CSSE 332 -- Operating Systems

Control Flow Hijacking

Name: Solution Key

The workflow below will guide you through working with `part2.c` in order to exploit it and have it run `print_bad_outcome` without the programmer calling this function explicitly. Before starting, please make sure that you have generated your cookie per the lab instructions. Also, please make sure to turn off `ASLR` using `./disable_aslr.sh`, and when you are done, please re-enable it using `./enable_aslr.sh`. **If you do not do so, all of your attempts below will fail.**

Question 1. (5 points) Examine `part2.c` and find where the vulnerability in this code is. To force the user to execute `print_bad_outcome`, what would an attacker need to do?

> **Solution:** The programmer is using the `gets` function, which is not safe and can lead to buffer overflows. The attacker can use that fact to overwrite the return address with that of `print_bad_outcome` so that when the vulnerable function returns, it jumps into the wrong spot.

Question 2. Launch `part2.bin` in `gdb` and find the following information by examining the stack and assembly code of the program.

Here are some helpful `gdb` and `gef` commands:

- `run`: run your program after loading it in `gdb`.
- `context`: refresh the `gef` context.
- `b num`: set a breakpoint at line number `num`.
- `b function`: set a breakpoint at the function `function`.
- `b *0xdeadbeef`: set a breakpoint at instruction at address `0xdeadbeef`.
- `continue` or `c`: continue executing after stopping at a breakpoint.
- `n`: move to the next line of code.
- `s`: step into the next line of code.
- `ni`: move to the next instruction.
- `si`: step into the next instruction.

- `x $esp`: examine memory at address contained in `$esp`.
- `x 0xdeadbeef`: examine memory at address `0xdeadbeef`.
- `x/16w $esp`: examine 16 words of memory, starting at address in `$esp`.
- `p var`: print the variable `var`.
- `p &var`: print the address of the variable `var`.

(a) (5 points) What is the address of the instruction that `vulnerable_fn` should return to? In other words, what is the *return address* of `vulnerable_fn`?

Solution: The answer will vary, need to pick this one up by launching `gdb`.

(b) (5 points) By examining the stack frame of `vulnerable_fn`, where is the return address stored? In other words, what is the memory address where the return address above is stored.

Solution: The answer will vary, need to pick this one up by launching `gdb`.

(c) (5 points) What is the address of the starting byte of the buffer `input`?

Solution: The answer will vary, need to pick this one up by launching `gdb`.

Question 3. (5 points) Using the above information, describe the exploit that you would want to run. Make sure to be specific as to what values should go in which addresses.

Solution: We would want to overwrite the `input` buffer with enough bytes so that we reach the return address (by reaching the address where the return address is stored). We then inject the address of our target function in there and watch the fireworks.

Question 4. (5 points) Using the above information, how many bytes should you overwrite before being able to modify your target return address?

Solution: Answer will vary, but it is the different between the address of the return address and the address of `input`.

Question 5. (5 points) What value would you want to replace the original return address with to achieve your goal? What would your exploit string look like?

Solution: The address of our target 'bad' function. The exploit string looks like a bunch of random bytes followed by the address of the target function.

Watch out for how the address should be passed.