SEPTEMBER 2010

virus BULLETIN

Fighting malware and spam

CONTENTS

ISSN 1749-7027

IN THIS ISSUE

EYE OPENER

'Why doesn't Windows tell me when a very important signature has been tampered with?' asks Roel Schouwenberg.
page 2

DUBIOUS LNKS

LNK files are everywhere in Windows – so ubiquitous that they are rarely even recognized for what they are. While, LNK files do not generally pose a direct threat, there are the LNK files produced by W32/Stuxnet, which allow the execution of arbitrary code without the need for any user interaction. Peter Ferrie has the details.
page 4

VBSPAM CERTIFICATION

A new spam feed and an expansion of the ham corpus ensured that anti-spam products in this month's test were tested to their full abilities. Martijn Grooten has the results.
page 22

'Why doesn't Windows tell me when that very important signature has been tampered with?'

Roel Schowenberg, Kaspersky Lab

BREAKING THE HABIT

It may seem like an age ago but it was only in July that the world was made aware of the W32/Stuxnet malware. In a nutshell, Stuxnet is an extremely sophisticated worm that targets SCADA environments while exploiting a zero-day vulnerability in all recent versions of *Microsoft*'s *Windows* operating system. To top it all off, the attacks appeared to target Iranian systems, with by far the majority of incident reports coming from Iran. All of a sudden, the most off-the-wall conspiracy theories began to seem plausible.

Stuxnet, much akin to the *Google* Aurora attack, is playing a crucial role in a new sense of user awareness that seems to be developing this year. Aurora and Stuxnet are tangible cases for different kinds of cyber-espionage. These ready-made examples will certainly help to make it clear to the people who aren't being attacked – or perhaps who aren't aware they're being attacked – that they need proper protection.

In our industry, we tend to be sceptical about user education – and rightfully so. And while it's definitely possible to put up shields against Aurora-type attacks, I'm extremely doubtful that this is the case with an attack of Stuxnet's class. Let's face it, with the exception of exfiltration and botnet infrastructure, it's hard to see where the Stuxnet authors could have done better. There are many lessons to be learned from Stuxnet, but there's one which clearly stands out. There's an extremely broken model of trust.

With the huge volume of malware we've been seeing in the last couple of years, the anti-malware industry is relying more and more on automation. That our current automation is less than perfect is something I pointed out a year ago in reference to W32/Induc.A (see *VB*, September 2009, p.2). W32/Induc basically infects the Delphi compiler so that any file created with it contains the virus. What we ended up with were many different applications that had contained the virus for quite a long while. A number of these applications were even digitally signed.

Which brings us back to Stuxnet. The Stuxnet authors stole *VeriSign*-issued certificates from two reputable companies – *RealTek* and *JMicron*. That's a double attack against reputation. Firstly, it's no easy task to obtain a certificate from *VeriSign*. Secondly, there's a long history of trust in the files originating from these companies.

Certificate-stealing malware is far from new. The Zeus trojan has been doing it since 2006. The malware authors have never needed to use those certificates over the years but that is slowly changing. Stuxnet proves this.

Does this mean we must completely rethink whitelisting? No, but it will burden us with having to contact companies directly and whitelisting by the hash of files rather than the hash of digital signatures.

Even beyond Stuxnet, there are other certificate-related issues to worry about. At the beginning of August this year, there was a report from our friends at *Trend Micro* that a variant of Zeus was using a *Kaspersky Lab* certificate. After the Stuxnet news, it certainly received a lot of attention. But was it really worth the attention? The creator of this particular variant had simply copied a digital signature belonging to one of *Kaspersky Lab*'s tools and pasted it into his Zeus variant.

Now this is where it gets really confusing. The security community places enormous value on digital signatures. *Microsoft Windows*, for instance, will tell you when a valid signature has been found in a file and who that certificate belongs to. It will ask you if you trust that particular publisher. Why, then, doesn't *Windows* tell me that someone has tampered with that very important signature? *Windows* will generally treat a file with a tampered or corrupted signature as if it weren't signed in the first place and will not warn the user in any way. That's an extremely broken model of trust

The issue I'm describing is far from new. But if Aurora can serve as an eye-opener to Fortune 500 companies, making them realise that they really shouldn't have been running *Internet Explorer 6* in 2009, then let's have Stuxnet serve indirectly as an eye-opener to *Microsoft*, making the company realize that it shouldn't allow execution of files that have tampered signatures.

NEWS

ISP IN HOT WATER WITH ICO

The UK's largest ISP *TalkTalk* has been rapped by the Information Commissioner's Office (ICO) over its covert trials of a new anti-malware system. The initial phase of *TalkTalk*'s new security measures involved logging every URL visited by each of its customers, then visiting each web page to scan for threats. Master blacklists and whitelists were then compiled from the information gathered. When the system is fully operational (which is expected to be later in the year and will be on an opt-in basis) the anti-malware service will use the blacklists to prevent its users from visiting malicious web pages.

TalkTalk began its data gathering in July this year, but failed to notify its users that it would be logging their web-browsing movements in this way. It was this lack of communication that provoked the ire of the ICO. In a letter to *TalkTalk* the Information Commissioner Christopher Graham wrote: 'I am concerned that the trial was undertaken without first informing those affected that it was taking place... You will be aware that compliance with one of the underlying principles of data protection legislation relies on providing individuals with information about how and why their information will be used.'

Just a couple of years ago UK telecoms company *BT* found itself in hot water after undertaking a test with *Phorm* – a company which used deep packet inspection at the ISP level to gather information on subscribers' web-surfing habits and subsequently deliver tailored advertising content. Although *Phorm* claimed that it had removed any personally identifiable information from the content it gathered, there was widespread outrage that the test had gone ahead without the knowledge or consent of *BT*'s user-base. Indeed, the two companies narrowly avoided criminal investigation after campaigners compiled a dossier of evidence against the two companies and presented it to the City of London Police.

TalkTalk has claimed that its technology and the trials it has undertaken comply with privacy laws – the ICO has requested documents to support these claims. David Evans of the ICO will give a presentation on data protection, privacy and security, outlining the ICO's view, at the VB Seminar later this year (in central London, 25 November 2010 – for details see http://www.virusbtn.com/seminar/).

RAP RAP?

Symantec's latest headline-hitting (possibly for the wrong reasons) publicity drive involves a collaboration between the security firm and US rap artist Snoop Dogg who, together, are running a competition to find the best cybercrime-themed rap. *VB* challenges its readers to compose a rap about the *VB* RAP testing – budding rappers can upload their entries at http://www.hackiswack.com/.

Prevalence Table – July 2010[1]

Malware	Type	%
Conficker/Downadup	Worm	10.70%
Autorun	Worm	8.24%
FakeAlert/Renos	Rogue AV	6.97%
Agent	Trojan	5.76%
Heuristic/generic	Virus/worm	4.03%
VB	Worm	3.74%
Ircbot	Worm	3.64%
OnlineGames	Trojan	3.49%
Adware-misc	Adware	3.27%
Downloader-misc	Trojan	3.12%
Mdrop	Trojan	2.98%
Injector	Trojan	2.60%
Crypt	Trojan	2.32%
Zbot	Trojan	2.28%
Virut	Virus	2.14%
Heuristic/generic	Trojan	2.11%
Delf	Trojan	1.86%
Alureon	Trojan	1.79%
AutoIt	Trojan	1.72%
Exploit-misc	Exploit	1.58%
Iframe	Exploit	1.51%
Hotbar	Adware	1.36%
Virtumonde/Vundo	Trojan	1.28%
Small	Trojan	1.26%
Sality	Virus	1.23%
Potentially Unwanted-misc	PU	1.08%
Tanatos	Worm	0.96%
Bifrose/Pakes	Trojan	0.77%
Hiloti	Trojan	0.77%
Dropper-misc	Trojan	0.75%
Redir	Trojan	0.74%
FakeAV-Misc	Rogue AV	0.73%
Others[1]		13.23%
Total		100.00%

[1]Figures compiled from desktop-level detections.

[2]Readers are reminded that a complete listing is posted at http://www.virusbtn.com/Prevalence/.

MALWARE ANALYSIS 1

THE MISSING LNK

Peter Ferrie
Microsoft, USA

LNK files are everywhere in *Windows*, so ubiquitous that they are rarely even recognized for what they are: complex structures containing pointers to Portable Executable files and, ultimately, executable code.

Some of the icons that appear in the Control Panel are visible because of LNK files. Many of the entries in the Start Menu and on the Desktop are LNK files. In most cases, the LNK references a file, and specifies an icon to display. When an application is used to view the LNK file, such as browsing a folder using *Windows Explorer*, the *Windows* shell parses the format and determines what to display. LNKs are not limited to just files, though. They can be shortcuts to drives such as a shared network location or a floppy disk (as used by the 'Send To' menu, for example). The 'Recent File List' in *Microsoft Office 2007* applications is composed of LNK files.

Overall, LNK files do not pose a direct threat. Of course, some LNK files can point to malicious executables that run when the LNK file is clicked, and some LNK files can point to harmless files and yet still perform malicious actions (such as when the command prompt is executed, but given the instructions to delete some files). Some LNK files can themselves be malicious by virtue of their contents (such as the self-executing LNK file virus from several years ago, where the LNK file carried an actual Portable Executable file, and executed it in a rather roundabout fashion). Then there are the LNK files produced by W32/Stuxnet, which allow the execution of arbitrary code without the need for any user interaction (other than browsing to a folder that contains such a file, with some further clarification below).

LNKS TO THE PAST

The LNK file format has existed since the days of *Windows 95*, but it was not documented publicly by *Microsoft* until 2009, and even then it was incomplete and interrupted. After the Stuxnet malware caused quite some interest in the LNK file format, the documentation was temporarily removed from the *Microsoft* website, and then re-released as 'new'. However, the only difference between the two versions is some formatting, and the dates of referenced external files. The two versions are equally useless as far as determining how to parse the LNK files produced by Stuxnet goes, because the section that is being exploited is not documented in either version.

So, in order to understand what Stuxnet did with LNK files, we first have to understand what is inside a LNK file.

LNK LAYER

A LNK file begins with a 0x4c-byte-long header. The first field of the header is the 'HeaderSize', which specifies the size of the header, including itself. It must contain the value 0x4c. The next field is the 'LinkCLSID' (though it resolves to a registry key called 'Shortcut'). The CLSID must be '{00021401-0000-0000-C000-000000000046}'. That's the extent of the constant bytes for the header, as far as Stuxnet-style files go.

There is a 'LinkFlags' field, which is supposed to specify information about the type of link and the presence of optional structures. Unfortunately, all but two of the flags can be set arbitrarily, despite the fact that the corresponding structures are missing from the file. Only one bit is required to be set, and that is the HasLinkTargetIDList. When the bit is set, it specifies that the file is saved with an item ID list, and the corresponding 'LinkTargetIDList' structure follows the ShellLinkHeader structure immediately.

One important-sounding bit in the LinkFlags field is the 'IsUnicode' bit. This is supposed to be set if a file contains Unicode strings, and the documentation states that the bit 'SHOULD' (in upper case in the documentation) be set. However, the 'should' apparently refers to the fact that LNK files should be in Unicode format, as opposed to ANSI format. Unfortunately, the bit is entirely useless because a file can have the bit set and still be in ANSI format. Alternatively, the bit can be clear and yet the file can still be in Unicode format. The way to determine the format of the strings will be described below.

The two bits that cannot be set arbitrarily are 'HasExpString' and 'HasDarwinID'. If either bit is set, then the corresponding structure must be present in the file, but the presence of either of them prevents the use of the structure that the Stuxnet LNK files require.

CUFF LNKS

The LinkTargetIDList structure is an 'IDList' structure which contains a collection of 'ItemID' structures. In the case of a Stuxnet LNK file, there are three ItemID structures. Each of these contains a size field, a type field, and a data field.

The first ItemID structure in a Stuxnet LNK file contains the type and CLSID for the 'My Computer' or 'Computer' element (the specific name depends on the version of *Windows*, but the name is not relevant). The type field is two bytes long, but only one of those bytes is checked by *Windows*. The type must be 0x1f, and the CLSID must be '{20D04FE0-3AEA-1069-A2D8-08002B30309D}'. Although the value in the size field is most commonly 0x14, the size of the structure is not a constant and the size of the

structure can be increased in a hand-crafted file (though Stuxnet does not do this), by adding additional data after the CLSID field. Then the value in the size field can be increased accordingly. This alteration is known to break at least one publicly available scanning tool. I attempted to contact the author of the tool regarding this attack, but received no reply, so the tool remains unchanged.

The second ItemID structure in a Stuxnet LNK file contains the type and CLSID for the 'Control Panel' or 'All Control Panel Items' element (again, the specific name depends on the version of *Windows*, but is not relevant). As before, the type field is two bytes long, but only one of those bytes is checked by *Windows*. The type must be 0x2e, and the CLSID must be '{21EC2020-3AEA-1069-A2DD-08002B30309D}'. Again, the size of the structure is not constant. The presence of the Control Panel CLSID leads us to the flaw that Stuxnet exploits.

The third ItemID structure in a Stuxnet LNK file contains an undocumented (to the point that I don't even know the correct name) 'Control Panel applet' structure, which contains a size field, an icon index, some truly 'spare' fields, as well as a path to the file that holds the icon to display. This is where the magic happens. When the Control Panel is displayed, *Windows* queries the corresponding LNK files for the icons to display. Based on the behaviour of the patch (see below), this was intended to apply only to registered Control Panel applets (that is, the registry key 'Software\Microsoft\Windows\CurrentVersion\Control Panel\Cpls', under either 'HKCU' or 'HKLM', contains subkeys or values that name them), and ideally they would have been placed in the '%windir%\system32' directory. Unfortunately, neither of these conditions was enforced, allowing LNK files to access files in arbitrary locations in any context in which an icon would be displayed. This is why we can reproduce the problem by simply 'browsing a folder using *Windows Explorer*'. The referenced files are *Windows* DLLs, usually with a suffix of '.CPL', and *Windows* will load them in order to retrieve the address of an exported function which is used to display the icon. Of course, in the case of Stuxnet, control is gained when *Windows* loads the file, and thus no exported function is necessary. Furthermore, no warning is given when the named file is loaded.

