Towards Advanced Malware Classification:

A Reused Code Analysis of Mirai Botnet and Ransomware

by

Yeonjung Lee

A Thesis Presented in Partial Fulfillment
of the Requirements for the Degree
Master of Science

Approved July 2020 by the
Graduate Supervisory Committee:

Youzhi Bao, Chair
Adam Doupé
Yan Shoshitaishvili

ARIZONA STATE UNIVERSITY

August 2020

ABSTRACT

Due to the increase in computer and database dependency, the damage caused by malicious codes increases. Moreover, gravity and the magnitude of malicious attacks by hackers grow at an unprecedented rate. A key challenge lies on detecting such malicious attacks and codes in real time by the use of existing methods, such as a signature-based detection approach. To this end, computer scientists have attempted to classify heterogeneous types of malware on the basis of their observable characteristics. Existing literature focuses on classifying binary codes, due to the greater accessibility of malware binary than source code. Also, for the improved speed and scalability, machine learning-based approaches are widely used. Despite such merits, the machine learning-based approach critically lacks the interpretability of its outcome, thus restricts understandings of why a given code belongs to a particular type of malicious malware and, importantly, why some portions of a code are reused very often by hackers. In this light, this study aims to enhance understanding of malware by directly investigating reused codes and uncovering their characteristics.

To examine reused codes in malware, both malware with source code and malware with binary code are considered in this thesis. For malware with source code, reused code chunks in the Mirai botnet. This study lists frequently reused code chunks and analyzes the characteristics and location of the code. For malware with binary code, this study performs reverse engineering on the binary code for human readers to comprehend, visually inspects reused codes in binary ransomware code, and illustrates the functionality of the reused codes on the basis of similar behaviors and tactics.

This study makes a novel contribution to the literature by directly investigating the characteristics of reused code in malware. The findings of the study can help cybersecurity practitioners and scholars increase the performance of malware classification.

DEDICATION

I would like to extend my sincere gratitude to my advisor, Professor Tiffany Bao, who introduced me the fascinating field of reused threats and supported my research program during my study.

First off, under her supervision, I started working on the malware evolution by analyzing the malware data from Symantec Inc., then delved into the ransomware operation by investigating Ghidra by conducting a reverse-engineering method. By utilizing Ghidra, I performed comparative analyses among the popular ransomware programs, such as, WannaCry, Petya and NotPetya. Then, she guided me to investigate malware source codes, such as Mirai botnet. Consequently, I was able to identify malware reused codes and code modification cases. She has been encouraging and nurturing me throughout this long journey, so I now embark on a new chapter of my academic endeavour in this fascinating area. So, I am very thankful to Tiffany for her support and guidance.

I would also like to thank Professor Fish Wang for his help and support by answering my unstructured questions and encouraging me. I am very thankful for Professors Adam Doupe and Yan Shoshitaishvili for providing invaluable comments and feedback on the earlier version of my manuscript. It is an absolute joy and honor to be part of SEFCOM and I would like to thank its current and past members. I cannot forget all the help from all lab members.

Last but not least, I would like to thank to my wonderful family who make me finish this work. Thank you my little girls, Chloe and Katie, and my hubby Sang-Pil. Without your support, I could not have made this far.

TABLE OF CONTENTS

Page

LIST OF TABLES . . . v

LIST OF FIGURES . . . vi

CHAPTER

1 INTRODUCTION . . . 1

2 BACKGROUND . . . 4

2.1 Definition of Reused Code . . . 4

2.2 Reused Code in Malware Source Code . . . 4

2.3 Reused Code in Malware Binary Code . . . 5

2.4 Motivations for Code Reusing in Malware . . . 5

2.5 Related Work . . . 6

3 EXPERIMENTS AND FINDINGS . . . 8

3.1 Functionality and Vulnerability . . . 8

3.1.1 Mirai Botnet . . . 8

3.1.2 WannaCry and NotPetya . . . 10

3.2 Mirai Botnet – Reused Code in Malware Source Codes . . . 12

3.3 Petya, WannaCry, and NotPetya - Reused Code in Binary Code . . . 17

4 LIMITATIONS AND FUTURE WORK . . . 20

5 CONCLUSION . . . 22

REFERENCES . . . 24

APPENDIX

A REUSED CODE IN AKIRU MIRAI CONNECTION.C FILE . . . 26

A.1 Reused Code in Wget . . . 27

A.2 Reused Code in Tftp . . . 27

B REUSED CODE IN PETYA AND NOTPETYA . . . 29

APPENDIX Page

B.1 Writing on Physical Drive 30

B.1.1 Petya 30

B.1.2 NotPetya 30

B.2 Encrypt the Data 30

B.2.1 Petya 30

B.2.2 NotPetya 31

B.3 Shutdown the OS 31

B.3.1 Petya 31

B.3.2 NotPetya 32

C REUSED CODE IN WANNACRY AND NOTPETYA 34

C.1 Vulnerability 35

C.1.1 WannaCry 35

C.1.2 NotPetya 35

LIST OF TABLES

Table Page

3.1 Reused Code in Mirai Variant 14

3.2 Functionality of Files 16

3.3 Similar Behaviors Between Petya and NotPetya 18

3.4 Vulnerability of WannaCry and NotPetya 18

LIST OF FIGURES

Figure Page

3.1 A Loader Structure 9

3.2 Bots and CNC Server 11

Chapter 1

INTRODUCTION

Code reusing saves resources and improves efficiency in software development. A reused code helps reduce development time, cost, and labor because programmers do not need to start from scratch nor to test the reused part. Consequently, programmers can spend more time and labor on their core tasks. Furthermore, the reused code is already tested and proved efficient. In this light, programmers can expect the same or higher level of performance by reusing existing codes for time saving and efficiency gain.

As much as code reuse is a common practice among programmers, it is a viable strategy for malicious actors, such as malware developers. Despite the apparent merit of code reusing, it comes with inevitable risks. For example, a reused code may entail bugs and backdoor issues and can be detected by a signature-based detection method with ease. If the benefit of code reusing outweighs its risk, reused code components would be observed in malware. There are many reported cases of code reusing in various malicious programs [1][2][3][4][5].

In light of rampant code reusing practices, it is imperative to understand and analyze reused codes in malware detection. First, the identification of reused codes can help malware scientists understand new malware that contains the same reused code. By investigating the reused code embedded in the new malware, malware scientists can ever infer the functionality of the new malware. Second, a reused code chunk can be used for detecting malicious programs. When a malicious program reuses the same vulnerability or the same strategy, it contains an exclusive string, such as error code, text, and methods. If the exclusive string is contained in the signature

of the malicious program, such malicious programs can be detected by signature-based detection tools. Lastly, the reused code can help predict the evolution of a malicious program. Reused code chunks in a malicious program. Provides valuable information for malware classification by projecting the evolution of the malicious program over time and geography.

