Casper Kristiansson 2022-10-17

Homework 2

Question 1

A)

We know that the purpose of a threat model is to provide the defenders with an idea of what needs to be protected in a system and how malicious users would attack a system. Depending on what type of software might be reversed engineering will change the threat model. The reason why people reverse engineer software is to get an understanding of how a system works.

If the software is in the form of an executable, it usually means that an attacker might reverse the steps to get its source code of it. There are multiple steps that the defender can take to make the reverse engineering part much harder.

B)

Linear disassembling is the process of reverse engineering a program by disassembling it sequentially. This means that you start the first byte in the code section and decode every byte by mapping a byte to the corresponding assembly instruction.

C)

Recursive disassembling is the process of reverse engineering a program by disassembling it in a recursive pattern. This means that rather than decoding every single byte of the code the disassembler will follow the jump and branch instructions.

The advantage of using **recursive** disassembling rather than **linear** disassembling is that linear disassembling could be vulnerable for example embedded data in the code section which could be intentionally left by the developer. While if this would be the case for recursive disassembling it wouldn’t be a problem because the algorithm will execute the jump and branch instructions.

D)

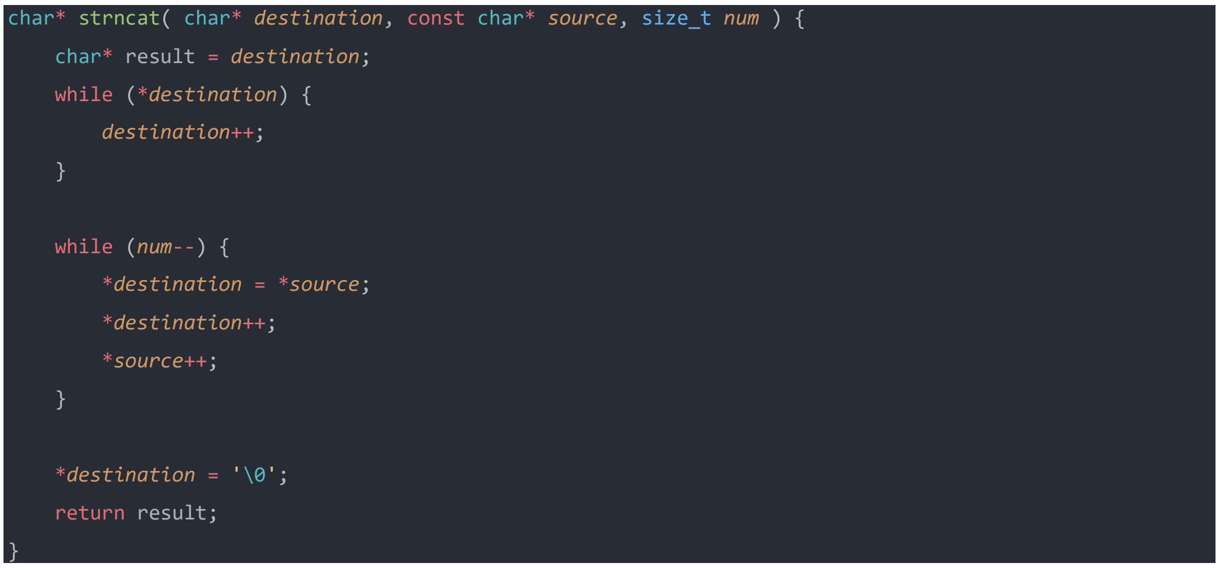
Control flow is the order in which statements and instructions are executed. This means that the process of control-flow recovery is by running assembly code and reconstructing it to C/C++ (Control flow Graph Recovery). This means that by using pattern matching after mapping the assembly instructions it can identify what type of assembly code resembles the C/C++ code. But doing this will not always replicate the source code because different types of code snippets could represent the same assembly code.

Question 2

A)

As stated the function **strcat()** will append two strings by removing the null character from the destination string and appending the new string. Because of this if the function is misused it allows a user to perform a buffer overflow attack. The meaning of misused is that if for example, the developer doesn't know exactly how big a source string a buffer overflow attack could be performed (source + destination is bigger than the assigned destination size).

