Midterm Review - Memory Safety

Question 1 ***True/false*** ()

Q1.1 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.

Q1.2 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.

Q1.3 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.

Q1.4 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.

Q1.5 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.

Question 2 *A Dangerous Game* **(35 min)**

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;
}
```

Q2.1 (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 0^{th} bucket will be larger than the 1^{st} bucket

○ (C) The 1^{st} bucket will be larger than the 0^{th} 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.

Q2.2 (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 0^{th} bucket will be larger than the 1^{st} bucket

○ (I) The 1^{st} bucket will be larger than the 0^{th} 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.

Q2.3 (3 points) Say a user stores a large number (ie. 10000) of `Player`s in a `HashTable`.

Which of the following would occur given the code above?

● (A) Integer overflow

○ (B) Buffer overflow

○ (C) Off-by-one

○ (D) —

○ (E) —

○ (F) —

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

Q2.4 (3 points) Which line number contains the vulnerability from the previous part?

- ● (G) Line 7
- ○ (H) Line 8
- ○ (I) Line 13
- ○ (J) ——
- ○ (K) ——
- ○ (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 `Player`s to a HashTable, assigns the group to a server based on size of the 0^{th} 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 }
```

Q2.5 (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 $-128 \le$ `group.buckets[0].size` ≤ -1.

Q2.6 (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 `Player`s 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 0^{th} 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.

Q2.7 (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.

Q2.8 (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.