# A Universal Windows Bootkit

## William Showalter

Information Security. Enjoys traveling. Mississippi State University graduate student. Formerly of the University of Alaska Fairbanks.

Email Twitter LinkedIn Github Flickr

**An analysis of the MBR bootkit referred to as "HDRoot"**

## Executive Summary

In October, 2015 Kaspersky released an analysis of a family of malware they dubbed "HDRoot" on their Securelist blog. It was an installment in their ongoing series on the WINNTI group, known for targeting gaming companies in their APT campaigns. The Securelist blog was dismissive of the HDRoot bootkit and called out a number of mistakes they claimed the authors made, which brought it to be the focus of their ridicule.

The bootkit in question uses two stolen signing certificates and is capable of running without problem on any Windows system that was released in the last 16 years, from Windows 2000 to Windows 10. The one limitation is that it will only run as an MBR bootkit and will not work on systems using UEFI. It contains the ability to install any backdoor payload to be launched in the context of a system service when Windows starts up on both 32 and 64-bit systems. It also does a fairly good job of concealing the actual bootkit code, only failing to remove the backdoor after running it at boot. This likely a conscious choice made by the authors to have the backdoor responsible for removing itself, and not an oversight.

HDRoot represents a serious commitment in time and effort to develop, and likely has been in use or development since at least 2006. The sample analyzed here dates to sometime in 2012 or 2013, and is the same sample Kasperky reports to have analyzed in their debut post on HDRoot. However, all evidence points to Kaspersky doing their analysis with a 2006 sample, criticizing problems in the malware that are not actually present. Additionally, they provide no hashes or other information on the actual sample they used.

The samples I analyzed in this report are detailed in appendix 1 and hashes are provided in appendix 2. They can be found in the following git repo below:

https://github.com/williamshowalter/hdroot-bootkit-analysis

Download White Paper

## Table of Contents

## Introduction

## Sample:

MD5: 2c85404fe7d1891fd41fcee4c92ad305
SHA1: 4c3171b48d600e6337f1495142c43172d3b01770
SHA256: a9a8dc4ae77b1282f0c8bdebd2643458fc1ceb3145db4e30120dd81676ff9b61

Original Filename: net.exe
Produce Name: Microsoft Windows Operating System
Product Version: 6.1.7600.16385
Time Stamp: 2012/08/06 13:12:39 UTC
Retrieved from malwr [1].

## This report

My analysis of this HDRoot sample began as an exercise to become more familiar with low-level malware and the techniques required for reverse engineering them. I had no prior knowledge of the WINNTI group who Kaspersky attributes this malware to, nor do I have any other samples beyond those associated with the dropper and bootkit analyzed here. The dropper is capable of installing any PE executable as the payload for the bootkit, but does not come bundled with any default payloads. As such, this report offers no insight into the various payloads used by the authors. For information on the other malware associated with the WINNTI group, see Trend Micro or Kaspersky's reports on the group [2][3]. This report does, however, offer a very in-depth look into the technical workings of the HDRoot bootkit and its components.

I also address a number of technical inaccuracies and misrepresentations from Kaspersky's SecureList post of October 2015, "I am HDRoot! Part 1," which is the only research on this sample to be published before my own [4]. Kaspersky's research was very helpful getting started, but I soon discovered that their analysis was not actually performed using the sample identified by MD5 in the article and thus could not be relied upon. I believe this to be the reason for many of their criticisms for HDRoot, which they call "quite conspicuous," and, "not what you expect from such a serious APT actor." The sample analyzed here is not free of criticisms, but none of the problems addressed by Kaspersky appear to be valid on the sample they claim to have analyzed.

I also freely acknowledge that the level of detail that this report goes into is impractical for almost all incident response purposes, and that this venture was largely done for my own education and curiosity.

## Overview

The malware examined here can be broken into several stages. The 64-bit dropper, which was signed with a stolen certificate that has since been revoked, is the first component that is executed. The dropper installs the bootkit to the hard drive along with a backdoor executable to be run on subsequent boots. The backdoor is supplied as a parameter to the dropper and can be any Win32 or Win64 executable.

Upon boot, the computer will execute the maliciously installed MBR, which loads a subsequent component that I named the "verifier". It is a single sector block that verifies that the rest of the bootkit and the backdoor are intact before running them. The bulk of the

bootkit's work is done by the next component, rkImage. The name rkImage actually comes from the interface of the dropper, which explicitly refers to it when installing the bootkit. rkImage works by manually reading the file system from the disk in order to write the backdoor (the generic term referring to the payload) into the filesystem and redirect a Windows system service to launch the backdoor.

When rkImage is finished it transfers execution back to the original, non-infected MBR and allows Windows to boot normally. The booting system will run the backdoor instead of the replaced system service, but will then restore and start the legitimate service after the backdoor has ran, hiding the fact it was ever replaced.

## Dropper

The dropper is designed to disguise itself as the Windows system utility net.exe. The properties on the executable attempt to mirror the settings found on a Windows 7 version of the utility, reporting it to a Microsoft program. When run without parameters, HDRoot shows the options menu as if it were net.exe. That is where the similarities end, however.

```
Administrator: Command Prompt                              —    □    ✕

C:\malware>net

net
NET [ ACCOUNTS | COMPUTER | CONFIG | CONTINUE | FILE | GROUP | HELP |
      HELPMSG | LOCALGROUP | NAME | PAUSE | PRINT | SEND | SESSION |
      SHARE | START | STATISTICS | STOP | TIME | USE | USER | VIEW ]


C:\malware>net help


C:\malware>
```
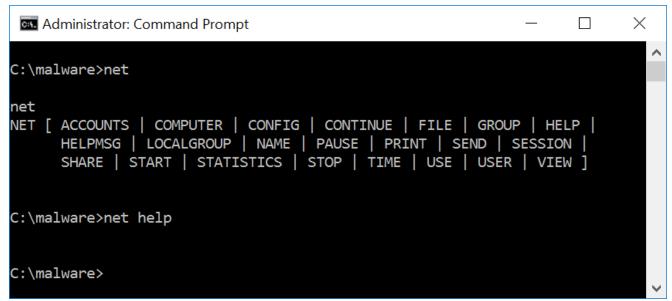
Figure 1: Dropper disguising itself as net.exe

