# Derusbi (Server Variant) Preliminary Analysis

## Overview

There are two types of Derusbi malware: a client-server model and a server-client model. Both types provide basic RAT functionality with the distinction between the two being largely the directionality of the communication. This report will focus on the server-client variant (or simply, the "server variant") of Derusbi which acts as a server on a victim's machine and waits for commands from a controlling client.

In and of itself, the Derusbi server variant is a largely unremarkable RAT when viewed from the perspective of functional capabilities. The server variant supports basic RAT functionality such as file management (uploading and download), network tunneling and remote command shell. What makes the server variant interesting is the device driver that the variant installs.

The server variant utilizes a device driver in order to hook into the Windows firewall by either using largely undocumented Windows Firewall hooking techniques found in Windows XP and older or by using the documented Windows Filtering Platform found in Windows Vista and later. The driver, after hooking the firewall using either of the two mentioned interfaces, will inspect incoming network packets. If a specific handshake occurs between the client and the server variant, the remainder of the communication session for the established session will be redirected to the server variant. If the driver does not detect the appropriate handshake, then the network traffic is allowed to pass unobstructed. This allows an attacker to hide their communication within a cluster of network sessions originating from a single IP such as would be the case for a client performing multiple HTTP requests against a web server.

## Functionality

The server variant appears to have a modular design allowing an attacker to compile in only the components that are necessary for any given operation. The malware supports up to 8 different modules per sample with each module designating its own ID code. Novetta observed the following modules:

| ID | Module Description |
|---|---|
| 0x81 | Remote command shell |
| 0x82 | Network tunneling |
| 0x84 | File management |
| 0xF0 | Derusbi administrative [built-in module that does not count against the maximum of 8 modules per variant sample] |

Given the spacing in ID numbers (as noted in the gap between 0x82 and 0x84 in an otherwise sequential ID scheme), it is conceivable that additional modules exist.

## Windows Device Driver (Firewall Hook)

The communication between the controlling client and the Derusbi server variant depends on the device driver being in place. If the device driver fails to load, the server variant will shut down silently.

The authors of the device driver designed the driver to work on Windows 2000 and greater versions of the Windows operating system. Depending on the version of the victim's OS, the driver will hook the Windows Firewall by either using the surprisingly undocumented IOCTL_IP_SET_FIREWALL_HOOK command of the \\Device\IP device for Windows XP or older machines or by using the documented Windows Filtering Platform (WFP) found in Windows Vista and later. The device driver inspects incoming network traffic from any client connecting to the victim machine, determines if an appropriate handshake packet occurs at the beginning of a new TCP session and then makes the decision to reroute the network traffic to the Derusbi malware or let the traffic continue unaltered to its original service.

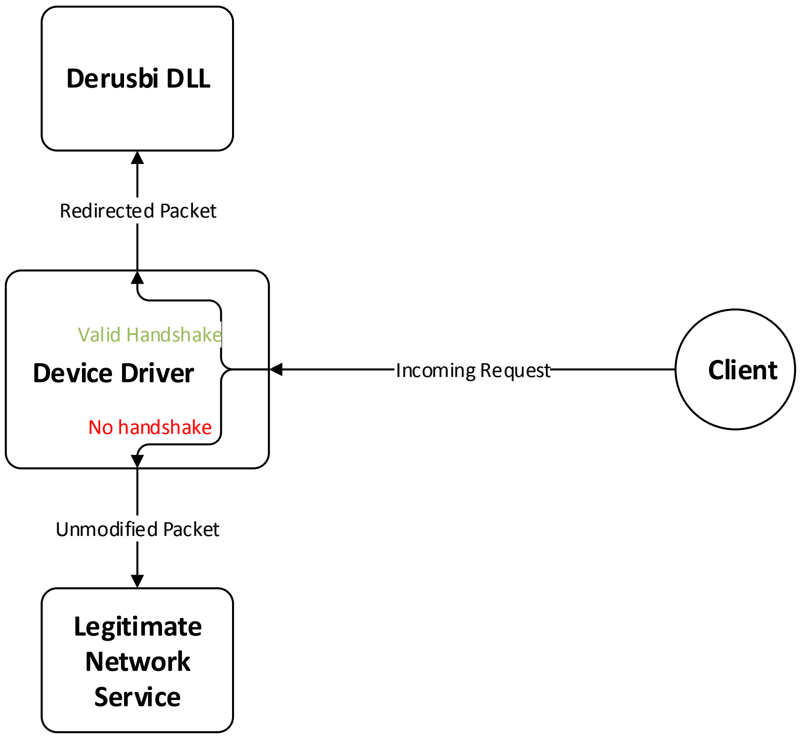

Figure 1: Device Driver Traffic Redirection

Once a session has been established by means of a valid handshake, any subsequent packets from the client for the given TCP session will automatically be directed by the device driver to the Derusbi server variant. The device driver does not capture or store any network traffic outside of the initial handshake inspection.

