Reverse Engineering a Malware

Valentin Lekov

Systemnähe Programmierung/Reverse Engineering
Erlangen, Germany

Keywords: Reverse Engineering, Malware, IDA, Ransomware, Obfuscation Techniques, String Obfuscation, Junk Code, Code Permutation, Opaque Predicates, Dead Code, Numbers Disguise, Merging, Import Hiding, Structured Exception Handling, Control Flow Obfuscation

October 23, 2020

Contents

1. Introduction

1.1. Context and Task

The idea of a computer malware precedes computer networks [1]. German mathematician John von Neumann first theorized the concept in the late 1940s [1]. He envisioned a computer virus as an automatically self-replicating entity [1]. But it was another 30 years before someone created one [1].

Malware or malicious software is any computer software intended to harm the host operating system or to steal sensitive data from users, organizations or companies [7]. Today, more than ever, there are thousands and thousands of malware in the wild.

Malware analysis is the study or process of determining the functionality, origin and potential impact of a given malware sample such as a virus, worm, trojan horse, rootkit, backdoor, etc [6].

This project is part of the module **Systemnähe Programmierung/Reverse Engineering**, where every student was given a different malware that they needed to analyze and write a report of. The analysis was to be divided into three subtasks:

1. reconstruction of the functionality of the malware
2. obfuscation techniques
3. validation of the ransom key

1.2. Structure of the Project Report

The project is divided into three important sections: **Introduction**, **Results** and **Summary and Conclusion**. The first section contains a short project diary, as well as a description of the working environment. The second sections consists of results obtained through the analysis of the malware and explains how the malware really works. The third sections is just a short explanation of the results, along with feedback.

1.3. Description of the Working Environment

The malware was analyzed in IDA on a virtual machine running Windows 7 x32 bit.

1.4. Description of the Approach

I started with the analysis in early August, but did not work on it every day. Because I had little experience with reverse engineering, I first started with static analysis, which immediately proved to be really hard.

Switching to (mostly) dynamic analysis made things easier because I could follow the execution of the program without the need to understand everything it was doing at that moment. I could also look at the memory and the registers and see which addresses were being modified, where the program was going to jump or return next, etc.

On some occasions, I did use pen and paper in order to see what was really going on. One such case was, for example, the instruction **call $ + 5**. Although I could follow the execution of the program and see what was happening, I could still not understand why it was happening.

Google, or mainly *Microsoft Docs*, were also of great help to me, because they helped me understand what arguments some of the functions are expecting and what their return value means.

2. Results

2.1. Overall Functionality of the Malware

2.1.1. Obfuscated Control Flow

Starting at the *main* function, the malware first copies the string **"Hier ist der Programmstart!"** into the variable *str_temp*. It then loads the address of the function *obfuscate1* into *eax* and by using the xor operation three times, it just swaps the values saved in the registers *eax* and *ebp*. After that, it will call the register *ebp*, i.e. the function *obfuscate1*.

The function *obfuscate1* pushes the address of the function *obfuscate2* onto the stack and calls the function *callOverwriteVars*. But because the address of the function *obfuscate2* that got pushed to the stack is never popped, it makes the stack misaligned and instead of returning back to the function *main*, it will return-jump to the function *obfuscate2*.

Function *callOverwriteVars* will just push the values of the registers *eax* and *ebx* onto the stack and call the function *overwriteVars*.

The code of the function *overwriteVars* is as follows:

```python
for i in range(0, 9):
    b_F5[i] = i << 2 + i - 11
```

Figure 1: *Source code example for overwriteVars*

While analyzing this function, I first came to the conclusion that it is just a junk function, because the variable that this function was writing to is only used here. But upon closer inspection, I found that this function actually overwrites some values that are later used for comparison, which produces control flow obfuscation. The values that it generates are [-11, -6, -1, 4, 9, 14, 19, 24, 29].

Returning back from *overwriteVars* to *callOverwriteVars*, the malware then returns to the function *obfuscate1*. As previously mentioned, because of the misaligned stack, it will return-jump to the function *obfuscate2*, instead of returning to the *main* function.

