Peyrin & Ryan
Summer 2020

CS 161
Computer Security

Midterm

For questions with **circular bubbles**, you may select exactly *one* choice on the answer sheet.

○ Unselected option

● Only one selected option

For questions with **square checkboxes**, you may select *one* or more choices on the answer sheet.

■ You can select

■ multiple squares

For questions with a **large box**, you need write and label your answer in the blank space below the question on the answer sheet.

You have 110 minutes. There are 5 questions of varying credit (150 points total).

The exam is open note. You can use an unlimited number of handwritten cheat sheets, but you must work alone.

Clarifications will be posted at https://cs161.org/clarifications.

MANDATORY - Honor Code (1 point)

Read the following honor code and sign your name on your answer sheet. ***Failure to do so will result in a grade of 0 for this exam.***

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 partial or complete loss of credit.

Q1 *True/false* (34 points)

Each true/false is worth 2 points. This question has 17 subparts.

Q1.1 True or False: If the discrete-log problem is broken (someone finds a way to efficiently calculate a given $g^a \bmod p$), ElGamal encryption is no longer secure.

[✓ True] False

Solution: True. ElGamal depends on Diffie-Hellman, which depends on the discrete-log problem. An attacker could learn r from the ciphertext, then calculate $(m \times B^r) \times B^{-r} \bmod p$ to obtain the original message.

Q1.2 True or False: Buffer overflows can occur on the stack and heap, but not in the static section of C memory.

True [✓ False]

Solution: False. Consider a program where a buffer is defined in static memory, and `gets` is called on the buffer.

Q1.3 True or False: The primary danger of format string vulnerabilities is that they let an attacker write more bytes into a buffer than the buffer has space for.

True [✓ False]

Solution: False. Calls to `printf` usually don't write into a buffer.

Q1.4 True or False: You create a Reddit bot but leave your secret credentials file in your public GitHub repo. You believe this is not a problem because attackers won't look at your Github repo. This is a failure to consider Shannon's Maxim.

[✓ True] False

Solution: True. The security of your bot is reliant upon nobody actually looking through the repo, which is security through obscurity.

Q1.5 True or False: If ASLR is enabled, leaking the address of a stack variable would give an attacker the address of heap variables.

True [✓ False]

Solution: False. Leaking the address of a stack variable would give an attacker the ability to determine other stack variables due to their deterministic spacing, but the address of the heap space is still random.

Q1.6 True or False: All cryptographic hash functions are one-to-one functions.

○ True [✓ False]

Solution: False. By definition, a hash function compresses an input which means you'll always have some collisions $\implies$ not one-to-one. Cryptographic hash functions try to make finding those collisions difficult, but they still exist.

Q1.7 True or False: Alice downloads a certificate for TikTok over a channel that is using encryption based on AES-ECB. She can **always** verify the validity of the certificate, assuming she has a validated copy of the parent certificate.

[✓ True] ○ False

Solution: True. As long as Alice verifies the signature on TikTok's certificate using the public key from Apple's certificate, she can trust that TikTok's certificate is valid.

Q1.8 True or False: Combining two independent detectors in parallel (alert when either detector alerts) is always more effective than combining them in serial (alert when both detectors alert).

○ True [✓ False]

Solution: False. It depends on the cost of getting a false positive vs the cost of a true positive and how frequently they occur.

Q1.9 Alice and Bob are communicating through RSA encryption, and both parties are attaching digital signatures to their messages. Alice and Bob each have a public encryption key, a private decryption key, a public verifying key, and a private signature key.

True or False: If Eve acquires access to **both** Alice and Bob's private signature keys, the communication channel is no longer confidential.

○ True [✓ False]

Solution: False. Even though Eve can now assume the identity of Alice or Bob, the actual messages sent between the real Alice and Bob remain encrypted, since Eve doesn't have access to either person's private decryption keys.

Q1.10 True or False: A company requires users to have long, complicated passwords. As a result, some employees write down their passwords on sticky notes to remember them. This is an example of not following the "Security is Economics" security principle.

○ True [✓ False]

Solution: False. this is an example of not following the 'Consider human factors' security principle.

Q1.11 True or False: If k is a 128 bit key selected uniformly at random, then it is impossible to distinguish $\text{AES}_k(\cdot)$ from a permutation selected uniformly at random from the set of all permutations over 128-bit strings.