To complicate matters further, the Control Panel applet structure comes in two forms, as noted above: Unicode and ANSI. The correct format can be determined by examining the values in particular locations within the structure. For the Unicode format, the six bytes beginning at offset 8 must be '00 00 00 00 00 6A'. Ten bytes later is the path in Unicode characters. This is the only format that Stuxnet LNK files use. For the ANSI format, the four bytes beginning at offset 8 can have almost any value (see below), including all zeroes, and which, if the following two bytes are not checked, might lead someone to assume that the Unicode format is in use. For the ANSI format, four bytes later is the path in ANSI characters.

D-LNKS

As far as the path goes, the typical case is to have a drive letter, directory and filename. Of course, this ties the file to a specific drive configuration. That would be fine if the drive letter were constant, but in the case of Stuxnet, there is no guarantee of that. Specifically, Stuxnet places LNK files on USB removable storage media (among other places). Since the drive letter is assigned dynamically, because the order of device insertion can vary, Stuxnet needed a way to refer to the file regardless of the drive letter. This was achieved by querying the registry for the hardware ID. The result looks like '\\.\STORAGE#RemovableMedia#7&xxxxxxxx&0&RM#{53f5630d-b6bf-11d0-94f2-00a0c91efb8b}\<file>', where the 'x's are a unique value specific to the device, and the CLSID is the Device Class GUID for a volume.

Furthermore, the path does not need to look like either of these examples. The path can be in UNC form, allowing a file to be loaded from a remote location, such as a network share within a corporation. However, since the WebDAV redirector is also running by default, the location can be very remote – that is, anywhere on the Internet. This variation is used by the exploit module in the Metasploit framework.

WEAK LNKS

One virus writer posted a message to a forum claiming that anti-virus researchers 'read docs instead of code' and that his post contained the '*real* format for LNK files'. While his description was mostly correct (even including the two bits that cannot be set, though he did not name them and did not describe why they cannot be used), and he did find something that I did not know (the use of a relative path without a drive letter was not supported by my detection at the time), he did manage to get two things wrong. The two things that were wrong result in essentially 'random' execution of the LNK files. Perhaps he didn't read the code carefully enough. Or perhaps that was the intended result.

The first wrong thing relates to the 'unused' bytes that live in the field before the path to the file. Two of those bytes are actually a separate field that is supposed to contain the size of the path. This is used as a relative pointer to the description text, and it is used by *Windows*. If the field contains a value that is too large (that is, beyond the end of the LNK file), then an exception might occur while *Windows* attempts to copy the description string. In that case, the LNK file will not be parsed any further. That might sound potentially interesting to an attacker,

since obviously there is no check that the pointer is valid. However, what lies beyond the end of the file is essentially random data, and the string copy operation has a limited length, so it cannot be exploited. The value is not supposed to be zero, either, since that would cause the filename to become the 'description', but such a situation causes no ill effects.

The second thing the virus writer got wrong relates to the non-terminated string. If the termination bytes are removed, then when the LNK file is loaded, whatever happened to be in memory at the corresponding location will be used instead. The bytes there might not be zero, resulting in a string that appears to be longer than it was supposed to be. Again, that might sound potentially interesting to an attacker, but as before, what lies beyond the end of the file is essentially random data, and the string copy operation has a limited length, so it cannot be exploited. A further result is that the check for the existence of the file is likely to fail, since those random bytes will become part of the filename that is examined.

STRONG LNKS

Microsoft patched the behaviour to check for CPL files in several locations, and a list is made of the files found for use later. The locations are the 'MMCPL' key in 'control.ini' (which is redirected to the registry, but the location is not constant), '%windir%\system32' and 'Software\Microsoft\Windows\CurrentVersion\Control Panel\CPLs' under both 'HKLM' and 'HKCU'. Any CPL file that is found is excluded if it is marked as 'don't load' according to its registry key.

Once the list has been made, the file referenced by the LNK file is checked against each entry in the list. If the file is not in the list, then *Windows* makes a change to the loading method, to force the use of a default system icon instead of the requested icon. This also prevents the referenced file from being loaded. Internally, the path is converted to the form '<path>,<icon>,<description>', where <icon> is zero in Stuxnet LNK files. However, if the file is not found in the list, then the path is converted to '<path>,-1,<description>'. Prior to creating the list, the original path is checked for the presence of a comma, in order to prevent a path that has the internal form prior to conversion, since that could have been used to defeat the other checks in the patch.

CONCLUSION

So now we know what the Stuxnet LNK files do. As for what else Stuxnet does, the details would fill a conference paper (or two!).

MALWARE ANALYSIS 2

INJECTION AS A WAY OF LIFE

Raul Alvarez
Fortinet, USA

Memory-residency is employed by malware to ensure that it is always active on the system. Techniques have been tried and tested; the good old DOS infector used Terminate and Stay Resident – TSR (using the infamous INT 21h function 31h) – and another well-known technique is code injection. Injecting code into a process is not a new technology, but it is still used by most prevalent malware today.

The main idea behind code injection is that the malware embeds itself into a running process to maintain residency. Well, of course we already know that. Behavioural analysis can tell us that a certain application has been infected; we use different tools to determine if a thread has been injected into a certain process. And lots of malware analysis online will tell us that a given piece of malware injects its code into a running process. But little has been said about the actual code-by-code steps that malware uses to inject its code.

This article will dissect two examples of recent prevalent malware and show how they inject their code into a running process. We will start with a variant of Virut, detected by *Fortinet* as W32/Virut.CE, which uses Zw*** APIs to implement code injection. Then we will explain how a variant of OnlineGames embeds its code into the Explorer.exe process.

PART I: VIRUT, VIRUT AND VIRUT

Virut's code injection starts by modifying the access token's privilege; the access token contains the security information for a logon session. Every time a user logs on, the system generates an access token which is also used by every process and application executed by the current user.

Virut uses the ZwOpenProcessToken API in order to get the handle for the access token of the user. After acquiring the handle of the token, Virut resolves the address of the LookupPrivilegeValueA API by using the LoadLibrary and GetProcAddress APIs. Virut calls for the LookupPrivilegeValueA API to get the locally unique identifier (LUID) for SeDebugPrivilege, also known as SE_DEBUG_NAME; this is a privilege required for memory modification of a given process, which Virut needs to freely inject its code. This is immediately followed by a call to the ZwAdjustPrivilegesToken API, which adjusts the privilege of the access token based on the new LUID.

Setting the privilege of the access token to SeDebugPrivilege enables Virut to perform code injection with ease; the malware doesn't need to concern itself with any issue regarding the opening of a process, writing to it, hooking code in its shared memory space, creating threads and executing instructions. Once the privilege is set to the proper attributes, Virut proceeds to enumerate the running processes.

Browsing active processes

Virut is a polymorphic virus, and after decryption and resolving the necessary APIs we can see that most variants don't go far from their intended purpose.

A typical way to enumerate the active processes in a given system starts with a call to the CreateToolhelp32Snapshot API; Virut calls the CreateToolhelp32Snapshot API to get a snapshot of the system. Using this API, a piece of malware can get a snapshot of every module, thread, heap and process, all depending on the dwFlags parameter supplied to it; Virut uses TH32CS_SNAPPROCESS to include all processes in the system. The malware enumerates the processes one by one using a single call to the Process32First API and concurrent calls to the Process32Next API. These two APIs use the PROCESSENTRY32 structure generated by the CreateToolhelp32Snapshot API which was called earlier (see Figure 1).

While enumerating the list of processes, Virut intentionally skips the first four processes without even checking their names. Interestingly, most often, the Winlogon.exe process is the fifth on the list. Winlogon is the first process into which Virut injects its code; Winlogon is infected not by choice but for the simple reason that it is one of the first processes available for Virut infection.

```
00CD0594   6A 00            PUSH 0
00CD0596   6A 02            PUSH 2
00CD0598   FF95 C8223512    CALL DWORD PTR SS:[EBP+123522C8]   CreateToolhelp32Snapshot
00CD059E   B9 28010000      MOV ECX,128
00CD05A3   97               XCHG EAX,EDI
00CD05A4   2BE1             SUB ESP,ECX
00CD05A6   890C24           MOV DWORD PTR SS:[ESP],ECX
00CD05A9   54               PUSH ESP
00CD05AA   57               PUSH EDI
00CD05AB   FF95 18233512    CALL DWORD PTR SS:[EBP+12352318]   Process32First
00CD05B1   33F6             XOR ESI,ESI
00CD05B3   83A5 6C5C3512 00 AND DWORD PTR SS:[EBP+12355C6C],0
00CD05BA   54               PUSH ESP
00CD05BB   57               PUSH EDI
00CD05BC   FF95 1C233512    CALL DWORD PTR SS:[EBP+1235231C]   Process32Next
00CD05C2   85C0             TEST EAX,EAX
00CD05C4  74 6E             JE SHORT 00CD0634
00CD05C6   46               INC ESI
00CD05C7   83FE 04          CMP ESI,4                          skips the first 4 processes
00CD05CA  72 EE             JB SHORT 00CD05BA
00CD05CC   FF7424 08        PUSH DWORD PTR SS:[ESP+8]
00CD05D0   6A 00            PUSH 0
00CD05D2   6A 2A            PUSH 2A
00CD05D4   FF95 14233512    CALL DWORD PTR SS:[EBP+12352314]   OpenProcess
00CD05DA   85C0             TEST EAX,EAX
00CD05DC  74 DC             JE SHORT 00CD05BA
```

Figure 1: Code snippets on enumerating the active processes and the skipping of the first four processes.

The next logical step, after acquiring the handle of the process to infect, is to open it. Virut opens the process by calling the OpenProcess API with the CREATE_THREAD|VM_OPERATION|VM_WRITE access parameter; this enables the malware to create a thread in the given process and to write the codes to inject.

Mapping a section of memory

Before the code injection stage, Virut creates a section of memory named \BaseNamedObjects\houtVt; this contains the complete code to be injected into the process. This is evident on any process that has already been injected with Virut's code. *Process Explorer* or any tool that can show the events, keys, sections and other objects of a process can be used to determine if the process is already infected.

Since the section already exists, Virut calls the ZwMapViewOfSection API to map a copy of \BaseNamedObjects\houtVt to the current process that it is working on. The actual Virut code is copied to the process's memory space by mapping the section of memory. Mapping a section of memory is like sharing a DLL in a process's memory space, thereby giving Winlogon (or other process) access rights to the section. Any viable code within the \BaseNamedObjects\houtVt section can now be executed by any process that maps it; calling a function from within the section is just a matter of pointing it to the right memory address.

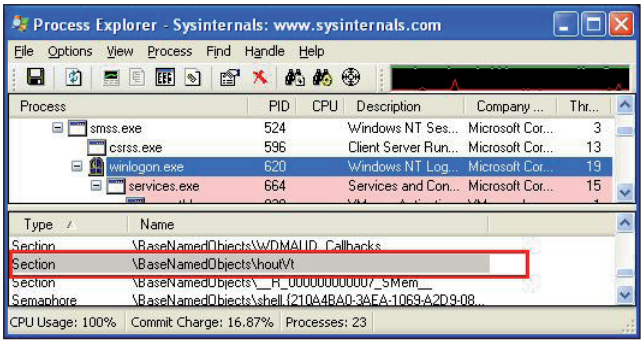


Figure 2: The mapped section named \BaseNamedObjects\houtVt in the Winlogon.exe process.

Hooking NTDLL.dll

Hooking is an old technique used by malware; old DOS viruses hooked INT functions to redirect calls to their code and new malware hooks DLL functions in a similar way. When a call to the hook function is performed, execution transfers to the malware code, which is executed, and then

control is transferred back to the original function routine; this is basically what happened to the hooked function.

Virut hooks some APIs from NTDLL, of a given process, simply by replacing the MOV EAX,yy instruction with a CALL xxxxxxxx, an address pointed to by the mapped \BaseNamedObjects\houtVt section. It uses the ZwProtectVirtualMemory to change the protection mode of NTDLL attached to the process to PAGE_READWRITE mode then proceeds to hook it by writing the CALL instruction using ZwWriteVirtualMemory. The PAGE_READWRITE mode ensures that the shared NTDLL can be written to by a call to ZwWriteVirtualMemory.

Virut hooks the following APIs:

- ZwCreateFile
- ZwOpenFile
- ZwCreateProcess
- ZwCreateProcessEx
- ZwQueryInformationProcess

By hooking the APIs above, Virut's code becomes available whenever a file is read, opened or created, and whenever a process is opened, created or queried.

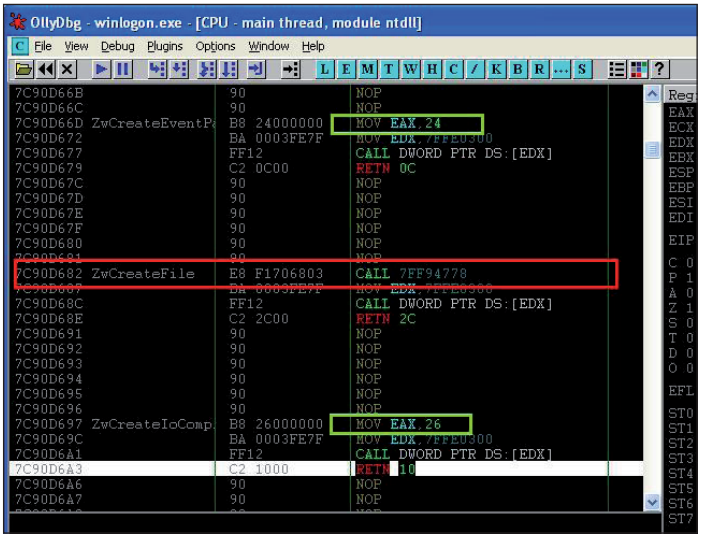


Figure 3: The hooked NTDLL.dll; the green boxes are the normal codes and the red box is the hooked ZwCreateFile API; the MOV instruction was replaced by a call to the mapped section.

Running the thread

Once everything is set – privileges have been set up, a process has been selected to infect, a section of memory has been mapped, and DLL hooked – the last thing for Virut to do is to execute a thread remotely.

