ESET Research White papers // September 2018

LOJAX

First UEFI rootkit found in the wild, courtesy of the Sednit group

TABLE OF CONTENTS

1. EXECUTIVE SUMMARY

Sednit also known as APT28, Sofacy, Strontium and Fancy Bear – has been operating since at least 2004, and has made headlines frequently in the past years: it is believed to be behind major, high profile attacks. For instance, several security companies [1] as well as the US Department of Justice [2] named the group as being responsible for the Democratic National Committee (DNC) hack just before the US 2016 elections. The group is also presumed to be behind the hacking of global television network TV5Monde [3], the World Anti-Doping Agency (WADA) email leak [4] and many others. Its targets are many and the group has a diversified set of malware in its toolbox several of which we have documented previously [5], but this white paper details the first time this group is known to have used a UEFI rootkit.

Key points in this white paper:

- Starting in at least early 2017, trojanized versions of an older userland agent of the popular LoJack anti-theft software from Absolute Software were found in the wild. We call this trojanized LoJack agent LoJax. LoJack attracted a lot of attention in recent years as it implements a UEFI/BIOS module as a persistence mechanism.
- The presence of known Sednit tools alongside LoJax samples as well as the fact that some of the C&C servers used by these trojanized agents were part of an earlier Sednit network infrastructure allows us to link this UEFI rootkit to the Sednit group with high confidence.
- Along with the LoJax agents, tools with the ability to read systems' UEFI firmware were found and in one case, this tool was able to dump, patch and overwrite part of the system's SPI flash memory. This tool's ultimate goal was to install a malicious UEFI module on a system whose SPI flash memory protections were vulnerable or misconfigured.
- This UEFI module has the responsibility to drop the LoJax agent on the system, making it the first Sednit UEFI rootkit identified. As it resides in the system's firmware, it can survive a Windows re-install as well as a hard drive replacement.
- There was at least one case where this rootkit was successfully installed in a system's SPI flash memory. To our knowledge, this is the first UEFI rootkit found in the wild.

For any inquiries related to this white paper, contact us at *threatintel@eset.com*

2. INTRODUCTION

The Sednit group is a resourceful APT group targeting people and organizations around the world. It has been in operation since at least 2004, using a wide range of malware families. For a complete description of the most prevalent tools this group uses, please refer to our Sednit white paper [5].

Throughout our multi-year tracking of this group, we released many reports on its activities, ranging from zero-day usage [6] to custom malware it develops, such as Zebrocy [7]. However, the component described in this white paper is in a league of its own.

There have been stories in the past of UEFI rootkits, such as "rkloader" described in a presentation [8] from the Hacking Team data leak or "DerStarke", a macOS EFI/UEFI boot implant described in the Vault7 leaks [9]. While we know of their existence, there has never been a published report detailing a real case of a victim compromised by such malware.

Not only were we able to confirm discovering an in the wild firmware including the malicious LoJax UEFI module, but we were also able to find the full toolchain that was presumably used to install it. It is interesting to note here that Sednit used the DownDelph bootkit in 2013 and 2014 as a persistence method for Downdelph, one of the group's first-stage backdoors. While the idea is similar, bootkits are no longer possible with the new UEFI implementation. Thus, these two components differ significantly in their behavior.

This white paper is divided into three sections. The first will deal with previous security research on LoJack/Computrace and how it could be used maliciously. The second section will examine the breadcrumbs found along our research route that ultimately led us to the UEFI rootkit. Finally, the third section will detail the different LoJax components and how they persist on a system even after a Windows re-install or a hard drive replacement.

Attribution

While many vendors have made attribution claims about the Sednit group in the past, ESET does not perform any type of geopolitical attribution. That was our position back when we published our white paper in 2016 [5] and is still the case today. As we wrote back then, performing attribution in a serious, scientific manner is a hard problem that is out of our scope as ESET security researchers. What we call "the Sednit group" is merely a set of software and the related network infrastructure, which we can hardly correlate authoritatively with any specific organization.

Victimology

We found a limited number of different LoJax samples during our research. Based on our telemetry data and on other Sednit tools found in the wild, we are confident that this particular module was rarely used compared to other malware components at their disposal. The targets were mostly government entities located in the Balkans as well as Central and Eastern Europe.

3. PREVIOUS RESEARCH ON COMPUTRACE/LOJACK

LoJack is anti-theft software made by Absolute Software Corporation. Earlier versions of this agent were known as Computrace. As its former name implies, once a user activated the service, the computer could call back to its C&C server and its user be notified of its location should it have gone missing or been stolen.

The rest of this section describes what LoJack architecture *used to* be. As only an old version of this software was trojanized by the threat actor, it makes sense to focus only on it. Also, Absolute Software issued a statement [10] in May 2018 stating that the vulnerabilities described below are not affecting recent versions of their agents.

Computrace attracted attention from the security community mostly because of its unusual persistence method. Since this software's intent is to protect a system hardware from theft, it is important that it resists OS re-installation or hard drive replacement. Thus, it is implemented as a UEFI/BIOS module, able to survive such events. This solution comes pre-installed in the firmware of a large portion of laptops' manufactured by various OEMs, waiting to be activated by its users. This activation step can be done through a BIOS option as depicted in Figure 1.

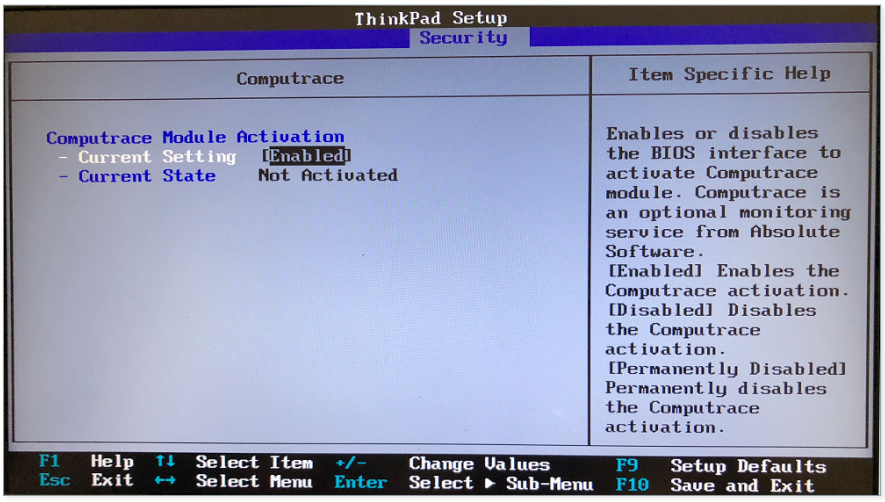

Figure 1 // Computrace BIOS activation

One of the first research reports providing information on how this solution is implemented was published in 2009 [11]. The global architecture of the product, at that time, was revealed, detailing how the UEFI/BIOS module was able to drop the userland agent on disk and how this agent was then able to call home by contacting a web server controlled by Absolute Software. The overall process of the LoJack/Computrace solution back then is best described in Figure 2.

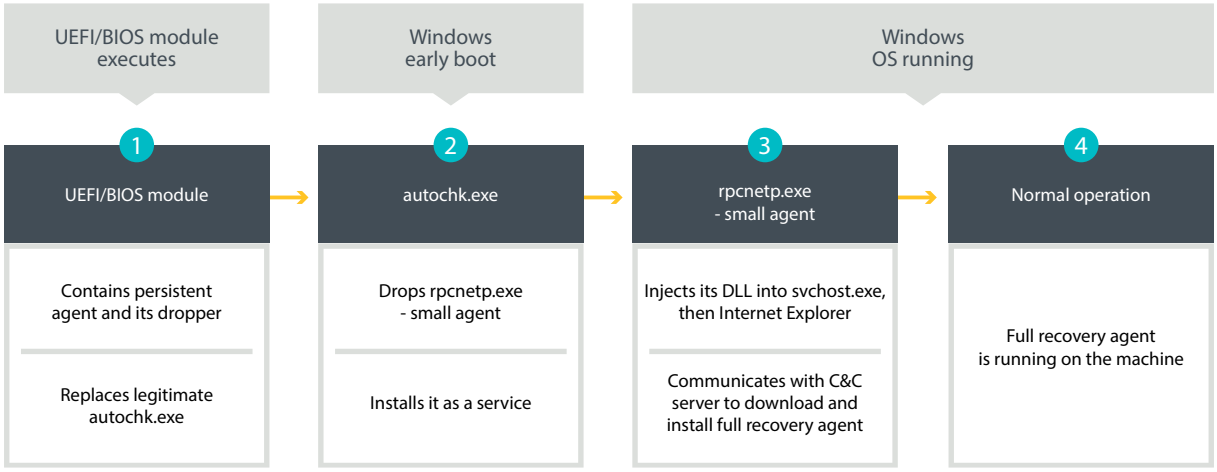

Figure 2 // LoJack persistence mechanism (circa 2008)

Here is a description of the different steps highlighted above:

1. At boot time, if activated, the UEFI/BIOS module is executed. It will try to find a FAT/FAT32/NTFS partition. Using an NTFS driver, it then creates a backup of `autochk.exe` and overwrites its content with a dropper responsible for installing the userland agent component. `autochk.exe` is a Windows executable that is run during the early stages of Windows initialization to check for possible hard drive corruption.
2. When the modified `autochk.exe` is run, its main purpose is to drop the small agent `rpcnetp.exe` and add it as a service so that it is started at each reboot. The last step of this component is to restore the original version of `autochk.exe`.
3. The small agent, `rpcnetp.exe`, is a small executable whose main purpose is to ensure that the main agent is running. If not, it will try to connect to Absolute Software's C&C server to download and execute it. The small agent will first make a copy of itself and modify the PE header so that it becomes a dynamic-link library (DLL). This DLL is then loaded in memory and it will spawn a `svchost.exe` process and inject the DLL there. It will then spawn an Internet Explorer `iexplore.exe` process and again inject its DLL into it. This last process will then be used to communicate over the Internet. The Computrace small agent's behavior of injecting code into foreign processes is commonly seen in malware and rarely associated with legitimate, reputable software.
4. The full featured agent is now running on the system and implements Computrace's various tracking and recovery functions.

