CS 161 Spring 2024 — Introduction to Computer Security — Final

Solutions last updated: Friday, May 10, 2024

Name: ______________________________

Student ID: ______________________________

This exam is 170 minutes long.

Question:	1	2	3	4	5	6	7	8	9	Total
Points:	0	10	13	15	13	11	17	11	10	100

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

○ Unselected option (completely unfilled)

● Only one selected option (completely filled)

✓ Don't do this (it will be graded as incorrect)

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

■ You can select

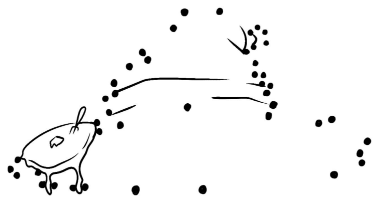
■ multiple squares (completely filled)

Anything you write outside the answer boxes or you ~~cross out~~ will not be graded. If you write multiple answers, your answer is ambiguous, or the bubble/checkbox is not entirely filled in, we will grade the worst interpretation.

Pre-exam activity:
(Just for fun, not graded.)

Connect the dots to form a drawing!
(I wonder what the result will be...)

Bonus: Once you're done, draw something below it!



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* **(10 points)**

Each true/false is worth 0.5 points.

Q2.1 The Caltopian Army designs a secure radio that encodes cryptographic keys into physical keys. To access a specific secure channel, the user inserts a key with a corresponding color.

TRUE or FALSE: This is an example of Consider Human Factors.

[✓ TRUE] FALSE

Solution: True. This is similar to the security keys example shown in lecture. By color-coding physical keys, we're making the system easier to use for the users.

Q2.2 EvanBook Inc. requires the approval of two separate engineers to unlock the server room.

TRUE or FALSE: This is an example of Shannon's Maxim.

TRUE [✓ FALSE]

Solution: False. This is an example of separation of responsibility.

Q2.3 TRUE or FALSE: Stack canaries are the same across different functions within the same program execution.

[✓ TRUE] FALSE

Solution: True (as seen in lecture).

Q2.4 TRUE or FALSE: AES-CBC is often said to act like a stream cipher.

TRUE [✓ FALSE]

Solution: False. AES-CBC does not efficiently support encrypting new data as it streams in, because input has to be padded. For example, assuming 16-bit blocks, if we have 20 bits initially, we have to pad the input to 32 bits before encrypting. Then, if an additional 5 bits arrive, there's no efficient way to "continue" the AES-CBC algorithm and encrypt additional data. (Instead, we would have to decrypt, un-pad, and re-encrypt and re-pad the message with the additional bits encrypted.)

By contrast, AES-CTR does efficiently support encrypting additional data. We can continue running AES encryption with successively higher counters to generate block cipher output, and then perform a bitwise XOR with the additional data that has been streamed in.

Q2.5 True or False: An attacker expects to have to try 2^{196} different values to find a collision in a 256-bit secure cryptographic hash function.

○ True [✓ False]

Solution: False. The attacker expects to have to try 2^{128} different values (see the birthday paradox).

Q2.6 True or False: RSA keys are usually chosen to be either 128 or 256 bits long.

○ True [✓ False]

Solution: False. RSA keys are usually 2048 bits or longer.

Q2.7 True or False: It is possible to have multiple valid certificates for a single website.

[✓ True] ○ False

Solution: True. Multiple certificate authorities can use their private keys to sign a message endorsing the same website's public key.

Q2.8 True or False: `www.google.com` and `google.com` have different origins under the Same-Origin Policy.

[✓ True] ○ False

Solution: True. Same-origin policy runs on string matching, and `www.google.com` and `google.com` are not an exact-string match.

Q2.9 True or False: CSRF tokens are often stored in cookies.

○ True [✓ False]

Solution: False. CSRF tokens are not encoded in cookies. CSRF is vulnerable because the browser automatically attaches cookies in requests, so CSRF tokens need to be stored elsewhere in order to be an effective defense.

Q2.10 TRUE or FALSE: It is possible for a cookie to have both `HttpOnly` and `Secure` set to true.

[✓ TRUE] FALSE

Solution: True. A common misconception is that `HttpOnly` means the message cannot be sent over HTTPS, but this is false. `HttpOnly` means that Javascript cannot access the cookie. `Secure` means that the cookie can only be sent over HTTPS. Both of these can be true at the same time.

Q2.11 TRUE or FALSE: Stored XSS attacks are often more severe than reflected XSS attacks, because users do not need to click on an attacker-controlled link in stored XSS attacks.

[✓ TRUE] FALSE

Solution: True. Stored XSS does not require the user to click on a link crafted by the attacker, since the XSS is stored on the server and could be displayed to the user on one of the server's pages.

By contrast, reflected XSS requires the attacker to place the malicious Javascript directly in the link, which means the user needs to click on the malicious link crafted by the attacker to execute the attack.

Q2.12 TRUE or FALSE: A MITM attacker can force the user's browser to execute malicious JavaScript by tampering with the HTTP response.

[✓ TRUE] FALSE

Solution: True. HTML and Javascript are sent over HTTP, which is sent over the network (TCP/IP). A MITM can change the HTML/Javascript as it's being sent across the network, which would cause the user to receive malicious Javascript. Also, recall that Javascript is executed in the user's browser, so the attacker can inject malicious Javascript and cause the user's browser to execute that Javascript.

Q2.13 TRUE or FALSE: ARP spoofing requires an off-path attacker to correctly guess the source port of the ARP request sender.

TRUE [✓ FALSE]

Solution: False. ARP spoofing works at the link layer, and ports are defined at the higher transport layer.

Q2.14 TRUE or FALSE: The correct order of the DHCP handshake is: Client Discover, DHCP Acknowledgement, Client Request, DHCP Offer.

○ TRUE [✓ FALSE]

Solution: False. Without memorizing the steps, you can reason that the acknowledgement comes last. You can also reason that the offer has to be acknowledged, so it cannot be the last step.

The actual sequence of steps is: Discover, Offer, Request, Acknowledgement.

Q2.15 An attacker who manages to impersonate a WPA2 access point has full access to the contents of the HTTPS requests from the network clients.

○ TRUE [✓ FALSE]

Solution: False. HTTPS is end-to-end secure, so someone impersonating the WPA2 access point cannot learn messages encrypted over HTTPS.

Q2.16 Every BGP AS requires a certificate signed by a certificate authority.

○ TRUE [✓ FALSE]

Solution: False. As seen in the BGP spoofing attack, there isn't really a way to verify the identity of a BGP AS.

For the next two subparts, you are a consultant for a large corporation that wishes to install intrusion detection equipment.

Q2.17 TRUE or FALSE: It would likely be cheaper to install a NIDS instead of installing HIDS.

[✓ TRUE] ○ FALSE

Solution: True. As seen in lecture, a NIDS is cheaper to install, because you only have to install one system to protect the entire network.

Q2.18 TRUE or FALSE: Installing a NIDS would allow the corporation to easily inspect the contents of the employees' HTTPS requests.

○ TRUE [✓ FALSE]

Solution: False. As seen in lecture, HTTPS is end-to-end secure, and the NIDS would be unable to read messages encrypted over HTTPS.

Q2.19 True or False: Double spending on the Bitcoin network is only possible by stealing the private key of another user.

○ True [✓ False]

Solution: False. Double-spending can be achieved by controlling 51% or more of the computing power, which allows the attacker to create a forged blockchain that is longer and accepted as the true blockchain. This can allow the attacker to convince user A that the original blockchain is correct, and convince user B that the forged blockchain is correct. The two different blockchains could spend a single coin in two different ways, thus allowing the attacker to double-spend a coin. Stealing a private key is not necessary for this attack, since the attacker could double-spend their own coin (e.g. sign the transaction with their own private key).

Q2.20 True or False: LLMs are vulnerable to prompt injection attacks because they generally cannot distinguish between input and commands.

[✓ True] ○ False

Solution: True, as mentioned in the LLM lecture. LLMs can have a hard time telling inputs and commands apart, which could allow an attacker to provide an input that's interpreted as a command.

(Note for future semesters: This lecture is not always taught in CS 161, but was taught in Spring 2024.)

