CS 161 Fall 2023 — Introduction to Computer Security — Midterm

Solutions last updated: October 7, 2023

Name: _____

Student ID: _____

This exam is 110 minutes long.

Question:	1	2	3	4	5	6	7	Total
Points:	1	13	14	17	16	22	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:

(Just for fun, not graded.)

Let's play Rock Paper Scissors with EvanBot! Circle your move below, and we will reveal your result after the exam!

Rock Paper Scissors

To prove EvanBot isn't cheating, here is the SHA256 hash of EvanBot's move:

```
406dba28c82f3d8ff18b2df401d5b02c
6b1debc42ca5866585497fdb23e51741
```

Q1 *Honor Code* (1 point)

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* **(13 points)**

Each true/false is worth 1 point.

Q2.1 EvanBot brings both an umbrella and a raincoat, so that EvanBot will stay dry even if the raincoat tears or the umbrella breaks.

True or False: This is an example of defense in depth.

● (A) True ○ (B) False

Solution: True. EvanBot is using two different defenses against the same attack, so that if one of the defenses fails, the other can still work.

Q2.2 A store installs sensors at the main exit to detect shoplifting, but there is another exit in the back of the store without sensors installed.

True or False: This is a failure to ensure complete mediation.

● (A) True ○ (B) False

Solution: True. In order to ensure complete mediation, the store should make sure that every exiting customer undergoes the same check.

For the next 3 subparts: Suppose we have a little-endian C program with a local variable `char buf[64]`. Consider the following possible GDB output after running the command `x/8wx buf`:

```
0xffff1430: 0x00003500 0x6ca59820 0x0000abcd 0x74392bc2
0xffff1434: 0x00000000 0x00000000 0x00000000 0x00000000
```

Q2.3 True or False: The addresses in the left-most column of this GDB output are valid GDB output.

○ (A) True ● (B) False

Solution: False. This output has address label `0xffff1430` followed by 16 bytes of output. Then, it has an address label `0xffff1434` (4 bytes higher) for the byte that actually comes 16 bytes later. This label should instead be `0xffff1440`.

Q2.4 True or False: `buf[5]` is `0x98`.

● (A) True ○ (B) False

Solution: True. `0x00003500` is the word that contains `buf[0]`, `buf[1]`, `buf[2]`, and `buf[3]`.

Then, in the word `0x6ca59820`, since the system is little-endian, `buf[4]` is the LSB of this word, `0x20`. Then, `buf[5]` is the second-least-significant byte of this word, `0x98`.

Q2.5 True or False: The command `x/16wx buf` would display all 64 bytes in `buf`.

● (A) True ○ (B) False

Solution: True. This command prints out 16 words = 64 bytes starting at `buf`.

Q2.6 True or False: A C program that never lets the user change the values of the RIP and SFP is safe against all memory safety attacks.

○ (A) True ● (B) False

Solution: False. It would still be possible to overwrite local variables, for example. (Other more sophisticated attacks may also exist, e.g. overwriting a pointer.)

Q2.7 True or False: A 128-bit stack canary would be secure against any practical brute-force attacks against the canary.

● (A) True ○ (B) False

Solution: True. An attacker would need to try 2^{128} possibilities, which is infeasible (in the same way that guessing a 128-bit key is infeasible).

Q2.8 True or False: When pointer authentication is enabled, an attacker who inspects the program memory cannot learn the values of any addresses.

○ (A) True ● (B) False

Solution: False. The address values still appear in memory, but with an extra pointer authentication code replacing the top bits (which used to be all zero). Pointer authentication stops an attacker from modifying addresses; it does not stop an attacker from reading addresses.

Q2.9 You are using GDB to debug a program with ASLR enabled.

True or False: Running a valid `x` command (e.g. `x/4wx var` for some local variable `var`) will always help you learn something about the address randomization on the **current** run of the program.

● (A) True ○ (B) False

Solution: True. The `x` command outputs the address of the variable, in addition to memory at that address. Therefore, you can use the output to learn something about address randomization.

Q2.10 You are using GDB to debug a program with ASLR enabled.

True or False: Running a valid `x` command will always help you learn something about the address randomization on a **different** run of the program.

○ (A) True ● (B) False

Solution: False. Even though the command will show you some addresses for the current run of the program, on a different run of the program, the addresses will probably change.

Q2.11 True or False: Publicly revealing the value of nothing-up-my-sleeve numbers does not compromise the security of the cryptographic system.

● (A) True ○ (B) False

Solution: True. Nothing-up-my-sleeve numbers are, by definition, publicly-known constants.

Q2.12 Alice decides to build a PRNG with a secure hash function H. For a seed s, she outputs $H(s\|0)$ and sets the new seed to $H(s\|1)$.

True or False: This PRNG is rollback-resistant.

● (A) True ○ (B) False

Solution: True by the one-wayness of H.

Q2.13 True or False: Creating a new certificate requires generating a secure digital signature.

● (A) True ○ (B) False

Solution: True. By definition, a certificate is a signed attestation of somebody's public key, so to generate a new certificate, we need to generate a digital signature.

Q3 *Memory Safety: Homecoming* **(14 points)**

Consider the following vulnerable C code:

```
1  void vulnerable() {
2      char vulnerable_buf[32];
3      fgets(vulnerable_buf, 32, stdin);
4      helper(vulnerable_buf);
5  }
6
7  void helper(char* arg) {
8      char helper_buf[32];
9      fgets(helper_buf, 37, stdin);
10 }
```

These assembly instructions occur in memory:

```
1 0x08076000: ret
2 0x08076004: pop %eax
3 ...
4 0x08076030: call helper
5 0x08076034: add $4, %esp
6 0x08076038: mov %ebp, %esp
7 0x0807603c: pop %ebp
8 0x08076040: ret
```

Stack at Line 8

RIP of `vulnerable`
(1)
(2)
(3)
(4)
(5)
(6)

Assumptions:

- You may use `SHELLCODE` as a 31-byte shellcode.
- ASLR is enabled, **not** including the code segment (i.e. the addresses in the code segment don't change).
- Using GDB, you find that the RIP of `helper` has value `0x08076034`.
- Unless otherwise specified, all other memory safety defenses are disabled.

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

- ○ (A) (1) `arg` (2) SFP of `vulnerable` (3) `vulnerable_buf`
- ○ (B) (1) SFP of `vulnerable` (2) `arg` (3) `vulnerable_buf`
- ● (C) (1) SFP of `vulnerable` (2) `vulnerable_buf` (3) `arg`
- ○ (D) (1) `vulnerable_buf` (2) SFP of `vulnerable` (3) `arg`

Q3.2 (1 point) What values go in blanks (4) through (6) in the stack diagram above?

- ○ (A) (4) `helper_buf` (5) RIP of `helper` (6) SFP of `helper`
- ○ (B) (4) RIP of `helper` (5) `helper_buf` (6) SFP of `helper`
- ● (C) (4) RIP of `helper` (5) SFP of `helper` (6) `helper_buf`
- ○ (D) (4) SFP of `helper` (5) `helper_buf` (6) RIP of `helper`

Solution:

RIP of `vulnerable`

SFP of `vulnerable`

`char vulnerable_buf[32]`

`char* arg`

RIP of `helper`

SFP of `helper`

`char helper_buf[32]`

Q3.3 (2 points) Recall the off-by-one exploit in Project 1, where we overwrite an SFP with a specific value that causes shellcode to execute.

