Homework 4

Team Members: Shu-Wei Hsu (sh2575), Aris Huang (th625)

Exercise 3.1) Briefly explain the vulnerabilities in target1.c, target2.c, target3.c, and target4.c.

target1.c: This is a classic buffer overflow vulnerability in the foo(). The name buffer is declared with size of 4 bytes, but *strncpy* is used to copy up to 12 bytes from temp, which is the argument. Thus, if temp is larger than 4 bytes, it will overwrite adjacent memory, which could include the function's return address. We can put the nop sled and shellcode in env since the buffer is too small and overwrite the return address that points to the nop sled.

target2.c: Similar to target0, but there's a check for input_size. We can utilize integer overflow to bypass the check.

target3.c:

```
int intvalues[4];
memcpy((char*)intvalues, values, sizeof(int)*5);
```

The buffer is 16 bytes. However, the `memcpy` copies 20 bytes. This gives us a chance to overwrite saved EBP. We can overwrite the saved EBP to the address of the shellcode. Since when a function returns, ebp is moved into esp.

target4.c: Similar to typical buffer overflow attack. However, instead of injecting shellcode, the attacker overwrites the return address with the address of an existing function in libc, like system(). The attacker also needs to set up the stack to pass the desired arguments to the libc function. For example, passing "/bin/sh" to system() to spawn a shell. This allows the attacker to execute arbitrary commands without injecting new code, bypassing the non-executable stack protection.

Exercise 3.2) Does ASLR help prevent return-to-libc attacks? If so, how? If not, why not?

Yes, ASLR can make return-to-libc attacks more difficult to execute. However, it does not completely prevent them. There are a few reasons why ASLR elevates the difficulty of such an attack. First, ASLR randomizes the memory addresses where system and library code segments are loaded, meaning that the specific addresses an attacker needs to use in a return-to-libc attack change every time the program is executed. This makes it particularly hard to predict the correct address to jump to.
However, this is only a partial mitigation strategy, meaning that if the attacker manages to find a way to discover the leaded or randomized base addresses, return-to-libc attack may still be possible. Furthermore, if the attacker can discover or bypass the randomization, ASLR alone is not sufficient to prevent such attacks. This is also why ASLR is most effective when combined with other security measures like non-executable stack or control-flow integrity. A way for an

attacker to still perform a return-to-libc attack is to leak memory addresses from vulnerable processes, use other methods to bypass ASLR (ex. Brute-force memory address, feasible if entropy of the randomization is low), or call functions like system() with correct arguments to spawn a shell.

In sum, ASLR is merely a mitigation technique that is designed to make exploiting vulnerabilities more difficult, but it is not an absolute protection on its own.

Exercise 3.3) Explain the stack protector countermeasure, which we turned off via the "-fno-stack-protector" option to gcc. Which of your exploits would be prevented by turning on this stack protection?

The stack protector in gcc is a security feature that aims to prevent buffer overflow by placing a small, random value (known as canary), on the stack just before the return address. When a function is called, the canary is set. Before the function returns, the canary's value is checked. If a buffer overflow does in fact occur and overwrote the stack, it is likely to also overwrite the canary. When the function attempts to return, the altered canary value will be detected, resulting in the program getting terminated with a "stack smashing detected" error. This effectively prevents the return address from being used to redirect the program flow to malicious code.

Thus, turning on the stack protection would prevent exploits that rely on overwriting the stack frame of a function to modify the return address. This means that since target1.c, target2.c, target3.c, and target4.c all deal with overflow, all four exploits would be prevented.

For target1 and target3, they involve direct stack buffer overflows that overwrite local variables and return address. It would be thwart by the stack protector, as the canary would be corrupted before reaching the return address.

As for target2 and target4, similar to target1 and target3, if the overflow does reach far up the stack, the exploits will be prevented by this measure.