STREAMLINE MALWARE ANALYSIS USING GHIDRA

A PROJECT REPORT

Submitted by

ANAND PANDEY	(22BCY10122)
AYUSH MONGA	(22BCY10267)
PRANJAL KHARE	(22BCY10020)
SYLAN PADMAKUMAR	(22BCY10176)

in partial fulfillment for the award of the degree

of

BACHELOR OF TECHNOLOGY

in

COMPUTER SCIENCE AND ENGINEERING

(Cyber Security and Digital Forensics)

SCHOOL OF COMPUTING SCIENCE AND ENGINEERING

VIT BHOPAL UNIVERSITY

KOTHRIKALAN, SEHORE
MADHYA PRADESH – 466114

May 2024

BONAFIDE CERTIFICATE

Certified that this project report titled **"STREAMLINED MALWARE ANALYSIS USING GHIDRA FOR VIT BHOPAL UNIVERSITY"** is the bonafide work of "**Pranjal Khare (22BCY10020), Aryan Singh (22BCY10034), Jatin Sharma (22BCY10050), , Sylan Padmakumar (22BCY10176), Kiran Kumar Ch (22BCY10109)"** who carried out the project work under my supervision. Certified further that to the best of my knowledge the work reported at this time does not form part of any other project/research work based on which a degree or award was conferred on an earlier occasion on this or any other candidate.

PROGRAM CHAIR

Dr. D.Saravanan

Assistant Professor Sr.,

Division of Cyber Security and Digital Forensics

School of Computing Science and Engineering
VIT BHOPAL UNIVERSITY

PROJECT SUPERVISOR

Dr. Hariharasitaraman S,

Assistant Professor-Senior,

Division of Cyber Security and Digital Forensics

School of Computing Science and Engineering
VIT BHOPAL UNIVERSITY

The Project Exhibition II Examination is held on _____.

ACKNOWLEDGEMENT

First and foremost, I would like to thank the Lord Almighty for His presence and immense blessings throughout the project work.

I wish to express my heartfelt gratitude to **Dr D. Saravanan**, Program Chair, Cyber Security and Digital Forensics for much of his valuable support and encouragement in carrying out this work.

I would like to thank our internal guide **Dr. Hariharasitaraman S**, for continually guiding and actively participating in my project, giving valuable suggestions to complete the project work.

I would like to thank all the technical and teaching staff of the School of Computer Science and Engineering, who extended directly or indirectly all support.

Last, but not least, I am deeply indebted to my parents who have been the greatest support while I worked day and night for the project to make it a success.

LIST OF ABBREVIATIONS

- URL - Uniform Resource Locator
- DLL - Dynamic Link Library
- DNS - Domain Name System
- RAT - REMOTE ACCESS TROJAN
- PE – Portable Executable
- ELF - Executable and Linkable Format
- ROP: - Return-Oriented Programming
- APT - Advanced Persistent Threat
- IOC - Indicators of Compromise
- Exe - Executable
- NSA - National Security Agency
- CVE - Common Vulnerabilities and Exposures
- SRE - Site reliability engineering

LIST OF FIGURES AND GRAPHS

ABSTRACT

In the field of cybersecurity, the process of reverse engineering is essential because it enables analysts to break down software binaries to find vulnerabilities and recognize malicious activity. In this work, we utilize the capabilities of Ghidra, an effective tool for reverse engineering, to create scripts that automatically identify harmful and vulnerable functionalities in executable binaries. Symbol tables and memory maps are analysed as part of our research to identify functions that have traits suggestive of malware vulnerabilities. We show how our method effectively and efficiently identifies possible security threats through methodical analysis and script development. The outcomes of executing the scripts on test programs demonstrate how well they can identify malicious code and weak points in programs, which helps with threat mitigation techniques. The development of automated reverse engineering methods is aided by this research.

Keywords: Symbol Table; Vulnerable Functions; String Analysis; Memory Mapping.

[PURPOSE-METHODOLOGY-FINDINGS]

PURPOSE:

Our main objective was to address the pressing need for effective and efficient methods for discovering and assessing vulnerabilities in malware binaries. The increasing complexity of cyber threats has created a pressing need for professionals with the certain skills so they can systematically and rapidly find possible malicious software and offer useful information for mitigation. Our objective was to improve safety measures and expedite malicious software discovery by creating customized scripts that made use of the Ghidra framework. Our main objective was to improve software binaries' security posture and reduce the risks related to malevolent actors seeking to take advantage of such vulnerabilities for illegitimate gain.

METHODOLOGY:

Our methodology was based on the creation and execution of scripts that were automated for the testing of malware binaries. We began by using the Ghidra framework to disassemble and analyze the binaries, extracting important information like mnemonic instructions and instruction counts within routines. We then created specific programs to examine these numbers in a systematic manner, discovering trends that could indicate potential security flaws. In addition, we used symbol table analysis and memory mapping techniques to acquire an improved awareness of the context and usage of vulnerable functions within the binary. Our methodology used a combination of static and dynamic analysis techniques to completely analyze the security posture of software binaries and deliver practical mitigation recommendations.

FINDINGS:

The findings from this project demonstrate that the blockchain-based system significantly improves the transparency and traceability of the grocery shop supply chain. It reduces the time required to track a product's journey, enhances the security of transactions, and minimizes inefficiencies in the supply chain. However, challenges such as scalability and integration with existing systems were also identified, indicating areas for future research and improvement. There is work to be done in coming up with strategies to retaliate Web3 attacks that threaten the supply chain.

TABLE OF CONTENTS

CHAPTER 1

PROJECT DESCRIPTION AND OUTLINE

1.1 INTRODUCTION

The initial section establishes the context by clarifying the crucial function of malware evaluation in modern cybersecurity. It highlights the changing character of cyberthreats and emphasizes the requirement for effective analysis methods to counteract advanced malware. Ghidra is also introduced as a well-known tool for its efficiency in malware analysis and reverse engineering.

The National Security Agency (NSA) created the open-source Ghidra software reverse engineering (SRE) framework, which has grown to be the standard tool for malware analysis and SRE. Because of its extensive feature set, it is very efficient at breaking down malicious software to comprehend its operation, behavior, and any weaknesses.

In addition, cybersecurity strategies have to constantly evolve and adapt because of the dynamic nature of cyber threats. Analysts have to stay alert and proactive in their approach to malware analysis as adversaries adapt their strategies to avoid detection and take advantage of vulnerabilities. Ghidra's adaptability and flexibility make it a vital tool for managing an evolving threat environment. Because of its flexible structure and modular architecture, it is possible to include unique scripts and plugins that are designed to meet certain analysis goals. Ghidra gives analysts the flexibility and scalability they need to tackle a broad range of cybersecurity challenges, whether they are conducting forensic investigations, reverse engineering unique protocols, or dissecting malware samples.

1.2 MOTIVATION FOR THE WORK:

Cyberattacks targeting organizations and individuals worldwide are increasing in frequency and complexity. Cybercriminals employ different methods to avoid being caught and infiltrate systems for malicious activities such as stealing data, committing financial crimes, and carrying out espionage. To address these threats, there is an urgent need to strengthen cybersecurity defenses through advanced analysis techniques. Ghidra, with its powerful suite of features for binary analysis and reverse engineering, is an invaluable resource in this endeavor.

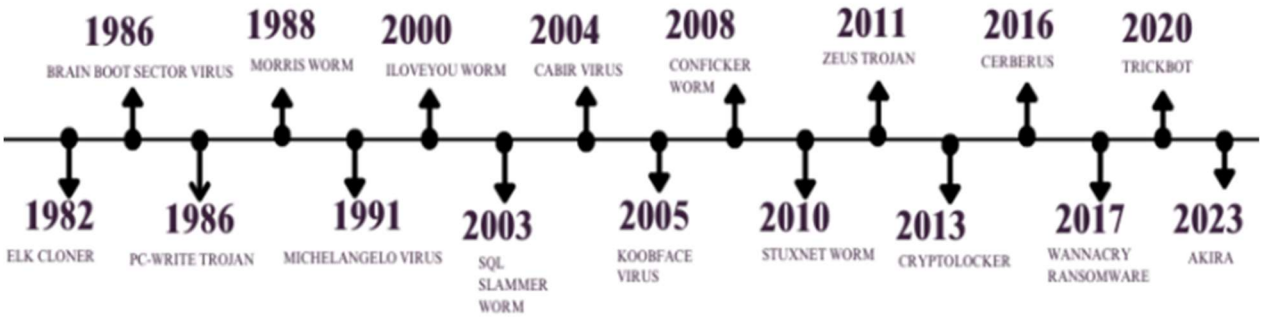

FIGURE 1.1: Evolution of Malware (1982-2023)

1.3 INTRODUCTION TO THE PROJECT

This section provides a detailed overview of the project, explaining the core concepts of malware analysis and the methodologies employed. Additionally, it explores the role of Ghidra in the analysis and reverse engineering process, highlighting its capabilities in disassembling binaries, decompiling code, and analysing program behaviour.

1.4 PROBLEM STATEMENT:

The problem statement articulates the primary challenges which security professionals face in battling malware threats. These problems include the propagation of polymorphic and obfuscated malware, the rapid evolution of attack techniques, and the sophistication of evasion techniques. Traditional signature-based detection systems are frequently unsuccessful against dynamic threats, necessitating the creation of more sophisticated analysis approaches and tools.

1.5 OBJECTIVE OF THE WORK:

The objectives of this project are multifaceted, aiming to address key aspects of malware analysis and enhance cybersecurity resilience. These objectives include:

- Investigating the behaviour and functionality of malware specimens.
- Identifying evasion techniques employed by malware authors to evade detection.
- Developing strategies for reverse engineering and analysis of malware.
- Developing scripts for easy and quick reverse engineering of malware samples.

- Enhancing proficiency in malware analysis and reverse engineering techniques, particularly with the utilization of Ghidra.

1.6 SUMMARY:

In short, this initiative is an intense attempt to take advantage of Ghidra's capabilities for comprehensive malware analysis. By carefully examining malware specimens and comprehending their inner workings, cybersecurity professionals can strengthen their security measures and proactively reduce emerging threats. This project aims to contribute to the security community's expertise.

CHAPTER 2
RELATED WORK INVESTIGATION

2.1 INTRODUCTION:

This section focuses on functions, symbol tree and mapping diagram analysis techniques obtained through analyzing the predefined scripts and modifying it for precise output. Our aim is to provide context for our project by reviewing existing methods and solutions, as well as to describe new benefits and innovations that emerge from our planned work.

2.2 EXISTING WORK:

Novel approaches to malware analysis that make use of the Ghidra framework's capabilities are being explored after its release. Brown and Lee used sophisticated classification algorithms with Ghidra's static analysis characteristics to present a machine learning method to automated malware detection. They utilized feature extraction from disassembled code, feature selection, and model training as part of their methodology to categorize malware samples into several families. Johnson and Adams concentrated on dynamic malware analysis with Ghidra at the same time, stressing the use of a behavior-based method to watch malware samples' runtime activity. They were able to gather system call traces, network traffic, and process activity by using dynamic instrumentation and behavioral logging, which allowed for a thorough behavioural study of the virus. Ghidra's flexibility in supporting both static and dynamic analytic methodologies was highlighted in both investigations, emphasizing its function as a memory mapping and combining the static part of analysis using c2 communcation systems for analysis.

The development of behavioral-based analysis, which relies on running and importing certain functions while evaluating the metrics, jump statements, and memory entry functions in dynamic as well as static evaluation, has been made possible by previous work in Ghidra reverse engineering. These solutions, at the same time, are specific and ineffective against a variety of malware.

2.3 PROPOSED WORK:

Our proposed work entails advancing malware analysis methodologies using Ghidra by developing custom scripts and plugins for efficient feature extraction, behavioral analysis, and signature generation. We aim to integrate static and dynamic analysis techniques within the Ghidra framework and create scripts to identify malicious functions,.

CHAPTER 3
REQUIREMENT ARTIFACTS

3.1 INTRODUCTION:

This section describes the components and prerequisites for successful reverse engineering of malware. Availability and correct installation of hardware, software and special requirements are important for effective and efficient evaluation. This section provides guidelines to ensure all necessary elements are available before starting the analysis process.

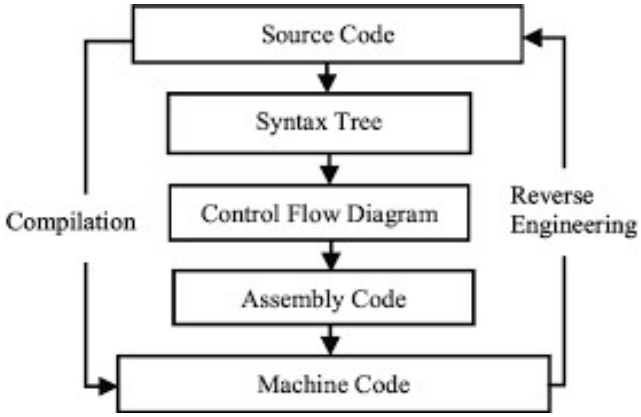

FIGURE 3.1 Reverse Engineering Workflow

3.2 HARDWARE AND SOFTWARE REQUIREMENTS:

3.2.1 Hardware Requirements

The following hardware components are essential for the malware analysis environment.

- **Host System:**
  - A high-performance computer with adequate CPU, RAM, and storage capacity to host virtual machines.
  - Sufficient disk space for storing VM images, snapshots, and analysis artifacts

- **Virtualization Platform:**
  - Installation of a compatible virtualization platform such as VMware Workstation, VirtualBox,

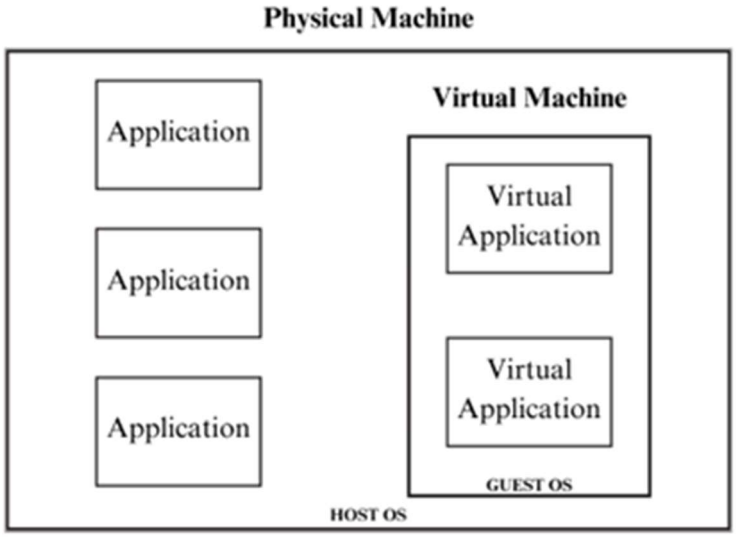

FIGURE 3.2 Visual representation OS

3.2.2 SOFTWARE REQUIREMENTS:

The software requirements include the necessary operating systems, virtual machine images, and analysis tools:

- **Operating Systems**:
  - Windows 10: For general operations and compatibility.
  - Linux: For general operations and compatibility
- **Virtual Machine Images**:
  - Downloaded and imported VM images for Windows 10 and LIinux from their respective official sources or trusted repositories.

- **Analysis Tools**:
  - Installation and configuration of malware analysis tools within the VM include Ghidra for reverse engineering , Eclipse IDE for script and plugin development and Graphiz.io

3.3 SPECIFIC PROJECT REQUIREMENTS:

- Ghidra
- Eclipse IDE
- Graphiz.io
- Ghidra Api’s
- Script Editor

CHAPTER 4
DESIGN METHODOLOGY AND ITS NOVELTY

4.1 METHODOLOGY AND GOAL:

The fundamental component of our design technique is the development and utilization of specialized scripts that automate and assist in malware analysis. By conducting a thorough analysis of software binaries, these scripts make use of the Ghidra framework's capabilities, which facilitates the detection and evaluation of potential security flaws and dangerous features.