A key challenge in analyzing reused codes in the malware context is obtaining malware source codes. Naturally, malware authors do not reveal or share their source codes. Thus, malicious programs are often analyzed with binary code files. To analyze binary codes, reverse-engineering is needed as a malware binary code is an already compiled code. There are several reverse-engineering tools, such as IDA-pro and Ghidra. However, it is practically impossible to obtain intact source code from reverse-engineering tools. For this reason, it is difficult to analyze a copied code in different malware with binary code. Therefore, a reused code in malware binary code can be detected not only clone code but also reused knowledge and strategies. These reused knowledge and strategies can be recognized via certain strings, flows of API calls and behaviors.

Even if malware uses the similar knowledge and strategy from another malware, the reused idea is hard to be detected for the following reasons. First, malware code is often modified, making it difficult to detect reused codes and threads in malware. Second, malware source is constantly improved for better functionality. Third, the expression is different because they do not know the expression of previous samples.

This thesis examines reused codes and investigates a modification of reused code, makes important practical and scholarly contributes. If the reused codes are possible to be classified by common characteristics over the modifications and different expressions, this information helps to detect classify the malware family. Moreover, it will

advance our knowledge as to how to recognize and to understand the new malware and to predict a future attack.

The rest of this thesis is organized as follows: Chapter II presents background information on reused codes and malware samples, followed by a practical example of reused code in malware source code and malware binary code in Chapter III. Chapter IV discusses limitations and future work. Chapter V concludes the thesis.

Chapter 2

BACKGROUND

This thesis analyzes reused cases with malware source code and malware binary code. For a reused code in malware source code, Mirai botnet variants source code is used. For reused code cases in malware binary code, Petya, WannaCry, and NotPetya binary code are considered. This chapter defines the reused code and discusses why Mirai botnet, Petya, WannaCry, and Notpetya binary code are chosen for this study.

2.1 Definition of Reused Code

A reused code is a code chunk that is reused multiple times for different malware or different purposes. Knight and Dunn mentioned [6] "Although software code is notable explicit knowledge that is both readable by humans and enables a computer to perform specific functions, knowledge reused may cover more than code reuse". Accordingly, a reused code concept can include reused knowledge or reused idea. In this thesis, a reused code is defined as a broad concept of reused code, which includes not only reused actual code but also reused ideas and knowledge such as tactics and vulnerabilities.

2.2 Reused Code in Malware Source Code

This thesis investigate Mirai botnet for reused source code. Mirai botnet is a self-propagating botnet virus and performs a large scale network attack by using controlled bots. The primary target is the IoT(internet-of-things). The Mirai botnet is an excellent platform to collect reused code chunks, since the first appearance of the Mirai botnet in August 2016, the author has released its source code to the public

to avoid surveilling eyes. After the release, many variants of the Mirai botnet have been created. Therefore, Mirai is a malware which I obtain source code and recognize the source code change via diverse variants with ease.

2.3 Reused Code in Malware Binary Code

Petya, WannaCry and NotPetya are frequently considered when reused code is discussed. Due to their discovery data and the similar behaviors, reused codes therein are studied. They are ransomware and emerged in March 2016, May 2017, and June 2017, respectively. Their functionality can be divided by four-parts: propagation, encryption, decryption, and payload. Due to their functional similarities and the similar time line of discovery, it is assumed that they contain reused code parts with each other. For example, Petya and NotPetya show very similar behaviors in terms of encryption and a ransom message. WannaCry and NotPetya attack the same vulnerability, which is Eternal Blue.

2.4 Motivations for Code Reusing in Malware

The benefits of code reusing in malware are in multi-folds. First, programmers can reduce their resources such as time, cost, and labors because there is no need to develop new functions from scratch and to test them. From such benefits, malware developers can reallocate their time and resources to more important tasks. NotPetya is a good case in point. From code reusing, NotPetya mimics Petya's behavior. Thus, NotPetya focused more on protecting its own code. NotPetya has a function that is detecting a sandbox and debugging an environment. When it detects a sandbox and a debugging environment, it does not show its normal behavior and terminates the program. This function makes it hard for detecting systems to understand and to see inside of the NotPetya program as well as to follow its' execution with a debugger.

Second, some tasks are to be accomplished in a specific manner. In this case, the code reusing is inevitable for any authors who wish to implement the function therein. Even though the source code is unavailable, the final code will be similar because there is only one way to do the task. WannaCry and Notpetya attack the same vulnerability which is EternalBlue. In doing so, it creats specific error code, which means “out of resource.” Therefore, the same error code can be detected in both type of malware. The last common benefit is efficiency gain. Even if there are multiple ways to carry out a task, malware authors prefer to use only one method for high efficiency. For instance, Mirai botnet uses the brute force attack to enslave vulnerable devices. Interestingly, Mirai botnet uses a pre-saved ID and password table which contains frequently used ID and password set information. The brute force attack with the table is efficient than the one without it. Even though a normal brute force attack is possible, it may take longer time to achieve the task. Therefore, many Mirai botnet variants use the same tactic in targeting select vulnerable devices.

2.5 Related Work

Several studies have investigated reused codes in software and documented their merites for programmers [7][8][9][10]. In the case of detecting reused code in benign software, the literature focuses on source codes. Due to the inaccessibility of malware source code, however the method to detect reused code in benign software cannot be readily applied for malware. For the malware, binary code is more available and accessible than the actual source code. Thus binary code analysis is considered in malware studies. Many scholars have reported effectively classified malware based on the common part in their binary code.

A stream of studies conducting malware classification[11][12][13][14][15][16]. However, many of them use machine learning to analyze the malware binary code. So,

the exact functionalities of reused code in malware are difficult to understand and often not interpretable. Moreover, if the reused code has few modifications, it cannot be detected as a cloned chunk. Therefore, even if malware analysis based on machine learning supports the speed and the scalability, the in-depth study of reused code is still necessary . This area is a burgeoning field. Only a few scholars have researched the reused code in malware. For example, Pfeffer et al. [17] introduced a novel system, MAAGI, based on the malware genetic information. He used diverse kinds of analyse such as header analysis, dynamic analysis, and trace analysis with API calls. Gregio et al. [18] detects code reuse based on tracking memory writes. Farhadi et al. [19] suggests BinClone, which is a new detecting code clone in malware, based on assembly code and clone database. By using the aformentioned systems, reused code detection is possible in malware to same extent. However, these methods cannot handle reused code that have little modification and use different expressions. For example, Manuel [1] reported a reused code in Mirai and 'Hide and Seek(hereinafter HNS)' malware, which changes reused code. According to the report, HNS malware reused part of the Mirai code. However, HNS changed "for loop" to "while loop," so it was not detected. Since detecting reused code in malware is relatively new, this study makes unique contribution to the malware classification field.

