virus BULLETIN Covering the global threat landscape

LEARNING ABOUT BFLIENT THROUGH SAMPLE ANALYSIS

Meng Su & Dong Xie
Fortinet, China

The first variant of Bflient was discovered as long ago as June 2010. Bflient is a worm that spreads via removable media. It steals sensitive information, updates configuration files and downloads other malware. In addition, it opens unsolicited commercial advertising links.

By studying different variants, we have deduced that they were built using ButterFly Flooder – a toolkit for building worms with bot functionality. ButterFly Flooder is an update to the ButterFly Bot toolkit [1], which was used, among other things [2], to create the Mariposa botnet [3]. The malware also includes a conspicuous text string, 'BFFclient v1.11b', which indicates its relationship with the toolkit. This analysis focuses on the latest variants of the worm.

PACKER & INJECTION

The function of the __setusermatherr() API is to replace the default _matherr() routine with a user-defined routine for handling maths errors. Bflient is built using the *Microsoft* Visual C++ compiler, and is probably packed with UPX. Under *Microsoft* Visual C++, if the user writes a self-defined routine named '_matherr()', the compiler will automatically call the __setusermatherr() API to replace the default _matherr() with the user-defined version. The malware takes advantage of this feature in an anti-debugging trick: it raises a maths error deliberately to trigger its _matherr() routine.

The malware unmaps its original section image view and reuses its memory to map the decrypted file. It checks the IMAGE_FILL_DLL flag to decide on the mapping method. After that, the malware creates a new thread and suspends the main one. The new thread adjusts the context of the main thread and wakes it up. The resumed thread will execute from the entry point of the decrypted file.

The injection procedure begins with the creation of a suspended svchost.exe process. The malware writes its malicious code and arguments into the pre-allocated memory of the remote process. It obtains the entry point address of the remote process and rewrites it using the following classic code snippet, then resumes the process:

```
mov [esp+4], DWORD_1 //arguments
push DWORD_2 //entry point
retn
```

INITIALIZATION

The malware uses more than half of its code for its initialization. It utilizes redundant code, for example encapsulating an API with several different routines which have the same functionality. It stores data and function addresses in memory, and accesses them indirectly. Although these tricks increase the complexity of reverse engineering, the initialization data is well structured, which makes things simpler. We can roughly divide the initialization process into the following parts:

Obtaining APIs

The malware gets the base address of kernel32.dll from the InInitializationOrderModuleList of the PEB_LDR_DATA structure in the PEB (Process Environment Block). It compares the length of each of the module names on the list with the length of kernel32.dll, and fetches the base address of the first match it finds. The malware uses LoadLibraryA() to get the base addresses of other modules, as it insists on walking the export table in order to obtain APIs instead of calling GetProcAddress().

The module and API information is saved into a table. Each record in the table points to a module information chain, and the index of the record is a hash byte value based on the module name. If different modules have the same index, the malware links them to the same chain. At the end of each module information chain, there is also a table for its API information. Here, the hash index is generated based on the API name. Whenever there is a request for an API, the malware looks up this table first. In the case of both the module and API information chains, if there is no record for the request, a new one is appended.

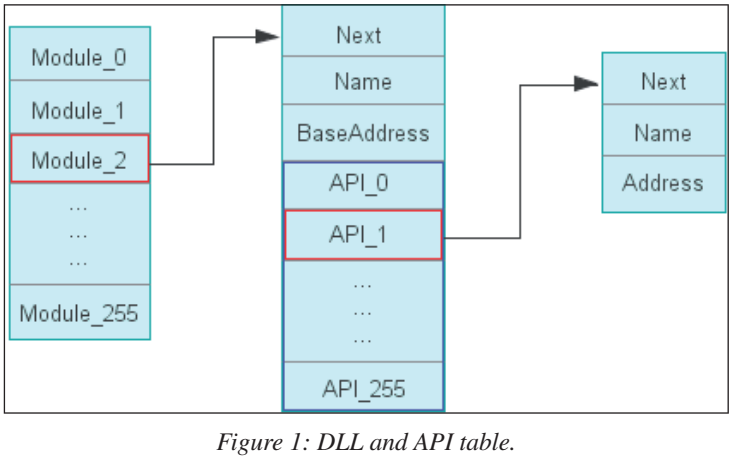

Figure 1: DLL and API table.

