CS 161 Spring 2026 — Introduction to Computer Security — Midterm

Name: _____

Student ID: _____

This exam is 110 minutes long. There are 7 questions of varying credit. (100 points total)

Question:	1	2	3	4	5	6	7	Total
Points:	0	10	16	22	9	24	19	100

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

- ○ Unselected option (Completely unfilled)
- ◎ Don't do this (it will be graded as incorrect)
- ● 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).
- ☑ (Don't do this)

Anything you write outside the answer boxes or you ~~cross out~~ will not be graded. If you write multiple answers, your answer is ambiguous, or the bubble/checkbox is not entirely filled in, we may grade the worst interpretation.

Pre-Exam Activity: (0 points): EvanBot is a hungry food critic! What does bot get to try tonight?

Art by: Soklynin Nou

Q1 *Honor Code* 📜 (0 points)

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.

Read the honor code above and sign your name: _____

This page left intentionally (mostly) blank

The exam continues on the next page.

Q2 *Potpourri*  **(10 points)**

Each question is worth 1 point.

Q2.1 EvanBot configures a file-sharing application so that it can only access a specific folder instead of the entire filesystem.

TRUE OR FALSE: This is an example of the principle of least privilege.

● TRUE ○ FALSE

Solution: This is true. Least privilege requires granting only the permissions necessary for correct functioning. Restricting access to a specific folder limits damage if the application is compromised.

For Q2.2–Q2.3, consider the C code and GDB output stopped at line 7.

C code:

```
1  void callee() {
2    char buf[16];
3    gets(buf);
4  }
5
6  void caller() {
7    callee();
8  }
```

GDB Output (line 7):

```
1  info frame
2  ...
3  Saved registers:
4  ebp at 0xffffdc58,
5  eip at 0xffffdc5c
```

GDB Output (line 3):

```
1  x/1x buf
2  0xffffdc60: 0xdeadbeef
```

Q2.2 TRUE OR FALSE: Running `print buf` will display the bytes in the local array buf.

○ TRUE ● FALSE

Solution: False. `buf` is local to `callee`, so it is only in scope while execution is inside `callee`. If GDB is currently stopped in `caller`, `buf` is out of scope.

Q2.3 TRUE OR FALSE: `0xffffdc5c` is the address on the stack where the RIP of `caller` is stored.

● TRUE ○ FALSE

Solution: True. `saved eip at ...` reports the stack location that stores the saved instruction pointer. To see the saved return address value, you would examine memory at that address (for example, `x/xw 0xffffdc5c` on 32-bit).

For Q2.4–Q2.5, consider the C code and GDB output stopped at line 3.

Q2.4 TRUE OR FALSE: `0xffffdc60` is the address of the first byte of `buf`.

● TRUE ○ FALSE

Solution: True. In the command `x/1x buf`, GDB examines memory starting at the address stored in `buf`, which is the address of the first element of the array. Therefore `0xffffdc60` is the address of the first byte of `buf`.

Q2.5 True or False: `0xde` is the first byte of `buf`.

○ True ● False

Solution: Because the program is in little-endian, The first byte is `0xef`

Q2.6 Which of these properties should a pseudo-random function exhibit? Select all that apply.

■ Deterministic ■ Efficient to compute ○ None of the above

■ Keyed ☐ Randomized on every call

Solution: A PRF should be deterministic, keyed, and efficient to evaluate, and it should be computationally indistinguishable from a truly random function for any efficient adversary without the key. It should **not** be randomized on each call because PRFs are deterministic by definition.

Q2.7 True or False: If a function is one-way, it is guaranteed to be a collision-resistant hash function.

○ True ● False

Solution: False. If a function is a CRHF then it is guaranteed to be one-way.

Q2.8 True or False: An adversary with a polynomial-time factoring algorithm can break the hashed ElGamal encryption scheme.

○ True ● False

Solution: The security of the hashed ElGamal is tied to the hardness of CDH / discrete-log-style assumptions, not factoring. Factoring is associated with RSA instead.

Q2.9 True or False: Merkle puzzles are more secure than Diffie-Hellman key exchange because they provide a larger asymptotic computational gap between the honest parties and the attacker.

○ True ● False

Solution: This is false. Merkle puzzles provide only a polynomial gap ($O(N)$ for Alice/Bob vs. $O(N^2)$ for the attacker). Diffie-Hellman, in contrast, relies on subexponential or exponential hardness assumptions (e.g., roughly $2^{N^{\frac{1}{3}}}$ or $2^{\frac{N}{2}}$ for known attacks), which grow much faster than any polynomial. Therefore, Diffie-Hellman provides a significantly stronger asymptotic security guarantee.

Q2.10 True or False: Deterministic encryption schemes can be secure against chosen-plaintext attacks.

○ True ● False

Solution: This is false. CPA security requires that an encryption scheme hide all information about the plaintext even when the attacker can request encryptions of messages of its choice. A deterministic encryption scheme always produces the same ciphertext for the same plaintext, so an attacker can simply encrypt the challenge messages itself and compare the outputs. This lets the attacker distinguish which message was encrypted with probability 1. Therefore, deterministic encryption cannot satisfy chosen plaintext security.

Q3 *Memory Safety: Football's Coming Home* 🏴 **(16 points)**

Consider the following vulnerable C code:

```
1  void somethingBritish() {
2      int value = 161;
3      char buf[24];
4      fread(buf, 40, 1, stdin);
5      memcpy((buf + value), *(void**)buf, 4);
6      if (value > 32) {
7        exit(0);
8      }
9  }
10
11 void main() {
12     somethingBritish();
13 }
```

Stack at Line 5

RIP of `main`
SFP of `main`
canary
RIP of `somethingBritish`
SFP of `somethingBritish`
(1)
(2)
(3)

Assumptions:

- Stack canaries are enabled and consist of non-null bytes.
- All other memory safety mitigations are disabled.
- You run GDB once and find that the RIP of `somethingBritish` is located at `0xffffdc90`.
- Your goal is to place and execute a 20-byte `SHELLCODE`.

(1 point each) What goes in the blanks (1) - (3) in the stack diagram above?

Q3.1 Blank (1): ○ `value` ○ `buf` ● canary

Q3.2 Blank (2): ● `value` ○ `buf` ○ canary

Q3.3 Blank (3): ○ `value` ● `buf` ○ canary

Solution: STACK

Address	Content
`0xffffdc9c`	`[4]` RIP of `main`
`0xffffdc98`	`[4]` SFP of `main`
`0xffffdc94`	`[4]` canary
`0xffffdc90`	`[4]` RIP of `somethingBritish`
`0xffffdc8c`	`[4]` SFP of `somethingBritish`
`0xffffdc88`	`[4]` canary
`0xffffdc84`	`[4]` `value`
`0xffffdc6c`	`[24]` `buf`

Q3.4 (2 points) Which of the following memory safety vulnerabilities are present in this code? Select all.

- ☐ Format string vulnerability
- ☐ Signed/unsigned vulnerability
- ■ Stack overflow
- ○ None of the above

Solution:

- `buf` is `char buf[24];` but `fread(buf, 40, 1, stdin);` reads **40 bytes** into it, overflowing `buf` by 24 bytes and smashing adjacent stack data .
- No format-string bug: `buf` is never used as a format string.
- No signed/unsigned issue: the bug is not from conversion; it's the fixed oversized read.

Q3.5 (6 points) Provide an input to `fread` on Line 4 that would cause the program to execute `SHELLCODE`.

If a part of the input can be any non-zero value, use `'A' * n` to represent `n` bytes of garbage.

Hint: `*(void**)X` *treats the value stored at address* `X` *as a pointer to another location in memory.*

Input to `fread` on line 4:

```
'\x94\xdc\xff\xff'+ SHELLCODE + '\x1c\x00\x00\x00' +
'A' * 8 + '\x70\xdc\xff\xff'
```

Solution: We exploit the fact that `fread` overflows `buf`, but then we get **one 4-byte arbitrary write** via

```
memcpy((buf + value), *(void**)buf, 4);
```

before the function returns (and before the canary check runs at epilogue).

- `buf` starts at `0xffffdc6c`
- `value` is at `buf + 24`
- canary of `SomethingBritish` is at `buf + 28` (address `0xffffdc88`)
- canary of `main` is at `0xffffdc94`
- `SHELLCODE` is at `buf + 4 = 0xffffdc70`

Plan:

1. Set the first 4 bytes of `buf` to the address of `main`'s canary (`0xffffdc94`), so `*(void**)buf = 0xffffdc94`.
2. Overflow `value` to 28 (so the destination `buf + value` becomes `buf + 28`, i.e. the canary slot of `SomethingBritish`), and also keep `value <= 32` to pass the `if (value > 32) exit(0)` check.
3. Overflow the saved RIP to `0xffffdc70` so control flow returns into our `SHELLCODE` in `buf`.
4. Even though `fread` smashes the canary, the subsequent `memcpy` **repairs** it by copying the real canary bytes from `0xffffdc94` into `buf+28`.

Field-by-field:

- `'\x94\xdc\xff\xff'` : makes `*(void**)buf` point to `main`'s canary.
- `SHELLCODE` : 20 bytes placed at `buf+4`.
- `'\x1c\x00\x00\x00'` : overwrites `value` with 28 so `memcpy` writes into the canary slot and the check passes.
- `'A' * 8` : overwrites canary + SFP
- `'\x70\xdc\xff\xff'` : overwrites RIP to jump to `buf+4` upon function return.

Q3.6 (3 points) Which changes would cause the correct exploit (without modification) to fail? Consider each choice independently. Select all that apply.

- [x] Line 3: Change `char buf[24];` to `char buf[32];`.
- [ ] Line 2: Change `int value = 161;` to `unsigned int value = 161;`.
- [x] Line 4: Swap this line with line 5, so `memcpy` happens before `fread`.
- [x] Line 4: Change `fread(buf, 40, 1, stdin);` to `fread(buf, 32, 1, stdin);`.
- ○ None of the above

Solution:

The following changes would make the exploit fail:

- **Line 3:** `char buf[24];` → `char buf[32];` You can no longer overwrite the RIP, and you can't even overwrite the SFP.
- **Line 4:** swap line 4 and line 5 This would no longer allow you to replace the value of the canary after your stack smashing attack
- **Line 4:** `fread(buf, 40, 1, stdin);` → `fread(buf, 32, 1, stdin);` The payload can no longer overflow far enough to reach the saved RIP with the same input.

Changing `int value` to `unsigned int value` does **not** break the exploit, because the exploit writes `28`, and `28 <= 32` whether `value` is signed or unsigned.

Q3.7 (2 points) Would the correct exploit (without modification) fail if ASLR was enabled?

- ● Yes, because the absolute address of the stack changes every time the program runs.
- ○ Yes, because ASLR makes the stack non-executable, preventing the shellcode inside `buf` from running even if the RIP is overwritten.
- ○ No, because the relative distance between `buf` and the RIP remains constant regardless of the absolute address.
- ○ No, because ASLR only randomizes the location of the heap and shared libraries, not the stack where `buf` is located.

Solution: Yes. The exploit would fail with ASLR enabled because it hardcodes absolute stack addresses, such as the address of the canary and the address of the shellcode in `buf`. With ASLR, those stack addresses change each run, so the overwritten RIP and the pointer used by `memcpy` would no longer be correct.

Q4 *Memory Safety: Stephen Curry* 🏀 **(22 points)**

Consider the following vulnerable C code:

```
1  void Kyrie() {
2      printf(_____________________);
3  }                    Q4.5
4  void LeBron() {
5      char buf[64];
6      fread(buf, 64, 1, stdin);
7      Kyrie();
8  }
9
10 void main() {
11     LeBron();
12 }
```

Stack at Line 2

RIP of (1)
SFP of (1)
RIP of (2)
SFP of (2)
`buf`
RIP of (3)
SFP of (3)

Assumptions:

- All memory safety defenses are disabled.
- You run GDB once and find that the RIP of `LeBron` is located at `0xffffdc34`.
- Your goal is to place and execute a 32-byte `SHELLCODE`.

(1 point each) What goes in the blanks (1) - (3) in the stack diagram above?

Q4.1 Blank (1): ● `main` ○ `LeBron` ○ `Kyrie`

Q4.2 Blank (2): ○ `main` ● `LeBron` ○ `Kyrie`

Q4.3 Blank (3): ○ `main` ○ `LeBron` ● `Kyrie`

Solution:

Address	Content
`0xffffdc3c`	`[4]` RIP of `main`
`0xffffdc38`	`[4]` SFP of `main`
`0xffffdc34`	`[4]` RIP of `LeBron`
`0xffffdc30`	`[4]` SFP of `LeBron`
`0xffffdbf0`	`[64]` `buf`
`0xffffdbec`	`[4]` RIP of `Kyrie`
`0xffffdbe8`	`[4]` SFP of `Kyrie`

Q4.4 (3 points) What is the **value** (not the address) of the SFP of `Kyrie`?

◯ `0xffffdbe8`	● `0xffffdc30`	◯ `0xffffdc38`
◯ `0xffffdbec`	◯ `0xffffdc34`	◯ None of the above

Solution: The SFP of `Kyrie` points to the the SFP of the function that called it, hence the value of the SFP of `Kyrie` will be the address of the SFP of `LeBron`, hence it is `0xffffdc30`

Q4.5 (3 points) What should Line 2 be in order to create a vulnerability that allows the attacker to execute `SHELLCODE`?

Hint: Your reference sheet contains definitions for these format string specifiers.

◯ `printf("%d")`	◯ `printf("%c")`	◯ `printf("%hn")`
◯ `printf("%s")`	◯ `printf("%n")`	● `printf("%hhn")`

Solution: The correct line is:

```
printf("%hhn");
```

This creates a format-string vulnerability. Since no argument is actually supplied for `%hhn`, `printf` will still try to read one from the stack and interpret it as a pointer. Then it writes one byte, namely the number of characters printed so far, to that address.

Q4.6 (2 points) What type of exploit can be constructed as a result of your answer to the previous subpart?

◯ Buffer Overflow	● Off-by-one
◯ Signed/unsigned vulnerability	◯ None of the above

Solution: The exploit is an **off-by-one** style saved-frame-pointer exploit.

Here `%hhn` writes exactly one byte. In this setup, that one-byte is `\x00` the least significant byte of `LeBron`'s saved frame pointer, which is enough to redirect the SFP into the attacker-controlled `buf`.

Q4.7 (6 points) Provide an input to `fread` on Line 6 that would cause the program to execute `SHELLCODE`, If a part of the input can be any non-zero value, use `'A' * n` to represent `n` bytes of garbage.

Input to `fread` on Line 6:

```
'A'*20 + '\x08\xdc\xff\xff' + SHELLCODE +  'A' * 8
```

Solution: Using `printf("%hhn")`, `printf` consumes the next word on the stack as a pointer. In this calling setup, that word is the value of `Kyrie`'s SFP, namely `0xffffdc30`, which is the address of `LeBron`'s SFP.

Because `%hhn` writes the number of characters printed so far, and `"%hhn"` prints zero characters before performing the write, `printf` writes `\x00` to address `0xffffdc30`. This changes the least significant byte of `LeBron`'s saved frame pointer from `0x38` to `0x00`, so its saved frame pointer becomes `0xffffdc00`.

When `LeBron` later returns, `main` inherits this corrupted frame pointer. Then when `main` executes its epilogue, it treats `0xffffdc00` as its stack frame. So we want the following layout starting at `0xffffdc00`:

- at `0xffffdc00`: fake SFP, any 4 bytes
- at `0xffffdc04`: fake RIP = address of `SHELLCODE`
- at `0xffffdc08`: `SHELLCODE`

Since `buf` starts at `0xffffdbf0`, we need:
- 16 bytes to reach `0xffffdc00`
- 4 more bytes for the fake SFP
- then the 4-byte address `0xffffdc08`
- then the 32-byte `SHELLCODE`
- then 8 bytes of padding to fill the rest of the 64-byte buffer

So a correct input is:

```
'A' * 20 + '\x08\xdc\xff\xff' + SHELLCODE + 'A' * 8
```

Q4.8 (2 points) Would the correct exploit (without modification) fail if Non-Executable Pages was enabled?

- ● Yes, because the CPU will refuse to execute instructions located on the stack, even if the RIP successfully points to them.
- ○ Yes, because Non-Executable Pages randomizes the stack addresses, making it impossible to determine the location of the shellcode.
- ○ No, because Non-Executable Pages only prevents the `RIP` from being overwritten, but does not stop the execution of shellcode.
- ○ No, because the `fread` function bypasses Non-Executable permissions when writing the shellcode into memory.

Solution: Yes. With Non-Executable Pages enabled, the stack cannot be executed as code. Even if the exploit successfully overwrites the control flow so that `RIP` points at `SHELLCODE` inside `buf`, the CPU will refuse to execute instructions from that stack page.

Q4.9 (3 points) Which **addresses** of the SFP of `LeBron` would prevent the type of exploit from Q4.7? Assume that the stack is shifted up/down accordingly and remains in-order. Select all that apply.

- ■ `0xffffdc50`
- ☐ `0xffffdc20`
- ■ `0xffffdbf0`
- ☐ `0xffffdc40`
- ☐ `0xffffdc10`
- ■ `0xffffdbe0`
- ☐ `0xffffdc30`
- ■ `0xffffdc00`
- ○ None of the above

Solution: The exploit works only if zeroing the least significant byte of `LeBron`'s **saved** frame pointer causes the new frame pointer used by `main` to land **inside** attacker-controlled `buf`.

If the address of the SFP of `LeBron` is `A`, then:
- `buf` occupies `[A - 64, A - 1]`
- the saved frame pointer stored at `A` is `A + 8`
- after the `%hhn` write, that saved frame pointer becomes `(A + 8)` with its low byte zeroed

We therefore need that zeroed value to still lie inside `buf`. Checking each option:

- `0xffffdc50` → SFP becomes `0xffffdc00`, but `buf` is `0xffffdc10` to `0xffffdc4f`, so this fails
- `0xffffdc40` → SFP becomes `0xffffdc00`, and `buf` is `0xffffdc00` to `0xffffdc3f`, so this still works
- `0xffffdc30` → SFP becomes `0xffffdc00`, and `buf` is `0xffffdbf0` to `0xffffdc2f`, so this still works
- `0xffffdc20` → SFP becomes `0xffffdc00`, and `buf` is `0xffffdbe0` to `0xffffdc1f`, so this still works
- `0xffffdc10` → SFP becomes `0xffffdc00`, and `buf` is `0xffffdbd0` to `0xffffdc0f`, so this still works
- `0xffffdc00` → SFP becomes `0xffffdc00`, but `buf` is `0xffffdbc0` to `0xffffdbff`, so this fails
- `0xffffdbf0` → SFP becomes `0xffffdb00`, but `buf` is `0xffffdbb0` to `0xffffdbef`, so this fails
- `0xffffdbe0` → SFP becomes `0xffffdb00`, but `buf` is `0xffffdba0` to `0xffffdbdf`, so this fails

So the correct choices are:

- `0xffffdc50`
- `0xffffdc00`
- `0xffffdbf0`
- `0xffffdbe0`

Q5 *Cryptography: "Sure, Whatever"* 🤷 **(9 points)**

Q5.1 (1 point) Alice sends (m, t) where $t = \text{MAC}(k, m)$. An attacker records one valid pair (m, t). Which attack on the protocol above is still possible even if the MAC is EUF-CMA secure?

- ● Replay the exact same (m, t) later and have Bob accept again.
- ○ Modify m into m' and compute a valid t' for m'.
- ○ Recover the MAC key k from (m, t).
- ○ Distinguish $\text{MAC}(k, \cdot)$ from a random function using one query.

Solution: EUF-CMA security says an attacker shouldn't be able to create a valid tag for a new message they didn't already get tagged. But replaying is not "creating" anything: the attacker can just send the exact same (m, t) again. If the receiver doesn't track freshness (nonces, sequence numbers, or state), the verification algorithm will accept the replayed pair.

Q5.2 (1 point) Why does a perfectly secret one-time pad still fail to provide integrity?

- ● Because an attacker can flip bits in the ciphertext and predictably flip bits in the plaintext.
- ○ Because OTP ciphertexts are distinguishable from random.
- ○ Because OTP requires a PRG to be secure.
- ○ Because OTP decryption is non-deterministic.

Solution: One-time pad encryption is malleable: ciphertext is essentially $c = m \oplus r$. An attacker can flip any chosen set of bits in c to obtain $c' = c \oplus \Delta$, and the decrypted plaintext becomes $m' = m \oplus \Delta$ in a completely predictable way. So even though the attacker can't learn m, they can still **change** it in controlled ways, which is exactly a failure of integrity.

Q5.3 (1 point) What is the core risk in MAC-then-encrypt constructions that has enabled real attacks?

- ○ The MAC key is sent in the clear.
- ○ The MAC is computed on ciphertext rather than plaintext.
- ● The receiver decrypts attacker-controlled ciphertexts before checking integrity, leaking information via behavior/timing.
- ○ The scheme cannot be CPA secure.

Solution: In MAC-then-encrypt, the receiver typically has to decrypt first in order to even know what message and tag to verify. That means the system is doing computation on attacker-controlled ciphertexts before it has established integrity, and any differences in error messages, timing, or behavior can leak information about the plaintext or about whether parts of the ciphertext are "well-formed." This "decrypt first, then reject" pattern is exactly what padding-oracle and timing-style attacks exploit.

Q5.4 (1 point) In CPA security for public-key encryption, the adversary:

- ○ Must not see the public key.
- ● Chooses two equal-length messages and receives an encryption of one of them under the public key.
- ○ Gets a decryption oracle for arbitrary ciphertexts.
- ○ Must guess the secret key.

Solution: In the IND-CPA game for public-key encryption, the attacker is given the public key, chooses two messages of the same length, and receives an encryption of one of them under that public key. The attacker's job is to guess which message was encrypted with non-negligible advantage over $\frac{1}{2}$. A decryption oracle is not part of CPA (that would be a CCA-style game), and the attacker never needs to guess the secret key directly.

Q5.5 (1 point) In "hashed ElGamal" (hybrid encryption), what is the ciphertext form for encrypting a message M to public key $\mathrm{PK} = g^a$?

