



Prevx

» ADVANCED MALWARE RESEARCH TEAM

ZeroAccess – an advanced kernel mode rootkit

(rev. 1.2)

Marco Giuliani

Head of Prevx Advanced Malware Research Team

PREFACE

When we write about ZeroAccess rootkit, it is essential to go back in 2009 and to remind when this rootkit had been discovered in the wild. It was the time of MBR rootkit and TDL2 rootkit – the second major release of the most advanced kernel mode rootkit currently in the wild – when security researchers came across a new, previously unknown, rootkit able to kill most of security software as soon as they tried to scan specified folders in the system. ZeroAccess was creating a new kernel device object called `__max++>`, this is the reason why the rootkit has quickly become known in the security field as the max++ rootkit, also known as ZeroAccess due to a string found in the kernel driver code, presumably pointing to the original project folder called ZeroAccess (*f:\VC5\release\ZeroAccess.pdb*).

This rootkit was storing its code in two alternate data streams, *win32k.sys:1* and *win32k.sys:2*. To avoid being detected, it was killing every security software that attempted to scan for alternate data streams. It created in the system folder a number of fake junctions (note: an NTFS junction point is a feature of the NTFS file system that allows a folder to be linked to another local folder, becoming an alias for such target folder) pointing to the fake rootkit device written above. When security software tried to scan such specified folders for Alternate Data Streams presence (*FileStreamInformation* class), the rootkit's self-defense queued a work item in the security process able to immediately kill it. It became a non-trivial job scanning the system without being killed.

Since then, ZeroAccess rootkit evolved, changing the way it infects the system, becoming yet more advanced and dangerous. In this paper we are going to analyze this threat and how it evolved to its current release.

DROPPER ANALYSIS

This rootkit is installed by a dropper which is usually downloaded in the system by crack or warez websites, or still by exploit packs. These are the usual infection vehicles. The dropper implements a number of anti-debugging techniques along with a classic *spaghetti code* able to slow down the job of code analysis. After the first stage unpacking, the code tries to acquire following privileges: SeDebugPrivilege, SeTakeOwnershipPrivilege, SeRestorePrivilege, SeSystemtimePrivilege, SeSecurityPrivilege. Then, it starts the infection payload.

Before analyzing the infection more in detail, it's necessary to briefly describe how ZeroAccess is infecting the system. The dropper chooses randomly a driver in the `systemroot\system32\drivers` folder and it overwrites the original code – saving it for backup purposes. Then, after loaded, the rootkit driver sets up a new disk device object, which will be used as a gate for the hidden volume drive created by the rootkit itself to store its files and data.

This is an effective technique, though similar to the TDL3 rootkit infection. While ZeroAccess sets up a new encrypted hidden volume in the system's filesystem, TDL3 creates a brand new encrypted filesystem in the last sectors of the hard drive, outside the system's filesystem. Both store their files inside these new encrypted volumes, making them totally inaccessible by the operating system. Both rootkits infect a random driver, though while ZeroAccess totally overwrites the driver's body, TDL3 rootkit hijacks the driver's entrypoint, overwriting less than 1KB in the driver's resource section. Other differences are in the disk's I/O filtering engine, much different and less powerful in ZeroAccess than in TDL3 rootkit.

Let's analyze more in depth how the driver's infection routine works in ZeroAccess and how the rootkit chooses the right driver to infect.

- ✓ The rootkit calculates a specific value that will be used as a check for the driver's image size. In the analyzed sample the value is 0x7410 (29712 bytes), which is the size of the rootkit kernel driver. Obviously the target driver should be bigger than that;
- ✓ The rootkit starts enumerating all the system drivers by calling `ZwQuerySystemInformation` with `SystemModuleInformation` class;
- ✓ The target driver must be located between `classnp.sys` driver and `win32k.sys`, every other driver is discarded;
- ✓ All the drivers between `classnp.sys` and `win32k.sys` that have an image size smaller than 0x7410 are discarded;
- ✓ All the drivers bigger than such value are subsequently analyzed. Following parameters are checked:
 - Driver file name must end with a ".sys" extension;
 - Start value in the driver's registry key must be greater than zero (driver should not start at system boot);
 - Driver's PE Export Table size must be zero (the driver should not export anything);
- ✓ If the above listed checks are positive, the driver is marked as "potential good target" by setting the value 1 to its `SYSTEM_MODULE->Id` structure;
- ✓ This analysis loops until all the drivers are analyzed and marked

This 1st loop is used by the rootkit to find all potential target drivers in the system machine. Then, after the loop is finished, the rootkit starts a 2nd loop, which is the one that actually chooses which driver will be infected.

- ✓ The rootkit calculates a random value by calling `GetTickCount` and then `RtlRandom` Win32 APIs;
- ✓ A counter is initialized with the value got from the operation (`RandomValue % NumberOfPotentialTargetsFound`);
- ✓ The rootkit starts again a loop to analyze all system drivers, decreasing the counter each time a potential target driver is found (`SYSTEM_MODULE->Id = 1`);

When the counter is equal to zero, the rootkit has found the target driver that will be infected. The rootkit then creates a new section, called `\<name of the driver that will be infected>` (e.g. `\.NdProxy`), where it temporarily stores a copy of the clean driver

body. The rootkit then creates a new section, called \.<name of the driver that will be infected> (e.g. \.NdProxy), where it temporarily stores a copy of the clean driver body. Then the rootkit creates a new service registry key under *HKLM\SYSTEM\CurrentControlSet\Services* with the value .<name of the driver that will be infected> (e.g. .NdProxy). Inside this registry key, the *ImagePath* value is set to *. This is an obfuscation trick to avoid security software from intercepting the file which is going to be loaded. By passing the value *, security software will be fooled because it apparently doesn't point to any real file. Actually the rootkit's dropper sets a new symbolic link by calling *ZwCreateSymbolicLinkObject* API, pointing * to the real file.

The dropper infects the target driver by fully overwriting the code with its own kernel mode driver and then loads it by calling *ZwLoadDriver*. Before overwriting the driver's body, the dropper makes sure to suspend the System File Checker (SFC) thread by suspending all threads related to the *sfc_os.dll* module. These threads are resumed after the infection routine is finished.

Before executing the real infection payload, the dropper checks if it is running in a WoW64 emulated environment. If so, the process immediately terminates. The rootkit currently doesn't infect x64 based Windows operating systems. Moreover the dropper checks if the infection is already running inside the system by making a specific call to *ZwOpenFile* to try opening the rootkit device. If the system is already infected, the rootkit device will give back the NTSTATUS error *STATUS_VALIDATE_CONTINUE*.

