Madison & Ana
Summer 2023

CS 161
Computer Security

Midterm

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

PRINT your student ID: ____

You have 120 minutes. There are 7 questions of varying credit (150 points total).

Question:	1	2	3	4	5	6	7	Total
Points:	3	30	23	24	20	24	26	150

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):
Word search! Circle your favorite mascot(s).

Q	E	N	2	A	M
5	E	V	E	L	A
C	O	D	A	O	L
B	E	E	P	N	L
O	C	E	I	Z	O
S	I	R	N	O	R
K	L	T	T	V	Y
I	A	B	O	B	6

Q1 *Honor Code* (3 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* (30 points)

Each true/false is worth 2 points.

Q2.1 True or False: Time of check to time of use (TOCTTOU) vulnerabilities violate the security principle "ensure complete mediation".

● (A) True ○ (B) False

Solution: True, since TOCTTOU relies on the fact that the condition is only enforced once and not checked again.

Q2.2 True or False: In the real world (outside of this class), stack canaries usually have a null byte.

● (A) True ○ (B) False

Solution: True

Q2.3 True or False: It is often worth disabling stack canaries in order to save compiler overhead.

○ (A) True ● (B) False

Solution: False, stack canaries add extremely little overhead.

Q2.4 True or False: In practice, PACs are a useful defense on an x86 little-endian 16-bit system.

○ (A) True ● (B) False

Solution: False, 16 bit addresses are very easy to forge PACs for.

Q2.5 True or False: Faster hash algorithms are strictly better than slow hash algorithms.

○ (A) True ● (B) False

Solution: False, both have their use cases (fast hashing for a bunch of different use cases, slow hashing for passwords in particular).

Q2.6 TRUE or FALSE: Using the same key to both encrypt and sign a message is good practice because you need to remember fewer keys.

○ (A) TRUE ● (B) FALSE

Solution: False, we always want to avoid key reuse.

Q2.7 TRUE or FALSE: A passive eavesdropper is able to compromise the security properties of Diffie-Hellman.

○ (A) TRUE ● (B) FALSE

Solution: False, Diffie-Hellman is resistant to passive eavesdroppers.

Q2.8 TRUE or FALSE: El Gamal encryption is a malleable scheme.

● (A) TRUE ○ (B) FALSE

Solution: True, multiplying s by k in an ElGamal ciphertext (r, s) makes the encrypted message go from m to km.

Q2.9 Consider the following scheme: A message M is encrypted with AES-CBC (with a random IV). The ciphertext is then tagged with a MAC scheme that fails EU-CPA security. Then, both the ciphertext and the tag are outputted.

TRUE or FALSE: This scheme preserves the confidentiality of M.

● (A) TRUE ○ (B) FALSE

Solution: True, since leaking information on the ciphertext does not leak information on the plaintext (if the ciphertext is IND-CPA). This scheme doesn't have integrity since the MAC is not EU-CPA, however.

Q2.10 TRUE or FALSE: IND-CPA secure schemes must be non-deterministic.

● (A) TRUE　　○ (B) FALSE

Solution: True, or else we can win the IND-CPA game with probability 1.

Q2.11 TRUE or FALSE: Once a certificate is issued, it is impossible to revoke it.

○ (A) TRUE　　● (B) FALSE

Solution: False, there are various methods of revocation (revocation lists combined with time limits, etc).

Q2.12 TRUE or FALSE: Certificate implementations using a trusted directory are scalable.

○ (A) TRUE　　● (B) FALSE

Solution: False, see lecture slides.

Q2.13 TRUE or FALSE: Storing passwords as $\mathsf{Enc}(K, \texttt{pwd})$ with secret key K is more secure than hashing them as $\mathsf{H}(\texttt{pwd})$

Clarification during exam: Instead of $\mathsf{H}(\texttt{pwd})$, passwords are stored as $\mathsf{H}(\texttt{pwd} \| \texttt{salt})$ where salt is a randomly chosen value for each user.

○ (A) TRUE　　● (B) FALSE

Solution: False, encrypted passwords can eventually be reversed if a key is leaked, unlike hash functions. It is not unlikely that a key or some form of key access will be leaked if the password database is also leaked, since the server needs to access the key to verify logins.

