Loki-Bot: Come out, come out, wherever you are!

r3mrum.wordpress.com/2017/05/07/loki-bot-atrifacts/

View all posts by R3MRUM May 7, 2017

Intro

I'm going to make my first post an easy one. I'm currently in the middle of writing up my GREM Gold paper, which focuses on the reverse engineering of a Loki-Bot v1.8 sample. This post is going to focus on how Loki-Bot creates its mutex and the folders, files, and registry keys that are created as a result.

Per PhishMe:

> Loki Bot is a commodity **malware** sold on underground sites which is designed to steal private data from infected machines, and **then** submit that info to a command and control host via HTTP POST. This private data includes stored passwords, login credential information from Web browsers, and a variety of cryptocurrency wallets.

What is a Mutex?

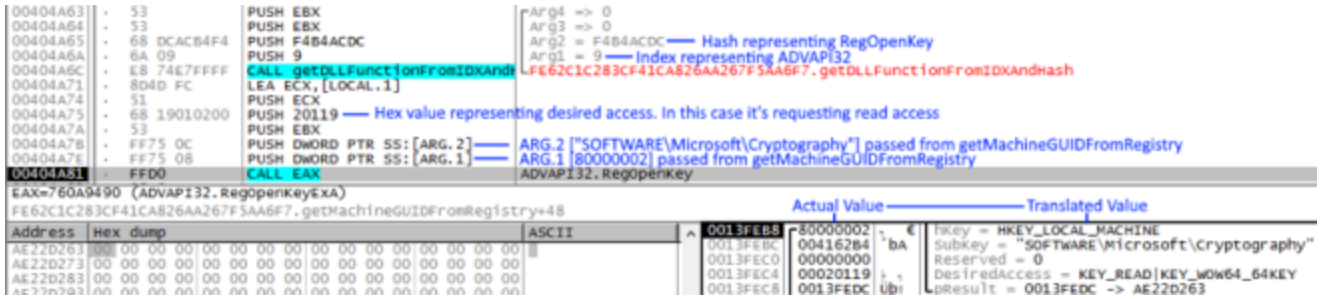
Understanding what a Mutex is can be a bit difficult to understand for those with little-to-no programming background. I found it best described on the SANS DFIR Blog:

> “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.”

Creating the Mutex

So, based on the mutex description, Loki-Bot uses a mutex to ensure that multiple versions of Loki-Bot cant be running at the same time. In order for this to happen, both versions of Loki-Bot need to have the same logic for naming the mutex. What we are going to talk about next is said logic.

Obtaining the Machine GUID

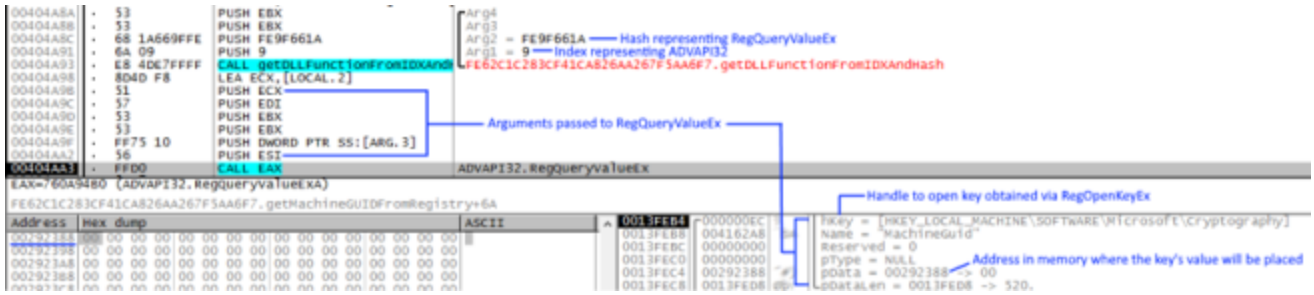


First and foremost, know that Loki-Bot employs function hashing to thwart analysis. This is what you are seeing from 0x404A63 to 0x404A6C. Two important arguments passed to the function labeled getDLLFunctionFromIDXAndHash are Arg1 (DLL Index) and Arg2 (Function Hash). In this instance, these values are set to 9 and ‘F4B4ACDC’. Without diving too deep into this, know that the DLL Index of 9 equates to ADVAPI32 and the hash ‘F4B4ACDC’ decodes to RegOpenKeyEx. At 0x404A81, we see the decoded function ADVAPI32.RegOpenKeyEx being called.

