CS 161 Spring 2023 — Introduction to Computer Security — Midterm

Last updated: March 13, 2023

PRINT your name: ______________________, ______________________
(last) (first)

PRINT your student ID: ______________________________

There are 7 questions of varying credit (150 points total).

Question:	1	2	3	4	5	6	7	Total
Points:	2	28	23	23	24	28	22	150

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

○ Unselected option (completely unfilled)

● 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)

Pre-exam activity (not graded, just for fun):
Look, a ~~shooting star~~ deorbited satellite! What will you wish for?

____

Q1 *Honor Code* (2 points)

Read the following honor code and sign your name.

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.

SIGN your name: ______________________

Q2 *True/False* (28 points)

Each true/false is worth 2 points.

For the next 2 subparts: Tired of biased referees, the Caltopia Sports Association (CSA) is now using EvanBot to officiate the upcoming games.

Q2.1 All changes to EvanBot's programming are logged in a secure file for auditing.

TRUE or FALSE: This is an example of detecting if you can't prevent.

[✓ (A) TRUE] (B) FALSE

Solution: True. If an attacker changes EvanBot's behavior, even if we can't prevent it, we can at least log evidence of the attack happening so that we can recover from the attack later.

Q2.2 CSA officials and team coaches are given full read/write access to EvanBot's programming to audit and ensure fairness.

TRUE or FALSE: This is an example of separation of responsibility.

(A) TRUE [✓ (B) FALSE]

Solution: False. Separation of responsibility involves requiring two parties to collaborate to exercise a privilege. There is no such case of two parties collaborating here.

The most relevant principle here is actually least privilege; officials and coaches probably don't need full read/write access to Bot's programming for auditing. Read access would be sufficient.

For the next 2 subparts: You run `x/4wx buf` and receive the following GDB output:

```
0xffff1244: 0x00ab0000 0x00000000 0x00000033 0xff000000
```

Q2.3 TRUE or FALSE: Byte `0xab` appears at address `0xffff1246` in memory.

[✓ (A) TRUE] (B) FALSE

Solution: True. Recall that the system is little-endian, so the least-significant byte (right-most two digits) of `0x00ab0000` are stored in the lowest address of memory, and the most-significant byte (left-most two digits) are stored in the highest address in memory.

The address of the word `0x00ab0000` is `0xffff1244`, and `0xab` is the third-least-significant byte in this word.

Q2.4 Suppose `buf` is a local variable defined in function `foo`.

TRUE or FALSE: It is possible that the address of `foo`'s RIP is `0xffff1260`.

[✓ (A) TRUE] (B) FALSE

Solution: True. From this output, we can conclude that `buf` is at address `0xffff1244`. It's plausible that the address of this stack frame's RIP is slightly higher up at `0xffff1260`.

Q2.5 Recall the off-by-one exploit from the project.

TRUE or FALSE: Without any modifications, this exploit will crash the program if stack canaries are enabled.

[✓ (A) TRUE] (B) FALSE

Solution: True. The exploit will overwrite the byte directly above the buffer, which used to be part of the SFP but would not be part of the canary. Changing the canary will crash the program.

Q2.6 An attacker writes `"rm -rf /"` into memory and calls the `system` function with this string as an argument.

TRUE or FALSE: Non-executable pages will stop this attack because `"rm -rf /"` was written to a non-executable part of memory.

(A) TRUE [✓ (B) FALSE]

Solution: False. `"rm -rf /"` is not being interpreted as executable instructions in C memory; it's a string being passed to the shell.

For the next 2 subparts: Alice and Bob share a symmetric key K not known to anyone else. Alice sends these two values to Bob:

$$M \text{ and } \mathsf{SHA\text{-}2}(M \| K)$$

Q2.7 TRUE or FALSE: A MITM can modify the message M to be some arbitrary amount of their choosing, without being detected.

(A) TRUE [✓ (B) FALSE]

Solution: Although hashes in general don't provide integrity, this particular setup is not trivially forgeable. The main attack we've talked about in this class regarding SHA-2 is a length extension attack, but this attack would not be useful here, because an extended hash output like $\mathsf{SHA\text{-}2}(M\|K\|M')$ would not be a valid tag for any message.

Q2.8 True or False: A MITM can modify the message M to be some different amount (not necessarily of their choosing), without being detected.

○ (A) True ● (B) False

Solution: As in the previous subpart, there is no way for an attacker to exploit the length extension attack we've seen in order to forge a valid tag for any message that's different from M.

Also, note that the question mentions that Alice is sending just one message to Bob, so there is also no way to replay an old message hashed with K.

For the next 3 subparts: You perform a Diffie-Hellman key exchange with EvanBot over an insecure channel. You use a randomly-generated secret a, but EvanBot uses $b = 0$.

Q2.9 True or False: You and EvanBot will derive the same shared secret.

● (A) True ● (B) False

Solution: The intended answer was true. You derive $(g^0)^a = g^0 = 1 \bmod p$. EvanBot derives $(g^a)^0 = g^0 = 1 \bmod p$.

However, if you assumed that there was a MITM in the insecure channel, then it's possible that they interfere with your exchange and cause you and EvanBot to derive different secrets. If you made this assumption, then the answer would be false.

Because we weren't clear about whether a MITM is present in this question, we'll accept both True and False as correct answers.

Q2.10 True or False: An eavesdropper on the insecure channel can derive the shared secret.

● (A) True ○ (B) False

Solution: True. As seen in the previous subpart, the shared secret is always going to be $1 \bmod p$.

Q2.11 True or False: An eavesdropper on the insecure channel can learn your shared secret a.

Clarification during exam: This question should read "An eavesdropper on the insecure channel can learn your ~~**shared**~~ secret a."

○ (A) True ● (B) False

Solution: False. The eavesdropper see $g^0 = 1 \bmod p$ and $g^a \bmod p$ exchanged over the channel. However, because of the discrete-log problem, they cannot derive a from $g^a \bmod p$.

Q2.12 Consider a world where everyone uses Diffie-Hellman to exchange shared secrets, and then uses those shared secrets to communicate with only symmetric-key cryptography.

TRUE or FALSE: In this world, there would be no benefit to introducing certificates.

○ (A) TRUE [✓ (B) FALSE]

Solution: False. In this world, you have no idea who you're talking to in any communication– you could have done that Diffie-Hellman exchange with an attacker (e.g. if the exchange was MITM'd). Certificates would solve this problem by providing signed attestations of the identity of who you're communicating with.

For the next 2 subparts: Consider the following certificate tree:

1. EvanBot (root of trust)
2. TAs (certificates signed by EvanBot)
3. Readers (certificates signed by a TA)

Q2.13 TRUE or FALSE: If an attacker compromises the private key of a reader, they can create a valid certificate endorsing the attacker as a reader.

