CS 161 Fall 2024 — Introduction to Computer Security — Midterm

Solutions last updated: October 15th, 2024

Name: _____

Student ID: _____

Question:	1	2	3	4
Points:	0	8	20	19

Question:	5	6	7	Total
Points:	16	19	18	100

This exam is 110 minutes long. For questions with **circular bubbles**, select only one choice.

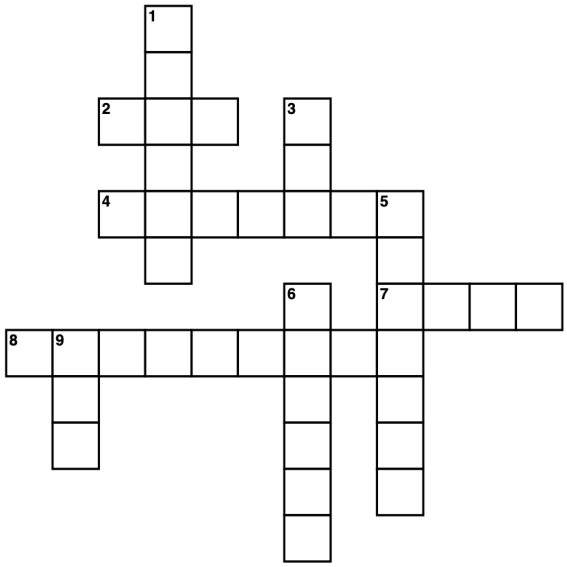
- ○ Unselected option (completely unfilled)
- ● Only one selected option (completely filled)
- ◉ Don't do this (it will be graded as incorrect)

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 may grade the worst interpretation.

Pre-exam activity (0 points):



Across:
(2) Block cipher previously known as Rijndael
(4) EvanBot's favorite food (singular)
(7) Randomizes addresses for each program execution
(8) $x \neq y$, $H(x) = H(y)$

Down:
(1) Hard to find x given $H(x)$
(3) Created by Rivest, Shamir, and Adleman
(5) The CS 161 mascot
(6) Used to encrypt and decrypt messages
(9) Perfectly-secure encryption

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* **(8 points)**

Each true/false is worth 1 point.

Q2.1 The employee entrance to the Caltopian Post Office is protected by a 20-digit keypad code that changes daily.

TRUE or FALSE: This violates Consider Human Factors.

● (A) TRUE ○ (B) FALSE

Solution: True. It's unreasonable to expect people to memorize a 20-digit code.

Q2.2 Caltopian Army counterintelligence installs devices on all secure networks to look for abnormally high levels of outgoing data.

TRUE or FALSE: This is an example of Detect If You Can't Prevent.

● (A) TRUE ○ (B) FALSE

Solution: True, since even though we didn't prevent the sensitive data being exfiltrated to the adversary, we would detect something is wrong.

Suppose we have a little-endian C program with a local variable `char easter_egg[32]`. Consider the following possible GDB output after running the command `x/8wx easter_egg`:

```
0xfffff000: 0x486F6D65 0x20706167 0x6520736F 0x75726365
0xfffff010: 0x20436865 0x636B2047 0x69744875 0x20707232
```

Q2.3 TRUE or FALSE: The last four bytes of `easter_egg` equal the word `0x20707232`.

● (A) TRUE ○ (B) FALSE

Solution: True.

Q2.4 TRUE or FALSE: The stack is always marked non-executable when non-executable pages is enabled.

● (A) TRUE ○ (B) FALSE

Solution: True. All writable pages (i.e., all of the stack and the heap) are marked non-executable.

Q2.5 True or False: When the ESP is moved up during function return, all memory that ends up below the ESP is set to zero.

○ (A) True ● (B) False

Solution: False. The stack pointer (ESP) is updated, but memory is not modified. Therefore, the memory below the ESP contains old stale data that is (normally) ignored by the program.

Q2.6 True or False: One-time pads require both a random key and a random IV to be secure.

○ (A) True ● (B) False

Solution: False. They don't use a IV. No IV is needed, since a one-time pad is designed to be used to encrypt only a single message.

Q2.7 True or False: Small changes to the input of a hash function usually result in only minor changes to its output.

○ (A) True ● (B) False

Solution: False

Q2.8 True or False: The generator and modulus (g, p) in Diffie-Hellman must not be reused between key exchanges.

○ (A) True ● (B) False

Solution: False. It is safe for everyone to use the same g, p value.

Q3 *EvanBot's Folly: Memory Safety* **(20 points)**

EvanBot accidentally deleted the C library files responsible for `fgets`! They've taken it upon themselves to code a replacement:

```
1  void efgets(char* ptr, size_t maxlen) {
2      int c, i;
3
4      for (i = 0; i < maxlen; i++) {
5          c = fgetc(stdin); // Get one character from stdin
6          if (c == EOF)
7              break;
8
9          ptr[i] = c;
10     }
11     ptr[maxlen] = '\0';
12 }
```

Assumptions:

- All memory safety defenses are disabled.
- There is no compiler padding.
- You may use `SHELLCODE` as a 16-byte shellcode.
- `efgets` can be called like a C standard library function (similar to `gets`, etc).

