Brian Truong
CS161-ht
SID: 23428686

Homework 1

Q1) a. False, "Security through obscurity" relies on the general concept that the secrecy of the design or implementation of a system provides a level of security, it does not refer to the practice of hiding a user's password when being entered.

b. False, it is possible to allow the malicious user to write arbitrary data to memory using format string bugs, so for example, overwriting the return address will lead to a control-hijack. Thus, it is definitely possible to use format string bugs to control-hijack.

c. True, the NX bit feature will prevent malicious code that is on the stack to be executed, this means that even though a malicious user may overflow the stack it will not be able to obtain a root shell since it requires execution.

d. False, by overwriting the exception handler we can prevent the stack canaries from raising any errors to the operating system and to overcome the NX we can use ROP. Lastly, for the ASLR we can still access instruction addresses and other non-ASLR modules so we can run the shellcode by using a jmp $esp like we did on the project. Thus, by using all of these methods in conjunction we can successfully perform a buffer overflow attack.

e. False, a library call from rich library may have more exploitable areas (maybe even have more powerful functionalities like libc) for a malicious user to take advantage of when trying to hack the system, thus it is not always a better idea to use library calls with rich functionalities at the cost of security.

f. False, line coverage only insures that each line is tested for an input, but does not guarantee that an untested malicious or malformed input will not result in a bug, such as string and format string exploits.

Q2) a. Buggy, when memmove on line 5 hits the last character in the string, then the p+2 input will go out of bounds, which is a memory-safety bug as you are now potentially writing into memory and corrupting other surrounding data.

b. Buggy, off-by-one bug on the line with the for loop (line 6), this can be easily seen when the step size is 1 and since the variable i in the for loop is initialized to 0 it will end up trying to index more elements than the array contains, which may lead to a memory error i.e. corrupting one of the surrounding variables.

c. Buggy, because for the integer “track” in the update function we can pass a negative integer value, which would end up setting memory preceding the tracklen buffer to newtracklen, which may lead to a memory-safety bug as we are writing over memory.

Q3) a. n1 == sizeof(s1), n2 == sizeof(s2) and char s1[], char s2[] are not NULL
b. i < n1
c. i + j < n-1, equivalent to j < n2 - 1
d. i + j < n

Q4) a. The vulnerability here involves the memcpy function call that tries to copy 8 bytes from the array "in", to a much smaller character array "a" that only holds 1 byte per index. By doing so, the contents of writing to a single index in array "a" would overflow into the following data spaces, which means when you get closer to the end of the array "a", this overflow would overwrite memory outside of the memory allocated for "a" and may lead to a buffer overflow attack.

b. For example, len of value 20 will allow the for loop to go up to i = 19, so accessing the 19th index of array "a" is valid, however memcpy will try to write 8 bytes of data from array "in" to array "a" which will overflow by 7 bytes since "a" can only take in 1 byte. Thus, data will be written at an index greater than sizeof(a[]), and memory will be written into. (perhaps the return address is overwritten to point to malicious code)
note: any i >= 19 would overflow the boundary of the char a[25] with index starting at 0

Q5) a. After drawing out of the path coverage of **target**, it turns out to get to line 15 you need to follow the true path for three branches. Solving for these constraints (x >= 0 && y >= 0, f(x) == x + 18, and y < 20) I found there is only one value x can hold (x = 8), and six values y can hold (y = 14, 15, 16, 17, 18, 19) such that line 15 tries to set a memory address outside of the space allocated to buf to 0 and crash the program.

To calculate the approximate number of test cases needed to generate a crash at line 15, we use the known values: 6 pairings that crash the program, 2^{64} total pairings, and $2^{64} - 6$ pairings that will not crash the program at line 15. Let X be all of the good inputs we can produce. $X = x_1 + x_2 + \cdots x_a = \sum_{i=1}^{a} x_i$ Calculating the expected value of the number of test cases needed:

$$E(x) = E\left(\sum_{i=1}^{2^{64}-6} x_i\right) = \sum_{i=1}^{2^{64}-6} E(x_i)$$