Q2.21 (0 points) True or False: `EvanBot is a real bot.`

[✓ True] ○ False

Solution: `True. You've solved the past exams and seen the answer to this already, haven't you?`

Q3 *Memory Safety: Everyone Loves PIE* **(13 points)**

Consider the following vulnerable C code:

```c
1  void cake() {
2    char buf[8];
3    char input[9];
4    int i;
5
6    fread(input, 9, 1, stdin);
7
8    for (i = 8; i >= 0; i--) {
9      buf[i] = input[i];
10   }
11   return;
12 }
13
14 void pie() {
15   char cookies[64];
16
17   // Prints out the 4-byte address of cookies
18   printf("%p", &cookies);
19
20   fgets(cookies, 64, stdin);
21   cake();
22   return;
23 }
```

Stack at Line 6
RIP of `pie`
SFP of `pie`
(1)
RIP of `cake`
(2)
`buf`
(3)
`i`

Assumptions:

- `SHELLCODE` is 63 bytes long.
- ASLR is enabled. All other defenses are disabled.

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

- ○ (1) &p (2) SFP of `cake` (3) `SFP of printf`
- [✓ ● (1) `cookies` (2) SFP of `cake` (3) `input`]
- ○ (1) `cookies` (2) SFP of `cake` (3) `RIP of fgets`
- ○ (1) `RIP of printf` (2) SFP of `printf` (3) input

Solution: Here's the stack diagram. It's not needed to solve this subpart, but to clarify later solutions, we'll label the addresses relative to the address of cookies (which is the one address we know, because of the print statement).

`&cookies + 68`	RIP of `pie`
`&cookies + 64`	SFP of `pie`
`&cookies`	`cookies`
`&cookies - 4`	RIP of `cake`
`&cookies - 8`	SFP of `cake`
`&cookies - 16`	`buf`
`&cookies - 25`	`input`
`&cookies - 29`	`i`

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

- [✓ ● Off-by-one]
- ○ Format string vulnerability
- ○ Signed/unsigned vulnerability
- ○ Time-of-check to time-of-use

Solution: The big clue that an off-by-one attack exists is `buf` being 8 bytes, and `input` being 9 bytes. In particular, the for loop is iterating 9 times and causing 9 bytes of `input` to be copied into the 8-byte `buf` array. This causes the byte directly after `buf` to be overwritten.

There's no format string vulnerability, because in the one and only call to `printf`, the attacker does not control the 0th argument where the percent formatters are placed.

There's no signed/unsigned vulnerability, because the numbers in `fgets` and `fread` are hard-coded, and `i` is never interpreted as an unsigned integer.

There is no time-of-check to time-of-use vulnerability, because the program never pauses (which might cause an input to be correct at time-of-check but incorrect at time-of-use).

In the next two subparts, you will provide inputs to cause `SHELLCODE` to execute with high probability.

Let `OUT` be the output from the `printf` call on Line 18. Assume that you can slice this value (e.g. `OUT[0:2]` returns the 2 least significant bytes of `&cookies`). You may also perform arithmetic on this value (e.g. `OUT[0:2] + 4`) and assume it will be converted to/from the correct types automatically.

Q3.3 (2 points) Provide a value for the `fgets` call on Line 20.

Solution: SHELLCODE

Solution: `buf` and `input` cannot fit the 63-byte shellcode, so `cookies` is the only possible place to put shellcode.

The `fgets` call writes at most 63 bytes into `cookies`, which means that after writing shellcode here, there's no more space to write anything else in `cookies`.

As we'll see in the next subpart, there's nothing else that needs to be placed in `cookies` to complete the exploit.

Q3.4 (5 points) Fill in each blank with an integer to provide an input to the `fread` call on Line 6.

You must put an integer for every blank even if the final slice would be equivalent – for example, you must put both "0" and "7" in the blanks for `OUT[0:7]`, even though `OUT[:7]` is equivalent.

Note that the `+` between terms refers to string concatenation (like in Project 1 syntax), but the minus sign in the third term refers to subtracting from the `OUT[_:_]` value.

```
'A'*________ + OUT[______:______] + (OUT[______:______] - ________)
```

Solution:

`A*'4' + OUT[0:4] + (OUT[0:1] - 16)`

The last blank can also be 25 instead of 16.

`OUT` prints the 4-byte address of `cookies` (which is where we put shellcode).

The for loop causes the 9 bytes of `input` to be copied into `buf`. This means that the byte immediately after `buf` can also be overwritten. This byte is the LSB of the SFP of `cake`.

In the off-by-one attack (as seen in Project 1), we can overwrite the SFP of `cake` to point 4 bytes below the place where we put the address of shellcode.

We can overwrite the SFP to point at the address of `buf`. Then, 4 bytes after the start of `buf`, we can write the address of shellcode.

The first 4 bytes of `buf` are garbage, then the next 4 bytes are `OUT[0:4]`, the address of shellcode. (Note that the slice here doesn't do anything since the output is already 4 bytes, but the question requires we put an integer in every blank.)

The 9th and final byte of input needs to change the SFP of `cake` to point at the address of `buf`. Per the stack diagram, we calculated this to be 16 bytes below the address we leaked. Since we can only overwrite a single byte, we slice out the LSB of the address of `cookies`, which is `OUT[0:1]`, and subtract 16 from this value.

`OUT[0:1] - 25` also works (i.e. last blank could also be 25), since this would cause the SFP to point at `input`. The first 4 bytes of input are also garbage, and the next 4 bytes of input are also the address of shellcode, so this solution also works.

Q3.5 (2 points) Which of these defenses, if enabled by itself, would prevent the exploit (without modifications) from working? For pointer authentication only, assume the program runs on a 64-bit system.

■ Stack canaries ■ Pointer authentication

■ Non-executable pages ☐ None of the above

Solution: Stack canaries: True. The off-by-one attack now overwrites the LSB of the canary, instead of the LSB of the SFP.

Non-executable pages: True. The shellcode was written on the stack, so if non-executable pages were enabled, it would not be possible to execute user-inputted code.

Pointer authentication codes would break the exploit, since we're changing a pointer value (SFP of `cake`) without modifying its corresponding pointer authentication code.

Q3.6 (2 points) Which of these variable values would cause the exploit to break?

○ RIP of `pie` = `0x10c3fa00`

○ RIP of `cake` = `0x10237acf`

[✓ address of `cookies` = `0xffff5fc0`]

[✓ SFP of `cake` = `0xffffcd04`]

Solution: Recall that the SFP of `cake`'s value is the address of the SFP of `pie`. If the SFP of `cake` is `0xffffcd04`, this means the address of the SFP of `pie` is `0xffffcd04`, and the stack looks like this:

`0xffffcd08`	RIP of `pie`
`0xffffcd04`	SFP of `pie`
`0xffffccc4`	`cookies`
`0xffffccc0`	RIP of `cake`
`0xffffccbc`	SFP of `cake` (value `0xffffcd04`)
`0xffffccb8`	`buf`
`0xffffccb4`	`input`
`0xffffccb0`	`i`

The value of the SFP of `cake` is `0xffffcd04`. However, we want to overwrite this value with the address of `buf`, which is `0xffffccb8`. It is no longer possible to perform the off-by-one exploit, since we have to change the 2 least-significant bytes in order to change the address correctly.

At a high level, the problem here is that the LSB of the addresses were close to 0x00, which caused the second-least significant byte to roll over, preventing the off-by-one exploit from working.

If you try drawing out a similar stack diagram with the address of `cookies`'s value set to `0xffff5fc0`:

`0xffff6004`	RIP of `pie`
`0xffff6000`	SFP of `pie`
`0xffff5fc0`	`cookies`
`0xffff5f80`	RIP of `cake`
`0xffff5f7c`	SFP of `cake` (value `0xffff6000`)
`0xffff5f78`	`buf`
`0xffff5f74`	`input`
`0xffff5f70`	`i`

we see that this is also broken, so both answers were accepted for credit. (This was not originally intended as a right answer, but became correct after the size of `cookies` was changed from 16 to 64.)

The two options with RIP values show addresses in the code section, which are irrelevant to our exploit.