EvanBot creates a sample program to demonstrate the use of `efgets`:

```
1  int main() {
2      char buf[16];
3      efgets(buf, 24);
4      return 0;
5  }
```

Q3.1 (2 points) Assume you have run this sample program using GDB and paused on **Line 3** of `main` (before `efgets` is called). You run the `info frame` command and see the following output (some lines excluded for brevity):

```
eip = 0x0100adb0 in main (sample.c:3); saved eip = 0x0100adfc
...
Saved registers:
ebp at 0xffffb550, eip at 0xffffb554
```

What is the address where **RIP of `main`** is stored on the stack?

○ (A) `0x0100adb0`	○ (C) `0xffffb550`	○ (E) `0xffffb558`
○ (B) `0x0100adfc`	● (D) `0xffffb554`	○ (F) `0xffffb55c`

The `efgets` function is reproduced for your convenience:

```
1  void efgets(char* ptr, size_t maxlen) {
2      int c, i;
3
4      for (i = 0; i < maxlen; i++) {
5          c = fgetc(stdin); // Get one character from stdin
6          if (c == EOF)
7              break;
8
9          ptr[i] = c;
10     }
11     ptr[maxlen] = '\0';
12 }
```

Q3.2 (3 points) You now run this program on a second machine where the RIP of `main` is stored at a different address, specifically, at address `0xffffff18`. (Ignore the GDB output in Q3.1 from here on. Don't forget that `SHELLCODE` is 16 bytes long.)

Select the correct option for an input to `stdin` that causes the program to execute `SHELLCODE`.

- ○ (A) `SHELLCODE + 'A'*4 + '\x18\xff\xff\xff'`
- ○ (B) `SHELLCODE + '\x18\xff\xff\xff'`
- ○ (C) `'A'*20 + SHELLCODE`
- ● (D) `SHELLCODE + 'A'*4 + '\x04\xff\xff\xff'`

Solution: With this input, `efgets` will write `SHELLCODE` into `buf`, then write `'A'*4` into the next 4 bytes, i.e., the SFP for `main`, then write `0xFFFFFF04` (recall that little-endian stores the least significant byte first) into the next 4 bytes, i.e., the RIP for `main`. When `main` returns, the CPU will branch to the value stored in the RIP for `main`, i.e., to address `0xFFFFFF04`. If we draw out the stack diagram, we find that `buf` starts 20 bytes before the address where the RIP for `main` is stored, i.e., `buf` starts at address `0xFFFFFF04`. Therefore, when `main` returns, the CPU branches and starts running the code stored in `buf` — which now contains `SHELLCODE`.

Q3.3 (2 points) Assuming that the correct exploit was given in the previous subpart, when will `SHELLCODE` be executed?

- ○ (A) Immediately after `fgetc` returns
- ○ (B) Immediately after `efgets` returns
- ○ (C) During the execution of `efgets`
- ● (D) Immediately after `main` returns

Solution: See solution for Q3.2.

Q3.4 (2 points) Which of the following changes would prevent the correct exploit from Q3.2 (without modifications) from executing `SHELLCODE`? Select all that apply.

- ■ (A) Enabling stack canaries
- ■ (B) Enabling non-executable pages
- ☐ (C) Removing the `EOF` check on Line 7
- ☐ (D) None of the above

Solution: `efgets` writes contiguously, so stack canaries would stop this attack: the canary would get overwritten, so before `main` returns, the canary-check would detect this and terminate without returning. Non-executable pages would prevent executing `SHELLCODE` from `buf`. The EOF check won't change anything because this input is already 24 bytes long, and `maxlen` = 24.

Impressed with your previous success, EvanBot decides to issue you a new challenge program **with ASLR enabled**!

```
1 void run_test() {
2     char buf[16];
3     efgets(buf, 16);
4 }
5
6 int main() {
7     run_test();
8     return 0;
9 }
```

Assume that EvanBot's program is the only program that modifies stack memory in this system, including memory that ends up below the ESP.

Stack at Line 3 of `efgets`

RIP of `main`
(1)
RIP of `run_test`
SFP of `run_test`
(2)
`maxlen`
(3)
RIP of `efgets`
SFP of `efgets`
`c`
`i`

Q3.5 (2 points) Fill in the stack diagram, assuming the program is paused on the third line of `efgets`.

- ● (A) (1) SFP of main (2) `buf` (3) `ptr`
- ○ (B) (1) SFP of main (2) `buf` (3) `c`
- ○ (C) (1) `buf` (2) `c` (3) `ptr`
- ○ (D) (1) `buf` (2) `ptr` (3) `c`

Q3.6 (2 points) Which vulnerability is present in the code?

- ● (A) Off-by-one
- ○ (B) Signed/unsigned vulnerability
- ○ (C) ret2ret
- ○ (D) Return-oriented programming

Solution: The vulnerability is that `efgets` writes the `'\0'` byte one after the end of `buf`.

Q3.7 (7 points) [Warning: This part is hard; consider coming back to it if you are stuck.]

Give an input to `stdin` that would cause `SHELLCODE` to be executed by the new program with probability $\geq 1/256$.

Solution: `SHELLCODE`

The attack will be successful (i.e., `SHELLCODE` will be executed) if the least significant byte of the **value initially stored in** SFP of `run_test` is equal to which of the following?

- ○ (A) `0x00`
- ○ (B) `0x04`
- ○ (C) `0x12`
- ○ (D) `0x20`
- ● (E) `0x24`
- ○ (F) `0x32`
- ○ (G) `0x36`
- ○ (H) `0x64`

Solution: Suppose that the address for the SFP of `main` is 0x0124. Then the value initially stored in the SFP of `run_test` is 0x0124 (since it points to the SFP of its caller). Drawing out the stack frame and working out the address of all of the other entries on the stack, we find that it looks like this when `efgets` starts executing:

Address	Entry	Value
0x0128	RIP of `main`	
0x0124	SFP of `main`	
0x0120	RIP of `run_test`	
0x011C	SFP of `run_test`	0x0124
0x010C	`buf`	
0x0108	`maxlen`	0x0010
0x0104	`ptr`	0x010C
0x0100	RIP of `efgets`	
0x00FC	SFP of `efgets`	0x011C
0x00F8	`c`	0x0000
0x00F4	`i`	0x0000

When line 11 of `efgets` executes, it writes `'\0'` one past the end of `buf`, i.e., to address 0x011C, i.e., to the first byte (least significant byte) of the SFP of `run_test`. Since that word on the stack previously contained the value 0x0124, it now contains the value 0x0100. So after `efgets` returns and `run_test` restores `%esp`, the call stack looks like this:

Address	Entry	Value
0x0128	RIP of `main`	
0x0124	SFP of `main`	
0x0120	RIP of `run_test`	
0x011C	SFP of `run_test`	0x0100
0x010C	`buf`	`SHELLCODE`
0x0108	`maxlen`	0x0010
0x0104	`ptr`	0x010C
0x0100	RIP of `efgets`	
0x00FC	SFP of `efgets`	0x011C
0x00F8	`c`	0x0000
0x00F4	`i`	0x0000

At this point `%esp` has the value 0x0104, so values below that on the stack are stale, but they remain in memory (they are not erased or overwritten with zeros; see Q2.5). Next `run_test` executes `mov %ebp, %esp` and `pop %ebp` from the epilogue, restoring `%esp` and setting `%ebp` to the value 0xFF00. At this point the value of `%ebp` has been corrupted by the off-by-one vulnerability. Next `run_test` executes `ret` and returns back to `main`.

Next `main` executes `mov %ebp, %esp` and `pop %ebp` from the epilogue. The first instruction sets `%esp` to 0x0100, so now `%esp` has been corrupted, and all subsequent use of the stack will be using the wrong stack pointer. The `pop %ebp` instruction now works relative to the current (corrupted) value of `%esp`: thus it reads the 4 bytes at address 0x0100, stores them into `%ebp`, and adds 4 to `%esp`. Afterwards `%esp` contains the value 0x0104. Notice that address 0x0104 corresponds to the place where `ptr` was stored on the stack, and the value stored there is the address of `buf`, i.e., 0x010C. Finally, main executes the `ret` instruction to return. The `ret` instruction looks at the address given by `%esp` (which, as a reminder, has been corrupted), so it looks at address 0x0104, reads the value stored there (namely, 0x010C), and the CPU starts executing code at that location, i.e., at address 0x010C. But looking at our chart above, we see that address 0x010C corresponds to the address of `buf`, so the CPU starts executing instructions from inside `buf`. Thanks to our choice of input, `buf` contains `SHELLCODE`, so the CPU starts executing the instructions of `SHELLCODE`.

You can see that this attack relied on the initial value stored in the SFP of `run_test` to have least significant byte 0x24, but the more significant bytes (e.g., 0x01 in this example) don't matter — all that matters is its least significant byte.

Intuitively, if the initial value of SFP of `run_test` starts with 0x24, then when it is overwritten with `'\0'` (i.e., 0x00), the value stored there will be 0x24 smaller. That means that by the time we return twice, the instructions of `main` are looking for the stack frame in the wrong place: they are looking 0x24 bytes lower than they should be. At 0x24 bytes lower than the RIP of `main`, we find

`ptr`, and the value stored in `ptr` is (conveniently) the address of `buf`, i.e., the address `SHELLCODE`. In other words, when `main` returns, because it is looking in the wrong place for the return address, instead of returning to the address in RIP of `main`, it'll return to the address of `SHELLCODE`.

Q4 *Copycat - Memory Safety* (19 points)

```
1  void foo(uint8_t offset, char* buf_ptr) {
2      int x = 0x11223344;
3
4      memcpy(buf_ptr + offset, &x, 8);
5  }
6
7  int main() {
8      char buf[132];
9      int8_t offset = 0;
10
11     gets(buf);
12     fread(offset, 1, 1, stdin);
13
14     if (offset > 20)
15         return 0;
16
17     foo(offset, buf);
18
19     return 0;
20 }
```

Stack at Line 4

RIP of `main`
SFP of `main`
Canary
`buf`
`offset`
(1)
(2)
RIP of `foo`
SFP of `foo`
(3)
`x`

Assumptions:

- Stack canaries are enabled, but all other memory safety defenses are disabled.
- There is no compiler padding.
- There is shellcode already stored at address `0xDEADBEEF`.
- `uint8_t` and `int8_t` are the C types for an 8-bit unsigned/signed integer, respectively.

Q4.1 (2 points) What values go in blanks (1) through (3) in the stack diagram above?

○ (A) (1) Canary	(2) `offset`	(3) `buf_ptr`
○ (B) (1) Canary	(2) `buf_ptr`	(3) `buf_ptr`
○ (C) (1) `offset`	(2) `buf_ptr`	(3) Canary
● (D) (1) `buf_ptr`	(2) `offset`	(3) Canary

Q4.2 (2 points) Which vulnerability is present in the code?