$$E(x_i) = (2^{64} - 6)E(x_i),$$

$$E(x) = (2^{64} - 6) * \left(\frac{1}{7}\right)$$

$$= (2^{64} - 6) * \left(\frac{1}{7}\right) + 1$$

this rounds to …

$$= 2.64\ x\ 10^{18} \text{ test cases for expected crash}$$

b. Path constraints = $((x_0 \geq 0)$ && $(y_0 \geq 0))$
&& $(3 * x_0 + 2 == x_0 + 18)$
&& $(y_0 < 20)$

c. **assert**$(\boldsymbol{x_0 + 3 * y_0 + 2 \leq 49})$

d. The first constraint gets rid of all negative values, the second constraint requires a single x_0 value that satisfies the equality, which is $x_0 = 8$, the third constraint narrows y_0 to $0 \leq y_0 < 20$. Finally, since we want a pair of values that will cause a buffer overflow, the assertion must fail, so we solve ! $(x_0 + 3 * y_0 + 2 \leq 49)$ which gives six valid pairings.
Instance of buffer overflow pairing: $\boldsymbol{x_0 = 8,\ y_0 = 15.}$
(all pairings that would cause crash: (8,14), (8,15), (8,16), (8,17), (8,18), (8,19))

Q6) a. Using probability, create a mapping of the probability of mistyping a particular character on the keyboard. Additionally, calculate other values such as most statistically popular mistypes for particular keyboard characters, such as ‘w’ being a popular mistype for ‘e’ since they are both adjacent to each other. With this information, generate a threshold that will allow a user to log on even with typos in their password as long as their error rate falls below the set threshold. As for the user error rate, it should be generated from several factors such as length of password, how many characters are mistyped, how common are the typos. Thus, the user error rate is properly weighted to account for multiple parameters in a mistyped password, so the threshold can apply any user password.

b. Due to the modification of allowing users to mistype their password and still being able to login (up to a certain error threshold) the security of password authentication would be weaker. For instance, brute forcing the password would be easier, as you only have to get a deviation of the password (under a certain error threshold) so there is more likelihood to guess a password that would grant access. As for human nature, a lot of people have similar passwords, such as ‘12345’ or ‘password’ and many of these passwords are used for their simplicity and ease of use, thus a hacker can take advantage of this human nature in conjunction with the modification from part (a) to increase malicious logins. Therefore, false positives would definitely increase as there will be more hackers gaining access to accounts they should not be having access to. As for false negatives, there will be less of them because users will be able to mistype their passwords and still gain access. Thus, this modification is good for legitimate users, but it makes it easier for malicious users to hack into accounts.

c. From the problem we have 2^{30} words, a 0.002 false negative rate, 10 characters and a 2% typo rate for users. Let X be the number of errors that we will allow:

Firstly, given we want a 0.002 false negative rate:

$$P(X \leq k) \geq 0.998$$

Then we have,

$$\sum_{i=0}^{k} P(x = i) = \sum_{i=0}^{k} (10 Choose i) * (0.98)^{10-5} * (0.02)^{i}$$

$$additionally\ k = 2, so\ we\ want\ 2\ or\ fewer\ typos$$

For our parameters we need to consider all combinations of differing positions, the chance of properly typing out the character correctly, and the chance of typing the character incorrectly and how these parameters influence the summation.

So, let user in if they have 2 or fewer typos.
Therefore, the false negative rate would be < 0.002 and the false positive would be $FP = 1/2^{30}$

Q7) a. (xorl) 0x2ef4face, (addl) 0x2ef4cafe, (movl) 0x2ef4beef

b.

D: 0x2ef4beef
C: 0x2ef4cafe
B: 0x2ef4face
A: 0x00000000

c.

E: 0xbfdecaf0
D: 0xbfdecae0
C: 0x1ffa4b28
B: 0x1ffa4b28
A: 0x00000000

d.

J: 0xbfdecaf8
I: 0x00000000 (this can be anything, we are done before eip gets here)
H: 0x1ffa4b28
G: 0xbfdecaf0
F: 0x2ef4dead
E: 0x1ffa4b28
D: 0xbfdecae0
C: 0x2ef4dead
B: 0x1ffa4b28
A: 0x00000000