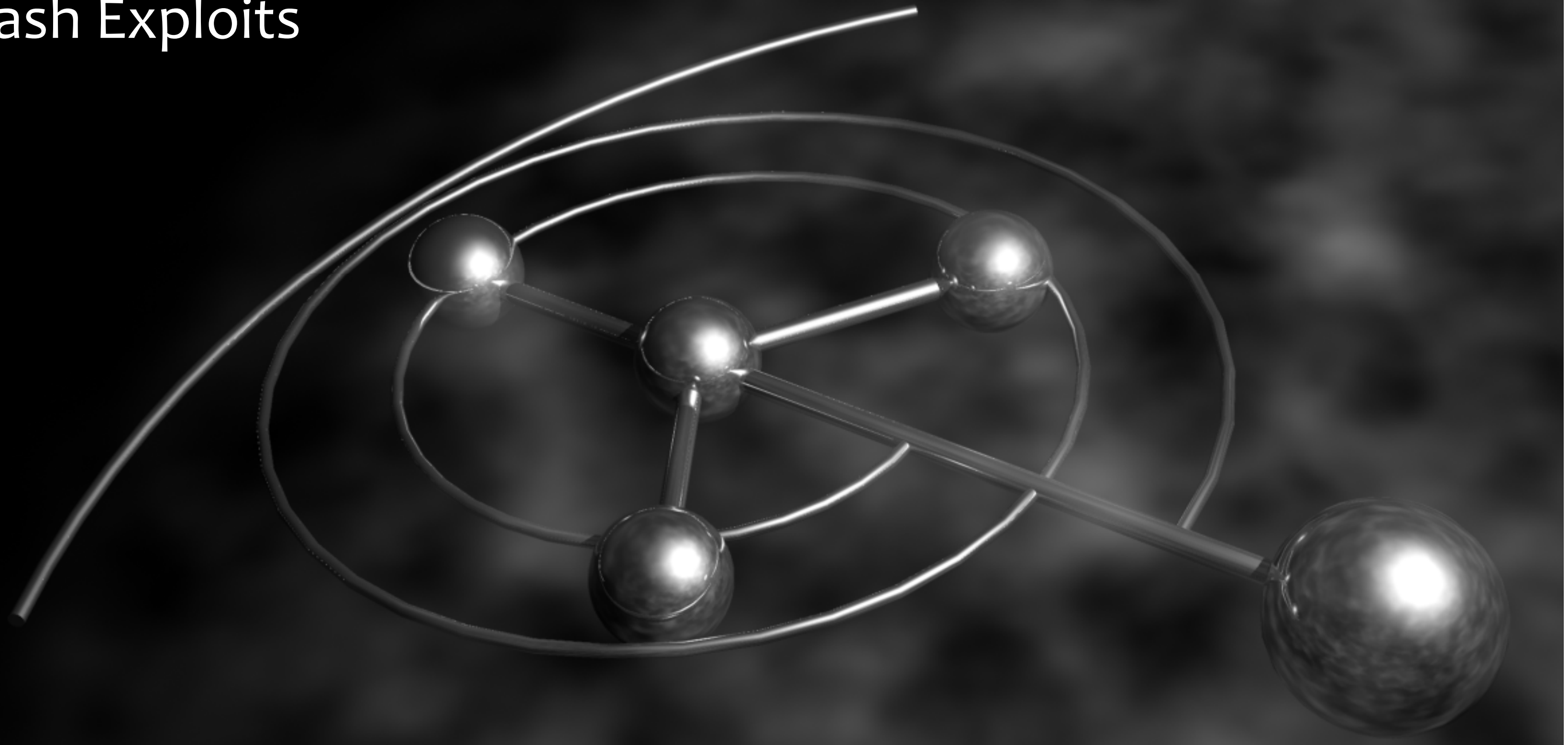
Blitzableiter – BETA Release

Countering Flash Exploits



Felix ‘FX’ Lindner
DEFCON XVIII

Agenda

- Motivation
- RIA Basics
- Flash (in)Security
- Flash Malware
- Flash Internals
- Defense approach
- Implementation
- Current functionality
- Measurements & Results
- Next steps

Motivation

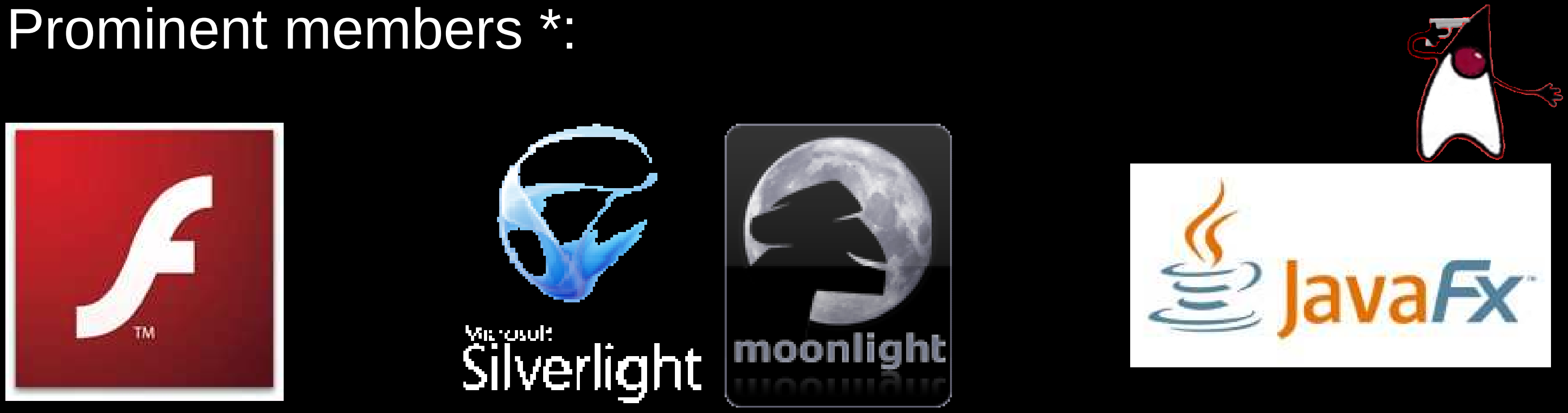
- Project initiated in late 2008 by the German Federal Office for Information Security (Bundesamt für Sicherheit in der Informationstechnik)
- Review of the current Rich Internet Application security situation
- Adobe Flash turned out to be far behind the curve in terms of security compared to other technologies
- That posed the question whether that fact could be helped
  - Preferably without firing everyone at Adobe

Who cares about Flash security?

