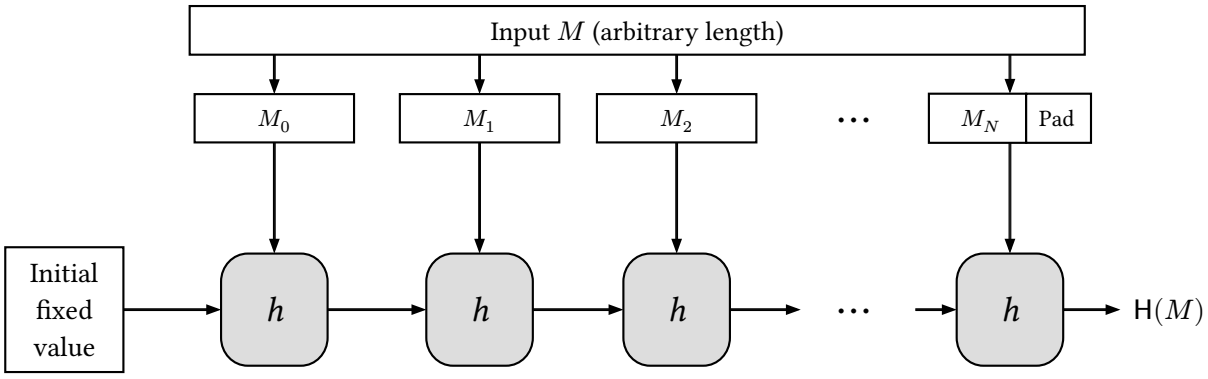
Q4.8 (2 points) Which of these defenses would cause the correct exploit (without modification) to fail? Consider each modification independently. Select all that apply.

■ Non-executable pages ■ ASLR Ⓒ None of the above

Solution:
- **NX:** `SHELLCODE` sits on the stack, which is no longer executable, so the indirect call to `0x0804b034` faults.
- **ASLR:** the heap base is randomized, so the hardcoded `q.payload` address in `q.handler` (and the offsets in `net1`) are wrong.

Q5 *Authentication: The Long Con* 🎭 **(18 points)**

Consider a hash function H built using the Merkle-Damgård construction from lecture:



Evanbot builds the following MAC scheme for key K and message M:

$$t = \mathsf{H}(K \parallel M)$$

Q5.1 (2 points) Why is $t = \mathsf{H}(K \parallel M)$ not EUF-CMA secure?

Ⓐ Mallory can find a collision, which lets her find a new message for the same tag.

Ⓑ Mallory can find a collision, which lets her find a new tag for the same message.

Ⓒ Mallory can recover K and find a tag for any message.

● Mallory can use the tag to find a new message-tag pair.

Solution: This is a length-extension attack. For a Merkle–Damgård hash, the tag t **is** the internal state after absorbing $k \parallel m$ (plus padding). Mallory can load that state and keep hashing more blocks, producing the tag for a longer message — all without knowing k. (SHA-256 is collision resistant and one-way, so the other options are false.)

Suppose a web server accepts GET requests with the following name-value pairs as parameters:
- `amount=100`, where `100` is an integer indicating an amount of money to transfer.
- `recipient=bob`, where `bob` is a string indicating the recipient.
- `tag=`$\mathsf{H}(K \parallel M)$, the MAC of all the other parameters (e.g. M = `amount=100&recipient=bob`).

If a parameter appears more than once, e.g. `amount=100&amount=200`, the last one is used.

Q5.2 (2 points) To use the Merkle-Damgård construction, the hash input must be padded.

To pad, we compute the number of padding bytes we need, and append that number (as a byte) repeatedly, until the message length is a multiple of 64.

If K is 16 bytes, and M is `amount=100&recipient=bob` (where each character is one byte), which of these is the correctly-padded input to the hash?

Ⓐ `K + 'amount=100&recipient=bob' + '\x28' * 40`

Ⓑ `K + 'amount=100&recipient=bob' + '\x24' * 24`

● `K + 'amount=100&recipient=bob' + '\x18' * 24`

Ⓓ `K + 'amount=100&recipient=bob' + '\x28' * 28`

Solution: Each block of this construction takes in 64 bytes. The Key (K = 16 bytes) and Message (M = 24 bytes) for a total of 40 bytes, therefore we need 24 more bytes. 24 as a byte is `'\x18'` . So according to the padding scheme, we construct padding = `'\x18' * 24`, and add it to the end of the current K and message: K + M + padding.

Q5.3 (4 points) Mallory sees a message-tag pair (M, t), where $M =$ `amount=100&recipent=bob`.

Let `pad_M` be the correct padding for $K \parallel M$ (from the previous subpart).