- ○ (A) Off-by-one
- ● (B) Signed/unsigned vulnerability
- ○ (C) ret2ret
- ○ (D) Out-of-bounds read

Solution: The vulnerability is that `offset` is declared as a signed int in `main`, but as an unsigned int in `foo`. In particular, if `offset` is negative in `main`, then the if-statement on line 14 will be false, so `foo` will be called; but then when the negative value is cast to an unsigned int (`uint8_t`), it becomes a large-ish positive value, causing the `memcpy` to write after the end of the buffer.

Q4.3 (1 point) True or False: The value of the stack canary is the same for each different function frame in the same program execution.

- ● (A) True
- ○ (B) False

In the next two subparts, provide inputs that would cause the program to execute `SHELLCODE`.

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

Q4.4 (3 points) Input to `gets` at Line 11:

Solution: `'A'*140 + '\xEF\xBE\xAD\xDE' + '\n'`

Q4.5 (5 points) Input to `fread` at Line 12 (**in hexadecimal**):

Solution: `'\x80'`

Solution: The call to `gets` will overwrite the canary with `'A'*4`, overwrite the SFP of `main` with `'A'*4`, and then overwrite the RIP of `main` with `'\xEF\xBE\xAD\xDE'`, i.e., with `0xDEADBEEF`. This looks promising, but the canary for `main` has been corrupted.

Next, `offset` will get set to `0x80`; considered as a `int8_t`, this is a negative number (-127), so the if-statement follows the else branch. Finally, `foo` is called, and now `offset` is cast to `uint8_t`, so it is considered as the large positive number `0x80`.

The call to `memcpy` writes `0x11223344` to `buf + 0x80`, i.e., to the last 4 bytes of `buf`, then it writes the 4 bytes above `x` to `buf + 0x84`. The 4 bytes above `x` are the canary for `foo`, and the address `buf + 0x84` refers to the address of the canary for `main`. Therefore, the `memcpy` copies the canary for `foo` (which is uncorrupted) over the canary for `main` (which was previously corrupted, but is now restored thanks to this step). This restores the canary to `main` to its correct value.

Finally, once `main` returns, the CPU will jump to the address `0xDEADBEEF` (since this was stored over the RIP for `main` by `gets`), and the stack corruption will not be detected by the canary (since the canary was restored to its correct value).

Q4.6 (3 points) Which of the following modifications would prevent this exploit (without any modifications) from working? Select all that apply.

- ☐ (A) Generating the canary such that its least-significant byte is a null terminator.
- ■ (B) Changing the type of `offset` on Line 9 to `uint8_t`.
- ☐ (C) Changing the condition on Line 14 to `offset > 128`.
- ☐ (D) None of the above

Solution: As described in the solution for 4.5, we are using `memcpy` to copy the canary for `foo` into the canary slot for `main`.

`memcpy` does not stop copying on a null terminator, so changing the canary to have a null terminator does not prevent the exploit.

If we change `offset` type in `main` to be unsigned `uint8_t`, then we can no longer do our signed/unsigned exploit and can't copy into `buf + 128` as required.

Changing the condition to `offset > 128` does not affect the exploit, as our `0x80` input still passes the check.

Q4.7 (3 points) Select all values for the size of `buf` such that it is still possible to exploit this code, assuming you are able to pick new inputs.

- ■ (A) 4
- ■ (B) 20
- ■ (C) 24
- □ (D) 64
- □ (E) 128
- ■ (F) 256

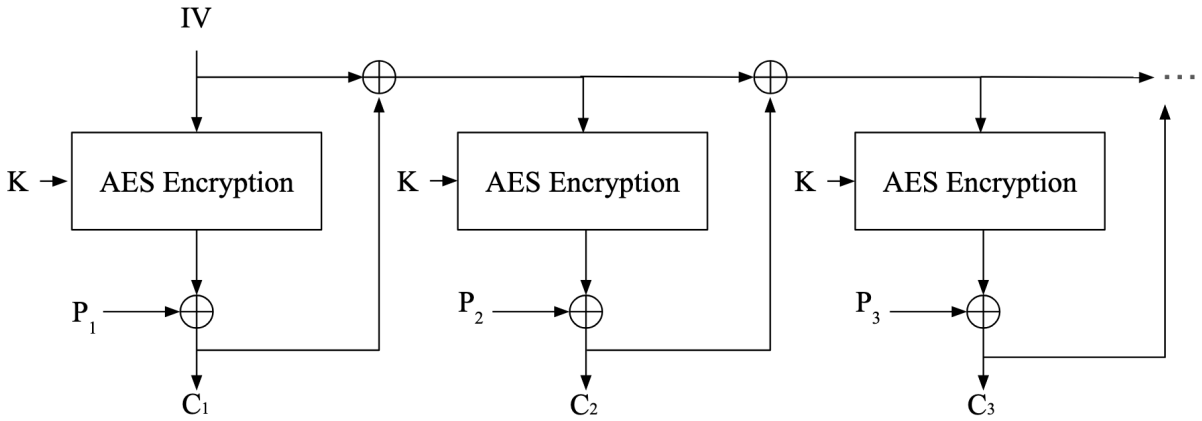
Solution: We require that `offset` be able to go exactly 4 bytes before the end of the buffer. Since $0 \leq$ `offset` ≤ 20 and $128 \leq$ `offset` ≤ 255 by the size check, we have the $4 \leq$ `len(buf)` ≤ 24 and $132 \leq$ `len(buf)` ≤ 259.

Q5 *AES-RFM — Symmetric Cryptography* **(16 points)**

EvanBot invents a new block cipher mode of operation: AES Repeated Feedback Mode.

