Exploiting ActionScript3 interpreter

Boris Larin
Anton Ivanov
Bio (Boris Larin)

• Malware Analyst (Heuristic Detection and Vulnerability Research Team)
• RE has been my main passion for 8+ years
• Author of Kaspersky Academy’s Malware Reverse Engineering course for universities
• Regular writer on https://securelist.com/

@oct0xor
Bio (Anton Ivanov)

• Head of Advanced Threat Research and Detection Team
• Detecting exploits for 8 years
• Leads the targeted attacks research team
• Regular writer on https://securelist.com/

@antonivanovm
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.

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.

Header   FileAttributes tag   Tag   Tag   ...   End tag
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@swiffle.d(132): Range violation
--------------
0x00400508
0x00400C41
0x00436710
C:\RABCDAsm_v1.18>
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 😞

Let’s do it!
ActionScript3 processor module
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

```java
static function var120() :
{
    if (var16)
    {
        new BufferControlParameters(0,0);
        new C1();
        new C2();
        c3 = new C1();
        new C4();
        new C5();
        new C7();
        var16 = c3;
        var16.var38 = 4660;
        var122(0, var16);
    }
    if (var16.var38 <= 4660)
    {
        var12 = true;
        if (var16)
        {
            return;
        }
        C12.var130();
    }
    else
    {
        var100();
    }
}
}

public static function var122(param1:*, param2:*)
{
    if (var8)
    {
        var16.var36 = Low(param1);
        var16.Var37 = Hi(param1);
    }
    else
    {
        var16.var36 = param1;
        var16.o = param2;
        var121 = true;
        new Call();
        return;
    }
    catch(e:*)
    {
        return;
    }
}

public static function var123(): Object
{
    var121.BufferControlParameters = var16;
    var122.** = var109(_loc1_initialBufferTime);
    var123.** = var109(_loc1_playBufferTime);
    return {
        "16": _loc2_low,
        "14": _loc2_hi,
        "4": _loc3_low,
        "6": _loc3_hi
    };
}
```

Var130 launches shellcode using a standard technique
Exploit

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

Var130 launches shellcode using a standard technique.
Exploit

Var130 launches shellcode using a standard technique

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

Where is the vulnerability?
First hints
First hints
AVM2 core

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

```c
// Verify the given method according to its type, with a CodeWriter
// pipeline appropriate to the current execution mode.
void BaseExecutor::verifyMethod(MethodInfo* m, Toplevel* toplevel, AbcEnv* abc_env)
{
    AvmAssert(m->declaringTraits())->isResolved());
    m->resolveSignature(toplevel);
    PERFNTPROF_BEGIN("verify-ticks");
    MethodSignature 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);
    PERFNTPROF_END("verify-ticks");
}
```
/** @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
};
/** @} */
```c
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 block
    // 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;
}
```
Interpreted

- try {} block
- static Init
Verification

```c
// 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 above get cleaned via their
// destructors, and passed-in CodeWriters are still valid.
void BaseExeImageRelation::verifyCommon(MethodInfo* m, MethodSignature& 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->unregister(); // Clean up verifier.
        vcoder->cleanup(); // Cleans up all coders.
        core->throwException(exception);
    }

END_TRY
```

```c
// 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

```c

const uint32_t argc = imm30h;
checkStack(argc-1, 1);

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

Always throw verifyFailed()
Exceptions in Flash

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

verifyFailed:

In which scenario would a legitimate SWF need to catch bytecode verify errors?
• Function var122 is called twice
• At first attempt verifyFailed exception is caught
• At second attempt exception is not thrown!
• Code interpreted without verification!
Vulnerability

```c
// 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, 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[n] == 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

#define VMCCG BFCARG OPTIMIZATION

void Verifier::parseExceptionHandlers()
{
    if (!abc_exceptions()) {
        AvmAssert(tryFrom && tryTo);
        return;
    }

    const uint8_t* pos = code_pos + code_length;
    int exception_count = toplevel->readU32(pos);  // will be nonnegative and less than 0x00000000
    if (exception_count != 0)
    {
        if (exception_count == 0 || (size_t(exception_count-1) >> SIZE_T_MAX / sizeof(ExceptionHandler))
            VerifyFailed(willlegalExceptionHandlerError);

        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

(1) On first run – set exceptions