1. **Symbol Tree Analysis**: Decoding the binary's symbol table and extracting data about variables, functions, and data structures.
2. **Memory Mapping**: Discovering executable code and data structure in a binary by mapping memory segments to the addresses at which they reside.
3. **String Analysis**: Gathering and evaluating human-readable strings written in the binaries, such as function names, API calls, and hardcoded URLs. Renaming it for leveraging the reverse engineering analysis.
4. **Vulnerable Function:** Identifying functions in the binary that might be susceptible to security vulnerabilities like buffer overflows, format string shortcomings or integer overflows.

STEPS:

1) Setting up the system which includes downloading the windows 10, Flare VM in your virtual machine.
2) Installation of Ghidra for proceeding with the reverse engineering.
3) Setting up of Eclipse IDE for script implementation.
4) Start by finding the Entry function of the imported malware sample.

```
void __fastcall f_p__zc_FUN_00012900__xref_02(uint param_1)

{
  uint Fn_param;
  uint condition;
  uint zero_int;
  int Test_val;

  Fn_param = param_1 ^ 0x7ef640f0;
  zero_int = 0;
  if (condition != 0) {
    do {
      *(byte *)(zero_int + Test_val) = *(byte *)(zero_int + Test_val) ^ (byte)Fn_param;
      Fn_param = Fn_param >> 5 | (Fn_param & OBJID_VSCROLL) << 0x1c;
      zero_int = zero_int + 1;
      Fn_param = Fn_param ^ (Fn_param * Fn_param) / 0x8677 + 1 + Fn_param * 0x787c956a;
    } while (zero_int < condition);
  }
  return;
}
```

FIGURE 4.1 Entry Function of Stuxnet

5) Begin by identifying the vulnerable functions of our PE file after locating the code's Entry function.

- Implement a script to iterate through all functions in the binary.
- For each function, extract the first two bits of the mnemonic instructions.

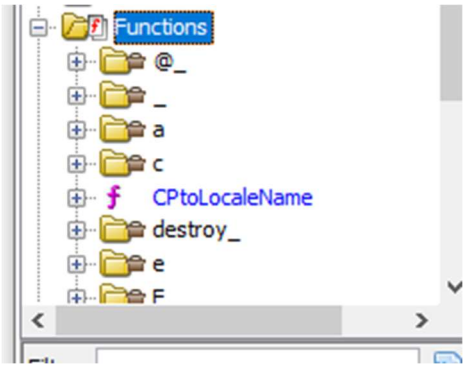

FIGURE 4.2 Functions Graph for analysis

- Count the total number of instructions within the function.
- If the first two bits of the mnemonics match a predefined pattern and the instruction count exceeds a threshold, mark the function as potentially vulnerable.

6) After Finding with Symbol Table and Memory Mapping of Vulnerable Functions

For each identified vulnerable function:

- To access the symbol table and obtain pertinent data, including function names, addresses, and parameters, use the Ghidra API.
- Retrieve the function's memory mapping details, including the start and finish addresses and the permissions of any related memory segments.

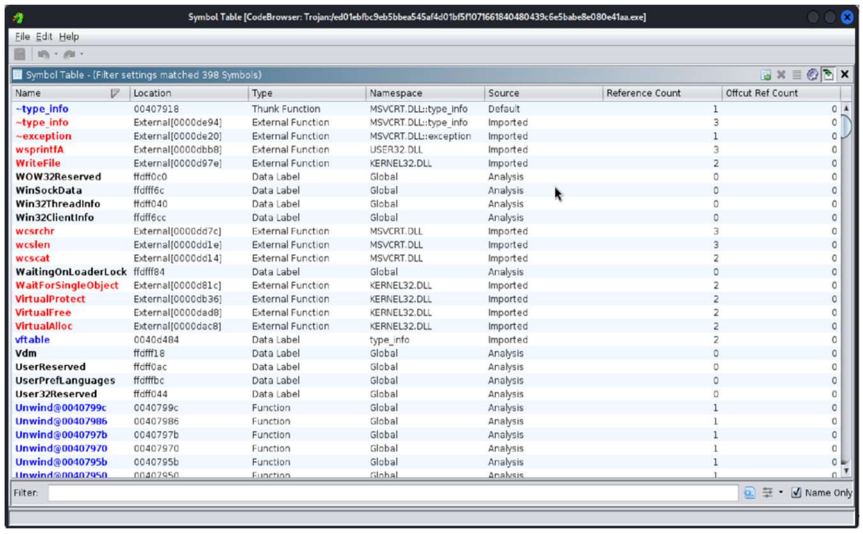

FIGURE 4.3 Symbol Table for the vulnerable functions**.**

NOVELTY

The novelty of our implementation lies in its automated approach to vulnerability detection within software binaries. Utilizing scripts for examining the first two bits of mnemonic instructions and the total number of instructions in functions, we present a new pattern-based approach to detect possible security flaws. This method offers an organized structure for vulnerability detection and streamlines the procedure of analysis. Also, by combining memory mapping and symbol table analysis, we can examine vulnerabilities in greater depth and provide a thorough understanding of the context and use of susceptible functions in the binary. Our approach emphasizes remediation over detection by providing specific recommendations for addressing discovered vulnerabilities and improve the rate of flagging of malicious binaries.

4.2 FLOW DIAGRAM:

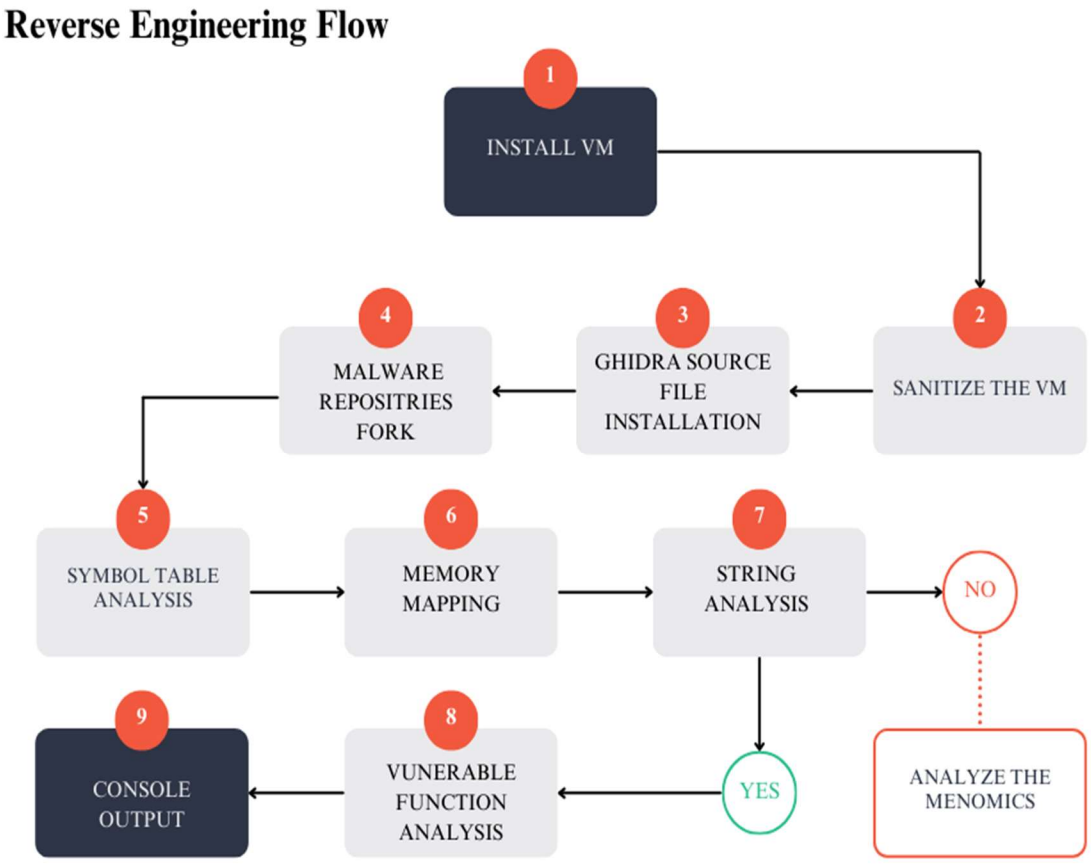

FIGURE 4.4: Reverse Engineering Process Flowchart

CHAPTER 5
TECHNICAL IMPLEMENTATION & ANALYSIS

5.1 STRING ANALYSIS:

This chapter provides an overview of malware analysis with Ghidra. It goes over functionalities present in Ghidra such as renaming functions, variables for better understanding, Function graph for better analysis of program flow, Script Manager to run scripts for rapid analysis of malware binaries , analysis options which provide automatic preliminary analysis of binaries.

5.1.1 Ghidra analysis of malicious executable

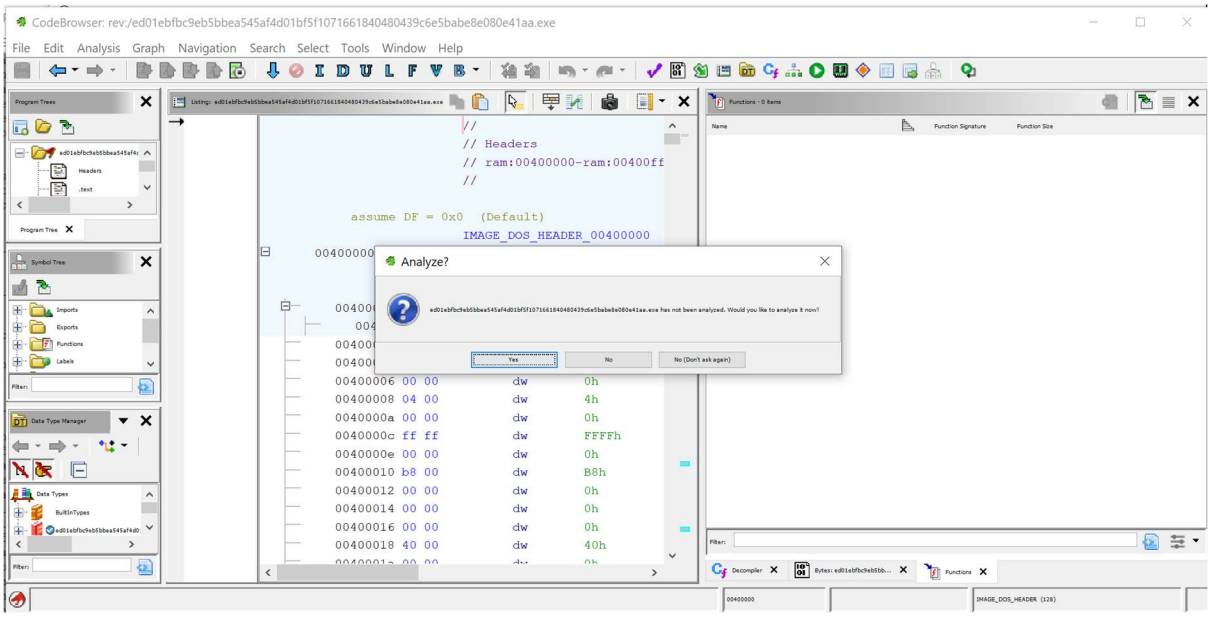

FIGURE 5.1: analysing malware sample in Ghidra

When a binary is selected for this window pops up prompting the user to allow Ghidra to analyse the binary, which Ghidra unpacks.

Function graph:

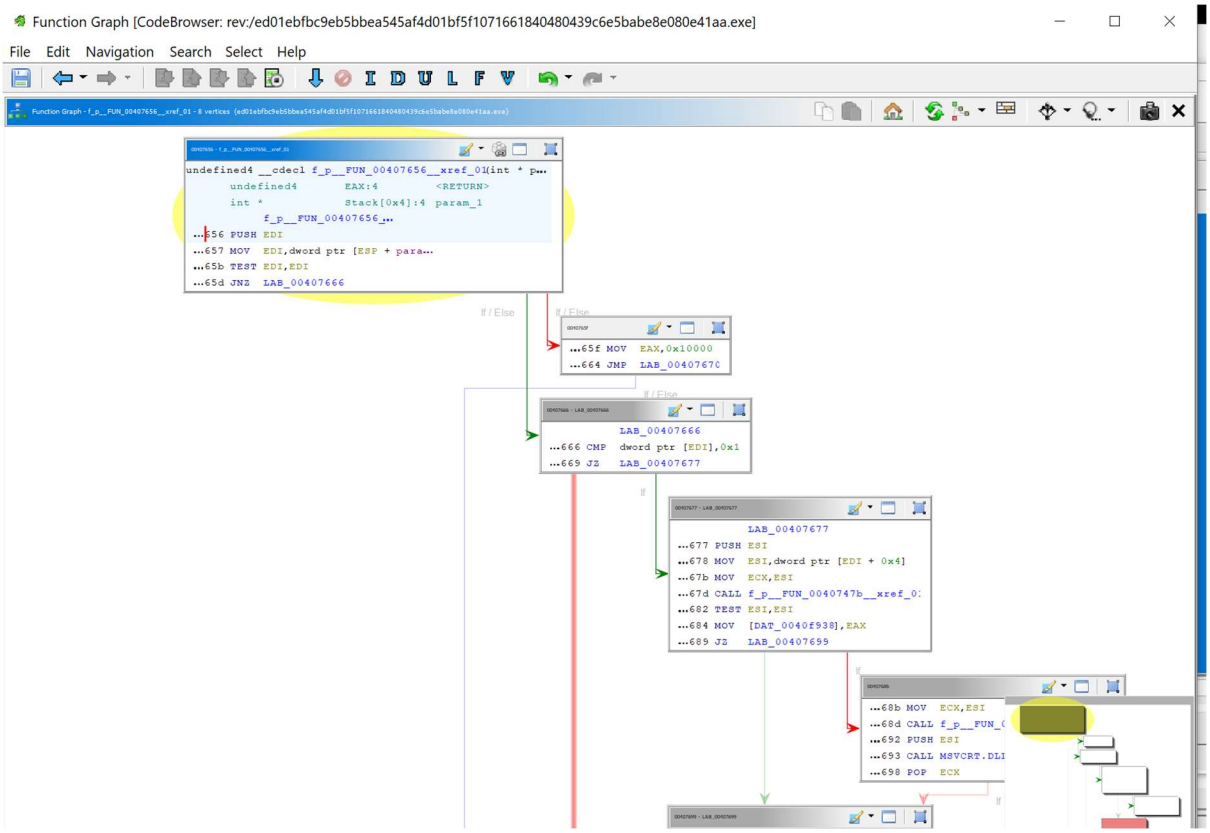

FIGURE 5.2: represents the functional flow of the program

In Ghidra, the "Function Graph" provides a visual representation of the control flow within a function, offering analysts an intuitive and interactive way to explore and understand the behavior of binary code and the structure of the code. This graphical view presents the function's basic blocks as nodes connected by directed edges, illustrating the flow of execution between instructions. Analysts can navigate the function graph, inspecting individual blocks and their relationships to identify loops, conditionals, function calls, and other control structures. The Function Graph also supports various analysis features, such as highlighting of data and control dependencies, making it a valuable tool for comprehensively analyzing and debugging complex functions during reverse engineering tasks.

