CS 161 Summer 2024 — Introduction to Computer Security — Midterm

Solutions last updated: July 11th, 2024

Name: _____

Student ID: _____

This exam is 110 minutes long.

Question:	1	2	3	4
Points:	0	16	17	17
Question:	5	6	7	Total
Points:	13	20	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):

Evanbot is in charge of managing a zoo. Uh oh! Evanbot lost the key and one animal escaped. Circle the one you believe is missing!

To prove EvanBot won't lie (to their boss), here's the SHA256 hash of the animal that escaped:

```
cd08c4c4316df20d9c30450fe776dcde4810029e641cde526c5bbffec1f770a3
```

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 True or False: A 64 byte char array on the stack starting at `0xFFFFD840` ends at `0xFFFFD8A4`.

○ True ● False

Solution: False, 64 bytes after `0xFFFFD840` is `0xFFFFD880`. Remember, the stack grows upwards and 64 bytes is $4 * 16^1$.

Q2.2 True or False: If the address `0x161ABDAC` is stored as a pointer on the stack, then `0x16` is stored at the lowest memory address **in a big-endian system.**

● True ○ False

Solution: True, in big-endian, the most significant bit (left-most) is stored at the lower memory address. The opposite is true for little-endian.

Q2.3 Evanbot just designed a completely new security system to protect their pancakes. Evanbot is convinced that nobody can learn about their system, so they don't need to worry about Shannon's Maxim.

True or False: This is the intended application of Shannon's Maxim.

○ True ● False

Solution: False, Shannon's Maxim is the premise that we assume an attacker knows how our system works and it applies regardless of whether or not it would be logistically impossible to know this information, i.e. we should always assume an attacker knows our system.

Q2.4 True or False: In CALL (compiler-assembler-linker-loader), the loader will create a binary executable of the program you're trying to run.

○ True ● False

Solution: False, this is created by the assembler/linker, not the loader.

Q2.5 True or False: In CALL, the bits in the code section of memory were originally outputted by the assembler and linker.

● True ○ False

Solution: True, the assembler and linker create the binary, which is what is ultimately loaded into the code section of memory.

Q2.6 True or False: The x86 push instruction decrements the `ESP` and stores the new value on the stack.

● True ○ False

Solution: True, this is the function of the push instruction as discussed in lecture.

Q2.7 True or False: Return-oriented programming is a way to subvert non-executable pages.

● True ○ False

Solution: True, in return-oriented programming we are essentially using already loaded code to construct shellcode, which circumvents NX pages.

Q2.8 True or False: A system implementing stack canaries, non-executable pages, ASLR, and PACs is still exploitable.

● True ○ False

Solution: True, while adding more defenses can make a system more safe, there may still be vulnerabilities in code–this is why we always seek to find new ways to improve our security systems even at the coding level!

Q2.9 True or False: AES-ECB encryption can be parallelized.

● True ○ False

Solution: True, each block of text in ECB is encrypted independently of the other blocks, so it can be done in parallel.

Q2.10 Alice is encrypting multiple messages with AES-CBC. She uses a PRNG to generate IVs for each encryption. Mallory knows the seed to the PRNG.

True or False: Given a ciphertext, Mallory can learn the plaintext.

○ True ● False

Solution: False, Mallory may be able to guess the values output by the PRNG (and thus the IVs), but since IVs are supposed to be public anyway, this makes no difference to the confidentiality of the ciphertext.

Q2.11 True or False: Let E_K be a secure block cipher. It is computationally feasible to find two messages M_0 and M_1 such that $M_0 \neq M_1$ and $E_K(M_0) = E_K(M_1)$, even if the attacker knows K.

○ True ● False

Solution: False. If the block cipher is secure, then following the bijection property of block ciphers, it is not feasible to find messages that are different but are the same when encrypted.

Q2.12 True or False: Let H be a secure hash function. It is computationally feasible to find two messages M_0 and M_1 such that $M_0 \neq M_1$ and $\mathsf{H}(M_0) = \mathsf{H}(M_1)$.

○ True ● False

Solution: False, this is the collision resistance property of hashes.

Q2.13 True or False: SHA-2 is vulnerable because given a message $\mathsf{H}(M)$ and the length, we are able to rederive M.

○ True ● False

Solution: False, SHA-2 is vulnerable because of length extension attacks, but these attacks are not able to rederive M.

Q2.14 Alice and Bob want to ensure they can send messages without Mallory tampering with them, therefore they use MACs. However, Mallory knows the key K used to compute their MACs.

TRUE or FALSE: Alice and Bob could attach $\mathsf{H}(M)$ or $\mathsf{HMAC}(K, M)$, and Mallory could tamper with the message either way.

● TRUE ○ FALSE

Solution: True, a MAC is essentially a Hash with a key, if this key is publicly known then an attacker can compute the MAC for any message which is equivalent to a Hash.

Q2.15 TRUE or FALSE: A man-in-the-middle attacker who cannot solve the discrete log problem can still exploit Diffie-Hellman key exchange.