After the rootkit driver has been loaded, the rootkit device *\\?\ACPI#PNP0303#2&da1a3ff&0* (in this sample, though it may change from release to release) can be accessed by user mode and the dropper is able to format the new volume using the NTFS file system. To do so, it loads the *fmifs.dll* module – the Format Manager for Installable File Systems module - and imports the *FormatEx()* API.

```

Format_Virtual_Drive proc near          ; CODE XREF: Infection_Payload+3904p
    push    esi
    push    offset LibFileName ; "fmifs"
    call    ds:LoadLibraryW
    mov     esi, eax
    test    esi, esi
    jz     short loc_402032
    push    offset ProcName ; "FormatEx"
    push    esi             ; hModule
    call    ds:GetProcAddress
    test    eax, eax
    jz     short loc_40202B
    push    offset sub_401FE8
    push    0
    push    1
    push    offset unk_40A3A0
    push    offset aNtFs ; "NTFS"
    push    0Bh
    push    offset a?AcpiPnp03032D ; "\\?\ACPI#PNP0303#2&da1a3ff&0"
    call    eax

loc_40202B:
    push    esi             ; CODE XREF: Format_Virtual_Drive+201j
    call    ds:FreeLibrary

```

The new hidden volume is now ready to store the clean copy of the original overwritten driver. The dropper doesn't use the real file name though, it generates a random file name, based on the following steps:

- ✓ The rootkit queries the following registry key: *HKLM\SYSTEM\CurrentControlSet\Control\agp* by calling *ZwQueryKey* with *KeyBasicInformation* parameter;
- ✓ The rootkit then queries the *_KEY_BASIC_INFORMATION->LastWriteTime* parameter;
- ✓ It generates two specific seed values: the first by doing a XOR between the *LowPart* and the *HighPart* of the *LastWriteTime* parameter (*LastWriteTime.LowPart ^ LastWriteTime.HighPart*); the second is by adding to the new generated seed the original *LowPart* value, then increasing it by 1;
- ✓ It uses a starting string from where it gets the "random" characters that will compose the new file name. The string is: *eaaimnqazwsxedcrfvtgbyhnujmikolp*;
- ✓ The file name that needs to be composed is 8 characters long, so it starts a loop by doing following steps:

- Seed value is and'd with 0x1F (length of the starting string), the returning value is the index of the character in the starting string that will be used in the new file name;
- Seed value is right shifted by 5 using a 64 bit right shift function exported by ntdll.dll (_allshr());

The loop continues until the eight-characters string is composed – starting from the end till the beginning of it. Then the file is stored in the following path:

\\?\ACPI#PNP0303#2&da1a3ff&0\L\Sniifer67, where Snifer67 is replaced with the just generated name.

```

setup_seed:
        ; CODE XREF: generate_name_clean_driver+28fj
        mov     eax, [ebp+LowPart] ; LastWriteTime.Lowpart
        mov     edx, [ebp+HighPart] ; LastWriteTime.HighPart
        xor     edx, eax           ; edx contains (LowPart ^ HighPart)
        push   7
        lea   eax, [eax+edx+1] ; eax contains (edx + LowPart + 1)
        pop    esi

generate_file_name:
        ; CODE XREF: generate_name_clean_driver+77lj
        mov     edi, [ebp+arg_0]
        mov     ecx, eax           ; eax is moved to ecx for math calcs
        and     ecx, 1Fh          ; ecx contains (ecx & 0x1F)
        movsx  cx, ds:Start_String[ecx] ; ecx is now the index used to choose letter from "eaoinmqazwsxedcrfvgtgbyhnujmikolp" string
        mov     [edi+esi*2], cx ; the choosed char is stored in the new string, starting from the end
        mov     cl, 5             ; 5 is the number of times the 64 bit shift should be executed
        call   _allshr           ; 64 bit right shift is executed
        mov     ecx, esi
        dec     esi
        test   ecx, ecx
        jnz    short generate_file_name

```

Asm code of the name generation routine

Which can be roughly translated to the following C code:

```

char* StartingString = "eaoinmqazwsxedcrfvgtgbyhnujmikolp";
char FileName[9];
DWORD index = 7;

RegOpenKeyA(HKEY_LOCAL_MACHINE, "SYSTEM\\CurrentControlSet\\Control\\agp", &regKey);
NtQueryKey(regKey, KeyBasicInformation, &KeyInfo, sizeof(KEY_BASIC_INFORMATION), &result);

seed2 = (KeyInfo.LastWriteTime.HighPart ^ KeyInfo.LastWriteTime.LowPart);
seed = (seed2 + KeyInfo.LastWriteTime.LowPart + 1);

while (index >= 0)
{
    FileName[index] = StartingString[(seed & 0x1F)];
    _allshr(&seed, &seed2, 5);

    if (index == 0)
        break;

    index--;
}

```

Asm code roughly translated to C code

When the file name is generated, the new file is created inside the rootkit device and a copy of the clean driver is stored there.

KERNEL MODE ROOTKIT INFECTION

In this paragraph we are going to analyze more in depth the job of the kernel mode driver dropped by the ZeroAccess rootkit.

As said in the previous paragraph, the rootkit sets up a new device object named *ACPI#PNP0303#2&da1a3ff&0*, which is the gate to access to the rootkit hidden device. Then, it intercepts Windows's disk I/O by hijacking the disk.sys connection to the lower port device. If an attempt to read or write the infected driver is intercepted, the rootkit fakes the file content by showing the original clean copy of the driver.

At driver's startup, the rootkit checks if it's the first time it runs on the system by checking the registry startup key from where it has been executed. If it comes from the *.<drivername>* (e.g. *.NdProxy*) service registry key, then it's the first time and the rootkit deletes that key – it isn't anymore needed.

Then the rootkit reads the path to the infected driver and calculates the hash of the driver path and file name by calling the *RtlHashUnicodeString* function. This hash will be used by the rootkit to check whether someone is trying to get access to the infected driver on the disk. The infected copy of the driver is then stored in memory and pointed by a specific MDL.