5.1.2 Analysis of Stuxnet (Trojan)

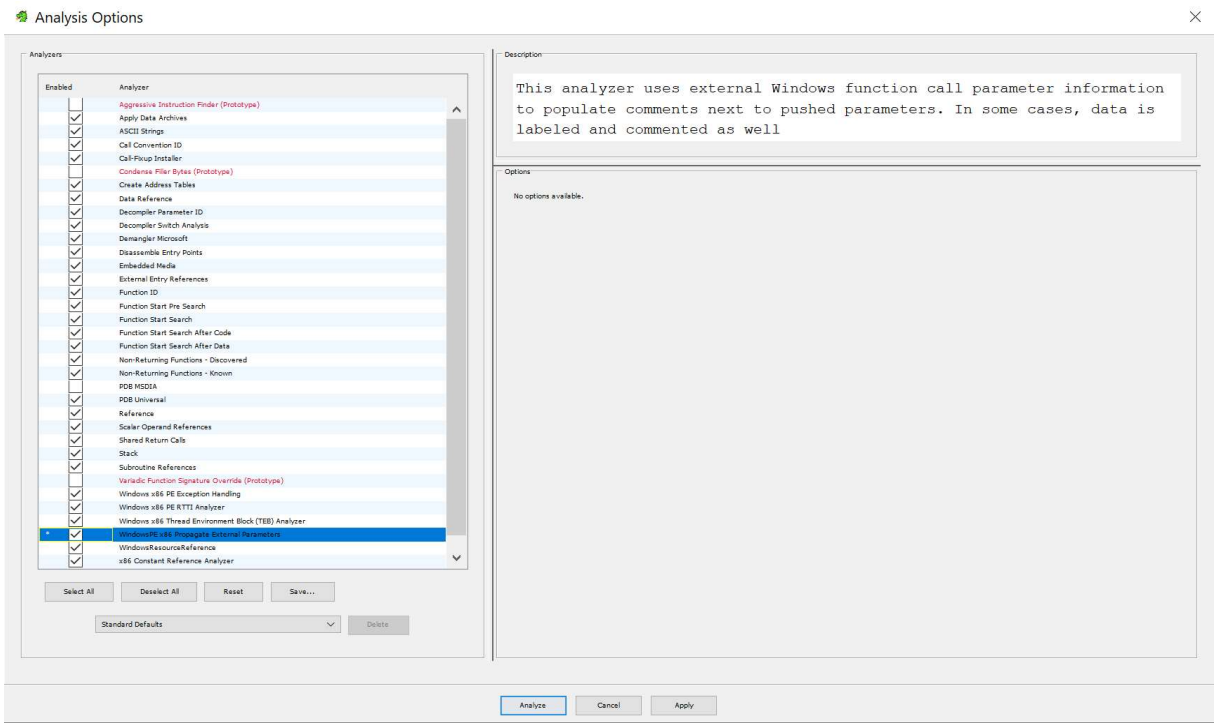

FIGURE 5.3: analysis options

A small brief on Stuxnet, it is a sophisticated Trojan discovered in 2010, jointly developed by American and Israeli intelligence agencies. Targeting Iran's nuclear program, Stuxnet was designed to sabotage industrial control systems, particularly those used in uranium enrichment facilities, by exploiting zero-day vulnerabilities in Windows operating systems and Siemens supervisory control and data acquisition (SCADA) systems. The malware utilized multiple propagation techniques, including USB drives and network shares, to infiltrate air-gapped networks and spread stealthily within target environments. Stuxnet's modular architecture allowed it to adapt and evolve over time, enabling precise control over centrifuge equipment through malicious.

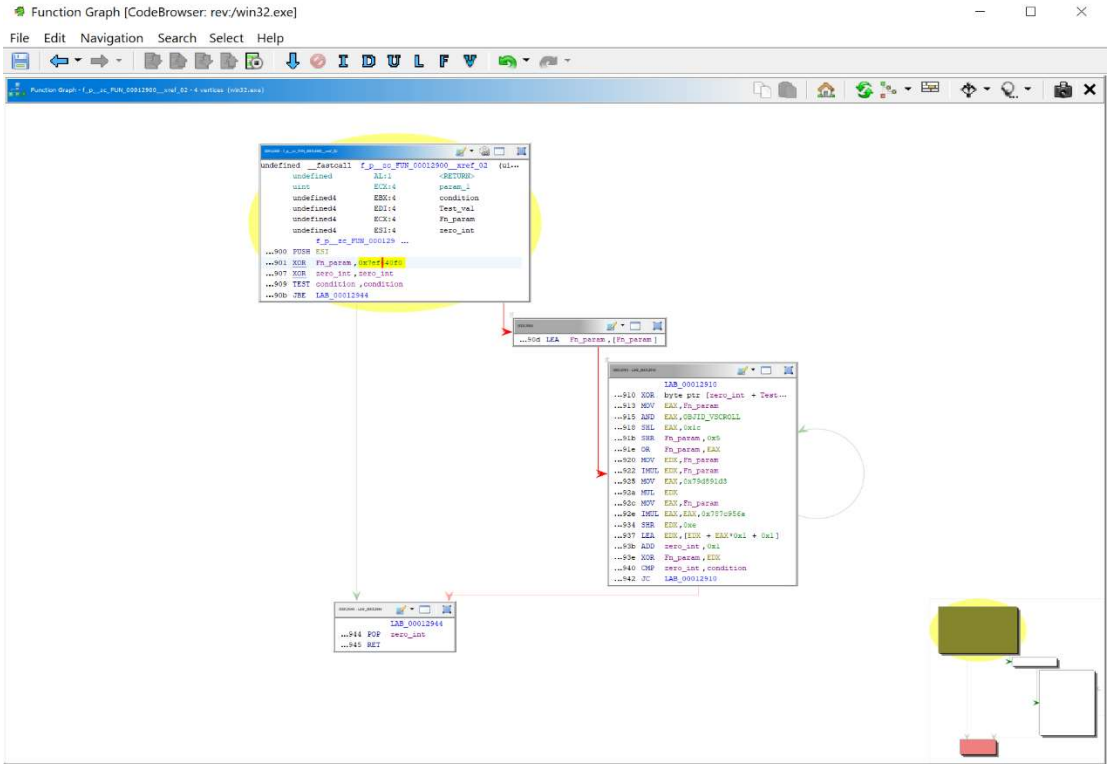

FIGURE 5.4: Represents the functional flow of the program

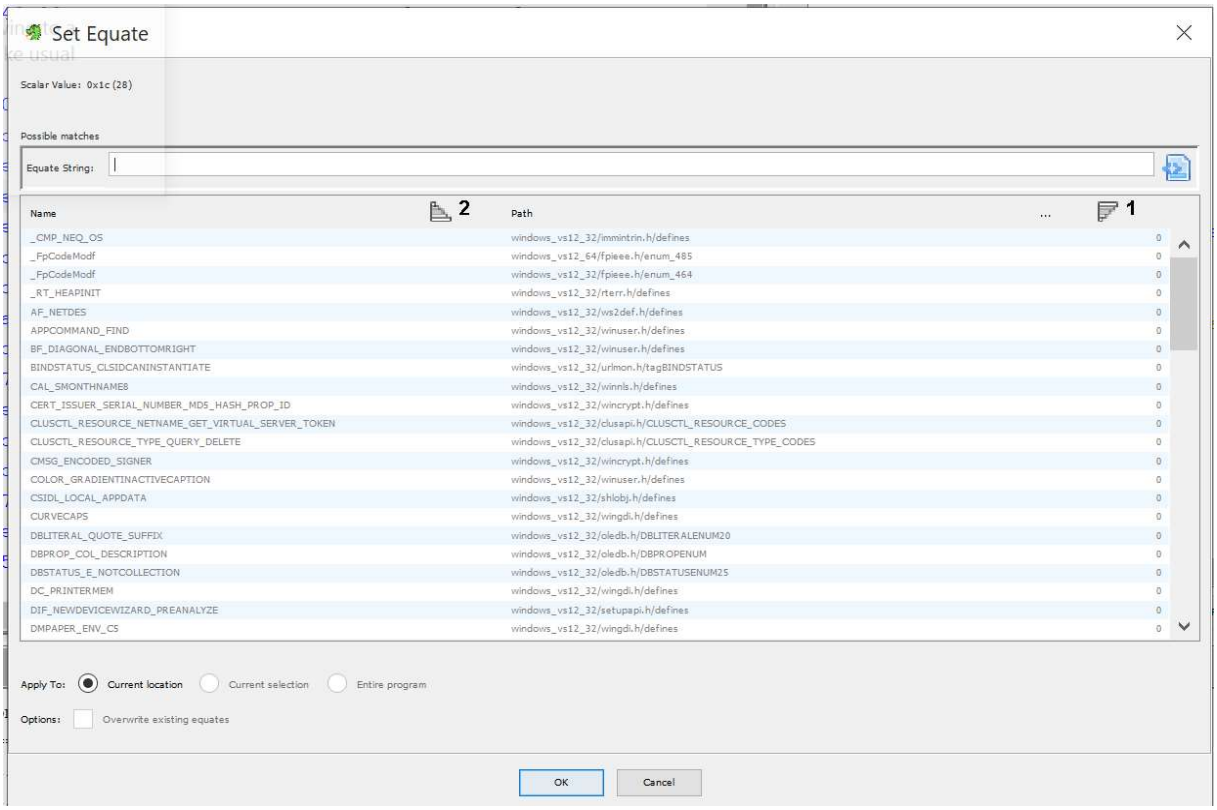

FIGURE 5.5: Set equate assigns symbolic names to the functions

Set Equate allows users to assign symbolic names to specific values or addresses within a program's memory space, enhancing readability and maintainability during analysis. By defining equates, users can create meaningful aliases for numerical constants or memory addresses, making it easier to understand and reference important values throughout the disassembly or decomplication process. Equates can represent various data types, including integers, pointers, and offsets, and can be applied to individual instructions, data structures, or entire sections of code. This feature enables analysts to annotate their analysis with descriptive labels, improving code comprehension and facilitating collaboration among team members. Additionally, equates can be organized into categories and shared across projects, ensuring consistency and standardization across multiple analyses.

Entry function:

```
void __fastcall f_p__zc_FUN_00012900__xref_02(uint param_1)

{
  uint Fn_param;
  uint condition;
  uint zero_int;
  int Test_val;

  Fn_param = param_1 ^ 0x7ef640f0;
  zero_int = 0;
  if (condition != 0) {
    do {
      *(byte *)(zero_int + Test_val) = *(byte *)(zero_int + Test_val) ^ (byte)Fn_param;
      Fn_param = Fn_param >> 5 | (Fn_param & OBJID_VSCROLL) << 0x1c;
      zero_int = zero_int + 1;
      Fn_param = Fn_param ^ (Fn_param * Fn_param) / 0x8677 + 1 + Fn_param * 0x787c956a;
    } while (zero_int < condition);
  }
  return;
}
```

FIGURE 5.6: Entry function for Stuxnet.

Entry function is the primary starting point of execution within a binary program or executable. It typically represents the first instruction executed when the program is launched and plays a crucial role in understanding the program's functionality and behavior. Ghidra automatically identifies the entry function during the analysis process, marking it as the entry point where control flow begins. Analysts rely on the entry function to initiate their analysis, often examining its code to gain insights into the program's initialization routines, runtime behavior, and overall structure. By identifying and analyzing the entry function, analysts can establish a foundational understanding of the program's

execution flow, guiding further investigation and exploration of its functionality and potential vulnerabilities.

Extracted DLL calls:

RtlInitUnicodeString function (wdm.h):

The RTL_CONSTANT_STRING macro creates a string or Unicode string structure to hold a counted string.

The escape sequence **\0** is a commonly used octal escape sequence, which **denotes the null character**, with value zero in ASCII and most encoding systems.

The Hal.dll-file lets the applications in "Microsoft Windows" communicate with your PC's hardware. It prevents applications from directly accessing your PC's memory, CPU, and other hardware devices, this feature is built to stabilize your operating system and prevent it from crashing.

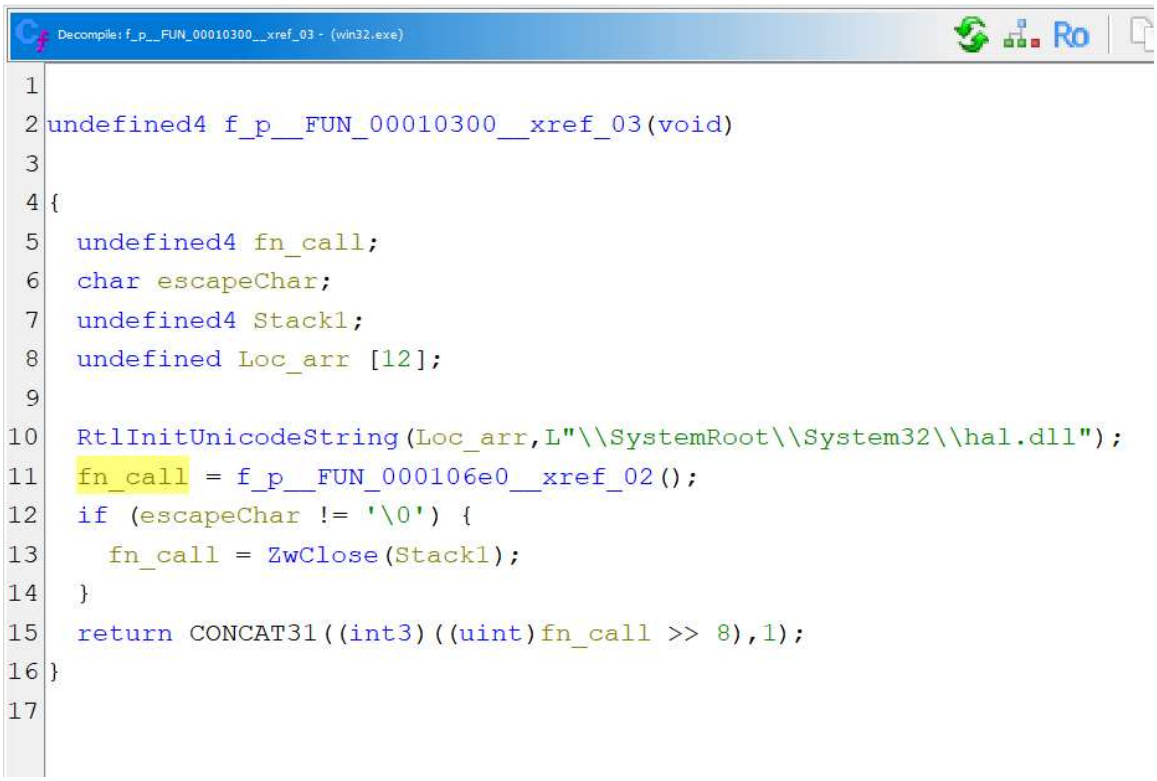

FIGURE 5.7: Renamed source code

The **ZwOpenFile** routine opens an existing file, directory, device, or volume

IoRegisterDriverReinitialization function (ntddk.h)

The IoRegisterDriverReinitialization routine is called by a driver during its initialization or reinitialization to register its Reinitialize routine to be called again before the driver's and, possibly the system's, initialization is complete.

KdDebuggerEnabled_exref

KERNEL32.DLL

Kernel32.dll file handles the memory usage in "Microsoft Windows". It is one of the primary files that are needed in order for "Windows" to function properly.

ZwQuerySystemInformation

Retrieves the specified system information.

ntoskrnl.exe

ntoskrnl.exe (short for Windows NT operating system kernel executable), also known as the kernel image, contains the kernel and executive layers of the Microsoft Windows NT kernel, and is responsible for hardware abstraction, process handling, and memory management. In addition to the kernel and executive layers, it contains the cache manager, security reference monitor, memory manager, scheduler (Dispatcher), and blue screen of death (the prose and portions of the code).[1]

ntdll.dll

The ntdll.dll file is one of the most important file in the "Microsoft Windows NT" OS family. Ntdll.dll is mostly concerned with system tasks and it includes a number of kernel-mode functions which enables the "Windows Application Programming Interface (API)". The ntdll.dll is also responsible for messages, timing, threading and synchronization in the operating system.

IoDeleteSymbolicLink

The IoDeleteSymbolicLink routine removes a symbolic link from the system.

ExFreePoolWithTag

The ExFreePoolWithTag routine deallocates a block of pool memory allocated with the specified tag.

IoRegisterDriverReinitialization

The IoRegisterDriverReinitialization routine is called by a driver during its initialization or reinitialization to register its Reinitialize routine to be called again before the driver's and, possibly the system's, initialization is complete.

IoDeleteDevice

The IoDeleteDevice routine removes a device object from the system, for example, when the underlying device is removed from the system.

IoFreeWorkItem

The IoFreeWorkItem routine frees a work item that was allocated by IoAllocateWorkItem.

MmUnmapIoSpace

The MmUnmapIoSpace routine unmaps a specified range of physical addresses previously mapped by MmMapIoSpace.

MmGetPhysicalAddress

The MmGetPhysicalAddress routine returns the physical address corresponding to a valid nonpaged virtual address.

ExAllocatePool

ExAllocatePool allocates pool memory of the specified type and returns a pointer to the allocated block

IoAllocateWorkItem-

IoAllocateWorkItem returns a pointer to the allocated IO_WORKITEM structure. The routine returns NULL if sufficient resources do not exist.