- ○ $C = \left(g^{\mathrm{ab}}, \mathrm{AE.Enc}\left(H\left(g^b\right), M\right)\right)$.
- ○ $C = \left(\mathrm{AE.Enc}\left(H\left(g^{\mathrm{ab}}\right), M\right)\right)$ with no group element sent.
- ○ $C = (g^a, \mathrm{AE.Enc}(H(g^a), M))$.
- ● $C = \left(g^b, \mathrm{AE.Enc}\left(H\left(\left(g^a\right)^b\right), M\right)\right)$ for fresh random b.

Solution: Hashed ElGamal works by choosing a fresh random exponent b, sending the ephemeral group element g^b, and deriving a symmetric key from the Diffie–Hellman shared secret $\left(g^a\right)^b = g^{\{\mathrm{ab}\}}$. The payload is then encrypted with a symmetric (typically authenticated) scheme under that derived key. So the ciphertext must include g^b plus an encryption under a key computed from $H\left(g^{\{\mathrm{ab}\}}\right)$, which is exactly the stated form.

For Q5.6–Q5.7, consider a "hash-and-sign" signature scheme where:

- The signer computes $\sigma := \mathrm{Invert}(\mathrm{SK}, H(m))$.
- The verifier checks $F(\mathrm{PK}, \sigma) = H(m)$, where F is a trapdoor permutation.

Assume H maps messages into the domain X of the trapdoor permutation.

Q5.6 (1 point) Why do many security proofs for this scheme model H as a random oracle rather than assuming only that H is collision resistant?

- ◯ Collision resistance already implies pseudorandomness, so random oracle is redundant.
- ● The proof needs to "program" H on adversary queries in a way not captured by collision resistance/one-wayness alone.
- ◯ One-wayness of H is too strong and must be weakened to a random oracle assumption.
- ◯ Random oracle is required only to argue correctness, not security.

Solution: Modeling H as a random oracle is used because typical security proofs for hash-and-sign need stronger structure than "no collisions" or "hard to invert." In reductions, the simulator often has to **control** the outputs of H on certain messages (for example, to embed a hard problem instance or to answer signing queries consistently). Plain collision resistance/one-wayness does not justify this kind of programmable behavior, while the random-oracle model does.

Q5.7 (1 point) Which fact best explains why an honestly generated signature on m will always be accepted by the verifier?

- ◯ H is collision resistant.
- ◯ H is one-way.
- ● $\mathrm{Invert}(\mathrm{SK}, \cdot)$ is the inverse of $F(\mathrm{PK}, \cdot)$ on the domain.
- ◯ The scheme is EUF-CMA secure.

Solution: Verification checks whether applying $F(\mathrm{PK}, \cdot)$ to the signature produces the hash of the message. When the signer computes $\sigma := \mathrm{Invert}(\mathrm{SK}, H(m))$, the signature is literally constructed as a preimage under $F(\mathrm{PK}, \cdot)$. If $\mathrm{Invert}(\mathrm{SK}, \cdot)$ is truly the inverse of $F(\mathrm{PK}, \cdot)$ on the domain, then $F(\mathrm{PK}, \sigma) = H(m)$ holds automatically for honest signing, so the verifier accepts.

Q5.8 (1 point) Which requirement is **not** part of the definition of a trapdoor one-way permutation?

- ◯ KeyGen outputs $(\mathrm{SK}, \mathrm{PK})$.
- ◯ $F(\mathrm{PK}, \cdot)$ is efficiently computable and is a permutation over a domain X.
- ◯ $\mathrm{Invert}(\mathrm{SK}, \cdot)$ efficiently computes the inverse of $F(\mathrm{PK}, \cdot)$.
- ● $F(\mathrm{PK}, \cdot)$ is collision resistant over X.

Solution: A trapdoor one-way permutation requires that the public function is efficiently computable and bijective over its domain, and that the holder of the trapdoor can invert it efficiently. Collision resistance is not an additional requirement, because permutations on a fixed domain already have no collisions by definition. Security is about one-wayness without the trapdoor not about collision resistance.

Q5.9 (1 point) Why is using a non-cryptographic PRNG to generate cryptographic keys dangerous?

- ○ Non-cryptographic PRNGs are often too inefficient for repeated cryptographic use.
- ● Non-cryptographic PRNGs often leak enough data that observing enough outputs lets an attacker predict future outputs/state.
- ○ Non-cryptographic PRNGs often fail enough statistical checks that attackers can directly recover hidden keys.
- ○ Non-cryptographic PRNGs often produce only short outputs at a time.

Solution: Many non-cryptographic PRNGs are designed for simulation, not adversarial settings. Their internal state is often small and update rules are predictable; after observing enough outputs, an attacker can frequently reconstruct the state and predict all future outputs. If you use such outputs as "keys" or nonces, the attacker can guess them, which breaks the security of the cryptosystem even if the outputs look random in basic tests.

Q6 *Cryptography: EvanCRHF* 🤖 **(24 points)**

EvanBot is designing a hash function $\mathsf{H} : \{0,1\}^{2n} \to \{0,1\}^{2n}$.

Notation for the entire question:
- 0^n is an n-bit bitstring of all zeros.
- $f : \{0,1\}^n \to \{0,1\}^n$ is a one-way, collision-resistant hash function (CRHF).

For each subpart with boxes, select whether the function is a CRHF.
- If "Yes," leave the boxes blank.
- If "No," give a collision pair $(x_1, x_2), (y_1, y_2)$ such that $\mathsf{H}(x_1, x_2) = \mathsf{H}(y_1, y_2)$ and $(x_1, x_2) \neq (y_1, y_2)$.

Example: If $\mathsf{H}(a, b) = a + b$, then a valid solution is $(x_1, x_2) = (2, 3),\ (y_1, y_2) = (4, 1)$.

Assumptions:
- x_1, x_2, y_1, y_2 are exactly n bits each, but you may answer with a simple integer and assume it is converted to the associated bitstring.
- There may be multiple correct answers. In the example above, $(x_1, x_2) = (1, 9),\ (y_1, y_2) = (5, 5)$ would also be correct.

