De-obfuscating and reversing the user-mode agent dropper

resources.infosecinstitute.com/step-by-step-tutorial-on-reverse-engineering-malware-the-zeroaccessmaxsmiscer-crimeware-rootkit/

Reverse engineering
November 12, 2010 by **Giuseppe Bonfa**

Part 1: Introduction and De-Obfuscating and Reversing the User-Mode Agent Dropper

Summary

This four part article series is a complete step-by-step tutorial on how to reverse engineer the ZeroAccess Rootkit. ZeroAcess is also known as the *Smiscer* or *Max++ rootkit.* You can either read along to gain an in-depth understand the thought process behind reverse engineering modern malware of this sophistication. The author prefers that you download the various tools mentioned within and reverse the rookit yourself as you read the article.

If you would like to use the malware sample used in these articles, download it here:

[download]

InfoSec Institute would classify ZeroAccess as a sophisticated, advanced rootkit. It has 4 main components that we will reverse in great detail in this series of articles. ZeroAccess is a compartmentalized crimeware rootkit that serves as a platform for installing various malicious programs onto victim computers. It also supports features to make itself and the installed malicious programs impossible for power-users to remove and very difficult security experts to forensically analyze.

At the conclusion of the analysis, we will trace the criminal origins of the ZeroAccess rootkit. We will discover that the purpose of this rootkit is to set up a stealthy, undetectable and un-removable platform to deliver malicious software to victim computers. We will also see that ZeroAccess is being currently used to deliver FakeAntivirus crimeware applications that trick users into paying $70 to remove the “antivirus”. It could be used to deliver any malicious application, such as one that steals bank and credit card information in the future. Further analysis and network forensics supports that ZeroAccess is being hosted and originates from the Ecatel Network, which is controlled by the cybercrime syndicate RBN (Russian Business Network).

Symantec reports that 250,000+ computers have been infected with this rootkit. If 100% of users pay the $70 removal fee, it would net a total of $17,500,000. As it is not likely that 100% of users will pay the fee, assuming that perhaps 30% will, resulting $5,250,000 in revenue for the RBN cybercrime syndicate.

It has the following capabilities:

- Modern persistence hooks into the OS – Make it very difficult to remove without damaging the host OS
- Ability to use a low level API calls to carve out new disk volumes totally hidden from the infected victim, making traditional disk forensics impossible or difficult.
- Sophisticated and stealthy modification of resident system drivers to allow for kernel-mode delivery of malicious code
- Advanced Antivirus bypassing mechanisms.
- Anti Forensic Technology – ZeroAccess uses low level disk and filesystem calls to defeat popular disk and in-memory forensics tools
- Serves as a stealthy platform for the retrieval and installation of other malicious crimeware programs
- Kernel level monitoring via Asynchronous Procedure Calls of all user-space and kernel-space processes and images, and ability to seamlessly inject code into any monitored image

In this tutorial, our analysis will follow the natural execution flow for a new infection. This will result in a detailed chronology of the infection methodology and “workflow” that the rootkit uses to infect hosts. This conceptual workflow is repeated in many other advanced rootkit that have been analyzed, so it behooves you to understand this process and therefore be able to apply it to new malware reversing situations.

Usually, when a rootkit infects a host, the workflow is structured as follows:

- Infection vector allows for rootkit agent reaches victim's system. (Drive-by-download, client side exploit or a dropper)
- User-mode agent execution
- Driver executable decryption and execution
- System hiding from Kernel-mode.
- Establishment on the host and Kernel-mode level monitoring/data-stealing.
- Sending of stolen data in a covert data channel.

Our analysis of ZeroAccess is split into a series of articles:

Part 1: Introduction and De-Obfuscating and Reversing the User-Mode Agent Dropper

Part 2: Reverse Engineering the Kernel-Mode Device Driver Stealth Rootkit

Part 3: Reverse Engineering the Kernel-Mode Device Driver Process Injection Rootkit

Part 4: Tracing the Crimeware Origins of ZeroAccess Rootkit by Reversing the Injected Code

Our analysis starts from analyzing the User-mode Agent and finishes at Kernel-mode where the rootkit drops two malicious device drivers.

Step-by-step analysis

The ZeroAccess rootkit comes in the form of a malicious executable that delivered via infected Drive by Download Approach. Drive-by download means three things, each concerning the unintended download of computer software from the Internet:

1. Downloads which a person authorized but without understanding the consequences (e.g. downloads which install an unknown or counterfeit executable program, ActiveX component, or Java applet).
2. Any download that happens without a person's knowledge.
3. Download of spyware, a computer virus or any kind of malware that happens without a person's knowledge.

Drive-by downloads may happen when visiting a website, viewing an e-mail message or by clicking on a deceptive pop-up window by clicking on the window in the mistaken belief that, for instance, an error report from the computer itself is being acknowledged, or that an innocuous advertisement pop-up is being dismissed. In such cases, the "supplier" may claim that the person "consented" to the download although actually unaware of having started an unwanted or malicious software download. Websites that exploit the Windows Metafile vulnerability may provide examples of drive-by downloads of this sort.

ZeroAccess has some powerful rootkit capabilities, such as:

- Anti FileSystem forensics by modifying and infecting critical system drivers (disk.sys, atapi.sys) as well as PIC driver object stealing and IRP Hooking.
- Infecting of System Drivers.
- User-mode Process Creation interception and DLL Injection, from KernelMode.
- DLL Hiding and Antivirus bypassing.
- Extremely resistant to Infection Removal.