Q2.14 Evanbot tried to design a secure file system, but forgot to add integrity and authenticity features. He's now struggling to rewrite the entire codebase to add these features.

TRUE or FALSE: Evanbot violated Design in Security from the Start.

● (A) TRUE ○ (B) FALSE

Solution: True

Q2.15 CS161 staff spent all semester implementing a single strong security feature to their new website. They figured this would be enough to prevent any pesky attackers, so they stored the exam on the website. Unfortunately, a student Mallory, was able to bypass the feature, and now has access to the exam!

TRUE or FALSE: CS161 staff violated Separation of Responsibility.

○ (A) TRUE ● (B) FALSE

Solution: False

Q3 *Memory Safety Exploit: Across the Security-Verse* **(23 points)**

Consider the following code:

```
 1 void verse() {
 2     char miles[256];
 3     fgets(miles, 257, stdin);
 4 }
 5
 6 void spider() {
 7     verse();
 8 }
 9
10 void main() {
11     char *peter = (char *) malloc(128);
12     gets(peter);
13     printf("%x", peter);
14     spider();
15 }
```

Stack at Line 2
RIP of `main`
SFP of `main`
(1)
(2)
(3)
(4)
(5)
(6)

Assumptions:

- You may use `SHELLCODE` as a 120-byte shellcode.
- **ASLR is enabled** (including the code segment), but all other memory safety defenses are disabled.
- Assume that ASLR will **always** randomize the value of the SFP of `verse` to end with a least significant byte (LSB) in the range of `0x10` - `0xfc` (e.g. the LSB will never be `0x00` or `0x04`).
- You may use the variable `ADDR` to represent the output of `printf` on line 13, converted into a 4-byte, little-endian, byte string. You can directly use this without casting, converting, or slicing.

For the following 2 subparts, one variable should go in each row of the stack diagram. Assume the program is paused at a breakpoint on line 2.

Q3.1 (2 points) What values go in blanks (1), (2), and (3) in the stack diagram above?

○ (A) (1) RIP of `spider` (2) SFP of `spider` (3) `peter`

● (B) (1) `peter` (2) RIP of `spider` (3) SFP of `spider`

○ (C) (1) RIP of `spider` (2) `peter` (3) SFP of `spider`

Q3.2 (2 points) What values go in blanks (4), (5), and (6) in the stack diagram above?

○ (A) (4) SFP of `verse` (5) RIP of `verse` (6) `peter`

● (B) (4) RIP of `verse` (5) SFP of `verse` (6) `miles[256]`

○ (C) (4) `miles[256]` (5) RIP of `verse` (6) SFP of `verse`

Solution:

RIP of main
SFP of main
peter
RIP of spider
SFP of spider
RIP of verse
SFP of verse
miles[256]

Q3.3 (10 points) Provide inputs to the program that will execute shellcode with 100% probability. Use Python syntax (from project 1).

Input to `gets` on Line 12:

Solution: `SHELLCODE + '\n'`

Input to `fgets` on Line 3:

Solution: `ADDR * 64`

Solution: The key to this question is the `fgets` on Line 3 with a size input of 257. This allows us to fill up the buffer and then execute an off-by-one exploit by overwriting the least-significant byte of the SFP of verse.

Once we overwrite the SFP of verse with a null terminator, we are guaranteed to point somewhere into `miles` with at least 16 bytes in the buffer left above the pointer, since the problem specified the LSB is at least `0x10`. We can use this to cause the EBP to move into this buffer, and eventually move the ESP into the buffer after the return from `spider`.

Since ASLR is enabled, however, we need a way to access the address of a buffer where SHELLCODE is located. The `printf("%x", peter)` line does this for us, printing out the hexadecimal value **of** `char *peter` (the address of the buffer on the heap), not the value **at** `char *peter`. This is represented by the variable `ADDR` per the assumptions in the problem statement. We then need to fill `peter` with SHELLCODE.

Given this address, all that remains is to fill the entire `miles` buffer with `ADDR` such that the off-by-one exploit will eventually pop one of the addresses into the EIP.

Q3.4 (4 points) If we replaced line 13 with `printf("%x", &peter);`, is an exploit that executes `SHELLCODE` still possible?