Which of these messages M^* can Mallory produce a forged tag for?

Ⓐ `'amount=100&recipient=bob' + '&recipient=mallory' + pad_M`

● `'amount=100&recipient=bob' + pad_M + '&recipient=mallory'`

Ⓒ `'amount=100&recipient=bob' + '&recipient=mallory'`

Ⓓ `'recipient=mallory' + '&amount=100&recipient=bob' + pad_M`

Solution: In order to extend this current message to add our own data, we need to exploit the internal structure of this hash. Starting with the existing message, when it is sent, the original tag is computed after it was padded, so we must first construct `amount=100&recipent=bob + pad_M` (assuming that pad_M is the correct padding). We are able to append our forged message after this construction.

Q5.4 (3 points) How does Mallory compute the forged tag t^* for the message M^* from Q5.3?

● Set H's initial value to t, then pad and hash the new part of M^* using this hash function.

Ⓑ Hash t concatenated with the padded new part of M^*.

Ⓒ Pad and hash M^* on its own, then XOR the result into the tag t.

Ⓓ Brute-force the key K from t, then hash $K \parallel M^*$.

Solution: The tag t is precisely the chaining state after $k \parallel m \parallel \mathsf{pad}$. By setting the hash's state to t and feeding in m' followed by the correct final padding, Mallory continues the computation exactly where the legitimate signer left off, yielding $t' = \mathsf{H}(k \parallel m \parallel \mathsf{pad} \parallel m')$ — a valid tag for the message $m \parallel \mathsf{pad} \parallel m'$. No key recovery is needed, and restarting from the standard initial state ($\mathsf{H}(t \parallel m')$) would **not** match.

For Q5.5-Q5.9, select whether the given scheme is secure against the length-extension attack from above.

Assume:
- H is a Merkle–Damgård hash.
- c, c_1, and c_2 are fixed, publicly known constants.

Q5.5 (1 point) $t = \mathsf{H}(M \parallel K)$

● Secure Ⓑ Insecure

Solution: Secure (against this attack). The key is now a **suffix**, not a prefix. To verify a message M' the server computes $\mathsf{H}(M' \parallel k)$, which always ends in k. A length-extension forgery would only let Mallory compute $\mathsf{H}(m \parallel k \parallel \mathsf{pad} \parallel x)$ — she cannot place her extension after the trailing k without knowing it, so the prefix-extension trick gains nothing.

Q5.6 (1 point) $t = \mathsf{H}(c \parallel K \parallel M)$

Ⓐ Secure ● Insecure

Solution: Insecure. Prepending a **public** salt changes nothing: the key is still a prefix and the full digest is still exposed. Mallory extends $\mathsf{salt} \parallel k \parallel m$ to $\mathsf{salt} \parallel k \parallel m \parallel \mathsf{pad} \parallel m'$ and updates the tag from t, forging a valid tag for the message $m \parallel \mathsf{pad} \parallel m'$ without ever learning k.

Q5.7 (1 point) $t = \mathsf{H}(K) \oplus \mathsf{H}(M)$

Ⓐ Secure ● Insecure

Solution: Insecure. From the single observed pair (m, t), Mallory computes the public value $\mathsf{H}(m)$ and recovers $\mathsf{H}(k) = t \oplus \mathsf{H}(m)$. She can then forge a tag for **any** message m' as $\mathsf{H}(k) \oplus \mathsf{H}(m')$. (This is not even length extension — exposing $\mathsf{H}(k)$ breaks it outright.)

Q5.8 (2 points) $t = \mathsf{H}(K \parallel M)$`[0:128]`, where H outputs 256 bits and the tag only keeps the first 128 bits.

● Secure Ⓑ Insecure

Solution: Secure (against this attack). Length extension needs the **entire** output of $\mathsf{H}(k \parallel m)$, since that value is exactly the internal state Mallory must resume hashing from. Releasing only half of it leaves her unable to reconstruct the state, so she cannot continue the computation. (Truncating the output is a standard defense against length extension.)

Q5.9 (2 points) $t = \mathsf{H}((K \oplus c_2) \parallel \mathsf{H}((K \oplus c_1) \parallel M))$

● Secure Ⓑ Insecure

Solution: This construction nests two hashes and re-applies the key at the outer layer. The published tag is $\mathsf{H}((k \oplus c_2) \parallel (\text{inner digest}))$, where the inner digest has a fixed length, so there is nothing for Mallory to "extend" into a message of the verifier's form. To forge a tag for a longer message she would need to recompute the inner hash $\mathsf{H}((k \oplus c_1) \parallel m')$, which requires k. (The constants c_1, c_2 are public, and the scheme does not encrypt m.) This nested construction is HMAC.

