AUGUST 2011

virus BULLETIN

Fighting malware and spam

CONTENTS

IN THIS ISSUE

RELOCATION ON BOARD

The idea of a virus carrying (or calculating) a relocation table allows virus writers to use a high-level language and high-level APIs without having to perform tricks with position dependence. Peter Ferrie details two such viruses, Linux/Relax.A and Linux/Relax.B.

page 4

WAR ON BOTNETS

Thanks to the introduction of IPv6, spammers will have access to a much larger pool of unique IP addresses, making it almost impossible for anti-spam companies to maintain useful blacklists. The 'IPv6whitelist.eu' was founded to try to solve this problem. The project assumes that all computers send out spam, unless they have been registered on the whitelist. One of the project's co-founders, Dreas van Donselaar, explains more.

page 15

RELOCK REVIVAL

Through analysis of an old piece of malware, researchers at the University of Verona have found unexpected vulnerabilities in Windows 7 and have demonstrated that with some slight tweaks, W32/Relock will run smoothly on the latest OS.

page 16

ISSN 1749-7027

'In the security industry we know the dangers of sharing personal information and we never stop warning about it.'

Luis Corrons, Panda Security

GOOGLE+, PRIVACY AND THE HUMAN BRAIN

Google+ has become the latest cool new social network that everyone wants to try. More than 10 million users joined the site in the space of just a few days – pretty impressive even for a company like *Google*. *Facebook* and *Twitter* had to wait years to build up such a large number of users. For those who have not yet experienced *Google+*, I would describe it as a mix between *Facebook* and *Twitter*. You can follow whoever you want, you add only the people you want, and you can create different circles of contacts, so that every time you share something you can select the circles with which you wish to share it. You can even go 'public', sharing with everybody, as in *Twitter*.

In the security industry we know the dangers of sharing personal information and we never stop warning about it. The idea of the circles in *Google+* is a simple one and a good one, making it very easy to share what you want with the people you want. (While this is technically also possible on *Facebook*, creating lists on *Facebook* is a rather more difficult process and users tend to share all of their information with all of their contacts.) Overall, we are enjoying the ease of use of *Google+*, and hoping that *Google* has learned from its past mistakes (remember *Buzz*?).

Editor: Helen Martin
Technical Editor: Morton Swimmer
Test Team Director: John Hawes
Anti-Spam Test Director: Martijn Grooten
Security Test Engineer: Simon Bates
Sales Executive: Allison Sketchley
Web Developer: Paul Hettler
Consulting Editors:
Nick FitzGerald, *Independent consultant, NZ*
Ian Whalley, *IBM Research, USA*
Richard Ford, *Florida Institute of Technology, USA*

However, there is still the issue of privacy. The following is an excerpt from the *Google+* terms of service:

> 'By submitting, posting or displaying the content you give Google a perpetual, irrevocable, worldwide, royalty-free, and non-exclusive license to reproduce, adapt, modify, translate, publish, publicly perform, publicly display and distribute any Content which you submit...'

The fact that people are worried about this and are discussing it publicly (e.g. http://www.zdnet.com/blog/projectfailures/google-plus-is-privacy-an-issue/13749), and the fact that the site has some close competitors (e.g. *Facebook*) means that *Google* will have plenty of motivation to fight to win the trust of its users. But in this battle over privacy the key is the user. Ultimately it is the user who decides how exposed he/she wants to be, and in this case, I'm afraid that the battle is already lost: privacy no longer exists for most users.

It doesn't matter how loudly we shout our warnings, all users will reach the point where they will have to decide between privacy and popularity – and in that moment the majority of users seem to opt for popularity. Social networks not only enable people to share thoughts, ideas, photographs, videos and more, but they also make all users equal. Anyone can talk, anyone can listen. This was impossible before the Internet came along, and we now have the opportunity for anyone to spread their beliefs and ideas, to become popular and feed their ego. In some cultures people compare the size of their muscles as a show of status; in social networks users compare the number of followers/contacts they have, or in the case of *Google+*, the size of their circles.

At some point in their lives a lot of people dream of becoming a 'rock star' (or the equivalent in their field) – gaining recognition from the masses – yet very few will ever achieve such a status in real life. Social networks give the opportunity to millions of ordinary people to build up networks of followers – people will throw themselves into *Google+* hoping to become something they are not. And thus the better privacy settings (or ones that are easier to use) will actually translate into less privacy.

Is there a perfect solution? No. The only way to resolve the issue once and for all would be to take away users' freedom and clamp down on privacy – which is not a viable option. We can only try to mitigate the problem through education: give users enough information to enable them to make their own decisions based on a better understanding of the implications of being a 'rock star'. We know that a rock star's life is not all glamour, and our responsibility is to share that knowledge. No more, no less.

NEWS

VB2011 CALL FOR LAST-MINUTE PAPERS

VB is inviting submissions from those wishing to present last-minute papers at VB2011 in Barcelona (5 to 7 October). Those selected to present last-minute papers will receive a 50% discount on the conference registration fee. The deadline for submissions is 8 September 2011 (speakers will be notified no later than 18 days prior to the start of the conference). The full call for papers can be seen at http://www.virusbtn.com/conference/vb2011/call/.

ERRATUM: VBSPAM COMPARATIVE JULY 2011

As a result of human error (and a reviewer in need of a holiday), a number of mistakes regrettably crept into last month's VBSpam comparative review. An incorrect formula was used to calculate products' final scores, making the majority of scores slightly higher than they should have been. With the correct calculations the pass/fail results remain unchanged, although the order of *FortiMail* and *SpamTitan* are reversed in the listings of products by final score, with *FortiMail* now achieving the third highest final score, and *SpamTitan* coming in fourth.

In addition, it was reported in the July review that in the May review, *OnlyMyEmail's MX-Defender* had missed four spam messages. In fact, the product missed only two spam messages on both occasions. *VB* apologises for the errors and any inconvenience they may have caused. The correct results are all now available at http://www.virusbtn.com/vbspam/archive/test?id=164 and in the updated PDF at http://www.virusbtn.com/pdf/magazine/2011/201107-vbspam-comparative.pdf.

PHISHER GETS 12 YEARS+ IN JAIL

A Californian man has been sentenced to 12 years and seven months in prison for his role in a phishing scam that targeted more than 38,000 victims.

34-year-old Tien Truong Nguyen and two accomplices used identities stolen from users of various financial services to set up lines of credit at instant credit kiosks at *Wal-Mart* stores, fraudulently obtaining merchandise worth around $200,000. When Nguyen was arrested in January 2007, police found stolen banking and credit card information belonging to 38,500 victims as well as 20 web templates used to make fake sites for *eBay* and a number of local banks.

Nguyen's accomplices have also received prison sentences in connection with the crime.

Prevalence Table – June 2011 [1]

Malware	Type	%
Autorun	Worm	8.57%
FakeAlert/Renos	Rogue AV	6.25%
VB	Worm	5.96%
Heuristic/generic	Virus/worm	5.40%
Adware-misc	Adware	5.12%
Sality	Virus	4.81%
Conficker/Downadup	Worm	4.43%
Agent	Trojan	3.60%
LNK	Exploit	3.30%
Downloader-misc	Trojan	3.05%
StartPage	Trojan	2.84%
OnlineGames	Trojan	2.61%
Virut	Virus	2.08%
Ircbot	Worm	1.99%
AutoIt	Trojan	1.81%
Zbot	Trojan	1.78%
Slugin	Virus	1.78%
Heuristic/generic	Trojan	1.71%
WinWebSec	Rogue AV	1.53%
Crack/Keygen	PU	1.45%
Dropper-misc	Trojan	1.39%
Crypt	Trojan	1.38%
Kryptik	Trojan	1.36%
Injector	Trojan	1.33%
Iframe	Exploit	1.31%
Virtumonde/Vundo	Trojan	1.30%
Alureon	Trojan	1.26%
Cycbot	Trojan	1.26%
Delf	Trojan	1.23%
HackTool	PU	1.16%
Qhost	Trojan	0.93%
Bifrose/Pakes	Trojan	0.86%
Others [2]		15.15%
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

FRANKIE SAY RELAX

Peter Ferrie
Microsoft, USA

When a virus infects a file, it usually needs to know its loading address so that it can access its variables. This is done most commonly by using a 'delta offset'. There are two main types of delta offset: one is the difference between the location where the virus is currently loaded and the original location where the virus was loaded when it was created; the other is the difference between the location of the variable and the start of the virus code. One alternative method is to append relocation items to the host relocation table (if one exists), so that the addresses in the virus code are updated appropriately by the operating system itself. However, touching the host relocation table can be a complex task, depending on the file format and its location within the file. Another alternative is to carry a relocation table in the virus body and use that to update the addresses to constant values during the infection phase. This is the method that is used by Linux/Relax.A. Linux/Relax.B uses the same method, but in this case the relocation table is generated dynamically.

ALL YOUR BASE

Both viruses begin by registering two signal handlers: one to intercept invalid regular memory accesses (for example, a pointer to unallocated memory), and the other to intercept misaligned addresses or invalid mapped-memory accesses (for example, in a file that is memory-mapped according to its original file size, but which is then truncated by another process). The viruses use the 'int 0x80' interface directly here, because no external symbols have yet been resolved (that is, the host has access to its own symbols, but the virus does not know where they are yet). These two int 0x80 calls are the only ones in the virus code. However, what might be considered a bug exists here – if an exception occurs, then the signal handlers are not restored to their default values. Thus, if an exception occurs in the host (perhaps due to the presence of the virus) and in the absence of another registered signal handler, the signal handlers will run the host entrypoint again – at which point further exceptions seem likely to occur, so the signal handlers will run again (and again, and again). The result is an infinite loop.

At this point, Relax.A finds the image base of its host (Relax.B does this the first time that a function is called in libc) by walking backwards one page at a time, beginning at the start of its code, until 'an' ELF header is found. The 'an' here refers to the fact that that no verification is made that the signature belongs to an actual header, as opposed to the (unlikely) case that the magic value happens to appear at the start of a page. However, the signal handler will intercept any problem relating to fake headers. This lack of verification could exclude certain files from being infected, but this is a minor point. It would be possible to inoculate files against these and similar viruses by placing the fake signature in the right place, but the idea is a little silly. The simplest approach would be to remove the writable flag on the file, since the viruses make no attempt to set it.

GET IT. 'GOT' IT? GOOD.

