virus BULLETIN Covering the global threat landscape

DRIDEX IN THE WILD

Meng Su
Tencent, China

Dridex is a descendent of the Cridex malware. Its initial spread occurred in late 2014 via spam and the malware is still active in the wild today. Dridex is a *Windows* executable which uploads system information to its C&C server before downloading a DLL. After the DLL has been installed by the executable, the C&C server will control the infected PC, sending it commands to carry out further harmful instructions. In this article, we will analyse the main executable, focusing on the following actions: obtaining APIs, getting server data, getting and encoding system information, and communicating with the C&C server.

OBTAINING APIs

All of the *Windows* APIs the bot uses are obtained by a function. The argument passed into this function is only an index. This 'index' is an index number of the API_Address array – the API-name-encode-data-block uses the same index value.

At first, the malware checks the API_Address array, which is initiated with a NULL value. If API_Address[API_index] is found with a valid value, the function returns the address. Otherwise the malware moves onto the next step.

In the second step, the malware decodes the API_Name from the API-name-encode-data-block with the API_index using an algorithm which is predefined by the malware itself. The decoded data contains two parts, DLL_index and API_Name:

```
API_Data
{
BYTE DLL_index;
BYTE[] API_Name;
};
```

The role of DLL_index is the same as that of the API_index. The malware has a DLL_Module array which is similar to the API_Address array and also a similar DLL-name-encode-data-block.

The malware checks the DLL_Module array. If it finds valid data at DLL_Module[DLL_index], then it returns the DLL module for the next step. Otherwise the malware will get the DLL module using the following method: similar to API_Name, the DLL_Name is decoded from the DLL-name-

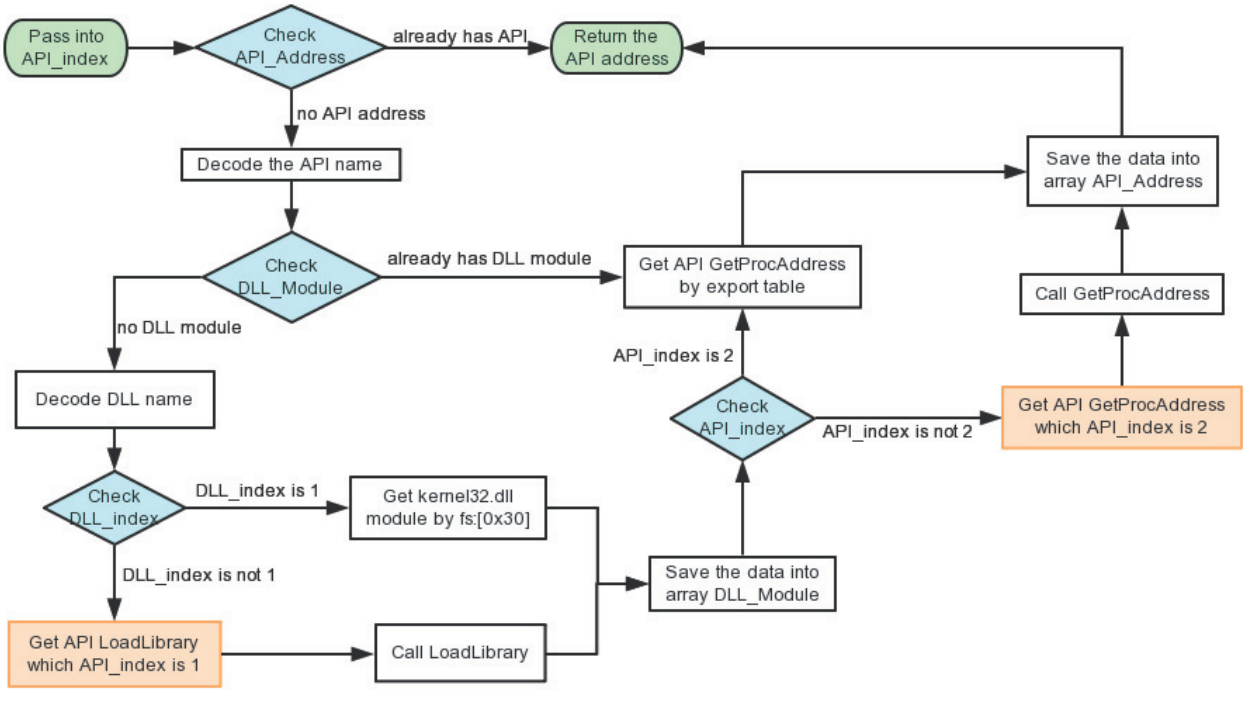


Figure 1: Obtaining APIs.

