An Exploration of shared code execution for malware analysis

Moses Ashawa, Nsikak Pius Owoh, Jackie Riley, Jude Osamor and Salaheddin Hosseinzadeh

Proc. of International Conference on Artificial Intelligence, Computer, Data Sciences and Applications (ACDSA 2024)
1-2 February 2024, Victoria-Seychelles

An Exploration of shared code execution for malware analysis

Moses Ashawa
Department of Cybersecurity & Networks
line Glasgow *Caledonian University*
line 4: Scotland, UK
line 5: Moses.Ashawa@gcu.ac.uk

Nsikak Pius Owoh
Department of Cybersecurity & Networks
line *Glasgow Caledonian University*
line 4: Scotland, UK
line 5: Nsikak.owoh@gcu.ac.uk

line 1: 3rd Jackie Riley
line 2: *Department of Cybersecurity & Networks*
line 3: *Glasgow Caledonian University*
line 4: Scotland, UK
line 5: j.riley@gcu.ac.uk

line 1: 4th Jude Osamor
line 2: *Department of Cybersecurity & Networks*
line 3: *Glasgow Caledonian University*
line 4: Scotland, UK
line 5: jude.osamor@gcu.ac.uk

Salaheddin Hosseinzadeh
Department of Cybersecurity & Networks)
Glasgow Caledonian University
Scotland, UK
Salaheddin.Hosseinzadehc@gcu.ac.uk

***Abstract*— In today's ever evolving technology, malware is one of the most significant threats faced by individuals and corporate organizations. With the increasing sophistication of malware attacks, detecting malware becomes harder as many malware variants use different techniques, such as obfuscation, to evade detection. Even though advanced techniques, such as use of deep learning, prove to be of great success in classifying malware, the high computational resources needed for training and deploying deep learning models may not be feasible for all organizations or individuals. It is therefore essential to use fewer computational techniques to understand how malware can be analysed using shared code execution, which uses less computational resources. In this paper, we explored shared code execution as a novel approach for analyzing and understanding the behavior of malware. We dynamically analysed the shared code execution of the malicious payloads by looking at the dynamic link library found in NTDLL.dll. We demonstrated how samples make use of the LoadLibrary function using inline hooking techniques to overwrite the actual function code to create service execution and persistence using shared code execution. We identified functions that address the problem of encoding routine and domain obfuscation when malware uses seDebug Privilege to escalate privileg. Through realistic experiments, we found that executables such as Mod_77D4 Module, change at different instances using XOR encoding operations for each payload byte with a pre-defined key. This helps sophisticated malware to create and bind address structures for remote control. Our proposed technique shows high analytical accuracy for sophisticated samples that use encoding and obfuscation methods to evade detection.**

Keywords—Malware, code execution, suspicious file, target,

I. Introduction

Harmful applications, generally known as malware, exist in different forms, such as adware, ransomware, and rootkits, each with its own specific methods of infiltration and objectives [1]. The increasing prevalence of malware attacks in today's digital landscape and the various methods cybercriminals employ to target individuals, businesses, and even governments have made it clear that no one is safe from the potential harm caused by malicious software. From phishing emails and ransomware attacks to sophisticated hacking techniques, cybercriminals constantly evolve tactics to exploit digital system vulnerabilities using malware [2]. The impact of malware attacks is multifaceted, encompassing financial losses, compromised personal information, and a loss of trust in online platforms and services.

The financial implications of malware attacks on individuals and organisations, including the costs associated with repairing systems, recovering stolen funds or data, and potential legal repercussions, can be significant. The psychological impact on victims of malware attacks, such as feelings of violation and loss of trust in technology or online platforms, should not be underestimated. The invasion of personal or sensitive information can impact individuals, making them hesitant to engage in online activities or share personal data. This loss of trust in technology can have far-reaching consequences, not only for the victims but also for the overall digital economy.

Various techniques are used by malware to attack a target. One common technique malware uses is shared code execution, which allows the malicious code to run alongside legitimate processes on the target system. This technique is particularly effective at evading detection, as it camouflages itself within the normal operations of the computer. Malware can also utilise techniques such as code injection [3] inserting code into legitimate processes, or DLL hijacking, replacing legitimate DLL files with malicious ones [4].

The malware binaries are often deleted from an infected device, but the malicious code residues are left behind on the evaluation target. These residues include registry, service execution, timestamping, processes, strings, reverse shell sessions, and changes made to folders and files. Effective malware analysis requires technical expertise, advanced tools, and a thorough understanding of the threat landscape. The study [5] highlighted that the best way to analyse malicious files is to place them in a virtualised or restricted environment to observe their behaviour. The major limitation of manually observing malware functionalities is that different malware variants keep emerging in thousands daily, and security companies may be unable to achieve this.

Even though automated tools can dynamically analyse the samples once loaded into the testing solutions to observe and determine their functionalities and interactions with the internet and other connected system resources. Tools such as *Nepenthes* and *CWSandbox [6]* examine the behaviour of a sample file and generate a report of all the system calls and other parameters involved by the malware at runtime. One of the major drawbacks of these solutions is that they may generate different reports based on the test environment, and their results are based on a single test execution trace. As a result, significant features of the sample, especially those hidden by the malware, might be missed during the analysis process. While there are website-based tools such as VirusTotal [7] for files and URL virus analysis, other evasive malware variants try to evade detection by using the sleeping technique. The sleeping technique is an evasive loop many variants use to evade automatic and dynamic analysis by speedily consuming system resources. Also, malicious applications often encompass indices to detect infected environments by checking if a mutex object exists [8] to enhance resource sharing and consumption [9]. Other sophisticated samples have functionalities to bypass authentication and obtain administrative access.

Other malware evasion techniques include obfuscation, sandbox detection, and polymorphism [10]. Use of various obfuscation techniques, such as code encryption, variable renaming, and code splitting enable malware to hide its true intentions and make it harder for security researchers to detect and analyse its behaviour. On the other hand, the malware uses sandbox detection to determine whether it is running in a controlled sandbox or a virtualised environment. This allows the malware to avoid being detected and analysed by security researchers who often use sandboxes to analyse and understand the behaviour of suspicious files without looking at the code pattern and structure. Shared code execution analysis helps in understanding the techniques and tactics employed by malware authors, providing insights into their motivations and potential targets.

By studying shared code execution, researchers can uncover vulnerabilities in software and operating systems that are being exploited by malware, which can then be patched and strengthened to enhance cybersecurity measures. Specifically, the study aims to analyse how shared code execution can potentially enable the malicious application to gain unauthorised access to sensitive data or perform unauthorised actions on the device. By examining the interplay between different processes and the function calls within the code execution paths, a comprehensive understanding of the potential risks and vulnerabilities can be obtained. This research is crucial in developing effective countermeasures and enhancing the security of Windows-based devices against such malicious attacks.

To summarise, this paper makes the following contributions.

- We dynamically analysed the shared code execution of the malicious payloads by looking at the dynamic link library found in NTDLL.dll. We demonstrate how samples use the LoadLibrary function using inline hooking techniques to overwrite the actual function code to create service execution and persistence.
- We proposed a function that addresses the problem of encoding routine and domain obfuscation when malware uses certain services such as seDebugPrivilege to escalate privileges. In this idea, different privilege escalation calls such as GetCurrentProcess, Local Unique Identifier, PToken Privileges, AdjustTokenPrivilges and CreateRemoteThread were analysed to determine how malware accesses these features. For example, the samples analysed in this paper used multiple imports, such as kernel32 and RegCreateKeyA, for registry modification.
- We explored the potential use of shared code execution to detect and mitigate advanced persistent threats (APTs) that often employ sophisticated evasion techniques. We explore using code similarity analysis to identify shared code among different malware samples by identifying common code patterns.
- We evaluated the encoding process on many real-world malware samples and demonstrated how simple loops use the key sign of XOR encoding to obfuscate data to make investigation difficult.

.

2. BACKGROUND

2.1 Share Code Execution

Sophisticated malware can wreak havoc on computer systems and compromise sensitive data using the concept of code execution [11]. One particularly insidious form of malware is shared code execution, where malicious code is injected into legitimate software. Shared code execution can create backdoors in a system, granting hackers persistent access and control over compromised devices. This can be especially concerning for organisations, as it opens the door for further attacks, data breaches, and espionage. Malware can use shared code execution vulnerabilities to spread throughout different operating system platforms such as Windows, Android or network [12].