Once the header is found, the viruses search within the Program Header Table for the segment that contains the virus code. The virus code segment is identified by finding the loadable segment which has the lowest virtual address. The viruses also search for the segment that holds the dynamic linking information. The viruses search within the tags in the dynamic segment for the one that describes the Global Offset Table. If the third entry in the Global Offset Table is non-zero then the viruses use that pointer to search for the segment that holds the dynamic linking information, and then search the tags within that segment for the one that describes the Global Offset Table. The Global Offset Table is a table of pointers. The third value in the table is a pointer to the '_resolve' symbol inside the dynamic linker. If the dynamic linker is not required (because the symbols have all been resolved statically before the process started) then the value at that location will be zero.

In either case, the viruses perform the same search for the dynamic segment and another Global Offset Table, using the fifth entry in the current Global Offset Table. The new table should point into libc. There is no requirement for it to do so, but there is no other library that the loader would need. The viruses search within the tags in the dynamic segment for the symbol table and the string table. In order to call external functions, the viruses need to resolve the external symbols. To do so, they would normally need to know how many symbols exist. They attempt to retrieve the number of symbols from a hash table which is located by searching the tags within the dynamic segment. The viruses know about two hash table tags. If neither of these is found, then they use a hack to calculate it by determining the number of symbol structures that can fit in the symbol table.

It is not known why the viruses determine the number of symbols, except perhaps as a leftover from code that used one of the hash tables correctly (see *VB*, August 2009 p.4 for details of how the hash table is used for symbol resolution). They could perform the symbol search without an upper limit (the symbols that the viruses need ought

to exist), and simply allow the signal handler to trap any error. Since the virus is using a brute-force search anyway, the performance is actually worse with the check for the upper limit than it would be without it. The virus author knows how to use the hash table correctly, but since the viruses recognize two types of hash table, which have different formats, there would need to be two parsing algorithms.

Relax.A uses the gathered information to resolve the address of a single function, mprotect(), while Relax.B uses it for multiple functions. Further, Relax.B waits until a function in libc is called for the first time, and then resolves the address of that function. Thereafter, the proper address is used directly.

PROTECTION DETAIL

Relax.A uses the mprotect() function to make the code section writable. Then the virus parses the relocation table that it carries in its data section, searching for the relocation items that correspond to external symbols. The virus resolves the addresses of the external symbols that it needs in order to infect files. The relocation table is in a custom format, and is produced by a standalone tool that is run after the file is compiled. The details of that tool are not relevant here. After applying all of the required relocations, the virus restores the section attributes, and then calls the main virus body.

Relax.B does not carry a relocation table in its data section. Instead, the virus disassembles its code at runtime and creates the relocation table dynamically. As a result, the mprotect() function is not needed by the virus. The virus has no concerns about the code versus data problem, since the entire virus body is known. Of course, if there were any misinterpretation, it would have prevented the first generation of the virus from running at all, and thus would have been detected instantly.

Since the viruses can run from any address thanks to the relocation table, they are also able to make use of external functions instead of calling the int 0x80 interface directly. In this case, the viruses use the ftw() function to search for files to infect instead of performing the file enumeration on their own. The ftw() function accepts a pointer to a function to be called for each item that is found. The infection routine begins by attempting to open the item and map the first 4KB of the file. The viruses are interested in ELF files that are at least 1KB long (this appears to be an oversight given the size of the map above), but not more than 3MB large. In contrast to all of the previous pieces of malware from this virus author, the viruses are quite strict about the file format:

- the ELF signature must match
- the size of the ELF header must be the standard value
- the file must be 32-bit format
- the file must use little-endian byte-ordering
- the file must be executable
- it must be for an *Intel* 386 or better CPU
- the version must be current
- the size of a program header table entry must be the standard value
- there must not be too many program header table entries
- the program header table must fit within the file
- the ABI must either not be specified or it must be for *Linux*
- the size of a section header table entry must be the standard value
- the section header table must fit within the file
- the file must not be infected already.

The infection marker for the viruses is the last byte of the e_ident field being set to 1. This has the effect of inoculating the file against a number of other viruses (including several by the same virus author), since a marker in this location is quite common.

HOLE-Y WORK

The viruses search the Program Header Table for the interpreter segment. The segment will be present if the file uses dynamic linking. If the segment is found, then the viruses check that it fits within the file, and that the virus code can fit in the space between the end of the interpreter segment and the start of the next page (though there is an off-by-one bug here such that an exact fit will not be accepted). There is an implicit assumption here that the interpreter segment is in the first page of the file. The viruses also search for the loadable segment which has the highest virtual address. If the interpreter segment is not found, then the viruses will try to place their code immediately after the Program Header Table, otherwise they will try to place their code immediately after the interpreter segment. There is an implicit assumption here that the Program Header Table appears before the interpreter segment. If the two elements are swapped, then the virus will overwrite the Program Header Table as a result.

The viruses initially increase the file size by 4KB and create a hole at the chosen location (into which the virus

code will be placed). The bytes between the end of the virus code and the start of the next page are zeroed. There is a bug here in that some bytes in the following page are also zeroed because the length is calculated incorrectly. The viruses add 4KB to the file offset of the Section Header Table, and to the file offset of each of the entries in the table, to compensate for the size of the hole that was inserted.

The viruses find the Program Header Table entry that corresponds to the file header, increase its physical and virtual size by 4KB, and decrease its physical and virtual addresses by 4KB. The physical and virtual addresses of the Program Header Table and the interpreter segment are also decreased by 4KB, to ensure that they remain within the first page of the file. All of the other Program Header Table entries have their physical address increased by 4KB.

The viruses increase the physical and virtual sizes of the loadable segment with the highest virtual address by the size of the virus data. They create a hole at the chosen location, into which the virus data will be placed. The viruses then increase by a corresponding amount the file offset of each entry in the Section Header Table whose previous offset was after the end of the affected loadable segment.

The viruses parse their relocation table again, and for each entry that is not an external symbol in the Relax.A code, or for each entry in Relax.B (Relax.B does not carry relocation information for the external symbols), the viruses apply the appropriate relocation value in the newly infected file, such that all of the addresses are made absolute according to the host loading address. Of course, this requires that the address is constant. It will not work if the file is a position-independent executable. To achieve that would require the use of a delta offset in order to locate the data section in the first place, and then to apply the relocations dynamically to the entire virus body.

Finally, the viruses set the entrypoint to point to the virus code, mark the file as infected, and then allow the search to continue for more files.

CONCLUSION

The idea of a virus carrying (or calculating) a relocation table is great for virus writers. It allows them to write the code in a high-level language, and use all of the high-level APIs that exist, without having to perform tricks with position dependence or having to use Assembler to fiddle with the bits. Best of all, it doesn't make any difference to anti-virus vendors, because whether it's high level or low level, we can still detect it without any trouble.

2011
BARCELONA

VB2011 BARCELONA
5–7 OCTOBER 2011

Join the VB team in Barcelona, Spain for the anti-malware event of the year.

What:
- Three full days of presentations by world-leading experts
- Rogue AV
- Botnets
- Social network threats
- Mobile malware
- Mac threats
- Spam filtering
- Cybercrime
- Last-minute technical presentations
- Networking opportunities
- Full programme at www.virusbtn.com

Where: The Hesperia Tower, Barcelona, Spain

When: 5–7 October 2011

Price: VB subscriber rate $1795

BOOK ONLINE AT
WWW.VIRUSBTN.COM

MALWARE ANALYSIS 2

SPYEYE BOT – AGGRESSIVE EXPLOITATION TACTICS

Aditya K Sood, Richard J Enbody
Michigan State University, USA

Rohit Bansal
SecNiche Security, USA

This paper sheds light on the exploitation techniques that are used by SpyEye to spread infections. Last month, we presented details of the SpyEye malware infection framework [1]. In this article, we continue our research and will discuss the SpyEye bot and the tactics used for stealing information from victim machines.

1. UNDERSTANDING THE SPYEYE BOT

The SpyEye bot [2] has to be installed on the victim machine to become a resident, and it is easiest to install code at ring 3. Conceptually, the OS is divided into four main rings starting from level 0 to level 3. The rings are used to define the access privileges within which code is allowed to execute. Ring 0 protects the kernel. Code that executes in ring 0 has very high privileges so malicious code running in ring 0 can be particularly virulent. In contrast, code executed in ring 3 is in the application layer, and has fewer privileges than ring 0. However, ring 3 rootkits can have significant capabilities. Ring 3 rootkits can use 'CreateRemoteThread', 'VirtualAllocEx' and 'WriteProcessMemory' to inject malicious code into running processes. It is also possible to enumerate and modify files, processes and registry keys. At ring 3 the rootkit can wait silently for keyboard strokes, and direct all the information to a centralized server using an HTTP communication interface. The SpyEye bot effectively runs as a user-mode (ring 3) rootkit as illustrated in Figure 1.

Rootkits are a class of stealthy malware which can be extremely difficult to detect because they sit between applications and the operating system. A rootkit running at ring 3 has the capability to hook application-level processes. The SpyEye bot will hook functions when a system call is initiated from an application. Rather than executing the normal operating system functions, malicious ones are hooked in. Hooking is efficient because dynamically linked libraries have predefined memory addresses and locations. This means that the locations of memory addresses are known and are not dynamically generated. The SpyEye bot hooks specific DLLs such as wininet.dll (*Windows* networking dynamic link library) to tamper with the HTTP data that flows between a victim's browser and the target website. It also hooks the

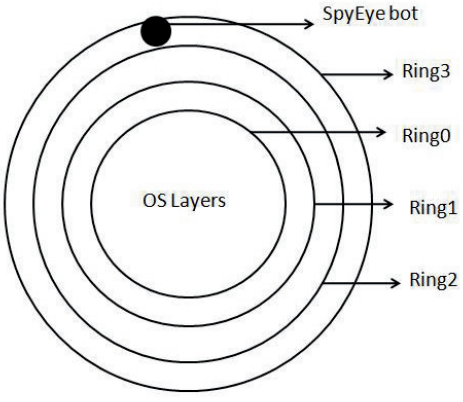

Figure 1: SpyEye ring 3 execution.

nspr4.dll routine, which is a core library used by the *Firefox*, *Netscape* and *Flock* browsers. The SpyEye bot uses *Windows*' built-in Application Programming Interface (API) to execute hooking modules in the context of running applications. Since all browser communication occurs at a user-mode level it becomes easy for SpyEye to perform modifications by manipulating function calls. SpyEye basically performs two major operations on the DLL:

- It completely removes and replaces the executable binary or DLL from the system.
- It performs direct binary modifications in the memory address space.

