2016 年 6 月 21 日，星期二

中毒的存档

漏洞发现者：“冰壁” Marcin Noga。博文作者：Marcin Noga 和 Jaeson Schultz。

libarchive 是一个开源库，可用于访问各种不同的文件存档格式，几乎所有地方都有使用。最近，思科 Talos 与 libarchive 维护人员合作，对该库中的三个极为严重的漏洞进行了修复。由于许多产品在处理压缩文件时都会用到 libarchive，所以 Talos 强烈建议所有用户修补/升级相关的易受攻击软件。

TALOS-2016-0152 [CVE-2016-4300]：7-ZIP READ_SUBSTREAMSINFO 整数溢出

这是另一个可能导致代码执行的 7-Zip 漏洞（有关 7-Zip 漏洞的上一篇博客，请点击此处）。在本实例中，经特殊设计的 7-Zip 文件可能导致整数溢出，造成后续内存损坏和代码执行。要利用此漏洞，攻击者只需将中毒的 7-Zip 文件发送给受害者供其使用 libarchive 处理。

7-Zip 支持格式模块 libarchive\archive_read_support_format_7zip.c 中存在易受攻击的代码：

```
(...)
#define UMAX_ENTRY       ARCHIVE_LITERAL_ULL(100000000)
(...)
Line 2129       static int
Line 2130       read_SubStreamsInfo(struct archive_read *a, struct _7z_substream_info *ss,
Line 2131               struct _7z_folder *f, size_t numFolders)
Line 2132       {
Line 2133               const unsigned char *p;
Line 2134               uint64_t *usizes;
Line 2135               size_t unpack_streams;
Line 2136               int type;
Line 2137               unsigned i;
Line 2138               uint32_t numDigests;
(...)
Line 2149       if (type == kNumUnPackStream) {
Line 2150               unpack_streams = 0;
Line 2151               for (i = 0; i < numFolders; i++) {
Line 2152                       if (parse_7zip_uint64(a, &(f[i].numUnpackStreams)) < 0)
Line 2153                               return (-1);
Line 2154                       if (UMAX_ENTRY < f[i].numUnpackStreams)
Line 2155                               return (-1);
Line 2156                       unpack_streams += (size_t)f[i].numUnpackStreams;
^^^^^^^^^  ---- INTEGER OVERFLOW
Line 2157               }
Line 2158               if ((p = header_bytes(a, 1)) == NULL)
Line 2159                       return (-1);
Line 2160               type = *p;
Line 2161       } else
Line 2162               unpack_streams = numFolders;
```

```
Line 2163
Line 2164       ss->unpack_streams = unpack_streams;
Line 2165       if (unpack_streams) {
Line 2166               ss->unpackSizes = calloc(unpack_streams,
^^^^^^^^^  ---- ALLOCATION BASED ON OVERFLOWED INT
Line 2167                   sizeof(*ss->unpackSizes));
Line 2168               ss->digestsDefined = calloc(unpack_streams,
Line 2169                   sizeof(*ss->digestsDefined));
Line 2170               ss->digests = calloc(unpack_streams,
Line 2171                   sizeof(*ss->digests));
Line 2172               if (ss->unpackSizes == NULL || ss->digestsDefined == NULL ||
Line 2173                   ss->digests == NULL)
Line 2174                       return (-1);
Line 2175       }
Line 2176
Line 2177       usizes = ss->unpackSizes;
Line 2178       for (i = 0; i < numFolders; i++) {
Line 2179               unsigned pack;
Line 2180               uint64_t sum;
Line 2181
Line 2182               if (f[i].numUnpackStreams == 0)
Line 2183                       continue;
Line 2184
Line 2185               sum = 0;
Line 2186               if (type == kSize) {
Line 2187                       for (pack = 1; pack < f[i].numUnpackStreams; pack++) {
Line 2188                               if (parse_7zip_uint64(a, usizes) < 0)                    ^^^^^^^^^  ---- BUFFER OVERFLOW
Line 2189                                       return (-1);
Line 2190                               sum += *usizes++;
Line 2191                       }
Line 2192               }
Line 2193               *usizes++ = folder_uncompressed_size(&f[i]) - sum;
Line 2194       }
```

在第 2149-2157 行的代码中，可以看到，我们对所有“folder”计算了“numUnpackStreams”的和，然后将结果存储在“unpack_streams”变量中。
这个变量名为“size_t”，从其定义可以看出，在 x86 平台上，它将是一个 32 位无符号整数。
不过，请注意代码允许的“numUnpackStreams”最大值为：

```
UMAX_ENTRY      100000000
```