Chapter 3

EXPERIMENTS AND FINDINGS

Two different experiments were conducted about reused code cases for malware: one for reused code in malware source code and the other for reused code in malware binary code. This thesis investigates reused source code in the Mirai botnet variant. After the disclosure of the source code, the Mirai botnet has been changed by diverse malware authors. Thus, the Mirai botnet is a good example to recognize code reusing and modification. For reused code cases in malware binary code, Petya, WannaCry, and NotPetya binary codes are considered. Their alike behaviors and the same vulnerability attack demonstrate the code reusing practices among them.

3.1 Functionality and Vulnerability

3.1.1 Mirai Botnet

Mirai botnet is composed of two parts: loader and bot. The loader is designed to make a connection with vulnerable devices and to load binary. The bot is assumed as a worker, searching for vulnerable devices and taking an attack to a victim server when it gets an attack command from the CNC server(hereinafter the CNC server) [20].

When Mirai botnet is installed, it starts from a loader(see Figure 3.1). When the loader server is initialized, the loader creates a designated number of threads. These threads work as a worker in Mirai. The workers start to search for vulnerable devices with brute-force attacks. A brute-force attack is a method by which an attacker explores all possible combinations of a password until finding the correct one. The

Mirai botnet uses a brute-force attack to find vulnerable devices, but modifies the original brute-force attack to increase efficiency. It also uses a brute-force attack based on statistical information to recruit vulnerable devices. A Mirai server has a pre-saved 'ID-Password' set table in the program and it executes brute-force attack accordingly. A Mirai author exploits on two regularities. First, many people do not change the default setting of a router. Second, there are some popular 'ID-Password' sets. In this light, Mirai authors target the popular 'ID-Password' sets. Because devices are more likely to have the pre-saved 'ID-Password' sets as their 'ID-Password', this brute-force attack shows high efficiency in detecting vulnerable devices. Once a worker recognizes vulnerable devices, it saves the information of the vulnerable devices, such as 'ip: port' and 'id: password'. Each bot sends the vulnerable device information to the loader to install a binary program and it shares the information with the CNC server to enslave the device as a bot.

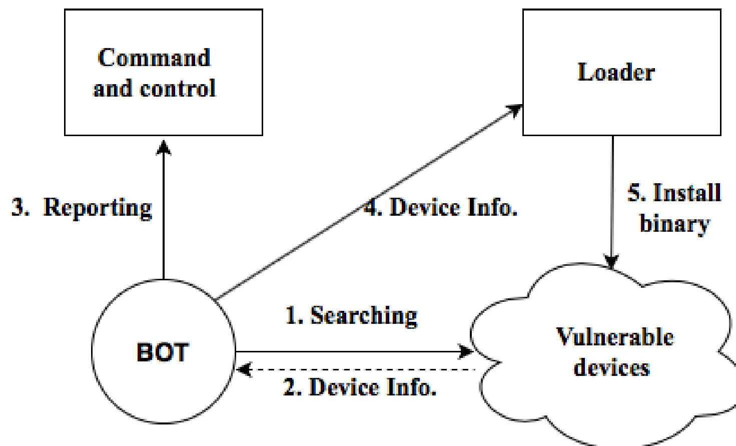

Figure 3.1: A Loader Structure

After installing binary, a bot initializes a attacker, a killer, and a scanner(see Figure 3.2). The killer inspects any pre-installed malicious programs in the device. The Mirai botnet can terminate the pre-installed malicious program for better performance. The scanner creates a random IP address to find the next vulnerable victim.

Scanning the next victim with an IP address is very similar to a worm process. However, the Mirai botnet uses a pre-defined IP address range rather than creates a random IP for high performance. After creating the IP, it attempts to connect the device with pre-saved 'ID-Password' sets. When it succeeds, it stores and sends the information of the device. This information is sent to the loader and the CNC server. The attacker participates in an actual attack to the victim server to some extent. The used attack to the victim is a Denial-of-service (hereinafter DDoS) attack. When the CNC server has an attack order with victim server IP, the CNC server creates an attack command with the victim IP and send it to all bots involved. Then the bots start to attack the target IP server in unison. Due to the superfluous requests which are created in a short burst of time period, the victim server becomes out of service. To sum up, the final goal of the Mirai botnet is to execute DDoS attacks to a target server and to cause the server down. For the DDoS attack, Mirai needs many bots that can participate in attacks. Thus, bots search vulnerable devices who can be a candidate for the next bot. The vulnerability of the Mirai botnet is devices that use common 'ID-Password' sets. Since the Mirai botnet takes a brute-force attack based on the frequently used 'ID-Password' sets. Therefore, devices with default 'ID-Password' sets, factory 'ID-Password' setting, and popular 'ID-Password' sets are most prone to attacks.

3.1.2 WannaCry and NotPetya

WannaCry and NotPetya are ransomware. Ransomware is a type of malicious program that aims to block access to a victim's system or files as a hostage, and asks money for unlocking the hacked data and system. When a victim pays the required money, the hacker gives the handle of their data back to the victim. Otherwise, ransomware destroys the data. A notable difference between WannaCry and

