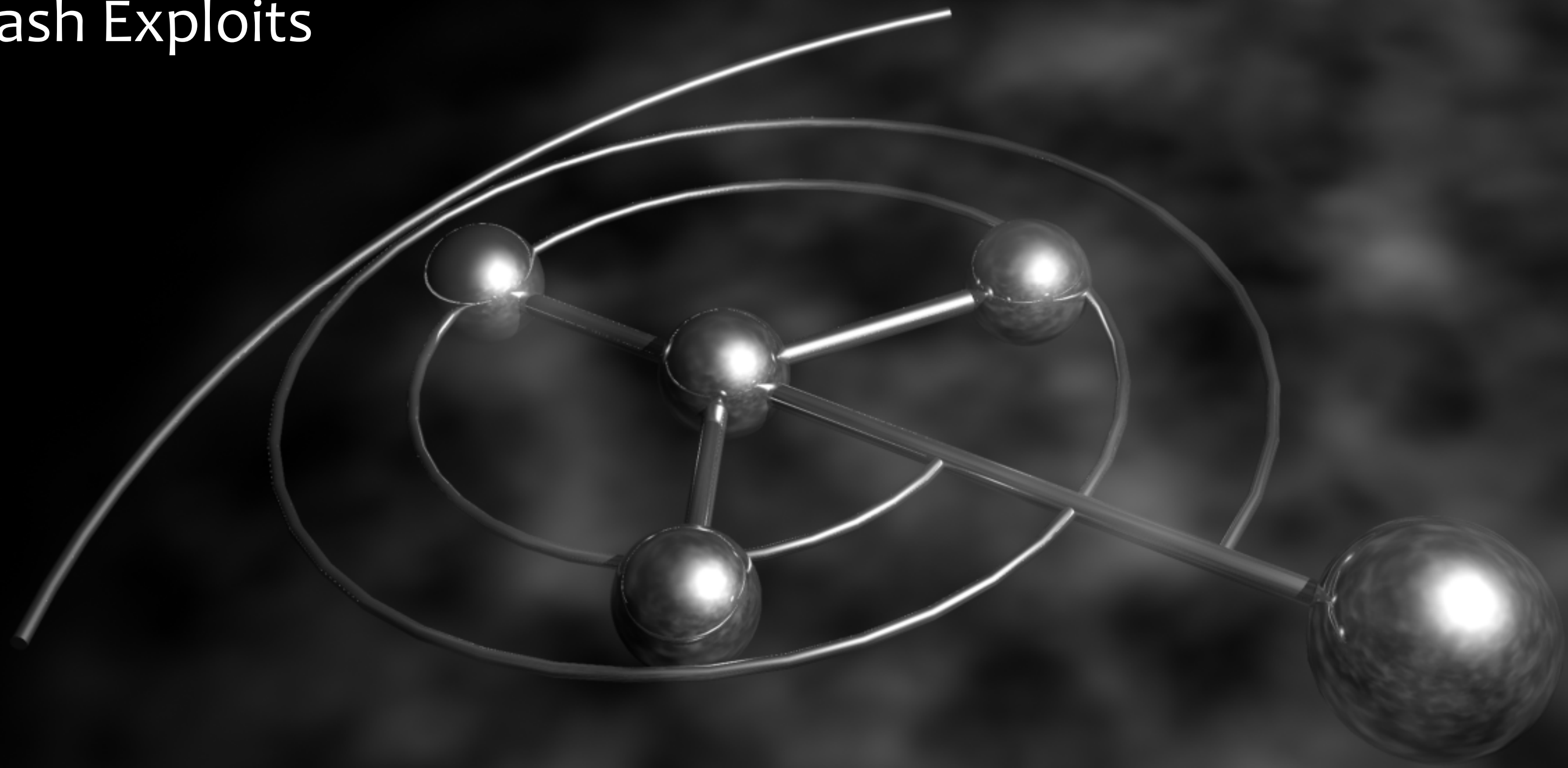
Blitzableiter – The Release (BETA)

Countering Flash Exploits



Felix 'FX' Lindner
BlackHat USA, July 2010

Agenda

- Motivation
- Flash Attack Surface
- Flash Victims
- Flash Security Options
- Introduction of Blitzableiter
- Flash Internals
- Blitzableiter Internals
- Adobe Virtual Machine 1
- AVM1 Code Analysis
- Enforcement of Functionality

Motivation

- Results from a project initiated in late 2008 by the German Federal Office for Information Security (Bundesamt für Sicherheit in der Informationstechnik) showed Adobe Flash to be the weakest Rich Internet Application technology
  - Adobe Flash runtime unfixable (at least for a third party)
  - Traditional detection mechanisms (AV/IDS) #fail
- The constant surfacing of new attacks against Flash requires a defense approach that doesn't depend on attack signatures
  - We didn't want to build yet another AV
  - The goal still is to be done with it at some point in time, once and for all.