IoAttachDeviceToDeviceStack-

The IoAttachDeviceToDeviceStack routine attaches the caller's device object to the highest device object in the chain and returns a pointer to the previously highest device object.

IoCreateSymbolicLink-

The IoCreateSymbolicLink routine sets up a symbolic link between a device object name and a user-visible name for the device.

IoInitializeRemoveLockEx-

Routine, Description. IoAcquireRemoveLockEx. See IoAcquireRemoveLock. IoInitializeRemoveLockEx. Use IoInitializeRemoveLock instead.

IoCreateDevice-

The IoCreateDevice routine creates a device object for use by a driver.

IoQueueWorkItem-

The IoQueueWorkItem routine associates a WorkItem routine with a work item, and it inserts the work item into a queue for later processing by a system worker thread.

IofCompleteRequest-

The IoCompleteRequest macro indicates that the caller has completed all processing for a given I/O request and is returning the given IRP to the I/O manager.

RtlDeleteElementGenericTable-

The RtlDeleteElementGenericTable routine deletes an element from a generic table.

5.1.3 Ghidra Script manager:

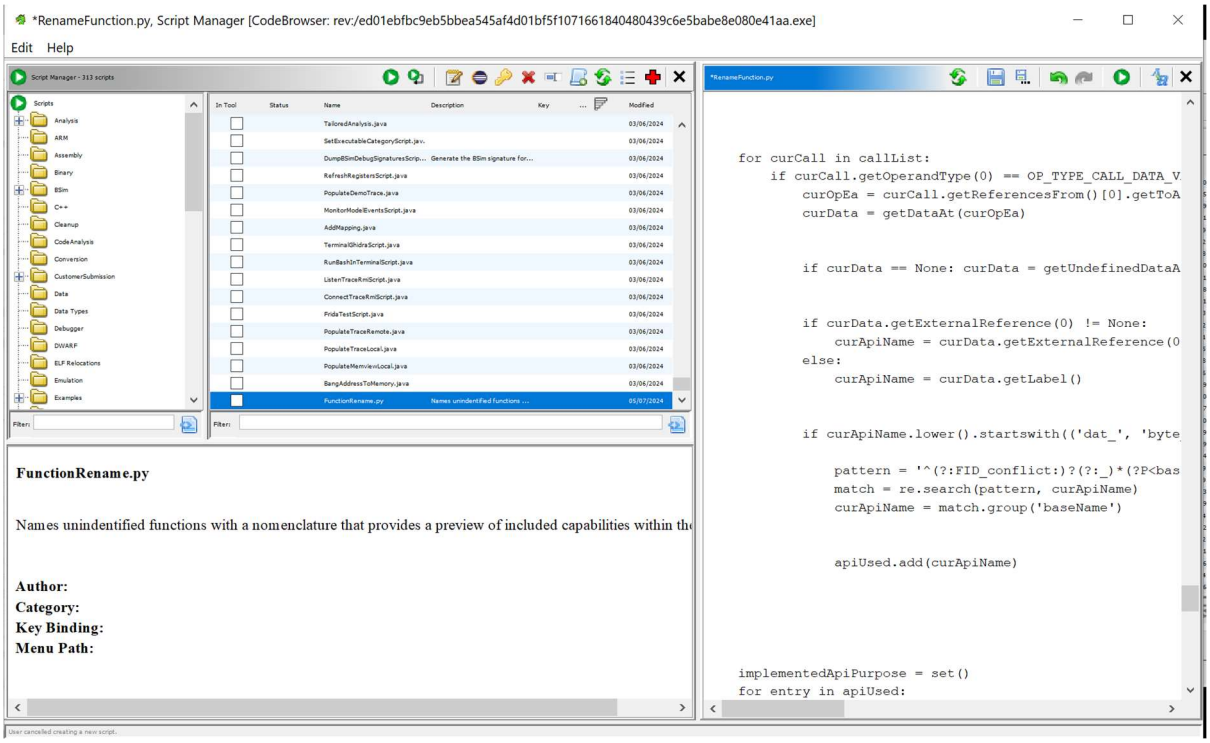

FIGURE 5.8: Ghidra Script manager

The Ghidra Script Manager is a component of the Ghidra software suite, offering users a streamlined environment for creating, organizing, and executing custom scripts to automate tasks in binary analysis and reverse engineering. With features such as built-in script templates, syntax highlighting, and code completion, the Script Manager simplifies the process of script creation and editing. Users can categorize and manage their scripts efficiently, executing them directly from the interface to interact with Ghidra's extensive API and perform diverse analysis tasks. Overall, the Script Manager empowers users to extend Ghidra's functionality, automate repetitive operations, and tailor analysis workflows to their specific needs, enhancing productivity and enabling deeper insights into binary programs.

Script to find all Malicious Strings and which functions they belong to:

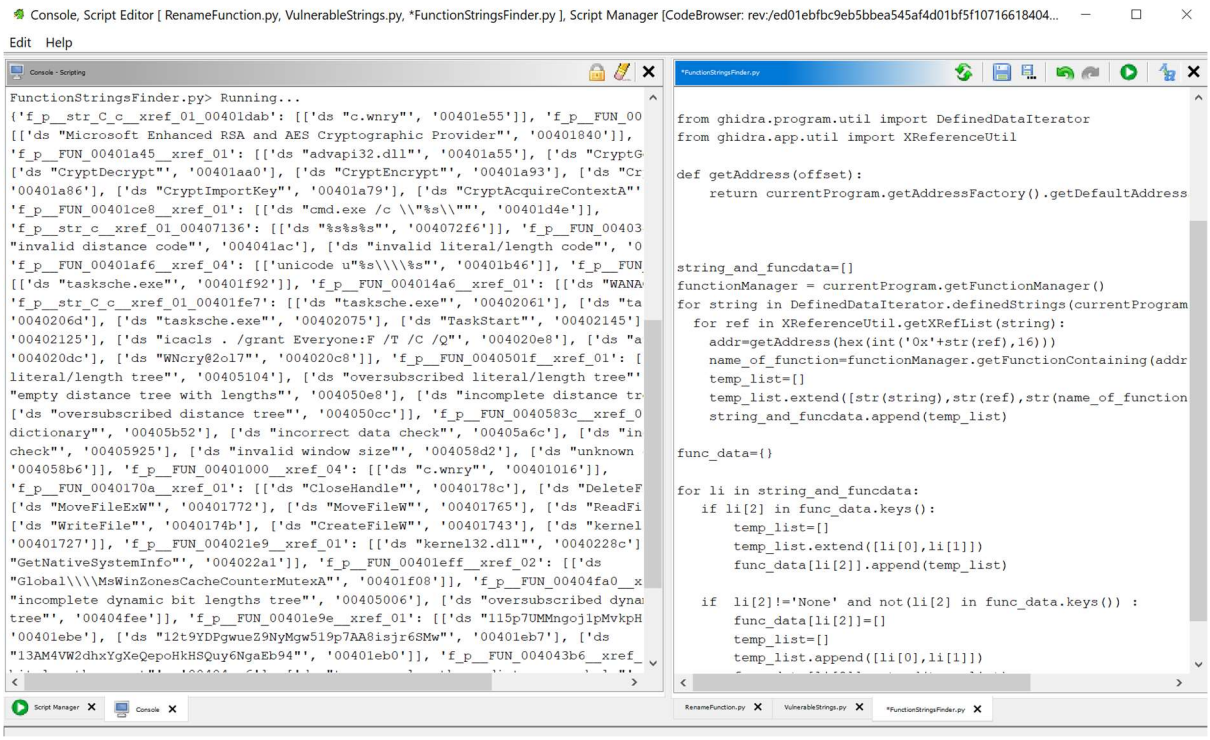

FIGURE 5.9: Script to find all Malicious Strings and which functions they belong to

```
from ghidra.program.util import DefinedDataIterator
from ghidra.app.util import XReferenceUtil

def getAddress(offset):
   return currentProgram.getAddressFactory().getDefaultAddressSpace().getAddress(offset)

string_and_funcdata=[]
functionManager = currentProgram.getFunctionManager()
for string in DefinedDataIterator.definedStrings(currentProgram):
 for ref in XReferenceUtil.getXRefList(string):

   addr=getAddress(hex(int('0x'+str(ref),16)))

   name_of_function=functionManager.getFunctionContaining(addr)
   temp_list=[]
   temp_list.extend([str(string),str(ref),str(name_of_function)])

   string_and_funcdata.append(temp_list)

func_data={}

for li in string_and_funcdata:
  if li[2] in func_data.keys():
     temp_list=[]
     temp_list.extend([li[0],li[1]])
     func_data[li[2]].append(temp_list)

  if  li[2]!='None' and not(li[2] in func_data.keys()) :
     func_data[li[2]]=[]
     temp_list=[]
     temp_list.append([li[0],li[1]])
     func_data[li[2]].extend(temp_list)

print(func_data)
```

This script is designed to analyse the interconnections between strings and functions within a binary program. It accomplishes this by first iterating over all defined strings in the program and then retrieving a list of cross-references for each string. For every cross-reference, it identifies the corresponding function and stores the string, cross-reference address, and function name in a structured format. Subsequently, the collected data is organized into a dictionary where each function name serves as a key, and the associated value is a list containing tuples of strings and cross-reference addresses referenced by that function. This organization allows for a clear understanding of which strings are utilized by each function. Finally, the script prints the resulting dictionary, providing a comprehensive overview of the relationships between strings and functions in the analysed binary program. By systematically gathering and presenting this information, the script facilitates efficient analysis and reverse engineering efforts.

Rename Function based on Function it performs script:

Before running the script,

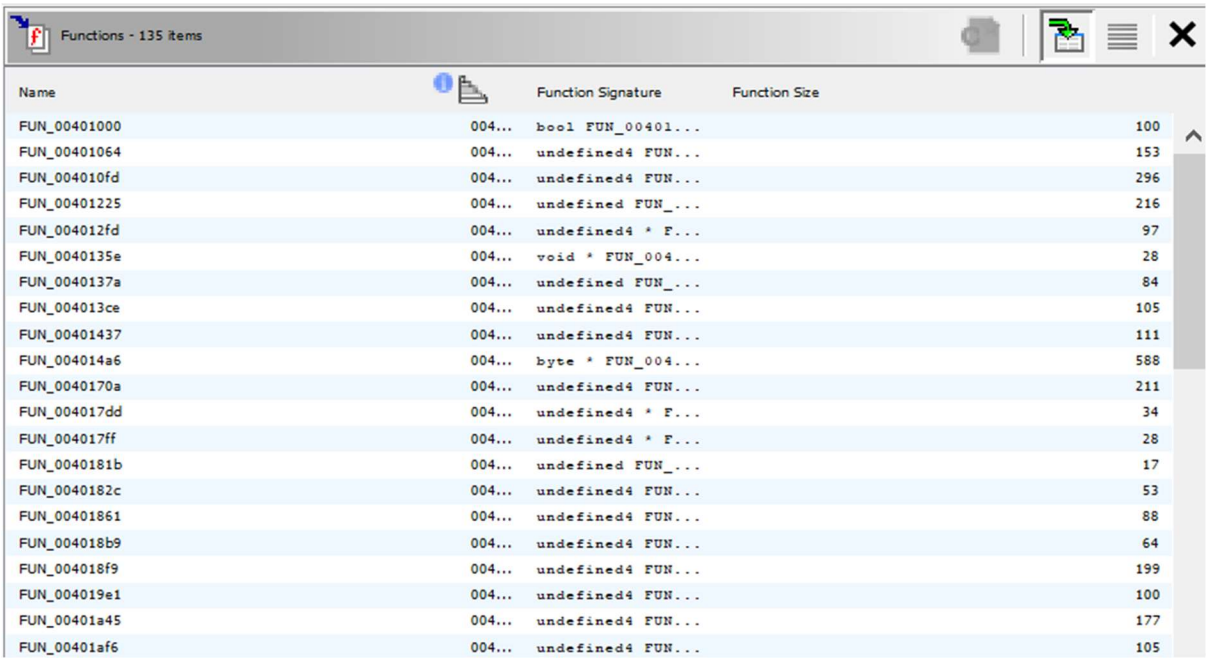

FIGURE 5.10: Rename Function based on Function it performs script.

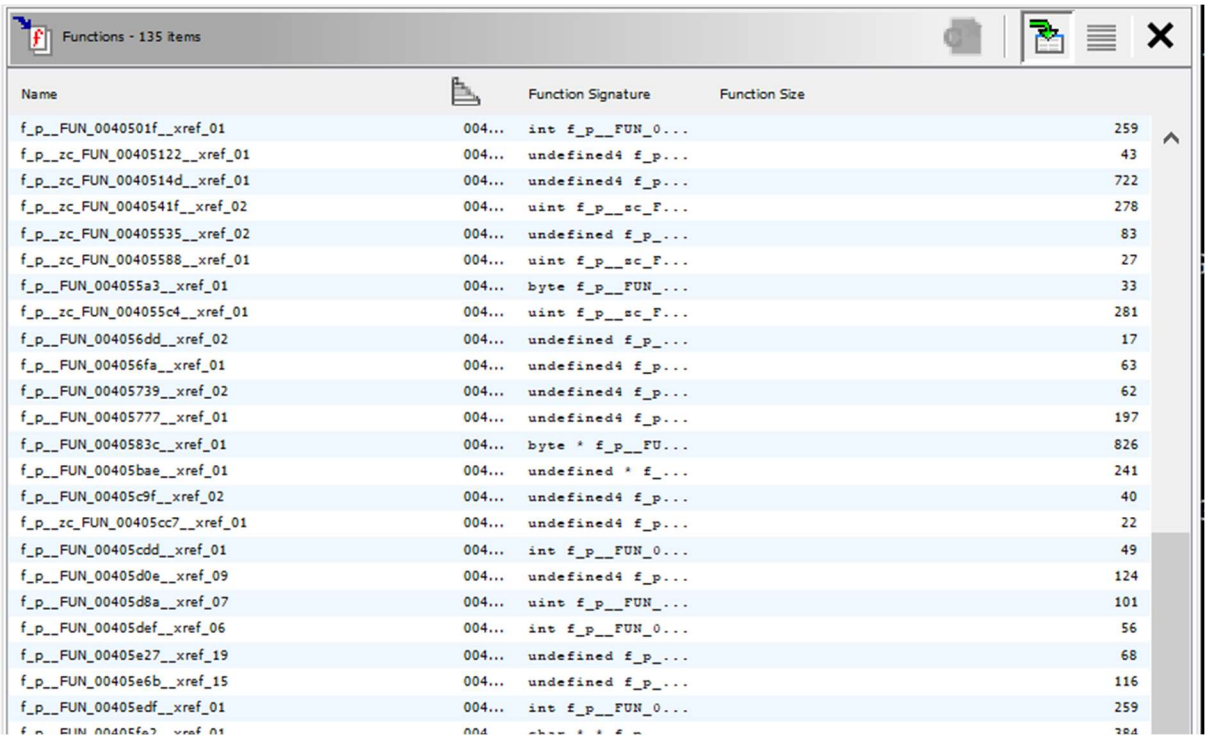

FIGURE 5.11: Renamed functions

The script uses a specific naming convention to generate new names for unidentified functions. This convention consists of prefixing each new function name with f_p__ followed by a series of capability indicators. Each capability within a function is represented by a single-letter indicator. These indicators are organized into categories such as networking (netw), registry manipulation (reg), file processing (file), process manipulation (proc), service manipulation (serv), thread functionality (thread), and string manipulation (str).

The script iterates over all unidentified functions in the binary, excluding those already named or identified by a library signature. It traverses through the functions, renaming leaf nodes first, and then propagating the capabilities up to parent functions. The script identifies the capabilities of each function by analysing the API calls within them. It maintains a dictionary of API calls and their corresponding capabilities. It checks calls to external references, statically linked functions, and function pointers stored in data variables to determine the capabilities. The script recursively renames functions until no further changes are made, ensuring that capabilities from child functions are propagated up to parent functions.

Overall, this script automates the process of providing meaningful names to unidentified functions in a binary, allowing users to quickly understand their capabilities without manually inspecting each function.