The rootkit is now ready to sets up its own code, so it makes a call to the *IoCreateDriver()* native API and sets its own driver object, hiding it from the *DriverSection* and pointing all its dispatch functions to a specific rootkit dispatch routine. To hide the new generated driver object, the rootkit steals the original *\driver\disk* driver object, making a one-to-one copy of the clean *disk.sys*'s driver object to the fake one

```
lkd> dt _DRIVER_OBJECT 0x8201f590
nt!_DRIVER_OBJECT
+0x000 Type           : 4
+0x002 Size           : 168
+0x004 DeviceObject   : 0x81fd5040 _DEVICE_OBJECT
+0x008 Flags          : 0x12
+0x00c DriverStart    : 0xf86cb000
+0x010 DriverSize     : 0x8e00
+0x014 DriverSection  : 0x821edbc0
+0x018 DriverExtension : 0x8201f638 _DRIVER_EXTENSION
+0x01c DriverName     : _UNICODE_STRING "\Driver\Disk"
+0x024 HardwareDatabase : 0x8066e9d8 _UNICODE_STRING "\REGISTRY\MACHINE\HARDWARE\DESCRIPTION\SYSTEM"
+0x028 FastIoDispatch : (null)
+0x02c DriverInit     : 0xf86d28ab long +ffffffff86d28ab
+0x030 DriverStartIo  : (null)
+0x034 DriverUnload   : (null)
+0x038 MajorFunction  : [28] 0xf4b79134 long +ffffffff4b79134
lkd> dt _DRIVER_OBJECT 821eb320
nt!_DRIVER_OBJECT
+0x000 Type           : 4
+0x002 Size           : 168
+0x004 DeviceObject   : 0x821a89f0 _DEVICE_OBJECT
+0x008 Flags          : 0x12
+0x00c DriverStart    : 0xf86cb000
+0x010 DriverSize     : 0x8e00
+0x014 DriverSection  : 0x821edbc0
+0x018 DriverExtension : 0x821eb3c8 _DRIVER_EXTENSION
+0x01c DriverName     : _UNICODE_STRING "\Driver\Disk"
+0x024 HardwareDatabase : 0x8066e9d8 _UNICODE_STRING "\REGISTRY\MACHINE\HARDWARE\DESCRIPTION\SYSTEM"
+0x028 FastIoDispatch : (null)
+0x02c DriverInit     : 0xf86d28ab long +ffffffff86d28ab
+0x030 DriverStartIo  : (null)
+0x034 DriverUnload   : 0xf86e253a void +ffffffff86e253a
+0x038 MajorFunction  : [28] 0xf86e1c30 long +ffffffff86e1c30
```

Fake and original disk driver objects

In the above image we can see both fake and original *disk.sys*'s driver objects. The first one is the fake copy built by the rootkit, the lower one is the original *disk.sys* copy. They are identical, except for the dispatch functions and the Device Object, which the rootkit's driver object points to its own objects.

The rootkit driver object sets up two different device objects, the first one is the device object used to intercept the *disk.sys*'s I/O while the second one is the one we talked about at the beginning of the current paragraph.

To intercept *disk.sys*'s I/O routine, the rootkit hijacks the *\driver\disk*'s *DR0* device object by altering its Device Extension structure. The *DR0_Device_Object->DevExtension->LowerDeviceObject* pointer is modified to point to the rootkit device. The rootkit then intercepts the IRP after it has been processed by *disk.sys* and before it can arrive to the port device driver (e.g. *atapi.sys*), analyzing it and filtering it if needed.

The rootkit analyzes whether the IRP is sent to its fake device `ACPI#PNP0303#2&da1a3ff&0`, if so then it calls its own dispatch routine to handle the request. Being a fake hidden volume, it can handle all the needed IOCTL like `IOCTL_DISK_CHECK_VERIFY`, `IOCTL_DISK_GET_DRIVE_GEOMETRY`, `IOCTL_DISK_IS_WRITABLE`, `IOCTL_STORAGE_CHECK_VERIFY`, `IOCTL_STORAGE_GET_DEVICE_NUMBER`, `IOCTL_DISK_GET_DRIVE_LAYOUT_EX`, `IOCTL_DISK_GET_PARTITION_INFO_EX`. The hidden volume is encrypted and the rootkit read/write routine is able to encode and decode the data on the fly. The fake volume is stored inside a file located to `systemroot\system32\config\<random file name>`, where the random file name is the same name generated by the dropper and used to store the clean copy of the infected driver. This file is always encrypted on the hard drive. The encryption algorithm used by the rootkit is RC4 with a 128 bit key, which is the following: `0xFF,0x7C,0xF1,0x64,0x12,0xE2,0x2D,0x4D,0xB1,0xCF,0x0F,0x5D,0x6F,0xE5,0xA0,0x49`. The RC4 encryption/decryption is done sector by sector.

```

crypt_sector:
push offset RC4_key ; RC4 key
lea esi, [ebp+Sbox] ; RC4 S-Box base address
call Generate_RC4_Table ; Generate RC4 S-Box
push SectorSize
xor eax, eax
push edi
call CryptBuffer ; Encrypt/Decrypt sector
mov eax, SectorSize
add edi, eax
sub ebx, eax
jnz short crypt_sector ;

