Reflash: practical ActionScript3 instrumentation with RABCDAsm

Jarkko Turkulainen

F-Secure
jarkko.turkulainen@f-secure.com

June 16, 2016

Abstract

Adobe Flash has been announced dead for almost ten years now. But it is still here, installed on almost every computing device on Earth, and it is getting more attention because of the alarming rate of zero-day vulnerabilities we have been witnessing over the past few years. Even though the Flash platform is such wide spread, it is still lacking comprehensive binary analysis tools.

In this paper we try to address this shortcoming with a set of tools and techniques for dynamic ActionScript3 (AS3) instrumentation and analysis. The techinques described in the paper covers generic AS3 opcode instrumentation and stack trace generation, and finally a toolchain to operate generated data outside the AS3 execution environment is presented.

The presented toolchain consists of a service component that listens on active network connections and attempts to inject an instrumentation module to flash files on the wire. The instrumentation module produces a generic stack trace that is sent back to the service for offline analysis. The last component of the framework is a set of tools for building and manipulating a SQL database of stack events. The paper shows that the presented client/server-architecture is scalable and relatively stable in a hostile execution environment.

I. Introduction

Adobe Flash platform is a very common target for malicious software. It has a huge install base, it is still enabled by default on most browsers and as a very complicated interpreted software platform it also has many bugs. Writing software for the Flash platform is relatively fast and its deep interaction with other Web tehnologies, such as *JavaScript* and *VBScript*, offers attractive opportunities for malicious actors.

It is quite common to include Flash components in so-called Exploit Kits, either as a form of payload delivery mechanism, or as a main target for vulnerability exploitation. As a part of *Exploit Kit* (EK), the malicious Flash can automate exploitation, obfuscate the EK actions, collect statistics and steal personal information from target systems.

Reverse engineering a complicated software stack including web pages, JavaScript, Flash and possibly other interpreted languages, ending up to a physical CPU using software bug, is a challenging and time-consuming task. Flash files usually contain several layers of embedded Flash files for further obfuscating the purpose of their existence. The nature of such interpreted environments offer good opportunities for complicated code obfucation, but on the other hand, it also lacks the control compared to native execution environments.

For complicating the matter even further, Flash is usually run on a Web environment, and it is dependent on external resources, such as JavaScript and other Web content. This again creates a twofold situation. On the other hand, having the entire Internet as a computing resource creates reverse engineering problems that are theoretically very difficult to solve [1] but it also rules out some problems existing on native platforms, such as timing attacks because timing network connections is not reliable.

The research described on this paper concentrates exclusively on the Flash part of this complete picture. One of the challenges the author had to face was almost complete nonexistence of binary analysis and debugging tools for the Flash platform. For native platforms, such as Intel x86/x64, there exists already multitude of good binary research tools, but not so much for the Flash platform. As an interpreted execution environment, the Flash relies a lot on the native execution runtime libraries, which do offer very good debugging support, but in order to use them, one would have to compile the software with debugging information. Once compiled, the code cannot anymore directly manipulate its instruction stream, which means that Flash software without debugging information is out of reach of debuggers and other profiling tools. This again is a clear difference to native platforms, where one loses a lot of information without debugging symbols, but it is still possible to manipulate the code, for example insert breakpoint instruction (INT3 on Intel) and catch the exception with external debugger tools.

One of the main motivations for the author was to create a set of tools analogous to familiar native platforms debugging tools. The very basic requirement for such a tool is the ability to run without any debugging information. This is absolutely required, since the malicious software usually leaves out such reverse engineering aids. The other major requirement is stablility and efficiency because malicious software quite often exploits the limits of execution and uses exotic platform features.

Proof of concept of this research is called *Reflash*. It is a set of tools for analyzing binary Flash files without any debugging information. Reflash grew out of curiosity for a quite promising platform. It is not limited only to reverse engineering use, altough that is its main purpose.

Reflash can run Flash files written in ActionScript, targeting the Adobe ActionScript Virtual Machine version 2.

i. ActionScript 2/3

The language used to write software for the Flash platform is called ActionScript. Its current version is 3. ActionScript is an object-oriented language originally designed by Gary Grossman [4]. ActionScript is a dialect of ECMAScript, standardized scripting language specification. Other known dialects include JavaScript and JScript.

Reflash operates on a lower level, using the underlying Virtual Machine instruction set and features. However, some parts of the instrumentation module is written in ActionScript, for conveniece.

The underlying parsing engine *RABCDAsm* [3] supports ActionScript 2 and 3, but programs written in version 2 has not been tested in the context of Reflash.

ii. Adobe AVM2

The execution environment for ActionScript sofware is ActionScript Virtual Machine, shortly AVM. For ActionScript 2/3, the execution environment is called AVM version 2, shortly AVM2.

AVM2 is a stack-based machine, unlike most native platforms and also some interpreted platforms such as *Dalvik* [5]. AVM2 instruction set consists of up to 256 different opcodes, most of them handling data values on stack, such as arithmetic operations.

AVM2 execution environment consists of method body (instructions), data stack, heap, local registers and scope stack. Of these the most relevant parts for Reflash are the actual instructions, data stack and local registers.

Instructions are a stream of binary operation codes (opcodes), such as ADD, CALL or JUMP. AVM2 instruction are *high-level opcodes*, meaning they can operate directly with ActionScript language constructs, such as classes and arrays.

Data stack is a memory area used for arithmetic operations, call parameters and other data. AVM2 stack operates on a familiar concept of pushing and popping data from the stack. There is no direct way to manipulate stack pointer.

Local registers are quite similar to stack, but instead of indirect data manipulation, data is moved in and out of local registers with specific opcodes (GETLOCAL/SETLOCAL). Both stack and local register can hold any type of value.

Heap is a data area managed by the runtime and it cannot be addressed directly from the AVM2. Only way to observe data in heap is through the objects created on the heap, such as new classes.

For more comprehensive introduction to AVM2, Adobe's AVM2 overview is a good reference [2]

iii. Reflash

Reflash toolkit consist of several components. In the following sections *Reflash* refers both to the instrumentation engine and to the overall concept. If there is a need to distinguish the two, it will be indicated clearly. When referring to the instrumentation engine, a term *Reflash executable* is used.

- *Reflash executable* is the actual underlying instrumentation engine. It is implemented as a standalone executable that can also be be used independently from other components.
- *Instrument* is an instrumentation module injected by the Reflash executable to all analyzed flash files. It is responsible for generating stack trace of instrumented flash program.
- *Proxy* is a service component acting as a HTTP proxy. It attempts to capture any flash file requests and executes Reflash executable before returning the actual flash content to the client.
- *Dbtool* is used for building a SQL database from stack trace. There are also some useful features in Dbtool, for example it can run YARA [10] over the database or produce a human readable decompilation from the stack trace.
- *Replay* is a graphical frontend for the SQL database. It presents a debugger-like interface that can be used for analyzing the database.
- *reflasher.py* is a driver script that binds together Proxy and Dbtool for automatic stack data collection.

In the following sections, some of the above components are described with details.

II. Reflash executable

Reflash is the main component of the toolkit. It is a standalone executable, written in D-language [6]. The choice of programming language was natural, since Reflash interfaces the *RABCDAsm* [3] classes directly. Reflash has an integrated disassembler, assembler and generic flash file injection mechanism. The purpose of flash file injection is to have a flexible approach for writing instrumentation modules.

i. Opcode instrumentation mechanism

When Reflash is instructed to perform opcode instrumentation, it will disassemble all instructions in a given flash file and replace opcodes with a generated assembly code that collects the stack arguments and transfers them to an instrumentation module written with ActionScript. Instrumentation decision is made by consulting a user-supplied configuration file where user can define a list of regular expressions that are evaluated against the opcode names during the disassembly. In the most generic setup, only one hook may be defined, a wild card ".*".

Opcode names correspond directly to names presented in the Adobe AVM2 document [2]. In addition to the normal opcodes, there is one pseudo opcode, *method_entry* available for instrumentation. This hooks the method prolog but instead of stack arguments, it collects arguments from local registers according to a protocol described in [2]: *this* pointer is always in the first local register, and rest of the arguments are placed to following local registers.

If an opcode was selected for instrumentation, certain preconditions needs to be specified before the actual code injection:

- Amount of stack items to be evaluated. This can be reliably determined from static compile time information, such as the actual operation the opcode is performing (for example: ADD always adds two top values on stack) or the amount of arguments for function calls.
- How many of the stack items are *object* arguments and *function* arguments. Object arguments are usually the *this* pointer, namespace and/or name information for runtime multinames [2] and other runtime-specific stack values, or for example arithmetic operation arguments. Function arguments are arguments for method calls and class constructors. Object arguments are never modified, but function arguments can be modified by the instrumentation module. There are always at least one object argument.
- Session context for the opcode. Session consists of symbolic user-defined prefix (usually just a single character, such as "s"), a number indicating flash file run in the session, a unique method identifier and an opcode index. This information is later sent back from instrumentation for connecting a specific dynamically generated stack event to statically generated data, such as disassembly.
- Available local registers. Instrumentation needs to be aware of local registers used by the code for avoiding collisions with the original code. This information is always available statically in flash file method body structure.
- Instrumentation module API. Instrumentation code needs to refer to the ActionScript module with a specific API that consists of user-defined package name, a fixed class *Instrument* and a fixed method name *InstrumentStack*. Method InstrumentStack takes three parameters: session context, amount of object arguments and an array of stack items. InstrumentStack returns back an array consisting of function arguments. If an argument was not modified, item in the array is *undefined*.

With the information defined above, a generic form of opcode instrumentation can be presented as following listing:

```
_start:
  setlocal        A+1
  setlocal        A+2
  ..
  setlocal        A+N
  getlocal        A+N
  ..
  getlocal        A+2
  getlocal        A+1
  newarray        [N]
  setlocal        A
  getlex          PACKAGE:Instrument
  pushstring      SESSION_ID
  pushint         X
  getlocal        A
  callproperty    :InstrumentStack (3)
  setlocal        A
  getlocal        A+N
  ..
  getlocal        A
  callproperty    :pop (0)
  dup
  pushundefined
  ifeq            L2
  jump            L3
L2:
  pop
  getlocal        A+2
L3:
  getlocal        A
  callproperty    :pop (0)
  dup
  pushundefined
  ifeq            L4

  jump            L5
L4:
  pop
  getlocal        A+1
L5:
  OPCODE
```

Listing 1: *Generic instrumentation opcodes*

Explanation of code listing line by line:

- *lines 2-5* Store stack items to local registers. Symbol *A* refers to first available local register, used later for saving the stack items as an array. Symbol *N* is the total number of stack items determined statically.
- *lines 6-9* Push back local registers to stack in reverse order.
- *line 10* Create an array of *N* items.
- *line 11* Store created stack item array to local register *A*.
- *line 12* Get *PACKAGE.Instrument* where *PACKAGE* is a user-defined package name corresponding the the package name used in instrumentation ActionScript module. *Instrument* is a fixed function name.
- *line 13* Push the session identifier to stack.
- *line 14* Push the number of object arguments to stack.
- *line 15* Push the stack item array to stack.
- *line 16* Call InstrumentStack with three parameters(array, object argument count, session identifier).
- *line 17* Store the returned array in *A*.
- *line 18* Start pushing the object arguments from local registers to stack, up to *X*. In this procedure, there is no need for any type checking.
- *lines 20-21* After *X* object arguments, pop first function argument to stack.
- *line 22* Duplicate the stack item.
- *line 23* Push *undefined* to stack.
- *line 24* If the stack items are equal, branch to L2.
- *line 25* If the stack items are not equal, branch to L3, leaving stack item intact.
- *line 27* Pop *undefined* from stack.
- *line 28* Push function argument from local register.
- *lines 30-39* Repeat steps at 20-28, ending up to last function argument *A+1*.
- *line 41* Call original opcode.

This method of moving stack arguments as an array was originally deviced by Jeong Wook Oh with his tool *FlashHacker* [7]. The reason why stack items has to be recycled using local registers is to retain the types. First version of Reflash was executing NEWARRAY and saving the full array returned by InstrumentStack directly to stack, without any type coercion, which led to obscure failures. Second revision was doing some manual type coercion for values indicated by debug Flash Player, but it proved to be futile. The current implementation is very careful to store only modified function arguments to stack from returned array, all other items are restored from local registers.

ii. Opcode relocations

When Reflash executable is inserting instrumented code blocks to the instruction stream, it has to relocate some of the branch targets and exception handler targets in the method body. This procedure is straightforward: if the target is after inserted code block, it needs to be adjusted with the size of the inserted block. This needs to be done to all branch targets, after each instrumentation. The following diagram illustrates the procedure:

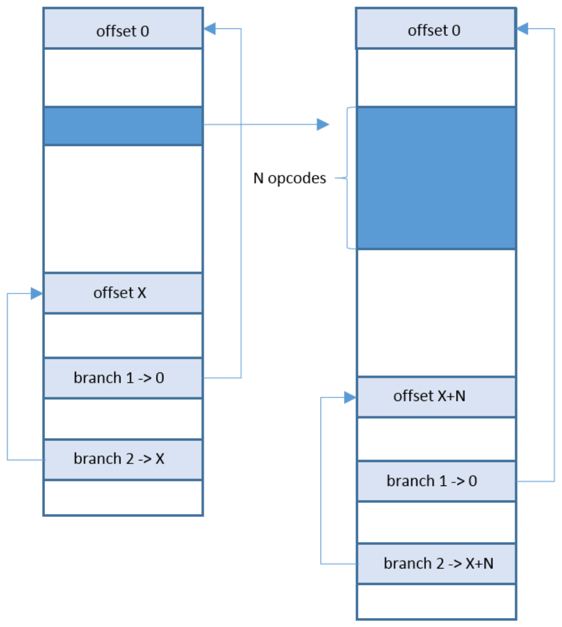


Figure 1: *Branch target relocations*

iii. Flash injection mechanism

Flash injection mechanism is generic mechanism for including the *Instrument* to the analyzed flash file. It is implemented using the built-in *RABCDAsm* engine [3]. When the analyzed flash file is being disassembled on disk, the injected flash is also disassembled to a subdirectory and its class files are included with RABCDAsm include-directive.

Injection mechanism is completely agnostic about the underlying API so it cannot do any verification for instrumentation engine and injected flash code interoperation. That is completely the responsibility of the user.

```

#include "wrutrofsoudkqvr.script.asasm"
#include "g/cdxjkyjrlav.script.asasm"
#include "g/iepfknfpdjngrrr.script.asasm"
#include "g/ekxapdubdwhxqmz.script.asasm"
..
#include "../sub/Instrument.script.asasm"
```

Listing 2: *Example includes*

iv. Metadata

In order to later correlate stack trace to the originally disassembled code, specific metadata is generated by the Reflash executable. This metadata is a simple stream disassembly of the original code.

```

0-0:4:wrutrofsoudkqvr/instance/init:
  00000000   getlocal0
  00000001   pushscope
  00000002   pushbyte            0
  00000003   setlocal            11
  00000004   pushbyte            0
  00000005   setlocal            7
  ..
```

Listing 3: *Example disassembly metadata*

In the above listing, *0-0* is the Session identifier. Leftside column is an opcode index in the unique method body *4; wrutrofsoudkqvr/instance/init*. Later when the Instrument returns stack event, it can be correlated to the metadata for presenting contextual background for the event.

v. Performance

For evaluating Reflash executable performance, we selected three files based on file size and features. These files were instrumented with four different configurations. The execution time was measured with Unix *time* command.

Small file is a malicious file of 717 bytes. It does nothing else than load additional flash payload from network using method *flash.display.Loader::load()*. This is fairly typical procedure used by flash components in Exploit Kits and other malware.

Typical file presents the most common type of flash malware. It is a compressed file with size of 69,5KB, consisting of 10138 opcodes. It is loading two embedded flash files with *flash.display.Loader::loadBytes()*. The final flash payload attempts to exploit vulnerability in the Flash Player. Embedded payloads are extracted and prepared with various manipulations over *ByteArrays*.

Large file is a benign game file of 13,0MB. The file contains total of 730608 opcodes.

Configurations present sets of instrumented opcodes.

First configuration is very minimalistic set of hooks instrumenting only CALL instructions.

Second configuration is a conservative set consisting of the following regular expressions:

```

"method_entry",
"call.*",
"init.*",
"setprop.*",
"construct.*"
```

Listing 4: *Configuration 2*

Third configuration is comprehensive set of hooks instrumenting arithmetic operations, bitwise manipulations and other typical opcodes in addition to the second set.

Fourth configuration is hooking only pseudo instruction *method_entry*, used for evaluation of the raw performance.

Table 1: *Small file*

Conf	Instrumented	Size	Time
1	3/59	3611 bytes	0, 0s
2	8/59	3702 bytes	0, 0s
3	14/59	3776 bytes	0, 0s

Table 2: *Typical file*

Conf	Instrumented	Size	Time
1	74/10138	73,0K	0, 1s
2	263/10138	74,8K	0, 1s
3	3487/10138	96,2K	0, 5s

Table 3: *Large file*

Conf	Instrumented	Size	Time
1	9763/730608	13,5M	5.79s
2	97714/730608	18,5M	44.5s
3	267455/730608	26,8M	238.0s
4	1742/730608	12,8M	3.6s

The results show that Reflash executable performs well in typical situations, but the performance starts to deteriorate when the amount of instrumented opcodes is in range 5000-10000. The execution time appears to grow linearly, which indicates there is room for optimization in the instrumentation algorithm itself. The underlying parsing engine *RABCDAsm* [3] performs reasonably well even in extreme situations, as we can see from large file configuration 4.

It should be noted that this section discusses only the perfomance of executing the static instrumentation. The performance of instrumented files is discussed in the following section.

III. Instrument module

Instrument module is the code injected by Reflash executable. It is written in ActionScript and compiled as a standard flash file.

As with any Internet content, flash is designed to run *untrusted* content over the network. This places some difficult restrictions on the data collection. For example, we cannot access disk directly for efficient data logging. Flash player also runs out of memory and CPU time if the data collection requires too much resources. All this makes the Instrument data collection a challenging task.

Typically, similar applications collect data on disk and do some preprocessing over the data, such as presenting textual information in readable form. Because disk access is restricted and any preprocessing require VM resources, we have developed a method for minimizing the amount of processing inside the VM. Stack argument processing has been divided in two parts: first the arguments are collected and packed in binary format and transferred over a binary TCP connection. After that the data is processed outside the VM.

i. Argument collection

Arguments are transferred from the instrumentation hook as an array of stack items. This array is rearranged to a new array consisting of:

Table 4: *Returned Array*

Index	Item
0	Session
1	Type of argument 0
2	Argument 0 as ByteArray
3	Type of argument 1
4	Argument 1 as ByteArray
N*2 - 1	Type of argument N
N*2	Argument N as ByteArray

All arguments are presented as *ByteArray* because AS3 ByteArray method *writeObject* is used for formatting the argument in Action Message Format [8].

The argument collection can be presented as the following ActionScript-like pseudo code:

```
function GetArguments(array, session)
{
    var len:int = array.length;
    var ba:ByteArray = new ByteArray();
    var ret:Array = new Array((len*2)+1);

    ret[0] = session;
    for(var i:int = 0; i < len; i++)
    {
        ba.writeObject(array[i]);

        ret[(i*2)+1] = ArgumentType;
        ret[(i*2)+2] = ba;
    }
    return ret;
}
```

Listing 5: *Argument collection*

session in the above listing is the session context prepared by Reflash executable for the particular hook. *ArgumentType* is class name as retuned by *getQualifiedClassName()*.

Instrument module tracks the visited code locations based on the session identifier for limiting the amount if collected stack data. If the amount of visits on particular location exceeds a preconfigured threshold (by default 20), data is not collected. This is to prevent exhausting the AVM2 resources, while still providing some details about loops. With the default settings, it is still possible to get an idea on what the target is running inside loops.

In the actual implementation, only safe types are written directly with *writeObject*. Safe types include *final* classes, like String and Integer, and classes from package *flash.**. This safety measure is to prevent infinite loops in class getters. If the class is not safe to write, explicit coercing to ByteArray is tried and if that fails, zero-sized ByteArray is written to the return array.

ii. Logging

After preparing the return Array, Instrument module packs up the Array again with writeObject() and then sends the packed object over a binary TCP connection to Proxy module. In effect, there is no log format or protocol present. The log simply consists of series of AMF-packed data objects that can be later unpacked with standard tools.

iii. loadBytes instrumentation

In addition to raw data logging, the Instrument module can also manipulate call arguments. This feature is utilized in one particular use case, namely instrumentation of embedded flash content. Many malware uses embedded flash files as a form of obfuscation, so it is essential for Reflash to detect this and run Reflash executable also for any detected embedded flash content, otherwise it will lose its visibility to the overall execution.

Embedded flash content loading is supported in ActionScript with method *flash.display.Loader::loadBytes*, which loads content from ByteArray. In practise, this method is always executed with CALL opcode, so the absolute minimum configuration should include all calls. Instrument module will detect CALL instructions with flash.display.Loader object and ByteArray argument. If this condition is present, it sends the ByteArray over a HTTP connection back to Proxy module for instrumentation. After receiving back instrumented content, the ByteArray argument is replaced with the new content.

This concept is simple in principle, but quite challenging to implement in the concurrent Flash Players. Flash Player runs AS3 code usually in a single thread and all networking is asynchronous, based on callbacks. Because the ByteArray content needs to be replaced synchronously, there is no definitive way to accomplish this task. The current implementation of Instrument uses AS3 *ExternalInterface*, which enables synchronous remote procedure calls from AVM2. The embedded flash content is wrapped as an argument to JavaScript function in the context of browser and the JS code can then perform synchronous HTTP request to Proxy.

iv. Performance

The performance of instrumented files is not trivial to measure, because there are too many details involved. For that reason, a simple test file was prepared. The test file runs a series of simple loops, copying bytes from ByteArray to another:

```
var ba:ByteArray = new ByteArray();
var bb:ByteArray = new ByteArray();

var len:int = ba.length;

var start:Number = getTimer();
var i:int;
var j:int;
var x:int;

for (i = 0; i < N; i++)
{
    ba.position = 0;
    bb.position = 0;
    for (j = 0; j < len; j++)
    {
        x = ba.readByte();
        bb.writeByte(x);
    }
}
..
    [repeated Round times]
..
for (i = 0; i < N; i++)
{
    ba.position = 0;
    bb.position = 0;
    for (j = 0; j < len; j++)
    {
        x = ba.readByte();
        bb.writeByte(x);
    }
}

trace(getTimer() - start);
```

Listing 6: *Perfomance test loop*

The fixed number *N* is the threshold for Instrument loop counter. This test procedure was repeated 10, 100, 1000 and 10000 times, producing the following results when measuring the execution time with *getTimer()*:

Table 5: *Instrumented code performance test*

Rounds	Instrumented	Time
10	96/375	40ms
100	726/2985	375ms
1000	7030/29095	3, 6s
10000	70030/290095	37, 1s

The test run with 10000 rounds was a good stress test also for the Reflash executable. The instrumented main class contained a method body with around 1,2 million instructions. As expected, the execution time grows linearly.

IV. Proxy module

Proxy module has two operational modes: a *sandbox* mode, which only serves the initial flash file to the browser and a *live* mode, which passes all connections through. When the Proxy detects a request to flash file, it will run the Reflash executable and return instrumented content instead of the original flash file.

In addition to the flash instrumentation, the Proxy is collecting the AMF-packed log generated by the Instrument.

Proxy is implemented as an inline *mitmproxy* script [9] .

V. Dbtool

Dbtool is a python module responsible for parsing the AMF-packed binary log, populating a SQL database from the parsed stack events and other tasks, such as producing readable reports from the database. In addition to stack events, other data, such as the metadata produced by Reflash executable, is written to the database.

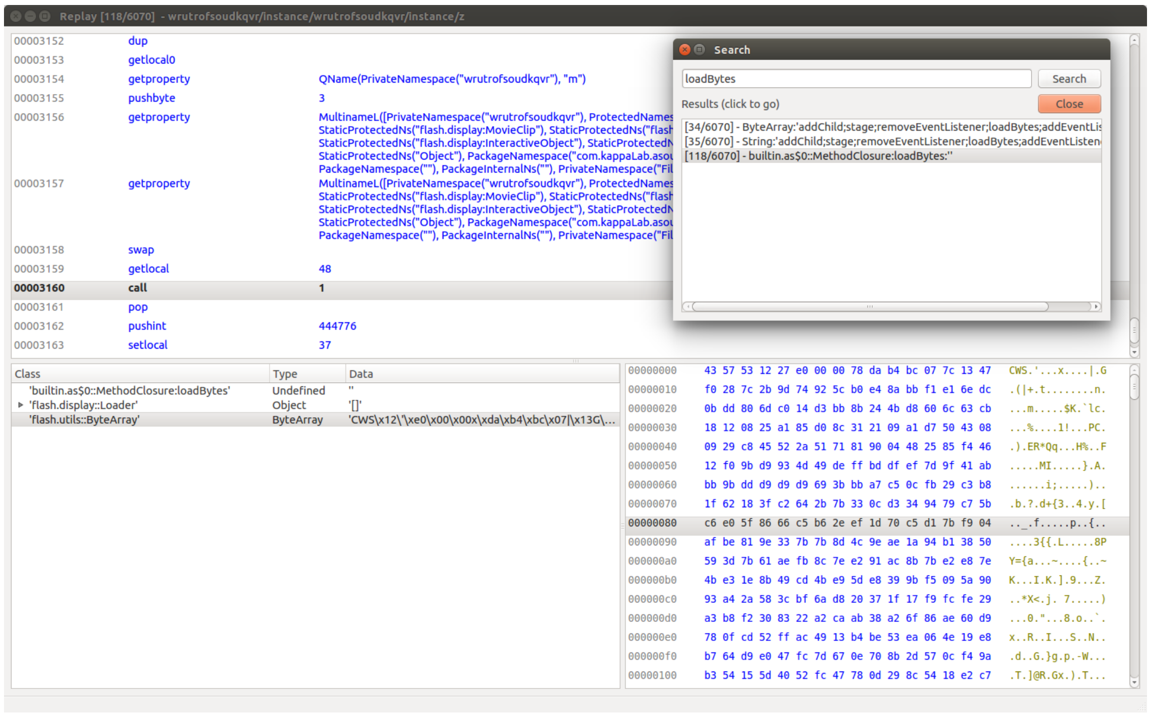
```
nw22/instance/nw22/get_skotina:
{
  [00005586]  (propvoid)::writeBytes
  (
    obj:ByteArray:''
    arg:ByteArray:'\xeb\x12X1\xc9f[...]
    arg:Integer:'0'
    arg:Integer:'1426'
  )
  [00005587]  (propvoid)::writeMultiByte
  (
    obj:ByteArray:'\xeb\x12X1\xc9f[...]
    arg:String:'uqmyijenjr'
    arg:String:'iso-8859-1'
  )
  [00005588]  (propvoid)::writeByte
  (
    obj:ByteArray:'\xeb\x12X1\xc9f[...]
    arg:Integer:'34'
  )
```

Listing 7: *Example database report*

VI. Replay

Replay is a graphical frontend for the SQL database prepared by Dbtool. With the stack events and metadata in SQL database, it can present a coherent view of AS3 execution. Replay was modelled after popular debugging tools for presenting a disassembly view, stack view and a hex data view. Some of the features included in Replay:

- Present a stream disassembly of the current stack event's method.
- Search textual data over stack items and disassembly.
- Step forward and backward the stack events.
- Go to specific stack item.
- View selected stack item in hex view.
- Save data from hex view to disk.
- Set breakpoints on stack events, run, single step.
- Run YARA [10] over stack items and disassembly.
- IDA-style [11] disassembly navigation using ENTER and ESC keys.


Replay [118/6070] - wrutrofsoudkqvr/instance/wrutrofsoudkqvr/instance/z

```
00003152    dup
00003153    getlocal0
00003154    getproperty    QName(PrivateNamespace("wrutrofsoudkqvr"), "m")
00003155    pushbyte       3
00003156    getproperty    MultinameL([PrivateNamespace("wrutrofsoudkqvr"), ProtectedNames
                           StaticProtectedNs("flash.display:MovieClip"), StaticProtectedNs("flash
                           StaticProtectedNs("flash.display:InteractiveObject"), StaticProtectedN
                           StaticProtectedNs("Object"), PackageNamespace("com.kappaLab.asou
                           PackageNamespace(""), PackageInternalNs(""), PrivateNamespace("Fil
00003157    getproperty    MultinameL([PrivateNamespace("wrutrofsoudkqvr"), ProtectedNames
                           StaticProtectedNs("flash.display:MovieClip"), StaticProtectedNs("flash
                           StaticProtectedNs("flash.display:InteractiveObject"), StaticProtectedN
                           StaticProtectedNs("Object"), PackageNamespace("com.kappaLab.asou
                           PackageNamespace(""), PackageInternalNs(""), PrivateNamespace("Fil
00003158    swap
00003159    getlocal       48
00003160    call           1
00003161    pop
00003162    pushint        444776
00003163    setlocal       37
```

```
[34/6070] - ByteArray:'addChild;stage;removeEventListener;loadBytes;addEventLis
[35/6070] - String:'addChild;stage;removeEventListener;loadBytes;addEventListen
[118/6070] - builtin.as$0::MethodClosure:loadBytes:''
```

Class	Type	Data
'builtin.as$0::MethodClosure:loadBytes'	Undefined	''
▸ 'flash.display::Loader'	Object	'[]'
'flash.utils::ByteArray'	ByteArray	'CWS\x12\'\xe0\x00\x00x\xda\xb4\xbc\x07|\x13G\...

```
00000000    43 57 53 12 27 e0 00 00 78 da b4 bc 07 7c 13 47   CWS.'...x....|.G
00000010    f0 28 7c 2b 9d 74 92 5c b0 e4 8a bb f1 e1 6e dc   .(|+.t........n.
00000020    0b dd 80 6d c0 14 d3 bb 8b 24 4b d8 60 6c 63 cb   ...m.....$K.`lc.
00000030    18 12 08 25 a1 85 d0 8c 31 21 09 a1 d7 50 43 08   ...%....1!...PC.
00000040    09 29 c8 45 52 2a 51 71 81 90 04 48 25 85 f4 46   .).ER*Qq...H%..F
00000050    12 f0 9b d9 93 4d 49 de ff bd df ef 7d 9f 41 ab   .....MI.....}.A.
00000060    bb 9b dd d9 d9 d9 69 3b bb a7 c5 0c fb 29 c3 b8   ......i;.....)..
00000070    1f 62 18 3f c2 64 2b 7b 33 0c d3 34 94 79 c7 5b   .b.?.d+{3..4.y.[
00000080    c6 e0 5f 86 66 c5 b6 2e ef 1d 70 c5 d1 7b f9 04   .._.f.....p..{..
00000090    af be 81 9e 33 7b 7b 8d 4c 9e ae 1a 94 b1 38 50   ....3{{.L.....8P
000000a0    59 3d 7b 61 ae fb 8c 7e e2 91 ac 8b 7b e2 e8 7e   Y={a...~....{..~
000000b0    4b e3 1e 8b 49 cd 4b e9 5d e8 39 9b f5 09 5a 90   K...I.K.].9...Z.
000000c0    93 a4 2a 58 3c bf 6a d8 20 37 1f 17 f9 fc fe 29   ..*X<.j. 7.....)
000000d0    a3 b8 f2 30 83 22 a2 ca ab 38 a2 6f 86 ae 60 d9   ...0."...8.o..`.
000000e0    78 0f cd 52 ff ac 49 13 b4 be 53 ea 06 4e 19 e8   x..R..I...S..N..
000000f0    b7 64 d9 e0 47 fc 7d 67 0e 70 8b 2d 57 0c f4 9a   .d..G.}g.p.-W...
00000100    b3 54 15 5d 40 52 fc 47 78 0d 29 8c 54 18 e2 c7   .T.]@R.Gx.).T...
```

Figure 2: *Replay graphical database frontend*

VII. Future development

Reflash is still very early in its development, and there are many things that can be improved. One of the culprits in the current implementation is the instrumentation of embedded flash content. Relying on *ExternalInterface* is somewhat fragile, because there is no guarantee that it will remain synchronous in any given situation. It appears to be reliable, but that can change with any Flash Player release.

There is no obvious solution to this problem, because essentially it requires to transform the asynchronous nature of AS3 to synchronous, for that particular case. If *ExternalInterface* suddenly stops working, one of the possible solutions could be including an instrumented code in the Instrument module. That would require a two-step approach. First detect the attempt to load embedded content, then instrument the content, rebuild a new Instrument module embedding the new content and finally restart the execution. That is somewhat problematic to fully automate and it cannot be done to polymorphic payloads.

Another possible solution to the embedded content problem could be using a method deviced by Timo Hirvonen with his tool *Sulo* [12]. Sulo is instrumenting the Flash Player outside the AS3 context, so it can see all calls to *loadBytes* no matter how many layers of embedding there is. On the other hand, it is still unknown if it is possible to feed back modified flash using the Sulo tracing approach.

Next target for development is the Instrument logging, which can never be efficient enough. Here also the Sulo approach could be interesting. Instead of sending the AMF-packed data over to a TCP connection, just fetch them off the AVM2 using Sulo.

There are also some interesting things that could be done with the Replay frontend. In the current implementation, there is no execution logic interception - it is just a dummy database frontend. It should be perfectly possible to implement an AVM2 emulation for stack data manipulation. In that scenario, only the instructions that require AVM2 runtime, such as class constructors and method calls, would be instrumented. The resulting stack values could be migrated to the emulation, thus creating a *hybrid*, or assisted emulator. Functionality that is not trivial to implement would be run on a real Flash Player, but all trivial operations would be emulated over the database. Example of such trivial functionality could be a simple ADD operation. It is not necessary to instrument that, thus saving precious AVM2 CPU cycles.

One interesting future development possibility is to replace Reflash stack data generation backend with *Mozilla Shumway* [13]. In that setup, Reflash instrumentation stack trace generation would be implemented directly to the Shumway runtime. Reflash executable's role would be reduced to provide only contextual information and metadata to the database, and Shumway would generate stack items. That should provide better performance and stability as there is no instrumentation involved. This approach sounds very promising in principle, but there would be a lot of compatibility issues with concurrent malware and exploit kits. They tend to require a very specific and realistic environments.

Finally, there is an interesting, albeit somewhat speculative analogy from native platforms instrumentation platforms, such as *Intel Pin* [14] . With that approach, Reflash would instrument entire method body instead of individual opcodes. There is already the pseudo opcode *method_entry* that could be extended for providing emulated instrumentation for the entire method body. Because AS3 code cannot access directly the opcode level, intermediate code could be provided forehand by the static instrumentation code. With this intermediate code, the method instrumentation code could emulate stack manipulation and proxy all opcodes requiring flash runtime, such as object manipulation on heap and all native methods. This could provide better perfomance and coverage.

VIII. Related work

The basic concept of dynamic instrumentation of flash files for malware analysis was introduced by Jeong Wook Oh in his 2012 presentation, *AVM Inception*. In the original presentation, a concept of *class hooking* was presented, but later he has also released a tool called *FlashHacker* [7] that uses the RABCDAsm [3] tools for instrumenting flash files, creating call traces and other manipulations. Reflash can be considered as a continuation of that work, extending and automating the approach.

F-Secure Sulo by Timo Hirvonen is also somewhat related to what Reflash is doing. Reflash's emphasis is in data postprocessing, and it could be beneficial to start using Sulo approach for data collection, instead of flash file instrumentation. The trade-off would be losing the portability of Reflash AS3 instrumentation, but there would be huge benefits in performance.

IX. Conlcusion

This paper shows that large-scale instrumentation of flash files is not only possible, but a practical solution for dynamic ActionScript analysis. The novel client/server-architecture used in the solution also opens up other possibilities for even more efficient stack trace collection in the future.

References

[1] TrendLabs Security Intelligence blog. How Exploit Kit Operators are Misusing Diffie-Hellman Key Exchange (2015).

[2] Adobe Systems inc. ActionScript Virtual Machine 2 (AVM2) Overview (2007). `https://www.adobe.com/content/dam/Adobe/en/devnet/actionscript/articles/avm2overview.pdf`

[3] Vladimir Panteleev. Robust ABC (ActionScript Bytecode) [Dis-]Assembler. `https://github.com/CyberShadow/RABCDAsm`

[4] Gary Grossman, Emmy Huang. ActionScript 3.0 overview (2006). `https://www.adobe.com/devnet/actionscript/articles/actionscript3_overview.html`

[5] Dan Bornstein. Dalvik VM Internals (2008). `https://sites.google.com/site/io/dalvik-vm-internals`

[6] The D programming language. `https://dlang.org/`

[7] Jeong Wook Oh. FlashHacker ActionScript Bytecode instrumentation framework. `https://github.com/ohjeongwook/FlashHacker`

[8] Adobe Systems inc. Action Message Format (2013). `http://wwwimages.adobe.com/www.adobe.com/content/dam/Adobe/en/devnet/amf/pdf/amf-file-format-spec.pdf`

[9] mitmproxy, the Man-In-The-Middle proxy. `https://mitmproxy.org/`

[10] YARA, The pattern matching swiss knife for malware researchers. `https://virustotal.github.io/yara/`

[11] IDA, a multi-processor disassembler and debugger. `https://www.hex-rays.com/products/ida/`

[12] Timo Hirvonen. Sulo: Dynamic instrumentation tool for Adobe Flash Player built on Intel Pin. `https://github.com/F-Secure/Sulo`

[13] Mozilla. Shumway: HTML5 technology experiment that explores building a faithful and efficient renderer for the SWF file format without native code assistance. `https://github.com/mozilla/shumway`

[14] Intel. Pin: A Dynamic Binary Instrumentation Tool `https://software.intel.com/en-us/articles/pintool`