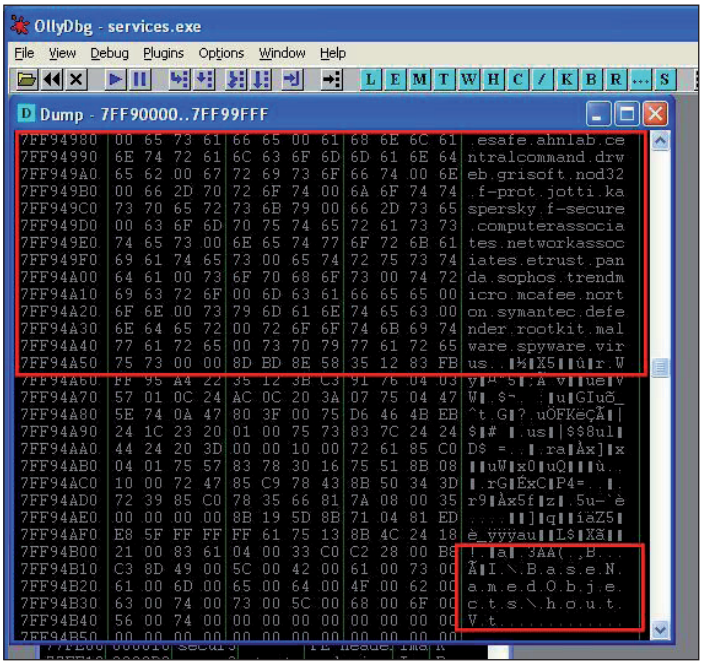


Figure 4: Strings found in services.exe's process indicative of Virut's mapped section.

Virut creates a remote thread using a call to CreateRemoteThread, with dwCreationFlags equal to 0. It executes the thread immediately. When a remote thread is created, it can be suspended or, in this case, executed immediately. Virut executes the thread as soon as it is created to speed up the infection process. When all is well, Virut relinquishes its control to the process and proceeds to look for a new process to inject its code into. As we now know, Virut doesn't only infect the Winlogon.exe process; it keeps looking for more processes to inject code into.

As discussed earlier, we can easily check if a process is infected by looking for the presence of the \BaseNamedObjects\houtVt section. To be certain, we can browse the process's memory and look for a sign that Virut is really there. Most often, Virut's favourite location is 7FF90000h and the size is 0A000h; however, some processes use that location, so Virut uses the next location on the block, 7FFA0000h, with the same virus size. Virut's code within the process's memory is not encrypted, thereby giving us the strings to look for. We can see strings like AV company names, the name of the section, resolved names of APIs, IRC-related strings, and registry key strings.

Virut's method of code injection is fairly common amongst malware. That being said, we will now look at another method of injecting code.

PART II: ONLINE GAMING

The next piece of malware we will look at is a variant of OnlineGames. Most malware families have their own style of decryption routine, and the same is true when it comes to the process of code injection. We have already noted that a variant of Virut skips the first four processes and injects its code into Winlogon.exe and succeeding processes after that. In this variant of OnlineGames, Explorer.exe is the sole target.

We will discuss some commonalities of Virut and OnlineGames when selecting the process for injection, how codes are copied to the process's memory space and what the remote code looks like before it is executed in the process.

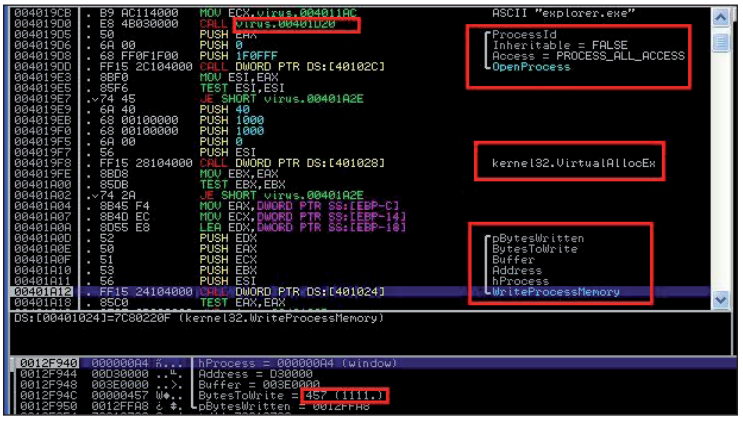


Figure 5: Code snippet showing the call to the OpenProcess, VirtualAllocEx and WriteProcessMemory APIs. It also shows a certain call to a memory location, 00401D20, where the pID searching can be found. Lastly, it shows where the length of the codes, 457h(1111), is used.

Choosing explorer.exe

Like Virut, OnlineGames uses the CreateToolhelp32Snapshot to enumerate the processes active in the system – using TH32CS_SNAPPROCESS as the dwFlags parameter. Although the malware knows what process to infect, it still uses the same pair of Process32First and Process32Next APIs to locate the pID (process ID) of Explorer.exe.

Interestingly enough, the malware has a longer code routine just to copy a string (process name) to a memory location; it also has a longer code routine comparing the process name to look for the 'Explorer.exe' string. Instead of copying the string using a single instruction, the malware copies it, character by character, to the memory. To compare the string, the malware first counts the number of characters of the name of the given process and compares it to the length of the 'Explorer.exe' string. If the size of the two strings matches, then it proceeds to check each character of both strings. After a successful attempt at getting the right process name, 'Explorer.exe', the malware captures the pID of the process.

The pID of Explorer.exe is now used by OpenProcess, with an access parameter of PROCESS_ALL_ACCESS – all possible access rights.

Writing codes to process

Virut's method of putting its codes into memory is by mapping the entire \BaseNamedObjects\houtVt section and hooking NTDLL.dll APIs linking to the mapped section. In comparison, OnlineGames uses the WriteProcessMemory API to write codes into the Explorer.exe process. But in this respect, the code written to the process's memory space is not the whole virus code yet.

Before OnlineGames writes some of its code to the process, it uses the VirtualAllocEx API to reserve some memory space from the process; the resulting value is the base address where OnlineGames can write to. It then proceeds to write 457h(1111) bytes of code – which, of course, is not the whole virus code.

Intercepting the remote thread

The WriteProcessMemory API is only called once within the malware body; it only writes 457h(1111) bytes of code. We can only assume that there should be more to it than just writing that small piece of code. Does OnlineGames use the same technique of mapping a section of memory to the

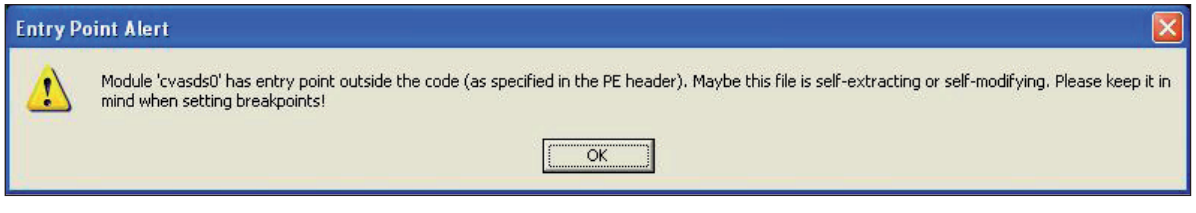


Figure 6: Message displayed when CreateRemoteThread API from OnlineGames was executed.

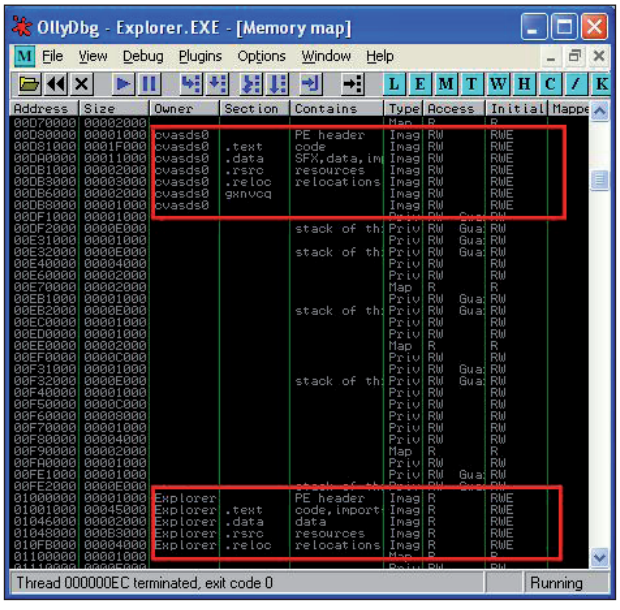


Figure 7: Memory map of the 'Explorer.exe' process within OllyDbg. It shows the map view of 'Explorer.exe' and the new file 'cvasdds0'.

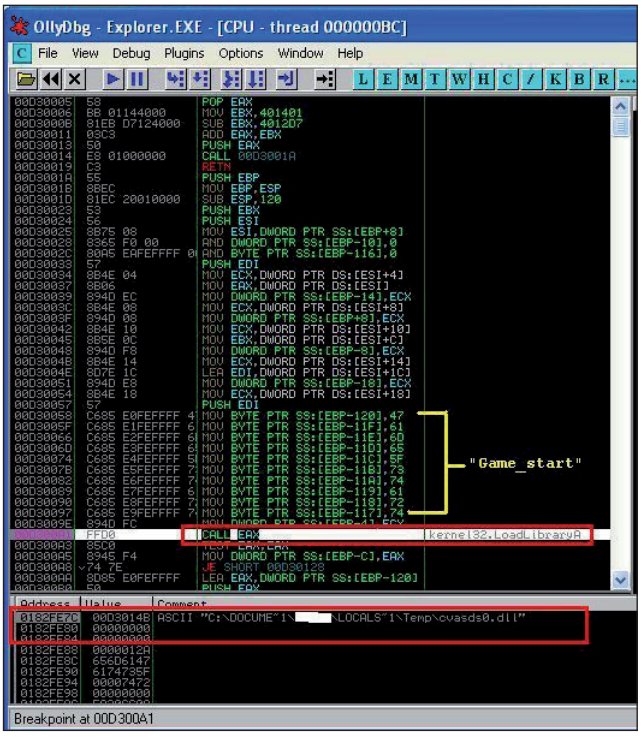


Figure 8: Code snippet of 457h bytes of code copied to the memory space of Explorer.exe, showing the call to the LoadLibraryA API and the string 'Game_start'.

running process as Virut? The answer is no, OnlineGames doesn't use memory mapping and it doesn't hook any functions in NTDLL, or any DLL for that matter. But how can OnlineGames copy the whole malware code to Explorer.exe? The answer lies in the 457h bytes of memory the malware wrote earlier.

The only logical way to look for the answer is to intercept the execution of the 457h mystery bytes. A remote thread is created when OnlineGames uses the CreateRemoteThread API; it points to the base address, the starting address of the 457h bytes of code taken from the call to VirtualAllocEx API earlier. Once the thread is created and Explorer.exe is within a debugger, such as *OllyDbg*, we will see a message box displaying *'Module "cvasds0" has entry point outside the code (as specified in the PE header). Maybe this file is self-extracting or self-modifying. Please keep it in mind when setting breakpoints!'* (see Figure 6). Note that the message will only show when Explorer.exe is within a debugger context.

Knowing that a file named 'cvasds0' is being accessed by Explorer.exe, it is safe to say that it is the same malware file that we are looking for. We haven't intercepted the code yet, so we need to go back and execute the CreateRemoteThread API; this time we are in intercept mode. Figure 8 shows a snippet of the intercepted code, the 457h bytes of code copied earlier using the WriteProcessMemory API.

The 457h bytes of code is responsible for loading 'cvasds0' into the Explorer.exe process; it calls the LoadLibraryA API to load the file, actually a DLL, that can be found at the 'c:\DOCUME~1\[varies]\LOCALS~1\Temp\' folder. 'cvasds0.dll' is a DLL file dropped by OnlineGames at an earlier stage of the malware's execution. The 457h bytes of code also contains the string 'Game_start', which is encoded character by character.

CONCLUSION

We have seen two different pieces of malware, each demonstrating different skills in performing code injection. They both start off by using the basic techniques of enumerating, searching and opening a process. Then, they each go a different way when they start preparing the code to be injected. Virut has chosen to map its code to the process and hook NTDLL, while OnlineGames has chosen to inject a small amount of code into Explorer.exe and let it load its complete code in a library form. There are several more tricks for code injection out there; we will encounter them in one way or another, yet they will always have one thing in common – the process.

MALWARE ANALYSIS 3

CHIM CHYMINE: A LUCKY SWEEP?

David Harley
ESET

The class of malware that exploits Autorun/Autoplay as an infection vector has been an irritating fact of life for a good while. The malware known as Win32/Stuxnet could be (and indeed has been [1]) described as a worm of a different colour. It can propagate making use of a zero-day vulnerability [2] listed by CVE as CVE-2010-2568 [3]. In a nutshell – or rather the *Windows* Shell – *Windows* can be tricked into executing malicious code presented in a specially crafted shortcut (.LNK) file, linking in turn to a malicious DLL, allowing compromise of a system even where Autoplay is disabled. (It should not be forgotten that USB devices aren't the only potential entry point: network and webDAV shares are also an issue.)

MISSING LINKS

There's already plenty of detailed information available on both the vulnerability and the very interesting Win32/Stuxnet family, so I won't go much beyond the minimum background in this article. However, apart from generating detection for Stuxnet, *ESET* also started to detect its approach heuristically, as LNK/Autostart.A, and subsequently as LNK/Exploit.CVE-2010-2568. It was, after all, reasonable to suppose that as the proof-of-concept code gained currency, other malware families would adopt the same trick. Sure enough, our telemetry systems soon picked up some interesting vibes.

THE OTHER VB

As early as 20 July, our lab was seeing several Autorun worms written in Visual Basic and experimenting with LNK files. However, by 23 July things were getting really interesting. We had identified a new family exploiting the still unpatched vulnerability in order to spread by code execution through malicious LNK files. This was promptly christened Win32/TrojanDownloader.Chymine.A. At the present time, this threat is used to download and install a keystroke logger which we detect as the Win32/Spy.Agent.NSO trojan.

