# Analysis of Adobe Flash Player ID3 Tag Parsing Integer Overflow Vulnerability (CVE-2015-5560)

January 12, 2016 by Nahuel Riva

## Vulnerability Overview

After Adobe released a patch for this vulnerability, it was made public that this bug was already being exploited in the wild by some exploit kits like *Angler* and *Nuclear Pack*.

This vulnerability is about an integer overflow in *Adobe Flash Player* when parsing a compressed ID3 tag which size exceed *0x2AAAAAAA* bytes. An error  in how the size of a dynamic allocated buffer is calculated, used as destination for final decompressed data, produces that too much data is copied to a small buffer. In other words, a heap-based buffer overflow.

This bug was fixed in *Adobe Flash player 18.0.0.23*2. However, this is an important fix because that version, the previous one *18.0.0.20*9, and new versions, introduce new exploit mitigations to avoid exploitation techniques as the one described in Haife's Li presentation using *Vector*.

In fact, the exploits included in the exploit kits mentioned above, perform new bypasses as the ones described in the Project Zero blog spot.

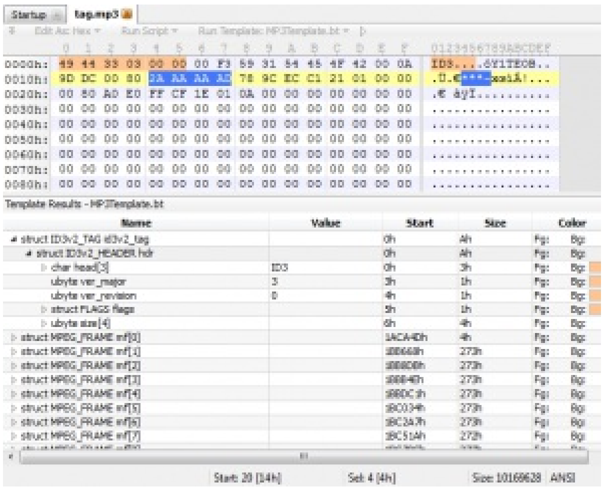
## Vulnerability analysis, finding the root cause

Natalie Silvanovich, from *Google Project Zero (PZ)*, made public a PoC in order to trigger this bug. As the PoC seems to be written for Flash CS (or some compliant compiler) and I like more Apache Flex, I re-wrote the PoC like this:

```
package
{
 import flash.display.*;
 import flash.media.*;
 import flash.utils.*;
 import flash.net.*;
 import flash.events.*;
 import flash.system.*;
 import flash.external.*;
 import avm2.intrinsics.memory.*;

 public class CVE_2015_5560 extends Sprite
 {
  public var mySound:Sound;

  function CVE_2015_5560()
  {
   logDebug("Loading MP3 file ...");

   mySound = new Sound();
   mySound.load(new URLRequest("CVE_2015_5560.mp3"));
   mySound.play();

   logDebug("Triggering corruption :)");

   setInterval(f, 1000);
  }

  private function f():void
  {
   mySound.id3;
  }
 }
}
```

Anyway, the important thing here is the MP3 referenced in the code; that’s the real file triggering the bug.

Let’s see a little bit the structure of the ID3 tag contained in the MP3 file with the help of the 010 Editor and its MP3 template:




We can observe the presence of an ID3v2 tag followed by another tag that the template couldn't recognize due to malformed data:

```
MP3: ID3v2 tag found
MP3: warning: invalid ID3v2 tag header --> ERROR HERE!!!
MP3: warning: invalid MPEG frame synchronization at offset 0xA
MP3: warning: invalid MPEG header in frame at offset 0x1ACA4D
MP3: warning: invalid MPEG frame synchronization at offset 0x1ACA51
MP3: first found MPEG frame parameters:
MP3: - header ofsset: 0x1BB668
MP3: - bitrate: 192 kbit
MP3: - MPEG-1 layer 3
MP3: - sampling frequency: 44100 Hz
MP3: - channel mode: stereo
MP3: - CRC protected: No
MP3: ID3v1 tag found
MP3: file parsing completed!
MP3: valid MPEG frames found: 13324
MP3: average frame bitrate: 192 kbit
```

In the previous image, we can see the **major** and **revision** fields in the *ID3* header with the values **3** and **0**, respectively, which means that this is an ID3v2.3.0. Hence I'm going to use that version of the specs to perform my analysis of the tag.