○ (A) TRUE [✓ (B) FALSE]

Solution: The attacker would need to compromise the private key of TA in order to sign a certificate endorsing the attacker as a reader. In this certificate tree, the private key of a reader cannot be used to sign any certificates.

Q2.14 TRUE or FALSE: If the TA certificates issued by EvanBot expire every 24 hours, then both the TAs and readers need to renew their certificates every 24 hours. Assume that renewing a certificate involves creating a new public/private keypair.

[✓ (A) TRUE] ○ (B) FALSE

Solution: When the TA certificates expire, they need to generate new public keys. This means the certificates on readers' public keys are now invalid as well (because they were signed by the old, expired TA public keys), so the readers also need new certificates signed by the new TA keys.

Q3 *IND-CPA and Block Ciphers: evanbotevanbotevanbotevan...* (23 points)

Consider a PRNG whose outputs repeat in a cycle of length 6:

$$\ldots 1, 5, 4, 6, 2, 3, 1, 5, 4, 6, 2, 3, 1, 5, 4, 6, 2, 3, 1, 5, \ldots$$

Let r_i denote the ith output from the PRNG. We don't know the initial seed of this PRNG, but we do know the first output (r_0) will be an integer between 1 and 6 (inclusive). After initially seeding or reseeding, the next output has an equal probability of being any of the 6 numbers.

Consider using this PRNG to build an encryption scheme:

$$C_i = E_K(H(r_i)) \oplus M_i$$

For each block we need to encrypt, we generate the next output r_i from the PRNG, and compute C_i using the equation above, where E_K denotes AES encryption.

Here's an example of this encryption scheme. If the first PRNG output will be 1, and we encrypt a 3-block message (M_0, M_1, M_2), then the ciphertext (C_0, C_1, C_2) would be computed as follows:

$$C_0 = E_K(H(1)) \oplus M_0$$
$$C_1 = E_K(H(5)) \oplus M_1$$
$$C_2 = E_K(H(4)) \oplus M_2$$

Q3.1 (3 points) Write the decryption formula for M_i.

Solution: $M_i = C_i \oplus E_K(H(r_i))$

One simple way to derive this decryption equation is to XOR both sides with $E_K(H(r_i))$.

Q3.2 (4 points) Select all true statements about this scheme.

Clarification during exam: P_i refers to M_i (the ith plaintext block) for this subpart.

■ (A) If an attacker flips the last bit of C_i, then the last bit of P_i will also be flipped.

☐ (B) If an attacker flips the last bit of C_i, then P_i will become random-looking garbage.

☐ (C) If an attacker flips the last bit of C_{i-1}, then the last bit of P_i will also be flipped.

■ (D) If an attacker flips every bit of C_i, then every bit of P_i will also be flipped.

☐ (E) None of the above

Solution: Note that XOR works bitwise here: each bit of C is XORed with each bit of the block cipher output $E_K(H(r_i))$ to compute each bit of P. Therefore, an attacker who flips any bit of C will also cause the corresponding bit of P to be flipped.

Q3.3 (3 points) For any two plaintext blocks M_i and M_j, which of the following conditions would cause the corresponding ciphertext blocks to be equal ($C_i = C_j$)?

Clarification during exam: Select the least restrictive condition that would cause the corresponding ciphertext blocks to be equal.

○ (A) $M_i \neq M_j$, and $i - j$ is a multiple of 6.

○ (B) $M_i \neq M_j$, and $i - j$ is exactly 6.

[✓ (C) $M_i = M_j$, and $i - j$ is a multiple of 6.]

○ (D) $M_i = M_j$, and $i - j$ is exactly 6.

○ (E) There is no way to deduce that $C_i = C_j$.

Solution: For any two plaintext blocks M_i and M_j, if $i - j \equiv 0 \bmod 6$ and $M_i = M_j$, then $C_i = C_j$.

The block cipher output repeats every 6 blocks, so if two plaintext blocks are a multiple of 6 blocks away from each other, they will be encrypted by being XORed with the same block cipher output block. If the two plaintext blocks are equivalent, and they get XORed with the same block cipher output, then the two ciphertext blocks will also be equivalent.

We can show this by writing an example:

$$C_0 = E_K(H(r_0)) \oplus M_0$$
$$C_6 = E_K(H(r_6)) \oplus M_6$$

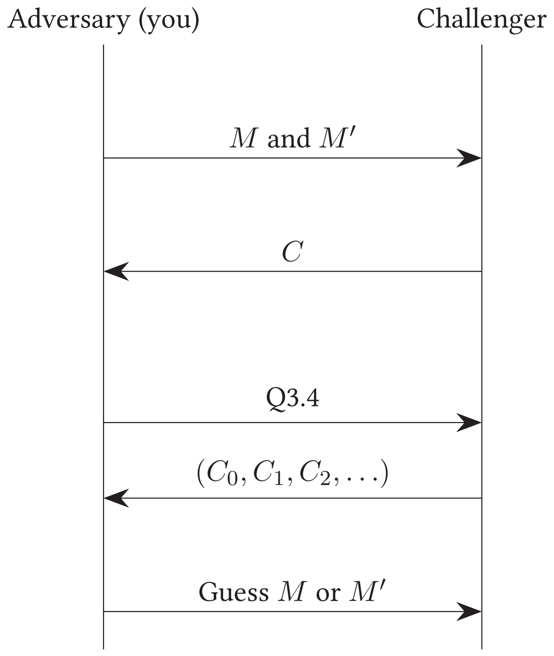
But we know that $r_0 = r_6$ because of the PRNG's repeating output. (If you're not convinced, check out the example PRNG output at the top of the question.) Therefore, the second equation could be rewritten using r_0 to be:

$$C_0 = E_K(H(r_0)) \oplus M_0$$
$$C_6 = E_K(H(r_0)) \oplus M_6$$

Now we can see that if $M_0 = M_6$, then $C_0 = C_6$.

Generalizing this, there was nothing special about using the indices 0 and 6, except for the fact that they were a multiple of 6 apart (so that we could apply the $r_6 = r_0$ property of the PRNG).

To show this scheme is insecure, you want to provide a strategy that always wins the IND-CPA game.



Challenge phase:
First, the adversary (that's you!) sends two different challenge messages, $M \neq M'$, to the challenger. For your strategy, you can assume **both M and M' are one block long**.

The challenger randomly picks one of M or M' to encrypt. The resulting **one-block ciphertext** C is returned to you.

Query phase:

You send **one** multi-block query message to the challenger...