Q4 *Memory Safety: Breaking Bot* (15 points)

EvanBot has decided to manufacture an industrial amount of pancakes and has gone to Walter White for help. Consider the following vulnerable C code:

```c
 1 typedef struct {
 2   char name[12];
 3   void (*task)(); // task is a pointer to a function
 4 } person;
 5 
 6 /* implementations not shown */
 7 void cook() { ... };
 8 void sell() { ... };
 9 
10 void rv() {
11     person *p;
12     char *formula;
13     char saved_name[12];
14 
15     p = (person *) malloc(sizeof(person));
16     fgets(p->name, 12, stdin);
17     if (strcmp(p->name, "Walter") == 0) {
18         p->task = cook;
19     } else {
20         p->task = sell;
21     }
22     strlcpy(saved_name, p->name, 12);
23     free(p);
24 
25     formula = (char *) malloc(17);
26     fgets(formula, 17, stdin);
27 
28     p->task();
29 }
```

Stack at Line 15

RIP of `rv`
(1)
(2)
(3)
`saved_name`

Assumptions:

- The heap starts at address `0x30000000` and grows upwards.
- `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.
- Your goal is to call `system("/bin/sh")`, which will spawn a shell.
- The function `system` is located in memory at address `0x08120161`.
- The address of `saved_name` is `0xffffca10`.
- Non-executable pages are enabled. All other defenses are disabled.

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

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

[✓ (1) SFP of `rv`]	(2) `p`	(3) `formula`
○ (1) SFP of `rv`	(2) `formula`	(3) `p`
○ (1) `formula`	(2) `p`	(3) RIP of `cook`
○ (1) `formula`	(2) RIP of `cook`	(3) SFP of `cook`

Solution: For the rest of the solution, we'll draw out not just the stack, but also the heap here:

Stack:

`0xffffca28`	RIP of `rv`
`0xffffca24`	SFP of `rv`
`0xffffca20`	`p` (value: `0x30000000`)
`0xffffca1c`	`formula` (value: `0x30000000`
`0xffffca10`	`saved_name`

Heap:

Address	value after Line 15	value after Line 25
`0x3000000c`	`p->task`	`formula[12:16]`
`0x30000008`	`p->name[8:12]`	`formula[8:12]`
`0x30000004`	`p->name[4:8]`	`formula[4:8]`
`0x30000000`	`p->name[0:4]`	`formula[0:4]`

Q4.2 (1 point) Which vulnerability is present in the code?

○ Off-by-one

○ Signed/unsigned vulnerability

○ Format string vulnerability

[✓ Use after free]

Solution: The big clue that this is a use-after-free attack is the use of `free` on Line 23. In particular, the block of heap memory that `p` is pointing at is freed on Line 23. But, at Line 28, we try to dereference `p` anyway to use that memory after it's been freed. This will cause issues, because Lines 25-26 have caused that same block of heap memory to be overwritten with other (attacker-chosen) input.

There's no off-by-one vulnerability in this code. The two calls to `fgets` both write in-bounds, and the call to `strncpy` also writes in-bounds.

There's no format string vulnerability. There's no call to `printf` in the code.

There's no signed/unsigned vulnerability. All numerical inputs throughout the code are hard-coded.

Q4.3 (1 point) What address is stored in the variable `formula` after Line 25 is executed?

- ○ `0xffffca10`
- ○ `0xffffca1c`
- [✓ `0x30000000`]
- ○ `0x30000010`

Solution: After Line 15, the bytes between `0x30000000` and `0x3000000f` (inclusive) are used to store a `person` struct.

After Line 23, these bytes are freed up.

After Line 25, we call `malloc` again, the bytes between `0x30000000` and `0x30000010` (inclusive) are used again to store a character array (that `formula` is pointing at).

Per the assumptions, `malloc` always allocates starting at the lowest possible address with enough free space, which is why both allocations start allocating memory at `0x30000000`.

In the next two subparts, provide inputs that would cause the program to execute `system("/bin/sh")`.

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

Q4.4 (4 points) Input to `fgets` at Line 16:

Solution: `"\x14\xca\xff\xff" + "/bin/sh"`

First, note that Line 16 and Line 22 together allow the user to overwrite `saved_name`.

This input puts the argument `"/bin/sh"` at the bottom of the stack (`saved_name`), so that the call to `p->task()` (overwritten to be `system` in the next part) will look for arguments and find `"bin/sh"`.

One thing we need to note is that string arguments are passed on the stack as pointers to character arrays, so instead of directly writing `"/bin/sh"`, we need to put this string elsewhere in memory (e.g. later in `saved_name`), and then at the bottom of the stack, put the address of the string, as shown below:

Address	Variable	Value at Line 28
`0xffffca28`	RIP of `rv`	
`0xffffca24`	SFP of `rv`	
`0xffffca20`	`p`	`0x30000000`
`0xffffca1c`	`formula`	`0x30000000`
`0xffffca14`	`saved_name[4:12]`	`/bin/sh\x00`
`0xffffca10`	`saved_name[0:4]`	`0xffffca14`

Now, when we call `p->task()` (which is pointing at `system`), we will start the protocol for calling a function. Normally, the first step is to push arguments on the stack, but `p->task()` passes in no arguments, so no arguments are pushed on the stack. However, we wrote the `"/bin/sh"` argument on the stack in the same place the program would be looking for arguments (the very bottom of the previous stack frame, before we create the new stack frame).

Then, we run the rest of the function prologue as normal, which will create a new stack frame to execute `system`, with the arguments already written on the stack beforehand.

Note that unlike other exploits, we are not exploiting a function return by overwriting the RIP. We are changing a function pointer so that it points at `system`, and allowing the program to naturally call that function pointer, which causes `system` to be called using the usual calling convention process (e.g. push arguments, function prologue, etc.).

Q4.5 (6 points) Input to `fgets` at Line 26:

Solution: `'A'*12 + "\x61\x01\x12\x08"`

Solution: Remember from the first stack diagram that `formula` and `p` are both pointing at the same block of memory in the heap. Therefore, this input to `formula` is actually allowing us to overwrite `p`.

We overwrite the first 12 bytes with garbage, representing `p->name` in memory. Then, we overwrite the last 4 bytes, representing `p->task`, with the address of `system`.

As a side note, Lines 17-21 and the ability to overwrite `p->name` and don't do anything useful for the attacker, since they have no control over what `cook` or `sell` actually do.

Q4.6 (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: Yes, the exploit is still possible, because we never overwrite above the local variables, and the stack canary is located above the local variables.

In particular, note that none of the `fgets` or `strncpy` calls write out-of-bounds.

Q4.7 (1 point) Would it still be possible for your exploit to work with high probability (without modifications) if ASLR is enabled, assuming the code section is randomized?

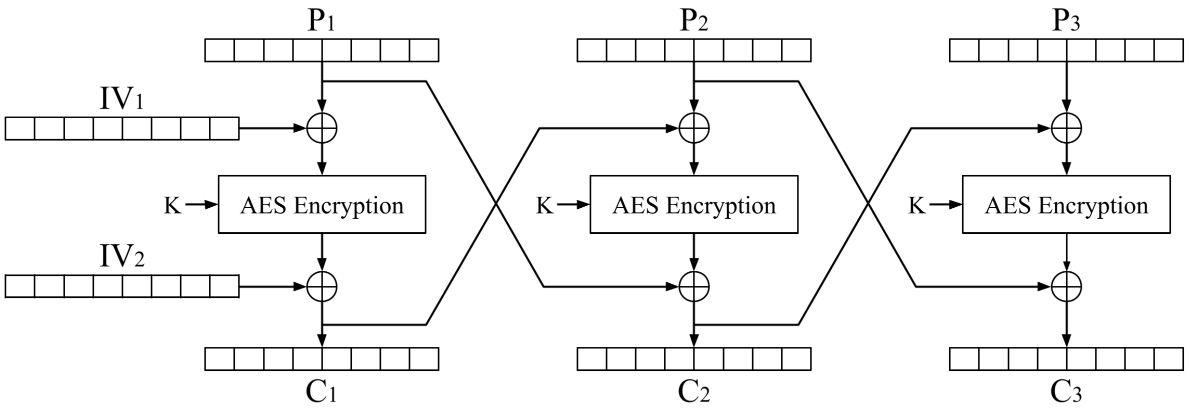
○ Yes [✓ No]

Solution: No. We needed to hard-code the address of `system` in our exploit, and the address of `system` in memory would be different every time if ASLR is enabled (including the code section).

Q5 *Symmetric Cryptography: AES-ROVW* (13 points)

EvanBot designs the AES-ROVW mode of operation as follows:

$$C_1 = E_K(P_1 \oplus IV_1) \oplus IV_2$$
$$C_i = E_K(P_i \oplus C_{i-1}) \oplus P_{i-1} \quad \text{(for } i \geq 2\text{)}$$



Q5.1 (1 point) Select the decryption formula for P_i, for $i \geq 2$.

