NOVETTA

Derusbi (Server Variant) 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 downloading), 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.

Startup Sequence

The server variant runs as a `svchost` dependent service. While the server variant binary does have exports related to the standard service DLL (e.g. `ServiceMain`, `DllRegisterService`, etc.), the startup sequence truly begins in the `DllEntryPoint` function.

When loaded into memory via a `LoadLibrary` or equivalent function call, the server variant will determine the name of the host binary (presumably `svchost.exe`) as well as its own DLL’s name. The binary then spawns a new thread that contains the main server variant code in order to allow the `DllEntryPoint` routine to return to the calling function.

Within the main server variant function (dubbed `mainThread`), the server variant loads a pointer to the API function `GetCommandLineW`, locates the pointer in memory to the command line string, and then locates the first space within the command line string and terminates the string by placing a NULL character at the location.

The server variant then attempts to determine if it has suitable access rights within the system in order to operation. The check for access rights effectively checks to see if the server variant process is running under the `NT Authority`. If the check is unsuccessful, then the server variant terminates.

With the command line patched and authority verified, the server variant sleeps for 5 seconds before verifying that the `fShutdown` flag is not set. The `fShutdown` flag can become set by the

process loading the server variant calling the `DllRegisterServer` export. The `DllRegisterServer` function, among other tasks, will attempt to install the server variant as a server on the victim's machine. Therefore, by waiting 5 seconds before continuing the `mainThread` functionality, the server variant is giving the `DllRegisterServer` time to activate and perform the necessary operations to ensure that the server variant is properly installed and activated as a service.

The `mainThread` calls the `mainLoop` function of the server variant. The mainLoop function begins by loading the unique infection ID for the victim's machine from the registry (under the key value located at `HKLM\SOFTWARE\Microsoft\Rpc\Security`). The infection ID, if present, must be decoded by XOR'ing each byte of the string with a static byte value (typically `0x5F`). If the infection ID does not exist within the registry, the server variant will attempt to load the configuration from an encoded buffer located immediately after the static string `XXXXXXXXXXXXXXX`, decode the buffer by starting at the last byte and XORing each previous byte by the current byte value in reverse order; the server variant will then use a specific portion of the configuration blob as the infection ID's base. Next, the server variant will append a hyphen and a four digit value to the end of the infection ID to generate the unique infection ID for the victim's machine. The newly generated infection ID is then saved to the registry location stated previously.

The `mainLoop` attempts to get the privileges for `SeDebugPrivilege`, `SeLoadDriverPrivilege`, `SeShutdownPrivilege`, and `SeTcbPrivilege` in order to perform the necessary operations to load the driver portion of the server variant. The `mainLoop` will attempt to open a handle to the driver (if it is already installed) by calling `CreateFile` with the filename of `\Device\{93144EB0-8E3E-4591-B307-8EEBFE7DB28F}`. Failing this, the `mainLoop` determines if the victim's machine is running the 360 antivirus product by looking for a process with the name ZhuDongFangYu.exe. If the process is running, the driver is not installed but the `mainLoop` continues regardless. If the process is not found, however, the `mainLoop` will extract the driver binary from an encoded buffer within itself, decode the file in memory (using a rotating 4-byte XOR key), and install the driver on the victim's machine as `%SYSDIR%\Drivers\{93144EB0-8E3E-4591-B307-8EEBFE7DB28F}.sys`.