The dropper executable, while masquerading as the Microsoft net command, has been signed with a digital certificate belonging to Guangzhou YuanLuo Technology Co, Ltd, a firm based in the city of Guangzhou, China who had their signing certificate stolen by the WINNTI group. The certificate has since been revoked, and, if the signing time and compilation dates on the executable are to be believed, it was signed in 2013 almost a year after this version was initially compiled.
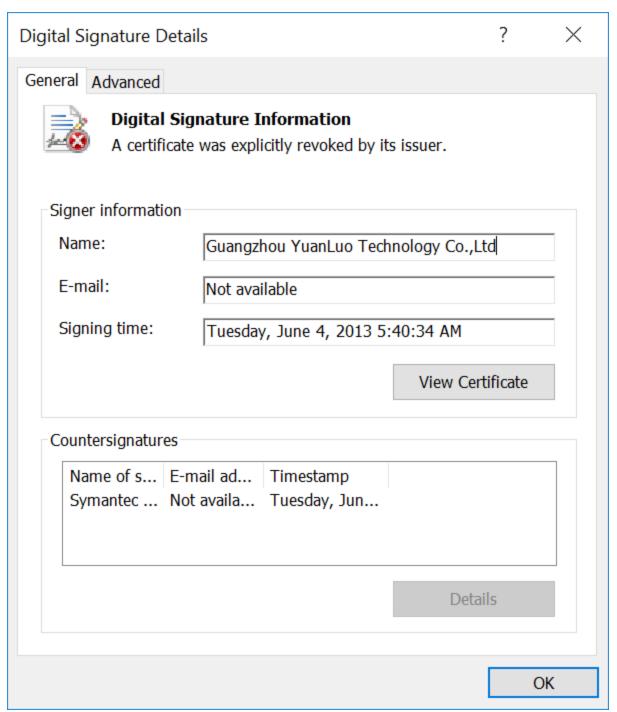
Figure 2: Dropper's revoked certificate

When run with any of the legitimate net command parameters or with unrecognized parameters, no output is given. The only commands that provide output are the valid HDRoot commands programmed by the authors. Kaspersky, by analyzing an older sample from 2006, was able to get a "help" output, rather than the "net" output, which contained a list of commands for that version. Most of these commands still worked on the newer sample. All of the commands were five or less characters in length, and even short words like install were abbreviated to "inst". Since the Kaspersky command listing was half a decade older than this sample, and that some of the commands from their listing were no longer present, I wrote a simple, and very slow, fuzzer to attempt to check all possible commands of five or less

characters. Given the length of the other commands and that input appears to be case insensitive, this appears to be a sensible approach. The code for the fuzzer can be found in the supplemental files detailed in appendix 1.1. Screenshots for each command can be found in appendix 3.1, as well. No additional commands to the ones Kaspersky detailed were found by the fuzzer, and the table below is the known list of commands.

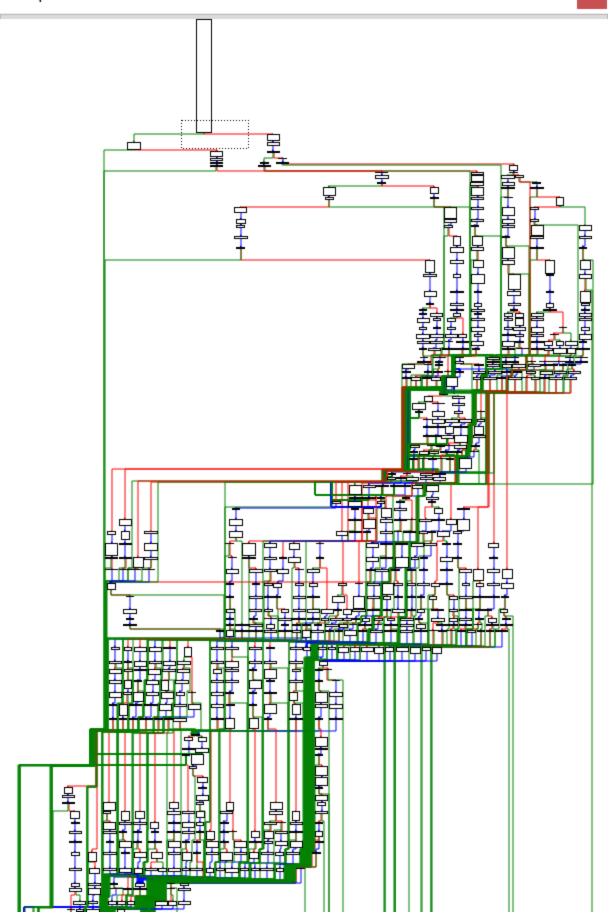| Command | Description |
| --- | --- |
| check | Checks for the presence of the bootkit and the integrity if present. |
| clean | Removes the bootkit. |
| inst <Backdoor> | Installs the bootkit |
| info <Backdoor> | Shows information about the checksums and requirements for an executable if it was installed as the backdoor. |

Table 1: HDRoot dropper commands

```
Administrator: Command Prompt                                    —    □    ✕

C:\malware>net inst proof.exe


I: System Env check
   end

I: rkImage size    56580 B, will use 120 sectors
I: Backdoor size 202240 B, will use 395 sectors

I: Total use  515 sectors

I: Crc16 0x0cef
I: DriverBitMap 0000000c


C: -

    1 file layout pairs
    Rkfile LCN 6245093, VCN 65
    Rkfile logical sector 49960744, 520
    Rkfile phyical disk 0, sector 50986792
    Image place PhyDisk 80,   50986792 -   50987307 sector

    Disk free 64%
    Lastfree   7101780 -  18221566,  11119787 clusters, 44479148 k; Gold place
    Maxfree    7101780 -  18221566,  11119787 clusters, 44479148 k; Gold place
    Image place  145772013 - 145772528 logical sector
    Rkfile phyical disk 0, sector 146798061
    Image place PhyDisk 80,  146798061 -  146798576 sector


I: Bootdisk phyid 0


done
```

Figure 3: inst command output

## VMProtect

A notable hindrance to reversing the dropper is that it was packed using VMProtect. Unlike most packers, which decompress and then jump to the original executable code, VMProtect converts the x86 opcodes into an automatically generated language of bytecodes to be interpreted in its own emulator. Attempting to statically analyze the sample would prove an arduous task. There have been a few unpacking plugins for Ollydbg written for certain versions of VMProtect, but these are generally found in forum posts and are not well maintained. I believe this to be the reason Kaspersky did the bulk of their analysis with a different sample that was almost, but not quite, functionally the same. Not wanting to spend my time tackling VMProtect either, I instead used a number of dynamic analysis techniques.
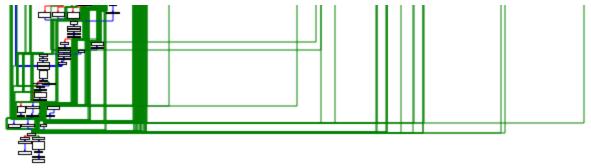
Figure 4: Graph overview of VMProtect's emulator. Not fun.