...and you receive the encryption of that query message.

Finally, you will guess whether M or M' was sent (see Q3.5).

Q3.4 (3 points) In general, you can ask the challenger to encrypt multiple query messages in the query phase. However, in your strategy, you only need to ask the challenger to encrypt **one** query message.

Which of these choices of query message would work in your strategy? Select all that apply.

(Note: 0 is a block of all zeroes.)

☐ (A) (M, M')

[✓ (B) $(0, 0, 0, 0, 0, 0)$]

☐ (C) $(0, M, M')$

☐ (D) $(0, 0, 0, 0)$

[✓ (E) (M, M, M, M, M, M)]

[✓ (F) (M', M', M', M', M', M')]

Solution: Suppose the challenger randomly encrypts M. Then the returned one-block ciphertext will be $C = E_K(H(r_0)) \oplus M$. The only value that we don't know here is $E_K(H(r_0))$, because the block cipher output is unpredictable without K.

However, we can at least narrow down the possible values. Note that there are only 6 possible outputs of the block cipher: $E_K(H(1)), E_K(H(2)), \ldots, E_K(H(6))$. If we know all six of these values, then we can XOR the ciphertext with these values to see if any of the resulting plaintext values matches M.

To determine what to send in our query phase, note that any message that is at least 6 blocks will let us figure out $E_k(H(i))$ for all $1 \leq i \leq 6$. We'd just need to XOR out our message from the returned ciphertext to receive the block cipher outputs.

For example, if we send $(1, 1, 1, 1, 1, 1)$, we receive $E_K(H(i)) \oplus 1$ for all $1 \leq i \leq 6$. Then we can XOR out the 1 to receive $E_K(H(i))$ for all $1 \leq i \leq 6$.

Now, you just have to compute $E_K(H(i)) \oplus M$ for all $1 \leq i \leq 6$. If the ciphertext matches one of the 6 messages you computed, then M was encrypted; otherwise, M' was encrypted.

Q3.5 (6 points) Describe the strategy that you will use to win the IND-CPA game.

In the query phase, the message you sent to the challenger is:

(Write one of the answers you selected in the previous subpart. All the answers you selected should work, but just pick one here to use in your strategy.)

Solution: You can write any of the options you selected from the previous subpart in this box, and then use it in your strategy below. Writing a valid answer here is not worth credit; it's just for reference of what you plan to use in your strategy later.

Note that the three correct answer choices lead to cleaner strategies for winning the IND-CPA game (as we'll show below).

Sending $(0, 0, 0, 0, 0, 0)$ will give us back $E_K(H(i))$ for $1 \leq i \leq 6$, because XORing the block cipher output with 0 doesn't change it.

Sending (M, M, M, M, M, M) will give us back $E_K(H(i)) \oplus M$ for $1 \leq i \leq 6$. This exactly corresponds to the six possible ways that M could have been encrypted.

Likewise, sending (M', M', M', M', M', M') will give us back the six possible ways that M' could have been encrypted.

The challenger encrypts the message you wrote in the box above and returns $(C_0, C_1, C_2, \ldots)$.

Explain how you would determine whether M or M' was encrypted, using:

- M and M' from the challenge phase,
- the encryption C from the challenge phase (C is either the encryption of M or M'), and
- the encryption of the message you wrote in the box above $(C_0, C_1, C_2, \ldots)$.

An example of how you could describe your strategy, that has nothing to do with this question: If C equals $C_1 \oplus 161$ or $C_2 \oplus C_5$, guess M. Else, guess M'.

Solution: Example answer 1:

First box: $(0, 0, 0, 0, 0, 0)$

Strategy: If C matches one of $C_0 \oplus M, C_1 \oplus M, \ldots, C_6 \oplus M$, guess M. Else, guess M'.

Explanation: In this strategy, the query phase returns $(C_0, \ldots, C_6) = (E_K(H(r_0)), \ldots, E_K(H(r_6)))$, i.e. all six possible block cipher outputs. We know that M or M' was XORed with one of these six block cipher outputs, so we just have to XOR M with each of these six possible outputs and see if any of them match the ciphertext given back to us.

See below for example answers 2 and 3.

Q3.6 (4 points) In general, what must be true about the query message (from Q3.4) in order for this strategy to work? You can answer in 10 words or fewer.

Solution: It must be 6 blocks or longer.

In short, you need at least 6 blocks in your query messages, so that from the returned ciphertext, you can extract all six possible block cipher outputs: $E_K(H(1)), E_K(H(2)), \dots, E_K(H(6))$.

See previous subparts for more details.

Note: In this question, it actually doesn't matter whether or not the PRNG is reseeded after every encryption. Regardless of the PRNG's state at the start of the query phase, any query input that's at least 6 blocks will run through a full PRNG cycle and yield all 6 possible block cipher outputs.

Solution:

Example answer 2:

First box: (M, M, M, M, M, M)

Strategy: If C matches one of $C_0, C_1, \dots, C_6$, guess M. Else, guess M'.

Explanation: In this strategy, the query phase returns

$$(C_0, \dots, C_6) = (E_K(H(r_0)) \oplus M, \dots, E_K(H(r_6)) \oplus M),$$

i.e. the six possible ways in which M could have been encrypted. We just have to check if the ciphertext matches one of the six possible encryptions of M to see if M was encrypted; if the ciphertext doesn't match any of the six possible encryptions, then M' must have been encrypted instead.

Example answer 3:

First box: (M', M', M', M', M', M')

Strategy: If C matches one of $C_0, C_1, \dots, C_6$, guess M'. Else, guess M.

Explanation: This is the same strategy as the second answer, except we've reversed the roles of M' and M. We receive the six possible ways where M' could have been encrypted, and see if the ciphertext C matches one of those six possible encryptions. If we get a match, we know M' must have been encrypted; otherwise, we know M must have been encrypted.

Q4 *Public-Key Cryptography: Mallory Forger* (23 points)

Alice wants to securely send a number M to the bank, to tell the bank to send M dollars to Bob. However, Mallory might try to read or tamper with M.

Assumptions:

- Both the bank and Alice have published, trusted RSA public keys denoted PK_{bank} and PK_{alice}, respectively.
- There are too many possible values of M for Mallory to try them all in a brute-force attack.
- Alice sends exactly one message to the bank.

Alice and her bank decide to use the RSA encryption scheme from class (no OAEP padding):

- $\mathsf{Enc}(PK, M) = M^e \bmod N$
- $\mathsf{Dec}(SK, C) = C^d \bmod N$

e, N are from the RSA public key PK, d is the RSA private key SK, and C is the ciphertext to be decrypted.

