Nicholas Weaver Spring 2022 — 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 (149 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)

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 and a corresponding notch on Nick's Stanley Fubar demolition tool.

SIGN your name: _____

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

Each true/false is worth 2 points.

Q2.1 True or False: In a big-endian system, consecutive bytes are written from high to low addresses, rather than from low to high addresses.

○ True ● False

Solution: False. Endianness doesn't affect the direction in which consecutive bytes are written. Consecutive bytes are always written from low to high addresses, regardless of endianness.

Q2.2 True or False: It's impossible to reseed a secure PRNG with an input that would cause it to lose entropy.

● True ○ False

Solution: True. Reseeding must not reduce the amount of entropy in the PRNG's state, only increase it.

Q2.3 True or False: If an attacker compromises the internal state of a secure PRNG, and then the PRNG is reseeded with a high-entropy input that the attacker doesn't know, the attacker is no longer able to predict future outputs of the PRNG.

● True ○ False

Solution: True. Reseeding the PRNG incorporates additional entropy, which the attacker doesn't know, even if they previously compromised the PRNG's internal state.

Q2.4 True or False: Using a memory safe language is the only way to defend against all serialization vulnerabilities.

○ True ● False

Solution: False. Memory-safe language defend against memory safety vulnerabilities, which are not exactly the same as serialization vulnerabilities. Log4J is in Java, which is considered a memory safe language.

Q2.5 Consider the following symmetric encryption scheme for encrypting and decrypting messages: $\mathsf{Enc}(K, M) = M^K$, and $\mathsf{Dec}(K, C) = C^{\frac{1}{K}}$. Assume that the key K, message M, and ciphertext C are positive integers.

True or False: This scheme is IND-CPA secure.

○ True ● False

Solution: False. There's no randomness, so this scheme is deterministic, and by definition, deterministic schemes are not IND-CPA secure.

Q2.6 True or False: In Diffie-Hellman, an attacker who sees $g^a \bmod p$ and b has enough information to learn the shared secret.

● True ○ False

Solution: True. The attacker can compute $(g^a)^b \bmod p$ to learn the secret. a and b are the secrets in Diffie-Hellman, so security is lost if an attacker learns either value.

Q2.7 True or False: A secure block cipher will always map the same input to multiple outputs, since passing the same plaintext through a block cipher will yield different ciphertexts.

○ True ● False

Solution: False. An n-bit block cipher is a random permutation that maps each of the 2^n inputs to exactly one of the 2^n outputs, uniquely. It is a bijective function.

Q2.8 EvanBot designs a new variant of Diffie-Hellman that requires a second public parameter g_2. EvanBot's scheme is secure using any constant, but everyone must use the same constant.

True or False: EvanBot should explain to everyone how g_2 is generated.

● True ○ False

Solution: True. This is the idea behind nothing-up-my-sleeve numbers. Explaining how a constant was generated allows users of the system to trust that the number doesn't contain a backdoor.

For example, using a random-looking number like 10928842 might make users suspicious that EvanBot has inserted some secret backdoor into the algorithm. Using a constant with obvious significance like 12345678 makes it less likely that it was chosen to exploit a weakness in the algorithm.

Q2.9 During a rocket launch, the launch mechanism depends on the precise positioning of the Earth and the local weather at launch time.

TRUE or FALSE: A possible communications delay between a weather beacon located on the ground and the rocket launch mechanism may present a time-of-check-to-time-of-use issue.

● TRUE ○ FALSE

Solution: True. The delay might allow an attacker to change the state of the system between when a message is sent from the ground and when the message arrives at the rocket mechanism. In general, any delay in the system could present a potential TOCTTOU vulnerability.

Q2.10 Consider a modification to the one-time pad scheme for encrypting a fixed-size message M using a secret key K and initialization vector IV, as described by the following algorithm:

$$C = (IV, K \oplus M \oplus \text{SHA-256}(IV))$$

TRUE or FALSE: If we choose a unique, random IV each time we encrypt a message, then this scheme is IND-CPA secure even when we reuse a key K.

○ TRUE ● FALSE

Solution: False. Since the attacker knows the initialization vector (it's directly included in the ciphertext), the attacker can just reverse the XOR of SHA-256$(IV||M))$ to rederive the original OTP ciphertext, which effectively reduces this to OTP, which is not IND-CPA secure.

Q2.11 TRUE or FALSE: Defense in depth is recommended when protecting legacy C code from memory safety vulnerabilities.