The function *obfuscate2* is not really big. First, it sets the *ebx* to 0 by using 3 instructions instead of just using *mov* or *xor*. Then, it uses 2 *mov* instructions in order to put the address of *obfuscate3* just after the return address. After that, it calls *$ + 5*, which just calls the next instruction. At the end, the function will return-jump to *obfuscate3* because of a misaligned stack.

In *obfuscate3* there is a comparison which always evaluates to false, thus allowing the function *loadDLLGetProcAddress* to be called.

Inside the function *loadDLLGetProcAddress*, through the use of *GetModuleHandleA*, the address of *kernel32.dll* is retrieved and saved into the variable *handle_kernel32DLL*. Then, the malware uses *handle_kernel32DLL* as well as *GetProcAddress* in order to retrieve the address of the function *GetProcAddress*, i.e. itself. This address is then saved into *addr_kernel 32_GetProcAddress*.

After this, the execution continues at *prepareObfuscate4*. In this block, the address of *obfuscate4* is pushed onto the stack using the *mov* instruction. Then, the malware jumps at address **0x00402004 + 1**.

The assembly code at address **0x00402004**, i.e. for *obfuscate3Epilogue* is as follows:

```
mov eax, C35DEC89h
```

Figure 2: *Assembly code for obfuscate3Epilogue*

Using an online assembler on the above intstruction, we get the following opcodes: **b8 89 ec 5d c3**. We can ignore the first opcode because the malware jumps at the address **0x00402005**, which is just after the first opcode. We can now use an online disassembler on the rest of the opcodes, which produces the following assembly code:

```
00402005: 89 ec -> mov esp, ebp
00402007: 5d    -> pop ebp
00402008: c3    -> ret
```

Figure 3: *Assembly code*

This is just an epilogue and because of yet another misaligned stack, it will return-jump at *obfuscate4*. The instructions at *obfuscate4* will push the address of the function *exceptionHandler* onto the stack and then call the function *loadDLLCheckConn*.

The function *loadDLL_CheckConn* first finds the addresses of the functions *LoadLibraryA* and *DeleteFileA* through the use of *GetProcAddress*. It then uses the function *LoadLibraryA* in order to load the dynamic link libraries **Wininet.dll**, **urlmon.dll** and **user32.dll**. After that, it once again uses *GetProcAddress* in order to find the address of the function *InternetGetConnectedState* from the dynamic link library named **Wininet.dll**. Finally, it uses *InternetGetConnectedState* in order to find if there is an internet connection, saves the result into the variable *int_connected* and returns back to the instructions at *obfuscate4*, i.e. from where it was called.

The last instruction inside the block at *obfuscate4* generates a hardware exception (access violation). Following the execution, the malware lands in the function *exceptionHandler*.

This function pushes the addresses of two different functions onto the stack: function *funcExit* (**0x0040205F**) and function *callMainFunctionality* (**0x00401DBD**). Upon executing the return instruction, the malware will land in the function *callMainFunctionality* because of a misaligned stack.

The function *callMainFunctionality* uses the *mov* instruction in order to insert the address of the function *mainFunctionality* onto the stack, right after the return address. It then pops the actual return address, so the function return-jumps to the address that it just inserted, i.e. in the function *mainFunctionality*.

2.1.2. Function mainFunctionality

This is where things start to get interesting, because this is one of the most important functions and everything happens inside here. This function will first load the *MessageBoxA* function using *GetProcAddress*. It will then check if there is an internet connection by using the variable *int_connected* and will only continue with execution if there is one available. Assuming the internet check is successful, the malware continues its execution.

Using the function *checkEncryptDecryptIDBFile* which is explained in the next subsection, it first checks if the IDB file is already encrypted. Based on the outcome, it will then either encrypt or try to decrypt the file.