This overall process, along with a detailed description of the network protocol used between the small agent and its C&C server, was published in 2014 [12]. As no authentication mechanism exists, if adversaries could control the server with which the small agent communicates, they could make it download and execute arbitrary code. There are several different mechanisms allowing an attacker to communicate directly with the small agent. The one that is the most relevant to our discussion involves how the address of the C&C server is retrieved by the small agent. In fact, this information is stored in a configuration file hardcoded in the executable itself.

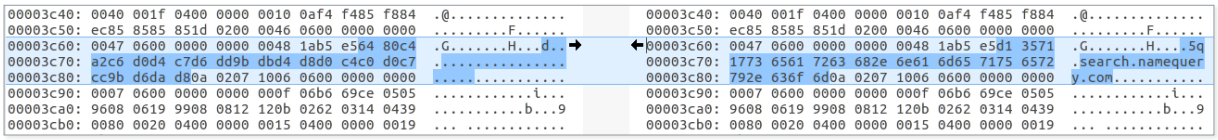

Figure 3 // Encrypted LoJack configuration file on the right with partial decryption on the right

Figure 3 shows both the encrypted and decrypted LoJack small agent configuration file. The "encryption" method used is a simple XOR operation using a one-byte key. This key, 0xB5, is the same for all small agents studied. As seen in Figure 3, the C&C domain name is clearly visible. The four bytes preceding that comprise a C&C server IP address. As there is no validation done on the configuration file content, adversaries with write access to `%WINDIR%` can change its content so that the small agent contacts a C&C server under their control instead of the legitimate one. By understanding the network protocol, it is then possible to make the small agent download and execute arbitrary code.

Although these risks were identified a long time ago, no noteworthy usage of this security risk was seen in the wild until recently.

LoJack becomes LoJax

In May 2018, an Arbor Networks blogpost [13] describing several trojanized samples of the LoJack small agent, `rpcnetp.exe`, was published. These malicious samples communicated with a malicious C&C server instead of the legitimate Absolute Software one, because their hardcoded configuration settings had been altered. Some of the domains found in LoJax samples had been seen before: they were used in late 2017 as C&C domains for the notorious Sednit first-stage backdoor, SedUploader. Figure 4 shows an example of the modified configuration in one such LoJax small agent.

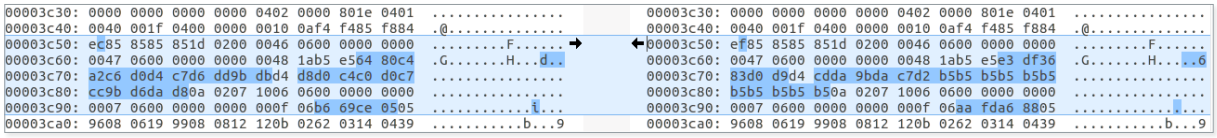

Figure 4 // Legitimate configuration file on the left, modified one on the right

The differences between the legitimate and trojanized agent are so small that the figures above actually show most of the changes between them. All the LoJax small agent samples we could recover are trojanizing the exact same legitimate sample of the Computrace small agent `rpcnetp.exe`. They all have the same compilation timestamp and only a few tens of bytes are different from the original one. Besides the modifications to the configuration file, the other changes include timer values specifying the intervals between connections to the C&C server.

At the time the blog was published, we had found different LoJax small agents targeting different entities in the Balkans as well as Central and Eastern Europe, but had no idea how they were installed. Of course, the obvious explanation was that some well-known Sednit backdoor installed them. After all, since LoJack was a well-known tool, it was whitelisted by many AV vendors. Thus, even if only the small agent was used in this campaign and that it could not survive a Windows re-install, it still had the benefit of being less likely to be flagged as malicious. However, what if the compromise was deeper than that? What if they tried to mimic the LoJack solution and go all the way to the system's firmware?

4. THE HUNT FOR A LOWER-LEVEL COMPONENT

We were able to uncover LoJax campaigns targeting a few organizations in the Balkans as well as Central and Eastern Europe. In all of them, we were able to find traces of other Sednit malware detections, namely:

- SedUploader, a first-stage backdoor
- XAgent, Sednit's flagship backdoor
- Xtunnel, a network proxy tool that can relay any kind of network traffic between a C&C server on the Internet and an endpoint computer inside a local network

Although we detected traces of Sednit tools on most of the systems we examined that were targeted by LoJax, we found a couple of systems where only LoJax was present. Thus, we can infer that in some cases, LoJax was used as a stand-alone tool, presumably as an additional backdoor used to regain admittance to the network should Sednit operators lose access.

As XAgent is routinely used to drop additional modules on a compromised system, it is tempting to jump to the conclusion that LoJax samples are dropped in the same way and that there are no other mechanisms in place. This would mean that the only part that was inspired by the LoJack solution would be the small agent. However, shortly after we started our analysis, we found some clues that led us to believe the inspiration went a bit further.

RWEverything driver (RwDrv) and info_efi.exe

The first piece of evidence comes from a custom tool created by the malicious actors that, when executed, dumps information about low level system settings to a text file. This tool was found alongside some LoJax samples. The following figure shows a snippet of the log file produced by this tool, aptly named `info_efi.exe`.

```
Get SMBIOS..
SMBIOS:
^@^X^@^@^A^BRê^C^@■Ð    |^@^@^@^@■^G^D^F^@^@Phoenix Technologies LTD^@6.00^@05/19/2017^@^@^A^[^A^@^A^B^C
^DUM^S■ÍÛW^MÓ\■^I~-(f^F^@^@VMware, Inc.^@VMware Virtual Platform^@None^@VMware-
                    ^@^@^B^O^B^@^A^B^C^D^@^@^@^@^@^A^@Intel Corporation^@440BX Desktop Reference Platf
orm^@None^@None^@^@^C^U^C^@^A^A^B^C^D^C^C^C^C4^R^@^@^@^@^@^@No Enclosure^@N/A^@None^@No Asset Tag^@^@^D*
^D^@^A^C^B^Bé^F  ^@ÿû■^O^C^B^@^@u¸^KA^D^U^@^U^@ÿÿ^@^@^@^A^A^@$^@^B^@CPU #000^@GenuineIntel^@Intel(R) Cor
e(TM) i5-7400 CPU @ 3.00GHz^@^@^E.^E^@^C^D^C^C^O^L^@^X^E^B^O^F^@^G^@^H^@          ^@
^@^K^@^L^@^M^@^N^@^O^@^P^@^Q^@^R^@^S^@^T^@^D^@^@^F^L^F^@^Aÿ^@^P^A
```

Figure 5 // Excerpt of log file as generated by info_efi.exe

In order to read this type of information, this tool embeds a driver called `RwDrv.sys`. This kernel driver is bundled with RWEverything, a free utility available on the web [14] that can be used to read information on almost all the computer low-level settings, including PCI Express, Memory, PCI Option ROMs, etc. As this kernel driver belongs to legitimate software, it is signed with a valid code-signing certificate.

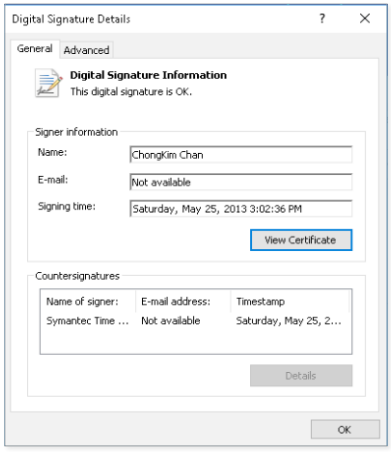

Figure 6 // RwDrv.sys code-signing certificate

RWEverything software comes with a GUI that lets the user access all of this data.

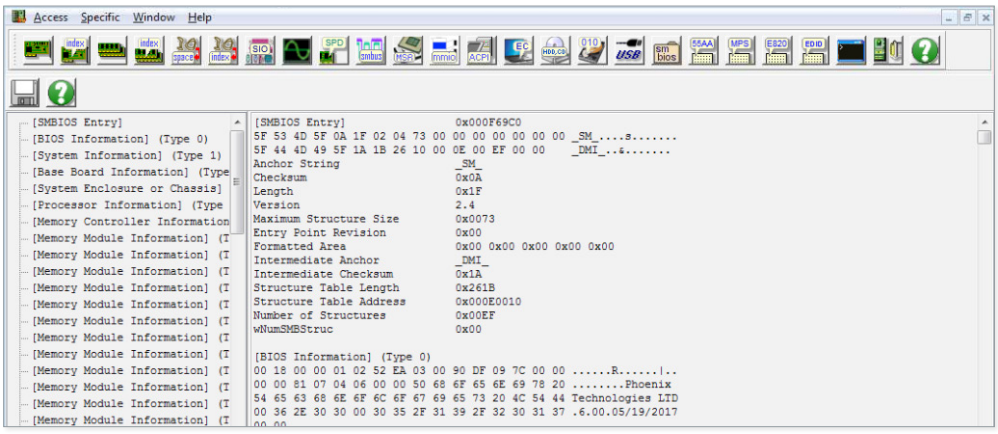

Figure 7 // Screenshot of RWEverything GUI

The `info_efi` tool discovery was the first sign that a LoJax UEFI module might exist. When trying to update a system's firmware, it is crucial to have information about the firmware vendor, its version, etc. As there are known vulnerabilities allowing userland processes to access and modify the content of the SPI flash memory where the UEFI modules are stored, getting data about the system's hardware is the first step towards a successful attack.