Part 1: Reverse engineering the user-mode agent/dropper

The rootkit is obfuscated via a custom packed executable typically called 'Max++ downloader install_2010.exe'. The hashes for this file are:

MD5: d8f6566c5f9caa795204a40b3aaaafa2

SHA1: d0b7cd496387883b265d649e811641f743502c41

SHA256: d22425d964751152471cca7e8166cc9e03c1a4a2e8846f18b665bb3d350873db

Basic analysis of this executable shows the following PE sections and imports:

Sections: .text .rdata .rsrc

Imports: COMCTL32.dll

The Import Table is left in a very poor condition for analysis. Typically this means that additional and necessary functions will be imported at Run Time. Let's now check the Entry Point Code:

```
00413BC8                 public start
00413BC8 start           proc near
00413BC8                 mov     edi, edi
00413BCA                 push    ebp
00413BCB                 mov     ebp, esp
00413BCD                 xor     ecx, ecx
00413BCF                 mov     edx, ecx
00413BD1                 inc     edx
00413BD2                 mov     eax, edx
00413BD4                 leave
00413BD5                 int     2Dh             ; Internal routine for MSDOS (IRET)
00413BD7                 retn
00413BD7 start           endp
```

The start code is pretty standard, except for an interesting particular, as you can see at 00413BD5 we have an int 2Dh instruction.

The interrupt 2Dh instruction is mechanism used by Windows Kernel mode debugging support to access the debugging interface. When int 2Dh is called, system creates an EXCEPTION_RECORD structure with an exception code of STATUS_BREAKPOINT as well as other specific informations. This exeception is processed by calling KiDebugRoutine.

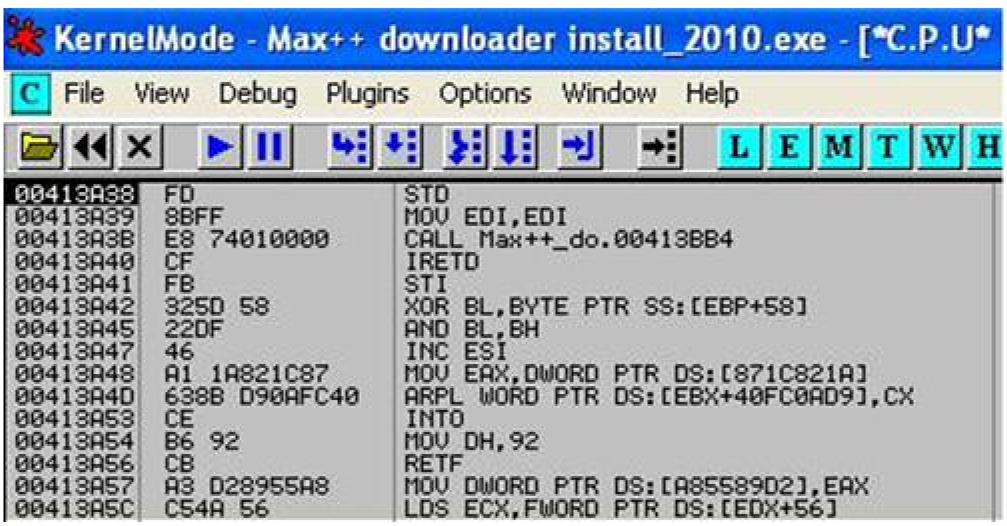
Int 2Dh is used by ntoskrnl.exe to interact with DebugServices but we can use it also in user-mode. If we try to use it in normal (not a debugged) application, we will get exception. However if we will attach debugger, there will be no exception.

(You can read more about this at the OpenRCE reference library http://www.openrce.org/reference_library/anti_reversing_view/34/INT%202D%20Debugger%20Detection/)

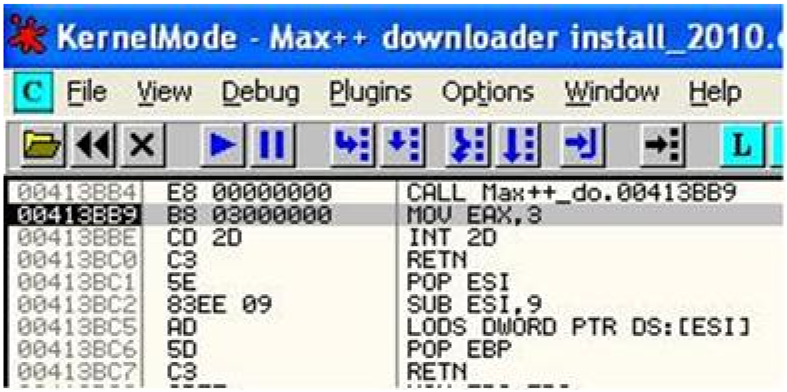
When int 2Dh is called we get our first taste of ZeroAccess anti-reversing and code obsfuction functionality. The system will skip one byte after the interrupt, leading to opcode scission. The actual instructions executed will differ from the apparent instructions that will be displayed in a dissassembler or debugger.

To continue further we need a mechanism to correctly handle int 2Dh call and mantain the jump-one-byte feature, and allow us to follow the opcode-splitted code. To do so, we are going to use StrongOD Olly plugin which can be downloaded here: http://reversengineering.wordpress.com/2010/07/26/strongod-0-3-4-639/

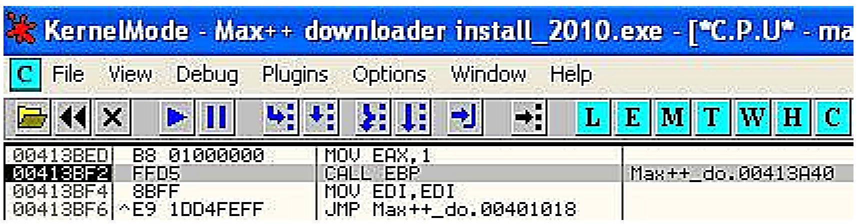
With StrongOD installed, after tracing over int 2Dh we are presenting with the following instructions:



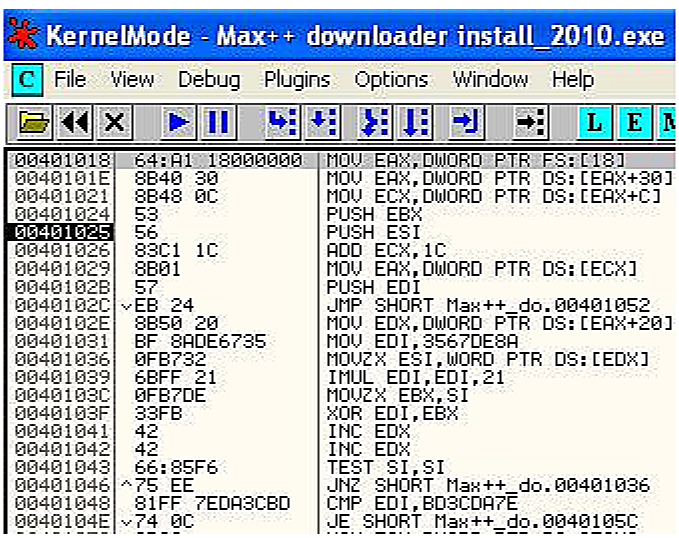
The most interesting instruction for us here is the Call 00413bb4. Immediately after this instruction we have garbage code. Let’s enter into this call, and you are now presented with the following code block:



Again, we see int 2Dh, which will lead us one byte after the RETN instruction. The next piece of code will decrypt the adjacent routine, after tracing further, finally we land here:



This call will decrypt another block of code, at after that call execution jump here:



FS:[18] corresponds to TEB (Thread Environment Block) address, from TEB is obtained PEB (Process Environment Block) which is located at TEB Address + 30h.

PEB+0C corresponds to PPEB_LDR_DATA LdrData.

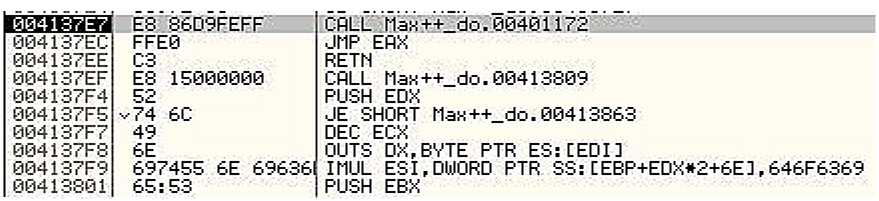
If you are using WinDBG, you can use this quick hint to uncover the link between structure -> offset ->involved member by issuing the following command:

```
0:004> dt nt!_PEB_LDR_DATA
ntdll!_PEB_LDR_DATA
```

+0x000 Length : Uint4B
+0x004 Initialized : UChar
+0x008 SsHandle : Ptr32 Void
+0x00c InLoadOrderModuleList : _LIST_ENTRY
+0x014 InMemoryOrderModuleList : _LIST_ENTRY
+0x01c InInitializationOrderModuleList : _LIST_ENTRY
+0x024 EntryInProgress : Ptr32 Void
+0x028 ShutdownInProgress : UChar
+0x02c ShutdownThreadId : Ptr32 Void

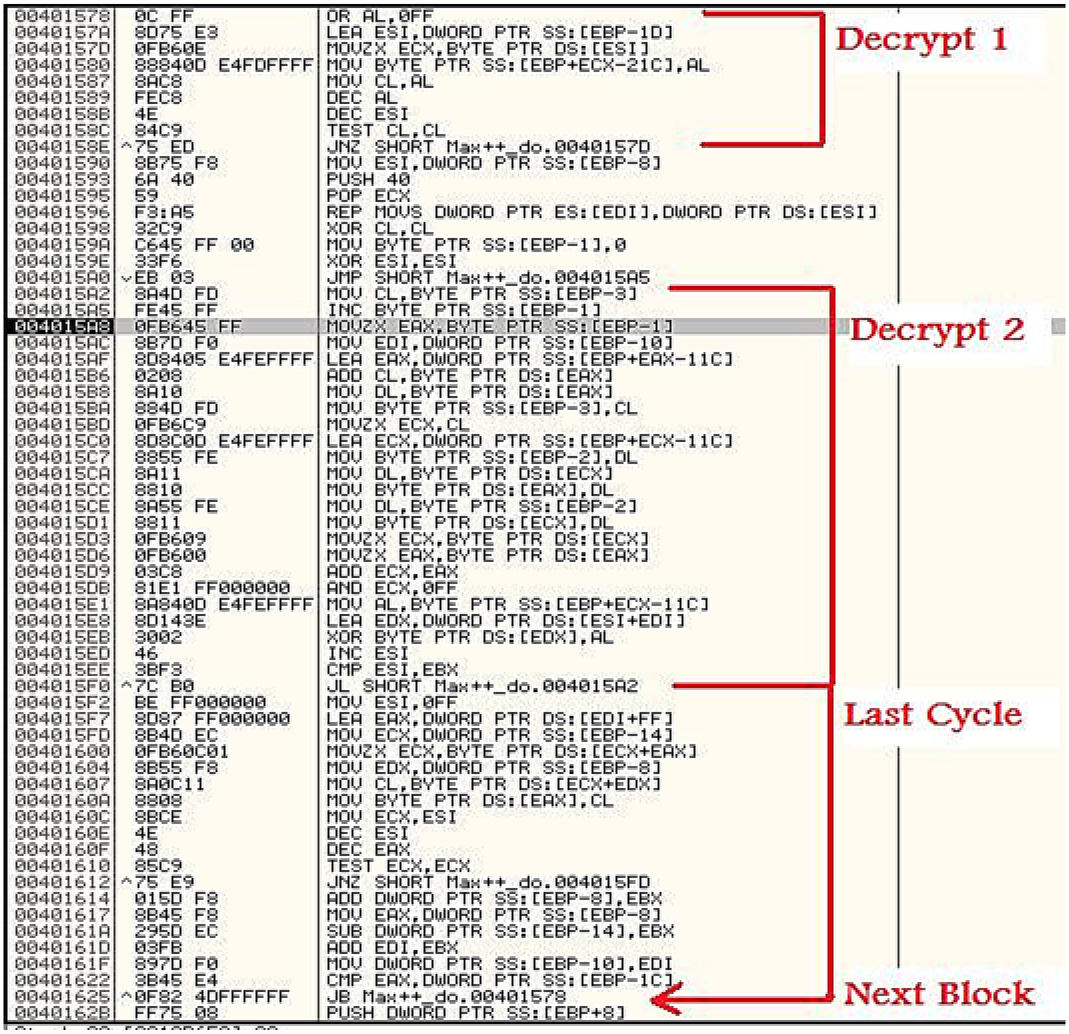
As you can see, the malicious code refers to _PEB_LDR_DATA + 1Ch, by checking the output of WinDbg you can see that ECX now points to InInitializationOrderModuleList. The code that follows is responsible for locating Import Function addresses and then from this information building an ImportTable on the fly dynamically. Next there is a complex sequence of nested calls that have the principal aim of decrypting, layer by layer, the core routines of ZeroAccess. We will not describe the analysis of this piece of multi-layer code; it is left as an exercise for the reader. This section of code is quite long, repetitive, and frankly boring, and not relevant from a functionality point of view.