```

Rootkit driver I/O encryption/decryption

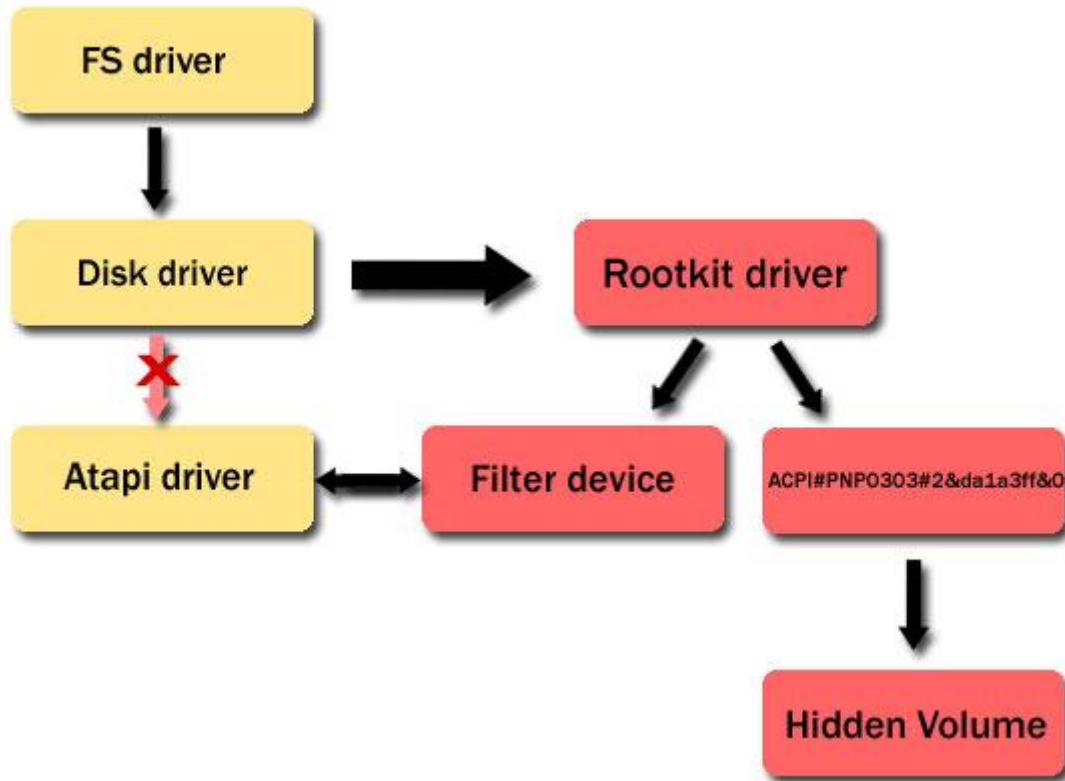


U		48 B	09/04/2011	21:38:25	09/04/2011
L		152 B	09/04/2011	21:38:25	09/04/2011
(Root directory)		4,1 KB	09/04/2011	21:38:25	09/04/2011
\$Extend		344 B	09/04/2011	21:38:25	09/04/2011
800000c0.sys	sys	44,5 KB	08/08/2021	13:57:08	09/04/2011
80000002.sys	sys	10,0 KB	29/05/2031	06:48:24	09/04/2011
80000001.sys	sys	21,5 KB	14/06/2031	10:52:17	09/04/2011
80000000.sys	sys	21,5 KB	22/06/2031	18:19:57	09/04/2011
000000c0.sym	sym	1,0 KB	08/08/2021	13:57:08	09/04/2011
00000011.sym	sym	38 B	08/08/2021	13:57:08	09/04/2011
00000002.sym	sym	6,8 KB	29/05/2031	06:48:24	09/04/2011
00000001.sym	sym	43,0 KB	29/05/2031	10:11:24	09/04/2011

Rootkit file system decrypted

If the IRP is not directed to the rootkit device, the dispatch routine analyzes the packet, looking for I/O requests to the infected driver file on the disk. The rootkit filters the `IRP_MJ_INTERNAL_DEVICE_CONTROL` major function, looking for SCSI request block structures. If the `SRB->Function` is `SRB_FUNCTION_EXECUTE_SCSI`, the filtering routine proceeds. The rootkit checks if a `FileObject` structure is filled in the incoming IRP request and, if so, calculates the hash of the file path located at the `FileObject->FileName`. The hash is calculated by calling the `RtlHashUnicodeString` and the result is checked against the hash of the infected driver's path calculated by the rootkit at the rootkit driver's startup. If the two hashes match, then the IRP request is faked by the rootkit.

If the SCSI_REQUEST_BLOCK packet operation is SCSIOP_READ, the read request is forwarded to the lower port device and the result is faked by the rootkit's CompletionRoutine; if the operation is SCSIOP_WRITE, the buffer is overwritten by the rootkit with the infected copy of the driver that was previously pointed to by a specific MDL.



Code flow after ZeroAccess infection

The rootkit queues a work item able to communicate with a list of C&C servers. It works at the TDI network layer, bypassing firewalls and security software that don't monitor network activities at this network level. The rootkit sends an encrypted request to all the servers in the list, the packet is always sent to the remote port TCP 13620. The rootkit allows the attacker to drop in the system further infections, by downloading and storing the relative files inside the hidden rootkit volume, so that they become invisible to security software. These dropped files are in the form of kernel mode driver. This is because the main rootkit driver is able to load them from the kernel by issuing a direct call to the IoCreateDriver() native API. These drivers will be invisible to most of security software which don't implement advanced anti rootkit features.

The rootkit presence in the system could be spotted by looking at suspicious system shutdown notification routines pointing to an unknown memory region. The rootkit sets up its own shutdown notification routine by calling the IoRegisterShutdownNotification() native API.

UPDATE (July, 2011)

ZeroAccess rootkit has been updated and several things changed since this first write-up.

The initial dropper, after executed, run a new instance of explorer.exe process and injects inside it the payload code. This behavior allows the dropper to evade from Windows 7's default UAC setting. Microsoft implemented a customizable version of the User Account Control in Windows 7. In its default configuration, the UAC automatically allows white listed system processes to acquire administrator rights without user interaction. This design allowed Microsoft to reduce the noise caused by the User Account Control, though opening a way for malicious code to reach the administrator rights in an easier way. Explorer.exe is one of the system white listed processes, so injecting malicious code inside the explorer process will allow the dropper to automatically get administrator rights without any user interaction.

Decrypting the dropper code will show the internal project name, which indeed confirms the target:

```
w E.getSystemTimeAsFileTime D.getTickCount E.ge
ltLangID ' ExitProcess U\Sleep " BindIoCompleti
■ DeleteTimerQueueTimer q CreateTimerQueueTimer
MD5Update ■ MD5Final ADUAPI32.dll B WSASocketW
SASend ; WSASendTo 6 WSARcvFrom WS2_32.dll ü_a
■ ■ e p:\vc5\release\_uac.pdb

cnqazwsxedcrfvtgeabyhnujmikoi jlp
B @ xEB @ ü*B
0MoC 106!1AW. è%B @ 33. .

(ü²€3Ü*³_è m s03ÉÈèd s.3Âè[ s#³_À°+è0 ↑Âs+u
Éè. *HÂâ□-è, = } s.Ëü | s-■ø■w_ AA■Â³ .U■÷+ðó*^è■_
Â+|$(■|$.aâ
```

ZeroAccess code injected inside explorer

The dropper now uses a domain generation algorithm to contact the command and control server. The DGA is able to generate a new domain each day. The domain generation routine is based on the current year, month and day and it works as follows:

- ✓ The dropper gets the current date by calling the API function `GetSystemTimeAsFileTime`;
- ✓ The dropper converts the resulted value to a `TIME_FIELDS` structure by calling the API function `RtlTimeToTimeFields`;
- ✓ The dropper calculates the CRC32 checksum value of the first 6 bytes of the `TIME_FIELDS` structure, which means calculating the CRC32 of the Year, Month, Day fields. The checksum is calculated by calling the API function `RtlComputeCrc32` and the initial CRC value is set to 0. We will call this checksum as `crc1`;
- ✓ The dropper calculates another CRC32 checksum like the step before, though this time initializing the CRC value to `0x33333333`. This is being called `crc2`;
- ✓ The dropper makes a shift logical left double (`shld` instruction) by 8, let's call the result `val2`;
- ✓ The dropper left shift the first `crc1` by 8;
- ✓ The dropper now `xor crc2 ^ crc1`, let's call the result value as `val1`;
- ✓ While `(val1 | val2)` is not zero, the dropper use the `(val1 & 0x1F)` result as index for the string: `"cnqazwsxedcrfvtgeabyhnujmikoi jlp"`, then it makes a full right shift by 5, calling `_allshr(val1, val2, 5)`. The routine loops until the "while" condition is met;
- ✓ The dropper appends to the string just composed the `.cn` top-level domain