The hooking procedure is illustrated in Figure 2.

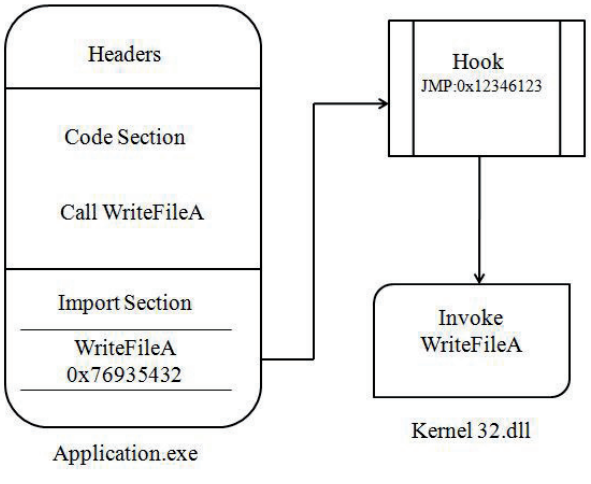

Figure 2: SpyEye bot hooking procedure.

The SpyEye bot monitors the types of applications running on a victim's machine and infects processes using its maliciously designed plug-ins. As mentioned in [1], the SpyEye bot can be customized because it supports plug-ins that are specified within the SpyEye framework. The SpyEye bot can issue commands in real time to infect specific application processes. SpyEye can easily hide processes and even has the capability of escalating the privilege level of a process. In addition, it can hide the TCP/UDP port binding on a victim's machine so that it becomes hard to detect the communication taking place between the bot and the backend collector. The SpyEye bot's rootkit functionality has resulted in robust control over applications which, in turn, make it hard to detect.

2. SPYEYE – TACTICS AND TECHNIQUES

In this section we will discuss specific tactics used by SpyEye to infect victim browsers and machines in order to spread. The tactics that we will discuss have been noticed in multiple versions and illustrate some stability in the SpyEye framework.

2.1 Malicious plug-ins

SpyEye's recent versions starting from 1.2.x have shown a great improvement in designing customized plug-ins based on requirements. In order to support this, the SpyEye frameworks include a Software Development Kit (SDK) which has self-defined APIs based on the framework. The API calls bind a plug-in directly to the bot. There is no restriction on the number of plug-ins that can be used with the SpyEye bot. The design of customized plug-ins has actually diversified the infection pattern of SpyEye. Now it is possible for a single plug-in to communicate directly with the bot and send data back to the database. This design has resulted in modular infections. In order to use plug-ins with the backend collector, the SpyEye framework requires certain modifications to the database. In fact, for plug-ins the collector generates a new database every day. Malicious plug-ins can perform operations based on the attacker's choice.

More technical details about specific plug-ins are discussed in the next section.

2.2 Malicious web fakes

Web fakes are one of the most prominent tactics used by SpyEye to circumvent the normal functioning of the browser. Web fakes are fake authentication windows generated by the SpyEye bot when a user visits a specific bank website. For example, consider a victim who is visiting a *Bank of America* website and his system is infected with SpyEye bot. The bot generates fake windows or pop-ups masquerading as *Bank of America* to fool the user into entering authentication credentials. These are then sent to the backend collector. As the SpyEye bot resides in the system in a stealthy manner, it becomes easy to hook processes. Web fakes are also defined and configured in a raw format as text. The text is interpreted by the plug-in and then commands are issued to the bot to infect browser processes. The web fakes are generated as follows:

- Web fakes have a direct interface with the SpyEye plug-ins. Once the bot is installed on the system, it hooks system DLLs as explained in the previous section.
- As web fakes relate to HTTP communication, SpyEye hooks all the functions in Wininet.dll so that communication through the browser can be modified and monitored. This process works through DLL injection (a technique used to execute code in the memory space of another process by forcing the process to load the attacker-specific DLL). This technique is widely used by virus writers to keep track of the activities in the system and for performing modifications when required. Module hooking and DLL injection work collaboratively to take control of various processes.
- The data is transferred to the processes by the same concept that is used by *Windows* OS, i.e. pipes. Plug-ins issue commands to the SpyEye bot which generates web fakes as described above and transfers data to backend servers via HTTP requests.

SpyEye uses a well-defined SDK for generating web fakes. The following functions are used:

- **DLLEXPORT bool IsGlobal()**: This function is called by a plug-in itself at the start. It provides full access for the plug-in to communicate with all the infected processes so that it is possible for the plug-in to take control of all the infected interfaces directly from the source.
- **DLLEXPORT void Callback OnBeforeLoadPage(IN PCHAR szUrl, IN PCHAR szVerb, IN PCHAR szPostVars, OUT PCHAR * lpszContent, OUT PDWORD lpdwSize)**: This function is called by plug-ins to set a hook on the HTTP/HTTPS request so that the contents of the page can be reported back to the centralized repository for analysing the type of information going out of the network.
- **DLLEXPORT void Callback ProcessContentOfPage(IN PCHAR szUrl, IN PCHAR szVerb, IN PCHAR szPageContent, OUT PCHAR * szOut, IN OUT PDWORD lpdwSize)**:

This function is used to infect the web page dynamically. It again performs a hook immediately before the page is about to render in the browser. It provides an edge to update page contents and injects additional web fakes into banking websites.

- **DLLEXPORT void FreeMem(LPVOID lpMem)**: This function is used to set the allocated resource free.

The list above provides a nice summary of how the SpyEye framework's standard APIs can be used for malicious purposes. Figure 3 shows the list of functions that are used by the SpyEye malware infection framework.

Figure 3: SpyEye SDK functions.

2.3 Anti-virus bot detection

The SpyEye framework has gone through a number of developmental changes since the first version was released. It has added a new anti-virus capability – an anti-virus module for third-party infection and self detection. This module actually enhances the SpyEye operations because the framework is capable of scanning the executables without any outside instruction. It looks quite strange for a malware framework to be using an anti-virus engine. Since the bot has the capability to send the data back to the collector module, it is also possible to scan the third-party executables when an HTTP URL is sent by a bot to the control panel. These features demonstrate the fact that malware is getting cleverer. The virus detection module is presented in Figure 4.

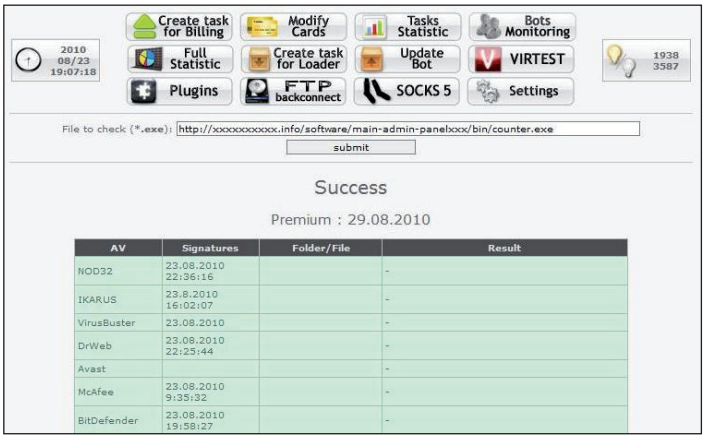

Figure 4: SpyEye anti-virus detection module.

2.4 Bypassing NAT – SOCKS with back connect

SpyEye has a built-in capability for supporting SOCKS connections. This feature was introduced in SpyEye version 1.2.x. When infection takes place in victim machines, it becomes hard to determine whether infected systems have leased IP addresses or systems that are behind NAT or firewalls. This feature helps in setting unanimous port connections through a SOCKS proxy for transfer of data between a victim machine and the control server. Basically, SOCKS is a network protocol supporting HTTP communication between client and server through the implementation of proxy servers to create a tunnel from a private network to the Internet. The SOCKS protocol is platform independent and can be implemented with ease, thereby supporting both *Windows* and *nix environments. This technique gets around firewall security protection because the HTTP traffic is relayed from different ports. The SOCKS proxy acts as a gateway. An IP authentication mechanism and identification protocol features are applied in the SpyEye framework so that the bot works appropriately. In addition, this protocol can be used to set up a stealth tunnel between a SpyEye bot and the centralized servers.

The SOCKS server is started on the same server as that on which the SpyEye framework is hosted. SpyEye uses the code shown in Figure 5 for configuring the SOCKS proxy on the server side.

The bot communicates with plug-ins and data is transferred directly to the SOCKS server, bypassing the

```
// ~~~
$socks = $_GET['s'];
// ~~~
$url = 'http://www.google.com';
$oktext1 = '302 Moved';
$oktext2 = 'Google';
$headers[] = 'Connection: Close';
$ua = 'Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1; .
NET CLR 1.0.3705; .NET CLR 1.1.4322; Media Center PC 4.0)';
$compression = 'gzip';
$timeout = 20;
$cnt = 1;
// ~~~
$process = curl_init($url);
curl_setopt($process, CURLOPT_HTTPHEADER, $headers);
curl_setopt($process, CURLOPT_HEADER, 0);
curl_setopt($process, CURLOPT_USERAGENT, $ua);
curl_setopt($process, CURLOPT_ENCODING , $compression);
curl_setopt($process, CURLOPT_TIMEOUT, $timeout);
curl_setopt($process, CURLOPT_PROXYTYPE, CURLPROXY_SOCKS5);
curl_setopt($process, CURLOPT_HTTPPROXYTUNNEL, 1);
curl_setopt($process, CURLOPT_SSL_VERIFYPEER, 0);
curl_setopt($process, CURLOPT_PROXY, $socks);
curl_setopt($process, CURLOPT_RETURNTRANSFER, 1);
//curl_setopt($process, CURLOPT_FOLLOWLOCATION, 1);
$start = microtime(1);
for ($i = 0; $i < $cnt; $i++) {
      $return = curl_exec($process);
}
$finish = microtime(1);
```