Imported Function addresses are successively protected and will be decrypted on fly only when they are called. Let’s take a look at how an API call actually looks:


```
004137E7  E8 86D9FEFF     CALL Max++_do.00401172
004137EC  FFE0            JMP EAX
004137EE  C3              RETN
004137EF  E8 15000000     CALL Max++_do.00413809
004137F4  52              PUSH EDX
004137F5 v74 6C           JE SHORT Max++_do.00413863
004137F7  49              DEC ECX
004137F8  6E              OUTS DX,BYTE PTR ES:[EDI]
004137F9  697455 6E 69636 IMUL ESI,DWORD PTR SS:[EBP+EDX*2+6E],646F6369
00413801  65:53           PUSH EBX
```

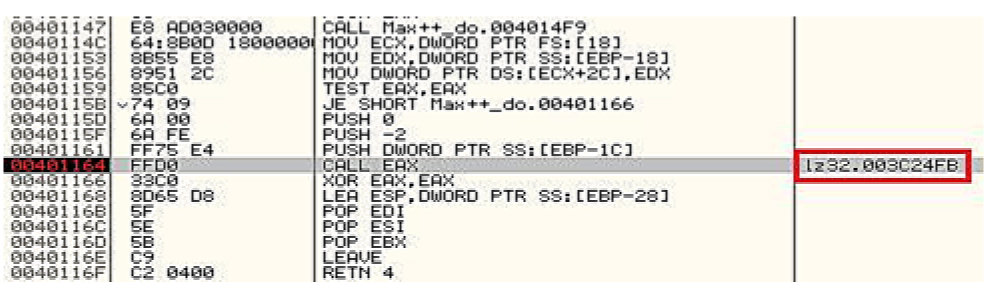
Call 00401172 decrypts and return the API’s address in EAX. In the above code snippet, the API called is VirtualAlloc. Allocated memory will be used in future execution paths to decrypt a number of different blocks of instructions. These blocks will eventually constitute an executable dropped by the original infection agent.

Main executable (the infection vector we are also referring to as the Agent) builds and drops various files into victim’s hard disk and as well as in memory. Whether on disk or in memory, the pattern used is always the same:



Next, let’s try to determine what is being decrypted in these blocks. We place a breakpoint at 0040162B, which is immediately after Next Block jump. The end of the Next Block corresponds to the end of decryption process, we will see in allocated memory the familiar ‘MZ’ signature, letting us know the executable is ready to be used. Before proceding we recommending dumping onto the the hard drive the full executable using the Backup functionality of Ollydbg.

The next block of code is protected with a VEH (Vectored Exception Handler) by using RtlAddVectoredExceptionHandler and RtlRemoveVectoredExceptionHandler. Inside this block we have a truly important piece of code. This block is loaded via the undocumented native API call, LdrLoadDll. A system DLL is called, lz32.dll, as well as the creation of a Section Object.



A Section Object represents a section of memory that can be shared. A process can use a section object to share parts of its memory address space (memory sections) with other processes. Section objects also provide the mechanism by which a process can map a file into its memory address space.

Take a look at the red rectangle, calling the value 003C24FB stored in EAX. As you can see this belongs to the previously loaded lz32.dll. Because of this call, execution flow jumps inside the lz32.dll, and which contains malicious code decrypted by the rootkit agent.