Adobe Flash Attack Surface

- Flash files (SWF) is a container format for:
  - Vector graphics data (shapes, morphing, gradients)
  - Pixel graphics formats (various JPEG, lossless bitmaps)
  - Fonts and text
  - Sound data (ADPCM, MP3, Nellymoser, Speex)
  - Video data (H.263, Screen Video, Screen Video V2, On2 Truemotion VP6)
  - Virtual machine byte code for the Adobe Virtual Machines (AVM)
- All data structures from file format version 3 until the current version 10 are still supported
- The parser is completely written in unmanaged languages (C/C++)

Flash Victims I: ~~Pr0n~~ End Users

- End user's Flash Player can be triggered by any web page
  - Commonly exploiting parser vulnerabilities (e.g. CVE-2007-0071*, CVE-2010-2174), yielding direct code execution within the victim's browser process
  - DNS rebinding attacks
  - CSRF-style attacks including additional HTTP headers (e.g. UPNP)
  - Exploit toolkits with Flash frontend: Determining exact OS and browser versions, then downloading the appropriate exploit.
- 97% of all web browsers report Flash installed

* "Application-Specific Attacks: Leveraging the ActionScript Virtual Machine", Mark Dowd

End User Requirements

- The vast majority of exploits use intentionally malformed Flash files to trigger a vulnerability
- End users need a verification or enforcement mechanism to ensure Flash files are well-formed
  - Technically, a property the Flash Player must ensure, but that’s exactly where the problem is
  - Preferably integrated into web browser or proxy server
- End users require said mechanism to perform well, i.e. not taking too long or requiring too many resources

Flash Victims II: Web Site Owners & Ad Networks

- Advertisement Networks are forced to accept pre-compiled Flash content from Ad-Agencies as banner material
  - Submitted content is manually inspected (if at all)
  - No way to verify or enforce contractual requirements
  - Flash byte code sometimes changes behavior after the banner was accepted: It pulls trigger or additional code from remote server.
- Malicious advertisements have hit major news sites
  - NYTimes.com, Handelsblatt.de, Zeit.de, Heise.de, etc.

Web Site Owner & Ad Network Requirements

- Ensuring the Flash file is well-formed and does not carry an exploit is only partially sufficient for web site operators
  - It helps, however, to protect the review people from Flash exploits
- Desired is the ability to define rules mapping contractual requirements
  - E.g.: a banner advertisement can only forward the user's browser to the previously agreed campaign URL
  - E.g.: a social network site widget is not allowed to load additional content from a third party server
- Computational expense is of less concern, thoroughness is
  - Processing happens upon submission of the content, on the server side

Native Security Functionality of Adobe Flash

(this slide is intentionally left blank)

Native Security Functionality of Adobe Flash

- Very limited settings within the Flash Player configuration page, using an actual Flash file
  - Camera and microphone access, local storage limits, hardware video acceleration, “older security system”, DRM licenses
- Much more useful settings can only be made in mms.cfg, a local user specific configuration
  - AutoUpdateDisable, AllowUserLocalTrust, LocalFileLegacyAction, LegacyDomainMatching, ThirdPartyStorage, FileDownloadDisable, FileUploadDisable
- There is no proof of origin for Flash files (i.e. no digital signatures)

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 (20.00%)**
SWF_TrojanDownloader.Small.DJ	**21/39 (53.85%)**	**11/41 (26.83%)**

Statistics generated using Virustotal.com on December 24, 2009

Blitzableiter – An Alternative Defense Approach

- Straight command-line filter program
  - “Blitzableiter” is the German term for lightning rod, since it turns dangerous lightning into a harmless flash
  - Implemented in fully managed C#, targeting the .NET 2.0 runtime
  - Binary compatible with the Microsoft CLR as well as Mono 1.2
- Receives a potentially malicious Flash file (SWF) as input
  - Grossly malformed files are rejected
- Produces a (hopefully) non-malicious Flash file as output
  - Well-formed input files produce functionally equivalent output files