- Some of the end users
  - Apple users running on PowerPC machines:
    The Adobe Flash Player 10.1 release, expected in the first half of 2010, will be the last version to support Macintosh PowerPC-based G3 computers. Adobe will be discontinuing support of PowerPC-based G3 computers and will no longer provide security updates after the Flash Player 10.1 release. This unavailability is due to performance enhancements that cannot be supported on the older PowerPC architecture.
  - People who don’t want to get owned while surfing pr0n
- Web site operators
  - Web sites that display advertisement banners (Heise or eWeek anyone?)
  - Owners of web sites allowing users to upload files

Rich Internet Applications

- Rich Internet Applications (RIA) are in general programmatic enhancements to the regular web browser that allow for enhanced interactivity, communication and media display.
- Prominent members *:



* Google Native Client is intentionally not mentioned here.

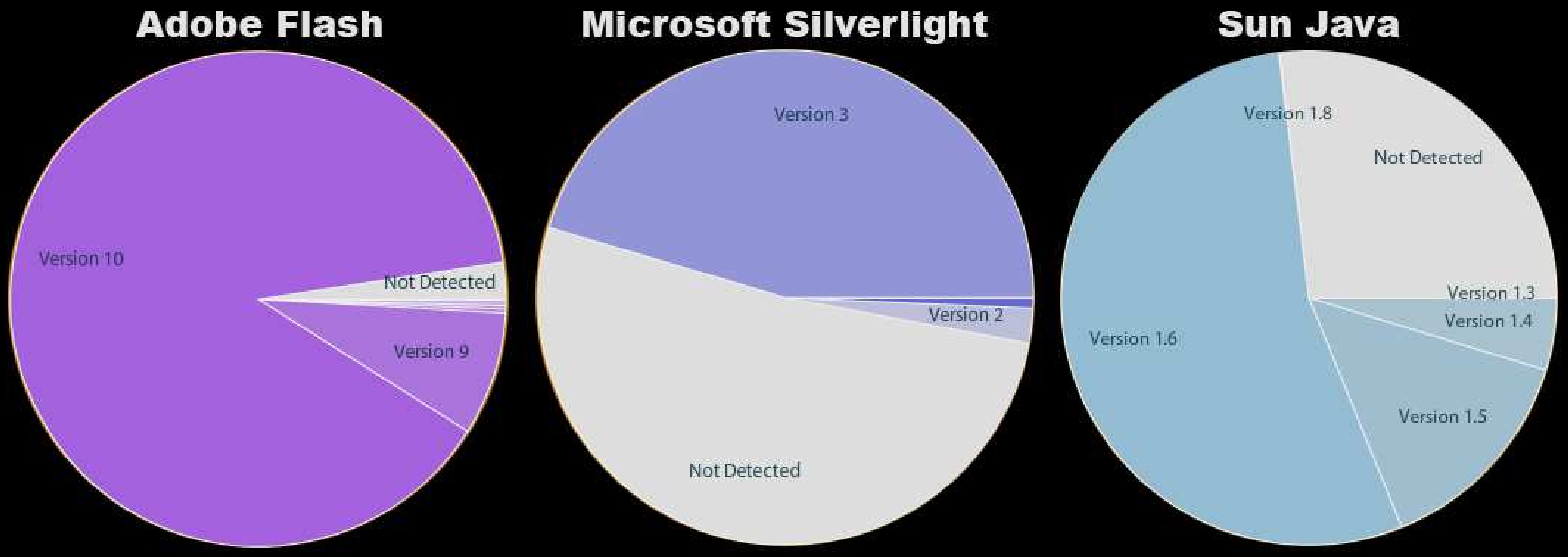
Common Properties of RIA Environments

- RIA functionality is implemented as plug-in for web browsers
  - Plug-in provides a runtime environment with one or more virtual machines that execute byte code specific to that RIA platform
  - RIA runtimes additionally provide media playback capabilities
  - Applications are distributed in integrated file formats, carrying byte code and resource files, such as image, video and audio data
  - RIA runtimes provide their applications independent local storage capabilities and browser independent network communication
  - RIA applications are portable between all platforms having a runtime

Web Browser Integration and Interaction

- Browser plug-in code base is usually fairly independent of the actual browser plug-in
  - The runtime is commonly an ActiveX component or an external program
  - Browser dependence only for JavaScript, DOM and HTTP stack
- Activation of RIA functionality in web pages through HTML tags
  - The embedding HTML often decides on security settings for the RIA
  - Interaction between RIA code (through the runtime) and the web browser is commonly achieved using JavaScript

