Weaver
Spring 2022

CS 161 Computer Security

Exam Prep 2

Q1 *Indirection* (0 points)

Consider the following vulnerable C code:

```c
 1 #include <stdlib.h>
 2 #include <string.h>
 3 
 4 struct log_entry {
 5     char title[8];
 6     char *msg;
 7 };
 8 
 9 void log_event(char *title, char *msg) {
10     size_t len = strnlen(msg, 256);
11     if (len == 256) return; /* Message too long. */
12     struct log_entry *entry = malloc(sizeof(struct log_entry));
13     entry->msg = malloc(256);
14     strcpy(entry->title, title);
15     strncpy(entry->msg, msg, len + 1);
16     add_to_log(entry); /* Implementation not shown. */
17 }
```

Assume you are on a little-endian 32-bit x86 system and no memory safety defenses are enabled.

Q1.1 (3 points) Which of the following lines contains a memory safety vulnerability?

○ (A) Line 10

○ (B) Line 13

[✓ (C) Line 14]

○ (D) Line 15

○ (E) ——

○ (F) ——

Solution: Line 14 uses a `strcpy`, which is not a memory-safe function because it terminates only when it sees a NULL byte, which is under the control of the attacker. Note that line 15 uses a `strncpy` whose length parameter comes from `strnlen`, so it is safe.

Q1.2 (3 points) Seeing an opportunity to exploit this program, you fire up GDB and step into the `log_event` function. Give a GDB command that will show you the address of the rip of the `log_event` function. (Abbreviations are fine.)

> **Solution:** `info frame` (abbreviated `i f`) would be the easiest command to use. Other solutions exist.

Q1.3 (3 points) Fill in the numbered blanks on the following stack and heap diagram for `log_event`. Assume that lower-numbered addresses start at the bottom of both diagrams.

Stack
`msg`
1
rip
sfp
`len`
`entry`

Heap
3
2

○ (A) 1 = `entry->title` 2 = `entry->title` 3 = `msg`

○ (B) 1 = `entry->title` 2 = `msg` 3 = `entry->title`

[✓ (C)] 1 = `title` 2 = `entry->title` 3 = `entry->msg`

○ (D) 1 = `title` 2 = `entry->msg` 3 = `entry->title`

○ (E) ——

○ (F) ——

> **Solution:** The two arguments, `title` and `msg`, must be on the stack, so 1 = `msg`.
>
> Structs are filled from lower addresses to higher addresses, so 2 = `entry->title` and 3 = `entry->msg`.

Using GDB, you find that the address of the rip of `log_event` is `0xbfffe0f0`.

Let `SHELLCODE` be a 40-byte shellcode. Construct an input that would cause this program to execute shellcode. Write all your answers in Python 2 syntax (just like Project 1).

Q1.4 (6 points) Give the input for the `title` argument.

Solution: The `title` will be used to overflow the `title` buffer in the struct to point the `msg` pointer to the RIP. The input should thus be

```
'A' * 8 + '\xf0\xe0\xff\xbf'
```

Q1.5 (6 points) Give the input for the `msg` argument.

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

Solution: The first 4 bytes will be written in the location of the RIP, which should point to the shellcode. Thus, our input should be

```
'\xf4\xe0\xff\xbf' + SHELLCODE
```

Q1.6 (3 points) Which of the following defenses on their own would prevent your exploit?

Note: If stack canaries are enabled, you can assume is still the correct address of the RIP.

☐ (G) Stack canaries

[✓ (H) W^X]

[✓ (I) ASLR]

☐ (J) None of the above

☐ (K) ——

☐ (L) ——

Solution: Stack canaries would not defend against this attack because the write operation starts at the RIP by pointing `entry->msg` to after the canary, jumping over it with a random access. W^X defends against this by preventing the shellcode on the stack from being executed, and ASLR defends against this by randomizing the addresses that `entry->msg` and the RIP need to be overwritten with.

Q2 *Memory Safety Vulnerabilities* **(23 points)**

Consider the following vulnerable C code:

```c
#include <stdio.h>
#include <string.h>

struct packet {
    char payload[300];
    char format[300];
};

void deploy(struct packet *ptr) {
    printf(ptr->format, ptr->payload);
}

int main(void) {
    struct packet p;
    do {
        strcpy(p.format, "%s\n");
        gets(p.payload);
        deploy(&p);
    } while (strcmp(p.payload, "END") != 0);
    // Assume loop always exits for subpart 3.
    return 0;
}
```

Assume you are on a little-endian 32-bit x86 system. Assume that there is no compiler padding or additional saved registers in all subparts. For the first 3 subparts, assume that **no memory safety defenses** are enabled.