## DEBUGFILE.sys - A signed kernel driver

At the time the dropper installs the bootkit, no changes to the filesystem or the registry are seen between snapshots taken before and after. I took the approach of running the dropper in a continuous loop in a virtual machine, suspending the VM, and analyzing the resulting memory image. Performing memory capture from outside the VM appeared to be the best option because there were a number of anti-debugging techniques employed along with the anti-disassembly. Using Volatility, I discovered two more PE files that were extracted inside the process, but none of the four clear text resources Kaspersky claimed to have extracted from a memory dump, providing further proof that they did their analysis on a different sample than is listed in their blog post. The two PE files I found were kernel drivers, one 32-bit and one 64-bit. The 64-bit driver is signed, as is required by 64-bit versions of Windows, using yet another stolen certificate, while the 32-bit driver is not signed. This certificate belongs to a South Korean video game company, Neowiz. The certificate, unlike the one for the dropper, has yet to be revoked (see Figure 6).

The use of the kernel drivers is fairly straightforward. Without kernel access there is no way for malware to write directly to the physical disk as there are no Windows API calls available to userland processes for doing so. The dropper writes out the appropriate driver to C:\Windows\system32\Drivers\DEBUGFILE.sys, and then creates a service for it. This shows up in the memory image as a handle to the registry key HKLM\System\ControlSet001\services\DEBUGFILE. The service runs and the driver \Driver\DEBUGFILE is created. DEBUGFILE.sys is also deleted from the disk. The driver is used by the dropper to proxy its direct access to the physical disk. A number of things are done in this process. The original MBR is backed up and then overwritten by the new bootkit MBR, and then weakly encrypted components are written to disk. Near the beginning of the disk is the component I've named the verifier, followed by two identical copies of the original MBR. In another section of the disk is the main component of the bootkit, rkImage, followed by the backdoor that was installed.
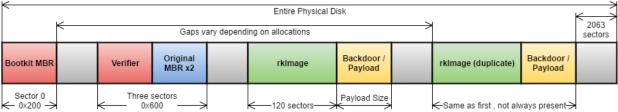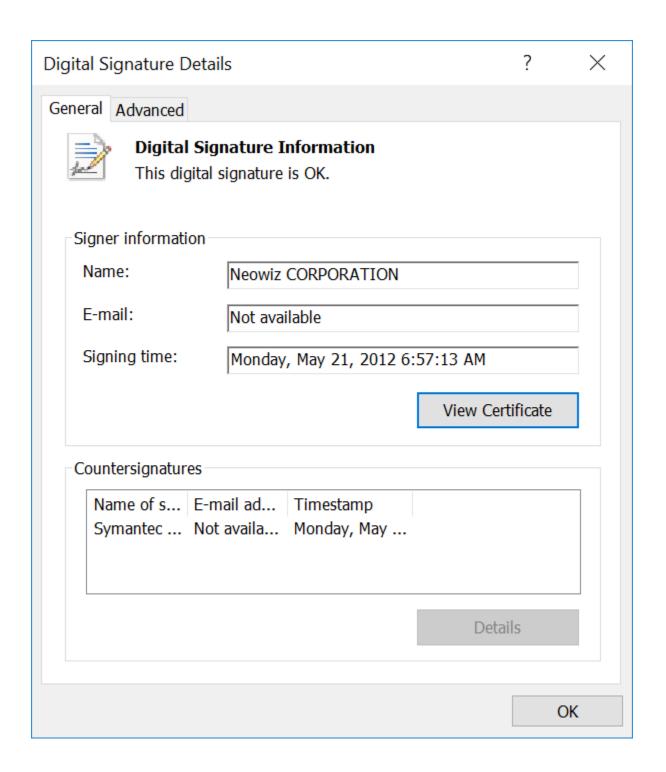
Figure 5: Physical Disk Layout written by DEBUGFILE.sys

One peculiar thing the malware does is install a second copy of the rkImage and backdoor files. This copy is encrypted identically to the first, and positioned such that it ends exactly 2063 sectors from the end of the drive. What makes this strange is that nothing in the bootkit will ever transfer execution to the second copy, and that the second copy is only installed if the drive has at least 30% free space. Kaspersky erroneously identified this behavior as only installing if the disk has greater than 30% free space, rather than installing a redundant copy of itself. As can be seen in a Windows 7 screenshot from the appendix 1.3 files, the bootkit is perfectly capable of installing with less than 30% free space. The only guess I make as to the purpose of this second copy is for the indented backdoor to be able to identify if one of the copies has been modified after it starts. The dropper will also detect a modified copy.

## Digital Signature Details

General | Advanced

### Digital Signature Information
This digital signature is OK.

**Signer information**

Name: Neowiz CORPORATION

E-mail: Not available

Signing time: Monday, May 21, 2012 6:57:13 AM

View Certificate

**Countersignatures**

| Name of s... | E-mail ad... | Timestamp | |
|---|---|---|---|
| Symantec ... | Not availa... | Monday, May ... | |