Figure 5: SpyEye SOCKS module.

```
66 61 6C 73 65 00 00 00 74 72 75 65 00 00 00 00  false...true....
43 00 00 00 62 61 64 20 63 61 73 74 00 00 00 00  C...bad cast....
62 61 64 20 6C 65 78 69 63 61 6C 20 63 61 73 74  bad lexical cast
3A 20 73 6F 75 72 63 65 20 74 79 70 65 20 76 61  : source type va
6C 75 65 20 63 6F 75 6C 64 20 6E 6F 74 20 62 65  lue could not be
20 69 6E 74 65 72 70 72 65 74 65 64 20 61 73 20   interpreted as
74 61 72 67 65 74 00 00 32 32 30 20 57 65 6C 63  target..220 Welc
6F 6D 65 20 74 6F 20 53 70 79 45 79 65 20 66 74  ome to SpyEye ft
70 20 73 65 72 76 65 72 0D 0A 00 00 33 33 31 20  p server....331
50 6C 65 61 73 65 20 73 70 65 63 69 66 79 20 74  Please specify t
68 65 20 70 61 73 73 77 6F 72 64 0D 0A 00 00 00  he password.....
32 33 30 20 4C 6F 67 69 6E 20 73 75 63 63 65 73  230 Login succes
73 66 75 6C 0D 0A 00 00 35 33 30 20 49 6E 76 61  sful....530 Inva
6C 69 64 20 6C 6F 67 69 6E 20 6F 72 20 70 61 73  lid login or pas
73 77 6F 72 64 0D 0A 00 32 31 35 20 57 69 6E 64  sword...215 Wind
6F 77 73 5F 4E 54 0D 0A 00 00 00 00 32 31 31 2D  ows_NT......211-
46 65 61 74 75 72 65 73 3A 0D 0A 20 50 41 53 56  Features:.. PASV
0D 0A 32 31 31 20 45 6E 64 0D 0A 00 32 30 30 20  ..211 End...200
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00 00 00 00 D8 24 04 10 70 C6 03 10 D1 00 00 00  ....Ø$..pÆ..Ñ...
52 53 44 53 D5 9B C7 5B CC 7A A2 4C 8F 78 56 68  RSDSÕ›Ç[Ìz¢L.xVh
29 47 EF CB 01 00 00 00 43 3A 5C 44 61 74 61 5C  )GïË....C:\Data\
44 6F 63 75 6D 65 6E 74 73 5C 4D 79 20 50 72 6F  Documents\My Pro
6A 65 63 74 73 5C 43 43 5C 43 61 72 64 4E 65 74  jects\CC\CardNet
5C 50 72 6F 67 73 5C 43 6C 69 65 6E 74 5C 53 70  \Progs\Client\Sp
79 45 79 65 5C 70 6C 75 67 69 6E 73 5C 42 43 5C  yEye\plugins\BC\
43 6C 69 65 6E 74 5C 52 65 6C 65 61 73 65 5C 66  Client\Release\f
74 70 62 63 2E 70 64 62 00 00 00 00 00 00 00 00  tpbc.pdb........
00 00 00 00 00 00 00 00 08 20 04 10 F0 AB 03 10  ......... ..ð«..
00 00 00 00 00 00 00 00 02 00 00 00 00 AC 03 10  .............¬..
0C AC 03 10 C0 B0 03 10 00 00 00 00 08 20 04 10  .¬..À°....... ..
01 00 00 00 00 00 00 00 FF FF FF FF 00 00 00 00  ........ÿÿÿÿ....
```

Figure 6: FTP server for SpyEye back connect.

NAT infrastructure. It works with insecure and secure connections such as HTTP and HTTPS respectively. Logging is also supported by the SOCKS server. Similarly, SpyEye supports an RDP and FTP back connect module. (Figure 6).

2.5 Web injects – manipulating the content

SpyEye is capable of injecting content into banking pages in real time as they are displayed. A number of techniques have been discussed in previous sections of this paper. The web injections [3] are more destructive in practice because they modify the content of the web page before the actual web page is rendered in the browser. The web injections occur on the client side. For example, a user with a SpyEye-infected machine visits a banking site. As soon as the website is about to load in the browser, the SpyEye bot injects custom content into the same web page. As a result, the content looks in line with the real web page, thereby implying the authenticity of the rendered content in the web browser. *Internet Explorer* and *Firefox* are injected in an extensible manner by SpyEye. Figure 7 shows the content injection.

2.6 Screen shot stealers and screen scrapers

SpyEye has an inbuilt key-logging mechanism that is perfectly designed for logging keystrokes from the victim machine. Basically, we have noticed form-grabbing activities by the SpyEye bot instead of complete keyboard hooking. In the form grabbing, all the content from HTML forms is stolen during the POST request and the bot sends that information back to the backend database. Form grabbing is one of the predominant features of SpyEye because all the user's monetary transactions and login activities take place via form submission. In order to perform efficient form grabbing, the SpyEye bot hooks into the browser dynamic link libraries and hooks the data submission functions so that sensitive information can be stolen from the victim machine. In addition to this, SpyEye also uses a screen scrapping feature in which the bot takes snapshots of the victim machine as the user is inputting sensitive information and sends them to the backend server [4] in a compressed format. Figure 8 shows how the snapshots of the system are retrieved at the main panel.

2.7 X.509 certificates stealer

SpyEye has an inbuilt plug-in that is primarily designed for stealing X.509 certificate information from victim machines. Basically, this is accomplished through Man-in-the-Browser (MitB) attacks. The SpyEye bot sits in between the browser and the destination domain, and since it has already hooked the HTTP communication interface, the bot is able to extract information from the certificates. This is done so that the bot can communicate with the legitimate domain without any hassles from the

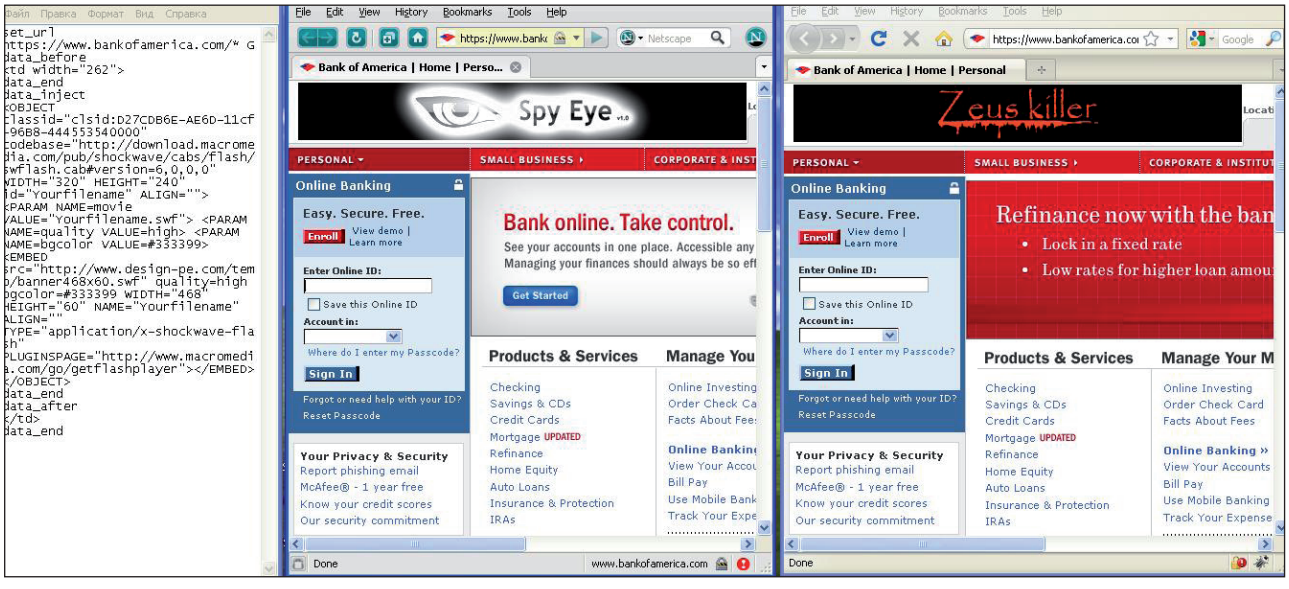

Figure 7: SpyEye's web inject in action.

Figure 8: SpyEye – screenshot stealer.

```
$bot_guid = $_POST['bot_guid'];
$certdatestart = $_POST['certdatestart'];
$certdateend = $_POST['certdateend'];
$data = $_POST['data'];
$limit = $_POST['limit'];
$showUseless = $_POST['showUseless'];

if (strlen($certdatestart) && strlen($certdateend)) {
      list($day, $month, $year) = split('[ :\/.-]', $certdatestart);
      $tstamp0 = "$year-$month-$day 00:00:00";
      list($day, $month, $year) = split('[ :\/.-]', $certdateend);
      $tstamp1 = gmmktime (0, 0, 0, $month, $day + 1, $year);
      $tstamp1 = gmdate('Y-m-d H:i:s', $tstamp1);
      $sqlp1 = " AND (UNIX_TIMESTAMP( cert.date_rep ) >=
      UNIX_TIMESTAMP('$tstamp0') AND UNIX_TIMESTAMP( cert.date_rep )
      < UNIX_TIMESTAMP('$tstamp1'))";
}
if (strlen($bot_guid))
      $sqlp2 = " AND cert.bot_guid LIKE '%$bot_guid%'";
if (strlen($data)) {
      if ( strpos($data, '*') !== false ) {
            $data = str_replace( '*', '%', $data );
      }
      if ($data{0} != '%')
            $data = '%' . $data;
      if ($data{ strlen($data) - 1 } != '%')
            $data .= '%';
```

Figure 9: Firefox – certificate collector.

victim browser. Apart from this, stolen certificates can also be used to generate fake certificates for malicious purposes. Figure 9 shows an implementation of the plug-in that steals certificates from the *Firefox* communication interface.

2.8 Distributed denial of service

SpyEye version 1.3.x has implemented the concept of distributed denial of service through inbuilt plug-ins. This functionality has been noted in the latest versions of the malware as a protection against anti-SpyEye detectors. Using this plug-in, the command and control server forces the installed bots to start sending packets against anti-detectors. Overall, the DDoS is achieved by harnessing the power of the victim machine through installed SpyEye bots.

Figure 10 shows how exactly the DDos.cfg plug-in is configured in SpyEye. This plug-in is not very effective at

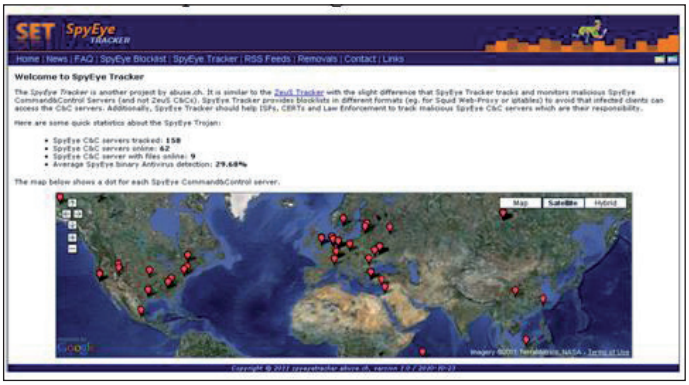

Figure 10: DDoS configuration.

conducting denial of service attacks in a distributed manner, but the design could improve and become more robust in the future.

CONCLUSION

In this article, we have presented SpyEye's most frequently used techniques. All the variants of SpyEye effectively use these tactics to exploit victim machines for malicious purposes.

The points discussed in this article demonstrate the advancements that have taken place in third-generation botnets.

REFERENCES

[1] Sood, A.K.; Enbody, R.J.; Bansal, R. SpyEye malware infection framework. http://www.virusbtn.com/virusbulletin/archive/2011/07/vb201107-SpyEye.

[2] Sood, A.K.; Enbody, R.J. Hack In The Box – Spying on SpyEye. http://secniche.blogspot.com/2011/05/hackinthebox-ams-spying-on-spyeye.html.

[3] Malware at Stake (SpyEye & Zeus) Web Injects – Parameters. http://secniche.blogspot.com/2011/07/spyeye-zeus-web-injects-parameters-and.html.

[4] Malware at Stake, SpyEye Backend Collector. http://secniche.blogspot.com/2010/08/spyeye-backend-collector-generating.html.

FEATURE 1

A NEW TREND IN EXPLOITATION

Abhishek Singh, Johnathan Norman
Alert Logic, USA

Understanding the exploitation of a vulnerability is important both for product security teams and for the research teams that generate signatures for network intrusion prevention/detection (NIS) devices.

Product security teams need to gain an understanding of the vulnerable part of the code and provide an update, or patch, to fix the vulnerability. In order to create a signature for an intrusion prevention/detection device, researchers must gain an understanding of the vulnerability and then derive the conditions that can lead to it being exploited. When deployed, the signature will protect the vulnerable application from being exploited via the network.

In order to develop a signature for traditional types of vulnerabilities such as buffer overflows, format string vulnerabilities and integer overflows, we have to refer to the vulnerable code itself. Once the vulnerable portion of the code has been identified, it can be used to determine the conditions that will lead to its exploitation, and a signature can be generated based on those conditions. Recently, however, we have observed a new type of exploitation technique that makes use of improper implementation of protocol specifications. This type of exploitation requires a different type of analysis.

WHAT MAKES THIS TREND DIFFERENT?

Even though improper implementation of protocol specifications can lead to traditional, well-defined classes of exploitation such as integer overflow, buffer overflow, denial of service attacks and remote code execution, exploitations arising in this manner can be classified as a new trend for the following reasons:

- Rather than analysing the vulnerable source code to derive the conditions that can be used to create a signature for NIS devices, the proprietary protocol specification document must be consulted. This document states the values for the arguments of a command as well as when and how the values can be used. The NIS signature is created based on the information provided in the documentation.
- Traditionally, when testing for security issues, product test teams find a vulnerable function and then generate various inputs for the function to test whether it can be exploited. In the case of vulnerabilities that arise due to the improper implementation of proprietary protocol specifications, test cases must be constructed according

to the values set by the protocol specifications and not by the exploitation techniques.

- There have been repeated occurrences of exploitations taking advantage of the improper implementation of protocol specifications, as outlined in Table 1.

In the following sections we will present analyses of two of the vulnerabilities listed here, CVE-2011-0654 and CVE-2009-3103, in each case looking first at the source code and then using the protocol specifications to derive the conditions upon which to base an NIS signature.

CVE ID of the vulnerability	Trigger conditions
CVE-2009-3103	Vulnerable condition is triggered due to the improper implementation of the Server Message Block (SMB) command negotiate protocol.
CVE-2009-3676	A denial of service vulnerability exists in *Microsoft Windows*' Server Message Block (SMB) implementation. Specifically, the vulnerability is due to improper parsing of the NetBIOS Length parameter. If the Length field does not match the size of the following SMB message, an infinite loop can result, causing a denial of service condition.
CVE-2010-0270	Vulnerability in improper implementation of the SMB Trans2 response for command type 0x32. If the sum of the values of the 'Data Count' and 'Data Offset' fields is larger than the total length of the SMB message header and the SMB message data structure, then an attack is underway.
CVE-2010-0477	Vulnerable condition is triggered when the message size is greater than the amount of data.
CVE-2011-0476	Vulnerability in improper implementation of the SMB response with command type =0x25. If the value of the 'TotalDataCount' field is larger than the actual length of the message data, the exploit is underway.
CVE-2011-0654	Vulnerable condition is triggered due to the improper implementation of the server name in *Microsoft Windows* Browser Protocol.

Table 1: List of vulnerabilities caused by the improper implementation of protocol specification documents.

ANALYSIS OF MS11-019 CVE-2011-0654

CVE-2011-0654 was a zero-day browser election vulnerability [1]. It exists in the way that the Common Internet File System (CIFS) browser protocol implementation [2] parses malformed browser messages. *Microsoft* has issued a patch for the vulnerability.

Figure 1 shows the packet capture when the exploit code is executed. It is obvious from the capture that the server name is the malicious field and is sending malicious bytes for the exploitation of the vulnerability.

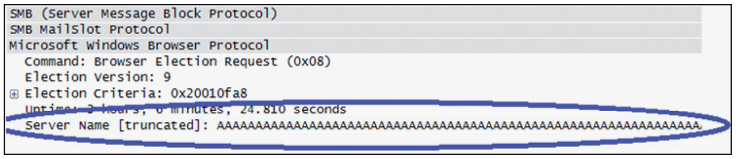

Figure 1: Packet capture for CVE-2009-3103 when malicious bits are sent over the wire.

When an overly long ServerName field is encountered, the code in the _BowserWriteErrorLogEntry function allocates a fixed buffer of size 112 (0x70) bytes to store multiple fields. Once the server name is copied, the remaining buffer size is calculated as

Remaining_Buffer_Size = 112 – (length (Server_Name)*2)

Hence a ServerName field that is 56 bytes long (including the NULL terminator) would cause the remaining buffer size to be zero.

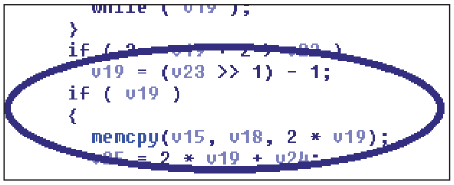

Figure 2: Vulnerable section of the code.

Later in the code, as shown in Figure 2, the variable v19 is used in memcpy. As shown in Figure 2, v19 is equal to v23>> -1. V23 is the variable Remaining_Buffer_Size. If the variable v23 is decreased by one, v19 being an unsigned integer becomes = 0xFFFFFFFF. The check 'if (v19)' becomes true and a large amount of data is copied to memcpy, leading to an overflow. So, from the analysis of the code, it can be inferred that in order to prevent such an overflow the sever name must be less than 56 bytes.

However, if we refer to *Microsoft*'s protocol specification, it can be seen that the server name must, in fact, be less than 16 bytes and must be null terminated:

> '**ServerName (variable)**: MUST be a null-terminated ASCII server name and MUST be less than or equal to 16 bytes in length, including the null terminator.' [2]

In this case a signature for an intrusion prevention/detection device can be created that checks the length of the server name in the Browser Election request – a server name that is greater than 16 bytes indicates exploitation of the vulnerability.

From the above analysis it can be seen that referring to the proprietary protocol specification is very important when creating an NIS signature. The document provides the correct values, whereas the analysis of the source code provided a value which would have been incorrect to base an NIS signature upon.

ANALYSIS OF CVE-2009-3103

Let's look at the analysis of another zero-day vulnerability, CVE-2009-3103. This is triggered due to an array indexing error while parsing SMB packets containing SMB2 dialect with an SMB Negotiate message [3].

In the source code the Process ID High (PIDHigh) value is used, without any bounds checking, to index an array of function pointers. This function pointer is later dereferenced and called for further processing. So, by using the process ID field, an attacker can index into an array of function pointers triggering the vulnerable conditions.

The analysis of the code does not provide an authoritative condition that can be used to author an NIS signature. However, if we check the publicly available proprietary protocol specification document for the legitimate values for PIDHigh, it states that for a 16-bit process ID the value must be 0 and for a 32-bit process ID the value is as per the CIFS/1.0 protocol specification:

> '**PIDHigh (2 bytes)**: This field MUST give the 2 high bytes of the process identifier (PID) if the client wants to use 32-bit process IDs, as specified in [CIFS] section 2.4.2. If a client uses 16-bit process IDs, this field MUST be set to zero.' [4]

Further referring to the CIFS protocol [5], the PIDHigh value is used only in the NtCreateAndX request. The command value of NtCreateAndX is 0xa2. Since the values are used in NtCreateAndX, for the command 'Negotiate (0x72)' the value of PIDHigh must be 0.

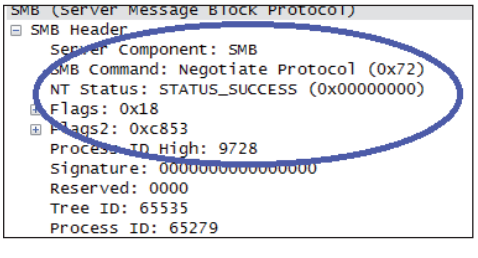

Figure 3: Packet capture for CVE-2009-3103.

Hence for network-level inspection devices, it must be assumed that if the value of the SMB command is 0x72, and if the value of PIDHigh is not equal to 0, the bits on the wire are an exploit. Once again, this case demonstrates that if we refer to the protocol specification documents, the conditions used to author an NIS signature can be derived in an authoritative manner.

INFERENCE DRAWN

Protocol specifications and/or RFCs generally define the structure of a protocol and the fields that are associated with it. In some cases proprietary protocol specification documents (or RFCs) can also define 'safe' values, including when and how these values are used.

The ideal approach to understanding any class of vulnerability is to reverse the code and perform an analysis of the vulnerability and then derive the conditions for a signature. The new trend of exploitations which arise due to the improper implementation of RFC/protocol specifications require a complete change in the thought process of a security researcher while performing the vulnerability analysis. The new trend will force security researchers to refer to protocol specifications, since they might contain the right values to author a signature.

In some cases, such as CVE-2011-0654, analysis of source code alone can lead to incorrect values being included in NIS signatures. In cases such as CVE-2009-3103, source code analysis is not sufficient to determine authoritative conditions for an NIS signature.