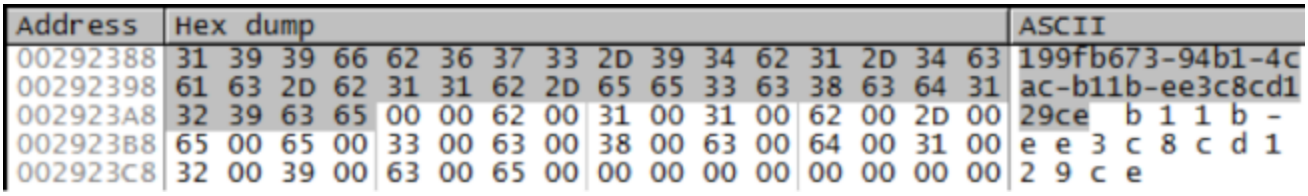
This will open the registry path:

“HKEY_LOCAL_MACHINE\SORTWARE\Microsoft\Cryptograpy\”

But it doesn’t actually *read* the value contained within the key it needs. For this to happen, ADVAPI32’s RegQueryValueEx function needs to be called.

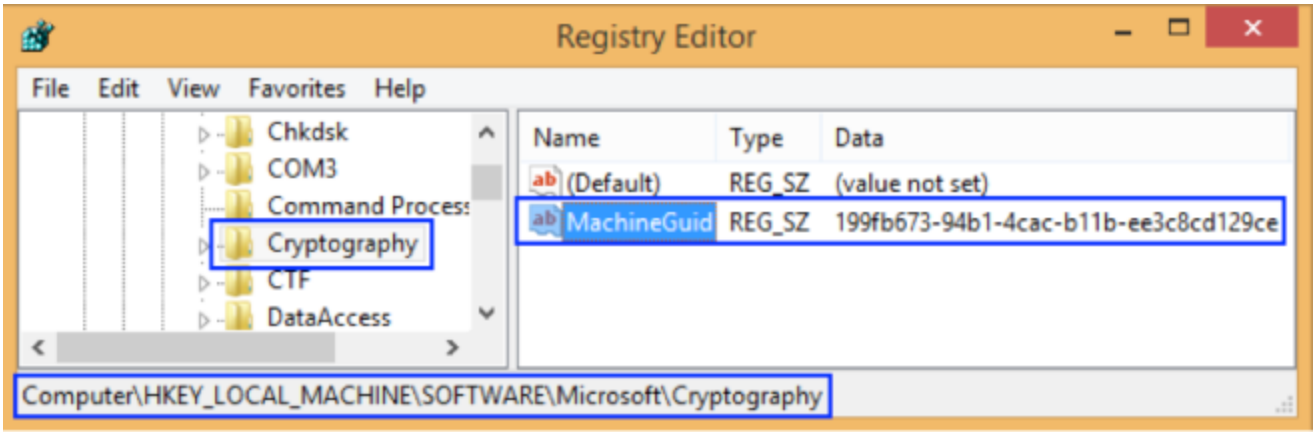


After successful execution, the value stored in the memory address referenced in the pData argument (0x292388) now contains the value that was in the HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Cryptography\MachineGuid registry key.



