Popa and Weaver
Fall 2021

CS 161
Computer Security

Midterm

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

PRINT your student ID: _____

You have 110 minutes. There are 8 questions of varying credit (150 points total).

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): Draw lines to match each character to their name.

Answer: Bob

Answer: Alice

Answer: Mallory

Answer: EvanBot

Answer: Eve

Q1 *Honor Code* **(4 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 and a corresponding notch on Nick's Stanley Fubar demolition tool.

SIGN your name: _____

Q2 *True/false* **(30 points)**

Each true/false is worth 2 points.

Q2.1 TRUE or FALSE: Assume stack canaries, non-executable pages, and pointer authentication codes are enabled, but ASLR is disabled. An attacker can execute the RET2ESP attack from Project 1, Question 6.

○ TRUE ● FALSE

Solution: False. A RET2ESP attack involves the CPU jumping to the ESP, which lives on the stack - but because we have non-executable pages enabled, the stack is non-executable (and thus the program will crash when the EIP moves to the ESP).

Q2.2 TRUE or FALSE: Assume that ASLR is enabled, but all other memory safety defenses are disabled. It's possible to find an exploit that will allow an attacker to learn the address of the RIP during program execution.

● TRUE ○ FALSE

Solution: True. If a format string vulnerability exists, an adversary could print a stack address (e.g. the value of the SFP), which would then allow the attacker to identify where the RIP was on the stack.

Q2.3 TRUE or FALSE: When designing a secure system, we often assume that an adversary has access to our source code.

● TRUE ○ FALSE

Solution: True. Remember Shannon's Maxim: Do not rely on security through obscurity!

Q2.4 TRUE or FALSE: If pointer authentication codes are enabled, the program will always crash if the user writes past the end of a buffer.

○ TRUE ● FALSE

Solution: False. PAC's only protect **pointers** on the stack, so it still could be possible to write past the end of a buffer and overwrite a boolean flag, for example.

Q2.5 TRUE or FALSE: Let $\mathsf{Enc}(K, M)$ be an IND-CPA secure encryption function. If Alice computes Enc("Hello", "World") and Bob computes Enc("Hello", "World"), they will always evaluate to the same ciphertext.

○ TRUE ● FALSE

Solution: False. Because this encryption function is IND-CPA secure, the scheme cannot be deterministic; consequently, encrypting the same thing twice (even with the same key and value) must yield two unique ciphertexts.

Q2.6 A PRNG has been securely initialized with truly random bits. Assume the attacker has not learned the PRNG's internal state.

TRUE or FALSE: It is infeasible for the attacker to distinguish n bits of PRNG output from n truly random bits.

● TRUE ○ FALSE

Solution: True. This is the security definition of a PRNG.

Q2.7 Alice and Bob are building a password management scheme for CalCentral. Carol tells them that they should store a record of $(\mathsf{username}, \mathsf{MD5}(\mathsf{password}))$ for each user in the database.

Clarification during exam: MD5 is a cryptographic hash function.

TRUE or FALSE: This design protects against offline password hashing attacks.

○ TRUE ● FALSE

Solution: False, for two reasons: MD5 is a weak hash, and the password isn't salted.

Q2.8 TRUE or FALSE: When designing a password management scheme, all security guarantees are lost if the attacker gains access to a table containing salt values.

○ TRUE ● FALSE

Solution: False. Salts can be public information - their purpose is to prevent dictionary attacks.

Q2.9 TRUE or FALSE: Assume that stack canaries are enabled, but all other memory safety defenses are disabled. A format string vulnerability in a vulnerable C program may allow an attacker to write arbitrary bytes anywhere on the stack.

● TRUE ○ FALSE

Solution: True. The `%n` specifier allows the attacker to write to an arbitrary address.

Q2.10 TRUE or FALSE: If a website requires all passwords to be at least 20 characters long, then it's more difficult for an adversary to perform rainbow table attacks on unsalted password hashes.

● TRUE ● FALSE

Solution: We dropped this subpart and gave everyone full points during the exam, because the phrasing "more difficult" was ambiguous. It was not clear what we are comparing this scheme to.

The intended answer was true. Brute-force attacks, online or offline, are harder when passwords are longer.

Q2.11 TRUE or FALSE: Diffie-Hellman is susceptible to man-in-the-middle attacks.

● TRUE ○ FALSE

Solution: True. Mallory can do a DH with both sides!

Q2.12 Bob has public key PK_B and private key SK_B. Assume that Sign is a secure digital signature algorithm.

TRUE or FALSE: $\{\text{“Bob's public key is } PK_B\text{”}\}_{SK_B^{-1}}$ is a valid certificate on Bob's public key.

○ TRUE ● FALSE

Solution: False. Certificates are signed by someone else's secret key. Bob cannot endorse his own public key.

Q2.13 Recall that in ElGamal, the ciphertext is $C_1 = g^r \bmod p$ and $C_2 = M \times (g^b)^r \bmod p$.