CHINA CHYMINE IN

The server used to deliver the components used in this attack is presently located in the US, hosted by the 'Managed Solutions group'. According to RWHOIS data from the hosting organization, the server IP address was assigned to a customer in China on 22 July. The DLL downloaded from here contains a number of strings in Chinese, translated here courtesy of *Google Translate*:

Comment:	'Fire Personal Firewall, building a fun-filled safe network for you'
File description:	Fair Personal Firewall, for you to create a safety net is full of fun.
Company name:	Fair Safety Laboratory
Product name:	Fair Personal Firewall

At the time of writing, neither Chymine.A nor any of the related files seem to be generating any malicious LNK files themselves. To date, we've only found LNKs exploiting the latest vulnerability and pointing to the downloader, suggesting to us that since Chymine.A doesn't spread by itself, there must be something (or someone) else 'helping' it along [4].

THIS WILL (AUTO)RUN AND RUN

Even while the lab team was in the process of sharing information about this new threat with other researchers, they observed a known threat which had been refurbished to include the CVE-2010-2568 exploit as a new propagation vector. Win32/Autorun.VB.RP looks very much like an updated version of the malware written in Visual Basic and described on 21 July by Adrian de Beaupre [5]. This class of threat hides folders in the root directory of any drive to which it has Write access, creates LNKs with the same name as the hidden folders, and drops autorun.inf, EXE and SRC files. It differs from the Internet Storm Center description, however, in that it *does* actually produce new LNK files exploiting the CVE-2010-2568 vulnerability to facilitate its own spreading; it doesn't simply rely on Autorun or wait for the victim to click on a malicious but uncrafted LNK. It now seems to download and install additional components on infected machines. The LNKs making use of the CVE-2010-2568 exploit use the following naming convention: z<two letters>.lnk (for example 'zTa.lnk').

There's not much doubt about which of the present crop of malware based on the original LNK vulnerability is the most novel and interesting. Win32/Stuxnet has two major points of interest.

- One, of course, is the targeting of *Siemens* control software on SCADA sites, injecting modules SystemRoot\inf\oem7A.PNF and SystemRoot\inf\oem7A.PNF into the address spaces of CCprojectMgr.exe and S7tftopx.exe processes using the mrxcls.sys driver, and reading configuration information from the registry key 'HKLM\System\CurrentControlSet\Services\MRxCLS. (This may

account for some of the interesting distribution patterns that have been noted by some sources [1, 6].)

- The second point of interest is Stuxnet's use of legitimate digital certificates to sign its device driver. These were stolen from *Realtek* and *JMicron Technology Corp* [7] and were subsequently revoked by *VeriSign* in order to prevent further misuse [8, 9, 10].

THE EARLY BIRD LAYS THE WORM?

However, the newer malware we're seeing is far less sophisticated than Stuxnet, and suggests bottom feeders seizing on vulnerabilities flagged by others, and hijacking exploitative techniques developed by the early birds. This is interesting in its own way (not least for the speed with which it has appeared). We expect to see plenty more worm cast [11] on the beach before (and after) *Microsoft*'s update [12] appears on the horizon.

This article synthesizes the research and thoughts of many people, not only within *ESET* but in the anti-malware community at large, and too many to mention individually. However, particular thanks are due to Richard Baranyi, Peter Košinár, Juraj Malcho, Pierre-Marc Bureau and Aleksandr Matrosov for sharing their research data and insights.

REFERENCES

[1] http://blog.eset.com/2010/07/17/windows-shellshocked-or-why-win32stuxnet-sux.

[2] http://www.microsoft.com/technet/security/advisory/2286198.mspx.

[3] http://www.cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2010-2568.

[4] http://blog.eset.com/2010/07/22/new-malicious-lnks-here-we-go.

[5] http://isc.sans.edu/diary.html?storyid=9229.

[6] http://blogs.technet.com/b/mmpc/archive/2010/07/16/the-stuxnet-sting.aspx.

[7] http://blog.eset.com/2010/07/19/win32stuxnet-signed-binaries.

[8] https://blogs.verisign.com/ssl-blog/2010/07/code_signing_certificates_used.php.

[9] http://blog.eset.com/2010/07/22/why-steal-digital-certificates.

[10] Goretsky, A. http://blog.eset.com/2010/07/22/a-few-facts-about-win32stuxnet-cve-2010-2568.

[11] http://en.wikipedia.org/wiki/Worm_cast.

[12] http://www.microsoft.com/technet/security/advisory/2286198.mspx.

TECHNICAL FEATURE

ANTI-UNPACKER TRICKS – PART TWELVE

Peter Ferrie
Microsoft, USA

New anti-unpacking tricks continue to be developed as older ones are constantly being defeated. This series of articles describes some tricks that might become common in the future, along with some countermeasures [1–12].

In this article we look at some anti-unpacking tricks that are specific to a range of debuggers.

Unless stated otherwise, all of the techniques described here were discovered and developed by the author.

1. HIDETOOLZ-SPECIFIC

HideToolz is an application that can hide another process under user control. It uses a driver to perform some of its work. The driver searches blindly within the ntoskrnl KeAddSystemServiceTable() function code for a particular instruction using a particular register. This combination is only present in *Windows XP* and later versions, so its presence is an unsafe assumption. The driver hooks many functions.

When the ntoskrnl NtQueryInformationProcess() function is called, the hook calls the original ntoskrnl NtQueryInformationProcess() function, and then exits if an error occurs. Otherwise, the hook checks the ProcessInformationClass parameter. If the ProcessBasicInformation class is specified, and if the specified process ID is on the hidden list, then the hook replaces the process ID of the parent process with the process ID of Explorer.exe in the InheritedFromUniqueProcessId field. This could be considered a bug, since the true parent might not be *Explorer*. The proper behaviour would be to use the process ID of the parent process.

If the ProcessDebugPort class is specified, then the hook zeroes the debug port, but without checking the process handle. The correct behaviour would be to zero the port only if the current process is specified.

When the ntoskrnl NtQuerySystemInformation() function is called, the hook calls the original ntoskrnl NtQuerySystemInformation() function, and then exits if an error occurs. Otherwise, the hook checks the SystemInformationClass parameter. If the SystemProcessInformation class is specified, then for each entry in the hidden list, the hook replaces the process ID of the parent process with the process ID of Explorer.exe in the

InheritedFromUniqueProcessId field. Once again, this could be considered a bug, since the true parent might not be *Explorer*. The proper behaviour would be to use the process ID of the parent process. A separate option exists which, for each entry in the hidden list, zeroes the entry in the buffer unless the list is requested by csrss.exe, smss.exe, or any entry in the hidden list.

If the SystemModuleInformation class is specified, then the hook walks the returned list and deletes any entry that contains the name of the driver by copying the entries that follow it over the top, and then reducing the returned length.

If the SystemHandleInformation class is specified, then the hook walks the returned list and deletes any handle for entries in the hidden list by copying the entries that follow it over the top, and then reducing the returned length. This change occurs unless the list is requested by csrss.exe, smss.exe, or any entry in the hidden list.

When any of the following ntoskrnl functions are called: NtWriteFile(), NtWriteFileGather(), NtShutdownSystem(), NtRaiseHardError(), NtSetSystemPowerState(), NtInitiatePowerAction(), or if the csrss ExitWindowsEx() function is called, the corresponding hook can be directed to ignore the request.

When the ntoskrnl NtSetInformationThread() function is called, the hook checks the ThreadInformationClass parameter. If the HideThreadFromDebugger class is specified, then the hook simply returns success. There is a bug in this code, which is that if an invalid handle is passed to the function, then an error code should be returned. A successful return would be an indication that *HideToolz* is running.

When the ntoskrnl NtClose() function is called, the hook calls the ntoskrnl NtQueryObject() function to verify that the handle is valid. If it is, then the hook calls the ntoskrnl NtClose() function. Otherwise, it returns STATUS_INVALID_HANDLE (0xC0000008). However, disabling the exception in this way, without reference to the 'HKLM\System\CurrentControlSet\Control\Session Manager\GlobalFlag' registry value, means that the absence of the exception might reveal the presence of *HideToolz*.

When either the ntoskrnl NtOpenProcess() function or the ntoskrnl NtOpenThread() function is called, the hook calls the original ntoskrnl function, and then exits if either an error occurs, or the resulting handle refers to any entry on the hidden list.

When the ntoskrnl NtDuplicateObject() function is called, the hook returns an error if the source handle refers to any entry on the hidden list.

HideToolz exposes a private interface via the ntoskrnl NtTerminateProcess() function. The interface is accessed by passing a handle that is actually a pointer to a memory block, and specifying a termination status of 0xDFF42AB7. The contents of the memory block must begin with the sequence 0x6B 0xB8 0xEC 0x75 0x47 0x46 0x0B 0xFB. Following that is an index into a table of function pointers. Only the values 0–5 are accepted. Next is a pointer to the input buffer, then the size of the input buffer, a pointer to the output buffer, and finally the size of the output buffer. The output buffer is verified to be writable, and the hook requires that the buffer is in user-mode memory.

The author of *HideToolz* could not be contacted.

2. OBSIDIAN-SPECIFIC

Obsidian is an unusual style of user-mode debugger. It does not attach to a process, but instead uses the kernel32 CreateToolhelp32Snapshot() function, and the kernel32 Thread32First() and Thread32Next() functions to access threads. It uses the kernel32 ReadProcessMemory() and WriteProcessMemory() functions to read and write process memory, including to set and clear breakpoints. Even the style of breakpoint is unusual. Rather than the common 'CC' opcode (short-form 'INT 3' instruction) and the T flag to raise single-step exceptions, *Obsidian* places an 'EBFE' opcode ('JMP $' instruction) at the desired location, and uses a timer with a 'sufficient' delay to allow the execution to complete.

2.1 FindWindow

Obsidian can be found by calling the user32 FindWindow() function, and then passing 'ObsidianGUI' as the window name to find.

Example code looks like this:

```
     push offset l1
     push 0
     call FindWindowA
     test eax, eax
     jne  being_debugged
     ...
l1:  db   "ObsidianGUI", 0
```

2.2 Escape

Because of the breakpoint style in *Obsidian*, it is vulnerable to self-modifying code which is aware of the format of the breakpoint.

Example code looks like this:

```
     mov b [offset l1], 0b0h
l1:  mov al, 1
     ;execution resumes freely here
```

This code also functions as a method to detect *Obsidian*, since the value in the AL register will be altered from 1 to 0xFE if *Obsidian* is running.

Obsidian does not handle exceptions, but this limitation is documented already.

Example code looks like this:

```
      xor  eax, eax
      push offset l1
      push d fs:[eax]
      mov  fs:[eax], esp
      int  3
      ...
l1: ;execution resumes freely here
```

The author of *Obsidian* is investigating the report.

3. UGDBG-SPECIFIC

UGDbg is a user-mode debugger with an interface that is similar to *SoftICE*. It can debug both 32-bit and 64-bit applications. It sets the PEB->BeingDebugged and PEB->NtGlobalFlag flags to zero, and does the same for the debugging-heap tail (0xBAADF00D, 0xFEEEFEEE), if it is present. *UGDbg* also attempts to set the PEB->Heap->ForceFlags flag to zero, however the location of the ForceFlags fields is different in *Windows Vista* and later versions, so the change fails on that platform.

UGDbg uses hardware breakpoints for both single-into and step-over. As a result, it is not vulnerable to the common step-over attack described in [10], nor to any of the variations described below.

4. ROCK DEBUGGER-SPECIFIC

Rock Debugger was described in a previous paper [8]. What follows is a bug that has been discovered since that paper was published.

4.1 Step-over

When *Rock Debugger* is asked to step over an instruction, it checks if stepping over the instruction is a meaningful request. *Rock Debugger* allows the stepping over of any instruction that can be decoded, and which starts with a REP prefix. This leaves the breakpoint vulnerable to self-modifying code.

Example code looks like this:

```
      rep
l1: mov b [offset l1], 90h
l2: nop
```

If a step-over is attempted at l1, then execution will resume freely from l2.

Rock Debugger refuses to disassemble code within 15 bytes of the end of a page, if the following page is not readable. Step-over is also disallowed in such cases. This can make it difficult to debug certain applications, since it is possible to fit several executable instructions within that space.

The author of *Rock Debugger* responded very quickly to the report. He is considering adding a user-defined option to control the behaviour when stepping over instructions that start with a REP prefix.

5. TURBO DEBUG32-SPECIFIC

Turbo Debug32 was described in a previous paper [6]. What follows are bugs that have been discovered since that paper was published.

5.1 Export table

Turbo Debug32 parses the debuggee's export table to identify the offered symbols. It assumes that an ordinal table always exists, and that the contents are valid, even though the table is not used if exports are exported by ordinal only. *Turbo Debug32* uses the values inside the ordinal table as indexes into a memory block within the *Turbo Debug32* process. The accesses are performed without any bounds checking. As such, by placing sufficiently large values into the table, it is possible to cause *Turbo Debug32* to crash.

5.2 Relocated code

Turbo Debug32 does not place the entrypoint breakpoint correctly if the executable file has been relocated in memory as a result of an invalid requested ImageBase (such as loading to offset zero). However, there is no problem if the image is loaded to a random address as a result of Address Space Layout Randomization.

When a process is started, a debugger typically wants to place a breakpoint at the main entrypoint. There are two common ways to locate this address. The first is to query the PEB->Ldr->InMemoryOrderModuleList->EntryPoint field value. Interestingly, *Microsoft* documentation labels this field as 'unsupported', even though the psapi32.dll uses it. The second way is to wait for the CREATE_PROCESS_DEBUG_EVENT event to occur, and then to query the CREATE_PROCESS_DEBUG_INFO->lpStartAddress field value.

However, there is a problem with the second method. *Windows* has supported the relocation of EXE files since *Windows 2000*. With the introduction of *Windows Vista* and Address Space Layout Randomization, this 'feature' came to be supported officially. As a result, a file can be loaded to an address other than the one that it requested. One case in particular is when the requested address is intentionally invalid, such as zero or greater than 2GB. This causes *Windows* to load the file to 0x10000. The problem is that for such files, the CREATE_PROCESS_DEBUG_

INFO->lpStartAddress field value contains the 'expected' (and incorrect) entrypoint value, which is calculated by summing the values from the PE->ImageBase and the PE->AddressOfEntryPoint. A breakpoint that a debugger places at that location will not be hit. If the debugger then resumes the process, the process will run freely. Further, the incorrect entrypoint can be calculated to point to a known-writable memory location. The process can then check for a breakpoint at this location and the presence of the debugger will be revealed. This is the problem in *Turbo Debug32*. Other debuggers, such as *OllyDbg*, handle this situation correctly because they use the first method in one form or another, such as by calling the psapi GetModuleInformation() function, which queries the PEB->Ldr->InMemoryOrderModuleList->EntryPoint, and which contains the correct entrypoint value.