● TRUE ○ FALSE

Solution: True, remember a MITM in the Diffie-Hellman key exchange is able to compute keys custom keys for the 2 communicating parties and can decrypt and encrypt any/all messages sent. This is regardless of whether or not the discrete log problem has a solution.

Q2.16 TRUE or FALSE: Even if we have a solution to the discrete log problem, El Gamal is semantically secure.

○ TRUE ● FALSE

Solution: False, given a way to solve the discrete log problem, an eavesdropper can recover the private keys used in El Gamal from the public keys.

More explicitly, say Alice has chosen prime p, generator g, and private key a. Say Bob has chosen private key b. Given the values of g^a and $g^b \pmod p$, a person who can solve discrete log can find a' such that $g^{a'} = g^a \pmod p$, then calculate $(g^b)^{a'} = (g^{a'})^b = g^{ab} \pmod p$.

Q3 *What Would C Do* **(17 points)**

There is a function `system(char* command)` in the C standard library. It can be used to execute the shell command passed in as the argument `command`.

- `system(char* command)` is located in memory at `0x08FECB3A`.
- `something[]` is located at `0xFF001020`.
- `padding[]` is located at `0xFF001048`.

```
1  void say_something(void) {
2      char something[32];
3      gets(something);
4  }
5
6  int main() {
7      char* command = "whoami";
8      char padding[4];
9      say_something();
10     return 0;
11 }
```

Our goal is to execute the command `whoami`. To do this, we will construct an input to the `gets` in line 3 that causes this program to call `system("whoami")`.

The input to `gets` will take the following form:

`"A" *` _____(1)_____ `+` _____(2)_____

Q3.1 (1 point) Which option provides the correct input and rationale for the first blank (1)?

- ○ **32**, to overwrite all of `something`
- ○ **32**, to overwrite all of `something` and the SFP of `say_something`
- ○ **36**, to overwrite all of `something`
- ● **36**, to overwrite all of `something` and the SFP of `say_something`

Solution: 32 (which is the size of something) + 4 (which is the size of the SFP) = 36 bytes to write to reach the RIP.

Q3.2 (1 point) Which option provides the correct input and rationale for the second blank (2)?

- ○ `0x08FECB3A`, to overwrite the RIP of `main` with the address of `system`
- ○ `0xFF001050`, to overwrite the RIP of `main` with the address of `system`
- ● `0x08FECB3A`, to overwrite the RIP of `say_something` with the address of `system`
- ○ `0xFF001050`, to overwrite the RIP of `say_something` with the address of `system`

Solution: We are overwriting the RIP of `say_something` (because this is the only function in the options whose RIP is below `command`, which must be higher up on the stack than the RIP to facilitate using it as an argument) with the address of system to change the flow of execution, as in a typical overflow attack.

Q3.3 (1 point) Is the stack variable `padding` necessary? Why or why not?

- ○ **No**, because `system` is expecting an `RIP` on the stack and looks above it for arguments
- ● **Yes**, because `system` is expecting an `RIP` on the stack and looks above it for arguments
- ○ **No**, because `system` is expecting an `SFP` on the stack and looks above it for arguments
- ○ **Yes**, to prevent the overflow attack from overwriting `whoami`

Solution: Clarification during exam: Q3.3: "necessary" means "necessary to conduct the attack." (7:36pm)

Yes–`system` expects to see an RIP on the stack and looks above it for arguments. `padding` serves as extra space on the stack where the RIP would be were the function being called normally, so the argument (char ptr to whoami) goes above it.

Q3.4 (2 points) What purpose does `command` have on the stack?

- ○ It is the string "whoami" that is passed as the argument to `system`
- ● It is the pointer to the string "whoami" that is passed as the argument to `system`

Solution: `system` expects a pointer to a string (char*).

Q3.5 (1 point) When does the execution of the `system` function begin?

- ○ After `main` returns
- ● After `say_something` returns
- ○ After `gets` returns
- ○ After `gets` begins execution

Solution: The RIP of say_something is overwritten, so when it returns the execution of system will begin.

Q3.6 (2 points) What address is the `ESP` pointing to when the execution of the `system` function begins? (i.e. just after the execution has been handed over to the `system` function)

- ○ `0xFF001044`
- ○ `0xFF00104C`
- ○ `0xFF001050`
- ● `0xFF001048`

Solution: 0xFF001048. This is because execution of the `system` function begins as soon as the RIP (of the function `say_something`) we have overwritten is popped off of the stack. Once this happens, the next thing up on the stack (since we assume no compiler padding) is the variable `padding`, which is located at 0xFF001048.

The following subparts are **independent**.

```
1  void special_printf(char* str) {
2      bool stop = false;
3      for (unsigned int i = 0; i < strlen(str) - 1; i++) {
4          if (str[i] == '%' && str[i+1] == 'x') {
5              stop = true;
6          } else if (str[i] == '%' && str[i+1] == 'd') {
7              stop = true;
8          }
9      }
10     if (stop) return;
11     int special = 0xABCD;
12     int not_special = 0xEEEE;
13     printf(str);
14 }
```

Q3.7 (3 points) What could you pass in as `str` that would allow the value of `special` to be leaked?