这意味着若要溢出“unpack_streams”变量，只需让 7-Zip 文件的文件夹数“numFolders”大于 42，然后使用足够大的值填写“numUnpackStreams”。

请注意，溢出值在第 2166-2171 行的 calloc 方法中用作大小参数。最值得注意的是，此处分配的缓冲区是“ss->unpackSizes”。之后，在第 2187-2194 行，根据每个文件夹的“numFolders”和“numUnpackStreams”值，代码会从文件读取 64 位无符号整数（最大值）并存储到缓冲区“usizes”（实际上是“ss->unpackSizes”，位于第 2177 行），这就会导致基于堆的缓冲区溢出（取决于溢出的值）。在经过某种迭代后，攻击者便可以完全控制用于溢出缓冲区的字节量及其值。

TALOS-2016-0153 [CVE-2016-4301]：
MTREE PARSE_DEVICE 基于堆栈的缓冲区溢出

在此漏洞中，代码尽其所能防止缓冲区溢出，但所使用的方法却存在错误。代码中会创建一个数组，其中最多保存三个无符号长整数。稍后，代码会尝试验证参数数量是否小于最大值 3，但却不会检查这些参数是否超过大小长整数。

在 mtree 支持格式模块 libarchive\archive_read_support_format_mtree.c 中存在易受攻击的代码：

```
Line 1353       static int
Line 1354       parse_device(dev_t *pdev, struct archive *a, char *val)
Line 1355       {
Line 1356       #define MAX_PACK_ARGS 3
Line 1357               unsigned long numbers[MAX_PACK_ARGS];
Line 1358               char *p, *dev;
Line 1359               int argc;
Line 1360               pack_t *pack;
Line 1361               dev_t result;
Line 1362               const char *error = NULL;
(...)
Line 1377               while ((p = la_strsep(&dev, ",")) != NULL) {
Line 1378                       if (*p == '\0') {
Line 1379                               archive_set_error(a, ARCHIVE_ERRNO_FILE_FORMAT,
Line 1380                                   "Missing number");
Line 1381                               return ARCHIVE_WARN;
Line 1382                       }
Line 1383                       numbers[argc++] = (unsigned long)mtree_atol(&p);
Line 1384                       if (argc > MAX_PACK_ARGS) {
Line 1385                               archive_set_error(a, ARCHIVE_ERRNO_FILE_FORMAT,
Line 1386                                   "Too many arguments");
Line 1387                               return ARCHIVE_WARN;
Line 1388                       }
Line 1389               }
```

在第 1357 行中，我们看到静态缓冲区被定义为预备包含三个元素。接下来，在 1377-1389 行的 while 循环中，存在一个应防止溢出“numbers”缓冲区的条件（第 1384 行）。不过，此条件在编码上存在错误，使攻击者有可能使用一个比无符号长整数大的元素造成缓冲区溢出。根据具体的平台和架构，覆盖可能为 4 字节或 8 字节，其内容完全可控。

TALOS-2016-0154 [CVE-2016-4302]：
LIBARCHIVE RAR RESTARTMODEL 堆溢出

在创建上下文时，下面的执行流会导致堆崩溃。

```
archive_read_next_header
        archive_read_format_rar_read_header
    head_type : 0x72
    head_type : 0x73
    head_type : 0x74
    read_header
```

```
        rar->packed_size : 0x1
        rar->dictionary_size = 0;
        archive_format_name : RAR
archive_read_extract
        rar->compression_method : 0x31
        read_data_compressed
          archive_read_format_rar_read_header
        head_type : 0x7a
        read_header
        parse_codes
              if (ppmd_flags & 0x20)
                archive_read_format_rar_read_header
              head_type : 0x7b
                archive_read_format_rar_read_header
              head_type : 0x74
              read_header
              rar->packed_size : 0x1
              rar->dictionary_size = 0;
                archive_read_format_rar_read_header
              head_type : 0x7a
              read_header
              rar->dictionary_size : 0x10000000
                archive_read_format_rar_read_header
              head_type : 0x7b
                archive_read_format_rar_read_header
              head_type : 0x74
              read_header
              rar->packed_size : 0x1
              rar->dictionary_size = 0;
                archive_read_format_rar_read_header
                archive_read_format_rar_read_header
              __archive_ppmd7_functions.PpmdRAR_RangeDec_CreateVTable(&rar->range_dec);
              ppmd_alloc : 0
                archive_read_format_rar_read_header
                archive_read_format_rar_read_header
                archive_read_format_rar_read_header
                archive_read_format_rar_read_header
                archive_read_format_rar_read_header
                archive_read_format_rar_read_header
                archive_read_format_rar_read_header
                archive_read_format_rar_read_header
              Ppmd7_Init
              RestartModel
        *** Heap corruption ***
```