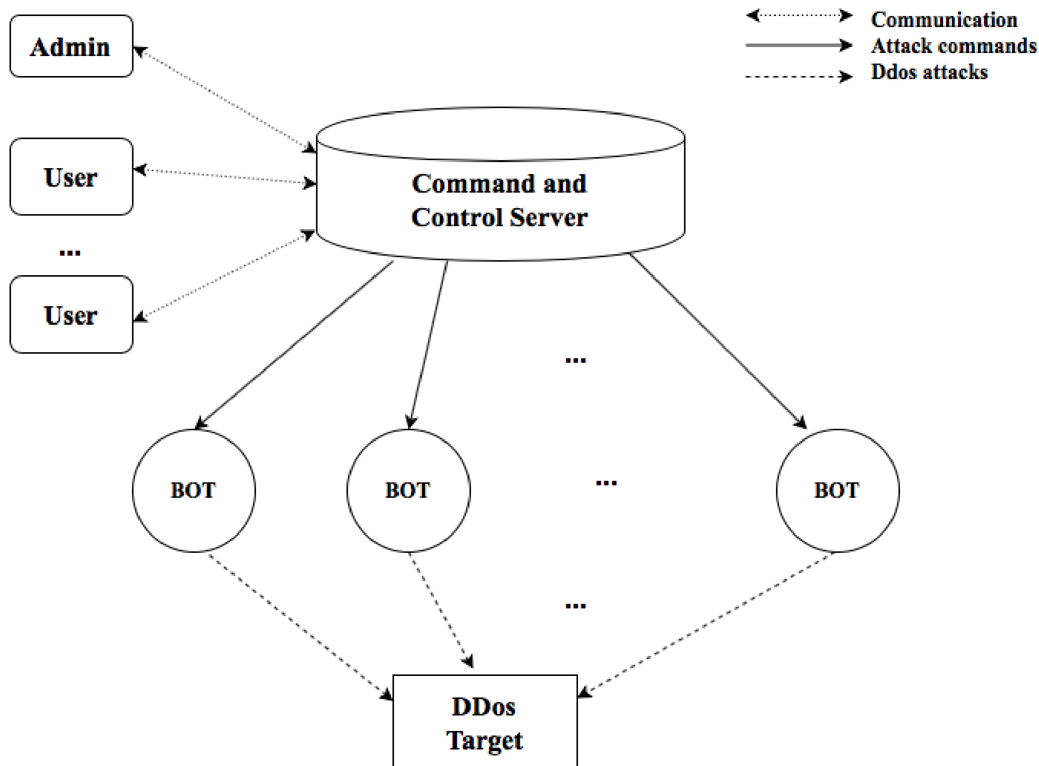

Figure 3.2: Bots and CNC Server

NotPetya is that WannaCry locks all the files in the system with special passwords while NotPetya blocks the access to the OS system to modify the master bootstrap. Two ransomware programs show little difference in the process, in that they attack the same vulnerability, Eternal Blue, to search new victims in the propagation stage. Eternal Blue is a vulnerability using the error code of the Server Message Block (hereinafter SMB) protocol. It exploits the fact that SMB server1 shows a malfunction with the special crafted packet from remote attackers in a certain version of Microsoft Windows, allowing attackers to execute random code on the target computer. When remote attackers send an SMB packet creating overflow, the SMB server sends an error message, STATUS_INSUFF_SERVER_RESOURCES(0xc0000205). When the error code is created, the attacker can execute an arbitrary code on the victim com-

puter. Following this process, WannaCry and NotPetya can infect many computer systems automatically.

3.2 Mirai Botnet – Reused Code in Malware Source Codes

To find clone code chunks among Mirai botnet variants, this study employs 'Deckard', a source code clone detector. One can access it at 'skyhover/Deckard (`https://github.com/skyhover/Deckard`)' [21]. Mirai botnet source code is downloaded from 'threadland/TL-BOTS' in github (`https://github.com/threatland/TL-BOTS`). There are 62 types of Mirai botnet variants and 118 versions of source codes exist. 'Deckard' executes 118 Mirai botnet variants source codes. From the experiment, 'Deckard' detects 28,721 duplicated code chunks from 692 C program source codes. These cloned source code chunks are then classified into 1,424 different reused code chunks. From the result of the 'Deckard,' valuable information of reused chunk is extracted such as file location, line information, and reused number. The extracted data set is represented by graphs using python. According to the graphs, the location of code chunks is presented and the relation among reused code chunks is recognized. To learn about the functionality of the reused code chuck, Akiru.Mirai is chosen.

Since there are 118 Mirai variants, the code clusters which are reused more than 100 times are considered and the result is reported in Table 3.1. From the table, some notable reused patters are observed. First, some files are reused without any changes in all 118 Mirai variants. Telnet_info.c file is responsible to obtain telnet information 'IP:port' and 'ID:password' and transmits information to the program with telnet_info structure. This file is composed with 64 lines and two functions: telnet_info_new() and telnet_info_parse(). In this file, line 9 to 25 (telnet_info_new()) and line 26 to 65 (telnet_info_parse()) are detected 121 times as cloned code chunks. That is, the Telnet_info.c file is reused in all Mirai variants without any modification. Second,

some clone code chunks are observed in different functions. Cluster 609 is found in both scanner.c and connection.c files. The reused chunk is Consume_iacs() function in scanner.c file and Connection_consume_iacs() function in connection.c file. The scanner.c file is used to connect a bot and the CNC server. In the Mirai architecture, a bot needs to set a connection with the CNC server to communicate. The connection.c file serves to connect the loader and bots. The loader tries to connect with a bots to get vulnerable device information. This reused code chunk is used for different functions. However, the Mirai botnet uses the same method to connect the CNC server and the loader, and this code chunk is reused. Last, some code chunks are reused many times for the same function. For example, cluster 1418 is recognized twice in lines 547 to 560 and lines 519 to 534, which are indicate functions connection_upload_tftp() and connection_upload_wget(), respectively. They upload binary files to vulnerable devices. One differences is, however, the method used to transport a binary file: ftp vs. wget. Therefore, these functions share the similar structure and have little difference (see Appendix A.1).

In a similar reason, attack_method.c file has many reused code chunks. Attack_method.c file is a core function of attacking of an assigned victim server. It involves several attacking functions, such as greip, greeth, std, tcpsyn, and tcpack. Because these functions perform a same goal of attacking the victim server, many parts of their codes are similar. Therefore, attack_method.c contains many reused code chunks. Another example is Amakano.Mirai. Amakano.Mirai uses a special scanner depending on the router. So, scanner files for different routers have clone code chunks. Lastly, diverse code chunks are founded in various versions of Mirai variants. Table 3.1 presents that 41 code chunks are observed more than 100 times in different Mirai botnet variants. Since we execute 118 Mirai variants, the code chunk which is founded more than 100 times means that it was founded from more than