5.2 SYMBOL TABLE- A STRUCTURED REPOSITORY:

This chapter provides a comprehensive exploration of symbol tables and symbol references in Ghidra, a powerful software reverse engineering tool. Let's understand these concepts fundamentally for effective code analysis and manipulation within Ghidra.

Symbol Table:

A symbol table in Ghidra acts as a meticulously organized database that stores symbolic information about a program. Each entry within this table, referred to as a symbol, represents a crucial association between:

- **Address:** A unique memory location within the program.
- **Name:** A human-readable string assigned to the address, enhancing readability compared to raw hexadecimal values.
- **Type:** Classification of the symbol, such as function, data, label, or namespace (explained further). Ghidra recognizes various symbol types, each with distinct properties.
- **Source:** (Optional) Information regarding how the symbol was created. This can be user-defined, analyser-generated during the automated analysis process, or imported from external sources.
- **Namespaces:** Namespaces offer a more comprehensive approach to organization within the symbol table. Imagine the symbol table as a bustling city. Namespaces function like districts within the city,
  grouping related symbols together to promote order and prevent confusion.
- **References:** A collection of addresses where the symbol is referenced throughout the code. These references provide valuable insight into how the symbol is used and interacted with within the program.

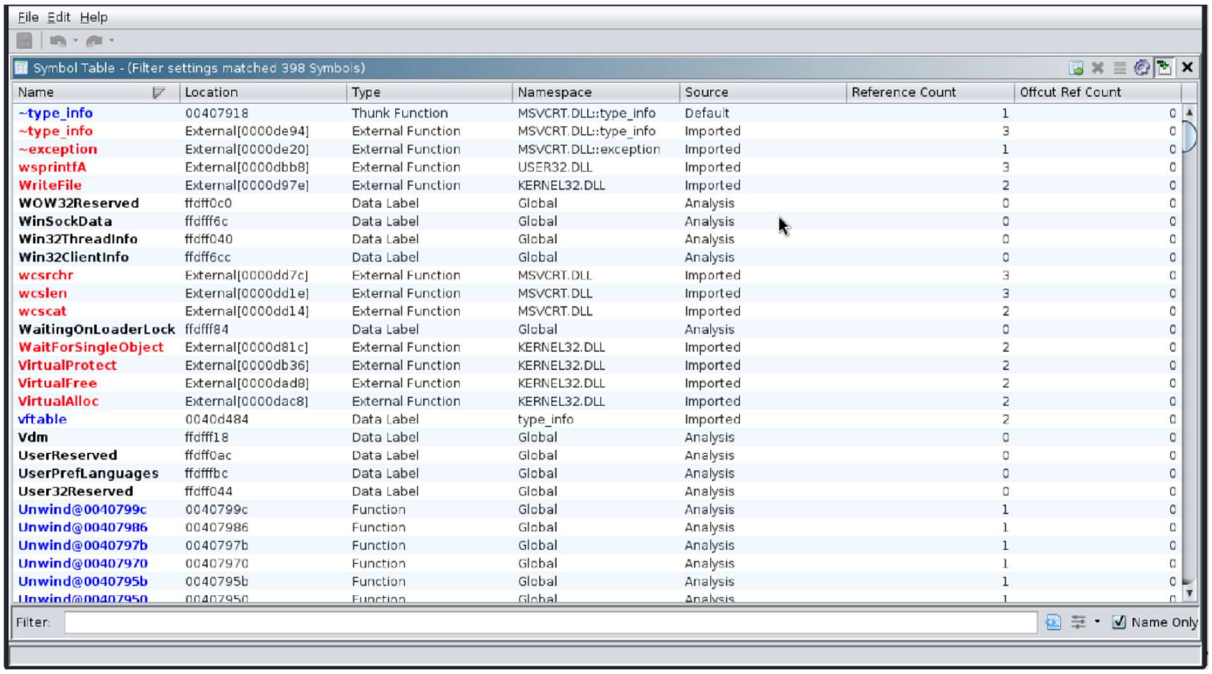

FIGURE 5.12: Symbol table information in Ghidra.

Symbol Table Benefits:

- **Improved Code Readability:** Symbolic names significantly enhance code understanding compared to cryptic memory addresses. This allows analysts to grasp the program's logic and functionality more efficiently.
- **Streamlined Analysis:** Symbols empower automated analysis processes within Ghidra. By providing context and meaning to code elements, symbols facilitate a more accurate and efficient analysis of the program.
- **Enhanced Navigation:** Ghidra's symbol table enables jumping to specific code sections by referencing symbols. This expedites navigation within the program, saving analysts valuable time.

5.2.1 Symbol Types: Tailored Representation:

Ghidra recognizes a variety of symbol types, each catering to specific elements within the program:

- **Function Symbols:** These symbols represent subroutines or procedures within the program. They typically include a name, address, and potentially arguments and return type information. Function symbols are crucial for understanding the program's modular structure and how different code sections interact.
- **Data Symbols:** Identifying named data elements like variables, constants, and arrays falls under the purview of data symbols. These symbols provide meaningful names to data elements, fostering a clearer comprehension of data manipulation within the program.
- **Label Symbols:** User-defined names assigned to specific memory locations are categorized as label symbols. Labels offer a convenient way to reference specific code sections with descriptive names, improving code organization and readability.

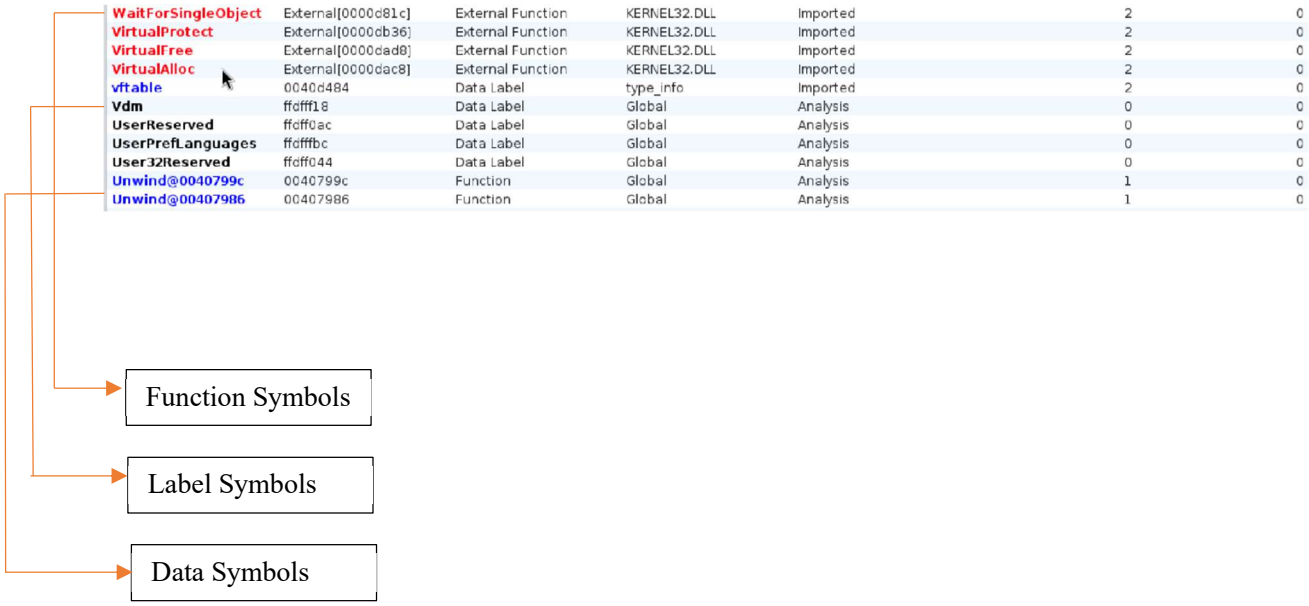

FIGURE 5.13: Symbol table information in Ghidra.

5.2.2 Investigate a Symbol Reference:

It is a common practice to identify interesting code based on Windows API references. There are multiple approaches to viewing API references within Ghidra's interface. Here, we will view references via Window >> Symbol References (note that this list includes more than Imports).

Select an API of interest and identify where that function is called. For example, the CreateProcessA API is worth investigating to understand what additional processes this program spawns. To locate a reference to this API simply click on the API name and choose a reference on the right-hand side. Below, we click on the second reference to CreateProcessA:

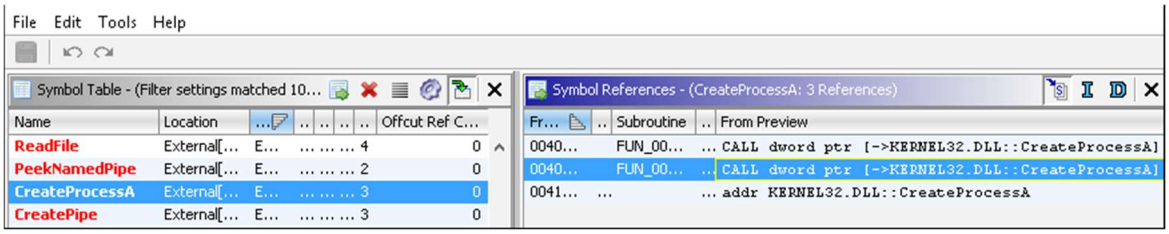

FIGURE 5.14: Identify CreateProcessA Symbol references.

Clicking a reference populates the Listing view with the appropriate disassembly:

```
0040a0bf 50          PUSH   EAX                                   ; LPPROCESS_INFORMATION lpProcessInformation for
0040a0c0 8d 45 ac    LEA    EAX=>local_58, [EBP + -0x54]
0040a0c3 50          PUSH   EAX                                   ; LPSTARTUPINFOA lpStartupInfo for CreateProcessA
0040a0c4 33 c0       XOR    EAX, EAX
0040a0c6 50          PUSH   EAX                                   ; LPCSTR lpCurrentDirectory for CreateProcessA
0040a0c7 50          PUSH   EAX                                   ; LPVOID lpEnvironment for CreateProcessA
0040a0c8 68 00 ...   PUSH   0x8000000                             ; DWORD dwCreationFlags for CreateProcessA
0040a0cd 50          PUSH   EAX                                   ; BOOL bInheritHandles for CreateProcessA
0040a0ce 50          PUSH   EAX                                   ; LPSECURITY_ATTRIBUTES lpThreadAttributes for Cr
0040a0cf 50          PUSH   EAX                                   ; LPSECURITY_ATTRIBUTES lpProcessAttributes for C
0040a0d0 68 ac ...   PUSH   s_/k_%windir%\System32\reg.exe_ADD_004157ac ; LPSTR lpCommandLine for CreateProcessA
0040a0d5 68 90 ...   PUSH   s_C:\Windows\System32\cmd.exe_00415790 ; LPCSTR lpApplicationName for CreateProcessA
0040a0da ff 15 ...   CALL   dword ptr [->KERNEL32.DLL::CreateProcessA]
```

FIGURE 5.15: Follow a CreateProcessA Symbol reference.

At virtual address 0x40A0DA we see the CALL to CreateProcessA. The PUSH instructions above the CALL refer to CreateProcessA's arguments. On the right, we see autogenerated comments that identify the function parameters (as described on Microsoft's website). The dwCreationFlags parameter specifies the process creation flags, which are characteristics of the spawned process.A complete list of flags can be found here.

The value 0x8000000 represents a symbolic constant. We can convert this to a more human-readable representation by right-clicking the value and selecting “Set Equate…”. The resulting window displays multiple options for the chosen hexadecimal value. To identify the correct choice for this API, notice that most of the process creation flag options on Microsoft’s website began with “CREATE_”. Typing this into the “Equate String” field yields only one option: CREATE_NO_WINDOW. This flag specifies that the spawned process should not launch a console window and is likely the correct choice. Select this value and press OK:

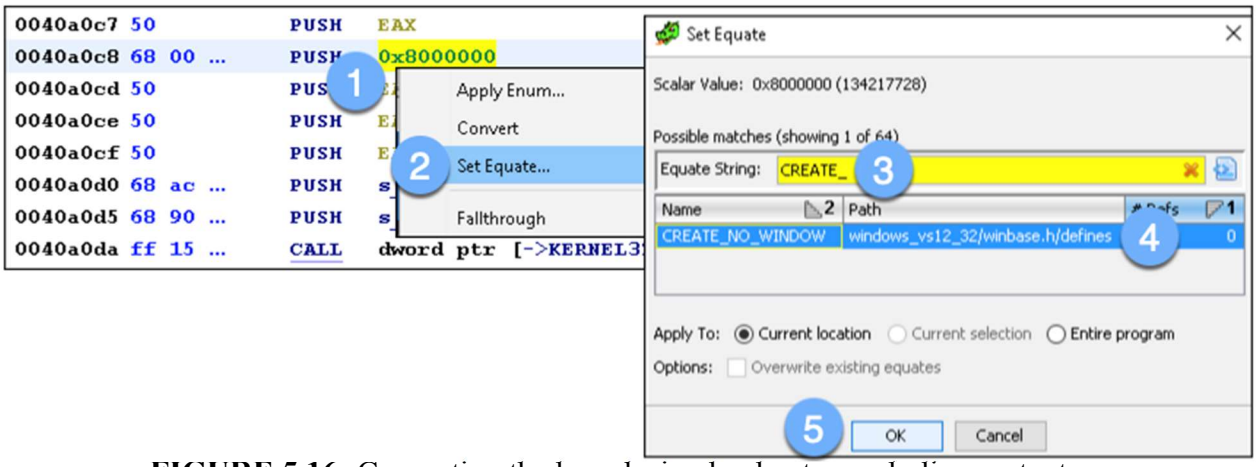

FIGURE 5.16: Converting the hexadecimal value to symbolic constant.

The assembly now displays the text representation of the creation flag instead of the hexadecimal value:

Scripting:

Ghidra includes support for writing Java and Python (via Jython) scripts to automate analysis. To view built-in scripts, go to *Window >> Script Manager*. There are over 200 scripts included with many more available online.

