

Solutions last updated: Friday, October 9, 2026

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	9	16	22	15	13	25	100

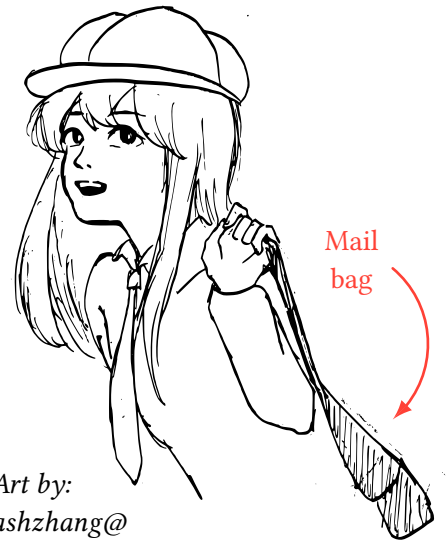
For questions with **circular bubbles**, you may select only one choice.

- ☐ 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)

According to the official CS 161 lore, what is Alice's day job?

- ☐ Professor
- ☐ Student
- ☒ Mailman
- ☐ Architect
- ☐ Spy
- ☐ Other: _____

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: A Little Bit of Everything 🍷

(9 points)

Each question is worth 1 point.

Q2.1 Consider a C compiler that leaves stack canaries off unless the programmer uses a special flag.

TRUE OR FALSE: This compiler is correctly following the “use fail-safe defaults” security principle.

- ☐ A TRUE ☒ B FALSE

Solution: False. A fail-safe default is the secure setting chosen automatically. If the protection is off unless someone remembers to turn it on, many programs will never enable it (lecture’s Cisco ASA example shipped without canaries). The fail-safe choice is to enable mitigations by default and let programmers opt out.

Q2.2 TRUE OR FALSE: In the struct shown to the right, `p.a` lives at a lower memory address than `p.b`.

```
struct {  
    int a;  
    int b;  
} p;
```

- ☒ A TRUE ☐ B FALSE

Solution: True. Inside a struct the first member is at the lowest address, so within `p`, `p.a` is below `p.b`.

Q2.3 A 32-bit program uses stack canaries whose first byte is always a null byte, as described in lecture. How many unique values of the canary are there?

- ☐ A 2^8 ☒ B 2^{24} ☐ C 2^{32} ☐ D 2^{56}

Solution: 2^{24} . A 32-bit canary has one byte fixed to `\x00` (so string-based overflows stop at it or must change it), which leaves $32 - 8 = 24$ random bits, about 16 million values. That is feasible to brute-force without rate limiting. 2^{56} is the 64-bit case.

Q2.4 TRUE OR FALSE: Pointer authentication is secure against an attacker trying to brute-force the value of the PAC (pointer authentication code).

- ☐ A TRUE ☒ B FALSE

Solution: False. The PAC has only about 11 to 31 bits, so there are millions to billions of possibilities. Guessing may be feasible depending on the threat model.

Q2.5 TRUE OR FALSE: To use a NOP sled, an attacker should overwrite the RIP/RA of a function with no-op instructions (e.g. write `0x90909090` at the RIP/RA).

- ☐ A TRUE ☒ B FALSE

Solution: False. The RIP/RA of the function should be overwritten with the *address* of no-op instructions, not literal no-op instructions.

(Question 2 continued...)

Q2.6 Alice sends $M, H(M)$ to Bob over a channel that Mallory controls, where H is SHA-256. Bob recomputes $H(M)$ and accepts M if the hash he computed matches the hash he received.

TRUE OR FALSE: This lets Bob detect any modification Mallory makes to M .

- ☐ (A) TRUE ☒ (B) FALSE

Solution: False. A hash has no key, so anyone can compute it. Mallory replaces the pair with $M', H(M')$, and Bob's check passes. Hashes give integrity only when the hash value itself comes over a trusted channel (e.g., a hash you stored locally or a hash on the original website). Against an active attacker you need a keyed primitive such as a MAC or a signature.

Q2.7 Bob uses Trust on First Use (TOFU) for Alice's public key. Mallory tampers with Alice's public key on Bob's very first connection to Alice.

TRUE OR FALSE: On Bob's later connections to Alice, Bob will flag Alice's real public key as a mismatch.

- ☒ (A) TRUE ☐ (B) FALSE

Solution: True. On first contact, Bob's client cached the key it received, which was Mallory's key. TOFU then treats that key as Alice's, so when the real Alice shows up with her real key, the client sees a mismatch and warns or rejects. This is why lecture calls TOFU "worse than nothing" when the attacker is present on the first connection. Its security depends entirely on the first connection not being attacked.

Q2.8 TRUE OR FALSE: If Bob receives a certificate for `alice.com` that is correctly signed by a certificate authority (CA) he trusts, he can conclude that the party who sent him the certificate is `alice.com`.

- ☐ (A) TRUE ☒ (B) FALSE

Solution: False. Certificates are public, signed statements of the form "this public key belongs to this name." Anyone can obtain a copy and send it, including Mallory. A certificate tells Bob which key belongs to `alice.com`. To authenticate the sender, Bob also needs proof that the sender holds the matching secret key, such as a fresh message signed with it.

Q2.9 TRUE OR FALSE: Decrypting an AES-CTR ciphertext runs AES only in its encode (encryption) direction, and never uses AES's decode (decryption) function.

- ☒ (A) TRUE ☐ (B) FALSE

Solution: True. CTR mode generates the pad $E_K(r) \parallel E_K(r+1) \parallel \dots$ in the same way when encrypting and decrypting, and decrypting just XORs that pad with the ciphertext. CTR therefore needs only a PRF, and AES's inverse function is never used. (Legacy chaining modes are the ones that need the block cipher's inverse.)

Q3 Memory Safety: Negative Attention

(16 points)

Consider the following vulnerable C code:

```

1 void set_score(int *scores, int idx, int val) {
2     if (idx > 7) {
3         return;
4     }
5     scores[idx] = val;
6 }
7
8 void game(int *packet) {
9     int scores[8];
10    set_score(scores, packet[0], packet[1]);
11 }

```

Stack at Line 5

RIP/RA of game
SFP of game
scores
(1)
(2)
(3)
RIP/RA of set_score
SFP of set_score

Assumptions:

- All memory-safety defenses are disabled.
- You run GDB once and find that **scores** starts at address **0xffffd6c0**.
- **packet** points to an integer array, and you control the values in this array.
- Your goal is to execute shellcode that is already placed in memory. You run GDB once and find that the shellcode starts at address **0xdeadbeef**.

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

Q3.1 Blank (1): ☒ A val ☐ B idx ☐ C scores

Q3.2 Blank (2): ☐ A val ☒ B idx ☐ C scores

Q3.3 Blank (3): ☐ A val ☐ B idx ☒ C scores

Solution:

Address	Content
0xffffd6e4	[4] RIP/RA of game
0xffffd6e0	[4] SFP of game
0xffffd6c0	[32] scores [0..7]
0xffffd6bc	[4] val (arg)
0xffffd6b8	[4] idx (arg)
0xffffd6b4	[4] scores (arg)
0xffffd6b0	[4] RIP/RA of set_score
0xffffd6ac	[4] SFP of set_score

Arguments are pushed right-to-left, so from just below **scores**[0..7] they appear in source order: blank (1) at **0xffffd6bc** is **val**, blank (2) at **0xffffd6b8** is **idx**, and blank (3) at **0xffffd6b4** is the **scores** pointer. Below them sit the return address (RIP/RA of **set_score**, **0xffffd6b0**) and its saved frame pointer.

(Question 3 continued...)

Q3.4 (8 points) Provide inputs to the `packet` array that would cause the program to execute shellcode.

In each box, write a single integer in decimal (e.g. `-71`) or hexadecimal (e.g. `0x12345678`).

You must write `0x` in front of any hexadecimal values. Do not worry about little-endianness.

`packet[0] =`

`packet[1] =`

Solution: We overwrite the RIP of `set_score` (`0xffffd6b0`) with the address of our shellcode. `scores[idx]` writes to `0xffffd6c0 + 4*idx`; setting this equal to the RIP address gives

$$0xffffd6c0 + 4 \cdot idx = 0xffffd6b0 \rightarrow 4 \cdot idx = -16 \rightarrow idx = -4,$$

and a signed `-4` (stored as `0xffffffffc`) sails past the `idx > 7` check.

Lay out `packet`, whose first `int` starts at the known address `0xffffcb00`:

- `packet[0] = idx = -4 = 0xffffffffc`, little-endian `'\xfc\xff\xff\xff'`. This passes the `idx > 7` check and selects offset `-4`.
- `packet[1] = val`, the 4 bytes written onto the RIP. The value we want written onto the RIP is `0xdeadbeef`, the address of the pre-placed shellcode.

On Line 6, `scores[-4] = 0xdeadbeef` writes the shellcode address over `set_score`'s RIP. When `set_score` returns, execution jumps to `0xdeadbeef` and runs `SHELLCODE`. This works because the `packet` buffer is executable (NX off) and lives at a fixed address (ASLR off).

Q3.5 (3 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 (with shellcode on a non-executable page)

☒ C ASLR

☒ B Replace Line 2 with `if (idx < 0 || idx > 7)`

☐ D None of the above

Solution:

- NX: `SHELLCODE` sits in the `packet` buffer, which is no longer executable, so the jump to `0xffffcb08` faults. Breaks the exploit.
- ASLR: both `0xffffcb08` (shellcode) and the target RIP/RA address are randomized, so the hardcoded values are wrong. Breaks the exploit.
- Replacing Line 2 with a check that bounds both ends of the index works. The missing piece is the `idx < 0` lower-bound check; the existing `idx > 7` upper bound is already correct for an 8-element array.

(Question 3 continued...)

Q3.6 (2 points) For this subpart only, suppose stack canaries are enabled. Why can the correct exploit (possibly with modifications) still succeed with stack canaries enabled?

- Ⓐ The exploit reads the canary, and overwrites the canary with itself.
- Ⓑ The exploit does not modify the canary, so the canary check passes.
- Ⓒ The exploit jumps to shellcode before the canary is ever checked.

Solution:

The canary is placed between `scores` and the SFP of `game` — at higher addresses than `scores`. Our write goes the other direction (a negative index, to lower addresses) and lands on `set_score`'s RIP, which is below `scores`. The write never crosses any canary, and `set_score` has no local buffer so no canary is inserted in its frame. The corrupted RIP is used before any canary check catches it — the canary does not stop this exploit.

In other words: A stack canary alone was insufficient because canaries only protect the words above a buffer (between the buffer and the saved RIP of the same frame). This attack writes below `scores` to a callee's return address, so it never disturbs the canary — the corruption is detected too late, if at all.

In other words: A canary only detects a write that overflows *upward* through the canary, between a buffer and its own frame's saved registers. The Line 6 write does not pass through `game`'s canary, so the canary is intact and `game`'s check succeeds — the program is redirected without any canary ever being disturbed.

Q4 Memory Safety: Null and Void

(22 points)

Consider the following vulnerable C code:

```
1 void copy_record(void) {
2     char name[64];
3     int len = fread(name, 1, 64, stdin);
4     name[len] = '\0';
5 }
6
7 void vulnerable() {
8     copy_record();
9 }
```

Stack at Line 4

RIP/RA of vulnerable
SFP of vulnerable
(1)
(2)
name (64 bytes)
(3)

Assumptions:

- All memory-safety defenses are disabled.
- You run GDB once and find that:
 - **name** starts at address **0xffffd700**.
 - The SFP of **copy_record** has the **value** **0xffffd748**.
- **fread** returns the number of bytes read.
- Your goal is to place and execute a 44-byte long **SHELLCODE**.

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

Q4.1 Blank (1): **A** RIP/RA of **copy_record** **B** SFP of **copy_record** **C** **len**

Q4.2 Blank (2): **A** RIP/RA of **copy_record** **B** SFP of **copy_record** **C** **len**

Q4.3 Blank (3): **A** RIP/RA of **copy_record** **B** SFP of **copy_record** **C** **len**

Solution:

Address	Content
0xffffd744	[4] RIP of copy_record
0xffffd740	[4] SFP of copy_record
0xffffd700	[64] name
0xffffd6fc	[4] len

name is the highest local, sitting directly below the saved frame pointer at **0xffffd740**; the return address is above that at **0xffffd744**, and **len** sits below **name**. So (1) = RIP of **copy_record**, (2) = SFP of **copy_record**, and (3) = **len**.

(Question 4 continued...)

Q4.4 (2 points) If you input exactly 64 bytes into `fread`, Line 4 writes one byte past the end of `name`.

What is the **value** of the SFP of `copy_record` immediately after Line 4 executes?

- Ⓐ 0xffffd600 Ⓑ 0xffffd700 Ⓒ 0xffffd740 Ⓓ 0xffffd748

Solution: `fread` returns `len = 64`, so `name[64]` writes `\x00` at `0xffffd740`, the least-significant byte of the saved frame pointer. That zeroes only the low byte, turning the stored value from `0xffffd748` into `0xffffd700`.

Q4.5 (8 points) Provide an input to the `fread` on Line 3 that causes 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.

`'A' * 4 + '\x08\x07\xff\xff' + SHELLCODE + 'A' * 12`

Solution: The off-by-one shifts `vulnerable`'s saved frame pointer down to `0xffffd700`, which is `&name[0]`. `copy_record` returns normally (its own RIP at `0xffffd744` is untouched), but `vulnerable` now runs with `EBP = 0xffffd700`. When `vulnerable` executes its `leave; ret:`

- `leave` does `mov esp, ebp` (so `esp = 0xffffd700`) then `pop ebp` (reading `name[0..3]`), leaving `esp = 0xffffd704`.
- `ret` pops the four bytes at `0xffffd704`, i.e. `name[4..7]`, as the return address and jumps there.

So `name[4..7]` is the address the program jumps to. Lay out the 64-byte input:

- `name[0..3]` — 4 filler bytes (`'A' * 4`); consumed by `pop ebp`.
- `name[4..7]` — the fake return address. Point it at the shellcode, which we place at `name[8] = 0xffffd700 + 8 = 0xffffd708`: little-endian `'\x08\x07\xff\xff'`.
- `name[8..51]` — the 44-byte **SHELLCODE**, sitting at `0xffffd708`.
- `name[52..63]` — 12 filler bytes (`'A' * 12`) so the total is exactly 64, which is what makes `fread` return 64 and fire the off-by-one.

When `vulnerable` returns, it jumps to `0xffffd708` and runs **SHELLCODE** (NX is off, so the stack is executable). No byte of the address or shellcode is `\x00`.

Final input: `'A' * 4 + '\x08\x07\xff\xff' + SHELLCODE + 'A' * 12`

(Question 4 continued...)

Q4.6 (3 points) When does the correct exploit start executing instructions in **SHELLCODE**?

- ☐ Ⓐ Immediately after the **fread** on Line 3 returns.
- ☐ Ⓑ Immediately after **name[len] = '\0'** on Line 4 executes.
- ☐ Ⓒ Immediately after **copy_record** returns.
- ☐ Ⓓ Immediately after **vulnerable** returns.

Solution: The off-by-one corrupts only the saved frame pointer, not any return address. **copy_record** returns normally to **vulnerable** using its intact RIP. The damage shows up one frame later: **vulnerable** runs with the shifted **EBP = 0xffffd700**, so its own **leave; ret** reads the fake return address out of **name** and jumps to the shellcode. Control is hijacked when **vulnerable** returns, not **copy_record**.

Q4.7 (3 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:

- NX: the shellcode sits in **name** on the stack; with non-executable pages the jump to **0xffffd708** faults. Breaks the exploit.
- ASLR: the payload hardcodes **0xffffd708** (and relies on the SFP shifting to **0xffffd700**), which change when the stack base is randomized. Breaks the exploit.
- Stack canaries: a canary sits between **name** and the SFP, so **name[64]** now writes **\x00** into the canary, not the SFP. The corrupted canary fails its check when **copy_record** returns, and the program aborts. Breaks the exploit.

(Question 4 continued...)

Q4.8 (3 points) Reminder: all defenses are still disabled.

Suppose that you run the same code on a different machine.
(Assume the program does not run any other code before the provided code.)

On this machine, you find that:

- **name** starts at address **0xffffd780**.
- The SFP of **copy_record** has the **value** **0xffffd7c8**.

As before, Line 4 modifies the value of the SFP of **copy_record**.

Is it possible to execute an off-by-one attack on this machine?

- Ⓐ Yes, because the modified SFP and the address of **name** have the same second-to-last byte.
- Ⓑ Yes, because the modified SFP is an address that points inside the **name** array.
- Ⓒ No, because the modified SFP and the address of **name** have different second-to-last bytes.
- Ⓓ No, because the modified SFP is an address that points far outside the **name** array.

Solution: No. The zeroed SFP **0xffffd700** is **0x80** bytes **below** the start of **name** (**0xffffd780**). When **vulnerable** runs **leave; ret**, it pops EBP from **0xffffd700** and the return address from **0xffffd704**. Both are stale stack memory below **copy_record**'s frame, not bytes the attacker wrote, so no input choice controls where **vulnerable** returns. The attack only works when the zeroed SFP lands inside **name** with room for the fake return address 4 bytes above it, as in the original layout (**0xffffd700 = &name[0]**).

Q5 Authentication: All You Can Answer!**(15 points)**

Each subpart is independent.

Q5.1 (2 points) Let F be a secure pseudorandom function (PRF) on fixed-length inputs, and let H be a collision-resistant hash function with output length equal to F 's input length. Alice and Bob share a random key K .

Which of these is an EUF-CMA secure MAC for arbitrary-length messages M ?

- Ⓐ $H(M)$ Ⓑ $K \oplus H(M)$ Ⓒ $F(K, H(M))$ Ⓓ $H(K) \oplus H(M)$

Solution: $F(K, H(M))$, the hash-then-PRF construction from lecture. H compresses M to a fixed length. A new message has a new hash (collision resistance), and F on that new input gives a random-looking value that cannot be predicted without K .

- $H(M)$ uses no key, so anyone can compute it.
- $K \oplus H(M)$: one observed tag gives Mallory $K = t \oplus H(M)$, and then she can tag anything.
- $H(K) \oplus H(M)$: one observed tag gives Mallory $H(K)$, which is all she needs to forge.

Q5.2 (2 points) In the RSA signature scheme from lecture, Alice's verification key is (N, e) , her signing key is d , and she signs M as $\sigma = H(M)^d \bmod N$.

Bob verifies (M, σ) by checking if the following condition holds:

- Ⓐ $H(M) = \sigma^e \bmod N$ Ⓒ $\sigma = H(M)^d \bmod N$
Ⓑ $\sigma = H(M)^e \bmod N$ Ⓓ $H(M)^e = \sigma^d \bmod N$

Solution: $H(M) = \sigma^e \bmod N$. Signing uses the trapdoor's invert direction (raise to d), and verifying uses the public eval direction (raise to e): $\sigma^e = H(M)^{de} = H(M) \bmod N$. Options using d cannot be checked, since Bob does not know Alice's secret d . Recomputing $H(M)^e$ and comparing it to σ does not match what Alice computed.

(Question 5 continued...)

Q5.3 (5 points) Alice encrypts two equal-length messages M_1 and M_2 with AES-CTR under the same key K , but uses the same nonce r both times.

Eve sees both ciphertexts C_1 and C_2 (and r) but does not know K . Select all pieces of information that Eve can learn.

- | | |
|---|--|
| ☐ A $M_1 \oplus M_2$ | ☐ D M_2 , given that she already knows M_1 |
| ☐ B Whether $M_1 = M_2$ | ☐ E M_2 , without knowing M_1 |
| ☐ C The key K | ☐ F None of the above |

Solution: With the same K and r , both messages are XORed with the same pad P , which is a two-time pad: $C_1 \oplus C_2 = (P \oplus M_1) \oplus (P \oplus M_2) = M_1 \oplus M_2$.

- $M_1 \oplus M_2$: computed directly, as shown above.
- Equality: $C_1 = C_2$ exactly when $M_1 = M_2$. (Reusing the nonce makes the scheme deterministic, which is what IND-CPA forbids.)
- M_2 from M_1 : $M_2 = C_2 \oplus C_1 \oplus M_1$. (Eve can also get the pad, $P = C_1 \oplus M_1$.)
- K : no. Eve learns only PRF outputs (the pad), and a secure PRF/block cipher does not reveal its key from its outputs.

Q5.4 (2 points) Given an IND-CPA secure encryption scheme Enc , and an EUF-CMA secure MAC, which construction is the Encrypt-then-MAC authenticated encryption scheme recommended in lecture?

- | | |
|--|---|
| ☐ A $\text{MAC}(K_1, M) \parallel \text{Enc}(K_2, M)$. | ☒ C $C \parallel \text{MAC}(K_1, C)$, where $C = \text{Enc}(K_2, M)$. |
| ☐ B $\text{Enc}(K_2, M \parallel \text{MAC}(K_1, M))$. | ☐ D $C \parallel \text{MAC}(K, C)$, where $C = \text{Enc}(K, M)$. |

Solution: Encrypt-then-MAC with separate keys: encrypt, then MAC the **ciphertext**. The receiver checks the tag first and decrypts only if it is valid, so it never decrypts attacker-made ciphertext.

- MAC-and-Encrypt ($\text{MAC}(K_1, M) \parallel \text{Enc}(K_2, M)$) is not recommended: EUF-CMA promises nothing about confidentiality, so the tag on M may leak information about M .
- MAC-then-Encrypt has to decrypt before checking the MAC, which exposes the decryption process to attacker-made input.
- Reusing one key for both is a trap. IND-CPA and EUF-CMA each assume their key is used only for that one job, so sharing a key can break both. Lecture says to use a different key for each.

(Question 5 continued...)

Q5.5 (4 points) Alice and Bob run ephemeral Diffie-Hellman to compute a shared key K . Select all true statements.

- ☐ A An eavesdropper can compute K as $g^a + g^b \bmod p$.
- ☐ B An eavesdropper can compute K as $g^a \cdot g^b \bmod p$.
- ☐ C The exchange by itself lets Alice confirm that the party she derived K with is really Bob.
- ☒ D If Alice and Bob generate a fresh a, b for each session and then discard a, b , and K , an attacker who later breaks into both machines cannot decrypt previously recorded sessions.
- ☐ E None of the above

Solution:

- **False:** $g^a + g^b \neq g^{ab}$.
- **False:** $g^a \cdot g^b = g^{a+b}$, not g^{ab} . Getting g^{ab} from g^a and g^b is the Diffie-Hellman problem, which is assumed to be hard.
- **False:** Diffie-Hellman provides no authentication. Alice has no idea who sent g^b . It might have been Mallory, who can run separate exchanges with each side.
- **True:** this is ephemeral DH (DHE), and the property is **forward secrecy**. Once the secrets are deleted, nothing on either machine can rebuild old session keys.

Q6 Authentication: Chain of Command 📄

(13 points)

EvanBot wants to learn the public key of `shop.oski.edu`.

EvanBot trusts the public verification key VK_{Root} as a trust anchor.

`shop.oski.edu` presents the following two certificates to EvanBot:

1. Signed by SK_{Root} : “CA has public key VK_{CA} , and may issue certificates for `*.oski.edu`.”
2. Signed by SK_{CA} : “`shop.oski.edu` has public key VK_{Oski} , and may issue certificates for no one.”

Q6.1 (2 points) What value should EvanBot use to verify the signature in Certificate 2?

- Ⓐ The trust anchor VK_{Root} .
- Ⓑ The value VK_{CA} from Certificate 1.
- Ⓒ The value VK_{Oski} from Certificate 2.

Solution:

Certificate 2 is signed by SK_{CA} , so EvanBot uses the corresponding public key VK_{CA} to verify that signature.

Q6.2 (2 points) EvanBot sees the following certificate:

Signed by SK_{M} : “`dev.oski.edu` has public key VK_{M} , and may issue certificates for no one.”

The signature in this certificate is valid. Can EvanBot trust VK_{M} ?

- Ⓐ Yes, because EvanBot can use VK_{M} in this certificate to verify the signature.
- Ⓑ Yes, because this certificate indicates that CA is endorsing VK_{M} .
- Ⓒ No, because this certificate must be signed by either VK_{Root} or VK_{CA} to be trusted.
- Ⓓ No, because this certificate must be signed by VK_{Root} to be trusted.

Solution: A signature only conveys trust if it chains back to a key already in the root store. A self-signed leaf is endorsed only by itself, so an attacker could trivially generate one for any domain. The cryptography is fine; the problem is that the trust chain terminates at an untrusted key.

(Question 6 continued...)

Q6.3 (3 points) In this subpart, Mallory is an attacker who knows SK_{CA} (but not SK_{Root} or SK_{Oski}).

Mallory's own key pair is $(VK_{Mallory}, SK_{Mallory})$.

Select all attacks that Mallory can carry out.

- ☒ **A** Mallory can forge a certificate binding **bank.oski.edu** to $VK_{Mallory}$ that EvanBot accepts.
- ☒ **B** Mallory can forge a certificate binding **www.google.com** to $VK_{Mallory}$ that EvanBot accepts.
- ☒ **C** Mallory can impersonate **shop.oski.edu** to EvanBot without knowing SK_{Oski} .
- ☐ **D** None of the above

Solution: With SK_{CA} Mallory can create leaf certificates for any name within the intermediate's delegated scope ***.oski.edu**. So a cert for **bank.oski.edu** (option 1) is forgeable and accepted, and Mallory can replace **shop.oski.edu**'s cert with one binding the domain to Mallory's own key $VK_{Mallory}$ (option 3) — she never needs SK_{Oski} , since she issues a fresh key she controls.

www.google.com is outside ***.oski.edu**, so the delegated scope in Certificate 1 makes EvanBot reject it (option 2 false). Compromising a private key gives no way to recover a different private key from a public key (option 4 false). The browser checks only that the chain validates and the name matches; it has no record of what key the real site uses, so a validly signed leaf with a different key raises no warning (option 5 false).

The text on the previous page, reprinted for your convenience:

EvanBot trusts the public verification key VK_{Root} as a trust anchor.

shop.oski.edu presents the following two certificates to EvanBot:

1. Signed by SK_{Root} : "CA has public key VK_{CA} , and may issue certificates for ***.oski.edu**."
2. Signed by SK_{CA} : "**shop.oski.edu** has public key VK_{Oski} , and may issue certificates for no one."

Q6.4 (2 points) In this subpart, suppose Mallory knows SK_{CA} (but not SK_{Root} or SK_{Oski}).

True or false: If we implement a short expiration time on all certificates, then Mallory will become harmless after a short time.

- ☒ **A** True
- ☐ **B** False

Solution:

True, because the certificate endorsing VK_{CA} will expire after a short time. Once the certificate expires, Mallory's stolen key cannot be used to generate any valid certificates.

(Question 6 continued...)

Q6.5 (2 points) In this subpart, suppose Mallory knows SK_{Root} (but not SK_{CA} or SK_{Oski}).

True or false: If we implement a short expiration time on all certificates, then Mallory will become harmless after a short time.

- ☐ A True ☒ B False

Solution:

False, because VK_{Root} is the trust anchor, and is not endorsed in any certificates. Therefore, Mallory can continue issuing malicious certificates for as long as she wants, and certificates expiring do not prevent her attacks.

Q6.6 (2 points) Suppose EvanBot checks for bad certificates by periodically downloading a certificate revocation list (CRL).

Which of these best describes the trade-off between security and performance of using this approach, compared to doing nothing?

- ☐ A Better security, better performance. ☐ C Worse security, better performance.
☒ B Better security, worse performance. ☐ D Worse security, worse performance.

Solution:

Better security because there is now a mechanism for checking for bad certificates (compared to doing nothing).

Worse performance because downloading CRLs can be expensive, e.g. uses lots of time and network bandwidth.

Q7 Cryptography: Maul of the Wild 🛒🇺🇸

(25 points)

Recall the ElGamal encryption scheme from lecture:

- The public parameters are p and g .
- Bob's private key is b , and his public key is $B = g^b \bmod p$.
- To encrypt a message m , the sender picks a fresh random exponent r and computes the ciphertext

$$(c_1, c_2) = (g^r \bmod p, m \cdot B^r \bmod p).$$

- To decrypt, Bob computes $m = c_2 \cdot (c_1^b)^{-1} \bmod p$.

Q7.1 (5 points) For this subpart only, use the following values:

$$p = 23, \quad g = 5, \quad b = 6, \quad B = 5^6 \bmod 23 = 8.$$

Alice encrypts a message m and sends ciphertext $(c_1, c_2) = (19, 7)$ to Bob.

Mallory does not know a or m , but she wants to change the ciphertext to (c'_1, c'_2) , so that when Bob decrypts, Bob receives the message $3m \bmod 23$. What should the tampered ciphertext be?

Write one integer between 0 and 22 in each box.

$c'_1 = \boxed{19}$

$c'_2 = \boxed{21}$

Solution: ElGamal is multiplicatively malleable. Multiplying the second component by a constant k multiplies the underlying plaintext by k , because the first component (and hence the mask A^r that decryption removes) is unchanged:

$$c'_2 = k \cdot c_2 = k \cdot m \cdot A^r \implies c'_2 \cdot (c'_1)^{-1} = k \cdot m \cdot A^r \cdot (A^r)^{-1} = km \bmod p.$$

Here $k = 3$, so Mallory leaves $c'_1 = c_1 = 19$ and sets $c'_2 = 3 \cdot 7 = 21 \bmod 23$. The mauled ciphertext is $(19, 21)$.

Sanity check: $c_1 = 19 = 5^{15} \bmod 23$, so $r = 15$ and $A^r = 8^{15} \equiv 2 \bmod 23$; thus $m \equiv 7 \cdot 2^{-1} \equiv 7 \cdot 12 \equiv 15 \bmod 23$. The mauled ciphertext decrypts to $21 \cdot 12 \equiv 252 \equiv 22 \equiv 3 \cdot 15 \bmod 23 = 3m$. It works, and Mallory never learned $m = 15$.

(Question 7 continued...)

Q7.2 (2 points) Alice encrypts two messages:

- She encrypts m_1 and sends ciphertext (c_1, c_2) to Bob.
- She encrypts m_2 and sends ciphertext (d_1, d_2) to Bob.

Mallory intercepts both ciphertexts. Which tampered ciphertext decrypts to $m_1 \cdot m_2 \bmod p$?

