CS 161 Spring 2024 — Introduction to Computer Security — Midterm

Solutions last updated: March 2nd, 2024

Name: _____

Student ID: _____

This exam is 110 minutes long.

Question:	1	2	3	4
Points:	0	16	16	16
Question:	5	6	7	Total
Points:	16	19	17	100

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

- ○ Unselected option (completely unfilled)
- ● Only one selected option (completely filled)
- ◍ Don't do this (it will be graded as incorrect)

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

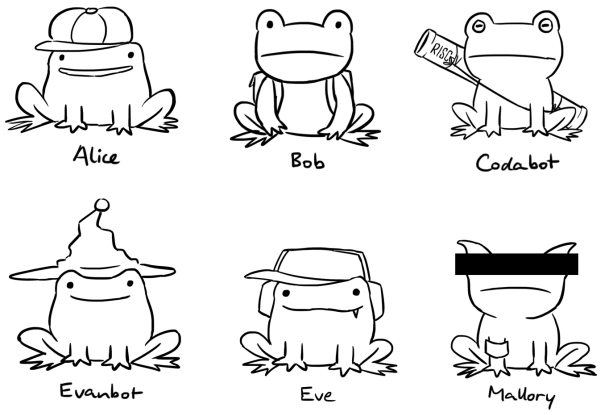
- ■ You can select
- ■ multiple squares (completely filled)

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 will grade the worst interpretation.

Pre-exam activity (0 points):
It's Leap Day, which means it's time for the quadrennial Leap-Frog Race! Everyone entered their frogs into the contest, but unfortunately the results got mixed up. EvanBot needs your help to piece together the standings from eyewitness reports!

1. Alice: "Bob's frog was right behind EvanBot's frog."
2. Bob: "CodaBot's frog was somewhere ahead of mine."
3. CodaBot: "I couldn't tell whose it was, but there was a frog with a hat somewhere ahead of Mallory's frog."
4. Eve: "My frog was always ahead of Alice's frog."
5. Mallory: "My frog definitely finished before CodaBot's frog. I'm telling the truth, I promise."

Circle the winning frog.



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: _____

Q2 *True/False* **(16 points)**

Each true/false is worth 1 point.

Q2.1 The Mallory Security Agency requires 128-bit AES encryption for all data, and "Top Secret" data is encrypted with an additional 256-bit key stored in a separate location.

True or False: This is an example of Defense in Depth.

● (A) True ○ (B) False

Solution: True, the top-secret data has two layers of defenses. The defenses are considered separate because the key is in a different location.

Q2.2 Devices storing cryptographic keys are often required to be fitted with tamper-evident seals.

True or False: This is an example of Detect If You Can't Prevent.

● (A) True ○ (B) False

Solution: True. Tamper-evident seals might not be able to stop all attacks, but they will leave a signal that the seal was broken, which allows us to detect that an attack has happened.

Q2.3 True or False: All strings in C are terminated with the character `'\n'`.

○ (A) True ● (B) False

Solution: False. Strings are terminated with the null terminator, `'\x00'`.

Q2.4 True or False: Enabling non-executable pages prevents the ret2libc attack.

○ (A) True ● (B) False

Solution: False. The ret2libc attack works by executing instructions that already exist in the code segment of memory.

For the next 2 subparts: Suppose we have a little-endian C program with a local variable `char buf[8]`. Consider the following possible GDB output after running the command `x/4wx buf`:

```
0xffffff58: 0x61657270 0x0000006d 0xc0dab0bb 0xbaa15691
```

Q2.5 True or False: The words `0x61657270` and `0x0000006d` correspond to the data in `buf`.

● (A) True ○ (B) False

Solution: True. These are the first 8 bytes in the memory dump (which starts at `buf`).

Q2.6 True or False: `buf[9]` is `0xda`.

○ (A) True ● (B) False

Solution: False, `buf[9]` would be 0xb0.

Q2.7 True or False: Every nondeterministic (randomized) encryption scheme is IND-CPA secure.

○ (A) True ● (B) False

Solution: False. It's possible for a nondeterministic scheme to still leak information about the message.

As an example, consider $\mathsf{Enc}(K, M) = M \| R$, where R is a block of random bits. This trivially leaks the entire contents of the message, even though the scheme is nondeterministic.

Q2.8 True or False: If Alice encrypts a message to Bob using ElGamal, an eavesdropper who can solve the discrete log problem would be able to correctly decrypt the message.

● (A) True ○ (B) False

Solution: True. ElGamal relies on the same assumptions as Diffie-Hellman, which includes the hardness of the discrete log problem.

Q2.9 True or False: Digital signature schemes often encrypt the message before signing to prevent existential forgery attacks.

○ (A) True ● (B) False

Solution: False, Digital signatures hash, not encrypt, their messages.

Q2.10 TRUE or FALSE: AES-CTR does not require the message to be padded.

● (A) TRUE ○ (B) FALSE

Solution: True. We can throw away the unused bits of the block cipher output before computing a bitwise XOR of the block cipher output and the plaintext.

Q2.11 TRUE or FALSE: HMAC is equivalent to NMAC with $K_1 = K_2$.

○ (A) TRUE ● (B) FALSE

Solution: False. The key is XOR'd with different pads in the inner/outer hash functions.

Q2.12 TRUE or FALSE: Using the output of a PRNG as the key for a one-time pad provides perfect security, even against an attacker with infinite computational power.

○ (A) TRUE ● (B) FALSE

Solution: False. An attacker with infinite computational power could try all the possible initial seeds for the PRNG, generate PRNG output for each possible seed, and use each possible seed to try and decrypt the ciphertext.

Q2.13 TRUE or FALSE: Length-extension attacks are often used to break the collision resistance of a hash function.

○ (A) TRUE ● (B) FALSE

Solution: False, the length-extension attack is concerned with creating $H(K\|M\|x)$ given $H(K\|M)$, but the two hashes aren't the same.

Q2.14 TRUE or FALSE: The security of Diffie-Hellman relies on the modulus being difficult to factor.

○ (A) TRUE ● (B) FALSE

Solution: False. The modulus in Diffie-Hellman is prime, so there's no point to factoring it. The difficulty of factoring a product of two large primes is used in RSA, not Diffie-Hellman.

Q2.15 TRUE or FALSE: The security of ElGamal relies on the difficulty of finding $g^{ab} \bmod p$ given $g^a, g^b \bmod p$.

● (A) TRUE ○ (B) FALSE

Solution: True, this is the Diffie-Hellman problem. The security of ElGamal relies on the Diffie-Hellman problem.

Q2.16 TRUE or FALSE: Certificates are generally not sent over insecure channels due to the risk of a replay attack.

○ (A) TRUE ● (B) FALSE

Solution: False, certificates are public values and anyone can copy them if they want.

Q3 *'Tis But a Scratch* (16 points)

King Arthur and his knights are searching for the Holy Grail. They find a castle protected by the following vulnerable C code:

```
1  void castle() {
2      int8_t holy_hand_grenade;
3      char[16] holy_grail;
4      char[128] cave;
5
6      // Implementation not shown.
7      find_grail(holy_grail);
8
9      memset(cave, 0, 128);
10
11     fread(&holy_hand_grenade, 1, 1, stdin);
12
13     if (holy_hand_grenade >= 128) {
14         return;
15     }
16
17     fread(cave, holy_hand_grenade, 1, stdin);
18     printf("%s", cave);
19 }
```

Stack at Line 8

RIP of `castle`
(1)
(2)
(3)
`cave`

The `find_grail` function writes a secret 16-byte string to `holy_grail`, and it's your job to make the program output the `holy_grail` string! You can assume `find_grail` does not modify the stack in any other way.