For product security testing teams, a complete change in the design of test cases is required. Fuzzing tools will have to be designed in such a way that the tool streams the values enforced by the protocol specifications. If fuzzers use the traditional technique of finding the vulnerable function and generating various inputs to test if it can be exploited, they will miss exploitations due to the improper implementation of protocol specifications.

REFERENCES

[1] http://www.securityfocus.com/bid/46360/exploit.

[2] http://msdn.microsoft.com/en-us/library/cc224428(v=prot.10).aspx.

[3] http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2009-3103.

[4] http://msdn.microsoft.com/en-us/library/cc246231%28v=PROT.13%29.aspx.

[5] http://www.microsoft.com/about/legal/protocols/BSTD/CIFS/draft-leach-cifs-v1-spec-02.txt.

FEATURE 2

IPV6 MAIL SERVER WHITELIST DECLARING WAR ON BOTNETS

Dreas van Donselaar
SpamExperts, The Netherlands

Although estimates differ between sources, around 95% of all email traffic currently consists of spam. Despite there having been some decreases in spam volumes recently[1], we are not likely to see a significant drop in spam levels any time soon, as spammers still earn a lot of money from their activities. Highly organized gangs operating from numerous countries make a professional living from sending spam and invest serious amounts of money and resources into their businesses to remain on top of their game – just like any legitimate industry. Since spam is a global problem, it can be difficult to track down and take legal action against these gangs.

A large proportion of spam is sent out by botnets. A botnet consists of a network of many infected computers that are controlled by a 'bot master' and may be used for any type of online crime, including sending spam. Botnets are generally made up of home computers whose owners do not realize that their machines have been infected and are being used as part of a botnet. Such infected machines can send out thousands of spam messages per day, and until the malware is cleaned from the computer, or the machine is disconnected from the Internet, it will continue to send spam via the control of the bot master.

Within the security industry, the spam problem is tackled in a number of different ways. For example, anti-virus companies provide software to detect the malware responsible for turning computers into bots, and firewall providers attempt to identify and block suspicious traffic coming from the computer. Anti-spam companies, meanwhile, resort to different methods to try and stop these bots from delivering spam.

Each computer on the Internet is assigned a unique number which is used for all Internet communication, the so-called IP address. When a computer visits a website or sends an email, the IP address is revealed to the destination server. Anti-spam companies monitor the activity of these IP addresses, and if they suddenly detect a stream of spam from a particular address, they add it to an IP blacklist. All email from that IP address will then be blocked, stopping the flow of spam to the recipient server. Removal of the IP address from the blacklist must be requested manually once the spam issue has been resolved.

There are a few problems with this method. First, spam has to be detected before the system can make a proper judgement as to whether or not to block the IP address. Spammers often send out small bursts of spam messages to try and keep the volume below the threshold that would trigger such a listing. Secondly, spammers can keep infecting new machines to gain access to new IP addresses which have not yet been listed.

In total there are around four billion IP addresses in IP version 4. Because there are an increasing number of devices on the Internet in need of an IP number, this pool of addresses is rapidly running out and will soon be exhausted. To get around this problem, a new version of the numbering system (IP version 6) has been introduced. To avoid running out of IP space again, this new standard will create a pool of approximately 340 undecillion (2^{128}) addresses. It is hard to comprehend such an enormous number, but to give an idea, it's greater than the number of stars in the sky.

Thanks to the introduction of IPv6, spammers will have access to a much larger pool of unique IP addresses, making it almost impossible for anti-spam companies to maintain useful blacklists. It will be a lot harder to accurately stop spam at an early stage, because there will be too many different IP sources from which spam can be delivered. Blacklists will grow too large for computers to handle efficiently, and spammers will be able to switch to a new address as soon as the current one gets blocked.

IPV6WHITELIST.EU

The not-for-profit project 'IPv6whitelist.eu' was founded in 2010 in The Netherlands by Dreas van Donselaar (*SpamExperts*), Ruud van den Bercken (*XS4ALL Internet/Stay-Secure*) and Raymond Dijkxhoorn (*Prolocation/SURBL*) to try to solve the quantity problem IPv6 introduces. Until now the mechanism has been to assume that computers don't send out spam, and then to blacklist them when they do. The Ipv6whitelist.eu project, however, assumes that all computers send out spam, unless they have been registered on the list. All IPv6 addresses are simply blacklisted unless they appear on the whitelist – addresses must be added to the whitelist manually via a simple web form.

The project is controversial because it goes against the openness of the Internet by obliging mail server administrators to register in a central database before sending out email. The situation is turned around and instead of the recipient deciding whether or not to accept email from a specific system, the sender is now obliged to specify that he/she would like to send email from a specific system.

The initiative will only succeed if sufficient recipient mail servers enforce the requirement for senders to join the IPv6 whitelist. If not enough recipients enforce the rule,

[1] See http://www.virusbtn.com/virusbulletin/archive/2011/07/vb201107-news1.

senders will simply ignore it and not bother registering their mail servers. The project currently only applies to IPv6 addresses assigned to the Netherlands. Thanks to the close collaboration of many IPv6-enabled access providers and web-hosting companies in the Netherlands, a critical mass of enforcing recipients has quickly been established, ensuring that IPv6 senders are forced to comply.

Email from any mail server in the Netherlands which is not yet registered to the central database is automatically temporarily rejected by recipient mail servers until the sending server has been registered (free of charge) via the API or website. More often than not, unregistered servers are hacked computers which are being used to send spam without their owners' knowledge.

In the long term, we foresee a significant reduction in spam originating from the Netherlands. Because this is a completely cost-free system, there has been little resistance from the market – people understand that the small inconvenience of having to register their mail servers resolves a major issue on the receiving side, keeping incoming spam under control.

The system is vulnerable to abuse though, since spammers could simply start registering their mail servers on the list as well. Besides verifying that the registration has been made by a human, there is no further control or judgement on an IP whitelisting. The IP netblock owner does have the option to delist certain IPs, if required. However, we do not envisage a problem if spammers start registering IPv6 addresses – even if there are millions of bad registrations that is still a very small number compared to the overall IPv6 pool. Thanks to that reduction, anti-spam companies can easily keep track of the reputation of sending servers as they currently do.

At the moment the volume of spam is so high that anti-spam companies will continue to play a vital role. The initiative will ensure that the problem remains manageable, not only now but also in the future.

All IPv6whitelist.eu software, APIs, systems and data are open to the public. There is no commercial incentive and the association is run by volunteers. Since the rollout of IPv6 has only just started, the effect of the project on live mail streams is currently minimal. However, because of the early launch, easy adoption on the recipient side has been ensured, and it is hoped that many more countries will either join the project or launch similar initiatives. A critical mass on the recipient side is the only requirement to be able to force senders to make changes to their sending behaviour – and there are no technical limitations or restrictions involved in the registration process, meaning that there are no barriers to making this a new standard requirement for email senders.

FEATURE 3

RELOCK-BASED VULNERABILITY IN WINDOWS 7

Andrea Fortunato, Marco Passuello, Roberto Giacobazzi
University of Verona, Italy

The new security features introduced with *Windows 7* prevent the relocation of an executable to a fixed address. Their aim is to make buffer overflow attacks harder, but they indirectly make the use of OS relocation procedures for hiding or obscuring information in files impossible, since a variable relocation address makes it impossible to reconstruct information while relocating executables. In this paper we present a *Windows 7* vulnerability related to the PE Header ImageBase field, which forces a relocation to a fixed address. This vulnerability is exploited to make an old obfuscation technique compatible with *Windows 7*. The technique, which is based on memory relocations, was first implemented in the W32/Relock virus.

INTRODUCTION TO RELOCK

In 2007 the virus writer roy g biv introduced W32/Relock for *Windows XP/2000* to demonstrate a new obfuscation technique called 'virtual code', based on a peculiar use of memory relocations for code stealthiness and polymorphism [1–3].

This malware does not have self-replicating features or network capabilities; it is an executable file infector because it only affects executable files (excluding libraries) recognized by detecting the Portable Executable (PE) format. Once executed, the virus infects the targets contained in its directory (and recursively in all subdirectories) but it does not reside in memory after completing its operations.

Designed as a proof of concept, the virus was not intended to be released into the wild to cause any damage. As such, it does not contain a harmful payload but only a PE header and a particular relocation table which represents an encryption of the malware code (Figure 1).

At run time, the OS will apply the relocation items specified in the table, decrypting the code and restoring the original malware. This avoids the use of a plain de-obfuscation procedure inside the virus, transferring the de-obfuscation duty to the OS instead, and making the malware highly stealthy and hard to catch by signature analysis.

VIRTUAL CODE TECHNIQUE

The virtual code obfuscation technique relies on a particular behaviour of the dynamic linker present in *Windows*

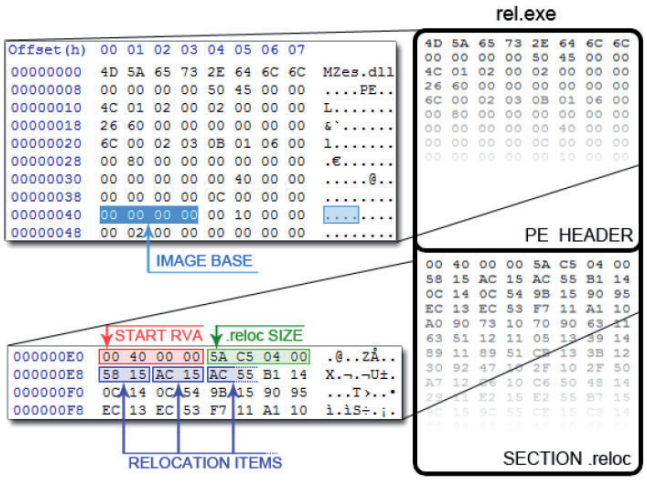

Figure 1: The file rel.exe consists of a shrunken PE header with the ImageBase set to zero and a huge relocation table. There is no executable code.

XP, which relocates executables with an ImageBase set to 0 (invalid) at the constant address 0x00010000. This behaviour is an essential condition in order for the obfuscating algorithm to work properly (Figure 2).

The basic idea behind virtual code can be summarized in a sequence of decrements which are applied to the .code section in order to set its bytes to zero, whilst symmetrically inserting relocation items of types 1, 5 and 9 in the relocation table of the virus. For simplicity, let's focus on the relocation type 1, which causes the addition of the highest 16 bits of the difference between the base address and the image base to the randomly chosen target byte.