Distribution of Runtimes



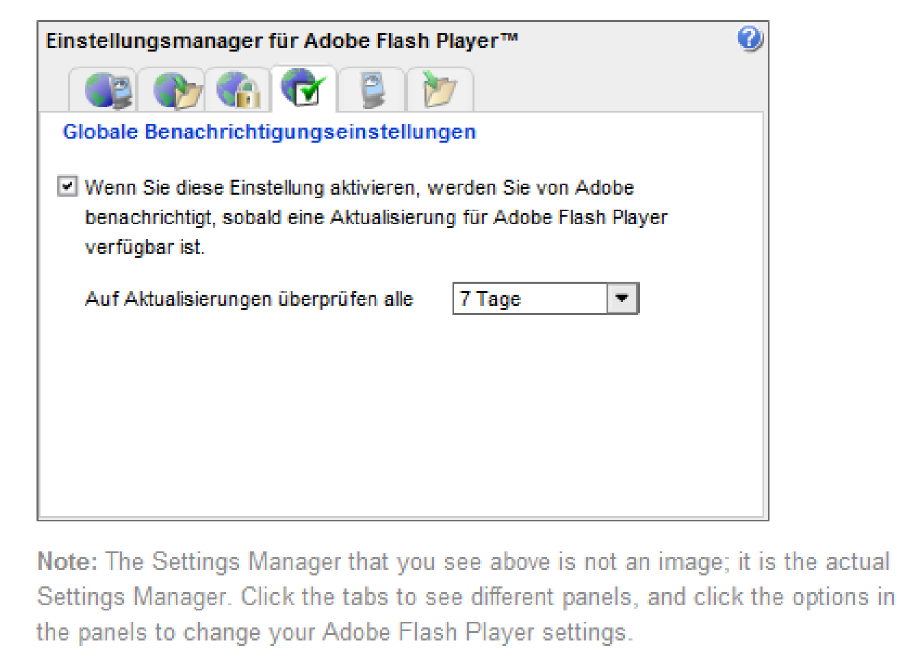
Source: RIAStats.com (18 million browsers, 110 sites, 30 days), 2009-12-24

The Flash Security Model

- Flash primarily relies on the virtual machine runtime environment for sealing off access from the RIA code to the native machine
- Permission decisions are based on so-called sandboxes
  - Generally, Flash code can either access local or remote resources, not both.
- The remote sandbox roughly follows the Same Origin Policy
  - Flash code can soften the Same Origin Policy itself
  - Flash code can offer a permissive Cross Domain Policy for “Flash Cookies”
  - Socket communication is controlled in a similar way using a policy file that is served from the remote TCP port the Flash code wants to connect to
- The embedding web page can give additional permissions to the Flash code using arguments to the OBJECT or EMBED tags

The Flash Security and the User

- There is some user control over the Flash Player
  - Using an actual Flash program
- Most controls are hidden as options in mms.cfg
- Flash does not support any proof of origin for the files
  - With the likely exception of DRM technologies, which are of no concern here



Einstellungsmanager für Adobe Flash Player™

Globale Benachrichtigungseinstellungen

☑ Wenn Sie diese Einstellung aktivieren, werden Sie von Adobe benachrichtigt, sobald eine Aktualisierung für Adobe Flash Player verfügbar ist.

Auf Aktualisierungen überprüfen alle: 7 Tage

Note: The Settings Manager that you see above is not an image; it is the actual Settings Manager. Click the tabs to see different panels, and click the options in the panels to change your Adobe Flash Player settings.

Flash Vulnerabilities

- Securityfocus.com lists about 40 vulnerabilities for the Flash Player, among them:
  - CVE-2008-3873: Copy
  - CVE-2007-3456: FLV i
  - CVE-2007-0071*: Integ arbitrary memory write
  - CVE-2009-3797 & CVE
  - CVE-2008-4546: SWF

```
package {
    import flash.display.*;
    import flash.net.*;

    public class A extends MovieClip {
        public function A() {
            load();
            load();
        }

        private function load():void {
            var loader:Loader = new Loader();
            loader.load(new URLRequest('/b'));
            addChild(loader);
        }
    }
}
```

* “Application-Specific Attacks: Leveragin

Attacks using Flash

- Using Flash to perform DNS rebinding, as shown by Dan Kaminsky
- Using the extensive operating system and browser information available to Flash code to determine exploits to use
  - Nowadays commonly used to carry PDF files
- Clickjacking aka. User Interface Redressing
- Sending additional HTTP Header in requests originating from Flash code
  - UPNP Requests through Flash to reconfigure home routers
  - CSRF exploits for the masses
- Appending HTML/JavaScript to Flash files
- Simply redirecting the web browser

Flash Malware

- Malware based on Flash is generally part of the following classes:

1. **Redirector and Downloader**
   The user browser is redirected to a new URL, either faking clicks for advertisement programs or simply downloading executable files.

2. **Binary Exploits**
   With the most prominent being CVE-2007-0071, these Flash files attempt to exploit parsing vulnerabilities in the Flash runtime.

3. **Web Attack Vehicle**
   Flash is commonly used in attacks leveraging XSS or weak coding in other Flash files in order to obtain personal identifiable information (PII) from the target.

Flash Malware Examples

- SWF.AdJack / Gnida
  - Malicious banner advertisement, which uses Local Shared Objects (LSO) to store campaign information on the user’s machine.
- CVE-2007-0071 Exploit
  - Various incarnations using the original published exploit technique to achieve arbitrary code execution. Your payload will vary.
- SWF/TrojanDownloader.Agent etc.
  - Simple browser forwarder
- SWF/TrojanDownloader.Agent.NAD
  - Multi-Exploit carrying a number of different attack codes and instantiating them depending on the operating system platform and Flash player version

Flash Malware and the Anti-Virus Industry

- Flash malware is not very well detected by anti-virus software
- AV software epically fails when the malware is uncompressed

Sample	Detection	Detection (uncompressed)
Simple generic downloader	**18/41 (43.91%)**	**16/39 (41.03%)**
Gnida.A	**29/41 (70.73%)**	**8/40 (20%)**
SWF_TrojanDownloader.Small.DJ	**21/39 (53.85%)**	**11/41 (26.83%)**

