CS161 Spring 2026 — Introduction to Computer Security — Exam Prep 3

Q1 *Echo, Echo, Echo* (20 points)

Consider the following vulnerable C code:

```
#include <stdio.h>
#include <stdlib.h>

char name[32];

void echo(void) {
    char echo_str[16];
    printf("What do you want me to echo back?\n");
    gets(echo_str);
    printf("%s\n", echo_str);
}

int main(void) {
    printf("What's your name?\n");
    fread(name, 1, 32, stdin);
    printf("Hi %s\n", name);

    while (1) {
        echo();
    }

    return 0;
}
```

The declarations of the used functions are as given below.

```
// execute the system command specified in 'command'.
int system(const char *command);
```

Assume you are on a little-endian 32-bit x86 system. Assume that there is no compiler padding or additional saved registers in all questions.

Q1.1 (2 points) Assume that execution has reached line 8. Fill in the following stack diagram. Assume that each row represents 4 bytes.

Stack
1
2
RIP of `echo`
SFP of `echo`
3
4

○ (1) - RIP of `main`; (2) - SFP of `main`; (3) - `echo_str[0]`; (4) - `echo_str[4]`

○ (1) - SFP of `main`; (2) - RIP of `main`; (3) - `echo_str[0]`; (4) - `echo_str[4]`

○ (1) - RIP of `main`; (2) - SFP of `main`; (3) - `echo_str[12]`; (4) - `echo_str[8]`

Q1.2 (3 points) Using GDB, you find that the address of the RIP of `echo` is `0x9ff61fc4`.

Construct an input to `gets` that would cause the program to execute malicious shellcode. Write your answer in Python syntax (like in Project 1). You may reference `SHELLCODE` as a 16-byte shellcode.

Q1.3 (4 points) Which of the following defenses on their own would prevent an attacker from executing the exploit above? Select all that apply.

□ Stack Canaries

□ Pointer authentication

□ Non-executable pages

□ ASLR

○ None of the above

Q1.4 (5 points) Assume that non-executable pages are enabled so we cannot execute `SHELLCODE` on stack. We would like to exploit the `system(char *command)` function to start a shell. This function executes the string pointed to by `command` as a shell command. For example, `system("ls")` will list files in the current directory.

Construct an input to `gets` that would cause the program to execute the function call `system("sh")`. Assume that the address of `system` is `0xdeadbeef` and that the address of the RIP of `echo` is `0x9ff61fc4`. Write your answer in Python syntax (like in Project 1).

Hint: Recall that a return-to-libc attack relies on setting up the stack so that, when the program pops off and jumps to the RIP, the stack is set up in a way that looks like the function was called with a particular argument.

Q1.5 (6 points) Assume that, in addition to non-executable pages, ASLR is also enabled. However, addresses of global variables are not randomized.

Is it still possible to exploit this program and execute malicious shellcode?

○ Yes, because you can find the address of both **`name`** and **`system`**

○ Yes, because ASLR preserves the relative ordering of items on the stack

○ No, because non-executable pages means that you can't start a shell

○ No, because ASLR will randomize the code section of memory

Q2 *The Way You Look Tonight* (22 points)

Consider the following vulnerable C code:

```
typedef struct {
    char mon[16];
    char chan[16];
} duo;

void third_wheel(char *puppet, FILE *f) {
    duo mondler;
    duo richard;
    fgets(richard.mon, 16, f);
    strcpy(richard.chan, puppet);
    int8_t alias = 0;
    size_t counter = 0;

    while (!richard.mon[15] && richard.mon[0]) {
        size_t index = counter / 10;
        if (mondler.mon[index] == 'A') {
            mondler.mon[index] = 0;
        }
        alias++;
        counter++;
        if (counter == ___ || counter == ___) {
            richard.chan[alias] = mondler.mon[alias];
        }
    }

    printf("%s\n", richard.mon);
    fflush(stdout); // no memory safety vulnerabilities on this line
}

void valentine(char *tape[2], FILE *f) {
    int song = 0;
    while (song < 2) {
        read_input(tape[song]); //memory-safe function, see below
        third_wheel(tape[song], f);
        song++;
    }
}
```

For all of the subparts, here are a few tools you can use:

- You run GDB once, and discover that the address of the RIP of `third_wheel` is `0xffffcd84`.
- For your inputs, you may use `SHELLCODE` as a 100-byte shellcode.
- The number `0xe4ff` exists in memory at address `0x8048773`. The number `0xe4ff` is interpreted as `jmp *esp` in x86.