Clarification during exam: Enc refers to the RSA encryption scheme from the class (no OAEP padding).

Scheme 1: Alice sends $\mathsf{Enc}(PK_{\text{bank}}, M)$.

Q4.1 (2 points) Can Mallory learn the value of M?

(A) Yes [✓ (B) No]

Solution: Without knowing the bank's secret key, Mallory cannot decrypt this message.

A single RSA encryption does not leak information about the message. Mallory would need to solve the factoring problem (which is assumed to be hard) to try and learn information about this message without the secret key.

(If we used this scheme to send multiple messages, then the deterministic nature of RSA would leak information, but the question says Alice only sends one message.)

Q4.2 (2 points) Can Mallory change M without being detected?

[✓ (A) Yes, Mallory can change M to any value of her choosing.]

(B) Yes, but Mallory can only change M to a specific value (not necessarily of her choosing).

(C) No, Mallory cannot change M.

Solution: In this case, nothing is stopping Mallory from computing $\mathsf{Enc}(PK_{\text{bank}}, M')$ by herself, since the bank's public key is known by all, and anybody can encrypt messages with the public key. Therefore, Mallory can cause the bank to send an arbitrary amount of money.

Scheme 2: Alice sends $\mathsf{Enc}(PK_{\text{bank}}, M)$ and $H(M)$.

Q4.3 (2 points) Can Mallory learn the value of M?

○ (A) Yes [✓ (B) No]

Solution: The only difference in this scheme is the addition of the hash. Since this is a secure cryptographic hash, there is no way to take $H(M)$ and learn the value of M.

Also, note that Mallory cannot brute-force all values of M, as stated in the question assumptions.

Q4.4 (2 points) Can Mallory change M without being detected?

[✓ (A) Yes, Mallory can change M to any value of her choosing.]

○ (B) Yes, but Mallory can only change M to a specific value (not necessarily of her choosing).

○ (C) No, Mallory cannot change M.

Solution: The addition of the hash does not stop the attack from Scheme 1.

Mallory can compute $H(M')$ herself as well and attach it to the forged message with no problems.

Alice and her bank decide to also use the **simplified** RSA signature scheme from class. This is similar to the full RSA signature scheme from class, except the message M is not being hashed:

- $\mathsf{Sign}(SK, M) = M^d \bmod N$
- $\mathsf{Verify}(PK, S, M)$: check if $S^e \equiv M \bmod N$

e, N are from the RSA public key PK, d is the RSA private key SK, and S refers to a signature output by Sign.

Scheme 3: Alice sends $C = \mathsf{Enc}(PK_{\text{bank}}, M)$ and $S = \mathsf{Sign}(SK_{\text{alice}}, M)$.

Q4.5 (5 points) Can Mallory learn the value of M?

[✓ (A) Yes] (B) No

Describe (in words or equations) how Mallory learns M, or explain why Mallory cannot learn M.

Solution: Mallory can compute $S^e \equiv M \bmod N$.

Remember that in discussion, we showed how RSA signatures without signatures is vulnerable to recovery of the original message. Intuitively, in $\bmod N$ space, raising something to the d power (secret key) "cancels out" raising something to the e power (public key), and vice-versa. Formally, $M^{ed} \equiv M \bmod N$.

In this simplified digital signature scheme, signing the message involves raising the message to the d power. Mallory can "cancel out" this effect and retrieve M by raising the signature to the e power (and Mallory knows e because it's part of the public key). Formally, the signature generated is:

$$S = \mathsf{Sign}(SK_{alice}, M) = M^d \bmod N$$

Mallory can compute:

$$S^e \equiv (M^d)^e \bmod N \equiv M \bmod N$$

which gives Mallory the value of M.

Q4.6 (5 points) Can Mallory change M without being detected?

- ○ (A) Yes, Mallory can change M to any value of her choosing.
- [✓ (B) Yes, but Mallory can only change M to a specific value (not necessarily of her choosing).]
- ○ (C) No, Mallory cannot change M.

Describe (in words or equations) how Mallory changes M (and possibly S), or explain why Mallory cannot change M.

Solution: Like in discussion, Mallory can fix some S' and find $(S')^{e_a} = M'$. She can then set $C = \mathsf{Enc}(PK_{\text{bank}}, M')$.

Alternate solution: Mallory can change C to C^2 and S to S^2.

Mallory cannot create a signature on any arbitrary message M', because Mallory does not have Alice's secret signing key.

However, let's take a closer look at the RSA encrypted message:

$$C = M^e \bmod N$$

(Here, e belongs to the bank's public key.) Mallory could square this value and obtain

$$C^2 = M^{2e} \bmod N$$

which is the encryption of the modified message M^2. This is similar to the El Gamal malleability attack we showed in lecture (where we multiplied the ciphertext by 2, which also multiplied the plaintext by 2).

Next, let's take a closer look at the RSA signature:

$$S = M^d \bmod N$$

(Here, d belongs to Alice's secret key.) Without knowing d, Mallory can compute

$$S^2 = M^{2d} \bmod N$$

which is a valid signature on the message M^2.

Scheme 4: Alice sends $C = \mathsf{Enc}(PK_{\text{bank}}, M)$ and $S = \mathsf{Sign}(SK_{\text{alice}}, C)$.

Q4.7 (2 points) Can Mallory learn the value of M?

○ (A) Yes [✓ (B) No]

Solution: If Mallory tries the same attack from the previous scheme (raising the signature to the e power), Mallory can recover $C = (m^{e_{bank}} \mod N_{bank}) \mod N_{alice}$. However, this does not help Mallory learn M. As discussed in Scheme 1, given the RSA encryption of M, Mallory cannot learn M.

Note that $S = \mathsf{Sign}(SK_{\text{alice}}, C) = (m^{e_b})^{d_a} \neq m \mod N$.

Q4.8 (3 points) Can Mallory change M without being detected?

○ (A) Yes, Mallory can change M to any value of her choosing.

[✓ (B) Yes, but Mallory can only change M to a specific value (not necessarily of her choosing).]

○ (C) No, Mallory cannot change M.

Solution: Both the normal and alternate solution attack from the previous scheme still works here. If we tried the first solution, we could recover some C' and send that to the bank, but cannot control C'.

Alternate solution:

The bank's encrypted message is $C = M^{e_{\text{bank}}} \mod N$. Mallory can square this value and obtain $C^2 = M^{2e_{\text{bank}}} \mod N$, the encryption of M^2.

The signature is $S = C^{d_{\text{alice}}} \mod N$. We can plug in C from the previous part to see that $S = M^{e_{\text{bank}} d_{\text{alice}}} \mod N$. If we square this value, we obtain $S^2 = M^{2e_{\text{bank}} d_{\text{alice}}} \mod N$. As before, this is a valid signature on the message M^2.