These vulnerabilities occur when multiple applications or processes share the same code, allowing an attacker to exploit a weakness in one application and use it to execute malicious code in another, leading to significant disruption to businesses and critical infrastructures. For instance, WannaCry ransomware in 2017 exploited a shared code execution vulnerability in Microsoft Windows, spreading rapidly across networks and infecting thousands of computers worldwide [13]. The attack caused significant disruption to businesses and critical infrastructure, highlighting the importance of patching and securing systems against known vulnerabilities.

Shared code execution of malware refers to the use of code that is shared among multiple malware samples. It involves using the same malicious code across multiple attacks. The technique allows cybercriminals to reuse code from previously successful attacks, making it easier and faster to develop new malware variants [14]. In this way, shared code execution has become an increasingly popular tactic among cybercriminals looking to maximise their impact and evade detection. This tactic is often used in targeted attacks against high-value targets. It can also be used in widespread attacks, such as malware campaigns. Furthermore, shared code execution can also create malware families, where multiple variants of the same malware share common code [15]. These families can have a long lifespan, with new variants developed over time and the shared code evolving with each iteration.

2.2 Share Code Vulnerabilities

This paper explores common forms of shared code execution vulnerabilities that malware exploits. Exploits such as code injection, dynamic link library hijacking, file inclusion and cross-site scripting are discussed.

Code Injection

One common type of shared code execution is code injection [16]. Such vulnerability can occur through user input fields, such as login forms or search bars, where the attacker inputs code that is then executed by the application [17]. Code injection attacks can be prevented by properly validating and sanitising user input. The countermeasure involves ensuring that user input is checked for any unexpected characters or the system could execute the system. Using parameterised queries or prepared statements can also help prevent injection attacks, especially when dealing with sensitive passwords or financial information. By implementing these security measures, the system can ensure that user data is protected from malicious attacks.

2.3 Dynamic Link Library Hijacking

DLL hijacking occurs when a malicious actor replaces a legitimate dynamic-link library (DLL) with a malicious one. When an application loads the compromised DLL, the attacker gains control over the execution flow and can manipulate the application's behaviour. DLL hijacking attacks are particularly dangerous as they can go unnoticed for a long time, allowing the attacker to maintain persistence and carry out various malicious activities undetected [18]. One common method of DLL hijacking is the manipulation of search order hijacking. This attack occurs when an attacker places a malicious DLL in a directory that the application searches before the legitimate DLL. As a result, when the application attempts to load the DLL, it unknowingly loads the malicious version instead.

This technique is effective because it takes advantage of the default search order used by the operating system, making it difficult for users or security tools to detect the malicious DLL. Once the malicious DLL is loaded, the attacker can execute arbitrary code, escalate privileges, or perform other malicious actions [19]. The consequences of loading a malicious DLL can be severe. Once the attacker has successfully executed arbitrary code or escalated privileges, they can gain control over the system and potentially compromise sensitive data. Furthermore, the presence of a malicious DLL can go undetected for an extended period, allowing the attacker to maintain persistence and continue their malicious activities undetected. It underscores the importance of robust security measures and regular vulnerability assessments to identify and mitigate such risks

2.4 File Inclusion

Malware file inclusion, also known as remote file inclusion (RFI), is a common type of malware-shared code execution used to exploit vulnerabilities in web applications [20]. It involves an attacker injecting malicious code into a web application, allowing them to execute arbitrary files on the server remotely. Malware file inclusion attacks often target websites that rely on user input to dynamically include files, such as those using PHP's include() or require() functions. By exploiting insecure coding practices or inadequate input validation, hackers can trick the application into including malicious files from a remote server.

For instance, a hacker may exploit a vulnerability in a website's file upload feature by injecting malicious code disguised as an image file. When the file is uploaded and accessed by the server, the injected code is executed, granting the attacker unauthorised access to sensitive data or allowing them to distribute malware to unsuspecting users. However, modern web application security measures such as input validation and file type checking can mitigate this vulnerability. Additionally, utilising server-side file handling techniques and sandboxed execution environments can further enhance the protection against remote code execution attacks by malicious threat agents.

Identify applicable funding agency here. If none, delete this text box.

3. RELATED WORKS

3.1 *Code Similarities & Functionalities*

Code similarities and shared functionalities are common in software development. These similarities can arise from using similar programming languages, frameworks, or design patterns. Shared functionalities refer to the features or capabilities that multiple pieces of code have in common. These similarities and shared functionalities can help developers save time and effort by reusing code components and leveraging existing solutions. One of the most fascinating aspects of studying malware is uncovering code similarities and shared functionalities among different strains. Analysis of code similarities and shared functionalities between different malware samples is essential to identify common attack patterns. These similarities can range from identical code snippets to similar evasion, propagation, or payload delivery techniques.

Dolnak [21] proposed a stochastical-aware code similarity extraction method. The method uses machine learning algorithms and probabilistic models to analyse the similarities between different malware code samples. The proposed method captures the inherent randomness and variability in malware code by incorporating stochastic elements into the analysis. The stochastic method considers code structure, syntax, and functionality, providing a holistic view of the similarities between different malware samples. However, a detailed counterexample of this approach could be a scenario where two malware samples have completely different code structures, syntax, and functionalities yet still pose the same threat level. In such cases, relying solely on stochastic methods may lead to misclassification or underestimation of the potential harm these malware threats pose.

Malware attack patterns refer to the specific techniques and strategies employed by malicious actors to infiltrate systems and compromise their security. These patterns can vary greatly in complexity and severity, ranging from simple phishing emails to sophisticated zero-day exploits. Gazzan and Sheldon [22] provided an overview of attack patterns ransomware deploys to attack industrial control systems (ICS) and the significant impacts. Industrial control systems are particularly vulnerable to ransomware attacks due to their interconnected nature and reliance on computer networks for operation and monitoring. Schmidbauer et al. [23] put forward a technique to illustrate how sensitive data is extracted by malware code snippets using a codebook approach. The codebook approach is malware's technique to extract sensitive data from a target system. This method involves using pre-defined code snippets or templates designed to search for and collect specific types of information from the target.

Moses and Sarah [24] compared different malware evolution and propagation strategies based on the malware variant's functionality and evasion mechanism. The study analysed various factors, such as the malware's ability to exploit software vulnerabilities, social engineering techniques, and reliance on botnets for propagation. Additionally, the researchers examined how these strategies evolved and their impact on the overall effectiveness of the malware. For example, the study found that malware variants utilising advanced social engineering techniques were more successful in tricking users into downloading malicious files or clicking on infected links. These variants are often disguised as legitimate software updates or enticing offers, increasing their chances of infiltrating systems [25].

Additionally, the study observed a shift in propagation strategies from relying solely on software vulnerabilities to leveraging botnets for faster and more widespread infection. However, a detailed counterexample to this observation can be seen in the case of phishing attacks. Despite not relying on advanced social engineering techniques, phishing emails have been incredibly successful in tricking users into divulging sensitive information or visiting malicious websites. These emails often pose as legitimate entities like banks or online services, preying on users' trust and urgency to act quickly.

3.2 COMMON CODE PATTERNS

Pattern refers to the similar structures and techniques used to develop and distribute malicious software. Common code pattern often found in malware is obfuscation techniques. By understanding this code pattern and shared components among malware samples, cybersecurity professionals can identify common sources and potential connections among different malware strains. Understanding these shared components can help the attribution process, as certain code patterns or components may be associated with specific threat actors or groups.

Choo and Dehghantanha [26] introduce a machine learning classification technique using consensus clustering to map the APT campaigns to their procedures which often vary and evolve. The proposed approach involved first clustering similar APT campaigns together using consensus clustering. This approach allowed for the identification of commonalities and patterns among different campaigns. For example, the proposed machine learning technique could be applied in a cybersecurity context to identify and track different APT campaigns targeting a specific industry. Cybersecurity analysts can uncover shared tactics, techniques, and procedures (TTPs) attackers use by clustering similar campaigns together. This knowledge can create robust defense strategies and improve real-time incident response capabilities against evolving APT threats. However, a counterexample to this approach is if the machine learning technique fails to cluster similar campaigns accurately. Cybersecurity analysts may mistakenly identify unrelated campaigns as part of the same APT group, leading to incorrect conclusions about shared TTPs, which can result in ineffective defence strategies and potentially leave the targeted industry vulnerable to evolving APT threats.