With the driver present (or recently installed), the `mainLoop` spawns another thread (dubbed `DerusbiThread::DerusbiThread`) that acts as the primary communication loop. `DerusbiThread::DerusbiThread` begins by generating a `PCC_SOCK` object. `PCC_SOCK` is an abstraction for the communication subsystem. The prototype for `PCC_SOCK` appears in Figure 1.

```
class BD_SOCK
{
  // members
  ~BD_SOCK();           // destructor
  BD_SOCK* Copy(bool fCopySocket);  // duplicate object

  // member variables
  DWORD dwLastError;
  char szHostName[256];
  DWORD dwListeningPortNumber;
  SOCKET sktRemoteEndpoint;
  SOCKET hListeningSocket;
  };

class PCC_SOCK: BD_SOCK
{
  ~PCC_SOCK(); // destructor

  // virtual members
  PCC_SOCK * Copy(bool fCopyListeningSocket);
  SOCKET ConnectToRemoteEndpointByNameAndAttemptChannelByPOSTOrHandshake(int
a2, int a3, int a4, int a5, char *pszHostname, int wHostPort);
  SOCKET ConnectToRemoteEndpointByNameAndHandshake(char *pszHostname, int
wHostPort);
  SOCKET ConnectToRemoteEndpointByNameAndPerformPOSTLogin(char *pszHostname,
int wHostPort);
  SOCKET WaitForClient();
  int SendEncodedData(int dwPktType, void *payload, size_t dwPayloadSize);
  int RecvEncodeData(DWORD *pdwPktType, char **pvPayload, DWORD
*pdwPayloadSize);
  void freeMemory(void *pMemory);

  // member functions
  int SendVictimInfo();
  int WaitForReadEvent(int dwTimeout);
  int SOCKSConnectWithRandomLocalPort(int dwEndPointIP, u_short hostshort);
  int BindToRandomPort(SOCKET s);
  int SendBuffer(SOCKET s, char *buf, int len);
  SOCKET AcceptIncomingConnection();
  SOCKET ConnectToRemoteEndpointByName(char *szHostName, int hostshort);
  SOCKET ConnectToRemoteEndpoint(int dwIP, u_short wPort);
  int ReadFromRemoteEndpoint(char *buf, int len);
  SOCKET NewSocket();
  int BindSocket(SOCKET s, int dwLowPort, int dwHighPort);
  int SendAuthenicationResponse(void *pvResponse);
  int SendHTTP200ResponseIfViaHeaderFound (char *Str);

 // member variables
 char compressionBuffer[65536];

};
```

Figure 1: PCC_SOCK Declaration in Pseudo-C++

With a new PCC_SOCK object allocates, `DerusbiThread::DerusbiThread` selects a port between 40,000 and 45,000 to use as a listening port. The port number is sent to the driver (via IOCTL `0x220200`) in order to inform the driver where to redirect incoming traffic. The “Windows

Device Driver (Firewall Hook)" section explains the functionality of the driver in greater detail. `DerusbiThread::DerusbiThread` binds to the specified port and opens the port as a listener. At this point `DerusbiThread::DerusbiThread` enters an infinite loop of waiting for new connections to the listening socket and dispatching a new thread (dubbed `CommLoop`) to handle the traffic for the socket until fShutdown is set. At this point, the startup sequence for Derusbi is complete and the server variant moves into a communication and command dispatch phase.

Windows Device Driver (Firewall Hook)

The communication between the controlling client and the Derusbi server variant depends on the device driver being in place. The authors of the device driver designed the driver to work on Windows 2000 and later 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.

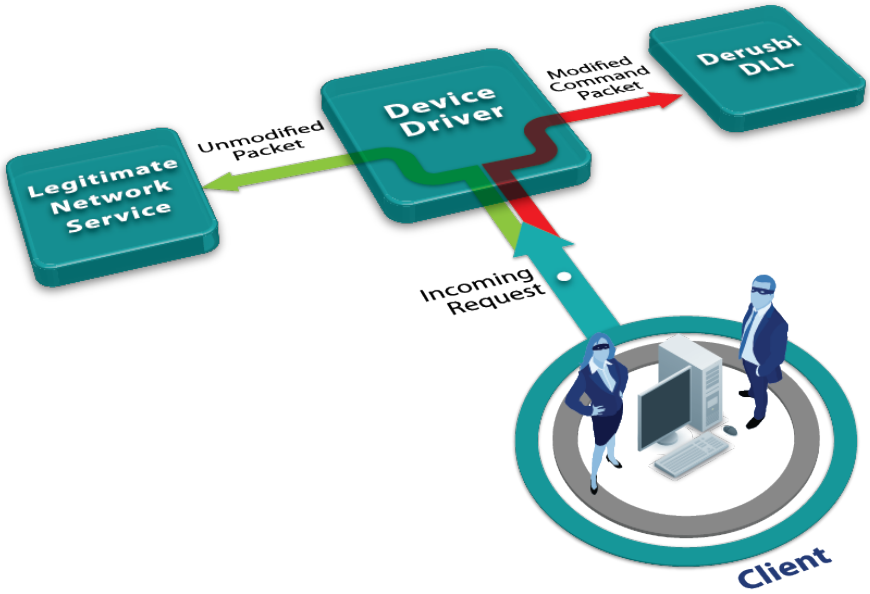

Figure 2: Device Driver Traffic Redirection

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.