This page left intentionally (mostly) blank

The exam continues on the next page.

Q6 *Web Security: PancakeStack* 🥞 **(15 points)**

EvanBot runs PancakeStack, a recipe-sharing site at `https://pancakestack.com`.
The site stores its data in the following two SQL tables:

Recipes	
`id`	`uint32`
`title`	`string`
`author`	`string`
`body`	`string`

Users	
`username`	`string`
`email`	`string`
`password`	`string`

PancakeStack has two features that place user input onto a page:

- **Search.** Visiting `https://pancakestack.com/search?q=QUERY` renders a page that begins with the text "Results for: `QUERY`", followed by matching recipes.
- **Reviews.** A logged-in user can post a review on a recipe. Reviews are saved in the database and shown to every user who later loads that recipe's page.

Unless otherwise stated, assume PancakeStack does not utilize input sanitization or escaping.

Q6.1 (3 points) To find recipes, the search feature runs the following query, substituting the URL's `q` parameter for `$q`:

```
SELECT title, author FROM Recipes WHERE title = '$q'
```

Which of these values of `$q` successfully leak `evanbot`'s `email` and `password`? Select all that apply.

■ `' UNION SELECT email, password FROM Users WHERE username = 'evanbot' --`

■ `' UNION SELECT password, email FROM Users WHERE username = 'evanbot' --`

C `' UNION SELECT title, author FROM Recipes WHERE author = 'evanbot' --`

D `' UNION SELECT id, email, password FROM Users WHERE username = 'evanbot' --`

E `' UNION SELECT email, password FROM Users WHERE username = 'evanbot'`

F None of the above

Solution: A `UNION` glues a second `SELECT` onto the results, but only if both `SELECT`s return the **same number of columns** (the outer query returns two: `title`, `author`). Both correct payloads select exactly two columns — `email` and `password` — from `Users` and comment out the trailing `'` with `-- ` . Column **order** doesn't matter: swapping to `password, email` still dumps both secrets.

The distractors fail:

- **Recipes table:** a valid two-column `UNION`, but it reads from `Recipes`, so it leaks recipe titles and authors — never the `email`/`password` in `Users`. (Matching the column count is necessary but not sufficient; you also need the right table.)
- **Three columns** (`id, email, password`) mismatches the two-column outer query, so the `UNION` is a syntax error.
- **No `-- `:** without the comment, the query's own trailing `'` is left dangling, producing an unterminated-string syntax error.

Q6.2 (4 points) PancakeStack's login page runs the query below, substituting the submitted username and password for `$user` and `$pass`:

```
SELECT username FROM Users WHERE username = '$user' AND password = '$pass'
```

The user is logged in if the query returns one or more results.

To stop injection, the server escapes every single-quote (`'`) in the user's input by replacing each one with `\'`. Give values of `$user` and `$pass` that log you in without knowing any user's password.

`$user`:

`$pass`:

Solution: Set `$user` to a single backslash `\` and `$pass` to `OR 1=1 --` . Neither contains a `'`, so the escaper leaves both untouched, and the server builds:

```
... WHERE username = '\' AND password = 'OR 1=1 -- '
```

The parser reads `'\'` as open-quote, an **escaped** quote (`\'`, a literal `'` inside the string), and then keeps reading — the string only closes at the next quote, which is the opening quote the server put before `$pass`. So `\' AND password =` becomes one string value, and the contents of `$pass` (`OR 1=1`) land **outside** any quotes as live SQL. The `--` comments out the final `'`. The condition collapses to `username = '...' OR 1=1`, true for every row — authentication is bypassed.

For Q6.3–Q6.4, assume an attacker injects a `<script>` tag using each of PancakeStack's two features.

Q6.3 (1 point) Injecting a malicious query into the search feature (via the `?q=` URL) is an example of:

- ● A reflected XSS attack
- Ⓒ A SQL injection attack
- Ⓑ A stored XSS attack
- Ⓓ A CSRF attack

Q6.4 (1 point) Which attack can compromise a victim who loads a recipe page, without clicking on any attacker-supplied link?

- Ⓐ Only the search injection
- Ⓒ Both injections
- ● Only the review injection
- Ⓓ Neither injection

Solution: The search page reflects input straight from the URL back onto the page, so the attacker must get the victim to open a crafted link — that's **reflected** XSS. The review is saved in the database and served to everyone who loads the recipe afterward — that's **stored** XSS, and it fires on a normal page load with no special link required.