The reversed algorithm is exposed in the following image:

```

call    ds:GetSystemTimeAsFileTime
lea    eax, [ebp+var_1C]
push   eax
lea    eax, [ebp+SystemTimeAsFileTime]
push   eax
call    ds:RtlTimeToTimeFields
mov    edi, ds:RtlComputeCrc32
push   6
lea    eax, [ebp+var_1C]
push   eax
push   0
call    edi ; RtlComputeCrc32
mov    esi, eax ; esi contains the first CRC
push   6
lea    eax, [ebp+var_1C]
push   eax
xor    ebx, ebx
shld  ebx, esi, 8 ; ebx contains 1st CRC shld 8
push   33333333h
shl   esi, 8 ; 1st CRC shl 8
call    edi ; RtlComputeCrc32
xor    edx, edx
xor    eax, esi ; 2nd CRC ^ 1st CRC (shl 8)
xor    edx, ebx ; edx contains the 1st CRC shld 8

build_domain:
mov    esi, [ebp+StringBuf] ; esi points to the destination string
mov    ecx, eax
and    ecx, 1Fh ; val1 & 0x1F (0x1F is the length of the seed string)
mov    c1, SeedString[ecx]
inc    [ebp+StringBuf]
mov    [esi], c1 ; put the indexed byte to the destination buffer
mov    c1, 5
call    _allshr
mov    ecx, eax
or     ecx, edx
jnz    short build_domain ; loop the build_domain routine
mov    eax, [ebp+StringBuf]
pop    edi
pop    esi
mov    dword ptr [eax], 'nc.' ; append the .cn top-level domain

```

Domain generation algorithm reversed

```

GetSystemTimeAsFileTime(&TimeFile);
RtlTimeToTimeFields(&TimeFile,&TimeDate);
crc1 = RtlComputeCrc32(0,&TimeDate,6);
crc2 = RtlComputeCrc32(0x33333333,&TimeDate,6);

_allshr(&val2,&crc1,8);
val1 = crc2 ^ crc1;

while ((val1 | val2))
{
    FileName[index] = StartingString[(val1 & 0x1F)];
    _allshr(&val1,&val2,5);
    index++;
}

strcat_s(FileName,MAX_PATH, ".cn");

```

DGA translated to C code

The command and control server checks the user agent, if matches the one showed below then the server will reply:
User-Agent: Opera/6 (Windows NT %u.%u; U; LangID=%x; x86) , where the %u and %x are the parameters retrieved from the infected operating system with a call to the GetVersion() and GetSystemDefaultLangID() Windows APIs.

The routine used by the rootkit to choose the right system driver to infect is not changed since the one described in this paper at page 3. What changed is the way how the rootkit generates the file name used to store the hidden RC4-encrypted rootkit volume. While the old routine is described at page 4, the new routine works as follows:

- ✓ The rootkit queries the following registry key: HKLM SYSTEM\Setup\Pid by calling ZwQueryKey with KeyBasicInformation parameter;
- ✓ The rootkit then queries the _KEY_BASIC_INFORMATION->LastWriteTime parameter;
- ✓ The rootkit hashes the LastWriteTime parameter, which is described as a LARGE_INTEGER value. It calls MD5Update() API by hashing the first 8 bytes of the _KEY_BASIC_INFORMATION structure filled by the ZwQueryKey call
- ✓ It retrieves the two first DWORD values of the four DWORD values composing the MD5 hash, these values will be used as seed values;
- ✓ It uses a starting string from where it gets the "random" characters that will compose the new file name. The string is: *eaoinnqazwsxedcrfvtgbyhnujmikolp;*
- ✓ The file name that needs to be composed is 8 characters long, so it starts a loop by doing following steps:
 - Seed value is and'd with 0x1F (length of the starting string), the returning value is the index of the character in the starting string that will be used in the new file name;
 - Seed value is right shifted by 5 using a 64 bit right shift functions (shrd/shr);

The loop continues until the eight-characters string is composed – starting from the end till the beginning of it.

```
RegOpenKeyA(HKEY_LOCAL_MACHINE, "SYSTEM\\Setup\\Pid", &regKey);
NtQueryKey(regKey, KeyBasicInformation, &KeyInfo, sizeof(KEY_BASIC_INFORMATION), &result);
CloseHandle(regKey);

Md5Init(&Md5Info);
Md5Update(&Md5Info, (unsigned char*)&KeyInfo, 8);
Md5Final(&Md5Info);

seed = *(DWORD*)((DWORD)&Md5Info + 0x58);
seed2 = *(DWORD*)((DWORD)&Md5Info + 0x5C);

while (index >= 0)
{
    FileName[index] = StartingString[(seed & 0x1F)];
    _allshr(&seed, &seed2, 5);

    if (index == 0)
        break;

    index--;
}
```

File name generation routine

The RC4 encryption implemented by the rootkit for its hidden volume is the same described at page 7, even the RC4 encryption key is still the same.

ROOTKIT SELF-DEFENSE TRICKS

The ZeroAccess rootkit now implements another kernel mode driver used to defend itself against security software along with its main driver analyzed in the previous pages. The driver sets up its own device object called `\Device\svchost.exe`, and it stores a fake PE executable to the fake path `\Device\svchost.exe\svchost.exe`. The rootkit self-defense driver attaches itself to the disk device stack so that it's able to handle IRP packets filtering the ones direct to its `\Device\svchost.exe` device object. The driver then sets up a fake system process, called `svchost.exe`, pointing its file path to the following one:

`\\.\globalroot\Device\svchost.exe\svchost.exe`. The rootkit then starts filtering every IRP packet directed to its fake device object.

The self-defense driver monitors every IRP packet, filtering the `IRP_MJ_CREATE` and `IRP_MJ_DIRECTORY_CONTROL` packets. If a software tries to open a handle to the fake `svchost.exe` or it executes an `IRP_MJ_DIRECTORY_CONTROL` query on the file – like a call to `ZwQueryDirectoryFile` – the rootkit driver kills the calling software.

If a software tries to open a handle to the fake file, the rootkit driver checks some PE settings before killing the calling process that attempted to open the handle. If the calling process's PE file has its `TimeDateStamp` set to `0x4CC3574B` and its `checksum` is set to `0xDEE3`, then the rootkit won't kill the process. Moreover, the rootkit won't kill the calling process if the PE fields `MajorOperatingSystemVersion` and `MinorOperatingSystemVersion` are equal to the current running operating system. This last check allows system tools like Windows Task Manager - `taskmgr.exe` - to work without being killed by the rootkit.