The final lead that allowed us to find Sednit's first UEFI rootkit was two different tools — one used to dump the SPI flash memory and one to write to it.

Dumping the SPI flash memory

The first piece of the puzzle was a file called `ReWriter_read.exe`. This file contained all the code required to dump a system SPI flash memory using the RWEverything driver, `RwDrv.sys`. In order for the device driver to perform the required operations, the dumper tool must send the correct I/O control (IOCTL) codes. While `RwDrv.sys` supports many different IOCTL codes, both the dumper and writer tool described in this section and the next use only four of them.

Table 1 **RwDrv.sys supported IOCTLs**

IOCTL code	Description
0x22280c	Writes to memory mapped I/O space
0x222808	Reads from memory mapped I/O space
0x222840	Reads a dword from given PCI Configuration Register
0x222834	Writes a byte to given PCI Configuration Register

`ReWriter_read` first creates a service with the embedded kernel driver `RwDrv.sys` and logs some information on the UEFI/BIOS configuration, namely the value of three fields contained in the BIOS Control Register (BIOS_CNTL): BIOS Lock Enable (BLE), BIOS Write Enable (BIOSWE) and SMM BIOS Write Protect Disable (SMM_BWP). While `ReWrite_read` does not use these values at all, the following sections will highlight why these three fields are of interest to this tool.

The tool's next task is to retrieve the BIOS region base address on the SPI flash memory as well as its size. This information is contained in the SPI Host Interface register "BIOS Flash Primary Region". All SPI Host Interface registers are memory-mapped in the Root Complex Register Block (RCRB) whose base address can be retrieved by reading the correct PCI Configuration Register. `ReWriter_read` obtains this address by using RwDrv IOCTL 0x22840 and reading the correct offset (0xF0 in our case). Once the BIOS region base address and size are known, the dump tool reads the relevant content of the SPI flash memory and writes it to a file on disk. The reading process of the SPI flash memory is illustrated in Figure 8. Please refer to the *Glossary* for the expansions of the acronyms used below.

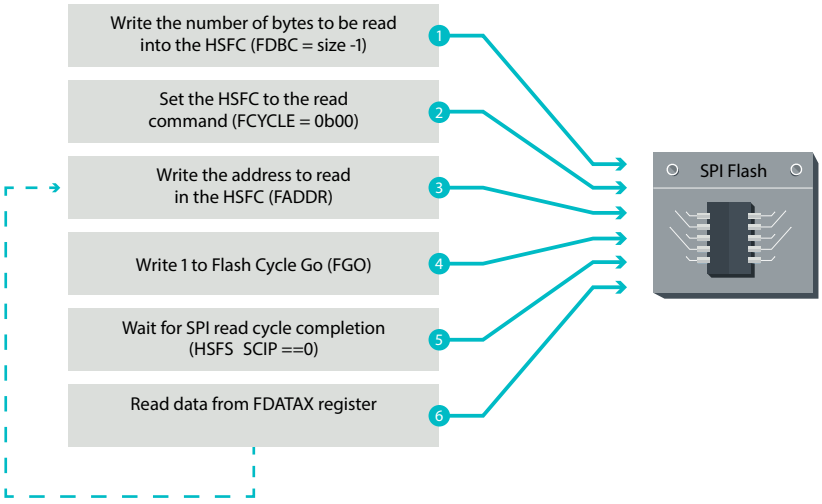

Figure 8 // Operation sequence to read from the SPI flash memory

Except for the first two steps that are executed only once, these operations are repeated in a loop until all the data is read from the SPI flash memory. This process is also well described in [15]. `ReWriter_read` will then validate the size of the dumped image. It will parse the image Flash descriptor to get the memory ranges of the BIOS, the Gigabit Ethernet (GbE) and the Management Engine (ME) regions. Adding the size of these three regions allows the dumper tool to compute the size of the entire content of the SPI flash memory. If this size is equal to the size obtained by reading the BIOS Flash Primary region register, the image is considered valid.

Patching the UEFI firmware

The second piece of the puzzle is a file called `ReWriter_binary.exe`. This file contains the evidence we were missing to prove that Sednit's operators went as far as targeting the firmware. This file contains the code to patch the dumped UEFI image and write the trojanized version back to the SPI flash memory. This section will detail the inner workings of this binary.

Once the flash memory content has been dumped and successfully validated by the aforementioned dumper tool, the malicious UEFI module is added to the image. To do so, the UEFI image must first be parsed to extract the information required for this task.

The data stored in the UEFI image are laid out in volumes using Firmware File System (FFS). As its name suggests, it is a file system specifically tailored for storing firmware images. Volumes contain files identified by GUIDs. Each file is usually composed of multiple sections, one of which contains the actual PE/COFF executable that is the UEFI image. To help visualize this layout, here's a screenshot from UEFITool [16], an open source project for manipulating UEFI firmware images.

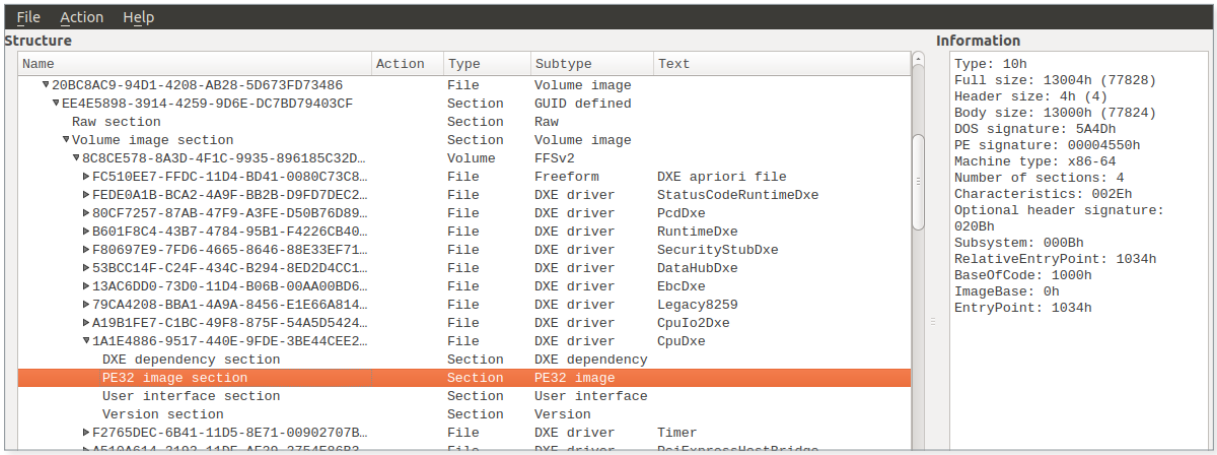

Figure 9 // Example UEFI firmware image loaded in UEFITool

`ReWriter_binary` parses all of the firmware volumes found in the BIOS region of the SPI flash memory searching for specific files:

- Ip4Dxe (8f92960f-2880-4659-b857-915a8901bdc8)
- NtfsDxe (768bedfd-7b4b-4c9f-b2ff-6377e3387243)
- SmiFlash (bc327dbd-b982-4f55-9f79-056ad7e987c5)
- DXE Core

```
if ( FileType == EFI_FV_FILETYPE_DRIVER )
{
  // If FileGuid == 8f92960f-2880-4659-b857-915a8901bdc8 (Ip4Dxe)
  if ( !((*&FileHeader->Name.Data4[4] - *&gIp4DxeGuid.Data4[4]) |
         (*&FileHeader->Name.Data2 - *&gIp4DxeGuid.Data2) |
         (FileHeader->Name.Data1 - gIp4DxeGuid.Data1) |
         (*FileHeader->Name.Data4 - *gIp4DxeGuid.Data4)) )
  {
    *Ip4DxeOffset = Index;
    Ip4DxeOffset[1] = 0;
    *Ip4DxeFileSize = FileSize;
    *Ip4DxeGuid = FileHeader->Name;
    Ip4DxeFound = 1;
  }
  // If FileGuid == 768bedfd-7b4b-4c9f-b2ff-6377e3387243 (NTFS)
  if ( !((*&FileHeader->Name.Data4[4] - *&gNTFSGuid.Data4[4]) |
         (*&FileHeader->Name.Data2 - *&gNTFSGuid.Data2) |
         (FileHeader->Name.Data1 - gNTFSGuid.Data1) |
         (*FileHeader->Name.Data4 - *gNTFSGuid.Data4)) )
  {
    *NtfsFileOffset = Index;
    NtfsFileOffset[1] = 0;
    *NtfsFileSize = *FileHeader->Size;
    *(NtfsFileSize + 2) = FileHeader->Size[2];
  }
  // If FileGuid == bc327dbd-b982-4f55-9f79-056ad7e987c5 (SmiFlash)
  if ( !((*&FileHeader->Name.Data4[4] - *&gSmiflashGuid.Data4[4]) |
         (FileHeader->Name.Data1 - gSmiflashGuid.Data1) |
         (*FileHeader->Name.Data4 - *gSmiflashGuid.Data4) |
         (*&FileHeader->Name.Data2 - *&gSmiflashGuid.Data2)) )
  {
    *SmiFlashOffset = Index;
    SmiFlashOffset[1] = 0;
  }
}
// If it's the DXE Core
else if ( FileType == EFI_FV_FILETYPE_DXE_CORE )
{
  *IsDxeCoreFirmwareVolume = 1;
```

Figure 10 // Hex-Rays decompiler output for the routine parsing the firmware volumes

Ip4Dxe and NtfsDxe are DXE drivers. In UEFI firmware, DXE drivers are PE/COFF images that are either meant to abstract the hardware or to produce services that can be used by other DXE drivers or by UEFI applications. Such drivers are discovered and loaded by the DXE Foundation through the DXE Dispatcher (DXE Core) early in the boot process. After completion of this phase, all services expected to be available by UEFI applications, such as an OS loader, are in place. Usually, all the DXE drivers are stored in the same volume. However, the DXE dispatcher may be on a separate one.