This problem has been documented publicly [13].

5.3 Step-over

When *Turbo Debug32* is asked to step over an instruction, it checks if stepping over the instruction is a meaningful request. *Turbo Debug32* allows stepping over only the CALL, REP[[N]E] <string>, and LOOP[[N]E] instructions. The CALL instruction is of particular interest because, as described in a previous paper [8], it does not support the SIB encoding. In the previous example, the bug was exploited to transfer control to an unexpected location. However, a more effective exploitation of the bug allows code to escape the control of the debugger. The bug occurs because *Turbo Debug32* assumes that any SIB-encoded instruction is six bytes long. Thus, if a CALL instruction followed by a JMP instruction can be encoded in no more than six bytes, then stepping over the CALL instruction will allow the JMP to be reached, after which the execution will resume freely from the destination of the JMP instruction.

Example code looks like this:

```
     xor  ebx, ebx
     push 40h
     mov  eax, esp
     push 3000h
     push esp
     push ebx
     push eax
     push -1 ;GetCurrentProcess()
     call NtAllocateVirtualMemory
     mov  b [ebx], 0c3h
     call d ds:[1]
     jmp  short l1
     nop ;replaced by int 3
l1: ...
```

Turbo Debug32 has another bug regarding instruction decoding, which is that it does not override the address-size (0x67) when applied to a long displacement. As above, the bug occurs because *Turbo Debug32* assumes that any SIB-encoded instruction is six bytes long. Thus, if a CALL instruction followed by a JMP instruction can be encoded in no more than six bytes, then stepping over the CALL instruction will allow the JMP to be reached, after which the execution will resume freely from the destination of the JMP instruction. Example code looks like this:

```
     xor  ebx, ebx
     push 40h
     mov  eax, esp
     push 3000h
     push esp
     push ebx
     push eax
     push -1 ;GetCurrentProcess()
     call NtAllocateVirtualMemory
     mov  b [ebx], 0c3h
     call d [bx+80h]
     jmp  short l1
     nop ;replaced by int 3
l1: ...
```

Turbo Debug32 ignores errors when a write occurs beyond writable memory. This bug can also be exploited to allow execution to resume freely from an arbitrary location, if a step-over is attempted at the end of a page. Example code looks like this:

```
     xor  ecx, ecx
     push offset l1
     push d fs:[ecx]
     mov  fs:[ecx], esp
     mov  eax, offset l2
     mov  w [eax], 0fee0h ;loopne $
     jmp  eax
l1: ;reached if step-over at l2
     ...
l2: ;place at 2nd-last byte in page
```

6. WINDBG-SPECIFIC

WinDbg was described in two previous papers [1, 3]. What follows is a detection method that has been discovered since those papers were published.

6.1 Step-over

When *WinDbg* is asked to step over an instruction, it checks if stepping over the instruction is a meaningful request. *WinDbg* allows stepping over of the CALL (0x9A, 0xE8 and 0xFF &38==0x10), INT (0xCC, 0xCD and 0xCE), REP[[N]E] <string> (including INS and OUTS), LOOP[[N]E] and BOP (0xC4 0xC4) instructions. The BOP support is especially interesting not least because it is undocumented, but also because *WinDbg* knows something

about the format. Specifically, *WinDbg* knows that the 0x50, 0x52-0x54, 0x57-0x58 and 0x5D indexes are four bytes long. For all other BOP indexes, *WinDbg* knows that the instruction is only three bytes long. This might appear to be vulnerable to a step-over bug, since some of the BOP indexes cause exceptions which the debuggee can intercept. However, *WinDbg* takes care to replace the breakpoint with the original byte value prior to dispatching the exception.

WinDbg also allows the stepping over of unknown instructions. This is achieved by using the single-step exception instead of a breakpoint. *WinDbg* also behaves in this way for instructions which contain redundant prefixes. This leaves *WinDbg* vulnerable to detection via the T flag. Example code looks like this:

```
cs:cs:pushfd
pop  eax
test ah, 1
jne  being_debugged
```

7. FDBG-SPECIFIC

FDBG is a debugger for the 64-bit platform. It can debug 64-bit executables.

7.1 Step-over

When *FDBG* is asked to step over an instruction, it checks if stepping over the instruction is a meaningful request. *FDBG* allows the stepping over of any instruction (valid or not) which starts with a REP prefix. This leaves the breakpoint vulnerable to self-modifying code. Example code looks like this:

```
    rep
l1: mov b [offset l1], 90h
l2: nop
```

If a step-over is attempted at l1, then execution will resume freely from l2.

FDBG refuses to disassemble code within 253 bytes of the end of a page if the following page is not readable. Step-over is disallowed within 31 bytes of the end of a page if the following page is not readable. This can make it difficult to debug certain applications, since it is possible to fit several executable instructions within that space.

The author of *FDBG* responded quickly to the report. A test version was shared privately, which solves these problems, and also the 'rep stos' problem (and its variations) described in the 'Self-modifying code' section in [10].

8. TITAN ENGINE

TitanEngine is a tool for reverse-engineering applications. It has a built-in debugger. *TitanEngine* uses breakpoints for any step-over request, regardless of the instruction which is being stepped over. *TitanEngine* supports three kinds of breakpoint instruction – the 'CC' opcode (short-form 'INT 3' instruction), 'CD03' (long-form 'INT 3' instruction), and '0F0B' opcode ('UD2' instruction). Because of the breakpoint style in *TitanEngine*, it is vulnerable to self-modifying code which is aware of the format of the breakpoint.

Example code looks like this:

```
    mov b [offset l1], 0b0h
l1: mov al, 1
    ;execution resumes freely here
```

This code also functions as a method to detect *TitanEngine* if the breakpoint style is not the 'CC' opcode form, since the value in the AL register will be altered from 1 to either 3 or 0x0B if *TitanEngine* is running.

The author of *TitanEngine* is investigating the report.

The next part of this series will look at *IDA* plug-ins.

The text of this paper was produced without reference to any Microsoft source code or personnel.

REFERENCES

[1] http://pferrie.tripod.com/papers/unpackers.pdf.

[2] http://www.virusbtn.com/pdf/magazine/2008/200812.pdf.

[3] http://www.virusbtn.com/pdf/magazine/2009/200901.pdf.

[4] http://www.virusbtn.com/pdf/magazine/2009/200902.pdf.

[5] http://www.virusbtn.com/pdf/magazine/2009/200903.pdf.

[6] http://www.virusbtn.com/pdf/magazine/2009/200904.pdf.

[7] http://www.virusbtn.com/pdf/magazine/2009/200905.pdf.

[8] http://www.virusbtn.com/pdf/magazine/2009/200906.pdf.

[9] http://www.virusbtn.com/pdf/magazine/2010/201005.pdf.

[10] http://www.virusbtn.com/pdf/magazine/2010/201006.pdf.

[11] http://www.virusbtn.com/pdf/magazine/2010/201007.pdf.

[12] http://www.virusbtn.com/pdf/magazine/2010/201008.pdf.

[13] http://pferrie.tripod.com/misc/lowlevel3.htm.

FEATURE

WHAT'S THE DEAL WITH SENDER AUTHENTICATION? PART 4

Terry Zink
Microsoft, USA

In the previous articles in this series [1–3], we've seen two relatively simple methods for authenticating email: SPF and SenderID. Both can be used to authenticate a sender and presumably trust a message, and both can also be used to detect spoofing. However, as we have also seen, both have their weaknesses: SPF can be circumvented by not using a domain in the P1 From that has SPF records, while SenderID can be prone to false positives when mail is sent on behalf of another. Neither technology works when mail is forwarded. Furthermore, both technologies tie a domain to a specific set of IP addresses.

To illustrate the problem, suppose my friend Tony has moved to Sacramento. I know that he always sends mail from Sacramento and so when I get the letter in the mail, I check the postmark, verify that it's from Sacramento and that it has his name on it. But what if Tony moves to St. Louis? He has to update the postal service. And, if he moves to Boston, he has to update the postal service again.

Furthermore, what if Tony gives the letter to Frank to deliver? Frank might have to make a stop in Memphis, Tennessee before he mails the letter. When I receive it, I see that it has Tony's name on it, but it came from Memphis. I know that Tony always sends mail from Sacramento or St. Louis. What's it doing coming from Tennessee? If that occurred, I would be tempted to think that it was not actually Tony's mail. At the very least, I wouldn't be able to verify that it was from him.

In email, in the case where I have a complex forwarding system set up to deliver mail to my personal domain, suppose I have a forwarding rule so that if Tony sends mail to my *Hotmail* account, it forwards to my *Gmail* account. I then have all of my *Gmail* sent to my personal domain.

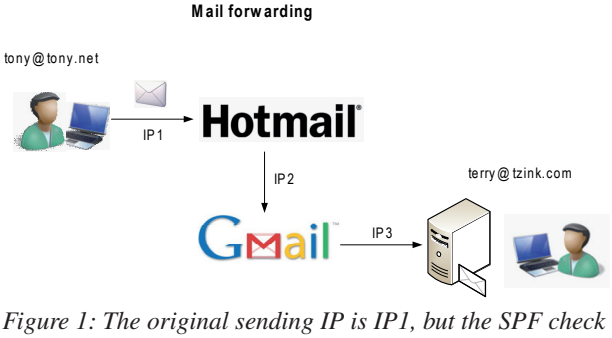


Figure 1: The original sending IP is IP1, but the SPF check is performed on IP3, which results in a fail.

The problem is that since the SPF or SenderID check is performed on the perimeter, the originating IP looks to my mail servers like *Gmail*'s IP. I cannot rely on header traversal to walk through IPs and search for the actual originating IP because received headers can be forged. A SenderID or SPF check will fail in this case, and it *should* fail; SPF and SenderID are only done on the headers that you can trust.

What would be handy would be if Tony put some sort of stamp of authenticity into his letter. What if Tony had a personal seal which he could dip into wax and stamp onto the bottom of his letter, and he was the only one in the world with this stamp? When I got the letter, rather than seeing where it came from, I could instead look for the seal at the bottom of his letter. Since Tony is the only one in the world that has this seal, I could be sure that the letter came from him.

In the postal system, and in letter writing, it would not be all that difficult to forge a seal that looked like Tony's. Fortunately, when it comes to technology, we can do better.

ENCRYPTION

Before we get into the technology used to establish identity, we first need to understand the basics of encryption. In the olden days, people needed ways of communicating messages securely between one another. At first, they simply sent their trusted companions on horseback. For example, a king would send a message with his assistant to the general out on the front lines. As time passed and technology moved on, people began sending messages electronically because this was much quicker and you could push through more data in a shorter period of time. Generals who can communicate between each other and transmit information faster have an advantage over those who can't. But the problem was security; if the message in transit was sensitive, then if somebody intercepted the message the secret information would no longer be secret.

The idea behind encryption is to encode the contents of the message such that even if the message is intercepted in transit, the person who intercepted it would be unable to read its contents. Consider the following message:

```
Ifmmp, J bn bo fodszqufe nfttbhf.
```

This text appears to be a bunch of gobbledygook but it is actually an example of a substitution cipher. The key is that each letter is actually the subsequent letter in the alphabet. In other words, B is substituted for A, C is switched for B, and so forth. For the above, the decrypted message is the following:

```
Hello, I am an encrypted message.
```

Different types of substitutions can be used. Above, I used a one-character algorithm, but others can be used such as a three-character substitution or an 11-character substitution. A three-character substitution would be the following:

```
khoor, L dp dq hqfubswhg phvvdjh.
```

While a substitution cipher is easy to implement, it is also very easy to break. The more text you have, the more you can use statistical analysis to break the cipher. For example, in the English language, the most common letter is 'e'. If you were to intercept a message in transit without knowing the substitution algorithm, you would look for the letter that occurs the most often and that would be pretty likely to be the letter 'e'. You could then look for a bunch of three-letter-words and make a guess that the first letter is 't' and the second letter is 'h'. In this way, you've guessed the letters for the word 'the'. Other commonly occurring consonants are r, s, l and n. Small, two-letter words are likely to be words such as in, of, on, at, it, and so forth. Once you start getting the smaller words you can use a process of elimination to work your way backwards in order to find the rest of the letters. Sometimes it is a process of trial and error to find the words that fit, but with enough iterations you can do it.

Computers are very good at iterating algorithms to find out patterns like this. Rather than using a simple substitution cipher, you could use a more complicated algorithm – for example by substituting the first letter of the message with the letter that follows it in the alphabet, the second with the letter that appears two letters after it in the alphabet, the third with the letter that appears three letters after it in the alphabet, and then repeating the sequence.

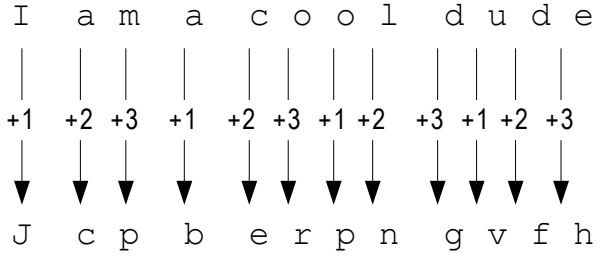


Figure 2: Example of a more complicated substitution algorithm.

However, given enough time, a computer could break this algorithm as well. It wouldn't take very long because substitution ciphers that work by switching one letter for another are not complicated to reverse engineer. An encryption-breaking algorithm works by trying every possible combination and then running the decrypted text against a text recognition program that detects recognizable word patterns in plain text. The swapping around of letters at various fixed points in the example above would be deciphered in a trivial fashion because this is something that computers can do extremely quickly.

Enter the concept of one-way functions. A one-way function is a mathematical function that is easy to calculate one way but very difficult to calculate in the inverse. For example, consider the process of squaring a number. It is easy to calculate x^2 but it is more difficult to calculate the square root of x, $\sqrt{x}$. The algorithm for a square root is more complicated than squaring a number and takes longer to evaluate. Another example would be a logarithm. It is easier to calculate 10^x than it is to evaluate $\log_{10}x$.