Note: Be careful in this solution explanation not to confuse the bank and Alice's public/private key pairs. The bank has its own key pair with its own e_{bank} and d_{bank}. Alice also has her own key pair with her own e_{alice} and d_{alice}, which are not the same as the bank's values.

Q5 *Passwords and Integrity: alice161* (24 points)

Consider a password storage server, with the following assumptions:

- There are N users who each choose from a common pool of N possible **alphanumeric** passwords.
- Usernames are unique. Passwords may not be unique.

Consider an attacker who is able to:

- Compute $O(N)$ hashes
- Read and modify what is stored in the password database
- Perform offline brute-force attacks (but not online attacks)

Clarification during exam: Enc refers to an IND-CPA secure symmetric key encryption scheme.

Clarification during exam: Assume that an attack is possible only if an attacker can do it without being detected by the server.

Scheme 1: For each user, the server stores the username and these two values:

$$\mathsf{H}(\texttt{password}) \text{ and } \mathsf{Sign}(SK, \mathsf{H}(\texttt{password}))$$

Sign is a secure digital signature scheme, and SK is a key known only by the server.

Q5.1 (3 points) In Scheme 1, is the attacker able to determine all pairs of users who share the same password?

[✓ (A) Yes, without computing any hashes]

(B) Yes, but only by computing $O(N)$ hashes

(C) No

Solution: The passwords aren't salted and hashes are deterministic, so the attacker can simply look at which users have matching hash outputs.

Q5.2 (3 points) In Scheme 1, how many users' passwords is the attacker able to learn?

(A) None (B) Only one [✓ (C) All of them]

Solution: The passwords aren't salted, so the attacker only needs to do one $O(N)$ dictionary attack to learn all possible password hashes.

Then, for a given password hash, the attacker just has to look up that hash in their list of hashed passwords to determine the original plaintext password.

Q5.3 (3 points) In Scheme 1, what is the attacker able to do to a specific user's password?

[✓ (A) Change the password to a specific value (not necessarily of the attacker's choosing)]

○ (B) Change the password to any value of the attacker's choosing

○ (C) Nothing

Solution: Nothing associates a password with a username, so the attacker could swap records in the database and change a user's password to another user's password.

However, the attacker could not change a user's password to any value, because without SK, the attacker could not generate a valid signature on the attacker-chosen password.

Scheme 2: For each user, the server stores the username and

$$\text{HMAC}(K, \texttt{username} \parallel \texttt{"161"} \parallel \texttt{password})$$

K is a key known only by the server.

Q5.4 (3 points) In Scheme 2, is the attacker able to determine all pairs of users who share the same password?

○ (A) Yes, without computing any hashes

○ (B) Yes, but only by computing $O(N)$ hashes

[✓ (C) No]

Solution: No. The HMAC values will be unique per user, because the username is part of the input, so the attacker cannot look for users with the same HMAC output value, as in the previous scheme.

Also, the attacker cannot compute HMAC outputs without knowing K, so the attacker cannot run any sort of brute-force attack to guess-and-check possible passwords.

Q5.5 (3 points) In Scheme 2, how many users' passwords is the attacker able to learn?

[✓ (A) None] ○ (B) Only one ○ (C) All of them

Solution: None. Without K, the attacker can't compute HMACs of their own. Also, because HMACs are based on hashes, the attacker can't try to reverse the HMAC output to figure out what the input to the HMAC was.

For the rest of the question, consider this scheme:

Scheme 3: For each user, the server stores the username and these two values:

$$C = \mathsf{Enc}(K_1, \texttt{username} \parallel \texttt{"161"} \parallel \texttt{password})$$

$$\mathsf{HMAC}(K_2, C)$$

Enc is an IND-CPA secure encryption scheme. K_1 and K_2 are secret keys known only by the server.

There are two ways a user can interact with the password server:

Create user: The user provides a new `username` and a new `password`. The server computes and stores the two values above for the new user. The attacker is able to see these new values in the database.

Log in:

- The user provides their existing `username` and their `password`.
- The server first verifies the ciphertext C with the stored HMAC value.
- Then, the server decrypts C, and compares the decrypted plaintext with the string `username161password` (replacing `username` and `password` with user-provided values).
- If the decrypted plaintext matches the user-provided string, the user is logged in.

The attacker is able to perform both of these operations, and read/modify the database at all times.

Clarification during exam: The attacker cannot create a user using an existing username.

Clarification during exam: The attacker is not able to modify the username of existing rows, nor can they delete an entire row from the database.

Q5.6 (4 points) The database contains one user, with username `alice161`. The attacker does not know their password.

Explain how the attacker could log in as `alice161`. You may not try to log in with all possible passwords.

Hint: Try writing out what string will be encrypted for user `alice161`.

Solution: Short solution:

Create a new user with username `alice` and password `161known`. This causes the server to encrypt-and-MAC the string `alice161161known`. The attacker can see the resulting ciphertext/MAC.

The attacker copies the encryption/MAC of `alice161161known` into `alice161`'s record. Now the attacker can sign in with username `alice161` and password `known`.

The key realization in this problem is that `alice161`'s record in the database will always be the encryption/MAC of a string like `alice161161...` where the first `161` is from Alice's username and the second `161` is the delimiter added by the scheme. However, the attacker could create a new user called `alice` and give this new user a password starting with `161...` which will also create the encryption/MAC of a string like `alice161161..` (where now the first `161` is the delimiter, and the second `161` is the start of the password).

Q5.7 (5 points) The database contains one user, with username `evanbot`. The attacker does not know their password.

Explain how the attacker could log in as `evanbot`. You may not try to log in with all possible passwords.

Solution: Create a new user with username `evanbot161` and password `known`. This causes the server to encrypt-and-MAC the string `evanbot161161known`.

The attacker copies the ciphertext/MAC of `evanbot161161known` into `evanbot`'s record. Now the attacker can sign in with username `evanbot` and password `161known`.

The attack here is quite similar to the attack from the previous subpart, where we exploit the fact that `161` could be either the delimiter, or part of the username/password. The string `evanbot161161known` could be generated in two different ways:

Username `evanbot161`, password `known`. This is generated by the attacker creating a new user.

Username `evanbot`, password `161known`. The attacker tricks the server into using this interpretation when signing in as `evanbot`.

Solution: Longer solution to Q5.6:

Long solution:

Following the hint: Suppose `alice161`'s password is `unknown` (and the attacker doesn't know this password). Then the string encrypted is `alice161161unknown`.