encode-data-block by DLL_index. After that, the malware checks whether the DLL_index value is equal to one. By the design of the malware, the DLL_index of kernel32.dll is one. The way to get this DLL's module is using register fs:[0x30], which points to the PEB structure, and then finding the PEB_LDR_DATA structure via the PEB. In the PEB_LDR_DATA structure we can find out the DLL base address by comparing the DLL name. If the DLL_index value is not one, the malware will get the LoadLibrary API whose API_index value is one. The malware then uses this API to get the DLL module. The malware records the DLL module into the DLL_Module array, regardless of whether or not the DLL_index is 1.

If the API_index passed is 2, which represents the GetProcAddress API in this malware, the bot will traverse the DLL's export table to get the API address. Otherwise the malware will get the GetProcAddress API first and then call this API to get the other API's address. The API address will be saved into the API_Address array.

Note that the addresses of the LoadLibrary and GetProcAddress APIs are always the first two addresses obtained by the malware. The flowchart in Figure 1 shows the full logic steps of getting any single API address.

GETTING SERVER DATA

In this malware, the C&C server address is not stored as plain text. The malware uses the GetModuleHandleW API to locate the IMAGE_DOS_HEADER, and then locates the section header. It will find one section's virtual address whose section name is '.sdata' (Figure 2).

This section contains only 0x7A valid bytes. The first DWORD (0xA9E97561 in this case) is a key which is used to XOR the other 0x76 bytes. Figure 3 shows the content after it has been XOR'ed.

As shown in Figure 3, this data is still encrypted. The 0x76 byte-long decoded data consists of three parts: the size of the

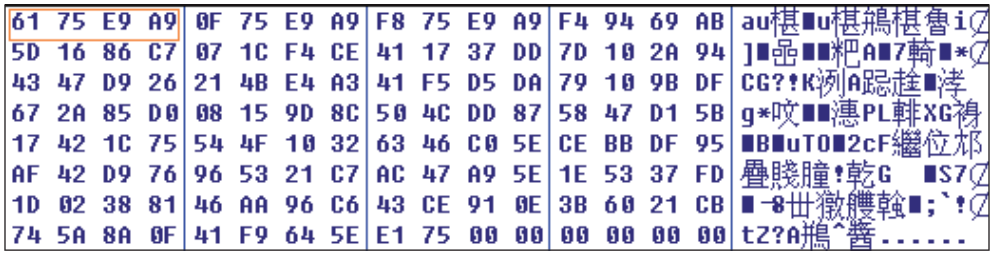


Figure 2: .sdata section content.

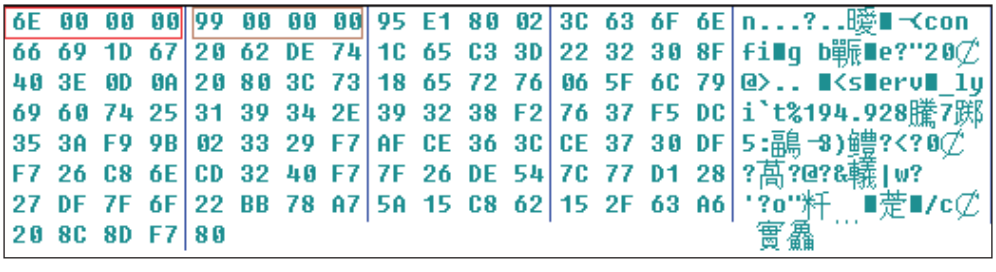


Figure 3: XOR content.

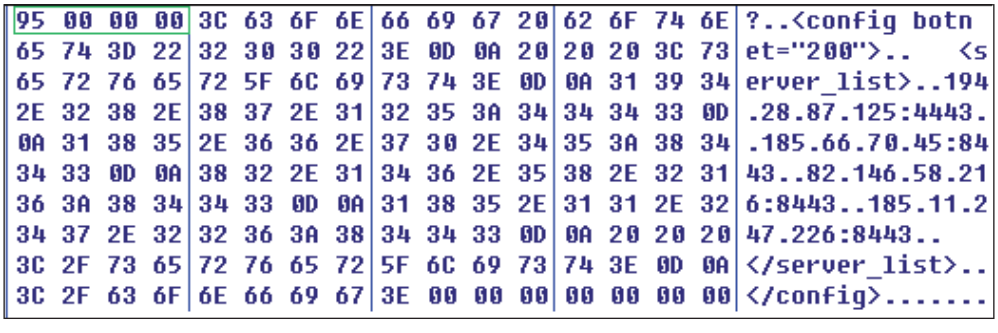


Figure 4: The raw data.

