# 

# 

# 

C2 Report

Group 1

Alexander Nuccitelli

Christiana Vancura

Killian Crittenden

Rose Chakraborty

Segun Oluwole

# 

# 

# 

# Table of Contents

[**Table of Contents 2**](#_vtudtacwvb9e)

[**Creation Task 3**](#_uj5te9jv0yah)

[Background 3](#_n042ozkj8qjf)

[Hidden Feature 3](#_eviuntx0uoby)

[Hiding Techniques 3](#_ouw9y4eygsym)

[Layer 1 3](#_ofjm8a4ax3vj)

[Layer 2 4](#_b8rcehj9akm8)

[Layer 3 4](#_898i0hlsfkn4)

[Red herrings & other inconveniences 4](#_vshs1fozlpld)

[**Reversing Task 5**](#_a0qt751kgcbq)

[Initial Discoveries 5](#_y5osa5tewzos)

[Reversing Process 5](#_salh0gabmh72)

[Ghidra 5](#_xsay2ecz6qfm)

[Layer 1 5](#_ec62vsphz7ds)

[Ghidra Decompiled Code: 7](#_p4xnt68d68ic)

[Helper Script 7](#_8f35576b6qt7)

[Layer 2 8](#_vwejh7hfkg3q)

[Ghidra Decompiled Code: 10](#_8msdh5mzeqtr)

[Layer 3 12](#_4yolihhv7o8c)

[Ghidra Decompiled Code: 13](#_j50f1jr4i8x4)

[Helper Script 14](#_of83n5ivr3ec)

[Additional Tools 15](#_pssasmx965ql)

[TA Confirmation 15](#_ewapen3trqhg)

[Outcomes 15](#_7nqqs8jntq2d)

[**Sources 16**](#_f3aflcufjube)

# Creation Task

## Background

Our program has a series of three layers that are divided into: a stream cipher utilizing a PRNG for static password encryption, a dynamically generated XOR password that changes every time the program runs, and a binary-modified function to decode a series of integers into the final password. The program also includes red herrings with image encryption and networking

Steps to run :

Run as sudo

sudo apt-get install libssl-dev

Run sudo ./C2 to run executable

We created the code in C++ and used GitHub for code sharing. The code was compiled with g++ C2.cpp -o C2 -lssl -lcrypto in the Makefile.

## Hidden Feature

Our program passes if a series of three passwords are entered in the right order. Our first layer password is **HfXpTcnk<&9{htN.F@A!Yw** which is found by figuring out which random values are used in the basic stream cipher. The second layer password is generated randomly every time the program is run and is found by combining all the split up passwords in different folders. The last layer password is **112, 47, 116, 83, 106, 82, 39, 70, 52, 49, 47, 110, 200, 100, 93, 93, 72, 48, 57, 47, 49**  is found by tracking the function responsible for transforming integers at runtime and reversing the arithmetic. Note: The integers need to be entered in order one after the other.

## Hiding Techniques

### Layer 1

* The hidden feature in this layer is the encryption of the password using a stream cipher, where the cipher’s randomness is based on a pseudo-random number generator (PRNG). The critical element to uncover is the key passed into the PRNG, as the sequence of random numbers it generates will dictate the encryption/decryption process. The password itself is encrypted using this random sequence, and to decrypt it, you need to reverse-engineer the PRNG setup.

### Layer 2

* The second layer password is dynamically generated during the program’s execution using an XOR function applied to a pre-defined string. This string, along with the XOR transformation, is hidden and not directly visible in the program’s source code or static assets. The password is split into several fragments, each hidden in different files on the system, which must be located, extracted, and recombined in a specific order(i.e. Based on the current system time) to reconstruct the full password. This means that at different times, the password fragments might yield different results depending on when the program is run.