As an example, we will write a simple Python script to print all CALL instructions within a specific function. To create this script, choose “Create New Script” from the top-right of the Script Manager. Select Python for the script type and give the script a name:

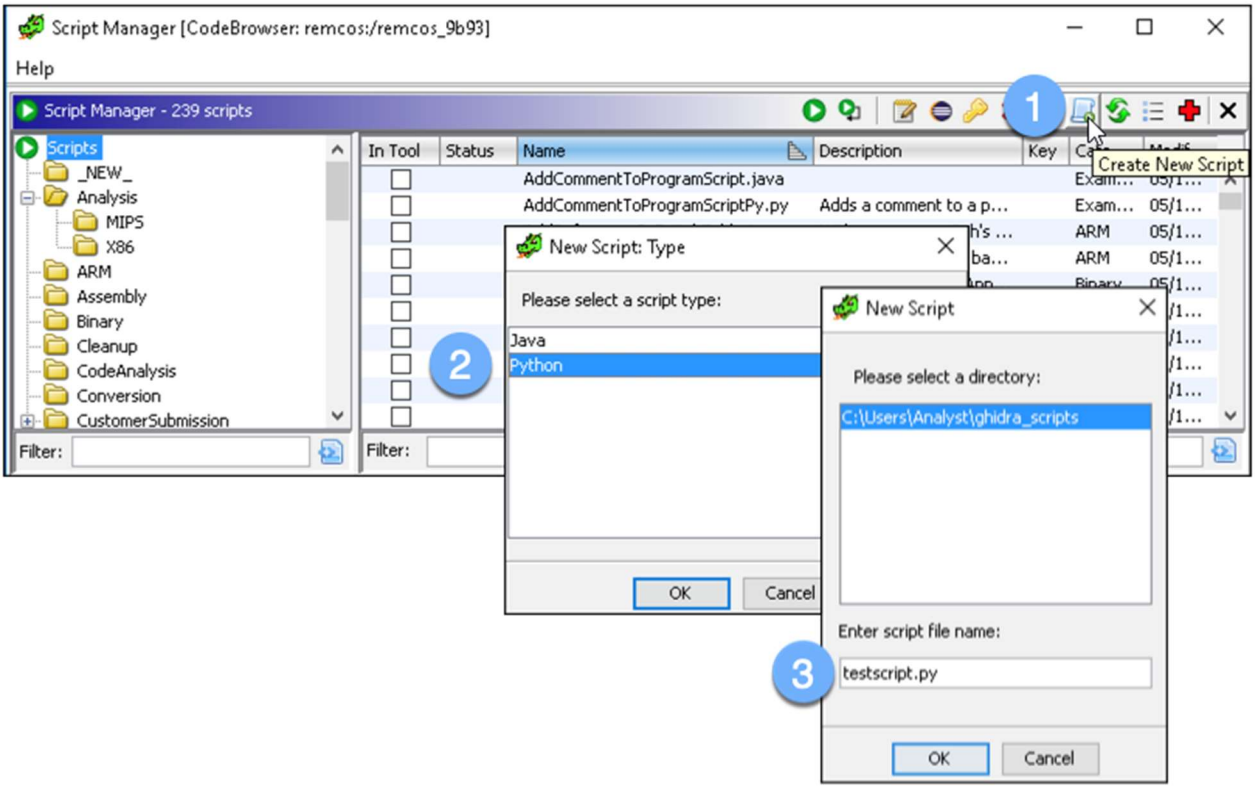

FIGURE 5.17: Choose to create a Python script.

```
import ghidra.app.script.GhidraScript;
import ghidra.program.model.lang.protorules.*;
import ghidra.program.model.mem.*;
import ghidra.program.model.lang.*;
import ghidra.program.model.pcode.*;
import ghidra.program.model.data.ISF.*;
import ghidra.program.model.util.*;
import ghidra.program.model.reloc.*;
import ghidra.program.model.data.*;
import ghidra.program.model.block.*;
import ghidra.program.model.symbol.*;
import ghidra.program.model.scalar.*;
import ghidra.program.model.listing.*;
import ghidra.program.model.address.*;

public class j extends GhidraScript {

   public void run() throws Exception {
        println("Analyzing functions and symbols...");

        // Get all functions in the program
        FunctionIterator iter = currentProgram.getListing().getFunctions(true);
        while (iter.hasNext()) {
            Function function = iter.next();

            // Check criteria for identifying important functions
            if (isImportantFunction(function)) {
                println("Potential Important Function: " + function.getName());

                // Get the corresponding symbol for the function
                Symbol symbol = getSymbolForFunction(function);
                if (symbol != null) {
                    println("Symbol: " + symbol.getName());
                } else {
                    println("Symbol not found for function: " + function.getName());
                }
            }
        }

        println("Analysis complete.");
    }

    private boolean isImportantFunction(Function function) {
        // Criteria for identifying important functions
        // Example criteria: functions with more than 50 instructions
        // You can modify this criteria based on your specific requirements
        return countInstructions(function) > 50;
    }

    private int countInstructions(Function function) {
        int count = 0;
        InstructionIterator instructions = getInstructions(function);
        while (instructions.hasNext()) {
            instructions.next();
            count++;
        }
        return count;
    }
```

FIGURE5.18: Symbol analysis script.

As with any script, there are various ways to accomplish the same objective. Our example provides a glimpse of how functions, instructions, and operands are accessed via a Python script.

To test the script, we will run it from the function that called the clipboard APIs. This is done by performing the following actions:

- Jump to 0x00406A29 via the “g” key within the Listing window
- Switch back to the Script Manager
- Double-click the new script

The Console window should appear with output showing the calls to clipboard APIs:

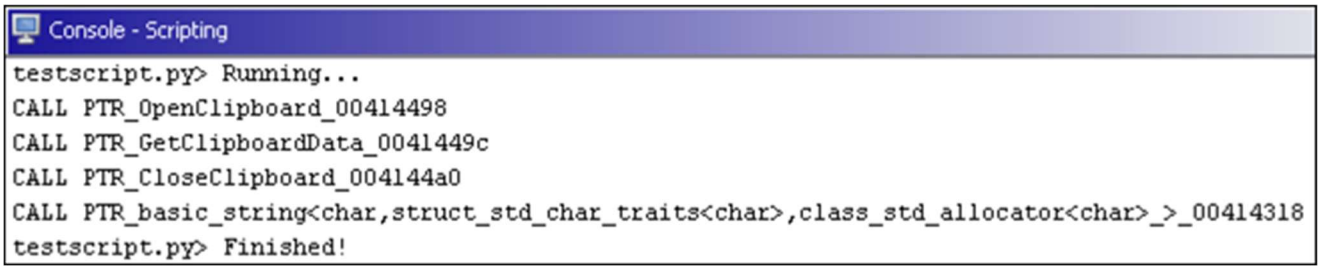

FIGURE 5.19: Functions Representation in Ghidra

5.3.3 Vulnerabilities of Functions

Vulnerable functions refers to specific functions within the malware code that represent points of weakness or potential exploitation. This Weaknesses can be exploited by attackers to take the remote access of a system and make changes in windows DLL Files and exploits the Headers of the Operating Systems.

Some of the most common vulnerable functions found in any malware include functions responsible for input/output operations (reading from or writing to files), network communication functions (socket connections), memory allocation and manipulation functions (malloc, Mampy), and cryptographic functions (encryption/decryption routines).

Understanding and identifying the vulnerable functions in malware is crucial for developing effective mitigation strategies. These strategies may include patching or updating vulnerable software components, implementing security controls (input validation, access controls), using security tools (antivirus, intrusion detection systems) with Ghidra we are able to analyze the important functions present inside the functions table and segregate according to their vulnerabilities.

5.3.4 Investigating a Vulnerable Functions

In our analysis of the malware using Ghidra, we focused on investigating the vulnerable functions present within the binary executable. This Functions play a important role in understanding the behavior and functionality of the malware codebase and serving as building blocks for its entry operations. Leveraging Ghidra's script analysis capabilities, we meticulously examined each function to uncover potential vulnerabilities and security risks

Implementation procedure:

1. **Open the Malware Binary**:

   Launch Ghidra and load the malware binary executable into the Ghidra project.

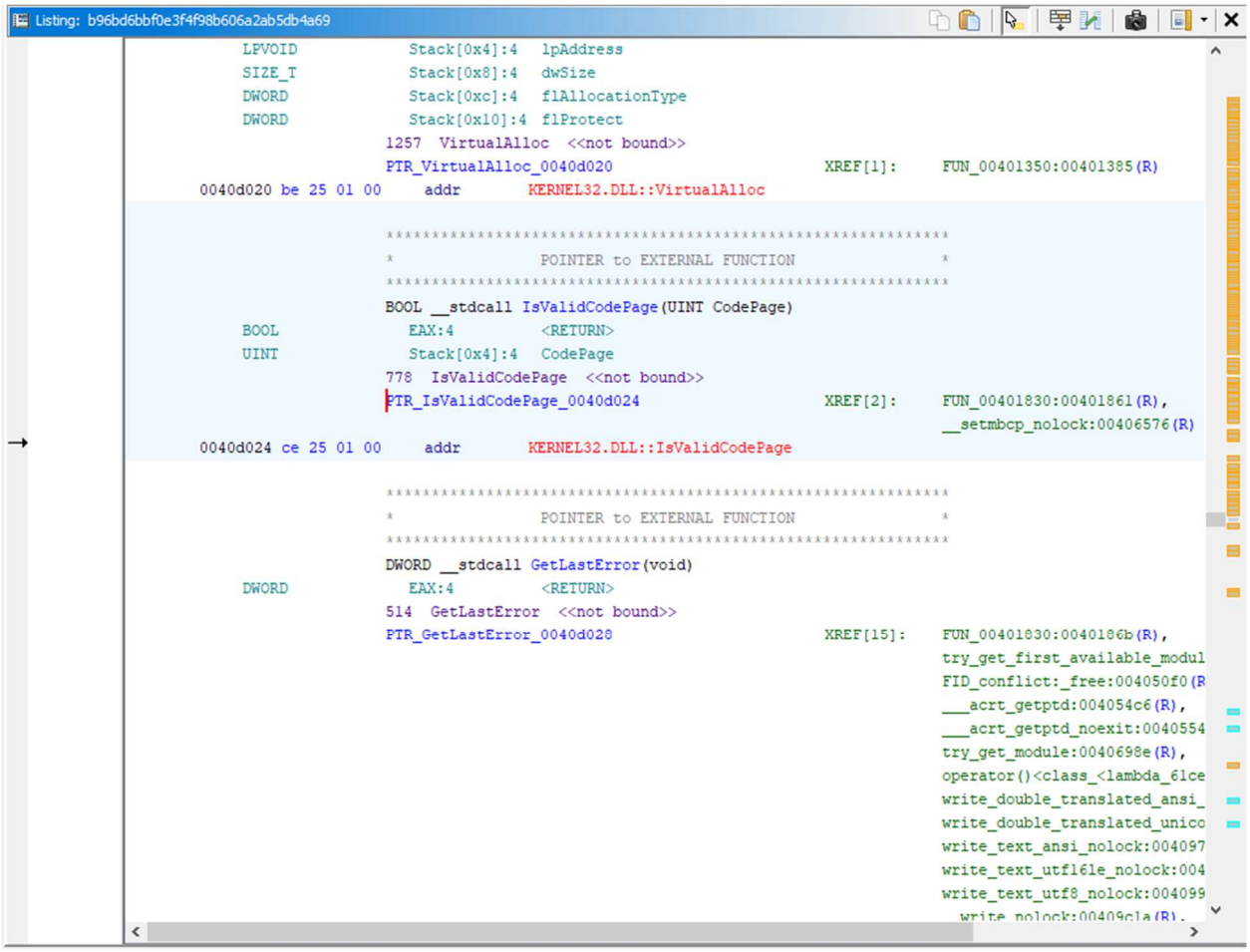

FIGURE 5.22: Binary File for the malware

2. Accessing script Manager:

The script manager is the part of ghidra functionality where we would be creating our own scripts with the help of existing ghidra API'S and the imports available.

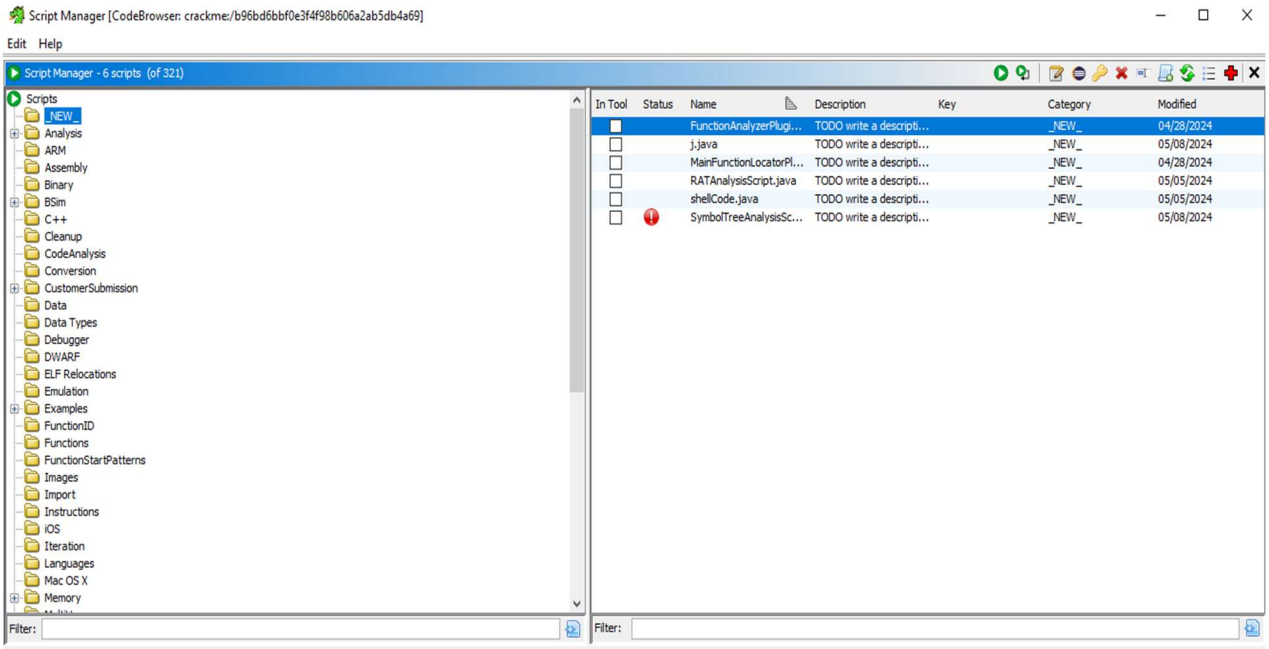

FIGURE 5.23: Representation window for script manager

3. Create New Script and Imports Ghidra API's:

Ghidra APIs (Application Programming Interfaces) provide us with a toolkit for extending and customizing the functionality of the Ghidra reverse engineering platform. These APIs allow access to Ghidra's core features, allowing us to automate the analysis tasks by providing us with access to ghidra functions, symbol table, functions and iterators.

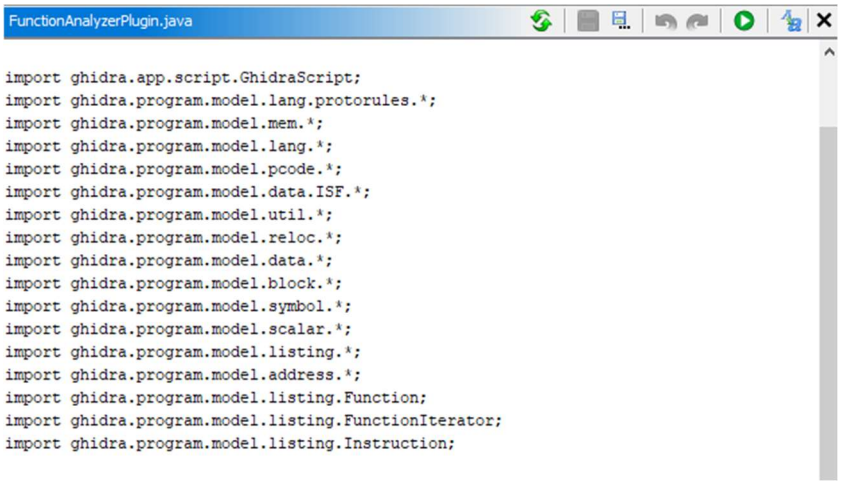

Figure 5.24: Some of the important Imports from API

4. Implementing the written script:

The written script provides us with its output by examining the initial two bits of each function (4F ,5E) and subsequently counting the total number of instructions, the script identifies functions that exhibit specific characteristics indicative of complexity or significance. Functions containing more than 50 instructions are flagged as potentially vulnerable, suggesting areas of interest for further investigation. Upon detection, these vulnerable functions are bookmarked within the Ghidra project, providing users with visual cues to facilitate navigation and reference during analysis. Furthermore, the script enhances code readability by highlighting jump statements, crucial elements in program control flow.

```
public void run() throws Exception {
    println("Analyzing functions...");

    // Get all functions in the program
    FunctionIterator iter = currentProgram.getListing().getFunctions(true);
    while (iter.hasNext()) {
        Function function = iter.next();

        // Check criteria for identifying important functions
        if (isImportantFunction(function)) {
            println("Potential Important Function: " + function.getName());
        }
    }

    println("Analysis complete.");
}

private boolean isImportantFunction(Function function) {
    // Criteria for identifying important functions
    //  functions with more than 50 instructions and first
    //two bits of executable is present in form of 4f , 5E or not
    return countInstructions(function) > 50;
}

private int countInstructions(Function function) {
    int count = 0;
    InstructionIterator instructions = getInstructions(function);
    while (instructions.hasNext()) {
        instructions.next();
        count++;
    }
    return count;
}
```

FIGURE 5.25: The implemented script in ghidra.

5. Result:

After the successful implementation of the script all the possible vulnerable functions will be available inside the console section of the ghidra.