encoded data (0x6E), the size of the raw data (0x99), and the encoded data:

```
Encoded_Server_Data
{
DWORD szEncodeData;
DWORD  szRawData;
BYTE[]  EncodedData;
};
```

The 'EncodedData' is compressed by the aPLib algorithm [1]. The decompressed raw data is shown in Figure 4.

The first four bytes of this raw data indicate the length of the data behind it. Figure 5 shows the configuration of the server. The 'botnet' attribute shows the botnet_id; the 'server_list' tag shows the server URLs.

```
<config botnet="200">
   <server_list>
194.28.87.125:4443
185.66.70.45:8443
82.146.58.216:8443
185.11.247.226:8443
   </server_list>
</config>
```

Figure 5: Server data.

After getting the server URL, the malware will collect system information for further communication.

GETTING AND ENCODING SYSTEM INFORMATION

The collected information will be stored in XML format in two parts. The first part is composed as follows:

```
<loader><get_module unique="%s" botnet="%d"
system="%dv" name="bot" bit="%d"/>
```

Meanwhile, the format of the other part is:

```
<soft><![CDATA[%s]]></soft></loader>.
```

In the first part, the value of the 'unique' attribute records a string relating to three registry entries:

```
Key: HKEY_LOCAL_MACHINE/SYSTEM/CurrentControlSet/
Control/ComputerName/ComputerName
Name: ComputerName

Key: HKEY_LOCAL_MACHINE/Volatile Environment
Name: USERNAME

Key: HKEY_LOCAL_MACHINE/SOFTWARE/Microsoft/Windows
NT/CurrentVersion
Name: InstallDate
```

The malware retrieves the values of these three keys and combines them as a data block, System_Info, then calculates the MD5 of this data block. The malware also checks every character of the 'ComputerName' value. If a character is not found on a list which contains the Latin letters and some special symbols, it will be replaced with the character '?'. I think the malware author made a mistake here: there is no letter 'D' on the letter list and there is an extra 'S' – I guess that's because 'D' is pretty close to 'S' on the keyboard and this was probably a typo. This means that the malware will replace 'D' with '?'. In the end, the changed 'ComputerName' value and the MD5 of the System_Info are joined with the character '_' (Figure 6), then set with the 'unique' attribute.

The value of the 'botnet' argument is a botnet_id which is the same botnet_id as in the server configuration (Figure 5). The value of the 'system' attribute is a hash value which indicates the version of the operating system (e.g. *XP* or *Win7*), whether it is an NT kernel or not, whether or not it is running as administrator, and whether or not the UAC is enabled. The value of the 'bit' attribute indicates whether the operating system is 32-bit (32) or 64-bit (64).

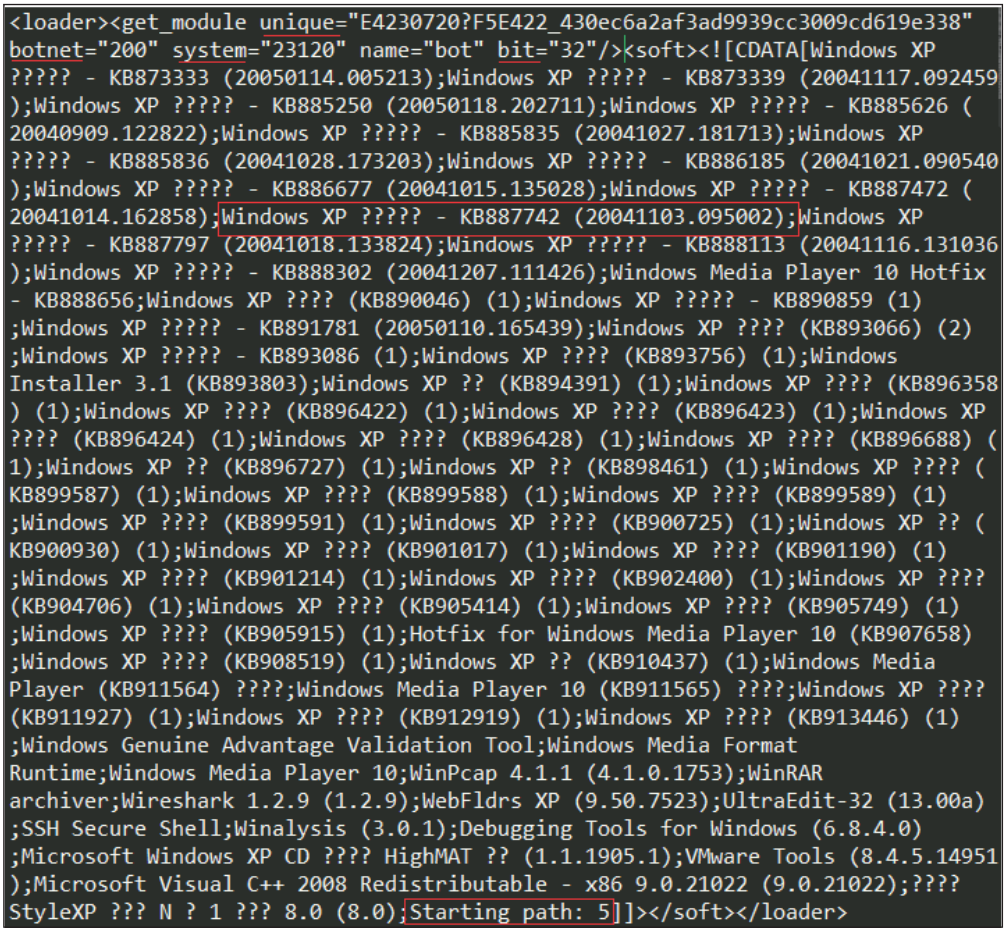
In the second part, the content in 'CDATA' is the information about the installed software. The malware enumerates all the subkeys of HKEY_LOCAL_MACHINE/SOFTWARE/Microsoft/Windows/CurrentVersion/Uninstall and gets the value of their key name, 'DisplayName' and 'DisplayVersion'. It will compose a string with the format 'DisplayName_value (DisplayVersion_value)' and connects every subkey's string with the character ';'. It should be noted that the malware only recognizes English characters, so it changes the non-English characters to '?'.

The malware attaches a string, 'Starting path: %d', to the end of the connected string. Despite what its name may suggest, the content of 'Starting path' is not a real path of the malware. Instead, it is a figure indicated in the MIC (Mandatory Integrity Control [2]) level of the path which the malware located. There are seven levels: untrusted, low, medium, medium plus, high, system and protect process, which correspond to the values 1–7. If the operating system version is higher than or equal to *Windows NT 6.0*, the malware uses the GetSidSubAuthority API to get the MIC level. Otherwise, it sets the figure to 5. Figure 7 shows the raw data which will be sent to the server.

45	34	32	33	30	37	32	30	3F	46	35	45	34	32	32	5F
34	33	30	65	63	36	61	32	61	66	33	61	64	39	39	33
39	63	63	33	30	30	39	63	64	36	31	39	65	33	33	38

E4	23	07	20	?F	5E	42	2_								
43	0e	c6	a2	af	3a	d9	93								
9c	c3	00	9c	d6	19	e3	38								