Details

OK
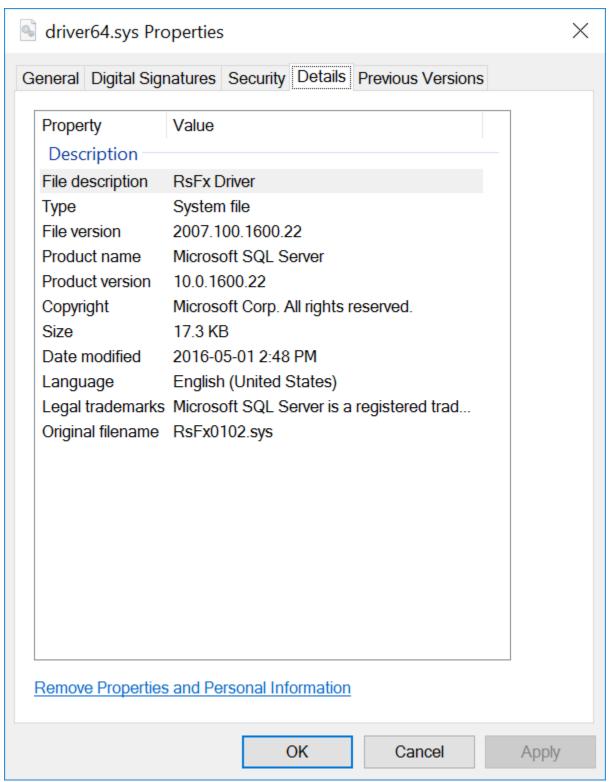
Figure 6: DEBUGFILE.sys

## MBR

For all x86 systems not running UEFI, the boot process starts with the BIOS loading the Master Boot Record into memory and jumping to it. By convention, the BIOS loads the MBR to the physical memory address of 0x7C00. Another convention that many MBRs follow is to copy themselves, a single 0x200 sized sector, to the address 0x600 and then transfer

execution to this location. The HDRoot bootkit is no exception. This is partly because the only code it actually changes in the original MBR is the jump address and the code jumped to. Most of the original MBR and partition table information is intact.

A normal MBR would look at the partition table to find the partition with the boot flag set, and then load the volume boot sector of that partition and transfer execution. HDRoot's MBR works similarly by calling interrupt 13 to read two sectors from disk into memory at the address 0x7A00 (through 0x7DFF). These are the verifier and the original MBR, which now has been loaded into the location where the MBR would have originally loaded on a non-infected system. The bootkit does not store these on disk in clear text, however. They are written to disk having been XOR'd with the byte value 0x76. Appendix 1.2 has a C utility that can be used to decrypt the values. A function at offset MBR+0x88 performs these read and decrypt operations, copies the partition table from the infected MBR to the original (incase the victim has changed any partitions since the bootkit was installed), and then transfers execution to the verifier.
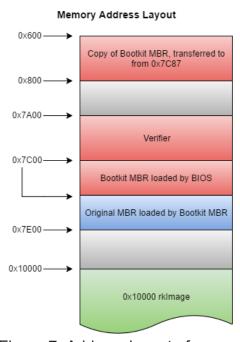


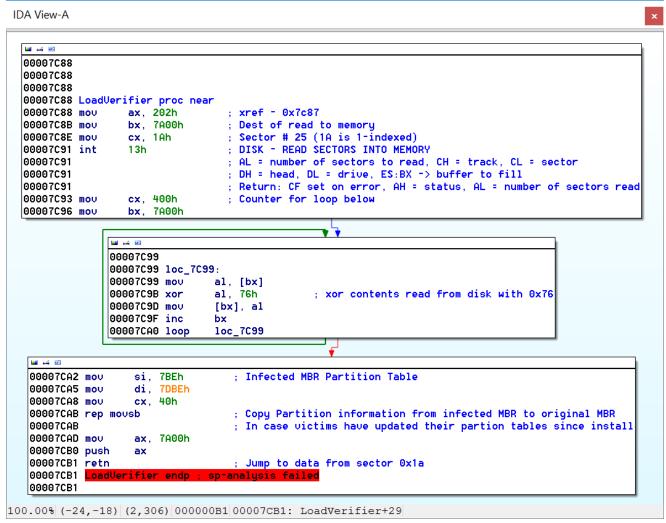Figure 7: Address layout of memory loaded before rkImage

```
IDA View-A                                                                                  ☒

  📊 📈 📉
 00007C88
 00007C88
 00007C88
 00007C88 LoadVerifier proc near
 00007C88 mov      ax, 202h       ; xref - 0x7c87
 00007C8B mov      bx, 7A00h      ; Dest of read to memory
 00007C8E mov      cx, 1Ah        ; Sector # 25 (1A is 1-indexed)
 00007C91 int      13h            ; DISK - READ SECTORS INTO MEMORY
 00007C91                         ; AL = number of sectors to read, CH = track, CL = sector
 00007C91                         ; DH = head, DL = drive, ES:BX -> buffer to fill
 00007C91                         ; Return: CF set on error, AH = status, AL = number of sectors read
 00007C93 mov      cx, 400h       ; Counter for loop below
 00007C96 mov      bx, 7A00h

                          📊 📈 📉
                         00007C99
                         00007C99 loc_7C99:
                         00007C99 mov      al, [bx]
                         00007C9B xor      al, 76h        ; xor contents read from disk with 0x76
                         00007C9D mov      [bx], al
                         00007C9F inc      bx
                         00007CA0 loop     loc_7C99

  📊 📈 📉
 00007CA2 mov      si, 7BEh       ; Infected MBR Partition Table
 00007CA5 mov      di, 7DBEh
 00007CA8 mov      cx, 40h
 00007CAB rep movsb               ; Copy Partition information from infected MBR to original MBR
 00007CAB                         ; In case victims have updated their partion tables since install
 00007CAD mov      ax, 7A00h
 00007CB0 push     ax
 00007CB1 retn                    ; Jump to data from sector 0x1a
 00007CB1 LoadVerifier endp ; sp-analysis failed
 00007CB1

100.00% (-24,-18) (2,306) 000000B1 00007CB1: LoadVerifier+29
```

Figure 8: Infected MBR code to load, decrypt verifier

## Verifier

The job of the verifier is to make sure that the bootkit is intact and that a specific set of criteria are met before allowing the bootkit to run. If any of these criteria fail the verifier will transfer the boot process to the original MBR, now at 0x7C00, without the bootkit executing. This mechanism helps prevent bricking the victim machines in the event that one or more of the hidden sectors are corrupted or overwritten.

The first criteria in the verifier process is a check for whether the alt key is pressed on the keyboard. If the alt key is pressed, the bootkit launch will be aborted. The verifier then checks for a value at 0x7A08 (+0x8 from the verifier start address). If the value is null, the startup is aborted. This value is the drive identifier of where rkImage and the backdoor are stored. This was 0x80 in all the systems I tested on, which indicates drive 0. The dropper sets this value, and the subsequent bytes, before writing them to disk. This ensures that the bootkit was properly setup during install, and allows for the bootkit to be stored on a separate disk from the system disk. Table 2 shows a breakdown of the rkImage information stored in the verifier.

| Address | Contents |
| --- | --- |
| 0x7A02 | 0x55AA, not used, signals start of rkImage location data. |
| 0x7A04 | CRC16 value for rkImage+Backdoor. |
| 0x7A06 | Sector count for rkImage+Backdoor. |
| 0x7A08 | Drive number |
| 0x7A09 | Sector where rkImage+Backdoor starts. |
| 0x7A0D | Next 0x55AA value, if present |
| … | … |

Table 2: rkImage location information in verifier

The third and final check done by the verifier is computing a CRC16 value on the encrypted contents of rkImage and the backdoor (still only encrypted with an XOR 0x76). It compares the results to the saved CRC, and if they do not match it aborts. Otherwise it reads the entire rkImage, but not the backdoor, to 0x10000, and then decrypts both. The last step is to copy the size and location information to the start of rkImage, so it can locate the backdoor for installing.