Hidden window

The malware takes advantage of the *Windows* message-handling mechanism for its own purposes. It creates a hidden window to handle messages dispatched elsewhere. If the message is WM_QUERYENDSESSION and the user is logging off from *Windows*, it notifies the C&C server. Any other messages will initially be forwarded to the malware-defined message-handling function. Any messages that the malware does not care about will be passed onto the default *Windows* procedure.

Module block

There are six hard-coded module blocks, each of which consists of a BLOCK_HEADER structure and encrypted data. The decrypted data includes a piece of code – if the code needs to be relocated, it also includes a relocation table. The malware XORs the checksum value of the decrypted data with the hash key, but the result is somewhat irrelevant, since the malware can also get the block from the C&C server. The result is probably used to verify the integrity of the downloaded data.

```
typedef struct _BLOCK_HEADER
{
        BYTE MajorID;
        BYTE MinorID;
        BYTE Key;
        BYTE Flag;
        LONG HashKey;
        LONG BlockSize;
        LONG CodeOffset;
        LONG DataOffset;
        LONG NumberOfRelocs;
        LONG EntryOffsetToCode;
} BLOCK_HEADER, *PBLOCK_HEADER;
```

The malware checks the major ID first. If it is not zero modulo four, the module block is a decryption or encryption function. Otherwise, the malware checks the minor ID and appends the block to a MODULE_BLOCK structure described block chain. The block is ignored if the minor ID is zero and the major ID is not equal to an odd multiple of four.

```
typedef struct _MONITOR
{
        struct _MONITOR* Next;
        PVOID fMonitor;
...
} MONITOR, *PMONITOR;
typedef struct _MODULE_BLOCK
{
        struct _MODULE_BLOCK* Next;
        PVOID fEntry;
        PVOID fFree;
        PVOID fMain;
        PVOID fDelete;
        PVOID fMessage;
        ...
PMONITOR pMonitor;
...
} MODULE_BLOCK, *PMODULE_BLOCK;
```

The structure defines which members rely on the concrete module block. According to the hard-coded blocks, the functionality of the members can be summarized as follows:

- fEntry: the entry point – sometimes performs initialization work instead of fMain.
- fFree: performs cleaning duties, such as freeing up memory, closing handles, etc.
- fMain: the core job of each block – if this function does not exist, the job is transferred to fEntry.
- fDelete: deletes traces of the malware, such as registry entries, copied files etc.
- fMessage: handles the *Windows* messages the malware is interested in.
- fMonitor: monitors some jobs, such as adding registry entries, downloading files, etc.

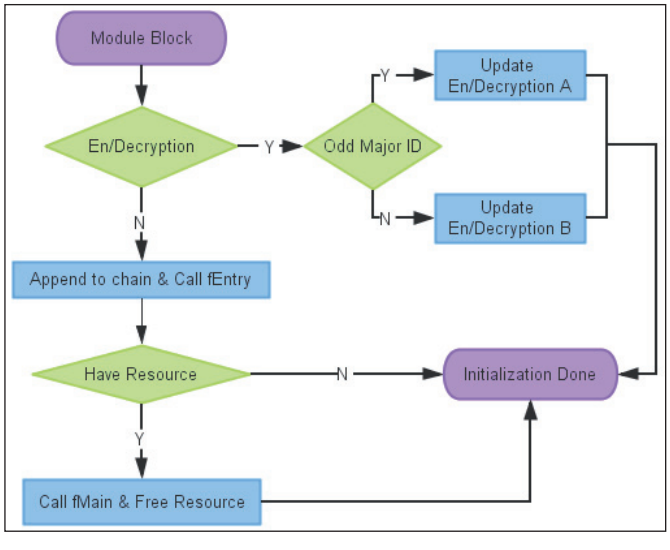

Figure 2: Handling module block.