Finally, depending on the outcome of the previous actions, the malware uses the function *fetchContentURL* to download the content from a particular URL, which will later be shown using a message box. The three different URLs that it uses are:

1. `https://faui1-files.cs.fau.de/public/OpenC3S/Sysprog/a`
2. `https://faui1-files.cs.fau.de/public/OpenC3S/Sysprog/b`
3. `https://faui1-files.cs.fau.de/public/OpenC3S/Sysprog/d`

2.1.3. Function checkEncryptDecryptIDBFile

The function *encryptDecryptIDBFile* will first retrieve the path of the executable file using the *GetModuleFileNameA* function and save it into the variable *str_temp*. It will then call *strstr* with *str_temp* and the substring **".exe"** as arguments and retrieve a pointer to the first occurrence in *str_temp* of the substring **".exe"**. The path to the executable is then copied into two variables *path_idb* and *path_ids*, but the **".exe"** part is omitted. After that, the malware appends the strings **".idb"** and **".ids"** to both variables accordingly.

Continuing with the execution, the malware checks the value of the first argument named *int_downloadReadCFile*. Depending on the value, there are two things that the malware can perform:

- *int_downloadReadCFile = 0*: The malware opens the IDB file and if it was successfully opened, it will read the first 7 bytes of it and save them into a variable named *idb_7_bytes*. Normally, the first seven bytes are **49 44 41 31 00 00 3E**, which is basically the string **"IDA\0\0>"**. Using the function *strcmp*, the malware will then compare the bytes in *idb_7_bytes* with the bytes **53 98 93 89 94 8f 8f** (**"Schrott"**).

  As seen later, the string **"Schrott"** is only used to tell the malware if the IDB file is encrypted or not. If the first seven bytes of the file are not equal to **"Schrott"**, that means that the IDB file is not yet encrypted. Thus, it will close the handle to it and return the value 1 to its caller function, telling it to encrypt the IDB file.

  If the content in the variable *idb_7_bytes* matches the string **"Schrott"**, the function will once again close the handle to the IDB file but return the value 2 to its caller function, telling it that the IDB file is already encrypted.

- *int_downloadReadCFile != 0*: The malware calls the function *downloadReadCFile*. This function uses *URLDownloadToFileA* in order to download the contents of the url `https://faui1-files.cs.fau.de/public/OpenC3S/Sysprog/2` and save them into a file named **c**.

  It then opens the file with read-bytes permission, reads the first 10 bytes and saves them into a variable named *str_encryptionDecryptionKey*. After that, it skips the space character, reads the next 34 bytes of the file and saves them into the variable *str_ransomKey*. Finally, it closes the handle to the IDB file, deletes the file and returns the value returned from the function *DeleteFileA*.

If the **c** file was successfully deleted, i.e. if *DeleteFileA* returned a non-zero value, the malware then uses the second argument *int_encryptIDBFile* in order to determine if the IDB file should be encrypted or decrypted. Here, depending on the value, there are also two things that the malware can perform: either encrypt (int_encryptIDBFile = 1) or decrypt (int_encryptIDBFile = 0) the file.

Encryption and decryption happen almost in the same way, except for two differences. One of the differences is that during encryption, the string **"Schrott"** is prepended to the content of the IDB file before it is encrypted. On the other hand, during decryption, the string **"Schrott"** is removed and the content of the IDB file is decrypted.

The other difference is that before decrypting the IDB file, the malware first checks if the key saved in the variable *str_ransomKey* is valid, which basically means that the victim has paid the ransom. The process of validation happens in the function *checkRansomKey* and is explained in subsection 2.3 through the use of flowcharts.

2.1.4. Encryption and Decryption

The process of encryption/decryption is really simple. The malware opens the IDB file with *read-bytes* permission and creates a new IDS file with *write-bytes* permission.

Ignoring the first one and using the remaining 9 bytes saved into the variable *str_encryptionDecryptionKey*, the malware XORs the first 9 bytes of the IDB file content accordingly. After that, the malware uses all 10 bytes from the variable *str_encryptionDecryptionKey*, XORs the next 10 bytes and repeats the process, until the whole content of the IDB file has been XOR-ed. All of the XOR-ed contents are then saved into the IDS file. The IDB file is then deleted and the IDS file is renamed into a IDB file.