○ $P_i = D_K(C_{i-1} \oplus P_i) \oplus C_i$

○ $P_i = D_K(C_i) \oplus C_{i-1} \oplus P_{i-1}$

● $P_i = D_K(C_i \oplus P_{i-1}) \oplus C_{i-1}$

○ $P_i = D_K(C_i \oplus C_{i-1}) \oplus P_{i-1}$

Solution: Using algebra (XORing both sides, or decrypting both sides), we get:

$$\begin{aligned} C_i &= E_K(P_i \oplus C_{i-1}) \oplus P_{i-1} \\ C_i \oplus P_{i-1} &= E_K(P_i \oplus C_{i-1}) \\ D_K(C_i \oplus P_{i-1}) &= P_i \oplus C_{i-1} \\ D_K(C_i \oplus P_{i-1}) \oplus C_{i-1} &= P_i \end{aligned}$$

Q5.2 (1 point) Select all true statements.

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

Solution:

A is false. Looking at the encryption formula or the diagram, we see that computing C_i requires knowing C_{i-1}.

B is false. Looking at the decryption formula or the diagram, we see that computing P_i requires knowing P_{i-1}.

Q5.3 (3 points) Select all true statements.

- ■ AES-ROVW is IND-CPA secure if IV_1 and IV_2 are independently randomly generated.
- ■ AES-ROVW is IND-CPA secure if IV_1 is randomly generated and $IV_2 = H(IV_1)$.
- ■ AES-ROVW is IND-CPA secure if IV_2 is randomly generated and $IV_1 = H(IV_2)$.
- ☐ None of the above.

Solution: This scheme looks most similar to AES-CBC, where we pass in a random IV before the first block cipher encryption. This causes the block cipher output to be unpredictable, and then we chain that unpredictable block cipher output into the input of the next block cipher.

In all three choices, IV_1 is being randomly-generated (in C, we're deriving IV_1 by hashing another randomly-chosen value), and IV_1 is passed into the first block cipher input, so this should intuitively cause all subsequent block cipher inputs/outputs to be unpredictable for an attacker.

(This isn't a formal proof, but the intuition you can use to approach the question in an exam setting.)

Alice has a two-block message (P_1, P_2). Alice encrypts this message with AES-ROVW to get (IV_1, IV_2, C_1, C_2).

Mallory, a MITM attacker, intercepts Alice's ciphertext (IV_1, IV_2, C_1, C_2), and Mallory **knows the original plaintext value** (P_1, P_2).

Mallory wants to change the ciphertext to $(IV_1', IV_2', C_1', C_2')$, such that when Bob receives the modified ciphertext and decrypts it, he sees (P_1', P_2'), a malicious message of Mallory's choosing.

In the next four subparts, give the values for Mallory's tampered ciphertext $(IV_1', IV_2', C_1', C_2')$. Select as many options as you need.

Q5.4 (2 points) IV_1' is equal to these values, XORed together.

For example, if you think $IV_1' = P_2 \oplus C_2$, then bubble in P_2 and C_2.

■ IV_1	■ P_1	■ P_1'	☐ C_1
☐ IV_2	☐ P_2	☐ P_2'	☐ C_2

Solution: $IV_1' = IV_1 \oplus P_1 \oplus P_1'$

Q5.5 (2 points) IV_2' is equal to these values, XORed together.

☐ IV_1	☐ P_1	☐ P_1'	☐ C_1
■ IV_2	■ P_2	■ P_2'	☐ C_2

Solution: $IV_2' = IV_2 \oplus P_2 \oplus P_2'$

Q5.6 (2 points) C_1' is equal to these values, XORed together.

☐ IV_1	☐ P_1	☐ P_1'	■ C_1
☐ IV_2	■ P_2	■ P_2'	☐ C_2

Solution: $C_1' = C_1 \oplus P_2 \oplus P_2'$

An alternate solution (must also choose the alternate solution for 5.7): $C_1' = C_0 \oplus P_1 \oplus P_2'$

Q5.7 (2 points) C_2' is equal to these values, XORed together.

☐ IV_1	■ P_1	■ P_1'	☐ C_1
☐ IV_2	☐ P_2	☐ P_2'	■ C_2

Solution: $C_2' = P_1' \oplus C_2 \oplus P_1$

If the previous subpart chose the alternate solution: $C_2' = P_1' \oplus C_1 \oplus P_0$

Solution:

Since we're dealing with P_1 and P_2 specifically, we can write down these two decryption formulas, and manipulate the attacker-controlled terms IV_1, IV_2, C_1, C_2 such that the formulas output P_1' and P_2'.

$$\begin{aligned} D_K(C_1 \oplus IV_2) \oplus IV_1 &= P_1 \\ D_K(C_2 \oplus P_1) \oplus C_1 &= P_2 \end{aligned}$$

There are two twists that make this derivation kind of tricky:

First, C_1 appears in both equations, so if we choose to tamper with C_1 in one of the equations, we'll need to go to the other equation and plug in the modified C_1' and make any necessary fixes to account for the modified C_1'.

Second, P_1 in the second equation comes from the P_1 value that Bob decrypts using the first formula, so when we tamper the first formula to output P_1', Bob will also be using P_1' in the second equation.

We can XOR both sides of the first equation by P_1 to "cancel out" P_1. Then, we can XOR both sides of the first equation by P_1' to "introduce" P_1':

$$\begin{aligned} D_K(C_1 \oplus IV_2) \oplus IV_1 &= P_1 \\ D_K(C_1 \oplus IV_2) \oplus \underbrace{IV_1 \oplus P_1 \oplus P_1'}_{IV_1'} &= P_1 \oplus P_1 \oplus P_1' \\ &= P_1' \end{aligned}$$

Matching terms between the original and modified equation, we see that both are equal if we set $IV_1' = IV_1 \oplus P_1 \oplus P_1'$. This answers Q5.4.

Now, we can use a similar trick to change P_2 to P_2', i.e. we XOR both sides by $P_2 \oplus P_2'$ to "cancel out" P_2 and "introduce" P_2':

$$\begin{aligned} D_K(C_2 \oplus P_1) \oplus C_1 &= P_2 \\ D_K(C_2 \oplus P_1) \oplus \underbrace{C_1 \oplus P_2 \oplus P_2'}_{IV_2'} &= P_2 \oplus P_2 \oplus P_2' \\ &= P_2' \end{aligned}$$

Matching terms again, we see that both are equal if we set $C_1' = C_1 \oplus P_2 \oplus P_2'$. This answers Q5.6.

First tricky part of the question: Since we changed C_1 to $C_1' = C_1 \oplus P_2 \oplus P_2'$, we have to go back and fix our first equation, which is now using C_1' instead:

$$D_K(C_1 \oplus IV_2) \oplus \underbrace{IV_1 \oplus P_1 \oplus P_1'}_{IV_1'} = P_1'$$
$$D_K(\underbrace{C_1 \oplus P_2 \oplus P_2'}_{C_1'} \oplus IV_2) \oplus \underbrace{IV_1 \oplus P_1 \oplus P_1'}_{IV_1'} = ???$$

We can fix this by changing IV_2 to IV_2' such that the extra $P_2 \oplus P_2'$ term is cancelled out:

$$\begin{aligned} &D_K(\underbrace{C_1 \oplus P_2 \oplus P_2'}_{C_1'} \oplus \underbrace{IV_2 \oplus P_2 \oplus P_2'}_{IV_2'}) \oplus \underbrace{IV_1 \oplus P_1 \oplus P_1'}_{IV_1'} \\ &= D_K(C_1 \oplus IV_2) \oplus \underbrace{IV_1 \oplus P_1 \oplus P_1'}_{IV_1'} \\ &= P_1' \end{aligned}$$

This tells us that $IV_2' = IV_2 \oplus P_2 \oplus P_2'$. This answers Q5.5.