- If needed, you may use standard output as `OUTPUT`, slicing it using Python 3 syntax.

Assume that:

- You are on a little-endian 32-bit x86 system.
- There is no other compiler padding or saved additional registers.
- `main` calls `valentine` with appropriate arguments.
- **Stack canaries** are enabled and no other memory safety defenses are enabled.
- The stack canary is four completely random bytes (**no null byte**).
- `read_input(buf)` is a memory-safe function that writes to `buf` without any overflows

Write your exploits in Python 3 syntax (just like in Project 1).

Q2.1 (4 points) Fill in the following stack diagram, assuming that the program is paused at **Line 14**. Each row should contain a struct member, local variable, the SFP of `third_wheel`, or canary. The value in each row does not have to be four bytes long.

Stack

Q2.2 (6 points) In the first call to `third_wheel`, we want to leak the value of the stack canary. What should be the missing values at line 21 in order to make this exploit possible?

Left:

Right:

For the rest of this question, **ASLR** is enabled in addition to stack canaries. Assume that the code section of memory has not been randomized.

Q2.3 (4 points) Provide an input to each of the lines below in order to leak the stack canary in the first call to `third_wheel`. If you don't need an input, you must write "Not Needed."

Provide a string value for `tape[0]`:

Provide an input to `fgets` in `third_wheel`:

Q2.4 (8 points) Provide an input to each of the lines below in order to run the malicious shellcode in the second call to `third_wheel`. If you don't need an input, you must write "Not Needed."

Q3 *Memory Safety: Everyone Loves PIE* (13 points)

Consider the following vulnerable C code:

```
1  void cake() {
2    char buf[8];
3    char input[9];
4    int i;
5
6    fread(input, 9, 1, stdin);
7
8    for (i = 8; i >= 0; i--) {
9      buf[i] = input[i];
10   }
11   return;
12 }
13
14 void pie() {
15   char cookies[64];
16
17   // Prints out the 4-byte address of cookies
18   printf("%p", &cookies);
19
20   fgets(cookies, 64, stdin);
21   cake();
22   return;
```

Stack at Line 6
RIP of `pie`
SFP of `pie`
(1)
RIP of `cake`
(2)
`buf`
(3)
`i`

Assumptions:

- `SHELLCODE` is 63 bytes long.
- ASLR is enabled. All other defenses are disabled.

Q3.1 (1 point) What values go in blanks (1) through (3) in the stack diagram above?

○ (1) `&p` (2) SFP of `cake` (3) SFP of `printf`

○ (1) cookies (2) SFP of `cake` (3) `input`

○ (1) `cookies` (2) SFP of `cake` (3) RIP of `fgets`

○ (1) RIP of `printf` (2) SFP of `printf` (3) `input`

Q3.2 (1 point) Which vulnerability is present in the code?

○ Off-by-one ○ Signed/unsigned vulnerability

○ Format string vulnerability ○ Time-of-check to time-of-use

In the next two subparts, you will provide inputs to cause `SHELLCODE` to execute with high probability.

(Question 3 continued...)

Let `OUT` be the output from the `printf` call on Line 18. Assume that you can slice this value (e.g. `OUT[0:2]` returns the 2 least significant bytes of `&cookies`). You may also perform arithmetic on this value (e.g. `OUT[0:2] + 4`) and assume it will be converted to/from types automatically.

Q3.3 (2 points) Provide a value for the `fgets` call on Line 20.

□

Q3.4 (5 points) Fill in each blank with an integer to provide an input to the `fread` call on Line 6.

You must put an integer for every blank even if the final slice would be equivalent – for example, you must put both “0” and “7” in the blanks for `OUT[0:7]`, even though `OUT[:7]` is equivalent.

Note that the `+` between terms refers to string concatenation (like in Project 1 syntax), but the minus sign in the third term refers to subtracting from the `OUT[_:_]` value.

```
'A' * □ + OUT[□:□] + (OUT[□:□] - □)
```

Q3.5 (2 points) Which of these defenses, if enabled by itself, would prevent the exploit (without modifications) from working? For pointer authentication only, assume the program runs on a 64-bit system.

□ Stack canaries

□ Non-executable pages

□ Pointer authentication

○ None of the above

Q3.6 (2 points) Which of these variables would cause the exploit to break?

○ RIP of `pie` = `0x10c3fa00`

○ address of `cookies` = `0xffff5fc0`

○ RIP of `cake` = `0x10237acf`

○ SFP of `cake` = `0xffffcd04`