(There are multiple possible answers; `0xABCD` is not one of them. Using Python syntax is acceptable.)

Solution: Clarification during exam: Q3.7: You are allowed to leak other information alongside the value of special. (7:27pm)

%c, %f, %X, %8x...

There are many possible solutions to this problem. All that was necessary was to use a format specifier other than %d or %x to print out `special` (we clarified during the exam that it was okay if you did that and also printed out `not_special`, for example).

Answers like ("%u", `special`) do not work, since we asked specifically for a value for the `char* str`.

Due to a mistake in how the code was formatted, the code we wrote to prohibit %d or %x is not run, so answers using those specifiers that printed out `special`'s value were also accepted.

In this **independent** code sample, assume that:

- Calls to `malloc` always succeed.
- `malloc` always allocates space at the lowest available memory address.
- This code will not segfault, and can successfully read memory at any memory address.
- Nothing but the program itself will change the heap.

```
1  void special_alloc() {
2      int* alloc_num = malloc(sizeof(int));
3      *alloc_num = 0xCDAB;
4      printf("Call 1: %x", *alloc_num);
5      free(alloc_num);
6      printf("Call 2: %x", *alloc_num);
7      int* new_num = malloc(sizeof(int));
8      *new_num = 0x1234;
9      printf("Call 3: %x", *alloc_num);
10 }
```

Q3.8 (2 points) What could the first call to `printf` possibly output? Select all that apply.

- ■ `Call 1: cdab`
- ☐ `Call 1: abcd`
- ☐ `Call 1:` followed by garbage bytes other than `cdab` or `abcd`
- ☐ `Call 1:` followed by the address of `alloc_num` on the heap
- ☐ None of the above

Solution: Option 1 only. This because this is just a normal call to `printf`–the %x specifier will print out the value in hexadecimal, lowercase format.

Q3.9 (2 points) What could the second call to `printf` possibly output? Select all that apply.

- ■ `Call 2: cdab`
- ☐ `Call 2: 1234`
- ☐ `Call 2:` followed by garbage bytes other than `cdab` or `1234`
- ☐ `Call 2:` followed by the address of `alloc_num` on the heap
- ☐ None of the above

Solution: Option 1 only. This because this is also just a normal call to `printf`–the %x specifier will print out the value in hexadecimal, lowercase format. Note that we specified that only this code will change the heap–despite the fact that the memory has been freed, the data there only changes if it is overwritten, which has not yet happened.

Q3.10 (2 points) What could the third call to `printf` possibly output? Select all that apply.

- ☐ `Call 3: cdab`
- ■ `Call 3: 1234`
- ☐ `Call 3:` followed by garbage bytes other than `cdab` or `1234`
- ☐ `Call 3:` followed by the address of `alloc_num` on the heap
- ☐ None of the above

Solution: Option 2 only. The `printf` call functions the same as before, but using the assumption that `malloc` always allocates at the lowest available memory address, we can know that `new_num` will occupy the same location on the heap as `alloc_num`, so printing out with a reference to the address of `alloc_num` will print the value given to `new_num`.

Q4 *evan86* (17 points)

EvanBot has modified x86 so that it's now impossible to directly overwrite the RIP of a function! If EvanBot sees that the value at the RIP's original stack location has been changed from its original value at any point **before** the function returns, the program will immediately terminate.

Your goal is to find a way to execute the shellcode located in memory at `0xABBA0161`. This shellcode is **outside the code section of memory**.

- `pancake_stack` is located at `0xBFFEED00`.

```
1  int get_user_input(int8_t read_amount) {
2      char buf[248];
3      if (read_amount > 248) return -1;
4      fread(buf, 1, read_amount, stdin);
5      memset(buf, 0, 248);
6      return 0;
7  }
8
9  int vuln() {
10     char pancake_stack[20];
11     fread(pancake_stack, 1, 20, stdin);
12     get_user_input(________);
13     return 0;
14 }
```

Stack at Line 2

SFP of `vuln`
(1)
(2)
RIP of `get_user_input`
SFP of `get_user_input`
(3)

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

● (1) `pancake_stack`	(2) `read_amount`	(3) `buf`
○ (1) `pancake_stack`	(2) `buf`	(3) `read_amount`
○ (1) RIP of `vuln`	(2) SFP of `fread`	(3) `buf`
○ (1) RIP of `vuln`	(2) `pancake_stack`	(3) `read_amount`

Solution: Option 1. From top to bottom: `pancake_stack` is the first stack variable declared in `vuln`, so it will be right under the SFP for that function. Next is the argument passed into `get_user_input`, which is `read_amount`, since arguments are pushed onto the stack right before the function is called. Finally, we see the stack variable that is declared first in the called function `get_user_input`, `buf`.

Q4.2 (2 points) Which of these values does the exploit have to overwrite, either directly or indirectly, to work? Select all that apply.