The attacker cannot decrypt the ciphertext to learn `unknown`. The attacker also can't generate valid HMACs on their own without knowing K_2.

However, we notice that the attacker has the power to create new users, and see the resulting values in the database. This means that if the attacker creates a new user, the server will encrypt-and-MAC this new user's username and password, and the attacker gets to retrieve the resulting ciphertext and (valid) MAC for this new username/password pair!

Now, we have to use this create user power to generate a ciphertext/MAC that could be valid for the user `alice161`. In other words, the string being encrypted must start with `alice161161...` to match the username `alice161` and the delimiter `161`.

We can't create a new user with the username `alice161`, because there can't be duplicate users. However, we could create a new user with the username `alice`. This would create a string starting with `alice161...` where the `161` comes from the delimiter being added. Then, we could start out our new user's password with `161`, so that the string starts with `alice161161...` as desired.

Q6 *Format Strings: Cake Without Pan* **(28 points)**

For this entire question, each subpart is independent of the others.

Consider the following stack diagram after `printf(buf)` is called:

Stack

Address	Value
`0xffff1248:`	`0x12345678` $\longrightarrow$ "evanbot"
`0xffff1244:`	`0xffff1234`
`0xffff1240:`	`0xdabbad04` $\longrightarrow$ `0x00000041`
`0xffff123c:`	`0xdeadbeef`
`0xffff1238:`	`0xffff1250`
`0xffff1234:`	`0x1234001d` $\longrightarrow$ "pancaketasty"
`0xffff1230:`	`&buf`
`0xffff122c:`	RIP of `printf`
`0xffff1228:`	SFP of `printf`

How to read this diagram: Each row of the diagram represents a value on the stack, and if relevant, the data located at that address (shown with arrows).

For example: at address `0xffff1234`, you'll find the address `0x1234001d`. At address `0x1234001d`, you'll find the null-terminated string `pancaketasty`.

Directions: For each subpart, provide an input to `buf` that performs the desired task.

- Your input can only contain percent formatters.
- Each blank can only contain exactly one of these formatters: `c`, `hn`, `n`, `s`, or `x`.
- You **may not** pad the output of a formatter (e.g. `%5c` is not permitted).
- You can assume that writing to any address will not crash the program.
- Not all blanks need to be used, but you may not use more blanks than provided.

Note: `printf("%x", 0x1234001D)` will output `1234001d`. Notice how there is no prefixing `0x` and all letters are in lowercase.

Q6.1 (6 points) Print a string containing `evanbot`. (The output can have other characters too, but `evanbot` needs to be in the output.)

%c%c%c%c%c%s

Solution: For the entirety of these solutions, assume that arguments to `printf` are zero-indexed. That is, &buf is its zero-th argument, `0x1234001d` is its first argument, `0xffff1236` is its second argument, etc...

The `%c` formatters could be replaced with any other format string.

The only place in memory where the string "evanbot" appears is at address `0x12345678`. On the stack, we have a pointer to the string, located where `printf` would find its 6th argument. Therefore, we need 5 percent formatters to skip past the first 5 arguments. Then, with the 6th formatter, we use `%s` to dereference the pointer to the string and print out the string.

We do allow %s, %hn, and %n for each of the first five formatters and allow for the assumption that all memory addresses on the stack dereference to valid memory locations.

Q6.2 (6 points) Write the integer 13 to address `0xdeadbeef`.

%s%c%n

Solution: No alternate solutions that we know of.

The reasoning behind `%n` being the third percent formatter is the same as the previous subpart (we need to skip past the first two arguments so that `%n` matches with `0xdeadbeef` on the stack).

This time, we need 13 characters to be outputted in total. Using two `%x`s or `%c`s will not be enough to get us to 13 characters printed. However, note that the first argument on the stack, `0x1234001d`, is a pointer to the string "pancaketasty" which is of length 12. Therefore, we can match `%s` with the first argument to dereference the pointer and cause 12 characters to be printed. Then, we can match `%c` with the second argument to cause 1 more character to be printed (for a total of 13).

Any solutions that attempts to use `%s` on `printf`'s second argument will not work, because it will attempt to dereference a pointer to the middle of the stack and print out values on the stack starting with `0x1234`, stopping when it encounters a null byte. However, this prints an unknown number of bytes, as it is not known when the next null byte on the stack occurs.

Q6.3 (6 points) Print a string containing exactly the capital letter `A`, and nothing else.

Remember that the hexadecimal ASCII value for the character `A` is `0x41`.

%n%n%n%s

Solution: Only alternate solutions (that we know of): `%hn` and `%n` are interchangeable in the given solution.

The only place where "A" appears in memory is at address `0xdabbad04`. This is one of the addresses on our stack. We want to use `%s` to dereference this value on the stack and print out `0x41` interpreted as a character, which is "A".

`0xdabbad04` is the fourth argument on the stack, so we need three percent formatters before the `%s`. Since we need exactly "A" to be printed, the first three percent formatters cannot produce any output; therefore, `%n` is the only valid percent formatter to use here.

Note that this solution will indeed print exactly the character `A`, no more and no less. The *%s* will not print out any extra bytes after `A` because the next byte in the integer `0x41` is a null byte.

Also, note that the `%n` (or `%hn`) will no cause a segfault because each address that will be dereferenced is a valid address in terms of this question.

Q6.4 (10 points) For this subpart, suppose that after `printf(buf)` returns, you overwrite `buf` with a second input, and then you immediately call `printf(buf)` again. The stack looks exactly the same on the second `printf` call, except any changes to memory from the first call will carry over to the second call.

You want the second `printf` call to output exactly `caketasty` and nothing else. It doesn't matter what the first `printf` call outputs.

Hint: What is the address of the string `caketasty` in memory?

Hint: Note that `0x20` is 32 in decimal.

First input to `buf`:

%x%x%x%x%hn%__________

Second input to `buf`:

%s%__________%__________%__________%__________%__________

Solution: No alternate solutions (that we know of).

To print exactly "caketasty", we need a pointer to the string "caketasty" somewhere on the stack. However, we only have a pointer to the string "pancaketasty" on the stack.

Note that if `0x1234001d` is the address of string "pancaketasty", then this is the address of the letter "p" in memory. Therefore, `0x1234001d + 3 = 0x12340020` is the address of the letter "c" in memory. We can use the first `printf` to change `0x1234001d` in memory to `0x12340020`, creating a pointer to the string "caketasty" instead.

To change `0x1234001d` in memory, we need a `%n` to match up with the address of `0x1234001d` on the stack. According to the GDB line, `0xffff1234` (the 5th argument on the stack) corresponds to the address that we want `%n` to match up with. (Note that `0xffff1234` isi the address of the lower bytes `0x001d`, and `0xffff1236` is the address of the upper bytes `0x1234`.)

