# Malware Report

## “Dridex Version 4”

August, 2017

## CONTENTS



## 1. Executive Summary

The present document gathers analysis of a new variant of harmful code called "Dridex", specifically the fourth version.

Dridex is a banking Trojan famous for its sophistication and its ability to go undetected on the devices it infects. These devices, once infected, are incorporated onto a modular botnet, at which point malicious characteristics, whether external or their own, can be freely added to them, via modules or libraries.

The first version appeared toward the end of 2014. At the beginning of 2015, a new, important update was launched, giving way to a second version. When looking at the earlier versions of Dridex, the most stable and resistant of them was the third, which was launched in April 2015 and was used in well-known cyberattacks up until the fourth version, the latest known version and subject of this report, which was found in February of 2017.

No new major updates for Dridex had been found since the dismantlement of important components of the botnet, carried out by government agencies in 2015. [1]

This new variant of the banking Trojan incorporates new functionalities. One of these is called AtomBombing, a functionality whose aim is to inject code without calling suspicious APIs to avoid being detected by monitoring systems. It incorporates the DLL hijacking technique to achieve persistence. Finally, various cryptographic methods were optimized and used to obtain the configuration. [2]

## 2. CHARACTERISTICS OF THE TROJAN

The following are some static properties of the analysed file.

The hash of the Trojan:

| | |
|---|---|
| MD5 | 001fcf14529ac92a458836f7cec03896 |
| SHA256 | a6db7759c737cbf6335b6d77d43110044ec049e8d4cbf7fa9bd4087fa7e415c7 |

The internal date of creation of the analyzed sample is May 16, 2017. The file in question was compiled to be executed in 64 bit environments and, at the same time, simulate the legitimate dll of Microsoft.

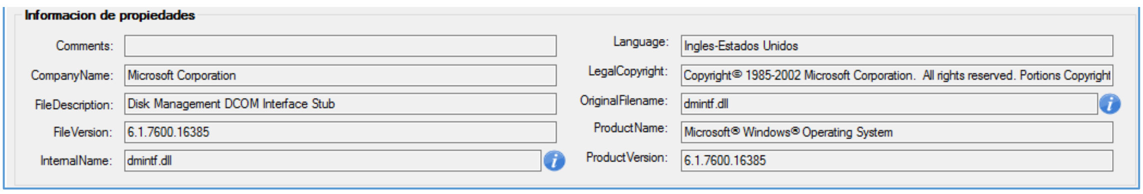



Figure 1. File properties

Additionally, it is encrypted with a distinctive algorithm to avoid detection by antiviruses.

It has been observed that the executable has a fairly high number of sections, 11 in total, as we can see in Figure 2:

| property | value | value | value | value | value | value | value | value | value | value | value |
|---|---|---|---|---|---|---|---|---|---|---|---|
| name | .text | .code | .sbss | .rdata | .data | .pdata | DATA | .crt0 | .rsrc | .reloc | .kwqrcd |
| virtual-size | 0x000005E6 (1510) | 0x00001AFC (6908) | 0x00000657 (1623) | 0x0001DDCB (122315) | 0x00002F62 (12130) | 0x000005FA (1530) | 0x00026B3F (158527) | 0x0001C16E (115054) | 0x000004C8 (1224) | 0x0000056C (1388) | 0x00000AE7 (2791) |
| virtual-address | 0x00001000 | 0x00002000 | 0x00004000 | 0x00005000 | 0x00023000 | 0x00026000 | 0x00027000 | 0x0004E000 | 0x0006B000 | 0x0006C000 | 0x0006D000 |
| raw-size | 0x00001000 (4096) | 0x00002000 (8192) | 0x00001000 (4096) | 0x0001E000 (122880) | 0x00003000 (12288) | 0x00001000 (4096) | 0x00027000 (159744) | 0x0001D000 (118784) | 0x00001000 (4096) | 0x00001000 (4096) | 0x00001000 (4096) |
| raw-data | 0x00001000 | 0x00002000 | 0x00004000 | 0x00005000 | 0x00023000 | 0x00026000 | 0x00027000 | 0x0004E000 | 0x0006B000 | 0x0006C000 | 0x0006D000 |
| PointerToRelocations | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 |
| PointerToLinenumbers | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 |
| NumberOfRelocations | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 |
| NumberOfLinenumbers | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 |
| md5 | A307CFABD21EE5... | 80A66726FB60EAD... | A38BB27F841B4F... | 258595205F0C58FBE... | B3AF0999C1DCB41... | 3848DA66328342... | D52A19B03FCF1B5... | A6F51B37F58FA376... | 3ECC94FD83DAC... | CB1AE24217F540... | 620F0B67A91F7F74151B... |
| cave | 0x00000A1A (2586) | 0x00000504 (1284) | 0x000009A9 (2473) | 0x00000235 (565) | 0x0000009E (158) | 0x00000A06 (2566) | 0x000004C1 (1217) | 0x00000E92 (3730) | 0x00000B38 (2872) | 0x00000A94 (2708) | 0x00000519 (1305) |
| entropy | 3.139 | 5.116 | 2.033 | 7.844 | 2.699 | 0.543 | 7.799 | 7.783 | 1.307 | 0.819 | 0.000 |
| entry-point | x | - | - | - | - | - | - | - | - | - | - |
| obfuscated | - | - | - | - | - | - | - | - | - | - | - |
| blacklisted | - | - | - | - | - | - | - | - | - | - | - |
| readable | x | x | x | x | x | x | x | x | x | x | x |
| writable | - | - | - | - | x | - | x | x | - | - | - |
| executable | x | x | x | - | - | - | - | - | - | - | - |
| shareable | - | - | - | - | - | - | - | - | - | - | - |
| discardable | - | - | - | - | - | - | - | - | - | x | - |
| cachable | x | x | x | x | x | x | x | x | x | x | x |
| pageable | x | x | x | x | x | x | x | x | x | x | x |
| initialized-data | - | - | - | x | x | x | x | x | x | x | x |
| uninitialized-data | - | - | - | - | - | - | - | - | - | - | - |