This is what the code of lz32.dll program looks like:

```
003C24FB   837C24 08 FE    CMP DWORD PTR SS:[ESP+8],-2
003C2500  ~75 05           JNZ SHORT lz32.003C2507
003C2502   E8 D4FEFFFF     CALL lz32.003C23DB
003C2507   B0 01           MOV AL,1
003C2509   C2 0C00         RETN 0C
```

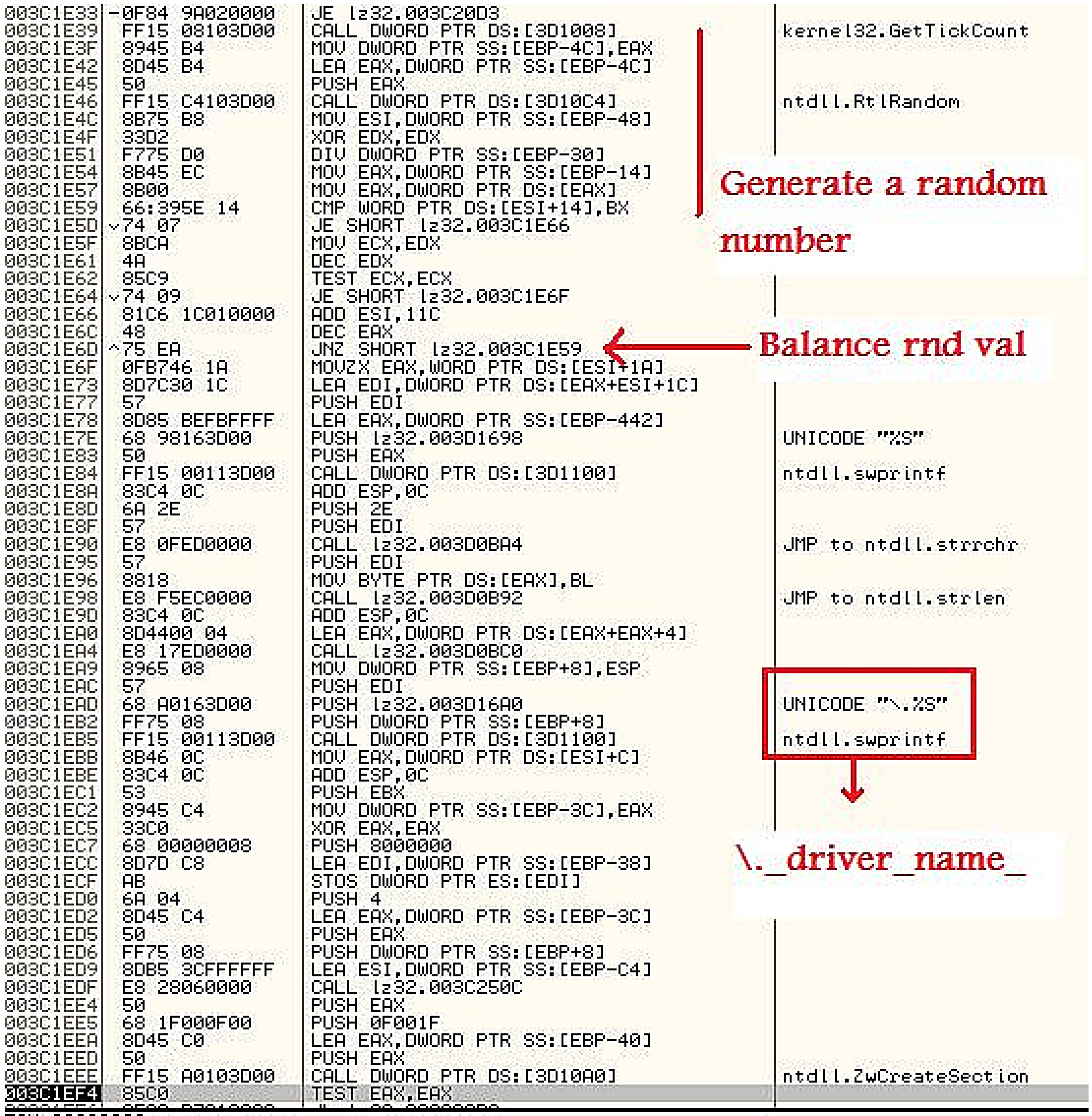
If we trace into the Call 003C23DB, we have a long routine that completes infection, and more precisely we have the kernel mode component installation phase. We will see a series of creative routines specifically written to elude classic Antivirus checks, such as the usage of Section Objects and Views placed into System Files.