○ (A) Yes, with 100% probability

● (B) Yes, with probability less than 100%

○ (C) No

Solution: Printing out `&peter` gives us a pointer to the stack (position (1) in the diagram). This means we cannot put SHELLCODE into the heap, since we have no way of finding the heap address with ASLR.

However, since ASLR preserves relative offsets, we can use the fact we know the distance between `&peter` and the start of the `miles` buffer to turn `&peter` into `&miles`. We can then place SHELLCODE into `miles` and fill the rest of the buffer with `&miles`. The probability that the SFP gets changed into the section with addresses is now less than 100% (actually $\approx 50\%$) but the exploit is still possible.

Q3.5 (5 points) Assume your exploit in Q3.3 was correct. **For this subpart only**, assume the LSB of the SFP of `verse` is `0x00`. Would this exploit still work?

○ (A) Yes

● (B) No

Briefly justify your answer.

Solution: If the LSB of the SFP of `function3` was `0x00`, we would not be able to redirect out pointer downwards into our buffer and complete the off by one exploit.

Q4 *Memory Safety Exploit: Snacktime* (24 points)

Consider the following code:

```
1  void goldfish(char* potato) {
2      fgets(potato, 256, stdin);
3
4      int8_t chip = strlen(potato);
5      printf("%s", &potato[chip]);
6
7      gets(potato);
8  }
9
10 void main() {
11     char cola[256];
12     goldfish(cola);
13 }
```

Stack at Line 4
RIP of `main`
SFP of `main`
(1)
(2)
(3)
(4)
(5)
(6)
(7)

Assumptions:

- **Stack canaries are enabled**, and all other memory safety defenses are disabled (unless otherwise specified).
- None of the bytes of any stack canaries are a null byte.
- You run GDB once and find that the `cola` buffer is located at the address `0xff00caf0`.

For the following 2 subparts, assume the program is paused at a breakpoint on line 4.

Q4.1 (2 points) Which values go in blanks (1), (2), and (3) in the stack diagram above?

○ (A) (1) `cola[256]` (2) canary (3) `potato`

○ (B) (1) canary (2) `cola[256]` (3) RIP of `goldfish`

○ (C) (1) canary (2) `potato` (3) `cola[256]`

● (D) (1) canary (2) `cola[256]` (3) `potato`

Q4.2 (2 points) Which values go in blanks (4), (5), (6), and (7) in the stack diagram above?

○ (A) (4) SFP of `goldfish` (5) canary (6) `chip` (7) `potato`

● (B) (4) RIP of `goldfish` (5) SFP of `goldfish` (6) canary (7) `chip`

○ (C) (4) SFP of `goldfish` (5) `potato` (6) canary (7) `chip`

○ (D) (4) RIP of `goldfish` (5) SFP of `goldfish` (6) `chip` (7) canary

Solution:

`rip main`
`sfp main`
`canary`
`cola[256]`
`potato`
`rip goldfish`
`sfp goldfish`
`canary`
`chip`

Q4.3 (3 points) Which vulnerability is present in the code?

○ (A) Off-by-one vulnerability

● (B) Signed/unsigned vulnerability

○ (C) Format string vulnerability

○ (D) None of the above

Q4.4 (10 points) Assuming you have a 252-byte SHELLCODE, provide inputs for fgets and gets that executes the SHELLCODE. You may use the variable OUTPUT to represent the output of printf on line 5. Use Python syntax (from project 1).

Reminder: On one execution of the program, all stack frames have the same canary value.

fgets:

Solution: `'A' * 240 + '\x00' + 'A' * 14`

gets:

Solution: `SHELLCODE + 'A' * 4 + OUTPUT[0:4] + 'A' * 4 + \xf0\xca\x00\xff`

Solution: The first input to fgets provides an input to the cola buffer such that a null byte is located at index 240. Thus, when strlen is called on this string, its length will be reported as 240. Note that strlen returns a number of type size_t (which is an unsigned, 4-byte integer), but we store its result in a variable of type int8_t (which a signed, 1-byte integer). Thus, the value 240 will be treated as -16 when stored in chip.

Then, the printf call on line 5 will print bytes beginning at -16th index of the cola array. This is where the canary for the goldfish stack frame is, so the first four bytes printed will be the canary value. Now that we have leaked the value of the canary, we can slice it from the OUTPUT variable with OUTPUT[0:4] and use it as part of a standard buffer overflow to the gets call on line 7.

