CS 161 Summer 2024 — Introduction to Computer Security — Midterm

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

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

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

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

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

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

○ True ○ False

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

○ True ○ False

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

○ True ○ False

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

○ True ○ False

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

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

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

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

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

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

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

○ TRUE ○ FALSE

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`

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`

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`

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`

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

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`

The following subparts are **independent**.

```c
void special_printf(char* str) {
    bool stop = false;
    for (unsigned int i = 0; i < strlen(str) - 1; i++) {
        if (str[i] == '%' && str[i+1] == 'x') {
            stop = true;
        } else if (str[i] == '%' && str[i+1] == 'd') {
            stop = true;
        }
    }
    if (stop) return;
    int special = 0xABCD;
    int not_special = 0xEEEE;
    printf(str);
}
```

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.)

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.

```c
void special_alloc() {
    int* alloc_num = malloc(sizeof(int));
    *alloc_num = 0xCDAB;
    printf("Call 1: %x", *alloc_num);
    free(alloc_num);
    printf("Call 2: %x", *alloc_num);
    int* new_num = malloc(sizeof(int));
    *new_num = 0x1234;
    printf("Call 3: %x", *alloc_num);
}
```

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

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

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

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`

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

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?

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

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

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

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

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.

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.

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:

```c
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`

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

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

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:

Input to `fgets` at Line 21:

Input to `fgets` at Line 24:

Input to `fgets` at Line 27:

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.

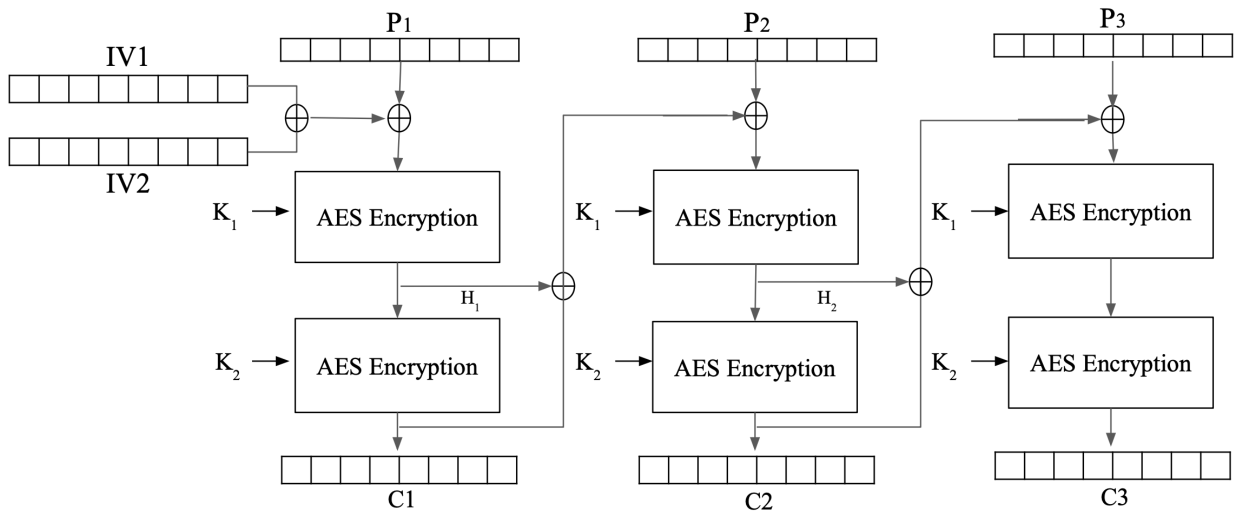
Q5.5 (2 points) Evanbot spilled syrup all over the stack, and now the value of the RIP of `placements` is randomized to 4 random bytes immediately before line 17! What is the probability that this exploit will still work now?

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

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 $i \geq 1$.

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

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

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

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

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

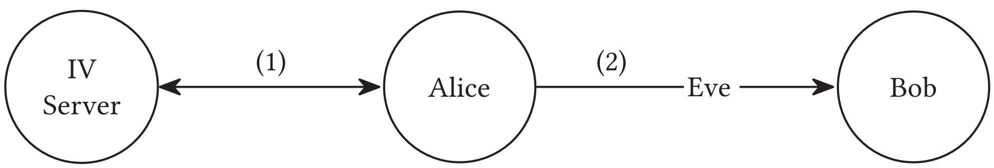
Q6.4 (2 points) Select all true statements.

- ☐ 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

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

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

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

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?

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

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

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

Given g^a (mod p) and g^b (mod p) for randomly generated a, b, what is the value of g^{ab} (mod 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

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

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

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.

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

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

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: