2

FESTI ROOTKIT: THE MOST ADVANCED SPAM AND DDOS BOT

This chapter is devoted to one of the most advanced spam and distributed denial of service (DDoS) botnets discovered—the Win32/Festi botnet, which we'll refer to simply as Festi from now on. Festi has powerful spam delivery and DDoS capabilities, as well as interesting rootkit functionality that allows it to stay under the radar by hooking into the filesystem and system registry. Festi also conceals its presence by actively counteracting dynamic analysis with debugger and sandbox evasion techniques.

From a high-level point of view, Festi has a well-designed modular architecture implemented entirely in the kernel-mode driver. Kernel-mode programming is, of course, fraught with danger: a single error in the code can cause the system to crash and render it unusable, potentially leading

the user to reinstall the system afresh, wiping the malware. For this reason, it's rare for spam-sending malware to rely heavily on kernel-mode programming. The fact that Festi was able to inflict so much damage is indicative of the solid technical skills of its developer(s) and their in-depth understanding of the Windows system. Indeed, they came up with several interesting architectural decisions, which we'll cover in this chapter.

The Case of Festi Botnet

The Festi botnet was first discovered in the fall of 2009, and by May 2012, it was one of the most powerful and active botnets for sending spam and performing DDoS attacks. The botnet was initially available to anyone for lease, but after early 2010, it was restricted to major spam partners, like Pavel Vrublebsky, one of the actors who used the Festi botnet for criminal activities as detailed in the book *Spam Nation* by Brian Krebs (Sourcebooks, 2014).

According to statistics from M86 Security Labs (currently Trustwave) for 2011, shown in Figure 2-1, Festi was one of the three most active spam botnets in the world in the reported period.

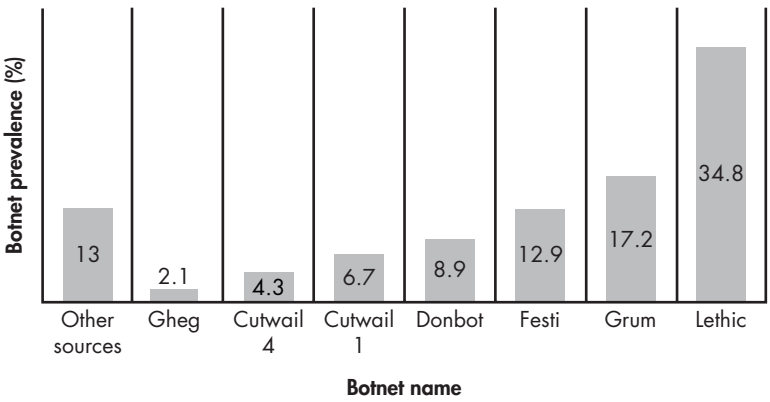

Figure 2-1: The most prevalent spam botnets according to M86 Security Labs

Festi's rise in popularity stemmed from a particular attack on Assist, a payment-processing company.[1] Assist was one of the companies bidding for a contract with Aeroflot, Russia's largest airline, but a few weeks before Aeroflot was due to make its decision, cybercriminals used Festi to launch a massive DDoS attack against Assist. The attack rendered the processing system unusable for an extended period of time, eventually forcing Aeroflot to award another company the contract. This event is a prime example of how rootkits may be used in real-world crime.

1. Brian Krebs, "Financial Mogul Linked to DDoS Attacks," *Krebs on Security* blog, June 23, 2011, *http://krebsonsecurity.com/2011/06/financial-mogul-linked-to-ddos-attacks/*.

Dissecting the Rootkit Driver

The Festi rootkit is distributed mainly through a PPI scheme similar to the TDL3 rootkit discussed in Chapter 1. The dropper's rather simple functionality installs into the system a kernel-mode driver that implements the main logic of the malware. The kernel-mode component is registered as a "system start" kernel-mode driver with a randomly generated name, meaning the malicious driver is loaded and executed at system bootup during initialization.

> **DROPPER INFECTOR**
>
> A *dropper* is a special type of infector. Droppers carry a payload to the victim system within itself. The payload is frequently compressed and encrypted or obfuscated. Once executed, a dropper extracts the payload from its image and installs it on a victim system (that is, drops it on the system—thus the name for this type of infector). Unlike droppers, *downloaders*—another type of infector—don't carry payloads within themselves but rather download it from a remote server.

The Festi botnet targets only the Microsoft Windows x86 platform and does not have a kernel-mode driver for 64-bit platforms. This was fine at the time of its distribution, as there were still many 32-bit operating systems in use, but now means the rootkit has largely been rendered obsolete as 64-bit systems have outnumbered 32-bit systems.

The kernel-mode driver has two main duties: requesting configuration information from the command and control (C&C) server and downloading and executing malicious modules in the form of plug-ins (illustrated in Figure 2-2). Each plug-in is dedicated to a certain job, such as performing DDoS attacks against a specified network resource or sending spam to an email list provided by the C&C server.

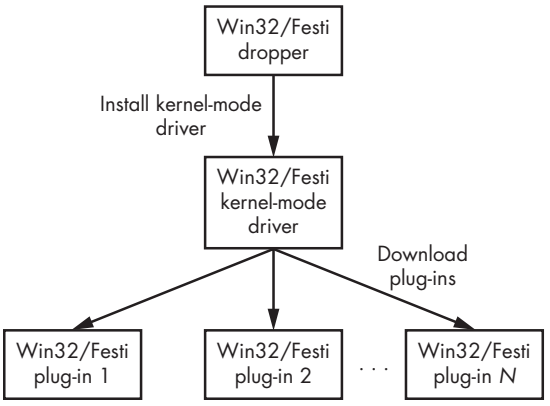

Figure 2-2: Operation of the Festi rootkit

Interestingly, the plug-ins aren't stored on the system hard drive but instead in volatile memory, meaning that when the infected computer is powered off or rebooted, the plug-ins vanish from system memory. This makes forensic analysis of the malware significantly harder since the only file stored on the hard drive is the main kernel-mode driver, which contains neither the payload nor any information on attack targets.

Festi Configuration Information for C&C Communication

To enable it to communicate with C&C server, Festi is distributed with three pieces of predefined configuration information: the domain names of C&C servers, the key to encrypt data transmitted between the bot and C&C, and the bot version information

This configuration information is hardcoded into the driver's binary. Figure 2-3 shows a section table of the kernel-mode driver with a writable section named `.cdata`, which stores the configuration data as well as strings that are used to perform the malicious activity.

Name	Virtual Size	Virtual Address	Raw Size	Raw Address	Reloc Address	Linenumbers	Relocations N...	Linenumber...	Characteristics
Byte[8]	Dword	Dword	Dword	Dword	Dword	Dword	Word	Word	Dword
.text	00003B27	00001000	00003C00	00000400	00000000	00000000	0000	0000	68000020
.rdata	000007C8	00005000	00000800	00004000	00000000	00000000	0000	0000	48000040
.data	00001098	00006000	00001000	00004800	00000000	00000000	0000	0000	C8000040
pagecode	0000A84C	00008000	0000AA00	00005800	00000000	00000000	0000	0000	C8000040
.cdata	00000582	00013000	00000600	00010200	00000000	00000000	0000	0000	C8000040
INIT	000008D8	00014000	00000A00	00010800	00000000	00000000	0000	0000	E2000020
.reloc	00000992	00015000	00000A00	00011200	00000000	00000000	0000	0000	42000040

Figure 2-3: Section table of Festi kernel-mode driver

The malware obfuscates the contents with a simple algorithm that XORs the data with a 4-byte key. The `.cdata` section in decrypted at the very beginning of the driver initialization.

The strings within the `.cdata` section, listed in Table 2-1, can garner the attention of security software, so obfuscating them helps the bot evade detection.

Table 2-1: Encrypted Strings in the Festi Configuration Data Section

String	Purpose
`\Device\Tcp` `\Device\Udp`	Names of device objects used by the malware to send and receive data over the network
`\REGISTRY\MACHINE\SYSTEM\CurrentControlSet\Services\SharedAccess\Parameters\FirewallPolicy\StandardProfile\GloballyOpenPorts\List`	Path to the registry key with the parameters of the Windows firewall, used by the malware to disable the local firewall
`ZwDeleteFile, ZwQueryInformationFile, ZwLoadDriver, KdDebuggerEnabled, ZwDeleteValueKey, ZwLoadDriver`	Names of system services used by the malware

Festi's Object-Oriented Framework

Unlike many kernel-mode drivers, which are usually written in plain C using the procedural programming paradigm, the Festi driver has an object-oriented architecture. The main components (classes) of the architecture implemented by the malware are:

Memory manager Allocates and releases memory buffers

Network sockets Send and receive data over the network

C&C protocol parser Parses C&C messages and executes received commands

Plug-in manager Manages downloaded plug-ins

The relationships among these components are illustrated in Figure 2-4.

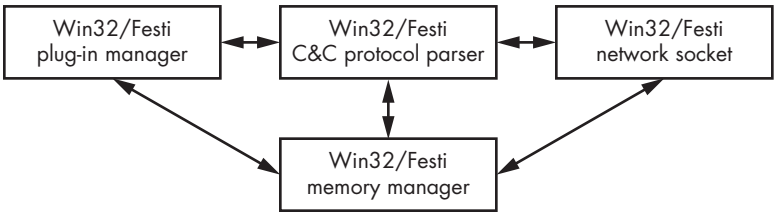

Figure 2-4: Architecture of the Festi kernel-mode driver

As you can see, the memory manager is the central component used by all other components.

This object-oriented approach allows the malware to be easily ported to other platforms, like Linux. To do so, an attacker would need to change only system-specific code (like the code that calls system services for memory management and network communication) that is isolated by the component's interface. Downloaded plug-ins, for instance, rely almost completely on the interfaces provided by the main module; they rarely use routines provided by the system to do system-specific operations.

Plug-in Management

Plug-ins downloaded from the C&C server are loaded and executed by the malware. To manage the downloaded plug-ins efficiently, Festi maintains an array of pointers to a specially defined `PLUGIN_INTERFACE` structure. Each structure corresponds to a particular plug-in in memory and provides the bot with specific entry points—routines responsible for handling data received from C&C, as shown in Figure 2-5. This way, Festi keeps track of all the malicious plug-ins loaded in memory.

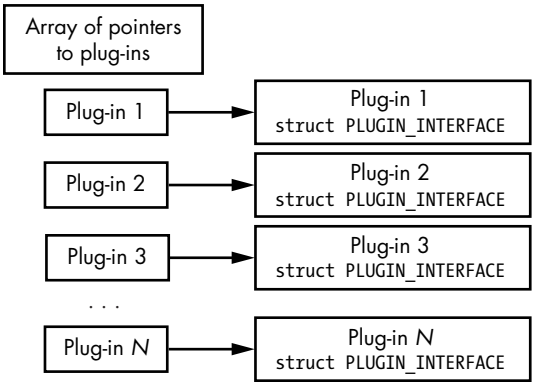

Figure 2-5: Layout of the array of pointers to PLUGIN_INTERFACE structures

Listing 2-1 shows the layout of the PLUGIN_INTERFACE structure.

```
struct PLUGIN_INTERFACE
{
  // Initialize plug-in
  PVOID Initialize;
  // Release plug-in, perform cleanup operations
  PVOID Release;
  // Get plug-in version information
  PVOID GetVersionInfo_1;
  // Get plug-in version information
  PVOID GetVersionInfo_2;
  // Write plug-in-specific information into tcp stream
  PVOID WriteIntoTcpStream;
  // Read plug-in specific information from tcp stream and parse data
  PVOID ReadFromTcpStream;
  // Reserved fields
  PVOID Reserved_1;
  PVOID Reserved_2;
};
```

Listing 2-1: Defining the PLUGIN_INTERFACE structure

The first two routines, Initialize and Release, are intended for plug-in initialization and termination, respectively. The following two routines, GetVersionInfo_1 and GetVersionInfo_2, are used to obtain version information for the plug-in in question.

The routines WriteIntoTcpStream and ReadFromTcpStream are used to exchange data between the plug-in and the C&C server. When Festi transmits data to the C&C server, it runs through the array of pointers to the plug-in interfaces and executes the WriteIntoTcpStream routine of each registered plug-in, passing a pointer to a TCP stream object as a parameter. The TCP stream object implements the functionality of the network communication interface.

On receiving data from the C&C server, the bot executes the plug-ins' `ReadFromTcpStream` routine, so that the registered plug-ins can get parameters and plug-in-specific configuration information from the network stream. As a result, every loaded plug-in can communicate with the C&C server independently of all other plug-ins, which means plug-ins can be developed independently of one another, increasing the efficiency of their development and the stability of the architecture.

Built-in Plug-ins

Upon installation, the main malicious kernel-mode driver implements two built-in plug-ins: the *configuration information manager* and the *bot plug-in manager*.

Configuration Information Manager

The configuration information manager plug-in is responsible for requesting configuration information and downloading plug-ins from the C&C server. This simple plug-in periodically connects to the C&C server to download the data. The delay between two consecutive requests is specified by the C&C server itself, likely to avoid static patterns that security software can use to detect infections. We describe the network communication protocol between the bot and the C&C server in "The Festi Network Communication Protocol" on page 26.

Bot Plug-in Manager

The bot plug-in manager is responsible for maintaining the array of downloaded plug-ins. It receives remote commands from the C&C server and loads and unloads specific plug-ins, delivered in compressed form, onto the system. Each plug-in has a default entry point—`DriverEntry`—and exports the two routines `CreateModule` and `DeleteModule`, as shown in Figure 2-6.

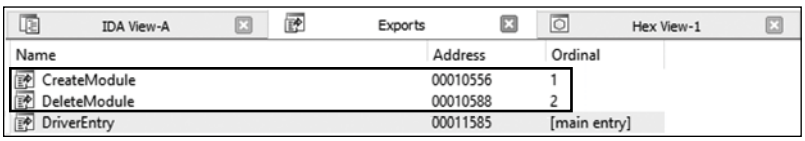

Figure 2-6: Export Address table of a Festi plug-in

The `CreateModule` routine is executed upon plug-in initialization and returns a pointer to the `PLUGIN_INTERFACE` structure, as described back in Listing 2-1. It takes as a parameter a pointer to several interfaces provided by the main module, such as the memory manager and network interface.

The `DeleteModule` routine is executed when the plug-in is unloaded and frees all the previously allocated resources. Figure 2-7 shows the plug-in manager's algorithm for loading the plug-in.

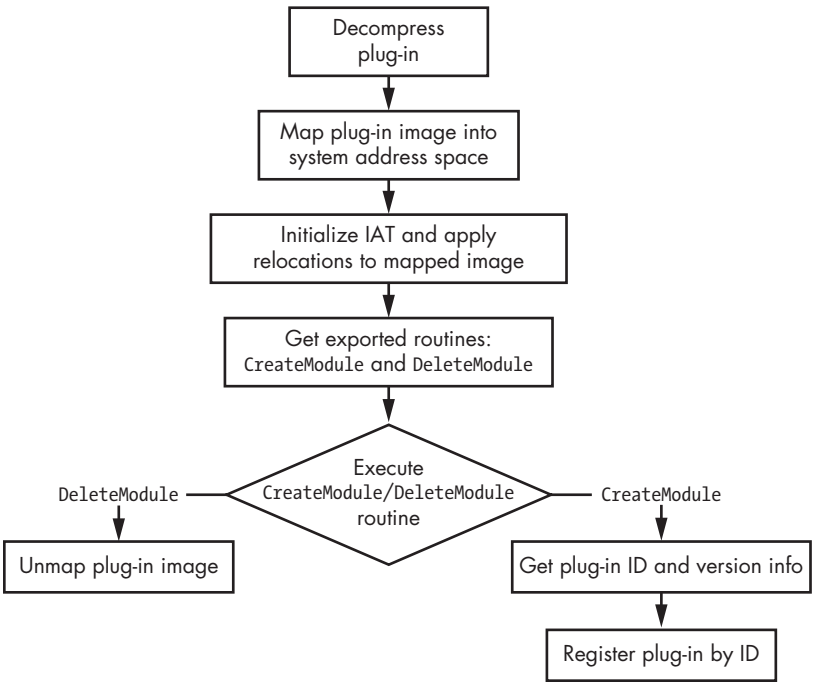

Figure 2-7: Plug-in manager algorithm

The malware first decompresses the plug-in into the memory buffer and then maps it into the kernel-mode address space as a PE image. The plug-in manager initializes the Import Address table (IAT) and relocates it to the mapped image. In this algorithm, Festi also emulates a typical operating system's runtime loader and dynamic linker of OS modules.

Depending on whether the plug-in is being loaded or unloaded, the plug-in manager executes either the `CreateModule` or `DeleteModule` routine. If the plug-in is being loaded, the plug-in manager obtains the plug-in's ID and version information, then registers it to the `PLUGIN_INTERFACE` structures.

If the plug-in is being unloaded, the malware releases all memory previously allocated to the plug-in image.

Anti–Virtual Machine Techniques

Festi has techniques for detecting whether it is running inside a VMware virtual machine in order to evade sandboxes and automated malware analysis environments. It attempts to obtain the version of any existent VMWare software by executing the code shown in Listing 2-2.

```
mov eax, 'VMXh'
mov ebx, 0
mov ecx, 0Ah
mov edx, 'VX'
in eax, dx
```

Listing 2-2: Obtaining the VMWare software version

Festi checks the `ebx` register, which will contain the value `VMX` if the code is being executed in a VMware virtual environment and `0` if not.

Interestingly, if Festi detects the presence of a virtual environment, it doesn't immediately terminate execution but proceeds as if it were being executed on the physical computer. When the malware requests plug-ins from the C&C server, it submits certain information that reveals whether it's being executed in the virtual environment; if it is, the C&C server may not return any plug-ins.

This is likely a technique for evading dynamic analysis: Festi doesn't terminate communication with the C&C server in an effort to trick the automatic analysis system into thinking Festi hasn't noticed it, while in fact the C&C server is aware of being monitored and so won't provide any commands or plug-ins. It's common for malware to terminate execution once it detects that it's running under a debugger or in a sandbox environment in order to avoid revealing the configuration information and payload modules.

However, malware researchers are savvy to this behavior: if the malware promptly terminates without performing any malicious activity, it can draw the attention of an analyst, who will likely then perform a deeper analysis to find out why it didn't work, eventually discovering the data and code the malware is trying to conceal. By not terminating its execution when a sandbox is detected, Festi attempts to avoid these consequences, but it does instruct its C&C to not provide the sandbox with malicious modules and configuration data.

Festi also checks for the presence of network traffic monitoring software on the system, which may indicate that the malware has been executed in a malware analysis and monitoring environment. Festi looks for the kernel-mode driver *npf.sys* (network packet filter). This driver belongs to the Windows packet capture library, WinPcap, which is frequently used by network monitoring software like Wireshark to gain access to the data link network layer. The presence of the *npf.sys* driver indicates that there are network monitoring tools installed on the system, meaning it is unsafe for the malware.

WINPCAP

The *Windows packet capture library* (WinPcap) allows applications to capture and transmit network packets, bypassing the protocol stack. It provides functionality for kernel-level network packet filtering and monitoring. This library is used extensively as a filtering engine by many open source and commercial network tools, like protocol analyzers, network monitors, network intrusion detection systems, and sniffers, including widely known tools such as Wireshark, Nmap, Snort, and ntop.

Antidebugging Techniques

Festi also checks for the presence of a kernel debugger in the system by examining the `KdDebuggerEnabled` variable exported from the operating system kernel image. If a system debugger is attached to the operating system, this variable contains the value `TRUE`; otherwise, it contains `FALSE`.

Festi actively counteracts the system debugger by periodically zeroing the debugging registers `dr0` through `dr3`. These registers are used to store addresses for breakpoints, and removing the hardware breakpoints hinders the debugging process. The code for clearing the debugging registers is shown in Listing 2-3.

```
char _thiscall ProtoHandler_1(STRUCT_4_4 *this, PKEVENT a1)
{
__writedr(0, 0); // mov dr0, 0
__writedr(1u, 0); // mov dr1, 0
__writedr(2u, 0); // mov dr2, 0
__writedr(3ut 0); // mov dr3, 0
  return _ProtoHandler(&this->struct43, a1);
}
```

Listing 2-3: Clearing debugging registers in Festi code

The highlighted `writedr` instructions perform write operations on the debugging registers. As you can see, Festi writes zeros to these registers before executing the `_ProtoHandler` routine, which is responsible for handling the communication protocol between the malware and C&C servers.

The Method for Hiding the Malicious Driver on Disk

To protect and conceal the image of the malicious kernel-mode driver stored on the hard drive, Festi hooks the filesystem driver so that it can intercept and modify all requests sent to the filesystem driver to exclude evidence of its presence.

A simplified version of the routine for installing the hook is shown in Listing 2-4.

```
NTSTATUS __stdcall SetHookOnSystemRoot(PDRIVER_OBJECT DriverObject,
                                       int **HookParams)
{
  RtlInitUnicodeString(&DestinationString, L"\\SystemRoot");
  ObjectAttributes.Length = 24;
  ObjectAttributes.RootDirectory = 0;
  ObjectAttributes.Attributes = 64;
  ObjectAttributes.ObjectName = &DestinationString;
  ObjectAttributes.SecurityDescriptor = 0;
  ObjectAttributes.SecurityQualityOfService = 0;

❶ NTSTATUS Status = IoCreateFile(&hSystemRoot, 0x80000000, &ObjectAttributes,
                                 &IoStatusBlock, 0, 0, 3u, 1u, 1u, 0, 0, 0, 0,
                                 0x100u);
  if (Status < 0 )
    return Status;

❷ Status = ObReferenceObjectByHandle(hSystemRoot, 1u, 0, 0,
                                     &SystemRootFileObject, 0);
  if (Status < 0 )
    return Status;

❸ PDEVICE_OBJECT TargetDevice = IoGetRelatedDeviceObject(SystemRootFileObject);
  if ( !_ TargetDevice )
      return STATUS_UNSUCCESSFUL;

  ObfReferenceObject(TargetDevice);
  Status = IoCreateDevice(DriverObject, 0xCu, 0, TargetDev->DeviceType,
                          TargetDevice->Characteristics, 0, &SourceDevice);
  if (Status < 0 )
    return Status;

❹ PDEVICE_OBJECT DeviceAttachedTo = IoAttachDeviceToDeviceStack(SourceDevice,
                                                                TargetDevice);
  if ( ! DeviceAttachedTo )
  {
    IoDeleteDevice(SourceDevice);
    return STATUS_UNSUCCESSFUL;
  }

  return STATUS_SUCCESS;
}
```

Listing 2-4: Hooking the filesystem device driver stack

The malware first tries to obtain a handle to the special system file `SystemRoot`, which corresponds to the Windows installation directory ❶. Then, by executing the `ObReferenceObjectByHandle` system routine ❷, Festi obtains a pointer to the `FILE_OBJECT` that corresponds to the handle for `SystemRoot`. The `FILE_OBJECT` is a special data structure used by the operating system to manage access to device objects and so contains a pointer

to the related device object. In our case, since we opened a handle for `SystemRoot`, the `DEVICE_OBJECT` is related to the operating system filesystem driver. The malware obtains the pointer to the `DEVICE_OBJECT` by executing the `IoGetRelatedDeviceObject` system routine ❸, then creates a new device object and attaches it to the acquired device object pointer by calling `IoAttachDeviceToDeviceStack` ❹, as shown in the layout of the filesystem device stack in Figure 2-8. Festi's malicious device object is located on top of the stack, meaning the I/O requests intended for the filesystem are rerouted to the malware. This allows Festi to conceal itself by altering request and return data to and from the filesystem driver.

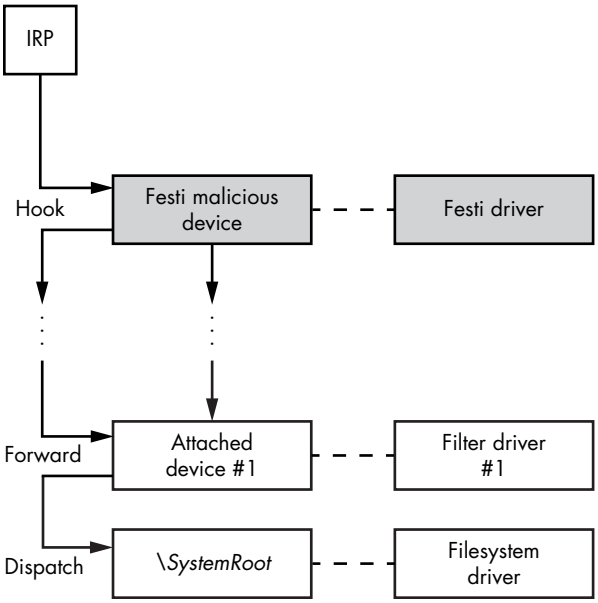

Figure 2-8: Layout of the filesystem device stack hooked by Festi

At the very bottom of Figure 2-8, you can see the filesystem driver object and the corresponding device object that handles OS filesystem requests. Some additional filesystem filters might be attached here too. Toward the top of the figure, you can see the Festi driver attached to the filesystem device stack.

This design uses and closely follows the Windows stacked I/O driver design, reproducing the design pattern of the native OS. By now, you probably see the trend: the rootkit aims to blend with the OS cleanly and reliably, emulating winning OS design patterns for its own modules. In fact, you can learn a lot about OS internals from analyzing aspects of rootkits, such as Festi's handling of input/output requests.

In Windows, a filesystem I/O request is represented as an IRP, which goes through the stack from top to bottom. Every driver in the stack can observe and modify the request or returned data. This means that, as shown in Figure 2-8, Festi can modify IRP requests addressed to the filesystem driver and any corresponding returned data.

Festi monitors the IRPs using the `IRP_MJ_DIRECTORY_CONTROL` request code, used to query the contents of the directory, watching for queries related to where the malware's kernel-mode driver is located. If it detects such a request, Festi modifies the returned data from the filesystem driver to exclude any entry corresponding to the malicious driver file.

The Method for Protecting the Festi Registry Key

Festi also hides a registry key corresponding to the registered kernel-mode driver using a similar method. Located in *HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services*, the registry key contains Festi's driver type and the path to the driver's image on the filesystem. This makes it vulnerable to detection by security software, so Festi must hide the key.

To do so, Festi first hooks the `ZwEnumerateKey`, a system service that queries information on a specified registry key and returns all of its subkeys, by modifying the *System Service Descriptor Table (SSDT)*, a special data structure in the operating system kernel that contains addresses of the system service handlers. Festi replaces the address of the original `ZwEnumerateKey` handler with the address of the hook.

WINDOWS KERNEL PATCH PROTECTION

It's worth mentioning that this hooking approach—modifying SSDT—works only on 32-bit Microsoft Windows operating systems. As mentioned in Chapter 1, the 64-bit editions of Windows implement *Kernel Patch Protection* (also known as PatchGuard) technology to prevent software from patching certain system structures, including SSDT. If PatchGuard detects a modification of any of the monitored data structures, it crashes the system.

The `ZwEnumerateKey` hook monitors requests addressed to the *HKLM\System\CurrentControlSet\Service* registry key, which contains subkeys related to kernel-mode drivers installed on the system, including the Festi driver. Festi modifies the list of subkeys in the hook to exclude the entry corresponding to its driver. Any software that relies on `ZwEnumerateKey` to obtain the list of installed kernel-mode drivers will not notice the presence of Festi's malicious driver.

If the registry is discovered by security software and removed during shutdown, Festi is also capable of replacing the registry key. In this case, Festi first executes the system routine `IoRegisterShutdownNotification` in order to receive shutdown notifications when the system is turned off. It checks the shutdown notification handler to see if the malicious driver and the corresponding registry key are present in the system, and if they're not (that is, if they've been removed), it restores them, guaranteeing that it will persist through reboot.

The Festi Network Communication Protocol

To communicate with C&C servers and perform its malicious activities, Festi employs a custom network communication protocol that it must protect against eavesdropping. In the course of our investigation of the Festi botnet,[2] we obtained a list of C&C servers it communicates with and found that while some focused on sending spam and others performed DDoS attacks, both types implemented a single communication protocol. The Festi communication protocol consists of two phases: the initialization phase, when it obtains C&C IP addresses, and the work phase, when it requests a job description from C&C.

Initialization Phase

During the initialization phase, the malware obtains the IP addresses of the C&C server, whose domain names are stored in the bot's binary. What's interesting about this process is that the malware manually resolves the C&C IP address from the C&C server domain names. Specifically, it constructs a DNS request packet to resolve the C&C server domain name and sends the packet to one of two hosts, 8.8.8.8 or 8.8.4.4 at port 53, both of which are Google DNS servers. In reply, Festi receives an IP address it can use in subsequent communication.

Manually resolving domain names makes the botnet more resilient to takedown attempts. If Festi had to rely on a local ISP's DNS servers for resolving domain names, it would be possible for the ISP to block access to the C&C servers by modifying DNS information on them—say, if a law enforcement agency issued a warrant to block those domain names. By manually crafting DNS requests and sending them to Google servers, however, the malware bypasses an ISP's DNS infrastructure and makes a takedown more difficult.

Work Phase

The work phase is when Festi requests information from the C&C server on what tasks it is to perform. Communication with the C&C servers is performed over the TCP protocol. The layout of the network packet request sent to the C&C server, shown in Figure 2-9, consists of a message header and an array of plug-in-specific data.

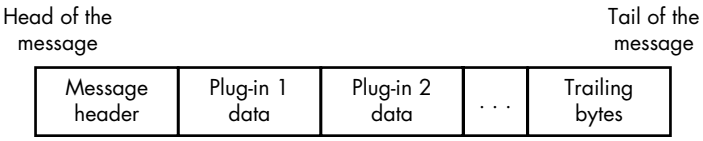

Figure 2-9: Layout of the network packet sent to the C&C server

2. Eugene Rodionov and Aleksandr Matrosov, "King of Spam: Festi Botnet Analysis," May 2012, *http://www.welivesecurity.com/wp-content/media_files/king-of-spam-festi-botnet-analysis.pdf.*

The message header is generated by the configuration manager plug-in and contains the following information:

- Festi version information
- Whether a system debugger is present
- Whether virtualization software (VMWare) is present
- Whether network traffic monitoring software (WinPcap) is present
- Operating system version information

The plug-in-specific data consists of an array of *tag-value-term* entries:

Tag A 16-bit integer specifying a type of value that follows the tag

Value Specific data in the form of a byte, word, dword, null-terminated string, or binary array

Term The terminating word, 0xABDC, signifying the end of the entry

The tag-value-term scheme provides a convenient way for malware to serialize plug-in-specific data into a network request to the C&C server.

The data is obfuscated with a simple encryption algorithm before being sent over the network. The Python implementation of the encryption algorithm is shown in Listing 2-5.

```
key = (0x17, 0xFB, 0x71,0x5C) ❶
def decr_data(data):
  for ix in xrange(len(data)):
    data[ix] ^= key[ix % 4]
```

Listing 2-5: Python implementation of the network encryption algorithm

The malware uses a rolling XOR algorithm with a fixed 4-byte key ❶.

Bypassing Security and Forensics Software

In order to communicate over the network with C&C servers, send spam, and perform DDoS attacks while eluding security software, Festi relies on a TCP/IP stack implemented in kernel mode in Windows.

To send and receive packets, the malware opens a handle to the *\Device\Tcp* or *\Device\Udp* devices depending on the protocol type being used, employing a rather interesting technique to acquire the handle without drawing the attention of security software. In designing this technique, Festi's authors again demonstrated a superb understanding of Windows system internals.

In order to control access to the network on the host, some security software monitors access to these devices by intercepting `IRP_MJ_CREATE` requests, which are sent to the transport driver when someone tries to open a handle to communicate with the device object. This allows the security software

to determine which process is trying to communicate over the network. Generally speaking, the most common ways for security software to monitor access to the device objects are:

- Hooking the `ZwCreateFile` system service handler to intercept all attempts to open the devices
- Attaching to *\Device\Tcp* or *\Device\Udp* in order to intercept all IRP requests sent

Festi cleverly bypasses both techniques to establish a connection with a remote host over the network.

First, instead of using the system implementation of the `ZwCreateFile` system service, Festi implements its own system service with almost the same functionality as the original one. Figure 2-10 shows the custom implementation of the `ZwCreateFile` routine.

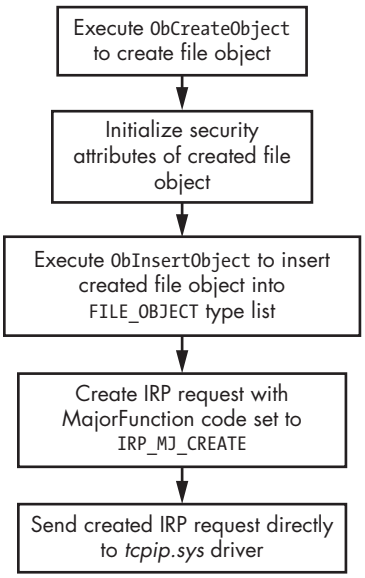

Figure 2-10: Custom implementation of `ZwCreateFile` routine

You can see that Festi manually creates a file object to communicate with the device being opened and sends an `IRP_MJ_CREATE` request directly to the transport driver. Thus, all the devices attached to *\Device\Tcp* or *\Device\Udp* will miss the request, and the operation goes unnoticed by security software, as illustrated in Figure 2-11.

On the left side of the figure, you can see how an IRP is normally processed. The IRP packet goes through the complete driver stack, and all the drivers hooked within it—including the security software—receive the IRP packet and inspect its contents. The right side of the figure shows how Festi instead sends the IRP packet directly to the target driver, bypassing all the intermediate ones.

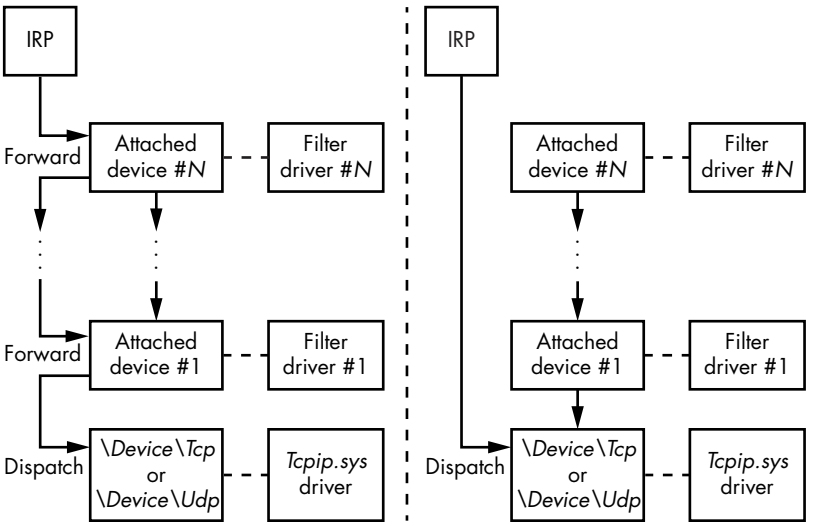

Figure 2-11: Bypassing network monitoring security software

Festi sidesteps the second security software technique just as deftly. To send a request directly to *\Device\Tcp* or *\Device\Udp*, the malware requires pointers to the corresponding device objects. The fragment of code responsible for this maneuver is presented in Listing 2-6.

```
RtlInitUnicodeString(&DriverName, L"\\Driver\\Tcpip");
RtlInitUnicodeString(&tcp_name, L"\\Device\\Tcp");
RtlInitUnicodeString(&udp_name, L"\\Device\\Udp");
❶ if (!ObReferenceObjectByName(&DriverName,64,0,0x1F01FF,
                               IoDriverObjectType,0,0,&TcpipDriver))
{
  DevObj = TcpipDriver->DeviceObject;
❷ while ( DevObj )                              // iterate through DEVICE_OBJECT
  {                                             // linked list
    if ( !ObQueryNameString(DevObj, &Objname, 256, &v8) )
    {
   ❸ if ( RtlCompareUnicodeString(&tcp_name, &Objname, 1u) )
      {
     ❹ if ( !RtlCompareUnicodeString(&udp_name, &Objname, 1u) )
        {
          ObfReferenceObject(DevObj);
          this->DeviceUdp = DevObj;         // Save pointer to \Device\Udp
        }
      } else
      {
        ObfReferenceObject(DevObj);
        this->DeviceTcp = DevObj;           // Save pointer to \Device\Tcp
      }
    }
    DevObj = DevObj->NextDevice;        // get pointer to next DEVICE_OBJECT
                                        // in the list
  }
```

```
  ObfDereferenceObject(TcpipDriver);
}
```

Listing 2-6: Implementing the network monitoring security software bypassing technique

Festi obtains a pointer to the *tcpip.sys* driver object by executing the `ObReferenceObjectByName` routine ❶, an undocumented system routine, and passing as a parameter a pointer to a Unicode string with the target driver's name. Then the malware iterates through the list of device objects ❷ corresponding to the driver object and compares its names with *\Device\Tcp* ❸ and *\Device\Udp* ❹.

When the malware obtains a handle for the opened device in this way, it uses the handle to send and receive data over the network. Though Festi is able to avoid security software, it's possible to see packets it sends by using network traffic filters operating at a lower level (for instance, at the Network Driver Interface Specification, or NDIS, level) than Festi.

The Domain Generation Algorithm for C&C Failure

Another of Festi's remarkable features is its implementation of a domain name generation algorithm (DGA), used as a fallback mechanism when the C&C servers' domain names in the bot's configuration data are unreachable. This can happen, for instance, if a law enforcement agency takes down the domain names of Festi C&C servers and the malware is unable to download plug-ins and commands. The algorithm takes the current date as input and outputs a domain name.

Table 2-2 lists the DGA-based domain names for a Festi sample. As you can see, all the generated domain names are pseudorandom, which is a characteristic of DGA-generated domain names.

Table 2-2: List of DGA Domain Names Generated by Festi

Date	DGA domain name
07/11/2012	*fzcbihskf.com*
08/11/2012	*pzcaihszf.com*
09/11/2012	*dzcxifsff.com*
10/11/2012	*azcgnfsmf.com*
11/11/2012	*bzcfnfsif.com*

Implementing DGA functionality makes the botnet resilient to takedown attempts. Even if law enforcement managed to disable the primary C&C server domains, the botnet master could still regain control of the botnet by falling back on DGA.

Malicious Functionality

Now that we've covered the rootkit functionality, let's look at the malicious plug-ins downloaded from the C&C servers. In the course of our investigation, we obtained a sample of these plug-ins and have identified three types:

- *BotSpam.sys* for sending spam emails
- *BotDos.sys* for performing DDoS attacks
- *BotSocks.sys* to provide proxy services

We found that different C&C servers tend to provide different types of plug-ins: some C&C servers provide only bots with spam plug-ins while others deal only in DDoS plug-ins, indicating that the malicious functionality of the malware depends on the C&C servers it reports to. The Festi botnet is not a monolith but rather comprises subbotnets dedicated to different targets.

The Spam Module

The *BotSpam.sys* plug-in is responsible for sending junk emails. The C&C server sends it a spam template and a list of recipient email addresses. Figure 2-12 illustrates the workflow for the spam plug-ins.

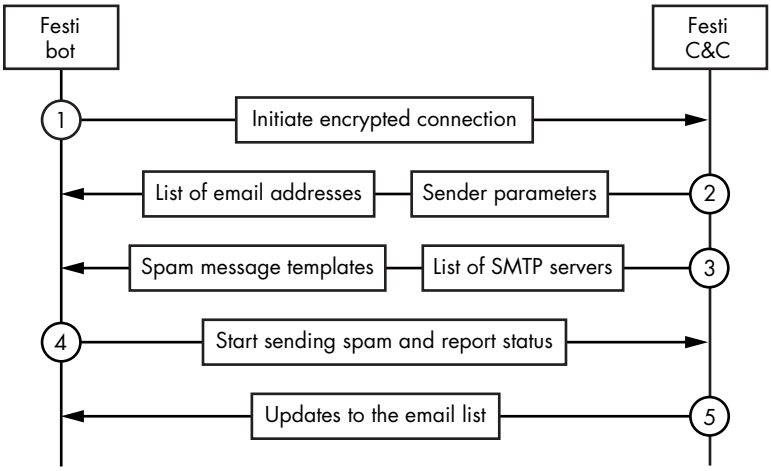

Figure 2-12: Workflow diagram of Festi spam plug-in

First, the plug-in initiates an encrypted connection with its C&C server to download a list of email addresses with sender parameters and the actual spam templates. It then distributes the spam letters to the recipients. Meanwhile, the malware reports the status to the C&C server and requests updates for the email list and spam templates.

The plug-in then checks the status of sent emails by scanning responses from an SMTP server for specific strings that signify problems—for instance, if there is no recipient with the specified address, an email wasn't received, or an email was classified as junk. If any of these strings is found in the

responses from the SMTP server, the plug-in gracefully terminates its session with the SMTP server and fetches the next address in the list. This precautionary step helps the malware to avoid an SMTP server blacklisting the infected machine's IP address as a spam sender and preventing the malware from sending any more spam.