● TRUE ○ FALSE

Solution: True. If we're dealing with a legacy C codebase, we can't rewrite the code in a memory safe language, meaning that we want to enable all memory safety protections we have access to (e.g. ASLR, canaries, etc.). This is defense in depth.

Q2.12 Consider a development tool where developers are prompted with two options when setting up a new project: a memory-unsafe language (the default option) and a memory-safe language.

True or False: This is a violation of Shannon's Maxim.

○ True ● False

Solution: False. Shannon's Maxim (which is related to security through obscurity) has nothing to do with this; rather, the principle at hand is "consider human factors." The memory-safe language should be the default option, so more people are likely to choose it!

Design in security from the start may also be a principle to use here.

Q2.13 Alice wants to verify that a public key, PK_B, belongs to Bob, and she knows that the public key PK_{CA} belongs to a trusted certificate authority. She receives a message from Bob: $\{\text{“}PK_B \text{ belongs to Bob”}\}_{SK_{\text{CA}}^{-1}}$.

True or False: Alice can now trust that PK_B belongs to Bob.

● True ○ False

Solution: True. Even though the message itself was sent from Bob, it was *signed* by the CA. Alice can verify that the CA signed the certificate using PK_{CA}, so she can trust PK_B.

Q2.14 Alice wants to verify that a public key, PK_B, belongs to Bob, and she knows that the public key PK_{CA} belongs to a trusted certificate authority. She receives a message from Mallory (whose public key is PK_M): $\left\{\{\text{“}PK_B \text{ belongs to Bob”}\}_{SK_{\text{CA}}^{-1}}\right\}_{SK_M^{-1}}$.

True or False: Alice can now trust that PK_B belongs to Bob.

● True ○ False

Solution: True. Even though Mallory signed the statement that was initially signed by the CA, since $\{\text{“}PK_B \text{ belongs to Bob”}\}_{SK_{\text{CA}}^{-1}}$ is still part of the message and has a trusted signature (you can ignore the outer signature and verify the inner signature against PK_{CA}), Alice can trust that the public key belongs to Bob.

Q2.15 True or False: Salting password hashes with an n-bit salt increases the difficulty of conducting an offline, brute-force attack on all users by a factor of n. Assume that each hash computation itself runs in constant time.

○ True ● False

Solution: False. Salting passwords increases the asymptotic complexity of brute-force attack against a database with M hashes and N possible passwords per hash from $O(M + N)$ to $O(MN)$, but the length of the salt is mostly irrelevant, and the important factor is that salts are unique per hash.

Q2.16 (0 points) True or False: Batman is EvanBot.

● True ○ False

Solution: Everyone is Batman if you look deep enough within your soul.

Q3 *The Joker's Schemes* **(17 point)**

The Joker has decided to evaluate the following encryption scheme. Assume the block cipher uses an n-bit block size, and the scheme uses a $2n$-bit $IV = IV_1 \| IV_2$, where IV_1 and IV_2 are each n bits:

$$C_1 = P_1 \oplus E_K(IV_1)$$
$$C_2 = P_2 \oplus E_K(IV_2 \oplus C_1)$$
$$C_i = P_i \oplus E_K(C_{i-2} \oplus C_{i-1})$$

Assume that the IV is sent along with the ciphertext in all instances.

Clarification issued during exam: Assume all IVs are generated per message.

Q3.1 (2 points) Write a formula to decrypt P_i (for $i > 2$) using this scheme.

Solution: $P_i = C_i \oplus E_K(C_{i-2} \oplus C_{i-1})$
One way to obtain this expression is to XOR both sides of the encryption equation by P_i.

Q3.2 (4 points) Is this scheme IND-CPA secure with a randomly generated IV? If you put yes, provide a brief justification (no formal proof necessary). If you put no, provide a strategy to win the IND-CPA game with probability greater than $\frac{1}{2}$.

● Yes ○ No

Solution: This scheme is similar to the CTR mode of encryption, in that the plaintext blocks are directly XORed with a psuedorandom stream of bits. The input to E_K is always guaranteed to be unique: IV_1 and $IV_2 \oplus C_1$ are pseudorandom, so they are unique. Because E_K is a pseudorandom permutation, the future inputs $C_{i-2} \oplus C_{i-1}$ will also be unique.
This construct is most closely related to the CFB mode of operation, which uses $C_i = P_i \oplus E_K(C_{i-1})$.

Q3.3 (4 points) Which of the following methods of choosing IV results in an IND-CPA secure scheme? Select all that apply.

- ☐ $IV = 0^{2n}$ (the bit 0 repeated $2n$ times)
- ■ $IV = H(i)$, where i is a monotonically increasing counter that increments for each message and H is a cryptographic hash function that outputs $2n$ bits
- ■ $IV = IV_1 \| IV_2$, where IV_1 is a randomly chosen, n-bit number and $IV_2 = 0^n$
- ☐ $IV = IV_1 \| IV_2$, where $IV_1 = 0^n$ and IV_2 is a randomly chosen, n-bit number
- ☐ None of the above

Solution: Using a constant IV (0^{2n}) doesn't introduce any randomness into the scheme, so the scheme would be deterministic, making it not IND-CPA.

Because the IV is always unique (even though it is predictable), the output of the block cipher is unpredictable, and all future encryptions using the block cipher result in further unpredictable outputs, similar to CBC mode. When XORed with the plaintext, this results in an IND-CPA secure scheme.

Q3.4 (4 points) The Joker encrypts a 5-block long message and sends it to the Mob. Batman intercepts the encrypted message and changes the second block of the cipher text C_2. Which of the following blocks of plaintext no longer decrypts to its original value? Select all that apply.

- ☐ P_1
- ■ P_2
- ■ P_3
- ■ P_4
- ☐ P_5
- ☐ None of the above

Solution: Based on the decryption formula, decrypting plaintext block i requires ciphertext blocks $i-2$, $i-1$, and i, so P_2, P_3, and P_4 would be corrupted.

Q3.5 (3 points) Which of the following statements are true about this encryption scheme? Select all that apply.

- ☐ Encryption is parallelizable
- ■ Decryption is parallelizable
- ☐ If C is the ciphertext of M, then $C' = C \oplus x$ decrypts to the plaintext $M \oplus x$
- ☐ None of the above

Solution: Encryption is not parallelizable because each ciphertext block relies on the previous ciphertext block. However, decryption is parallelizable since no decryption relies on any other plaintext block.

Even though this scheme is kind of like a stream cipher (like CTR mode), flipping a bit of the ciphertext (i.e. XORing with x) doesn't result in exactly one bit flipped in the plaintext. If a bit in ciphertext block C_i is flipped, then the decryption of P_{i+1} and P_{i+2} will be gibberish, because $E_K(C_{i-2} \oplus C_{i-1} \oplus x)$ results in a completely different output since E_K is a random permutation.

Q4 *The Red Hood* **(15 point)**

Jason Todd decides to launch a communications channel in order to securely communicate with the Red Hood Gang over an insecure channel. Jason wants to test different schemes in his attempt to attain confidentiality and integrity.

Notation:

- M is the message Jason sends to the recipient.
- K_1, K_2, and K_3 are secret keys known to only Jason and the recipient.
- ECB, CBC, and CTR represent block cipher encryption modes for a secure block cipher.
- Assume that CBC and CTR mode are called with randomly generated IVs.
- H is SHA2, a collision-resistant, one-way hash function.
- HMAC is the HMAC construction from lecture.

Decide whether each scheme below provides confidentiality, integrity, both, or neither. For all question parts, the ciphertext is the value of C; t **is a temporary value that is not sent as part of the ciphertext.**

Q4.1 (3 points)

$$t = \mathsf{CBC}(K_1, M) \qquad C_1 = \mathsf{ECB}(K_2, t) \qquad C_2 = \mathsf{HMAC}(K_3, t) \qquad C = (C_1, C_2)$$

- ○ Confidentiality only
- ○ Integrity only
- ● Both confidentiality and integrity
- ○ Neither confidentiality nor integrity

Solution: This is a typical encrypt-then-MAC scheme with a twist: Instead of including the ciphertext t directly, the ciphertext (but not the MAC) is additionally encrypted with ECB mode. Even though both the HMAC and ECB leak information about t, t doesn't leak information about the plaintext, so the scheme is confidential. The HMAC over t ensures that the input passed to CBC decryption can't be tampered with, so the scheme maintains integrity.

Q4.2 (3 points)

$$t = \mathsf{ECB}(K_1, M) \qquad C_1 = \mathsf{CBC}(K_2, t) \qquad C_2 = \mathsf{HMAC}(K_3, t) \qquad C = (C_1, C_2)$$

- ○ Confidentiality only
- ● Integrity only
- ○ Both confidentiality and integrity
- ○ Neither confidentiality nor integrity

Solution: Notice that t leaks information about the message because it uses insecure ECB mode. C_2 then leaks information about t, which leaks information about the plaintext, so confidentiality is lost (in this case, C_2 is deterministic). However, because the HMAC is computed over t, which is decryptable to the message, integrity is maintained.

Q4.3 (3 points)

$$C_1 = \mathsf{ECB}(K_1, M) \qquad C_2 = H(K_2 \| C_1) \qquad C = (C_1, C_2)$$

- ○ Confidentiality only
- ○ Integrity only
- ○ Both confidentiality and integrity
- ● Neither confidentiality nor integrity

Solution: C_1 leaks information about M it uses insecure ECB mode, so confidentiality is lost. C_2 does not maintain integrity as it vulnerable to length extension attacks—an attacker could forge $C_2' = H(K_2 \| C_1 \| x)$ and $C_1' = C_1 \| x$, which would be accepted by anyone verifying the hash.

Q4.4 (3 points) For this subpart only, assume that i a monotonically, increasing counter incremented per message.

$$C_1 = \mathsf{CTR}(K_1, M) \qquad C_2 = \mathsf{HMAC}(i, H(C_1)) \qquad C = (C_1, C_2)$$

Clarification issued during exam: Assume that the counter, i, starts at 0.

- ● Confidentiality only
- ○ Integrity only
- ○ Both confidentiality and integrity
- ○ Neither confidentiality nor integrity

Solution: Because i is a known value, the key to the HMAC can be predicted, and the scheme does not maintain integrity. However, since the ciphertext is encrypted with secure CTR mode, and the insecure HMAC is computed only over the ciphertext, the scheme maintains confidentiality.

Q4.5 (3 points) For this subpart only, assume that the block size of block cipher is n, the lengths of K_1 and K_2 are n, the length of M must be $2n$, and the length of the hash produced by H is $2n$.

$$C_1 = \mathsf{CBC}(K_1, K_2) \qquad C_2 = M \oplus C_1 \oplus H(C_1) \qquad C = (C_1, C_2)$$

- ○ Confidentiality only
- ○ Integrity only
- ○ Both confidentiality and integrity
- ● Neither confidentiality nor integrity

Solution: Notice that the attacker already knows the value of C_1 since it is sent with the ciphertext. Because of this, the adversary can just compute $H(C_1)$ then $C_2 \oplus C_1 \oplus H(C_1)$ in order to recover M, so the scheme is not confidential. Additionally, there is no MAC, so the scheme does not have integrity.

Q5 *Ra's Al Gamal* **(10 point)**

Recall the ElGamal scheme from lecture:

- KeyGen$() = (b, B = g^b \bmod p)$
- Enc$(B, M) = (C_1 = g^r \bmod p, C_2 = B^r \times M \bmod p)$

Q5.1 (3 points) Is the ciphertext (C_1, C_2) decryptable by someone who knows the private key b? If you answer yes, provide a decryption formula. You may use C_1, C_2, b, and any public values.

● Yes ○ No

Solution: The decryption formula is $M = C_1^{-b} \times C_2$.

Q5.2 (4 points) Consider an adversary that can efficiently break the discrete log problem. Can the adversary decrypt the ciphertext (C_1, C_2) without knowledge of the private key? If you answer yes, briefly state how the adversary can decrypt the ciphertext.

● Yes ○ No

Solution: An adversary that can break the discrete log problem can recover r from $C_1 = g^r$ or b from $B = g^b$, so they can compute g^{br} and recover the original message.

Q5.3 (3 points) Consider an adversary that can efficiently break the Diffie-Hellman problem. Can the adversary decrypt the ciphertext (C_1, C_2) without knowledge of the private key? If you answer yes, briefly state how the adversary can decrypt the ciphertext.

● Yes ○ No

Solution: An adversary that can break the Diffie-Hellman problem can recover g^{br} from $C_1 = g^r$ and $B = g^b$, so they can recover the original message.

Q6 *Probability of MANBAT* **(21 point)**

Consider the following vulnerable C code:

```
1 void unsafe() {
2     char buffer[8];
3     gets(buffer);
4 }
5
6 int main(void) {
7     unsafe();
8     return 0;
9 }
```

Assume that there is no compiler padding or additional saved registers in all subparts. Also assume that `SHELLCODE` represents 8-byte shellcode.

You run this code in GDB once and discover that the address of `buffer` is `0x3a9d2800`.

Q6.1 (3 points) *Suppose that no memory safety defenses are enabled.* Which of the following exploits would cause shellcode to execute? Select all that apply.

- ■ `'A' * 12 + '\x10\x28\x9d\x3a' + SHELLCODE`
- ■ `SHELLCODE + 'A' * 4 + '\x00\x28\x9d\x3a'`
- ■ `'A' * 4 + SHELLCODE + '\x04\x28\x9d\x3a'`
- ☐ None of the above

Solution: This is a simple `gets` buffer overflow. The stack frame looks something like this: |

RIP of `unsafe`
SFP of `unsafe`
`buf`
`buf`

Since the SHELLCODE fits within the 12 bytes of space, all we need to do is place the SHELLCODE somewhere within that space, fill the rest of the buffer with garbage bytes, and then point the RIP to the address of the shellcode. Another possible solution is to completely fill up the buffer with garbage bytes, place the SHELLCODE immediately after the RIP, and make the RIP point to RIP+4, which is the address of the SHELLCODE (just like Q1 of Project 1)

For the rest of this question, if ASLR is enabled, each segment of memory is exactly `0x10000` bytes long, and each starting address always has `0x0000` as the lower (least significant) bits and has 16 random upper (most significant) bits. For example, the stack segment might be located between addresses `0x3a9d0000` to `0x3a9e0000`, but it will not be located between addresses `0x3a9d0100` to `0x3a9e0100`, because the bottom 16 bits are not all zeros.

With ASLR enabled, you run the program in GDB three times and print the address of `buffer` each time. You see the following addresses: `0xef062800`, `0x2aec2800`, and `0x10702800`.

Q6.2 (3 points) *Suppose that ASLR is enabled, and no other memory safety defenses are enabled.* Consider the following exploit: `'A' * 12 + '\x10\x28\x9d\x3a' + SHELLCODE`.

What is the approximate probability that the above exploit will work on any given execution of the program?

- ○ 0
- ● $1/2^{16}$
- ○ $1/2$
- ○ $1/2^{32}$
- ○ $1/2^{4}$
- ○ 1

Solution: The upper 16 bits will change each time each time the program is executed, but the relative addresses on the stack will not change, so the lower bits will stay consistent.

The probability that the random upper 16 bits match our 16-bit guess is $1/2^{16}$.

Q6.3 (3 points) *Suppose that ASLR and stack canaries are enabled, and no other memory safety defenses are enabled.* Assume that the stack canary is four completely random bytes (no null byte).

Consider the following exploit: `'A' * 16 + '\x14\x28\x9d\x3a' + SHELLCODE`.

What is the approximate probability that the above exploit will work on any given execution of the program?

- ○ 0
- ● $1/2^{48}$
- ○ $1/2^{16}$
- ○ $1/2^{16\times 32}$
- ○ $1/2^{32}$
- ○ 1

Solution: There are 16 bits of entropy from ASLR and 32 bits of entropy from the stack canary. In total, there are 48 bits we need to guess.

Note that the extra 4 bits of padding are there to overwrite the stack canary. With probability $1/2^{32}$, the stack canary will just happen to be four `A`s and match what we overwrote the canary with.

Q6.4 (4 points) Which of the following additional vulnerabilities in the code would increase the probability of success of your exploit from the previous part? Select all that apply.

Clarification during exam: This question part has been dropped. Everyone will receive points on this question part.

- ■ A vulnerability that lets you read any memory in the program
- ■ A vulnerability that lets you read any memory from the stack only
- ■ A bug that causes one byte of the stack canary to always be `0x61`
- ■ A null byte in the stack canary
- ☐ None of the above

Solution: This question has been dropped.
The intent of this question was that the exploit could be tweaked to accommodate for the last two options, but as that was not clear, the question has been dropped. The intended answer explanation is that (a) a bug that decreases the amount of entropy (randomness) in the canary makes the canary easier to brute-force, and (b) reading memory on the stack (or anywhere in the program) will leak addresses that can be used to break ASLR.

Q6.5 (3 points) *Suppose that ASLR, stack canaries, and non-executable pages are enabled, and no other memory safety defenses are enabled.* Assume that the stack canary is four completely random bytes (no null byte).

Consider the following exploit: `'A' * 16 + '\x14\x28\x9d\x3a' + SHELLCODE`

What is the approximate probability that the above exploit will work on any given execution of the program?

- ● 0
- ○ $1/2^{48}$
- ○ $1/2^{16}$
- ○ $1/2^{16\times 32}$
- ○ $1/2^{32}$
- ○ 1

Solution: Non-executable pages prevents the attacker from executing any code they wrote into memory, so this exploit now has a probability of zero.

Q6.6 (3 points) Consider the following modified version of the original code:

```
1  char buffer[8];
2
3  void unsafe() {
4      gets(buffer);
5  }
6
7  int main(void) {
8      unsafe();
9      return 0;
10 }
```

Suppose that ASLR is enabled, and no other memory safety defenses are enabled.

As before, you run GDB and discover that the address of `buffer` is `0x3a9d2800`.

Consider the following exploit: `'A' * 12 + '\x10\x28\x9d\x3a' + SHELLCODE`.

What is the approximate probability that the above exploit will cause malicious shellcode to execute on any given execution of the program?

- ● 0
- ○ $1/2^{16}$
- ○ $1/2$
- ○ $1/2^{32}$
- ○ $1/2^{4}$
- ○ 1

Solution: The main difference here is that the buffer has been relocated to static memory. Overflowing the buffer in static memory will not overwrite the RIP of any function, so this exploit will almost certainly not cause the shellcode to execute.

Q6.7 (2 points) Generally, if an exploit succeeds with probability $1/2^{20}$, an attacker might try the exploit 2^{20} times until it succeeds. However, whether an attacker is willing to try 2^{20} times depends on whether the code has a timeout after each failed attempt, and whether the code is even worth attacking in the first place.

Which security principle or example is most relevant to this situation?

- ○ Least privilege
- ○ Don't rely on security through obscurity
- ○ Fail-safe defaults
- ● Know your threat model
- ○ Time-of-check to time-of-use

Solution: Thinking about an attacker's capabilities and motivations is related to knowing your threat model. One threat model may allow an attacker to try 2^{20} times, and another may not.

Q7 *Robin* **(29 point)**

Consider the following code snippet:

```
1  void robin(void) {
2      char buf[16];
3      int i;
4
5      if (fread(&i, sizeof(int), 1, stdin) != 1)
6          return;
7
8      if (fgets(buf, sizeof(buf), stdin) == NULL)
9          return;
10
11     _________
12 }
```

Clarification issued during exam: `fread` returns the number of members read. `fgets` returns NULL if an error occurs.

Clarification issued during exam: If you put "possible," writing "not needed" for a line means that any input to that line would work for the exploit.

Assume that:

- There is no compiler padding or additional saved registers.
- The provided line of code in each subpart compiles and runs.
- `buf` is located at memory address `0xffffd8d8`
- Stack canaries are enabled, and all other memory safety defenses are disabled.
- The stack canary is four completely random bytes (**no null byte**).

For each subpart, mark whether it is possible to leak the value of the stack canary. If you put possible, provide an input to Line 5 and an input to Line 8 that would leak the canary. If the line is not needed for the exploit, you must write "Not needed" in the box.

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

Q7.1 (3 points) Line 11 contains `printf("%x", buf[i]);`.

Clarification during exam: This question part has been dropped. Everyone will receive points on this question part.

● Possible ● Not possible

Line 5:

Solution: N/A (intended to be `'\x10\x00\x00\x00'`)

Line 8:

Solution: N/A

Solution: This was intended to be a simple out-of-bounds memory read by passing in `i == 16`. However, when clarifications for this question came in, we realized that this doesn't actually work: `buf` is a `char` array, so indexing into `buf[i]` returns only a single character. This character would then be cast to a 4-byte argument, but it still only contains the byte in the least-significant position.

Technically, the solution is "Not possible," but because the knowledge needed for this question was ultimately not what we intended to test, the question part was dropped.

Q7.2 (3 points) Line 11 contains `printf("%s", buf[i]);`.

Clarification during exam: This question part has been dropped. Everyone will receive points on this question part.

● Possible ● Not possible

Line 5:

Solution: Not possible (intended to be `'\x00\x00\x00\x00'`)

Line 8:

Solution: Not possible (intended to be `'\xe8\xd8\xff\xff`)

Solution: As before, the intent was for the student to set up `buf[i]` to contain the address of the canary so that `%s` would dereferencing the expected string and leak the value of the canary. However, because `buf[i]` only returns a single byte, the resulting pointer will always be (probably) invalid with only the lowest byte set and the top three bytes set to NULL.

Technically, the solution is "Not possible," but because the knowledge needed for this question was ultimately not what we intended to test, the question part was dropped.

