Samuel Petit
Homework 2 - COMP3632
Student no. 20683298

Written Assignment

Question 1

(a) Linear disassembling tries to disassemble everything sequentially, starting from the first byte in the code to the last byte, which means that it is easy for it to fail in multiple cases, one of them being treating data embedded within the code as an instruction.

(b) Recursive disassembly is less susceptible to these mistakes because the code isn't disassembled in a linear manner and jump/branch instructions are followed, meaning that it wouldn't treat embedded data as an instruction and jump to a correct instruction instead.

Question 2

(a) Metamorphic malware is a malware which changes the form for each of its instances of software, meaning any single instance of such a malware will be different from the other.

(b) It is usually used in order to avoid being detected by pattern matching malware detection. By creating a unique instance of malware, it will result in the same malware with a different signature, making it harder to detect.

(c) Metamorphic malware would not evade such detection because while it may disguise itself in order to change its signature and such, however its goal will remain to exploit a system somehow which means it will need to make system calls. A classic metamorphic malware with no particular protection against behavior-based malware detectors would then be detected.

(d) Yes it can be used, for example it could be effective to protect against reverse engineering of some license verification by making it a lot harder to crack.

Question 3

(a) Encrypted malware will have a different signature for each instance because it is uniquely encrypted. Similarly, polymorphic malware will also encrypt its body giving it a different signature every time, however it is different in that it will change the decryption code such that the decryption will always have a different signature.

(b) As I previously stated in the previous answer, a polymorphic malware will encrypt its body and have a changing decryption method. On the other hand, a metamorphic virus fully rewrites itself with every iteration so that every version of the code is different from the preceding one.

Question 4

(a) Strcpy is unsafe because it will not check has no way of knowing the size of the destination buffer as it contains no size parameter with the amount of values to copy. So if it happens that the source of values to copy is bigger than the destination, then it could lead to buffer overflows as it will keep writing into further memory addresses that do not belong to the destination variable.

(b)

```
char* strncpy(char *dest, const char *src, size_t n) {
    size_t i = 0;

    for(; i < n && src[i] != '\0'; i++) {
        dest[i] = src[i];
    }
    for(; i < n; i++) {
        dest[i] = '\0';
    }
    return dest;
}
```

(c) Strncpy has a set amount of values to copy whereas strcpy will copy all the values from the src pointer. Making it possible to avoid buffer overflows by making sure to use a safe amount of values to copy.

(d) Yes it can by not using the size parameter safely: let’s say we set the size to be the length of ‘src’, or that we don’t block the size from being bigger than the dst variable then it could still lead to buffer overflow.

Question 5

(a) The program output is :
   - BEFORE: buf2 = 22222222 AFTER: buf2 = 11122222

(b) Heap memory is implemented in decrementing order, which means buf1 will have a higher starting memory address than buf2. Thus when we compute the difference in addresses of buf2 - buf1. The fact that printing the values in buf2 gives us 3 ones at the beginning shows us that the call memset on buf1 overflowed into a part of the heap which has been assigned to the second buffer, thus a heap overflow occurred.

(c) Stack is used for static memory which the heap is dynamically allocated, thus, in C programmers need to allocate and deallocate memory from the heap while the program is running. The stack is also fast, it will hold local variables, return address and such, while the heap is slower (not limited in size though if the machine uses virtual ram) and hold objects, big buffers and such.

(d) Both stack and heap overflow essentially work in the same way, except that the stack is static and heap memory is dynamically allocated. This makes it generally harder to overflow in the heap than on the stack, it is also harder to know what will happen in a heap overflow as adjacent memory addresses could hold anything (or not be allocated) and thus lead to undefined behaviors.

Question 6

(a) An opaque predicate is a condition where the result is known in advance by the programmer, it is a branch that always executes in one direction, that direction is known by the programmer but not by the analyst. It makes it really hard or in some cases impossible until the code runs to know to know what the next instruction might be. It is a good defence against reverse engineering as it makes the software more complicated to break.