Why does the off-by-one exploit not work here with 100% probability?

- ● (A) ASLR is enabled, so we don't know what value to overwrite an SFP with.
- ○ (B) ASLR is enabled, so instructions written on the stack are not executable.
- ○ (C) The off-by-one exploit requires two returns after overwriting an SFP, and this code only returns once after overwriting an SFP.
- ○ (D) It is not possible to overwrite an SFP in this code.

Solution:

Recall that in the off-by-one exploit, we overwrite the SFP with the address of some part of memory we control. (The program then thinks that the caller function's RIP/SFP are located in some part of memory we control.)

In this question, we control the values that are written to the two buffers, both on the stack. However, if ASLR is enabled, we don't know the address of either buffer, so we cannot overwrite the SFP with the address of either buffer.

Option (B) is false. Non-executable pages is the defense that causes instructions written on the stack to be non-executable. ASLR does not affect whether segments of memory are executable.

Option (C) is false. The `fgets` call inside `helper` overwrites the SFP of `helper`. Then, we return from `helper` back to `vulnerable`. Then, `vulnerable` returns. This is a sequence of two function returns after the SFP is overwritten.

Option (D) is false. The `fgets` call at Line 9 allows us to overwrite 4 bytes above `helper_buf`, where the SFP of `helper` is located.

Here is an exploit that causes shellcode to execute with 100% probability:

Input to `fgets` in `vulnerable`: `SHELLCODE`

Input to `fgets` in `helper`: `SHELLCODE + 5*'A'`

Q3.4 (2 points) The function epilogue for `helper` runs:

```
mov %ebp, %esp
pop %ebp
ret
```

What is the next instruction executed after `ret`?

- ○ (A) `pop %eax`
- ○ (B) `pop %ebp`
- ● (C) The `ret` instruction at `0x08076000`
- ○ (D) The `ret` instruction at `0x08076040`
- ○ (E) The `SHELLCODE` in `vulnerable_buf`
- ○ (F) The `SHELLCODE` in `helper_buf`

Solution: The exploit overwrites the LSB of the RIP of `helper` with a null byte. The RIP of `helper` now has value `0x08076000`. When the `helper` function returns, we jump to this address and start executing instructions. The next instruction executed is the `ret` instruction at address `0x08076000`.

Longer solution available at the bottom of this question.

Q3.5 (2 points) What is the next instruction executed after your answer to Q3.4?

- ○ (A) `pop %eax`
- ○ (B) `pop %ebp`
- ○ (C) The `ret` instruction at `0x08076000`
- ○ (D) The `ret` instruction at `0x08076040`
- ● (E) The `SHELLCODE` in `vulnerable_buf`
- ○ (F) The `SHELLCODE` in `helper_buf`

Solution: Recall that the `ret` instruction treats the next value on the stack as an address, and causes the program to start executing instructions at that address.

When the `ret` instruction executes, the next value on the stack is the value directly above the RIP of `helper`. This is the argument to helper, which is a pointer to (aka the address of) `vulnerable_buf`.

The program jumps to `vulnerable_buf` and starts executing the shellcode we put in there.

Q3.6 (2 points) For this subpart only, suppose we run this exploit on a **big-endian** system instead of a little-endian system.

Does the exploit listed above still work?

- ○ (A) Yes, because the same bytes are written into memory.
- ○ (B) Yes, because endianness does not affect shellcode (the instructions are always executed from low to high addresses).
- ● (C) No, because a different byte of the RIP is overwritten.
- ○ (D) No, because endianness affects shellcode (we would have to reverse the shellcode to make the exploit work).

Solution: This exploit relies on the `fgets(helper_buf, 37, stdin)` call overwriting the LSB of the RIP of `helper`.

In a little-endian system, the LSB is stored at the lowest address, so when we overwrite the byte immediately above the SFP, we end up overwriting the LSB of the RIP.

In a big-endian system, the MSB is stored at the lowest address instead, so this exploit would instead cause the MSB of the RIP to be overwritten.

This would cause the RIP's value, `0x08076034`, to be replaced with `0x00076034`. This is no longer the address of a `ret` instruction, so the exploit no longer works.

Option (D) is incorrect because the statement is false. Instructions in memory are executed from lower addresses to higher addresses, so we do not need to reverse the shellcode. (If you've taken 61C, recall that the program counter increments by 4 after executing an instruction, indicating that the next instruction is at a higher address.)

Q3.7 (2 points) For this subpart only, suppose that we swap the `ret` instruction at address `0x08076000` and the `pop %eax` instruction at address `0x08076004`.

Does the exploit listed above still work?

- ○ (A) Yes, with no modifications.
- ○ (B) Yes, but only if we replace `'A'*5` with `'\x04'*5`.
- ● (C) No, because we cannot change the last byte that `fgets` writes in memory.
- ○ (D) No, because `pop %eax` is not a valid x86 assembly instruction.

Solution: Option (A) is incorrect. If we leave the exploit unmodified, then the program will still try to execute instructions starting at address `0x08076000`. Now, the first instruction executed is `pop %eax`, which pops `arg` off the stack.

Next, when we execute the `ret` instruction at `0x08076004`, the next value on the stack is `vulnerable_buf`, which is not the address of shellcode, so the address we jump to is not the address of shellcode. (Not needed to solve this question, but since `vulnerable_buf` contains shellcode, we'll attempt to read the first 4 bytes of shellcode as an address and jump to that address, which will likely crash the program.)

Option (B) is incorrect. The 5 garbage bytes are written to `helper_buf` and the SFP of `helper`. They are not written to the RIP of `helper`. With this modification, the RIP of `helper` is still overwritten to be `0x08076000` (because of the null byte appended by `fgets`), and the exploit fails for the same reasoning as in Option (A).

Option (C) is the best answer choice. To make this exploit work again, we would need to overwrite the RIP of `helper` with `0x08076004` instead of `0x08076000`, but this is not possible, because the null byte is not part of the input we control; it is automatically appended by `fgets`.

Option (D) is incorrect because the statement is false. Executing a `ret` instruction results in the same behavior, no matter where that instruction is located in memory.

Q3.8 (2 points) For this subpart only, suppose that we run this code on a 64-bit system, with pointer authentication codes (PACs) enabled.

Also, suppose that `fgets(helper_buf, 37, stdin)` is changed to `fgets(helper_buf, 41, stdin)`.

Does the exploit listed above still work?

- ○ (A) Yes, because the exploit never changes the value of `arg` in memory.
- ○ (B) Yes, because the exploit modifies addresses without overwriting any PACs.
- ○ (C) No, because PACs prevent all buffer overflows.
- ● (D) No, because the exploit changes the value of an address in memory.

Solution: Recall that in a 64-bit system, addresses are 64 bits = 8 bytes long.