4. METHODOLOGY

4.1 *Testbed*

Using the V2 Cloud emulator, we created a testbed on Netlab. Netlab was used as a sandbox environment to analyse the malware sample safely. The choice to use V2 Cloud instead of other emulators such as Qemu [27] is that V2 Cloud is an integrated Desktop-as-a-Service (DaaS) with better speed and simplicity. It also supports more platforms like Saas/Web than the Qemu emulator. Though predominant malware is designed for Windows, which makes many analysts focus on using Intel x86 architecture for analysis, V2 Cloud emulation has these features all embedded with faster speed [28]. The testbed on the Netlab contains three virtual machines, as illustrated in Fig.1. Malware samples were then loaded into the V2 Cloud emulator and then copied into the shared folder of Windows 10 C: drive (with the IP address ending with .130). Copying the sample to these two locations enable Nmap scan and easy access from "My Network Places" using the listen and accept socket function to analyse the processes, directories, registry entries, enumeration functions, system calls, and network interactions that were invoked by the malware.hap

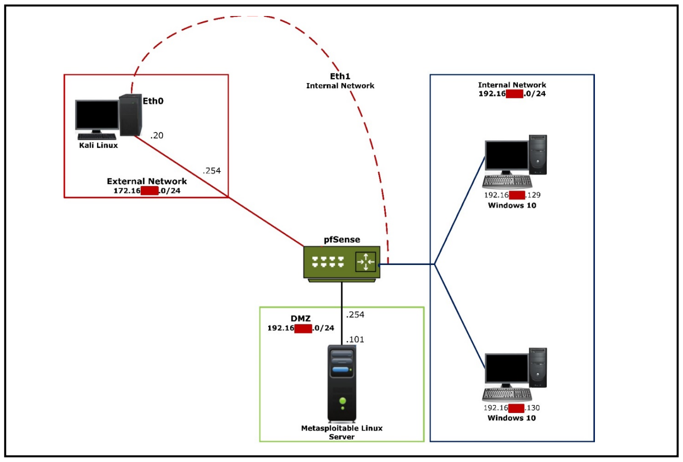
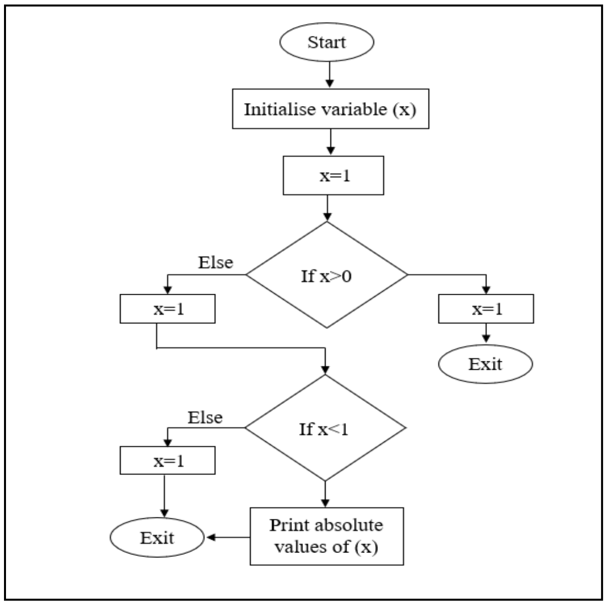


Fig. 1. Testbed environment for testing and analysis.

The sample was then executed in Windows 10 (IP address ending with .129), while Kali and Windows 10 (IP address ending with .130) were used to monitor the sample characteristics. pfSense, an open-source firewall and router platform, provides a robust network security solution that creates secure network environments. Its advanced features, such as traffic shaping, VPN support, and intrusion detection, make it an invaluable tool for simulating real-world network scenarios and testing the effectiveness of malware detection and prevention mechanisms. We intentionally added the Metasploitable Linux server to provide a controlled environment and to allow us to understand the behaviour and impact of different malware strains.

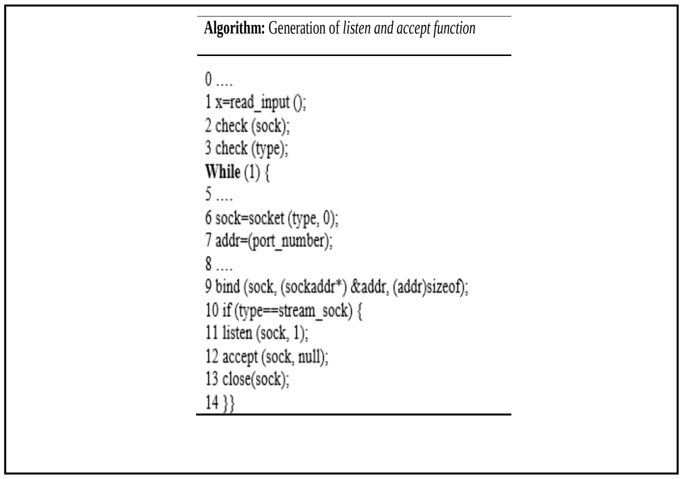


4.2 Paths Execution and Feature Generation

In this paper, the generation of malicious feature vectors using binary attributes from algorithm 1 of the existing framework [29] was used to attribute each malicious instruction the emulated processor executed to the guest system kernel. This technique allows all the interprocess communication and function calls from the emulated devices to be copied to the host system to examine different execution paths of the malicious program. As summarised in Fig. 2, the binary attributes used by our approach consist of 0s and 1s, with x representing the sample feature variable and input.

Fig. 2. Paths exploration flowchart for shared code execution. The exploration flowchart for shared code execution is a valuable tool for developers to navigate the various paths and options available when working with shared code. For example, if the value of x = 0 during the system runs, the system compares the value with a 1 and determines if many explorations can be performed. This process continues because x has to be greater than 0 to satisfy the condition. Anytime a value of 0 or less than 1 is returned, the system assumes other processes and states that are not visited. This process continues until the check thrives. The analysis process is terminated as soon as the check succeeds, and the output is generated when the feature variable is 1. This makes it possible to trace at every point of the system execution of all the functions and instructions that transfer control to a specific address of the value in the accumulator that the malware could use to create a loop and socket, which is a network function in Windows found in WSOCK32 library as summarise in (see algorithm).In the flowcharet, Line 1: The code reads input from the user and assigns it to a variable called "X ". Line 2: The code calls a function called "check" and passes a variable called "sock" as an argument. Line 3: The code calls the =check" function again, this time passing a variable called "type" as an argument. Line 4: The code starts a while loop that will run indefinitely. Line 6: The code creates a socket using the "socket" function. The "type" argument specifies the type of socket to create if the argument is not "0".

Line 7: The code creates an address structure using the "port_number" variable to be used in the address structure of the socket. Line 9: The code binds the socket to the address structure using the "bind" function with the help of the "sockaddr" argument which is a data type used to represent socket addresses. The "sizeof" argument is used to specify the size of the address structure. Line 10: The code checks if the socket type is a stream socket. Line 11: If the socket type is a stream socket, the code calls the "listen" function to listen for incoming connections on the socket with the "1" argument identifying connections that are yet to be established. Line 12: Code calls "accept" function to accept an incoming connection on the socket or reject it with the "null" argument. Line 13: the code closes the socket Line 14: the code ends the if statement and the while loop.