`ReWriter_binary` looks for Ip4Dxe only as an indication that the volume being parsed is the volume that contains the DXE drivers. As we will describe later, this volume will be a candidate for the installation of the malicious DXE driver. It also looks for DXE Core and adds the volume where it's located as another candidate volume for where to write the rootkit. The free space available on each of these volumes is stored and is used later to verify whether there is enough space available to add the malicious driver.

NtfsDxe is the AMI NTFS DXE driver. If present in a firmware volume, its location is stored and is later used to remove the file from the volume. We will see why the tool removes this driver in the section dedicated to the analysis of the UEFI rootkit.

As for the SmiFlash image, the information related to this image is stored but is not used anywhere in the malware. Interestingly, this image is vulnerable [17]. Thus, we believe that Sednit's operators might be working on some exploit for these vulnerabilities. This could allow them to write to the SPI flash memory even on properly configured systems. As we will describe later: in its current state, the tool is only able to write to the BIOS region of misconfigured or fairly old systems (on motherboards older than Platform Controller Hub chipsets introduced around 2008).

After the extraction of required metadata, `ReWriter_binary` proceeds to patching the dumped UEFI image, adding its malicious DXE driver. First, it creates a file header structure (EFI_FFS_FILE_HEADER). Then, it selects the destination volume based on the location of Ip4Dxe and DXE Core as well as the free space available on these volumes. `ReWriter_binary` embeds a compressed section containing the PE image and a *User interface* section specifying the human-readable name of the file: SecDxe. The compressed section is appended to the file header and written at the end of the volume, where the volume free space is located. Figure 11 shows the file structure as viewed with UEFITool.

```
▼682894B5-6B70-4EBA-9E90-A607E5676297    File      DXE driver     SecDxe
 ▼Compressed section                     Section   Compressed
   PE32 image section                    Section   PE32 image
   User interface section                Section   User interface
```

Figure 11 // UEFITool view of SecDxe file

Finally, if the NtfsDxe driver is present in the image, it is removed. Since the firmware file system stores files and their content sequentially, it is a fairly simple process:

- It finds the offset to the free space at the end of the volume
- The NtfsDxe image is overwritten by 0xFF bytes
- The trailing part of the firmware volume is copied starting at the offset where NtfsDxe was located
- The remainder of the file system is padded with 0xFF bytes, which means free space

Writing the patched firmware back to the SPI flash memory

Once the dumped firmware image is successfully modified, the next step is to write it back to the SPI flash memory. Before we dive into this process, we need to introduce some of the BIOS write protections that are relevant to this case. Other existing mechanisms, like BIOS Range Write Protection, are left aside since they are not checked by `ReWriter_binary`.

The platform exposes multiple protection mechanisms to block unauthorized attempts to write to the BIOS region. These mechanisms are nonetheless not enabled by default. The firmware is responsible for configuring them properly. Such configurations are exposed via the BIOS control register (BIOS_CNTL). This register contains the BIOS Write Enable (BIOSWE) bit, which needs to be set to 1 to be able to write to the BIOS region of the SPI flash memory. Since the platform shouldn't allow all attempts to write to the BIOS region, another bit is available in the BIOS_CNTL to protect BIOSWE: the BIOS Lock Enable (BLE). When enabled, this mechanism is meant to lock the BIOSWE bit to 0. However, the implementation is vulnerable. Indeed, when there is a request to set the BIOSWE bit to 1, the BIOSWE bit is actually set to 1. Only then does the platform issue a System Management Interrupt (SMI) and the handler for this SMI is responsible for setting the BIOSWE bit back to 0.

Multiple issues arise from this implementation. First, the implementation of the SMI handler is left to the firmware developers. Thus, if the firmware doesn't implement this handler, the BLE bit is useless since there won't be any routine setting the BIOSWE bit back to 0. Second, there's a race condition vulnerability [18] that allows complete bypass of this mechanism, even if the SMI handler is properly implemented. To exploit this vulnerability, an attacker needs to start a thread that continuously sets BIOSWE to 1 while another thread writes data to the SPI flash memory. According to Kallenberg and Wojtczuk's paper [19], this attack works on multi-core processors and can also succeed on a single-core processor if it has hyper-threading enabled.

To remediate this issue, a new protection mechanism configured via the BIOS_CNTL was added to the platform. It was introduced in the Platform Controller Hub (PCH) family of Intel chipsets. If its configuration bit is set, SMM BIOS Write Protect Disable (SMM_BWP) will ensure that the BIOS region is writable only if all the cores are running in System Management Mode (SMM) and BIOSWE is set to 1. This effectively protects a system against the race condition vulnerability explained above. However, as is the case for BLE, SMM_BWP needs to be activated by the firmware. Hence, a firmware that doesn't configure these mechanisms properly leaves the system at risk of unauthorized writes to the BIOS region.

`ReWriter_binary` reads the content of the BIOS control register to choose the proper path to take. It first checks if BIOSWE is set. If it is, it goes to the writing phase. If BIOSWE is disabled, it checks the value of the BLE bit. If it is not set, it flips the BIOSWE bit and starts to write the patched firmware. If BLE is set, it makes sure that SMM_BWP is disabled and exploits the race condition mentioned above. If the SMM_BWP bit is set, it fails. Figure 12 illustrates this process.

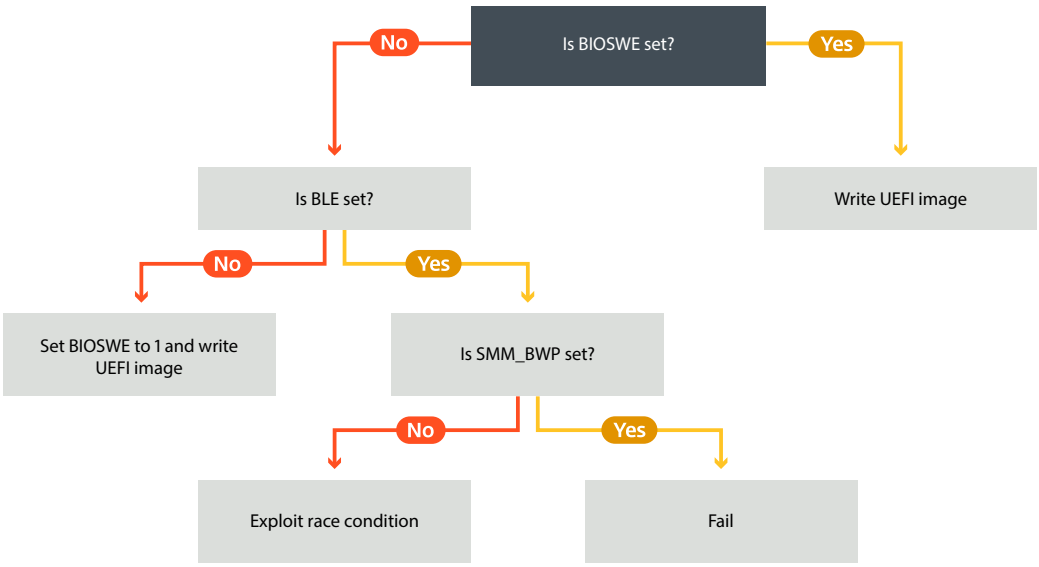

Figure 12 // Decision tree of the writing process

Assuming that the exact build of `ReWriter_binary` we analyzed was the one that was used to deploy the UEFI rootkit, we can conclude that either the firmware did not properly configure the BIOS write protection mechanisms or the victim's machine had a chipset older than the Platform Controller Hub. `ReWriter_binary` wouldn't have succeeded at flashing the UEFI firmware on a properly configured modern system. However, looking for the vulnerable SmiFlash UEFI image when parsing the UEFI firmware volumes suggests that the operators might have been fiddling with more advanced techniques to bypass BIOS write protections [17].