Clarification made during the exam: $\text{AES}_k(\cdot)$ refers to the encryption function of AES using key k.

[✓ True] False

Solution: True. AES is believed to be secure, which means that no known algorithm can distinguish between $\text{AES}_k(\cdot)$ and a truly random permutation so long as k is selected uniformly at random.

Q1.12 True or False: Enabling stack canaries, ASLR, and DEP prevents all buffer overflow attacks.

True [✓ False]

Solution: False. it makes it harder for buffer overflow attacks, but doesn't eliminate the possibility.

Q1.13 True or False: Coding in a memory-safe language prevents all buffer overflow attacks.

[✓ True] False

Solution: True. Memory-safe languages abstract away memory allocation and memory management or check memory bounds during runtime, avoiding buffer overflows and many other memory safety attacks.

Q1.14 True or False: To use ElGamal encryption efficiently on very long messages, you should break up the message into small blocks and encrypt each block individually with ElGamal.

True [✓ False]

Solution: False. To use asymmetric cryptography with large messages, it is most appropriate to randomly generate a symmetric key, encrypt the message using symmetric encryption, and encrypt the key with asymmetric encryption to protect the confidentiality of the message. Using asymmetric cryptography directly on a very long message is very inefficient.

Q1.15 True or False: A hash function that is one-way but not collision resistance can be securely used for password hashing.

[✓ True] False

Solution: True. Collisions don't matter in this context as the only property we want is that an attacker can't invert a hash.

Q1.16 True or False: A hash function whose output always ends in 0 regardless of the input can't be collision resistant.

○ True [✓ ● False]

Solution: False. Consider $H(x) = SHA256(x)||0$. This hash is collision resistant but always ends in a 0.

Q1.17 True or False: Compared to the trusted directories model, digital certificates are less dependent on a central point of availability.

[✓ ● True] ○ False

Solution: True. Digital certificates can be distributed by anyone, not just the trusted directory.

This is the end of Q1. Proceed to Q2 on your answer sheet.

Q2 *Asymmetric* (29 points)

This question has 7 subparts.

Q2.1 (5 points) In Diffie-Hellman key exchange, which of the following elements would be known to an attacker observing network traffic between Alice and Bob? Assume the same syntax from notes and lecture, and Alice has a SK = a. Select all that apply.

■ (A) g ☐ (C) $a \bmod p$ ☐ (E) $g^a \bmod a$

■ (B) p ■ (D) $g \bmod p$ ☐ (F) None of the above

Solution: g and p are both public values, which also means the attacker can directly calculate $g \bmod p$.

To calculate $a \bmod p$ or $g^a \bmod a$, the attacker needs Alice's secret a. However, in Diffie-Hellman, Alice sends $g^a \bmod p$, and because the discrete-log problem is hard, the attacker cannot learn the value of a from this.

Q2.2 (5 points) Consider the un-padded ElGamal encryption scheme as shown in lecture. Alice sends the number 10, but a man-in-the-middle attacker intercepts the message. If Alice sends out the encrypted message (R, S), write an expression for a modified message that would cause Bob to receive the number 20.

Please clearly label your final answer on your answer sheet.

Solution: $(R, 2 \times S)$

The encrypted message is is $(R, S) = (g^r \bmod p, m{\times}B^r \bmod p)$. We can change the message from 10 to 20 by replacing m with $2 \times m$, so the expression becomes $(g^r \bmod p, 2 \times m \times B^r \bmod p) = (R, 2 \times S)$.

Bob is tired of having his email hacked, so he devises a personal encryption method for students to send him messages. Define g and p the same way they are defined in ElGamal. Assume that there are p students, each with a unique SID in the range $[0, p - 1]$.

Just like in ElGamal, Bob generates a secret key b, and a public key $B = g^b \pmod p$. Students with a valid student ID (sid) will encrypt their plaintext message m and send (R, S), where $R = g^{sid} \bmod p$ and $S = (m \times B)^{sid} \bmod p$.

Q2.3 (5 points) Assume Bob is expecting a message from a student with SID sid. Write an expression for m in terms of $p, b, R, S,$ and sid.

Please clearly label your final answer on your answer sheet.