Table 3.1: Reused Code in Mirai Variant

	Cluster	Count	Location	Line	Functions
1	609	240	/Akiru/bot/scanner.c	863:916(53)	Consume_iacs()
			/Akiru/loader/src/connection.c	94:147(53)	Connection_consume_iacs()
2	990	232	/Akiru/bot/scanner.c	169:196(27)	In a scanning while loop scanner process initializing.
3	1418	168	/Akiru/loader/src/connection.c	547:560(13)	Connection_upload_tftp()
			/Akiru/loader/src/connection.c	519:534(15)	Connection_upload_wget()
4	789	155	/Akiru/bot/attack_method.c	478:510(32)	In attack_method_tcpack() while loop to send to sockets
			/Akiru/bot/attack_method.c	380:411(31)	In attack_method_tcpsyn() while loop send to sockets
5	950	135	/Amakano/bot/linksys_scanner.c	418:441(23)	Linksysscanner_get_random_ip()
			/Amakano/bot/scanner.c	678:701(23)	Get_random_ip()
			/Amakano/bot/thinkphp.c	415:439(24)	Thinkphp_get_random_ip()
6	1067	125	/Akiru/bot/attack_method.c	266:293(27)	In attack_method_std() for loop
			/Akiru/bot/attack_method.c	1079:1106(27)	In attack_method_udpplain() for loop
7	1086	121	/Akiru/loader/src/binary.c	63:90(27)	Load function()
8	1383	121	/Akiru/loader/src/connection.c	465:481(16)	Connection_consume_upload_methods()
9	1384	121	/Akiru/loader/src/telnet_info.c	9:25(16)	telnet_info_new()
10	806	121	/Akiru/loader/src/telnet_info.c	26:65(26)	telnet_info_parse
11	1019	121	/Akiru/dlr/main.c	218:246(28)	Xconnect()
12	585	121	/Akiru/bot/attack_method.c	249:304(55)	Attack_method_std()
			/Akiru/bot/attack_method.c	1062:1117(55)	Attack_method_udpplain()
13	1023	119	/Akiru/bot/scanner.c	920:939(19)	Consume_any_prompt()
14	1455	119	/Akiru/bot/scanner.c	1031:1044(13)	Add_auth_entry()
15	611	118	/Amakano/bot/gpon443.c	167:220(53)	In Gpon443_scanner()
			/Amakano/linksys_scanner.c	165:217(53)	In Scanner_init() insie of scanning process, waiting SYN+ACKs
			/Amakano/bot/thinkphp.c	163:216(53)	In thinkphp_scanner()
16	1164	118	Akiru/bot/scanner.c	1006:1030(24)	Consume_resp_prompt()
17	1006	117	Akiru/bot/checksum.c	24:54(30)	Checksum_tcpudp()
18	1099	117	Akiru/bot/killer.c	329:350(21)	In killer_kill_by_port() killing previous process
19	583	116	Akiru/bot/util.c	99:154(55)	Util_atioi()
20	1155	116	Akiru/bot/util.c	232:256(24)	Util_stristr()

	Cluster	Count	Location	Line	Functions
21	936	114	Akiru/loader/src/main.c	47:94(47)	Watchdog_maintain()
22	1096	114	Akiru/bot/scanner.c	321:347(26)	In Scanner_Init() trying to connect.
23	880	113	Amakano/bot/gpon443.c Amakano/bot/linksys_scanner.c Amakano/bot/thinkphp.c	225:260(35) 222:251(29) 221:250(29)	In gpon443_scanner() scanning Ip part In linksysscanner_scanner_init() In thinkphp_scanner()
24	279	113	/Akiru/bot/attack_method.c /Akiru/bot/attack_method.c	684:778(94) 539:633(94)	In attack_method_tcpxmas while loop to wait receiving socket Attack_method_tcpstomp while loop to wait receiving socket
25	861	113	Amakano/bot/gpon443.c Amakano/bot/linksys_scanner.c Amakano/bot/thinkphp.c	454:490(36) 379:412(33) 376:409(33)	Gpon443_setup_connection() Linksysscanner_setup_connection() Thinkphp_setup_connection()
26	1069	108	/Akiru/bot/attack_method.c /Akiru/bot/attack_method.c	778:800(22) 633:655(22)	In attack_method_tcpxmas() last for loop In attack_method_tcpstomp() last for loop
27	1159	110	/Akiru/bot/resolv.c	21:45(24)	Resolv_domain_to_hostname()
28	471	108	/Akiru/bot/resolv.c	152:211(59)	In Resolve_lookup() while loop
29	1283	110	/Akiru/bot/resolv.c	46:66(20)	Resolv_skip_name()
30	526	108	/Akiru/bot/attack_method.c	333:380(47)	Attack_method_tcpsyn()
31	860	108	/Akiru/bot/attack_method.c	383:410(27)	Attack_method_tcpsyn()
32	380	106	/Akiru/bot/attack.c	41:110(69)	In attack_parse()
33	574	114	/Akiru/bot/scanner.c	198:253(55)	In scanner_init while loop to recv socket
34	1072	105	/Akiru/bot/scanner.c	596:623(27)	Setup_connection()
35	624	105	/Akiru/bot/scanner.c	256:307(51)	In scanner_init to try connect
36	291	103	/Akiru/bot/resolv.c	94:214(120)	In resolv_lookup() while loop 5 times try
37	227	103	/Akiru/bot/killer.c	317:351(34)	In killer_kill_by_port() explorer directory
38	896	102	/Akiru/bot/scanner.c	535:568(33)	In scanner_init() case waiting_token_resp and wrong auth or correct auth.
39	932	101	/Akiru/bot/scanner.c	976:1005(29)	Donsume_pass_prompt()
40	455	101	/Akiru/bot/attack_method.c	1117:1174(57)	Get_dns_resolver()
41	1188	100	/Akiru/bot/rand.c	34:57(23)	Rand_str()

Table 3.2: Functionality of Files

	File name	functionality
Bot	Main.c	Initializing watchdog, attack, kill, and scanner Connecting with the CNC server and monitoring a connection with CNC This file has the attack parse function
	Killer.c	Scanning the device. If there are pre-installed programs, kill them
	Scanner.c	Searching for vulnerable devices with pre-saved id and password sets
	Attacker.c	Parsing an attack command and starting an attack
	Attack_method.c	Diverse functions depend on attack methods
Loader	Main.c	Creating threads for bots. Waiting for vulnerable device information form a bot When the loader gets a report from a bot, it creates a connection with the vulnerable device then installs a binary
	Connection.c	Creating a connection with vulnerable devices
	Binary.c	Read the binary file and Load the binary file
	Telnet_info.c	Parsing the telnet information
	Util.c	Binding a socket address

84.74 percent of Mirai variants. It indicates that large parts of Mirai botnet codes are recycled across diverse Mirai variants.

In sum up, Miria botnet source codes are reused many different ways. They are reused not only in one program but also across different programs. The frequently reused program code chunk is likely to be reused in different programs. Also, these popular code chunks can be reused with little modification. For example, the 'while loop' can be used instead of the 'for loop', and the 'if - else' statement can be changed to 'switch' statement. These small modifications can cause a nontrivial impact and be an obstacle in detecting cloned codes. As remembers to categorize frequently reused code chunks and classify modified versions of reused code in one category, reused code chunk detection rate can increase significantly.