Resource block

Resource blocks are usually downloaded from the C&C server, with the exception of one that is hard coded. Although the content of the block is variable, it always contains a minor ID and an action ID. Table 1 shows the actions based on different IDs.

The case of the action 0xA0 is a little more complicated. Table 2 shows the actions of the different commands.

Minor ID	Action ID	Actions
!=0	0xC0	Contains a new module block which will be appended to the module block chain.
	0xA0	Searches MODULE_BLOCK in chain by minor ID and calls fMain. If there is no match, the resource block is saved.
==0	0x21	Gets an IP address by randomly picking a C&C server URL from the hard-coded list; frees the saved resource blocks.
	0x10	Sends gathered information to the C&C server, such as whether the local IP is changed or not, the system version, etc.
	0xA0	The resource block contains a command ID.

Table 1: Actions based on minor and action IDs.

Block size	Command	Actions
==1	0x1	Resets a flag, the malware frees and exits.
	0x2	Sends confirmation of received data to the C&C server.
	0x3	Sends the 'BFFclient v1.11b' text string to the C&C server. Since the string is constant from one variant to another, the intent is not clear.
	0x7	Sends the minor IDs of the module and resource blocks to the C&C server.
>1	0x4	Drops an EXE file and runs it.
	0x6	Calls each fDelete function of MODULE_BLOCK in the chain.

Table 2: Actions based on commands of action 0xA0.

```
typedef struct _RESOURCE_BLOCK
{
        struct _RESOURCE_BLOCK* Next;
        PVOID pResource;
        LONG Size;
        ...
        BYTE MinorID;
...
} RESOURCE_BLOCK, *PRESOURCE_BLOCK;
```

WHAT BLOCKS DO

Since we know about the mechanisms of module and resource blocks, let's take a quick look at what the hard-coded blocks do.

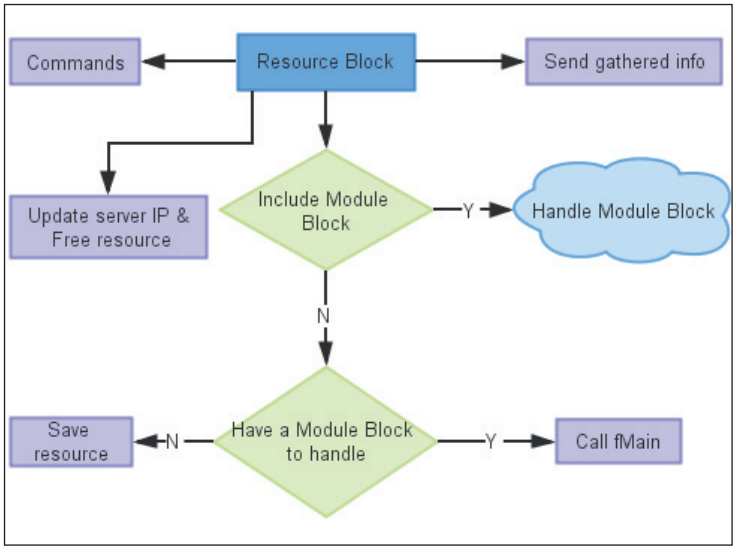

Figure 3: Handling resource block.

En/decryption

The malware provides two different en/decryption functions. The encryption function encrypts the information before it is sent to the C&C server. The decryption function decrypts the downloaded configuration data from the server. The en/decryption functions can be updated from the C&C server, for instance to use new algorithms.

Auto-run

The malware copies itself to the %user profile% folder. In order to perform this duplication each time the system boots up, it adds a new registry entry. The malware also has the ability to delete the copy and corresponding registry entry.

