CS 395 Final Report

Ransomware Lab Project

JT Reagor

4/16/2023

Introduction

With the rise of helpful resources like ChatGPT and platforms such as Stack Overflow, computer scientists have been able to increase their efficiency by orders of magnitude. They no longer need to fiddle with boilerplate code for hours. Rather, they can get a template from some generous Stack Overflow user and use ChatGPT to adapt it to their needs. Programming is getting quicker and easier. This is mostly a very good development. However, as the coding skill floor is lowered with these tools, Pandora's box of incompetency is opened. This problem is seen most evidently in classrooms. Professors can assign incredibly difficult homework that students pass with flying colors, but the exams will not reflect their apparent genius. In order to pass classes, students often resort to finding answers online. Then, when it comes to the exam, their crutch is nowhere to be found, and they are left with unsatisfactory knowledge of the topic.

This was very clear when I took CS 270, introduction to systems programming, with Dr. Calvert, specifically on one project, the bomb lab. The bomb lab is a project created by Drs. O'Hallaron and Bryant from Carnegie Mellon University [1]. Through the project, its creators intended to teach students about concepts such as reverse engineering, bit diddling, and assembly code. The project also teaches students how to use the debugger GDB effectively. It is a very well-designed project, and it has been incredibly useful to me. I have talked about it multiple times during interviews with companies that are looking for students with these exact skills. However, when discussing the project with other students, it was clear that not many of them struggled through it as intended. Instead, students gave up quickly and googled the answers [2].

The aim of my project is not to solve this cheating crisis, but to provide a new alternative to the bomb lab that does not have its answers online yet. Hopefully, my project will provide professors at the University of Kentucky with more ammunition with which to teach students well. In my project, I sought to teach similar concepts to the bomb lab in a different context.

Learning Outcomes

When creating this project, I had the following outline for desired learning outcomes. These learning outcomes are similar to those of the bomb lab. These outcomes helped guide the design of the project by determining what clues, resources, and prompts I would give them.

1. Students will demonstrate competent use of debuggers like GDB.
2. Students will demonstrate understanding and proper use of cryptographic building blocks.
3. Students will demonstrate understanding of assembly instructions.
4. Students will demonstrate understanding of bitwise operations.

5. Students will demonstrate their ability to reverse engineer executables.

Project Context

While the context of the bomb lab is that a malicious actor has deployed an assembly-code bomb that you must defuse by inputting the correct codes, I decided to set my project in a slightly more realistic scenario. In my project, the student takes the role of a cybersecurity contractor that has been contacted by a company in response to a ransomware attack that encrypted some of their critical files.

This context hopefully introduces the students to a simulation of some situation that could actually happen. Getting students familiar with real world situations will prepare them for interviews and future careers.

Project Content

Professors using this project will obtain a package containing 8 files. These 8 files make up the entirety of what I coded for this project. The manifest of these files is below:

1. driver.py: Run by professors to create student tarballs
2. Makefile: Compiles malware.c, decrypter.c
3. README: Useful instructions for professors
4. malware.c: Drives the encryption process
5. support.c: Contains code hidden from students like the encryption algorithm
6. support.h: Contains constants necessary for support.c
7. decrypter.c: Drives example decryption algorithm
8. supersecretfileorig.txt: Original file to be encrypted and copied into student tarballs

In describing the above files and their content, I will not provide any code snippets in this document, but all the files are available on my GitHub [3].

1. Driver.py

Driver.py is to be used by professors to generate a large amount of tar files to distribute to students. It takes two stdin inputs: number of packages and difficulty. The difficulty parameter is currently unused, as I only completed one level. For each student, the driver generates a new 4-byte hexadecimal key and places it in a file named “keyseed”. Then, it makes a copy of “supersecretfileorig.txt” and encrypts it using the malware executable and corresponding keyseed. It creates a new folder and places in it a copy of the malware executable, a copy of the malware.c source code, the newly created keyseed, and the newly encrypted text file. The driver then compresses the folder and deletes the old folder. In total, the student receives 4 files:

1. Malware.c (source file)

2. Malware (executable)

3. Supersecretfile.txt (encrypted text file)

4. Keyseed (text file)

2. Makefile

The Makefile has a few notable features, otherwise it is fairly standard and unimportant. The first feature is the –O0 flag. I learned to use this flag by looking at the bomb lab's Makefile. This flag restricts almost all compiler optimization. This keeps the compiler from performing most magic and leaves the assembly code pretty true to the original c code. The second feature is the –ggdb flag. This flag compiles the executable with support for GDB built in. This makes the student's lives easier. The Makefile has three routines: malware, decrypter, and clean. The default is malware, and it compiles the malware executable. This must be done before the driver is used as instructed in the README.

3. README

The README contains a list of the files necessary for the driver and makefile to work. It also contains instructions for professors to create the project packages.

4. Malware.c

Malware.c is one of the files that the students will receive. It has one function, the main function, that takes 1 argument. This argument is the name of the file to be encrypted. It opens the file to be encrypted and the keyseed file. Then, it reads in the file data from both, placing the data to be encrypted in a buffer and the keyseed in an unsigned integer. It calculates how many blocks will be encrypted. These parts of the encryption algorithm are shown in C code to give the students some general information about the general algorithm structure — that it is a block cipher with a key of size 4 bytes. Then, it passes a pointer to the beginning of the buffer into the encrypt function. The encrypt function is not contained in this source file.