**Figure 2. Static information of the analyzed binary**

In the DATA section, we can observe that the entropy is at 7.799, and is a fairly large in size. It is in this section that the highly encrypted and packaged binary (which, once decrypted, becomes the real malicious code) can be found.

In the first decrypted layer, the executable stores memory in the process, then copies the code and, finally, summons it and runs it, as we see in Figure 3:

```
call    AddressDOSHeader
mov     [rsp+1C8h+var_C8], rax
mov     rcx, rax
call    AddressPEHeader
mov     r8d, [rax+50h]
mov     eax, r8d
mov     [rsp+1C8h+var_C0], rax
mov     rdx, [rsp+1C8h+var_18]
lea     rax, [rsp+1C8h+dwSize]
mov     rcx, rax
mov     [rsp+1C8h+var_1A8], rax
call    CopyPEtoMemory
mov     rcx, [rsp+1C8h+var_1A8]
call    rax                 ; Execute shellcode
```

**Figure 3. Jump to shellcode**

The first thing the code does is to obtain the addresses of the functions that it will eventually be using. It does this with a dynamic search through the libraries downloaded by the program.

To carry out this task, it runs through the PEB_LDR_DATA structure and the LDR-MODULE structures to locate the base address of the loaded dlls. It proceeds to access the offset of the export table in order to run through all of the functions exported by the dll and find the address of the sought function in he computer's memory.

```
mov r9,qword ptr gs:[30]          Acceso al TEB
mov rax,qword ptr ds:[r9+60]      Acceso al PEB
mov rax,qword ptr ds:[rax+18]     Acceso al PEB_LDR_DATA
mov r9,rax
add r9,20
mov rax,qword ptr ds:[rax+20]     Acceso a la lista InMemoryOrderList
cmp rax,r9
```

**Figure 4. Enumeration of loaded modules**

The shellcode, in turn, checks to see whether there is a hook in the undocumented LdrLoadDll function, accessing its address and checking whether the first byte is the same as E9, the equivalent of a jmp assembler.

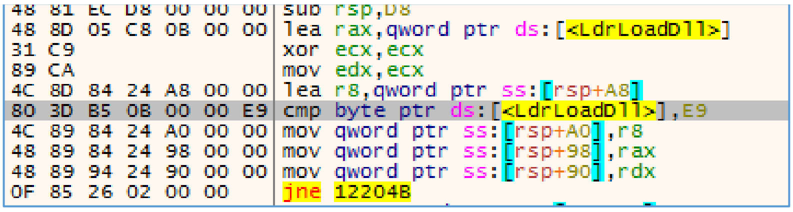



Figure 5. Hook Verification

If the previous verification was successful, it proceeds to demap the dll memory process with the name "snxhk.dll" which is an Avast and AVG library that creates hooks to monitor processes happening in the sandbox.

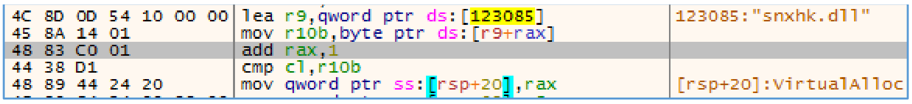



Figure 6. Library: snxhk.dll

Finally, the shellcode decrypts the executable found in the DATA section in the computer's memory, copies it into the base image's address, and then runs the new resulting executable.

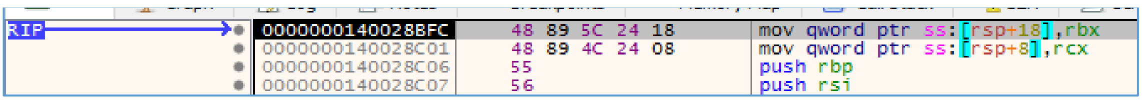



Figure 7. Decrypted executable

In summary, the full process of the sample being unpacked can be seen in Figure 8, where it is detailed more schematically.