我们着重来看提取阶段（archive_read_extract 下的所有内容）。此处的关键变量/字段是 rar->dictionary_size。首先，我们看到其值设置为：

rar->dictionary_size : 0x10000000

libarchive\archive_read_support_format_rar.c

```
Line 2073    /* Memory is allocated in MB */
Line 2074    if (ppmd_flags & 0x20)
Line 2075    {
Line 2076      if (!rar_br_read_ahead(a, br, 8))
Line 2077        goto truncated_data;
Line 2078      rar->dictionary_size = (rar_br_bits(br, 8) + 1) << 20;
Line 2079      rar_br_consume(br, 8);
Line 2080    }
```

接下来，由于在其他内容中，

rar->packed_size 的较小值为 0x1，

因此另一部分数据从文件读取，并通过调用 archive_read_format_rar_read_header 和 read_header 函数，在 archive_read_next_header 中的“读取阶段”进行解析。在这些调用中，我们看到有一个调用的 dictionary_size 值设置为零。接下来，代码会根据 dictionary_size 的值，对 Ppmd 上下文执行分配：

libarchive\archive_read_support_format_rar.c：

```
Line 2115     if ( !__archive_ppmd7_functions.Ppmd7_Alloc(&rar->ppmd7_context,rar->dictionary_size, &g_szalloc) )
```

libarchive\archive_ppmd7.c

```
Line 125      static Bool Ppmd7_Alloc(CPpmd7 *p, UInt32 size, ISzAlloc *alloc)
Line 126      {
Line 127        if (p->Base == 0 || p->Size != size)
Line 128        {
Line 129            Ppmd7_Free(p, alloc);
Line 130            p->AlignOffset =
Line 131              #ifdef PPMD_32BIT
Line 132                  (4 - size) & 3;
Line 133              #else
Line 134                  4 - (size & 3);
Line 135              #endif
Line 136            if ((p->Base = (Byte *)alloc->Alloc(alloc, p->AlignOffset + size
Line 137                  #ifndef PPMD_32BIT
Line 138                  + UNIT_SIZE
Line 139                  #endif
Line 140                  )) == 0)
Line 141              return False;
Line 142            p->Size = size;
Line 143        }
Line 144        return True;
Line 145      }
```

正如我们可以看到的那样，dictionary_size 等于 0，我们将分配 0 字节，这会分配可能最小的数据块。

```
python import gdbheap
p p->Base
$5 = (Byte *) 0x80e2480 " \b\r\brn"
heap select size==16
Reading in symbols for malloc.c...done.
   Start       End        Domain       Kind    Detail                                        Hexdump
---------- ---------- ------------- ----------- -------- ---------------------------------------------------------------------------------
0x080d0798  0x080d07a7  uncategorized           16 bytes  a8 07 0d 08 03 00 00 00 57 4d 54 00 19 00 00 00 c0 07 0d 08 |........WMT.........|
0x080d07c0  0x080d07cf  uncategorized           16 bytes  d0 07 0d 08 03 00 00 00 43 45 54 00 19 00 00 00 e8 07 0d 08 |........CET.........|
0x080d07e8  0x080d07f7  uncategorized           16 bytes  00 00 00 00 03 00 00 00 45 45 54 00 21 00 00 00 72 61 72 2d |........EET.!...rar-|
0x080d0818  0x080d0827  uncategorized           16 bytes  00 00 00 00 50 a4 d8 f7 00 00 00 00 11 00 00 00 50 70 6d 64 |....P...........Ppmd|
0x080d0828  0x080d0837             C  string data          50 70 6d 64 37 5f 49 6e 69 74 0a 00 21 00 00 00 44 00 00 00 |Ppmd7_Init..!...D...|
0x080d0cd8  0x080d0ce7  uncategorized           16 bytes  e8 0c 0d 08 00 00 00 00 00 00 00 00 f9 01 00 00 c5 b0 01 c0 |....................|
0x080e2470  0x080e247f  uncategorized           16 bytes  00 69 62 61 32 2e 68 74 6d 6c c0 cc 11 00 00 00 20 08 0d 08 |.iba2.html..........|
0x080e2480  0x080e248f             C  string data          20 08 0d 08 72 6e 00 00 00 00 00 00 21 00 00 00 72 61 72 2d | ...rn......!...rar-|
```

最后，我们以 RestartModel 函数结束：

libarchive\archive_ppmd7.c

```
Line 314    static void RestartModel(CPpmd7 *p)
Line 315    {
Line 316      unsigned i, k, m;
Line 317
Line 318      memset(p->FreeList, 0, sizeof(p->FreeList));
Line 319      p->Text = p->Base + p->AlignOffset;
Line 320      p->HiUnit = p->Text + p->Size;
Line 321      p->LoUnit = p->UnitsStart = p->HiUnit - p->Size / 8 / UNIT_SIZE * 7 * UNIT_SIZE;
Line 322      p->GlueCount = 0;
Line 323
Line 324      p->OrderFall = p->MaxOrder;
Line 325      p->RunLength = p->InitRL = -(Int32)((p->MaxOrder < 12) ? p->MaxOrder : 12) - 1;
Line 326      p->PrevSuccess = 0;
Line 327
Line 328      p->MinContext = p->MaxContext = (CTX_PTR)(p->HiUnit -= UNIT_SIZE); /* AllocContext(p); */
Line 329      p->MinContext->Suffix = 0;
Line 330      p->MinContext->NumStats = 256;
Line 331      p->MinContext->SummFreq = 256 + 1;
Line 332      p->FoundState = (CPpmd_State *)p->LoUnit; /* AllocUnits(p, PPMD_NUM_INDEXES - 1); */
Line 333      p->LoUnit += U2B(256 / 2);
Line 334      p->MinContext->Stats = REF(p->FoundState);
```

此处键值为：

p p->AlignOffset
$6 = 0
p p->Size
$7 = 0

如上所示，二者都等于 0（在第 328 行产生结果的值）。
代码会从 p->HiUnit 减去 UNIT_SIZE 值：

```
Line 30 #define UNIT_SIZE 12
```

并赋值给 p->MinContext。

所有内容都正确后，代码会在 Ppmd 上下文的已分配空间结尾设置 p->HiUnit，但是由于 AlignOffset 和 Size 等于 0，它仍指向已为 Ppmd 分配空间的开始处。因此，减法得到的 p->MinContext 值会设置为前一堆数据块空间内的值。

p p->MinContext
$8 = (CPpmd7_Context *) 0x80e2474

请注意，我们的堆 [a][b] 为：

```
(...)
0x080e2470  0x080e247f  uncategorized              16 bytes  00 69 62 61 32 2e 68 74 6d 6c c0 cc 11 00 00 00 20 08 0d 08 |.iba2.html.....|
0x080e2480  0x080e248f            C  string data          20 08 0d 08 72 6e 00 00 00 00 00 00 21 00 00 00 72 61 72 2d | ...rn.....!...rar-|
(...)
```

对此结构的进一步写入会覆盖堆数据块。在执行第 329 行之前进行堆检查：

```
python from heap.glibc import *
python print MChunkPtr(gdb.Value(0x080e2480-8).cast(MChunkPtr.gdb_type()))
<MChunkPtr chunk=0x80e2478 mem=0x80e2480 PREV_INUSE inuse chunksize=16 memsize=8>
```

在执行后进行堆检查：

```
python print MChunkPtr(gdb.Value(0x080e2480-8).cast(MChunkPtr.gdb_type()))
<MChunkPtr chunk=0x80e2478 mem=0x80e2480 prev_size=3435162733 free chunksize=0 memsize=-8&gt
```

总结

编写安全代码会很困难。这些 libarchive 漏洞的根本原因是未能正确验证输入 - 从压缩文件读取的数据。令人遗憾的是，这些类型的编程错误一再发生。当软件的一部分（例如 libarchive）中发现漏洞时，依赖该软件的许多第三方程序和捆绑 libarchive 都会受到影响。这就是所谓的共模故障，可能导致攻击者使用单一攻击破坏多个不同的程序/系统。我们鼓励用户尽快修补所有相关程序。

我们通过 ClamAV 提供对 TALOS-CAN-0152 的覆盖，因为要进行检测，完整的文件是必不可少的。TALOS-CAN-0153 可通过 SID 39034 和 39035 规则进行检测。TALOS-CAN-0154 可通过 SID 39045 和 39046 规则进行检测。

发布者：JAESON SCHULTZ；发布时间：10:36

标签：LIBARCHIVE、SNORT 规则、漏洞研究