### Layer 3

* The third layer involves the binary modification of certain functions during runtime. These functions are designed to perform an encryption operation (such as addition or right-shifting), but during runtime, their functionality is dynamically modified to perform the inverse operation (e.g., subtraction or left-shifting). This process allows the program to decrypt the password, but this behavior is not immediately obvious without understanding how the program is modifying its own execution. The password for this layer is given as a sequence of integer values, which represent characters or data that must be decoded after the function has been altered. The binary patching technique is effectively hiding the decryption logic in the program’s runtime state rather than in its static code.

### Red herrings & other inconveniences

Various other red herrings and inconveniences were implemented in order to slow down the reversing process.

* Various shutdowns were implemented based on the time of day, if a debugger was present, and other miscellaneous checks. These would shutdown your virtual machine when triggered. Since the code had to be run in sudo, the permissions for these commands were already present.
* All system calls were scrambled in a function that took a big long string, and decrypted parts of the string using set seed randomly generated values. This allowed for important system calls to be scrambled and mixed with unimportant system calls, adding further confusion and also allowed for the masking of shutdown commands.
* All function names were scrambled to random movies that the team enjoyed. Since function names are stored in the binary, scrambling the function names gave the reversers no insight into what the different functions did.
* There are lots of different anti debugger checks that are present throughout the code. These checks shutdown or exit the program if a debugger is detected.
* When the code is forcibly stopped, an interrupt handler removes all important files. Although the files it removes aren’t actually important and are red herrings.
* If a debugger is present the code will also ping google and write to a file called “network\_part1.txt”. This is a complete red herring.
* Functions were written that actually just always return 0 or 1. These functions are used in place of some 0’s and 1’s used throughout the code.
* Code for creating a pixelated image was present. This code simply created a pixelated image and is a red herring.
* Code for a SHA256 hash was present. This code did nothing relevant and is a red herring.

# Reversing Task

## Initial Discoveries

The file was unzipped and the binary was the only thing that was present. The binary was then run, it asked for an integer. The code was then run through Ghidra and the reversing process began. We did not try running Wireshark or gdb because upon initial viewing of the code it was apparent that we did not need them as there were no network interactions and all values could be figured out without a debugger.

## Reversing Process

### Ghidra

Ghidra was used to reverse all of the code that was present.

### Layer 1

Layer 1 asked for 10 integer inputs to pass to the next layer, it also printed the correct integer if the wrong one was input at any step along the path. Upon inspection of the decompiled code it was apparent that these “random” numbers were actually pseudo-random and were generated by the start time of the code. In lines 34 and 35 the code calls time(0) to get the system time in seconds since 1970, in line 36 the code sets the seed using a formula and the variables defined in lines 34 and 35. Another seed is then generated and set in lines 37 and 38 and the code then loops while waiting for the 10 inputs from the user. Inside the loop it calls rand() to generate a seeded number and then the expected input is that random number % 10000 + 1.

After understanding how the code generates the seeds, we wrote a script that could be run at the same time as the program and generate the 10 random numbers needed to crack layer 1. However, because of different start up latencies between the reversed code and the helper function they could not be run together in real time. Despite the fact that they cannot be run at the same time, the script showed only 7 possible combinations of 10 numbers that would be generated at runtime, so it became a brute force game. After a couple runs the correct first value will be found and after that the rest are known and layer 1 is passed.