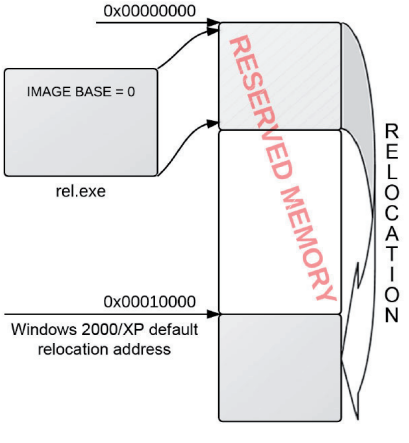

Figure 2: Relocation mechanism occurring in Windows XP/2000 when the ImageBase is set to zero: the executable is relocated to 0x00010000.

Since this delta is always 0x00010000, the dynamic linker will always apply a unitary increment, and for this reason the obfuscation algorithm decrements the target byte by one for each relocation item successfully created. The diagram in Figure 3 illustrates this procedure.

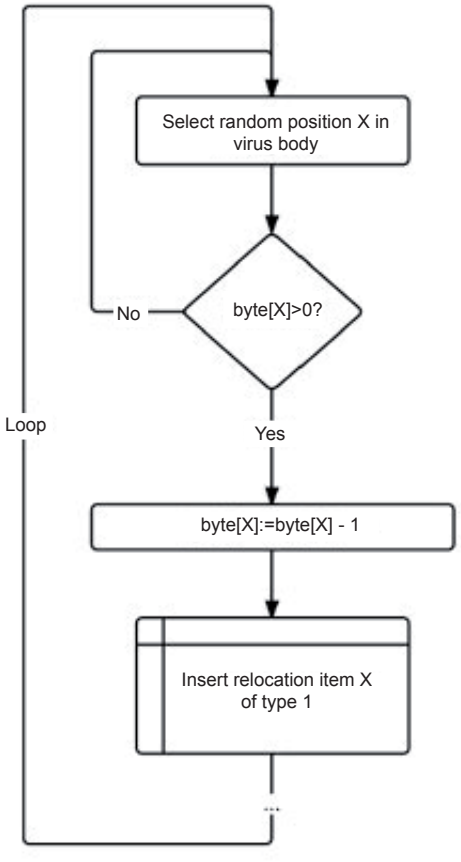

Figure 3: Flow diagram of a simplified version of virtual code.

The following pseudo-code represents the core mechanism of the obfuscation procedure:

```
1 while (virus code contains non -null byte){
2   generate random number R
3   if R < code size {
4      if byte[R] != 0 {
5         rel_item = R + 1000h
6         relocation_table .append(rel_item)
7         byte[R]= byte[R]--
8      }
9   }
10 }
```

The loop is executed until all the virus code is completely zeroed. For each iteration, when a valid position is found, the instruction at line 5 creates a relocation item of type

1 by adding 0x1000 to the previously selected position, and at line 7 the target byte is decremented to reflect the relocation item stored in the relocation table by the instruction at line 6.

VIRTUAL CODE OBFUSCATION IN WINDOWS 7

The advent of *Windows 7* has seen the introduction of effective security measures that can block relocation-based obfuscation techniques: the execution of files with ImageBase 0 has been disabled, with the error message 'The parameter is incorrect' appearing. We still need a fixed memory relocation but the presence of the Address Space Layout Randomization (ASLR) prevents this, randomizing the relocation address of the executable within its virtual space. ASLR techniques are typically used to prevent buffer overflow attacks [4] and their effectiveness relies on there being only a very small chance that an attacker could guess where randomly placed data and code are located. Security is increased by increasing the search space: the more entropy is present in the random offsets, the more effective address space randomization becomes. Entropy is typically increased by raising the amount of virtual memory area space over which the randomization occurs. It is widely believed that randomizing the address space layout of a software program prevents attackers from using the same exploit code effectively against all instantiations of the program containing the same flaw. To defeat the randomization, attackers must successfully guess the positions of all their targets, which is made harder by the randomization of the address space layout each time the program is restarted.

The effect of ASLR on Relock is to make virtual code unusable: it is no longer possible to force the relocation of an executable to a fixed address and therefore, without a constant offset, it is no longer possible to use virtual code to polymorphically hide the viral code in the relocation table. The only possible solution would be to include a plain-text procedure in the dropped virus which would patch the virus code at runtime to compensate for the difference between an assumed loading address and the real base address selected by ASLR.

FORCING FIXED ADDRESS RELOCATIONS

The search for possible solutions to the countermeasures used in *Windows 7* led to the analysis of the aligned values for the ImageBase inside the kernel memory space. When using *OllyDbg* to debug an executable with an ImageBase set to the aligned upper bound (0xFFFF0000) of the kernel memory space, we observed an unexpected behaviour of the

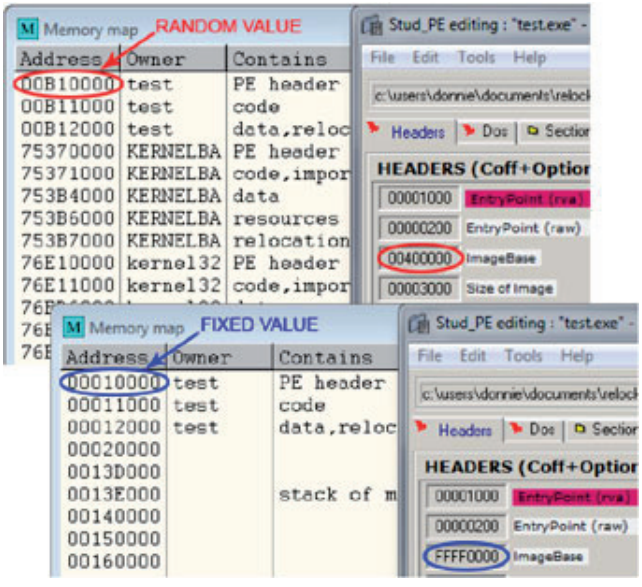

Figure 4: Canonical values of ImageBase (with ASLR enabled) produce relocations to random addresses, whereas the value 0xFFFF0000 forces relocation to the fixed address 0x00010000.

OS (Figure 4): the program is relocated to the fixed address 0x00010000, thus obtaining the same vulnerability as that present on *Windows XP* when the ImageBase is set to 0.

Subsequent analysis showed that the same effect can be obtained using any value inside the interval [0x7FFE0000;0xFFFF0000]: all aligned values for the ImageBase in this range cause the relocation of the executable to 0x00010000. This behaviour exists even with ASLR enabled. Figure 5 shows how relocation addresses grow almost linearly, except for a local randomness limited to the 256 positions underneath the ImageBase. This holds until the value 0x7FFE0000 is reached; from that moment forward all values cause fixed relocations to 0x00010000.

EXPLOITING THE IMAGEBASE: RELOCK 2.0

The knowledge of those particular values for the ImageBase provides a method to obtain, at each run, the relocation of the executable to a fixed address. It is therefore possible to reuse virtual code on *Windows 7*, with its advantages in terms of stealthiness. Considering the characteristics of this obfuscation technique, particular interest resides in the value 0xFFFF0000, which produces a round delta equal to 0x00010000 - 0xFFFF0000 = 0x00020000.

Thanks to this vulnerability it is possible to fully restore the functionality of the virus, thus obtaining a working

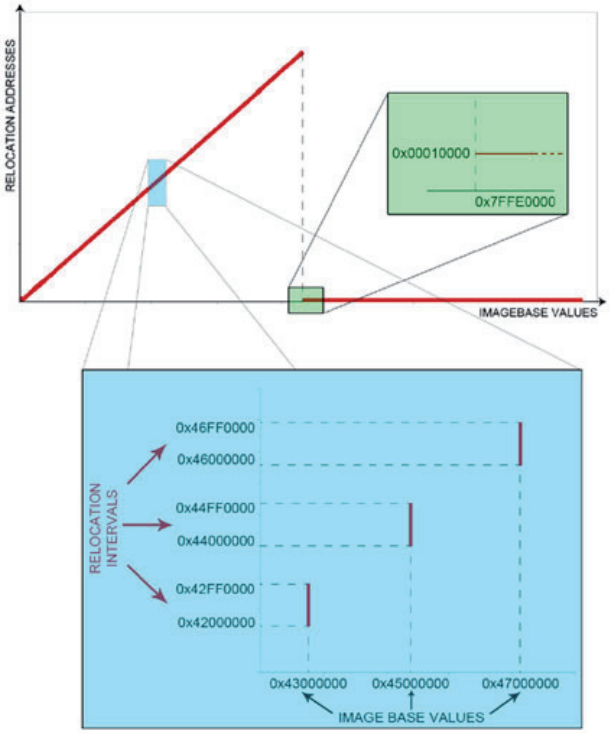

Figure 5: Relocation behaviour in Windows 7, an ImageBase value chosen between 0x7FFE0000 and 0xFFFF0000 causes a fixed relocation at 0x00010000, while lower values cause random relocations within the 256 aligned addresses underneath the current ImageBase.

implementation of virtual code once again. However, this new version cannot use relocation types 1, 5 and 9 which were used in the original Relock since they are no longer supported under *Windows 7*. Instead it uses type 3, which will cause the entire delta value to be added to each relocation item during the relocation phase.

In a similar manner to the original Relock, the obfuscation procedure must decrement the corresponding RVA for each relocation item successfully created. However, the new algorithm is based on relocation items of type 3 and will therefore have to subtract all the 32 bits of the delta (0x00020000), whereas the old Relock would have subtracted only the higher 16 bits of its delta (high[0x00010000] = 1). With this procedure all the four-byte blocks whose hex values are greater than 0x00020000 (null blocks are excluded) will leave a remainder once the obfuscation phase is concluded and all these remainders constitute the .code section of the virus executable. For this reason the .code section of this new version of Relock will contain some bytes (in contrast to the original Relock whose .code section was empty). These bytes will be polymorphically different for each dropped version of the virus thanks to the presence of random decisions relating to the choice of the blocks to decrement. See Figure 6 for a graphical representation of this procedure.