If we keep the exploit as-is, then we only overwrite the lower 5 bytes of the SFP with garbage/null. This will still cause the program to crash as shown below.

If we modify the exploit to use 9 garbage bytes, we overwrite `helper_buf` with 31 bytes of shellcode, followed by a garbage byte. Then, we overwrite the 8-byte SFP with 8 garbage bytes. Finally, `fgets` overwrites the LSB of the 8-byte RIP with a null byte.

Recall that with PACs enabled, the most-significant bits of a pointer (which are all zeros) are replaced with a PAC that is unique to that address. When we change the address without changing the PAC, the PAC is no longer correct, so the program will crash when trying to dereference this address. This happens when we overwrite the LSB of the RIP without changing the PAC.

Option (C) is incorrect because the statement is false. Buffer overflows are still possible even with PACs enabled, as long as no addresses are overwritten (e.g. overwriting local variables).

Solution: Longer solution:

Recall that the `fgets` function writes a maximum of `n-1` bytes (one fewer than the number supplied by the user) into memory, and then appends a null byte at the end of the user's input. Both of our inputs are exactly `n-1` bytes long, and each will cause `n` bytes to be written into memory: our input, followed by a null byte. (The value of `n` is different for each call.)

We pass the 31-byte shellcode into the `fgets(vulnerable_buf, 32, stdin)` call on Line 3. This causes the 32-byte `vulnerable_buf` to be filled with 31 bytes of shellcode, followed by a null byte. No part of memory outside of the buffer is modified as a result of this `fgets` call.

We pass 31 bytes of shellcode and 5 garbage bytes into the `fgets(helper_buf, 37, stdin)`. This causes the 32-byte `helper_buf` to be filled with 31 bytes of shellcode, followed by one garbage

byte. Then, immediately above `helper_buf` is the SFP of `helper`, which gets overwritten by the other 4 garbage bytes. Finally, the null byte is written into memory. This overwrites the byte in memory directly above the SFP of `helper`, namely: the least-significant byte (LSB) of the RIP of `helper`.

Here is the stack diagram after our inputs are written into memory:

RIP of `vulnerable`

SFP of `vulnerable`

`char vulnerable_buf[32]` - overwritten with 31 bytes of shellcode, followed by a null byte.

`char* arg`

RIP of `helper` - LSB is overwritten with a null byte.

SFP of `helper` - overwritten with 4 garbage bytes.

`char helper_buf[32]` - overwritten by 31 bytes of shellcode, followed by 1 byte of garbage.

From the assumptions, the RIP of `helper` had value `0x08076034`. When we overwrite the LSB of the RIP with a null byte, the value changes to `0x08076000`.

When `helper` returns, we start executing instructions in memory starting at address `0x08076000`. According to the assembly dump, there is a `ret` instruction at `0x08076000` that gets executed.

Just before the `ret` instruction in `helper`'s epilogue executes, the ESP register is pointing at the RIP of `helper`. This is standard function behavior; the ESP points at the RIP just before the `ret` instruction.

After we execute the `ret` instruction in `helper`'s epilogue, the ESP moves up by 4, so that it is now pointing at `char* arg`, the argument to `helper`. This is also standard function behavior; immediately after the function returns, the ESP is pointing at the lowest argument on the stack.

We now execute the `ret` instruction at `0x08076000`. At this point, the ESP is pointing at `arg`, so the value of `arg` will be treated as an address, and we will jump to that address and start executing instructions.

Note: `0x08076034` is the *value* of the RIP stored in memory. The RIP's value is an address, telling us which instruction in memory to execute next after the function returns. Typically the value of the RIP will be an address in the code segment of memory.

`0x08076034` is *not* the *address* of the RIP. The address of the RIP would be some address on the stack, but the address of the RIP is not given in this question, and is not relevant to the exploit.

Q4 *Memory Safety: Forbidden Instruction* **(17 points)**

Consider the following vulnerable C code:

```
1 void vulnerable() {
2     char buf[8];
3     gets(buf);
4 }
```

Assumptions:

- Each x86 instruction is 4 bytes long in machine code.
- The address of `buf` is `0xffff1230`.
- In the entire question, the `SHELLCODE` we want to execute is 16 bytes long.
- The system runs a special variant of non-executable pages: the instruction that assembles to the machine code `0xdeadbeef` cannot be executed on any page marked as non-executable. All other instructions can be executed anywhere in memory.

Q4.1 (3 points) For this subpart only, suppose that the 16-byte shellcode appears in the code section of memory at address `0x10101234`.

Select all inputs to the `gets` call that would cause the shellcode to execute. (Assume every input is followed by a newline character.)

- ☐ (A) `'A' * 8 + '\x34\x12\x10\x10'`
- ■ (B) `'A' * 12 + '\x34\x12\x10\x10'`
- ■ (C) `'\x34\x12\x10\x10' * 4`
- ☐ (D) `'A' * 12 + '\x10\x10\x12\x34'`
- ☐ (E) `'A' * 8 + '\x10\x10\x12\x34'`
- ☐ (F) None of the above.

Solution: The exploit in this subpart is a "classic" buffer overflow like in Project 1, Question 1.

We need to write 12 bytes of garbage to overflow the buffer and the SFP of `vulnerable`. Then, we need to overwrite the RIP of `vulnerable` so that the program executes the shellcode at `0x10101234` upon returning.

Option (A) doesn't work because it writes too few bytes of garbage; it causes the address of shellcode to appear in the SFP of `vulnerable`.

Options (B) and (C) both work, because they have 12 bytes of input, followed by the address of shellcode.

Options (D) and (E) both don't work, because they write the address of shellcode in big-endian, not little-endian. This would cause the little-endian program to jump to address `0x34121010`, which is not where we placed shellcode.

In the next few subparts, suppose that only the *last* instruction of shellcode is the special instruction. In other words, `SHELLCODE[12:16] = 0xdeadbeef`. All other instructions in the shellcode are not the special instruction.

Q4.2 (2 points) Which sequence of bytes must exist in the code section of memory in order for us to execute shellcode?

- ○ (A) `0x00000000`
- ● (B) `0xdeadbeef`
- ○ (C) `0xffff1230`
- ○ (D) `0xffff123c`
- ○ (E) A `nop` instruction
- ○ (F) A `ret` instruction

Solution: The special instruction (`0xdeadbeef`).

We need the special instruction to appear in the code section of memory in order to execute it.

Q4.3 (2 points) Assume that the byte sequence you provided in the previous subpart appears in the code section of memory at address `0x10101234`.

In addition to shellcode, the exploit will need to contain a single extra x86 instruction. What should this extra instruction do?

- ○ (A) Jump to the address `0xdeadbeef`.
- ○ (B) Jump to the address `0xffff1230`.
- ● (C) Jump to the address `0x10101234`.
- ○ (D) The special instruction (`0xdeadbeef`).
- ○ (E) The `ret` instruction.
- ○ (F) The `nop` instruction.

Solution: From the previous subpart, we found that the special instruction appears in memory at address `0x10101234`, so we'll need an instruction that jumps to that memory address to execute that special instruction.