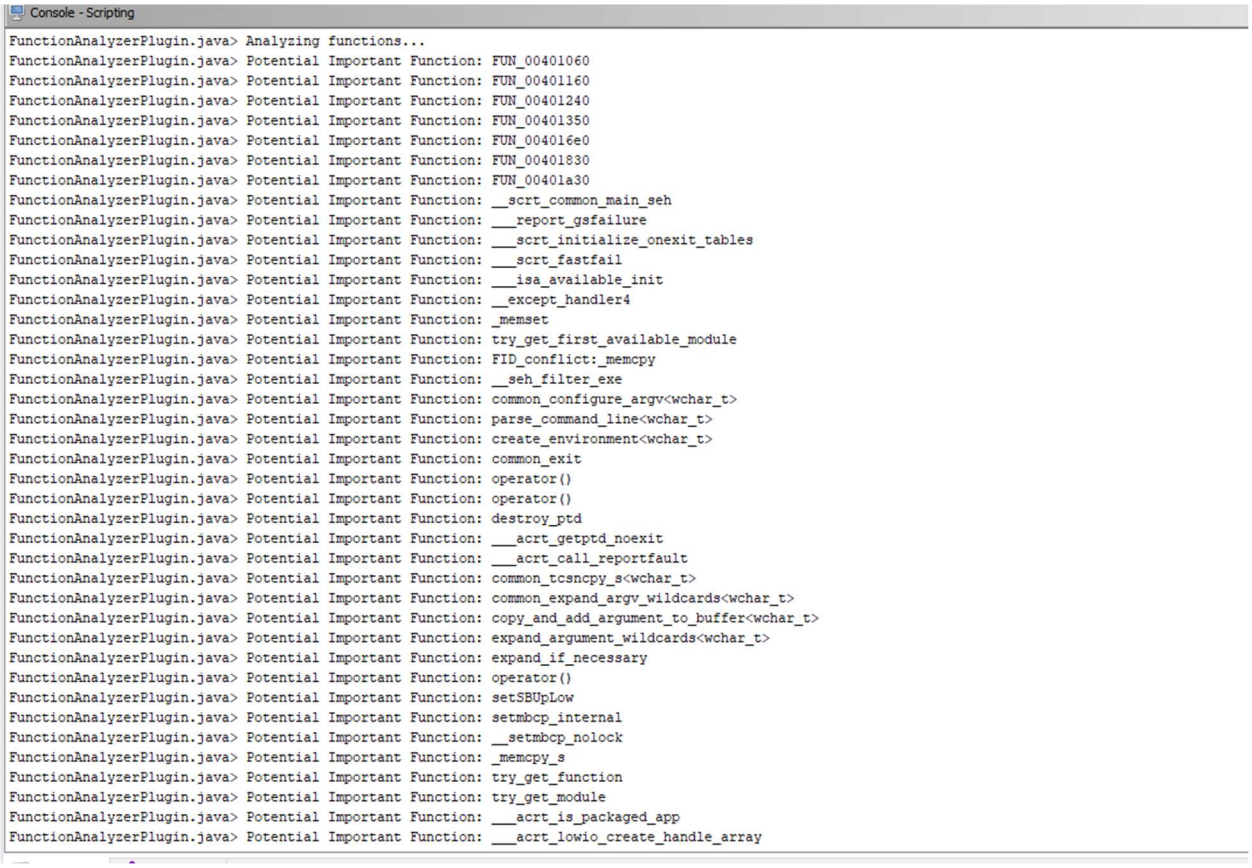

FIGURE 5.26: Shows the important functions for our PE

5.4 MEMORY MAPS:

A memory map is a list of all the different sections of memory that a program is using. Each section of the memory map has a name, a start address, an end address, a length, and some flags that indicate how the memory is being used.Our project aims to conduct a comprehensive analysis of the memory map function within the Ghidra software framework. The primary objectives of this analysis are:

- To understand the structure and organization of memory within the binaries being analysed.
- To identify and categorize different memory regions, including code, data, stack, heap, and memory-mapped I/O.
- To examine the permissions and attributes associated with each memory region, such as read, write, execute, and memory protection settings.

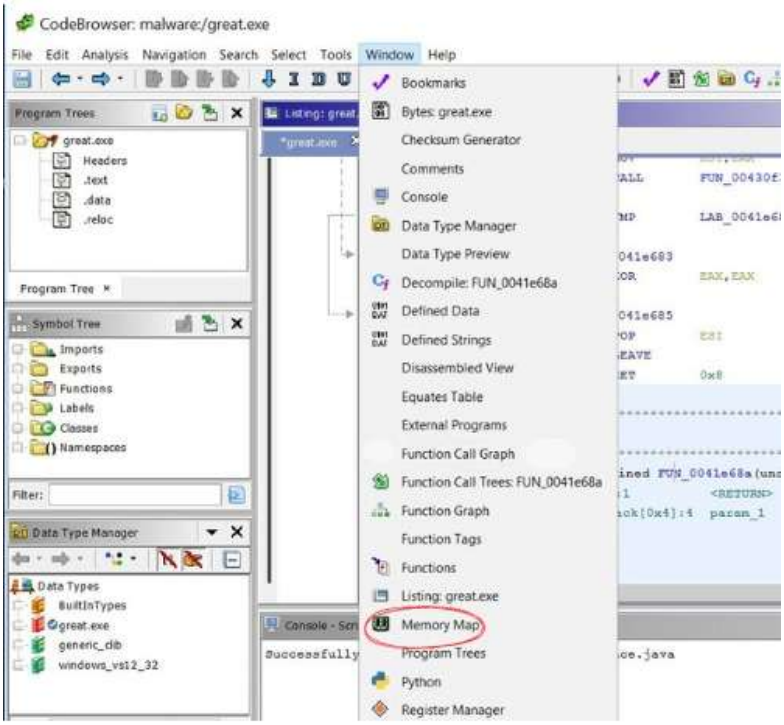

FIGURE 5.27: Locating Memory Map in Ghidra

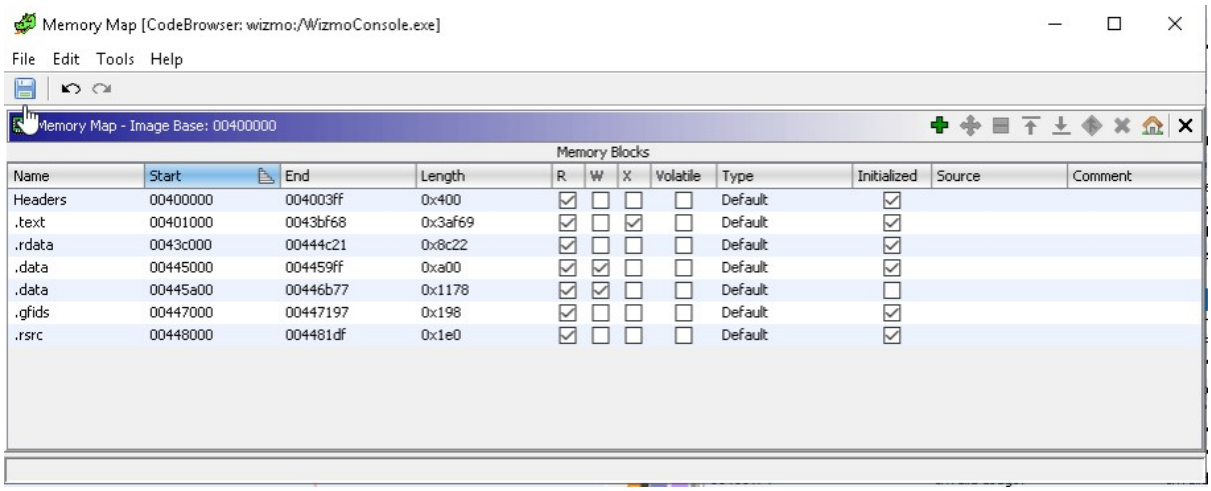

FIGURE 5.28: Memory Map in Ghidra

Here is a brief explanation of some of the columns in the memory map:

Name	The name of the memory section.
Start	The starting address of the memory section.
End	The ending address of the memory section.
Length	The length of the memory section in bytes.
R	Read permission.
W	Write permission.
X	Execute permission.
Volatile	Indicates whether the memory section can change its contents unexpectedly.
Overlayed	Indicates whether the memory section is overlaid by another section.
Type	The type of memory section.
Byte Source	The source of the bytes in the memory section.
Source	The source of the memory section.
Comment	A comment about the memory section.

Some of the common memory section types include:

Headers	This section contains the headers of the executable file.
.text	This section contains the executable code of the program.
.rdata	This section contains read-only data that is part of the program.
.data	This section contains data that can be read and written to by the program.

.bss	This section contains uninitialized data that is allocated by the program at runtime.
.idata	This section contains imported data that is used by the program from other libraries.

5.4.1 Memory Map Benefits

1. Understanding Memory Layout: The memory map provides a structured overview of how memory is organized within the binary being analysed. This includes the layout of code, data, stack, heap, and other memory regions. Understanding the memory layout is crucial for analysing and interpreting the behavior of the program.

2. Identifying Memory Regions: By examining the memory map, we can identify and categorize different memory regions used by the program. This includes code sections, data sections, stack space, heap space, and memory-mapped I/O regions.

3. Analyzing Memory Permissions: The memory map also provides information about the permissions associated with each memory region, such as read, write, and execute permissions. This is crucial for identifying potential security vulnerabilities, such as buffer overflows or code injection attacks, by understanding how memory is accessed and manipulated by the program.

4. Debugging and Analysis: Memory map information is essential for debugging and analyzing the program. It helps in understanding memory-related issues, such as memory leaks, invalid memory accesses, or segmentation faults. By examining the memory map, analysts can trace memory-related errors back to their source and understand their impact on program behavior.

5. Optimization and Performance Analysis: Memory map analysis can also aid in optimizing memory usage and improving program performance.

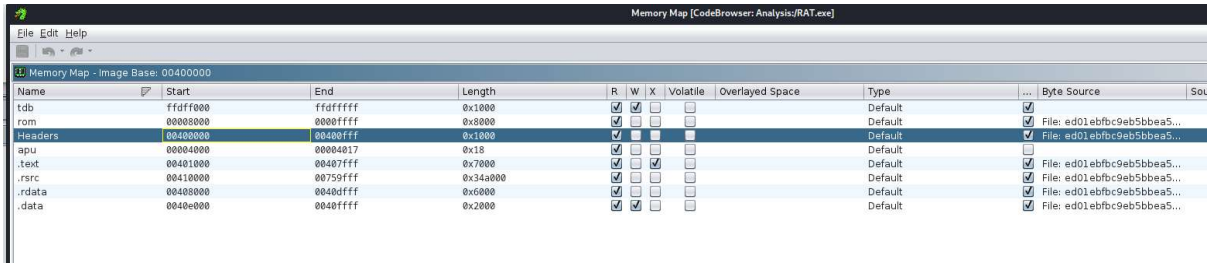

FIGURE 5.29: Headers Function in Ghidra

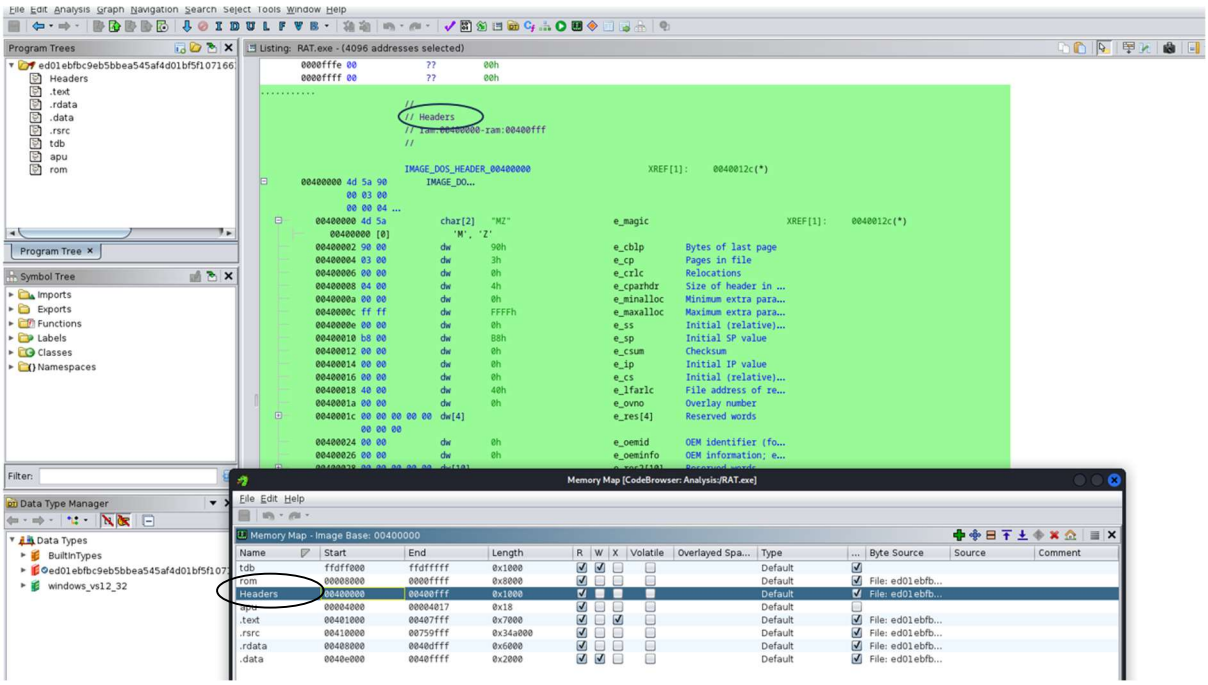

FIGURE 5.30: Highlighted lines are Headers

5.4.2 Adding new Block to Memory Map

There are two methods to add a ROMCOPY region to memory map.

Both methods are equally good, so select which one is easier for you.

You will need to repeat this step for each ROMCOPY region.

via Memory map:

In Window → Memory Map click green "+" symbol. This will open Add Memory Blockpanel.

Block Types	select “Byte mapped”
Source address	Self explanatory
Start address	Start of “destination” block“
Length	length of a block (note it will accept input as decimal if you don't use 0x prefix)
Block name	.bss

Select Read, Write, Execute flags.

.bss - represents uninitialized data, typically static variables or arrays that are not explicitly initialized by the program but are initialized to zero by default.

