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

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

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

Ⓐ True Ⓑ False

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

Ⓐ True Ⓑ False

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

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

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

Ⓐ True Ⓑ False

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

Ⓐ True Ⓑ False

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

Ⓐ True Ⓑ False

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

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

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

Ⓐ True Ⓑ False

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

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

Ⓐ True Ⓑ False

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.

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

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):

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

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.

A 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)
```

D 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

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.

A Non-executable pages

B ASLR

C Stack canaries

D None of the above

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`

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

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

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`:

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

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.

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

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

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

D Swap Lines 14 and 15.

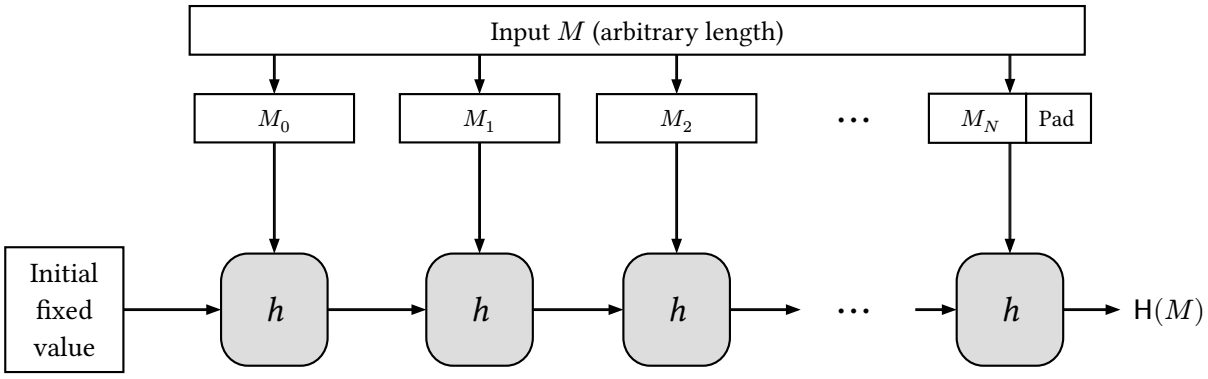
E None of the above

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.

A Non-executable pages B ASLR C None of the above

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.

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`

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`

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 $K \parallel M^*$.

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

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

Ⓐ Secure Ⓑ Insecure

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

Ⓐ Secure Ⓑ Insecure

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

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

Ⓐ Secure Ⓑ Insecure

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' --`

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

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

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

Ⓕ None of the above

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`:

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 stored XSS attack
- Ⓒ A SQL injection 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
- Ⓑ Only the review injection
- Ⓒ Both injections
- Ⓓ Neither injection

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.

- [A] Send a request to `pancakestack.com` that the server treats as coming from the victim.
- [B] 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`.
- Ⓕ None of the above

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

- [A] Setting a Content Security Policy that blocks all JavaScript on the page.
- [B] 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`.
- Ⓔ None of the above

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.

- [A] `gradebot.cs161.org`
- [B] `cs161.org`
- [C] `www.gradebot.cs161.org`
- [D] `.org`
- [E] `berkeley.edu`
- (F) None of the above

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.

- [A] `https://gradebot.cs161.org/exams/final`
- [B] `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

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.

- [A] `Secure=True`
- [B] `HttpOnly=True`
- [C] A short `Expires` time (e.g. 15 minutes).
- [D] A very long `Expires` time (e.g. 10 years).
- (E) None of the above

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.

- [A] Mallory emails a logged-in TA a link to the malicious URL, and the TA clicks it.
- [B] 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?

- (A) 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.

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.

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

- [A] 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.
- [C] The server sets `SameSite=Strict` on the session cookie.
- [D] The server sets `HttpOnly=True` on the session cookie.
- [E] The server checks the `Referer` header and rejects requests not from `gradebot.cs161.org`.
- (F) None of the above

(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: