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

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

Consider the following vulnerable C code:

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  char name[32];
5
6  void echo(void) {
7      char echo_str[16];
8      printf("What do you want me to echo back?\n");
9      gets(echo_str);
10     printf("%s\n", echo_str);
11 }
12
13 int main(void) {
14     printf("What's your name?\n");
15     fread(name, 1, 32, stdin);
16     printf("Hi %s\n", name);
17
18     while (1) {
19         echo();
20     }
21
22     return 0;
23 }
```

The declarations of the used functions are as given below.

```
1  // execute the system command specified in 'command'.
2  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]`

Solution: The first two items on the stack are the RIP and SFP of `main`, respectively. Since the stack grows down, lower addresses are at the bottom of the diagram, and arrays are filled from lower addresses to higher addresses and are zero-indexed. As such, row (3) contains `echo_str[12]`, and row (4) contains `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.

Solution: Where to put the `SHELLCODE` does not matter. This is a simple stack-smashing attack: we want to redirect execution to `SHELLCODE` when the `echo` function returns.

Approach 1: Place the Shellcode in the Buffer

```
SHELLCODE + 'A' * 4 + '\xb0\x1f\xf6\x9f'
```

Approach 2: Place the Shellcode above the RIP

```
'A' * 20 + '\xc8\x1f\xf6\x9f' + SHELLCODE
```

There may be a few other correct answers here (with the shellcode placed at slightly different offsets within the buffer or above the RIP), but these are the most common.

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
- ■ ASLR
- ■ Pointer authentication
- ○ None of the above
- ■ Non-executable pages

Solution: Stack canaries defend against this attack because we are consecutively writing from the local variables to the RIP. The canary would be checked when the `echo` function returns, and because we don't have a way to leak the value of the canary in our exploit, canaries effectively stop our exploit from succeeding.

Non-executable pages defend against our exploit by preventing the shellcode on the stack (a write-not-execute region of memory) from being executed.

If ASLR were enabled, we wouldn't be able to reliably find the address of the RIP - it would change every time! We'd have to use one of our special attacks specifically for ASLR to bypass this (e.g. ROP). As such, ASLR stops our original exploit from succeeding.

Pointer authentication would require us to forge a valid pointer authentication code along with our new RIP. We don't have a way to do this, so pointer authentication stops our exploit from succeeding.

Note: we received a handful of questions asking if the "Pointer Authentication" answer choice was referring to a 32-bit system, which is what was stated in the prologue of this question, or a 64-bit system, which is what we originally intended. As such, we awarded credit for both answer choices.

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.

```
'A' * 20 + '\xef\xbe\xad\xde' + 'B' * 4 + \xd0\x1f\xf6\x9f' + 'sh' + '\x00'
```

Solution: Our goal is to make `echo` return to the system function by changing the `RIP of echo` to the address of `system`. When `echo` returns to `system`, the stack should look like the stack diagram below, because by calling convention the callee expects its arguments and its `RIP` to be pushed onto the stack by the caller. **It's the callee's responsibility to push the `SFP` onto the stack as its first step.**

Therefore we need to first place garbage bytes from the beginning of `name` up to the `RIP of echo` (`'A' * 20`) and replace the `RIP of echo` with the address of `system` (`'\xef\xbe\xad\xde'`) so that `echo` will return to `system`. Now, we want to create the stack diagram above to make the stack in line with what the `system` method expects. Thus, we add four bytes of garbage where the `system` method expects `RIP of system` to be. Note that, `RIP of system` is the address that `system` method will return to. Then, we place the address of "sh" at the location where `system` expects an argument, and place the string "sh" at that location (which is 8 bytes above the `RIP of system`).

Stack

`command` (pointer to "sh")
(Expected) RIP of `system`

As such, our exploit may look something like the above answer.

NOTE: Since the stack below the `RIP of echo` will get invalidated (because it's below the ESP) after `echo` returns, we cannot make any assumptions about whether the values placed there would remain as-is. Therefore, you should not place the string `"sh"` in `name`.

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

Solution: If ASLR is enabled, the address of `system`, a line of code in the *code* section of memory, will be randomized each time the program is run. Because our exploit uses this address, ASLR will effectively prevent us from using our approach!

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

Consider the following vulnerable C code:

```
1  typedef struct {
2      char mon[16];
3      char chan[16];
4  } duo;
5
6  void third_wheel(char *puppet, FILE *f) {
7      duo mondler;
8      duo richard;
9      fgets(richard.mon, 16, f);
10     strcpy(richard.chan, puppet);
11     int8_t alias = 0;
12     size_t counter = 0;
13
14     while (!richard.mon[15] && richard.mon[0]) {
15         size_t index = counter / 10;
16         if (mondler.mon[index] == 'A') {
17             mondler.mon[index] = 0;
18         }
19         alias++;
20         counter++;
21         if (counter == ___ || counter == ___) {
22             richard.chan[alias] = mondler.mon[alias];
23         }
24     }
25
26     printf("%s\n", richard.mon);
27     fflush(stdout); // no memory safety vulnerabilities on this line
28 }
29
30 void valentine(char *tape[2], FILE *f) {
31     int song = 0;
32     while (song < 2) {
33         read_input(tape[song]); //memory-safe function, see below
34         third_wheel(tape[song], f);
35         song++;
36     }
37 }
```

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

RIP of `third_wheel`
SFP of `third_wheel`
Stack Canary
`mondler.chan`
`mondler.mon`
`richard.chan`
`richard.mon`
`alias`
`counter`

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:255	Right:47

Solution: Both `fgets` and `strcpy` insert a null byte at the end of their inputs, so we need to overwrite the null bytes that are located at `richard.mon[15]` and `mondler.chan[15]` (since we can use `strcpy` to write more than 16 bytes into `richard.chan`). Since `alias` is a signed value, we can use 255 to overwrite the null byte in the `richard.mon` buffer, and 47 to overwrite the null byte in the `mondler.chan` buffer.

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]`:

```
'B' * 47
```

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

`'B' * x`, where $x \geq 15$

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."

```
'B' * 48 + OUTPUT[64:68] + 'B' * 4 + '\x73\x87\x04\x08' + SHELLCODE
```

`\x00 or 'B' * x`, where $x \geq 15$

Solution: The solution to 9.5 and 9.6 follow the same logic as 9.3 and 9.4 except that we replace the address of `(RIP + 4)` with the address of the `jmp *esp` instruction since ASLR is enabled.

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`

Solution: Here's the stack diagram. It's not needed to solve this subpart, but to clarify later solutions we'll label the addresses relative to the address of cookies (which is the one address we know, because of the print statement).

`&cookies + 68`	`RIP of pie`
`&cookies + 64`	`SFP of pie`
`&cookies`	`cookies`
`&cookies - 4`	`RIP of cake`
`&cookes - 8`	`SFP of cake`
`&cookies - 16`	`buf`
`&cookies - 25`	`input`
`&cookies - 29`	`i`

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

Solution: The big clue that an off-by-one attack exists is `buf` being 8 bytes, and `input` being 9 bytes. In particular, the for loop is iterating 9 times and causing 9 bytes of `input` to be copied into the 8-byte `buf` array. This causes the byte directly after `buf` to be overwritten.

There's no format string vulnerability, because in the one and only call to `printf`, the attacker does not control the 0th argument where the percent formatters are placed.

There's no signed/unsigned vulnerability, because the numbers in `fgets` and `fread` are hard-coded, and `i` is never interpreted as an unsigned integer.

There is no time-of-check to time-of-use vulnerability, because the program never pauses (which might cause an input to be correct at time-of-check but incorrect at time-of-use).

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

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.

SHELLCODE

Solution: `buf` and `input` cannot fit the 63-byte shellcode, so `cookies` is the only possible place to put shellcode.

The `fgets` call writes at most 63 bytes into `cookies`, which means that after writing shellcode here, there's no more space to write anything else in `cookies`.