Fill in the following stack diagram, assuming that execution has entered the call to `printf`:

RIP of `main`
SFP of `main`
(1a)
(1b)
(1c)
(2a)
(2b)
(2c)
(2d)
RIP of `printf`
SFP of `printf`

Q2.1 (3 points) For (1a), (1b), and (1c):

- [✓ (A) (1a) - `p.format`; (1b) - `p.payload`; (1c) - `ptr`]
- (B) (1a) - `p.payload`; (1b) - `p.format`; (1c) - `ptr`
- (C) (1a) - `ptr`; (1b) - `p.payload`; (1c) - `p.format`
- (D) (1a) - `ptr`; (1b) - `p.format`; (1c) - `p.payload`
- (E) ——
- (F) ——

Q2.2 (3 points) For (2a), (2b), (2c), and (2d):

(G) (2a) - RIP of `deploy`; (2b) - SFP of `deploy`; (2c) - `&ptr->format`; (2d) - `&ptr->payload`

(H) (2a) - SFP of `deploy`; (2b) - RIP of `deploy`; (2c) - `&ptr->format`; (2d) - `&ptr->payload`

(I) (2a) - `&ptr->payload`; (2b) - `&ptr->format`; (2c) - RIP of `deploy`; (2d) - SFP of `deploy`

(J) (2a) - `&ptr->payload`; (2b) - `&ptr->format`; (2c) - SFP of `deploy`; (2d) - RIP of `deploy`

[✓ (K) (2a) - RIP of `deploy`; (2b) - SFP of `deploy`; (2c) - `&ptr->payload`; (2d) - `&ptr->format`]

(L) ——

Solution:

The local variable in the `main` stack frame is `struct packet p`. Within a struct, the first variable is stored at the lowest memory address, so `p.payload` is at a lower address than `p.format`.

Before the execution enters `printf`, the `main` function first calls the `deploy` function. When calling a function, the argument is pushed on the stack first. For the `deploy` function, the argument is `ptr`.

Next, the program pushes the RIP of `deploy` and the SFP of `deploy`. Recall that the RIP is at a higher address on the stack than the SFP.

Finally, the `deploy` function calls the `printf` function. Recall that the arguments are pushed on the stack in backwards order, so `&ptr->payload` is pushed first, then `&ptr->format` is pushed next. This means that `&ptr->payload` is at a higher address than `&ptr->format`.

Here is the stack diagram filled in:

RIP of `main`
SFP of `main`
`p.format`
`p.payload`
`ptr`
RIP of `deploy`
SFP of `deploy`
`&ptr->payload`
`&ptr->format`
RIP of `printf`
SFP of `printf`

Q2.3 (3 points) For this subpart only, assume that you may only execute one iteration of the while loop and that the call to `printf` will not segfault. For this subpart, assume that no memory safety defenses are enabled.

If the address of `p` is `0x7ff3ec10`, construct an input at line 18 that would cause the program to execute malicious shellcode. You may reference `SHELLCODE` as a 30-byte malicious shellcode. Write your answer in Python 2 syntax (just like in Project 1).

Clarification during exam: Instead of "Line 18," the question should say "Line 17."

Solution: The `gets` function allows us to overflow the `p.payload` buffer on the stack. We have 600 bytes between the start of the buffer and the shellcode, so we can place our 30-byte shellcode at the beginning, followed by 570 dummy bytes, followed by the address of our shellcode, which is the address of `p`:

```
SHELLCODE + 'A' * 574 + '\x10\xec\xf3\x7f'
```

For the remaining subparts, assume that **stack canaries are enabled**. Note that this changes the stack diagram!

Q2.4 (5 points) For your exploit, construct a one-line Python helper function `write_byte(addr, byte)` that returns an input for line 17 of the vulnerable C code. This input should ensure that `byte` is written to the address at `addr`. This function may change bytes **above** `addr` (but not below), as long as the correct byte is written at `addr` itself. **The returned input only needs to work for values of `byte` greater than 8.**

Assume that `addr` is given as a 4-byte Python string containing the bytes of the address in little-endian, and assume that `byte` is given as a Python integer between 9 and 255. For example, `write_byte('\xef\xbe\xad\xde', 128)` would be a valid call to this function. Write your answer in Python 2 syntax (just like in Project 1).

```
def write_byte(addr, byte):
    return # Your answer here
```

Hint: You may find the `%c` format specifier useful: Read 4 bytes off the stack and print as a single character.

Solution: We take advantage of the `%n` specifier, which writes the number of bytes printed so far to the next pointer on the stack. This gives us the ability to write a value to any aribtrary place in memory, as long as we know that `printf` will look for the pointer in a piece of memory that we control.

We start by reasoning about the format specifier in `p.format`, which is 300 bytes after the start of `p.payload`. To do this, we first need to add 5 `%c` sequences to force `printf` to read 5 arguments off the stack, which would read the values of `&ptr->payload`, a stack canary, RIP of `deploy`, SFP of `deploy`, and `ptr`, causing it to print 5 characters. Now, we know that the first 5 bytes of `payload` will be read as the next argument, so we remember to place our address there. To trick `printf` to write our desired byte, we need to print an additional `byte - 5` bytes, since we already printed 5 bytes to set up our buffer as the next argument. The format specifier ends with a `%n` to execute the write.