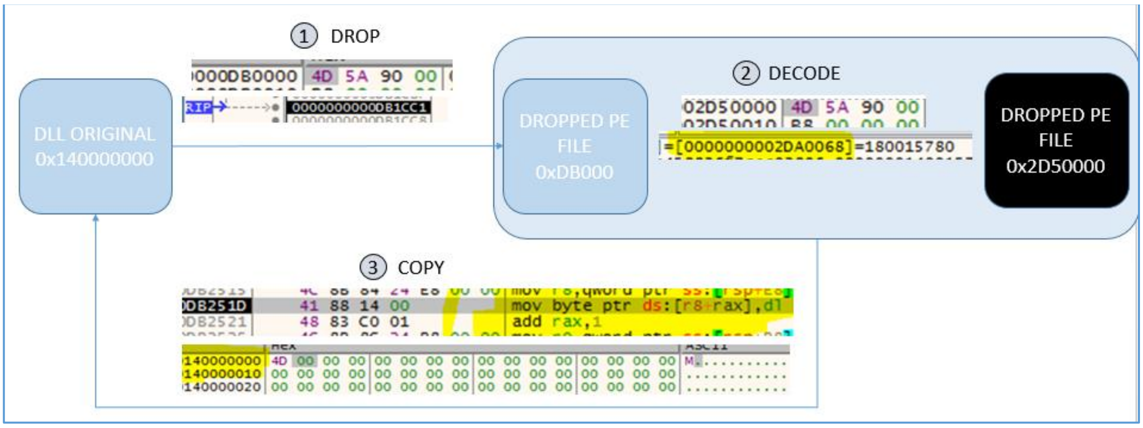



Figure 8. Complete unpacking process

# 3. INFECTION PROCESS

## 3.1. Infection Vectors

The infection of the device is not clearly understood. It may come by way of an exploit kit or spam campaign.

## 3.2. Interactions with the Affected System

Once it is run, the Trojan will proceed to verify if it is the only instance of malware running on the device, as well as to verify if it has already been injected in the explorer.exe process.

All of this is carried out by creating and opening a mutex. In order to achieve this, it first strings together the device name and the username, then calculates its MD5 hash.

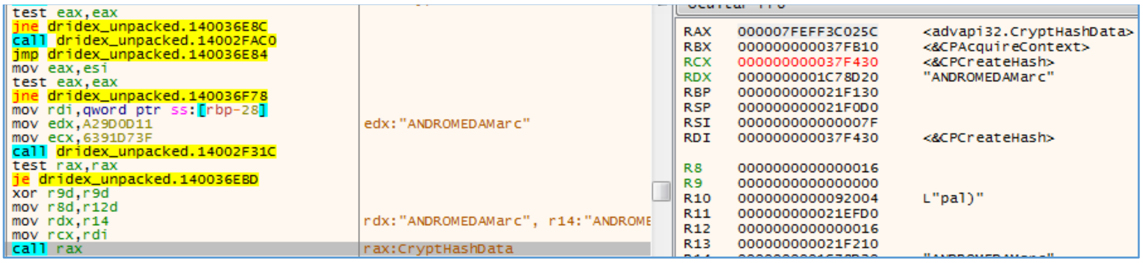
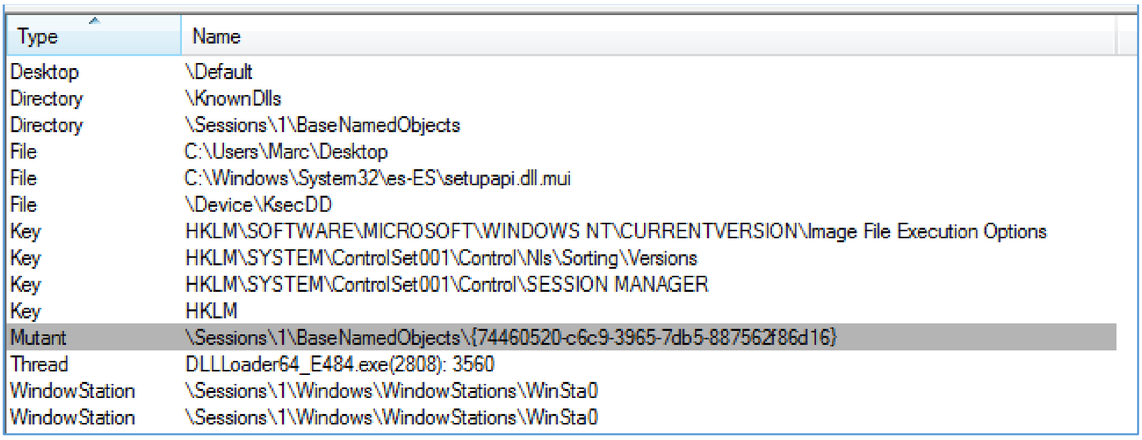