## rkImage

A significant component to reverse engineering the functionality of this bootkit was becoming familiar with the mechanics of low-level, pre-OS x86. For anyone looking to get into this I would highly recommend the Intermediate Intel x86 videos on OpenSecurityTraining.info [5]. Even just following the transition from the verifier to rkImage requires some understanding, as the processor is still in 16-bit real mode at this time and a far jump is being performed, crossing a barrier between segments. This seems like a trivial thing until you find out that GDB, even operating in 8086 mode remotely debugging the bootkit running in QEMU, has absolutely zero understanding of segment addressing and completely falls apart trying to set breakpoints at any address higher than 0xFFFF. In retrospect Bochs might have been a better choice for this over QEMU, but not being familiar with it either I struggled through with QEMU, learning as I went and performing most my analysis indirectly, either statically or dynamically through the clues left behind by the bootkit's actions.

The first task rkImage sets itself to, like any sane code booting up, is to transfer itself from real mode to protected mode, and then to 32-bit mode. In order to enable protected mode, the Global Descriptor Table must be setup and loaded. This is actually fairly unimportant to the operation of the malware but understanding it helped getting the disassembly properly setup in IDA to assist with the process. I will not get into the details and the contents of the segment descriptors, but I will state that I found the clearest explanations and diagrams in

the AMD Architecture Programmer's Manual, Vol 2., for system programming [6]. Something that caused confusion was that most diagrams detailing the structure of segment descriptors (the entries in the Global Descriptor Table) are for the descriptors in 64-bit mode, since the 32-bit descriptors, called legacy segment descriptors in AMD's documentation, have a different structure. The work done reassembling the GDT can be seen in the rkImage IDB file in appendix 1.4.

Once setup in a 32-bit environment, rkImage decrypts two sections in itself, the first containing a 32-bit DLL, and the second containing a 64-bit DLL. These are used to launch the backdoor from a Windows system service upon Windows booting. Each DLL has a 4-byte XOR key. The data is stored in rkImage in the format: 4-byte key, 4-byte length, encrypted PE file contents. The 32-bit DLL and its data are located at an offset of 0x4BE0 and the 64-bit values are at 0x6DE8, immediately after the previous DLL. These DLLs as packaged contain the registry path to the LanmanServer service DLL. The version of rkImage in this sample, installed by the 64-bit dropper, overwrites the LanmanServer information with the paths and values for the Schedule service. This allows for changing the target service without the need to recompile the DLLs embedded in rkImage. Kaspersky's observations that there are a number of different services that different samples have targeted supports this.

The backdoor executable is then loaded into memory and decrypted with the 0x76 XOR operation. At the end of this preparation work of loading, decrypting, and copying data, rkImage calls a function that I mark in my disassembly as being named DETERMINE_VERSION_NT_32_64_BIT. This is the function that determines what Windows version is installed, what malicious DLL to use, and where to install it. Since the malware will attempt to boot in any Windows version from Windows 2000 to Windows 10, there is a considerable nest of switch and if statements happening here. It first checks whether there is a "\winnt" directory, which is present in Windows 2000, and if "\winnt" is not found it will check for "\windows\system32\kernel32.dll". The check for kernel32.dll prevents the bootkit from continuing install on Windows 98 and lower systems, as kernel32.dll was stored in "\windows\system\kernel32.dll". If kernel32.dll is found it will check for "\users" and "\documents and settings" to determine if it is XP/2003, or Vista or newer. If it is not able to locate any of these, it fails out of the switch statement and no bootkit is installed.

If the directory selected was not "\winnt", it will check for "\windows\syswow64" and set a variable indicating if the system is 64-bit. This is used later when choosing which DLL to install (which means it is even 64-bit Windows XP / Server 2003 compatible). Then for each of the three operating system categories it will write the backdoor to %TEMP%\Explorer.exe (wherever %TEMP% is located for that version of Windows), and iterate through a list of files. If the file is present it will copy the appropriate DLL into the beginning of the file, overwriting the contents already there. The files to be overwritten in question are shown in Table 3, and appear to be carefully selected to cause the least potential problems, with all but one of them being for different architectures than the running host.

| Windows Version | Path |
|---|---|
| Windows 2000 | \winnt\help\access.hlp |
| Windows 2000 | \winnt\system\OLESVR.DLL |
| Windows XP or 2003 | \windows\twain.dll |
| Windows XP or 2003 | \windows\system\OLESVR.DLL |
| Windows Vista/2008 + | \windows\syswow64\C_932.NLS |
| Windows Vista/2008 + | \windows\system\OLESVR.DLL |
| Windows Vista/2008 + | \windows\syswow64\kmddsp.tsp |
| Windows Vista/2008 + | \windows\syswow64\Irclass.dll |

Table 3: Service DLL Paths by OS, in order attempted

Windows 2000 will attempt to first overwrite the access.hlp file, which, if anyone has not already disabled the help popups, may cause errors. Similarly, the 16-bit OLESVR.DLL file is overwritten if access.hlp does not exist. This will only cause issues if it is used by a 16-bit application, as 32-bit applications will be using the system32\OLESVR32.DLL file. Windows XP will attempt to overwrite the 16-bit version of the twain.dll library for scanners (even using old scanners, twain32.dll should be used), and then the 16-bit OLESVR.DLL if the twain.dll is not found.