Q3.1 (1 point) What values go in blanks (1) through (3) in the stack diagram above?

- ● (A) (1) SFP of `castle` (2) `holy_hand_grenade` (3) `holy_grail`
- ○ (B) (1) SFP of `castle` (2) `holy_grail` (3) `cave`
- ○ (C) (1) RIP of `castle` (2) `holy_grail` (3) `holy_hand_grenade`
- ○ (D) (1) `&holy_grail` (2) RIP of `find_grail` (3) SFP of `find_grail`

Q3.2 (1 point) Which vulnerability is present in the code?

- ○ (A) Off-by-one
- ○ (B) Format string vulnerability
- ● (C) Signed/unsigned vulnerability
- ○ (D) Heap overflow

Solution: `holy_hand_grenade` is a signed type `int8_t`, but the `fread` function takes in an unsigned type `size_t`.

In the next two subparts, provide inputs that would cause the program to output the `holy_grail` string (possibly as part of a larger output).

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

Q3.3 (4 points) Input to `fread` at Line 11:

Solution: `\xff`

Other values would also work, as long as the byte, when read as an unsigned number, is at least 128.

Q3.4 (4 points) Input to `fread` at Line 17:

Solution: `'A' * 128`

This overwrites all the null bytes in `cave`, so that the `printf` at Line 18 prints past the end of `cave` and prints out `holy_grail`.

Alternate solution: `'A'*n` where $n > 128$. This only works if the answer to Q3.3 is exactly `\x80` (which is exactly 128 in two's complement signed representation), because this would cause the `fread` call to stop after 128 characters are written to memory.

If the answer to Q3.3 is not `\x80`, then this alternate solution fails, because it will overwrite `holy_grail`.

The next four subparts are independent from each other.

Q3.5 (1 point) Would it still be possible for your exploit to leak `holy_grail` if `fread(&holy_hand_grenade, 1, 1, stdin)` on Line 11 is replaced with `gets(&holy_hand_grenade)`?

- ○ (A) Yes, because `gets` would allow the attacker to overwrite the address of `cave`.
- ● (B) Yes, because `holy_hand_grenade` is above `holy_grail` in the stack diagram.
- ○ (C) No, the null terminator added by `gets` will cause the final `printf` to terminate before reaching `holy_grail`.
- ○ (D) No, because the null terminator added by `gets` will partially overwrite the SFP of `castle`.

Solution: Yes, the exploit is still possible. `gets` can still overwrite the single byte of `holy_hand_grenade`.

While it is true that `gets` appends a null terminator, this won't affect our exploit. If we assume there is no padding for address alignment, then the null byte is written into the SFP of `castle`, which is not used in our exploit.

If we assume there is padding, then the null byte is written into the padding, where the null byte doesn't affect program execution.

Choice (A) is incorrect because `gets` writes from lower to higher memory addresses, and would not allow the attacker to overwrite the address of `cave`.

Q3.6 (1 point) Would it still be possible for your exploit to leak `holy_grail` with stack canaries enabled?

- ○ (A) Yes, because the exploit writes around the canary to overwrite values above the canary.
- ● (B) Yes, because the exploit never tries overwriting values above the canary.
- ○ (C) No, because all 4 bytes of the canary are overwritten by garbage.
- ○ (D) No, because the least-significant byte of the canary is overwritten by a null byte.

Solution: Yes, the exploit is still possible, because we never overwrite above the local variables, and the stack canary is located above the local variables.

This is somewhat moot because the printf would run before the function returns anyway (so `holy_grail`'s value would be leaked before the program crashes on a failed canary check).

Choice (A) is incorrect because the exploit never attempts to write to any values above the canary. We only overwrite the local variables.

Q3.7 (1 point) Would it still be possible for your exploit to leak `holy_grail` with non-executable pages enabled?

- ● (A) Yes, because the exploit never writes any executable instructions on the stack.
- ○ (B) Yes, because the malicious instructions being executed are not on the stack.
- ○ (C) No, because the exploit writes executable instructions on the stack.
- ○ (D) No, because non-executable pages stops the exploit from writing anything on the stack.

Solution: Yes, the exploit is still possible, because the exploit does not write any executable instructions on the stack.

Choice (B) is incorrect because there are no malicious instructions being executed. All of the instructions being executed are instructions that were already in the code section of memory (the malicious part of the exploit is just causing `printf("%s", cave)` to not find a null terminator and print out-of-bounds).

Q3.8 (1 point) Would it still be possible for your exploit to leak `holy_grail` with ASLR enabled?

● (A) Yes, because the exploit does not require knowing any absolute addresses.

○ (B) Yes, because the `printf` call will always leak an address on the stack.

○ (C) No, because we need to know the address of `holy_grail` in order to leak its value.

○ (D) No, because we need to know the address of `castle` in order to complete the exploit.

Solution: Yes, the exploit is still possible, because the exploit doesn't require using any absolue addresses.

Choice (B) is false because `printf` is not guaranteed to leak addresses on the stack. For example, `printf("%s", cave)` may only print out the contents of `cave`, which is not guaranteed to contain a memory address.

For the following subparts, your goal is now to execute a 72-byte shellcode, rather than leaking `holy_grail`.

Q3.9 (1 point) Assuming no memory safety defenses are enabled, would it be possible to exploit this program to execute shellcode?

● (A) Yes ○ (B) No

Solution: By inputting `\xff` on Line 11, we can now write up to 255 bytes into memory on Line 17, which is more than enough to write the 72-byte shellcode and also overwrite the RIP to point to shellcode.

Q3.10 (1 point) Assuming canaries are enabled, would it still be possible to exploit this program to execute shellcode?

○ (A) Yes, because we can leak the canary and overwrite the canary with itself.

○ (B) Yes, because the `printf` call can write directly to the RIP, skipping the canary.

● (C) No, because we run the final `fread` before we can leak the canary.

○ (D) No, because it is not possible to run shellcode at all in this program.

Solution: No, this exploit is no longer possible.

The naive exploit from the previous subpart doesn't work because `fread` writes contiguously from lower to higher memory addresses, which would cause the canary to be overwritten before the RIP gets overwritten.

You could try and use the `printf` call to somehow leak the canary value (and write it back later), but this won't work either, because the `fread` call comes before the `printf` call. By the time you need to supply an input to `fread`, the `printf` call has not executed yet.

Choice (D) is incorrect because without stack canaries, it is possible to run shellcode in this program.

Q4 *I Sawed This Shellcode In Half!* **(16 points)**

Consider the following vulnerable C code:

```
1  void boat(void* shellcode_first_half, void* shellcode_second_half) {
2      // fp contains the address of the fgets function
3      uintptr_t fp = (uintptr_t) fgets;
4
5      char[32] buf;
6      char* buf_ptr = &buf;
7
8      fgets(buf, 32, stdin);
9      printf(buf);
10
11     fgets(buf, 32, stdin);
12     printf(buf);
13 }
```

This is the result of running `disas fgets` in GDB:

```
1 0x08076030: push %ebp
2 0x08076034: mov %esp, %ebp
3 0x08076038: sub %ebp, 20
4 ...
5 0x08076050: mov %ebp, %esp
6 0x08076054: pop %ebp
7 0x08076058: ret
```

`shellcode_first_half` is a pointer to the first half of shellcode and `shellcode_second_half` is a pointer to the second half of shellcode. Both halves of shellcode have a `ret` instruction at the end.

Assumptions:

- ASLR is enabled, but all other memory safety defenses are disabled.
- The program can print an arbitrarily large number of bytes using `printf`.
- All addresses are at least `0x01000000`.
- `esp` is not modified by the shellcode.

Q4.1 (2 points) Which of these inputs to the `fgets` on Line 8 will always leak the values of all the local variables in the `boat` stack frame? Select all that apply.

- [x] (A) `"%x" * 10`
- [ ] (B) `"%s" * 10`
- [ ] (C) `"%n" * 10`
- [ ] (D) `"%c" * 10`
- [x] (E) `"%d" * 10`
- [ ] (F) None of the above

Solution: `%x` and `%d` both read values directly off the stack (as hexadecimal or decimal numbers) and prints them out. Repeating many of these formatters will cause subsequent values on the stack to each get printed out.

`%c` will not work because each `%c` reads 4 bytes off the stack but only prints out 1 byte. This will not allow us to learn the entire value of all the local variables.

`%n` will not work because it produces no output (it writes to memory instead).

Note that `%s` does not work, because even though the first `%s` matches up to `buf_ptr = &buf`, the `fgets` will add a null byte to `buf`, preventing the program from printing any memory past the end of `buf`.

Then, subsequent `%s` formatters will get matched up to values inside `buf`, and there's no guarantee that any of the values inside `buf` are addresses on the stack (so that `%s` would dereference them and print out values on the stack). Note that the question asks for inputs that will *always* leak the values of all local variables.

Now that we have the values of all the local variables in the `boat` stack frame, we need to compute two values to use later in our exploit.

You can assume that the values outputted from the `printf` on Line 9 have all been converted to integers that you can perform arithmetic on.

Q4.2 (2 points) Which of these values do we need to use later in our exploit?

`&buf` is the address of `buf`, which you leaked in Q4.1.

Hint: What is the address in memory we want to write to?

- ○ (A) `&buf - 4`
- ○ (B) `&buf`
- ○ (C) `&buf + 32`
- ○ (D) `&buf + 36`
- ● (E) `&buf + 40`
- ○ (F) `&buf + 44`

Solution: This is the address of the RIP.

```
[&buf + 48] shellcode_first_half
[&buf + 44] shellcode_first_half
[&buf + 40] RIP of boat
[&buf + 36] SFP of boat
[&buf + 32] fgets
[&buf + 0] buf
[&buf - 4] buf_ptr
```

Q4.3 (2 points) Which of these values do we need to use later in our exploit?

`fp` is the address of the `fgets` function, which you leaked in Q4.1.

Hint: What is the value we want to write into memory?

- ○ (A) `fp`
- ○ (B) `fp + 0x4`
- ○ (C) `fp + 0x8`
- ○ (D) `fp + 0x20`
- ○ (E) `fp + 0x24`
- ● (F) `fp + 0x28`

Solution: This is the address of the `ret` instruction, which is the value we'll be overwriting the RIP with.

Let `x` be the value you computed in Q4.2, and `y` be the value you computed in Q4.3.

You can perform arithmetic on these values (e.g. `x + 5`), and you can assume that the resulting value is converted to the proper format (e.g. raw bytes, or decimal representation), so you don't have to manually do any conversions.

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

Q4.4 (8 points) Provide an input to the `fgets` call on Line 11 that will execute shellcode.

[ ] + [ ] + `'%c' + '%` [ ] `u' + '%n'`

Solution: `'A'*4 + x + '%c' + '% (y-9) u' + '%n'`

First box: `'A'*4`, which will be consumed by the `%u`.

Second box: `x`. This is the address of the RIP, which is consumed by the `%n`.

Third box: `(y-9)`. The value we want to write into memory is `y` (the address of `ret`), but we've already printed 9 bytes so far, so we need to print out `(y-9)` more bytes so that the total number of bytes printed so far is `y`.

```
[arg3] buf[4:8] (matches to the %n)
[arg2] buf[0:4] (matches to the %u)
[arg1] buf_ptr (matches to the %c)
[arg0] &buf
RIP printf
printf stack frame
```

This exploit causes the address of `ret` to be written to the RIP of `boat`. When `boat` returns, we will execute a `ret` instruction, which causes us to take the next value on the stack, `shellcode_first_half`, and start executing the instructions at that address.

Q4.5 (1 point) If canaries are enabled, would it still be possible to exploit this program to execute shellcode?

● (A) Yes, because the exploit writes directly to the RIP instead of smashing through the canary.

○ (B) Yes, because overwriting the RIP directly disables the canary check altogether.

○ (C) No, because all 4 bytes of the canary are overwritten by garbage.

○ (D) No, because the `ret` instruction at the end of the first shellcode will fail the canary check.

Solution: The format string vulnerability allows us to write only to the RIP of `boat`, without having to contiguously write from lower to higher addresses.

Choice (B) is incorrect because overwriting the RIP does not disable the canary check (which is some sequence of instructions in the code section of memory).

Q4.6 (1 point) What happens to the exploit if `boat` is called with the arguments reversed?

In other words, instead of `boat(shellcode_first_half, shellcode_second_half)`, what if we called `boat(shellcode_second_half, shellcode_first_half)`?

○ (A) The instructions of shellcode would all execute in reverse order.

○ (B) The shellcode would execute unchanged.

● (C) The second half of shellcode executes, followed by the first half of shellcode.

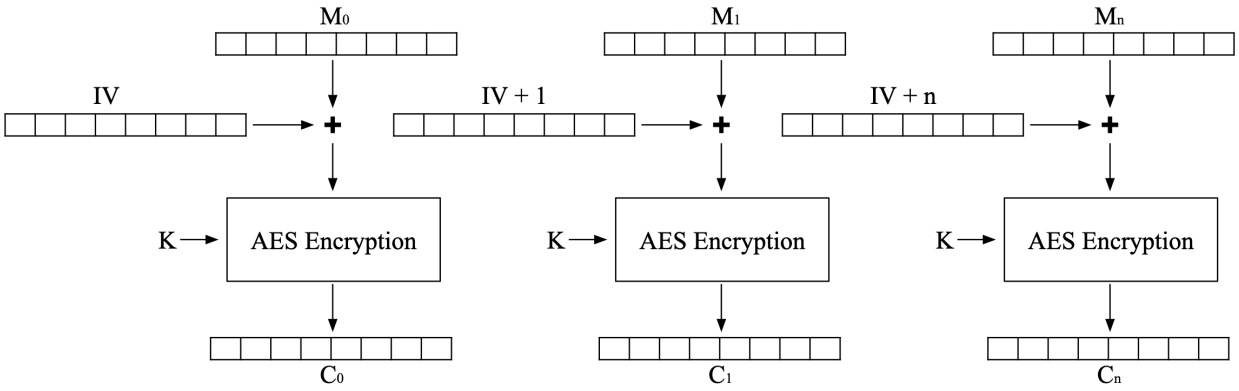
○ (D) The program would crash after returning from `boat`, but before executing shellcode.