Figure 6: The content of the 'unique' argument.



```
<loader><get_module unique="E4230720?F5E422_430ec6a2af3ad9939cc3009cd619e338"
botnet="200" system="23120" name="bot" bit="32"/><soft><![CDATA[Windows XP
????? - KB873333 (20050114.005213);Windows XP ????? - KB873339 (20041117.092459
);Windows XP ????? - KB885250 (20050118.202711);Windows XP ????? - KB885626 (
20040909.122822);Windows XP ????? - KB885835 (20041027.181713);Windows XP
????? - KB885836 (20041028.173203);Windows XP ????? - KB886185 (20041021.090540
);Windows XP ????? - KB886677 (20041015.135028);Windows XP ????? - KB887472 (
20041014.162858);Windows XP ????? - KB887742 (20041103.095002);Windows XP
????? - KB887797 (20041018.133824);Windows XP ????? - KB888113 (20041116.131036
);Windows XP ????? - KB888302 (20041207.111426);Windows Media Player 10 Hotfix
- KB888656;Windows XP ???? (KB890046) (1);Windows XP ????? - KB890859 (1)
;Windows XP ????? - KB891781 (20050110.165439);Windows XP ???? (KB893066) (2)
;Windows XP ????? - KB893086 (1);Windows XP ???? (KB893756) (1);Windows
Installer 3.1 (KB893803);Windows XP ?? (KB894391) (1);Windows XP ???? (KB896358
) (1);Windows XP ???? (KB896422) (1);Windows XP ???? (KB896423) (1);Windows XP
???? (KB896424) (1);Windows XP ???? (KB896428) (1);Windows XP ???? (KB896688) (
1);Windows XP ?? (KB896727) (1);Windows XP ?? (KB898461) (1);Windows XP ???? (
KB899587) (1);Windows XP ???? (KB899588) (1);Windows XP ???? (KB899589) (1)
;Windows XP ???? (KB899591) (1);Windows XP ???? (KB900725) (1);Windows XP ?? (
KB900930) (1);Windows XP ???? (KB901017) (1);Windows XP ???? (KB901190) (1)
;Windows XP ???? (KB901214) (1);Windows XP ???? (KB902400) (1);Windows XP ????
(KB904706) (1);Windows XP ???? (KB905414) (1);Windows XP ???? (KB905749) (1)
;Windows XP ???? (KB905915) (1);Hotfix for Windows Media Player 10 (KB907658)
;Windows XP ???? (KB908519) (1);Windows XP ?? (KB910437) (1);Windows Media
Player (KB911564) ????;Windows Media Player 10 (KB911565) ????;Windows XP ????
(KB911927) (1);Windows XP ???? (KB912919) (1);Windows XP ???? (KB913446) (1)
;Windows Genuine Advantage Validation Tool;Windows Media Format
Runtime;Windows Media Player 10;WinPcap 4.1.1 (4.1.0.1753);WinRAR
archiver;Wireshark 1.2.9 (1.2.9);WebFldrs XP (9.50.7523);UltraEdit-32 (13.00a)
;SSH Secure Shell;Winalysis (3.0.1);Debugging Tools for Windows (6.8.4.0)
;Microsoft Windows XP CD ???? HighMAT ?? (1.1.1905.1);VMware Tools (8.4.5.14951
);Microsoft Visual C++ 2008 Redistributable - x86 9.0.21022 (9.0.21022);????
StyleXP ??? N ? 1 ??? 8.0 (8.0);Starting path: 5]]></soft></loader>
```


Figure 7: The raw data sent to the server.

Finally, the malware gets a random DWORD key and uses a XOR operation to encode the raw data of every DWORD.

COMMUNICATION

Before the communication begins, the malware will parse the server data (Figure 5). The parsing function checks the special characters (e.g. '://', '@', '/', ':', '?', '#') to locate the communication protocol, server address, file path, arguments, port, user name and password. If the hard-coded URL does not have a communication protocol, the malware will set 'HTTP' as default. The port field also has default values: for HTTP it is 80, for HTTPS it is 443 and for FTP it is 21. Other fields default to NULL if no matching value is found in the string. In this sample, the server data is very simple, with only server address and port. As shown in Figure 5, the server URL is of the format 194.28.87.125:4443. By design, the malware uses HTTPS for communication, so before calling the parsing function, the malware will prepend the URL string with the HTTPS protocol. After calling the parsing function, the malware will get the server address as 194.28.87.125 and the port as 4443. After parsing, the malware uses the InternetConnectW API to connect to the server, sends the encrypted data using the HttpSendRequestW API, and finally reads the response from the server using the InternetReadFile API.

The data received from the server is also encrypted. The first four bytes is a DWORD key which is used as the XOR key to decode the data after it by DWORD (Figure 8).

The decoded data is encased in XML code which starts with a '<root></root>' element. In the root node, there are two sub nodes, <nodes></nodes> and <module name="bot" bit="32"></module>. The content in the 'module' node is encoded with the BASE64 algorithm. Figure 9 shows a piece of the data after decoding.