Address	Hex dump	ASCII
00292388	31 39 39 66 62 36 37 33 2D 39 34 62 31 2D 34 63	199fb673-94b1-4c
00292398	61 63 2D 62 31 31 62 2D 65 65 33 63 38 63 64 31	ac-b11b-ee3c8cd1
002923A8	32 39 63 65 00 00 62 00 31 00 31 00 62 00 2D 00	29ce b 1 1 b -
002923B8	65 00 65 00 33 00 63 00 38 00 63 00 64 00 31 00	e e 3 c 8 c d 1
002923C8	32 00 39 00 63 00 65 00 00 00 00 00 00 00 00 00	2 9 c e

We can validate this by simply loading up RegEdit on the Windows host that is about to be compromised and navigating to the referenced registry key.



The Machine GUID is supposed to be a value that is unique for each system. This means that your Machine GUID will be different from the Machine GUId depicted here; thus, your mutex will be different from mine.

MD5 Hash Machine GUID

Once the Machine GUID is obtained from the registry, Loki-Bot obtains the MD5 hash of the Machine GUID by making calls to ADVAPI's CryptAcquireContext, CryptCreateHash, CryptHashData, and CryptGetHashParam.

```
00403874  .  56            PUSH ESI                                 ┌Arg4
00403875  .  56            PUSH ESI                                 │Arg3
00403876  .  68 D10741D3   PUSH D34107D1                            │Arg2 = D34107D1 ── Hash representing CryptAcquireContext
0040387B  .  6A 09         PUSH 9                                   │Arg1 = 9 ── Index representing ADVAPI32
0040387D  .  E8 63F9FFFF   CALL getDLLFunctionFromIDXAndH           └FE62C1C283CF41CA826AA267F5AA6F7.getDLLFunctionFromIDXAndHash
00403882  .  68 000000F0   PUSH F0000000
00403887  .  6A 01         PUSH 1
00403889  .  56            PUSH ESI
0040388A  .  56            PUSH ESI
0040388B  .  8D4D F8       LEA ECX,[LOCAL.2]
0040388E  .  51            PUSH ECX
0040388F  .  FFD0          CALL EAX                                 ADVAPI32.CryptAcquireContext
00403891  .  85C0          TEST EAX,EAX
00403893  .v 0F84 84000000 JZ 00403910
00403899  .  8B75 F8       MOV ESI,DWORD PTR SS:[LOCAL.2]
0040389C  .  6A 00         PUSH 0                                   ┌Arg4 = 0
0040389E  .  6A 00         PUSH 0                                   │Arg3 = 0
004038A0  .  68 F3A6B0ED   PUSH EDB0A6F3                            │Arg2 = EDB0A6F3 ── Hash representing CryptCreateHash
004038A5  .  6A 09         PUSH 9                                   │Arg1 = 9 ── Index representing ADVAPI32
004038A7  .  E8 39F9FFFF   CALL getDLLFunctionFromIDXAndH           └FE62C1C283CF41CA826AA267F5AA6F7.getDLLFunctionFromIDXAndHash
004038AC  .  8D4D FC       LEA ECX,[LOCAL.1]
004038AF  .  51            PUSH ECX
004038B0  .  6A 00         PUSH 0
004038B2  .  6A 00         PUSH 0
004038B4  .  68 03800000   PUSH 8003 ── ALG_ID
004038B9  .  56            PUSH ESI
004038BA  .  FFD0          CALL EAX                                 ADVAPI32.CryptCreateHash
004038BC  .  85C0          TEST EAX,EAX
004038BE  .v 74 51         JZ SHORT 00403911
004038C0  .  6A 00         PUSH 0                                   ┌Arg4 = 0
004038C2  .  53            PUSH EBX                                 │Arg3
004038C3  .  FF75 08       PUSH DWORD PTR SS:[ARG.1]                │Arg2 => [ARG.1] ── Machine GUID
004038C6  .  FF75 FC       PUSH DWORD PTR SS:[LOCAL.1]              │Arg1 => [LOCAL.1]
004038C9  .  E8 98FDFFFF   CALL cryptHashData                       └FE62C1C283CF41CA826AA267F5AA6F7.cryptHashData
004038CE  .  83C4 10       ADD ESP,10
004038D1  .  85C0          TEST EAX,EAX
004038D3  .v 74 3C         JZ SHORT 00403911
004038D5  .  8B75 FC       MOV ESI,DWORD PTR SS:[LOCAL.1]
004038D8  .  6A 00         PUSH 0                                   ┌Arg4 = 0
004038DA  .  6A 00         PUSH 0                                   │Arg3 = 0
004038DC  .  68 FDDBA8FE   PUSH FEA8DBFD                            │Arg2 = FEA8DBFD ── Hash representing CryptGetHashParam
004038E1  .  6A 09         PUSH 9                                   │Arg1 = 9 ── Index representing ADVAPI32
004038E3  .  E8 FDF8FFFF   CALL getDLLFunctionFromIDXAndH           └FE62C1C283CF41CA826AA267F5AA6F7.getDLLFunctionFromIDXAndHash
004038E8  .  6A 00         PUSH 0
004038EA  .  8D4D F4       LEA ECX,[LOCAL.3]
004038ED  .  51            PUSH ECX
004038EE  .  57            PUSH EDI ── Address where the resulting MD5 hash of the Machine GUID will be stored
004038EF  .  6A 02         PUSH 2
004038F1  .  56            PUSH ESI
004038F2  .  FFD0          CALL EAX                                 ADVAPI32.CryptGetHashParam
004038F4  .  85C0          TEST EAX,EAX
004038F6  .v 74 19         JZ SHORT 00403911
004038F8  .  FF75 FC       PUSH DWORD PTR SS:[LOCAL.1]              ┌Arg1 => [LOCAL.1]
004038FB  .  E8 2FFDFFFF   CALL cryptDestroyHash                    └FE62C1C283CF41CA826AA267F5AA6F7.cryptDestroyHash
00403900  .  6A 00         PUSH 0
00403902  .  FF75 F8       PUSH DWORD PTR SS:[LOCAL.2]
00403905  .  E8 3FFDFFFF   CALL cryptReleaseContext
```