Q4.4 (4 points) Again, assume that the byte sequence you provided in Q4.2 appears at address `0x10101234`.

Provide an input to the `gets` call that would cause the shellcode to execute.

You may use the variable `INST` to represent the 4-byte x86 instruction you answered in the previous subpart.

> **Solution:** `'A' * 12 + \x40\x12\xff\xff + SHELLCODE[0:12] + INST`
>
> The first three parts of this exploit are a classic buffer overflow that redirects execution to the start of shellcode.
>
> After executing the first three instructions (12 bytes) of shellcode, we still need to execute the special instruction. However, we can't write this special instruction on the stack, since the non-executable page defense stops us from executing this instruction when we write it into memory ourselves. Instead, our solution executes a jump instruction that jumps to address `0x10101234`, where the special instruction appears in the code section (which is executable).

For the rest of the question, suppose that only the *first* instruction of shellcode is the special instruction. In other words, `SHELLCODE[0:4] = 0xdeadbeef`. All other instructions in the shellcode are not the special instruction.

Also, suppose that the following x86 instructions appear in the code section of memory. (The sequence of bytes you provided in Q4.2 are no longer in the code section of memory.)

```
1 0x10101280: jmp 0x1010129c
2 0x10101284: pop %eax
3 0x10101288: call helper
4 0x1010128c: add $4, %esp
5 0x10101290: mov %esp %ebp
6 0x10101294: 0xdeadbeef (special instruction)
7 0x10101298: ret
8 0x1010129c: 0xdeadbeef (special instruction)
9 0x101012a0: call exit (terminates the program)
```

Q4.5 (6 points) Provide an input to the `gets` call that would cause the shellcode to execute.

Solution: `'A' * 12 + \x94\x12\x10\x10 + \x44\x12\xff\xff + SHELLCODE[4:16]`

—

The correct sequence of bytes in memory that we need to dereference is: The special instruction (`0xdeadbeef`), followed by the machine code for a `ret` instruction.

As in the previous exploit, we need the special instruction to appear in the code section of memory in order to execute it. Additionally, we need a `ret` instruction immediately afterwards so that we can redirect execution somewhere else and execute the rest of our shellcode.

This answer uses a total of 8 bytes (2 instructions, 4 bytes per instruction). The entire shellcode is 16 bytes long, so it is not the shortest possible sequence.

A 4-byte sequence (e.g. only the special instruction) would not work here, because after the program execute the special instruction, it would try to execute the instruction immediately afterwards in memory, which might not correspond to the next instruction in our shellcode.

—

The pattern of this exploit is similar to a return-oriented programming (ROP) attack we saw from lecture. The two "gadgets" (code snippets) we want to execute are the special instruction, which already exists in memory at `0x10101294`, and the rest of the shellcode, which we'll place in memory ourselves.

As in the classic buffer overflow, we use 12 bytes of garbage to overwrite both the buffer and the SFP. Then, we overwrite the RIP with `0x10101294`, the address of the first gadget.

When the function returns, it will redirect execution to `0x10101294` and execute the special instruction. Then, it will execute the `ret` instruction immediately after the special instruction. The `ret` instruction will take the next item on the stack, read it as an address, and start executing instructions at that address.

Therefore, we should make the next item on the stack (the 4 bytes directly above the RIP) the address of our remaining shellcode. We have to write the remaining shellcode in memory ourselves, so we place it at the very end of our exploit, which happens to be at address `0xffff1244` (20 bytes after the start of buffer). Then, in the 4 bytes directly after the RIP, we place the address of our remaining shellcode.

Q5 *Symmetric-Key Cryptography: Not Quite by DESign* **(16 points)**

Q5.1 (2 points) Suppose we take AES-CBC encryption and replace all AES encryption blocks with HMAC. The new encryption formula for a message $M = (M_1, M_2, \ldots, M_n)$ is:

$$C_0 = IV$$
$$C_i = \mathsf{HMAC}(K, M_i \oplus C_{i-1})$$

Select all true statements about this new scheme.

- ☐ (A) M must be padded until M is a multiple of the HMAC output size before computing C.
- ☐ (B) Someone who knows K is able to decrypt any ciphertext.
- ☐ (C) Someone who does not know K is able to decrypt any ciphertext.
- ■ (D) None of the above.

Solution: HMAC is not invertible, so ciphertext cannot be decrypted, even if you know K.

Padding is not required in this scheme. Suppose the last block M_i is shorter than the HMAC output size. Note that C_{n-1} is the output of HMAC, so it will always be the HMAC output size (and therefore longer than M_i). Then, when computing $M_n \oplus C_{n-1}$, we could throw away the extra bits of C_{n-1} until M_n and C_{n-1} are the same length. Then, we can compute the XOR and pass the result into HMAC, because HMACs take arbitrary-length input.

Q5.2 (2 points) Now suppose we take AES-CTR encryption and replace all AES encryption blocks with HMAC. The new encryption formula is:

$$C_i = M_i \oplus \mathsf{HMAC}(K, IV + i)$$

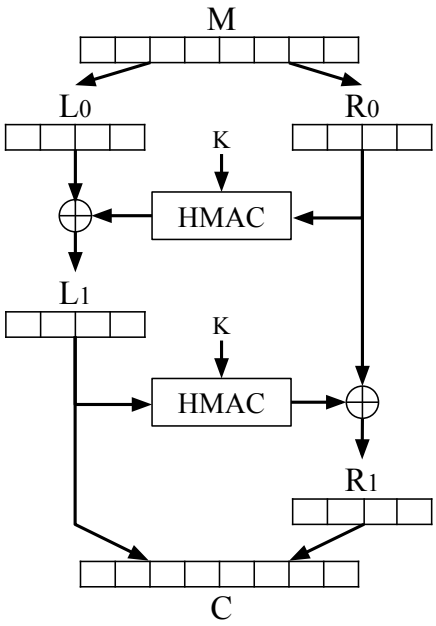
Select all true statements about this new scheme.

- ☐ (A) M must be padded until M is a multiple of the HMAC output size before computing C.
- ■ (B) Someone who knows K is able to decrypt any ciphertext.
- ☐ (C) Someone who does not know K is able to decrypt any ciphertext.
- ☐ (D) None of the above.

Solution: Even though HMAC is not invertible, someone who knows K can create the same one-time pad and correctly decrypt the ciphertext.

Padding is not required in this scheme. If M_n is less than the HMAC output size, we can throw away the extra bits of $\mathsf{HMAC}(K, IV + n)$ to compute the XOR of the last block. This is similar to AES-CTR encryption, where we can throw away the unneeded extra bits of $E(K, IV + n)$.

The rest of the question is independent of the first two subparts.



For the rest of the question, consider this scheme for encrypting 128-bit messages:

1. Split the message into two halves.
   $L_0 = M$`[:64]`
   $R_0 = M$`[64:128]`
2. Set $L_1 = L_0 \oplus \mathsf{HMAC}(K, R_0)$.
3. Set $R_1 = R_0 \oplus \mathsf{HMAC}(K, L_1)$.
4. Concatenate L_1 and R_1 to get the ciphertext.
   $C = L_1 \| R_1$

For this question, you can assume that HMAC outputs 64 bits.