The first 0x80 bytes of the decoded data is junk code, after which is a DLL. The malware writes this DLL into a TEMP file whose directory is the same as the malware. Then it creates a registry entry: HKEY_CURRENT_USER/Software/

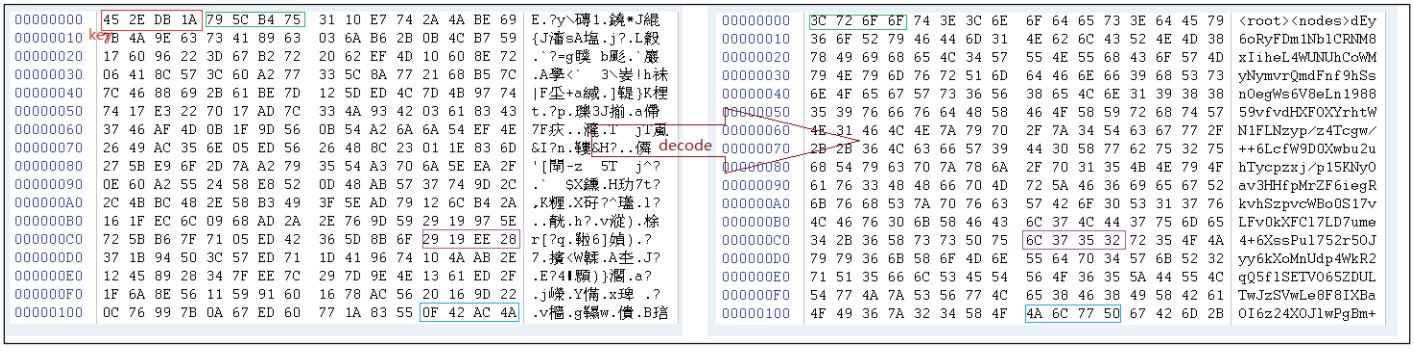


Figure 8: XOR the downloaded data.

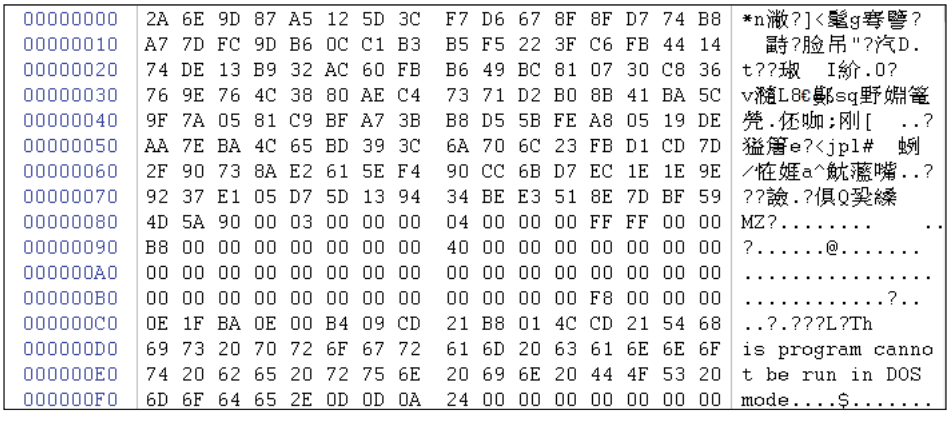


Figure 9: A piece of data decoded with BASE64.

Microsoft/Windows/CurrentVersion/Explorer/CLSID/%s/ShellFolder. The '%s' is a GUID which is transformed from the MD5 of a variant of the System_Info data block (described earlier). The System_Info variant only adds a byte, 0x13, following the System_Info block. The value of this registry is encrypted by a customized algorithm. Its raw data is in the format <cfg net="%d" build="0"><startup>%s</startup>~~%S~~</cfg>. The value of the 'net' attribute is botnet_id, the content in the 'startup' section is retrieved from the 'nodes' section, which is sent from the C&C server, the content of the 'del' section is the path of the malware. Finally, the malware calls the CreateProcessW API to run the DLL with argument 'rundll32.exe "<DLL_path>" NotifierInit'. The DLL has an export function, NotifierInit, and this DLL will carry out further orders received from the server.

CONCLUSION

By analysing the malware in detail, we have learned about its working mechanism and how it gathers information and communicates with the C&C server. We can now forge data and send it to the server, decode the response and check the server commands. In this way, we might obtain more commands for further research or obtain the latest variants in order to keep track of this malware.

REFERENCES

[1] http://ibsensoftware.com/products_aPLib.html.

[2] http://en.wikipedia.org/wiki/Mandatory_Integrity_Control.

Editor: Martijn Grooten

Chief of Operations: John Hawes

Security Test Engineers: Scott James, Tony Oliveira, Adrian Luca

Sales Executive: Allison Sketchley

Editorial Assistant: Helen Martin

Developer: Lian Sebe

Consultant Technical Editor: Dr Morton Swimmer

Tel: +44 (0)1235 555139. Fax: +44 (0)1865 543153

Email: editorial@virusbtn.com

Web: http://www.virusbtn.com/