Solution: Now, when the `ret` instruction executes, the next value on the stack is `shellcode_second_half`, so we jump to that address and begin executing the second half of shellcode.

Per the assumptions in the question, both halves of shellcode end in a `ret` instruction. At the end of the second half of shellcode, the `ret` instruction will take the next value on the stack, `shellcode_first_half`, and execute instructions at that address, causing the first half of shellcode to execute after the second half finishes executing.

Q5 *Challenging Constructions* **(16 points)**

Consider the following encryption scheme:



The encryption formula for this scheme is

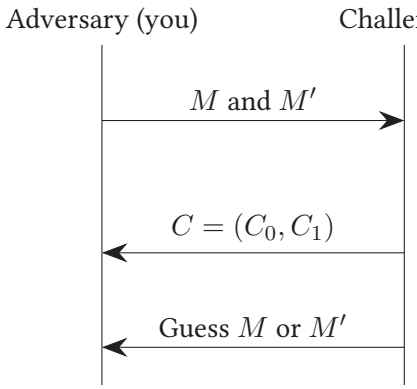
$$C_i = E_K(M_i + IV + i)$$

Q5.1 (1 point) Select the correct decryption formula for the given scheme.

- ○ (A) $M_i = D_K(C_i - IV - i)$
- ● (B) $M_i = D_K(C_i) - IV - i$
- ○ (C) $M_i = D_K(C_i) + IV + i$
- ○ (D) $M_i = D_K(C_i + IV - i)$

Solution: Algebraically, apply D_K to both sides, and then add $-IV - i$ to both sides.

To show this scheme is insecure, you want to provide a strategy that always wins the IND-CPA game.



First, the adversary (that's you!) sends two different challenge messages, $M \neq M'$, to the challenger. For your strategy, you can assume M and M' are each two blocks long.

Then, the challenger randomly encrypts either M or M'. The resulting two-block ciphertext $C = (C_0, C_1)$ is returned to you.

Finally, you guess whether M or M' was encrypted.

In this strategy, the query phase is not needed (i.e. you never have to ask the challenger to encrypt messages of your choosing beforehand).

Assume that the second challenge message $M' = (?, ?)$ is chosen completely at random.

Clarification during exam: Assume that each "?" value is independently randomly generated, so (?, ?) would not necessarily be two identical values.

Q5.2 (2 points) What must be true of M for this strategy to work?

(Note: ? denotes a randomly-chosen value)

- ○ (A) $M_0 = 0$ and $M_1 = ?$
- ○ (B) $M_0 = ?$ and $M_1 = 0$
- ○ (C) $M_0 = ?$ and $M_1 = ?$
- ○ (D) $M_1 = M_0 + 1$
- ● (E) $M_1 = M_0 - 1$
- ○ (F) $M_0 = M_1$

Solution: We require that $M_1 = M_0 - 1$. In other words, within the two-block message M, the second block M_1 has value one less than the first block M_0 (treating both blocks as 128-bit binary numbers).

See next subpart for why this is useful.

Q5.3 (3 points) The challenger encrypts one of M, M' from the previous subparts and returns $C = (C_0, C_1)$.

Explain how you would determine whether M or M' was encrypted. Your answer can use these values:

- $M = (M_0, M_1)$ and $M' = (M'_0, M'_1)$
- $C = (C_0, C_1)$. Recall: C is either the encryption of M or M'.

An example of how you could describe your strategy, that has nothing to do with this question: If $C_0 + 161 = M_0$, guess M. Else, guess M'.

Solution: If $C_0 = C_1$, guess M, otherwise M'.

If the challenger chooses to encrypt M, which has the property that $M_1 = M_0 - 1$, then:

$$\begin{aligned} C_0 &= E_K(M_0 + IV + 0) \\ &= E_K(M_0 + IV) \\ C_1 &= E_K(M_1 + IV + 1) \\ &= E_K(M_0 - 1 + IV + 1) \\ &= E_K(M_0 + IV) \end{aligned}$$

Since E_K is a deterministic function for a fixed K, and IV stays the same within a single encryption, this tells us that when M is encrypted, we will notice that $C_0 = C_1$.

This leads us to the strategy of guessing that M is encrypted if $C_0 = C_1$. (Otherwise, we guess that M' was encrypted.)

Q5.4 (1 point) What is the probability (excluding negligible factors) that an optimal attacker (i.e. using the correct answer to Q5.3) wins the IND-CPA game?

○ (A) 50% ○ (B) 66.7% ○ (C) 75% ● (D) 100%

Solution: The exact calculations (including negligible factors) aren't needed to solve this question, but intuitively speaking:

If M is encrypted, $C_0 = C_1$ will always hold. This means you will always correctly guess that M was encrypted.

If M' is encrypted, the probability that $C_0 = C_1$ is negligible. (See footnote below for more details). This means that you will almost-always correctly guess that M' was encrypted.

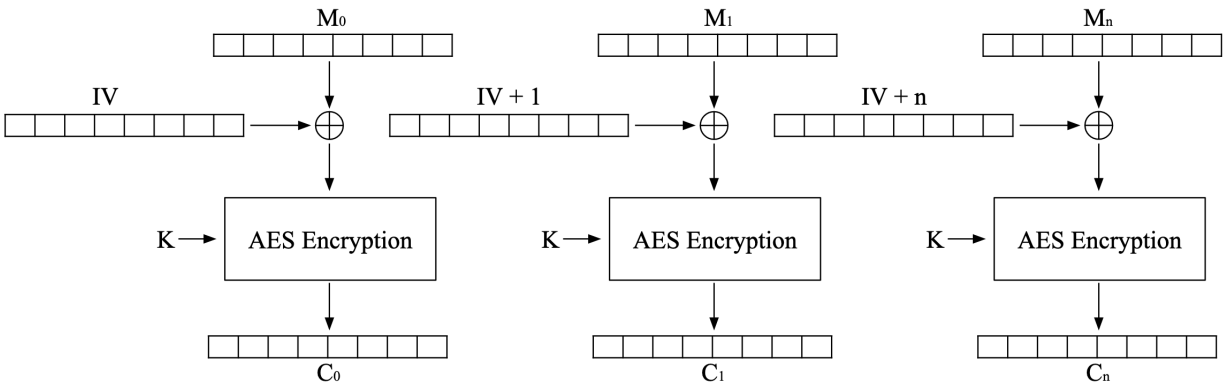
In total, your probability of winning the game is closest to 100% of all the options here.

Footnote: If you encrypt two completely random blocks (M'_0, M'_1), then:

$$C_0 = E_K(M'_0 + IV + 0)$$
$$C_1 = E_K(M'_1 + IV + 1)$$

In order for $C_0 = C_1$ (which would cause us to make an incorrect guess), we would need $M_1 = M_0 - 1$ to hold (the same assumption that allowed us to guess M earlier). The probability that two randomly-chosen blocks differ by just 1 is negligible, so we don't need to consider this in our probability calculations.