3.3 Petya, WannaCry, and NotPetya - Reused Code in Binary Code

A malware binary file is an executable malicious program. A malware binary file is created after compiling. Thus, a malware binary file cannot be analyzed itself. To analyze a malware binary file, reverse engineering is widely used. Reverse engineering of a malware binary involves a process to reveal deconstructed object's designs and architecture. However, during compilation, a program loses much of detailed information such as variable names and function names. Even if a binary code is reverse engineered, the result cannot be the same as the original source code. Therefore, the meaning of reused code in malware binary code only includes reused knowledge and attacking tactics. This thesis uses Ghidra for reverse engineering and compares the similarity among different malware programs after manual analysis. To understand the process of three different ransomware, OllyDbg is used to inspect the operation, and WireShark is used to examine the packet exchange. Moreover, Cuckoo sandbox is used to obtain the API calls and perform dynamic analysis.

From the manual analysis after reverse engineering, some code chunks were uncovered and thry exhibit the following regularities.

First, Petya and Netpetya shares similar behaviors. They both write on physical drive, encrypt with XOR, and shut down the program at the end of the execution. Even though they demonstrate many identical behaviors, the expressions are different. In the case of encrypting demonstrate data, Petya xors with '0x37' which is ASCII number '7' while NotPetya uses number '07.' Also, the shutdown method is different. Petya uses an API call, LookupPrivilegeValueA-shutdownPrivilege, but NotPetya sets a time to shut down when it creates a process. After certain amount of time past, the scheduler comment for shut down is issued. Then, the OS is shut down. In

Table 3.3: Similar Behaviors Between Petya and NotPetya

Function	Malware	Code
Writing on Physical Drive	Petya	Save the "PhysicalDrive" string in the param_1 array
	NotPetya	Save the "PhysicalDrive" string in the local_168 stack
Encrypt the data	Petya	XORing with '0x37' which is ASCII value of number '7'
	NotPetya	XORing with number '7'
Shutdown the OS	Petya	Using API 'LookupPrivillegeValueA – SeShutdownPrivilege.' If GetLastError API return a value '0', it raise 'ntdll.ZwRaiseHardError' and shut down.
	NotPetya	Schedules a rebute by issuing the command "shutdown.exe /r/f" at a set time using CreateProcessW API

Table 3.4: Vulnerability of WannaCry and NotPetya

Malware	Code	
	Similarity	**Difference**
WannaCry	Both using EternalBlue vulnerability. They expect returning '0xc0000205.'	Comparing the return value in each location Ex) It compares '05','02','00','04' respectably.
NotPetya		Comparing the return as a whole. Ex) It compares '-0x3ffffdfb.' This is 1's complement of '0xc0000205.'

summary, Petya and NotPetya show very similar behaviors but the actual codes are different.(Table 3.3 and Appendix A.2)

WannaCry and NotPetya use the same vulnerability, EternalBlue. Because both use the same vulnerability, the expected error code is the same (0xc0000205). However, the way to detect the error code differs. WannaCry compares each byte where the error code is saved. In contrast, NotPetya compares the error code(4bytes) directly (see Table 3.3 and Appendix A.3). Thus, even if different malware programs

use the same vulnerability, the expression can be different. These different expressions for the identical behavior pose a great challenge for malware classification. As a result, these same behavior is hard to recognized as identical. However, if these diverse expressions for the similar behaviors are recognized and are classified in the same category, it can reduce the false positive and true negative rates in malware classification.

Chapter 4

LIMITATIONS AND FUTURE WORK

This study investigated, four different malware programs from two distinct categories: malware with source code and malware with a binary file. In both cases, static analysies were mainly performed to examine reused codes. For the malware analysis with source code, it is possible to detect reused code chunks among the same version of a source file and across the different versions of a source file. As a result, scanner.c, connector.c, and attack_method files have many reused code chunks. Such frequently reused code chunks can be used to detect another kind of malware that has similar functionality. This result can offer practical benefits in detecting reused code in different malware because the frequently reused codes are more likely to be detected. If there were notable patters of code reusing across malware program, it can help predict a future attack. This is an interesting avenue for future research. Since obtaining malware source code is a key obstacle for malware analysis, detecting reused code in malware source code remains as a limitation.

In many cases, malware binary code is available, but it is impossible to get the original source code from binary code. The binary code is hard to read and difficult to understand because much essential information, such as function name and variable names, is removed. To this end, static analysis with binary code is conducted but is time-consuming, and difficult to understand the underlying operation. To overcome the obstacle, dynamic analysis can be a potential remedy. According to the dynamic analysis, extracting API calls by chronological order will help understand the functions and predict behaviors of malware. Moreover, it requires less time than manual analysis. The information from the dynamic analysis will be essential to recognize the

reused code in binary code. Hence, a consolidated approach of static and dynamic analysis is expected to be more valuable to detect reused codes and improve malware classification.

Chapter 5

CONCLUSION

This thesis documents and investigates reused codes in diverse malware programs. For the malware with source code, Mirai botnet variants are investigated. Petya, WannaCry, and Notpetya are used for malware with binary code.

Results from Mirai botnet variants reveal that reused code chunks are detected many times in connection.c, scanner.c, and attack_method.c files. These files are used to connect between two devices and to execute DDoS attacks. These frequently reused code chunks are more likely to be recycled in a variety of malware. Since the original code checks can be modified for various reasons, many reused code chunks are difficult to detect with machine learning methods because the machine learning methods cannot recognize a reused code chunk with little modification as an identical code chunk. Consequently, malware classification with machine learning methods may result in poor performance (i.e. a high false-positive rate).

Due to the difficulty to understand binary files and the required time to manually analyze them, this thesis focuses on four malware binary files. Results show that there exists a behavioral similarity between Petya and Notpetya. WannaCry and Notpetya use the same vulnerability. However, the codes are different because they use different expressions. It poses an unique challenge for researches and practitioners to detecting reused source codes. This thesis elucidates the differences between ransomware programs for reused behaviors. From the analysis, it demonstrated that, even if the expressions are different, modified reused codes can be detected as the same in nature for the purpose of malware classification.

In conclusion, this thesis is a case study of reused codes and diverse expressions of reused codes. This study provides practitioners with actionable insights as to how to identify the reused codes with different expressions and to classify into the same category. The methodological framework and empirical findings in this thesis can help increase the accuracy rate of malware classification.