Second tricky part of the question: Since we changed P_1 to P_1', we need to revisit the second equation, which is now using P_1' instead of P_1:

$$D_K(C_2 \oplus P_1) \oplus \underbrace{C_1 \oplus P_2 \oplus P_2'}_{IV_2'} = P_2'$$
$$D_K(C_2 \oplus P_1') \oplus \underbrace{C_1 \oplus P_2 \oplus P_2'}_{IV_2'} = ???$$

We can fix this by changing C_2 to C_2' such that P_1' is “cancelled out” and P_1 is “reintroduced”:

$$
\begin{aligned}
&D_K(\underbrace{C_2 \oplus P_1 \oplus P_1'}_{C_2'} \oplus P_1') \oplus \underbrace{C_1 \oplus P_2 \oplus P_2'}_{IV_2'} \\
&= D_K(C_2 \oplus P_1) \oplus \underbrace{C_1 \oplus P_2 \oplus P_2'}_{IV_2'} \\
&= P_2'
\end{aligned}
$$

This tells us that $C_2 = C_2 \oplus P_1 \oplus P_1'$. This answers Q5.7, and we are done.

Q6 *Asymmetric Cryptography: Plentiful Playlists* (11 points)

Alice and Bob wish to create a music playlist for their upcoming road trip to Pittsburgh. Alice and Bob each come up with a list of n songs. Some (but not all) of the songs might appear on both lists.

Alice and Bob want to learn the songs that are on both lists, without revealing their individual lists to each other.

For example, say Alice's list has the songs "One Little Victory" and "Motivation", while Bob's list has "Motivation" and "Give It All". Both Alice and Bob should be able to learn that both lists contain "Motivation". However, Alice should learn nothing about Bob's other songs, and vice versa.

They decide to use the following protocol, but need your help to fill in the blanks. Assume that p is a large prime, like those used in Diffie-Hellman, and each song is represented as an integer $\bmod p$.

1. Alice and Bob denote their lists as $a_1, \ldots, a_n$ and $b_1, \ldots, b_n$ respectively.
2. Alice generates a random number $r \pmod p$.
3. Bob generates a random number $k \pmod p$.
4. For each element a_i in Alice's list, Alice sends $\texttt{STEP4}_i = a_i^r \pmod p$ to Bob.
5. For each element $\texttt{STEP4}_i$ received from Alice in the previous step, Bob computes $\texttt{STEP5}_i =$ ____, and sends $\texttt{STEP5}_i$ to Alice.
6. For each element $\texttt{STEP5}_i$ received from Bob in the previous step, Alice computes $\texttt{STEP6}_i =$ ____.
7. For each element b_i in Bob's list, Bob sends $\texttt{STEP7}_i =$ ____ to Alice.
8. Alice compares the set of $\texttt{STEP6}_i$ (for all i) and $\texttt{STEP7}_j$ (for all j) to find matching pairs, i.e. all (i, j) such that $\texttt{STEP6}_i = \texttt{STEP7}_j$. For the pairs that match, Alice finds the corresponding a_i.
9. Alice sends the set of all matching a_i to Bob over a secure channel.

Q6.1 (2 points) Replace the blank in Step 5.

- ○ $(\texttt{STEP4}_i) - k \pmod p$
- ○ $\texttt{STEP4}_i + k \pmod p$
- ○ $k \cdot \texttt{STEP4}_i \pmod p$
- [✓ ● $(\texttt{STEP4}_i)^k \pmod p$.]

Q6.2 (2 points) Define r^{-1} such that $g^{r \cdot r^{-1}} \equiv g \pmod p$ for all $g \pmod p$. Assume that such an r^{-1} always exists.

Replace the blank in Step 6.

- [✓ ● $(\texttt{STEP5}_i)^{r^{-1}} \pmod p$]
- ○ $(\texttt{STEP5}_i)^{r \cdot r^{-1}} \pmod p$
- ○ $r^{-1} \cdot (\texttt{STEP5}_i) \pmod p$
- ○ $(\texttt{STEP5}_i)^r \pmod p$

Q6.3 (2 points) Replace the blank in Step 7.

○ $b_i^r \pmod{p}$

○ $b_i + k \pmod{p}$

[✓ ● $b_i^k \pmod{p}$]

○ $(b_i^r)^k \pmod{p}$

Solution:

Step 5 gives Alice $a_i^{rk} \pmod{p}$ for all songs in Alice's list.

In Step 6, Alice will "cancel out" the r by computing $(a_i^{rk})^{r^{-1}} = a_i^k \pmod{p}$, for all songs in Alice's list.

In Step 7, Bob will send $b_i^k \pmod{p}$ for all songs in Bob's list.

If song i in Alice's list matches song j in Bob's list, we'll get $a_i^k = b_j^k$, which allows Alice to learn which songs match.

Q6.4 (1 point) Which option best explains why Alice and Bob cannot learn the songs in the other person's list (besides the songs that are in both lists)?

○ The values are encrypted with a shared symmetric key derived via Diffie-Hellman.

[✓ ● Alice does not know k (Bob's random number), and Bob does not know r (Alice's random number).]

○ Exponentiation is commutative modulo p, i.e. $(x^a)^b \equiv x^{ab} \equiv (x^b)^a \pmod{p}$.

○ Factoring the product of two large primes is considered to be difficult.

Solution: Alice cannot learn the other songs in Bob's list, because given $b_i^k \pmod{p}$, Alice has no way to derive b_i. In order for Alice to derive b_i, she would need to know k, so that she can derive $k^{-1} \pmod{p}$ and compute $(b_i^k)^{k^{-1}} \equiv b_i \pmod{p}$.

A is false because there is no shared symmetric key ever derived in this question (i.e. no value is ever explicitly derived as "the key").

C is a true statement, but it does not explain why Alice and Bob are unable to learn each other's songs.

D is false because the factoring problem is not used in this protocol (contrast with RSA, where we multiply together two large primes, and rely on the fact that factoring this number is hard).

Q6.5 (2 points) What information is leaked to a third-party eavesdropper? Select all that apply.

☐ All of Alice's songs

■ The number of songs in Alice's list

☐ All of Bob's songs

■ The number of songs in Bob's list

☐ All songs on both lists

☐ None of the above

Solution:

A: False. An attacker does see $a_i^r \pmod p$, but because the attacker does not know r, they cannot derive r^{-1} and compute $(a_i^r)^{r^{-1}}$ to retrieve the values a_i.

B: False. An attacker does see $b_i^k \pmod p$, but because the attacker does not know k, they cannot derive k^{-1} and compute $(b_i^k)^{k^{-1}}$ to retrieve the values b_i.

C: False. In the final step, Alice sends the matching a_i to Bob over a secure channel.

D: True. An attacker can see the number of values $a_i^r \pmod p$ sent to Bob in Step 4, even though they don't know what the values are.

E: True. An attacker can see the number of values $b_i^k \pmod p$ sent to Alice in Step 7, even though they don't know what the values are.

Q6.6 (2 points) In the current scheme, a third-party eavesdropper learns some information about the number of songs on both lists.

Which option(s) prevent the eavesdropper from learning any information about the number of songs on both lists? Assume there are ℓ songs on both lists. Reminder: There are n songs on each list.

- ■ Add $n - \ell$ dummy values to the message in Step 9.
- ☐ Add $\min(n, \ell)$ dummy values to the message in Step 9.
- ☐ Add $\max(n, \ell)$ dummy values to the message in Step 9.
- ☐ Add $n + \ell$ dummy values to the message in Step 9.
- ☐ None of the above

Solution: For all options, remember that in Step 9, a secure channel still leaks the length of the message.

A: True. This forces Step 9 to always send $\ell + (n - \ell) = n$ items, which prevents an attacker from learning ℓ.

B: False. We know that $\ell \leq n$, so Step 9 will send $\ell + \min(n, \ell) = \ell + \ell = 2\ell$ items, which still leaks ℓ.

C: False. We know that $\ell \leq n$, so Step 9 will send $\ell + \max(n, \ell) = \ell + n$ items, which still leaks ℓ.

D: False. This causes Step 9 to send $\ell + n + \ell = n + 2\ell$ items, which still leaks ℓ.

Q6.7 (0 points) **A+ Question: This subpart is worth no points and is considerably more difficult than the rest of the exam. If you correctly solve this subpart and are in the "A" grade bin at the end of the semester, you may be moved up to the "A+" bin.** .

In this scenario, you are controlling Alice who is running the aforementioned protocol with Bob. Design an attack to recover Bob's ephemeral key $k \pmod{p-1}$, and by extension reveal his entire set.