Q6.1 (3 points) Is $\mathsf{H}(a, b) = a \| f(b)$ a CRHF?

● Yes (leave the boxes blank) ○ No (fill in the boxes)

x_1 : x_2 :

y_1 : y_2 :

Solution: $\mathsf{H}(a, b) = a \| f(b)$ is a CRHF.

If $\mathsf{H}(x_1, x_2) = \mathsf{H}(y_1, y_2)$ then $x_1 = y_1$ and $f(x_2) = f(y_2)$.

So any collision for H would give a collision for f, contradicting that f is collision-resistant.

Q6.2 (3 points) Does $\mathsf{H}(a, b) = a \| f(b)$ possess one-wayness?

● Yes, because an attacker must still invert $f(b)$ to find the complete pre-image (a, b).

○ Yes, because it is computationally difficult to determine where a ends and $f(b)$ begins.

○ No, because the first part of the input (a) is leaked directly in the output.

○ No, because the output length is strictly greater than the input length.

Solution: $\mathsf{H}(a, b) = a \| f(b)$ possesses one-wayness.

Given an output $a \| f(b)$, the attacker can read a, but must still find a b such that $f(b)$ equals the second half of the output. That requires inverting f, which is assumed to be hard.

Q6.3 (3 points) Is $\mathsf{H}(a,b) = 0^n \| f(a)$ a CRHF?

○ Yes (leave the boxes blank) ● No (fill in the boxes)

x_1 :	x	x_2 :	1
y_1 :	x	y_2 :	0

Solution: $\mathsf{H}(a,b) = 0^n \| f(a)$ is not a CRHF.

The output ignores b, so different inputs collide.

Example collision:

$x_1 = x$ $x_2 = 1$

$y_1 = x$ $y_2 = 0$

Then

$\mathsf{H}(x_1, x_2) = 0^n \parallel f(x)$ $\mathsf{H}(y_1, y_2) = 0^n \parallel f(x)$

but $(x_1, x_2) \neq (y_1, y_2)$.

Q6.4 (3 points) Does $\mathsf{H}(a,b) = 0^n \| f(a)$ possess one-wayness?

○ Yes, because fixing the first n bits prevents collisions across different inputs, which is enough to make it one-way.

● Yes, because an attacker still needs to invert $f(a)$ to find a valid a for the pre-image.

○ No, because the construction leaks b directly in the output, so inverting is trivial by reading b.

○ No, because the output only covers strings with prefix 0^n, so the function cannot be one-way for all outputs.

Solution: $\mathsf{H}(a,b) = 0^n \| f(a)$ possesses one-wayness.

Given an output $0^n \| f(a)$, an attacker must find some a such that $f(a)$ equals the second half of the output. That requires inverting f, which is assumed to be hard.

For the following **two subparts**, let the hash function from Q6.3 be G, so $G(a,b) = 0^n \| f(a)$.

Q6.5 (3 points) Is $\mathsf{H}(a,b) = G(G(a,b))$ a CRHF?

Note: The output of G is a 2n bit string. By splitting this in half, it is possible to produce a tuple of form (a,b) that can serve as the input to G.

○ Yes (explain your answer below) ● No (explain your answer below)

Why or why not? Use 10 words or fewer (the staff answer is 5 words).

The hash always outputs 0.

Solution: Let $G(a,b) = 0^n \| f(a)$.

Then

$G(G(a,b)) = G(0^n, f(a)) = 0^n \parallel f(0^n)$.

Thus $\mathsf{H}(a,b) = G(G(a,b))$ always outputs the same constant value, independent of the input.

Therefore it is not a CRHF.

Q6.6 (3 points) Does $\mathsf{H}(a,b) = G(G(a,b))$ possess one-wayness?

○ Yes, because a is not used to compute the output $\mathsf{H}(a,b)$.

○ Yes, because b is not used to compute the output $\mathsf{H}(a,b)$.

● No, because the output is a constant for all inputs, making inversion trivial.

○ No, because the output length is smaller than the input length, which always breaks one-wayness.

Solution: $\mathsf{H}(a,b) = G(G(a,b))$ does not possess one-wayness.

Since the output is always the constant $0^n \parallel f(0^n)$, an attacker can trivially produce a valid preimage such as $(0,0)$.

For the **remaining subparts**, EvanBot is attempting to design CRHF $\mathsf{H} : \{0,1\}^{2n} \to \{0,1\}^n$ that outputs n-bit strings.

Q6.7 (3 points) Is $\mathsf{H}(a, b) = a \oplus b \oplus f(a \oplus b)$ a CRHF?

○ Yes (leave the boxes blank) ● No (fill in the boxes)

x_1 :	x	x_2 :	x
y_1 :	y	y_2 :	y

Solution: $\mathsf{H}(a, b) = a \oplus b \oplus f(a \oplus b)$ is not a CRHF.

Let $z = a \oplus b$. Then

$\mathsf{H}(a, b) = z \oplus f(z)$,

so the hash only depends on $a \oplus b$.

Choose two distinct inputs with the same XOR:

$x_1 = x$ $x_2 = x$

$y_1 = y$ $y_2 = y$

Then

$x_1 \oplus x_2 = 0$ $y_1 \oplus y_2 = 0$

so both hash to

$0 \oplus f(0) = f(0)$.

Q6.8 (3 points) Is $\mathsf{H}(a, b) = a^b \bmod p$ a CRHF?

Assume that a and b are cast to integers, and then the resulting integer $a^b \bmod p$ is cast back to an n-bit string.