We want to change `0x001d` to `0x0020`, which means we need `0x20` = 32 bytes to be printed in total. Also, we want four percent formatters to consume arguments so that the `%n` matches with the 5th argument on the stack. The simplest (and probably only) way to print out 32 characters with four percent formatters is `%x` four times (each `%x` prints out 8 characters).

In summary, the first input needs to create a pointer to "cake" by incrementing the "pancake" pointer. This is achieved by using four `%x` formatters to skip past four arguments and print 32 characters, and then a `%hn` to dereference the 5th argument and write `0x0020` into memory.

Now that the address `0x12340020` appears in memory, our second input just needs to dereference that address (the 1st argument on the stack) to print out "cake". This is achieved with a single `%s`. Since we just want "cake" printed, no further format strings should be used.

Q7 *Memory Safety Exploit: Valentine's Day* (22 points)

EvanBot wrote a program to exchange valentines with CodaBot. Mallory got her hands on EvanBot's code, and wants to send a valentine that executes her shellcode and ruins Valentine's day!

```
 1 typedef struct message {
 2     char *ptr;
 3     char text[64];
 4 } message_t;
 5
 6 typedef struct reply {
 7     char *ptr;
 8     char text[8];
 9 } reply_t;
10
11 void valentine() {
12     reply_t coda;
13     coda.ptr = &(coda.text[____]);
14     fgets(coda.ptr, 5, stdin);
15 }
16
17 void main(){
18     message_t evan;
19     reply_t bot;
20     fgets(evan.text, 64, stdin);
21     evan.ptr = &evan.text[0];
22     valentine();
23 }
```

Stack at Line 13:

RIP of `main`
SFP of `main`
canary
(1)
(2)
(3)
`bot.ptr`
RIP of `valentine`
SFP of `valentine`
(4)
(5)
(6)

ASLR is enabled on the stack, but not in the code section. (In other words, the stack section is moved on each run, but the code section stays in the same place.) **Stack canaries** are also enabled. No other memory safety defenses are enabled.

Q7.1 (2 points) What values go in Blanks (1)-(3) in the stack diagram above?

- ○ (A) (1) - `bot.text[8]` (2) - `evan.text[64]` (3) - `evan.ptr`
- ○ (B) (1) - `bot.text[8]` (2) - `evan.ptr` (3) - `evan.text[64]`
- [✓ (C) (1) - `evan.text[64]` (2) - `evan.ptr` (3) - `bot.text[8]`]
- ○ (D) (1) - `evan.ptr` (2) - `evan.text[64]` (3) - `bot.text[8]`

Q7.2 (2 points) What values go in Blanks (4)-(6) in the stack diagram above?

- ○ (A) (4) - `coda.text[8]` (5) - `coda.ptr` (6) - canary
- ○ (B) (4) - `coda.text[8]` (5) - canary (6) - `coda.ptr`
- [✓ (C) (4) - canary (5) - `coda.text[8]` (6) - `coda.ptr`]
- ○ (D) (4) - canary (5) - `coda.ptr` (6) - `coda.text[8]`

Solution:

	RIP of `main`
	SFP of `main`
	canary
(1)	`evan.text[64]`
(2)	`evan.ptr`
(3)	`bot.text[8]`
	`bot.ptr`
	RIP of `valentine`
	SFP of `valentine`
(4)	canary
(5)	`coda.text[8]`
(6)	`coda.ptr`

Mallory runs GDB and finds some useful assembly in the executable. (Assume that all x86 instructions are 4 bytes long.)

Clarification during exam: Some of the addresses had typos; they're fixed in this version.

```
0x08988158: xor %ebx, %ebx
0x0898815c: mov %ebp, %esp
0x08988160: pop %edx
0x08988164: pop %ecx
0x08988168: pop %ebx
0x0898816c: pop %eax
0x08988170: pop %ebp
0x08988174: ret
```

Create an exploit that would cause this program to execute shellcode.

Hints (there's no new information here, but it might give you ideas if you're stuck):

- The `pop` instruction takes the lowest value (where the stack pointer register `esp` is pointing) off the stack, and puts that value in the specified register. For example, `pop %eax` puts the lowest value on the stack into register `%eax`. When a value is popped off the stack, the `esp` moves up by 4.
- The `ret` instruction behaves like `pop %eip`.

Q7.3 (5 points) What integer should go in the blank on Line 13?

Solution: 16

Note that whatever goes in the blank on Line 13 affects the value of `coda.ptr`, which in turn affects where the `fgets(coda.ptr, 5, stdin)` call on Line 14 starts writing in memory. We could put an unusually large value (i.e. greater than 8, the real size of `text`) in this blank, which would cause `fgets` to write out-of-bounds!

In particular, we need the `fgets` call on Line 14 to start writing after the stack canary. If `fgets` started writing below the stack canary, it might clobber out the value of the canary and crash the program. To write above the canary, we'd like `coda.ptr` to point somewhere after the stack canary.

The `fgets` call can only write 4 bytes to memory. The most useful 4 bytes to overwrite here are the RIP of `valentine`, so we'd like `coda.ptr` to point to the RIP of `valentine`.

`&coda.text[16]` is the address of the RIP of `valentine`, so the index we're looking for is 16.

We tried to design this question so that even if you didn't get the bulk of the exploit in the next few subparts, you could use your intuition about defeating the stack canary to arrive at 16 as a useful number in the blank, because this numbers let us write to an RIP on the stack.

Q7.4 (9 points) What should be the inputs to the two `fgets` calls?

You can use `SHELLCODE` to represent the 60-byte shellcode you want to execute.

Input to `fgets` in `main`, at **Line 20**:

Solution: SHELLCODE

By process of elimination, you have to put `SHELLCODE` here. It doesn't fit in the other `fgets` call, and it's not in memory yet (so you have to put it in memory yourself).

`fgets` will read a total of 63 bytes of input from the user and write it into `evan.text`, so technically we do have 3 more bytes here that we could write after shellcode. It turns out that we won't need them, though - keep reading to see that the rest of the solution doesn't use this space.

Input to `fgets` in `valentine`, at **Line 14**:

Solution: `'\x68\x81\x98\x08'`

Short solution: You need to pop 3 words off the stack (`bot.ptr` and the two words in `bot.text[8]`, and then execute a `ret` instruction that dereferences `evan.ptr` to execute the shellcode.

By overwriting the RIP to point to `0x08988168`, we cause 3 pops to execute, taking `bot.ptr` and `bot.text[8]` off the stack, then a `ret` to execute, which dereferences `evan.ptr`.