Although the encryption/decryption process above is explained in terms of blocks of data, it actually happens as a stream of data, i.e. byte per byte. Assuming the content of the IDB file is saved in the variable *content* and the key is saved in the variable *key*, this is what the code will look like:

```python
content, key = ..., ...
j = 1

for i in range(len(content)):
    content[i] ^= key[j]
    j = (j + 1) % 10
```

Figure 4: *Source code example for encryption/decryption*

2.1.5. Function encryptDecryptString

Although this function is never mentioned in the above text, it is heavily used by the malware in order to obfuscate strings, e.g. for loading functions and dinamically linked libraries, as well as when downloading content from URLs.

This function ORs the first character of the string with the hex value **0xBA**. It then uses that value and XORs every remaining character with it. The following is an example of how this function may look like:

```
char_first = 0xBA | edString[0]

for i in range(1, len(edString)):
    edString[i] ^= char_first
```

Figure 5: *Source code example for encryptDecryptString*

2.2. Disguise Measures in Tabular Form

2.2.1. Scattered obfuscation procedures that are used multiple times

Obfuscation Procedure	**Program Reference**
String Obfuscation	0x00401BE5, 0x00401C05, 0x00401C11, 0x00401C3A 0x00401DF3, 0x00401E1F, 0x00401E2B, 0x00401E57, 0x00401E63, 0x00401E8, 0x00401E91, 0x00401EB3, 0x00401EBF, 0x00401EE1, 0x00401EED, 0x00401F16 0x00401C55, 0x00401C81 0x00401683, 0x004016AC, 0x004016BE, 0x00401707 0x00401B09, 0x00401BD0
Junk Code	0x00401F5F - 0x00401F74, 0x00401F8A - 0x00401F90 0x00401FA7 - 0x00401FA8 0x00401FC3, 0x00401FCD - 0x00401FCE
Code Permutation	0x00401F3C 0x00401FB2 - 0x00401FBB, 0x00401FC5 0x00401FF2 - 0x00401FF7
Opaque Predicates	0x00401FD3 - 0x00401FDA
Dead Code	0x00401FFC - 0x00402001
Numbers Disguise	0x00401F4D
Merging	0x00401F3B 0x00401802 - 0x00401AEE
Import Hiding	0x00401DF8 - 0x00401E0E, 0x00401E30 - 0x00401E46, 0x00401E6D - 0x00401E74, 0x00401E9B - 0x00401EA2, 0x00401EC9 - 0x00401ED0, 0x00401F20 - 0x00401F2F 0x00401C5A - 0x00401C70
Structured Exception Handling	0x0040200B - 0x00402040
Control Flow Obfuscation	0x00401F7F - 0x00401F88 0x00401F9C 0x00401FBE - 0x00401FC8 0x00401FE1 - 0x00401FEA, 0x00402004 0x00401DE5 - 0x00401DEA 0x00402052 - 0x0040205D 0x00401DC0 - 0x00401DD8 0x00401C48, 0x00401D04 - 0x00401D09 0x004018DC, 0x00401A05 - 0x00401A0A, 0x00401A57 - 0x00401A5C 0x004016B1, 0x00401756 - 0x0040175B 0x004014A7, 0x004014F0, 0x004015CA 0x00401B7F - 0x00401B84