○ Yes (leave the boxes blank) ● No (fill in the boxes)

x_1 :	1	x_2 :	x
y_1 :	1	y_2 :	y

Solution: $\mathsf{H}(a, b) = a^b \bmod p$ is not a CRHF.

Example collision:

$x_1 = 1$ $x_2 = x$

$y_1 = 1$ $y_2 = y$

with $x \neq y$.

Then

$\mathsf{H}(x_1, x_2) = 1^x \bmod p = 1$ $\mathsf{H}(y_1, y_2) = 1^y \bmod p = 1$,

so the outputs are equal while the inputs differ.

Q7 *Cryptography: Man Your Stations* 🏞️ **(19 points)**

Alice and Bob are trying to design their own key-exchange protocol.

Assume:
- Alice and Bob agree on a large prime p and generator g.
- They have signature key pairs (sk_A, pk_A) and (sk_B, pk_B), and both know each other's public keys.
- $\text{Enc}(K, M)$ is an IND-CCA secure encryption function with a corresponding decryption function.
- $\mathsf{H}(x)$ is a cryptographic hash function, e.g., SHA256.

The Protocol

1. Alice to Bob: Alice chooses a random x and computes $X = g^x \bmod p$. She sends X to Bob.

2. Bob to Alice: Bob chooses a random y and computes: $Y = g^y \bmod p$.
Bob derives the session key: $K = \mathsf{H}(\underline{\quad \text{Q7.1} \quad})$. He signs the transcript: $\sigma_B = \text{Sign}_{sk_B}(X \parallel Y)$.
Bob sends (Y, C_B) where $C_B = \text{Enc}(K, \sigma_B)$.

3. Alice to Bob: Alice computes: $K = \mathsf{H}(\underline{\quad \text{Q7.2} \quad})$. She decrypts C_B and verifies σ_B using pk_B.
If verification passes, Alice accepts K and she signs the transcript $\sigma_A = \text{Sign}_{sk_A}(X \parallel Y)$.
Alice sends $C_A = \text{Enc}(K, \sigma_A)$.

4. Final Step: Bob decrypts C_A and verifies σ_A using pk_A. If verification passes, Bob accepts K.

Q7.1 (2 points) Which of the following should fill in the blank in Step 2?

○ $X \bmod p$	○ $Y^x \bmod p$	○ $gXY \bmod p$
○ $Y \bmod p$	● $X^y \bmod p$	○ $g \bmod p$
○ $XY \bmod p$	○ $g^{x+y} \bmod p$	○ $g^g \bmod p$

Solution: Bob should derive the Diffie–Hellman shared secret using Alice's value $X = g^x \bmod p$ and his secret exponent y, so the correct input is

$X^y \bmod p$.

Since $X = g^x$, this equals $(g^x)^y = g^{xy} \bmod p$.

Q7.2 (2 points) Which of the following should fill in the blank in Step 3?

○ $X \bmod p$	● $Y^x \bmod p$	○ $gXY \bmod p$
○ $Y \bmod p$	○ $X^y \bmod p$	○ $g \bmod p$
○ $XY \bmod p$	○ $g^{x+y} \bmod p$	○ $g^g \bmod p$

Solution: Alice should derive the Diffie–Hellman shared secret using Bob's value $Y = g^y \bmod p$ and her secret exponent x, so the correct input is

$Y^x \bmod p$.

Since $Y = g^y$, this equals $(g^y)^x = g^{xy} \bmod p$.

Q7.3 (2 points) Suppose Alice and Bob know authentic $(\mathrm{pk}_A, \mathrm{pk}_B)$ in advance. Why is the protocol secure against a man-in-the-middle attack by Mallory?

- ○ Because Enc is IND-CPA secure, Mallory cannot read C_A or C_B, so MITM is impossible.
- ● Because each party signs the full transcript $(X \parallel Y)$ under an authentic public key, Mallory cannot alter X or Y (or swap in her own values) without causing signature verification to fail; she can relay messages but cannot make either party accept a key that Mallory knows.
- ○ Because H is collision resistant, Mallory cannot create a collision that would let her impersonate either party.
- ○ Because Diffie–Hellman alone is already secure against MITM, the signatures are unnecessary.

Solution: The protocol is secure against a man-in-the-middle attack because each party signs the full transcript $X \parallel Y$ using an authentic public key.

So Mallory may relay messages, but she cannot replace X or Y with her own values without causing signature verification to fail. Therefore she cannot make Alice or Bob accept a key that Mallory knows.

For Q7.4–Q7.6, Alice and Bob change their encryption protocol so that Bob no longer sends a signature to Bob in step 2. The steps of their modified protocol now look like:

1. Alice to Bob: Alice chooses a random x and computes $X = g^x \bmod p$. She sends X to Bob.

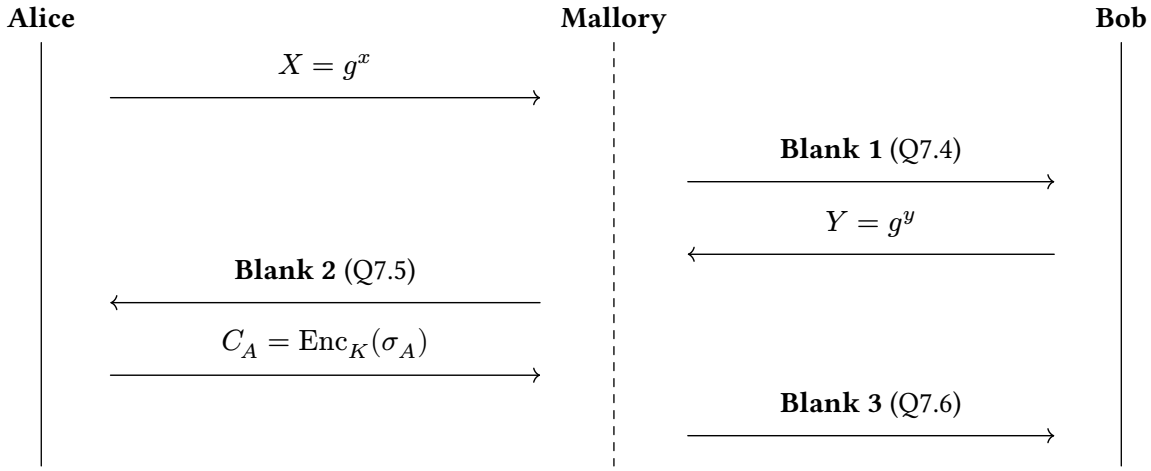
2. Bob to Alice: Bob chooses a random y and computes: $Y = g^y \bmod p$.
Bob derives the session key: $K = \mathsf{H}(\underline{\quad \text{Q7.1} \quad})$. **(No Signature!)**