```c
void Verifier::parseExceptionHandler()
{
    if (!abc_exceptions()) {
        AvmAssert(tryFrom && tryTo);
        return;
    }

    const uint8_t* pos = code_pos + code_length;
    int exception_count = toplevel->readU32(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(illegalExceptionHandlerError);

        ExceptionHandlerTable* table = ExceptionHandlerTable::create(core->GetGC(), exception_count);
        ExceptionHandler*handler = table->exceptions;
        for (int i=0; i < exception_count; i++, handler++)
        {
            handler->from = toplevel->readU32(pos);
            handler->to = toplevel->readU32(pos);
            handler->target = toplevel->readU32(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;
        }
    }...
```

```c
// Verify in two passes. Phase 1 does type modeling 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().
...
```

```c
parseBodyHeader(); // set code_pos & code_length
checkFrameDefinition();
parseExceptionHandler(); // resolve catch block types
checkParamO();
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);
```
Vulnerability

(1) On first run – set exceptions

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

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

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

<table>
<thead>
<tr>
<th>Line</th>
<th>Status</th>
<th>Source</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>103</td>
<td>Fixed</td>
<td><a href="mailto:forshaw@google.com">forshaw@google.com</a></td>
<td>Windows Acrobat Reader 11 Sandbox Escape in MoveFileEx IPC Hook CProjectZeroMembers</td>
</tr>
<tr>
<td>108</td>
<td>Fixed</td>
<td><a href="mailto:cevans@google.com">cevans@google.com</a></td>
<td>Flash logic error in bytecode verifier CProjectZeroMembers</td>
</tr>
<tr>
<td>107</td>
<td>Fixed</td>
<td><a href="mailto:hawkes@google.com">hawkes@google.com</a></td>
<td>Microsoft Office 2007 TTDdeleteEmbeddedFont handle double delete CProjectZeroMembers</td>
</tr>
<tr>
<td>108</td>
<td>Fixed</td>
<td><a href="mailto:hawkes@google.com">hawkes@google.com</a></td>
<td>Microsoft Office 2007 TcAbPlcfindTxt/TcPlguid/Un memory corruption CProjectZeroMembers</td>
</tr>
<tr>
<td>109</td>
<td>Fixed</td>
<td><a href="mailto:cevans@google.com">cevans@google.com</a></td>
<td>Flash heap overflow in bytecode verifier CProjectZeroMembers</td>
</tr>
<tr>
<td>110</td>
<td>Fixed</td>
<td><a href="mailto:hawkes@google.com">hawkes@google.com</a></td>
<td>Microsoft Office 2007 PapxFkp rbx bOffset memory corruption CProjectZeroMembers</td>
</tr>
<tr>
<td>111</td>
<td>Fixed</td>
<td><a href="mailto:hawkes@google.com">hawkes@google.com</a></td>
<td>Microsoft Office 2007 VBA ExtendedControl use-after-free CProjectZeroMembers</td>
</tr>
<tr>
<td>112</td>
<td>Fixed</td>
<td><a href="mailto:cevans@google.com">cevans@google.com</a></td>
<td>Adobe Flash incorrect jit optimization with op_pushwith CProjectZeroMembers</td>
</tr>
<tr>
<td>113</td>
<td>Fixed</td>
<td><a href="mailto:jsenr@google.com">jsenr@google.com</a></td>
<td>Flash 14 on IE11, readAV crash on xmm instruction CProjectZeroMembers</td>
</tr>
<tr>
<td>114</td>
<td>Fixed</td>
<td><a href="mailto:cevans@google.com">cevans@google.com</a></td>
<td>Adobe Flash incorrect jit optimization with op_pushscope CProjectZeroMembers</td>
</tr>
<tr>
<td>115</td>
<td>Fixed</td>
<td><a href="mailto:cevans@google.com">cevans@google.com</a></td>
<td>Adobe Flash incorrect jit optimization with op_setglobalslot CProjectZeroMembers</td>
</tr>
<tr>
<td>116</td>
<td>Fixed</td>
<td><a href="mailto:cevans@google.com">cevans@google.com</a></td>
<td>Flash heap buffer overflow calling Camera.copyToByteArray() with a large byte array CProjectZeroMembers</td>
</tr>
</tbody>
</table>

But targeted another part of a check…

- if (pc < tryTo && pc >= tryFrom && (opcodeInfo[opcode].canThrow))
CVE-2017-11292 fix

- Code found on GitHub