A copy of the diagram on the previous page is available on the appendix.

Q5.3 (2 points) What is the decryption formula for L_0?

L_0 is equal to these values, XORed together. Select as many options as you need.

For example, if you think $L_0 = R_1 \oplus L_1$, then bubble in R_1 and L_1.

- ☐ (A) R_0
- ■ (B) $\mathsf{HMAC}(K, R_0)$
- ☐ (C) R_1
- ☐ (D) $\mathsf{HMAC}(K, R_1)$
- ■ (E) L_1
- ☐ (F) $\mathsf{HMAC}(K, L_1)$

Solution: $L_0 = L_1 \oplus \mathsf{HMAC}(K, R_0)$

Manipulate the equation in encryption step 2 to solve for L_0.

Note: This question is graded all-or-nothing, because the answer choices are not independent.

Q5.4 (2 points) What is the decryption formula for R_0?

R_0 is equal to these values, XORed together. Select as many options as you need.

- ☐ (A) L_0
- ■ (C) R_1
- ☐ (E) L_1
- ☐ (B) $\mathsf{HMAC}(K, L_0)$
- ☐ (D) $\mathsf{HMAC}(K, R_1)$
- ■ (F) $\mathsf{HMAC}(K, L_1)$

Solution: $R_0 = R_1 \oplus \mathsf{HMAC}(K, L_1)$

Manipulate the equation in encryption step 3 to solve for R_0.

Note: This question is graded all-or-nothing, because the answer choices are not independent.

In the next two subparts, consider this scenario: Alice encrypts a message M and gets ciphertext C. Later, Alice encrypts M again, but with one bit in L_0 flipped.

Q5.5 (2 points) What happens to L_1 in the resulting ciphertext?

- ○ (A) It is unchanged.
- ● (B) One bit is flipped.
- ○ (C) It is unpredictably different (garbage).
- ○ (D) None of the above.

Solution: $L_1 = L_0 \oplus \mathsf{HMAC}(K, R_0)$

If we flip one bit in L_0, this causes one bit in L_1 to change.

Q5.6 (2 points) What happens to R_1 in the resulting ciphertext?

- ○ (A) It is unchanged.
- ○ (B) One bit is flipped.
- ● (C) It is unpredictably different (garbage).
- ○ (D) None of the above.

Solution: $L_1 = L_0 \oplus \mathsf{HMAC}(K, R_0)$

$R_1 = R_0 \oplus \mathsf{HMAC}(K, L_1)$

If we flip one bit in L_0, then that causes L_1 to be different in one bit. Then, this causes $\mathsf{HMAC}(K, L_1)$ to be unpredictably different, which in turn causes R_1 to be unpredictably different.

Q5.7 (4 points) Does this scheme provide integrity?

○ (A) Yes ● (B) No

Briefly explain your answer (two sentences or fewer is sufficient).

Solution: Despite using HMAC as the internal function, we never verify the MAC integrity. HMAC is simply a random-looking, keyed function in this case.

Even if we tried to add steps to verify the HMACs, any verification requires an expected input/output pair (message/tag). In our case we always have one of those two variables unknown and dependent on the other, precisely because we use HMAC to derive them instead of verify them.

For example, if we tried to verify $\text{HMAC}(K, L_1)$ we would need to recover R_0. To recover R_0 with need to use $\text{HMAC}(K, L_1)$, meaning R_0 ends up taking whatever value is needed to make the equation work. Effectively, since we need to use HMAC to recover the values required to verify it, we can never verify that the original values were as intended.

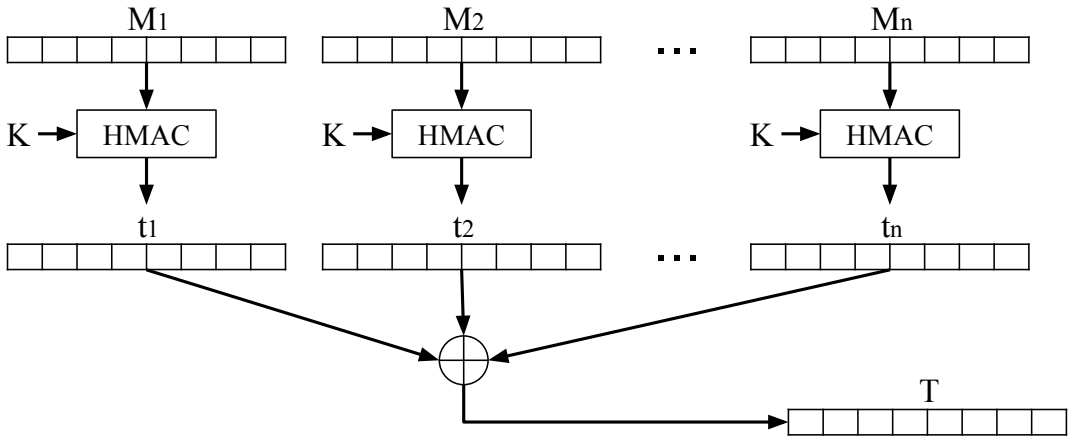
For example, using our decryption equations:

$L_0 = L_1 \oplus \text{HMAC}(K, R_0)$
$R_0 = R_1 \oplus \text{HMAC}(K, L_1)$

We are given some possibly-modified L_1', R_1'. Using $\text{HMAC}(K, L_1')$ and R_1' we can find R_0' that makes the equation true, but this is not necessarily the original R_0. For example, if we had $C = 0^{128}$ (all zeroes) which is extremely unlikely to be a "real" ciphertext, we would just end up setting $R_0 = \text{HMAC}(K, 0)$ and $L_0 = \text{HMAC}(K, \text{HMAC}(K, 0))$ with no way to verify that these are intended or not, since running the encryption algorithm on L_0', R_0' would end up with $C = 0^{128}$.

Q6 *Integrity and Authenticity: Mix-and-MAC* **(22 points)**

Alice designs a scheme that generates a single MAC on a list of n messages $M_1, M_2, \ldots, M_n$.



1. Compute HMACs on each individual message. $t_i = \mathsf{HMAC}(K, M_i)$, for $1 \leq i \leq n$.
2. XOR all the HMAC outputs (t_i) together to get the final MAC output. $T = t_1 \oplus t_2 \oplus \ldots \oplus t_n$.

Q6.1 (2 points) Does this scheme require the message length to be less than or equal to the length of the HMAC output?

- ○ (A) Yes, because HMAC processes messages one block at a time.
- ○ (B) Yes, because XOR cannot be done between two different-length bitstrings.
- ○ (C) No, because HMAC pads shorter messages to the block length.
- ● (D) No, because HMAC takes in arbitrary-length inputs and outputs fixed-length outputs.

Solution: By definition, HMAC can take in arbitrary-length inputs.

Option (C) is false. HMAC does not pad shorter messages.

Q6.2 (2 points) Alice computes the MAC for the message list $[M_1, \ldots, M_n]$. She sends the message list and the MAC to Bob.