Figure 9. Hash calculation

Next, it adds brackets to the beginning and the end, and separates it with hyphens, similar to a COM object.



| Type | Name |
|---|---|
| Desktop | \Default |
| Directory | \KnownDlls |
| Directory | \Sessions\1\BaseNamedObjects |
| File | C:\Users\Marc\Desktop |
| File | C:\Windows\System32\es-ES\setupapi.dll.mui |
| File | \Device\KsecDD |
| Key | HKLM\SOFTWARE\MICROSOFT\WINDOWS NT\CURRENTVERSION\Image File Execution Options |
| Key | HKLM\SYSTEM\ControlSet001\Control\Nls\Sorting\Versions |
| Key | HKLM\SYSTEM\ControlSet001\Control\SESSION MANAGER |
| Key | HKLM |
| Mutant | \Sessions\1\BaseNamedObjects\{74460520-c6c9-3965-7db5-887562f86d16} |
| Thread | DLLLoader64_E484.exe(2808): 3560 |
| WindowStation | \Sessions\1\Windows\WindowStations\WinSta0 |
| WindowStation | \Sessions\1\Windows\WindowStations\WinSta0 |

Figure 10. Mutex created in the system

Using this algorithm, it may be possible to develop a vaccine that creates these mutexes in systems to avoid infection by Dridex.

Malware that is not running creates a folder in %WINDOWS%\system32\[0-9]{4}

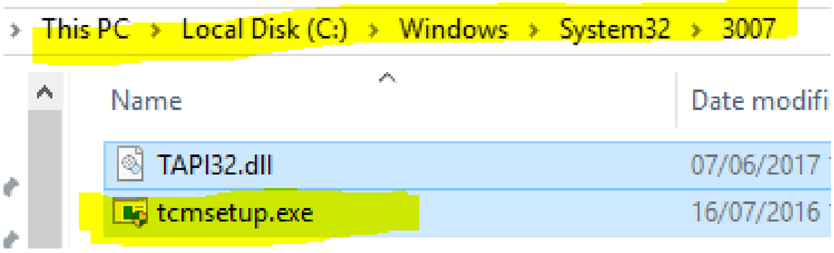



Figure 11. Created folder

The malware copies a legitimate .exe into the folder along with an associated .dll or .cpl. This .dll or .cpl is not legitimate — it's a Trojan. Upon running the .exe from the folder, the malicious .dll or .cpl will load via a technique known as hijacking.

It also programs a task with a randomized name ("Domitxtdoi" in our example in Figure 12), which will run every 60 minutes.

```
schtasks.exe  /Create /F /TN "Domltxtdoi" /SC minute /MO 60 /TR "C:\Windows\system32\3007\tcmsetup.exe" /RL highest
```

Figure 12. Creation of task

In this example, we see that the tcmsetup.exe runs so that the malicious .dll, TAPI32.dll, loads, thus beginning the infection process.

After programming the task, it launches a series of commands: it creates a rule in the firewall for explorer.exe, which is where it will be injected:

```
netsh  advfirewall firewall add rule name="Core Networking - Multicast Listener Done
(ICMPv4-In)" program="C:\Windows\Explorer.EXE" dir=in action=allow protocol=TCP
localport=any
```

Creation of the malicious task

```
schtasks.exe  /Create /F /TN "Utdcm" /SC minute /MO 60 /TR
"C:\Windows\system32\3007\tcmsetup.exe" /RL highest
```

During this process, the malicious .dll will have been injected into the explorer.exe process using the AtomBombing technique. It will then wait for the user to open a browser like Internet Explorer, Firefox, Chrome, etc.

The moment the user opens a browser, a new shellcode will be injected from explorer.exe to the browser using the same AtomBombing technique.

## 4. PERSISTENCE IN THE SYSTEM

To ensure its persistence in the system, it carries out the following actions.

It creates a folder with four random numbers on C:\Windows\System32, inside of which it copies a legitimate Windows executable (not always the same one) and a .dll that it knows will be loaded by the executable. This .dll will be modified with the harmful code.