2.2.2. Chronological listing of the obfuscation measures

Obfuscation Procedure	**Program Reference**
	main:
Junk Code	0x00401F5F - 0x00401F74, 0x00401F8A - 0x00401F90
Control Flow Obfuscation	0x00401F7F - 0x00401F88
	obfuscate1:
Control Flow Obfuscation	0x00401F9C
	callOverwriteVars:
Junk Code	0x00401FA7 - 0x00401FA8
	overwriteVars:
Code Permutation	0x00401F3C
Numbers disguise	0x00401F4D
	obfuscate2:
Code Permutation	0x00401FB2 - 0x00401FBB, 0x00401FC5
Control Flow Obfuscation	0x00401FBE - 0x00401FC8
Junk Code	0x00401FC3, 0x00401FCD - 0x00401FCE
	obfuscate3:
Opaque Predicates	0x00401FD3 - 0x00401FDA
Control Flow Obfuscation	0x00401FE1 - 0x00401FEA, 0x00402004
Code Permutation	0x00401FF2 - 0x00401FF7
Dead Code	0x00401FFC - 0x00402001
	loadDLLGetProcAddress:
String Obfuscation	0x00401BE5, 0x00401C05, 0x00401C11, 0x00401C3A
	obfuscate4:
Structured Exception Handling	0x0040200B - 0x00402040
	loadDLLCheckConn:
Control Flow Obfuscation	0x00401DE5 - 0x00401DEA
Merging	0x00401DEC - 0x00401F3B
String Obfuscation	0x00401DF3, 0x00401E1F, 0x00401E2B, 0x00401E57, 0x00401E63, 0x00401E8, 0x00401E91, 0x00401EB3, 0x00401EBF, 0x00401EE1, 0x00401EED, 0x00401F16
Import Hiding	0x00401DF8 - 0x00401E0E, 0x00401E30 - 0x00401E46, 0x00401E6D - 0x00401E74, 0x00401E9B - 0x00401EA2, 0x00401EC9 - 0x00401ED0, 0x00401F20 - 0x00401F2F
	exceptionHandler:
Control Flow Obfuscation	0x00402052 - 0x0040205D
	callMainFunctionality:
Control Flow Obfuscation	0x00401DC0 - 0x00401DD8
	mainFunctionality:
Control Flow Obfuscation	0x00401C48, 0x00401D04 - 0x00401D09
String Obfuscation	0x00401C55, 0x00401C81
Import Hiding	0x00401C5A - 0x00401C70
	checkEncryptDecryptIDBFile:
Merging	0x00401802 - 0x00401AEE
Control Flow Obfuscation	0x004018DC, 0x00401A05 - 0x00401A0A, 0x00401A57 - 0x00401A5C
	downloadReadCFile:
String Obfuscation	0x00401683, 0x004016AC, 0x004016BE, 0x00401707
Control Flow Obfuscation	0x004016B1, 0x00401756 - 0x0040175B
	checkRansomKey:
Control Flow Obfuscation	0x004014A7, 0x004014F0, 0x004015CA
	fetchContentURL:
String Obfuscation	0x00401B09, 0x00401BD0
Control Flow Obfuscation	0x00401B7F - 0x00401B84

2.3. Reversing the Verification Process for a Given Key

As explained before, the malware fetches the key used for encryption and decryption, as well as the key used for indicating if the ransom was paid or not. In the flowcharts below, the keyword *edKey* refers to the first case and *rKey* to the second case.

Although the flowchart in **Figure 1** calls three other functions, the real*checkRansomKey* in the malware is just one, whole function and does not make any calls to other functions. It is only presented in this way for a better visibility.

Using the rules from the flowcharts, one valid key would be: *rKey* = *"zzg-JABCfDbzzzzzzzzzzzmnopqrst1234"* when *edKey* = *"jdjfhzusdf"*.