Our input to gets starts with the shellcode and this will fill up cola upto the last four bytes. We then put four garbage bytes to fill up the rest of cola Right after that, we overwrite the canary with itself (which we just got from OUTPUT[0:4]. Then, we write four garbage bytes to overwrite the SFP of main. Finally, we overwrite the RIP of main with the address of the shellcode, which is the address of the cola buffer. From the assumptions, we found that this address is 0xff00caf0. We just need to ensure that we store it in little-endian form.

Q4.5 (2 points) **For this subpart only**, assume that non-executable pages are enabled (in addition to stack canaries).

True or False: The exploit from the previous subpart would still cause the shellcode to be executed.

○ (A) True ● (B) False

Solution: False. The only place in memory we can write the shellcode to in this code is on the stack, which is necessarily a writeable portion of memory. Thus, whereever we place our shellcode will therefore be non-executable and our shellcode will not be able to run.

Q4.6 (5 points) **For this subpart only**, assume that ASLR is enabled (in addition to stack canaries).

Would it still be possible to exploit this program with overwhelming probability?

● (A) Possible ○ (B) Not Possible

If you selected *Possible*, what conditions need to be true for the exploit to succeed? If you selected *Not Possible*, list which addresses in your exploit from above could no longer be learned by the attacker.

Solution: This exploit is still possible, because the `printf` may read past the canary and also leak the SFP of `goldfish`. This address can be used as a relative address to find the address of the RIP of `main`. However, we need the canary and the 3 least significant bytes of the SFP to be non-null to avoid terminating the `printf` early.

Q5 *Cryptography: All or Nothing Security* (20 points)

EvanBot decides to modify AES-CTR in order to provide **all-or-nothing security**. All-or-nothing security means that modifying *any* part of the ciphertext will make the *entire* plaintext decrypt to some sort of "garbage" output.

EvanBot designs the following scheme to encrypt $M = (M_1, M_2, \ldots, M_n)$:

1. EvanBot generates a new random key K_2 on top of the original key K_1. Note that K_2 is **not** known to the decryptor, even though K_1 is.
2. EvanBot transforms M into a "pseudomessage" M' by setting $M'_i = M_i \oplus E_{K_2}(i)$.
3. EvanBot adds the block $M'_{n+1} = H(M'_1 \oplus 1) \oplus H(M'_2 \oplus 2) \oplus \ldots \oplus H(M'_n \oplus n) \oplus K_2$.
4. EvanBot derives the ciphertext $C = \mathsf{Enc}(K_1, M')$ using AES-CTR with key K_1 and IV IV.

First, we will walk through the decryption process for this all-or-nothing scheme. Fill in the blanks for the following by answering the multiple-choice subparts below:

1. CodaBot receives C.
2. CodaBot decrypts C with key K_1 to recover ________________.
3. CodaBot sets K_2 = $M'_{n+1} \oplus$ ________________.
4. CodaBot finds i-th original message block as $M_i =$ ________________.

Q5.1 (2 points) Select the correct option for the blank on Step 2:

○ (A) K_2

○ (B) $H(M'_1 \oplus 1) \oplus \ldots \oplus H(M'_n \oplus n)$

○ (C) $M'_i \oplus E_{K_2}(i)$

● (D) M'

Solution: We first need to decrypt the ciphertext C, which decrypts to M' (the pseudomessage) as stated in Step 4 of the encryption process.

Q5.2 (2 points) Select the correct option for the blank on Step 3:

○ (A) K_2

● (B) $H(M'_1 \oplus 1) \oplus \ldots \oplus H(M'_n \oplus n)$

○ (C) $M'_i \oplus E_{K_2}(i)$

○ (D) M'

Solution: We now need to recover K_2 in order to decrypt the pseudomessage into the real message. By re-arranging the formula from Step 3 of the encryption process, we find that $K_2 = M'_{n+1} \oplus H(M'_1 \oplus 1) \oplus \ldots \oplus H(M'_n \oplus n)$.