Detailed solution below.

Q7.5 (4 points) Mallory runs GDB and discovers that the address of `bot.text` is `0xffff1234`.

Can Mallory use this address to exploit this program?

(A) Yes [✓ (B) No]

Briefly justify your answer. You can answer in 15 words or fewer.

Solution: ASLR changes addresses on the stack every time, so this address won't be the same on future runs of the program.

Solution: Detailed exploit solution:

Because we put 16 in the blank earlier, this `fgets` is going to start writing at the RIP of `valentine`.

We want to execute shellcode that lives at `evan.text` (since we put shellcode there with our last `fgets`).

At this point, a good first step is to think about the steps of a buffer overflow exploit, and which steps are being made difficult because of the code or the defenses. In this case, we know that we're eventually going to need the address of shellcode (aka address of `evan.text`). However, ASLR on the stack is enabled, so this address changes every time! Luckily, we look at our stack diagram or the code and notice that thanks to Line 21, the address of shellcode actually does appear on the stack; it's the value of `evan.ptr`.

In normal function execution, the program would never dereference `evan.ptr` and start executing instructions there. However, we read through the reminders for inspiration and notice that we could get this program to potentially execute a `ret` instruction. Remember: the `ret` instruction takes a value on the stack, reads it as an address, and starts executing instructions at that address. If we could force the program to execute `ret`, it might look on the stack, read `evan.ptr` as an address, and start executing instructions at that address (aka our shellcode).

Our first attempt, recalling Project 1 Q7, might be to overwrite the RIP of `valentine` with the address of `ret`. When the program returns from `valentine`, it will jump to this address and execute the `ret` instruction. What will this do? It will take the next value on the stack, `bot.ptr`, jump to that address, and start executing instructions there.

Unfortunately, the value `bot.ptr` isn't the address of shellcode. In order to reach the address of shellcode, we need to delete a few more values off the stack so that when the `ret` is executed, it dereferences `evan.ptr`.

Reading through the reminders again, we see that the `pop` instruction is also available to us in memory, and it seems useful to us because it will move ESP up by 4 and take a value off the stack.

After executing one `ret`, the program is going to jump to `bot.ptr` and move ESP up by 4. This will make `bot.text[0:4]` the next lowest value on the stack. We can see that 8 bytes, or 2 words/pointers, on the stack separate the new lowest value and the desired lowest value of `evan.ptr`. These 2 values on the stack are `bot.text[0:4]` and `bot.text[4:8]`. We can't repeatedly execute `ret` instructions to remove these extra values off the stack to reach `evan.ptr`, but we can execute the sequence of `pop` instructions to remove these values off the stack!

In particular, if the RIP of `valentine` points to the first `pop` instruction, then the program will execute:

`pop %ebx`: Take `bot.ptr` off the stack, put its value in `%ebx`. We don't care about the value in the `%ebx` register, so this operation is harmless and useful for removing a value off the stack.

`pop %eax`: Take `bot.text[0:4]` off the stack, put its value in `%eax`. Again, we don't care about the value in `%eax`, so this operation is harmless and useful.

`pop %ebp`: Take `bot.text[4:8]` off the stack, put its value in `%ebp`. Again, we don't care about the value in the `%ebp` register, so this operation is harmless and useful.

`ret`: Take the next value on the stack, `evan.ptr`, off the stack, and jump to this address. Remember that `evan.ptr` held the address of shellcode, so we've jumped to shellcode!

In summary:

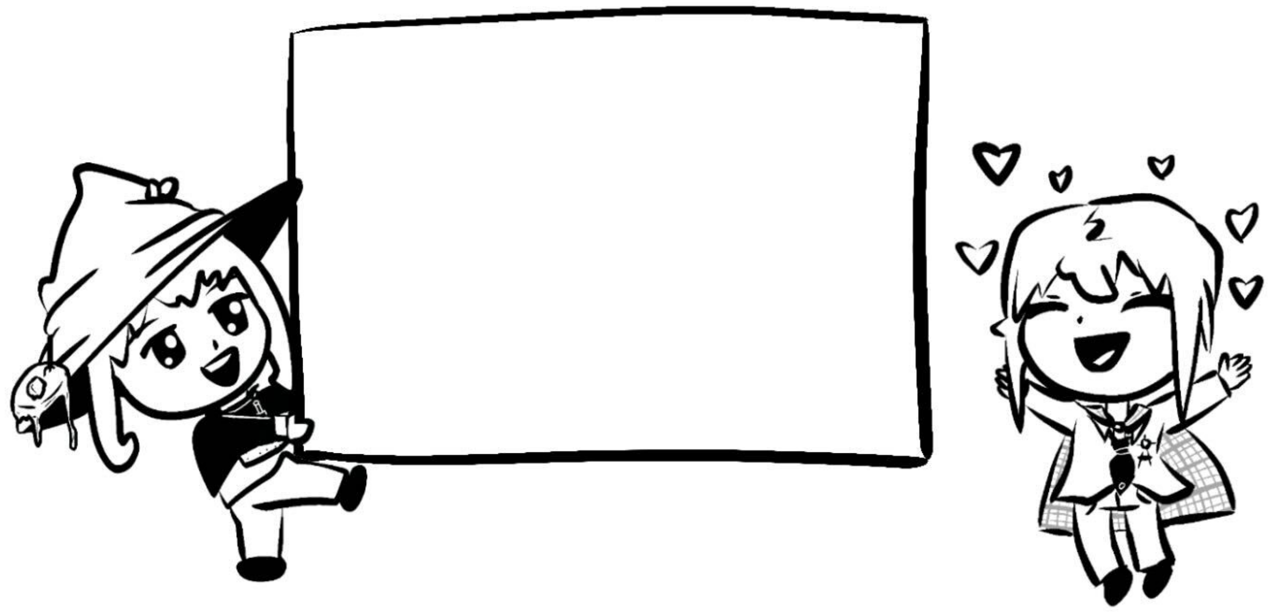
Overwrite RIP of `valentine` with `0x08988168`, the address of a pop-pop-pop-ret sequence of instructions. When `valentine` returns, the program will jump to this address and execute three pop instructions and a ret instruction.

The first pop instruction takes `bot.ptr` off the stack. The second pop instruction takes `bot.text[0:4]` off the stack. The second instruction takes `bot.text[4:8]` off the stack. The ret instruction dereferences `evan.ptr`, which is the address of shellcode, and starts executing instructions there.

Nothing on this page will affect your grade in any way.

Post-Exam Activity: Valentine

What was in EvanBot's valentine to CodaBot? Draw it here!



Comment Box

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