According to the specification, after the main **ID3 header**, we can find the so called **ID3 frames**, with the following structure:

```
Frame ID   $xx xx xx xx  (four characters)
Size       $xx xx xx xx
Flags      $xx xx
```

So, we have 10 bytes, adjusting this to what we have seen previously, our tag ends like this:

```
Frame ID: 'TEOB'
Size: 0x000A9DDC
Flags: 0x0080
```

First, we have the **Frame ID**. In our case, it is a user defined text information frame.

After the **Frame ID**, we have the **Size**, *0xA9DDC*. This is the *frame size* excluding the *frame header*, that is *frame size* – 10.

At the end, we have the **Flags**, **0x0080**. This field is composed by two bytes, **0x00** and **0x80**. The first byte, *0x00*, is the **status message** and the second one, *0x80*, is used for **encoding purposes**.

According to the mentioned in the 3.3.1. Frame header flags section, our first byte is *0x00*, so, our result is:

- **Tag alter preservation**: Frame should be preserved.
- **File alter preservation**: Frame should be preserved.
- **Read only**: no read-only.

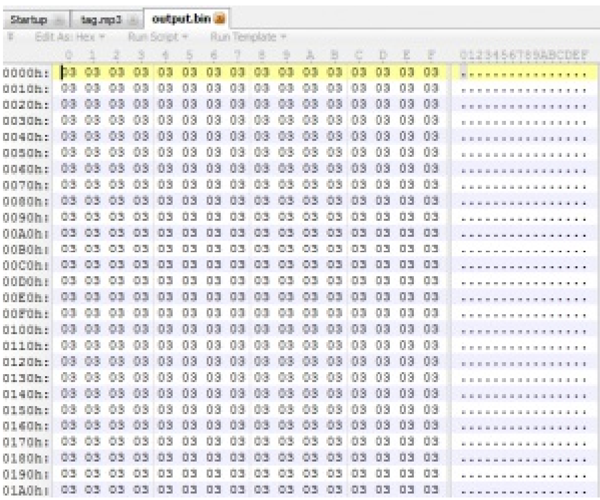
These are not important flags for us, let's see the other byte, *0x80* (*10000000* in binary):

- **Compression**: 1 Frame is compressed using zlib with 4 bytes for 'decompressed size' appended to the frame header.
- **Encryption**: 0 Frame is not encrypted.
- **Grouping identity**: 0 Frame does not contain group information.

In this second byte, only the most significant bit, the **Compression** bit, is on so that indicates that our frame is compressed using **zlib**. Therefore, according to the specs, there must be **decompression size** (4 bytes) field after the *frame header*. In our case, our *decompression size* is *0x2AAAAAAA*. What's next is the compressed data using *zlib*. Just to be sure, I wrote a small and ugly script to decompress the data:

```
import sys
import zlib

from struct import unpack

print 'Opening file ...'
fd = open(sys.argv[1], 'rb')
data = fd.read()
fd.close()

print 'Reading tag size ...'
tag_size = unpack('>L', data[0x0E:0x0E+4])[0]
print 'Tag size: %x' % tag_size

print 'Getting compressed data ...'
compressed_data = data[0x18:0x18+tag_size]

print 'Decompressing ...'
decompressed_data = zlib.decompress(compressed_data)

print 'Saving decompressed data to a file ...'
fd = open('output.bin', 'wb')
fd.write(decompressed_data)
fd.close()
print 'Done. Saved to output.bin'
```

This is the result:



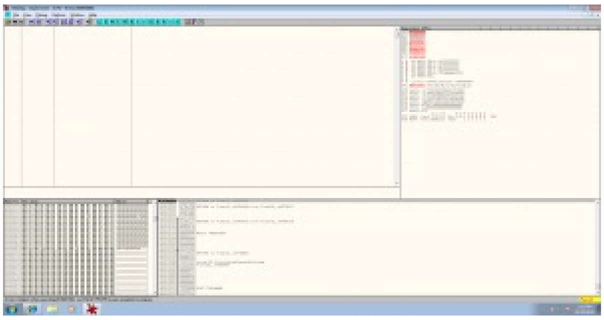

The decompressed data contains a lot of *0x03* bytes.