Q7.3 (3 points) Line 11 contains `gets(buf);`.

○ Possible ● Not possible

Line 5:

Solution: N/A

Line 8:

Solution: N/A

Solution: There's not much we can do here as an attacker: there's no way to execute arbitrary shellcode to leak the canary, because we'd have to bypass the canary somehow; and there's no way of leaking the canary value directly as there are no read commands, only write commands.

Q7.4 (3 points) Line 11 contains `printf("%s", buf);`.

○ Possible ● Not possible

Line 5:

Solution: Not possible

Line 8:

Solution: Not possible

Solution: There's not much we can do here as an attacker: `fread` and `fgets` are used correctly, and we've safely escaped our `printf` call here.

Q7.5 (5 points) **For this subpart only, enter an input that allows you to leak a single character from memory address `0xffffd8d7`. Mark "Not possible" if this is not possible.** Line 11 contains `printf("%c", buf[i]);`.

● Possible ○ Not possible

Line 5:

Solution: `'\xff\xff\xff\xff'`

Line 8:

Solution: Not needed

Solution: We can set `i` to -1 to read a value one byte below the buffer. We know that -1 is `0xffffffff` in two's complement, so we just enter that for the integer.

Q7.6 (6 points) Line 11 contains `printf(buf);`.

● Possible ○ Not possible

Line 5:

Solution: Not needed

Line 8:

Solution: `'%c%c%c%c%c%x'`

Solution: This is just a simple format string attack: We just need to walk our way up the stack using `%c` specifiers until we reach `canary`, at which point we can dump the value of the canary using a `%x`.

Q7.7 (6 points) Line 11 contains `printf(i);`.

● Possible ○ Not possible

Line 5:

Solution: Approach 1: `'\xe8\xd8\xff\xff'`
Approach 2: `'\xd8\xd8\xff\xff'`

Line 8:

Solution: Approach 1: Not needed
Approach 2: `'%c%c%c%c%c%x'`

Solution: The first option is simple: Use the integer as a pointer directly to the stack canary, which causes it to be leaked since it's contents will be treated as the format string and directly printed out (since it's unlikely for it to contain a format specifier).

The second option is identical to the previous subpart, except for the fact that we're printing `i` instead of `buf` - as such, we need to set this up such that `i` is a pointer to the format string specifier, which resides at `buf`. We can do this by setting `i` to this address, so that when it's passed into `printf`, it's treated identically to passing in `buf` directly.

Q8 *Copperhead* **(26 point)**

Consider the following vulnerable C code:

```
1  struct vtable {
2      void (*hiss)(void);  // function pointer
3      void (*snack)(void); // function pointer
4      void (*bite)(void);  // function pointer
5  };
6
7  struct snake {
8      char name[16];
9      struct vtable *vtable_ptr;
10 };
11
12 void snake_bite(void) { printf("ouch!\n"); }
13 void viper_bite(void) { printf("ouch + venom!\n"); }
14
15 int main(void) {
16     struct vtable snake_vtable = { NULL, NULL, snake_bite };
17     struct vtable viper_vtable = { NULL, NULL, viper_bite };
18     struct snake copperhead;
19     char venom[32];
20     int i;
21
22     /* Make copperhead. */
23     copperhead.vtable_ptr = &viper_vtable;
24     fgets(venom, sizeof venom, stdin);
25     for (i = 0; i <= 16; i++) {
26         copperhead.name[i] = venom[i];
27     }
28
29     copperhead.vtable_ptr->bite();
30
31     return 0;
32 }
```

Assume you are on a little-endian 32-bit x86 system. Assume that there is no compiler padding or saved additional registers in all questions. Assume there are **no memory safety defenses enabled**.

Q8.1 (5 points) Fill in the following stack diagram, assuming that the program is paused at **Line 22**. There are no extra rows. Each row should contain one struct member or variable from the following (not all options will be used):

`copperhead.name`	`copperhead.vtable_ptr`	address of `fgets`
`i`	address of `printf`	`snake_vtable.bite`
`snake_vtable.hiss`	`snake_vtable.snack`	`venom`
`viper_vtable.bite`	`viper_vtable.hiss`	`viper_vtable.snack`

Stack

RIP of `main`
SFP of `main`

Solution:

RIP of `main`
SFP of `main`
`snake_vtable.bite`
`snake_vtable.snack`
`snake_vtable.hiss`
`viper_vtable.bite`
`viper_vtable.snack`
`viper_vtable.hiss`
`copperhead.vtable_ptr`
`copperhead.name`
`venom`
`i`

Q8.2 (3 points) Which of the following lines contains a memory safety vulnerability?