The encryption formulas for AES-RFM are as follows:

$$C_1 = E_K(IV) \oplus P_1$$
$$C_i = P_i \oplus E_K(IV \oplus C_1 \oplus \cdots \oplus C_{i-1})$$



Q5.1 (2 points) Select the correct decryption formula for the i-th ($i \geq 2$) plaintext block in AES-RFM.

- ● (A) $P_i = C_i \oplus E_K(IV \oplus C_1 \oplus \cdots \oplus C_{i-1})$
- ○ (B) $P_i = C_{i-1} \oplus E_K(IV \oplus C_1 \oplus \cdots \oplus C_{i-1})$
- ○ (C) $P_i = D_K(C_i) \oplus IV \oplus C_1 \oplus \cdots \oplus C_{i-1}$
- ○ (D) $P_i = C_i \oplus D_K(IV \oplus C_1 \oplus \cdots \oplus C_{i-1})$

Solution: Starting from $C_i = P_i \oplus E_K(IV \oplus C_1 \oplus \cdots \oplus C_{i-1})$, we XOR $E_K(IV \oplus C_1 \oplus \cdots \oplus C_{i-1})$ on both sides to isolate $P_i = C_i \oplus E_K(IV \oplus C_1 \oplus \cdots \oplus C_{i-1})$.

Q5.2 (3 points) Alice has a 4-block message (P_1, P_2, P_3, P_4). She encrypts this message with AES-RFM and obtains the ciphertext $C = (IV, C_1, C_2, C_3, C_4)$, which she then sends to Bob.

During transit, network errors flip a single bit in C_1. That is, Bob receives the ciphertext $C' = (IV, C_1 \oplus 1, C_2, C_3, C_4)$.

What message will Bob compute when he decrypts the modified ciphertext C'?

G represents some unpredictable "garbage" output (individual G blocks do not necessarily have the same value).

- ○ (A) (G, G, G, G)
- ● (D) $(P_1 \oplus 1, G, G, G)$
- ○ (B) (G, P_2, P_3, P_4)
- ○ (E) $(P_1 \oplus 1, P_2, G, G)$
- ○ (C) (P_1, G, G, G)
- ○ (F) $(P_1, P_2 \oplus 1, G, G)$

Solution: Let's consider what happens when we try to decrypt $(IV, C_1 \oplus 1, C_2, C_3, C_4)$. First, P'_1:

$$\begin{aligned} P'_1 &= C'_1 \oplus E_K(IV') \\ &= (C_1 \oplus 1) \oplus E_K(IV) \\ &= 1 \oplus (C_1 \oplus E_K(IV)) \\ &= P_1 \oplus 1 \end{aligned}$$

Now for P'_2:

$$\begin{aligned} P'_2 &= C'_2 \oplus E_K(IV' \oplus C'_1) \\ &= C_2 \oplus E_K(IV \oplus C_1 \oplus 1) \end{aligned}$$

Since $IV \oplus C_1 \neq IV \oplus C_1 \oplus 1$, the value of $E_K(IV \oplus C_1 \oplus 1)$ is going to be very different from $E_K(IV \oplus C_1)$, making P'_2 effectively pseudorandom garbage. The same will happen with all other ciphertext changes, so all P'_i with $i > 1$ will be garbage.

Alice has a 3-block message (P_1, P_2, P_3). She encrypts this message with AES-RFM and obtains the ciphertext $C = (IV, C_1, C_2, C_3)$.

As Mallory, you will modify the ciphertext C in transit to C', and you wish to choose C' so it decrypts to $P' = (X, Y, 1)$ (**where X and Y can be any value, garbage or otherwise**). In other words, you want to ensure that the last block of P' will be 1, but you don't care what the first two blocks of P' turn out to be. You have access to the original message (P_1, P_2, P_3).

Q5.3 (6 points) Select values for the modified ciphertext $C' = (IV', C_1', C_2', C_3')$ such that Bob will decrypt C' to $P' = (X, Y, 1)$.

Each value below will be represented as the XOR of multiple variables. Select as many as you need. For example, if you want to set $IV' = P_1 \oplus C_2$, then bubble in P_1 and C_2.

IV' is equal to the XOR of:

☐ (A)	☐ (B)	☐ (C)	■ (D)	☐ (E)	☐ (F)	☐ (G)	☐ (H)
P_1	P_2	P_3	IV	C_1	C_2	C_3	1

C_1' is equal to the XOR of:

☐ (A)	☐ (B)	☐ (C)	☐ (D)	■ (E)	☐ (F)	☐ (G)	☐ (H)
P_1	P_2	P_3	IV	C_1	C_2	C_3	1

C_2' is equal to the XOR of:

☐ (A)	☐ (B)	☐ (C)	☐ (D)	☐ (E)	■ (F)	☐ (G)	☐ (H)
P_1	P_2	P_3	IV	C_1	C_2	C_3	1

C_3' is equal to XOR of:

☐ (A)	☐ (B)	■ (C)	☐ (D)	☐ (E)	☐ (F)	■ (G)	■ (H)
P_1	P_2	P_3	IV	C_1	C_2	C_3	1

Solution: Our core idea here is that the value of P_i' is determined by $C_i' \oplus E_K(IV' \oplus C_1' \oplus \cdots \oplus C_{i-1}')$. If we can keep the value of $E_K(IV \oplus C_1 \cdots \oplus C_{i-1})$ constant, then any change in C_i will be reflected in P_i'. For example, if we have IV through C_{i-1} be the same and set $C_i' = C_i \oplus 5$, then

$$\begin{aligned} P_i' &= (C_i \oplus 5) \oplus E_K(IV \oplus C_1 \cdots \oplus C_{i-1}) \\ &= 5 \oplus (C_i \oplus E_K(IV \oplus C_1 \cdots \oplus C_{i-1})) \\ &= P_i \oplus 5 \end{aligned}$$

Solution: Applying this to our concrete case, if we XOR C_3' with $P_3 \oplus 1$, that will cancel out the old value and replaces it with 1:

$$\begin{aligned} P_3' &= (C_3 \oplus P_3 \oplus 1) \oplus E_K(IV \oplus C_1 \oplus C_2) \\ &= 1 \oplus P_3 \oplus (C_3 \oplus E_K(IV \oplus C_1 \oplus C_2)) \\ &= 1 \oplus P_3 \oplus P_3 \\ &= 1 \end{aligned}$$

Q5.4 (5 points) Which values of P' can Mallory cause Bob to decrypt to, given that she can modify C and knows the original value of P? As in the previous subpart, X and Y represent values that Mallory doesn't need to control or predict and might be garbage.

Assume that none of the original P_i values were equal to 1.

■ (A) $(X, 1, Y)$ ☐ (C) $(P_1, 1, P_2)$ ■ (E) $(P_1, 1, X)$

■ (B) $(P_1, P_2, 1)$ ☐ (D) $(1, P_2, P_2)$ ☐ (F) $(1, 1, X)$

Solution: Intuitively, we can use the idea from the previous subpart to set any specific plaintext block to a value we want.

However, all plaintext blocks after that block will become garbage. Note that the plaintext blocks before this are unaffected, since we kept the ciphertext blocks before this the same.

For example, $C' = (IV, C_1, C_2 \oplus P_2 \oplus 1, G)$ decrypts to $P' = (P_1, 1, X)$ (G can be any value).

Q6 *To HMAC and Back — Cryptography* **(19 points)**

For each of the following subparts, indicate whether the given construction is an EU-CPA secure MAC.

Assume that the message M is variable length and does not require padding.

Q6.1 (2 points) $\mathsf{MAC}(K, M) = H(M) \oplus H(K)$.

○ (A) Secure ● (B) Insecure

Solution: An attacker can ask for the MAC on X, receive $H(X) \oplus H(K)$, and XOR with $H(X)$ to find $H(K)$. Then they can compute the MAC tag for every other message.

Q6.2 (2 points) $\mathsf{MAC}(K_1, K_2, M) = H(K_2 \| H(K_1 \| M))$.

● (A) Secure ○ (B) Insecure

Solution: This is the same as NMAC (a secure precursor of HMAC).

Q6.3 (2 points) $\mathsf{MAC}(K, M) = \mathsf{HMAC}(K, M) \| M$.

● (A) Secure ○ (B) Insecure

Solution: MACs don't require or promise confidentiality. We can treat this as HMAC if we just discard the last half. Predicting the MAC tag with this scheme is at least as hard as predicting the HMAC MAC tag (because an attacker must predict strictly more).

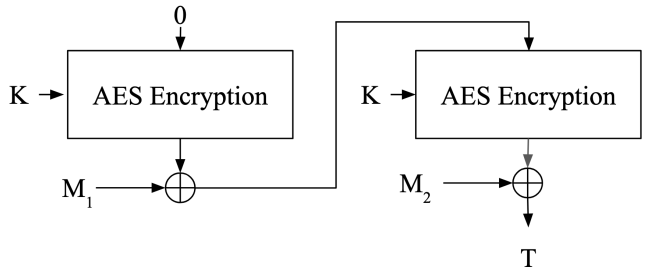
Q6.4 (2 points) $\mathsf{MAC}(K, M) = (IV, C_n)$ where C_n is the last block of the AES-CBC encryption of $H(M)$ under key K, and IV is the corresponding randomly-generated IV.

For this subpart only, assume the cryptographic hash function H has an output size of 128 bits.

○ (A) Secure ● (B) Insecure

Solution: An attacker can edit the IV. For instance, given a MAC tag (IV, C_n) on $P = (P_1, \ldots, P_n)$, the attacker can modify the message to $P' = (P'_1, P_2, \ldots, P_n)$ and compute a valid MAC tag on this message as $(IV \oplus P_1 \oplus P'_1, C_n)$. In this way, the attacker obtains a valid MAC tag on the new message P', which violates EU-CPA.

Q6.5 (5 points) $\mathsf{MAC}(K, M) = C_n$, where C_n is the last block of the encryption of M with AES-CFB under key K and $IV = 0$. For example, $\mathsf{MAC}(K, [M_1, M_2]) = M_2 \oplus E_K(M_1 \oplus E_K(0))$:



Given $M = (M_1, M_2)$ and its MAC T, provide a new **two-block** message $M' = (M'_1, M'_2)$ and its MAC T'. (You must not use $M' = M$.)