Very similar to the read operation described above, the following sequence of events occurs to write to the SPI flash memory:

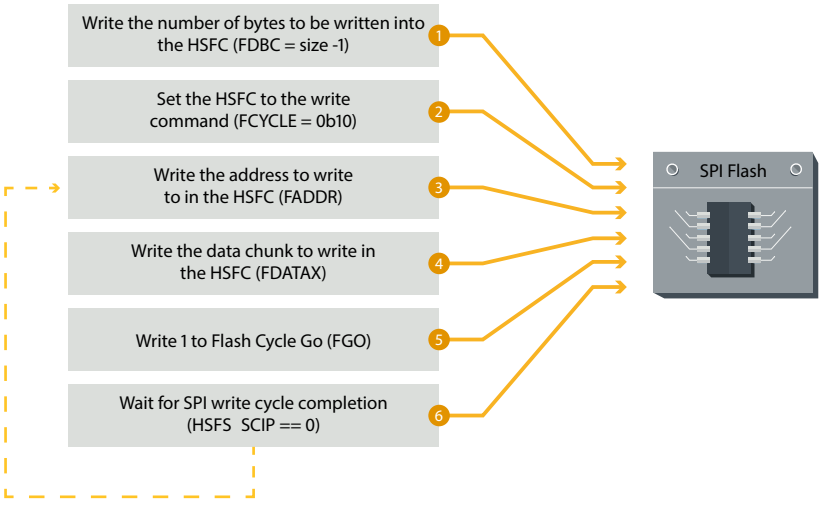

Figure 13 // Operation sequence to write to the SPI flash memory

Except for the first two steps that are only executed once, these operations are repeated in a loop until all the data is written to the SPI flash memory.

When the writing process is done, the content of the SPI flash memory is once again dumped into the file `image.bin`. The same integrity check that was done by `ReWriter_read` is performed on the new dumped image. Then, the image read from the SPI flash memory is compared to the patched image in-memory. If some bytes differ, the address where it happened is logged. Whether they differ or not has no effect on the execution of the malware. It is just logged for the operators to know what happened.

As final steps, this registry key is set:

```
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manager\BootExecute =
"autocheck autochk *"
```

Then, RwDrv service is stopped and uninstalled. It is important that the Windows Registry value is set to that string because the UEFI Rootkit looks for that exact string to modify it and thus execute its payload during Windows startup. We will give more details about this modification of the Windows Registry when we describe the UEFI Rootkit and its payloads.

5. LOJAX TECHNICAL ANALYSIS

While the tool to dump, patch and write to the SPI flash memory is customized for a particular firmware image and cannot be re-used easily on any given system, the full UEFI module can be extracted from it. The first step we did after recovering this module was to go through our telemetry to see whether we had seen this module before. However, as this is a UEFI module, we had to rely on the new ESET UEFI scanner that is able to access and scan a system's firmware. Using telemetry coming from this module, we were able to find at least one case where the Sednit's UEFI module was installed on a system, meaning that this UEFI rootkit was truly deployed in the wild.

We do not know for sure how the different tools ended up on the compromised systems. The most likely guess at this point is that it was dropped by another tool, likely XAgent, as part of the post-compromise steps done by the operators. Since the dumper and the writer tools were found on the same system but at different times, it is likely the operators worked in two steps. First, they dumped the firmware on the target machine, made sure that their patching tool would work fine before uploading it again and patching the firmware for real. While we were able to find only one version of the dumper and writer tools, there is a possibility that different versions exist for different vulnerable firmware they were able to locate.

Figure 14 gives a high-level overview of the UEFI rootkit workflow until the OS boots. First, SecDxe DXE driver is loaded by the DXE dispatcher. It sets a Notify function callback on the `EFI_EVENT_GROUP_READY_TO_BOOT` event group. When the firmware is about to choose a boot device and to run the OS loader, the Notify function is called. The callback does three things:

- It loads an embedded NTFS DXE driver to be able to access and write to NTFS partitions
- It writes two files to the Windows NTFS partition: `rpcnetp.exe` and `autoche.exe`
- It modifies this registry key `'HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manager\BootExecute'`:
  - Before: `'autocheck autochk *'`
  - After: `'autocheck autoche *'`.

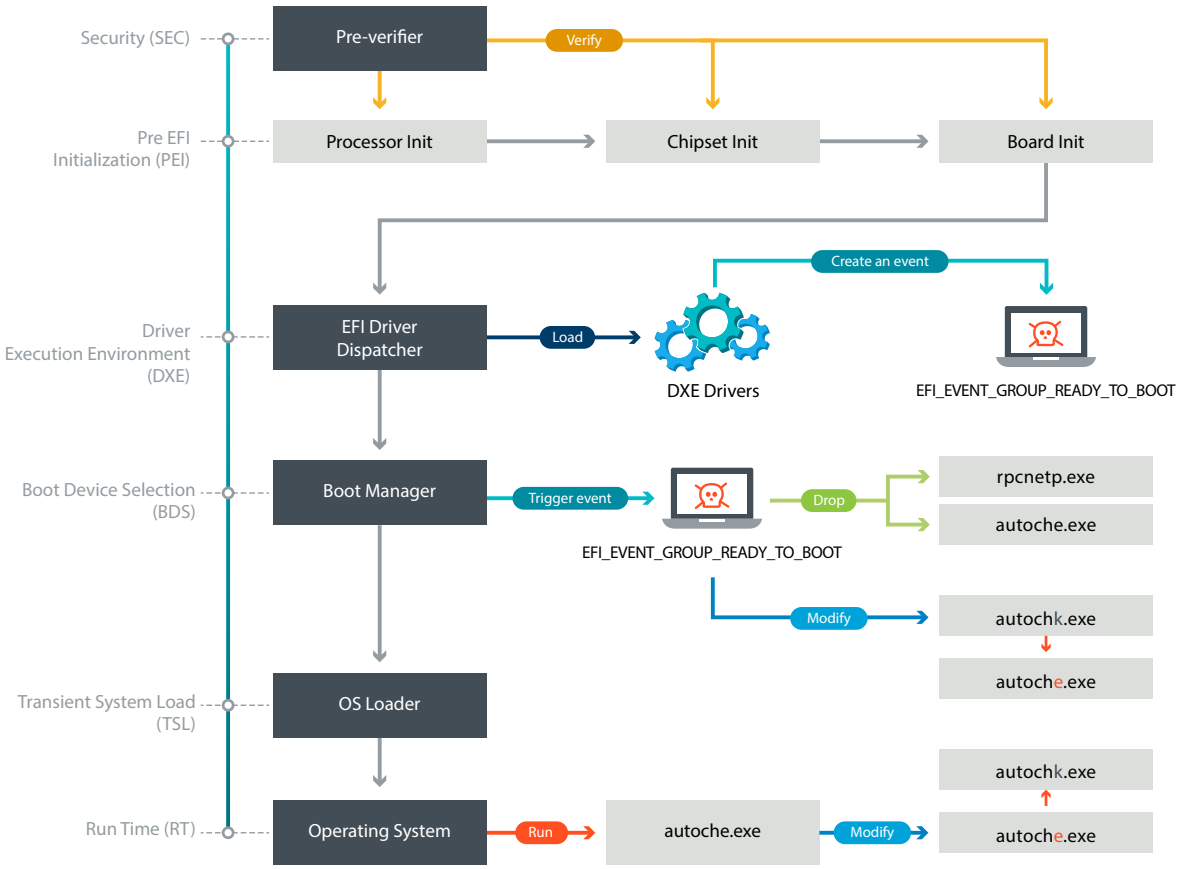

Figure 14 // Boot process of a system infected by the UEFI rootkit

SecDxe: The malicious DXE driver

Now that we have covered the details of the deployment of the UEFI rootkit, this section focuses on the chain of events that occurs on a compromised machine. We adopt a bottom-up approach where we begin by describing the UEFI rootkit itself and then follow the chain of events up to the final payloads that are deployed at the operating system level.

Sednit's UEFI rootkit is a DXE driver, identified by the GUID `682894B5-6B70-4EBA-9E90-A607E5676297`. It is unsigned; thus it cannot run on a system with Secure Boot enabled. Once deployed in one of the firmware volumes, the DXE Foundation loads it every time the system boots.

SecDxe is a small DXE driver that mainly does two things. It installs a protocol identified by the GUID `832d9b4d-d8d5-425f-bd52-5c5afb2c85dc` that is never used. Then, it creates an event associated with a Notify function. The Notify function is set to be called when the `EFI_EVENT_GROUP_READY_TO_BOOT` event group is signaled. This event group is signaled by the boot manager when it is about to choose a device to boot from.

```
__int64 __fastcall fnCreateEventEx(EFI_TPL NotifyTpl_1, EFI_EVENT_NOTIFY NotifyFunction_, __int64 NotifyContext_, EFI_EVENT *Event_1)
{
  int *v4; // r8
  int *v5; // r9
  __int64 result; // rax
  EFI_TPL NotifyTpl; // [rsp+50h] [rbp+8h]
  EFI_EVENT_NOTIFY NotifyFunction; // [rsp+58h] [rbp+10h]
  __int64 NotifyContext; // [rsp+60h] [rbp+18h]
  EFI_EVENT *Event; // [rsp+68h] [rbp+20h]

  Event = Event_1;
  NotifyContext = NotifyContext_;
  NotifyFunction = NotifyFunction_;
  NotifyTpl = NotifyTpl_1;
  if ( fnRetZero() && !Event )
    sub_19CC("c:\\edk2\\MdePkg\\Library\\UefiLib\\UefiNotTiano.c", &byte_40[117], "ReadyToBootEvent != ((void *) 0)");
  if ( gEfiSystemTable->Hdr.Revision >= 0x20000 )
  {
    if ( NotifyFunction )
      result = (gEfiBootServices->CreateEventEx)(
                 EFI_EVENT_NOTIFY_SIGNAL,
                 NotifyTpl,
                 NotifyFunction,
                 NotifyContext,
                 &gEfiEventReadyToBootGuid,
                 Event);
    else
      result = (gEfiBootServices->CreateEventEx)(
                 EFI_EVENT_NOTIFY_SIGNAL,
                 NotifyTpl,
                 fnDefaultNotifyFunction,
                 NotifyContext,
                 &gEfiEventReadyToBootGuid,
                 Event);
  }
  else
  {
    if ( fnRetZero() && sub_19E0(0x80000000i64) )
      sub_19B4(0x80000000i64, "EFI1.1 can't support ReadyToBootEvent!", v4, v5);
    if ( fnRetZero() )
      sub_19CC("c:\\edk2\\MdePkg\\Library\\UefiLib\\UefiNotTiano.c", (&dword_B8 + 1), "((BOOLEAN)(0==1))");
    result = EFI_UNSUPPORTED;
  }
  return result;
}
```