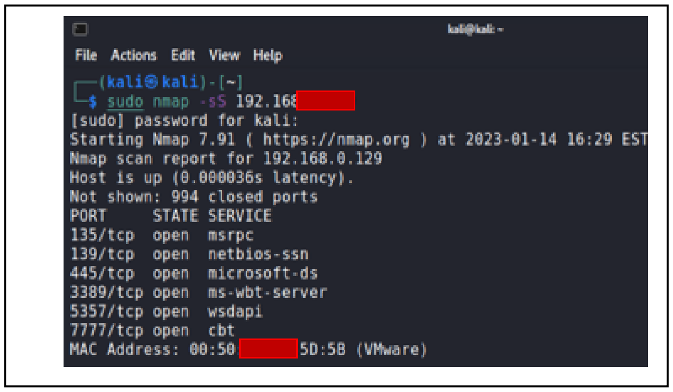


Algorithm: Generation of *listen and accept function*

```
0 ....
1 x=read_input ();
2 check (sock);
3 check (type);
While (1) {
5 ....
6 sock=socket (type, 0);
7 addr=(port_number);
8 ....
9 bind (sock, (sockaddr*) &addr, (addr)sizeof);
10 if (type==stream_sock) {
11 listen (sock, 1);
12 accept (sock, null);
13 close(sock);
14 }}
```

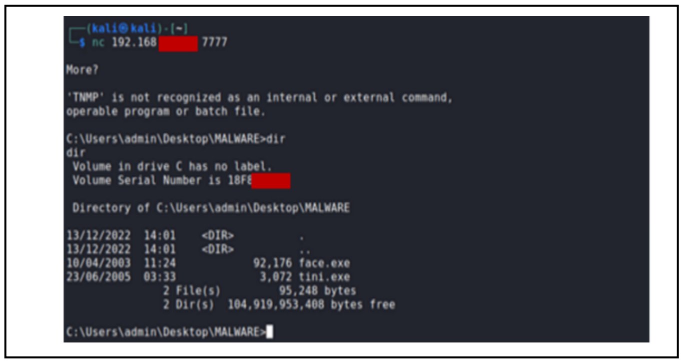
4.2 Connection Function

As illustrated in Fig 3, using Network Mapper (Nmap) [30] on Kali, Windows 10 was scanned to identify if the sample has established a connection and is listening to any available open ports on the victim's machine using the sock function. Using Netcat on Kali, remote and command line access were found to be established by the sample using port number 7777, revealing that the sample bypassed the Windows firewall and established a covert communication channel with the attacker's machine. This finding raised concerns about the potential for data exfiltration and unauthorised control of the victim's system. The use of port 7777 indicated a deliberate choice by the attacker to hide their activities within a commonly used port, making it harder to detect and block the malicious traffic.



```
┌──(kali㉿kali)-[~]
└─$ sudo nmap -sS 192.168[redacted]
[sudo] password for kali:
Starting Nmap 7.91 ( https://nmap.org ) at 2023-01-14 16:29 EST
Nmap scan report for 192.168.0.129
Host is up (0.000036s latency).
Not shown: 994 closed ports
PORT     STATE SERVICE
135/tcp  open  msrpc
139/tcp  open  netbios-ssn
445/tcp  open  microsoft-ds
3389/tcp open  ms-wbt-server
5357/tcp open  wsdapi
7777/tcp open  cbt
MAC Address: 00:50[redacted]5D:5B (VMware)
```

Fig. 3a Nmap scan



```
┌──(kali㉿kali)-[~]
└─$ nc 192.168[redacted] 7777

More?

'TNMP' is not recognized as an internal or external command,
operable program or batch file.

C:\Users\admin\Desktop\MALWARE>dir
dir
 Volume in drive C has no label.
 Volume Serial Number is 18F8[redacted]

 Directory of C:\Users\admin\Desktop\MALWARE

13/12/2022  14:01    <DIR>          .
13/12/2022  14:01    <DIR>          ..
10/04/2003  11:24            92,176 face.exe
23/06/2005  03:33             3,072 tini.exe
               2 File(s)         95,248 bytes
               2 Dir(s)  104,919,953,408 bytes free

C:\Users\admin\Desktop\MALWARE>
```

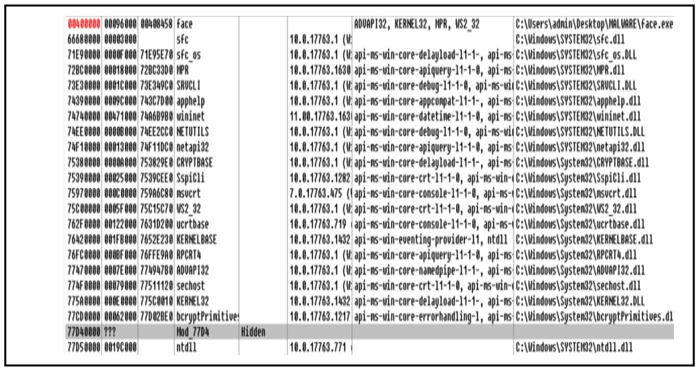
Fig. 3b Netcat Command Line Access

Fig. 3 Sock function connection using Nmap to establish a connection with a specific port on a target host. This gathered valuable vulnerabilities about the open ports, protocols, and services running on that host.

5. Results and Discussion

5.1 Executable Modules

Executable modules are a critical component of any malicious software. It is responsible for carrying out the malicious activities that the attacker intends. This module is often designed to be covert and evasive, making it challenging for security measures to detect and mitigate its impact. During the analysis, it was found that the malicious payload was executed by loading shared modules. For example, the dynamic link libraries (DLL) were loaded by the Windows OS loader directly from the local paths of the infected device conventions network location. This pattern of module loading is similar to the technique highlighted by Alsaheel and Pande [31]. The first time the malware was executed, it used an executable module named Mod_77D4 (See Fig. 4). This module was designed to exploit a specific vulnerability in the target system, allowing the malware to gain unauthorised access. Once inside, the malware quickly spread its roots, infecting various files and directories, making it

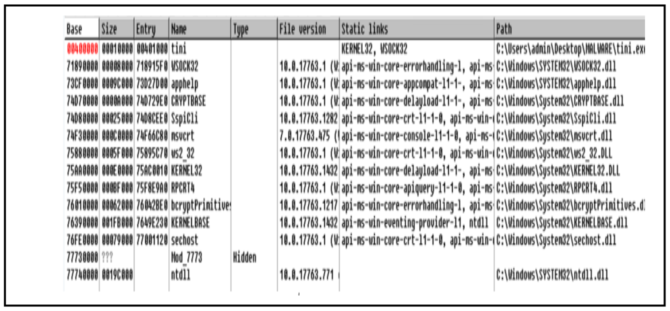


00400000	00096000	00408458	face			ADVAPI32, KERNEL32, MPR, WS2_32	C:\Users\admin\Desktop\MALWARE\face.exe
66680000	00003000		sfc		10.0.17763.1 (W		C:\Windows\SYSTEM32\sfc.dll
71E90000	0000F000	71E95E70	sfc_os		10.0.17763.1 (W	api-ms-win-core-delayload-l1-1-, api-ms	C:\Windows\SYSTEM32\sfc_os.DLL
72BC0000	00018000	72BC33D0	MPR		10.0.17763.1630	api-ms-win-core-apiquery-l1-1-0, api-ms	C:\Windows\SYSTEM32\MPR.dll
73E30000	0001C000	73E349C0	SRVCLI		10.0.17763.1 (W	api-ms-win-core-debug-l1-1-0, api-ms-wi	C:\Windows\SYSTEM32\SRVCLI.DLL
74390000	0009C000	743C7D00	apphelp		10.0.17763.1 (W	api-ms-win-core-appcompat-l1-1-, api-ms	C:\Windows\SYSTEM32\apphelp.dll
74740000	00471000	74A6B9B0	wininet		11.00.17763.163	api-ms-win-core-datetime-l1-1-0, api-ms	C:\Windows\SYSTEM32\wininet.dll
74EE0000	0000B000	74EE2CC0	NETUTILS		10.0.17763.1 (W	api-ms-win-core-debug-l1-1-0, api-ms-wi	C:\Windows\SYSTEM32\NETUTILS.DLL
74F10000	00013000	74F11DC0	netapi32		10.0.17763.1 (W	api-ms-win-core-apiquery-l1-1-0, api-ms	C:\Windows\SYSTEM32\netapi32.dll
75380000	0000A000	753829E0	CRYPTBASE		10.0.17763.1 (W	api-ms-win-core-delayload-l1-1-, api-ms	C:\Windows\System32\CRYPTBASE.dll
75390000	00025000	7539CEE0	SspiCli		10.0.17763.1282	api-ms-win-core-crt-l1-1-0, api-ms-win-	C:\Windows\System32\SspiCli.dll
75970000	000C0000	759A6C80	msvcrt		7.0.17763.475 (	api-ms-win-core-console-l1-1-0, api-ms-	C:\Windows\System32\msvcrt.dll
75C00000	0005F000	75C15C70	WS2_32		10.0.17763.1 (W	api-ms-win-core-crt-l1-1-0, api-ms-win-	C:\Windows\System32\WS2_32.dll
762F0000	00122000	7631D200	ucrtbase		10.0.17763.719	api-ms-win-core-console-l1-1-0, api-ms-	C:\Windows\System32\ucrtbase.dll
76420000	001FB000	7652E230	KERNELBASE		10.0.17763.1432	api-ms-win-eventing-provider-l1, ntdll	C:\Windows\System32\KERNELBASE.dll
76FC0000	000BF000	76FFE9A0	RPCRT4		10.0.17763.1 (W	api-ms-win-core-apiquery-l1-1-0, api-ms	C:\Windows\System32\RPCRT4.dll
77470000	0007E000	774947B0	ADVAPI32		10.0.17763.1 (W	api-ms-win-core-namedpipe-l1-1-, api-ms	C:\Windows\System32\ADVAPI32.dll
774F0000	00079000	77511120	sechost		10.0.17763.1 (W	api-ms-win-core-crt-l1-1-0, api-ms-win-	C:\Windows\System32\sechost.dll
775A0000	000E0000	775C0010	KERNEL32		10.0.17763.1432	api-ms-win-core-delayload-l1-1-, api-ms	C:\Windows\System32\KERNEL32.DLL
77CD0000	00062000	77D02BE0	bcryptPrimitives		10.0.17763.1217	api-ms-win-core-errorhandling-1, api-ms	C:\Windows\System32\bcryptPrimitives.dll
77D40000	???		Mod_77D4	Hidden			
77D50000	0019C000		ntdll		10.0.17763.771		C:\Windows\SYSTEM32\ntdll.dll