In 64-bit Windows Vista and newer systems, the default target is C_932.NLS, which is a 32-bit National Language Support file for the Japanese language [7]. This assumes that authors did not plan on infecting targets running 32-bit applications in Japanese, as this would cause issues. The only file that will be tried for 32-bit Windows Vista and newer is the same 16-bit OLESVR.DLL. This will only cause issues for applications run in 16-bit compatibility/emulation mode, as they are not natively supported by Vista and newer, and is therefore unlikely to affect most targets. The other two potential target files, which are also unlikely to be used, are both 16-bit DLLs found in the syswow64 (32-bit compatibility) directory. They are actually only labeled as compatible with Windows Server 2003 and earlier operating systems on MSDN, but are for some reason still included in the syswow64 directory. Kmddsp.tsp is a "kernel mode device driver" for "telephony service provider" network drivers, and IRClass.dll is an Infrared Class Coinstaller [8]. Neither should ever be used on a 64-bit system and therefore won't cause any issues if overwritten.

Once the DLL has been written to the appropriate file, the registry is patched to overwrite the Schedule service's DLL path with the path to the overwritten file. This should be approximately:

HKLM\SYSTEM\CurrentControlSet\Services\Schedule\Parameters\ServiceDLL.

rkImage will then return to 16-bit real mode, handing execution back to the original MBR at 0x7C00, allowing the boot process to continue and Windows to load. It is worth noting that since Windows NT also used the C:\WINNT directory, it will match the first section of the bootkit which chooses the files to write the DLL into. However, since Windows did not introduce the svchost.exe process until Windows 2000, services did not have a Parameters sub-key or a ServiceDLL value in Windows NT. As such, if installed on Windows NT the bootkit wouldn't be able to locate the registry key for editing, and would fail out of the installation process. Additionally, it is likely that 32-bit versions of the dropper would not allow the install to a Windows 2000 system.
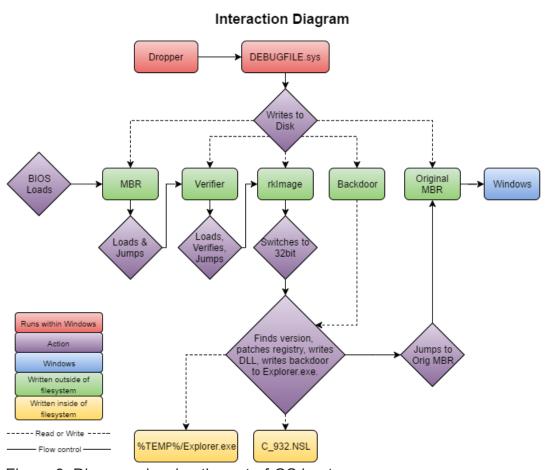


Figure 9: Diagram showing the out-of-OS boot process.

## Schedule service DLL

The final component of the bootkit, responsible for running the backdoor within Windows, is the DLL that replaced the Schedule service's DLL. The bootkit did not change the rest of the registry key, so it will be loaded into a svchost.exe executable. The Schedule service is part of the NetworkService group, so the DLL will be loaded into the svchost.exe containing the other services for the group, and a new thread will be spawned to run the ServiceMain for that DLL. Additionally, as happens every time a DLL is loaded, the DLL's entry point (DLLMain, in this case) is called by the Windows loader in another thread.

The HDRoot authors chose to use the DLLMain function to start the backdoor process and ServiceMain to revert the service registry entry back to the original path. The DLLMain thread creates another thread running the function I identified in my disassembly as SpawnBackdoorThread. That thread creates a process running the backdoor, which rkImage saved to %TEMP%\Explorer.exe. It then sets a global variable in the DLL to signal that it successfully launched the backdoor, and suspends itself before continuing.

Simultaneously the ServiceMain thread reverts the registry, and waits for the backdoor to start, sleeping and periodically checking for the flag to be set. After the flag is set it resumes the SpawnBackdoor thread, and then exits. In turn, the SpawnBackdoor thread unloads the DLL from memory and then exits itself.
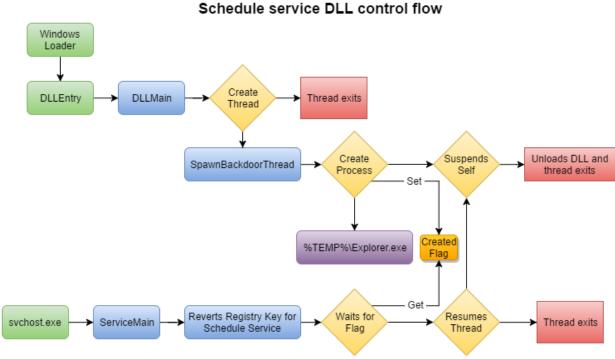


Figure 10: Flow of the malicious Schedule service

This has the effect of both threads exiting and the DLL unloading at almost the exact same time, guaranteeing that the service manager will have to restart the service, causing the legitimate service DLL to be loaded and run from the patched registry entry. In all my tests of this sample, not once did the real Windows service ever fail to start after running the bootkit. This is completely contrary to Kaspersky's claim that the bootkit breaks the service and that all the victims must just not have cared or noticed that the service failed to start.

## Conclusions

My analysis of the HDRoot malware shows Kaspersky's claims that this bootkit was written sloppily is patently wrong. It also leads me to no other conclusion than that they did their entire analysis with and presented research on a ten-year-old sample, passing it off as a

sample from 2012 that had been in modern use. It also shows that the authors, who have been designated as the WINNTI group, have been around for a significant period of time, dating back to at least 2006 if Kaspersky's sample is to be believed.

The one stage of the attack in which the bootkit did not make good use of hiding techniques was in covering its tracks for the service and backdoor executables. Both the modified file hosting the DLL (C:\Windows\syswow64\C_932.NLS in most my tests) and the backdoor in %TEMP%\Explorer.exe were left intact on the file system. However, it is likely that a sophisticated backdoor run by the bootkit would know to remove these two pieces of evidence after starting itself, and it may just have been a choice of segregation of duties made by the authors.

Another criticism that can be made is the extremely weak use of encryption. The XOR cipher is little more than obfuscation and was trivial to figure out even just looking at the encrypted sectors on the disk. It can be argued, however, that since the entire contents of the bootkit is code that will be decrypted before it can be run, there is little point in hiding it from anything but simple scans, as it could just be captured from memory by analysts. To that end, the simple cipher serves its purpose of not matching the signatures for executables or of a boot sector while on disk.