Figure 15 // Hex-Rays decompiler output for the routine creating the event

The Notify function implements the malicious behavior of Sednit's UEFI rootkit. It writes the payloads to Windows' NTFS file system. Since UEFI firmware normally deals solely with the EFI system partition, an NTFS driver usually is not included. Only FAT-based file systems are supported as boot partitions. Thus, it is not mandatory for a UEFI firmware to ship with NTFS drivers. For that reason, SecDxe embeds its own NTFS driver. This driver is first loaded and connected to the disk device. Hence, it installs an `EFI_SIMPLE_FILE_SYSTEM_PROTOCOL` on disk devices with NTFS partitions, enabling file-based access to it.

Now that everything is in place to write files on the Windows partition, SecDxe drops `rpcnetp.exe` and `autoche.exe`. Next, `rpcnetp.exe` is installed to `%WINDIR%\SysWOW64` on 64-bit Windows versions or to `%WINDIR%\System32` on 32-bit versions. As for `autoche.exe`, it is installed to `%WINDIR%\SysWOW64`. Figure 16 shows the routine responsible for writing these files to disk.

```
EfiSimpleFileSystemProtocol->OpenVolume(EfiSimpleFileSystemProtocol, Root);
v2 = (*Root)->Open(*Root, WindowsDirHandle, WindowsDir, 1ui64, 0x10ui64);
if ( !v2 )
{
  if ( (*WindowsDirHandle)->Open(*WindowsDirHandle, SystemDirHandle, SysWOW64Dir, 1ui64, 0x10ui64) )
  {
    if ( !(*WindowsDirHandle)->Open(*WindowsDirHandle, SystemDirHandle, System32Dir, 1ui64, 0x10ui64) )
    {
      if ( (*SystemDirHandle)->Open(*SystemDirHandle, NewHandle, L"rpcnetp.exe", 1ui64, 0x20ui64) )
      {
        (*SystemDirHandle)->Open(*SystemDirHandle, NewHandle, L"rpcnetp.exe", 0x8000000000000003ui64, 0x20ui64);
        (*NewHandle)->Write(*NewHandle, &RpcnetpFileSize, &gRpcnetp_exe);
      }
      (*NewHandle)->Close(*NewHandle);
    }
  }
  else
  {
    if ( (*SystemDirHandle)->Open(*SystemDirHandle, NewHandle, L"rpcnetp.exe", 1ui64, 0x20ui64) )
    {
      (*SystemDirHandle)->Open(*SystemDirHandle, NewHandle, L"rpcnetp.exe", 0x8000000000000003ui64, 0x20ui64);
      (*NewHandle)->Write(*NewHandle, &RpcnetpFileSize, &gRpcnetp_exe);
    }
    (*NewHandle)->Close(*NewHandle);
  }
  v2 = (*WindowsDirHandle)->Open(*WindowsDirHandle, SystemDirHandle, System32Dir, 1ui64, 0x10ui64);
  if ( !v2 )
  {
    if ( (*SystemDirHandle)->Open(*SystemDirHandle, NewHandle, L"autoche.exe", 1ui64, 6ui64) )
    {
      (*SystemDirHandle)->Open(*SystemDirHandle, NewHandle, L"autoche.exe", 0x8000000000000003ui64, 6ui64);
      (*NewHandle)->Write(*NewHandle, &AutocheFileSize, &gAutoche_exe);
    }
    v2 = (*NewHandle)->Close(*NewHandle);
  }
}
```

Figure 16 // Hex-Rays decompiler output for the routine writing files to disk

SecDxe then opens `%WINDIR%\System32\config\SYSTEM`, which is the file backing the `HKLM\SYSTEM` registry hive. It parses the file until it finds `'autocheck autochk *'` and replaces the `'k'` of `'autochk'` with `'e'`. This sets `'HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manager\BootExecute'` to `'autocheck autoche *'`. Next time Windows boots, autoche.exe will be launched instead of `autochk.exe`.

Hacking Team's NTFS driver

As previously discussed, SecDxe module embeds an NTFS driver. There is strong evidence that Sednit's operators did not write their own driver, but rather compiled their own copy of Hacking Team's leaked NTFS DXE driver.

Hacking Team's NTFS driver uses the ntfs-3g open source project at its core. It is merely a wrapper around it to make it work as a UEFI DXE Driver. As such, the INF file build information of Hacking Team's driver lists filenames from the ntfs-3g project. SecDxe's NTFS driver strings also lists many of these filenames:

- `c:\edk2\NtfsPkg\NtfsDxe\ntfs\inode.c`
- `c:\edk2\NtfsPkg\NtfsDxe\ntfs\volume.c`
- `c:\edk2\NtfsPkg\NtfsDxe\ntfs\bootsect.c`
- `c:\edk2\NtfsPkg\NtfsDxe\ntfs\unistr.c`
- `c:\edk2\NtfsPkg\NtfsDxe\ntfs\attrib.c`
- `c:\edk2\NtfsPkg\NtfsDxe\ntfs\mft.c`
- `c:\edk2\NtfsPkg\NtfsDxe\ntfs\index.c`
- `c:\edk2\NtfsPkg\NtfsDxe\ntfs\cache.c`
- `c:\edk2\NtfsPkg\NtfsDxe\ntfs\misc.c`
- `c:\edk2\NtfsPkg\NtfsDxe\ntfs\dir.c`
- `c:\edk2\NtfsPkg\NtfsDxe\ntfs\runlist.c`
- `c:\edk2\NtfsPkg\NtfsDxe\ntfs\logfile.c`
- `c:\edk2\NtfsPkg\NtfsDxe\ntfs\uefi_io.c`
- `c:\edk2\NtfsPkg\NtfsDxe\ntfs\ntfsinternal.c`
- `c:\edk2\NtfsPkg\NtfsDxe\ntfs\mst.c`

- `c:\edk2\NtfsPkg\NtfsDxe\ntfs\lcnalloc.c`
- `c:\edk2\NtfsPkg\NtfsDxe\ntfs\compress.c`
- `c:\edk2\NtfsPkg\NtfsDxe\ntfs\bitmap.c`
- `c:\edk2\NtfsPkg\NtfsDxe\ntfs\collate.c`
- `c:\edk2\NtfsPkg\NtfsDxe\ntfs\security.c`

Another interesting thing to note is that the project path is the same as those found in vector-edk, Hacking Team's EFI development leaked project. In vector-edk, there is a subproject `NtfsPkg` with the exact same directory layout. The ntfs-3g source code files are located in the same path. While these paths are generic, we believe this is not a coincidence.