Statistics generated using Virustotal.com on December 24, 2009

Flash Files from the Inside

- Flash files (also called movies) follow the SWF (apparently pronounced “swiff”) file format specification
  - Version 3 to Version 10 are specified
- SWF files can be compressed using zlib methods
- Type-Length-Value structure
  - The elements are called “Tags”
  - The element ordering determines (partially) the rendering
  - 63 Tag types are documented for Version 10
- Data structures are heavily version dependent

Adobe Virtual Machines

- The Flash Player contains ***two*** virtual machines
- AVM1 is a historically grown, weakly typed stack machine with support for object oriented code
  - AVM1 is programmed in ActionScript 1 or ActionScript 2
  - Something between 80%-90% of the Flash files out there are AVM1 code, including YouTube, YouPorn, etc.
- AVM2 is an ECMA-262 (JavaScript) stack machine with a couple of modifications to increase strangeness
  - AVM2 is programmed in ActionScript 3
  - The Flash developer community struggles to understand OOP

The History of AVM1

- First scripting capability appears in SWF Version 3
  - Something like a very simple click event handler
- SWF Version 4 introduces the AVM
  - Turing complete stack machine with variables, branches and sub-routine calls
  - All values on the stack are strings, conversion happens as needed
- SWF 5 introduces typed variables on the stack
  - Addition of a constant pool to allow fast value access
  - Introduction of objects with methods

The History of AVM1

- SWF 6 fixes SWF 5
    - New Tag type allows initialization code to be executed early
    - Checking of the type of an object instance is added
    - Type strict comparisons are added
- SWF 7 brings more OOP
    - New function definition byte code
    - Object Inheritance, extension and test for extension (implements)
    - Exception generation and handling (Try/Catch/Finally)
    - Explicit type casting

The History of AVM1

- SWF 8 never happened
- SWF 9 already brings the AVM2 into the format
  - They call the byte code “ABC”
- SWF 10 is the currently specified standard

Keep in mind that all this is still supported!

AVM1 Code Locations in a Flash File

- A Flash file can contain AVM1 code in 5 different types of locations
  - DoAction Tag contains straight AVM1 code
  - DoInitAction Tag contains AVM1 code for initialization
  - DefineButton2 Tag contains ButtonRecord2 structure that can carry conditional ButtonCondActions, which are AVM1 code
  - PlaceObject2 and PlaceObject3 Tags can contain ClipActions whose ClipActionRecords may contain AVM1 code
- Many tools, including security tools, only handle DoAction

AVM1 Code Properties

- AVM1 byte code is a variable length instruction set
  - 1-Byte instructions
  - n-Byte instructions with 16 Bit length field
- Branch targets are signed 16 Bit byte offsets into the current code block
- Function declarations are performed using one of two byte codes inline with the other code
  - Function declarations can be nested
  - Functions may be executed inline or when called
- Try/Catch/Finally blocks are defined by byte code similar to functions

Design Weaknesses in AVM1

- The byte offset in branch instructions allows:
  - Jumps into the middle of other instructions
  - Jumps outside of the code block (e.g. into image data)
- The signed 16 Bit branch offset prevents large basic blocks
  - The Adobe Flash Compiler emits illegal code for large IF statements
- Instruction length field allows hiding of additional data
  - Length field is parsed even for instructions with defined argument sizes
- Argument arrays contain their own length fields after the instruction length field

Design Weaknesses in AVM1

- The order of code execution appears to be non-deterministic
  - Depends on the Tag order and type
  - Depends on references to other Flash files
  - Depends on the conditions set to execute
  - Depends on the visibility of the object (z-axis depth)

Considerations for the Defense Approach

- There are two types of attacks to be handled
    - Malformed SWF Files that cause memory corruption in the player
    - Well formed SWF Files that use the player’s API for evilness
- Instrumentation of the player is bound to fail
    - The player is closed source and changes permanently
    - The player is very forgiving with invalid / malformed code
    - The player only parses code and data when it hits it
    - The player is written in an unmanaged language
- Nobody wants to write a new Flash player from the ground up

Normalization through Recreation

If the final consumer is fragile, we must try to ensure that it will not choke on the data passed to it:

1. Safely parsing the complete SWF file, strictly checking specification compliance of everything
2. Discarding of the original file
3. Verification and modification of the AVM code
4. Creation of a new “normalized” SWF file

Introducing the Tool

- The aforementioned and now further discussed approach is implemented in the project called:

Blitzableiter

- Yes, that’s a German term. It means “lightning rod”. It turns dangerous lightning into harmless flashes.

A Flash File Parser in C#

- The CLR ensures buffer boundaries and prevents integer overflow / signdness issues
- Native or unsafe code must not be used
  - This can be checked easily
- Strategic defense advantage: .NET CLR 0-day is rare, expensive and unlikely to be used against a Flash parser
- By strictly targeting .NET 2.0, it runs nicely on Mono as well

Enforcing Container Boundaries

- Parsing container type data structures (e.g. a Tag) must ensure they are completely used
  - Trailing data could be shell code
  - Exceeding data could be the actual attack
- Place the container’s content into a new memory block
  - CLR ensures boundaries
  - Post parsing code ensures complete consumption

Only Parse Documented Data

- For the approach to work, all data must be parsed completely
  - Simply copying byte arrays from the original file is dangerous, as it may contain the attack
- All input data must be checked thoroughly against the specification
  - Are the correct format and fields for the declared version of the SWF file used?
  - Are all reserved Bits clear?
  - Are conditions declared for objects correct and make sense?
  - Are all type fields using documented values?

AVM1 Code Verification

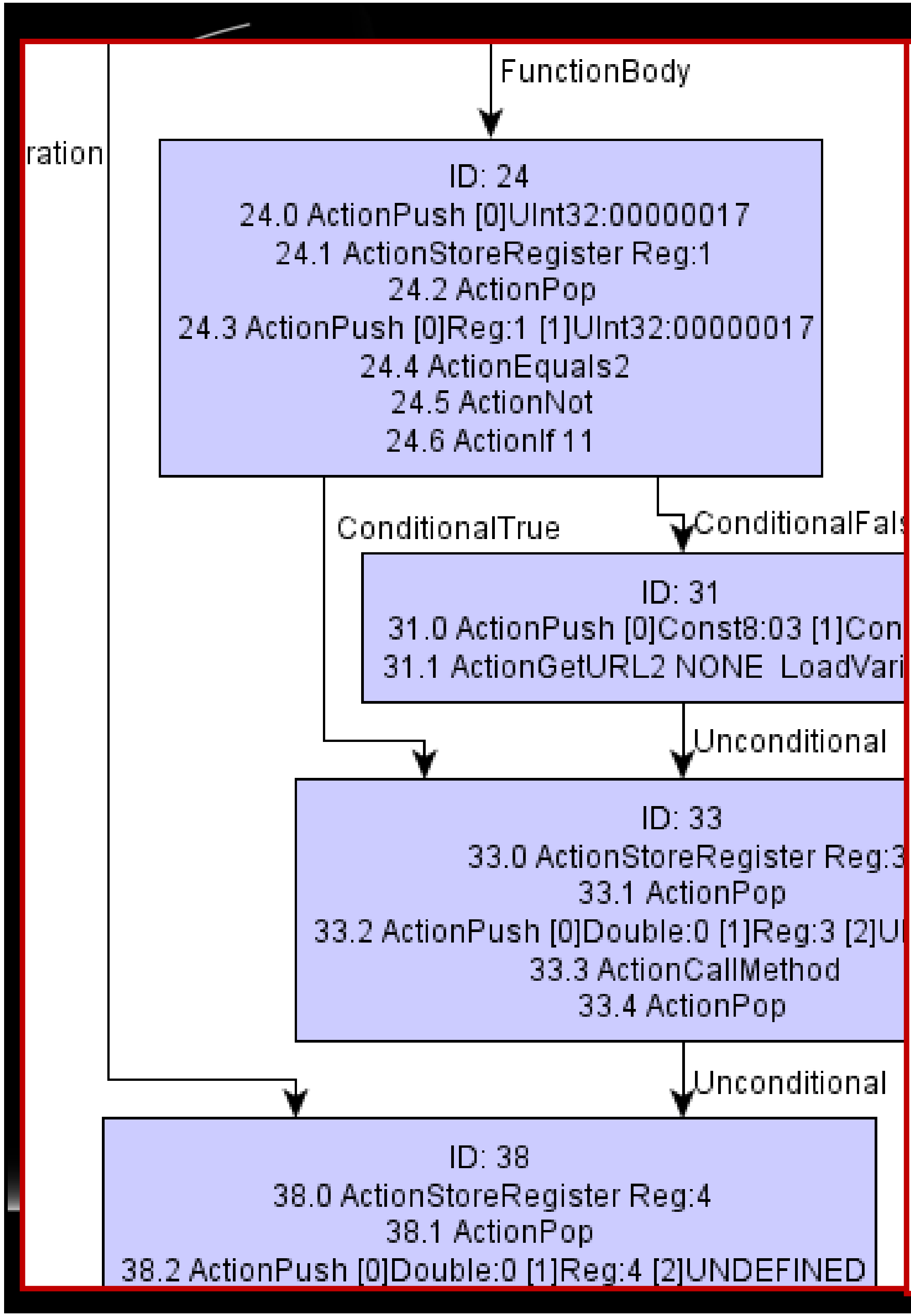
- Is the instruction legal within the declared SWF Version?
- Does the instruction have exactly the number of arguments specified?
- Is the declared instruction length correct and completely used?
- Does the code flow remain within the code block?
- Do all branches, try/catch/finally and all function declaration target addresses point to the beginning of an instruction?
    - This is ensured using linear disassembly instead of code flow disassembly
- Do all instructions belong to exactly one function?

Countering Functional Attacks

- If done correctly and completely, the approach so far leaves you with a representation of a nice and tidy SWF file that you completely understand.
- Static analysis will provably not be able to determine what any given code is actually doing.
- Emulation will cause a state discrepancy between your emulation and the Flash player's interpretation of the same code.

Patching the Point of Execution

- In runtime analysis, you verify the arguments to the final API call before the call is made.
- We are not part of the show when execution actually happens.
- But we can introduce AVM1 code before the final API call that inspects and verifies the arguments for us when executed.