- ☐ SFP of `vuln`
- ☐ SFP of `fread`
- ■ SFP of `get_user_input`
- ☐ RIP of `get_user_input`
- ☐ None of the above

Solution: Only the SFP of `get_user_input` needs to be overwritten. Doing this will cause the EBP to be misdirected during a return to a place of our choosing–in this exploit, inside `pancake_stack`–which means that when there is a second return, the program will look above this new EBP for an address to use as an RIP, which we can set to whatever we want. The misdirection means that nothing else here needs to be overwritten for the exploit-by-sfp to work.

In the next three subparts, provide inputs that would cause the program to execute the shellcode.

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

Q4.3 (3 points) What is a value you could give for `read_amount` (the blank in line 12) that would allow the exploit to work, **AND** would **NOT** allow overwriting the RIP of any function?

Solution: -6 is the intended answer, and -5 and -4 also work and received full credit.

This question required some knowledge of two's complement. Because of the if statement in the code preventing numbers greater than 248 from being used, we need to circumvent this check by exploiting a signed/unsigned vulnerability. The question asks for a value which allows overwriting the SFP to get the exploit to work (see 4.2), but also a value which won't allow overwriting the RIP located right above the SFP. Because of the layout of the stack, it was necessary to overwrite at least 2 bytes of the SFP to complete the exploit as intended. Therefore, we need to write at least 250 bytes, but no more than 252. Represented as signed `int8_t` values, this becomes -6, -5, or -4.

Q4.4 (4 points) Input to `fread` at Line 4:

Solution: "A" * 248 + "\x00\xED"

As mentioned in 4.2, our goal is to redirect the SFP of `get_user_input` to be inside of `pancake_stack`. `pancake_stack` begins at the address `0xBFFEED00`–meanwhile, the SFP as it is contains an address certainly beginning with `0xBFFE`. So, all that is needed is to overwrite the last two bytes to get the SFP to point to `pancake_stack`.

Q4.5 (4 points) Input to `fread` at Line 11:

Solution: "A" * 4 + "\x61\x01\xBA\xAB" + "A" * 12

Having pointed the SFP to `pancake_stack` in the previous part, we now fill out `pancake_stack`. The first 4 bytes represent the "fake SFP"–where the program believes the SFP will be located after it executes one return. Above that is the "fake RIP"–in other words, where we should put the address of shellcode. The rest of the array can then be filled with garbage. It is not necessary to have the garbage at the end of the array.

Q4.6 (1 point) When does the shellcode execute in this problem?

- ○ When `get_user_input` returns
- ○ When `fread` returns
- ● When `vuln` returns
- ○ When `buf` is filled

Solution: When vuln returns. This is because `get_user_input` returning will cause the esp to point to `pancake_stack`–then, when vuln returns, the value 4 bytes above the esp is used as the RIP.

Consider the following parts separately from one another.

Q4.7 (1 point) If ASLR were enabled for this problem, but you could correctly predict the addresses of shellcode and `pancake_stack`, is this same exploit still possible?

- ● Yes, because the layout of the stack itself will be arranged in the same way as before.
- ○ Yes, because ASLR wouldn't change the addresses of things on the stack anyway.
- ○ No, because we couldn't know for sure that the values on the stack will be arranged in the same way as before.
- ○ No, because this would simply prevent overwriting the RIP, which is already prevented in this problem.

Solution: Yes, option 1. ASLR may change where the stack is located, but it won't change the arrangement of data on the stack itself. If you can predict the addresses that are relevant to the exploit, it can still be executed.

Q4.8 (1 point) If non-executable pages were enabled for this problem, is this same exploit still possible?

- ○ Yes, because non-executable pages cannot be applied to anywhere in memory but the heap.
- ○ Yes, because non-executable pages can be circumvented, allowing us to execute shellcode in the same way as before.
- ● No, because the shellcode is located outside the code section, so it couldn't be executed directly.
- ○ No, because non-executable pages prevent overflow attacks in the first place.

Solution: No. The shellcode for this problem isn't located in the code section of memory–and NX pages would be enabled for all memory outside of the code section. Therefore, the code would not be able to be executed, and the exploit could not work in the same way.

Q5 *Memory Safety: An "Off" Trip to the Zoo* **(13 points)**

Evanbot and Codabot are volunteering as zookeepers today. Their jobs are to set up the exhibits for the day. Consider the following vulnerable C code:

```
1  typedef struct {
2    char body[16];
3  } giraffe;
4
5  typedef struct {
6    char body[24];
7  } zebra;
8
9  typedef struct {
10   char body[24];
11 } elephant;
12
13 void placements() {
14     char zoo[64];
15     char list[74];
16
17     memset(zoo, 0, 64);
18     fgets(list, 74, stdin);
19
20     giraffe* g = malloc(sizeof(giraffe));
21     fgets(g->body, 17, stdin);
22
23     zebra* z = malloc(sizeof(zebra));
24     fgets(z->body, 25, stdin);
25
26     elephant* e = malloc(sizeof(elephant));
27     fgets(e->body, 25, stdin);
28
29     for (int i = 0; i < 71; i++) {
30         zoo[i] = list[i];
31     }
32 }
```

Stack at Line 31

RIP of `placements`
(1)
`zoo`
(2)
(3)
...