TRUE or FALSE: Alice encrypts M and sends the corresponding ciphertext C_1 and C_2 to Bob. If Mallory replaces the ciphertext with $2 \times C_1$ and $2 \times C_2$, Bob will decrypt the ciphertext as $2 \times M$.

○ TRUE ● FALSE

Solution: False. Mallory should replace the ciphertext with C_1 and $2 \times C_2$.

Q2.14 Alice and Bob share a symmetric key K not known to anyone else, and MAC is a secure (unforgeable) MAC scheme. Alice presents the statement $M =$ “Bob owes Alice 100 dollars” and the MAC $\mathsf{MAC}(K, M)$ to a judge.

TRUE or FALSE: The judge can be sure that the message was sent by Bob to Alice.

○ TRUE ● FALSE

Solution: False. Alice could have constructed the MAC of the message herself.

Q2.15 TRUE or FALSE: In the Bitcoin ledger, everyone can add new entries and modify/delete existing entries.

○ TRUE ● FALSE

Solution: False. Bitcoin uses a public ledger that is append-only and immutable meaning that everyone can add entries to the ledger, but nobody can modify or delete existing entries.

Q3 *EvanBot Alpha* **(15 points)**

Earlier releases of EvanBot sometimes came with security vulnerabilities. For each subpart, select the most relevant security principle. Each option is used exactly once.

Q3.1 (3 points) When debugging students' code, EvanBot Alpha ran all code with administrator privileges. The latest version of EvanBot runs untrusted code in an isolated environment (sandbox) instead.

- ◯ (A) Consider human factors
- ● (B) Least privilege
- ◯ (C) Separation of responsibility
- ◯ (D) Don't rely on security through obscurity
- ◯ (E) Design in security from the start

Solution: Least privilege. Administrator privileges are not needed to run all student code.

Q3.2 (3 points) Any TA by themselves could instruct EvanBot Alpha to release exam solutions. The latest version of EvanBot requires two TAs to approve the instruction before releasing exam solutions.

- ◯ (G) Consider human factors
- ◯ (H) Least privilege
- ● (I) Separation of responsibility
- ◯ (J) Don't rely on security through obscurity
- ◯ (K) Design in security from the start

Solution: Separation of responsibility. Releasing exam solutions is a dangerous privilege, so we require multiple TAs to approve before the privilege can be exercised.

Q3.3 (3 points) The default setting for EvanBot Alpha was to store exam solutions with no encryption. To encrypt exam solutions with AES, TAs had to manually change a setting in EvanBot Alpha's source code. The latest version of EvanBot encrypts exam solutions by default.

- ● (A) Consider human factors
- ◯ (B) Least privilege
- ◯ (C) Separation of responsibility
- ◯ (D) Don't rely on security through obscurity
- ◯ (E) Design in security from the start

Solution: Consider human factors. If it is difficult to find the secure option (encrypting exam solutions), then it will not be used.

Q3.4 (3 points) To protect exam solutions, EvanBot Alpha used a secret encryption algorithm. The latest version of EvanBot uses a well-known encryption algorithm with a secret key.

◯ (G) Consider human factors
◯ (H) Least privilege
◯ (I) Separation of responsibility
● (J) Don't rely on security through obscurity
◯ (K) Design in security from the start

Solution: Don't rely on security through obscurity. If the secret encryption algorithm leaked, we would need to change the algorithm entirely. If the secret key leaks, we only need to change the key.

Q3.5 (3 points) EvanBot Alpha required frequent patches to fix vulnerabilities. The latest version of EvanBot's source code was rewritten from scratch and requires far fewer security patches.

◯ (A) Consider human factors
◯ (B) Least privilege
◯ (C) Separation of responsibility
◯ (D) Don't rely on security through obscurity
● (E) Design in security from the start

Solution: Design in security from the start. When code is not written with security in mind, it will need to be patched later to fix vulnerabilities.

Q4 *CIA* **(15 points)**

Alice and Bob want to communicate securely over an insecure channel. Mallory can read and modify messages sent over the channel. Determine whether each scheme provides confidentiality, integrity, both, or neither. Each subpart is independent.

Assumptions:

- ECB and CBC denote AES-ECB and AES-CBC encryption, respectively.
- MAC is a secure (existentially unforgeable) MAC scheme.
- Sign denotes RSA signatures.
- Alice and Bob share two symmetric keys, K_1 and K_2, not known to anyone else.
- Alice has an RSA key pair (PK_a, SK_a). Bob knows PK_a, and SK_a is not known to anyone other than Alice.

Clarification during exam: Assume that Eve knows the padding scheme.

Q4.1 (3 points) Alice sends over $\mathsf{CBC}(K_1, M)$ and $\mathsf{MAC}(K_2, M)$.

◯ (A) Confidentiality ◯ (C) Both

● (B) Integrity ◯ (D) Neither

Solution: The MAC provides integrity, since the MAC can only be computed when K is known. However, the MAC provides no guarantee of confidentiality, as the MAC can leak bits of the plaintext.