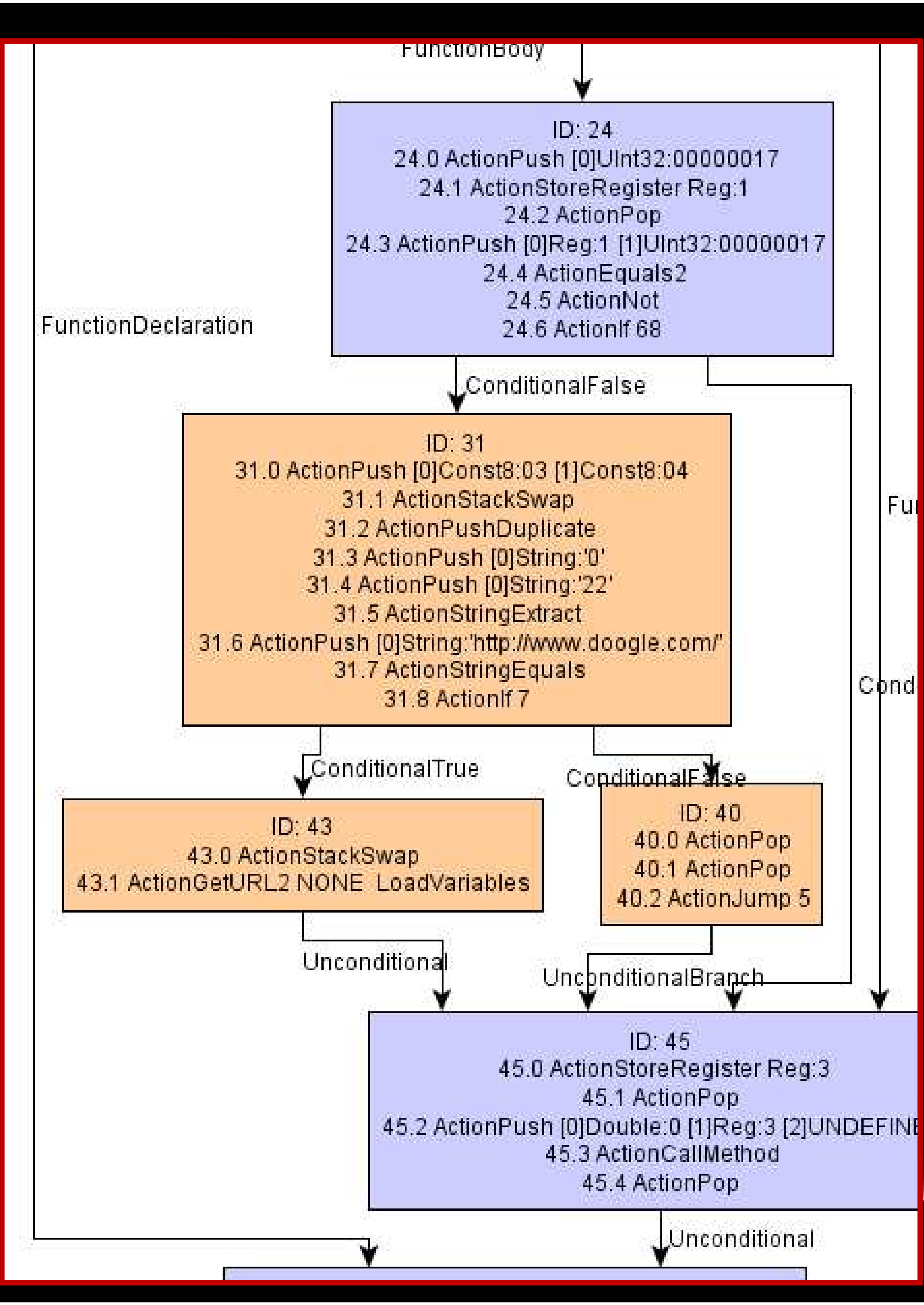


ID: 24
24.0 ActionPush [0]UInt32:00000017
24.1 ActionStoreRegister Reg:1
24.2 ActionPop
24.3 ActionPush [0]Reg:1 [1]UInt32:00000017
24.4 ActionEquals2
24.5 ActionNot
24.6 ActionIf 11

ID: 31
31.0 ActionPush [0]Const8:03 [1]Con
31.1 ActionGetURL2 NONE LoadVari

ID: 33
33.0 ActionStoreRegister Reg:3
33.1 ActionPop
33.2 ActionPush [0]Double:0 [1]Reg:3 [2]U
33.3 ActionCallMethod
33.4 ActionPop

ID: 38
38.0 ActionStoreRegister Reg:4
38.1 ActionPop
38.2 ActionPush [0]Double:0 [1]Reg:4 [2]UNDEFINED



ID: 24
24.0 ActionPush [0]UInt32:00000017
24.1 ActionStoreRegister Reg:1
24.2 ActionPop
24.3 ActionPush [0]Reg:1 [1]UInt32:00000017
24.4 ActionEquals2
24.5 ActionNot
24.6 ActionIf 68

ID: 31
31.0 ActionPush [0]Const8:03 [1]Const8:04
31.1 ActionStackSwap
31.2 ActionPushDuplicate
31.3 ActionPush [0]String:'0'
31.4 ActionPush [0]String:'22'
31.5 ActionStringExtract
31.6 ActionPush [0]String:'http://www.doogle.com/'
31.7 ActionStringEquals
31.8 ActionIf 7

ID: 43
43.0 ActionStackSwap
43.1 ActionGetURL2 NONE LoadVariables

ID: 40
40.0 ActionPop
40.1 ActionPop
40.2 ActionJump 5

ID: 45
45.0 ActionStoreRegister Reg:3
45.1 ActionPop
45.2 ActionPush [0]Double:0 [1]Reg:3 [2]UNDEFINE
45.3 ActionCallMethod
45.4 ActionPop

What Should be Patched?

- According to Adobe*, this is:
  - Functions and objects that accept URLs as strings
  - Functions that display or accept HTML
    - Which in fact are fields of the respective objects, but never mind.
  - Functions that communicate with the web browser
  - Functions for accessing FlashVars
  - Functions for accessing shared objects
  - Functions that make networking calls
- Additionally, we need to patch any undocumented voodoo, e.g. calls to ASNative

* http://www.adobe.com/devnet/flashplayer/articles/secure_swf_apps_13.html

How To Do Patching

- Blitzableiter features a full, if somewhat spartanian, AVM1 assembler, so patches can be written in text files
  - Supports all 100 documented AVM1 instructions
- Support for variable names to allow the patch to interface with the verification code
  - Variables are used to store case-by-case information, e.g. the origin URL of the Flash file
  - Variable names can be randomized in order to prevent the surrounding code from checking them
- This is the low level view of the patching

Determining What Method Is Called

- Method calls are implemented in AVM1 as a sequence of:

```
ActionConstantPool 0:'receiving_lc' [...] 8:'connect'
ActionPush [0]Const8:07 [1]UInt32:00000001 [2]Const8:00
ActionGetVariable
ActionPush [0]Const8:08
ActionCallMethod
```

- Therefore, we need to check if we are dealing with an instance of the object first and then determine the method:

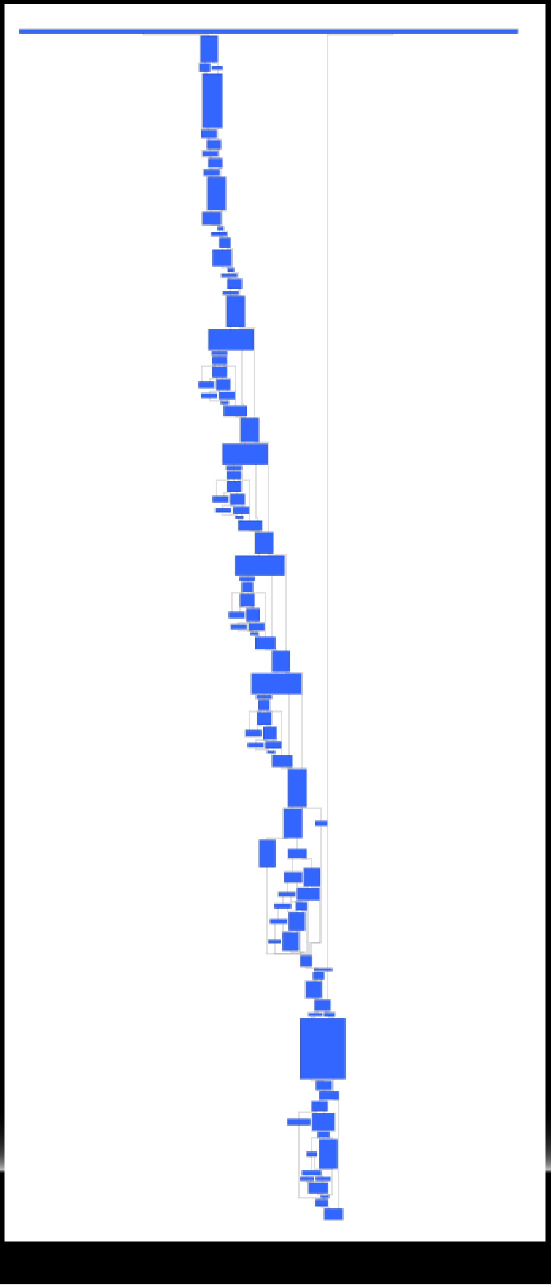
```
ActionStackSwap
ActionPushDuplicate
ActionPush String:OBJECTTYPE
ActionGetVariable
ActionInstanceOf
ActionNot
ActionIf ExitPatch:
ActionStackSwap
ActionPushDuplicate
ActionPush String:connect
ActionStringEquals
ActionIf CleanUp:
```

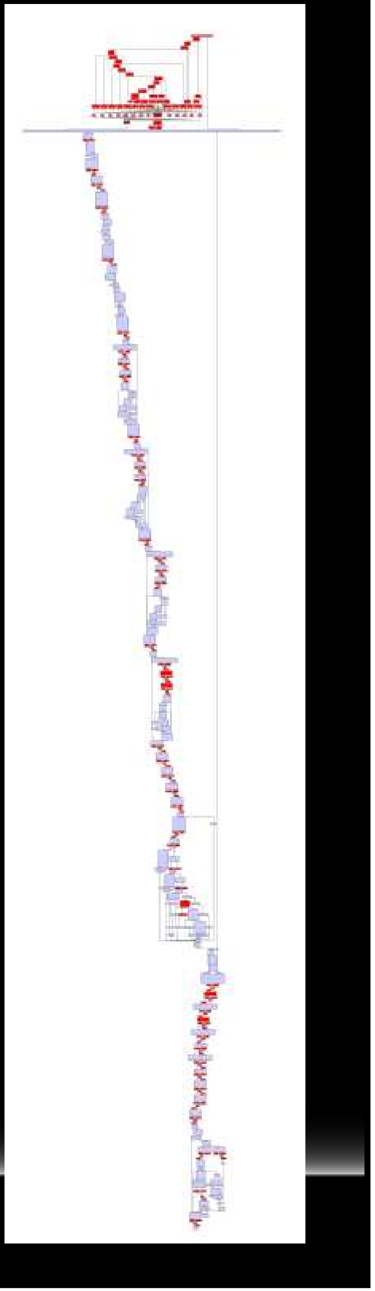
Cleaning Up In Case We Don’t Want Something

```
ActionPop                         # Remove method name
ActionPop                         # Remove object reference
ActionPush String:$RANDOM         # Create a variable with a random name
ActionStackSwap                   # Swap variable name and number of arguments
ActionSetVariable                 # Store number of arguments
RemovalLoop:
ActionPush String:$RANDOM         # Push random variable name
ActionPushDuplicate               # Duplicate
ActionGetVariable                 # Get number of arguments
ActionPush UInt32:0               # Push 0
ActionEquals2                     # Compare
ActionIf RemovalLoopDone:         # If number of arguments == 0, we are done
ActionPushDuplicate               # Duplicate random variable name again
ActionGetVariable                 # Get number of arguments
ActionDecrement                   # Decrement it
ActionSetVariable                 # Store in random variable name
ActionPop                         # Now remove one of the original arguments
ActionJump RemovalLoop:           # Repeat
ActionPop                         # Remove remaining string
ActionPush UNDEFINED              # Return UNDEFINED to the code that called
                                  # the method
```

Example: Gnida

- Adding a function to the top of the code sequence in order to perform all the object and method checks in one place
- Patching all ActionCallMethod places to verify the call using our check function





Feeding Malware to Blitzableiter

- Out of 20 real malware samples:
  - All AVM2 based files were rejected, as we don’t support that yet
  - All exploits targeting the player where rejected for format violations
  - All attacks using obfuscation were rejected for code violations
  - All exploits targeting the browser were patched to harmlessness
- So far, nothing survived Blitzableiter

Collateral Damage

- SWF obfuscation software is used to prevent decompilation
  - Ambiera irrFuscator, http://www.ambiera.com/irrfuscator/
  - Dcomsoft SWF Protector, http://www.dcomsoft.com/
  - Kaiyu Flash Encryption Genius, http://www.kaiyusoftware.com/products.html
  - Swfshield, http://swfshield.com/
  - Amayeta SWF Encrypt, http://www.amayeta.com/software/swfencrypt/
  - Kindisoft secureSWF, http://www.kindisoft.com/
- Except for irrFuscator, all obfuscation tools produce invalid SWF files, which were accordingly rejected during tests
  - Another good argument why obfuscation is zarking bullocks

Feeding Blinking Things to Blitzableiter

Item	Result
Sum of test cases	95780
Total size of all test case	45,47 GB
Sum of all Tags	120111586
Total number of test cases passing parsing and validation	88080
Percentage of test cases passing parsing and validation	92%
Number of AVM1 instructions before modification	472.698.973
Number of AVM1 instructions after modification	1.155.737.360
Average code size increase per file	224%
Total number of successfully patched files	82214
Percentage of test cases successfully patched	82%
Total number of unhandled exceptions during processing	82

Performance (on a fairly decent machine)

- Average read and validation time:
  0,447932 seconds over 96408 Files
- Average patch and write time:
  0,450058 seconds over 82203 Files
- Generating Code Flow Graphs (CFG) for all code takes significantly more time
  - Not fit for smaller machines (Blitzableiter within Web browser)
  - Graph operations on the CFG are computationally expensive

The Obvious and the Less Obvious Problem

- Patching every instance of a method call inflates the code significantly but still ignores the arguments to said call
  - This is a major problem for in-line patching
- AVM1 does not have any long branch instruction
  - 15 Bit is all we get: 32768 byte distance is the maximum
- Some Flash files already max out the branch distance
  - Patching such files results in an integer overflow (which we catch, thanks to the .NET CLR)

Extrospection and Code Flow Analysis

- To overcome the branch target overflow issue, we need to introduce “jump pads” into the code
    - Determining the Code Flow Graph (CFG) for the AVM1 code block
    - Determining which branch instructions would overflow
    - Placing jump pads for them halfway into the code
    - Ensuring that this does not cause more overflows
- This isn’t algorithmically trivial as far as I can tell
    - It also depends on how the Flash player likes jumping into and out of functions without them being actually called

Stack Tracing

- We can provably not determine all call arguments using static analysis
  - But we can determine calls and arguments that are loaded directly from the constant pool or static values on the stack
  - In order to determine values, we need to be able to track the stack state backwards
- A couple of approaches have been tried so far, but it’s not easy
  - Trivial to implement within the same basic block, but notion of basic block requires more expensive CFG generation
  - Even constant pools can be overwritten by AVM1 code. Therefore, the even constant pools are conditional and need CFG inspection.

Higher Level Verification Modeling

- The goal is to model:
  “Does the 2nd argument of this instruction begin with the following string?”
- The current implementation uses a dual stack machine approach
  - An internal stack machine performs individual static analysis operation steps to model conditions we want to verify
  - If the internal stack machine cannot deterministically continue, all basic operations emit AVM1 code to perform the same operation within the file.
- The individual operations are of small granularity
  - Example: ArgN determines the value of the n-th argument on the stack
  - Easier to verify the equivalence of the internal and the AVM1 representation

Things Blitzableiter Cannot Do Anything About

- Heap Spraying using Flash cannot be prevented using this approach
  - There is no obvious way to tell legitimate and malicious heap allocations apart
  - We could try the approach outlined by Microsoft’s Nozzle *
- Flash API overflows (as seen by the metric ton in PDF vs. Adobe Reader JavaScript API currently)
  - This would introduce checking for specific call arguments
  - We could consider a general call argument length limit

* http://research.microsoft.com/en-us/projects/nozzle/

The Current State

- This talk gives you the first official BETA
- We are covering 54 out of 63 Tag types so far
  - That’s 47 more tag types than the initial release at 26C3 supported!
- Parsing media data, as it is also often used for attacks
  - Currently using the .NET classes wherever possible
  - Need to find documentation on the Adobe proprietary things
- Support for AVM2 code
  - Really needed, already started, a whole new can of worms by itself

Why Blitzableiter is Open Source

- Apply Kerckhoffs' Principle to defense
  - No yellow box solution that magically protects you
- FX would like people to look over his zarking code and find bugs
- Flash developers shall to be able to test their stuff
- We want allow people to integrate Blitzableiter
  - Web browser extensions
  - Proxy server filter module
  - File upload filter module

Where To Get It From

- The project site is up and running at `http://blitzableiter.recurity.com`
  - Full source code for the class library under GPLv3
  - Developer documentation for quick starts
  - Test cases for malicious and non-malicious Flash
  - Public bug tracking system

Summary

- The longer we work on this, it becomes evident that:
  - Format validation is doable, practical and can prevent a lot of attacks
    - 3rd party tools in Flash world produce even less specification compliant files than any of the Adobe tools
  - Code validation is, not surprisingly, quite a bit more complex
    - It might only make sense to do this on server side (upload)
- FAQ: What about Adobe AIR, Flex, Flirr, Fluff, Fart, etc?
  - A: This is about Flash. Get the code and try it out yourself if you want.

Thank you!

fx@recurity-labs.com