#### Ghidra Decompiled Code:

### 

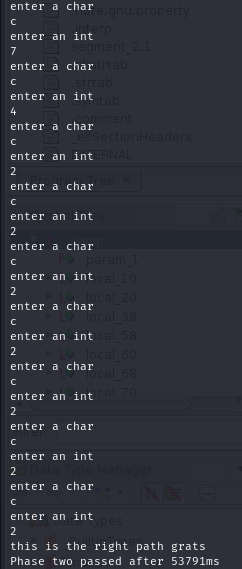
#### Helper Script

#### Passing Layer One:

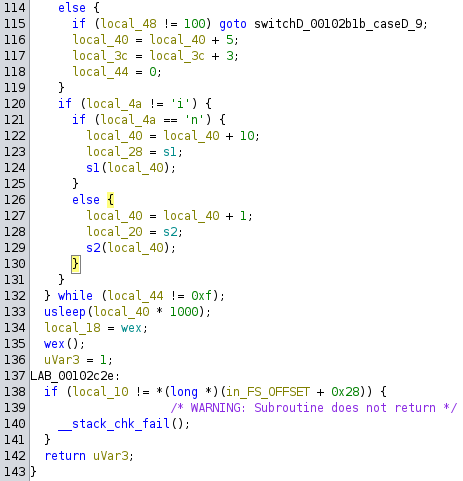
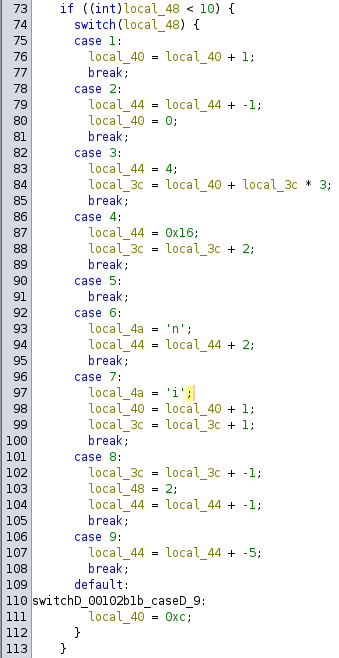
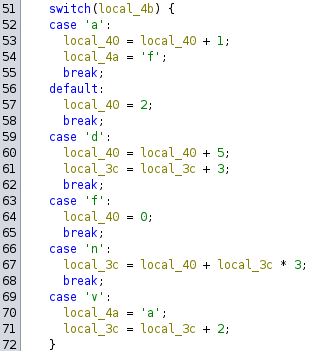
#### 

### Layer 2

Layer 2 called for the input of one character(local 4b) and one integer(local 48) repetitively until the variable named local 44 is equal to 0xf(line 132 below). Entering any character that was not a,d,f,n,v, and then the integer 7 in the first prompt would result in continuous execution of the program. If the integer 7 was not entered in the first prompt then the variable local 4a would not be set to ‘i’ and the executable would be deleted(lines 120 to 130 where function s1 calls n\_we() and s2 calls function s1 which calls n\_we() and results in deletion of the executable). Because the desired value for local 44 is 0xf then in the second prompting for a character and an integer, the same character would be entered along with the integer 4 which would set local 44 to 0x16. Then in the next prompting, the same character would be entered along with the integer 2 which would then subtract 1 from the value stored in local 44(lines 78-80 below). This is done 7 times to set the value in local 44 equal to 0xf which results in the passing of phase 2. We reversed this layer by examining the decompiled code in Ghidra and testing out different characters and integers as we reasoned out what was occurring in this part of the executable.