Advertisement

The C&C server provides a list of browser window titles and a URL list. The malware enumerates each of the victim's windows to determine the title of each, then compares the titles against its list of window titles. If a window title matches one on the list – for example, a title belonging to a window of the *Internet Explorer* browser – the malware selects a URL randomly from its URL list and opens it in a new browser window. All of the URLs in the list direct to advertising sites controlled by the attackers.

Downloader

The data that comes from the C&C server includes a URL that is used by the malware to download a new file. The downloaded file varies – one example is W32/Oderoor, a notorious backdoor trojan.

Worm

In terms of hard-coded blocks, this is the only one that handles the messages transferred by the hidden window. In fact, this block only handles the WM_DEVICECHANGE message. The resource block provides an autorun.inf configuration file. By intercepting the WM_DEVICECHANGE message, the malware is able to drop both a copy of itself and an autorun.inf file onto the removable drive.

OVERVIEW

The malware retrieves a message from the message queue and then dispatches it to the hidden window. The message will be transferred to the MODULE_BLOCK chain and handled by fMessage. The malware sets a flag to control the communication with the C&C server. Whether it communicates with the server or not, it calls each fMonitor in the MODULE_BLOCK chain. The process is repeated until the malware resets the exit flag.

CONCLUSION

Through analysis, we are able to understand this piece of malware, and in particular its flexible module-handling mechanism. By adding or removing module blocks, the malware can adjust functionalities at will. Referring to the module block handling mechanism, the malware can also update its resource block handling functions in the future, although it has had a handful of action and command control IDs.

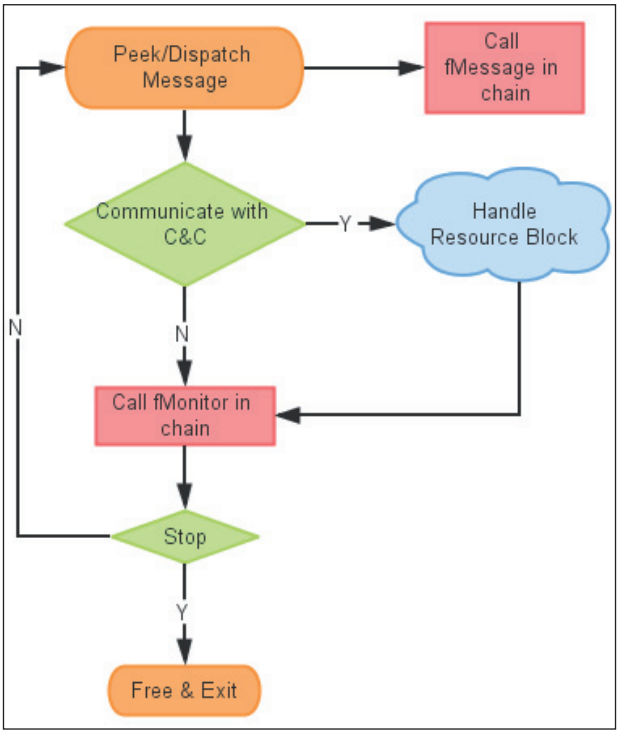

Figure 4: General work flow.

REFERENCES

[1] Coogan, P. The Mariposa / Butterfly Bot Kit. http://www.symantec.com/connect/blogs/mariposa-butterfly-bot-kit.

[2] Krejdl, M. Lazy Friday? Maybe next time. https://blog.avast.com/2012/04/06/lazy-friday-maybe-next-time/.

[3] Corrons, L. Shedding some light on Mariposa. http://pandalabs.pandasecurity.com/shedding-some-light-on-mariposa/.

Editor: Martijn Grooten
Chief of Operations: John Hawes
Security Test Engineers: Scott James, Tony Oliveira
Sales Executive: Allison Sketchley
Editorial Assistant: Helen Martin
Perl Developer: Tom Gracey
Consultant Technical Editors: Dr Morton Swimmer, Ian Whalley

Tel: +44 (0)1235 555139. Fax: +44 (0)1865 543153
Email: editorial@virusbtn.com
Web: http://www.virusbtn.com/