Assumptions:

1. Assume the correct answers were selected for the first three subparts (if you did not select the correct responses, you will receive no credit for this subpart).
2. You may choose an arbitrary set of elements for Alice, and also deviate from the protocol in any of Alice's steps – for example, you may set $r = 1$ in Step 2. Bob will follow the protocol honestly and will not try to actively detect any sort of attack.
3. $p = p_1 p_2 \cdots p_n + 1$, where each individual prime p_i is **small enough that you can solve the discrete logarithm problem** $\mod p_i$. The attacker knows this factorization.

   Clarification after exam: The ability to solve the discrete logarithm $\pmod{p_i}$ *should instead be: "p_i is very small, such that you may feasibly brute force p_i possibilities to find a discrete logarithm".*
4. k is invertible modulo $p - 1$.

HINT: Use the Chinese Remainder Theorem.

Solution:

The key to this attack in the "for-loop" in Step 4, where Alice sends $a_i^r \pmod p$ and gets back $(a_i^r)^k \pmod p$ for all i. By modifying the order (smallest k such that $g^k = 1 \pmod p$) of the underlying a_i^r element, we can leak the value of k modulo the order. By collecting a set of coprime orders and their respective values for k, we can use CRT to recover the full value of k.

It is important to note that simplying evaluating $(a_i)^k \bmod p_i$ and solving for k does **not** work here. This is because we would end up finding $k \pmod{p_i - 1}$.

To do this, we need a way to get an element of order p_i for each individual p_i. This is most easily done by finding a generator g and setting $r = 1$ and

$$a_i = g^{\frac{p-1}{p_i}}$$

When we send a_i and receive $y = a_i^k$, there are only p_i many possibilities for y. We then solve the discrete log for y to recover $k \pmod{p_i}$. Note that technically we are still $\pmod p$, but since there's only p_i possibilities to check, this is within the range of an attacker. (The setup says the attacker can solve $\pmod{p_i}$, which is imprecise here since we are solving over a order p_i cyclic group and not a modular field. The cyclic subgroup generated does not lend itself to efficient modular arithmetic-only attacks like the general number field sieve, and rather relies on generic group algorithms. However, the intention for the question was that p_i is so small that these would both be feasible for the attacker).

1. Set $r = 1$.
2. Set $a_i = g^{\frac{p-1}{p_i}}$ for $i \in [1, n]$.
3. After Step 4, we have a_i^k for $i \in [1, n]$. For each of these values, find $k \bmod p_i$ as aforementioned.
4. We now have $k \pmod{p_1}, k \pmod{p_2}, \ldots, k \pmod{p_n}$. Solve this system using the Chinese Remainder Theorem to recover $k \pmod{p_1 p_2 \cdots p_n} = k \pmod{p-1}$.
5. Find $k^{-1} \pmod{p-1}$, which exists per the assumptions.
6. For each b_i^k sent by Bob, evaluate $(b_i^k)^{k^{-1}} \equiv b_i \pmod p$.

Q7 *Web Security: Suspicious SQL* (17 points)

A bank website, `bank.com`, decides to test out a new form of user authentication.

- **When a user signs up**: The user chooses an alphanumeric username. The bank securely gives the user a symmetric key (known only to the user and the bank).

  The bank has a SQL table named `keys`, which maps usernames to keys. The table has two columns: `username` (of type string) and `key` (of type integer).
- **When a sender wants to send money**: the sender encrypts the recipient's username with the sender's key, and makes a GET request to:

  `www.bank.com/transfer?sender=______&recipient=______`

  The first blank contains the sender's username. The second blank contains the recipient's username, encrypted with the sender's key.
- **When the bank receives a GET request**: the bank first runs this SQL query:

  `SELECT key FROM keys WHERE username='______'`

  where the blank is replaced with the first URL parameter.

  Then, the bank will use the key returned by the query to decrypt the the recipient's username.

  Finally, the bank transfers money from the sender (whose username comes from the first URL parameter) to the recipient (whose username comes from the second URL parameter, decrypted).

Note: For this entire question, you do not need to consider URL escaping.

Q7.1 (1 point) If requests to `bank.com` are made over HTTP, which of these values can an on-path attacker see? Select all that apply.

■ Sender's IP address ☐ Recipient's plaintext username

■ Sender's plaintext username ☐ None of the above

Solution:

A: True. HTTP runs on top of TCP/IP, so the IP address is unaffected by the fact that we're using HTTP.

B: True. HTTP is unencrypted, so the URL parameters, including the unencrypted sender username, are visible.

C: False. HTTP is unencrypted, but the recipient's username is encrypted by the protocol, so an on-path attacker cannot see it.

Q7.2 (1 point) If requests to `bank.com` are made over HTTPS, which of these values can an on-path attacker see? Select all that apply.

- ■ Sender's IP address
- ☐ Sender's plaintext username
- ☐ Recipient's plaintext username
- ☐ None of the above

Solution:

HTTPS encrypts the URL parameters (since that's inside the HTTPS payload), so neither username is visible to an attacker.

However, HTTPS is still running on top of TCP/IP, so the IP address, which is in an "outer" layer of headers wrapped around the HTTPS packet, is still unaffected by the HTTPS/TLS encryption.

For the next four subparts: Mallory (username: `mallory`) wants to cause the user Bob (username: `bob`) to run the Javascript function `hack()`, once he clicks on a `bank.com` link provided by Mallory.

In each subpart, select whether Mallory's attack is possible. If you select "Yes," provide the URL parameters (`sender` and `recipient`) in the link that Bob clicks. In your answer(s), you may use Mallory's key K_{mallory} and the encryption function $\mathsf{Enc}(\cdot, \cdot)$, where the first argument is the key and the second argument is the plaintext.

Q7.3 (3 points) For this subpart only: If the decrypted recipient username does not exist in the `keys` table, the bank will return an HTML page with the text:

"`____` does not exist"

replacing the blank with the decrypted recipient username.

Can Mallory cause Bob to call `hack()`?

[✓ Yes] No

If you selected "Yes":

`sender`:

Solution: mallory

`recipient`:

Solution: $\mathsf{Enc}(K_{\text{mallory}},$ `<script>hack()</script>`$)$

Solution: The server will look up Mallory's key, since the sender field is set to Mallory. Then, the server uses Mallory's key to decrypt the recipient field and gets `<script>hack()</script>`. This is not a valid username, since the bank enforces alphanumeric usernames only.

As a result, the bank will return an HTML page with the text: "`<script>hack()</script> does not exist`", which will cause Bob's browser to run Javascript calling `hack()`.

Q7.4 (3 points) For this subpart only: if the decrypted recipient username does not exist in the `keys` table, the bank will return an HTTP 404 response.

Can Mallory cause Bob to call `hack()`?

○ Yes [✓ No]

If you selected "Yes":

`sender`:

Solution: N/A

`recipient`:

Solution: N/A

Solution: The attack from the previous part is no longer possible, since the bank is only returning an HTTP 404 response, without any actual content specified by the attacker in the URL.

Q7.5 (1 point) What type of attack is Mallory trying to execute in the previous two subparts?

○ Stored XSS ○ SQL injection

[✓ Reflected XSS] ○ CSRF

Solution: Mallory wants Bob to click on a link with Javascript in the URL parameter, such that the page returned by the bank contains the same Javascript that was in the URL parameter. This is a reflected XSS attack.

Note that this is not a CSRF attack, since we're not relying on the fact that Bob is sending any cookies. (In fact, cookies aren't used in the question at all.)

Q7.6 (1 point) Select the true statement about Mallory's attack (assuming it succeeds).

Assume `mallory.com` is a website controlled by Mallory.

- ● The `hack()` function runs with the origin of `bank.com`.
- ○ The `hack()` function runs with the origin of `mallory.com`.
- ○ `hack()` is able to read cookies with the HttpOnly flag set.
- ○ Instead of making Bob click on a `bank.com` link, Mallory could make Bob click on a link like `mallory.com/hack`, and the attack would be the same.

Solution: Reflected XSS is dangerous because it allows the attacker's Javascript to run with the bank's origin, since the Javascript is being copied onto the HTML page and returned by the bank.

C is false because Javascript cannot read cookies with the HttpOnly flag set (by definition).

D is false, because if Mallory's webpage returned the same Javascript, that Javascript would be running with Mallory's origin instead of the bank's origin.

In the rest of the question, Mallory wants to create a list of one or more URLs such that when **Mallory** clicks on every URL in the list, one after the other, the bank sends money from Bob to Mallory.