Q5.3 (2 points) Select the correct option for the blank on Step 4:

○ (A) K_2

● (C) $M'_i \oplus E_{K_2}(i)$

○ (B) $H(M'_1 \oplus 1) \oplus \ldots \oplus H(M'_n \oplus n)$

○ (D) M'

Solution: We can now recover the real message by XOR-ing out $E_{K_2}(i)$ with the i-th block per Step 2 of the encryption process.

Q5.4 (5 points) Explain how modifying an arbitrary ciphertext block prevents recovery of ***any block*** of the original message.

HINT: Show that we cannot recover K_2 if any ciphertext block is modified.

Solution: Say we modify some C_i to C'_i. We then decrypt M'_i (the i-th pseudomessage block) to some garbage M_i^{*i}.

Recall that we recover K_2 by XOR-ing the hashes of all M'_i with the last ciphertext block. Therefore, since one of the inputs to these hashes is wrong, the entire XOR will be irrecoverably incorrect, since a small change in a hash input will lead to a wildly different output (avalanche effect). This is important to note, because otherwise an attacker could predictably modify the ciphertexts to cancel out their differences and recover the same K_2 (see next subpart).

Q5.5 (5 points) EvanBot wonders if it's really necessary to have the hash function used in Step 3, and decides to replace Step 3 with this new step:

3. EvanBot adds the block $(M'_1 \oplus 1) \oplus (M'_2 \oplus 2) \oplus \ldots \oplus (M'_n \oplus n) \oplus K_2$ to the end of M'.

Show that it is possible to tamper with the order of the message blocks, i.e. by swapping two blocks. Note that "tamper" means the message will be decrypted to something different, but not all blocks will turn to garbage (i.e. not "all or nothing").

Solution: Say we swap M'_1 and M'_2. When decrypting, the client will then *successfully* compute K_2 with the expression above.

Since we are using AES-CTR, we decrypt $M_1 = E_K(IV+1) \oplus C_2$ and $M_2 = E_K(IV+2) \oplus C_1$. Note that the C_1, C_2 in the decryption equations are swapped since we swapped the ciphertext. We then see that (since XOR is commutative):

$$\begin{aligned}&((E_K(IV+1) \oplus C_2) \oplus 1) \oplus ((E_K(IV+2) \oplus C_1) \oplus 2) \ldots \\ &= ((E_K(IV+1) \oplus C_1) \oplus 1) \oplus ((E_K(IV+2) \oplus C_2) \oplus 2) \ldots \\ &= (M'_1 \oplus 1) \oplus (M'_2 \oplus 2) \ldots\end{aligned}$$

This does not hold with the hash version, since the inputs to the hash changing even a little bit change the output dramatically (i.e. the XOR does not commute through the hash function).

Q5.6 (4 points) Does the original all-or-nothing scheme (from the beginning of the question) provide integrity?

○ (A) Yes ● (B) No

Explain why or why not.

Solution: This scheme does not provide integrity, since we cannot **detect tampering**. The all-or-nothing property just causes them to decrypt garbage, but this is not sufficient to provide integrity. For example, tampering with a normal AES ciphertext (without MAC) also causes them to decrypt a (at least partially) garbage message, but does not provide integrity.

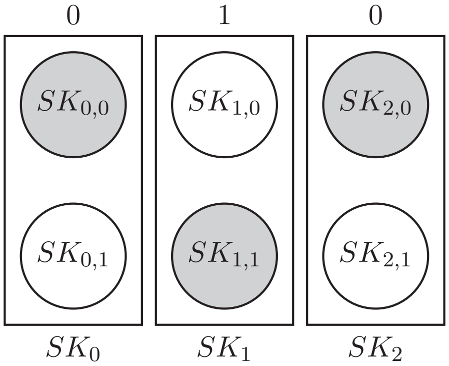
Q6 *Cryptography: One-Time Signatures* (24 points)

One-time signatures are a class of digital signatures that can only sign a single message before becoming insecure.

Consider a scheme to sign a n-bit message m using a hash function H with 256 bit output:

1. Generate n pairs of random 256-bit numbers as the private key, denoted SK_i for the i-th pair. $SK_{i,j}$ for $j = 0$ or $j = 1$ represents the first or second item in the pair, respectively.
2. Publish the public key as the list of all hashed secret key pairs: $PK_{i,j} = H(SK_{i,j})$.
3. To sign a n-bit message m, we consider its binary representation: $m_0, m_1, \ldots, m_{n-1}$. The signature $S = (S_0, S_1, \ldots, S_{n-1})$ is defined as $S_i = SK_{i,m_i}$.