If the rootkit intercept a `IRP_MJ_DIRECTORY_CONTROL` request, it immediately kills the calling process without any further check.

```
lea    eax, [ebp+MinorVersion]
push   eax                ; MinorVersion
lea    eax, [ebp+MajorVersion]
push   eax                ; MajorVersion
call   ds:PsGetVersion
cmp    dword ptr [esi+8], 4CC35748h ; PE TimeDateStamp == 0x4CC3574B?
jnz    short Check_MajorOS ;

cmp    dword ptr [esi+58h], 0DEE3h ; PE checksum == 0xDEE3?
jz     short set_nokill_flag

Check_MajorOS:
; CODE XREF: Check_Process_Flags+761j
movzx  eax, word ptr [esi+40h] ;
; PE MajorOperatingSystemVersion
cmp    eax, [ebp+MajorVersion]
jnz    short clear_nokill_flag
movzx  eax, word ptr [esi+42h] ; PE MinorOperatingSystemVersion
cmp    eax, [ebp+MinorVersion]
jnz    short clear_nokill_flag

set_nokill_flag:
; CODE XREF: Check_Process_Flags+7F1j
xor    eax, eax
inc    eax
```

PE file check by the rootkit driver

The killing routine is composed by two main step: the process killing routine and the file killing routine. To kill the process that is trying to access to the fake file, the rootkit driver allocates inside the target process 165 bytes of memory and injects there its malicious payload. Then, the code is executed by scheduling an APC. When executed from inside the target process space, the malicious payload walks the module list, looking for `kernel32.dll`. Then, it parses the export table and looks for the `ExitProcess()` Win32 API. When found, the payload calls the exit process function. By doing so, the rootkit forces the target process from killing itself, bypassing every kind of security protection eventually implemented by the process.

```

Injected_Code:
mov     eax, large fs:18h ; Teb
mov     eax, [eax+30h] ; Teb->ProcessEnvironmentBlock
mov     eax, [eax+0Ch] ; Peb->Ldr
lea     ebp, [eax+0Ch] ; Peb->Ldr.InLoadOrderModuleList
mov     ebx, ebp

walk_module_list: ; CODE XREF: .text:00012D47lj
mov     ebx, [ebx]
cmp     ebx, ebp
jz     short loc_12D59
mov     esi, [ebx+30h]
xor     edi, edi
mov     eax, edi
mov     ecx, 1003Fh

hash_name: ; CODE XREF: .text:00012D3Dlj
or      ax, 20h
movzx   eax, ax
add     eax, edi
mul     ecx
mov     edi, eax
lodsw
test    ax, ax
jnz    short hash_name
cmp     edi, 86B11DEEh ; kernel32.dll hash
jz     short Parse_Export_Table ; module found
jmp     short walk_module_list

;
Parse_Export_Table: ; CODE XREF: .text:00012D45lj
mov     ebx, [ebx+18h] ; ExitProcess
push    0CEDFBD AFh
call    ImportFunction
push    eax
jmp     eax ; jmp to ExitProcess()

```

killer payload injected by the rootkit

The rootkit doesn't simply kill the process that tried to analyze the svchost.exe fake process, it will even prevent the software from being executed again. Thus, after the process has been killed, the rootkit resets the ACL setting of the software executable, preventing it from being executed again. The file's ACL is reverted to a customized ACL of the Everyone built-in Windows group.

```

push    edi ; OpenOptions
push    7 ; ShareAccess
lea     eax, [ebp+IoStatusBlock]
push    eax ; IoStatusBlock
lea     eax, [ebp+ObjectAttributes]
push    eax ; ObjectAttributes
push    40000h ; DesiredAccess
lea     eax, [ebp+Handle]
push    eax ; FileHandle
mov     [ebp+ObjectAttributes.Length], 18h
mov     [ebp+ObjectAttributes.ObjectName], ebx
mov     [ebp+ObjectAttributes.RootDirectory], edi
mov     [ebp+ObjectAttributes.Attributes], 40h
mov     [ebp+ObjectAttributes.SecurityDescriptor], edi
mov     [ebp+ObjectAttributes.SecurityQualityOfService], edi
call    ds:ZwOpenFile ; open handle to the target file
test    eax, eax
jl     short loc_1119B
push    offset Custom_SecurityDescriptor ; SecurityDescriptor
push    4 ; DACL_SECURITY_INFORMATION
push    [ebp+Handle] ; File handle
call    ds:ZwSetSecurityObject ; fix the DACL configuration
push    [ebp+Handle] ; Handle
call    ds:ZwClose

```

ZeroAccess kills the file ACL

```

C:\>cacls procexp.exe
C:\procexp.exe BUILTIN\Administrators:F
NT AUTHORITY\SYSTEM:F
TESTPC\PCtest:F
BUILTIN\Users:R

C:\>procexp

C:\>cacls procexp.exe
C:\procexp.exe Everyone:(NP)<accesso speciale:>
DELETE
READ_CONTROL
WRITE_DAC
WRITE_OWNER
STANDARD_RIGHTS_REQUIRED
FILE_READ_DATA
FILE_WRITE_DATA
FILE_APPEND_DATA
FILE_READ_EA
FILE_WRITE_EA
FILE_EXECUTE
FILE_DELETE_CHILD
FILE_READ_ATTRIBUTES
FILE_WRITE_ATTRIBUTES

```

ACL before and after the fix

UPDATE 2 (July, 2011)

ZeroAccess rootkit has been updated again this month, with another major update. Many things changed since the review we did in the previous pages, starting from the hidden volume container to the encryption algorithm change and self-defense improvement.

As documented above, the rootkit used to create a file under the %systemroot%\system32\config folder which worked as a hidden volume container. The file was mounted as a NTFS volume, which was encrypted and decrypted on the fly by using a plain RC4 encryption. The new release of ZeroAccess totally changes this approach: the rootkit doesn't use anymore its hidden NTFS volume and it doesn't use anymore the RC4 encryption – at least not as it is defined.

ZeroAccess now creates a folder under the %systemroot% directory (usually C:\WINDOWS), called \$NtUninstallKBxxxx\$ - where the Xs represent a unique number generated from characteristics of the infected system. The folder name has been chosen to be familiar with the usual folders created by Windows during the installation of the Windows updates.

The unique number is generated by querying the system volume's file system information, by calling the NtQueryVolumeInformationFile native API with the *FileFsVolumeInformation* information class. The CreationTime field is then hashed and the MD5 value is split into four DWORDs. The rootkit then xor the 1st DWORD with the 2nd one, then the result is xor'd with the 3rd one and again the same applies with the 4th one. The resulted DWORD value is then divided and some math operations are executed to derive the unique ID.