Communication and Command Dispatch

The Derusbi server variant will select an available, random port between the range of 40,000 and 45,000 on the victim's machine upon which to listen. 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.

In order to establish a valid communication channel between the server variant and a controlling client, a specific handshake is required. The handshake between a client and the server variant is well defined and consisting of 64 bytes, the data within the handshake is entirely random with the exception of the 3rd and 8th DWORD. 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.

Some older versions of the server variant use a different set of DWORDs to validate the handshake, also the tests are the same. These other versions have been observed to use the following DWORDs:

```
   DWORD1 == ~DWORD0
DWORD2 == ROR(DWORD0, 7)
```

If the handshake fails, the server variant provides a secondary means to authenticate a client. Presumably a failsafe if the driver is unable to load, the secondary method requires the client to send a POST request with the following form:

```
POST /forum/login.cgi HTTP/1.1\r\n
```

In addition, the POST request must contain a `Via` field. If the request and the `Via` field exist, the server variant authenticates the client and responds with

```
HTTP/1.0 200
Server: Apache/2.2.3 (Red Hat)
Accept-Ranges: bytes
Content-Type: text/html
Proxy-Connection: keep-alive
```

If the client's request does not meet the appropriate authentication criteria, the server variant sends:

```
HTTP/1.0 400  Bad Request
Server: Apache/2.2.3 (Red Hat)
Connection: close
```

With a communication channel between the server variant and the client established, the server sends information about the victim's computer. Consisting of a 180 byte data structure (Figure 3), the server variant provides the client with a variety of details about the victim's machine. The `VictimInfoPacket` has an identifier of `2` (see the `dwPktType` explanation below). The communication between the server and the client at this point, and for the remainder of the session, is encrypted.

```
#pragma pack(push, 1)
struct VictimInfoPacket
{
  int magicValue;
  char szInfectionID[64];
  char szComputerName[64];
  char szSelfIP[16];
  char unknownArray[16];
  int dwOSandSPVersionInfo;
  int dwBuildNumber;
  char unknownValue;
  char OemId;
  __int16 unused_align2;
  int fCampaignCodeMatch;
};
#pragma pack(pop)
```

Figure 3: VictimInfoPacket Structure Definition

Communication between the client and the server variant 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 4 provides a graphical representation of the presentation types of the payload section.

Figure 4: Possible Presentations of the Payload Section of a Derusbi Server Variant's Datagram

After sending the server information via the `VictimInfoPacket`, the server variant spins off a `CommLoop` thread for the connection and returns to waiting for new connections from clients to appear.

The `CommLoop` thread begins by establishing the set of internal command handlers available to the server variant. With the exception of the administrative command handler (which is built into the `CommLoop`), each of the internal commands consists of an object derived from a base object `PCC_BASEMOD`.

```
class PCC_BASEMOD
{
	~PCC_BASEMOD();	// destructor

	// virtual member functions
	void *return1(); // always returns 1
	void Cleanup(void);
	void ProcessPacket(void *pkt, DWORD dwPktSize);
	int ReadWaitingData(void **pPacket, DWORD *dwPktSize);
	int MallocWithClear(size_t Size);
	int Free(void *Memory);

	// member variables
	DWORD dwPacketIdentifierCode;
};
```

Figure 5: PCC_BASEMOD Pseudo-C++ Declaration

The server variant appears to have a modular design allowing an attacker to compile 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 has observed the following modules:

ID	Class Name	Module Description
0x81	PCC_CMD	Remote command shell
0x82	PCC_PROXY	Network tunneling
0x84	PCC_FILE	File management
0xF0	n/a	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.

After establishing the tools, an infinite loop (`CommLoop`) is entered in which the server variant will wait for up to 1/100 of a second for input from the network; if such input arrives, the server routes the packet to the appropriate handler. If the network input does not arrive, the `CommLoop` queries each of the command handlers for any packets they may have queued (by calling each command handler's `ReadWaitingData` function) and transmits the packets the handlers have generated. Additionally, if more than 60 seconds passes between network inputs from the client or network outputs from the server variant, the `CommLoop` will send out a beacon packet (`dwPktType = 4`).