Overall I was impressed with the level of detail that went into making this malware which is capable of installing itself on any Windows version, 32 or 64-bit, dating back to Windows 2000, with the exception of newer installs using UEFI. The lack of UEFI support is unlikely to be an issue when targeting server systems, however, especially with virtualization on the rise - very few virtual environments are virtualizing UEFI in their guests. The small touches, such as anticipating that the drive may have been repartitioned, are particularly impressive. Clearly significant thought and work went into the creation of this bootkit, and it is a mistake to dismiss it as amateur. While different versions of the dropper are geared toward different targets (the observed sample here targets the Schedule service on 64-bit systems), the overall framework is very flexible. The choices made in the dropper or when compiling the dropper are able to be tuned toward the target, choosing a service compatible with that version of Windows. This makes narrowing down the traits of the bootkit from which services they target to be very difficult, as it is trivial for the authors to change their target.

# References

[1] "malwr," [Online]. Available:
https://malwr.com/analysis/NGFiNDBmMWNmYjM0NDVmZWIxNTg5OWFkMDUwYmIzNTQ/.
[2] E. A. II, "Backdoor Built With Aheadlib Used in Targeted Attacks?," Trend Micro, [Online].
Available: http://blog.trendmicro.com/trendlabs-security-intelligence/backdoor-built-with-aheadlib-used-in-targeted-attacks/.
[3] Securelist, "WINNTI: More than just a game," [Online]. Available:
https://securelist.com/analysis/internal-threats-reports/37029/winnti-more-than-just-a-game/.

[4] Securelist, "I am HDRoot Part 1," [Online]. Available:
https://securelist.com/analysis/publications/72275/i-am-hdroot-part-1/.
[5] X. Kovah, "Intermediate Intel x86," [Online]. Available:
http://opensecuritytraining.info/IntermediateX86.html.
[6] AMD, "AMD64 Architecture Programmer's Manual, Volume 2," [Online]. Available:
http://developer.amd.com/wordpress/media/2012/10/24593_APM_v21.pdf.
[7] Microsoft, "National Language Support (NLS) API Reference," [Online]. Available:
https://www.microsoft.com/resources/msdn/goglobal/default.mspx.
[8] Microsoft, "Kernel-Mode Device Driver TSP," [Online]. Available:
https://msdn.microsoft.com/en-us/library/ms725209(v=vs.85).aspx.

# Appendix 1. Index of Supplemental File Repository

Files are available at: https://github.com/williamshowalter/hdroot-bootkit-analysis

## 1.1 Binary Files – hdroot-bootkit-analysis/binaries

| File | Description |
| --- | --- |
| C_932.NLS | 64-bit bootkit service DLL sample, as installed |
| driver32.sys.bin | 32-bit kernel driver used by the dropper to write directly to the physical disk. |
| driver64.sys.bin | 64-bit signed kernel driver used by the dropper to write directly to the physical disk. |
| dropper64.bin | 64-bit dropper sample that installs bootkit |
| mbr-clean.bin | MBR before modification, for comparison. |
| mbr-inst.bin | MBR that has been modified after install. |
| pe1_decrypted.bin | 32-bit bootkit service DLL sample, extracted and decrypted from decrypted rkimage |
| pe1_encrypted_b61e1dcf.bin | 32-bit bootkit service DLL sample, extracted in original form from decrypted rkimage. XOR key is 0xb64e1dcf. |
| pe2_decrypted.bin | 64-bit bootkit service DLL sample, extracted and decrypted from decrypted rkimage |
| pe2_encrypted_b61e8d81.bin | 64-bit bootkit service DLL sample, extracted in original form from decrypted rkimage. XOR key is 0xb64e8d81. |
| rkimage_decrypted.bin | rkImage sample, extracted from harddrive and decrypted. |

| File | Description |
| --- | --- |
| rkimage_encrypted.bin | rkImage sample, extracted from harddrive and decrypted. |
| rkimage_backdoor_decrypted.bin | rkImage sample with example backdoor, extracted from harddrive and decrypted. |
| rkimage_backdoor_encrypted.bin | rkImage sample with example backdoor, extracted from harddrive. Obfuscated with 0x76 byte-XOR. |
| verifier_win7_decrypted.bin | verifier sample, containing the verifier sector followed by two copies of the original mbr sector. |
| verifier_win7_encrypted.bin | Verifier sample, containing the verifier sector followed by two copies of the original mbr sector. Obfuscated with 0x76 byte-XOR. |
| verifier_win10_decrypted.bin | verifier sample, containing the verifier sector followed by two copies of the original mbr sector. |
| verifier_win10_encrypted.bin | Verifier sample, containing the verifier sector followed by two copies of the original mbr sector. Obfuscated with 0x76 byte-XOR. |

## 1.2 Code Files – hdroot-bootkit-analysis/code

| File | Description |
| --- | --- |
| convert.c | C utility to decrypt verifier and rkimage samples. |
| dll_decryptor.c | C utility to decrypt service DLL samples with 4-byte XOR keys. |
| fuzzer.py | Simple python fuzzer to discover commands to dropper64.bin |
| proof.cpp | C++ program to install as backdoor. Writes C:\proof.txt as evidence that bootkit ran successfully. |

## 1.3 Evidence Files – hdroot-bootkit-analysis/evidence

| File | Description |
| --- | --- |
| crc_error.PNG | Error message shown by check command when secondary bootkit image is modified after install. |
| driver64_certificate.PNG | Screenshot of the stolen certificate used by the 64-bit kernel driver. |

| File | Description |
| --- | --- |
| driver64_valid.PNG | Screenshot showing that the certificate on the kernel driver has not been revoked. |
| dropper64_certificate.PNG | Screenshot of the stolen certificate used by the 64-bit dropper. |
| dropper64_revoked.PNG | Screenshot showing that the certificate on the dropper has been revoked. |
| hashes_after.txt | Hashes taken of files after the bootkit has run on a Windows 7 virtual machine. |
| hashes_before.txt | Hashes taken of files before the bootkit has run on a Windows 7 virtual machine. |
| hashes_win10.txt | Hashes of the first and second rkImage locations on a Windows 10 virtual machine with > 30% free space. |
| install_win10.PNG | Screenshot of installing a backdoor on Windows 10. |
| install_win10_cmd.PNG | Screenshot of installing cmd.exe as the backdoor. |
| install_win7.PNG | Screenshot of installing a backdoor on Windows 7 with low disk space. |
| installer_cmd.txt | The text output of installing a backdoor on Windows 10. |
| Neowiz.p7b | Extracted certificate used in the 64-bit kernel driver. |
| reg_service_after.txt | Registry after boot, with timestamps showing it was written to, even if the values didn't change. |
| reg_service_before.txt | Registry before rebooting, with timestamps. |
| vol_modules.txt | Volatility output snippet from listing modules that shows the kernel driver. |
| vol_reg_debugfile.txt | Volatility output that shows a registry key for the DEBUGFILE service used by the kernel driver. |