The assembly code that shows the math operations is listed here below:

```
mov     eax, [ebp+Format] ; eax contains the MD5 hash xor'd
movzx  ecx, word ptr [ebp+Format] ; ecx contains the low 16 bit of the MD5 hash
shr    eax, 10h          ; eax << 0x10 (eax contains higher 16 bit)
not    eax              ; ~eax
movzx  eax, ax          ; get low 16 bit
xor    eax, ecx         ; and xor together eax and ecx
push   eax              ; Format
lea    eax, [ebp+String]
push   offset aSystemrootNtun ; "\\systemroot\\$NtUninstallKB%u$"
push   eax              ; String
call   ds:swprintf
```

ZeroAccess code that generates folder name

Which can be roughly translated to the following C code:

```
Md5Init(&Md5Info);
Md5Update(&Md5Info, (unsigned char*)&VolumeInfo.VolumeCreationTime, 8);
Md5Final(&Md5Info);

FinalVal = *(DWORD*)((DWORD)&Md5Info.digest);
FinalVal ^= *(DWORD*)((DWORD)&Md5Info.digest + 4);
FinalVal ^= *(DWORD*)((DWORD)&Md5Info.digest + 8);
FinalVal ^= *(DWORD*)((DWORD)&Md5Info.digest + 12);

tmp = (WORD)FinalVal;
FinalVal >>= 16;
FinalVal = ~FinalVal;
FinalVal = (WORD)FinalVal;
FinalVal ^= tmp;
```

Code translated to C code

The folder will be used by the rootkit to store the payload modules as long as the clean copy of the infected driver – basically everything that was previously stored in the hidden volume. Giving the fact that the rootkit is using a more comfortable folder instead of a fully hidden volume could make us think that it's easier to get access to the rootkit components. In fact, it isn't trivial for a couple of reasons: the folder is configured by the rootkit as a reparse point, a symbolic link pointing to a fake path. This prevent every attempt to get access to the folder, redirecting every attempt to an invalid path.

```

push    edi                ; Size
mov     [esi+0Ch], ax      ; PrintNameOffset
lea    eax, [esi+14h]     ; PathBuffer
push   offset Str         ; "\\Device\\suchost.exe\\setup"
push   eax                ; Dst
mov     dword ptr [esi], 0A000000Ch ; IO_REPARSE_TAG_SYMLINK
mov     [esi+0Eh], bx     ; printName length
call   memcpy
movzx  eax, bx
push   eax                ; Size
lea    eax, [edi+esi+16h]
push   offset aCWindowsSystem ; "c:\\windows\\system32\\setup"
push   eax                ; Dst
call   memcpy
add    esp, 24h
xor    eax, eax
push   eax
push   eax
push   [ebp+Size]
lea    ecx, [ebp+var_10]
push   esi
push   900A4h             ; FSCTL_SET_REPARSE_POINT
push   ecx
push   eax
push   eax
push   eax
push   [ebp+var_4]
call   ds:2wFsControlFile

```

Even if the folder is accessed – e.g. through an application able to manually parse NTFS file system – all the files stored are encrypted, so that they can be read only through the rootkit's device object which is able to encrypt and decrypt them on the fly. The rootkit checks if the system's file system supports named streams and reparse points as well.

In the previous variants of ZeroAccess, the rootkit was using the RC4 encryption with a static RC4 key embedded inside the rootkit code. That's a weak approach, trivial to decode. This is the reason why the authors of the rootkit have totally changed the implementation of it. The rootkit now generates a unique encryption key based on specific information gathered by the infected system. This means that every file can be decrypted only on the specific system where it has been encrypted.

The encryption key is calculated in the same way how the folder unique ID number is generated. In fact, the MD5 hash of the Volume Creation Date is the system encryption key.

```

GetEnvironmentVariableA("systemroot", SystemPath, MAX_PATH);
hDir = CreateFileA(SystemPath, GENERIC_READ, FILE_SHARE_READ, NULL, OPEN_EXISTING, FILE_FLAG_BACKUP_SEMANTICS, NULL);

if (hDir == INVALID_HANDLE_VALUE)
{
    printf("\n[!] Error getting handle! (%d)", GetLastError());
    return 0;
}

NtQueryVolumeInformationFile(hDir, &IoStatus, (PVOID)&AttribInfo, sizeof(FILE_FS_ATTRIBUTE_INFORMATION), FileFsAttributeInformation);

if (AttribInfo.FileSystemAttributes & (FILE_NAMED_STREAMS | FILE_SUPPORTS_REPARSE_POINTS))
{
    NtQueryVolumeInformationFile(hDir, &IoStatus, (PVOID)&VolumeInfo, sizeof(FILE_FS_VOLUME_INFORMATION), FileFsVolumeInformation);

    Md5Init(&Md5Info);
    Md5Update(&Md5Info, (unsigned char*)&VolumeInfo.VolumeCreationTime, 8);
    Md5Final(&Md5Info);

    memcpy_s((PVOID)Key, 16, (PVOID)((DWORD)&Md5Info.digest), 16);
}

```

C code to generate RC4 key

The encryption algorithm used by the last version of ZeroAccess is a slightly modified variant of the RC4 encryption, which still uses the S-Box generated by the RC4 algorithm though it doesn't use the classic encryption algorithm. Given the system-generated 128 bit key, the rootkit generates the RC4 S-Box and then it generates another S-Box, which is actually the first S-Box inverted. Then, the file encryption/decryption process is based on bytes permutation – first S-Box is used to encrypt the file, second one is used to decrypt.

```

index = 255;

while (index >= 0)
{
    IdByte = StreamTable[index];
    ShadowTable[IdByte] = (BYTE)index;

    if (index == 0)
        break;

    index--;
}

```

C code - Inverted Table Gen

```

PVOID EncryptDecryptBuf(PVOID buf, DWORD size, PVOID KeyStream)
{
    BYTE* BufBin = (BYTE*)buf;
    BYTE* KeyStr = (BYTE*)KeyStream;
    BYTE * NewBuf;
    DWORD index;
    BYTE tmpByte;

    NewBuf = (BYTE*)malloc(size);

    for (index = 0; index < size; index++)
    {
        tmpByte = BufBin[index];
        NewBuf[index] = KeyStr[tmpByte];
    }

    return (PVOID)NewBuf;
}