Now, let’s take a look at the core routine of the Agent, which we will analyze piece by piece:

```
003C23F3  FF15 E8103D00    CALL DWORD PTR DS:[3D10E8]           ntdll.RtlAdjustPrivilege
003C23F9  8D4424 24        LEA EAX,DWORD PTR SS:[ESP+24]
003C23FD  50               PUSH EAX
003C23FE  E8 26EDFFFF      CALL lz32.003C1129
003C2403  E8 8BEDFFFF      CALL lz32.LZCopy
003C2408  85C0             TEST EAX,EAX
003C240A v0F84 E4000000    JE lz32.003C24F4
003C2410  68 00000100      PUSH 10000                           UNICODE "=::=::\"
003C2415  6A 07            PUSH 7
003C2417  8D4424 24        LEA EAX,DWORD PTR SS:[ESP+24]
003C241B  50               PUSH EAX
003C241C  BF 50313D00      MOV EDI,lz32.003D3150
003C2421  57               PUSH EDI
003C2422  BE 00001000      MOV ESI,100000
003C2427  56               PUSH ESI
003C2428  8D4424 28        LEA EAX,DWORD PTR SS:[ESP+28]
003C242C  50               PUSH EAX
003C242D  FF15 DC103D00    CALL DWORD PTR DS:[3D10DC]           ntdll.ZwOpenFile
003C2433  3BC3             CMP EAX,EBX
003C2435  894424 18        MOV DWORD PTR SS:[ESP+18],EAX
003C2439 v7C 0A            JL SHORT lz32.003C2445
003C243B  FF7424 14        PUSH DWORD PTR SS:[ESP+14]
003C243F  FF15 E4103D00    CALL DWORD PTR DS:[3D10E4]           ntdll.ZwClose
003C2445  817C24 18 71020  CMP DWORD PTR SS:[ESP+18],C0000271
003C244D v75 07            JNZ SHORT lz32.003C2456
003C244F  53               PUSH EBX
003C2450  FF15 04103D00    CALL DWORD PTR DS:[3D1004]           kernel32.ExitProcess
003C2456  B8 700B3D00      MOV EAX,lz32.003D0B70
003C245B  2D 60BF3C00      SUB EAX,lz32.003CBF60
003C2460  50               PUSH EAX
003C2461  E8 C6F7FFFF      CALL lz32.003C1C2C
003C2466  E8 F7FCFFFF      CALL lz32.003C2162
003C246B  8D4424 24        LEA EAX,DWORD PTR SS:[ESP+24]
003C246F  50               PUSH EAX
003C2470  E8 32FDFFFF      CALL lz32.003C21A7
003C2475  8D4424 24        LEA EAX,DWORD PTR SS:[ESP+24]
003C2479  50               PUSH EAX
003C247A  E8 8BFDFFFF      CALL lz32.003C220A
003C247F  68 50AD3C00      PUSH lz32.003CAD50
003C2484  68 68313D00      PUSH lz32.003D3168
003C2489  B8 40253C00      MOV EAX,lz32.003C2540
003C248E  E8 26F2FFFF      CALL lz32.003C16B9
003C2493  68 60BF3C00      PUSH lz32.003CBF60
003C2498  68 80313D00      PUSH lz32.003D3180
003C249D  B8 50AD3C00      MOV EAX,lz32.003CAD50
003C24A2  E8 12F2FFFF      CALL lz32.003C16B9
003C24A7  53               PUSH EBX
003C24A8  53               PUSH EBX
003C24A9  68 00200000      PUSH 2000
003C24AE  6A 02            PUSH 2
003C24B0  53               PUSH EBX
003C24B1  53               PUSH EBX
003C24B2  68 98313D00      PUSH lz32.003D3198
003C24B7  8D4424 38        LEA EAX,DWORD PTR SS:[ESP+38]
003C24BB  50               PUSH EAX
003C24BC  57               PUSH EDI
003C24BD  56               PUSH ESI
003C24BE  8D4424 3C        LEA EAX,DWORD PTR SS:[ESP+3C]
003C24C2  50               PUSH EAX
003C24C3  FF15 74103D00    CALL DWORD PTR DS:[3D1074]           ntdll.ZwCreateFile
003C24C9  8BF0             MOV ESI,EAX
003C24CB  3BF3             CMP ESI,EBX
003C24CD v7C 0A            JL SHORT lz32.003C24D9
```

During the analysis of complex pieces of malware it’s a good practice to leave open the HandleView and ModuleView panes within OllyDbg. This will help you keep track of what is loaded/unloaded and what files/objects/threads/etc. are opened. Let’s see what happens in Call 003C1C2C at address 003C2461.

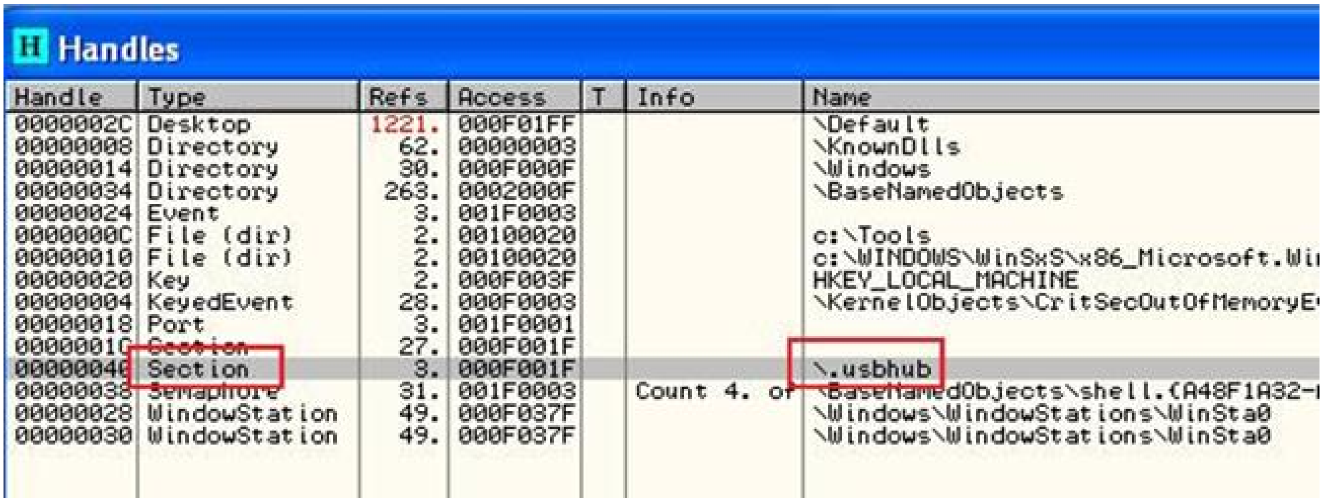
At first, we see the enumeration of Drivers placed into system32drivers, and next we have the following piece of code:



We have an interesting algorithm here, after driver enumeration a random number is generated, next fitted within a range of [0 – 0xFF] and used to randomly select from the driver list a file to be infected. Finally the string formatted as:

._driver_name_

Now let’s watch what is going on in HandleView:



Handles

Handle	Type	Refs	Access	T	Info	Name
0000002C	Desktop	1221.	000F01FF			\Default
00000008	Directory	62.	00000003			\KnownDlls
00000014	Directory	30.	000F000F			\Windows
00000034	Directory	263.	0002000F			\BaseNamedObjects
00000024	Event	3.	001F0003			
0000000C	File (dir)	2.	00100020			c:\Tools
00000010	File (dir)	2.	00100020			c:\WINDOWS\WinSxS\x86_Microsoft.Wi
00000020	Key	2.	000F003F			HKEY_LOCAL_MACHINE
00000004	KeyedEvent	28.	000F0003			\KernelObjects\CritSecOutOfMemoryE
00000018	Port	3.	001F0001			
0000001C	Section	27.	000F001F			
00000040	Section	3.	000F001F			\.usbhub
00000038	Semaphore	31.	001F0003		Count 4. of	\BaseNamedObjects\shell.{A48F1A32-
00000028	WindowStation	49.	000F037F			\Windows\WindowStations\WinSta0
00000030	WindowStation	49.	000F037F			\Windows\WindowStations\WinSta0

As you can see a Section Object is created according to the randomly selected driver file, and next will be opened as View inside this Section.