Q4.2 (3 points) Alice computes $C = \mathsf{CBC}(K_1, M)$. Then, Alice sends $\mathsf{ECB}(K_2, C)$ and $\mathsf{Sign}(SK_a, C)$.

◯ (G) Confidentiality ● (I) Both

◯ (H) Integrity ◯ (J) Neither

Solution: The use of an IND-CPA insecure encryption scheme (ECB) in series with an IND-CPA secure encryption scheme (CBC) will be IND-CPA secure and thus provide confidentiality, since the security of the modified scheme reduces to the security of the IND-CPA secure scheme (CBC). In addition, Alice's valid signature on the message provides integrity.

Q4.3 (3 points) Alice computes $C_0 = \mathsf{CBC}(K_1, M)$ and $C_1 = \mathsf{ECB}(K_1, M)$. Then, Alice sends C_0, C_1, and $\mathsf{Sign}(SK_a, C_0)$.

◯ (A) Confidentiality

◯ (C) Both

● (B) Integrity

◯ (D) Neither

Solution: The use of an IND-CPA insecure encryption scheme (ECB) in parallel with an IND-CPA secure encryption scheme (CBC) will be IND-CPA insecure and thus not provide confidentiality, since the security of the modified scheme reduces to the security of the IND-CPA insecure scheme (ECB). Alice's valid signature on the message provides integrity.

Q4.4 (3 points) Alice computes $C = \mathsf{CBC}(K_1, M)$. Alice sends C and $\mathsf{MAC}(K_2, C)$.

◯ (G) Confidentiality

● (I) Both

◯ (H) Integrity

◯ (J) Neither

Solution: This scheme is an authenticated encryption scheme denoted as encrypt-then-MAC. Unlike the previous part, the MAC is computed over the encryption of the plaintext, so plaintext bits are not leaked.

Q4.5 (3 points) Alice sends $\mathsf{CBC}(K_1, M)$ and $\mathsf{Sign}(SK_a, \text{“This message was sent by Alice”})$.

● (A) Confidentiality

◯ (C) Both

◯ (B) Integrity

◯ (D) Neither

Solution: This scheme provides confidentiality since CBC is IND-CPA secure. This scheme does not provide integrity because a man-in-the-middle (MITM) attacker can perform a replay attack on the signature, attaching the signature to an arbitrary message.

Q5 *Ivy Why* **(21 points)**

Alice wants to send a 400-bit message M to Bob. Assume that Alice is using a block cipher with 128-bit blocks and that Alice is using PKCS#7 for padding (this is the padding scheme from Homework 2).

Eve observes the encryption of M. Assume that Eve knows which encryption scheme is being used. For each encryption scheme, what is the most specific information Eve can learn about $\mathsf{len}(M)$, the length of M in bits?

Clarification during exam: PKCS #7 is a correct padding scheme that pads the number of padding bytes added.

Q5.1 (3 points) Alice pads the plaintext to the nearest multiple of the block size and then encrypts the padded plaintext with CBC mode.

◯ (A) $\mathsf{len}(M) = 400$ ◯ (C) $256 \leq \mathsf{len}(M) < 384$ ◯ (E) $512 \leq \mathsf{len}(M)$

◯ (B) $\mathsf{len}(M) < 256$ ● (D) $384 \leq \mathsf{len}(M) < 512$ ◯ (F) None of the above

Solution: Eve has no way of knowing the exact length of the message since we are using padding. However, CBC mode does leak the length to the nearest block size, so we leak the fact that we use four blocks.

Q5.2 (3 points) Alice pads the plaintext to the nearest multiple of the block size and then encrypts the padded plaintext with CTR mode.

◯ (G) $\mathsf{len}(M) = 400$ ◯ (I) $256 \leq \mathsf{len}(M) < 384$ ◯ (K) $512 \leq \mathsf{len}(M)$

◯ (H) $\mathsf{len}(M) < 256$ ● (J) $384 \leq \mathsf{len}(M) < 512$ ◯ (L) None of the above

Solution: Padded plaintext looks indistinguishable from random, so again we only leak length to the nearest block size, so we leak the fact that we use four blocks.

Q5.3 (3 points) Alice encrypts the message with CTR mode and then pads the ciphertext to the nearest multiple of the block size.

● (A) $\mathsf{len}(M) = 400$ ◯ (C) $256 \leq \mathsf{len}(M) < 384$ ◯ (E) $512 \leq \mathsf{len}(M)$

◯ (B) $\mathsf{len}(M) < 256$ ◯ (D) $384 \leq \mathsf{len}(M) < 512$ ◯ (F) None of the above

Solution: Since the padding happens on the ciphertext, Eve can de-pad to retrieve the ciphertext without padding. Since we're using CTR mode, Eve learns the exact length of M.