(b) Collatz conjecture works as following :
   - If x is even then : f(x) = x/2
   - If x is odd then : f(x) = 3x + 1

   Thus we can build opaque predicates by knowing that any starting value we pick, going into this function a number of times will always give us 1.

(c) I think it could be feasible however it may be really hard and require a lot of trial and error. Assuming someone is able to train a model well enough that it generates accurate enough opaque predicates, there will always be a small chance that such a generated predicate does not work or is wrong. There is lots of research happening at the moment to try and understand at a deep level how such machine learning modules work, and until we understand this, we might not be able to create such a model that is always correct.

Question 7

(a) It is possible to reobfuscate p1. The advantage to doing so is that it would become more diverse, however the costs of computation becomes higher the more obfuscation we apply to it as we have to evaluate every single one to get the final value.

(b) We assume that the obfuscation program does not bring any bug to the software, thus if any of the obfuscated programs have a bug then it must be the original code which has a bug and thus any other piece of obfuscated code will also have this same erroneous output.

Question 8

(a) We could use mutation based fuzzing (or dumb fuzzers), they will test our function with unexpected inputs and we will be able to make sure that it works as expected.

(b) One problem it might find is that there is nothing preventing this function from trying to divide by 0 and thus crashing in the process. By testing many inputs there will be one which tries to divide by 0 and thus detect it that way.

(c) It may be more difficult to detect the problem in integer division in this program. Dividing an integer by another integer in java will also result in an integer value and thus loose the decimals, in this case we want to keep the decimal values and thus this would be a problem. A fuzzer would not detect this however since it is not going to prevent the program from running.

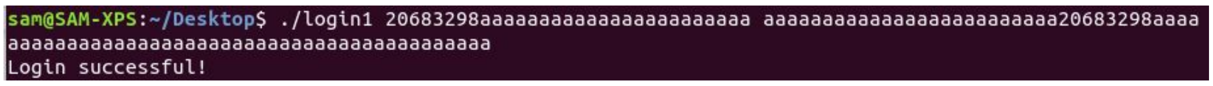
Programming Assignment: Buffer Overflow

<u>Login 1</u>

The username and password I used (in this order):

20683298aaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaa20683298aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaa

The command I ran:



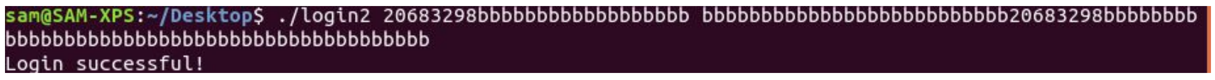
```
sam@SAM-XPS:~/Desktop$ ./login1 20683298aaaaaaaaaaaaaaaaaaaaaaa aaaaaaaaaaaaaaaaaaaaaaaaa20683298aaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
Login successful!
```

The reason this works is that we fill the username and make sure to put more characters in the input password such that it overflows the buffer used to check the username and password. This is possible in this case by exploiting the unsafe C method strcpy.
What will happen is that the input password will be copied into the “good password” and “good username” thus we just need to make sure to align my student number such that the 2 is the first value that is overflowed into the “good username”.

Login 2

The username and password I used (in this order):
20683298bbbbbbbbbbbbbbbbbb
bbbbbbbbbbbbbbbbbbbbbbbbb20683298bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
bbbbb

The command I ran:



```
sam@SAM-XPS:~/Desktop$ ./login2 20683298bbbbbbbbbbbbbbbbbb bbbbbbbbbbbbbbbbbbbbbbbbbbbbbb20683298bbbbbbb
bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
Login successful!
```

The reason this works is the same as for login1. The only difference being that login 2 uses a canary which checks for buffer overflows. The canary is set to be ‘b’ as we can see from the code, which means that the program won’t be able to detect an overflow as long as the value in the address of the canary remains ‘b’ (we make sure that the value overflowed into the canary memory address remains indeed ‘b’).