In other words, for each bit m_i, if $m_i = 0$ we choose the first item from the i-th secret key pair. Otherwise, we choose the second item.



Pictured: Signing the binary message "010". The final signature is $(SK_{0,0}, SK_{1,1}, SK_{2,0})$.

Q6.1 (4 points) Alice has received a message M from Bob, with the one-time signature S using key PK. Select the correct option for verifying whether this signature is valid.

● (A) Verify that $H(S_i) = PK_{i,m_i}$ for $0 \le i < n$

○ (B) Verify that $S_i = H(PK_{i,m_i})$ for $0 \le i < n$

○ (C) Verify that $S_i = PK_{i,m_i}$ for $0 \le i < n$

○ (D) Invert H to verify $H^{-1}(PK_{i,m_i}) = S_{i,m_i}$

Solution: Since the public key is just the hash of the secret key, verifying the signature is equivalent to verifying that the given secret key corresponds to that public key by hashing and comparing.

Since hash functions are one-way, this proves we knew of the secret key to the given public key.

Alice and Bob forget that these signatures are one-time use only and accidentally sign two different n-bit messages with the same secret key: a message of all zeroes, and a message of all ones.

Q6.2 (8 points) Explain how Eve can forge **arbitrary** n-bit messages using (possibly not all of) Alice's public key and the two previous signatures.

Solution: The first signature (on all zeroes) is just a collection of $SK_{i,0}$ for all i. Likewise, the second signature is just $SK_{i,1}$ for all i, meaning Eve now sees the entire secret key.

Consider a different situation: Alice sends two different signed messages, but they only differ in b bits instead of differing in all n bits.

Q6.3 (6 points) How many new messages (excluding those already sent by Alice) can Eve forge in this new situation?

○ (A) 0 ● (B) $2^b - 2$ ○ (C) $2^{n-b} - 2$ ○ (D) $2^n - 2$

Explain your reasoning.

Solution: For the positions in which the bits differ, Eve can forge that bit position arbitrarily since she now knows both elements of that given pair. In the positions in which they do not differ, Eve is forced to use that specific bit value, since they only know one of two $S_{i,b}$.

Therefore, since b bits are different between the two messages, we can forge $2^b - 2$ different messages. The subtracted 2 is there because 2 messages have already been sent.

Q6.4 (6 points) Alice and Bob want to reduce the size of the public keys for an individual one-time signature scheme, without reducing its security.

Design a scheme that reduces the size of a one-time signature public key to exactly 256 bits.

HINT: The constraint only applies to the publicly trusted public key. The signature itself may or may not have additional information included that does not have to be constant space.

HINT: The output of H is 256 bits.

Describe how to construct the new public key:

Solution: We can use a Merkle tree, hash list, or similar to reduce the public key representation to a single hash value. When we publish a signature, we then include a proof that the used public key corresponds to the public hash.

For example, our new public key can simply be $H(PK_{0,0} \| PK_{0,1} \| \ldots \| PK_{n,0} \| PK_{n,1})$.

There are plenty of alternative solutions.

Describe how to produce a signature using this new scheme:

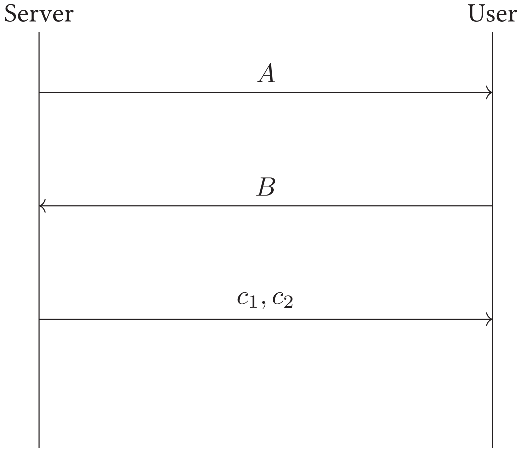
Solution: Using the previous example, our signature must now include the entire public key (or at least the public keys whose secret key entries weren't used). We send S and $PK_{0,0}, PK_{0,1}, \ldots, PK_{n,0}, PK_{n,1}$. The verifier can check that secret keys in the signature hash to values in the public key, and check that the real public key values hash to the short public key.

Q7 *Cryptography: Oblivious Transfer* **(26 points)**

A user of a popular document storage website wishes to access one of two possible documents (labeled m_1 and m_2), without the server knowing which document was accessed. They decide to use an **oblivious transfer** scheme. We will consider a 1-of-2 oblivious transfer scheme, in which the user accesses either m_1 or m_2 (but not both).

Clarification during exam: You may assume Enc refers to an IND-CPA symmetric encryption scheme.

Clarification during exam: i refers to the document the user wishes to access (if $i = 1$, they access m_1, if $i = 2$, they access m_2).



1. The server chooses a random number $a \mod p$ and sends $A = g^a \mod p$.

2. The user chooses a random number b. If $i = 1$, they send $B = g^b \mod p$. If $i = 2$, they send $B = Ag^b \mod p$.

3. The server computes $k_1 = B^a$ and $k_2 = \left(\frac{B}{A}\right)^a$ Next, they send $c_1 = \mathsf{Enc}(k_1, m_1)$ and $c_2 = \mathsf{Enc}(k_2, m_2)$.

4. [Answer to Q7.5]

Q7.1 (2 points) When $i = 1$, what value does the server derive for k_1?

● (A) $g^{ab} \mod p$, ○ (B) $g^{ab-a^2} \mod p$ ○ (C) $g^{ab+a^2} \mod p$ ○ (D) $Ag^{ab} \mod p$

Solution: $(g^b)^a \equiv g^{ab} \mod p$

Q7.2 (2 points) When $i = 1$, what value does the server derive for k_2?

○ (A) $g^{ab} \mod p$, ● (B) $g^{ab-a^2} \mod p$ ○ (C) $g^{ab+a^2} \mod p$ ○ (D) $Ag^{ab} \mod p$

Solution: $(\frac{g^b}{A})^a \equiv g^{ab-a^2} \mod p$

Q7.3 (2 points) When $i = 2$, what value does the server derive for k_1?

○ (A) $g^{ab} \mod p$, ○ (B) $g^{ab-a^2} \mod p$ ● (C) $g^{ab+a^2} \mod p$ ○ (D) $Ag^{ab} \mod p$

Solution: $(Ag^b)^a \equiv g^{ab+a^2} \mod p$

Q7.4 (2 points) When $i = 2$, what value does the server derive for k_2?

● (A) $g^{ab} \mod p$, ○ (B) $g^{ab-a^2} \mod p$ ○ (C) $g^{ab+a^2} \mod p$ ○ (D) $Ag^{ab} \mod p$

Solution: $(\frac{Ag^b}{A})^a \equiv g^{ab} \mod p$

Q7.5 (4 points) Give an expression for k_i (the i-th document key) in terms of (possibly not all of) A, B, b, c_i, g, p.

Solution: Alice should recover $k_i = A^b$ and then decrypt the respective ciphertext c_i using k_i.

To see why $k_i = A^b$, we can consider simple casework. Note that $A = g^a \mod p$ regardless of i, meaning $A^b \equiv (g^a)^b \equiv g^{ab} \mod p$ across both cases.

For $i = 1$, $B = g^a \mod p$. The server computes k_1 as $k_1 = B^a \equiv (g^b)^a \equiv g^{ab} \mod p$, so both k_1 match.

For $i = 2$, $B = Ag^a \mod p$. The server computes k_2 as $k_2 = (\frac{B}{A})^a \equiv (\frac{Ag^b}{A})^a \equiv g^{ab} \mod p$, so both k_2 match.

Q7.6 (3 points) Which option best describes why the server is not able to tell which document the user has chosen to read?

○ (A) Since b is randomly chosen, $g^b \mod p$ will look random to the server.

○ (B) The server cannot solve the discrete logarithm to recover a from $g^a \mod p$.

● (C) The server cannot tell the difference between $g^b \mod p$ and $Ag^b \equiv g^{a+b} \mod p$.

Solution:
Option (A) is technically true but not the best solution.

Option (B) is not relevant, since the server knows a.

Option (C) is the best choice, since the problem of identifying i comes down to detecting whether the client has sent $g^b \mod p$ or $Ag^b \mod p$.