- ○ Line 12
- ○ Line 16
- ○ Line 19
- ○ Line 23
- ○ Line 24
- ● Line 25

Solution: The vulnerable line of code is Line 25 as there is a <= operator which will cause an off-by-one vulnerability.

Line 12 is not vulnerable as it is a function definition and the `printf` statement inside of the function does not take any arguments from the attacker.

Line 16 is not vulnerable as it simply is the initialization of the `struct vtable`.

Line 19 is not vulnerable as it is the initialization of the `venom` array.

Line 23 is not vulnerable as it is setting the `vtable_ptr` to point to the address of `viper_vtable`.

Line 24 is not vulnerable as the length of the input to `venom` is controlled by the `fgets` statement, so we cannot overflow `venom`.

Q8.3 (10 points) Assume that, at Line 22, the value of the ESP is `0xffff9308`. Provide an input to the program that will cause a malicious 8-byte shellcode to be executed. You may reference the variable `SHELLCODE` in your exploit. Write your answer in Python 2 syntax (just like in Project 1).

Solution: Off-by-one overwrites the LSB of `copperhead.vtable_ptr`, so we can trick the program into thinking that the `struct vtable` exists at the start of the `venom` buffer. Based on this, we can place a pointer to the shellcode at the location at which the program will look for the `bite` function (eight bytes offset from the start of the vtable), and place the shellcode in the remaining space that we have.

```
0xffff9348 [ ][ ][ ][ ]     viper_vtable.bite
0xffff9344 [ ][ ][ ][ ]     viper_vtable.snack
0xffff9340 [ ][ ][ ][ ]     viper_vtable.hiss
0xffff933c [40 --> X = 0c][93][ff][ff] copperhead.vtable_ptr

0xffff9338 [ ][ ][ ][ ]     copperhead.name
0xffff9334 [ ][ ][ ][ ]     copperhead.name
0xffff9330 [ ][ ][ ][ ]     copperhead.name
0xffff932c [ ][ ][ ][ ]     copperhead.name

0xffff9328 [ ][ ][ ][ ]venom
0xffff9324 [ ][ ][ ][ ] venom
0xffff920 [ ][ ][ ][ ] venom
0xffff931c [ ][ ][ ][ ]    venom

0xffff9318 ['A']['A']['A']['A']     venom
0xffff9314 [0c][93][ff][ff] venom  [fake viper_vtable.bite]
0xffff9310 [SH][SH][SH][SH] venom  [fake viper_vtable.snack]
0xffff930c [SH][SH][SH][SH] venom  [fake viper_vtable.hiss]

0xffff9308 [ ][ ][ ][ ]     i

answer: SHELLCODE + '\x0c\x93\xff\xff\' + 'A' * 4 + '\x0c'
```

Q8.4 (3 points) Assume an attacker has successfully carried out the above exploit. At what point will execution jump to shellcode?

- ○ When Line 24 is executed
- ○ When Line 26 is executed
- ● When Line 29 is executed
- ○ When `snake_bite` returns
- ○ When `viper_bite` returns
- ○ When `main` returns

Solution: The basis of our exploit relied on changing the `vtable_ptr` to point to inside our buffer, where it will interpret the 3rd word above the beginning of the pointer as the new address of `bite`. Because the address of the `bite` function pointer now appears to be our shellcode, it will execute when `bite` is called on line 29.

Q8.5 (5 points) Which of the following memory safety defenses would prevent an attacker from executing the exploit above? Select all that apply.

- ☐ Stack canaries
- ■ ASLR
- ■ Pointer authentication (assuming a 64-bit system)
- ■ Non-executable pages
- ■ Rewriting the code in a memory-safe language
- ☐ None of the above

Solution: Stack canaries would not defend against out exploit since we are not overwriting the rip. With this off-by-one vulnerability, we aren't even able to reach the RIP.

ASLR and pointer authentication would protect against our exploit as there are no printf vulnerabilities or magic numbers which would allow us to leak addresses.

Implementing non-executable pages would also protect against this exploit as we are writing our shellcode into venom and copperhead.name, thus those segments of memory would be non-executable. As a result we won't be able to execute our malicious shellcode.

Rewriting the code in a memory-safe language should also prevent the exploit since it prevents memory safety vulnerabilities.