B)



```
char* strncat( char* destination, const char* source, size_t num ) {
    char* result = destination;
    while (*destination) {
        destination++;
    }

    while (num--) {
        *destination = *source;
        *destination++;
        *source++;
    }

    *destination = '\0';
    return result;
}
```

C)

The problem that **strncat()** solves is that the developer can specify the exact amounts of characters that should be appended to the destination string. By doing this we can make sure that the destination buffer never overflows.

D)

If the function **strncat()** is properly used it is safe. But if the destination is declared for example with the size 10 and then the $size\_t\ num$ + the destination is bigger than 10 buffer overflows will happen.

Question 3

A)

$$BEFORE: buf2 = 22222222\ AFTER: buf1 = 1111111111111111111111111111111122222222$$

B)

The security issue with using $memset()$ and then $printf()$ is because how strings work in c++. Strings uses the character "$\backslash 0$" to inform that it is the end of the string. When we use $memset()$ to fill the chars the character is never added. For example, the function $printf()$ will keep writing to stdout until it finds the character. If the character is not existing, it might for example print garbage or other malicious code. The problem with $memset()$ is also that we specify the length of characters to add to the buffers. This means if we specify anything bigger than their actual size, we will end up with overflowing the buffer.

C)

Stack memory is where the local declared variables of functions are (it also includes methods and references to variables). This means that the variables are declared and stored during runtime, meaning that they are only temporarily stored and erased. Stack memory is allocated and deallocated automatically when a certain method is executed.

Heap memory for example is used to store global variables. But also, every time when an object is created it is created in the Heap memory where then the stack contains referencing information to it. The difference between stack memory is that it doesn't allocate and deallocate automatically, and therefore garbage collectors need to be used to remove old objects if the goal is to use the memory efficiently.

Question 4

A)

The problem with this function is that if the user tries to input an integer value bigger then 2147483647 will cause the function to be vulnerable to malicious users. For example, if we use the input value 4294967296 the function will return the item located at index 0 in the array. The security issue that is triggered is stack overflow because the integer array is located on the stack.

B)

The best way to solve this is to have a check at the beginning of the program that the integer that is passed as an argument does not overwrite the maximum size of a int. By doing this the function can detect if the input is invalid and then return for example -1.

Question 5

A)

Opaque predicate is an expression that will be evaluated to either true or false. The outcome of the evaluation is already known by the developer by the statement still needs to be executed. Example:

```
int x = 100;
if (x+x >= x*x) {}

time_t now = time(NULL);
time_t yesterday = now - 24 * 60 * 60;
if (now < yesterday) {}
```

Both examples are basic but, in both situations, we can clearly see what the outcome of the evaluation will be. This means that in both statements we could store a lot of unused code to make the process of reverse engineering harder.

B)

Opaque predicate works that for example, the compiler cannot statically resolve the condition and therefore needs to evaluate it. Because of this, the program will contain instructions that are unnecessary and has no effect on the result which will eat away at its performance of it. Therefore, it is important when opaque predicates are used the condition should not take long for the compiler to evaluate. For example, you should not use extreme math calculations which are hard to evaluate but rather simple ones that are hard for someone who hasn't written the expression to understand but is quick to evaluate.

C)

As mentioned above because of the conditions there will be a lot of instructions that will be executed that have no effect on the result of the program. But in a lot of cases, developers will also use a lot of junk code in the wrong path to make it harder to understand exactly what will be executed. Meaning that there exists a lot of unreachable code in the executable which will impact its size of it.

Questions 6

A)

Fuzz testing could be launched against this function by simply testing random values of the parameter $string\ s$.

B)

The first error that fuzzer would identify would be the divide by 0 error because running the program triggers an error where the program will crash. This happens if you input an empty string. There is also an error if the value $NULL$ is sent to the function.

C)

More difficult bugs for fuzzer to find is usually logical errors. In the code snippet we can see that the logical error that exists is that we divide an int with the size to get the ratio. The only problem is that we want the function to return a value between 0 and 1 but instead it will only return the values 0 or 1. This can simply be fixed by changing the variable $int\ n$ to $float\ n$.