Assumptions:

- `malloc` always allocates starting at the lowest possible address with enough free space.
- `malloc` always allocates the exact amount of memory required by its input, with no metadata.
- No other process is modifying the heap either before this function runs or concurrently.
- The heap starts at address `0x53ABFF08` and grows upwards.
- Your goal is to place and execute a 60-byte `SHELLCODE`.
- The address stored **in the RIP** of `placements` is `0x08AA7F3C`.
- One-byte NOPs exist in memory at `0x53ABFF04`, `0x53ABFF05`, `0x53ABFF06`, `0x53ABFF07`.

EvanBot says you should go re-read the assumptions before proceeding!

The following x86 instructions exist in memory at the following locations listed below. Use this table for the following subparts!

0x0861321A	jmp *0x53ABFF04
0x01BAFFFF	jmp *0x53ABFF08
0x08AA7F3F	addl 0x8, %ebx
0xDEADBEEF	jmp *0x08AA7F3C
0xffffca1c	ret

Q5.1 (1 point) What values go in blanks (1) through (3) in the stack diagram above?

- ● (1) SFP of `placements` (2) `list` (3) `&g`
- ○ (1) SFP of `placements` (2) `list` (3) `i`
- ○ (1) `list` (2) SFP of `placements` (3) `i`
- ○ (1) `list` (2) SFP of `placements` (3) `&g`

Solution: Option 1. SFP of `placements` follows the RIP. Below `zoo` is list, and the next variable defined is the giraffe pointer g, so `&g` is put on the stack.

Q5.2 (1 point) Which vulnerability is present in the code?

- ○ ret2libc
- ○ Format string vulnerability
- ○ Signed/unsigned vulnerability
- ● None of the above

Solution: None of the above. This is not ret2libc as this requires exploiting a function which is part of the C standard library. There is no signed/unsigned vulnerability either as no signed number are converted to unsigned numbers or vice-versa. There is no printf for a format string vulnerability.

In the next 4 subparts, provide inputs that would cause the program to execute `SHELLCODE`.

Q5.3 (8 points) Input to `fgets` at Line 18:

Solution: `"A" * 68 + \x1A\x32\x61`

We input 68 bytes of garbage to overwrite all of zoo and 4 bytes for the SFP.

In order to change the RIP to the address of `SHELLCODE`, we need to point to the bottom of the heap (see next parts for what that is). This is not possible as we are only able to modify the 3 least significant bytes of the RIP which means we can't overwrite it directly. With our given x86 instructions, we can however rewrite to a jmp instruction that sends the instruction pointer to the beginning of the heap. The only 2 instructions that do this are `jmp *0x53ABFF04` stored at `0x0861321A` and `jmp *0x53ABFF08` stored at `0x01BAFFFF`.

Notice how the latter doesn't have a matching MSB as the address stored at the RIP so we cannot actually redirect the RIP to this instruction. The other one is reachable but doesn't point to the beginning of the heap, but 4 bytes below. The assumptions tell us that there are 4 NOPs below the heap though so if the IP was set to `0x53ABFF04`, it would go down the NOP sled and execute the shellcode. Therefore, after garbage, we write the least significant 3 bytes of this instruction in little-endian.

Input to `fgets` at Line 21:

Solution: `SHELLCODE[:16]`

The indicator that this problem requires split `SHELLCODE` is that we have no way of overwriting the RIP to a place on the stack as we cannot modify the MSB of the RIP. The heap is reachable though as shown above. We write the SHELLCODE starting here going up. We write 16 bytes as `fgets` will automatically append one null byte to fill the 17 bytes allocated. Since we call malloc for only size of the animals, we will continuously overwrite the null bytes which each subsequent `fgets` call.

Input to `fgets` at Line 24:

Solution: `SHELLCODE[16:40]`

Refer to the explanations above.

Input to `fgets` at Line 27:

Solution: `SHELLCODE[40:]`

Refer to the explanations above.

Q5.4 (1 point) Would it still be possible for your exploit to work (without modifications) if stack canaries are enabled?

- ○ Yes, because the exploit writes around the canary to overwrite values above the canary.
- ○ Yes, because the exploit never tries overwriting values above the canary.
- ● No, because we cannot leak the canary value before overwriting it.
- ○ No, because the least-significant byte of the canary is overwritten by a null byte.

Solution: As we are overwriting the stack from zoo to the RIP, we need to know the canary (which would exist below the SFP but above zoo) in order to overwrite it properly, but the code above does not do so.

Q5.5 (2 points) Evanbot spilled syrup all over the stack, and now the value of the RIP of `placements` is randomized to 4 random bytes immediately before line 17! What is the probability that this exploit will still work now?