For the rest of this question, Alice is sending two 400-bit messages M_1 and M_2 to Bob. M_1 and M_2 are identical except in the 200th bit.

Eve observes the encryption of M_1 and the encryption of M_2. Assume that Eve knows which encryption

scheme is being used. For each encryption scheme, what is the most specific information that Eve can learn about how the messages differ? Assume that inputs for block cipher modes that require padding are correctly padded.

Q5.4 (3 points) Alice encrypts the messages with ECB mode.

◯ (G) The messages differ in only the 200th bit

● (H) The messages differ in only the second block (Eve doesn't know which bit)

◯ (I) The messages differ in the second block and possibly subsequent blocks

◯ (J) The messages differ somewhere (Eve doesn't know which blocks)

◯ (K) None of the above

Solution: ECB shows which blocks differ from one another (since it is a deterministic block cipher). As such, it shows that all the blocks are the same except the second block, but because block ciphers are indistinguishable from random, Eve can't tell that only one bit is different.

Q5.5 (3 points) Alice encrypts the messages with CBC mode. Alice uses the same IV for both encryptions.

◯ (A) The messages differ in only the 200th bit

◯ (B) The messages differ in only the second block (Eve doesn't know which bit)

● (C) The messages differ in the second block and possibly subsequent blocks

◯ (D) The messages differ somewhere (Eve doesn't know which blocks)

◯ (E) None of the above

Solution: CBC mode with a reused IV leaks the first block in which messages differ. After this, the inputs to the next block of CBC mode change unpredictably, and security is restored for the remainder of the message.

Q5.6 (3 points) Alice encrypts the messages with CTR mode (with no padding). Alice uses the same nonce for both encryptions.

● (G) The messages differ in only the 200th bit

◯ (H) The messages differ in only the second block (Eve doesn't know which bit)

◯ (I) The messages differ in the second block and possibly subsequent blocks

◯ (J) The messages differ somewhere (Eve doesn't know which blocks)

◯ (K) None of the above

Solution: CTR with the same IV is effectively one-time pad, so Eve sees exactly which bit was different.

Q5.7 (3 points) Alice encrypts the messages with CTR mode (with no padding). Alice uses a different, random nonce for each encryption.

◯ (A) The messages differ in only the 200th bit

◯ (B) The messages differ in only the second block (Eve doesn't know which bit)

◯ (C) The messages differ in the second block and possibly subsequent blocks

◯ (D) The messages differ somewhere (Eve doesn't know which blocks)

● (E) None of the above

Solution: Since nonces are public values, knowing them doesn't give Eve extra information. CTR mode is IND-CPA secure with different nonces so Eve didn't actually learn anything.

Q6 *Bonsai* (22 points)

EvanBot wants to store a file in an *untrusted* database that the adversary can read and modify.

Before storing the file, EvanBot computes a hash over the contents of the file and stores the hash separately. When retrieving the file, EvanBot re-computes a hash over the file contents, and, if the computed hash doesn't match the stored hash, then EvanBot concludes that the file has been tampered with.

Clarification during exam: Assume that EvanBot does not know if hashes or files have been modified in the untrusted datastore.

Clarification during exam: The assumption that only F_2 is modified applies for all parts after subpart 4.

NOTE: Subparts 6.4–6.7 have been dropped from this question due to ambiguities in wording and clarifications.

Q6.1 (4 points) What assumptions are needed for this scheme to guarantee integrity on the file? Select all that apply.

■ (A) An attacker cannot tamper with EvanBot's stored hash

☐ (B) EvanBot has a secret key that nobody else knows

☐ (C) The file is at most 128 bits long

■ (D) EvanBot uses a secure cryptographic hash

☐ (E) None of the above

Solution: In order to guarantee integrity on this file, we need two assumptions to hold.

First, the attacker shouldn't be able to tamper with the stored hash. If they could, then the attacker could simply replace the file with an arbitrary file of the attacker's choice, and replace the original stored hash with a hash over this new file. EvanBot's check on the file would succeed.

If EvanBot had a secret key, then EvanBot could change the scheme to use a MAC using the secret key instead of a hash. However, since this scheme uses a hash, a secret key doesn't help us here.

The file being 128 bits long has no relevance to this question.

Finally, the hash must be a secure cryptographic hash. A quick counterexample: if EvanBot used a hash function that mapped every input to the hash value "1", then the attacker could choose an input of their choice, and the check on the hash would always succeed.

For the rest of this question, we refer to two databases: a *trusted database* that an adversary cannot read or modify, and an *untrusted database* that an adversary can read and modify.

Assume that H is a secure cryptographic hash function and $\|$ denotes concatenation.

EvanBot creates and stores four files, F_1, F_2, F_3, and F_4, in the untrusted database. EvanBot also computes and stores a hash on each file's contents in the untrusted database:

$$h_1 = \mathsf{H}(F_1) \qquad h_2 = \mathsf{H}(F_2) \qquad h_3 = \mathsf{H}(F_3) \qquad h_4 = \mathsf{H}(F_4)$$

Then, EvanBot stores $h_{\mathsf{root}} = \mathsf{H}(h_1 \| h_2 \| h_3 \| h_4)$ in the *trusted* database.

Q6.2 (3 points) If an attacker modifies F_2 stored on the server, will EvanBot be able to detect the tampering?

● (G) Yes, because EvanBot can compute h_{root} and see it doesn't match the stored h_{root}

○ (H) Yes, because EvanBot can compute h_2 and see it doesn't match the stored h_2

○ (I) No, because the hash doesn't use a secret key

○ (J) No, because the attacker can re-compute h_2 to be the hash of the modified file

Solution:

In this scheme, we have a trusted database that an adversary cannot read or modify. Because we have this trusted database, it's possible to ensure integrity through the use of hashes, despite them not being signed (like MAC's).

Let's walk through what happens if an attacker modifies F_2. If the attacker modifies this file and nothing else, then it's easy for Bot to detect tampering: Bot just has to recompute a hash over F_2 and realize that it doesn't match h_2.

However, an attacker can also modify h_2 to be the hash of the malicious file, since it's in the untrusted database. Because of this, in order to detect tampering, Bot has to use the only thing that the attacker doesn't have access to: h_{root}, which is stored in the trusted database.

Based on this information: the simplest way to verify the integrity of F_2 is to:

1. Recompute a hash over F_1, F_2, F_3, and F_4.
2. Recompute h_{root} using these hashes.
3. Compare this h_{root} to the stored version of h_{root}.

If the attacker modifies F_2, then Bot will **always** be able to detect the tampering, since the check on the root hashes will fail.

Q6.3 (3 points) What is the minimum number of hashes EvanBot needs to compute to verify the integrity of all four files?

◯ (A) 1 ◯ (C) 3 ● (E) 5

◯ (B) 2 ◯ (D) 4 ◯ (F) More than 5

Solution:

Because the attacker has the ability to modify all files and hashes in the insecure database, Bot needs to make sure that the attacker hasn't modified any single file/hash pair. To do this, Bot need to follow the procedure discussed in Q6.2's solution - recompute a hash over each file (4 hashes in total), and recompute the root hash (1 hash in total).

For the rest of the question, assume that none of the other files besides F_2 and none of the hashes have been modified.

Q6.4 (3 points) What is the minimum number of hashes EvanBot needs to compute to verify the integrity of only F_2?

● (G) 1 ● (I) 3 ● (K) 5

● (H) 2 ● (J) 4 ● (L) More than 5

Solution:

The wording of this question threw a lot of students off during the exam — we received many, many questions about this. We'll try to explain our original intention here, but due to the confusion — and the fact that a clarification was issued pretty late — and the fact that we, as course staff, decided that this question is fundamentally broken: we decided to drop this subpart, and all remaining subparts of this question.

Let's start by thinking about the work that EvanBot, who knows nothing about what the adversary is going to do with these files, has to do.

As EvanBot, verifying the integrity of **only** F_2 is challenging here, because Bot doesn't know what's going on with the other files and hashes.

If they are modified, then Bot cannot make any determinations about the integrity of only F_2. It's easy for Bot to know if **any** of the files have been tampered with (as an ensemble), but computing the integrity for a **particular** file is provably impossible in this scheme.

Realizing this in exam pre-testing, we added an optimistic assumption above this question, stating that none of the files other than F_2 and none of the hashes in the untrusted database have been modified.

This assumption makes it possible for an individual to use this scheme to verify F_2's integrity. Without it, this scheme just doesn't work.

From the perspective of the reader, once we make this assumption, we can simply recompute F_2's hash and the root hash, and if the root hash doesn't match the stored root hash, we know that F_2 has been modified. Our knowledge (which is something that EvanBot could not possibly have) is what lets us check the integrity of F_2.

All of this said, let's go back to what the question is asking: What is the minimum number of hashes EvanBot needs to compute to verify the integrity of only F_2. **Because the question phrased from the perspective of EvanBot, it is impossible for EvanBot to come to the same conclusions that we can about this system.** Because of this, this question is broken.

During the exam, we issued a clarification that stated the following:

6, all subparts: Assume that EvanBot does not know if hashes or files have been modified in the untrusted datastore.

This clarification, unfortunately, solidified the impossibility of this question: because EvanBot has no knowledge over the assumptions that we're making, it's just not possible for EvanBot to verify the integrity of only F_2.

As such, we threw out subparts 6.4 through 6.7.

Q6.5 (3 points) If EvanBot uses the minimum number of hash computations, how many hashes from the untrusted database are used to verify the integrity of only F_2?

◯ (A) 1 ● (C) 3 ◯ (E) 5

◯ (B) 2 ◯ (D) 4 ◯ (F) More than 5

Solution: This question was dropped. See solution to 6.4.

EvanBot changes the protocol as follows:

Before storing the files, EvanBot first computes the hash of each file individually and stores the following in the *untrusted* database:

$$h_1 = \mathsf{H}(F_1) \qquad h_2 = \mathsf{H}(F_2) \qquad h_3 = \mathsf{H}(F_3) \qquad h_4 = \mathsf{H}(F_4)$$

Then, EvanBot computes and stores the following in the *untrusted* database:

$$h_{\mathsf{left}} = \mathsf{H}(h_1||h_2)$$

$$h_{\mathsf{right}} = \mathsf{H}(h_3||h_4)$$

Finally EvanBot computes and stores the following in the *trusted* database:

$$h_{\mathsf{root}} = \mathsf{H}(h_{\mathsf{left}}||h_{\mathsf{right}})$$

Q6.6 (3 points) What is the minimum number of hashes EvanBot needs to compute to verify the integrity of only F_2?

◯ (G) 1 ● (I) 3 ◯ (K) 5

◯ (H) 2 ◯ (J) 4 ◯ (L) More than 5

Solution: This question was dropped. See solution to 6.4.

Q6.7 (3 points) If EvanBot uses the minimum number of hash computations, how many hashes from the untrusted database are used to verify the integrity of only F_2?

◯ (A) 1 ◯ (C) 3 ◯ (E) 5

● (B) 2 ◯ (D) 4 ◯ (F) More than 5

Solution: This question was dropped. See solution to 6.4.

Q7 *Returnless* **(20 points)**

Consider the following vulnerable C code:

```
1  int main(void) {
2      int i;
3      char input[16];
4      int stored_nums[4];
5
6      for (i = 0; i < 4; i++) {
7          printf("Enter stored number for %d: ", i);
8          gets(input);
9          stored_nums[i] = atoi(input);
10     }
11
12     while (i >= 0) {
13         printf("Which stored number do you want to print? ");
14         gets(input);
15         i = atoi(input);
16         if (i >= 0) {
17             printf("%d\n", stored_nums[i]);
18         }
19     }
20     return 0;
21 }
```

Assume you are on a little-endian 32-bit x86 system. Assume that there is no compiler padding or additional saved registers in all subparts. Assume that **stack canaries are enabled**, and all other memory safety defenses are disabled (unless otherwise specified).

Q7.1 (3 points) Assume the program is paused immediately after executing line 4. Complete the stack diagram below.

(a)
SFP of `main`
(b)
(c)
(d)
(e)

◯ (A) (a) - canary; (b) - RIP of `main`; (c) - `input`; (d) - `stored_nums`; (e) - `i`

◯ (B) (a) - canary; (b) - `i`; (c) - `input`; (d) - `stored_nums`; (e) - RIP of `main`

● (C) (a) - RIP of `main`; (b) - canary; (c) - `i`; (d) - `input`; (e) - `stored_nums`

◯ (D) (a) - RIP of `main`; (b) - canary; (c) - `stored_nums`; (d) - `input`; (e) - `i`

◯ (E) (a) - canary; (b) - `printf`; (c) - `stored_nums`; (d) - `input`; (e) - `i`

◯ (F) (a) - canary; (b) - `i`; (c) - `input`; (d) - `stored_nums`; (e) - `printf`

Solution: Recall the SFP is stored below the RIP and that the local variables are stored below the SFP.

Q7.2 (3 points) Which of the following vulnerabilities is present in this code?

● (G) Buffer overflow

◯ (H) Format string vulnerability

◯ (I) Integer overflow vulnerability

◯ (J) Signed/unsigned vulnerability

◯ (K) None of the above

Solution: The `gets` functions in this question lead to a clear buffer overflow vulnerability, since `gets` doesn't check the length of the buffer.

There are no format string vulnerabilities because all instances of `printf` take a static string as an input.

There are no integer overflow vulnerabilities because we never perform arithmetic on integer values.

There are no signed/unsigned vulnerabilities because we never cast a signed value to an unsigned value or vice-versa, and the for loop iterates from 0 to 7 regardless of whether `i` is signed or unsigned.

Q7.3 (4 points) Which of these inputs to the `gets` call at line 14 will leak the value of the canary? Select all that apply.

☐ (A) `'-1'`
■ (C) `'9'`
☐ (E) None of the above
☐ (B) `'\x00'`
☐ (D) `'12'`

Solution: Notice that this code allows for any non-negative integer value index. According to the stack diagram, the canary must be stored at index 9 (0–3 are `stored_nums`, 4–7 are `input`, and 8 is `i`).

```
print('12\n')
print('34\n')
print('56\n')
print('78\n')
print('9\n')
```

Q7.4 (4 points) Assume that the address of `input` is `0x7ffc9180`.

Fill in the blanks to construct another input to the `gets` call at line 14 that will overwrite the canary with itself and cause the program to execute malicious shellcode.

Write your answer in Python 2 syntax (just like Project 1). You can use the variable `CANARY` as the leaked 4-byte canary value and the variable `SHELLCODE` as the 32-byte shellcode.