Solution:

$S = (m \times B)^{sid} \bmod p$

$S = m^{sid} \times B^{sid} \bmod p$

Using the fact that $B = g^b \bmod p$:

$S = m^{sid} \times (g^b)^{sid} \bmod p$

$S = m^{sid} \times (g^{sid})^b \bmod p$

Using the fact that $R = g^{sid} \bmod p$:

$S = m^{sid} \times R^b \bmod p$

$S \times R^{-b} = m^{sid} \bmod p$

$m = (S \times R^{-b})^{1/sid} \bmod p$

Q2.4 (3 points) Will Bob be able to decrypt a message from someone he is not expecting in polynomial time?

○ (G) Yes, because Bob can try every *sid* in polynomial time

○ (H) Yes, because the decryption does not require Bob to know *sid*

[✓ (I) No, because the discrete-log problem is hard]

○ (J) No, because the factoring problem is hard

○ (K) None of the above

○ (L) ——

Solution: No, p is assumed to be large enough that this is not possible. Note, that if bruteforcing every value of $\mathbb{Z}_p$ was possible in polynomial time, than an attacker could break discrete log in a polynomial amount of time. Furthermore, there would be no guaranteed indication as to which decryption is the intended message because the message m is not padded. So, in general, this will be a bad encryption scheme.

Q2.5 (3 points) TRUE OR FALSE: The same attack from Q2.2 will succeed under this new schema.

Clarification made during the exam: Subpart 5 is asking if the exact expression you wrote in subpart 2 will have the same effect on the modified scheme.

○ (A) True [✓ (B) False] ○ (C) —— ○ (D) —— ○ (E) —— ○ (F) ——

Solution: False. Multiplying $2 \times S = 2 \times (m \times R)^{sid} \neq (2 \times m \times R)^{sid}$, which is what we'd want to recover.

Q2.6 (5 points) Suppose Alice is sending Bob your grade (R, S), and you know Alice's *sid*. You have the ability to launch a man-in-the-middle attack. Write an expression for a modified message that would change your grade to be 10 times your original grade.

Please clearly label your final answer on your answer sheet.

Solution: The encrypted message is $(R, S) = (g^{sid} \bmod p, (m \times B)^{sid} \bmod p)$. We can change the message to be 10 times its original value by replacing m with $10 \times m$, so the expression becomes:

$(g^{sid} \bmod p, (10 \times m \times B)^{sid} \bmod p)$

$(g^{sid} \bmod p, 10^{sid} \times (m \times B)^{sid} \bmod p)$

$(R, 10^{sid} \times S)$

Q2.7 (3 points) Assuming that the recipient knows the sid used, what does this scheme provide? Select all that apply.

☐ (A) Integrity [✓ (C) Confidentiality] ☐ (E) ——

☐ (B) Authentication ☐ (D) None of the above ☐ (F) ——

Solution: The previous part showed that the message could be manipulated without Bob's knowledge, so it lacks authentication and integrity. Without Bob's private key, the message maintains confidentiality.

This is the end of Q2. Proceed to Q3 on your answer sheet.

Q3 *IV-e got a question for ya* (24 points)

Determine whether each of the following schemes is IND-CPA secure. This question has 6 subparts.

Q3.1 (6 points) AES-CBC where the IV for message M is chosen as HMAC-SHA256(k_2, M) truncated to the first 128 bits. The MAC key k_2 is distinct from the encryption key k_1.

Provide a short justification for your answer on your answer sheet.

[✓ (A) Insecure] (C) —— (E) ——

(B) Secure (D) —— (F) ——

Solution: For any given message, the IV will be the same each time it's encrypted $\Longrightarrow$ deterministic scheme.

Q3.2 (6 points) AES-CTR where the IV for message M is chosen as HMAC-SHA256(k_2, M) truncated to the first 128 bits. The MAC key k_2 is distinct from the encryption key k_1.

Provide a short justification for your answer on your answer sheet.

Clarification made during the exam: You can assume that IV refers to the nonce for CTR mode.

[✓ (G) Insecure] (I) —— (K) ——

(H) Secure (J) —— (L) ——

Solution: For any given message, the IV will be the same each time it's encrypted $\Longrightarrow$ deterministic scheme.

Q3.3 (3 points) AES-CBC where the IV for message M is chosen as SHA-256(x) truncated to the first 128 bits. x is a predictable counter starting at 0 and incremented *per message*.