After CryptGetHashParam executes, the MD5 hash of the Machine GUID is returned.

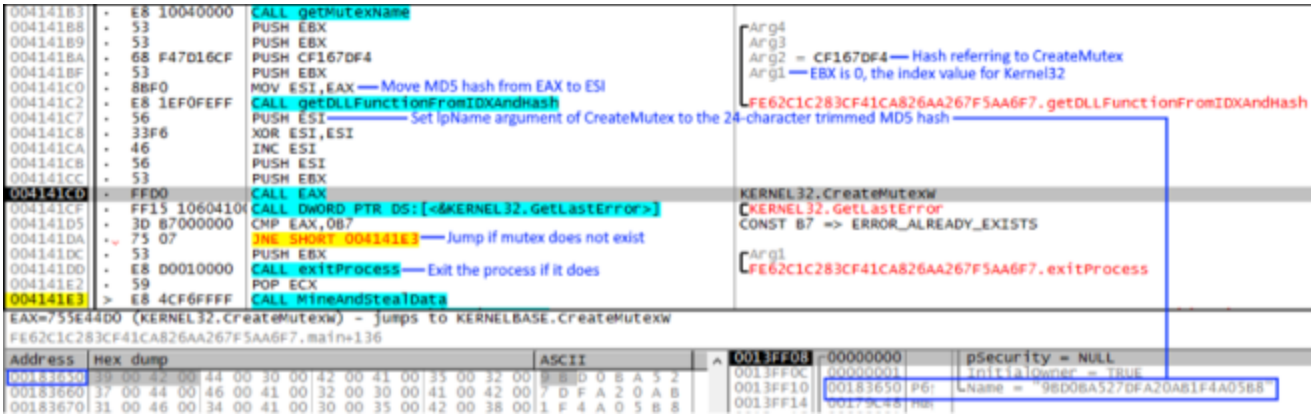
```
004038F2  .  FFD0          CALL EAX                                 ADVAPI32.CryptGetHashParam
004038F4  .  85C0          TEST EAX,EAX ── Just executed CryptGetHashParam
004038F6  .v 74 19         JZ SHORT 00403911
004038F8  .  FF75 FC       PUSH DWORD PTR SS:[LOCAL.1]              ┌Arg1 => [LOCAL.1]
EAX=00000001
FE62C1C283CF41CA826AA267F5AA6F7.getMD5Hash+0CA

Address   Hex dump                                        ASCII
00202708  9B D0 BA 52 7D FA 20 AB 1F 4A 05 B8 D0 D4 E0 4B  ›Đ°R}ú « J¸ĐÔàK
00202718  AB AB AB AB AB AB AB AB 00 00 00 00 00 00 00 00  «««««««« ........
```