`CommLoop` routes packets to the appropriate command handler object by locating the `dwPacketIdentifierCode` within each of the registered command handlers that matches the incoming packets `dwPktType`. When the appropriate command handler is found, `CommLoop` passes the payload of portion of the packet to the command handler's `ProcessPacket` function.

PCC_CMD

The `PCC_CMD` object contains the remote shell functionality of the server variant along with the ability to execute arbitrary programs. Derived from the `PCC_BASEMOD` class, the `PCC_CMD`

class's operations are focused largely in the `ProcessPacket` and `ReadWaitingData` functions. The `PCC_CMD::ProcessPacket` function works as a stub function that merely passes the packet's payload data (`pkt`) data to `PCC_CMD::ProcessPacketEx` while ignoring the `dwPktSize` parameter. The packet's payload data is, in and of itself, another datagram with a header and optional payload section. The payload of each `PCC_CMD` destined packet contains the following header:

```
struct PCCCMDPacketHeader
{
  DWORD dwPacketSize;
  DWORD field_4; // purpose unknown, seemingly unused.
  DWORD dwCommandType;
  DWORD dwLastError;
};
```

The `dwCommandType` field specifies the specific `PCC_CMD` command that the client is requesting the server variant perform. There are four commands that `PCC_CMD` supports:

dwCommandType	Functionality
0x04	Activate the remote shell
0x08	Execute the specified file
0x0C	Send input to remote shell
0x10	Terminate the remote shell

For each of the commands, any output from or acknowledgement of the commands comes in the form of another packet consisting of a `PacketHeader` followed by a `PCCCMDPacketHeader` and any optional payload data. The `dwCommandType` of the newly constructed packet matches the command's original `dwCommandType` value (e.g. responses from `0x04` commands will reply with `dwCommandType` set to `0x04`). `PCC_CMD::ProcessPacketEx` will queue the response packets in an internal buffer.

The `PCC_CMD::ReadWaitingData` member function is responsible for transmitting any of the previously queued packets from `PCC_CMD::ProcessPacketEx`. If there are no queued packets, `PCC_CMD::ReadWaitingData` will perform a queue of the console output pipe for the remote shell process (if it is active); the function will also attempt to read the entirety of the waiting data, which then becomes the payload of a `PacketHeader`/`PCCCMDPacketHeader` based packet with the `dwCommandType` set to `0x0C`. If the read is unsuccessful, the function returns a `PacketHeader`/`PCCCMDPacketHeader` based packet with the `dwCommandType` set to `0x10` indicating an error and terminating the remote shell session.

PCC_FILE

The `PCC_FILE` object provides a large range of file system administration functions. `PCC_FILE` is derived from the `PCC_BASEMOD` class meaning that the processing of commands should be contained within the `PCC_FILE::ProcessPacket` member function with some additional processing done in the `PCC_FILE::ReadWaitingData` member function. This is not necessarily the case, however.

The PCC_FILE::ProcessPacket member function, much like PCC_CMD::ProcessPacket, is little more than a stub function that passes only a copy of the payload data (pkt) to PCC_FILE::ProcessPacketEx. PCC_FILE::ProcessPacketEx performs no file management operations but instead adds any incoming command packets to a queue for processing by PCC_FILE::ReadWaitingData if the packet is not already within the queue (thus avoiding duplication of commands).

The PCC_FILE::ReadWaitingData member function is a stub function that calls PCC_FILE::ProcessQueue and returns the resulting packet from the queue processing. This means that file operations are surprisingly low priority, and potentially, high latency operations.

Each packet that arrives within the packet queue of PCC_FILE contains a standard header followed by a (quasi-optional) payload data blob. The header for the PCC_FILE command packets takes the following form:

```
struct PCCFilePacketHeader
{
  DWORD dwTotalPayloadSize;
  DWORD dwCommandType;
};
```

The dwCommandType field specifies the specific PCC_FILE command that the client is requesting the server variant to perform. PCC_FILE supports 17 (of which 15 are unique) commands. While the general form within the Derusbi server variant communication model is to return a packet with the same dwCommandType as the original command, many of the PCC_FILE commands return a status packet type (dwCommandType = 0x04).