Now consider a modified version of the previous encryption scheme:



The encryption formula for this scheme is

$$C_i = E_K(M_i \oplus (IV + i))$$

You want to devise a strategy to win the IND-CPA game.

Assume that the second challenge message $M' = (?, ?)$ is chosen completely at random.

Q5.5 (1 point) What value should you pick for M?

○ (A) $(0, ?)$ ○ (C) $(?, ?)$ ● (E) $(1, 0)$

○ (B) $(?, 0)$ ● (D) $(0, 1)$ ○ (F) $(1, 1)$

Solution: See next subpart for more detailed explanation.

Q5.6 (5 points) The challenger encrypts one of M, M' from the previous subparts and returns $C = (C_0, C_1)$.

Explain how you would determine whether M or M' was encrypted. Your answer can use these values:

- $M = (M_0, M_1)$ and $M' = (M'_0, M'_1)$
- $C = (C_0, C_1)$. Recall: C is either the encryption of M or M'.

Solution: If $C_0 = C_1$, guess M, otherwise M'. (The same strategy as the previous scheme.)

If the challenger chooses to encrypt $M = (1, 0)$, then:

$$\begin{aligned} C_0 &= E_K(M_0 \oplus (IV + 0)) \\ &= E_K(1 \oplus (IV + 0)) \\ &= E_K(1 \oplus IV) \\ C_1 &= E_K(M_1 \oplus (IV + 1)) \\ &= E_K(0 \oplus (IV + 1)) \\ &= E_K(IV + 1) \end{aligned}$$

If the challenger chooses to encrypt $M = (0, 1)$, then:

$$\begin{aligned} C_0 &= E_K(M_0 \oplus (IV + 0)) \\ &= E_K(0 \oplus (IV + 0)) \\ &= E_K(IV) \\ C_1 &= E_K(M_1 \oplus (IV + 1)) \\ &= E_K(1 \oplus (IV + 1)) \\ &= E_K(IV) \end{aligned}$$

Note that the above only happens when IV is even, meaning the least-significant bit of IV is 0, which happens around half the time.

In summary, if M is encrypted, then around half the time we will get a signal of that by noticing that $C = C'$. This gives us a non-negligible advantage of winning the IND-CPA game, as we'll see in the next subpart.

Q5.7 (3 points) What is the probability (excluding negligible factors) that an optimal attacker (i.e. using the correct answer to Q5.6) wins the IND-CPA game?

○ (A) 50% ○ (B) 62.5% ○ (C) 66.7% ● (D) 75% ○ (E) 87.5% ○ (F) 100%

Justify your answer.

Solution: There are two properties that can differ in each given IND-CPA instance:

1. Whether the IV's least-significant bit is zero
2. Whether M or M' is encrypted

Recall that we guess M if $C_0 = C_1$ and M' otherwise.

1. IV LSB is zero, M encrypted. In this case, our attack succeeds ($C_0 = C_1$) and we guess M correctly.
2. IV LSB is one, M encrypted. In this case, $C_0 \neq C_1$, so we guess M' incorrectly.
3. IV LSB is zero, M' encrypted. In this case, $C_0 \neq C_1$ and we guess M' correctly.
4. IV LSB is one, M' encrypted. In this case, $C_0 \neq C_1$ and we guess M' correctly.

Counting the probabilities up gives us 75% chance of success, since each case is equally likely.

Q6 *Authentic Auctions* **(19 points)**

EvanBot wants to hold a charity auction and needs your help! They have designed a secure bidding protocol as follows:

1. Each party chooses a bid x.
2. Each party applies a function `COMMIT` to x and broadcasts $\texttt{COMMIT}(x)$ to every other party.
3. Once everyone has posted their commitment, each party reveals their bid and any other information relevant to the scheme.

There are two key properties we want to guarantee:

1. **Binding**: If a party chooses x, sends $\texttt{COMMIT}(x)$, and then reveals $x' \neq x$, other parties can detect that x' is invalid with overwhelming probability.
2. **Hiding**: $\texttt{COMMIT}(x)$ should not leak more than a negligible amount of information about x.

For each of the following schemes, select whether the scheme provides binding, hiding, both, or neither.

You should assume that the range of all possible bids is small enough to brute-force, each subpart is independent, and the auction is run exactly once. If a reveal step is not specified, assume that the reveal step is to send x.

Hint: EvanBot says you should reread the above assumptions and make sure you understand them before proceeding!

Clarification during exam: Assume that all parties have access to the trusted public keys of every other party.

Q6.1 (1 point) $\texttt{COMMIT}(x) = x$

● (A) Binding ○ (B) Hiding ○ (C) Both ○ (D) Neither

Solution: This is not hiding because we send x in plaintext for other parties to see.

This is binding because an adversary cannot choose x, send x, and later claim to reveal $x' \neq x$. Other parties would be able to detect that the adversary changed their mind after sending the commit.

Q6.2 (1 point) $\texttt{COMMIT}(x) = H(x)$.

● (A) Binding ○ (B) Hiding ○ (C) Both ○ (D) Neither

Solution: This scheme is not hiding because we can bruteforce all possible x against the $H(x)$ value that was sent, in order to determine the value of x (see assumptions).

This scheme is binding due to the collision-resistance of H. It is infeasible for an adversary to find two values $x \neq x'$ that both hash to the same output.

Q6.3 (1 point) $\texttt{COMMIT}(x) = E_K(x)$, where K is a publicly-known symmetric key shared by everyone.

● (A) Binding ○ (B) Hiding ○ (C) Both ○ (D) Neither

Solution: This scheme is not hiding because everyone knows K and can decrypt to learn x.

This scheme is binding precisely because everyone can learn the original value of x by decrypting the COMMIT(x) value, so changing the reveal value will always be detected.

Q6.4 (1 point) $\texttt{COMMIT}(x) = x \oplus K$, where K is a randomly-generated one-time-pad key. To reveal, each party sends x, K.

○ (A) Binding ● (B) Hiding ○ (C) Both ○ (D) Neither

Solution: This scheme is hiding by virtue of OTP. With no knowledge of K, other parties cannot determine the value of x.

This scheme is not binding because we can change our revealed value arbitrarily. Suppose an adversary chose x, published $x \oplus K$, but wanted to reveal a different x'. Then the adversary could reveal x' and $K' = K \oplus x \oplus x'$.

Other parties would take the published commitment $x \oplus K$, decrypt this with the revealed K', and receive $(x \oplus K) \oplus (K \oplus x \oplus x')$, and be convinced that the adversary really chose x' all along.

Q6.5 (1 point) $\texttt{COMMIT}(x) = H(x) \oplus K$, where K is a randomly-generated one-time-pad key. To reveal, each party sends x, K.

○ (A) Binding ● (B) Hiding ○ (C) Both ○ (D) Neither

Solution: This scheme is perfectly hiding by virtue of OTP. With no knowledge of K, other parties cannot determine the value of x.

This scheme is not binding because we can change our revealed value arbitrarily. To reveal y from $H(x) \oplus K$, first evaluate $H(y)$. Then set $K' = H(x) \oplus H(y) \oplus K$ and send y, K'.