[✓ (A) Insecure] (C) —— (E) ——

(B) Secure (D) —— (F) ——

Solution: CBC mode requires its IVs to be random and thus unpredictable. To break IND-CPA, the adversary could send its first challenge as M = SHA-256(0), which would result in C = AES-CBC$_k$(SHA-256(0) $\oplus$ SHA-256(0)) = AES-CBC$_k$(0). Next, the adversary would send the challenge M_0 = SHA-256(1), $M_1 \neq M_0$, and the adversary knows that the challenger encrypted M_0 if $C_b = C$ and M_1 otherwise.

Q3.4 (3 points) AES-CTR where the IV for message M is chosen as SHA-256(x) truncated to the first 128 bits. x is a predictable counter starting at 0 and incremented *per message*.

Clarification made during the exam: You can assume that IV refers to the nonce for CTR mode.

○ (G) Insecure ○ (I) —— ○ (K) ——

● (H) Secure ○ (J) —— ○ (L) ——

Solution: CTR mode is secure even with predictable nonces, so long as you never reuse a counter in any block. Note that if x were used directly as the nonce, this would be insecure. Consider two 2-block messages M_0 and M_1. The first message would be encrypted with $x = 0$, so the two blocks encrypt the counter 0 and 1. THe second message would be encrypted with $x = 1$, so the two blocks encrypt the counter 1 and 2, breaking security.

Q3.5 (3 points) AES-CBC where the IV for message M is chosen as HMAC-SHA256($k_2 + x, M$) truncated to the first 128 bits. The MAC key k_2 is distinct from the encryption key k_1 and x is a predictable counter starting at 0 and incremented *per message*.

○ (A) Insecure ○ (C) —— ○ (E) ——

● (B) Secure ○ (D) —— ○ (F) ——

Solution: The IV is unpredictable to the attacker, even though the adversary can view previous IVs due to the properties of the HMAC.

Q3.6 (3 points) AES-CTR where the IV for message M is chosen as HMAC-SHA256($k_2 + x, M$) truncated to the first 128 bits. The MAC key k_2 is distinct from the encryption key k_1 and x is a predictable counter starting at 0 and incremented *per message*.

Clarification made during the exam: You can assume that IV refers to the nonce for CTR mode.

○ (G) Insecure ○ (I) —— ○ (K) ——

● (H) Secure ○ (J) —— ○ (L) ——

Solution: The IV is unpredictable to the attacker, even though the adversary can view previous IVs due to the properties of the HMAC.

This is the end of Q3. Proceed to Q4 on your answer sheet.

Q4 *steg* **(27 points)**

This question has 9 subparts.

Consider a new C function, `steg(char *s)`. It is similar to `gets`, but instead of writing to higher memory addresses, `steg` stores the user input by writing to lower memory addresses, starting at the memory address pointed to by `s`.

For example, if I call `steg(str)` and `&str = 0xdeadbeef`, and I type in `xyz` as input, the byte `x` will be stored at `0xdeadbeef`, the byte `y` will be stored at `0xdeadbeee`, and the byte `z` will be stored at `0xdeadbeed`.

Consider the following vulnerable C code that uses `steg`:

```c
void display(char *buf) {
  steg(buf);
  printf("%s", buf);
}

int main() {
  char door[4];
  display(&door);
}
```

(3 points) Fill in the numbered blanks for this incomplete stack diagram. Each box in the diagram represents 4 bytes. Each blank is worth 3 points.

rip of `main`
sfp of `main`
(1)
(2)
(3)
sfp of `display`

Q4.1 Blank (1)

[✓ (A) `door`]   (C) rip of `display`   (E) ——

(B) `buf = &door`   (D) ——   (F) ——

Q4.2 Blank (2)

(G) `door`   (I) rip of `display`   (K) ——

[✓ (H) `buf = &door`]   (J) ——   (L) ——

Q4.3 Blank (3)

○ (A) `door` [✓ (C) rip of `display`] ○ (E) ——

○ (B) `buf = &door` ○ (D) —— ○ (F) ——

Solution: First, in the `main` stack frame we allocate space for the local variable `door`. Then, to call the function `display`, we push the argument `buf = &door` on the stack. Finally, we push the rip of `display` (which points to the instructions for `main`) on the stack.