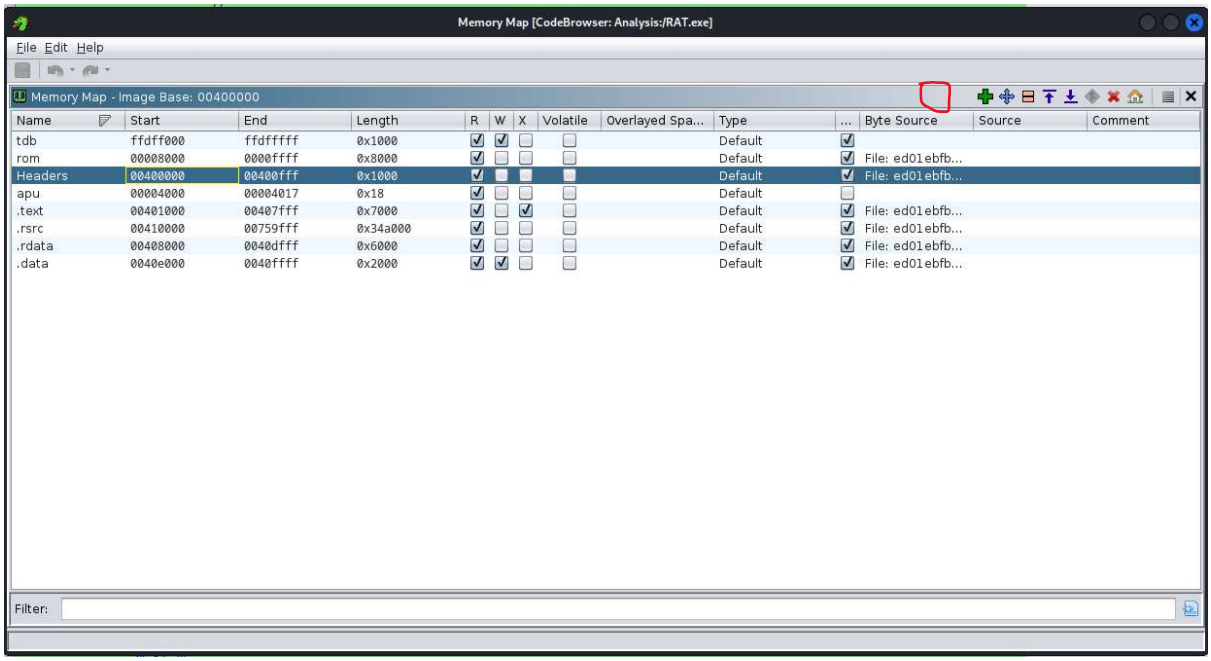

FIGURE 5.31: Creating New Memory Block in Ghidra

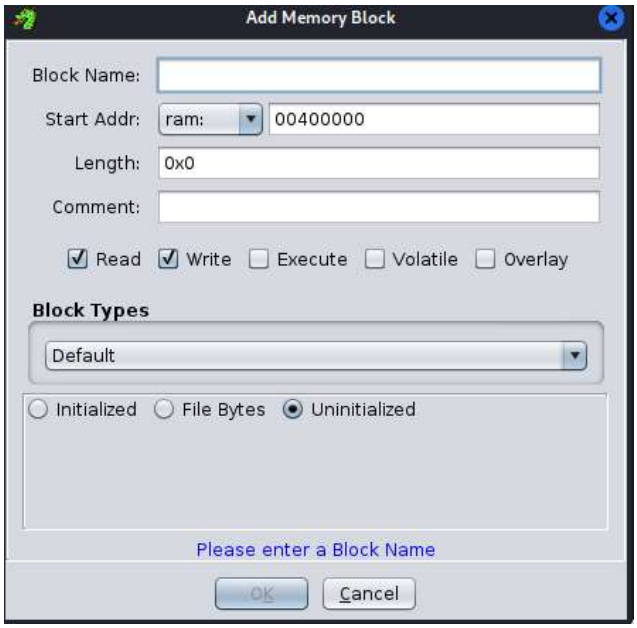

FIGURE 5.32: Adding New Memory Block in Ghidra

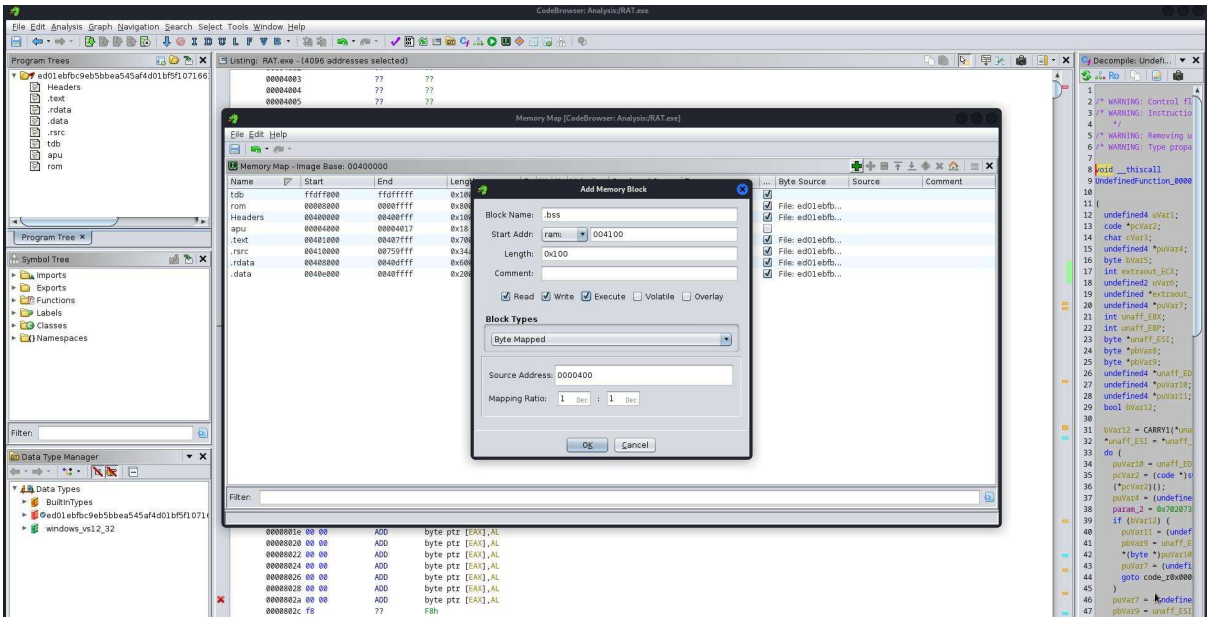

FIGURE 5.33: Assigning values in New Memory Block

5.4.3 Scripting:

Scripting can be a powerful tool for automating repetitive tasks, customizing analyses, and extending the functionality of tools like Ghidra.

Ghidra supports scripting in multiple languages, including Python and Java. Python is the most commonly used language for scripting due to its simplicity and popularity.

Script Manager: Ghidra includes a built-in Script Manager where you can create, edit, and run scripts. This interface provides access to script templates, documentation, and debugging tools to facilitate script development.

5.4.4 Scripting Use Cases

Automating Analyses: Scripts can automate repetitive tasks such as data extraction, code analysis, and vulnerability detection, saving time and effort for analysts.

Customizing Workflows: Scripts can be tailored to specific analysis workflows or project requirements, allowing users to adapt Ghidra's functionality to their needs.

Extending Functionality: Scripts can extend Ghidra's capabilities by adding new features, tools, or plugins that enhance its functionality for specific tasks or domains.

Integration with External Tools: Scripts can integrate Ghidra with external tools and systems, enabling seamless data exchange and interoperability in complex analysis environments.

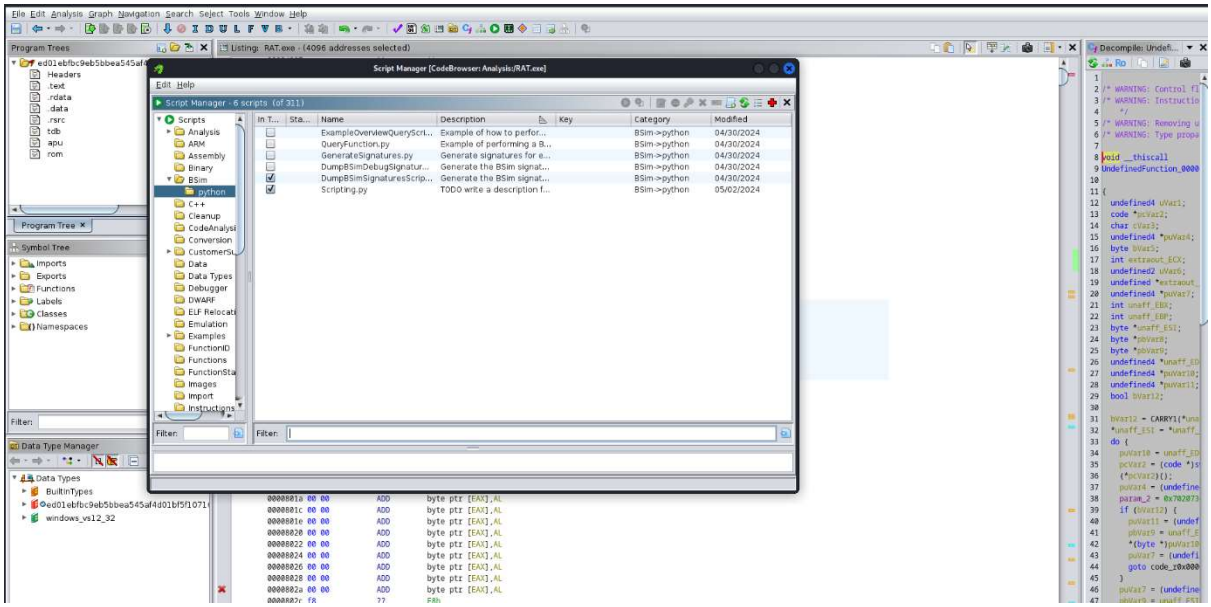

FIGURE 5.34: Script Manger in Ghidra

CHAPTER 6

PROJECT OUTCOME AND APPLICABILITY

6.1 OUTLINE:

In this chapter, we present the outcomes of our malware analysis project using the Ghidra platform. We begin by outlining the key implementations of our system and then proceed to discuss the significant findings from our analysis. We explore how these findings can be practically applied in real-world cybersecurity scenarios and draw conclusions regarding the implications of our project.

6.2 KEY IMPLEMENTATIONS OUTLINE OF THE SYSTEM:

Our system not only relies on the native functionalities of the Ghidra platform but also incorporates custom scripting to enhance the analysis process:

- **Function Analysis:** We developed custom scripts to automate the identification and classification of functions based on their signatures, calling conventions, and control flow structures. These scripts streamlined the analysis workflow and provided deeper insights into the functionality of the malware.

- **String Analysis:** Custom scripts were utilized to extract and categorize strings based on their types (e.g., ASCII, Unicode, encrypted) and usage patterns. These scripts enabled efficient string identification and helped uncover hidden messages or malicious payloads embedded within the code.

- **Symbol Table Analysis:** We developed scripts to parse and analyse the symbol table, extracting valuable information about imported functions, libraries, and external dependencies. These scripts facilitated the identification of key system calls and API functions used by the malware, enhancing our understanding of its capabilities and behaviour.

- **Memory Map Analysis:** Custom scripts were employed to analyse the memory map, mapping out memory regions, segments, and allocations within the binary. These scripts aided in the

detection of memory manipulation techniques, such as code injection or heap spraying, and provided insights into the malware's runtime behaviour and memory usage patterns.

6.3 SIGNIFICANT PROJECT OUTCOMES:

Our custom scripting efforts played a crucial role in achieving significant project outcomes:

- **Function analysis** script not only automated the process of function identification but also enabled advanced analysis techniques, such as identifying recursive functions or detecting obfuscated control flow structures.
- **String analysis** script facilitated the extraction and classification of strings, allowing us to differentiate between benign and malicious strings and uncovering encoded or encrypted data within the binary.
- **Symbol table analysis** script provided insights into the malware's external dependencies and API usage, enabling us to identify potential evasion techniques or system-level interactions.
- **Memory map analysis scripts** aided in the visualization and interpretation of memory layout, highlighting anomalous memory regions or suspicious memory.

6.4 APPLICABILITY IN REAL-WORLD APPLICATIONS:

The integration of custom scripting in our analysis workflow enhances the applicability of our project in real-world cybersecurity scenarios:

- **Threat Intelligence:** Our custom scripts can be adapted for automated malware triage and classification, accelerating threat intelligence efforts and enabling rapid detection of emerging threats.
- **Incident Response:** Script-assisted analysis facilitates quick and effective incident response, allowing security teams to identify, analyse, and mitigate malware infections in real-time.
- **Malware Reverse Engineering:** Custom scripts empower malware analysts with automation tools for reverse engineering tasks, reducing manual effort and accelerating the development of detection signatures and mitigation strategies.

6.5 INFERENCE:

In conclusion, the integration of custom scripting with Ghidra's powerful analysis capabilities significantly enhances the effectiveness and efficiency of malware analysis. By automating repetitive tasks and enabling advanced analysis techniques, our project underscores the importance of scripting in modern cybersecurity practices, with implications for threat intelligence, incident response, and malware reverse engineering.

CHPATER 7
CONCLUSIONS AND RECOMMENDATION

7.1 OUTLINE:

In our project, we describe the details of malware analysis done by the use of the Ghidra software reverse engineering tool. Our analysis focused on four key aspects: symbolic analysis of string, vulnerable function analysis, malicious symbols of symbol table and vulnerable memory maps analysis.

With the help of custom scripts to process the analysis in Ghidra, we aimed at leading automation and streamlining the procedure. These scripts helped us retrieve the important data from the malware binary and the generated reports identified relevant insights from the malware under investigation.

7.2. THE EXISTING WORK:

The malware analysis in cyber security is an essential part of cyber-security research; mostly, scholars have conducted research in this field. A number of studies have been conducted by the field specialists who have implemented a selection of methods and techniques to execute sample analysis, e. g. static and dynamic analysis. One critical work in the field of this subject is the paper "Malware Analysis Techniques," authored by Michael Sikorski and Andrew Honig [1] which covers in depth the different analysis techniques and their application. Another relevant study is "Automated Malware Analysis: "An Efficient Data-Mining Approach" article by Uppal et al. [2], which is about creating an automatic method for data mining using the mining methods.

The use of Ghidra tools by many experts has also been brought to focus as the main subjects that challenge with malware analysis. For instance, the paper "Malware Analysis Using Ghidra: Step-by-Step Guide written by Prabakaran et al. [3] providing

a detailed walkthrough of how to use Ghidra tool and its associated application for malware analysis operations.

However, they have done immense work, but our project aspires to supply more end-to-end and automated approach inside the Ghidra framework, with the usage of the custom scripts, and the designated analysis areas including string analysis, vulnerable function analysis, malicious symbol analysis, and exploited memory maps analysis.

7.3 INFERENCE:

Based on the findings from the string analysis, vulnerable function analysis, malicious symbol analysis, and vulnerable memory maps analysis, we can draw the following inferences:

- **Malware Functionality:** Specifically, the strings, functions, and symbols located allow consideration of malicious code intended to emulate data loss, instructing remote code execution, or raising privileges.

- **Potential Attack Points:** The damaged functions and memory correlates that can be misused by intruders illustrate where the attackers might get a way in and switch to a higher level of privileges on the attacked system.

- **Code Reuse and Modularity:** The analysis of symbols and function calls indicates that the malware may have been written in the manner of modular design and so that the code of other authors or existing components were, possibly, reused.

- **Masking Techniques:** The presence of obfuscated strings and other techniques implemented by malware developers is a proof of them being afraid of being caught and not only tangled in reverse engineering process.

- **Threat Intelligence:** Connecting such data with known malware to campaigns may contribute towards malware attribution to specific threat actors, which may, in turn, support targeting defence strategies.

REFERENCE

1. blogs.blackberry.com. *An Introduction to Code Analysis with Ghidra*. https://blogs.blackberry.com/en/2019/07/an-introduction-to-code-analysis-with-ghidra. Accessed 10 May 2024.
2. David, A. P. *Ghidra Software Reverse Engineering for Beginners: Analyze, Identify, and Avoid Malicious Code and Potential Threats in Your Networks and Systems*. Packt Publishing, 2020.
3. Dennis , Yurichev. *Ghidra Software Reverse Engineering for Beginners*. Creative Commons, 2015.
4. Figueroa, Marco. "A Guide to Ghidra Scripting Development for Malware Researchers - SentinelLabs." *SentinelOne*, 3 Mar. 2021, https://www.sentinelone.com/labs/a-guide-to-ghidra-scripting-development-for-malware-researchers/.
5. "Ghidrathon | Snaking Ghidra with Python 3 Scripting | Mandiant." *Google Cloud Blog*, https://cloud.google.com/blog/topics/threat-intelligence/ghidrathon-snaking-ghidra-python-3-scripting. Accessed 10 May 2024.
6. Hooper, Justin H. "How to Install Ghidra." *Medium*, 7 Mar. 2023, https://medium.com/@ecojumper30/how-to-install-ghidra-f6592ab002bb.
7. *Introduction to Reverse Engineering with Ghidra*. https://hackaday.io/course/172292-introduction-to-reverse-engineering-with-ghidra. Accessed 10 May 2024.
8. LLC, VoidStar Security. "Extending Ghidra Part 1: Setting up a Development Environment." *Voidstar Security Research Blog*, 24 Dec. 2021, https://voidstarsec.com/blog/ghidra-dev-environment.
9. Rusco, Kendall. "Automating GHIDRA: Writing a Script to Find Banned Functions." *VDA Labs*, 1 Oct. 2020, https://www.vdalabs.com/2019-03-09-automating-ghidra-writing-a-script-to-find-banned-functions/.
10. Sikorski, Michael, and Andrew Honig. *Practical Malware Analysis: The Hands-on Guide to Dissecting Malicious Software*. No Starch Press, 2012.
11. wrongbaud. "Introduction to Reverse Engineering with Ghidra: A Four Session Course." *Wrongbaud's Blog*, 30 July 2020, https://wrongbaud.github.io/posts/ghidra-training/.
12. https://ghidra.re/ghidra_docs/api/ghidra/program/model/symbol/SymbolTable.html. *An Introduction to Script Analysis with Ghidra*. Accessed 10 May 2024.