Comparing the leaked source code with Hex-Rays decompiler output, it becomes evident that it is the same project. Figure 17 is an example comparing the function `NtfsDriverBindingStart` took from `vector-edk/NtfsPkg/NtfsDxe/Ntfs.c`. Comments have been removed from the original HT's source code for clarity. The logic and the order of the function calls are the same. Both projects even use the same variable (LockedByMe) to keep the state of the lock.

```
ControllerHandle_1 = ControllerHandle;
EfiDriverBindingProtocol = This;
LockedByMe = 0;
if ( fnNtfsAcquireLockOrFail() >= 0 )
  LockedByMe = 1;
Status = fnInitializeUnicodeCollationSupport(EfiDriverBindingProtocol->DriverBindingHandle);
if ( Status >= 0 )
{
  v4 = EFI_OPEN_PROTOCOL_GET_PROTOCOL;
  Status = (gEfiBootServices->OpenProtocol)(
             ControllerHandle_1,
             &gEfiBlockIoProtocolGuid,
             &EfiBlockIoProtocol,
             EfiDriverBindingProtocol->DriverBindingHandle,
             ControllerHandle_1,
             v4);
  if ( Status >= 0 )
  {
    LODWORD(v6) = EFI_OPEN_PROTOCOL_BY_DRIVER;
    Status = (gEfiBootServices->OpenProtocol)(
               ControllerHandle_1,
               &gEfiDiskIoProtocolGuid,
               &EfiDiskIoProtocol,
               EfiDriverBindingProtocol->DriverBindingHandle,
               ControllerHandle_1,
               v6);
    if ( Status >= 0 )
    {
      Status = fnNtfsAllocateVolume(ControllerHandle_1,
          EfiDiskIoProtocol, EfiBlockIoProtocol);
      if ( Status < 0 )
      {
        LODWORD(v7) = 4;
        Status = (gEfiBootServices->OpenProtocol)(
                   ControllerHandle_1,
                   &gEfiSimpleFileSystemProtocolGuid,
                   0i64,
                   EfiDriverBindingProtocol->DriverBindingHandle,
                   ControllerHandle_1,
                   v7);
        if ( Status < 0 )
          (gEfiBootServices->CloseProtocol)(
            ControllerHandle_1,
            &gEfiDiskIoProtocolGuid,
            EfiDriverBindingProtocol->DriverBindingHandle,
            ControllerHandle_1);
      }
    }
  }
}
if ( LockedByMe )
  fnNtfsReleaseLock(v3);
return Status;
}
```

```
Status = NtfsAcquireLockOrFail ();
if (!EFI_ERROR (Status)) {
  LockedByMe = TRUE;
}

Status = InitializeUnicodeCollationSupport (This->DriverBindingHandle);
if (EFI_ERROR (Status)) {
  goto Exit;
}
Status = gBS->OpenProtocol (
                ControllerHandle,
                &gEfiBlockIoProtocolGuid,
                (VOID **) &BlockIo,
                This->DriverBindingHandle,
                ControllerHandle,
                EFI_OPEN_PROTOCOL_GET_PROTOCOL
                );
if (EFI_ERROR (Status)) {
  goto Exit;
}
Status = gBS->OpenProtocol (
                ControllerHandle,
                &gEfiDiskIoProtocolGuid,
                (VOID **) &DiskIo,
                This->DriverBindingHandle,
                ControllerHandle,
                EFI_OPEN_PROTOCOL_BY_DRIVER
                );
if (EFI_ERROR (Status)) {
  goto Exit;
}
Status = NtfsAllocateVolume (ControllerHandle, DiskIo, BlockIo);
if (EFI_ERROR (Status)) {
  Status = gBS->OpenProtocol (
                  ControllerHandle,
                  &gEfiSimpleFileSystemProtocolGuid,
                  NULL,
                  This->DriverBindingHandle,
                  ControllerHandle,
                  EFI_OPEN_PROTOCOL_TEST_PROTOCOL
                  );
  if (EFI_ERROR (Status)) {
    gBS->CloseProtocol (
           ControllerHandle,
           &gEfiDiskIoProtocolGuid,
           This->DriverBindingHandle,
           ControllerHandle
           );
  }
}

Exit:
  if (LockedByMe) {
    NtfsReleaseLock ();
  }
```

Figure 17 // Comparison between Hex-Rays decompiler output of Sednit's NTFS driver (left) and HT's NTFS driver (right)

The comparison above shows code from Hacking Team developers and is not present in the ntfs-3g open source code.

As mentioned in the `ReWriter_binary` section, when parsing the firmware file system the executable tries to remove the AMI NTFS driver. We wanted to understand why they remove it instead of using it. We analyzed the driver and found out that it can only perform read operations. As writing to the file system is not supported, they couldn't use it for their purposes. It is also likely that Sednit's operators may have run into some issues when another NTFS driver was already present in the firmware, so they simply decided to remove it. In addition to implementing read and write operations, Hacking Team's driver does not enforce file permissions. For instance, it is possible to overwrite a read-only file without raising any error.

At this point in this paper, we have described the various operations performed by the UEFI rootkit to compromise the host operating system. We also discussed the reasons why we believe that Sednit operators used the source code of Hacking Team's vector-edk to build their NTFS driver to write files on the Windows NTFS partition. In the following sections, we will provide our analysis of the payloads dropped by SecDxe.

autoche.exe vs. autochk.exe

The malicious `autoche.exe` is used to set up persistence for the small agent `rpcnetp.exe`. As can be seen in Figure 18, it uses native Windows API calls to create this service.

```
if ( NtOpenKey(&KeyHandle, 0xF003Fu, &ObjectAttributes) < 0 )
{
  NtCreateKey(&KeyHandle, KEY_ALL_ACCESS, &ObjectAttributes, 0u, 0u, 0u, 0u);
  RtlInitUnicodeString(&ValueName, L"DisplayName");
  RtlInitUnicodeString(&v5, L"Remote Procedure Call (RPC) Net")
  if ( NtSetValueKey(KeyHandle, &ValueName, 0u, 1u, v5.Buffer, v5.MaximumLength) >= 0 )
  {
    RtlInitUnicodeString(&ValueName, L"ObjectName");
    RtlInitUnicodeString(&v5, L"LocalSystem");
    if ( NtSetValueKey(KeyHandle, &ValueName, 0u, 1u, v5.Buffer, v5.MaximumLength) >= 0 )
    {
      RtlInitUnicodeString(&ValueName, L"ErrorControl");
      Data = 1;
      if ( NtSetValueKey(KeyHandle, &ValueName, 0u, 4u, &Data, 4u) >= 0 )
      {
        RtlInitUnicodeString(&ValueName, L"ImagePath");
        v19 = NtCreateFile(&FileHandle, 1u, &v24, &IoStatusBlock, 0u, 128u, 1u, 1u, 1u, 0u, 0u);
        RtlInitUnicodeString(&v5, L"C:\\Windows\\SysWOW64\\rpcnetp.exe");
        if ( v19 < 0 )
          RtlInitUnicodeString(&v5, L"C:\\Windows\\System32\\rpcnetp.exe");
        if ( NtSetValueKey(KeyHandle, &ValueName, 0u, 2u, v5.Buffer, v5.MaximumLength) >= 0 )
        {
          RtlInitUnicodeString(&ValueName, L"Start");
          v20 = 2;
          if ( NtSetValueKey(KeyHandle, &ValueName, 0u, 4u, &v20, 4u) >= 0 )
          {
            RtlInitUnicodeString(&ValueName, L"Type");
            v21 = 16;
            NtSetValueKey(KeyHandle, &ValueName, 0u, 4u, &v21, 4u);
```

Figure 18 // Malicious autoche.exe setting up rpcnetp.exe persistence

It should be noted that the service name is the same as the one used by the legitimate Computrace agent. Once the service is created, it then restores the `BootExecute` registry key to its previous value.

```
NtClose(FileHandle);
RtlInitUnicodeString(&v28, L"\\REGISTRY\\MACHINE\\SYSTEM\\CurrentControlSet\\Control\\Session Manager");
ObjectAttributes.Length = 24;
ObjectAttributes.RootDirectory = 0;
ObjectAttributes.Attributes = 512;
ObjectAttributes.ObjectName = &v28;
ObjectAttributes.SecurityDescriptor = 0;
ObjectAttributes.SecurityQualityOfService = 0;
NtOpenKey(&v23, 0xF003Fu, &ObjectAttributes);
*SourceString = 'u\0a';
v8 = 'o\0t';
v9 = 'h\0c';
v10 = 'c\0e';
v11 = ' \0k';
v12 = 'u\0a';
v13 = 'o\0t';
v14 = 'h\0c';
v15 = ' \0k';
v16 = '*';
v17 = 0;
RtlInitUnicodeString(&ValueName, L"BootExecute");
RtlInitUnicodeString(&v5, SourceString);
NtSetValueKey(v23, &ValueName, 0u, 7u, SourceString, 0x2Au);
return NtTerminateProcess(0xFFFFFFFF, 0);
```

Figure 19 // Malicious autoche.exe restoring BootExecute's original registry value

Since this process takes place while Windows is booting, the user can hardly notice the `BootExecute` registry key value modification. It should be noted that autoche.exe shows some similarities with Computrace's `autochk.exe` module, such as the API calls used and the service registration, but the rest is quite different. Computrace's module is bigger and restores the original autochk.exe executable instead of changing the registry key. It is also responsible for dropping the small agent on disk, while this is handled by the UEFI rootkit in the LoJax case.

rpcnetp.exe

While the small agent `rpcnetp.exe` can be dropped by the UEFI rootkit, it is probable that most instances we saw of a trojanized LoJack small agent did not use this component. It is likely that they were opportunistic and installed the UEFI rootkit only when possible and in organizations of high importance.

Throughout our investigation, we were able to uncover different LoJax small agent versions. The IOC section lists their hashes and the associated malicious domains/IPs. As discussed previously, all LoJax small agent samples we were able to recover were a trojanized version of the same old Computrace small agent compiled in 2008.

While we never witnessed LoJax agent download and install additional modules, we do know that this functionality exists. As LoJax's best quality is to be stealthy and persistent, it could definitely be used to help ensure that access to key resources is maintained.

6. PREVENTION AND REMEDIATION

How could such an attack have been prevented? This involves a complex ecosystem composed of multiple actors. The first security mechanism that could have blocked such an attack is Secure Boot. When Secure Boot is enabled, each and every firmware component that is loaded by the firmware needs to be properly signed, thus ensuring the integrity of the firmware. We strongly suggest that you enable it. This is the base defense against attacks targeting UEFI firmware.

As is the case for software, the UEFI firmware should always be kept up-to-date. Visit your motherboard website to make sure that you have the latest version available.

You should also make sure that all of your systems have modern chipsets with Platform Controller Hub (starting from Intel Series 5 chipsets onwards). This will ensure that the security mechanism against the race condition vulnerability we mentioned [18] is available on the platform.

The other part of firmware security is in the hands of UEFI/BIOS vendors. The security mechanisms provided by the platform need to be properly configured by the system firmware to actually protect it. Thus, firmware must be built with security in mind from the ground up. Fortunately, more and more security researchers are looking at firmware security thus contributing to improve this field and raise awareness of firmware vendors. It is also worth mentioning CHIPSEC [16], an open source framework to perform low-level security assessments, which is very helpful to determine if your platform is properly configured.