According to the *PZ* advisory, the *decompression size* is the value that causes the integer overflow so, it’s important to pay a attention to this value during our debugging session.

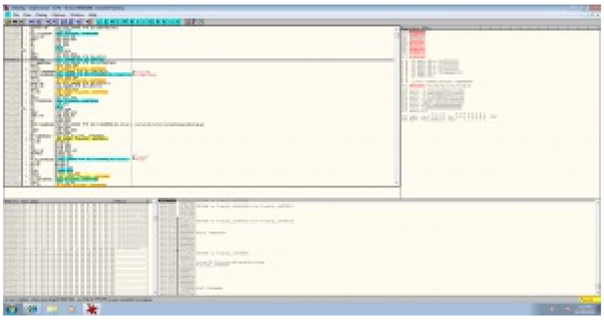
Just to see how *Adobe Flash Player* behaves using this *PoC*, I compiled the *AS* code and put the generated *SWF* file plus the *MP3* file and an *HTML* file that loads the *SWF* in a folder and then started a webserver using *Python* like this: *python -m SimpleHTTPServer 8888*.

My testing environment is Windows Ultimate SP1 with *Adobe Flash Player* 18.0.0.209 (32 bits).

Once I requested the *HTML* and the *SWF* was loaded, this was the result:



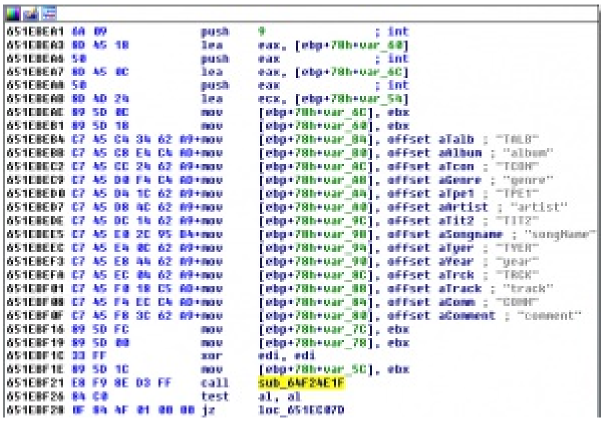
In the previous image, *EIP* has the value *0x01000100*  and in *EAX* we can recognize a know value, *0x03030303* (compressed data). In this case, we were a lucky guys because *0x03030303* is a mapped address and it content ended up in *EIP*. When this address is not mapped, the program crashes earlier, in a *CALL [EAX+8]*, just in the virtual call of the id3 property from the **mySound** object.



Next step is to identify where the integer overflow is produced so we must find the function responsible for parsing the *ID3* tag.

Generally, an *ID3* tag is used to show meta-information as the album name, music genre, artist name, etc; of a *MP3* file (or any audiovisual container file) so I started to look for strings in *IDA* like “*album*“, “*genre*“, “*author*“, etc.