dwCommandType	Functionality	Response dwCommandType
0x0C	Purge PCC_FILE Commands from Queue Based on dwCommandType	(no response)
0x10	Enumerate Attached Drives	0x10
0x14	Get File Attributes	0x14
0x18	File Search	0x18
0x1C	Rename File	0x04
0x20	Delete File	0x04
0x24	Create Directory	0x04
0x28	Upload File to Client	0x28 and 0x04
0x2C	Recursively Enumerate Directory	0x2C
0x30	Download File from Client	0x04
0x34	Copy File	0x04
0x38	Move File	0x04
0x3C	Duplicate File Timestamp	0x04
0x40	Execute File	0x04
0x44	Recursively Enumerate Directory	0x44
0x48	Recursively Enumerate Directory	0x48
0x4C	Enumerate All Drives and Files	0x4C

Commands 0x2C, 0x44, and 0x48 all appear to be the exact same base command with only slight variations in their response format. It is unclear why this particular command is included three times in PCC_FILE.

PCC_PROXY

The PCC_PROXY object provides the platform for a tunneling network traffic to and from the client to a specific endpoint (or endpoints if multiple tunnels are activated by the client). Derived on the PCC_BASEMOD class, the PCC_PROXY class performs very little network tunneling within the CommLoop interactive PCC_PROXY::ProcessPacket and PCC_PROXY::ReadWaitingData member functions. The PCC_PROXY::ProcessPacket member function queues incoming PCC_PROXY packets into a received queue while PCC_PROXY::ReadWaitingData returns packets from a transmit queue, with the directionality from the perspective of the server variant. The core of the PCC_PROXY's network tunneling comes from a spawned processing thread (PCC_PROXY::MainThread) that is generated when the PCC_PROXY object is instantiated.

The PCC_PROXY::MainThread function consist an infinite loop that only terminates when the PCC_PROXY::fShutdown flag is set. Otherwise, the loop will inspect another internal flag (PCC_PROXY::fNetworkEnabled) to determine if the network tunneling is currently active. If the PCC_PROXY::fNetworkEnabled flag is set to false, then tunneling is disabled but command processing continues.

It is possible to have more than one tunnel active at any given time. In order to firewall tunnels from each other over the backbone of the server variant's command channel, each tunnel is assigned a specific channel identifier. This allows the client to specify which specific tunnel data is transmitted to as well as telling the client which tunnel is returning data. If the PCC_PROXY::fNetworkEnabled flag is set to true, PRC_PROXY::MainThread will loop through all active channels, perform a select on the socket connected to the endpoint and -- if the select indicates that there is data waiting on a particular socket -- the data is read. A new PCC_PROXY based packet is then generated and the packet is queued for delivery to the client.

After processing each of the channels for new data, PCC_PROXY::MainThread processes incoming command packets from the client (an operation usually handled by the PCC_BASEMOD::ProcessPacket function). Packets belonging to the PCC_PROXY subsystem have a common header, much like the other PCC_BASEMOD derived classes. To this end, the PCC_PROXY packets have the same packet header as the PCCFilePacketHeader packet header.

The `PCC_PROXY` supports five commands:

dwCommandType	Functionality
0x04	Connect to Specified Endpoint (Creates New Channel)
0x08	Send Data to Endpoint
0x0C	Terminate Channel
0x10	Enable Network Tunneling (`PCC_PROXY::fNetworkEnabled` set to `true`)
0x14	Disable Network Tunneling (`PCC_PROXY::fNetworkEnabled` set to `false`)

Administrative Commands

The administrative commands are built-in to the server variant and are not derived from the `PCC_BASEMOD` class. Each of the administrative command packets contains the same header structure as the `PCCFilePacketHeader` structure followed by an optional payload data blob.

The administrative commands consist of the following five commands:

dwCommandType	Functionality
0x08	Terminate the current network connection between the client and the server variant.
0x10	Run the `Cleanup` member function of each of the registered `PCC_BASEMOD` derived objects, effectively resetting the state of each of the modules.
0x14	Write infection ID to the registry and immediately terminate the server variant.
0x18	Shutdown the server variant (set `fShutdown` to `true`)
0x1C	Drop a new DLL to `%TEMP%\tmp1.dat`, load the DLL into memory and call `DllRegisterServer` to install a new server variant binary on the victim's system.

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
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)
}
```