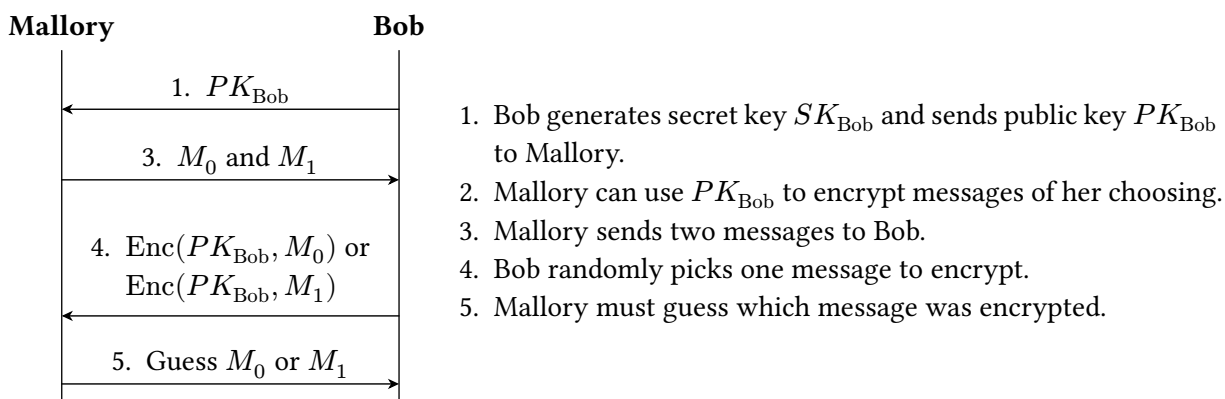
- ☒ A $(c_1 \cdot d_1 \bmod p, c_2 \cdot d_2 \bmod p)$
☐ B $(c_1 \cdot d_2 \bmod p, c_2 \cdot d_1 \bmod p)$
☐ C $(c_1 + d_1 \bmod p, c_2 + d_2 \bmod p)$
☐ D $(c_1^{d_1} \bmod p, c_2^{d_2} \bmod p)$

Solution: Write the two ciphertexts with exponents r and s : $(g^r, m_1 A^r)$ and $(g^s, m_2 A^s)$. Multiplying componentwise gives

$$(g^r \cdot g^s, m_1 A^r \cdot m_2 A^s) = (g^{r+s}, (m_1 m_2) \cdot A^{r+s}),$$

which is exactly a valid ElGamal encryption of $m_1 m_2$ under the fresh exponent $r + s$. This componentwise product is the multiplicative homomorphism of ElGamal, and it needs neither a nor either plaintext.

Recall the IND-CPA game for public-key encryption schemes (from lecture):



Q7.3 (4 points) Mallory tries several strategies to win the IND-CPA game against ElGamal.

For all of her strategies in this subpart:

- In Step 2: Mallory uses PK_{Bob} to compute the encryptions of two messages x and $2x \bmod p$.
- In Step 3: Mallory sends $M_0 = x$ and $M_1 = 2x \bmod p$ to Bob.
- In Step 5: *see answer choices*

Which strategies allow Mallory to win the IND-CPA game with probability $> 0.5 + \text{negligible}$?
Select all that apply.

- ☐ A Use Bob's encryption in Step 4 to compute the r he used. Then, use r to compute the shared secret $B^r = g^{br} \bmod p$ and decrypt the message that Bob chose.
- ☐ B If Bob's encryption in Step 4 matches Mallory's encryption of x in Step 2, guess M_0 .
Else, guess M_1 .
- ☐ C If Bob returns a value $\geq p/2$, guess that $M_1 = 2x \bmod p$ was encrypted.
Else, guess that $M_0 = x$ was encrypted.
- ☐ D Flip a fair coin and guess M_0 and M_1 at random.
- ☒ E None of the above

Solution:

A: Fails because the discrete log problem makes it hard to find r from the ciphertext.

B: Fails because Bob uses a random r each time, so his encryption in Step 4 does not match Mallory's encryptions in Step 2.

C: Fails because encrypting larger values does not necessarily result in larger ciphertexts.

D: This strategy succeeds 50% of the time, which does not exceed $50\% + \text{negligible}$.

Q7.4 (4 points) Mallory tries another strategy to win the IND-CPA game against ElGamal:

- In Step 3: Mallory sends two randomly-generated messages M_0 and M_1 to Bob.
- In Step 5: Use Bob's encryption in Step 4 to compute the r he used. Then, use r to compute the shared secret $B^r = g^{br} \bmod p$ and decrypt the message that Bob chose.

Does this allow Mallory to win the IND-CPA game with probability $> 0.5 + \text{negligible}$?