- ○ 0
- ○ `1/64`
- ○ `1/16`
- ● `1/256`

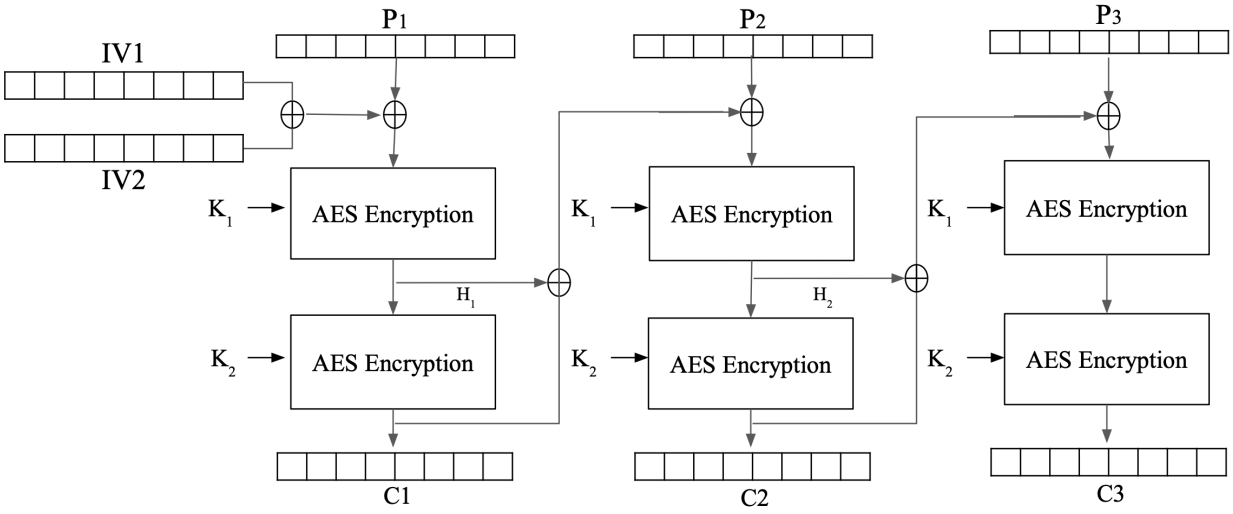
Solution: Clarification during exam: "this exploit" specifically refers to the solution you wrote in the previous subparts.

In order to conduct the exact same exploit the, the MSB of the RIP would have to be `0x08`, which has a $1/16^2$ or `1/256` chance of occurring.

Q6 *Symmetric Cryptography: AES-OHP* **(20 points)**

EvanBot designs the AES-OHP mode of operation. Here are the encryption equations for $i \geq 2$:

$$H_1 = E_{K_1}(P_1 \oplus IV_1 \oplus IV_2) \qquad H_i = E_{K_1}(P_i \oplus C_{i-1} \oplus H_{i-1})$$
$$C_1 = E_{K_2}(H_1) \qquad C_i = E_{K_2}(H_i)$$



Q6.1 (1 point) Select the decryption formula for H_i, for $i \geq 1$.

- ● $H_i = D_{K_2}(C_i)$
- ○ $H_i = D_{K_2}(C_{i-1})$
- ○ $H_i = D_{K_1}(C_i)$
- ○ $H_i = D_{K_2}(C_{i-1})$

Solution: Clarification during exam: Q6.1: The right two options are interchangeable, i.e. we will treat them as the same answer

The encryption formula for C_i is $C_i = E_{K_2}(H_i)$, if we reverse this we get $H_i = D_{K_2}(C_i)$.

Q6.2 (1 point) Select the decryption formula for P_i, for $i \geq 2$.

- ● $P_i = D_{K_1}(D_{K_2}(C_i)) \oplus H_{i-1} \oplus C_{i-1}$
- ○ $P_i = D_{K_2}(E_{K_1}(C_i)) \oplus H_i \oplus C_{i-1}$
- ○ $P_i = D_{K_2}(D_{K_1}(C_i)) \oplus H_{i-1} \oplus C_{i-1}$
- ○ $P_i = D_{K_1}(E_{K_2}(C_i)) \oplus H_i \oplus C_{i-1}$

Solution: The encryption formula is $H_i = E_{K_1}(P_i \oplus C_{i-1} \oplus H_{i-1})$.

Let's decrypt both sides: $D_{K_1}(H_i) = P_i \oplus C_{i-1} \oplus H_{i-1}$.

Now XOR the values and we get: $P_i = D_{K_1}(H_i) \oplus C_{i-1} \oplus H_{i-1}$.

When we plug the answer to the above question in we get $P_i = D_{K_1}(D_{K_2}(C_i)) \oplus H_{i-1} \oplus C_{i-1}$.

Q6.3 (1 point) Select all true statements.

- ☐ Encryption is parallelizable.
- ☐ None of the above
- ■ Decryption is parallelizable.

Solution: The decryption formula does not rely on prior plaintexts, whereas the encryption formula contains the previous ciphertext to compute.

Q6.4 (2 points) Select all true statements.