Q7.7 (3 points) Which option best describes why the user is not able to read more than a single document per request?

○ (A) The encryption function used is IND-CPA secure.

● (B) The user cannot derive $g^{ab \pm a^2} \mod p$ despite knowing $g^{ab} \mod p$.

○ (C) The server will only derive k_1 or k_2, but not both.

Solution:
Option (A) is technically true but not the best solution.

Option (B) is correct, since to decrypt the other document, the client needs to find $g^{ab \pm a^2}$. This problem is intractable, since it reduces to recovering $g^{a^2} \mod p$ from $g^a \mod p$.

Option (C) is incorrect, the server derives both keys.

The user wants to expand their document storage from 2 to 3 documents, and need to update their oblivious transfer scheme to account for this. They know that Steps 1 and 4 from the previous scheme will stay the same, but still need to update Steps 2 and 3.

Q7.8 (4 points) The new Step 2 is as follows:

"The user chooses a random number b. If $i = 1$, they send $B = g^b \mod p$. If $i = 2$, they send $B = Ag^b \mod p$. If $i = 3$, they send $B =$ ________________."

Give an expression for the blank in this new step.

Solution: Replace step 2 blank with $B = A^2 g^b \mod p$.

Q7.9 (4 points) The new Step 3 is as follows:

"The server computes $k_1 = B^a$, $k_2 = \left(\frac{B}{A}\right)^a$, and $k_3 =$ ________________. Next, they send $c_1 = \mathsf{Enc}(k_1, m_1)$, $c_2 = \mathsf{Enc}(k_2, m_2)$, and $c_3 = \mathsf{Enc}(k_3, m_3)$."

Give an expression for the blank in this new step.

Solution: Replace step 3 blank with $k_3 = \left(\frac{B}{A^2}\right)^a$ for m_3.

The proof of correctness follows much the same: when $i = 3$, the server gets $B = A^2 g^b \mod p$ and computes $\left(\frac{B}{A^2}\right)^a \equiv \left(\frac{A^2 g^b}{A^2}\right)^a \equiv g^{ab} \mod p$.

(this page is intentionally blank)

Nothing on this page will affect your grade in any way.

(Optional) Post-exam Activity: Battleship

EvanBot wants to play battleship with CS 161 students. EvanBot has chosen the positions of the ships on their board, but won't reveal them until after the exam. However, they have published a SHA−3 hash of their board and ship locations so you can verify they haven't changed them later:

$\mathsf{H}(\text{board}) =$ `0xd726c59246ede23df594586246d5983924d00a06a7736ab67aa89c1db1461688`

Now you have the chance to try and hit their ships. On the grid below, mark **five** squares with an **X** where you believe EvanBot has placed their ships. After the exam, EvanBot will reveal their board and you can see how many ships you hit.

	1	2	3	4	5	6	7	8	9	10
A										
B										
C										
D										
E										
F										
G										
H										
I										
J										

Solution: Below is a grid where a o in a square indicates one of EvanBot's boats is located there.

	1	2	3	4	5	6	7	8	9	10
A			o							
B			o					o		
C			o					o		
D			o							
E			o					o	o	o
F										
G										
H							o			
I							o			
J	o	o	o	o			o			

To verify that EvanBot has not modified their board from before the exam, you can verify the hash they published by taking the $\mathsf{SHA} - 3$ hash of the following ASCII form of the above board.

```
+---+---+---+---+---+---+---+---+---+---+----+
|   | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
+---+---+---+---+---+---+---+---+---+---+----+
| A |   |   | o |   |   |   |   |   |   |    |
| B |   |   | o |   |   |   |   | o |   |    |
| C |   |   | o |   |   |   |   | o |   |    |
| D |   |   | o |   |   |   |   |   |   |    |
| E |   |   | o |   |   |   |   | o | o | o  |
| F |   |   |   |   |   |   |   |   |   |    |
| G |   |   |   |   |   |   |   |   |   |    |
| H |   |   |   |   |   |   | o |   |   |    |
| I |   |   |   |   |   |   | o |   |   |    |
| J | o | o | o | o |   |   | o |   |   |    |
+---+---+---+---+---+---+---+---+---+---+----+
```

Comment Box

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