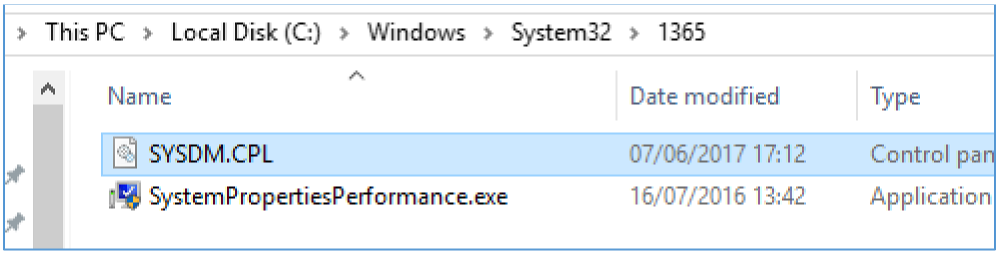



**Figure 13. Persistence in the system**

This technique is known as DLL hijacking. It takes advantage of the command that allows the system to search libraries/files that it's going to load/use. In the case of the image above, the executable "SystemPropertiesPerformance.exe" will load "SYSDM.CPL" among other libraries. By default, the first place that it will search for the "SYSDM.CPL" file will be in the directory where the application is running, in this case C: \ Windows \ System32 \ 1365. If it does not find it, it will look it up on other routes depending on how the search order of .dlls in the system is set.

When it copies an executable and a modified .dll in the same directory, Dridex's aim is to raise as little suspicion as possible, since its malicious actions are carried out by way of a legitimate program.

To execute the file, it creates a scheduled task to run it in the random number folder (C: \ Windows \ System32 \ 1365) every hour, as indicated in the previous section.

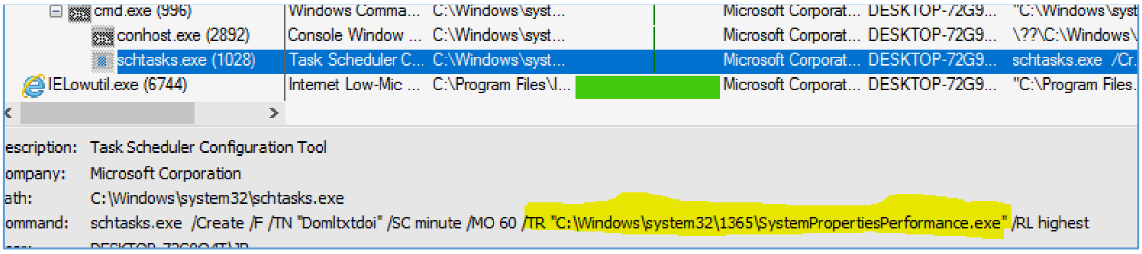



**Figure 14. Creation of the programmed task**

As already mentioned, the folder is composed of four random numbers, and the executable it creates is not always the same, just like the .dll, so it is aware of which executable loads which library at all times, and is able to modify said library with harmful code.

Going further in our analysis, we see that it acts in the following manner:

1. It will list all executables in the folder "C: \ Windows \ System32 \"
2. It will hash the name of each executable and compare it with a value that has been previously saved. If it matches, it will remain with that executable (in each execution that the hash is different).
3. It will read the IAT of the selected executable and from there choose a .dll for eventual hijacking.

4. It will read the IAT of the .dll selected in point 3.
5. It will make a copy of the malicious code (the .dll itself) and add a section at the end with a random name to copy the IAT obtained in point 4.
6. It will copy both the selected executable (3) and the modified malicious .dll (5) into a random folder.

In this way it obtains persistence in the system and every time that file is executed it will load the malicious .dll.

The malware will also create a copy of itself in executable format along with a registry key in the AppData\Roaming\[random folder name] with the route in "HKCU\Software\Microsoft\Windows\CurrentVersion\Run".