Q6.5 (3 points) An attacker successfully executes an XSS attack. A victim viewing the injected JS is logged into `https://bank.com` in another tab. PancakeStack's session cookie is set with `HttpOnly=True`.

Which of the following can the injected JS do? Select all that apply.

■ Send a request to `pancakestack.com` that the server treats as coming from the victim.

■ Read and modify the content currently shown on the victim's `pancakestack.com` page.

C Read and modify the content currently shown on the victim's `bank.com` page.

D Read the `bank.com` session cookies from the victim's other browser tab.

E Read the `pancakestack.com` session token by accessing `document.cookie`.

F None of the above

Solution: Injected JavaScript runs with the origin of the page that served it — `pancakestack.com` — because the browser has effectively "copy-pasted" it onto that page. So it can do anything a `pancakestack.com` script can: read and modify the current page's content, and issue requests to `pancakestack.com` that carry the victim's cookies automatically.

The same-origin policy blocks the two `bank.com` options: `bank.com` is a different origin, so the script can neither read/modify its page nor read its cookies.

The trap: even with PancakeStack's origin, the session cookie is `HttpOnly`, which specifically prevents JavaScript from reading it via `document.cookie`. The attacker can still **act as** the victim (requests attach the cookie automatically) but cannot exfiltrate the token itself.

Q6.6 (3 points) Which of the following, deployed on its own, would stop a stored XSS attack from executing? Select all that apply.

■ Setting a Content Security Policy that blocks all JavaScript on the page.

■ Using an HTML templating engine that auto-escapes the review before display.

C Sanitizing only `<script>` and `</script>` tags from the review.

D Setting the session token cookie's `HttpOnly` attribute to `True`.

E None of the above

Solution: A CSP that blocks inline JavaScript stops **both** inline `<script>` blocks and inline event handlers like `onerror` — exactly where the @pancake-onerror payload hides — so nothing runs. HTML templating escapes the special characters, so `<img ...>` is shown as literal text instead of being parsed as a tag.

The script-only sanitizer is still defeated (the payload contains no `<script>`), and `HttpOnly` is a red herring: it protects the cookie from being **read** by JavaScript but does nothing to stop the JavaScript from executing.

Q7 *Web Security: GradeBot*  **(16 points)**

CS 161 runs GradeBot, a course grading service hosted at `https://gradebot.cs161.org`. When a user logs in, the server authenticates them and returns a session token cookie that the browser attaches to future requests.

Q7.1 (3 points) `https://gradebot.cs161.org/login` returns a response with a `Set-Cookie` header.

Which `Domain` attribute values will the browser accept for this cookie? Select all that apply.

- ■ `gradebot.cs161.org`
- ■ `cs161.org`
- C `www.gradebot.cs161.org`
- D `.org`
- E `berkeley.edu`
- F None of the above

Solution: When **setting** a cookie, the `Domain` attribute must be the server's own domain or one of its (non-TLD) parents. From `gradebot.cs161.org`, that's `gradebot.cs161.org` (itself) and `cs161.org` (a parent). The others fail: `www.gradebot.cs161.org` is **not** a parent of the server (it's a different, more specific name), `.org` is a top-level domain and far too general, and `berkeley.edu` is unrelated. (The `Path` attribute, by contrast, may be set to anything.)

Q7.2 (3 points) A cookie is stored with `Domain=cs161.org` and `Path=/exams`.

The user opens each of these URLs. Select all requests that the browser attaches this cookie for.

- ■ `https://gradebot.cs161.org/exams/final`
- ■ `https://cs161.org/exams`
- C `https://gradebot.cs161.org/grades`
- D `https://gradebot.com/exams/final`
- E `https://evil.org/exams`
- F None of the above

Solution: A cookie is **sent** only when both the domain and the path match the request URL. Domain matches if the request host is the cookie's `Domain` or a subdomain of it, and path matches if the request path begins with the cookie's `Path`.

- `gradebot.cs161.org/exams/final`: subdomain of `cs161.org` ✓, path starts with `/exams` ✓ — sent.
- `cs161.org/exams`: exact domain ✓, path matches ✓ — sent.
- `gradebot.cs161.org/grades`: domain ✓, but `/grades` does not start with `/exams` — **not** sent.
- `gradebot.com/exams/final`: `gradebot.com` is a different domain from `cs161.org` — **not** sent.
- `evil.org/exams`: domain mismatch — **not** sent.

Q7.3 (2 points) Which attributes should GradeBot set on the session token cookie to best protect it? Select all choices that improve the token's security.

■ `Secure=True`

■ `HttpOnly=True`

■ A short `Expires` time (e.g. 15 minutes).

D A very long `Expires` time (e.g. 10 years).