The stack looks like this (the address of each slot is in parentheses):

rip of `main`	(0xbfffff24)
sfp of `main`	(0xbfffff20)
`door`	(0xbfffff1c)
`buf = &door`	(0xbfffff18)
rip of `display`	(0xbfffff14)
sfp of `display`	(0xbfffff10)

Q4.4 (3 points) Which rip is vulnerable to being changed during the call to `steg`?

Remember that the rip of a function `f` refers to the EIP of the previous function that is pushed onto the stack when calling `f`.

[✓ (G) `display`] ○ (I) None of the above ○ (K) ——

○ (H) `main` ○ (J) —— ○ (L) ——

Solution: The call to `steg` writes to lower addresses, starting at `door`, so the vulnerable rip is the rip of `display`, which is below `door`.

Suppose we have an **8-byte** shellcode. Denote `REV_SHELLCODE` as a reversed version of this shellcode.

We find the address of `door` to be `0xbfffff1c`. Complete the exploit in the following three parts to cause the shellcode to execute.

Hint: x86 is little-endian (ie. the least significant byte of a word is stored at the lowest address), and we are writing from higher addresses to lower addresses.

Hint: `0xbfffff1c - 16 = 0xbfffff0c`, *and* `0xbfffff1c - 8 = 0xbfffff14`.

Q4.5 (3 points) At the call to `steg` at line 2, first input this many bytes of garbage to reach the rip:

○ (A) 0 ○ (B) 1 [✓ (C) 5] ○ (D) 9 ○ (E) 13 ○ (F) 17

Solution: The RIP is 4 bytes below `door`, and `steg` starts writing at the first byte of `door`, so we have to write 1 byte of garbage to fill the first byte of door, then another 4 bytes to skip over the argument `buf`.

Q4.6 (3 points) Then overwrite the rip with these bytes:

[✓ (G) `\xbf\xff\xff\x0c`]   ○ (J) `\x14\xff\xff\xbf`

○ (H) `\x0c\xff\xff\xbf`   ○ (K) `REV_SHELLCODE`

○ (I) `\xbf\xff\xff\x14`   ○ (L) —

Solution: The address of the start of shellcode will be 8 bytes below `&rip` (or 16 bytes below `door`), which is `0xbfffff0c`.

We want to input the address `0xbfffff0c` in little-endian. If we were writing from lower addresses to higher addresses, like in project 1, this would be `\x0c\xff\xff\xbf`. Partial credit was given for this answer.

However, we are writing from higher addresses to lower addresses, so the correct answer is actually the reverse of this, which is `\xbf\xff\xff\x0c`.

Q4.7 (3 points) Then input these bytes:

○ (A) `\xbf\xff\xff\x0c`   ○ (D) `\x14\xff\xff\xbf`

○ (B) `\x0c\xff\xff\xbf`   [✓ (E) `REV_SHELLCODE`]

○ (C) `\xbf\xff\xff\x14`   ○ (F) —

Solution: Finally, we input the shellcode, remembering to input it backwards because we're writing from higher addresses to lower addresses.

Q4.8 (3 points) Would the exploit from the previous parts still work if stack canaries were enabled? Assume there is no way for the attacker to learn the value of the stack canary.

○ (G) Yes   [✓ (H) No]   ○ (I) —   ○ (J) —   ○ (K) —   ○ (L) —

Solution: No. This exploit overwrites 12 bytes below `door`, which would overwrite the stack canary just below the sfp of `display`.

A common mistake was to answer yes, because the exploit does not affect the canary just below the sfp of `main`, but remember that when we enable stack canaries, a canary is added below the sfp in every stack frame.

Q4.9 (3 points) What is the length (in bytes) of the longest shellcode that can be executed using the exploit in the previous parts without triggering a stack canary? Assume there is no way for the attacker to learn the value of the stack canary.

Please clearly label your final answer on your answer sheet.

Solution: The intended answer was 4 bytes. After the rip of `display`, there are 4 bytes of the sfp you can overwrite with shellcode before the canary is overwritten.

However, you could also put the shellcode at the beginning of your exploit by writing shellcode all the way until the rip of `display`. There are 5 bytes you can overwrite here (see subpart 5), so technically 5 bytes is the correct answer.