#### Ghidra Decompiled Code:

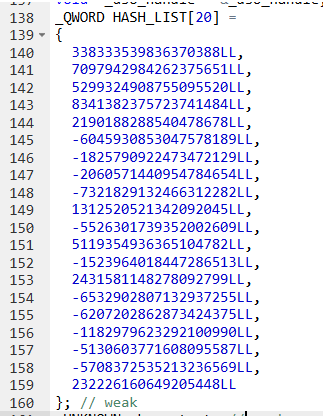


### Layer 3

#### Ghidra Decompiled Code:

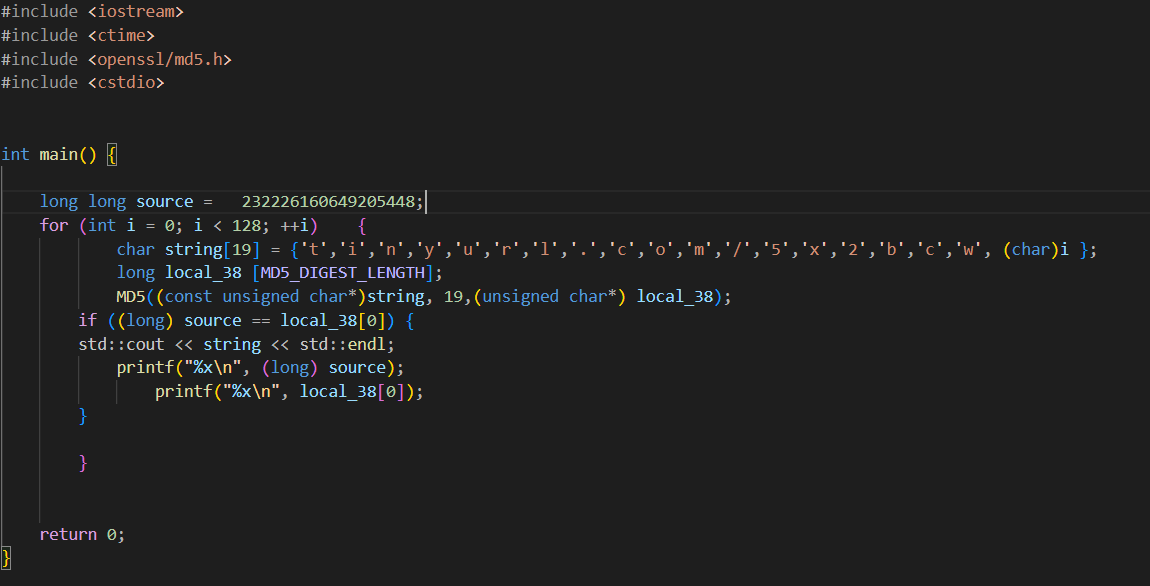


Layer 3 called for the input of a string. Upon further investigation, this layer takes in a 20 character string and then hashes it with an MD5 hashes. It first hashes the first character, then it hashes the first two characters, all the way to hashing all 20 characters. Each time it hashes a character, it is compared with a hashed value in the hash list. The hash list didn’t immediately appear in Ghidra, so the code was plugged into dog bolt and the hash list was found using Hex-Rays.

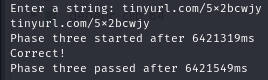


#### Helper Script

Since the input is a string, and it first only hashes one character, and since there are only 128 different ascii characters, a brute force attack is possible. A helper script was written that hashes all 128 characters and compares them with index 1 of the hash list (the for loop in their code started with 1). Once the first character was correctly identified, the script was modified to keep the first character constant and change the second character. Then the 2 character strings were hashed and compared to index 2 of the hash list. This process was repeated until 19 of the 20 characters were found.



After finding almost all of the characters we were left with this string, “[tinyurl.com/5x2bcwj](http://tinyurl.com/5x2bcwj)”. The last character was missing and not present in the hash list. However the string appeared to be a tiny url leading to a website. So a brute force attack was used by plugging the string into the website with different letters and numbers (tiny urls can only be upper or lowercase letters and numbers), until a valid link was reached. The valid link was “[tinyurl.com/5x2bcwjy](http://tinyurl.com/5x2bcwjy)” which led to a rick-roll. When inputting the final string into layer 3 the output said success and was shown below.



### Additional Tools

Dogbolt ([Decompiler Explorer](https://dogbolt.org/?id=63a5f0bb-dead-4cc7-95f4-e29d3f5b8989#BinaryNinja=1022&RetDec=193&Ghidra=972&Hex-Rays=142&angr=280&Reko=884&Relyze=6)) was also used to extract the hash list.

## Outcomes

The code was successfully reversed and all layers were passed to reveal the hidden payload, which was a rick-roll. The code was relatively simple to reverse and we ran into no major hiccups while reversing.

# Sources

The video, the source code and the binary are all present in the submission folder.