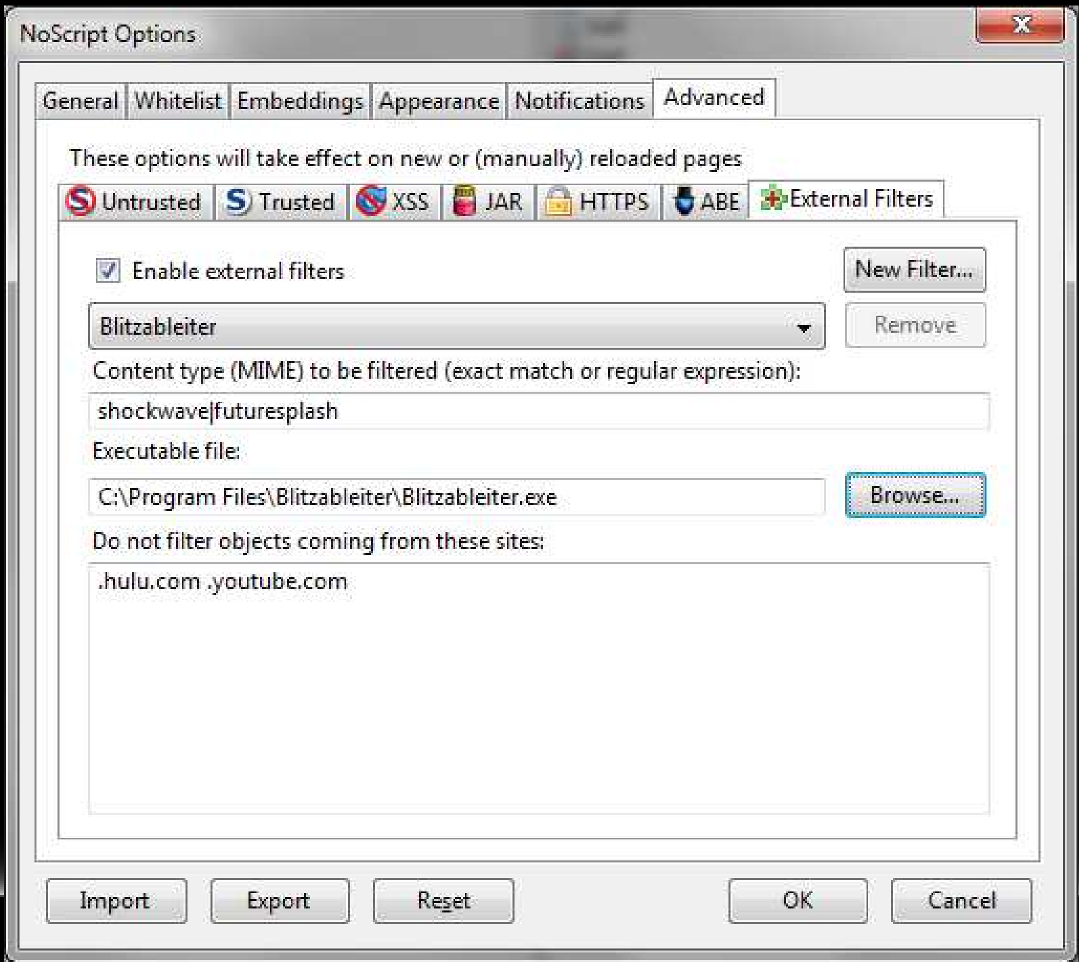
NoScript Supports Blitzableiter

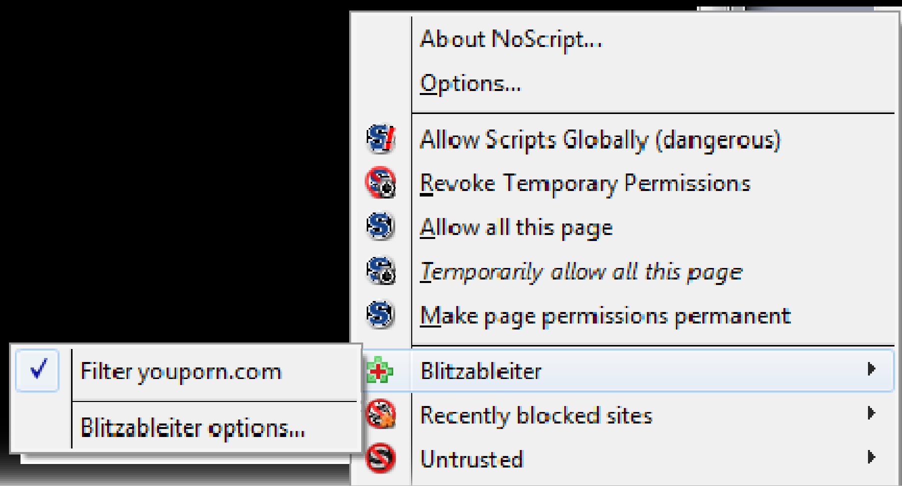
- Giorgio Maone introduced support for external filters in his popular NoScript add-on for Mozilla Firefox
  - MIME-Type based filtering using external programs
  - Required some serious design and code changes to allow for processing in background threads
  - Current versions (1.9.9.x and above) already support external filters, development versions (2.0rc2 and above) provide additional information to the filter (origins of page and content)
- We would like to thank Giorgio very much for his support!
  - His extraordinary willingness to cooperate, responsiveness, speed and quality of implementation should be an example for many others.

Recurity Labs

Introducing the Blitzableiter Security Tool

NoScript Supports Blitzableiter





Integrating Blitzableiter into Web Sites

- Web Site integration as post-processing step for upload functionality is trivial
  - Simply start Blitzableiter with the uploaded file as input
  - If OS return value is 0, move the output to the intended destination
  - If OS return value is < 0, present upload user with log output
- A Blitzableiter SOAP API is under consideration / construction

Blitzableiter is Open Source under GPLv3

- This project is open source, so you can apply something like Kerckhoffs' Principle and verify its protection value yourself
  - No yellow box solution that magically protects you
- We would love to see more integration in other software that must deal with Flash files
- Bug reports are also very welcome

http://blitzableiter.recurity.com

Demo Time

… what can possibly go wrong?

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

A few Example Tag Types

- Control Tags manage general aspects of the file
  - SetBackgroundColor, FrameLabel, Protect, End, EnableDebugger, EnableDebugger2, FileAttributes, Metadata, …
- Display List Tags define and show graphic elements
  - PlaceObject, PlaceObject2, PlaceObject3, RemoveObject, RemoveObject2, ShowFrame, …
- Bitmap Tags hold bitmap graphics data
  - DefineBits, DefineBitsJPEG2, DefineBitsJPEG3, DefineBitsLossless, …