This is the end of Q4. Proceed to Q5 on your answer sheet.

Q5 *A Dangerous Game* (35 points)

This question has 9 subparts.

Note: This is the hardest question on the exam. We recommend trying the other questions on the exam before this one.

A new online game, *HackMe*, splits 128-512 players into groups of 16 and has all groups compete to hack each other. *HackMe* uses a hash table to create groups and store info about each player.

Recall that a hash table is an array of "buckets" (here each bucket is a linked list). To add a player to the table, a hash function is evaluated to decide which bucket the player goes into, and they are appended to the linked list of that bucket.

```c
typedef struct Player {
    int id;
    int hacking_ability;
} Player;

typedef struct Bucket {
    int8_t size;    // 8 bit signed integer
    LinkedList *b; // Pointer to a linked list implementation
} Bucket;

typedef struct HashTable {
    int players;
    Bucket buckets[16];
} HashTable;

void add_player(HashTable *t, Player p) {
    size_t idx = hash(p.id + t->players); // hash range is [0, 16)
    append(t->buckets[idx].b, p);         // appends p to LinkedList
    t->buckets[idx].size += 1;
    t->players += 1;
}
```

Q5.1 (3 points) Assume that `hash()` outputs an unsigned integer equal to the last 4 bits of a pseudorandom, cryptographic hash function. If the table contains a number of `Player`s with random `id`s, what do you expect about the size of the buckets?

[✓ (A) They will all roughly be the same size]

○ (B) The 0th bucket will be larger than the 1st bucket

○ (C) The 1st bucket will be larger than the 0th bucket

○ (D) ——

○ (E) ——

○ (F) ——

Solution: Since the hash function is pseudorandom and all the inputs to the hash function will be different with high probability, the `Player`s should be uniformly distributed.

Q5.2 (3 points) Assume that `hash()` outputs an unsigned integer equal to the last 4 bits of a pseudorandom, cryptographic hash function. If the table contains a number of `Player`s with the same `id`, what do you expect about the size of the buckets?

[✓ (G) They will all roughly be the same size]

○ (H) The 0th bucket will be larger than the 1st bucket

○ (I) The 1st bucket will be larger than the 0th bucket

○ (J) ——

○ (K) ——

○ (L) ——

Solution: The inputs to the hash function will still all be different since the id is added to the player count. Thus, the `Player`s should be uniformly distributed.

Q5.3 (3 points) Say a user stores a large number (ie. 10000) of `Players` in a `HashTable`.

Which of the following would occur given the code above?

[✓ (A) Integer overflow]	○ (C) Off-by-one	○ (E) ——
○ (B) Buffer overflow	○ (D) ——	○ (F) ——

Solution: Each bucket will contain more than 127 elements so the `int8_t size` variable will overflow.

Q5.4 (3 points) Which line number contains the vulnerability from the previous part?

[✓ (G) Line 7]	○ (I) Line 13	○ (K) ——
○ (H) Line 8	○ (J) ——	○ (L) ——

Solution: The `int8_t size` variable is defined at line 7.

To register a group for playing *HackMe*, one inputs a list of `Player`s to the following function which adds all `Players` to a HashTable, assigns the group to a server based on size of the 0th bucket, and sets a group name.

```
 1 void register_group(Player *players, size_t num_players) {
 2     char *server_names[128] = { /* Contains 128 server names */ };
 3     char *a_gift = 0xffffd528; // Pointer to the stack canary
 4     char group_name[16];
 5     HashTable group;
 6     for (int i = 0; i < num_players; i++) {
 7       add_player(&group, players[i]);
 8     }
 9     printf("Use server: %s\n", server_names[group.buckets[0].size]);
10     printf("Please provide 16 character group name: \n");
11     gets(group_name);
12     ...
13 }
```

Q5.5 (5 points) Consider line 9:

```
printf("Use server: %s\n", server_names[group.buckets[0].size]);
```

Which *valid* values of `group.buckets[0].size` would cause this statement to print something outside of `server_names`?

______ ≤ `group.buckets[0].size` ≤ ______

Please clearly label your final answer on your answer sheet.