Q7.7 (1 point) Mallory thinks that she could create a single URL to execute this attack.

Why is it not possible to execute this attack with a single URL?

- ● The SQL injection has to be placed in `sender`, but this will cause the sender username to be invalid.
- ○ The SQL injection has to be placed in `recipient`, but this will cause the recipient username to be invalid.
- ○ The SQL injection has to be placed in both `sender` and `recipient`, but this causes both fields to be invalid.
- ○ The SQL injection needs to be split across two separate URLs.

Solution: As specified in the question, the SQL query accepts user input from the `sender` URL parameter, which eliminates B and C. There's no reason we would need to split the injection across two separate URLs, which eliminates D.

Q7.8 (6 points) Construct two URLs, such that when Mallory clicks on the first URL, and then the second URL, the bank will send money from username `bob` to username `mallory`.

You may use K_{mallory} and the encryption function $\mathsf{Enc}(K, M)$ in your answer. If a value could be anything, you must write the word "`anything`" in the box.

First URL:

`sender`:

Solution: `'; UPDATE keys SET key=`$K_{\texttt{mallory}}$` WHERE username='bob';--`

`recipient`:

Solution: `anything`

Second URL:

`sender`:

Solution: bob

`recipient`:

Solution: $\mathsf{Enc}(K_{mallory}, \texttt{mallory})$

Solution: The first link, when plugged into the SQL query, creates:

`SELECT key FROM keys WHERE username='';`

`UPDATE keys SET key=`$K_{\texttt{mallory}}$` WHERE username='bob';--'`

The injected query will change the `keys` table, such that Bob is now associated with Mallory's key.

Since Bob is now associated with Mallory's key, the second link claims to be Bob in the sender field, and uses Mallory's key (which the server thinks is Bob's key) to encrypt the recipient, Mallory.

Q8 *DNS: Check Please* (11 points)

Suppose that in DNS, we introduce a single, additional name server that can be used to check if the records returned by other name servers are correct.

Every time the user makes a DNS query, the user also sends the same query to the check server. The user receives the answer directly from the check server, and can compare that answer against the answer received from the other name servers.

Q8.1 (1 point) The zone of the check server must be set to `.` (root). Otherwise, ____ would cause records from the check server to be rejected. What term goes in the blank?

- ○ Kaminsky attack
- ○ NSEC3 hashing
- [✓ Bailiwick checking]
- ○ Glue validation

Solution: Recall that bailiwick checking prevents a name server from providing answers outside of its zone (e.g. the `.com` name server cannot return a record for `www.cs161.org`). Since the check server needs to return answer records for every record, we need to set its zone to root.

Q8.2 (1 point) Is the Kaminsky attack still possible when users also issue a request to the check server for every query?

- [✓ Yes, assuming the user doesn't validate glue records with the check server.]
- ○ Yes, even if the user validates glue records with the check server.
- ○ No, because the check server is in the root zone.
- ○ No, because in the Kaminsky attack, the malicious record is returned in the Answer section.

Solution: Remember that in the Kaminsky attack, the attacker tricks the user into making requests for non-existent domains like `fake1.cs161.org`. In the response, the attacker adds a malicious glue record in the Additional section with the IP address of `www.cs161.org`.

If the user validates glue records with the check server, the attacker would not be able to spoof a response with a malicious glue record (since the check server would reveal that this record is incorrect).

However, if the user does not validate glue records, then the check server verifying the actual answer (i.e. `fake1.cs161.org` doesn't exist) would not affect the additional record being added to the cache.

For the rest of question, suppose we're now using DNSSEC instead of DNS.

Q8.3 (2 points) Which of these options, by itself, would ensure that every recursive resolver trusts the check server? Select all that apply.

- ■ The root name server provides a DS and RRSIG record endorsing the check server.
- ☐ The check server sends an RRSIG record over its own public key.
- ☐ The check server sends an RRSIG record over every record it sends.
- ■ The check server's public key is hard-coded in all resolvers.
- ☐ None of the above

Solution:

A: True. Resolvers implicitly trust the root name server. DS and RRSIG records are used to endorse trust to another name server, so this would allow all resolvers to trust the check server as well.

B: False. Anybody can sign their own public key (e.g. an attacker could pretend to be the check server and sign their own public key).

C: False. The check server signing records is not sufficient; we need some way to verify the check server's public key.

D: True. This is similar to how the root server is implicitly trusted by all resolvers.

For the rest of the question, suppose we're using DNSSEC, and every client trusts the check server.

Q8.4 (1 point) Which record type(s) does the check server need to send to securely answer the user's query? Select all that apply.

- ■ A type record
- ☐ NS type record
- ■ RRSIG type record
- ☐ DS type record
- ☐ None of the above

Solution: An A and RRSIG record are used to deliver the answer and a signature on the answer in standard DNSSEC.

Q8.5 (2 points) For this subpart only, assume an attacker has compromised the check server.

Also, assume that the user will re-send the query if they receive different answers from the check server and the other name servers.

Select all attacks that this attacker could carry out.

- ■ DoS attack
- ☐ Cache poisoning attack
- ☐ None of the above

Solution:

A: True. The check server can repeatedly send the wrong answer, forcing the user to re-send the query forever.

B: False. The check server can send the wrong answer, but cannot tamper with the answer returned by the other name servers (which are using DNSSEC).

Q8.6 (2 points) Select all true statements about the check server verifying non-existent domain names.

■ If the server uses **offline** signing, it will need to store a large amount of NSEC records.

☐ If the server uses **online** signing, it will need to store a large amount of NSEC records.

■ If the server uses **online** signing, it is more vulnerable to DoS attacks (compared to using offline signing).

■ If the server uses **online** signing, it is more vulnerable to having its private key stolen (compared to using offline signing).

☐ None of the above

Solution:

A: True. Offline signing means we sign the records ahead of time and deliver them as needed. This means the check server, which is answering queries for everything in the root zone, would need to store a large number of NSEC records.

B: False. Online signing means that the server signs answers for non-existent domains as they are requested. No pre-storage of records is needed.

C: True. Online signing requires on-demand computation, and an attacker can DoS the server by sending lots of fake domains that the server needs to sign on demand.

D: True: Online signing requires the server to keep its private key on the server itself which is connected to the Internet. By contrast, in offline signing, the server can sign records with its private key ahead of time, and does not need to keep the private key on a server connected to the Internet.

Q8.7 (2 points) Name one usability disadvantage to having a single additional check server. You can answer in 10 words or fewer (the staff answer is 1 word).

Solution: Staff answer: Scalability

A single check server will not not scale well to the volume of requests from the entire Internet.

Redundancy could also be an answer. If the check server goes down, the entire system would stop working. Alternate solutions could include bandwidth problems, etc.

Q9 *TLS: Key Rotation* **(10 points)**

Consider modifying TLS so that within a long-running connection, the server and client switch to using a different set of symmetric keys every hour.

For the entire question, you can assume TLS does **not** use record numbers.

Q9.1 (2 points) Suppose the server and client switch to using a different randomly-generated key every hour. Select all true statements.

- ■ Within one connection, a MITM attacker can only replay a message from the same hour.
- ☐ A MITM can replay messages from an earlier connection, if it was made during the same hour of the day.
- ☐ A MITM attacker could feasibly brute-force the symmetric keys if they were not switched every hour, but could not brute-force the keys if they only had one hour per key.
- ☐ This scheme has no practical purpose because TLS connections cannot last more than an hour, or else the TCP sequence numbers would start being reused.
- ☐ None of the above

Solution:

A: True. We assume TLS does not use record numbers, so an attacker can replay messages within the same hour, when the same key is being used.

B: False. Different connections have different values of ClientRandom and ServerRandom, so their initial symmetric keys will be different. Also, this subpart says that a different randomly-generated key is used every hour. Therefore, messages between different connections won't be encrypted with the same key.

C: False. Even if keys are not switched, it's infeasible for an attacker to brute-force a 128-bit or 256-bit symmetric key.

D: False. TCP sequence numbers don't depend on how long a connection has been open (they depend on the number of bytes sent).

Q9.2 (1 point) Suppose the client and server's system clocks are out of sync, and the client uses an old key to send a message to the server, which is using a newer key.

Assume the server has discarded the old key, and is only using the newer key.

What will the server do with this message?

○ Accept the ciphertext, and decrypt it to the correct plaintext.