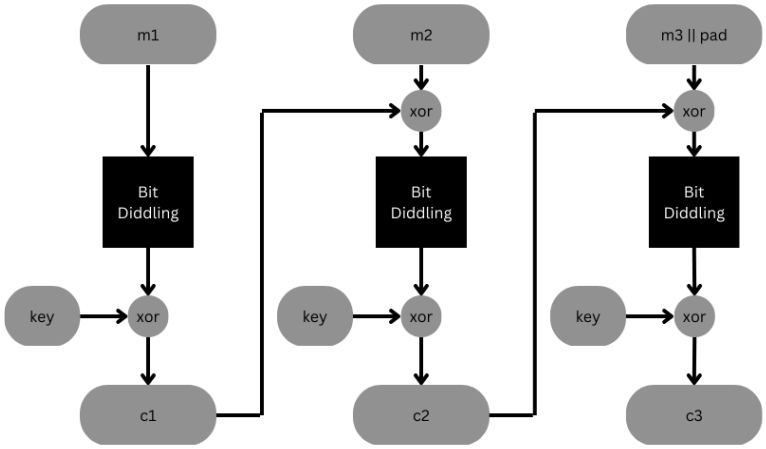
The hope for this source file is that it provides enough information to the students to give them a starting point in their reverse engineering process. Then, it references the support.h file that contains the actual implementation of the encryption algorithm.

5. Support.c

This file contains the implementation for the encrypt function referenced in malware.c. This file is not given to students, but it is referenced by malware.c. This way, the C code is not shown to the students, but the assembly instructions are incorporated into the executable. This file also contains the implementation for the example decryption function.

Encryption Algorithm

I designed the encryption algorithm in support.c with the intent that it would be both vulnerable and modular. The structure that I chose to use is a block cipher with block size 4 bytes with cipher block chaining. The general structure can be seen below.



There are three main steps in this encryption process: convolution, key xoring, and the block cipher mode of operation. The convolution step, labeled in the diagram as "bit diddling", is the step with the most modularity and room for expansion. In the current state of the project, the bit diddling flips the nibbles of each byte of plaintext. In reverse engineering the assembly instructions for this, the students will learn about bit masks and logical shifts. This could be swapped out for some other convolution process like an S-box operation or more complex bit diddling. The key xoring is self-explanatory, but to understand what is happening, the student will have to keep track of what register is holding the key. Finally, the cipher block chaining will require students to keep track of what is being referenced on the stack. The assembly code peeks backwards 4 bytes from the pointer's current location on the stack. The student should see this and understand that it is referencing the previous block of ciphertext.

6. **Support.h**

This is the header file for support.c. It contains the function declarations and one macro definition: KEY_LEN. KEY_LEN is used in malware.c and is set to 4.

7. **Decrypter.c**

Decrypter.c drives the implementation of the example decryption function in support.c. It is compiled by the makefile using the "decrypter" target. It takes one argument, like the malware executable, the name of the file to be decrypted.

8. **Supersecretfileorig.txt**

This is a template file for the encrypted file that each student will receive. It is a plaintext file containing important TIC Corp passwords. This file is copied and encrypted by each student's unique key.

Additional Project Content

During the creation of this project, I also wrote up a student assignment document. This document sets up the context for the assignment and instructs the student on how to complete it. This file can be found on my GitHub [3] in the "writeups" folder.

As outlined in the student document, the student's objective is to reverse engineer the encryption algorithm and produce a program that can decrypt the sample encrypted file. The deliverable is a source file in Python, C, or C++ depending upon the professor's preference. It must be able to take a file name as input and decrypt the file in place. A satisfactory example can be found in the decrypter.c source file.

Lessons Learned

In the production of this project, I learned lessons that developed my technical skills and lessons that sharpened my general understanding of cybersecurity practices. In creating the encryption algorithm, I was able to write my first full symmetric key block cipher. Translating my knowledge of cryptography into practice took more work than I anticipated. However, I was encouraged to see that the work that I had put into learning the theory translated well into practice. Understanding modes of operation helped me develop the algorithm into a more fleshed out piece of work. In working out the reverse engineering process, I learned the most effective ways to use GDB. This made me realize how painful I had made the bomb lab for myself. I am curious how long the bomb lab would take me now if I went back and tried it with my enhanced debugging skills.

The most significant takeaway that I got from this project was this: never write your own encryption. Even when I was trying to make a vulnerable encryption algorithm, I found myself discovering holes in it in places that I hadn't expected. It only takes one oversight to leave a critical vulnerability. Using time-tested algorithms and standards is the most secure way to protect important information.

References

1. Bomb lab (date accessed: 4/20/2023): http://csapp.cs.cmu.edu/3e/labs.html
2. Bomb lab answers (date accessed: 4/19/2023): https://john.coffee/pages/binary-bomb-lab
3. GitHub repository (date accessed: 4/19/2023): https://github.com/jt-reagor/cs395proj