3. Alice to Bob: Alice computes: $K = \mathsf{H}(\underline{\quad \text{Q7.2} \quad})$. She signs the transcript $\sigma_A = \text{Sign}_{sk_A}(X \parallel Y)$. Alice sends $C_A = \text{Enc}(K, \sigma_A)$.

4. Final Step: Bob decrypts C_A and verifies σ_A using pk_A. If verification passes, Bob accepts K.

For Q7.4–Q7.6, Mallory is a MITM attacker between Alice and Bob, with two goals:
1. Mallory wants to trick Alice into forming a shared key between Alice and Mallory.
2. She also wants to trick Bob into forming a shared key between Bob and Mallory.

Mallory may or may not succeed in both goals, but she should succeed in at least one.



Clarification During Exam: The modified scheme contains a typo. It should read "Bob no longer sends a signature to ~~Bob~~ Alice in Step 2."

Clarification During Exam: The diagram should read $X = g^x \bmod p$ and $Y = g^y \bmod p$.

In the boxes below, you can use m to denote Mallory's secret. If a value can be anything, write "Anything."

Q7.4 (3 points) What should Mallory send to Bob in **Blank 1**?

Anything

Solution: Mallory can't get a shared key with Bob.

Q7.5 (3 points) What should Mallory send to Alice in **Blank 2**?

$Y_M = g^m \bmod p$

Solution: Mallory sends Alice her own DH share: $Y_M = g^m \bmod p$.

Q7.6 (3 points) What should Mallory send to Bob in **Blank 3**?

Anything

Solution: Mallory can't get a shared key with Bob.

Q7.7 (2 points) What shared K will Alice compute?

$K_A = \mathsf{H}(g^{xm})$

Solution: Alice computes the DH secret $(Y_M)^x = (g^m)^x = g^{xm}$, so $K_A = \mathsf{H}(g^{xm})$.

Q7.8 (2 points) After Mallory's attack, does Bob accept K?

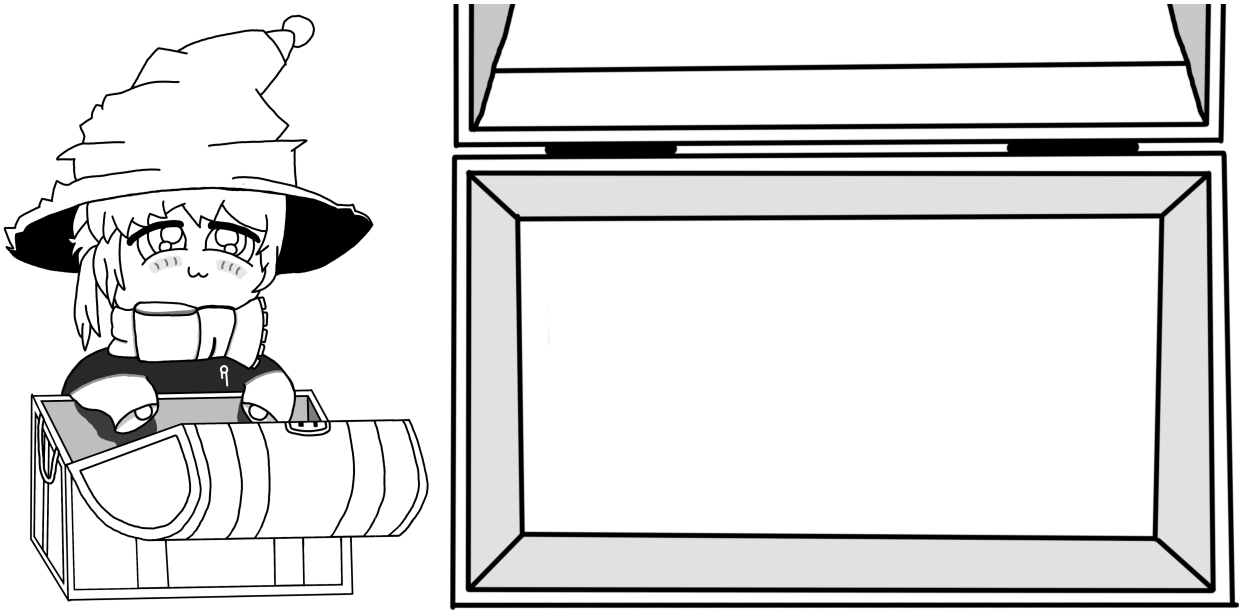
○ Yes ● No

If "Yes", write the value in the box. If "No", leave the box blank.

Solution: Mallory can't forge Alice's signature, so the value that she sends over to Bob does not matter, seeing as he will reject the Key regardless.

Post-Exam Activity: Beyond Bot's End

On a quest for magical grimoires, EvanBot sticks their head inside a chest in the depths of a dungeon. What's inside?



Art by: Soklynin Nou

Comment Box

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