Remediation of a UEFI firmware-based compromise is a hard problem. There are no easy ways of cleaning the system from such threat nor are there any security products that can save the day. In the case we described in this paper, the SPI flash memory needs to be reflashed to remove the rootkit. This is not a trivial task and that definitely is not a recommended procedure for the average computer owner. Upgrading the UEFI firmware may remove the rootkit given that the update rewrites the whole BIOS region of the SPI flash memory. If reflashing the UEFI firmware is not an option for you, the only alternative is to change the motherboard of the infected system.

7. CONCLUSION

UEFI rootkits are one of the most powerful tools in an attacker's arsenal as they are persistent across OS re-install and hard disk changes and are extremely difficult to detect and remove. While it is hard to modify a system's UEFI image, few solutions exists to scan system's UEFI modules and detect malicious ones. Moreover, cleaning a system's UEFI firmware means re-flashing it, an operation not commonly done and certainly not by the average user. These advantages explain why determined and resourceful attackers will continue to target systems' UEFI.

8. ACKNOWLEDGEMENT

We'd like to express our gratitude to the people behind opensecuritytraining.info for the great material that they share with the community. The course 'Introduction to BIOS & SMM' [20] was of great help to us when it came the time to analyze interactions with the SPI flash chip.

9. GLOSSARY

Please refer to Intel specifications [21] for more details on each fields and more.

- **BIOS_CNTL:** BIOS Control Register
- **BIOSWE:** BIOS Write Enabled
- **BLE:** BIOS Lock Enabled
- **FADDR:** Flash Address
- **FDATAX:** Flash Data from FDATA0 to FDATAN
- **FDBC:** Flash Data Byte Count
- **FGO:** Flash Cycle Go
- **HSFC:** Hardware Sequencing Flash Control
- **HSFS:** Hardware Sequencing Flash Status
- **IOCTL:** Input/Output Control
- **PCH:** Platform Controller Hub
- **RCBA:** Root Complex Base Address Register
- **RCRB:** Root Complex Register Block
- **SCIP:** SPI Cycle in Progress
- **SMI:** System Management Interrupt
- **SMM:** System Management Mode
- **SMM_BWP:** SMM BIOS Write Protect Disable
- **SPI:** Serial Peripheral Interface

10. REFERENCES

1 D. Alperovitch, "Bears in the Midst: Intrusion into the Democratic National Committee," Crowdstrike, 15 June 2016. [Online]. Available: *https://www.crowdstrike.com/blog/bears-midst-intrusion-democratic-national-committee/*

2 US Department of Justice, July 2018. [Online]. Available: *https://assets.documentcloud.org/documents/4598895/DOJ-Russia-DNC-Hack-Indictment.pdf*

3 G. Corera, "How France's TV5 was almost destroyed by 'Russian hackers'," BBC, 10 October 2016. [Online]. Available: *https://www.bbc.com/news/technology-37590375*

4 L. Matsakis, "Hack Brief: Russian Hackers Release Apparent IOC Emails in Wake of Olympics Ban," WIRED, 1 January 2018. [Online]. Available: *https://www.wired.com/story/russian-fancy-bears-hackers-release-apparent-ioc-emails/*

5 ESET Research, "En Route with Sednit," ESET, 2016. [Online]. Available: *http://www.welivesecurity.com/wp-content/uploads/2016/10/eset-sednit-full.pdf*

6 ESET Research, "Sednit adds two zero-day exploits using 'Trump's attack on Syria' as a decoy," ESET, 9 May 2017. [Online]. Available: *https://www.welivesecurity.com/2017/05/09/sednit-adds-two-zero-day-exploits-using-trumps-attack-syria-decoy/*

7 ESET Research, "Sednit update: Analysis of Zebrocy," ESET, 24 April 2018. [Online]. Available: *https://www.welivesecurity.com/2018/04/24/sednit-update-analysis-zebrocy/*

8 P. Lin, "Hacking Team Uses UEFI BIOS Rootkit to Keep RCS 9 Agent in Target Systems," Trend Micro, 13 July 2015. [Online]. Available: *https://blog.trendmicro.com/trendlabs-security-intelligence/hacking-team-uses-uefi-bios-rootkit-to-keep-rcs-9-agent-in-target-systems/*

9 WikiLeaks, "DerStarke 2.0," [Online]. Available: *https://wikileaks.org/ciav7p1/cms/page_13763820.html*

10 Absolute, "Absolute Response to Arbor Research," May 2018. [Online]. Available: *https://www.absolute.com/en-gb/resources/faq/absolute-response-to-arbor-research*

11 A. Ortega and A. Sacco, "Deactivate the Rootkit: Attacks on BIOS anti-theft," Core Security Technologies, 24 July 2009. [Online]. Available: *https://www.coresecurity.com/system/files/publications/2016/05/Paper-Deactivate-the-Rootkit-AOrtega-ASacco.pdf*

12 V. Kamlyuk, S. Belov and A. Sacco, "Absolute Backdoor Revisited," BlackHat, June 2014. [Online]. Available: *https://www.blackhat.com/docs/us-14/materials/us-14-Kamluk-Computrace-Backdoor-Revisited-WP.pdf*

13 ASERT team, "Lojack Becomes a Double-Agent," 1 May 2018. [Online]. Available: *https://asert.arbornetworks.com/lojack-becomes-a-double-agent/*

14 "RWEverything Read & Write Everything," [Online]. Available: *http://rweverything.com/*

15 A. Matrosov and E. Rodionov, "UEFI Firmware Rootkits: Myths and Reality," Black Hat Asia, 2017. [Online]. Available: *https://www.blackhat.com/docs/asia-17/materials/asia-17-Matrosov-The-UEFI-Firmware-Rootkits-Myths-And-Reality.pdf*

16 "GitHub repository for UEFITool," [Online]. Available: *https://github.com/LongSoft/UEFITool*

17 Cylance, "Researchers Disclose Vulnerabilities in GIGABYTE BRIX Systems," [Online]. Available: *https://threatvector.cylance.com/en_us/home/gigabyte-brix-systems-vulnerabilities.html*

18 Carnegie Mellon University SEI CERT, "Vulnerability Note VU#766164, Intel BIOS locking mechanism contains race condition that enables write protection bypass," [Online]. Available: *https://www.kb.cert.org/vuls/id/766164*

19 C. Kallenberg and R. Wojtczuk, "Speed Racer: Exploiting an Intel Flash Protection Race Condition," January 2015. [Online]. Available: *https://bromiumlabs.files.wordpress.com/2015/01/speed_racer_whitepaper.pdf*

20 J. Butterworth, "Advanced x86: Introduction to BIOS & SMM," 2014. [Online]. Available: *http://opensecuritytraining.info/IntroBIOS.html*

21 Intel, "Intel 7 Series / C216 Chipset and Family Platform Controller Hub (PCH)," June 2012. [Online]. Available: *https://www.intel.com/content/dam/www/public/us/en/documents/datasheets/7-series-chipset-pch-datasheet.pdf*

11. IOCs

ReWriter_read.exe

ESET detection name

Win32/SPIFlash.A

SHA-1

`ea728abe26bac161e110970051e1561fd51db93b`

ReWriter_binary.exe

ESET detection name

Win32/SPIFlash.A

SHA-1

`cc217342373967d1916cb20eca5ccb29caaf7c1b`

SecDxe

ESET detection name

EFI/LoJax.A

SHA-1

`f2be778971ad9df2082a266bd04ab657bd287413`

info_efi.exe

ESET detection name

Win32/Agent.ZXZ

SHA-1

`4b9e71615b37aea1eaeb5b1cfa0eee048118ff72`

autoche.exe

ESET detection name

Win32/LoJax.A

SHA-1

`700d7e763f59e706b4f05c69911319690f85432e`

Small agent EXE

ESET detection names

Win32/Agent.ZQE

Win32/Agent.ZTU

SHA-1

`1771e435ba25f9cdfa77168899490d87681f2029`

`ddaa06a4021baf980a08caea899f2904609410b9`

`10d571d66d3ab7b9ddf6a850cb9b8e38b07623c0`

`2529f6eda28d54490119d2123d22da56783c704f`

`e923ac79046ffa06f67d3f4c567e84a82dd7ff1b`

`8e138eecea8e9937a83bffe100d842d6381b6bb1`

`ef860dca7d7c928b68c4218007fb9069c6e654e9`

```
e8f07caafb23eff83020406c21645d8ed0005ca6
09d2e2c26247a4a908952fee36b56b360561984f
f90ccf57e75923812c2c1da9f56166b36d1482be
```

C&C server domain names

secao[.]org
ikmtrust[.]com
sysanalyticweb[.]com
lxwo[.]org
jflynci[.]com
remotepx[.]net
rdsnets[.]com
rpcnetconnect[.]com
webstp[.]com
elaxo[.]org

C&C server IPs

185.77.129[.]106
185.144.82[.]239
93.113.131[.]103
185.86.149[.]54
185.86.151[.]104
103.41.177[.]43
185.86.148[.]184
185.94.191[.]65
86.106.131[.]54

Small agent DLL

In this section, we list only the DLL for which we never obtained the corresponding EXE

ESET detection names

Win32/Agent.ZQE

SHA-1

```
397d97e278110a48bd2cb11bb5632b99a9100dbd
```

C&C server domain names

elaxo[.]org

C&C server IPs

86.106.131[.]54