Provide a value for M':

Solution: $(M_1, M_2 \oplus T)$

Provide a value for T':

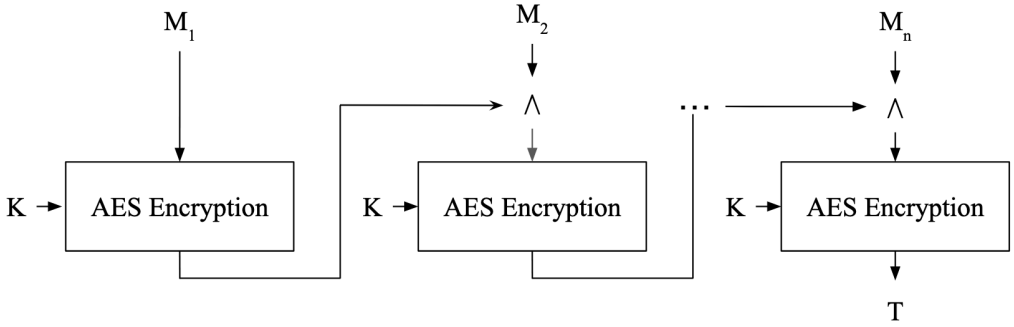
Solution: 0

Solution: This is one of many alternate solutions. The idea is that if we have the first block of M' (namely, M'_1) equal to the first block of M (namely, M_1), then the intermediate value $E_K(E_K(0) \oplus M'_1)$ will be the same as it was in the MAC of M, i.e., the same as $E_K(E_K(0) \oplus M_1)$. That is, the value that gets XOR'd with M_2 at the end will be the same for both.

Now $T = M_2 \oplus E_K(M_1 \oplus E_K(0))$. Since $E_K(M_1 \oplus E_K(0))$ will be constant between the two messages if we ensure $M'_1 = M_1$, changing M_2 will change T in the same way. For example, to set $T' = T \oplus 1$ we would set M'_2 to $M_2 \oplus 1$.

Q6.6 (6 points) Define AES-AND-MAC$(K, M) = C_n$, where $C_i = E_K(M_i \wedge C_{i-1})$ and $C_1 = E_K(M_1)$.

$\wedge$ represents **logical AND** between two 128-bit blocks (bitstrings).



You will describe an attack on this MAC. First, you request a MAC over a **one-block** message M of your choosing.

Provide a value for M:

Solution: $M = (0)$

You then receive a MAC T over M. Given (M, T) from the previous step, provide a new **two-block** message M' with MAC T' that you can compute from the information available to you (without knowing the key K).

HINT: M must not equal M', but T can equal T'.

Provide a value for M':

Solution: $M' = (X, 0)$ where X can be anything.

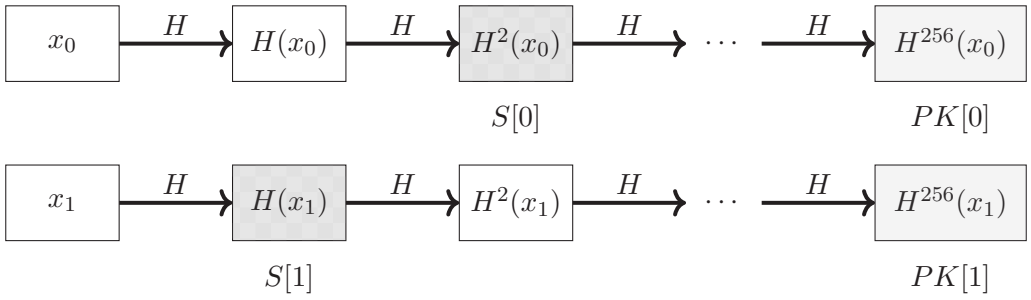
Alternatively, $M' = (0, !T)$ (i.e. logical NOT T, since $T \wedge !T = 0$)

Provide a value for T':

Solution: T. This works because $M_1 = M'_1 \wedge M'_2 = 0$, so in both cases the correct tag will be $T = E_K(0)$.

Q7 *Opaque Only Once — Digital Signatures* **(18 points)**

The rise of quantum computing worries EvanBot, who decides to invent a post-quantum signature scheme using only hash functions.



Pictured: A signature over the two-byte message $M = [$`0x02`, `0x01`$]$, with signature and public key: $S = [H^2(x_0), H(x_1)]$, $PK = [H^{256}(x_0), H^{256}(x_1)]$.

Clarification after exam: The example pictured above has a bug: it doesn't hash the message. The real scheme hashes the message first and then signs the bytes of the hash.

Key Generation:

1. Generate a list of 32 randomly-generated 256-bit values x_i: $[x_0, x_1, \ldots, x_{31}]$ as the private key.
2. Derive the public key by applying H to each x_i 256 times: $[H^{256}(x_0), H^{256}(x_1), \ldots, H^{256}(x_{31})]$.

Signing a Message:

1. Hash the message M to receive a 256-bit hash $H(M)$. Split $H(M)$ into 32 bytes n_i: $[n_0, n_1, \ldots, n_{31}]$.
2. For each $i \in [0, 31]$, apply n_i iterations of H to x_i to receive $H^{n_i}(x_i)$.
3. Publish the signature $S = [H^{n_0}(x_0), H^{n_1}(x_1), \ldots, H^{n_{31}}(x_{31})]$.

Verifying a Signature:

1. Given a signature S, let $S[i]$ refer to the i-th entry in the signature ($H^{n_i}(x_i)$). Let $PK[i]$ refer to the i-th entry in the public key ($H^{256}(x_i)$).
2. For each $i \in [0, 31]$, [ANSWER TO Q7.1].

Q7.1 (3 points) Let n_i be the i-th byte of $H(M)$, treated as an unsigned 8-bit integer.

Select the best option to fill in the blank from the signature verification protocol.