```cpp
void Verifier::parseExceptionHandlers()
{
    if (info->abc_exceptions())
    {
        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

```cpp
void Verifier::parseExceptionHandlers()
{
    if (info->abc_exceptions())
    {
        CFG_HAPPY_RUN
        // In halfmoon, Analyze mode, Verifier is run twice.
        // Exception parsing was happening twice and duplicate scope traits were generated.
        // Which lead to verify error for following sample action script code
        // function f1: void {
        //     try {
        //         // <code inside try>
        //         // } catch(e) {
        //         //     function f2: void{
        //         //         // <function - body>
        //         //     }
        //     }
        //     }
        // }

        // 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?
Exploitation

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

Var16 is passed as “this”!
Exploitation

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

- Debugging of interpreted code
  - avmplus::interpBoxed – main function responsible for interpretation
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?

“We debug 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
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 versions of Flash
- Not very obvious how to get it to work with newer versions

JIT debugging - 2014
DbgFlashVul - First (?) public release of Flash WinDbg plugin to debug JIT

- Works on different Flash versions with the use of signatures

```
! EnableTraceJit

0:000> !SetBaseAddress 0x620000
0:000> !EnableTraceJit

*** ERROR: Symbol file could not be found. Defaulted to export symbols

0:000> g
Call [Function1/createEmptyFunction]
Call [Object1/_dontEnumPrototype]
Call [Object1/initialize]
Call [flash cc::Rectangle]
Call [flash display::Stage]
Call [flash display::DisplayObjectContainer]
Call [flash display::InteractiveObjectVector <flash.display::Stage3>]
Call [flash display::DisplayObject]
Call [flash events::EventDispatcher]
Call [test]
Call [flash display::Sprite]
Call [test/launch]
Call [test/Starting]
```

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#static_method

4. if package name is empty then no ‘package::’ prefix

```
! 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
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
JIT codegen

```c
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 has ReachableExceptions() 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, 18h
mov edx, dword ptr [ebp+var_170+4]
mov ecx, dword ptr [ebp+var_168]
mov ebx, eax
mov [ebp+var_124], 56h ; convert_u
mov [ebp+var_124], 57h ; setlocal 7
mov [ebp+var_124], 59h ; getlocal2
mov [ebp+var_124], 58h ; pushstring "VirtualProtect"
mov [ebp+var_124], 58h ; getlocal
mov [ebp+var_124], 5Fh ; callproperty QName(_G, __3), 2; "PackageNamespace()"

mov eax, [ecx+4]
lea ecx, [ebp+var_160]
mov DWORD PTR [ebp+var_160], edx
mov eax, eax
mov DWORD PTR [ebp+var_158+4], offset unk_83C8BAC0
mov DWORD PTR [ebp+var_158], eax
mov edi, [edi+4]
sub esp, 4
push ecx
push 5
push edi
call eax
add esp, 10h
mov [ebp+var_124], 62h ; convert_u
mov [ebp+var_124], 63h ; setlocal 8
mov [ebp+var_E9], eax
mov [ebp+var_124], 65h ; getlex QName(_G, __2); "PackageNamespace()"
lea ebx, [ebp+28h]
mov DWORD PTR [ebp+var_160+4], ebx
lea edx, [ebp+var_158]
mov ecx, DWORD PTR [ebp+var_160+4]
call sub_689900E0
mov ebx, [eax+10h]
mov [ebp+var_124], 67h ; getproperty QName(_G, __17); "PackageNamespace()"
test ebx, ebx
jz loc_7560BC4
```
DEMO
Conclusions

- AVM core was and still is a source of critical vulnerabilities
  - Bypass of bytecode verification
  - 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
• CVE-2018-5002

• Exception handler will be called if instructions in range from 0x4666 to 0x466A cause exception

• In this range there is only one instruction: “jump”

• “jump” never causes exception…
**Bonus**

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

- Let's take a look at li8 handler

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

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

```c
#define MOPS_RANGE_CHECK(addr, type) \n    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

```c
} // End TRY

CATCH (Exception *exception)
{
    // find handler; rthrow 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? 😊
```
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

```c
} // End TRY

CATCH (Exception *exception)
{
    // find handler; rthrow 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?

@oct0xor – Boris Larin
@antonivanovm – Anton Ivanov