- Buttons are special graphic objects that allow interaction (programming)
  - DefineButton, DefineButton2, DefineButtonCxform, DefineButtonSound

A Tag Data Structure Example

- Every Tag type has its own data structures, often deeply nested ones
- Many data structures are composed of lists of sub-structures, great places for integer overflows and signedness issues
- The Tag to the right is what caused CVE-2007-0071 by using a negative SceneCount and a missing allocation return value check in Flash Player

DefineSceneAndFrameLabelData

Field	Type	Comment
Header	RECORDHEADER	Tag type = 86
SceneCount	EncodedU32	Number of scenes
Offset1	EncodedU32	Frame offset for scene 1
Name1	STRING	Name of scene 1
...	...	...
OffsetN	EncodedU32	Frame offset for scene N
NameN	STRING	Name of scene N
FrameLabelCount	EncodedU32	Number of frame labels
FrameNum1	EncodedU32	Frame number of frame label #1 (zero-based, global to symbol)
FrameLabel1	STRING	Frame label string of frame label #1
...	...	...
FrameNumN	EncodedU32	Frame number of frame label #N (zero-based, global to symbol)
FrameLabelN	STRING	Frame label string of frame label #N

Preventing Format Based Exploits: Normalization through Recreation

1. Safely parse the complete SWF file
   - Strictly verify all data structures against their specified properties
2. Discard the original file
3. Verify inter-Tag consistency and AVM byte code
   - Potentially adjust the AVM byte code
4. Create a new, “normalized” SWF file for the final consumer (e.g. the Flash Player)

Implementation Details

- The Blitzableiter parser is completely managed code
  - Out-of-bounds conditions and integer overflows are caught by the runtime and cause an exception to be raised
- All TLV-style data structures are handled in individual memory streams, thus only offering as much data as declared in the TLV header
  - Trailing data is therefore discarded before parsing
  - Parser modules ensure that all content of the TLV container is used
- The parser only accepts well-documented SWF data structures
  - To provide the desired security level, this approach requires to parse every known data structure within the SWF specification
- The parser also verifies version dependencies of data structures

Recurity Labs

Blitzableiter Internals

Example: Catching CVE-2007-0071

```
protected override void Parse()
{
    log4net.ILog log = log4net.LogManager.GetLogger(System.Reflection.MethodBase.GetCurrentMethod()
    log.DebugFormat("0x{0:X08}: reading DefineSceneAndFrameLabelData-Tag", this.Tag.OffsetData);

    BinaryReader br = new BinaryReader(_dataStream);
    _sceneCount = SwfEncodedU32.SwfReadEncodedU32(br);
    _sceneOffset = new ulong[_sceneCount];
    _sceneName = new string[_sceneCount];

    for (ulong i = 0; i < _sceneCount; i++
    {
        _sceneOffset[i] = SwfEncodedU32.Sw
        _sceneName[i] = SwfStrings.SwfStri
        log.DebugFormat("0x{0:X08}:\tScene
    }

    _frameNum = SwfEncodedU32.SwfReadEncode
    _frames = new ulong[_frameNum];
    _frameLabel = new string[_frameNum];

    for (ulong i = 0; i < _frameNum; i++)
    {
        _frames[i] = SwfEncodedU32.SwfReadE
        _frameLabel[i] = SwfStrings.SwfStr
        log.DebugFormat("0x{0:X08}:\tFrame
    }
```

OverflowException occurred

Arithmetic operation resulted in an overflow.

Troubleshooting tips:

Make sure you are not dividing by zero.

Get general help for this exception.

Search for more Help Online...

Actions:

View Detail...

Enable editing

Copy exception detail to the clipboard

```
reading rejected: Tag handler failed parsing: System.OverflowException
```

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 around 80% 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)

AVM1 Code Verification performed by Blitzableiter

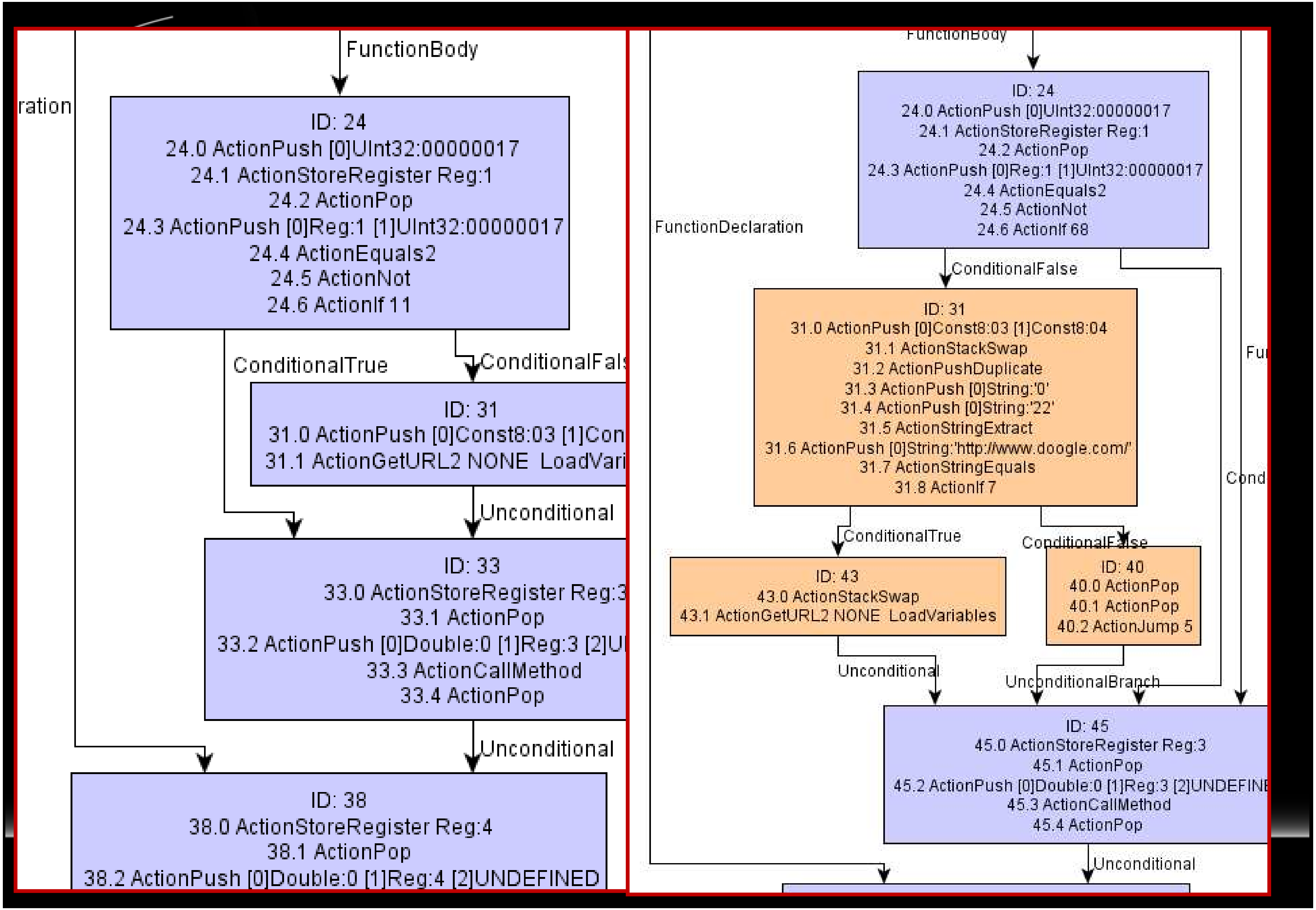
- Is the instruction legal within the declared SWF Version?
- Does the instruction have exactly the number of arguments specified?
- Is the declared instruction length correct and completely used?
- Does the code flow remain within the code block?
- Do all branches, try/catch/finally and all function 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 one and only one function?

Countering Functional Attacks

- If done correctly and completely, the format normalization 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.



FunctionBody
ration
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
ConditionalTrue
ConditionalFals
ID: 31
31.0 ActionPush [0]Const8:03 [1]Con
31.1 ActionGetURL2 NONE LoadVari
Unconditional
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
Unconditional
ID: 38
38.0 ActionStoreRegister Reg:4
38.1 ActionPop
38.2 ActionPush [0]Double:0 [1]Reg:4 [2]UNDEFINED
FunctionBody
FunctionDeclaration
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
ConditionalFalse
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
Fu
Cond
ConditionalTrue
ConditionalFalse
ID: 43
43.0 ActionStackSwap
43.1 ActionGetURL2 NONE LoadVariables
ID: 40
40.0 ActionPop
40.1 ActionPop
40.2 ActionJump 5
Unconditional
UnconditionalBranch
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
Unconditional

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:
```

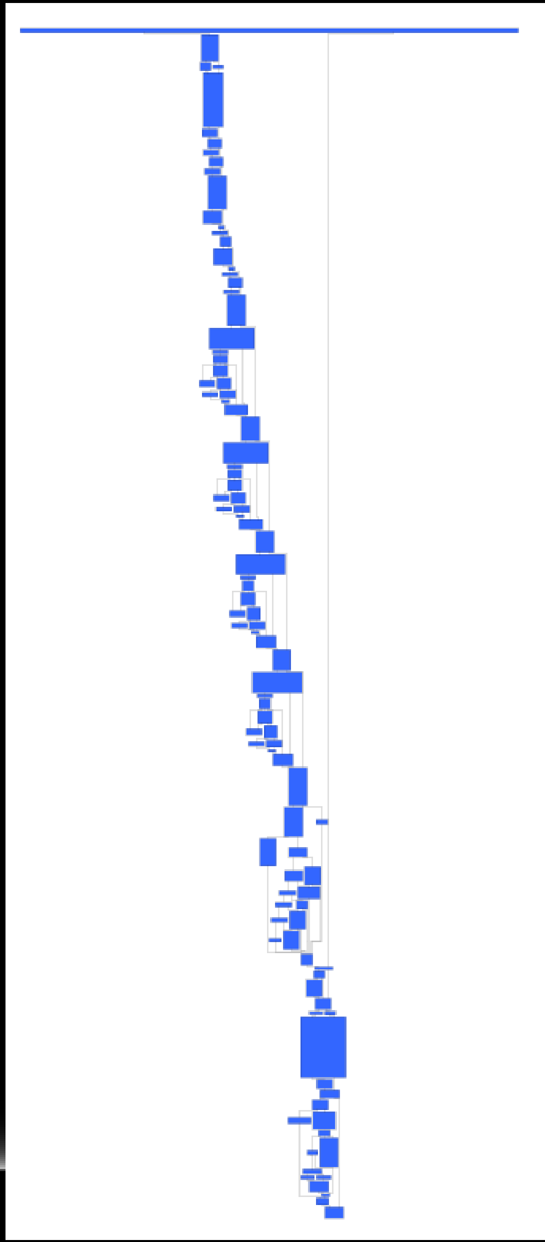
Generically Cleaning Up The Stack

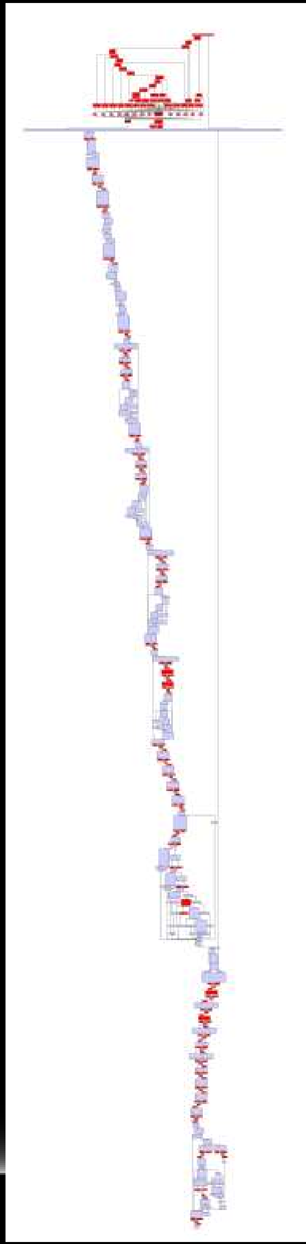
```
ActionPop                           # Remove method name
ActionPop                           # Remove object reference
ActionPush String:$RANDOM           # Create a variable with a random name
ActionStackSwap                     # Swap variable name and number of arguments
ActionSetVariable                   # Store number of arguments
RemovalLoop:
ActionPush String:$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
```

Recurity Labs

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
- One can easily see the significant code blow-up (~250% the original size)





Static Analysis

- We can provably not determine all call arguments using static analysis, therefore a code patch is the safer method
  - 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:
  - Backward tracing of the virtual machine stack using code flow
  - Code Flow Graphs in order to trace along basic blocks and edges
    - E.g.: even the constant pool can be overwritten anywhere in AVM1 code

Higher Level Verification Modelling

- The goal is to model:
  "Does the 2nd argument of any call to ObjectA.MethodB 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