Through the “*genre*” string, I finally found a nice basic block where the tag seems to be parsed:


```
651EBEA1 6A 09             push    9                      ; int
651EBEA3 8D 45 18          lea     eax, [ebp+78h+var_60]
651EBEA6 50                push    eax                    ; int
651EBEA7 8D 45 0C          lea     eax, [ebp+78h+var_6C]
651EBEAA 50                push    eax                    ; int
651EBEAB 8D 4D 24          lea     ecx, [ebp+78h+var_54]
651EBEAE 89 5D 0C          mov     [ebp+78h+var_6C], ebx
651EBEB1 89 5D 18          mov     [ebp+78h+var_60], ebx
651EBEB4 C7 45 C4 34 62 A9+mov     [ebp+78h+var_B4], offset aTalb ; "TALB"
651EBEBB C7 45 C8 E4 C4 A0+mov     [ebp+78h+var_B0], offset aAlbum ; "album"
651EBEC2 C7 45 CC 24 62 A9+mov     [ebp+78h+var_AC], offset aTcon ; "TCON"
651EBEC9 C7 45 D0 F4 C4 A0+mov     [ebp+78h+var_A8], offset aGenre ; "genre"
651EBED0 C7 45 D4 1C 62 A9+mov     [ebp+78h+var_A4], offset aTpe1 ; "TPE1"
651EBED7 C7 45 D8 4C 62 A9+mov     [ebp+78h+var_A0], offset aArtist ; "artist"
651EBEDE C7 45 DC 14 62 A9+mov     [ebp+78h+var_9C], offset aTit2 ; "TIT2"
651EBEE5 C7 45 E0 2C 95 B4+mov     [ebp+78h+var_98], offset aSongname ; "songName"
651EBEEC C7 45 E4 0C 62 A9+mov     [ebp+78h+var_94], offset aTyer ; "TYER"
651EBEF3 C7 45 E8 44 62 A9+mov     [ebp+78h+var_90], offset aYear ; "year"
651EBEFA C7 45 EC 04 62 A9+mov     [ebp+78h+var_8C], offset aTrck ; "TRCK"
651EBF01 C7 45 F0 18 C5 A0+mov     [ebp+78h+var_88], offset aTrack ; "track"
651EBF08 C7 45 F4 EC C4 A0+mov     [ebp+78h+var_84], offset aComm ; "COMM"
651EBF0F C7 45 F8 3C 62 A9+mov     [ebp+78h+var_80], offset aComment ; "comment"
651EBF16 89 5D FC          mov     [ebp+78h+var_7C], ebx
651EBF19 89 5D 00          mov     [ebp+78h+var_78], ebx
651EBF1C 33 FF             xor     edi, edi
651EBF1E 89 5D 1C          mov     [ebp+78h+var_5C], ebx
651EBF21 E8 F9 8E D3 FF    call    sub_64F24E1F
651EBF26 84 C0             test    al, al
651EBF28 0F 84 4F 01 00 00 jz      loc_651EC07D
```

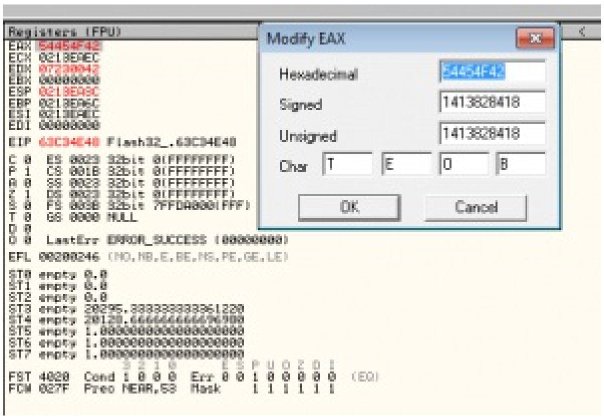
The *sub_64F24E1F* function is responsible for parsing all the frames in the archive. We can set a breakpoint at the very beginning of the function and start tracing to identify the important pieces of the code.

Here’s a brief summary of the most important pieces of the code. Just for convenience, I renamed the function to *parse_id3_tag*.

What happens first in *parse_id3_tag+24*, when calling *GetFrameInfo*, is that the *Frame ID* is read:

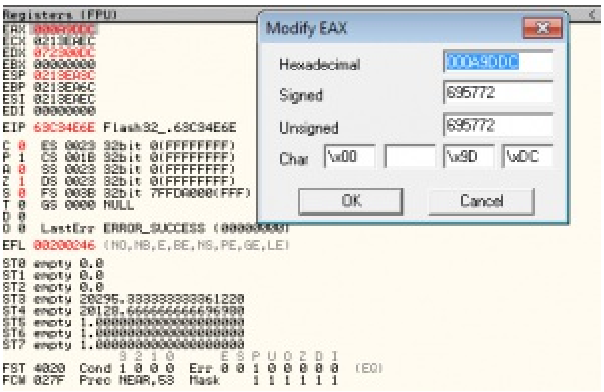
```
parse_id3_tag+1D   loc_64F24E3C:
parse_id3_tag+1D   xor     ebx, ebx
parse_id3_tag+1F   push    ebx
parse_id3_tag+20   push    4
parse_id3_tag+22   mov     ecx, esi
parse_id3_tag+24   call    GetFrameInfo
parse_id3_tag+29   mov     [ebp+var_14], eax
parse_id3_tag+2C   cmp     eax, ebx
parse_id3_tag+2E   jz      loc_64F25160
```

