ZeroAccess – an advanced kernel mode rootkit

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 classpnp.sys driver and win32k.sys, every other driver is discarded;
- ✓ All the drivers between classpnp.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+390↓p
                     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:                             ; CODE XREF: Format_Virtual_Drive+20↑j
                     push    esi             ; hLibModule
                     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: *eaoimnqazwsxedcrfvtgbyhnujmikolp;*
- ✓ 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\Snifer67, where Snifer67 is replaced with the just generated name.

```
setup_seed:                             ; CODE XREF: generate_name_clean_driver+28↑j
                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+77↓j
                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 "eaoimnqazwsxedcrfvtgbyhnujmikolp" 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:

```c
char* StartingString = "eaoimnqazwsxedcrfvtgbyhnujmikolp";
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  +fffffffff86d28ab
   +0x030 DriverStartIo    : (null)
   +0x034 DriverUnload     : (null)
   +0x038 MajorFunction    : [28] 0xf4b79134     long  +fffffffff4b79134
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  +fffffffff86d28ab
   +0x030 DriverStartIo    : (null)
   +0x034 DriverUnload     : 0xf86e253a     void  +fffffffff86e253a
   +0x038 MajorFunction    : [28] 0xf86e1c30     long  +fffffffff86e1c30
```

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.

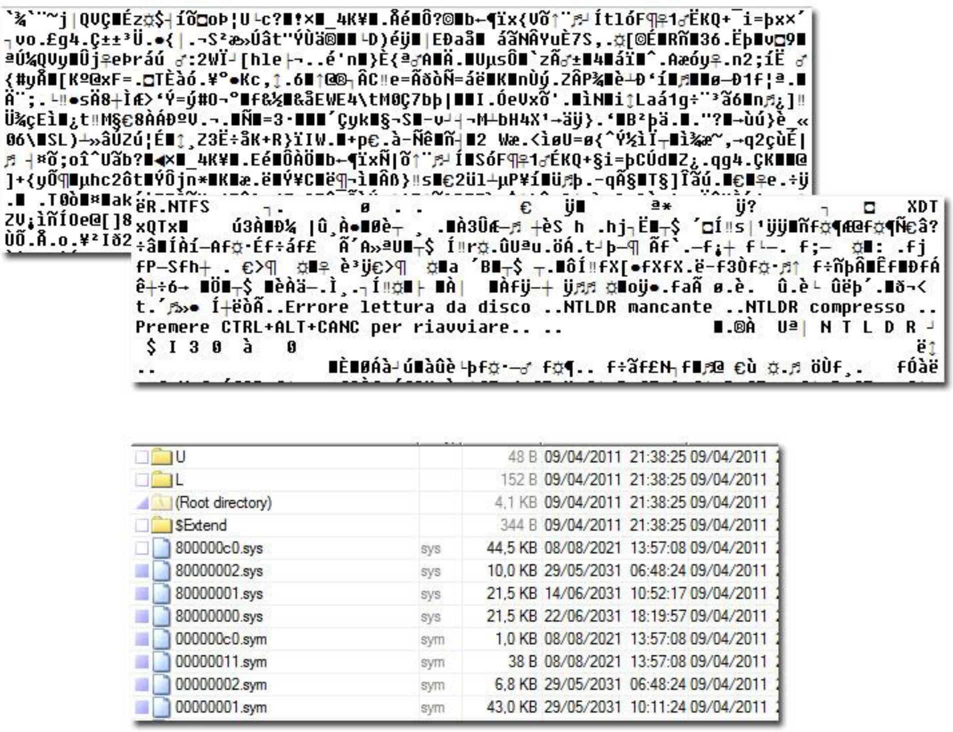
The rootkit driver object sets up two different device objects, the first one is the device object used to intercept the disk.sys's I/O while the second one is the one we talked about at the beginning of the current paragraph.

To intercept disk.sys's I/O routine, the rootkit hijacks the \driver\disk's DR0 device object by alterating 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   ;
```

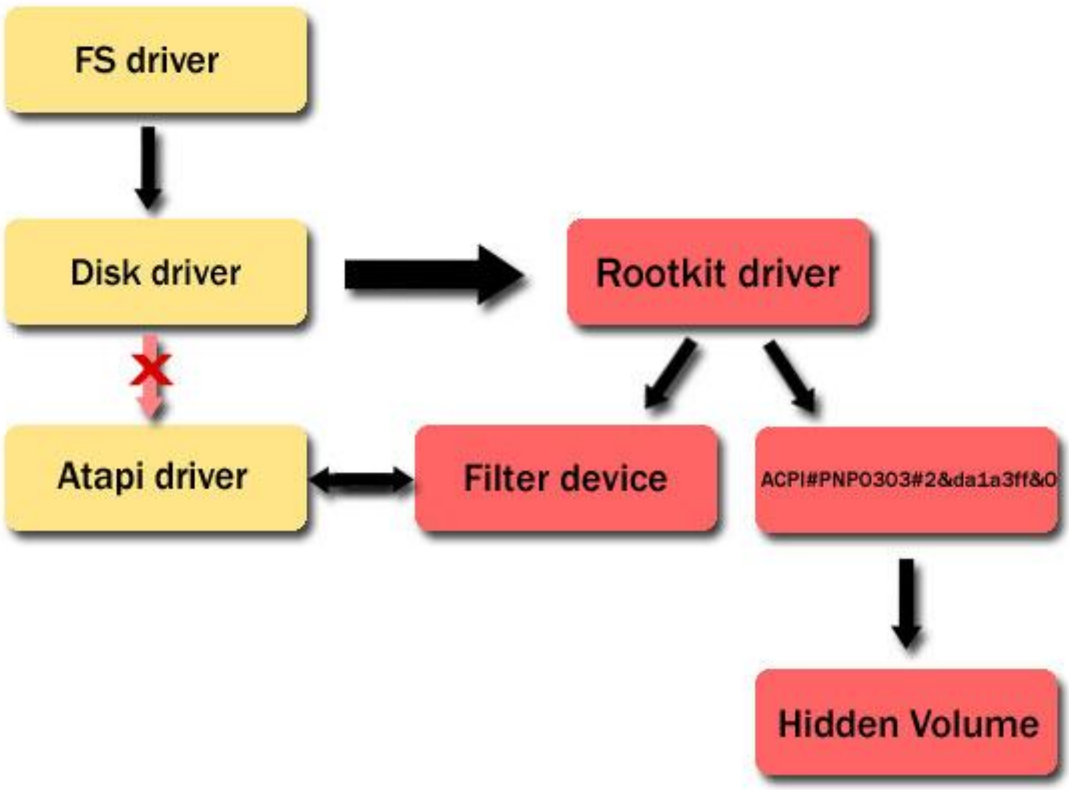
Rootkit driver I/O encryption/decryption



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.

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.