Ⓔ None of the above

Solution:

- `Secure=True`: the cookie is only ever sent over HTTPS, so a network eavesdropper on a plaintext connection can't capture the token.
- `HttpOnly=True`: JavaScript can't read the cookie via `document.cookie`, so an XSS bug on the site can't steal the session token.
- **Short** `Expires`: limits how long a stolen token remains usable.

The other two **reduce** security: broadening `Domain` to `cs161.org` sends the token to every `cs161.org` subdomain (a larger attack surface than just `gradebot.cs161.org`), and a very long expiration widens the window during which a leaked token works.

GradeBot lets users logged in as TAs change a grade by issuing a `GET` request to `https://gradebot.cs161.org/setgrade?student=X&score=Y`.

Q7.4 (2 points) Mallory is not a TA, but wants to update her grade. She crafts this malicious URL: `https://gradebot.cs161.org/setgrade?student=mallory&score=100`.

Which of these attacks would successfully change Mallory's grade? Select all that apply.

- [■] Mallory emails a logged-in TA a link to the malicious URL, and the TA clicks it.
- [■] Mallory posts `<img src="`the malicious URL`">` on a forum the TA browses while logged in.
- [C] Mallory, logged in as herself, visits the malicious URL in her own browser.
- [D] Mallory puts `<img src="`the malicious URL`">` on a page that only she (not the TA) ever loads.
- [E] None of the above

Q7.5 (1 point) Why does the GradeBot server accept the forged request from Q7.4?

- (●) The victim's browser automatically attaches GradeBot's session cookie to the request.
- (B) The `<img>` tag runs with GradeBot's origin under the same-origin policy.
- (C) The `Referer` header is stripped, so the server cannot tell where the request came from.
- (D) Mallory's own session token is copied onto the TA's request.

Solution: A CSRF attack needs the **victim's** authenticated browser to send the request. Both correct options do that: the TA clicking the link issues the `GET`, and the `<img>` tag causes the TA's browser to fetch the URL when the forum page loads (the image "fails," but the request — with cookies — is still sent). The two distractors use **Mallory's** browser or a page the TA never loads, so the TA's session cookie is never attached and the TA's authority is never borrowed.

The server is fooled because the browser **automatically** attaches the `gradebot.cs161.org` session cookie to any request to that domain, so the forged request looks exactly like a legitimate one from the TA. (The request does **not** run with GradeBot's origin — it's an ordinary cross-site request — and Mallory's own token is never placed on the victim's request.)

Q7.6 (2 points) GradeBot changes `/setgrade` to require an HTTP `POST` request. Describe how Mallory can still force the TA's browser to update her grade.

Answer in 10 words or fewer; the staff answer is 1 word.

JavaScript

Solution:

Host a page whose JavaScript auto-submits a hidden `POST` form.

(Any answer capturing "an attacker page that auto-submits a `POST` form/JS" earns credit. When the TA loads it, the browser sends the `POST` with the TA's session cookie attached, so the grade change goes through.)

Q7.7 (3 points) Which of the following defenses prevents Mallory's attack? Select all that apply.

■ The server includes a CSRF token in the page body that the client must attach to each request.

B The server includes a CSRF token in a cookie that the client must attach to each request.

■ The server sets `SameSite=Strict` on the session cookie.

D The server sets `HttpOnly=True` on the session cookie.

■ The server checks the `Referer` header and rejects requests not from `gradebot.cs161.org`.

F None of the above

Solution:

- **CSRF token (in the page body):** Mallory can't read GradeBot's page (the same-origin policy blocks her), so she can't learn the per-request token to put in her forged request, and the server rejects it.
- `SameSite=Strict`**:** the browser refuses to attach the cookie to a **cross-site** request, so Mallory's forged request arrives unauthenticated and is rejected.
- `Referer` **check:** the forged request's `Referer` is Mallory's site, not `gradebot.cs161.org`, so the server rejects it. (Imperfect in practice, since the header is sometimes absent, but it does stop **this** request.)

The two traps do **not** help:

- `HttpOnly` only stops JavaScript from **reading** the cookie. CSRF never reads the token — it relies on the browser **auto-attaching** it — so `HttpOnly` is irrelevant to CSRF.
- **CSRF token in a cookie:** a cookie is auto-attached to every request to the domain, including Mallory's forged one, so her request carries the correct token for free. The token only works when the client must actively supply it from the page body, which is something the attacker cannot do.

(Question 7 continued...)

Post-Exam Activity: Defending Caltopia

EvanBot is ready to do everything possible to defend Caltopia! What is bot defending us against?

Art by: Soklynin Nou *Artists' Note: Hands are hard*

Comment Box

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