The DDoS Engine

The *BotDos.sys* plug-in allows the bot to perform DDoS attacks against specified hosts. The plug-in supports several types of DDoS attacks against remote hosts, covering a variety of architectures and hosts with different software installed. The types of attacks depend on the configuration data received from the C&C and include TCP flood, UDP flood, DNS flood, and HTTP flood attacks.

TCP Flood

In the case of TCP flooding, the bot initiates a large number of connections to a port on the target machine. Every time Festi connects to a target port on a server, the server allocates resources to handle the incoming connection. Soon the server runs out of resources and stops responding to clients.

The default port is the HTTP port, port 80, but this can be changed with corresponding configuration information from the C&C server, allowing the malware to attack HTTP servers that listen on ports other than 80.

UDP Flood

In a UDP flood, the bot sends UDP packets of randomly generated lengths, filled with random data. The length of a packet can be anywhere from 256 to 1,024 bytes. The target port is also randomly generated and is therefore unlikely to be open. As a result, the attack causes the target host to generate an enormous number of ICMP Destination Unreachable packets in reply, and the target machine becomes unavailable.

DNS Flood

The bot is also able to perform DNS flood attacks by sending high volumes of UDP packets to port 53 (DNS service) on the target host. The packets contain requests to resolve a randomly generated domain name in the *.com* domain zone.

HTTP Flood

In HTTP flood attacks against web servers, the bot's binary contains many different user-agent strings, which are used to create a large number of HTTP sessions with the web server, overloading the remote host. Listing 2-7 contains the code for assembling the HTTP request that's sent.

```
int __thiscall BuildHttpHeader(_BYTE *this, int a2)
{
❶ user_agent_idx = get_rnd() % 0x64u;
  str_cpy(http_header, "GET ");
  str_cat(http_header, &v4[204 * *(_DWORD *)(v2 + 4) + 2796]);
  str_cat(http_header, " HTTP/1.0\r\n");
  if ( v4[2724] & 2 )
  {
    str_cat(http_header, "Accept: */*\r\n");
    str_cat(http_header, "Accept-Language: en-US\r\n");
    str_cat(http_header, "User-Agent: ");
  ❷ str_cat(http_header, user_agent_strings[user_agent_idx]);
    str_cat(http_header, "\r\n");
  }
  str_cat(http_header, "Host: ");
  str_cat(http_header, &v4[204 * *(_DWORD *)(v2 + 4) + 2732]);
  str_cat(http_header, "\r\n");
  if ( v4[2724] & 2 )
    str_cat(http_header, "Connection: Keep-Alive\r\n");
  str_cat(http_header, "\r\n");
  result = str_len(http_header);
  *(_DWORD *)(v2 + 16) = result;
  return result;
}
```

Listing 2-7: Fragment of Festi DDoS plug-in assembling an HTTP request

At ❶ the code generates a value that's then used at ❷ as an index in the array of user-agent strings.

Festi Proxy Plug-in

The *BotSocks.sys* plug-in provides remote proxy service to the attacker by implementing the SOCKS server over the TCP and UDP protocols. The SOCKS server establishes a network connection to another target server on behalf of a client, then routes all the traffic back and forth between the client and the target server.

As a result a Festi-infected machine becomes a proxy server that allows attackers to connect to remote servers through the infected machine. Cybercriminals may use such a service for anonymization—that is, to conceal the attacker's IP address. Since the connection happens via the infected host, the remote server can see the victim's IP address but not that of the attacker.

Festi's *BotSocks.sys* plug-in doesn't use any reverse-connect proxy mechanisms to bypass NAT (Network Address Translation), which enables multiple computers in the network to share a single externally visible IP address. Once the malware has loaded the plug-in, it opens a network port and starts listening for incoming connections. The port number is chosen at random in a range from 4000 to 65536. The plug-in sends the port number it's listening on to the C&C server so that an attacker could establish a network

connection with the victim computer. The NAT would normally prevent such incoming connections (unless port forwarding is configured for the target port).

The *BotSocks.sys* plug-in also attempts to bypass the Windows firewall, which may otherwise prevent the port from being opened. The plug-in modifies the registry key *SYSTEM\CurrentControlSet\Services\SharedAccess\Parameters\FirewallPolicy\DomainProfile\GloballyOpenPorts\List*, which contains a list of ports that may be opened in the Windows firewall profile. The malware adds two subkeys in this registry key to enable incoming TCP and UDP connections from any destination accordingly.

SOCKS

Socket Secure (SOCKS) is an internet protocol that exchanges network packets between a client and server through a proxy server. A SOCKS server proxies TCP connections from a SOCKS client to an arbitrary IP address and provides a means for UDP packets to be forwarded. The SOCKS protocol is often used by cybercriminals as a circumvention tool that allows traffic to bypass internet filtering to access content that's otherwise blocked.

Conclusion

You should now have a complete picture of what the Festi rootkit is and what it can do. Festi is an interesting piece of malware with well-designed architecture and carefully crafted functionality. Every technical aspect of the malware accords with its design principles: be stealthy and be resilient to automated analysis, monitoring systems, and forensic analysis.

The volatile malicious plug-ins downloaded from C&C servers don't leave any trace on the hard drive of the infected machine. Using encryption to protect the network communication protocol that connects it with C&C servers makes it hard to detect Festi in the network traffic, and advanced usage of kernel-mode network sockets allows Festi to bypass certain Host Intrusion Prevention Systems (HIPS) and personal firewalls.

The bot eludes security software by implementing rootkit functionality that hides its main module and the corresponding registry key in the system. These methods were effective against security software at the height of Festi's popularity, but they also constitute one of its major flaws: it targets 32-bit systems only. The 64-bit editions of the Windows operating systems implement modern security features, such as PatchGuard, that render Festi's intrusive arsenal ineffective. The 64-bit versions also require kernel-mode drivers to have a valid digital signature, which is obviously not an easy option for malicious software. As mentioned in Chapter 1, the solution malware developers came up with to circumvent this limitation was to implement bootkit technology, which we'll cover in detail in Part 2.