The access values for this section are set to 0xF001F. Let’s first talk about why this is important. During a malware analysis session, much like a forensic investigation, is fundamental to know what the access potential the various components have, so we can direct our investigation down the right path. This can be determined by checking the access rights assigned to various handles.

Let’s lookup what the access right of 0xF001F corresponds by looking in winnt.h:

#define SECTION_ALL_ACCESS 0xf001f

SECTION_ALL_ACCESS means the handle has the ability to Read, Write, Query and Execute. This is the optimal environment to place a malicious portion of code. Now, lets analyze further:

```
003C1979  8D85 6CFDFFFF    LEA EAX,DWORD PTR SS:[EBP-294]
003C197F  68 E0143D00      PUSH lz32.003D14E0                UNICODE "\registry\MACHINE\SYSTEM\CurrentControlSet\services\%s"
003C1984  50               PUSH EAX
003C1985  FF15 00113D00    CALL DWORD PTR DS:[3D1100]        ntdll.swprintf
003C198B  83C4 0C          ADD ESP,0C
003C198E  33FF             XOR EDI,EDI
003C1990  57               PUSH EDI
003C1991  57               PUSH EDI
003C1992  57               PUSH EDI
003C1993  57               PUSH EDI
003C1994  8D85 6CFDFFFF    LEA EAX,DWORD PTR SS:[EBP-294]
003C199A  50               PUSH EAX
003C199B  8D75 AC          LEA ESI,DWORD PTR SS:[EBP-54]
003C199E  E8 690B0000      CALL lz32.003C250C
003C19A3  50               PUSH EAX
003C19A4  68 3F000F00      PUSH 0F003F
003C19A9  8D45 F8          LEA EAX,DWORD PTR SS:[EBP-8]
003C19AC  50               PUSH EAX
003C19AD  FF15 2C113D00    CALL DWORD PTR DS:[3D112C]        ntdll.ZwCreateKey
003C19B3  85C0             TEST EAX,EAX
003C19B5 v0F8C DB010000    JL lz32.003C1B96
003C19BB  8B35 98103D00    MOV ESI,DWORD PTR DS:[3D1098]     ntdll.ZwSetValueKey
003C19C1  6A 04            PUSH 4
003C19C3  5B               POP EBX
003C19C4  53               PUSH EBX
003C19C5  8D45 F0          LEA EAX,DWORD PTR SS:[EBP-10]
003C19C8  50               PUSH EAX
003C19C9  53               PUSH EBX
003C19CA  57               PUSH EDI
003C19CB  68 5C153D00      PUSH lz32.003D155C
003C19D0  FF75 F8          PUSH DWORD PTR SS:[EBP-8]
003C19D3  C745 F0 0100000  MOV DWORD PTR SS:[EBP-10],1
003C19DA  FFD6             CALL ESI
003C19DC  53               PUSH EBX
003C19DD  8D45 F0          LEA EAX,DWORD PTR SS:[EBP-10]
003C19E0  50               PUSH EAX
003C19E1  53               PUSH EBX
003C19E2  57               PUSH EDI
003C19E3  68 70153D00      PUSH lz32.003D1570
003C19E8  FF75 F8          PUSH DWORD PTR SS:[EBP-8]
003C19EB  C745 F0 0300000  MOV DWORD PTR SS:[EBP-10],3
003C19F2  FFD6             CALL ESI
003C19F4  6A 06            PUSH 6
003C19F6  68 94153D00      PUSH lz32.003D1594                UNICODE "\*"
003C19FB  6A 01            PUSH 1
003C19FD  57               PUSH EDI
003C19FE  68 8C153D00      PUSH lz32.003D158C
003C1A03  FF75 F8          PUSH DWORD PTR SS:[EBP-8]
003C1A06  FFD6             CALL ESI
```

This block of code takes the driver previously selected and now registers it into:

registryMACHINESYSTEMCurrentControlSetservices

The services entry under CurrentControlSet contains parameters for the device drivers, file system drivers, and Win32 service drivers. For each Service, there is a subkey with the name of the service itself. Our registry entry will be named ._driver_name_