A good encryption algorithm makes use of these one-way functions. A message sender would encrypt his message using a one-way function and send it to the receiver and even if somebody intercepted the message in transit, they would have a difficult time decrypting it. The algorithm is computationally intensive, which makes breaking it cost-prohibitive (in terms of time). The non-intended recipient *could* break the message since all it takes to break it is a matter of time and enough computing power, however, the idea is that by the time they did this, the contents of the message would be stale. In other words, it would not be useful to the non-intended recipient. For example, if a military commander was going to organize his troops for a surprise attack on the enemy in a week's time, he might send a message and encrypt it using an algorithm that is breakable but which would take a long time to break, at least two months on average. In transit, the message is intercepted and the enemy proceeds to attempt to break it. The enemy will eventually be successful but by the time they do, the original commander will have made his attack and the information will be stale-dated and no longer useful.

Thus, if you wanted to encrypt the contents of an email message such that it was resistant to people who might try to intercept it, you would use an algorithm that takes a long time to decrypt.

This is all well and good for the people who you don't want reading your message, but what about the person who you *do* want to read it? What good is it if it takes *them* forever to read the message? You might as well not send it at all.

This is where secret key encryption comes in. With secret key encryption, you use a mathematical algorithm to encode your message, and use a secret key to do it. So, a message would be scrambled by using the mathematical function that returns a different result each time you use a different key.

Here's a very basic example. Suppose you wanted to encode the number sequence:

```
4 8 15 16 23 42
```

Let's suppose your secret key, n, is 2 and you are using the algorithm $f(x) = x^n$. We'd encode the sequence this way:

```
16 64 225 256 529 1764
```

If our secret key were 4, we would encode the sequence this way:

```
256 4096 50625 65536 279841 3111696
```

The recipient would receive the encoded message and would also know the algorithm. Therefore they also would know the decryption algorithm (in our case, the square root of x or the 4th root of x). Secret key encryption works not by keeping the algorithm secret (as is the case of a substitution cipher) but by keeping the *key* secret. If you don't know the key, it will take you a long time to figure out the contents of the message. By the time you do, the data will no longer be useful. In my example above, that doesn't look too difficult to break. What would happen if our secret key were 8? Or 16? Or 3.14159? There are still algorithms out there that are computationally expensive for computers to break.

```
77.70847 685.0189 4930.904 6038.607 18872.32 125026.7
```

Someone who intercepted this piece of cipher text wouldn't know what the algorithm for encryption was so they would start trying every possible combination. They might suspect that it is an exponential function and so start by attempting to decrypt using the square root of x, then the cube root of x, and then the fourth root of x. When none of those worked they might start working on decimal points, and so forth. While modern computers today can blaze through mathematical functions in the blink of an eye, it does take more CPU time to evaluate these mathematical functions. If the mathematical function is more complicated, then even powerful computers can start to slow down and take a long time to process it.[1]

This brings me to my next point: in order to increase the security of an encrypted message, you don't need to change the algorithm; you only need to increase the length of the key. For example, it is easier to calculate the inverse of x^4 than $x^{4.5}$, which is easier still than $x^{4.59}$, which is easier than $x^{4.591234}$, and so forth. Knowing what the secret key is makes it possible to decrypt the secret message in a shorter time frame, but it is always going to take longer than it did to encrypt it. For example, encrypting it might take two seconds but decrypting it takes three seconds. On the other hand, without knowing the secret key, the amount of computational processing time makes decryption infeasible from a usefulness perspective because of stale-dating of information.

[1] Modern encryption algorithms do not rely exclusively on mathematical functions. They swap bits around and make use of prime numbers. Furthermore, standard algorithms get reviewed by the encryption community looking for weaknesses (back doors that make them easy to reverse engineer).

DISTRIBUTION

The basic idea behind secret key encryption is the following:

1. The encryption algorithm should be secure (i.e. one-way).
2. You don't have to keep the algorithm a secret.
3. It should only be able to be decrypted by use of a secret key.
4. You do need to keep the key a secret.
5. To increase the security of the contents, you lengthen the size of the key.

The next question arises: how do you distribute the key to your recipients? And what do you do if you want to update your key? Do you have to send them a letter containing it, talk on the telephone and verbalize it, or maybe send a representative on horseback carrying a new key? That's a bit of a hassle.

This is where public key encryption comes in. Whereas with secret key encryption, the same key is used to encrypt the message as to decrypt it, with public key encryption, you use two different keys in the process: one to encrypt and one to decrypt. The public key algorithm is similar to secret key encryption except that the keys are pairs and are designed to work together. You cannot decrypt a message encoded with one key without the other (if you lose one, you're out of luck). The keys are unique (or nearly unique) to each other. Suppose that Bob wanted to send an encrypted message to Alice. Here's how the process works:

1. Alice picks two keys and makes one public and keeps the other private.
2. Bob asks Alice for her public key, and Alice gives it to him.
3. Bob encrypts the message with Alice's public key and transmits the message to Alice.
4. Alice receives the message and decrypts it with her private key. Alice is the only one that can decrypt the message with her private key.

Note that after Bob encodes his message, he can't decrypt it with the public key to double-check its contents. Once it's encoded, it's encoded and he can't check it over. So, Bob can transmit the message to Alice and, just like secret key encryption, without the secret key to decrypt the message, the message contents are protected if it is intercepted in

transit by an unintended party. Eventually, it could be broken but it would be time-prohibitive to do so.

Public key encryption solves the problem of key distribution. Using public key encryption, you don't have to worry about distributing your key to others, they simply ask you for your public key, you give it to them and then they send you the message. Note that you can use either key to encrypt or decrypt, but you have to keep one of them secret. Again, the strength of this process is that you don't have to keep the algorithm or the public key secret, only your private key.

DIGITAL SIGNATURES

However, recall that *either* key can be used to encrypt and decrypt. That is, we can encrypt with the private key and decrypt with the public key. This means that anyone can intercept the message, and anyone (with knowledge of the public key) can then read the message. Why would we want to do this? Don't we always want to keep the message contents a secret? As it turns out, there are times when we don't care about keeping the contents a secret, we only care about *who* encrypted the message.

This brings us to the concept of a digital signature. In real life, a signature is something that does the following:

1. Provides proof that a person authorized the contents of the document.
2. Is unique to the individual.

If a document is signed with a person's signature (such as Tony's signature), I am not concerned about the contents of the document (his letter to me), I am only concerned that Tony authorized the letter, and that the signature is unique to him.

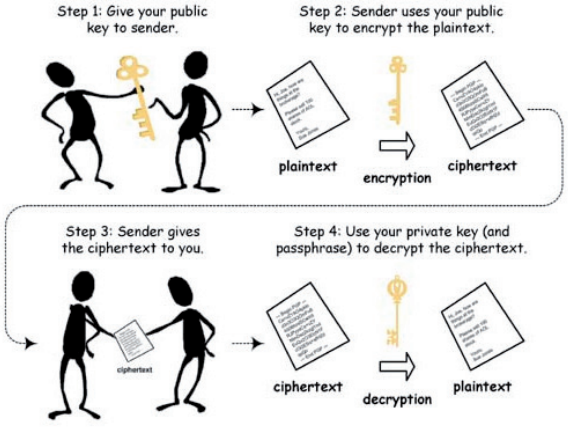


Figure 3: Public key encryption[2].

Public key encryption allows us to *digitally* sign a document. Here is how the process of authentication works between Bob and Alice:

1. Bob creates a document and signs it with his signature (i.e. 'I am Bob and I signed this document').
2. Bob encrypts the document with his private key and sends it to Alice.
3. Alice receives the message, reportedly from Bob, and asks Bob for his public key. Bob sends it to Alice.
4. Alice takes the public key and decrypts the message. The contents of the message contain Bob's signature, which verifies that the message came from Bob.

What would happen if Bob sent a key that was not part of the public/private key pair? Assume someone claims to be Bob and sent Alice a message. Alice asks the real Bob for his public key, who sends it to Alice. Alice decrypts the message, but because Bob's public key only works with his private key, the contents of the message do not decrypt properly. Alice judges that the message did not actually come from Bob.

If the contents of the message did decrypt properly, then Alice could have judged that the message did come from Bob. Since the keys can only work in pairs, only the private key that was used to encrypt the message could have been the one used to create the signature, and only the public key could have decrypted it. In other words, encrypting a message with a private key allows others with a public key to verify (authenticate) the original signer of a message.

In the world of email, if Tony were to send a message to me, it might look something like the following[3]:

1. Tony decides that he will proceed to sign all of his messages with the following signature: 'I am Tony and I approve this message'. He uploads the signature to his public DNS at diamond.net as well as his public key[4].
2. Tony next wants to send me a message. At the bottom of it, he adds a signature – 'I am Tony and I approve this message'. He places it between two XML tags which makes it easy for me to parse:

```
From the desk of <person>tony@diamond.net</person>

Hey Terry, you're an awesome person.

<signature>
I am Tony and I approve this message.
</signature>
```

[2] Image from http://www.data-processing.hk/uploads/images/public_key_encryption%281%29.jpg.

[3] There is no actual protocol that uses this flow of events, it is for illustrative purposes.

[4] This means that there are two entries in DNS – a secret key and a clear text signature establishing Tony's identity.

3. Tony encrypts the signature with his private key using the TZFA[5] algorithm. He does not encrypt the entire message, only his signature. It now looks like the following:

```
From the desk of <person>Tony</person>

Hey Terry, you're an awesome person.

<signature>
ksxal;q1254naa;lkasdf\a;kz7a890asd\2;
</signature>
```

   He then proceeds to send me the email.

4. I receive the message and I don't bother to do an SPF check. Instead, I see that Tony has placed his name between the <person> XML tags. I extract the signature between the <signature> tags, trimming any leading and trailing white space. I see that the message is purportedly from tony@diamond.net and since there is a signature at the bottom, I attempt to decrypt it.
5. I retrieve Tony's public key which is stored in public DNS at diamond.net. I run the TZFA algorithm on the contents of the signature and it reads the following: 'I am Tony and I approve this message'.
6. I proceed to retrieve Tony's clear text signature from DNS. I compare the signature from the email against the one from DNS. The two of them match, and I decide that the message really did come from Tony. My day has just got better because my friend has told me I'm an awesome person.

In this example, nobody could ever send me a message where the signature decrypts to 'I am Tony and I approve this message' while claiming to be from the person tony@diamond.net. The public key *only* works to decrypt messages encrypted with Tony's private key. If someone attempted to forge the message and encrypt it with a different secret key, then when I decrypted the signature it would be a different string of text and it would not match Tony's signature which he had uploaded to DNS.

Digital signatures solve the problem of email forwarding; you no longer have to identify the correct source IP address of the mail. So long as the originator of the message always adds their signature and the signature can be extracted[6], you will be able to validate it. Mail can be forwarded any number of times, but as long as the contents of the signature are not modified, it will always be properly validated. The message is validated securely and reliably by the contents, *not* the sending IP. It is not spoofable in any practical sense.

Similarly, with digital signatures, a domain doesn't have to tie all of its outbound mail to a particular set of IPs, it only needs to ensure that it signs with the same private key. If IPs change, it won't matter because it is with the private/public key pair that mail is validated, not a rotating set of IPs. To be sure, keys need to be rotated every so often, but you can add more servers and outbound IPs with less overhead. The receivers of your mail will be able to validate it with a DNS query to the sending domain without worrying about your IP addresses.

A word of caution, however. Digital signatures work to establish identity and trust. They do not necessarily work to establish forgery:

1. If a message comes to me purportedly from Tony and does not contain a signature, it doesn't mean that the message didn't come from him (i.e. is being spoofed). He may have not signed this particular message. Perhaps he forgot, or perhaps he is in the process of rotating keys, or doing server maintenance and didn't have time to update the keys.
2. If a message comes to me purportedly from Tony and does contain a signature, but the signature doesn't validate properly (i.e. doesn't match what he has uploaded in DNS), it doesn't mean that the message didn't come from him. The signature may have the wrong private/public key pair (i.e. a misconfiguration), or it could mean that the message was modified in transit since changing characters in a string affects how it is decrypted. Modifications in transit can be intentional (such as line wrapping by a mail transfer agent) or unintentional (such as line noise that changes the bit stream).

This example of digital signature validation is essentially what is done in the actual world of email. The discussion on the main technology used to do it, Domain Keys Identified Mail, or DKIM, will have to wait until next month.

REFERENCES

[1] Zink, T. What's the deal with sender authentication? Part 1. Virus Bulletin, June 2010, p.7. http://www.virusbtn.com/pdf/magazine/2010/201006.pdf.

[2] Zink, T. What's the deal with sender authentication? Part 2. Virus Bulletin, July 2010, p.16. http://www.virusbtn.com/pdf/magazine/2010/201007.pdf.

[3] Zink, T. What's the deal with sender authentication? Part 3. Virus Bulletin, August 2010, p.16. http://www.virusbtn.com/pdf/magazine/2010/201008.pdf.

[5] Terry Zink's Fictional Algorithm.

[6] A signing algorithm generally specifies how to extract the signature.

COMPARATIVE REVIEW

VBSPAM COMPARATIVE REVIEW SEPTEMBER 2010

Martijn Grooten

My spam isn't the same as your spam, which then isn't the same as the spam of the man playing with his *iPhone* next to you on the bus. That isn't too surprising: our respective email addresses may have ended up on different spammers' lists, and different spammers send different spam. But spam sent to addresses on one domain also differs from that sent to addresses on a different domain, and even groups of domains – where one might expect such differences to average out – receive spam that differs significantly.

We have always kept this in mind when running our anti-spam tests. Since we wanted the tests to provide a measure of performance that would be relevant to any organization, we didn't want to use the spam sent to a single domain, or even group of domains. This is the reason why we have been using *Project Honey Pot*'s spam feed for our tests. *Project Honey Pot* receives spam sent to a large number of spam traps on a large number of domains, distributed all over the world. By using this feed, we can be sure that products are being tested against spam that isn't any more likely to be received by someone in the UK than by someone in, say, New Zealand.

However, we always like to see things from a different perspective, and this is why we are very pleased to have developed a relationship with *Abusix*, a German company that also manages a large number of spam traps. From this test onwards, *Abusix* will provide us with a second spam corpus; in this test, and in all future tests, products will see spam from both streams (as well as a number of legitimate emails) and will be required to filter all of these emails correctly.