extremely difficult to detect and remove. The Mod_77D4 module found in NTDLL.DLL could

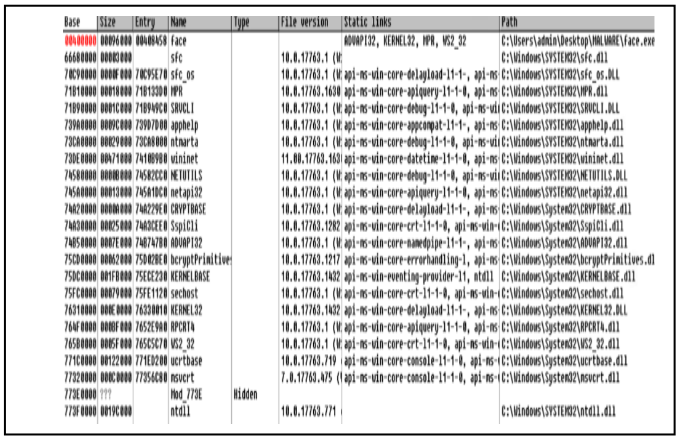
Fig. 4 Executable Mod_77D4 Module

NTDLL.DLL being an essential component of Windows native API, the malware uses it to deliver the executables on the socket address of the Windows victim machine (as specified in line 9 of the socket function). The LoadLibrary allows the malware to load a DLL from a specified path or UNC location. Malware authors frequently use this method to avoid detection and get around security measures. Apart from the base information, the malware completely hid every other detail such as the size, entry point, type, static links and path. This level of sophistication in concealing crucial information made it incredibly challenging to analyse and understand the inner workings of the malware. The creators had meticulously crafted the code to ensure that any attempts to dissect it would be met with a labyrinth of obfuscation with an intricate web of nested functions and constantly changing variable names. The analysis of the executable indicated that the malware-shared module was loaded in stealth mode. The malware changed the Mod_77D4 module to three additional Mod names after every execution, Mod_7773, Mod_773E, and Mod_7712 (see Fig. 5: 5a, 5b, and 5c). This tactic is to avoid detection and hinder reverse engineering attempts.



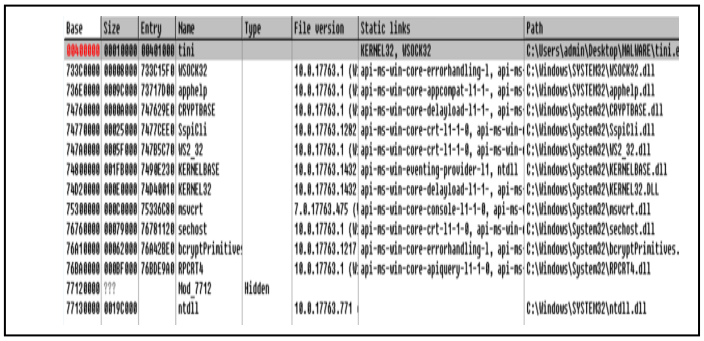
Base	Size	Entry	Name	Type	File version	Static links	Path
00400000	00010000	00401000	tini			KERNEL32, WSOCK32	C:\Users\admin\Desktop\MALWARE\tini.exe
71890000	00008000	718915F0	WSOCK32		10.0.17763.1 (W	api-ms-win-core-errorhandling-l, api-ms	C:\Windows\SYSTEM32\WSOCK32.dll
73CF0000	0009C000	73D27D00	apphelp		10.0.17763.1 (W	api-ms-win-core-appcompat-l1-1-, api-ms	C:\Windows\SYSTEM32\apphelp.dll
74D70000	0000A000	74D729E0	CRYPTBASE		10.0.17763.1 (W	api-ms-win-core-delayload-l1-1-, api-ms	C:\Windows\System32\CRYPTBASE.dll
74D80000	00025000	74D8CEE0	SspiCli		10.0.17763.1282	api-ms-win-core-crt-l1-1-0, api-ms-win-(	C:\Windows\System32\SspiCli.dll
74F30000	000C0000	74F66C80	msvcrt		7.0.17763.475 ((	api-ms-win-core-console-l1-1-0, api-ms-(	C:\Windows\System32\msvcrt.dll
75880000	0005F000	75895C70	ws2_32		10.0.17763.1 (W	api-ms-win-core-crt-l1-1-0, api-ms-win-(	C:\Windows\System32\ws2_32.DLL
75AA0000	000E0000	75AC0010	KERNEL32		10.0.17763.1432	api-ms-win-core-delayload-l1-1-, api-ms	C:\Windows\System32\KERNEL32.DLL
75F50000	000BF000	75F8E9A0	RPCRT4		10.0.17763.1 (W	api-ms-win-core-apiquery-l1-1-0, api-ms	C:\Windows\System32\RPCRT4.dll
76010000	00062000	76042BE0	bcryptPrimitives		10.0.17763.1217	api-ms-win-core-errorhandling-l, api-ms	C:\Windows\System32\bcryptPrimitives.dl
76390000	001FB000	7649E230	KERNELBASE		10.0.17763.1432	api-ms-win-eventing-provider-l1, ntdll	C:\Windows\System32\KERNELBASE.dll
76FE0000	00079000	77001120	sechost		10.0.17763.1 (W	api-ms-win-core-crt-l1-1-0, api-ms-win-(	C:\Windows\System32\sechost.dll
77730000	???		Mod_7773	Hidden			
77740000	0019C000		ntdll		10.0.17763.771 (		C:\Windows\SYSTEM32\ntdll.dll

Fig. 5a. Shared executable Mod_7773 Module



Base	Size	Entry	Name	Type	File version	Static links	Path
00400000	00096000	00408458	Face			ADVAPI32, KERNEL32, MPR, WS2_32	C:\Users\admin\Desktop\MALWARE\face.exe
66680000	00003000		sfc		10.0.17763.1 (W		C:\Windows\SYSTEM32\sfc.dll
70C90000	0000F000	70C95E70	sfc_os		10.0.17763.1 (W	api-ms-win-core-delayload-l1-1-, api-ms	C:\Windows\SYSTEM32\sfc_os.DLL
71B10000	00018000	71B133D0	MPR		10.0.17763.1630	api-ms-win-core-apiquery-l1-1-0, api-ms	C:\Windows\SYSTEM32\MPR.dll
71B90000	0001C000	71B949C0	SRVCLI		10.0.17763.1 (W	api-ms-win-core-debug-l1-1-0, api-ms-wi	C:\Windows\SYSTEM32\SRVCLI.DLL
739A0000	0009C000	739D7D00	apphelp		10.0.17763.1 (W	api-ms-win-core-appcompat-l1-1-, api-ms	C:\Windows\SYSTEM32\apphelp.dll
73CA0000	00029000	73CA8000	ntmarta		10.0.17763.1 (W	api-ms-win-core-debug-l1-1-0, api-ms-wi	C:\Windows\SYSTEM32\ntmarta.dll
73DE0000	00471000	7410B9B0	wininet		11.00.17763.163	api-ms-win-core-datetime-l1-1-0, api-ms	C:\Windows\SYSTEM32\wininet.dll
74580000	0000B000	74582CC0	NETUTILS		10.0.17763.1 (W	api-ms-win-core-debug-l1-1-0, api-ms-wi	C:\Windows\SYSTEM32\NETUTILS.DLL
745A0000	00013000	745A1DC0	netapi32		10.0.17763.1 (W	api-ms-win-core-apiquery-l1-1-0, api-ms	C:\Windows\SYSTEM32\netapi32.dll
74A20000	0000A000	74A229E0	CRYPTBASE		10.0.17763.1 (W	api-ms-win-core-delayload-l1-1-, api-ms	C:\Windows\System32\CRYPTBASE.dll
74A30000	00025000	74A3CEE0	SspiCli		10.0.17763.1282	api-ms-win-core-crt-l1-1-0, api-ms-win-(	C:\Windows\System32\SspiCli.dll
74B50000	0007E000	74B747B0	ADVAPI32		10.0.17763.1 (W	api-ms-win-core-namedpipe-l1-1-, api-ms	C:\Windows\System32\ADVAPI32.dll
75CD0000	00062000	75D02BE0	bcryptPrimitives		10.0.17763.1217	api-ms-win-core-errorhandling-l, api-ms	C:\Windows\System32\bcryptPrimitives.dl
75DC0000	001FB000	75ECE230	KERNELBASE		10.0.17763.1432	api-ms-win-eventing-provider-l1, ntdll	C:\Windows\System32\KERNELBASE.dll
75FC0000	00079000	75FE1120	sechost		10.0.17763.1 (W	api-ms-win-core-crt-l1-1-0, api-ms-win-(	C:\Windows\System32\sechost.dll
76310000	000E0000	76330010	KERNEL32		10.0.17763.1432	api-ms-win-core-delayload-l1-1-, api-ms	C:\Windows\System32\KERNEL32.DLL
764F0000	000BF000	7652E9A0	RPCRT4		10.0.17763.1 (W	api-ms-win-core-apiquery-l1-1-0, api-ms	C:\Windows\System32\RPCRT4.dll
765B0000	0005F000	765C5C70	WS2_32		10.0.17763.1 (W	api-ms-win-core-crt-l1-1-0, api-ms-win-(	C:\Windows\System32\WS2_32.dll
771C0000	00122000	771ED200	ucrtbase		10.0.17763.719 (	api-ms-win-core-console-l1-1-0, api-ms-(	C:\Windows\System32\ucrtbase.dll
77320000	000C0000	77356C80	msvcrt		7.0.17763.475 ((	api-ms-win-core-console-l1-1-0, api-ms-(	C:\Windows\System32\msvcrt.dll
773E0000	???		Mod_773E	Hidden			
773F0000	0019C000		ntdll		10.0.17763.771 (		C:\Windows\SYSTEM32\ntdll.dll

Fig. 5b. Shared Executable Mod_773E Module

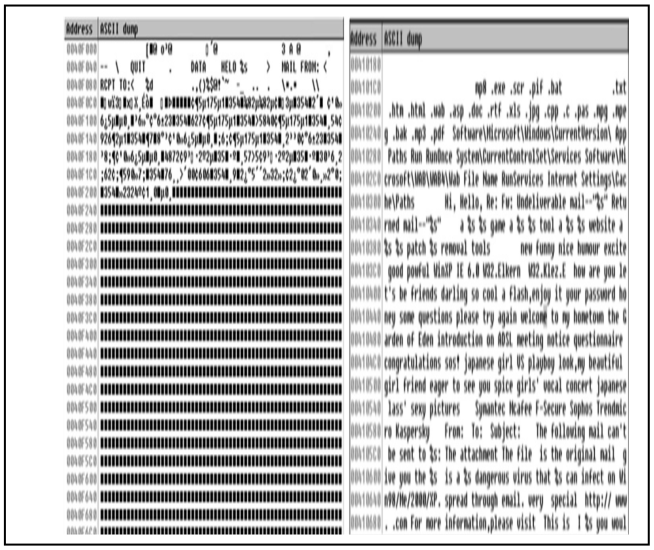


Base	Size	Entry	Name	Type	File version	Static links	Path
00400000	00010000	00401000	tini			KERNEL32, WSOCK32	C:\Users\admin\Desktop\MALWARE\tini.e
733C0000	00008000	733C15F0	WSOCK32		10.0.17763.1 (W	api-ms-win-core-errorhandling-l, api-ms	C:\Windows\SYSTEM32\WSOCK32.dll
736E0000	0009C000	73717D00	apphelp		10.0.17763.1 (W	api-ms-win-core-appcompat-l1-1-, api-ms	C:\Windows\SYSTEM32\apphelp.dll
74760000	0000A000	747629E0	CRYPTBASE		10.0.17763.1 (W	api-ms-win-core-delayload-l1-1-, api-ms	C:\Windows\System32\CRYPTBASE.dll
74770000	00025000	7477CEE0	SspiCli		10.0.17763.1282	api-ms-win-core-crt-l1-1-0, api-ms-win-(	C:\Windows\System32\SspiCli.dll
747A0000	0005F000	747B5C70	WS2_32		10.0.17763.1 (W	api-ms-win-core-crt-l1-1-0, api-ms-win-(	C:\Windows\System32\WS2_32.dll
74800000	001FB000	7490E230	KERNELBASE		10.0.17763.1432	api-ms-win-eventing-provider-l1, ntdll	C:\Windows\System32\KERNELBASE.dll
74D20000	000E0000	74D40010	KERNEL32		10.0.17763.1432	api-ms-win-core-delayload-l1-1-, api-ms	C:\Windows\System32\KERNEL32.DLL
75300000	000C0000	75336C80	msvcrt		7.0.17763.475 ((	api-ms-win-core-console-l1-1-0, api-ms-(	C:\Windows\System32\msvcrt.dll
76760000	00079000	76781120	sechost		10.0.17763.1 (W	api-ms-win-core-crt-l1-1-0, api-ms-win-(	C:\Windows\System32\sechost.dll
76A10000	00062000	76A42BE0	bcryptPrimitives		10.0.17763.1217	api-ms-win-core-errorhandling-l, api-ms	C:\Windows\System32\bcryptPrimitives.
76BA0000	000BF000	76BDE9A0	RPCRT4		10.0.17763.1 (W	api-ms-win-core-apiquery-l1-1-0, api-ms	C:\Windows\System32\RPCRT4.dll
77120000	???		Mod_7712	Hidden			
77130000	0019C000		ntdll		10.0.17763.771 (		C:\Windows\SYSTEM32\ntdll.dll