As we'll see in the next subpart, there's nothing else that needs to be placed in `cookies` to complete the output.

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' *` [4] `+ OUT[` [0] `:` [4] `] + (OUT[` [0] `:` [1] `] -` [16] `)`

Solution: The last blank can also be 25 instead of 16.

`OUT` prints the 4-byte address of `cookies` (which is where we put shellcode).

The for loop causes the 9 bytes of `input` to be copied into `buf`. This means that the byte immediately after `buf` can also be overwritten. This byte is the LSB of the SFP of `cake`.

In the off-by-one attack (as seen in Project 1), we can overwrite the SFP of `cake` to point 4 bytes below the place where we put the address of shellcode.

We can overwrite the SFP to point at the address of `buf`. Then, 4 bytes after the start of `buf`, we can write the address of shellcode.

The first 4 bytes of `buf` are garbage, then the next 4 bytes are `OUT[0:4]`, the address of shellcode. (Note that the slice here doesn't do anything since the output is already 4 bytes, but the question requires we put an integer in every blank.)

The 9th and final byte of input needs to change the SFP of `cake` to point at the address of `buf`. Per the stack diagram, we calculated this to be 16 bytes below the address we leaked. Since we can only overwrite a single byte, we slice out the LSB of the address of `cookies`, which is `OUT[0:1]`, and subtract 16 from this value.

`OUT[0:1] - 25` also works (i.e. last blank could also be 25), since this would cause the SFP to point at `input`. The first 4 bytes of input are also garbage, and the next 4 bytes of input are also the address of shellcode, so this solution also works.

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 ■ Pointer authentication

■ Non-executable pages ○ None of the above

Solution: Stack canaries: True. The off-by-one attack now overwrites the LSB of the canary instead of the LSB of the SFP.

Non-executable pages: True. The shellcode was written on the stack, so if non-executable pages were enabled, it would not be possible to execute user-inputted code.

Pointer authentication: True. Pointer authentication codes would break the exploit since we're changing a pointer value (SFP of `cake`) without modifying its corresponding pointer authentication code.

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

- ○ RIP of `pie` = `0x10c3fa00`
- ○ RIP of `cake` = `0x10237acf`
- ● address of `cookies` = `0xffff5fc0`
- ● SFP of `cake` = `0xffffcd04`

Solution: Recall that the SFP of `cake`'s value is the address of the SFP of `pie`. If the SFP of `cake` is `0xffffcd04`, this means the address of the SFP of `pie` is `0xffffcd04`, and the stack looks like this:

`0xffffcd08`	RIP of `pie`
`0xffffcd04`	SFP of `pie`
`0xffffccc4`	`cookies`
`0xffffccc0`	RIP of `cake`
`0xffffccbc`	SFP of `cake` (value: `0xffffcd04`)
`0xffffccb8`	`buf`
`0xffffccb4`	`input`
`0xffffccb0`	`i`

The value of the SFP of `cake` is `0xffffcd04`. However, we want to overwrite this value with the address of `buf`, which is `0xffffccb8`. It is no longer possible to perform the off-by-one exploit, since we have to change the 2 least-significant bytes in order to change the address correctly.

At a high level, the problem here is that the LSB of the addresses were close to 0x00, which caused the second-least significant byte to roll over, preventing the off-by-one exploit from working.

If you try drawing out a similar stack diagram with the address of `cookies`'s value set to `0xffff5fc0`:

`0xffff6004`	RIP of `pie`
`0xffff6000`	SFP of `pie`
`0xffff5fc0`	`cookies`
`0xffff5f80`	RIP of `cake`
`0xffff5f7c`	SFP of `cake` (value: `0xffff6000`)
`0xffff5f78`	`buf`
`0xffff5f74`	`input`
`0xffff5f70`	`i`

We see that this is also broken, so both answers were accepted for credit. **This was not originally intended as a right answer, but became correct after the sized of `cookies` was changed from 16 to 64.**

The two options with RIP values show addresses in the code section, which are irrelevant to our exploit.