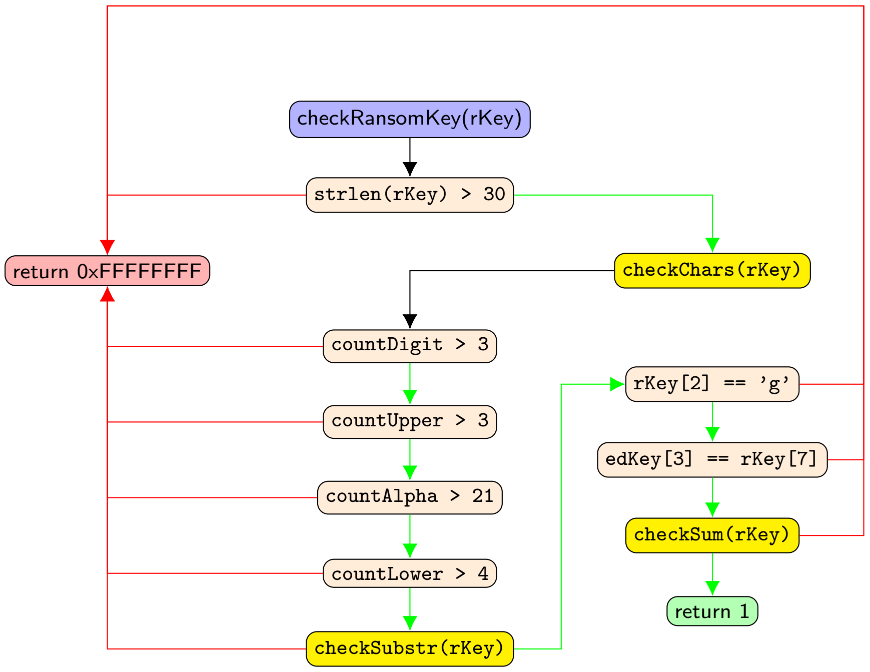

Figure 1: Flowchart of the function *checkRansomKey*

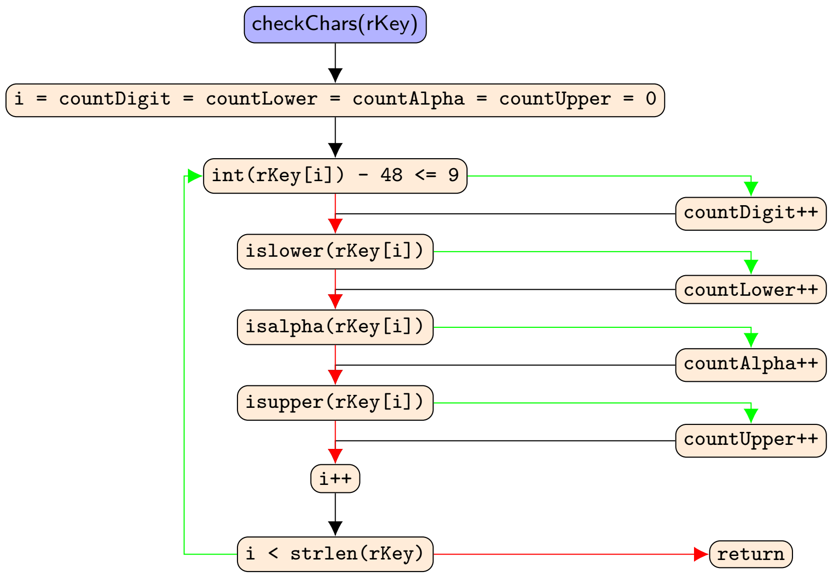

Figure 2: Flowchart of the function *checkChars*

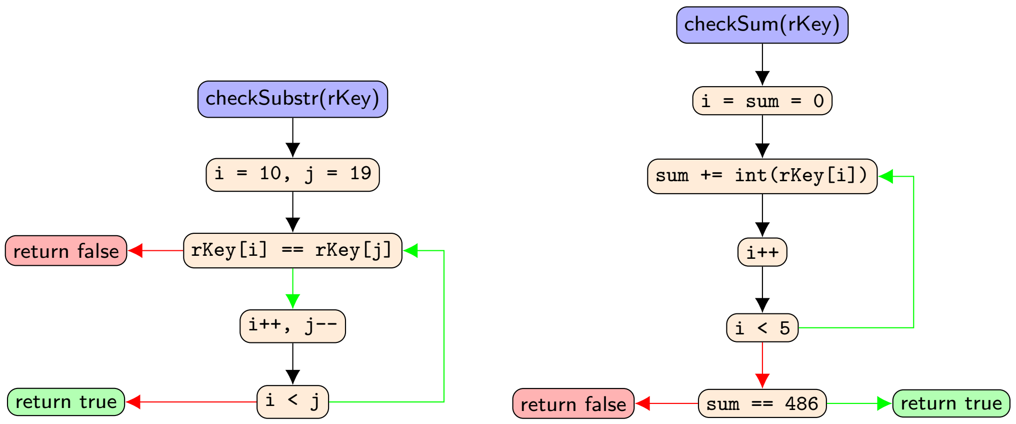

Figure 3: Flowcharts of the functions *checkSubstr* and *checkSum*

3. Summary and Conclusion

3.1. Short Summary of the Results

As we can see from the results, this type of malware is a crypto-ransomware [8], because it encrypts our IDB file and also asks for a ransom.