Now we reason about the payload in `p.payload`. The format specifier needs to be placed 300 bytes after the start of `p.buffer`, and the first 4 bytes of `p.payload` will be our address, so we need 296 dummy bytes to fill `p.payload`. and our final helper is as follows:

```
addr + 'A' * 296 + '%c' * 5 + 'B' * (byte - 5) + '%n'
```

Q2.5 (5 points) If the address of `p` is `0x7ff3ec10` and the address of the RIP of `main` is `0x7ff3ee68`, construct a series of inputs for repeated calls at line 18 that would cause the program to execute malicious shellcode. Assume that `write_byte` is implemented correctly, and you may call `write_byte` for as many inputs as you would like. Write your answer as a series of `print` statements, all in Python 2 syntax (just like in Project 1).

Hint: You may write hex literals to represent integers in Python, such as `0x36`.

Clarification during exam: Instead of "Line 18," the question should say "Line 17."

Solution: We use our `write_byte` helper to overwrite the RIP with the address of our shellcode, one byte at a time. We know that `write_byte` will potentially modify bytes above the destination byte but not below, so we start with the LSB and write upwards. Afterwards, we place our shellcode in `p.payload` and then print `END` to exit the loop. Notice that `END` will need to be at the beginning of the buffer, so we actually need to place the shellcode 4 bytes after beginning of `p.payload` so that it doesn't get overwritten (3 characters plus 1 NULL).

```
print(write_byte('\x68\xee\xf3\x7f', 0x14))
print(write_byte('\x69\xee\xf3\x7f', 0xec))
print(write_byte('\x6a\xee\xf3\x7f', 0xf3))
print(write_byte('\x6b\xee\xf3\x7f', 0x7f))
print('A' * 4 + SHELLCODE)
print('END')
```

Q2.6 (4 points) Which of the following changes, if made on their own, would prevent the attacker from executing malicious shellcode (not necessarily using your exploit above)?

☐ (G) Enabling non-executable pages in addition to stack canaries

☐ (H) Enabling ASLR in addition to stack canaries

■ (I) Rewriting the code in a memory-safe language

■ (J) Using `fgets(p.payload, 300, stdin)` instead of `gets(p.payload)` on line 17

☐ (K) None of the above

☐ (L) ——

Solution: Because stack canaries can be bypassed, it would be fairly easy to bypass non-executable pages and execute a return-to-libc or ROP chaining attack using the `write_byte` helper.

ASLR can be bypassed because the format string vulnerability allows the address of the stack to be leaked, such as by leaking the value of the SFP using `%x` or `%p`.

Rewriting the code in a memory-safe language prevents all memory safety attacks, since the language is memory-safe.

Rewriting the code to use the safe function `fgets` removes the last memory safety vulnerability in this code, making it memory-safe.

Q3 *Memory Safety Mitigations* **(12 points)**

Suppose we are on a 64-bit system, and we have an address space of 2^{50} bytes.

Q3.1 (3 points) How many unused bits are available for pointer authentication in each address?

○ (A) None ○ (B) 4 ○ (C) 11 [✓ (D) 14] ○ (E) 17 ○ (F) 32

Solution: Addresses are 64 bits, and we need 50 bits to address the entire address space, so there are $64 - 50 = 14$ unused bits available for pointer authentication.

Q3.2 (3 points) Regardless of your answer to the previous part, for the rest of the question, assume that 10 bits are used for pointer authentication in each address.

Additionally, for the rest of the question, assume that 64-bit stack canaries are enabled. The first byte of the stack canary is always a null byte.

Assume the attacker does not have the ability to create their own pointer authentication codes (PACs). How many bits does the attacker have to guess correctly to guess the stack canary and the PAC?

○ (G) 0 ○ (H) 10 ○ (I) 56 ○ (J) 64 [✓ (K) 66] ○ (L) 74

Solution: The stack canary has 56 bits to be brute-forced (the canary is 64 bits long, but there is a constant null byte, which is 8 bits). The attacker must also guess the 10 bits in the PAC. In total, there are $10 + 56 = 66$ bits that must be guessed correctly.

Q3.3 (3 points) Now assume that the attacker has a format string vulnerability that lets them read any part of memory while the program is running.

Assume the attacker does not have the ability to create their own PACs. How many bits does the attacker have to guess correctly to guess the stack canary and the PAC?

○ (A) 0 [✓ (B) 10] ○ (C) 56 ○ (D) 64 ○ (E) 66 ○ (F) 74

Solution: Since the attacker can read memory, they can read the stack canary value, so they don't need to guess the stack canary. However, they still need to guess the 10-bit PAC. (Recall that the secrets used for generating PACs are stored in the CPU and are not accessible to the program memory.)

Q3.4 (3 points) Assume the attacker is interacting with a remote system. Provide one defense that would make brute-force attacks infeasible for the attacker. (Please answer in 10 words or fewer.)

Solution: Possible answers: Timeouts. Rate-limiting. Attacker is blocked from making too many guesses.

Other answers are possible too.