- ■ AES-OHP is IND-CPA secure if IV_1 and IV_2 are independently randomly generated.
- ■ AES-OHP is IND-CPA secure if IV_1 is known but IV_2 is randomly generated.
- ☐ AES-OHP is IND-CPA secure if both IV_1 and IV_2 are predictable.
- ☐ AES-OHP is IND-CPA secure if both IV_1 and IV_2 are fixed constants.
- ☐ None of the above

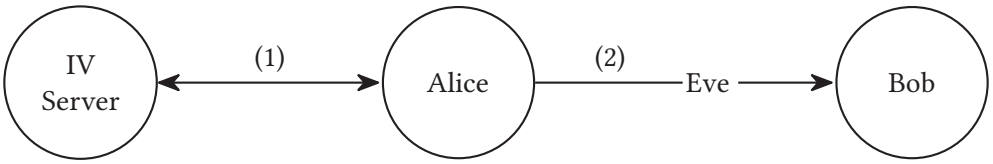
Solution: The key realization here is that $IV_1 \oplus IV_2$ is equivalent to a singular random IV and as long as there is an element of randomness in this IV, then AES-OHP is IND-CPA secure.

By this logic Option 1 is true. Option 2 is also true as one of the IVs is random which means the outputted IV will be random as well. Options 3 and 4 are equivalent and there is no element of randomness which means AES-OHP is not IND-CPA secure.

Alice uses AES-OHP mode to encrypt and send two 3-block messages to Bob. Alice obtains her IVs from a server that provides IVs.

Eve is an attacker with these capabilities:

- Eve is an eavesdropper who can see the ciphertexts.
- Eve knows the value of K_2, which means that given ciphertext C, she can compute the intermediate H_i values.
- In between the two encryptions, Eve hacks into the IV server, which means that she can provide malicious IVs for Alice's second encryption.



Alice encrypts the first message, (P_1, P_2, P_3):

(1) Alice requests an IV pair, IV_1 and IV_2, from the server.

(2) Alice computes and sends $(IV_1, IV_2, C_1, C_2, C_3)$. Eve can read this, and also derive (H_1, H_2, H_3).

Between the two encryptions, Eve hacks into the IV server. **Eve can now make the server return IVs of her choice.**

Then, Alice encrypts the second message, (P_1', P_2', P_3'):

(1) Alice requests another IV pair, IV_1' and IV_2' (**values chosen by Eve**), from the server.

(2) Alice computes and sends $(IV_1', IV_2', C_1', C_2', C_3')$. Eve can read this, and also derive (H_1', H_2', H_3').

For each subpart, select whether it is possible for Eve to answer the specified question with high probability.

If you select "Eve can answer this," write the values for IV_1' and IV_2', and write a strategy for answering the question.

A completely unrelated sample answer:

> $IV_1' = C_2' \oplus H_1$, and $IV_2' = IV_2$.
>
> Strategy: If $IV_2' = C_3'$ and $H_2 = IV_1$, Eve answers yes. Else, no.

Q6.5 (5 points) Are Alice's two messages identical? i.e. is it true that $P_1 = P_1'$, $P_2 = P_2'$, $P_3 = P_3'$?

● Eve can answer this ○ Eve cannot answer this

Solution: Clarification during exam: Q6.5-6.7: if you say Eve cannot answer this, you may leave the box empty. (8:30pm)

Many students wrote: $IV_1' = IV_1$ and $IV_2' = IV_2$ but the following explanation works too.

$IV_1' = IV_1 \oplus IV_2 \oplus IV_2'$ and $IV_2' = IV_1 \oplus IV_2 \oplus IV_1'$
Strategy: If $C_i' = C_i \forall i$ Eve answers yes, else No.

In order to see if the two messages are equal the consolidated IV's must be the same and equal $IV_1 \oplus IV_2$, therefore $IV_1' \oplus IV_2'$ = $IV_1 \oplus IV_2$.

Q6.6 (5 points) Do the first two blocks of the second message match the second and third blocks of the first message? i.e. is it true that $P_1' = P_2$ and $P_2' = P_3$?

● Eve can answer this ○ Eve cannot answer this

Solution: Many students wrote: $IV_1' = H_1$ and $IV_2' = C_2$ but the following explanation works too.

$IV_1' = P_1 \oplus H_1 \oplus IV_2'$ and $IV_2' = P_1 \oplus H_1 \oplus IV_1'$
Strategy: If $C_1' = C_2$ and $C_2' = C_3$, Eve answers yes, else No.

Note that in AES-OHP mode, $P_i \oplus H_i$ is treated as the IV value past block 1. Therefore if we set $IV_1' \oplus IV_2'$ = $P_1 \oplus H_1$, we are passing in equivalent IVs into P_1' and P_2 so we can see if these blocks and onwards are equal.

Q6.7 (5 points) Assuming the first blocks of both messages are different and Eve knows this—are the last blocks of both messages the same? i.e. is it true that $P_3 = P_3'$?