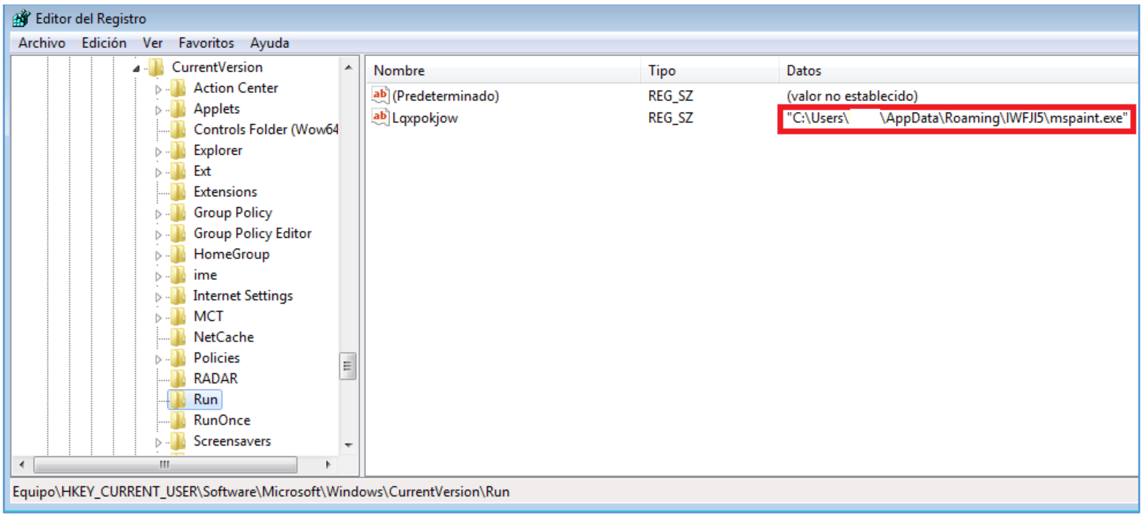



Figure 15. Registry key

## 5. INJECTION VIA ATOMBOMBING

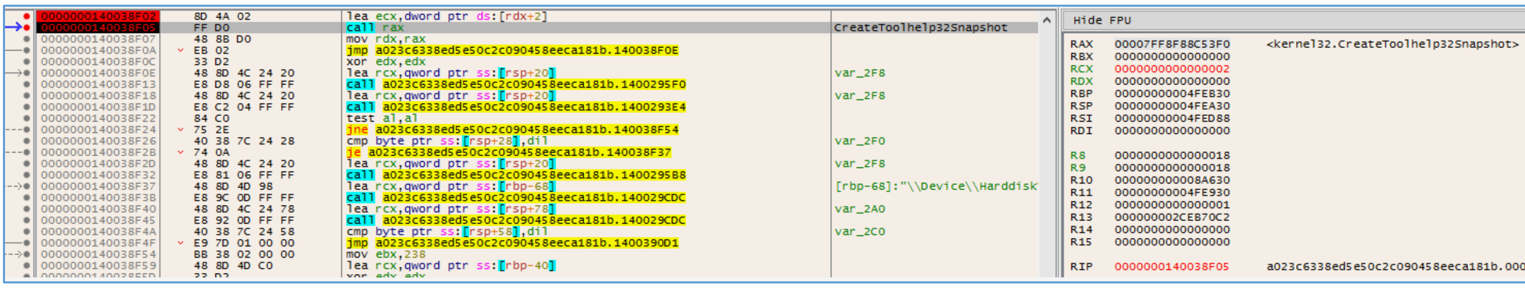
Dridex uses the AtomBombing technique to write a shellcode in other processes without raising suspicions.

It achieves this through APC calls and one of the most used Windows Executive Objects, called Atoms.

Below are the different phases of injection into another process.

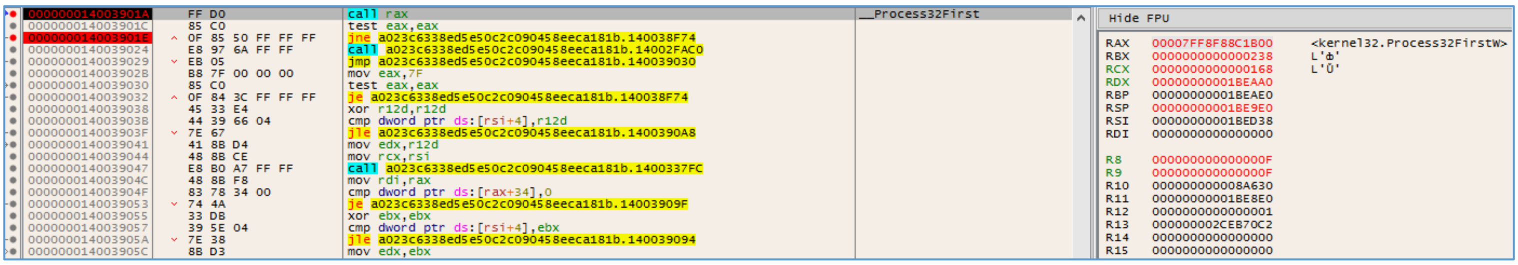
### 5.1. Search for the target process

The target process in this case is explorer.exe, and to inject into it, it must first be accessed in order to perform an enumeration of the processes involved, making use of functions such as the following:




Once it finds the process explorer.exe, it calls the OpenProcess function to begin enumerating alertable threads.

## 5.2. Search for alertable threads




At this point, the malware will try to find some thread in an alertable state, as this will allow it to make APC calls in order to execute code in the target process.

To find an alertable thread, it first obtains a handle for each thread in explorer.exe. It will then launch a call to NtQueueApcThread as NtSetEvent and wait for any of the threads to respond.

If it works correctly, it will obtain the first thread that answers the call and start with the injection.

## 5.3. Injection of shellcode in the target process

First, the malicious .dll makes a call to GlobalAddAtomW and creates a new Atom with the content it wishes to inject in the target process, in this case explorer.exe.

Second, the malicious .dll calls the NtQueueApcThread and sends as a parameter the function to be run by explorer.exe.