In EAX, we have the return value:




After that, the *Size* is read:

```
parse_id3_tag+43
parse_id3_tag+43   loc_64F24E62:
parse_id3_tag+43   movzx   eax, byte ptr [esi+20h]
parse_id3_tag+47   push    eax
parse_id3_tag+48   push    4
parse_id3_tag+4A   call    GetFrameInfo
```

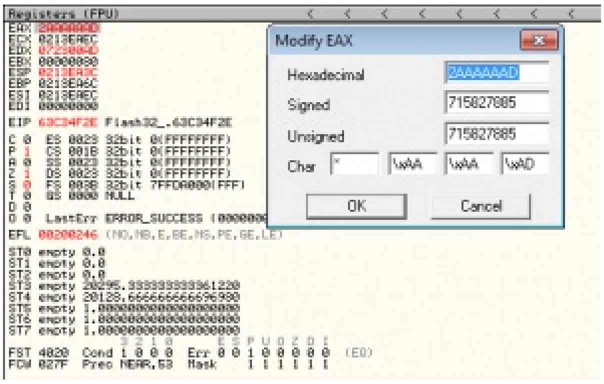



Then, we get the *Flags*:

```
parse_id3_tag+61    push    ebx
parse_id3_tag+62    push    1
parse_id3_tag+64    mov     ecx, esi
parse_id3_tag+66    call    GetFrameInfo >> encoding
parse_id3_tag+6B    push    ebx
parse_id3_tag+6C    mov     edi, eax
parse_id3_tag+6E    push    1
parse_id3_tag+70    shl     edi, 8
parse_id3_tag+73    call    GetFrameInfo >> status message
parse_id3_tag+78    mov     ecx, [ebp+var_8]
parse_id3_tag+7B    mov     ebx, eax
parse_id3_tag+7D    mov     al, [esi+20h]
parse_id3_tag+80    or      ebx, edi
parse_id3_tag+82    cmp     byte ptr [esi+21h], 4
parse_id3_tag+86    mov     [ebp+var_20], ebx
parse_id3_tag+89    mov     [ebp+var_10], al
parse_id3_tag+8C    mov     [ebp+var_1C], ecx
parse_id3_tag+8F    mov     [ebp+var_1], 1
parse_id3_tag+93    jnz     short loc_64F24F12
```

As we mentioned earlier, the *Compression* bit is on so there must be a call to get the *decompression size* value and that’s exactly what happens next:

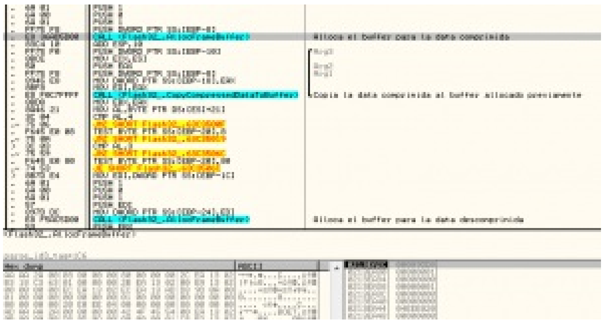
```
parse_id3_tag+103    push    dword ptr [ebp+var_10]
parse_id3_tag+106    mov     ecx, esi
parse_id3_tag+108    push    4
parse_id3_tag+10A    call    GetFrameInfo
parse_id3_tag+10F    sub     [ebp+var_8], 4
parse_id3_tag+113    mov     [ebp+var_1C], eax
```




Now that the frame size and the size for the decompressed data have been read, it’s time to allocate the corresponding buffers to hold the data.

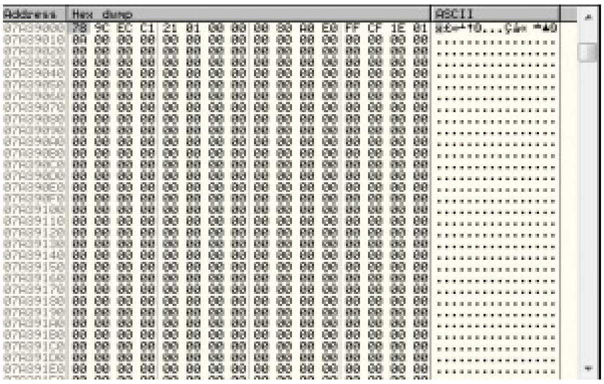
Starting at *parse_id3_tag+1BD*, it allocates a buffer using *frame size – 4* as the size for the buffer:

```
parse_id3_tag+1BD    push    1               ; char
parse_id3_tag+1BF    push    0               ; int
parse_id3_tag+1C1    push    1               ; int
parse_id3_tag+1C3    push    [ebp+var_8]     ; int >> frame size
parse_id3_tag+1C6    call    AllocFrameBuffer
parse_id3_tag+1CB    add     esp, 10h
parse_id3_tag+1CE    push    dword ptr [ebp+var_10] ; char
parse_id3_tag+1D1    mov     ecx, esi
parse_id3_tag+1D3    push    eax             ; int >> buffer to store data
parse_id3_tag+1D4    push    [ebp+var_8]     ; int >> frame size - 4
parse_id3_tag+1D7    mov     [ebp+var_18], eax
parse_id3_tag+1DA    mov     edi, eax
parse_id3_tag+1DC    call    CopyCompressedDataToBuffer
parse_id3_tag+1E1    mov     ebx, eax
parse_id3_tag+1E3    mov     al, [esi+21h]
parse_id3_tag+1E6    cmp     al, 4
parse_id3_tag+1E8    jnz     short loc_64F2500F
```



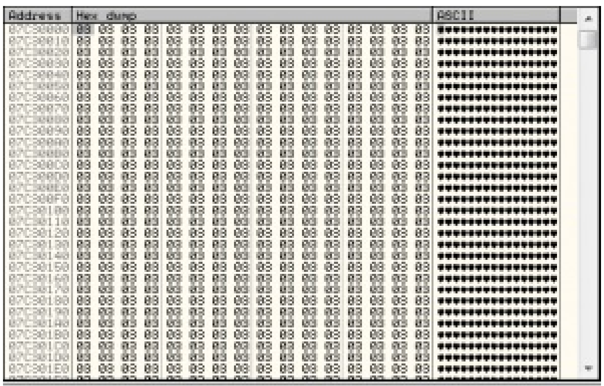
Remember that the size was *0x0A9DDC* and now it is *0x0A9DD8*.

In this case, 0x07A39000 is the buffer used to hold the compressed data. This data is copied there using the *CopyCompressedDataToBuffer* function:




After that, it allocates a buffer using the *decompression size* and then decompress and copy the data to the buffer:

```
parse_id3_tag+1FA
parse_id3_tag+1FA    loc_64F25019:
parse_id3_tag+1FA    mov     edi, [ebp+var_1C]
parse_id3_tag+1FD    push    1               ; char
parse_id3_tag+1FF    push    0               ; int
parse_id3_tag+201    push    1               ; int
parse_id3_tag+203    push    edi             ; int >> decompression size
parse_id3_tag+204    mov     [ebp+var_24], edi
parse_id3_tag+207    call    AllocFrameBuffer
parse_id3_tag+20C    push    ebx             ; int >> frame size - 4
parse_id3_tag+20D    push    [ebp+var_18]    ; int >> buffer with compressed data
parse_id3_tag+210    lea     ecx, [ebp+var_24]
parse_id3_tag+213    push    ecx             ; int
parse_id3_tag+214    push    eax             ; int >> buffer to store the data
parse_id3_tag+215    mov     [ebp+var_20], eax
parse_id3_tag+218    call    DecompressZlibData
parse_id3_tag+21D    add     esp, 20h
parse_id3_tag+220    test    eax, eax
parse_id3_tag+222    jnz     loc_64F25160
```




At this point, we don't notice anything unusual, everything seems to be fine. We keep in mind that in *0x07C30000* we have the decompressed data but a little bit further, in *parse_id3_tag+281*, we see this:

```
parse_id3_tag+281
parse_id3_tag+281    loc_64F250A0:
parse_id3_tag+281    mov     eax, ebx
parse_id3_tag+283    imul    eax, 6
parse_id3_tag+286    add     eax, 2
parse_id3_tag+289    cmp     [esi+28h], eax
parse_id3_tag+28C    mov     [ebp+var_20], eax
parse_id3_tag+28F    jge     short loc_64F250DA
```

In *EBX*, we have the *decompression size – 1* (this has to do with a little code that we overlooked [1]), *0x2AAAAAAC*. Then, *EBX* is copied to *EAX* and multiplied by 6. The result is stored in *EAX*.

Let's do some math:

(0x2AAAAAAC * 6) + 2 = 0x10000000A

Now we do find the bug, the integer overflow is very obvious, what only fits in 32 bits is *0x0A*. Then, the result of this operation is used as a size to allocate a buffer in *parse_id3_tag+2A6*:

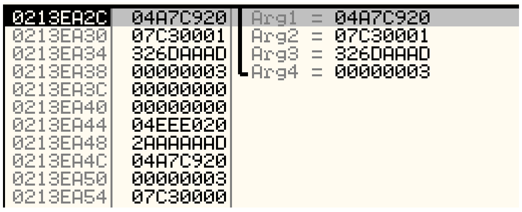
```
parse_id3_tag+2A6
parse_id3_tag+2A6    loc_64F250C5:            ; char
parse_id3_tag+2A6    push    1
parse_id3_tag+2A8    push    0                ; int
parse_id3_tag+2AA    push    1                ; int
parse_id3_tag+2AC    push    eax              ; int >> overflowed size
parse_id3_tag+2AD    mov     [esi+28h], eax
parse_id3_tag+2B0    call    AllocFrameBuffer
parse_id3_tag+2B5    add     esp, 10h
parse_id3_tag+2B8    mov     [esi+24h], eax
```

In this case, the buffer starts at *0x04A7C920* and its size is, as we shown before, *0x0A*.

Then, in *parse_id3_tag+2D5*, there is a function call to copy the remaining decompressed data to the previously allocated buffer with the overflowed size:

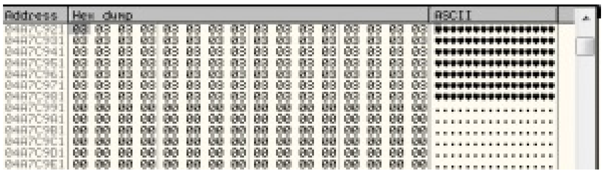
```
parse_id3_tag+2CA    push    [ebp+var_1C]    ; int
parse_id3_tag+2CD    lea     eax, [ebx+edi]
parse_id3_tag+2D0    push    eax             ; int >> value used as MAX counter(EBX+EDI)
parse_id3_tag+2D1    push    edi             ; int >> src buffer (with decompressed data)
parse_id3_tag+2D2    push    ecx             ; int >> dest buffer (allocated with overflowed size)
parse_id3_tag+2D3    mov     ecx, esi
parse_id3_tag+2D5    call    InternalMemcpy
parse_id3_tag+2DA    mov     ecx, [ebp+var_20]
```

If we look it in Ollydbg, we see this:




We can see that one of the parameters used in the function call is *0x326DAAAD.* This is the result of executing the lea eax, [ebx+edi] instruction located at *parse_id3_tag+2CD*. *EBX* had *decompression size – 1* while *EDI* pointed to the buffer where the *decompressed data +1* was, so, *0x02AAAAAAC + 0x07C30001 = 0x326DAAAD*. This value it's going to be used in *InternalMemcpy+D8* as one of the conditions in the loop. The other condition in the loop is to copy until a null byte is found.

After tracing some rounds of the loop, we can see how the copy operation was performed beyond the *0x0A* bytes:



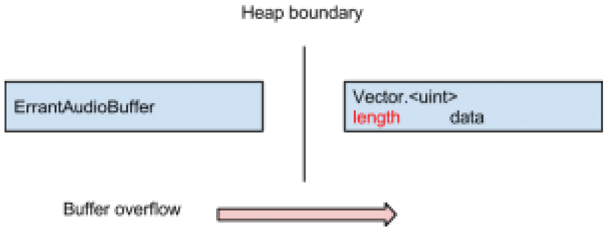
Boom, heap overflow in sight!

# Some words about the exploitation process

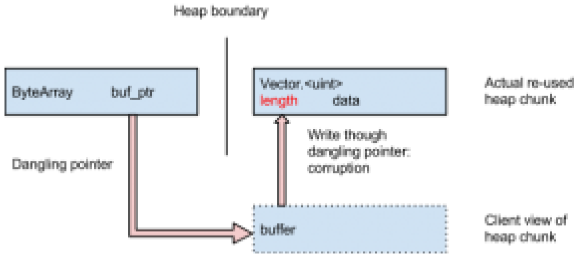
To summarize, the root vulnerability is an integer overflow that then ends up in a heap-based buffer overflow. In a very basic exploitation scenario, we must massage the heap in order to place a buffer with our data just after the buffer we overflow. Also, as we have protections such as *ALSR*/*DEP*/*CFG* that we must bypass in order to get reliable code execution, it would be a good idea to build some kind of read/write primitives that help us during the exploitation process.

For *Flash 18.0.0.209*, the most used technique to exploit *Adobe Flash* was the one described by ***Haifei L***i, entitled "Smashing the Heap with Vector: Advanced Exploitation Technique in Recent Flash Zero-day Attack". By incrementing the value stored in the *length* field of a *Vector.<uint>* object, we can read and write beyond the limits of the Vector.

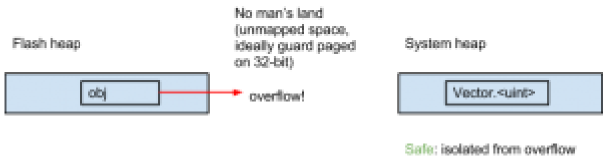
So, we could place a *Vector.<uint>* after our overflowed buffer and overwrite the metadata, the *length* field to be more accurate, and build our read/write primitive. The following picture (borrowed from *Project's Zero* post):




During the exploitation process, it ends up like this:




But starting from *Adobe Flash 18.0.0.209* we have some new challenges to face before getting reliable code execution because new exploit mitigation mechanisms were added in order to avoid the mentioned method from *Haifei Li*. For example, now *Flash* has some heap isolation mechanism called ***heap partitioning*** in which objects like *Vector.<uint>* are isolated from the *Flash* heap and stored in the ***System heap***. So, our exploitation scenario turns into something like this:




As if this were not enough, now we have better*ASLR* in the heap. To highlight:

- Allocations > 1 *MB* have better randomization
- In *x64*, the *Flash* heap ends up far away of any mapped area

Last but not least, a validation to the length fields of the *Vector.<*>* object was added.

If you want to read more detailed information about all these new mechanisms introduced in *Flash* you can go the the excellent write up published by *Project Zero*.

So, the only thing to think at this point is: we are screwed!.

In order to exploit this bug in a reliable way, at least in*Flash 18.0.0.209*, we are going to not only allocate in a deterministic way (to avoid randomization) an object that places after our overflowed buffer (*Flash* heap) but also it must have the capability to give us the possibility (as the length of a *Vector.<uint>* object did it before) to generate a read/write primitive.

So, please, stay tuned to see how this story ends!

# Notes

- **[1]** We only get to the zone where the overflow occurs just if the*JG* jump located at *parse_id3_tag+27B* is not executed, that is, if the condition is *False*. Once the data was decompressed it takes the first byte of the buffer, it decrements the *decompression size* by 1 and compares it with 3. If it is greater, it goes to the end of the loop and process the next frame. If not, it goes to the overflow zone.



Latest from CoreLabs

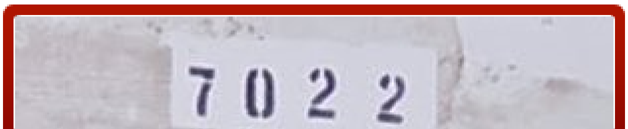
Core Security Selected as SC Magazine 2016 Excellence Award Finalist

## Leave a Comment

Name *

Email *

Website




Type the text

Privacy & Terms

reCAPTCHA™

Post Comment

## Categories

Blog Home
Latest from CoreLabs
What's New At Core

## Resources

Case Studies
Data Sheets
Quick Reads
Webcasts
White Papers
Videos

## Core Impact Pro Demo



## Core News Sign Up

**Name**

First

Last

**Email** *

**Agreement**

☐ I agree to receive information regarding Core products and services.

Submit

## RSS Feed

Subscribe to our RSS feed to get the latest blog posts - formatted for your favorite feed reader