REFERENCES

[1] Jasper Manuel. Searching for the reuse of mirai code: Hide ‘n seek bot, 2018. www.fortinet.com/blog/threat-research/searching-for-the-reuse -of-mirai-code–hide–n-seek-bot.

[2] Anonymous. Mmd-0056-2016-linux/mirai, how an old elf malcode is recycled, 2016. blog.malwaremustdie.org/2016/08/mmd-0056-2016-linuxmirai-just.html.

[3] How similar are wannacry and perya ransomware?, 2017. www.forbes.com/sites/quora/2017/07/05/how-similar-are-wannacry-and-petya-ransomware/7e1464fe46eb.

[4] Mrac Laliberte. Why hackers reuse malware, 2017. www.helpnetsecurity.com/2017/11/20/hackers-reuse-malware/.

[5] Xavier Mertens. Code data reuse in the malware ecosystem, 2019. isc.sans.edu/forums/diary/Code+Data+Reuse+in+the+Malware+Ecosystem /25598/.

[6] John C Knight and Michael F Dunn. Software quality through domain-; driven certification. *Annals of Software Engineering*, 5(1):293, 1998.

[7] Stefan Haefliger, Georg Von Krogh, and Sebastian Spaeth. Code reuse in open source software. *Management science*, 54(1):180–193, 2008.

[8] Audris Mockus. Large-scale code reuse in open source software. In *First International Workshop on Emerging Trends in FLOSS Research and Development (FLOSS'07: ICSE Workshops 2007)*, pages 7–7. IEEE, 2007.

[9] Tomoya Ishihara, Keisuke Hotta, Yoshiki Higo, and Shinji Kusumoto. Reusing reused code. In *2013 20th Working Conference on Reverse Engineering (WCRE)*, pages 457–461. IEEE, 2013.

[10] Saed Alrabaee, Paria Shirani, Lingyu Wang, and Mourad Debbabi. Sigma: A semantic integrated graph matching approach for identifying reused functions in binary code. *Digital Investigation*, 12:S61–S71, 2015.

[11] George E Dahl, Jack W Stokes, Li Deng, and Dong Yu. Large-scale malware classification using random projections and neural networks. In *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 3422–3426. IEEE, 2013.

[12] Mu Zhang, Yue Duan, Heng Yin, and Zhiruo Zhao. Semantics-aware android malware classification using weighted contextual api dependency graphs. In *Proceedings of the 2014 ACM SIGSAC conference on computer and communications security*, pages 1105–1116, 2014.

[13] Alexey Malyshev, Timur Biyachuev, and Dmitry Ilin. Systems and methods for malware classification, January 21 2014. US Patent 8,635,694.

[14] Younghee Park, Douglas Reeves, Vikram Mulukutla, and Balaji Sundaravel. Fast malware classification by automated behavioral graph matching. In *Proceedings of the Sixth Annual Workshop on Cyber Security and Information Intelligence Research*, pages 1–4, 2010.

[15] Joris Kinable and Orestis Kostakis. Malware classification based on call graph clustering. *Journal in computer virology*, 7(4):233–245, 2011.

[16] Lakshmanan Nataraj, Sreejith Karthikeyan, Gregoire Jacob, and BS Manjunath. Malware images: visualization and automatic classification. In *Proceedings of the 8th international symposium on visualization for cyber security*, pages 1–7, 2011.

[17] Avi Pfeffer, Catherine Call, John Chamberlain, Lee Kellogg, Jacob Ouellette, Terry Patten, Greg Zacharias, Arun Lakhotia, Suresh Golconda, John Bay, et al. Malware analysis and attribution using genetic information. In *2012 7th International Conference on Malicious and Unwanted Software*, pages 39–45. IEEE, 2012.

[18] André Ricardo Abed Grégio, Paulo Lício De Geus, Christopher Kruegel, and Giovanni Vigna. Tracking memory writes for malware classification and code reuse identification. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 134–143. Springer, 2012.

[19] Mohammad Reza Farhadi, Benjamin CM Fung, Philippe Charland, and Mourad Debbabi. Binclone: Detecting code clones in malware. In *2014 Eighth International Conference on Software Security and Reliability (SERE)*, pages 78–87. IEEE, 2014.

[20] Joel Margolis, Tae Tom Oh, Suyash Jadhav, Young Ho Kim, and Jeong Neyo Kim. An in-depth analysis of the mirai botnet. In *2017 International Conference on Software Security and Assurance (ICSSA)*, pages 6–12. IEEE, 2017.

[21] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *29th International Conference on Software Engineering (ICSE'07)*, pages 96–105. IEEE, 2007.

APPENDIX A

REUSED CODE IN AKIRU MIRAI CONNECTION.C FILE

A.1 Reused Code in Wget

```
int connection_upload_wget(struct connection *conn)
{
    int offset = util_memsearch(conn->rdbuf, conn->rdbuf_pos,
        TOKEN_RESPONSE, strlen(TOKEN_RESPONSE));

    if(offset == -1)
        return 0;

    if(util_memsearch(conn->rdbuf, offset, "Permission denied"
        , 17) != -1)
        return offset * -1;
    else if(util_memsearch(conn->rdbuf, offset, "nvalid option
        ", 13) != -1)
        return offset * -1;
    else if(util_memsearch(conn->rdbuf, offset, "llegal option
        ", 13) != -1)
        return offset * -1;
    else if(util_memsearch(conn->rdbuf, offset, "No space left
         on device", 23) != -1)
    {
        conn->clear_up = 1;
        return offset * -1;
    }

    return offset;
}
```

A.2 Reused Code in Tftp

```
int connection_upload_tftp(struct connection *conn)
{
    int offset = util_memsearch(conn->rdbuf, conn->rdbuf_pos,
        TOKEN_RESPONSE, strlen(TOKEN_RESPONSE));

    if(offset == -1)
        return 0;

    if(util_memsearch(conn->rdbuf, offset, "Permission denied"
        , 17) != -1)
        return offset * -1;
    else if(util_memsearch(conn->rdbuf, offset, "timeout", 7)
        != -1)
        return offset * -1;
```

```
    else if(util_memsearch(conn->rdbuf, offset, "nvalid option
        ", 13) != -1)
        return offset * -1;
     else if(util_memsearch(conn->rdbuf, offset, "llegal
         option", 13) != -1)
        return offset * -1;
    else if(util_memsearch(conn->rdbuf, offset, "No space left
         on device", 23) != -1)
    {
        conn->clear_up = 1;
        return offset * -1;
    }

    return offset;
}
```