- ○ (A) Let $T = H^{n_i}(x_i)$. Verify that $T = S[i]$.
- ● (B) Let $T = H^{256-n_i}(S[i])$. Verify that $T = PK[i]$.
- ○ (C) Let $T = (H^{-1})^{256-n_i}(PK[i])$. Verify that $T = S[i]$.
- ○ (D) Let $T = H^{256}(x_i)$. Verify that $T = PK[i]$.

Solution: We know that $S[i] = H^{n_i}(x_i)$ and $PK[i] = H^{256}(x_i)$ from the definition provided in the start of the question. The core idea behind this scheme is that only the original public key owner knows x_i, and only they can evaluate $H^{n_i}(x)$ for $0 \leq n_i \leq 255$. Other parties know $H^{256}(x_i)$ but can't invert the hash to get any of the earlier values.

To verify that $H^{n_i}(x_i)$ is truly H applied n_i times to x_i, we use that fact that we know $H^{256}(x_i)$ and that $H^{256}(x_i) = H^{256-n_i}(H^{n_i}(x_i))$. By hashing $S[i]$ the remaining $256 - n_i$ times, we should get the original public key value $H^{256}(x_i)$.

Q7.2 (4 points) Which properties are *necessary conditions* for the signature scheme to be secure?

- ■ (A) $H, H^2, \ldots, H^{256}$ are one-way
- ■ (B) H is collision resistant
- ☐ (C) H is secure against length-extension attacks
- ☐ (D) The message never has a byte of all ones
- ☐ (E) The output of H never has a byte of all zeroes
- ☐ (F) None of the above

Solution: If $H, \cdots, H^{256}$ aren't one-way, then an attacker can take $S[i] = H^{n_i}(x_i)$ and recover either x_i, or some other preimage — some other value that is just as good as x_i (as it hashes to the same thing as x_i). Given all the x_i values, the attacker can create a signature on any arbitrary message by following the original protocol.

If H is not collision-resistant, then an attacker can find a collision $H(M) = H(M')$, ask for the signature on M, and use it as a valid signature on M'.

Length-extension attacks aren't relevant here, because the internal steps are over constant-length inputs (since we hash the message, and the output of the hash function has a constant length).

It does not matter if the message or its hash has a byte of all zeroes or ones, since those value still lie in the range $0 \leq n_i \leq 255$, and $H^{n_i}(x_i)$ can't be reached from $H^{256}(x_i)$.

Q7.3 (6 points) Alice sends Bob a message M with signature S, generated with her private key. Mallory is eavesdropping on their conversation and learns (M, S). Mallory wishes to find a new message/signature pair (M', S') that verifies under Alice's public key PK.

Let n_i, n'_i be the i-th bytes, parsed as an 8-bit unsigned integer, for $H(M), H(M')$ respectively. What must be true **for all** $i \in [0, 31]$ for Mallory to succeed in forging a signature for M'? Select the most accurate option.

● (A) $n_i \leq n'_i$ ○ (C) $n_i = n'_i$ ○ (E) $n_i > n'_i$

○ (B) $n_i \geq n'_i$ ○ (D) $n_i < n'_i$ ○ (F) $n'_i < PK[i]$

Assuming the message M' satisfies the correct condition, Mallory then calculates S'. For each $i \in [0, 31]$, give an expression for $S'[i]$, in terms of $n_i, n'_i, H, S[i], PK[i]$ (you do not need to use all those variables):

Solution: $H^{n'_i - n_i}(S[i])$

To forge a signature on M', we need to find $S'[i] = H^{n'_i}(x_i)$ where n'_i is the i-th byte of $H(M')$. We don't know x_i, so we can't evaluate $H^{n'_i}(x_i)$ directly. However, we *do* know $S[i] = H^{n_i}(x_i)$, and if $n'_i \geq n_i$, then $H^{n'_i} = H^{n'_i - n_i}(H^{n_i}(x_i)) = H^{n'_i - n_i}(S[i])$.

Q7.4 (2 points) Assuming H is a secure hash function, what is the approximate probability of the condition in the previous subpart being true for two randomly-selected messages M, M'?

○ (A) 2^{-8} ● (C) 2^{-32} ○ (E) 2^{-128}

○ (B) 2^{-16} ○ (D) 2^{-64} ○ (F) 2^{-256}

Solution: The probability that $k'_i \geq k_i$ is approximately $\frac{1}{2}$, since both are (pseudo-)randomly picked from the same uniform range.

Using the (reasonable) assumption that each byte is independent, with 32 bytes that comes out to $\frac{1}{2^{32}} = 2^{-32}$.

Q7.5 (3 points) Select all options which would decrease the probability of the condition being true.

- ☐ (A) Increasing the length of each x_i to be greater than 256 bits.
- ☐ (B) Using 2 bytes for each n_i instead of 1 byte (e.g., n_0 would now be the first 2 bytes of $H(M)$ parsed as a 16-bit integer). Assume the public key values change to $H^{2^{16}}(x_i)$ accordingly.
- ■ (C) Using a hash function with a 512-bit output (assume that there would be 64 1-byte entries in S and PK rather than the original 32 entries accordingly).
- ☐ (D) None of the above

Solution: Option (C) reduces the probability of the condition to about 2^{-64}, since we "flip the coin" on $n_i' \geq n_i$ twice as much.

Nothing on this page will affect your grade.

Post-Exam Activity

Help EvanBot out by drawing some toppings on their pancakes!

Comment Box

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