The server variant DLL can access the driver via \\Device\{93144EB0-8E3E-4591-B307-8EEBFE7DB28F}. The driver exists on the victim's machine as %SYSTEMDIR%\drivers\{93144EB0-8E3E-4591-B307-8EEBFE7DB28F}.sys.

## Communication

The Derusbi server variant will select an available, random port between the range of 5000 and 40000 on the victim's machine to listen on. After selecting the port, the server variant will wait for incoming connections and instruct the driver to redirect appropriate TCP sessions to the listening port.

The handshake between a client and the server variant is well defined. Consisting of 64 bytes, the data within the handshake is entirely random with the exception of the 3rd and 8th bytes. The handshake begins when the client sends a 64 byte random buffer with the 3rd (offset 12) and 8th (offset 32) DWORDs defined as:

```
DWORD3 == ~DWORD0
DWORD8 == ROR(DWORD0, 7)
```

The server will acknowledge the handshake by sending a 64 byte random buffer with the same pattern for the 3rd and 8th DWORDs based on the new, randomly generated 1st DWORD (offset 0). It is the client's handshake that the driver for the server variant triggers off of.

With a communication channel between the server variant and the client established, the remainder of the communication exists in the form of a sequence of encrypted datagrams. Each datagram consists of a 24 byte header followed by an optional payload section. The header is not encrypted but if the optional payload is attached, the payload is encrypted using a DWORD XOR. The format of the header is as follows:

```
struct PacketHeader
{
	DWORD dwTotalPacketSize;
	DWORD dwPktType;
	DWORD dwChecksum;
	DWORD dwEncryptionKey;
	DWORD fCompressedPayload;
	DWORD dwDecompressedSize;
};
```

The dwTotalPacketSize field defines the total size of the datagram including both the size of the header and the size of the optional payload. The dwPktType field correlates to the module ID which allows the server variant to route the datagram to the appropriate module without further inspection of the payload data. The dwChecksum value is sum of all of the bytes within the optional header (the field is ignored, but present, if there is no payload section). The dwEncryptionKey is the 32-bit XOR encryption key for the payload section. If the fCompressedPayload field is non-zero, then the data within the payload is compressed using LZO compression (prior to XOR encoding) and the dwDecompressedSize field represents the final size of the payload data after decompression. The payload section can have up to three different presentations depending on if compression is used. The first presentation is the original payload data as generated by the client or server, the second presentation is the LZO compressed form, and the final presentation (the presentation that exists going across the network) is the 32-bit XOR encoded data blob. Figure 2 provides a graphical representation of the presentation types of the payload section.

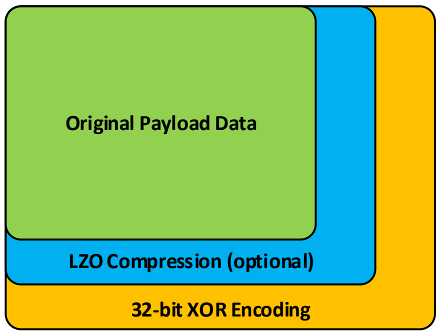

Figure 2: Possible Presentations of the Payload Section of a Derusbi Server Variant's Datagram

For each module (as defined by the dwPktType field) the optional payload section will contain a module specific format.

## Detection

Given the encrypted, and potentially compressed, nature of Derusbi server variant network traffic, detecting the traffic on a network can be problematic using traditional IDS signatures. Using a heuristic approach, it would be possible to detect the handshake of a possible Derusbi server variant session by looking for the following pattern:

| Client | Server |
|---|---|
| Exactly 64 bytes transmitted | |
| | Exactly 64 bytes transmitted |
| 40 bytes transmitted with the first 8 bytes taking the pattern of 0x28 0x00 0x00 0x00 0x02 0x00 0x00 0x00 | |

Detecting Derusbi server variants on disk is possible using the following YARA signature:

```
rule Derusbi_Server
{

        strings:
                $uuid = "{93144EB0-8E3E-4591-B307-8EEBFE7DB28F}" wide ascii
                $infectionID1 = "-%s-%03d"
                $infectionID2 = "-%03d"
                $other = "ZwLoadDriver"
        condition:
                $uuid or ($infectionID1 and $infectionID2 and $other)
}
```