Start Type has 0x3 value that means -> Load on Demand

Type: 0x1 -> Kernel Device Driver

Image Path -> *

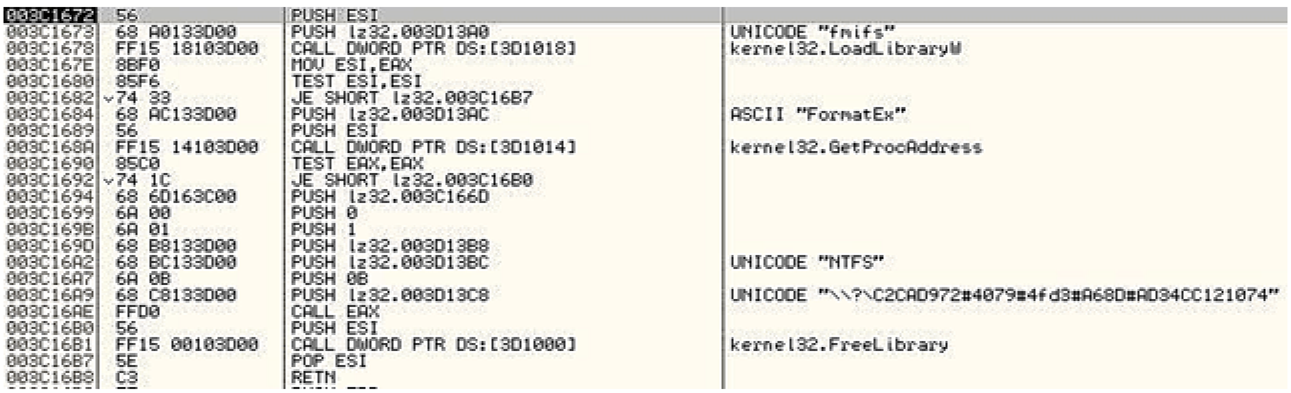
```
003C1A3C  50               PUSH EAX
003C1A3D  FF15 DC103D00    CALL DWORD PTR DS:[3D10DC]        ntdll.ZwOpenFile
003C1A43  85C0             TEST EAX,EAX
003C1A45 v0F8C B8000000    JL lz32.003C1B03
003C1A4B  57               PUSH EDI
003C1A4C  57               PUSH EDI
003C1A4D  6A 02            PUSH 2
003C1A4F  68 AC313D00      PUSH lz32.003D31AC
003C1A54  68 40C00900      PUSH 9C040
003C1A59  8D45 CC          LEA EAX,DWORD PTR SS:[EBP-34]
003C1A5C  50               PUSH EAX
003C1A5D  57               PUSH EDI
003C1A5E  57               PUSH EDI
003C1A5F  57               PUSH EDI
003C1A60  FF75 F4          PUSH DWORD PTR SS:[EBP-C]
003C1A63  FF15 9C103D00    CALL DWORD PTR DS:[3D109C]        ntdll.ZwFsControlFile
003C1A69  FF75 F4          PUSH DWORD PTR SS:[EBP-C]
003C1A6C  8D45 E8          LEA EAX,DWORD PTR SS:[EBP-18]
003C1A6F  68 00000008      PUSH 8000000
003C1A74  53               PUSH EBX
003C1A75  57               PUSH EDI
003C1A76  57               PUSH EDI
003C1A77  6A 06            PUSH 6
003C1A79  50               PUSH EAX
003C1A7A  FF15 A0103D00    CALL DWORD PTR DS:[3D10A0]        ntdll.ZwCreateSection
```

The same driver is always opened. Next, its handle used to send, via ZwFsControlCode, a FSCTL (File System Control Code). Taking a look at the API parameters at run time reveals that the FSCTL code is 9C040. This code corresponds to FSCTL_SET_COMPRESSION. It sets the compression state of a file or directory on a volume whose file system supports per-file and per-directory compression.

Next, a new executable will be built with the aforementioned decryption scheme and then loaded via ZwLoadDriver. This process will result in two device drivers:

1. The first driver is unnamed and will perform IRP Hooking and Object and disk.sys/pci.sys Object Stealing (we will analyze this in greater detail later)
2. The second driver, named B48DADF8.sys, is process creation aware and contains a novel DLL injection system (we will also analyze it greater detail later)

Once the driver infection is complete we land in an interesting piece of code:



Here, we see the loading of fmifs.dll. This DLL is the Format Manager for Installable File Systems, and it offers a set of functions for FileSystem Management.

In this case the exported function is FormatEx. A bit of documentation on FormatEx follows:

```
VOID
STDCALL
FormatEx(
PWCHAR        DriveRoot,
DWORD        MediaFlag,
PWCHAR Format,
PWCHAR        Label,
BOOL        QuickFormat,
DWORD        ClusterSize,
PFMIFSCALLBACK    Callback
);
```

This function, as the name suggests is used to Format Volumes. In our case the DriverRoot is ?C2CAD972#4079#4fd3#A68D#AD34CC121074 and Format is NTFS. This is a remarkable feature unique to this rootkit. This call creates a hidden volume, and the volume will contain the driver and DLLs dropped by the ZeroAccess Agent. These files remain totally invisible to the victim (something we teach in our ethical hacking course).

The next step the Agent takes is to build, with the same decryption routine previously described, the remaining malicious executables that will be stored into the newly created hidden volume. These two files are:

- B48DADF8.sys
- max++.00,x86.dll

Both located into the hidden volume, ?C2CAD972#4079#4fd3#A68D#AD34CC121074L. We now we have a good knowledge of what user-mode side of ZeroAccess does, we can focus our attention to Kernel Mode side, by reversing the two drivers and dropped DLL.

Let's continue to follow the workflow of the rootkit. If you are reversing along with us, analysis will logically follow the order of binaries dropped by the Agent. Our first driver to reverse will be the randomly named one, which will be in Part 2 of this tutorial.

Posted: November 12, 2010

Author

Giuseppe Bonfa

VIEW PROFILE

Giuseppe is a security researcher for InfoSec Institute and a seasoned InfoSec professional in reverse-engineering and development with 10 years of experience under the Windows platforms. He is currently deeply focused on Malware Reversing (Hostile Code and Extreme Packers) especially Rootkit Technology and Windows Internals. He has previously worked as Malware Analyst for Comodo Security Solutions as a member of the most known Reverse Engineering Teams and is currently a consultant for private customers in the field of Device Driver Development, Malware Analysis and Development of Custom Tools for Digital Forensics. He collaborates with Malware Intelligence and Threat Investigation organizations and has even discovered vulnerabilities in PGP and Avast Antivirus Device Drivers. As a technical author, Giuseppe has over 10 years of experience and hundreds of published pieces of research.