Fig. 5c. Shared Executable Mod_7712 Module

5.2 Data Obfuscation

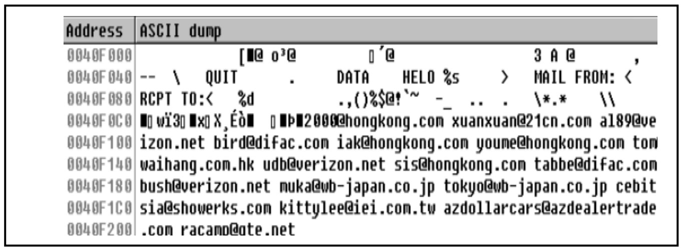
Data obfuscation is another shared code technique that the malware uses to obfuscate the data section of the executable (see Fig. 6a). The malware uses the XOR encoding method to transform the data in such a way that it appears as random noise to anyone trying to inspect it. Thus, hiding data within other seemingly harmless sections of the executable. However, by carefully studying the encryption algorithms and patterns employed by the malware, we successfully decode the seemingly random noise and retrieve meaningful information, which is a list of emails and contents (see Fig. 6b and 6c).



```
Address   ASCII dump
0040F000            [▮@ o'@      ▯´@              3 A @     ,
0040F040  -- \  QUIT    .    DATA   HELO %s    >   MAIL FROM: <
0040F080  RCPT TO:<  %d          .,()%$@!`~  -_  ..  .   \*.*   \\
0040F0C0  ▮▯ wï3▯ ▮x▯ X¸Éò▮  ▯▮Þ▮▮▮▮▮¢¶5µ175µ1▮354▮¼82µ¼82µ¢▮▯3µ▮354▮2´▮ ¢'0»
0040F100  6¿5µ▮µ0¸▮'6»°¢°6±23▮354▮627¢¶5µ175µ1▮354▮>5840¢¶5µ175µ1▮354▮¸54¢
0040F140  926¶2µ1▮354▮¶7▮8°'¢'0»6¿5µ▮µ0¸▮;6;¢¶5µ175µ1▮354▮¸2''0¢°6±23▮354▮
0040F180  '8;¶¢'0»6¿5µ▮µ0¸▮4872¢9'▯·2º2µ▮35▮·º▮¸57>5¢9'▯·2º2µ▮35▮·º▮30'6¸2
0040F1C0  ;62¢;¶590»7;▮354▮76¸¸>´00¢606▮354▮¸9▮2¿°5´´2»32»;¢2¿°02´0»¸»2°0;
0040F200  ▮354▮»2324º¢1¸0▮µ0¸▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮
0040F240  ▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮
0040F280  ▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮
0040F2C0  ▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮
0040F300  ▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮
0040F340  ▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮
0040F380  ▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮
0040F3C0  ▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮
0040F400  ▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮
0040F440  ▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮
0040F480  ▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮
0040F4C0  ▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮
0040F500  ▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮
0040F540  ▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮
0040F580  ▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮
0040F5C0  ▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮
0040F600  ▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮
0040F640  ▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮
0040F680  ▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮
0040F6C0  ▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮
```

```
Address   ASCII dump
00410180
004101C0                      mp8 .exe .scr .pif .bat          .txt
00410200   .htm .html .wab .asp .doc .rtf .xls .jpg .cpp .c .pas .mpg .mpe
00410240  g .bak .mp3 .pdf  Software\Microsoft\Windows\CurrentVersion\ App
00410280   Paths Run RunOnce System\CurrentControlSet\Services Software\Mi
004102C0  crosoft\WAB\WAB4\Wab File Name RunServices Internet Settings\Cac
00410300  he\Paths     Hi, Hello, Re: Fw: Undeliverable mail--"%s" Retu
00410340  rned mail--"%s"    a %s %s game a %s %s tool a %s %s website a
00410380  %s %s patch %s removal tools       new funny nice humour excite
004103C0   good powful WinXP IE 6.0 W32.Elkern  W32.Klez.E  how are you le
00410400  t's be friends darling so cool a flash,enjoy it your password ho
00410440  ney some questions please try again welcome to my hometown the G
00410480  arden of Eden introduction on ADSL meeting notice questionnaire
004104C0  congratulations sos! japanese girl VS playboy look,my beautiful
00410500  girl friend eager to see you spice girls' vocal concert japanese
00410540   lass' sexy pictures   Symantec Mcafee F-Secure Sophos Trendmic
00410580  ro Kaspersky   From:  To:  Subject:   The following mail can't
004105C0   be sent to %s: The attachment The file  is the original mail  g
00410600  ive you the %s  is a %s dangerous virus that %s can infect on Wi
00410640  n98/Me/2000/XP. spread through email. very  special  http:// www
00410680  . .com For more information,please visit  This is  1 %s you woul
```

Fig. 6a. Obfuscated data **Fig. 6b.** Decrypted data



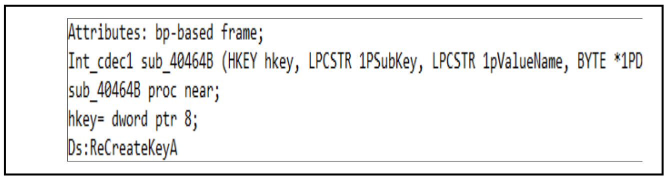
```
Address   ASCII dump
0040F000            [▮@ o³@        ▯´@              3 A @      ,
0040F040  --  \   QUIT      .     DATA    HELO %s     >   MAIL FROM: <
0040F080  RCPT TO:<   %d          .,()%$@!`~  -_  ..  .   \*.*    \\
0040F0C0  ▮▯ wï3▯ ▮x▯ X¸Éò▮   ▯▮Þ▮2000@hongkong.com xuanxuan@21cn.com al89@ve
0040F100  izon.net bird@difac.com iak@hongkong.com youme@hongkong.com tom
0040F140  waihang.com.hk udb@verizon.net sis@hongkong.com tabbe@difac.com
0040F180  bush@verizon.net muka@wb-japan.co.jp tokyo@wb-japan.co.jp cebit
0040F1C0  sia@showerks.com kittylee@iei.com.tw azdollarcars@azdealertrade
0040F200  .com racamp@gte.net
```

Fig. 6c. Data Information Containing Emails

Even though data obfuscation may initially hide data from casual inspection, it is not foolproof and can be overcome with sufficient knowledge and expertise. The results indicate that the variant is sophisticated and can spread through email attachments and network shares, creating a chain reaction of infections.

5.3 XOR Encoding

The sample employs the XOR encoding reversible cypher technique to obfuscate and share its executable modules (see Fig. 7.). XOR encoding works by applying an XOR operation to each payload byte with a specific key, making it appear as random data. For example, malware may use XOR encoding to obfuscate its malicious code in network communication. Before sending it over the network, the malware will apply an XOR operation to each payload byte with a pre-defined key. This obfuscation technique makes it challenging to detect the true nature of the malicious payload, as the XOR-encoded data appears as random noise. To successfully decipher the payload, the recipient must possess the same key used for encoding. Without the correct key, the payload remains unintelligible, adding an extra layer of security for the malware. Moreover, the malware can bypass traditional detection methods that rely on pattern matching or signature-based analysis by employing XOR encoding during network communication. However, when access to the hkey is available, the XOR encoding technique becomes ineffective.



```
Attributes: bp-based frame;
Int_cdec1 sub_40464B (HKEY hkey, LPCSTR 1PSubKey, LPCSTR 1pValueName, BYTE *1PD
sub_40464B proc near;
hkey= dword ptr 8;
Ds:ReCreateKeyA
```

Fig. 7 XOR Encoding Function

Sub_40464B is a procedure that takes in attributes such as a base pointer (bp) based frame. This procedure uses the input parameters HKEY hkey, LPCSTR IPSubKey, LPCSTR 1pValueName, and BYTE *1PData. The code within the procedure includes a declaration for the variable hkey and a specific line that invokes the Sub_407760 function with the XOR operation of 0FFFFFFFFh as its argument. Additionally, the line "Ds: ReCreateKey A" suggests that the malware may have a call to recreate a registry key to hide its activities and evade detection. The malware then made a call to RegCloseKey to close the handle. The function loops through all the Windows' API structures to zero out all the prologue functions for each associated API structure configuration.

All the XOR occurrences were searched during the analysis to determine where the malware cleared registers. When an interesting branch of the loop is located, instructions are filters to identify the XOR instructions, usually having a register and a constant. This approach is effective in identifying encoding functions used by malware. Existing research [32] shows that the call flow where the instructions originate has to be analysed to identify different loops, which is another indicator that the program uses XOR encoding using memory snapshots as evidence for malware obfuscation. The call flow is insufficient to identify encryption and obfuscation of malware activities. For instance, if the loop instructions used by a malicious application are jz and jnz, it means that if the zero flags are cleared, the malware can jump to a specific location, and the loop will continue without termination.

5.4 Limitations

Even though the framework demonstrated efficacy in analysing different executables, it may have compatibility issues when tested with samples from different platforms, such as Android and iOS. Also, some sophisticated malware variants may avoid common code patterns, which our framework may result in a false negative to identify their presence. To overcome this limitation in the future, we would consider incorporating AI models for pattern matching into the framework.

6.0 Conclusions

This paper explores the impact of shared code execution by malicious applications by looking at the interprocess communication and function calls of the code execution paths of the malicious program on Windows-based devices. We utilised Netlab, a comprehensive testing environment, to conduct this analysis, and employed three distinct virtual machines. Additionally, we set up two separate networks to ensure accurate traffic shaping and capture. One of the key findings of this research was the identification of a specific function call that indicates the creation of an address structure using any combination of IP address and port number. This function call was observed in multiple instances of the malicious program, suggesting a common pattern in their behaviour. Further analysis revealed that this address structure was used for establishing communication with external servers, potentially for command and control purposes by changing the executable module at different instances. Based on the results, we conclude that shared code execution analysis can help identify similarities and connections between seemingly unrelated malware samples, leading to a more comprehensive understanding of the malware threat landscape. Our future study will focus on leveraging on Long short-term memory (LSTM) algorithm to implement and optimize the performance of shared code execution to ensure it can handle large-scale analysis of malware samples in a timely fashion

AUTHORS AFFILIATIONS

Dr. Moses Ashawa is affiliated with Glasgow Caledonian
……………. is affliated with the University of Bolton
Dr Jackie riley is affiliated with Glasgow Caledonian University
……………. is affiliated with Glasgow Caledonian University
……………. is affiliated with Glasgow Caledonian Uni

ACKNOWLEDGMENT

Special acknowledgment goes to the Authors for their immense support of this research work.

References

[1] Wazid, M., Zeadally, S., Das, A. K.: Mobile Banking: Evolution and Threats: Malware Threats and Security Solutions. IEEE Consumer Electronics Magazine, 8(2), 56-60 (2019). doi: 10.1109/MCE.2018.2881291.

[2] Ashawa, M., Morris, S.: Analysis of Mobile Malware: A Systematic Review of Evolution and Infection Strategies. Journal of Information Security and Cybercrimes Research, 4(2), 103-131 (2021). doi: 10.26735/krvi8434.

[3] Paul, R.: 2020 11th IEEE Annual Ubiquitous Computing, Electronics & Mobile Communication Conference (UEMCON): 28th-31st October 2020, New York, USA, virtual conference (2020).

[4] Fernández-Álvarez, P., Rodríguez, R. J.: Module extraction and DLL hijacking detection via single or multiple memory dumps. Forensic Science International: Digital Investigation, 44 (2023). doi: 10.1016/j.fsidi.2023.301505.

[5] Ceron, J. M., Margi, C. B., Granville, L. Z.: MARS: An SDN-based malware analysis solution. In Proceedings - IEEE Symposium on Computers and Communications (pp. 525-530) (2016). doi: 10.1109/ISCC.2016.7543792.

[6] Dang, F., et al.: Understanding fileless attacks on linux-based IoT devices with HoneyCloud. In MobiSys 2019 - Proceedings of the 17th Annual International Conference on Mobile Systems, Applications, and Services (pp. 482-493) (2019). doi: 10.1145/3307334.3326083.

[7]

[8] Peng, P., Yang, L., Song, L., Wang, G.: Opening the blackbox of virustotal: Analyzing online phishing scan engines. In Proceedings of the ACM SIGCOMM Internet Measurement Conference, IMC (pp. 478-485) (2019). doi: 10.1145/3355369.3355585.

[9] Stiborek, J., Pevný, T., Rehák, M.: Multiple instance learning for malware classification. Expert Systems with Applications, 93, 346-357 (2018). doi: 10.1016/j.eswa.2017.10.036.

[10] Laurenza, G., Lazzeretti, R., & Mazzotti, L.: Malware triage for early identification of advanced persistent threat activities. Digital Threats: Research and Practice, 1(3) (2020). doi: 10.1145/3386581.

[11] IEEE Communications Society, IEEE Computer Society, IEEE Consumer Electronics Society, and Institute of Electrical and Electronics Engineers, 2016 IEEE 3rd World Forum on Internet of Things ((WF-IoT)?: 12-14 Dec. (2016).

[12] Alenezi, M. N., Alabdulrazzaq, H., Alshaher, A. A., Alkharang, M. M.: Evolution of Malware Threats and Techniques: A Review. International Journal of Communication Networks and Information Security, 12(3), 326-337 (2020). doi: 10.17762/ijcnis.v12i3.4723.

[13] Faruki, P., et al.: Android security: A survey of issues, malware penetration, and defenses. IEEE Communications Surveys and Tutorials, 17(2), 998-1022 (2015). doi: 10.1109/COMST.2014.2386139.

[14] Askarifar, S., Rahman, N.A.A., Osman, H.: A review of latest wannacry ransomware: Actions and preventions. J. Eng. Sci. Technol, 13, pp.24-33. (2018).

[15] IEEE Computer Society, Wireless Systems Security Research Laboratory, and Inc. Trend Micro Devices. Proceedings of the 2012 7th International Conference on Malicious and Unwanted Software: October 16-18, 2012, El Conquistador Hotel & Resort, Fajardo, Puerto Rico, USA (2012).

[16] Qiu, J., et al.: Cyber Code Intelligence for Android Malware Detection. IEEE Trans Cybern, 53(1), 617-627 (2023). doi: 10.1109/TCYB.2022.3164625.

[17] Kasim, Ö.: An ensemble classification-based approach to detect attack level of SQL injections. Journal of Information Security and Applications, 59 (2021). doi: 10.1016/j.jisa.2021.102852.

[18] Popl 12 Conference Committee. Popl 12 Proceedings of the 39th Annual Acm Sigplan-Sigact Symposium on Principles of Programming Languages. Association for Computing Machinery (2012).

[19] Fernández-Álvarez, P., Rodríguez, R. J.: Module extraction and DLL hijacking detection via single or multiple memory dumps. Forensic Science International: Digital Investigation, 44 (2023). doi: 10.1016/j.fsidi.2023.301505.

[20] Kwon, T., Su, Z.: Automatic detection of unsafe dynamic component loadings. IEEE Transactions on Software Engineering, 38(2), 293-313 (2012). doi: 10.1109/TSE.2011.108.

[21] Jin, D., & Lin, S.: Advances in computer science, intelligent system and environment. Vol. 1. Springer (2011).

[22] Dolnák, I.: Content Security Policy (CSP) as countermeasure to Cross Site Scripting (XSS) attacks. [Online]. Available: www.attackers.com.

[23] Gazzan, M., Sheldon, F. T.: Opportunities for Early Detection and Prediction of Ransomware Attacks against Industrial Control Systems. Future Internet, 15(4) (2023). MDPI. doi: 10.3390/fi15040144.

[24] Schmidbauer, T., Keller, J., Wendzel, S.: Challenging Channels: Encrypted Covert Channels within Challenge-Response Authentication. In ACM International Conference Proceeding Series. Association for Computing Machinery (2022). doi: 10.1145/3538969.3544455.

[25] Ashawa, M., Morris, S.: Analysis of Mobile Malware: A Systematic Review of Evolution and Infection Strategies. Journal of Information Security and Cybercrimes Research, 4(2), 103-131 (2021). doi: 10.26735/krvi8434.

[26] Wang, Q., Yan, H., Han, Z: Explainable APT Attribution for Malware Using NLP Techniques. In IEEE International Conference on Software Quality, Reliability and Security, QRS. Institute of Electrical and Electronics Engineers, pp. 70-80 (2021). doi: 10.1109/QRS54544.2021.00018.

[27] Choo, K. K. R., Dehghantanha, A.: Handbook of big data analytics and forensics. Springer International Publishing (2021). doi: 10.1007/978-3-030-74753-4.

[28] Zhao, Z., Jiang, Z., Chen, Y., Gong, X., Wang, W., Yew, P. C.: Enhancing Atomic Instruction Emulation for Cross-ISA Dynamic Binary Translation. In CGO 2021 - Proceedings of the 2021 IEEE/ACM International Symposium on Code Generation and Optimization. Institute of Electrical and Electronics Engineers Inc., pp. 351-362 (2021). doi: 10.1109/CGO51591.2021.9370312.

[29] Wang, Y., Yang, S., Ren, X., Zhao, P., Zhao, C., Yang, X.: IndustEdge: A Time-Sensitive Networking Enabled Edge-Cloud Collaborative Intelligent Platform for Smart Industry. IEEE Transactions on Industrial Informatics, 18(4), 2386-2398 (2022). doi: 10.1109/TII.2021.3104003.

[30] Ashawa, M., Morris, S.: Android Permission Classifier: a deep learning algorithmic framework based on protection and threat levels. Security and Privacy, 4(5) (2021). doi: 10.1002/spy2.164.

[31] Bagyalakshmi, G., et al.: Network Vulnerability Analysis on Brain Signal/Image Databases Using Nmap and Wireshark Tools. IEEE Access, 6, 57144-57151 (2018). doi: 10.1109/ACCESS.2018.2872775.

[32] Alsaheel, A., & Pande, R.: Using EMET to Disable EMET. www.fireeye.com., last accessed 2016

[33] Sadek, I., Chong, P., Rehman, S. U., Elovici, Y., Binder, A.: Memory snapshot dataset of a compromised host with malware using obfuscation evasion techniques. Data Brief, 26 (2019). doi: 10.1016/j.dib.2019.104437.

[34]

IEEE