The MD5 hash of our Machine GUID appears to be “9BD0BA527DFA20AB1F4A05B8D0D4E04B“. There are a number of different ways that we could validate this result but I find that it’s easiest using the linux command line.

```
remnux@remnux: ~
File  Edit  Tabs  Help
remnux@remnux:~$ echo -n '199fb673-94b1-4cac-b11b-ee3c8cd129ce' |md5sum | tr [a-z] [A-Z]
9BD0BA527DFA20AB1F4A05B8D0D4E04B  -
remnux@remnux:~$
```

Trim Hash & Create Mutex

Finally, Loki-Bot trims the MD5 hash of the Machine GUID to 24-characters: “9BD0BA527DFA20AB1F4A05B8“.



It then passes this trimmed value to Kernel32’s CreateMutexW function as the *lpName* attribute. If the function succeeds, it means that no other version of Loki-Bot is running on the system at that time and execution continues on. If it fails, it means another version of Loki-Bot is running, so Loki-Bot quietly exits.

Identify Folder/Files

Now that we know the mutex, we can identify the folders and files that are related to Loki-Bot. As part of setting up persistence, Loki-Bot will create a hidden folder within your %APPDATA% path whose name set by the 8th thru 13th characters of the mutex.

Once the hidden folder “%APPDATA%\27DFA2\” has been created, Loki-Bot will store several different types of files within it; all with the same filename but with different extensions. The filename used for the different files is also extracted from the mutex.

With the filename known, we can then identify the following files:

- **%APPDATA%\27DFA2\20AB1F.exe** – A copy of the malware that will execute every time the user account is logged into.
- **%APPDATA%\27DFA2\20AB1F.hdb** – A database of hashes for data that has already been exfiltrated to the C2 server.
- **%APPDATA%\27DFA2\20AB1F.lck** – A lock file created when either decrypting Windows Credentials or Keylogging to prevent resource conflicts.
- **%APPDATA%\27DFA2\20AB1F.kdb** – A database of keylogger data that has yet to be sent to the C2 server.

Identify Registry Key

The path for the specific persistence registry key used is encrypted within the binary using Triple DES encryption, which is why static analysis wont yield much. Once decrypted, my sample returned the following registry path used for persistence:

“HKEY_LOCAL_MACHINE\ Software\Microsoft\Windows\CurrentVersion\Run\”

The registry key within this path is then derived from the Mutex exactly how our %APPDATA% subfolder was:

“HKEY_LOCAL_MACHINE\ Software\Microsoft\Windows\CurrentVersion\Run**27DFA2**“

The value assigned to this key is the executable that is stored within the %APPDATA% subfolder:

“%APPDATA%\27DFA2\20AB1F.exe”

Conclusion

That pretty much covers all artifacts related to Loki-Bot that could be present on a compromised system. First step is to identify your system’s Machine GUID. Once you do that, MD5 hash and then trim that value. The result will help you identify all the different folders, files, and registry keys associated with the malware.