Clarification during exam: You do not need to use all blanks.

```
'____________' + '\x_____' + ('A' * _____) + _________________ +

('B' * _____) + --'\x_____\x_____\x_____\x_____' + _________________ + '\n'
```

Solution:

```
'-1' + '\x00' + 'A' * 17 + CANARY + 'B' * 4 +

'\xA0\x91\xfc\x7f' + SHELLCODE + '\n'
```

The buffer must start with `'-1\x00'` so that `atoi` returns a negative number, causing the loop to exit and the function to return. Since `-1` takes up 2 bytes and we add in a null byte, we have to overwrite the remaining 13 bytes of `input` and the four bytes of `i` for a total of 17 garbage btyes. Then we overwrite the SFP with four more dummy bytes followed by the address of the SHELLCODE (which we place at RIP + 4, which is the address of the input + 16 bytes to jump over `input` + 4 bytes to jump over `i` + 4 bytes to jump over the canary + 4 bytes to jump over the SFP + 4 bytes to jump over the RIP) – this is the address of the input plus 0x20. Then we place the SHELLCODE 4 bytes above the RIP.

Q7.5 (3 points) Is it possible to exploit this program without overwriting the stack canary (even with itself)?

◯ (A) Yes, using a format string vulnerability at line 7

● (B) Yes, using a write at line 9

◯ (C) Yes, using a format string vulnerability at line 13

● (D) Yes, using the call to `gets` at line 14

◯ (E) No, because you can't learn the address of the RIP

◯ (F) No, because stack canaries prevent the value of the RIP from changing at all

Solution: The intended solution was (B), but we were not clear about what "exploit" means (whether that's crashing the program, overwriting another variable, or executing shellcode), so we accepted any valid memory safety vulnerability: (B) and (D) are both correct answers.

Note that (A) and (C) are not correct answers because those `printf` calls do not have format string vulnerabilities (the first argument is not under attacker control).

Q7.6 (3 points) For this subpart only, assume that the code is run on a 64-bit system with stack canaries disabled but pointer authentication enabled.

Assume that you have found a vulnerability that would allow the used bits of a 64-bit address to be overwritten without touching the bits used by the PAC. Would this vulnerability, by itself, allow an attacker to execute malicious shellcode?

◯ (G) Yes, because this vulnerability leaves the PAC bits undisturbed

◯ (H) Yes, because this vulnerability overwrites the PAC bits with itself

● (I) No, because the value of the PAC depends on the address of the pointer

◯ (J) No, because the PAC is deterministic

Solution: PACs are dependant on the value of the address, so even if the attacker can leave the PAC undisturbed, changing the return would cause verification of the PAC to fail. A successful attack must overwrite the address and overwrite the PAC with a valid PAC for the new address.

Q8 *161 Meets 61A* **(23 points)**

Consider the following buggy C code:

```
1  void add_letter(int i, char *buf) {
2      char word[4];
3      printf("Enter Word %d:\n", i);
4      fgets(word, 4, stdin);
5      buf[i] = word[0];
6      if (i > 0) {
7          add_letter(i - 1, buf);
8      }
9  }
10
11 void make_acronym(void) {
12     char result[4];
13     add_letter(4, result);
14     printf("%s\n", result);
15 }
16
17 void word_games(void) {
18     make_acronym();
19 }
20
21 int main(void) {
22     word_games();
23     return 0;
24 }
```

Assume you are on a little-endian 32-bit x86 system. Assume that there is no compiler padding or additional saved registers in all subparts. Assume all memory-safety defenses (ASLR, stack canaries, pointer authentication codes, and non-executable pages) are disabled, unless otherwise specified.

Q8.1 (3 points) How many times will the `add_letter` function be run each time the `make_acronym` function is called?

○ (A) 0 ○ (B) 1 ○ (C) 2 ○ (D) 3 ○ (E) 4 ● (F) 5

Solution: Five times; i = 4, i = 3, i = 2, i = 1, and i = 0.

Q8.2 (4 points) Which value(s) will be overwritten (partially or completely) when you provide an input for the prompt to "Enter Word 4:"? Select all that apply.

☐ (G) RIP of `word_games`

☐ (H) SFP of `word_games`

☐ (I) RIP of `make_acronym`

■ (J) SFP of `make_acronym`

☐ (K) None of the above

Solution: At a high level, this code contains a buggy implementation of an acronym generator: it prompts the user for a series of words, and stores the resulting acronym (a word consisting of the first word of every letter) in `result`.

Realistically, the function should add a null byte to the end of `result`, but it doesn't. Instead, it provides the user with a mechanism to write to the zero'th, first, second, third, and fourth byte of `result`.

Because the user has no way to avoid writing to the fourth byte of `result`, the SFP of `make_acronym` (which is above `result` on the stack) will always be overwritten, which is why the function might crash during normal execution.