Solution:

-128 ≤ `group.buckets[0].size` ≤ -1

`server_names` is size 128, so 0 ≤ `group.buckets[0].size` ≤ 127 are all valid memory accesses. However, `group.buckets[0].size` is a signed variable (`int8_t`), so it can also take on negative values that will cause illegal memory accesses. The negative range of an `int8_t` variable is −128 ≤ `group.buckets[0].size` ≤ −1.

Q5.6 (10 points) Mallory challenges you to hack *HackMe*. Assume you can invoke `register_group` with a list of `Player`'s of your choosing, but the list must have length between [128, 512] and `num_players` must always be correct.

HackMe uses a 32-bit x86 system with **stack canaries enabled** (assume that canaries don't contain null bytes) but no W^X bit or ASLR. In order to help you out, Mallory has added a pointer to the stack canary: `a_gift`.

Describe the list of `Players` you input. Assume that `hash()` is a publicly-known function that you can query before making your list.

Clarification made during the exam: `a_gift` is a pointer to the stack canary of the `register_group` frame.

Clarification made during the exam: Your answer to subpart 6 should give you information to complete the exploit in subpart 7.

○ (G) —— ○ (H) —— ○ (I) —— ○ (J) —— ○ (K) —— ○ (L) ——

If you need more space on your answer sheet, you can write on a blank sheet of paper and attach it with your submission.

Solution: The overarching idea is we want to fill up the 0th bucket such that we overflow the `size` variable, making it negative and causing the print statement at line 8 to print out the stack canary. To do this, we take advantage of the fact that a hash function is deterministic.

Since `hash` is public, we query it until we find an input x that maps to 0. We set this number to be `id` of our first `Player`. We set `id` of our second `Player` to be $x - 1$, `id` of third `Player` to be $x - 2$, etc. This causes each Player to be placed in the 0 bucket. We repeat this 255 times so that the `size` variable is equal to -1. This will cause the array access to `server_names` to print out whatever `a_gift` points at - which is given to be the stack canary.

Q5.7 (5 points) Write down your exact input to the `gets` call at line 11. Assume that `SHELLCODE` holds 64-byte shellcode, `GARBAGE` is an arbitrary byte, and `OUTPUT` is the output from the print statement at line 9.

You can write constants using hex (e.g., 0xFF or 0xA02200FC). For instance, `4*GARBAGE + OUTPUT[:1] + SHELLCODE` would represent four irrelevant bytes, followed by the first byte of the print result, followed by the 64-byte shellcode.

○ (A) —— ○ (B) —— ○ (C) —— ○ (D) —— ○ (E) —— ○ (F) ——

Solution: `GARBAGE`*(16 + 4 + 128*4) + `OUTPUT`[12:16] + `GARBAGE`*4 + `0xffffd534` + `SHELLCODE`

First, we write 16 bytes of garbage to overwrite local variable `char group_name[16]`. Then, we write 4 bytes of garbage to overwrite local variable `char *a_gift`. Then, we write 128*4 bytes to overwrite local variable `char *server_names[128]`.

Next, we write the canary leaked from the previous part, when the `printf` call at line 9 accesses `server_names[-1].size` which is the canary value. This is `OUTPUT`[12:16] since we need to skip past the "Use server: ".

Next, we write 4 more bytes of garbage to overwrite the sfp of `register_group`.

Next, we write a pointer to the start of shellcode, which is 4 bytes after the rip of `register_group`. We know that the canary of `register_group` is located at `0xffffd528`, so the sfp is 4 bytes above the canary at `0xffffd52c`. The rip is 4 bytes above the sfp, at `0xffffd530`. So 4 bytes after the rip is `0xffffd534`.

Finally, we write the shellcode above the rip.

Q5.8 (3 points) Which of the following could prevent this attack? Assume `a_gift` always correct points to the stack canary.

■ (G) ASLR

■ (H) $W \wedge X$ protection (NX bit)

□ (I) Increasing the size of `server_names` to 256

□ (J) None of the above

□ (K) ——

□ (L) ——

Solution: Even though `a_gift` always points correctly, we never have the opportunity to read its address so ASLR will still stop us.

$W \wedge X$ protection (making the stack non-executable) will stop us because the exploit involves running shellcode that we placed on the stack.

Increasing the size of `server_names` doesn't have any effect since the exploit is reading lower memory addresses like `server_names[-1]`, not higher addresses.

This is the end of Q5. You have reached the end of the exam.