The properties, repeated for your convenience:

1. **Binding**: If a party chooses x, sends $\texttt{COMMIT}(x)$, and then reveals $x' \neq x$, other parties can detect that x' is invalid with overwhelming probability.
2. **Hiding**: $\texttt{COMMIT}(x)$ should not leak more than a negligible amount of information about x.

Q6.6 (1 point) $\texttt{COMMIT}(x) = H(x \oplus K)$, where K is a randomly-generated one-time-pad key. To reveal, each party sends x, K.

○ (A) Binding ● (B) Hiding ○ (C) Both ○ (D) Neither

Solution: This scheme is perfectly hiding by virtue of OTP.

This scheme is not binding because we can change our revealed value arbitrarily. To reveal y from $x \oplus K$, send $x' = y \oplus x \oplus K$. The hash function does not affect this, because the final hash function input doesn't change.

Q6.7 (1 point) $\texttt{COMMIT}(x) = \mathsf{HMAC}(K, x)$, where K is a publicly-known symmetric key shared by everyone.

● (A) Binding ○ (B) Hiding ○ (C) Both ○ (D) Neither

Solution: This scheme is not hiding, because we can bruteforce every possible $\texttt{HMAC}(K, x)$ (K is fixed).

This scheme is binding due to the collision-resistance of HMAC.

Q6.8 (1 point) $\texttt{COMMIT}(x) = \mathsf{Sign}(SK, x)$, where $\mathsf{Sign} = x^d \bmod N$ (naive RSA signature) and SK is the secret key for the user sending the bid.

● (A) Binding ○ (B) Hiding ○ (C) Both ○ (D) Neither

Solution: This scheme is not hiding, because we can immediately leak x from $x^d \bmod N$ by raising it to the power of e (part of the public key).

This scheme is binding because we can immediately leak x from the original scheme (among other reasons).

Q6.9 (1 point) $\texttt{COMMIT}(x) = \mathsf{Sign}(SK, x)$, where $\mathsf{Sign} = H(x)^d \bmod N$ (hash-based RSA signature).

● (A) Binding ○ (B) Hiding ○ (C) Both ○ (D) Neither

Solution: This scheme is not hiding, because we can immediately leak $H(x)$ from $H(x)^d \bmod N$ by raising it to the power of e (part of the public key). Given $H(x)$ we can bruteforce x per the question assumptions.

This scheme is binding because we can immediately leak $H(x)$ from the original scheme (among other reasons).

Q6.10 (1 point) $\texttt{COMMIT}(x) = g^x \bmod p$, where g is a generator and p is large prime (like in Diffie-Hellman).

● (A) Binding ○ (B) Hiding ○ (C) Both ○ (D) Neither

Solution: This scheme is not hiding, because we can bruteforce all $g^x \bmod p$.

This scheme is binding because g^x is a unique value modulo p given that g is a generator.

Q6.11 (1 point) Note: (x, y) denotes a tuple of two values, x and y.

$$\texttt{COMMIT}(x) = \begin{cases} (x, x+1) & \text{with probability 0.5} \\ (x-1, x) & \text{with probability 0.5} \end{cases}$$

○ (A) Binding ○ (B) Hiding ○ (C) Both ● (D) Neither

Solution: This scheme is not hiding, because we can learn, within 1, the value of x, which leaks some information about x.

This scheme is not binding because you could choose x, send $(x, x+1)$, and claim that your bid was actually $x+1$.

Specifically, EvanBot might choose $x = 4$, and send $(4, 5)$. Then, EvanBot could claim that their bid was actually $x' = 5$, and that they sent $(x-1, x) = (4, 5)$. Other users cannot detect whether EvanBot's bid was $x = 4$ or $x' = 5$, since the commit $(4, 5)$ is valid for both bids.

The properties, repeated for your convenience:

1. **Binding**: If a party chooses x, sends $\texttt{COMMIT}(x)$, and then reveals $x' \neq x$, other parties can detect that x' is invalid with overwhelming probability.
2. **Hiding**: $\texttt{COMMIT}(x)$ should not leak more than a negligible amount of information about x.

Q6.12 (8 points) Design a commitment scheme that was not mentioned in a previous subpart. Your scheme should be both binding and hiding.

Assume you have access to an unlimited source of random bits.

Given x, how do you compute $\texttt{COMMIT}(x)$?

Solution: There are many possible solutions, the most common being:

- $H(x\|r)$ for random r,
- $\mathsf{HMAC}(K, x)$ for random K
- $H(x) \oplus H(K)$

The core idea is that we need some sort of randomness to avoid brute-force (hiding), and some sort of one-way, collision resistant function to achieving binding.

$\mathsf{HMAC}(K, x)$ works because it is similar $H(x\|R)$, with K being the source of randomness inside a hash function. Lots of submissions used $\mathsf{HMAC}(K, E_K(x))$ or $(\mathsf{Enc}(K, x), \mathsf{HMAC}(K, \mathsf{Enc}(K, x)))$ but this was unnecessary – $\mathsf{HMAC}(K, x)$ by itself is hiding if K is unknown and randomly generated.

There was an edge case for our definitions: $E_K(x)$ for randomly generated (private) K. The intention was this was not binding, since we can generate a new key K', run $D_{K'}(E_K(x)) = x'$ to get a new x', K' pair that will validate correctly. However, the assumption that the range of valid bids is small enough to brute-force means the probability this new x' is valid is negligible, so we are giving credit for this response.

See the box on the next page for some examples of incorrect schemes.

List of values sent in the reveal step (including x):

Solution: For $H(x\|r)$ we would send x, r.

List of steps to verify $\texttt{COMMIT}(x)$ given the values from the reveal step:

Solution: Reconstruct $H(x\|r)$ from x, r and check that it matches $\texttt{COMMIT}(x)$.

Solution:

Some common ideas that do NOT work:

$g^{xr} \bmod p$ for random x. We can change x to x' as long as we also change r to r' such that $x'r' \equiv xr \bmod (p-1)$. This works when x' is invertible modulo $p-1$, which for correctly-chosen p is about 50%. (Essentially this is equivalent to solving $Ar = xr \bmod N$ where $A = x'$ (the new x value), so we can solve it if A is invertible. Therefore we just need to pick a new x' that is coprime to $p-1$.

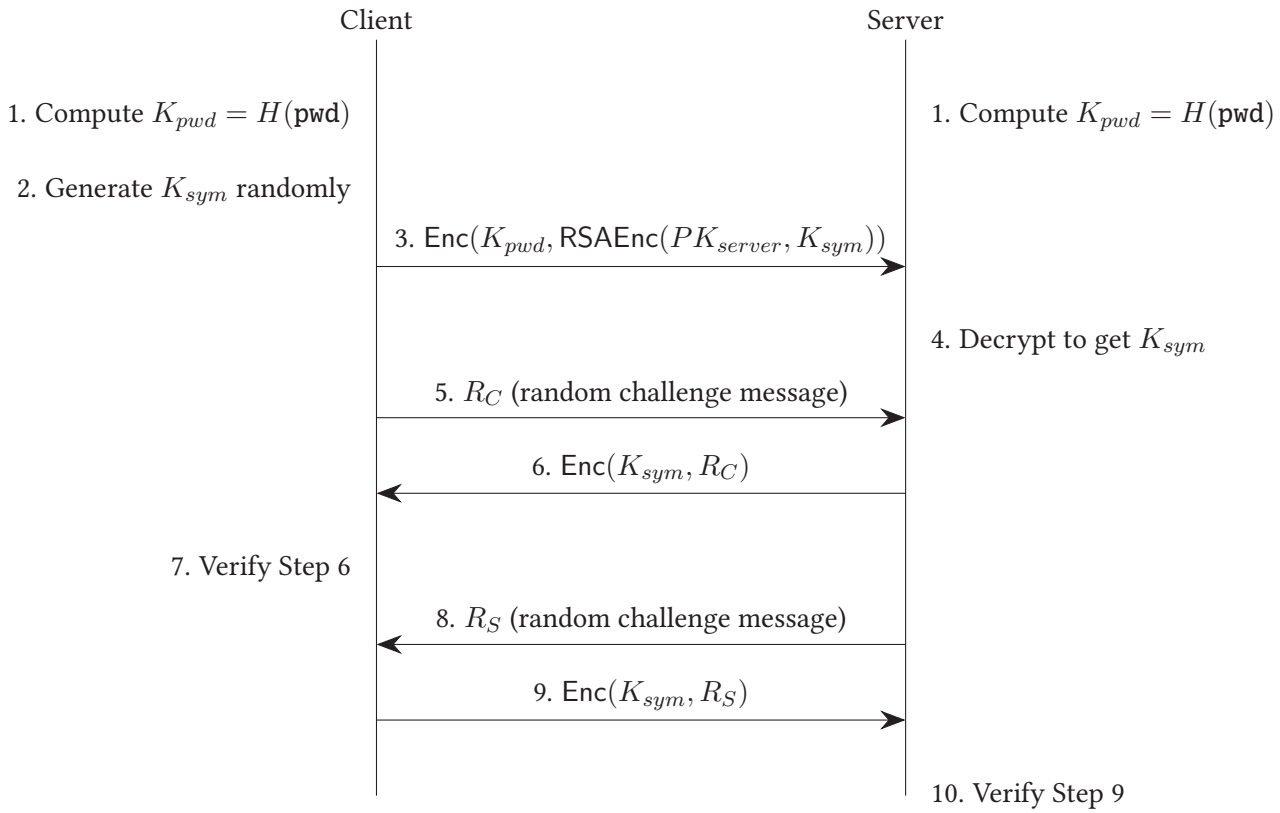
$x^r \bmod p$ for random r. Represent $x = g^k \bmod p$ for some generator g. Thus $x^r \equiv g^{kr} \bmod p$. Like in the previous example, we can find k', r' such that $k'r' \equiv kr \bmod (p-1)$, then set $x = g^{k'} \bmod p$. It's not too difficult to make sure these new x are valid bids.

Any scheme that has $x \oplus K$ or $x \oplus R$ as some input, since we can replace x with x' and change K to be $K' = K \oplus x \oplus x'$.

$H(x\|s)$ for a publicly-known salt s. We can bruteforce x because we know all the other inputs to H, so this is not hiding.

Q7 *Ephemeral Exchanges* **(17 points)**

Consider the following authentication protocol. `pwd` is a standard-strength password (i.e. vulnerable to brute-force). PK_{server} is a long-term, trusted public key for the server. Assume there's only a single user/password stored on the server.



Here is an equivalent description of the protocol:

1. Both the client and server derive $K_{pwd} = H(\texttt{pwd})$.
2. The client generates a random symmetric key K_{sym}.
3. The client sends $\mathsf{Enc}(K_{pwd}, \mathsf{RSAEnc}(PK_{server}, K_{sym}))$ to the server.
4. The server decrypts the message from the previous step to get K_{sym}.
5. The client sends a randomly generated number R_C to the server (challenge message).
6. The server replies with $\mathsf{Enc}(K_{sym}, R_C)$.
7. The client verifies that the server's response is valid.
8. The server sends a randomly generated number R_S to the client (challenge message).
9. The client replies with $\mathsf{Enc}(K_{sym}, R_S)$.
10. The server verifies that the client's response is valid.

Q7.1 (1 point) Which equation does the client use in Step 7 to verify the server's response from Step 6 (denoted STEP6)?

- (A) $\mathsf{Dec}(K_{sym}, \texttt{STEP6}) = R_C$.
- (B) $\mathsf{Dec}(K_{sym}, \texttt{STEP6}) = R_S$.
- (C) $\mathsf{Dec}(K_{pwd}, \texttt{STEP6}) = R_C$.
- (D) $\mathsf{Enc}(K_{sym}, \texttt{STEP6}) = R_C$.

Solution: The server sends $\texttt{STEP6} = \mathsf{Enc}(K_{sym}, R_C)$, so $\mathsf{Dec}(K_{sym}, \texttt{STEP6}) = \mathsf{Dec}(K_{sym}, \mathsf{Enc}(K_{sym}, R_C)) = R_C$.

Q7.2 (1 point) Which option best explains why the two parties don't use $H(\texttt{pwd})$ for the final shared symmetric key K_{sym}?

Clarification during exam (for 7.2, 7.3, 7.4): Assume that the attacker also records messages encrypted and MAC-d with K_{sym} after the protocol finishes.

- (A) The attacker can brute force values of $H(\texttt{pwd})$ and check candidate keys using messages from the resulting secure channel.
- (B) H outputs too many bits to be used as a symmetric key.
- (C) $H(\texttt{pwd})$ would be the same across different authentications.
- (D) $H(\texttt{pwd})$ would require a salt to prevent dictionary attacks.

Solution: If the parties encrypted/MAC-d all their real messages with K_{pwd}, then the attackers can bruteforce all possible pwd, derive the respective K_{pwd}, and try verifying a MAC to check if the guess is right.

Q7.3 (2 points) Is it possible for an eavesdropper to dictionary attack `pwd` given $\mathsf{Enc}(K_{pwd}, \mathsf{RSAEnc}(PK_{server}, K_{sym}))$ from Step 3?

- ○ (A) Yes, they can try decrypting $\mathsf{Enc}(K_{pwd}, \mathsf{RSAEnc}(PK_{server}, K_{sym}))$ with each guess of `pwd`.
- ○ (B) Yes, because anyone can evaluate $\mathsf{RSAEnc}(PK_{server}, K_{sym})$ and check that $\mathsf{Enc}(K_{pwd}, \mathsf{RSAEnc}(PK_{server}, K_{sym}))$ evaluates to the existing value.
- ● (C) No, because there is no way to tell when a test decryption is successful.
- ○ (D) No, because RSAEnc is slow to evaluate, preventing brute force.

Solution: A successful test decryption would produce $\mathsf{RSAEnc}(PK_{server}, K_{sym})$, which appears random since K_{sym} was randomly generated.

Q7.4 (2 points) An attacker records all messages sent between a client and server during a successful login.

Later, the attacker learns the value of SK_{server}, the server's secret key (corresponding to PK_{server}).