Assume that malicious shellcode is stored at `0x44332211` and the address of `result` is `0xAABBCCB8`. In the next five subparts, provide a series of inputs to `fgets` that would cause the program to execute shellcode.

Q8.3 (1 point) First input:

◯ (A) `\xA9` ◯ (B) `\xAC` ◯ (C) `\xB0` ● (D) `\xB4` ◯ (E) `\xB8` ◯ (F) `\xBC`

Q8.4 (1 point) Second input:

◯ (G) `\x00` ◯ (H) `\x11` ◯ (I) `\x22` ◯ (J) `\x33` ● (K) `\x44` ◯ (L) `\x48`

Q8.5 (1 point) Third input:

◯ (A) `\x00` ◯ (B) `\x11` ◯ (C) `\x22` ● (D) `\x33` ◯ (E) `\x44` ◯ (F) `\x48`

Q8.6 (1 point) Fourth input:

◯ (G) `\x00` ◯ (H) `\x11` ● (I) `\x22` ◯ (J) `\x33` ◯ (K) `\x44` ◯ (L) `\x48`

Q8.7 (1 point) Fifth input:

○ (A) \x00 ● (B) \x11 ○ (C) \x22 ○ (D) \x33 ○ (E) \x44 ○ (F) \x48

Solution:

```
print('\xB4') (&result - 4)
print('\x44') (shellcode MSB)
print('\x33')
print('\x22')
print('\x11') (shellcode LSB)
```

In an earlier sub-part, we established that we could write to the fourth byte of `result`. Because that byte is the LSB of the SFP of `make_acronym`, we can attempt to perform an off-by-one attack!

We follow the general off-by-one structure in the textbook, and in the project, where our goal is to make the SFP point to a spot somewhere lower on the stack, and then place the address of our shellcode four bytes above that. For the reasoning behind this attack structure, refer to the textbook.

To determine where to point the Forged SFP to, notice that our buffer is only four bytes large, so we have to point the Forged SFP to the memory address four bytes below the buffer. The buffer is at 0xAABBCCB8, and the original SFP is 0xAABBCCD0, so we set the last byte of the SFP to 0xB4.

Finally, we notice that our recursive `add_word` function enters inputs starting at `result[4]` and working down to `result[0]`, so we start with the most significant byte of the shellcode address and work our way to the LSB.

Q8.8 (3 points) Assume that you've successfully executed the exploit above. At what point will the function jump to your shellcode?

◯ (G) When `main` returns

● (H) When `word_games` returns

◯ (I) When `make_acronym` returns

◯ (J) When `add_letter` (called with `i == 4`) returns

◯ (K) When `add_letter` (called with `i == 3`) returns

◯ (L) None of the above

Solution: We're using a version of an off-by-one exploit here, so the `word_games` function has to return in order for the CPU to look for its RIP (and mistakenly find a malicious RIP that we've placed in our buffer instead).

Q8.9 (3 points) **For this subpart only, assume stack canaries are enabled.** Suppose the CPU generates a different 4-byte canary for each function call by taking the SHA-512 hash of the RIP and using the first 4 bytes as the canary.

Assume the attacker can execute this program in GDB. Using this scheme, which canary values would an attacker would be able to learn? Select all that apply.

■ (A) `main`

■ (B) `word_games`

■ (C) `make_acronym`

☐ (D) None of the above

Solution: If the stack canaries are generated using RIP values, then it should be trivial to inspect these values (e.g. using GDB) and construct the appropriate canaries! This is only possible because ASLR is disabled.

Q8.10 (5 points) **For this subpart only, assume stack canaries are enabled.** Assume that the CPU generates a random four-byte canary, but the least-significant byte is always `0xAA`. Which series of inputs to `fgets` will cause the program to leak the value of the stack canary? Select all that apply.

■ (G) `\xAA`, `\xAA`, `\xAA`, `\xAA`, `\xAA`

■ (H) `\xAA`, `\x11`, `\x22`, `\x33`, `\x44`

☐ (I) `\xB8`, `\xCC`, `\xFF`, `\xFF`, `\xAA`

☐ (J) `\xAA`, `\x00`, `\xFF`, `\xFF`, `\xAA`

☐ (K) `\xB8`, `\xFF`, `\xFF`, `\xFF`, `\x00`

☐ (L) None of the above

Solution:

```
print('\xAA')
print('\xFF') (any non-null string works)
print('\xFF') (any non-null string works)
print('\xFF') (any non-null string works)
print('\xFF') (any non-null string works)
```

First, observe that the canary will be placed between the SFP and the `result` buffer, so the off-by-one bug that causes one byte of `result` to be overwritten will overwrite the least significant byte of the canary. However, we're given the value of that byte - so all we need to do is ensure the rest of the buffer is filled with non-null values. The `printf` on line 14 will print starting from the address of `result` until a null-byte is encountered, so if we fill everything up to the canary with non-null bytes, the canary should be printed to the console.

Doodle Page