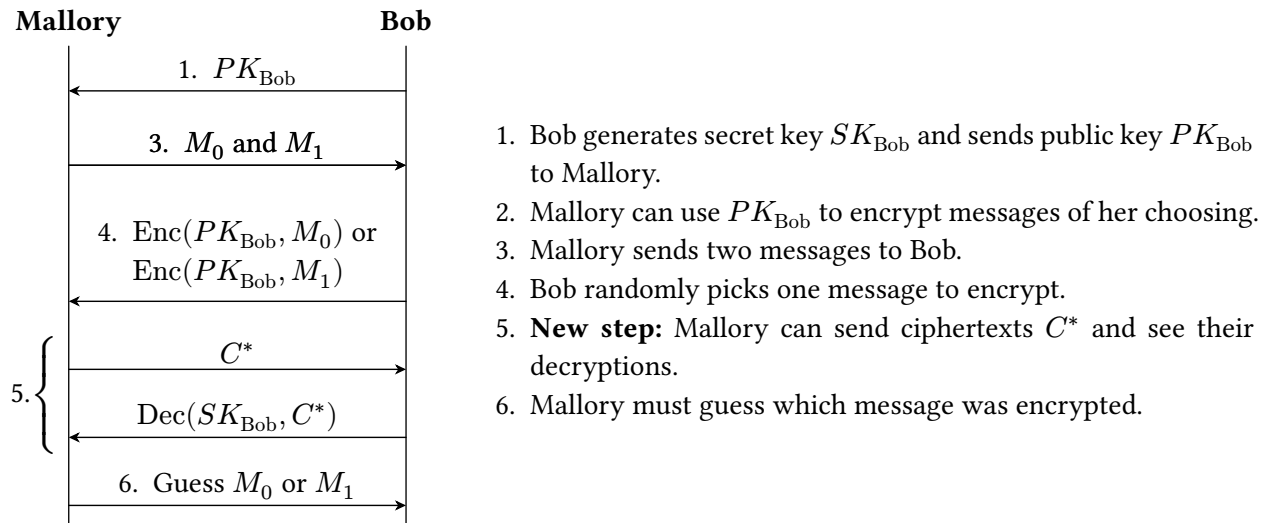
- ☒ A Yes ☐ B No

Explain why or why not in 20 words or fewer.

Mallory cannot compute r because discrete log is hard.

Now, suppose we modify the IND-CPA game so that Mallory can ask for *decryptions* of arbitrary ciphertexts of her choosing:

(Question 7 continued...)



Q7.5 (2 points) Suppose the game allows C^* to be any message, including the ciphertext sent in Step 4. Why would this not be a useful game for analyzing encryption schemes?

- (A) Decrypting the ciphertext from Step 4 always reveals SK_{Bob} to Mallory.
 (B) Decrypting the ciphertext from Step 4 always reveals the scheme's randomness to Mallory.
 (C) Mallory has a winning strategy for every encryption scheme.
 (D) Mallory never has a winning strategy for any encryption scheme.

Solution: If the challenge could be decrypted, Mallory would just ask for it, read m_b , and win with probability 1 against *any* scheme. Banning that one query is the smallest restriction that keeps the game meaningful.

Now, suppose C^* can be any message, *except* the ciphertext sent in Step 4.

Mallory has a strategy to win this game against ElGamal with 100% probability.

First, in Step 3, Mallory sends two randomly-generated messages M_0 and M_1 to Bob.

Q7.6 (4 points) In Step 5, what ciphertext should Mallory ask Bob to decrypt?

Your answer can be in terms of (c_1, c_2) , the encryption Bob returns in Step 4.

Your answer cannot be exactly (c_1, c_2) .

$$\left(\begin{array}{c} \text{ } \end{array} \right),$$

Solution: See solution to next subpart.

(Question 7 continued...)

Q7.7 (4 points) In Step 6, what strategy does Mallory use to win the game?

Your answer can use M_0 , M_1 , and M^* (the decryption Bob returns in Step 5).

A completely unrelated sample answer: “If $M_0 + M^* = M_1$, guess M_0 . Else, guess M_1 .”

If $M_0 \times 2 = M^*$, guess M_0 . Else, guess M_1 .

Solution: Mallory mauls the challenge so that it is no longer the challenge but still decrypts to a known function of m_b . Doubling the second component gives $(19, 2 \cdot 7 \bmod 23) = (19, 14)$. Since $(19, 14) \neq (19, 7)$, the oracle must decrypt it, and it returns $2m_b \bmod 23$.

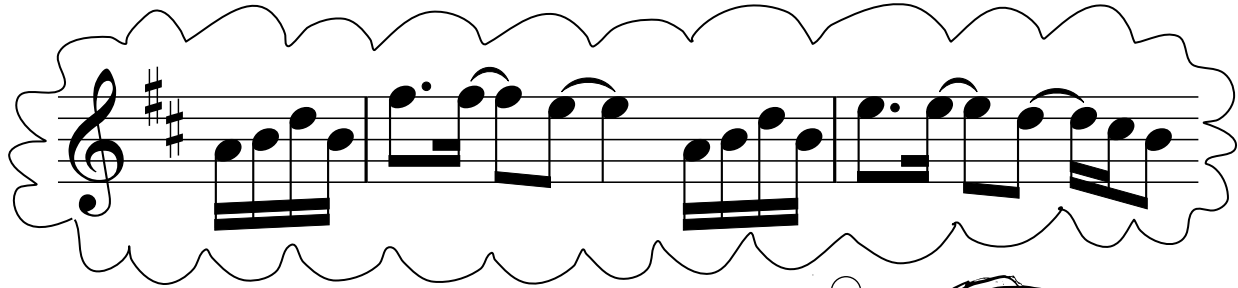
The oracle returns 7. Mallory compares this with the two possibilities: $2m_0 = 30 \equiv 7 \bmod 23$ and $2m_1 = 8 \bmod 23$. It matches $2m_0$, so $b = 0$. (Equivalently, she can multiply by $2^{-1} \equiv 12$ to get $7 \cdot 12 \equiv 15 = m_0$.)

This works for any challenge: multiply c_2 by a known $k \neq 1$, query, and divide by k . Mallory wins every time, so ElGamal is not secure once the adversary has a decryption oracle, even though re-encrypting and comparing fails.

Nothing on this page will affect your grade.

Post-Exam Activity: Earworm

Got a song stuck in your head? If you let Eve eavesdrop a few bars, maybe she can figure out what it is!



Comment Box

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