Bob adds a new message M_{n+1} to the list, and wants to compute the MAC of the new message list $[M_1, \ldots, M_n, M_{n+1}]$.

What is the minimum number of HMACs that Bob needs to compute in order to compute the MAC of the new message list?

○ (A) 0 ● (B) 1 ○ (C) 2 ○ (D) $n/2$ ○ (E) n ○ (F) $n+1$

Solution: Bob computes t_{n+1}, which only takes one HMAC to compute. Then Bob can compute $T \oplus t_{n+1}$ to get the MAC of the new list.

A copy of the diagram on the previous page is available on the appendix.

Q6.3 (4 points) Alice computes the MAC for two message lists:

- The list $A = [A_1, A_2, \ldots, A_n]$ has MAC T_A.
- The list $B = [B_1, B_2, \ldots, B_n]$ has MAC T_B.

Mallory observes both message lists and both MACs. Mallory does not know K.

Mallory wants to compute a valid MAC on some message list that is not A or B.

Give a valid (message list, MAC) pair that Mallory could compute.

The message list is:

Solution: $[A_1, A_2, \ldots, A_n, B_1, B_2, \ldots, B_n]$

This message list contains all the messages in A and B.

The MAC on the above message list is:

Solution: $T_A \oplus T_B$

This tag can be computed without knowing K. Solutions that try to use K in their expression for the tag would be incorrect, because Mallory does not know K.

The MAC of the combined list of messages is:

$$\mathsf{HMAC}(K, A_1) \oplus \ldots \oplus \mathsf{HMAC}(K, A_n) \oplus \mathsf{HMAC}(K, B_1) \ldots \oplus \mathsf{HMAC}(K, B_n)$$

But we know that

$$T_A = \mathsf{HMAC}(K, A_1) \oplus \ldots \oplus \mathsf{HMAC}(K, A_n)$$
$$T_B = \mathsf{HMAC}(K, B_1) \oplus \ldots \oplus \mathsf{HMAC}(K, B_n)$$

So we can simplify the MAC expression to $T_A \oplus T_B$.

Note: Answers where the MAC is an expression in terms of K were graded as incorrect, because Mallory does not know K (and would not be able to compute such a MAC).

Q6.4 (4 points) Mallory does not know K. Mallory wants to compute a valid MAC on [pancake], which is a list containing only one message (namely "pancake").

Mallory is allowed to ask for the MAC of two message lists that are not the list [pancake], and Alice will provide the correct MACs for each of the message lists.

The first message list that Mallory queries for is:

Solution: Many alternative solutions exist for this subpart, but most should follow a similar idea to the following.

[pancake, waffle]

"waffle" can be replaced with any arbitrary message in this solution.

Alice reports that the MAC of the message list in the box above is T_1.

The second message list that Mallory queries for is:

Solution: [waffle]

Alice reports that the MAC of the message list in the box above is T_2.

Now, Mallory can compute that the MAC of the message list [pancake] is:

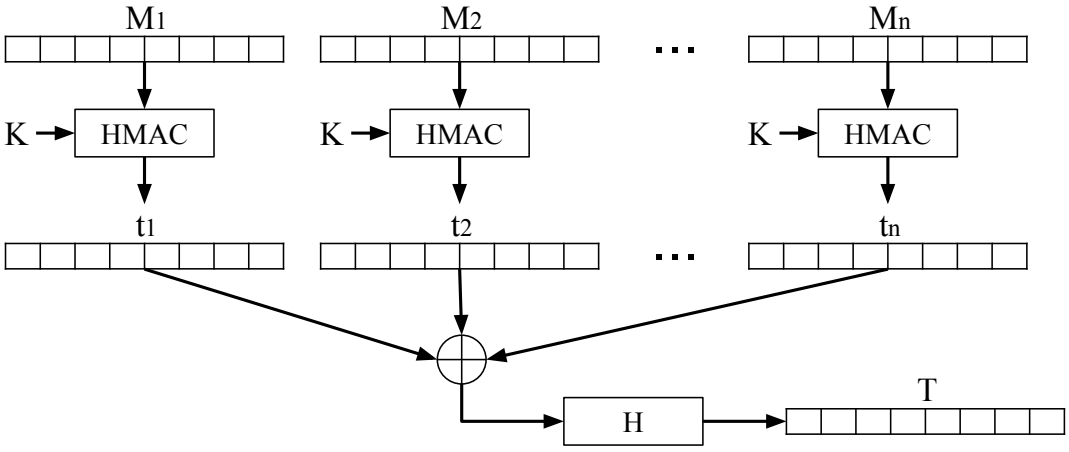
Solution: $T_1 \oplus T_2$

$T_1 = \mathsf{HMAC}(K, \text{pancake}) \oplus \mathsf{HMAC}(K, \text{waffle})$

$T_2 = \mathsf{HMAC}(K, \text{waffle})$

If we XOR these two tags together, then the $\mathsf{HMAC}(K, \text{waffle})$ cancels, leaving us with just $\mathsf{HMAC}(K, \text{pancake})$, as desired.

In the next two subparts, Alice modifies her scheme by adding an extra hashing step at the end:



1. Compute HMACs on each individual message. $t_i = \mathsf{HMAC}(K, M_i)$, for $1 \leq i \leq n$.
2. XOR all the HMAC outputs (t_i) together, **and hash the result**, to get the final MAC output. $T = \mathsf{H}(t_1 \oplus t_2 \oplus \ldots \oplus t_n)$.

Q6.5 (2 points) Using this new scheme, Alice computes the MAC for the message list $[M_1, \ldots, M_n]$. She sends the message list and the MAC to Bob.

Bob adds a new message M_{n+1} to the list, and wants to compute the MAC of the new message list $[M_1, \ldots, M_n, M_{n+1}]$.

What is the minimum number of HMACs that Bob needs to compute in order to compute the MAC of the new message list?

○ (A) 0 ○ (B) 1 ○ (C) 2 ○ (D) $n/2$ ○ (E) n ● (F) $n+1$

Solution: Given T, the output of a hash, there is no way for Bob to learn what the input to the hash was (the XOR of the $t_1, \ldots, t_n$). Therefore, Bob would have to recompute $t_1, \ldots, t_n$ from scratch, and also compute the new t_{n+1}, for a total of $n + 1$ HMAC computations.

Q6.6 (2 points) Does the attack in Q6.3 still work with this new scheme?

○ (A) Yes, with no modifications.

○ (B) Yes, if we apply H to the MAC produced by the attack.

○ (C) No, because Mallory cannot compute the hash without knowing K.

● (D) No, because the hash function is one-way.