The first time this is done, it makes a call to memset to make sure that the zone where it will write the shellcode is at 0.

| | | |
|---|---|---|
| RAX | 00000000774DC180 | <ntdll.NtQueueApcThread> |
| RBX | 0000000000000000 | |
| RCX | 00000000000000F8 | 'ø' |
| RDX | 00000000774DD910 | <ntdll.memset> |
| RBP | 00000000000000F8 | 'ø' |
| RSP | 00000000002CEB78 | |
| RSI | 00000000774DD910 | <ntdll.memset> |
| RDI | 00000000775CAAA0 | ntdll.00000000775CAAA0 |
| R8 | 00000000775CAAA0 | ntdll.00000000775CAAA0 |
| R9 | 0000000000000000 | |
| R10 | 0000000000000080 | |
| R11 | 00000000002CE678 | "q7Uw" |
| R12 | 00000000775CAAA0 | ntdll.00000000775CAAA0 |
| R13 | 00000000758C5F0 | ntdll.00000000758C5F0 |
| R14 | 00000000773D6BF0 | <kernel32.GlobalGetAtomNameA> |
| R15 | 0000000000000000 | |
| RIP | 00000000774DC180 | <ntdll.NtQueueApcThread> |

Figure 19. Memory wipe

It is important to indicate that the zone that Dridex has chosen for copying the shellcode is in ntdll as we can see in R8. This is because ntdll is always loaded on the same offset in all processes, regardless of the ASLR.

In the following iterations the function passed as parameter of NtQueueApcThread will be GlobalAtomGetAtomNameW, which will cause the target process to get the Atom that just created the malicious .dll and write it in the indicated zone, in such a way that it will write its contents inside the explorer.exe without raising suspicions.

First it will create an IAT for the shellcode.

| Address | Address | Comments |
|---|---|---|
| 00000000775CAAA0 | 00000000774DBFB0 | ntdll.NtMapViewOfSection |
| 00000000775CAAA8 | 00000000774DBFD0 | ntdll.ZwUnmapViewOfSection |
| 00000000775CAAB0 | 00000000774DBEB0 | ntdll.ZwAllocateVirtualMemory |
| 00000000775CAAB8 | 000000007494AF0 | ntdll.RtlCreateUserThread |
| 00000000775CAAC0 | 00000000774DBE10 | ntdll.NtSetEvent |
| 00000000775CAAC8 | 00000000774D9110 | ntdll.RtlCopyMemory |
| 00000000775CAAD0 | 00000000774DC230 | ntdll.ZwProtectVirtualMemory |
| 00000000775CAAD8 | 00000000774DBE20 | ntdll.NtClose |
| 00000000775CAAE0 | 0000000000000CA0 | |
| 00000000775CAAE8 | 0000000000000408 | |
| 00000000775CAAF0 | 0000000000000C28 | |
| 00000000775CAAF8 | 000000000000091A | |
| 00000000775CAB00 | 000000000000076C | |
| 00000000775CAB08 | 0000000000000758 | |
| 00000000775CAB10 | 00000000773D6BF0 | kernel32.GlobalGetAtomNameA |
| 00000000775CAB18 | 0044894438EC8348 | |
| 00000000775CAB20 | 0000000000000000 | |

Figure 20. IAT creation in explorer.exe

And after several iterations it will copy the shellcode in explorer.exe completely.

Figure 21. Shellcode in explorer.exe

## 5.4. Execution of the shellcode in the target process

Once the shellcode is copied to the explorer, it must be executed.

To do this, Dridex modifies the GlobalAtomGetAtomNameA function in the same way that it has injected the shellcode, using Atoms.

Original code of the function:

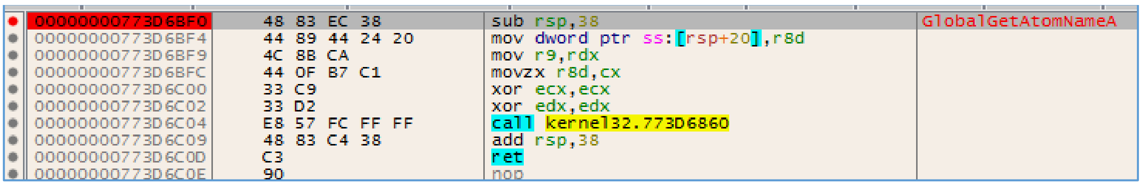



Figure 22. Original function

Here's how the function has been modified:

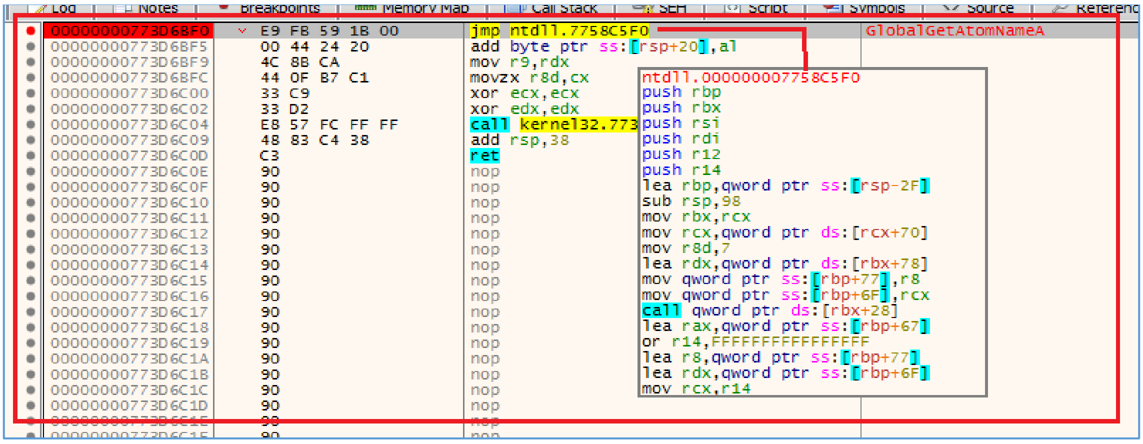



Figure 23. Modified function

As you can see, when you call GlobalAtomGetAtomNameA in explorer.exe the program will execute the shellcode.

After the modification, from the malicious .dll, a call will be made to GlobalAtomGetAtomNameA using NtQueueApcThread.

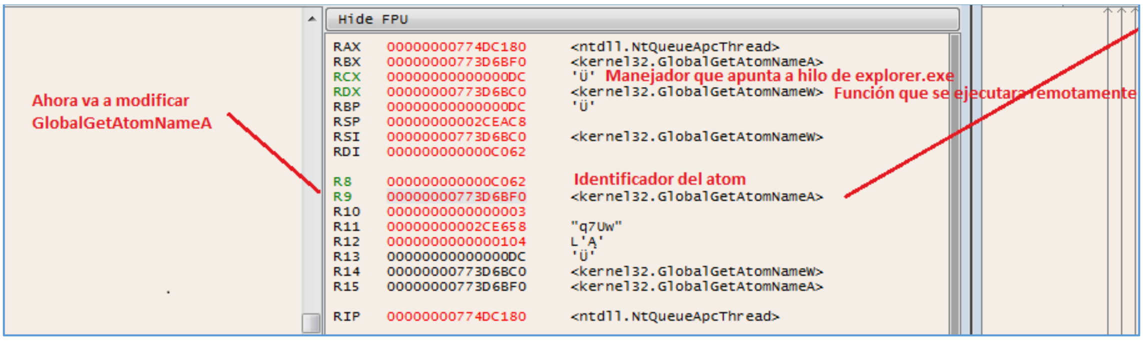



Figure 24. Remote execution of the shellcode

At this point the shellcode will start executing.

After this, GlobalAtomGetAtomNameA is returned to its original state, to avoid suspicion.

## 6. NETWORK CONNECTIONS

The Trojan, once it has been injected into the explorer.exe process, opens port 443 (usually used for the HTTPS protocol) and waits for some connection.

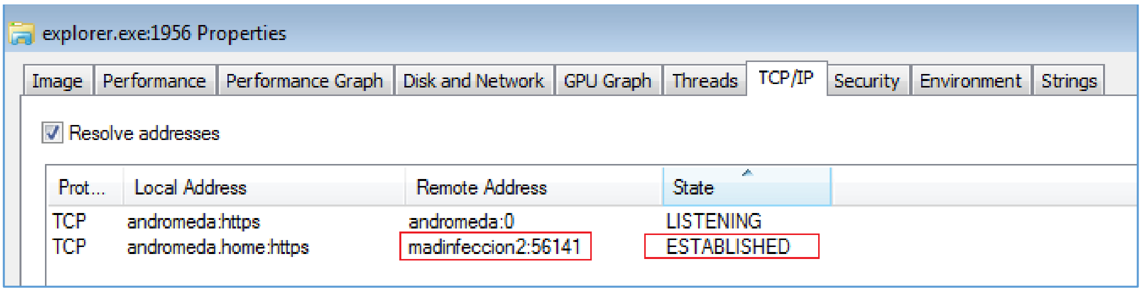



Figure 25. Port 443 opened

## 7. IOCs

To check if a computer has been compromised by this version of Dridex, the following points should be considered:

- The explorer.exe process has port 443 listening and there is a firewall rule in place allowing network traffic for that process.
- Directories that match the expression %SYSTEM%\[0-9] {4}, and contain a legitimate executable next to a .dll or .cpl file.
- Scheduled tasks that execute a file in path %SYSTEM%\[0-9] {4} in periods of 60 minutes.

## 8. REFERENCIAS


| | |
|---|---|
| [1] | **Inside the Dridex Malware Takedown**<br>Link: http://www.bankinfosecurity.com/dridex-botnet-disruption-lessons-learned-a-8594 |
| [2] | **Dridex v4 - AtomBombing and other surprises**<br>Link: https://www.virusbulletin.com/conference/vb2017/abstracts/dridex-v4-atombombing-and-other-surprises/ |
| [3] | **Dridex Banking Malware Sample Technical Analysis and Solution**<br>Link: http://blog.nsfocus.net/dridex-banking-malware-sample-technical-analysis-solution/ |