Covering the AVM2

- The AVM2 implementation is its own can of worms
- AVM2 is currently still incomplete in Blitzableiter ☹

Real World Behavior

- We are “eating our own dog food” and are happy so far
  - YouTube and YouPorn work, and so do many other sites
  - Just in case, you can switch individual Tag type parsers in the configuration file from parsing and normalization to simple byte array copy mode
- Flash files with code obfuscation will in almost all cases be rejected for format violations within the AVM byte code
  - This also affects some larger sites, such as hulu.com
- Many third party SWF generators emit invalid Flash files
  - Use of undocumented Tag types for unknown purposes
  - Use of reserved fields or undocumented AVM byte codes
  - Simply ridiculously broken files, which the Flash Player will accept anyway (the problem!)

Please Report Compatibility Issues

- When a Flash file is rejected by Blitzableiter, you receive an error log dialog (configurable)
  - The dialog allows you to send the log to us, in case you are convinced the Flash file in question was not malformed
    - Please keep in mind that many non-malicious Flash files are nevertheless malformed files and should be filtered
  - We only store the API request and the log file content
    - It’s HTTP, sniff it yourself if you don’t believe us
- We also want to know about Flash files that are visually or audibly different from the not normalized input file. We need your help to fix those cases!

Conclusions

- We think that Blitzableiter shows the viability of signature-free protections against file format based attacks using a managed language parser and format normalization.
- Automated code property verification and enforcement allow distributors of Flash content to enforce contractual regulations and requirements right when they receive it.
  - Not surprisingly, it’s also a fairly tricky area.
- We hope the tool is a useful addition to your browser protection measures and we rely on your feedback!

Recurity Labs

Finishing Up

Acknowledgements

- Robert Tezli for his commitment to the project
- Dirk Breiden for being an awesome team mate
- Mumpi for general awesomeness
- Thomas Caspers and Daniel Loevenich for their support

Thank you!

fx@recurity-labs.com