○ Eve can answer this ● Eve cannot answer this

Solution: Similar to AES-CBC mode, in cases of IV reuse, the ciphertexts blocks will appear the same for equal plaintext blocks until the first difference between plaintext blocks, after which ciphertexts block will appear completely different as the values XOR'd with the plaintexts have changed. So here if $P_1 \neq P_1'$, then the following ciphertext blocks are uncomparable regardless of whether they are equal.

Q7 *3-Way Diffie-Hellman* **(17 points)**

Alice, Bob, and Charlie are interested in what it would mean to do a 3-way Diffie-Hellman handshake. They decide on the following procedure.

1. Agree on a large prime p, and generator g.
2. Alice, Bob, and Charlie randomly choose private keys a, b, c (mod p).
3. They publish $g^a \pmod p$, $g^b \pmod p$, $g^c \pmod p$ respectively.
4. Using the information from step 3, they publish _____.

After steps 1-4 are completed, there is a shared key with the following security property: Alice, Bob, and Charlie all know the value of the shared key, but an eavesdropper with access to all communications cannot feasibly determine the shared key.

Q7.1 (2 points) What should the shared key be in this scheme?

Solution: The shared key should be $g^{abc} \pmod p$.

To understand our current scheme, first recall "normal" Diffie-Hellman. Alice and Bob want to agree on a key. They do so by choosing secret values a and b, then publishing g^a and g^b. Then they can derive a shared key g^{ab}: for example Alice will derive g^{ab} by calculating $(g^b)^a$.

The idea behind the current scheme is for Alice, Bob, and Charlie to first publish g^a, g^b, and g^c (step 2). After this, Alice or Bob can publish g^{ab}, Bob or Charlie can publish g^{bc}, and Charlie or Alice can publish g^{ca}. At this point, all three parties are able to derive g^{abc} using the published values and their own secret key.

Q7.2 (3 points) What should go in the blank for step 4? (Hint: it should be three values.)

Solution: They should publish $g^{ab}, g^{bc}, g^{ca} \pmod p$. (There is more than one way in which this can be done, see solution to part 1. Students don't need to specify which way, i.e. who publishes which one.)

Q7.3 (3 points) Explain how Alice derives the shared key using a and the published values. Write a clear equation and/or sentence.

Solution: She takes the published value g^{bc} and takes it to the ath power: $(g^{bc})^a = g^{abc}$.

Suppose we are given a prime p and generator g. The Diffie-Hellman problem asks:

Given $g^a \pmod p$ and $g^b \pmod p$ for randomly generated a, b, what is the value of $g^{ab} \pmod p$?

Q7.4 (1 point) Suppose that Mallory is an attacker who can solve the Diffie-Hellman problem. Is the current scheme used by Alice, Bob, and Charlie necessarily insecure against Mallory?

● Yes ○ No

Solution: It is insecure because if Mallory eavesdrops on the communications and learns the value of g^a and g^{bc}, solving the Diffie-Hellman problem, she can obtain $g^{a \cdot bc} = g^{abc}$.

Q7.5 (1 point) Suppose we're given a black box that solves the discrete log problem. Can we use this to solve the Diffie-Hellman problem?

● Yes ○ No ○ Don't know

Solution: Consider the Diffie-Hellman problem: we are given g^a, g^b. To find g^{ab}, first use discrete log solver to find a' such that $g^a = g^{a'}$, then take $(g^b)^{a'} = (g^{a'})^b = (g^a)^b = g^{ab}$.

Q7.6 (1 point) Mallory is a man-in-the-middle who is able to modify messages before they are published. Mallory has read all messages but has not modified any messages before step 4 of the handshake.

Can Mallory force everyone to derive a secret key that she knows? (Note: different people may derive different keys.)

● Yes ○ No

Solution: Mallory simply needs to replace g^{ab}, g^{bc}, g^{ca} with g^x, g^y, g^z for values x, y, z that she chooses. Alice will then derive the secret key $(g^x)^a = (g^a)^x$, which Mallory also knows. This is similarly the case for Bob and Charlie.

Q7.7 (1 point) Suppose now that we only know a and $g^{ab} \pmod{p}$. Assume GCD$(a, p-1) = 1$.

Is it computationally feasible to compute $g^b \pmod{p}$ with the information given?

- ● Yes
- ○ No
- ○ Depends on whether the discrete log problem is computationally feasible.

Solution: Q7.7: The public values (p and g) are still known. (8:31pm)

First find c such that $ac = 1 \pmod{p-1}$. (For example, use Euclidean algorithm.) Then we can calculate $(g^{ab})^c = (g^{ac})^b = g^b$.

Q7.8 (3 points) How does Diffie-Hellman provide forward secrecy? (Answer in 10 words or fewer.)

Solution: One can discard private keys after the handshake, so a future attacker will not discover their value by looking at records.

Q7.9 (2 points) Describe a drawback of asymmetric encryption. (The staff answer is one word.)

Solution: Slow

Post-Exam Activity

Nothing on this page will affect your grade.

Evanbot needs help putting on the fireworks show! Draw in your own fireworks below:

Comment Box

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