## 1.4 Ida Pro Files – hdroot-bootkit-analysis/ida pro

| File | Description |
| --- | --- |
| driver32.sys.idb | Ida Pro file for the 32-bit kernel driver. Functionally same as the 64-bit driver. |

| File | Description |
|---|---|
| driver64.sys.idb | Ida Pro file for the 64-bit kernel driver. Functionally same as the 32-bit driver. |
| dropper64.i64 | Ida Pro file for the dropper sample. Largely not reversed, as the static sample is packed with VMProtect. |
| mbr_infected.idb | Ida Pro file for the bootkit MBR. Disassembly is 16-bit. |
| pe1_decrypted.idb | Ida Pro file for the 32-bit service DLL. Functionally same as the 64-bit DLL. |
| pe2_decrypted.i64 | Ida Pro file for the 64-bit service DLL. Functionally same as the 32-bit DLL. |
| rkimage_decrypted.idb | Ida Pro file for rkImage. Contains real mode (16-bit) and protected mode (32-bit) segments. Also has undefined data at the end because the sample disassembled was mistakenly longer than the rkimage+bootkit length. |
| verifier_decrypted.idb | Ida Pro file for the verifier. Contains verifier and original MBR. Disassembly is 16-bit. |

# Appendix 2: Sample Hashes

## MD5

2c85404fe7d1891fd41fcee4c92ad305 dropper64.bin
4dc2fc6ad7d9ed9fcf13d914660764cd driver32.sys.bin
8062cbccb2895fb9215b3423cdefa396 driver64.sys.bin
c7fee0e094ee43f22882fb141c089cea pe1_decrypted.bin
d0cb0eb5588eb3b14c9b9a3fa7551c28 pe2_decrypted.bin
76e1e42988befbf13b4f934604206250 rkimage_encrypted.bin
613fd19d0abc3d018ead52afabd59fec rkimage_decrypted.bin
287fac6f4dac57253ac0061be1508f9d C_932.NLS.bin

## SHA1

4c3171b48d600e6337f1495142c43172d3b01770 dropper64.bin
7ff22bd8667ce23e7db8c759bd03c15fb7226c76 driver32.sys.bin
268dd909933c187d2798b5815674d70b930b498e driver64.sys.bin
24a80cd100274e2c39180741aa688a4e73282552 pe1_decrypted.bin
5d6c1a3c2d827c714b764b1c5a3e7370ed737986 pe2_decrypted.bin

aaf677acc05ae94f98f836fb44fd672a4b2d90db rkimage_encrypted.bin
3c22ef94a737484e2f708393dcbabdfdb9d6cfbc rkimage_decrypted.bin
88912b5227145d3a715ae6eeebd5935c89955721 C_932.NLS.bin
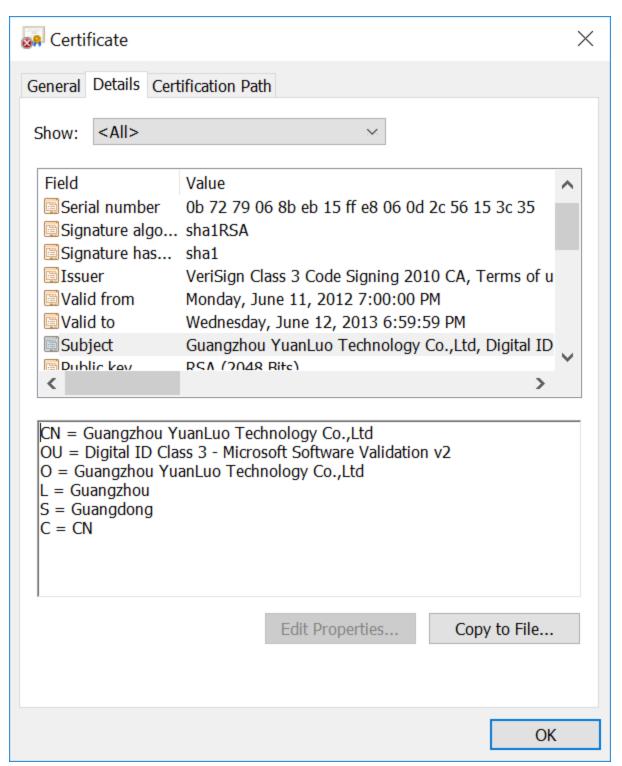
## Appendix 3: Screenshots

## Dropper

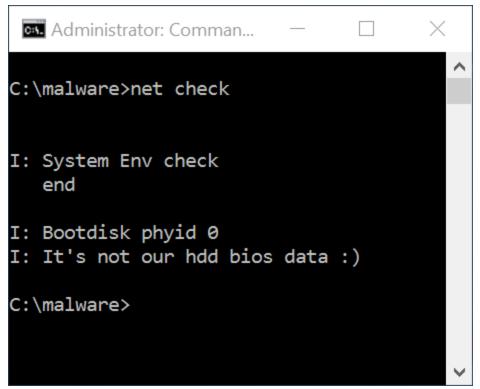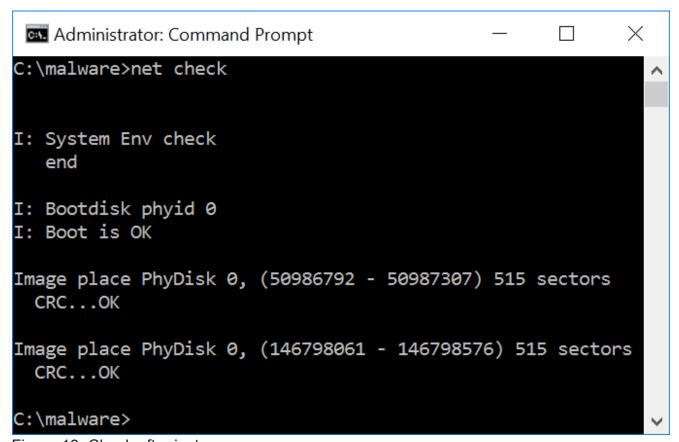Figure 11: Dropper's certificate

Figure 12: Check before inst


Figure 13: Check after inst

Figure 14: Info for backdoor