● Accept the ciphertext, and decrypt it to garbage.

● Reject the message because the MAC is invalid.

○ Reject the message because the underlying IP packets will get dropped.

Solution: Recall that in TLS, messages after the handshake are both encrypted and MACed. Therefore, the message will be rejected because the MAC was computed with the old key, but the server is checking the MAC using the new key.

B) was intended to mean the server would decrypt the message to garbage and still process it as a "real" message, which is false. However, since TLS MAC-then-encrypts, the server does technically decrypt the ciphertext before rejecting it, so this was accepted as a valid answer as well.

A is false because the key is different.

D is false because the IP packet will not be dropped because of an invalid MAC. IP is best-effort and operates below the TLS layer, so IP will try its best to deliver the message, even if the MAC inside the message is incorrect.

In each of the next three subparts, a scheme for switching keys is provided.

An on-path attacker has observed the entire TLS connection so far (starting from the handshake).

The attacker wants to learn K_t, the current key in the connection. Can the attacker learn K_t, and if so, do they need to know K_{t-1}, the previous key in the connection?

Clarification during exam: For 9.4 and 9.5, $K_0 = \mathsf{HKDF}(\texttt{PremasterSecret}, \text{"start"})$.

Q9.3 (1 point) To compute the new key, the client and the server compute:

$$\mathsf{HMAC}(\texttt{PremasterSecret}, t)$$

where t is the number of hours elapsed since the beginning of the connection, rounded down.

- ○ The attacker can learn K_t, even if they don't know K_{t-1}.
- ○ The attacker can learn K_t, but only if they know K_{t-1}.
- [✓ ● The attacker cannot learn K_t, even if they know K_{t-1}.]

Solution: For the next three parts, recall that in order to compute the HMAC output, the attacker would need to know both HMAC inputs.

The attacker doesn't know the premaster secret, so they have no way to compute the HMAC output (even if they know K_{t-1}).

Q9.4 (1 point) To compute the new key, the client and the server compute:

$$\mathsf{HMAC}(\texttt{ClientRandom}, K_{t-1})$$

- ○ The attacker can learn K_t, even if they don't know K_{t-1}.
- [✓ ● The attacker can learn K_t, but only if they know K_{t-1}.]
- ○ The attacker cannot learn K_t, even if they know K_{t-1}.

Solution: ClientRandom is not encrypted when sent during the handshake, and the attacker has observed the entire TLS connection (according to the question assumptions).

Therefore, if the attacker knows K_{t-1}, they can compute the HMAC output.

However, if the attacker does not know K_{t-1}, they cannot compute the HMAC output.

Q9.5 (1 point) To compute the new key, the client and the server compute:

$$\mathsf{HMAC}(\texttt{PremasterSecret}, K_{t-1})$$

- ○ The attacker can learn K_t, even if they don't know K_{t-1}.
- ○ The attacker can learn K_t, but only if they know K_{t-1}.
- [✓ ● The attacker cannot learn K_t, even if they know K_{t-1}.]

Solution: As in the earlier subpart, an attacker who doesn't know the premaster secret can't compute the HMAC output, even if they know K_{t-1}.

In each of the next two subparts, a scheme for switching keys is provided.

An on-path attacker and a MITM attacker have observed the entire TLS connection so far (starting from the handshake). Select all true statements.

Q9.6 (2 points) To compute the new key, the client chooses a new, random $\texttt{PremasterSecret}_t$, encrypts it with the server's public key (not any symmetric keys), and sends the encrypted $\texttt{PremasterSecret}_t$ to the server.

K_t is derived from the new $\texttt{PremasterSecret}_t$, the original `ClientRandom`, and the original `ServerRandom`.

☐ A on-path attacker can learn K_t, even if they don't know K_{t-1}.

☐ A on-path attacker can learn K_t, but only if they know K_{t-1}.

■ A MITM attacker can trick the server into accepting a new key known by the attacker, even if they don't know K_{t-1}.

☐ A MITM attacker can trick the server into accepting a new key known by the attacker, but only if they know K_{t-1}.

☐ None of the above

Solution: A and B are false. The attacker has no way to learn the new premaster secret, since they don't know the server's private key (which would be needed to decrypt and learn the premaster secret). The premaster secret is not encrypted with K_{t-1}, so knowing this value does not help the attacker.

C is true. The MITM attacker can select their own $\texttt{PremasterSecret}_t$, encrypt it with the server's public key (known to the attacker since they observed the handshake and the server's public key is sent in the certificate), and send their own encrypted $\texttt{PremasterSecret}_t$ to the server. There's no way for the server to check whether this value was generated by the server or the attacker, so the server would accept the malicious $\texttt{PremasterSecret}_t$ value.

ClientRandom and ServerRandom are known to the attacker (since they observed the handshake), so this allows the attacker to derive the symmetric key with those two values and the attacker-selected premaster secret.

K_{t-1} is not needed to carry out this attack, so D is false.

Q9.7 (2 points) To compute K_t, the client and server perform a Diffie-Hellman key exchange, with the server signing its half of the exchange. (The exchange is not encrypted with any symmetric keys.)

K_t is derived from the new Diffie-Hellman shared secret, the original `ClientRandom`, and the original `ServerRandom`.

- ☐ A on-path attacker can learn K_t, even if they don't know K_{t-1}.
- ☐ A on-path attacker can learn K_t, but only if they know K_{t-1}.
- ■ A MITM attacker can trick the server into accepting a new key known by the attacker, even if they don't know K_{t-1}.
- ☐ A MITM attacker can trick the server into accepting a new key known by the attacker, but only if they know K_{t-1}.
- ☐ None of the above

Solution: A and B are false, because an on-path attacker who sees the Diffie-Hellman exchange (e.g. $g^a \bmod p$ and $g^b \bmod p$) cannot derive the Diffie-Hellman shared secret ($g^{ab} \bmod p$). Knowing K_{t-1} would not help the attacker learn the Diffie-Hellman shared secret.

C is true. The MITM attacker can replace the client's $g^a \bmod p$ with their own malicious $g^m \bmod p$. In TLS, the client does not sign their own half of the Diffie-Hellman exchange, so there's no way for the server to distinguish between the client-chosen and attacker-chosen values. This would cause the server to derive $g^{bm} \bmod p$ as the shared secret.

The attacker knows m (chosen by themselves) and $g^b \bmod p$ (sent by the server), so the attacker can also derive the shared secret $g^{bm} \bmod p$, which allows the attacker to derive the new key as well.

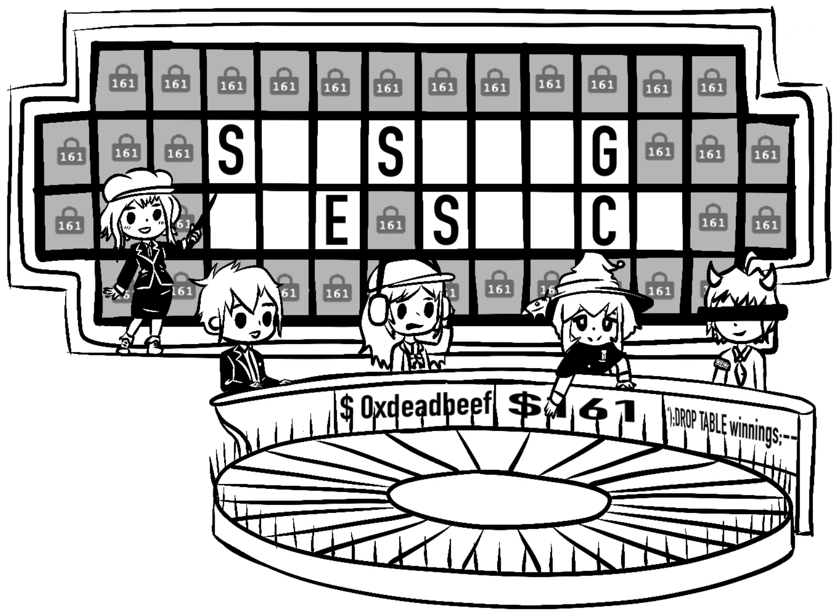
The attacker does not need to know K_{t-1} to carry out this attack, so D is false.

Nothing on this page will affect your grade.

Post-Exam Activity: Wheel of Fortune

Bot, I'd like to solve the puzzle:

Category: Memory Safety



Comment Box

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