The malware uses multiple obfuscation techniques in order to make the reverse engineering a lot more difficult. The three most used ones are **Control Flow Obfuscation**, **String Obfuscation** and **Import Hiding**.

The main functionality of the malware starts in the *mainFunctionality* function. Through the use of other functions like *checkEncryptDecryptIDBFile*, *downloadReadCFile* or *fetchContentURL*, the malware can download files from the internet, as well as encrypt and decrypt files.

The encryption and decryption processes are basically the same, because both of them use the XOR operator and the same key. The second difference that was mentioned in the results is the key validation. When the victim pays the ransom, the author of the malware will change the *ransomKey*, thus allowing the IDB file to be decrypted.

At the end, the malware shows a message box with a text stating if the IDB file was encrypted or decrypted.

3.2. Feedback

I really liked the project and enjoyed working on it and I have no negative feedback to give.

I learned a lot about the obfuscation techniques and how they may look like. I also learned that backups are really useful, because my IDB file was somehow not working the next day. Further, I learned to pay more attention to messages, because they may contain information regarding my corrupted IDB file.

The most interesting task during the whole analysis was reversing the verification process for a given key, probably because there were not many obfuscation techniques inside this functions (only a couple of control flow obfuscations).

4. References

1. T. Matthews, "Creeper: The World's First Computer Virus - Exabeam", Exabeam, 2019. [Online]. Available: https://www.exabeam.com/information-security/creeper-computer-virus. [Accessed: 01 Oct, 2020].

2. ”GetModuleHandleA function (libloaderapi.h) - Win32 apps”, Docs. microsoft.com, 2020. [Online]. Available: https://docs.microsoft.com/en-us/windows/win32/api/libloaderapi/nf-libloaderapi-getmodulehandlea. [Accessed: 23 Oct, 2020].
3. ”GetProcAddress function (libloaderapi.h) - Win32 apps”, Docs. microsoft.com, 2020. [Online]. Available: https://docs.microsoft.com/en-us/windows/win32/api/libloaderapi/nf-libloaderapi-getprocaddress. [Accessed: 23 Oct, 2020].
4. ”GetModuleFileNameA function (libloaderapi.h) - Win32 apps”, Docs. microsoft.com, 2020. [Online]. Available: https://docs.microsoft.com/en-us/windows/win32/api/libloaderapi/nf-libloaderapi-getmodulefilenamea. [Accessed: 23 Oct, 2020].
5. ”DeleteFileA function (fileapi.h) - Win32 apps”, Docs.microsoft.com, 2020. [Online]. Available: https://docs.microsoft.com/en-us/windows/win32/api/fileapi/nf-fileapi-deletefilea. [Accessed: 23 Oct, 2020].
6. ”Malware analysis”, En.wikipedia.org, 2020. [Online]. Available: https://en.wikipedia.org/wiki/Malware_analysis. [Accessed: 23 Oct, 2020].
7. S. Gadhiya and K. Bhavsar, ”Wayback Machine”, Web.archive.org, 2020. [Online]. Available: https://web.archive.org/web/20160418151823/http://www.ijarcsse.com/docs/papers/Volume_3/4_April2013/V3I4-0371.pdf. [Accessed: 23 Oct, 2020].
8. R. Desai, ”What Are the Different Types of Ransomware?”, Tech-Wonders.com, 2020. [Online]. Available: https://www.tech-wonders.com/2020/08/what-are-the-different-types-of-ransomware.html. [Accessed: 23 Oct, 2020].