```

C code Encrypt/Decrypt buffer

```

[*] System RC4 key is: 4F B4 F4 87 30 91 6C 6D 35 4E 5A 7C 3F A9 B0 4B

[*] RC4 S-Box is:
67 05 BC 12 20 52 63 13 6E C5 E5 B0 6D 8E 10 C9
53 ED B4 27 D1 77 E2 3E CA 9F 1C 3C C2 44 2B BE
92 D9 EF 99 AE 41 35 0F 48 98 87 5A FC 8A B3 39
08 55 3A 49 8B EB BD AA 7D 68 0B E9 43 66 1A A1
E1 A3 5B 6C 71 47 9D AB 6F 89 42 6B CD 59 60 D2
A6 9E 82 4C DD 90 85 A9 23 15 9B 3B 83 B8 54 FA
EC 81 97 36 B9 7F 86 30 7E 02 F2 F5 F6 FB 33 2F
B7 D4 88 50 3F DE 32 D7 BA F4 7C BF 0A 72 51 4B
E3 19 65 75 F1 18 C1 01 34 14 78 73 FF BB 2D 00
C3 96 B5 A4 E4 C0 9A A5 2E 5E FD 03 4F 16 7A 07
C7 DB 56 7B 17 38 AD F7 24 D3 8C AF 76 DF E6 E0
93 A8 79 25 F3 EA 4A DC 4E 6A 5C 61 5F 4D 95 E7
22 D0 62 F8 40 EE 31 2C 5D 8D CF CC 58 B6 2A 29
C8 57 1F C4 0E 26 09 94 28 F9 1D D8 C6 D6 D5 F0
AC 9C 21 A7 FE B2 11 3D CB DA 04 64 0D 06 46 70
1B 0C 1E 84 A2 CE 91 74 69 37 A0 E8 45 8F 80 B1

[*] Inverse RC4 S-Box is:
8F 87 69 9B EA 01 ED 9F 30 D6 7C 3A F1 EC D4 27
0E E6 03 07 89 59 9D A4 85 81 3E F0 1A DA F2 D2
04 E2 C0 58 A8 B3 D5 13 D8 CF CE 1E C7 8E 98 6F
67 C6 76 6E 88 26 63 F9 A5 2F 32 5B 1B E7 17 74
C4 25 4A 3C 1D FC EE 45 28 33 B6 7F 53 BD B8 9C
73 7E 05 10 5E 31 A2 D1 CC 4D 2B 42 BA C8 99 BC
4E BB C2 06 EB 82 3D 00 39 F8 B9 4B 43 0C 08 48
EF 44 7D 8B F7 83 AC 15 8A B2 9E A3 7A 38 68 65
FE 61 52 5C F3 56 66 2A 72 49 2D 34 AA C9 0D FD
55 F6 20 B0 D7 BE 91 62 29 23 96 5A E1 46 51 19
FA 3F F4 41 93 97 50 E3 B1 57 37 47 E0 A6 24 AB
0B FF E5 2E 12 92 CD 70 5D 64 78 8D 02 36 1F 7B
95 86 1C 90 D3 09 DC A0 D0 0F 18 E8 CB 4C F5 CA
C1 14 4F A9 71 DE DD 77 DB 21 E9 A1 B7 54 75 AD
AF 40 16 80 94 0A AE BF FB 3B B5 35 60 11 C5 22
DF 84 6A B4 79 6B 6C A7 C3 D9 5F 6D 2C 9A E4 8C

[*] Crypted File: [piswhjin]
[*] Decoded file: [piswhjin.dec]

```

The rootkit kernel driver is now packed with a runtime packer to obfuscate the rootkit code, which is a very questionable choice for a couple of reasons: it doesn't help hiding the rootkit code at all and adds another layer of potential incompatibility /bugs that should be avoided in kernel mode to prevent any kind of stability issue and system crashes.

The self-defense layer we have analyzed in our previous white paper update has been further improved and it has been embedded in the main rootkit driver. Not only the rootkit sets up a fake process used as a trap, but the self-defense routine now monitors every attempts to open a handle to the rootkit device `ACPI#PNP0303#2&da1a3ff&0`, which is the main door to communicate with the rootkit and access to the rootkit's encrypted folder. If a software tries to open a handle to the device without specifying the full path to a specific file (e.g. `ACPI#PNP0303#2&da1a3ff&0\L\xxxxxxx`), the self-defense routine interprets it as an attempts to analyze the device and immediately kills the calling application.

The self-defense routine itself has been improved: the killing routine consists of two parts, the process killing part and the file killing part. The process killing routine allocates some memory inside the target process through a call to the `ZwAllocateVirtualMemory` native API, then it copies there the malicious payload and schedules its execution with a call to the `KeInsertQueueApc` native API. This approach had a weak point: the call to the `ZwAllocateVirtualMemory` passes through the SSDT, thus hooking the pointer on the SSDT would allow a security application to prevent the rootkit from injecting the malicious code. The last update of ZeroAccess fixed this weak point by directly calling the `ZwAllocateVirtualMemory` API by knowing the memory address, bypassing the System Service Descriptor Table.

CONCLUSIONS

ZeroAccess is definitely one of the most advanced kernel mode rootkits out there. While it isn't as powerful as TDL rootkit family yet, it implements a number of unique features that make it quite dangerous and a potential vector of other infections. The way how it creates and handles the hidden volume allows ZeroAccess to be distributed along with any other kind of infection, storing it in the rootkit's encrypted file system and giving it full access to the system.

As already written in the paper, ZeroAccess strongly resembles TDL3 rootkit in many ways: they both implemented the same idea of storing their code outside the system's filesystem, both use RC4 encryption, both choose randomly the driver to be infected, both filter SCSI_REQUEST_BLOCK packets at lower level than disk.sys (though TDL3 hijacks the lowest miniport driver while ZeroAccess hits disk.sys's DR0 device by hijacking it and redirecting it to its filtering device). The disk filtering engine implemented by ZeroAccess is not as advanced as the one implemented by TDL3 rootkit, this is the reason why ZeroAccess infection is easier to be detected and removed than the TDL3 rootkit. Sadly this is a minor problem that could be easily improved by the ZeroAccess authors, making its creature more complete and powerful than ever, moreover if it'll be combined with other kind of infections.

If ZeroAccess will evolve in the same way how TDL3 quickly evolved, we'll probably see a bigger significant number of computers worldwide hit by this infection.

ABOUT PREVX

Prevx provides cloud-based products with unparalleled capabilities for protecting consumers, SMEs and enterprises, banks, and government organizations from the latest malware threats.

The entire Prevx suite is underpinned by its award-winning flagship security agent, Prevx 3.0, and connects to the world's largest cloud-based threat database. Prevx 3.0 is the world's smallest, fastest, and lightest endpoint security agent yet its detection, protection and removal capabilities rival the largest antivirus solutions. Prevx specializes in detecting zero day attacks, reducing the time exposed to danger and providing real-time protection against the latest and the most malicious forms of malware, including keyloggers, Trojans, and rootkits - catching the threats that are missed by traditional antivirus providers.

Prevx is a division of Internet security service company Webroot. With its main operations in the United Kingdom, Prevx products are also sold and supported across Europe and in the United States. Before acquisition by Webroot in 2010, Prevx was formed by IT entrepreneur Mel Morris who acquired Immunify Ltd in 2005 and re-launched it as Prevx. Now vice president and general manager of the Prevx division at Webroot, Morris named Prevx to reflect the organization's mission to help customers - from consumers and small businesses to the largest financial institutes and global organizations - to best protect themselves against the evolving and unknown nature of malicious software. Prevx: preventing the unknown.

Prevx's family of security software is deployed by leading banks, enterprises, and government agencies and supports over 15 million users worldwide.