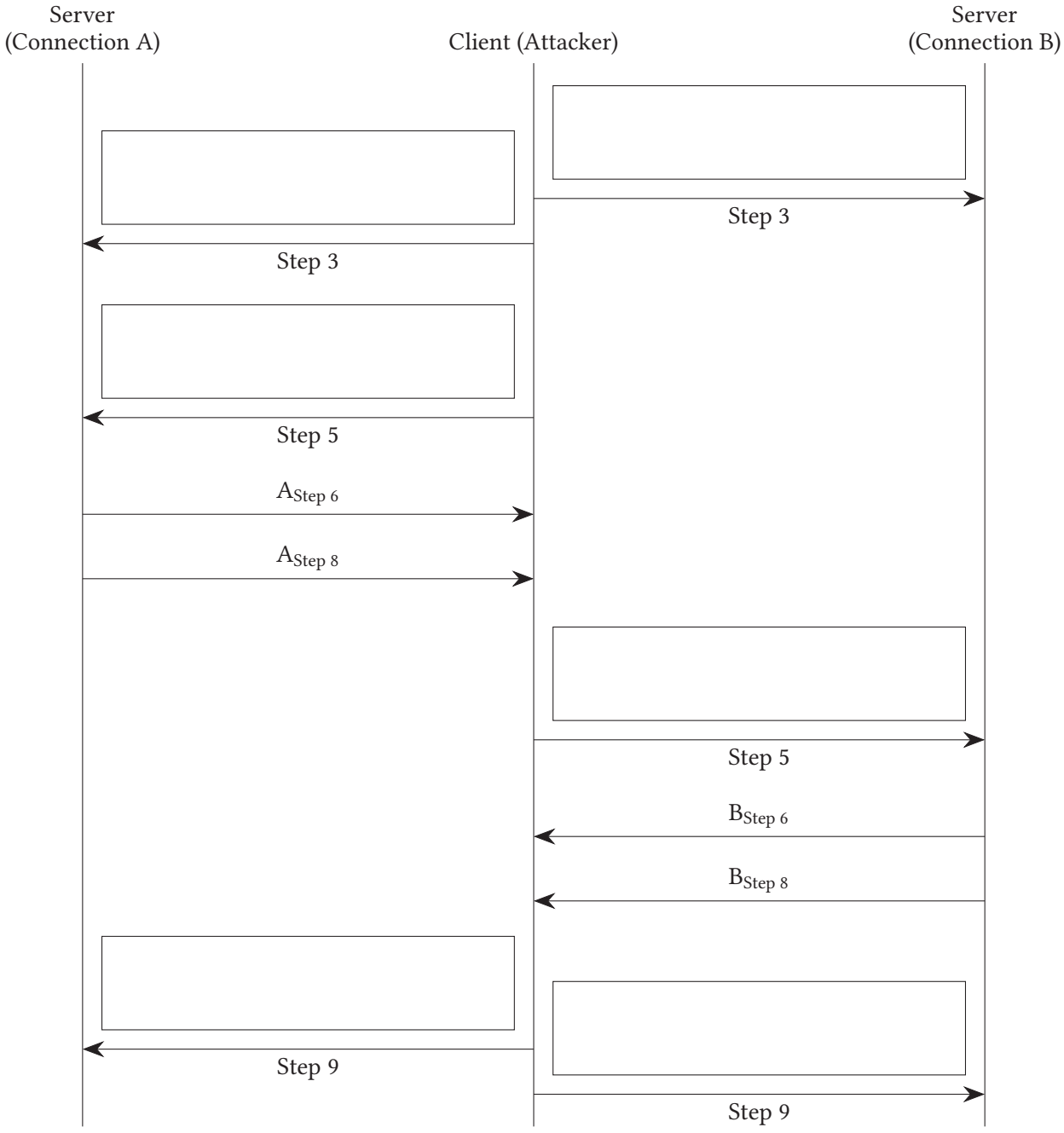
Can the attacker learn the value of K_{sym} from the earlier recorded login?

- ● (A) Yes, because the attacker can try all possible values of `pwd` and find K_{sym}.
- ○ (B) Yes, because the attacker can try all possible values of K_{sym} once SK_{server} is leaked.
- ○ (C) No, because K_{sym} is randomly generated for each authentication and deleted immediately afterwards.
- ○ (D) No, because there is no way to tell when a test decryption is successful, even if we leak SK_{server}.

Solution: If we remember the solution from 7.3, a test decryption using K_{pwd} gives $\mathsf{RSAEnc}(PK_{server}, K_{sym})$, which we can now also decrypt since we are given SK_{server}. Now we would have a K_{sym} to test verify on a real message from the resulting secure channel.

Q7.5 (8 points) Design an attack to successfully pass Step 10 (server verification) without knowing `pwd`. Assumptions:

1. You start two simultaneous connections to the server: Connection A and Connection B. The same server is shown twice in the diagram below.
2. You do not need to verify the server's response in Step 7.
3. In each of the six boxes below, you can write:
   - $A_{\text{Step X}}$ or $B_{\text{Step X}}$ to denote the value from step X of Connection A/B.
   - If any value would work for a box, write the word `anything`.



Solution:

1. Connection B step 3: `anything`
2. Connection A step 3: $B_{\text{Step 3}}$
3. Connection A step 5: `anything`
4. Connection B step 5: $A_{\text{Step 8}}$
5. Connection A step 9: $B_{\text{Step 6}}$
6. Connection B step 9: `anything`

The key idea here is to force both connections to have the same K_{sym} by using the same value in step 3. From there, we can send any value for connection A step 5 (client challenge) and ignore the server's response in step 6 since we don't need to verify it. Next, the server's challenge is given in $A_{\text{Step 8}}$. To pass it, we use $A_{\text{Step 8}}$ as the client challenge for connection B, so the server returns $\mathsf{Enc}(K_{sym}, A_{\text{Step 8}})$ in connection B step 6. This is the value the server expects from us in connection A, so we send it in connection A step 9 to pass the server's challenge. We only need one connection to succeed, so we can send anything for connection B step 9 (which causes connection B to fail).

Alternate solutions for step 3 are possible as long as both connections have the same value for step 3. However, any solution that uses K_{pwd} does not earn credit, since the premise of this question is that the attacker does not know `pwd`.

Q7.6 (1 point) In which of the two connections does the server successfully verify the challenge response sent by the client in Step 9?

● (A) Connection A ○ (B) Connection B ○ (C) Both connections

Solution: See above.

Q7.7 (2 points) Which of the following modifications would prevent an optimal attack from Q7.5? Select all that apply.

■ (A) Adding an HMAC keyed with K_{sym} on the challenge messages (Steps 5, 8).

■ (B) Having the server send its challenge message first (i.e. swap steps 5-7 with steps 8-10).

■ (C) Replacing Step 6 with "The server replies with $\mathsf{Enc}(K_{sym}, R_C \| R_S)$"

☐ (D) Generating the challenge numbers using the output of a PRNG seeded with `pwd`.

☐ (E) None of the above

Solution: The first option prevents the attack because we have no way of either replaying or creating an HMAC on the client's challenge message.

Having the server send its challenge message first does prevent the attack, as we can never get past this step in order to have the server answer its own challenge.

Having the server's challenge include both R_C and R_S does help, as we can never use a server's response as a reply to its own challenge, as the format will be different from a normal client response.

Generating the challenge numbers using the output of a PRNG seeded with `pwd` does not help as we simply replay existing messages.

Nothing on this page will affect your grade.

Post-Exam Activity

What is Mallory cooking?

Comment Box

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