This month's test included 19 full solutions and one partial solution. For various reasons, a number of products that have participated in previous tests decided to sit this one out, but most of them expect to be back on the test bench next time. All of the full solutions tested this month achieved a VBSpam award. However, for several products there is still significant room for improvement and no doubt their developers will be working hard to see their products move towards the top right-hand corner of the VBSpam quadrant.

THE TEST SET-UP

The test methodology can be found at http://www.virusbtn.com/vbspam/methodology/. Email was sent to the products in parallel and in real time, and products were given the option to block email pre-DATA. Five products chose to make use of this option.

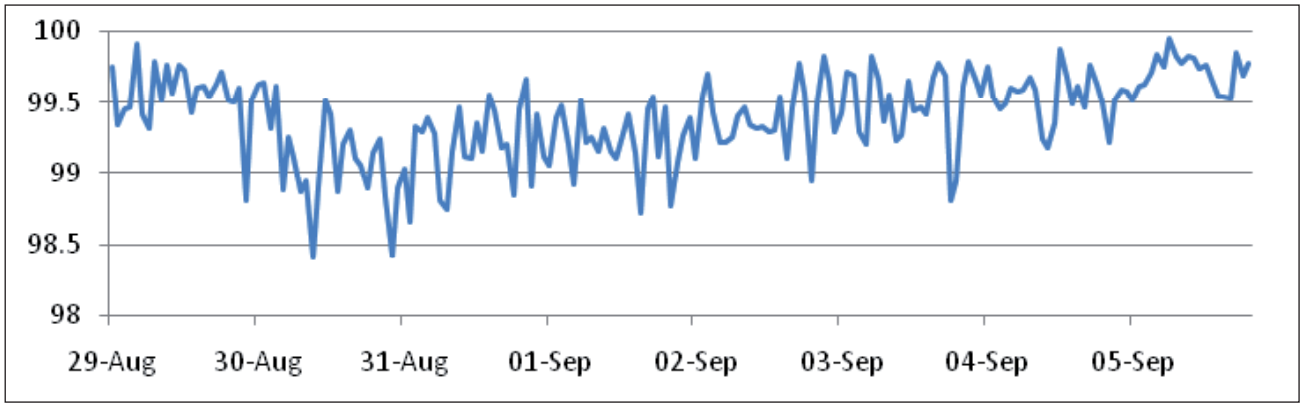
As in previous tests, the products that needed to be installed on a server were installed on a *Dell PowerEdge R200*, with a 3.0GHz dual core processor and 4GB of RAM. The *Linux* products ran on *SuSE Linux Enterprise Server 11*; the *Windows Server* products ran on either the 2003 or the 2008 version, depending on which was recommended by the vendor.

To compare the products, we calculate a 'final score', defined as the spam catch (SC) rate minus three times the false positive (FP) rate. Products earn VBSpam certification if this value is at least 96:

$$SC - (3 \times FP) \geq 96$$

THE EMAIL CORPUS

The test ran from midnight on 29 August 2010 to midnight on 6 September 2010, a period of eight full days. This was



Average spam catch rate throughout the test.

a shorter testing period than usual. A number of system crashes had caused the test network to be unreliable for several days after the test was initially started, and rather than using results from periods between the crashes when the network appeared to be working well, we decided to err on the side of caution and restart the whole test. The addition of a second spam stream and an increase in the size of the ham corpus gave us quantities of email comparable to those of previous tests despite the shorter test period.

The corpus contained 211,968 emails, 209,766 of which were spam. Of these spam emails 148,875 were provided by *Project Honey Pot* and 60,891 were provided by *Abusix*; in both cases they were relayed in real time, as were all legitimate messages, of which there were 2,202. The introduction of some new mailing lists (see *VB*, May 2010, p.24 for details), some of which are in foreign languages not previously included, means that seven out of the ten most commonly spoken languages in the world are now represented in the ham corpus.

The graph on the previous page shows the average spam catch rate for all products during every hour that the test ran (with the best and worst performing products removed from the computation of the averages). The graph shows that spam was harder to filter in certain periods than in others; for instance new spam campaigns tend to be harder to filter than ones that have been running for a while.

RESULTS

Anubis Mail Protection Service

SC rate: 99.93%
SC rate (image spam): 99.77%
SC rate (large spam): 99.63%
SC rate pre-DATA: N/A
FP rate: 0.05%
Final score: 99.80

Lisbon-based *AnubisNetworks*, the largest email security provider in Portugal, made a good debut in the previous test. The product's developers, however, were only mildly satisfied with the test results as they believed the product was capable of better. They were right – this month the product's spam catch rate increased, and the false positive rate was reduced to just a single missed email. With the second highest final score of this test, the developers should be very pleased with these results and the accompanying VBSpam award.

BitDefender Security for Mail Servers 3.0.2

SC rate: 99.91%
SC rate (image spam): 99.81%
SC rate (large spam): 99.20%
SC rate pre-DATA: N/A
FP rate: 0.00%
Final score: 99.91

I like it when developers have confidence in their product, and *BitDefender*'s developers demonstrated plenty of confidence when they were among the first to submit their product to the VBSpam tests in the early days. Despite this, they have never stopped trying to find ways to improve the product and have always been eager to hear feedback on its performance. *BitDefender* is the only product to have won a VBSpam award in every single VBSpam test – and with one of the highest catch rates in this test, and no false positives, it outperforms all other products and achieves the highest final score this month.

Fortinet FortiMail

SC rate: 98.44%
SC rate (image spam): 97.34%
SC rate (large spam): 95.98%
SC rate pre-DATA: N/A
FP rate: 0.05%
Final score: 98.30

One of the products that has been filtering mail quietly ever since its introduction to the tests, *FortiMail* wins its eighth VBSpam award in as many attempts. It does so with a nicely improved performance, demonstrating that the product's developers are keeping up with the latest spam campaigns.

Kaspersky Anti-Spam 3.0

SC rate: 98.30%
SC rate (image spam): 98.25%
SC rate (large spam): 97.37%
SC rate pre-DATA: N/A
FP rate: 0.05%
Final score: 98.16

Neither the new ham nor the new spam stream proved to be a problem for *Kaspersky*.

The company's *Linux* product saw its false positive rate improve, while barely compromising on the spam catch rate. *Kaspersky* easily wins another VBSpam award.

Libra Esva 2.0

SC rate: 99.96%
SC rate (image spam): 99.92%
SC rate (large spam): 99.71%
SC rate pre-DATA: 97.93%
FP rate: 0.32%
Final score: 99.01

Once again, *Libra Esva* had one of the highest spam catch rates of all products. Compared to previous tests, the product scored a slightly higher false positive rate – whilst this is something for the developers to pay attention to, the FP rate was still only average. With another very respectable final score, the Italian product wins its third consecutive VBSpam award.

M86 MailMarshal SMTP

SC rate: 99.97%
SC rate (image spam): 99.96%
SC rate (large spam): 99.93%
SC rate pre-DATA: N/A
FP rate: 0.5%
Final score: 98.47

M86's *MailMarshal* blocked the second largest amount of spam of all the products in this test, which is quite an achievement. Unfortunately, the product also missed almost a dozen legitimate emails, which lowered its final score quite significantly. It was still decent though, earning the product its sixth VBSpam award, but the developers will need to concentrate on reducing the FP rate, while not compromising too much on the amount of spam caught.

McAfee Email Gateway (formerly IronMail)

SC rate: 97.81%
SC rate (image spam): 93.08%
SC rate (large spam): 96.85%
SC rate pre-DATA: N/A
FP rate: 0.45%
Final score: 96.45

McAfee's *Email Gateway Appliance* suffered what appeared to be a temporary glitch during this test – with a disappointing spam catch rate towards the start of the test improving to see scores of over 99% during the final days of the test. Despite a small number of false positives, the product still earns a VBSpam award, but the product's developers will no doubt be working hard to determine the cause of the earlier problems and to ensure its spam catch rate remains consistently high in future.

McAfee Email and Web Security Appliance

SC rate: 99.05%
SC rate (image spam): 92.98%
SC rate (large spam): 90.20%
SC rate pre-DATA: N/A
FP rate: 0.27%
Final score: 98.23

McAfee's *Email and Web Security Appliance* achieved a VBSpam award in the previous test, but with a rather low final score. It was good to see that this appears to have been a one-off dip, rather than a serious problem with the installation; a high spam catch rate combined with a small handful of false positives easily earns the product its seventh VBSpam award.

MessageStream

SC rate: 99.08%
SC rate (image spam): 99.68%
SC rate (large spam): 99.56%
SC rate pre-DATA: N/A
FP rate: 0.09%
Final score: 98.81

As a product whose customers are based mostly in the Anglo-Saxon world, correctly filtering email in foreign languages may not be a high priority for *MessageStream*. However, in an industry where the devil is in the details, the developers have taken good care of even these details: a spam catch rate of over 99%, combined with just two false positives, means that the hosted solution more than deserves its eighth VBSpam award.

	True negative	False positive	FP rate	False negative	True positive	SC rate	Final score
AnubisNetworks	2201	1	0.05%	144	233321	99.93%	99.80
BitDefender	2202	0	0.00%	192	233232	99.91%	99.91
FortiMail	2201	1	0.05%	3281	229721	98.44%	98.30
Kaspersky	2201	1	0.05%	3567	229709	98.30%	98.16
Libra Esva	2195	7	0.32%	76	233387	99.96%	99.01
M86 MailMarshal	2191	11	0.50%	60	233408	99.97%	98.47
McAfee Email Gateway	2192	10	0.45%	4587	207284	97.81%	96.45
McAfee EWS	2196	6	0.27%	1999	231362	99.05%	98.23
MessageStream	2200	2	0.09%	1931	231179	99.08%	98.81
Messaging Architects M+Guardian	2182	20	0.91%	111	233291	99.95%	97.22
Pro-Mail	2201	1	0.05%	3607	229309	98.28%	98.15
Sophos	2199	3	0.14%	173	233273	99.92%	99.51
SPAMfighter	2199	3	0.14%	2795	230473	98.67%	98.26
SpamTitan	2188	14	0.64%	1942	231321	99.07%	97.17
Symantec Brightmail	2202	0	0.00%	745	232511	99.64%	99.64
The Email Laundry	2197	4	0.18%	392	233014	99.81%	99.27
Vade Retro	2194	8	0.36%	1175	232230	99.44%	98.35
Vamsoft ORF	2194	8	0.36%	1418	231616	99.32%	98.24
Webroot	2188	14	0.64%	18	233294	99.99%	98.08
Spamhaus ZEN	2202	0	0.00%	18119	211304	91.36%	91.36

Messaging Architects M+Guardian

SC rate: 99.95%
SC rate (image spam): 99.94%
SC rate (large spam): 99.85%
SC rate pre-DATA: 94.89%
FP rate: 0.91%
Final score: 97.22

Quite understandably, *M+Guardian*'s developers were not happy with their product's performance in the last test – in which it failed to achieve a VBSpam award. They looked into the settings of the appliance and among the changes they made was to turn on XCLIENT; this way they could use pre-DATA filtering, which they believe is one of the core benefits of the product.

Indeed, almost 94.9% of the spam was blocked this way, while the subsequent content filtering left less than 0.1% of spam unfiltered. An excellent spam catch rate, and *M+Guardian* easily reclaims its VBSpam award. However, there will be some disappointment for the developers over an incorrectly blocked domain which accounted for 15 of the 20 false positives.

Pro-Mail (Prolocation)

SC rate: 98.28%
SC rate (image spam): 99.66%
SC rate (large spam): 93.93%
SC rate pre-DATA: N/A
FP rate: 0.05%
Final score: 98.15

Like several anti-spam solutions, *Pro-Mail*, the hosted solution that debuted in the last test, classifies email into not

	Project Honey Pot		Abusix		Image spam*		Large spam*		pre-DATA†		St. dev‡
	FN	SC rate	FN	SC rate	FN	SC rate	FN	SC rate	FN	SC rate	
AnubisNetworks	138	99.91%	6	99.99%	11	99.77%	5	99.63%			0.14
BitDefender	112	99.93%	80	99.87%	9	99.81%	11	99.20%			0.15
FortiMail	2449	98.36%	832	98.63%	126	97.34%	55	95.98%			0.89
Kaspersky	2680	98.21%	887	98.54%	83	98.25%	36	97.37%			2.38
Libra Esva	58	99.96%	18	99.97%	4	99.92%	4	99.71%	4452	97.93%	0.09
M86 MailMarshal	39	99.97%	21	99.97%	2	99.96%	1	99.93%			0.11
McAfee Email Gateway	4576	96.94%	11	99.98%	328	93.08%	43	96.85%			2.10
McAfee EWS	1891	98.73%	108	99.82%	333	92.98%	134	90.20%			1.38
MessageStream	1066	99.29%	865	98.58%	15	99.68%	6	99.56%			0.69
Messaging Architects M+Guardian	100	99.93%	11	99.98%	3	99.94%	2	99.85%	10970	94.89%	0.13
Pro-Mail	2886	98.07%	721	98.82%	16	99.66%	83	93.93%			1.60
Sophos	144	99.90%	29	99.95%	17	99.64%	3	99.78%			0.22
SPAMfighter	1567	98.95%	1228	97.98%	139	97.07%	94	93.12%			2.55
SpamTitan	1423	99.05%	519	99.15%	2	99.96%	24	98.24%			1.10
Symantec Brightmail	429	99.71%	316	99.48%	4	99.92%	4	99.71%			0.36
The Email Laundry	336	99.78%	56	99.91%	2	99.96%	4	99.71%	11136	94.82%	0.24
Vade Retro	645	99.57%	530	99.13%	10	99.79%	27	98.02%			0.81
Vamsoft ORF	1232	99.18%	186	99.69%	46	99.03%	40	97.07%			0.51
Webroot	15	99.99%	3	100.00%	2	99.96%	9	99.34%	77506	63.92%	0.05
Spamhaus ZEN	10749	92.69%	7370	86.55%	378	92.03%	126	90.78%	18119	91.36%	3.35

* There were 4,743 spam messages containing images and 1,367 considered large; the two are not mutually exclusive.
† Pre-DATA filtering was optional and was applied on the full spam corpus.
‡ The standard deviation of a product is calculated using the set of its hourly spam catch rates.

two but three categories: ham, spam and 'possibly spam'. Messages that fall into the 'possibly spam' category are not blocked by the product but, as a header is added, can be put into a separate folder. Emails in this category were considered to have been marked as ham in this test, which may explain the product's relatively low spam catch rate. It was still decent enough for the product to win a VBSpam award though, and with just one false positive, it would be interesting to see what effect a stricter filtering policy would have.