Solution: Mallory knows the two message lists and MACs:

$$T_A = H(\mathsf{HMAC}(K, A_1) \oplus \ldots \oplus \mathsf{HMAC}(K, A_n))$$

$$T_B = H(\mathsf{HMAC}(K, B_1) \oplus \ldots \oplus \mathsf{HMAC}(K, B_n))$$

If we try to XOR these two tags together, we would get:

$$H(\mathsf{HMAC}(K, A_1) \oplus \ldots \oplus \mathsf{HMAC}(K, A_n)) \oplus H(\mathsf{HMAC}(K, B_1) \oplus \ldots \oplus \mathsf{HMAC}(K, B_n))$$

Which is not the same as the tag on the combined message list:

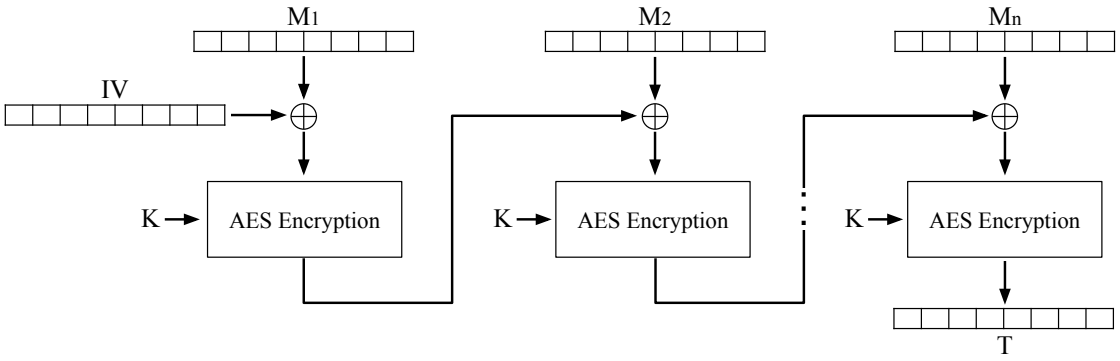
$$H(\mathsf{HMAC}(K, A_1) \oplus \ldots \oplus \mathsf{HMAC}(K, A_n) \oplus \mathsf{HMAC}(K, B_1) \oplus \ldots \oplus \mathsf{HMAC}(K, B_n))$$

Option (B) is incorrect because even if we hashed the XOR of the two tags, we would still not get the correct tag on the combined message list.

In order to get from the XORed tags to the tag of the combined message, we would somehow have to invert the hashes to recover the XOR of the HMACs, $\mathsf{HMAC}(K, A_1) \oplus \ldots \oplus \mathsf{HMAC}(K, A_n)$, but this is not possible because the hash function is one-way.

Option (C) is false because Mallory can compute the hash without knowing K.

For the rest of the question, consider this scheme for computing a single MAC on a list of n messages $M_1, M_2, \ldots, M_n$. For the rest of the question, you may assume each message is exactly one block long.



$$T = E_K(M_n \oplus E_K(\ldots M_2 \oplus E_K(M_1 \oplus \text{IV})))$$

The final MAC output is (T, IV).

Q6.7 (2 points) Select all true statements about the scheme above.

☐ (A) Given the list $[M_1, \ldots, M_n]$ and its MAC, it is possible to compute the MAC of list $[M_1, \ldots, M_n, M_{n+1}]$ without knowing K.

☐ (B) The MAC of list $[M_1, M_2, M_3]$ is equal to the MAC of list $[M_3, M_2, M_1]$.

■ (C) None of the above.

Solution:

Option (A) is false. As seen in Q6.9, if we want to compute the MAC of the list with an extra message, we need to compute an AES encryption E_K, but we can't do that unless we know K.

Option (B) is false. Each AES encryption scrambles the input unpredictably, so the order that we chain the inputs into the AES encryption blocks affects the final output value.

Q6.8 (2 points) Suppose that you know the MAC of list $[M_1, M_2, M_3]$ and the MAC of list $[M_4, M_5, M_6]$. You want to compute the MAC of the merged list $[M_1, M_2, \ldots, M_6]$. Select all true statements below.

☐ (A) If you know the individual messages $M_1, M_2, \ldots, M_6$, you can compute the merged MAC without knowing K.

☐ (B) If you know K, you can compute the merged MAC without knowing the individual messages.

■ (C) None of the above.

Solution: In order to chain the messages together, we would need to perform additional computation to account for the fact that the two individual MACs have two different IVs, which requires knowing both K and the individual messages.

Q6.9 (2 points) You receive the MAC (T, IV) of the message list $[M_1, M_2, M_3]$. You want to compute a MAC (T', IV) on the new message list $[M_1, M_2, M_3, M_4]$. Select the correct expression for T'.

○ (A) $T' = E_K(T) \oplus M_4$

● (C) $T' = E_K(T \oplus M_4)$

○ (B) $T' = D_K(T) \oplus M_4$

○ (D) $T' = D_K(T \oplus M_4)$

Solution: Looking at the diagram, we need to take the latest AES output, namely T, and chain it forward to the next message. We take the next message M_4, XOR it with T, and compute the AES encryption on the result.

This could also be derived from the equation:

$$T = E_K(M_3 \oplus E_K(M_2 \oplus E_K(M_1 \oplus IV)))$$

The expression we want is

$$E_K(M_4 \oplus E_K(M_3 \oplus E_K(M_2 \oplus E_K(M_1 \oplus IV))))$$

We can substitute in T to get

$$E_K(M_4 \oplus T)$$

as desired.

Q7 *Public-Key Cryptography: Does EvanBot Snore?* **(17 points)**

EvanBot decides to make a new authentication scheme that works over insecure connections.

Assume all parties have agreed on generator g for prime modulus p in advance.

To create a new account:

1. The user generates a random number a as their password and sends $A = g^a \bmod p$.
2. The server stores a mapping between the user and their corresponding A.

To login:

1. The user chooses a random $r \bmod p$ and sends $R = g^r \bmod p$ to the server.
2. The server chooses a random $b \bmod p$ and sends $b \bmod p$ to the user.
3. The user sends $z = r + ab \pmod{p-1}$ to the server.
4. The server checks if [ANSWER TO Q7.1].

Q7.1 (2 points) Fill in the missing value in the last step.

○ (A) $g^b \equiv A \cdot R^z \bmod p$

○ (C) $g^z \equiv R^b \cdot A \bmod p$

○ (B) $g^A \equiv R \cdot b \cdot A \bmod p$

● (D) $g^z \equiv R \cdot A^b \bmod p$

Solution: Note: All work shown is $\bmod p$.

$$
\begin{aligned}
g^z &\equiv g^{r+ab} && \text{definition of } z \\
&\equiv g^r \cdot g^{ab} && \text{exponent rule} \\
&\equiv R \cdot g^{ab} && \text{definition of } R \\
&\equiv R \cdot A^b && \text{definition of } A
\end{aligned}
$$

Q7.2 (3 points) Select all true statements.

■ (A) $r + ab \pmod{p-1}$ hides the value of a, even though the server knows b.

■ (B) $R = g^r \bmod p$ hides the value of r.

☐ (C) r being chosen randomly prevents replay attacks.

☐ (D) None of the above.

Solution:
Option (A) is correct, since r is randomly derived by the user and unknown to the server, $r + ab \pmod{p-1}$ is effectively an additive one-time pad $\pmod{p-1}$.

Option (B) is correct, since the discrete logarithm problem states that it is hard to recover r from $g^r \bmod p$.