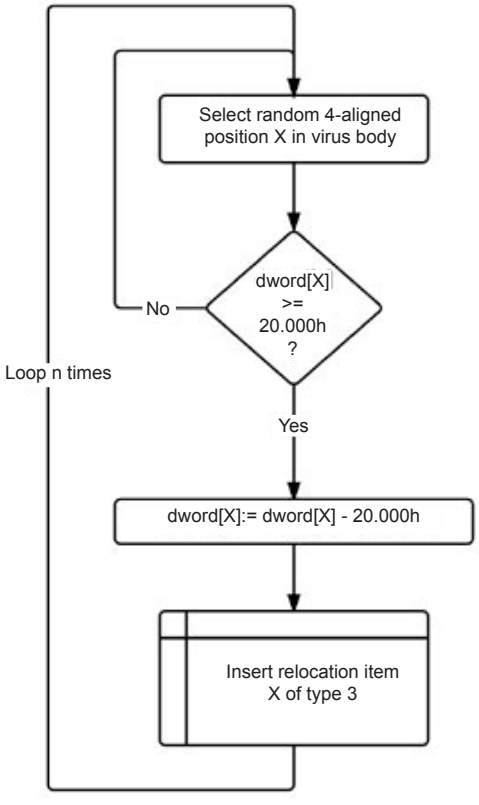

Figure 6: Diagram representing the execution flow of the new virtual code obfuscation procedure.

The following pseudo-code represents a proposal for the new version of virtual code:

```
1 choose N relocation item to create
2 while (N > 0) {
3   generate random number R
4   if R < (code size - 4) {
5      align R to 4 bytes
6      if dword[R] >= 20000 {
7        dword[R] = dword[R] - 20000
8        rel_item = R + 3000h
9        relocation_table .append(rel_item)
10       N--
11     }
12  }
13 }
```

The first instruction chooses the number of relocation items that will be generated by the new obfuscation procedure.

This value can be randomized, meaning that the relocation table size will be different at each obfuscation and will make the virus even more polymorphic.

Next, lines 3 to 5 generate a random number which represents a position inside the virus body. Note that the value 4 must be subtracted from the total virus size in order to avoid selecting a dword in the last four bytes, which would cause an overflow outside the virus body. At line 5 the chosen position is aligned to four bytes, hence avoiding non-aligned overlapping relocations. The instruction at line 6 ensures that the dword at the selected random position is greater than or equal to 0x20000, and only in such a case does the instruction at line 7 subtract this amount from the selected dword.

Finally, the instruction at line 8 generates the relocation item of type 3 (by adding 0x3000), which is then stored in the relocation table of rel.exe at line 9. This loop is executed until the number of relocation items to generate is decremented to zero.

The number of relocation items to produce is decided randomly, which therefore has an important impact both on the size of the file and on the time required for the obfuscation procedure. The relationship between time and number of items has been analysed in a series of tests whose results are displayed in Figure 7. The function maintains an acceptable growth rate as long as the number of relocation items to produce does not exceed 200,000. With higher values this function should assume an exponential behaviour since the more items are produced, the more bytes are brought to a zero value and this causes a frequent number of failures in the compare check at line 10.

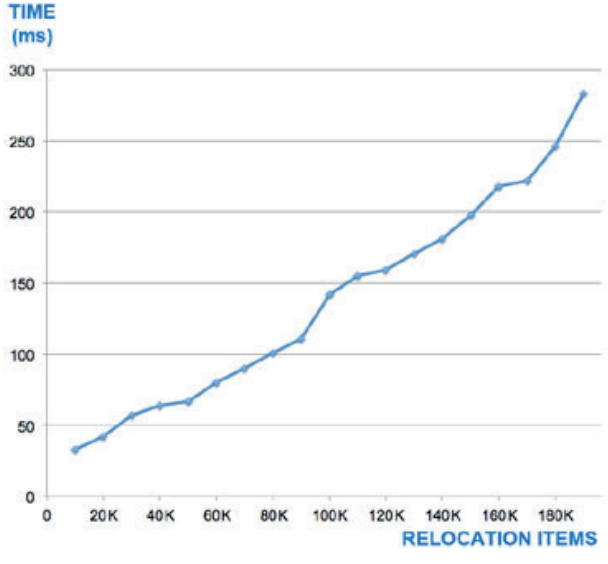

Figure 7: Obfuscation procedure performances with different sizes for the relocation table.

CONCLUSION

The analysis of a relatively dated piece of malware such as W32/Relock has shown that it is possible to deeply understand the inner structure of an OS and find unexpected vulnerabilities in new OS releases. This has both pedagogical and technical outcomes. Pedagogically, it proves the importance of an accurate analysis of the code of dated malware, which can be an incredible source of inspiration both for understanding protection mechanisms and for identifying possible unexpected vulnerabilities in new OS releases.

The peculiar nature of Relock exploits a flaw in the relocation mechanism to dynamically rebuild the malware code out of a relocation table in a polymorphic by relocation code obfuscation. This idea has been restored for *Windows 7* where the discovery of sensible values for ImageBase has led to the adaptation of Relock for the new OS. During this process some important modifications have been made to the structure of the virus, in particular to the file header and to the obfuscation procedure which has been altered to compensate for the removal of relocation types 1, 5 and 9. Another important change to the structure of the virus executable resides in the .code section: instead of being empty, it contains the leftovers from the obfuscating procedure. The stealth effectiveness of the new Relock has not been compromised since heuristic analysis conducted by a range of anti-virus products gives the same results as the original malware. In conclusion, these modifications have not compromised the essence of the obfuscation algorithm and the virus runs smoothly on *Windows 7*, bringing these relocation-based obfuscation techniques to modern times.

REFERENCES

[1] Roy g biv. Virtual Code. October 2007. http://eof-project.net/articles/roygbiv/vcode.html.

[2] Roy g biv. W32.Relock. 2009. http://eof-project.net/sources/roygbiv/Win32.Relock.

[3] Ferrie, P. Doin' the eagle rock. Virus Bulletin, March 2010, p.4.

[4] Shacham, H.; Page, M.; Pfaff, B.; Goh, E.J.; Modadugu, N.; Boneh, D. On the Effectiveness of AddressSpace Randomization. ACM Conference on Computer and Communications Security, CCS'04, October 25-29, 2004, Washington, DC, USA.

END NOTES & NEWS

The 20th USENIX Security Symposium will be held 10–12 August 2011 in San Francisco, CA, USA. See http://usenix.org/.

The 8th Annual Collaboration, Electronic messaging, Anti-Abuse and Spam Conference (CEAS 2011) will be held in Perth, Australia 1–2 September, 2011. See http://ceas2011.debii.edu.au/.

(ISC)² Security Congress takes place 19–22 September 2011 in Orlando, FL, USA. The first annual (ISC)² Security Congress offers education to all levels of information security professionals, not just (ISC)² members. For more information visit http://www.isc2.org/congress2011.

Cairo Security Camp takes place 30 September to 1 October 2011 in Cairo, Egypt. This annual event targets the information security community of the Middle East and North Africa. IT professionals and security practitioners from throughout the region are invited to attend. See http://www.bluekaizen.org/cscamp.html.

VB2011 takes place 5–7 October 2011 in Barcelona, Spain. For full programme details including abstracts for each paper, and online registration see http://www.virusbtn.com/conference/vb2011/.

RSA Europe 2011 will be held 11–13 October 2011 in London, UK. For details see http://www.rsaconference.com/2011/europe/index.htm.

The MAAWG 23rd General Meeting takes place 24–27 October 2011 in Paris, France. See http://www.maawg.org/.

The Hacker Halted Conference takes place 25–27 October 2011 in Miami, FL, USA. The conference is preceded by the Hacker Halted Academy (a range of technical training and certification classes) 21–24 October. For more information about both events see http://www.hackerhalted.com/2011/.

The CSI 2011 Annual Conference will be held 6–11 November 2011 in Washington D.C., USA. See http://www.CSIannual.com/.

The sixth annual APWG eCrime Researchers Summit will be held 7–9 November 2011 in San Diego, CA, USA. The summit will bring together academic researchers, security practitioners and law enforcement to discuss all aspects of electronic crime and ways to combat it. For more details see http://www.antiphishing.org/ecrimeresearch/2011/cfp.html.

The 14th AVAR Conference (AVAR2011) and international festival of IT Security will be held 9–11 November 2011 in Hong Kong. For details see http://aavar.org/avar2011/.

Ruxcon takes place 19–20 November 2011 in Melbourne, Australia. The conference is a mixture of live presentations, activities and demonstrations presented by security experts from the Aus-Pacific region and invited guests from around the world. For more information see http://www.ruxcon.org.au/.

Takedowncon 2 – Mobile and Wireless Security will be held 2–7 December 2011 in Las Vegas, NV, USA. EC-Council's new technical IT security conference series aims to bring industry professionals together to promote knowledge sharing, collaboration and social networking. See http://www.takedowncon.com/ for more details.

Black Hat Abu Dhabi takes place 12–15 December 2011 in Abu Dhabi. Registration for the event is now open. For full details see http://www.blackhat.com/.

ADVISORY BOARD

Pavel Baudis, *Alwil Software, Czech Republic*
Dr Sarah Gordon, *Independent research scientist, USA*
Dr John Graham-Cumming, *Causata, UK*
Shimon Gruper, *NovaSpark, Israel*
Dmitry Gryaznov, *McAfee, USA*
Joe Hartmann, *Microsoft, USA*
Dr Jan Hruska, *Sophos, UK*
Jeannette Jarvis, *Independent researcher, USA*
Jakub Kaminski, *Microsoft, Australia*
Eugene Kaspersky, *Kaspersky Lab, Russia*
Jimmy Kuo, *Microsoft, USA*
Costin Raiu, *Kaspersky Lab, Russia*
Péter Ször, *McAfee, USA*
Roger Thompson, *AVG, USA*
Joseph Wells, *Independent research scientist, USA*

SUBSCRIPTION RATES

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

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

Corporate rates include a licence for intranet publication.

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

- Comparative subscription: $100

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

Editorial enquiries, subscription enquiries, orders and payments:

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

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

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

No responsibility is assumed by the Publisher for any injury and/or damage to persons or property as a matter of products liability, negligence or otherwise, or from any use or operation of any methods, products, instructions or ideas contained in the material herein.

This publication has been registered with the Copyright Clearance Centre Ltd. Consent is given for copying of articles for personal or internal use, or for personal use of specific clients. The consent is given on the condition that the copier pays through the Centre the per-copy fee stated below.

VIRUS BULLETIN © 2011 Virus Bulletin Ltd, The Pentagon, Abingdon Science Park, Abingdon, Oxfordshire OX14 3YP, England. Tel: +44 (0)1235 555139. /2011/$0.00+2.50. No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form without the prior written permission of the publishers.