Sophos Email Appliance

SC rate: 99.92%
SC rate (image spam): 99.64%
SC rate (large spam): 99.78%
SC rate pre-DATA: N/A
FP rate: 0.14%
Final score: 99.51

There is a reason why we run an anti-spam test every two months: while one decent performance is certainly a promising sign, what really matters is that a product

manages to perform well repeatedly. With four good sets of results in as many VBSpam tests – each time achieving a final score among the top seven in the test – the *Sophos Email Appliance* certainly satisfies that criterion and adds another VBSpam award to its collection.

SPAMfighter Mail Gateway

SC rate: 98.67%
SC rate (image spam): 97.07%
SC rate (large spam): 93.12%
SC rate pre-DATA: N/A
FP rate: 0.14%
Final score: 98.26

It has been a while since I last needed to log into the admin interface of *SPAMfighter*. That is a good thing, but what is even better is that the product's developers have been working on their product in the meantime and upgrades have been downloaded automatically. This test saw improvements to both the spam catch rate and the false positive rate and, consequently, a significant improvement to the product's final score, winning *SPAMfighter* its sixth consecutive VBSpam award.

SpamTitan

SC rate: 99.07%
SC rate (image spam): 99.96%
SC rate (large spam): 98.24%
SC rate pre-DATA: N/A
FP rate: 0.64%
Final score: 97.17

SpamTitan is one of several products that suffered from more than a handful of false positives in this test. False positives are undesirable and customers are unlikely to accept them unless the spam catch rate of the product is exceptional. *SpamTitan*'s spam catch rate is very good – pushing the product's final score up to above the VBSpam threshold – but the developers will no doubt be spending some time scrutinizing the false positive samples in an attempt to improve the product's position on the VBSpam quadrant.

Symantec Brightmail Gateway 9.0

SC rate: 99.64%
SC rate (image spam): 99.92%
SC rate (large spam): 99.71%
SC rate pre-DATA: N/A
FP rate: 0.00%
Final score: 99.64

A product that manages to increase an already excellent spam catch rate, while eliminating the single false positive that pestered it in the previous test clearly deserves a VBSpam award. *Symantec*'s *Brightmail Gateway* virtual appliance did exactly that, completing this test with the third highest final score and the product's fifth VBSpam award.

The Email Laundry

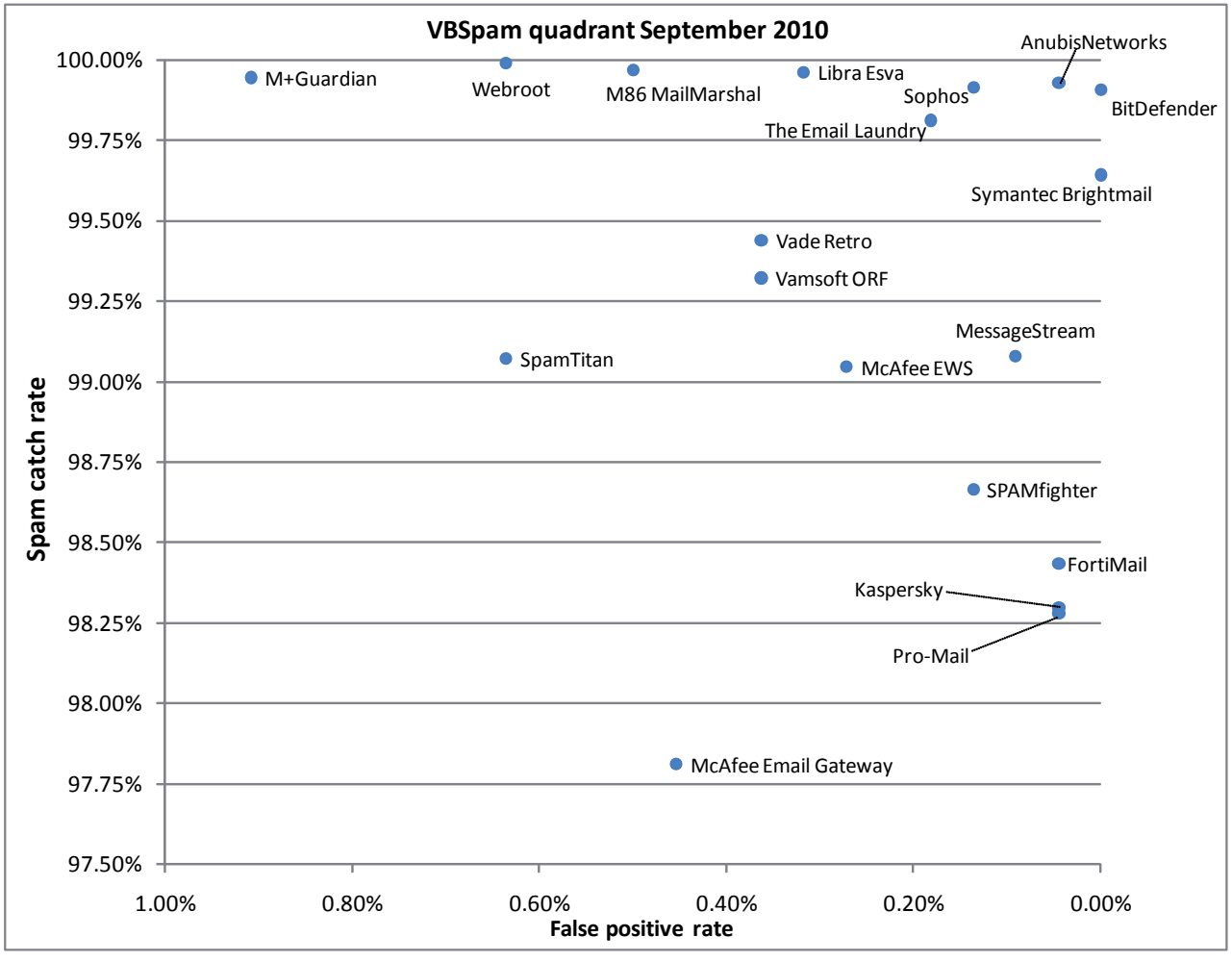
SC rate: 99.81%
SC rate (image spam): 99.96%
SC rate (large spam): 99.71%
SC rate pre-DATA: 94.82%
FP rate: 0.18%
Final score: 99.27

The significant drop in *The Email Laundry*'s pre-DATA catch rate since the last test deserves some explanation. The drop does not necessarily mean that the product's spam-filtering performance has worsened, but that spam has changed and, consequently, blocking on senders' domains and IP addresses wasn't as effective this month as it was in previous months.

What matters to the user is the percentage of spam that makes it to the inbox and this has decreased a fraction. There were a few false positives this time, but not enough to stop the hosted solution from achieving the fifth highest final score and earning a VBSpam award.

Vade Retro Center

SC rate: 99.44%
SC rate (image spam): 99.79%
SC rate (large spam): 98.02%
SC rate pre-DATA: N/A
FP rate: 0.36%
Final score: 98.35



Vade Retro is the market leader in France, but international spam is no problem for the product and it saw its catch rate improve significantly this month. With a small number of exceptions, legitimate email in foreign languages proved no problem either. The product thus wins its third VBSpam award in as many tests and with its best results to date.

Vamsoft ORF

SC rate: 99.32%
SC rate (image spam): 99.03%
SC rate (large spam): 97.07%
SC rate pre-DATA: N/A
FP rate: 0.36%
Final score: 98.24

No doubt *ORF*'s developers will be frustrated with two senders in this month's ham corpus, each of which caused four false positives, thus breaking their zero false positive record to date. However, it should be seen as a gentle reminder to all developers that no one can ignore the problem of false positives. Moreover, an improved spam catch rate means the product still achieved a decent final score and thus wins its third VBSpam award.

Webroot Email Security Service

SC rate: 99.99%
SC rate (image spam): 99.96%
SC rate (large spam): 99.34%
SC rate pre-DATA: 63.92%

FP rate: 0.64%

Final score: 98.08

Webroot was one of five products filtering email pre-DATA. It did not block as many emails during this stage as other products did, but this is not a sign that something is wrong with the product: it reflects a choice made by the developers as to where spam is filtered. And with more spam blocked than any other product, *Webroot*'s choice appears to be a good one. Unfortunately, there were a number of false positives this time, but the product easily earned another VBSpam award – its seventh to date.

Spamhaus ZEN

SC rate: 91.36%

SC rate (image spam): 92.03%

SC rate (large spam): 90.78%

SC rate pre-DATA: 91.36%

FP rate: 0.00%

Final score: 91.36

We owe an apology to *The Spamhaus Project*, as a bug on our side caused the *Spamhaus DBL* – the domain blacklist that in previous tests worked so well alongside *Spamhaus*'s *ZEN* blacklist – to fail during the running of this test. This is a shame, especially since *Spamhaus ZEN* – which combines three IP blacklists – performed significantly less well here than in previous tests.

It is important to realize that *Spamhaus* is a partial solution and is not supposed to be applied on its own. And while, together with the *DBL*, it is still recommended that the blacklists be supplemented with a content filter, the *DBL* is supposed to work especially well together with the organization's IP blacklists. What we can see is that during a period when pre-DATA filtering has produced worse results than during previous periods, *Spamhaus* is still a reliable first line of defence against spam – in particular because, once again, no legitimate emails were blocked.

CONCLUSION

For some products, the addition of a second spam stream and/or the new emails added to the ham corpus this month has given them something to work on; developers of other products will be trying to repeat this month's performance. As always, we will be working hard too – perhaps even harder than before. After nine successful tests, the VBSpam set-up is ready to go '2.0'.

Products ranked by final score	Final score
BitDefender	99.91
AnubisNetworks	99.80
Symantec Brightmail	99.64
Sophos	99.51
The Email Laundry	99.27
Libra Esva	99.01
MessageStream	98.81
M86 MailMarshal	98.47
Vade Retro	98.35
FortiMail	98.30
SPAMfighter	98.26
Vamsoft ORF	98.24
McAfee EWS	98.23
Kaspersky	98.16
Pro-Mail	98.15
Webroot	98.08
M+Guardian	97.22
SpamTitan	97.17
McAfee Email Gateway	96.45

For readers of the comparative reviews, little to nothing will change, but the new set-up will ensure greater system stability and allow room for the tests to grow bigger. Moreover, the provision of feedback on products' performance to the participants – most of which has been done manually until now – will be semi-automated, saving considerable time.

The next test is due to run throughout October, with results published in the November issue of *Virus Bulletin*. The deadline for submission of products will be Friday 24 September. Any developers interested in submitting a product should email martijn.grooten@virusbtn.com.

END NOTES & NEWS

The 8th German Anti Spam Summit takes place 15–16 September 2010 in Wiesbaden, Germany. The event – covering a number of spam and other Internet-related topics – will be held mainly in English. Participation is free of charge, but registration is required. See http://www.eco.de/veranstaltungen/7752.htm.

SOURCE Barcelona will take place 21–22 September 2010 in Barcelona, Spain. See http://www.sourceconference.com/.

VB2010 will take place 29 September to 1 October 2010 in Vancouver, Canada. For the full conference programme including abstracts for all papers and online registration, see http://www.virusbtn.com/conference/vb2010/.

A Mastering Computer Forensics masterclass will take place 4–5 October 2010 in Jakarta, Indonesia. For more information see http://www.machtvantage.com/computerforensics.html.

MAAWG 20th General Meeting takes place 4–6 October 2010 in Washington, DC, USA. MAAWG meetings are open to members and invited guests. For invite requests see http://www.maawg.org/contact_form.

Hacker Halted USA takes place 9–15 October 2010 in Miami, FL, USA. For more information see http://www.hackerhalted.com/.

HITBSecConf Malaysia takes place 11–14 October 2010 in Kuala Lumpur, Malaysia. For more information see http://conference.hackinthebox.org/hitbsecconf2010kul/.

RSA Conference Europe will take place 12–14 October 2010 in London, UK. For details see http://www.rsaconference.com/2010/europe/index.htm.

The fifth annual APWG eCrime Researchers Summit will take place 18–20 October 2010 in Dallas, TX, USA. For more information see http://www.ecrimeresearch.org/.

Malware 2010, The 5th International Conference on Malicious and Unwanted Software, will be held 20–21 October 2010 in Nancy, France. For details see http://www.malware2010.org/.

CSI 2010, takes place 26–29 October 2010 in National Harbor, MD, USA. For details see http://www.csiannual.com/.

The Computer Forensics Show takes place 1–2 November 2010 in San Francisco, CA, USA. For more information see http://www.computerforensicshow.com/.

Infosecurity Russia takes place 17–19 November 2010 in Moscow, Russia. See http://www.infosecurityrussia.ru/.

AVAR 2010 will be held 17–19 November 2010 in Nusa Dua, Bali, Indonesia. See http://www.aavar.org/avar2010/.

The VB 'Securing Your Organization in the Age of Cybercrime' Seminar takes place 25 November 2010 in London, UK. The seminar gives IT professionals an opportunity to learn from and interact with security experts at the top of their field and take away invaluable advice and information on the latest threats, strategies and solutions for protecting their organizations. For programme details and to book online see http://www.virusbtn.com/seminar/.

The 26th Annual Computer Security Applications Conference will take place 6–10 December 2010 in Austin, TX, USA. See http://www.acsac.org/2010/.

SOURCE Boston 2011 will be held 20–22 April 2011 in Boston, MA, USA. For more details see http://www.sourceconference.com/.

The 6th International Conference on IT Security Incident Management & IT Forensics will be held 10–12 May 2011 in Stuttgart, Germany. See http://www.imf-conference.org/.

SOURCE Seattle 2011 will be held 16–17 June 2011 in Seattle, WA, USA. For more details see http://www.sourceconference.com/.

VB2011 will take place 5–7 October 2011 in Barcelona, Spain. See http://www.virusbtn.com/conference/vb2011/.

SUBSCRIPTION RATES

Subscription price for 1 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.

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/