APPENDIX B

REUSED CODE IN PETYA AND NOTPETYA

B.1 Writing on Physical Drive

B.1.1 Petya

```
if (hDevice != (HANDLE)0xffffffff) {
        DeviceIoControl(hDevice,0x560000,(LPVOID)0x0,
                         0,local_24,0x20,&local_30,(LPOVERLAPPED)0
                             x0);
        CloseHandle(hDevice);
        *param_1 = 0x5c2e5c5c;
        param_1[1] = 0x73796850;
        param_1[2] = 0x6c616369;
        param_1[3] = 0x76697244;
        *(char *)(param_1 + 4) = 'e';
        *(char *)((int)param_1 + 0x11) = local_1c + '0';
        *(undefined *)((int)param_1 + 0x12) = 0;
        return 1;
    }

     1000a26c 5c 5c 2e        ds        "\\\\.\\PhysicalDrive"
              5c 50 68
              79 73 69
```

B.1.2 NotPetya

```
     memset(param_1,0,0x104);
     local_168 = 0x5c2e5c5c;
     uStack356 = 0x73796850;
     uStack352 = 0x6c616369;
     uStack348 = 0x76697244;
     cStack344 = 'e';
     local_157 = 0;

     1000ff38 5c 5c 2e        ds        "\\\\.\\PhysicalDrive"
              5c 50 68
              79 73 69
```

B.2 Encrypt the Data

B.2.1 Petya

```
     do {
         local_c40 = (undefined *)((int)uVar8 >> 0x1f);
```

```
        FUN_100088ee((LPCSTR)local_c3c,local_a08,uVar8,(int)
            local_c40);
        uVar3 = 0;
        do {
        local_a08[uVar3] = local_a08[uVar3] ^ 0x37;
        uVar3 = uVar3 + 1;
        } while (uVar3 < 0x200);
        uVar3 = FUN_10008963((LPCSTR)local_c3c,local_a08,uVar8
            ,(int)local_c40);
        if (uVar3 == 0) {
        return 0;
        }
        uVar8 = uVar8 + 1;
    } while ((int)uVar8 < 34);
```

B.2.2 NotPetya

```
    do {
         *(byte *)((int)local_79c + uVar8) = *(byte *)((int)
            local_79c + uVar8) ^ 7;
         uVar8 = uVar8 + 1;
    } while (uVar8 < 0x200);
    memset(&local_99c,7,0x200);
```

B.3 Shutdown the OS

B.3.1 Petya

```
    if (BVar1 != 0) {
        LookupPrivilegeValueA((LPCSTR)0x0,"SeShutdownPrivilege
            ",(PLUID)local_1c.Privileges);
        local_1c.PrivilegeCount = 1;
        local_1c.Privileges[0].Attributes = 2;
        AdjustTokenPrivileges
              (local_8,0,(PTOKEN_PRIVILEGES)&local_1c,0,(
                 PTOKEN_PRIVILEGES)0x0,(PDWORD)0x0);
        DesiredAccess = GetLastError();
        if (DesiredAccess == 0) {
            lpProcName = "NtRaiseHardError";
            hModule = GetModuleHandleA("NTDLL.DLL");
            pFVar2 = GetProcAddress(hModule,lpProcName);
                    /* ntdll.ZwRaiseHardError */
            (*pFVar2)(0xc0000350,0,0,0,6,local_c);
            return 1;
```

```
        }
    }
```

B.3.2 NotPetya

```
    FUN_CreateProcessorFor_cmd100083bd(3);
    if ((DAT_MFlag_AdjustedPrivileges1001f144 & 1) != 0) {
        hModule = GetModuleHandleA("ntdll.dll");
        if ((hModule != (HMODULE)0x0) &&
            (pFVar4 = GetProcAddress(hModule,"NtRaiseHardError"
                ), pFVar4 != (FARPROC)0x0)) {
                        (*pFVar4)(0xc0000350,0,0,0,6,&param_3)
                            ;
            }
                BVar2 = InitiateSystemShutdownExW((LPWSTR)0x0
                    ,(LPWSTR)0x0,0,1,1,0x80000000);
                        if (BVar2 == 0) {
                                ExitWindowsEx(6,0);
                        }
    }

  /* Function for cmd.exe*/
    uVar7 = 0;
    GetLocalTime((LPSYSTEMTIME)&local_14);
    uVar1 = FUN_10006973();
    if (uVar1 < 10) {
        uVar1 = 10;
    }
    uVar6 = ((uint)local_14.wHour + (uVar1 + 3) / 60) % 24;
    iVar8 = (uint)local_14.wMinute + (uVar1 + 3) % 60;
    UVar2 = GetSystemDirectoryW(local_62c,0x30c);
    if (UVar2 != 0) {
        BVar3 = PathAppendW(local_62c,L"shutdown.exe /r /f");
      if (BVar3 != 0) {
            iVar4 = FUN_10008494();
             if (iVar4 == 0) {
               wsprintfW(local_e2c,L"at %02d:%02d %ws",uVar6,
                   iVar8,local_62c);
             }
             else {
                 pwVar5 = L"/RU \"SYSTEM\" ";
                 if (((byte)DAT_MFlag_AdjustedPrivileges1001f144
                     & 4) == 0) {
                         pwVar5 = L"";
                 }
```

```
            wsprintfW(local_e2c,L"schtasks %ws/Create /SC
               once /TN \"\" /TR \"%ws\"     /ST %02d:%02d",
             pwVar5,local_62c,uVar6,iVar8);
          }
        local_62e = 0;
        uVar7 = FUN_CreateProcessorFor_cmd100083bd(0);
      }
    }
    return uVar7;
```

APPENDIX C

REUSED CODE IN WANNACRY AND NOTPETYA

C.1 Vulnerability

C.1.1 WannaCry

```
len = send(s,&DAT_0042e4f4,0x4e,0);
if (len != -1) {
    /* Check return value of previous recv function. When
        0xc0000205 is returned,
     the propagation is started  */
     len = recv(s,&local_400,0x400,0);
     if ((((len != -1) && (cStack1015 == 5)) && (
        cStack1014 == 2)) &&((cStack1013 == 0 && (
        cStack1012 == -0x40)))) {
                    closesocket(s);
                    return 1;
    }
}
```

C.1.2 NotPetya

```
iVar6 = FUN_EternerBlueTriger100035fa
                    (0x2801,local_2c,local_34,0xfeff,
                       local_3c,local_38,2,0xff,0);
    if (iVar6 != -0x3ffffdfb) goto LAB_10006618;
```