Option (C) is incorrect because the server doesn't check or care that R is different between logins. Randomly choosing R is primarily to provide confidentiality for the user against the server – if R is reused then the one-time pad protection from Option (A) is compromised.

Replay attacks are prevented by the server sending a random b in Step 2 – see Q7.5 for more details on why this is the case.

Q7.3 (2 points) What happens if r is leaked to an eavesdropping attacker that has seen the full login process?

● (A) The attacker can use r to recover the password a.

○ (B) The attacker can tell when two different logins use the same password.

○ (C) The attacker can use r to recover b.

○ (D) The attacker is able to solve the discrete log problem on logins from different users.

Solution: The attacker knows r because it is leaked.

The attacker knows b because it was sent by the server.

The attacker knows z because it was sent by the user.

From the question, we know that $z = r + ab \pmod{p-1}$. The attacker knows every value except a, so the attacker can solve this equation and learn a.

Option (B) is false. First, note that R and b don't leak any information about the password, because they're randomly-derived values. Then, note that $z = r + ab$ involves the random values r and b, so even if two logins use the same password a, the z values that the attacker sees will be different.

Option (C) is false. b is sent over the channel, and it is derived separately from r, so there is no need to use r to recover b.

Option (D) is false. Solving the discrete log problem is infeasible in general. Even if we knew the value of r for a login, the values of r and R will be different in future logins, so the leaked value won't help us solve the discrete log problem.

For the rest of this question, consider this modified, non-interactive authentication scheme.

1. The user chooses a random $r \bmod p$ and sets $R = g^r \bmod p$.
2. The user sets $b = H(R)$.
3. The user sends R and $z = r + ab \pmod{p-1}$.
4. The server derives $b = H(R)$.
5. The server checks if [ANSWER TO Q7.1].

Q7.4 (3 points) Which option best explains why this scheme convinces the server that the user knows a, despite b not being a server-provided random input?

- ○ (A) Sending $g^r \mod p$ forces the user to stick with the same r throughout the protocol.
- ○ (B) The hash function is collision resistant, making it hard to find a different value that hashes to the same b.
- ● (C) An attacker cannot predict the value of b due to the unpredictability of the hash function.
- ○ (D) The server can check that $b = H(R)$ because the hash function is publicly known.

Solution: The original (interactive) scheme has a random input b from the server to prevent trivial forgery attacks (see Q7.6), forcing the attacker to "commit" to their R by sending it to the server *before* they receive b. If the user can predict what this b value will be **or** change their R after they receive/derive b, they can execute the attack in Q7.6.

Therefore, we need a solution that prevents the user from predicting b or changing their R afterward. Setting $b = H(R)$ accomplishes both: we can't change R because the server uses the "old" value to derive b – changing R to something else would cause the verification to fail. Moreover, b is unpredictable to the client even though they control R, they can't predict the value of $H(R)$ until they evaluate it (even though they can evaluate it at any time). The key here is that we would like to choose an R such that $R \equiv (A^{H(R)})^{-1} \mod p$ to execute the Q7.6 attack, but since we can't find $H(R)$ until we choose an R we are stuck.

The other options are technically true but not the primary reason outlined above.

Q7.5 (3 points) Is this scheme secure against replay attacks?

- ○ (A) Yes, because setting $b = H(R)$ effectively replaces the server's random input.
- ○ (B) Yes, because any change to r will cause $b = H(R)$ to have a wildly different output.
- ● (C) No, because eavesdroppers can record R and z from the login transcript and resend R, z on their own.
- ○ (D) No, because the value of R is the same between authentications made by the same user.

Solution: Since this scheme is completely determined by the user's actions and the server only sees the "final" result (R, z), there is nothing stopping an eavesdropper from simply copying that and resending it themselves.

Q7.6 (4 points) Consider a modified scheme where Step 2 and Step 4 set $b = H(A)$ instead of $b = H(R)$. Provide values for R and z that will validate for any given A.

R:

Solution: $R = (A^b)^{-1} \bmod p$

z:

Solution: $z = 0$

Solution: Other solutions exist, as long as the equation $g^z \equiv R \cdot A^b \mod p$ verifies given their values.

This verifies correctly, using the verification protocol described in Q7.1:

$$\begin{aligned} g^z &\stackrel{?}{\equiv} R \cdot A^b \bmod p \\ g^0 &\equiv (A^b)^{-1} \cdot A^b \bmod p \\ g^0 &\equiv 1 \bmod p \\ 1 &\equiv 1 \bmod p \end{aligned}$$

Note that $(A^b)^{-1} \bmod p$ is the multiplicative inverse of $A^b \bmod p$. We know the multiplicative inverse is guaranteed to exist because p is prime.

Q7.7 (0 points) **This A+ question is not worth points. It can only affect your course grade if you have a high A and might receive an A+, however, there are other ways to earn an A+ if you are not able to complete this question. We strongly recommend completing the rest of the exam first. Ask your proctor for additional paper if you need more space to write.**

Design a secure digital signature scheme based on the security of the non-interactive authentication scheme used from Q7.4 onwards. **Your scheme should not significantly deviate from the non-interactive authentication protocol.**

Describe an algorithm to sign a message. The input will be a secret key sk (such that $pk = g^{sk} \bmod p$ for publicly-known g, p) and a message M.

HINT: Use the steps from the non-interactive authentication protocol as a starting point.

Solution: Recall that $sk = a$ and $pk = A = g^a \bmod p$ using notation from the earlier subparts.

We can proceed as usual in Step 1 by generating a random $r \bmod p$ and deriving $R = g^r \bmod p$.

For Step 2, we instead set $b = H(M \| R)$ instead of $b = H(R)$.

For Step 3, we set $z = r + ab$ as usual and output $S = (R, z)$.

Describe an algorithm to verify the signature produced in your given signing algorithm. The input will be a public key pk, signature S, and message M.

Solution: Step 1: Derive $b = H(M \| R)$

Step 2: Verify that $g^z \equiv R \cdot A^b$

Explain why your algorithm is a secure digital signature scheme, assuming the non-interactive protocol used from Q7.4 onwards is secure.

*HINT: You don't need to write a formal proof, but you should explain **precisely** why the changes you made to turn the authentication scheme into a signature scheme are **necessary** and **secure**.*

Solution: The original scheme is secure because setting $b = H(R)$ effectively produces a random value that the user cannot predict ahead of time. The user is also effectively committing to use the value of R. These two properties allow us to replace the interactive steps of sending R to the server (commit) and receiving b (random server input) into one user-executed step.

In that case, however, we are simply proving that we know the private key. A signature scheme requires showing that we know the private key **and commit to a specific message**. Therefore, we have to include the message as an argument to the hash function $b = H(M, R)$. If we didn't, an attacker could change M after we derive b to perform an existential forgery attack. Since b is now a random function of R and M, any party that can output a valid (R, z) must have knowledge of sk and actively decided to sign M.

It turns out that it's possible to turn any interactive proof of knowledge protocol into a signature scheme via the Fiat-Shamir transformation!

Nothing on this page will affect your grade.

Post-Exam Activity: Pumpkin

It's spooky season! How will you carve this huge pumpkin?

Comment Box

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