# SWF and the Malware Tragedy
## Detecting Malicious Adobe Flash Files

fukami[1] and Ben Fuhrmannek[2]

[1] SektionEins, fukami@sektioneins.de, http://sektioneins.de/
[2] ben@fuhrmannek.de, http://fuhrmannek.de/

March 9, 2008

**Abstract.** Security of Adobe Flash based Rich Internet Applications (RIA) has become a subject for many concerns over the last year. Numerous tools for decompiling, disassembling and analysis are available, although most of them are not intended be used for security-related analysis. The recent attacks supplying malicious banner ads through high profile web sites are an example how easy it is to reach a large number of targets with relatively primitive techniques such as redirects from within a Flash banner. This paper is focussed on the detection of malicious SWF files.

# Table of Contents



## 1 Introduction

Flash has became one of the major topics for security concerns over the last year. While Adobe began to respond to these concerns and started to provide a lot of possibilities for protection by configuration [24], there are still attacks using Flash. The most recent attacks at larger scale utilised Flash banner ads for supplying malicious content. While these malicious banners are mainly redirectors tricking users to download and install malware, much nastier attacks are possible.

From the attackers perspective using banner ads on high profile sites has a lot of benefits. Beside the fact that banner on these sites have many page impressions and so many targets for exploitation, most ad networks and publishers still only superficially test Flash banners for appearance (i.e. if it fits into the layout without messing around). In most cases no tests for ActionScript functions happen. Many ad related companies also still think about SWF as pictures, but since Flash files can somewhat carry full applications it is no real surprise this kind of exploitation is quite successful.

But even if tests for ActionScript are present there are numerous ways to obfuscate SWF to hide malicious intentions. The attacker for example doesn't want the exploit to be triggered before a certain date (for example before the SWF is actually deployed on the target site) and wants to hide its present from the people in the content providers network so that testing doesn't reveal the real nature of the SWF.

For non-security people it is quite complicate to understand when a SWF is malicious or not since all tools available by now produce rather crude disassemblies. Also the knowledge of critical functions is not widespread enough, maybe beside functions which are directly loading arbitrary data from URLs (`getURL`, `loadMovie` etc.).

This paper will focus on possible exploit techniques with Flash and approaches for analysing SWF.

## 2 Common Attacks

The most common and best known attacks using Flash are redirections, XSS and CSRF as well as other approaches involving URLLoader or socket functions. Also privacy concerns with so-called Local Shared Objects aka Flash Cookies have been widely discussed in the past.

This paper is focussing on the following 4 types of attacks.

### 2.1 Redirections And Arbitrary Requests

Most of the well known attacks with Flash banners are redirections to malicious sites. Exploits like the one which recently hit Heise and some other bigger news sites used a very aggressive approach to convince users to download and install malware [2,3]. In the end this attack is more or less harmless since it doesn't directly try to exploit its victims. But this approach could also be used to redirect to sites hosting attack frameworks like MPack and alike.

The most common way for redirection is the use of ActionScript 2 based scripts with functions like `getURL` [26] and `loadMovie` [27] since it's the easiest way to file arbitrary requests. The `getURL` function for example is also capable of directly executing JavaScript, so it is well known to be critical. With ActionScript 3 the function `navigateToURL` [29] is used. A meaningful example use for this function in conjunction with the `URLRequest` [30] object can be found in the proof of concept exploit for requesting UPnP functionality via SOAP [4].

### 2.2 Attacks Using ExternalInterface

ActionScript 3 based URLLoader functions are not prone to JavaScript injections like the loader functions present in ActionScript 2. But there is a function present in ActionScript 2 and 3 which deals with JavaScript called `ExternalInterface` [28,31]. Since `ExternalInterface.call()` function behaves exactly like JavaScript's `eval()` it is possible to do for example DOM injections and call JavaScript cross domain [5]. This doesn't even violate security policies with the recent Flash Player as long as the embed code doesn't disallow scripting at all. `ExternalInterface` functions also make the development of attack back channels like BeEF [17] or Backframe [18] very easy.

Although not many attacks have been seen by now using `ExternalInterface` it is most likely to change in the future since it is a very flexible approach.

### 2.3 Socket Functions

The Princeton attack back in 1996 also used banner ads to demonstrate DNS rebinding attacks with Java [6]. This kind of attack was also present with Flash in the past and its discovery was an important step in sense of securing the Flash player [7,8,9]. However, simple port scans with the functions `Socket` [32] and `XMLSocket` [33] are still possible using a side channel attack as long as the socket functions are not completely disabled by Player configuration [10,23].

### 2.4 Attacking The Parser And Analysis Tools

Past Flash player versions had buffer and heap overflows as well as memory violations [11] and it still suffers from various parser problems. So it is likely that there are still attacks possible using low level exploits. Banner carrying such exploits probably have a big impact.

But not only the player can be attacked. Also some of the analysis tools like Flare [13] suffer problems from parsing SWF, and runtime analysis tools such as SWFIntruder [16] could also lead to an exploitation. While normal users won't probably be a target for such exploits it could be used to attack security researchers and other people analysing SWF.

## 3 Preventing Attacks

We would like to minimise the risk of being exploited by a seamless Flash application. There are several approaches of how to check such an application for security issues. A risky way would be just to run the code and see what happens. This method is known as runtime code analysis or dynamic code analysis. It may be exactly what we want in order to analyse reoccurring behaviours such as file downloads or network socket activity while residing in a safe testing environment. An ordinary debugger can easily track values of variables and trace function calls thus giving an accurate overview of the application's internal structure. Runtime code analysis, however, is hardly practical for use outside a testing environment since the code is actually being executed here.

As a result our approach to recognise potentially malicious Flash applications is purely based on the static code analysis, covered by the remainder of this section.

### 3.1 Static Code Analysis

The decision whether to trust an application enough to execute it will purely be the result of analysing the disassembly of the SWF file in question. During the process no part of the application's code is being executed.

The basis for our static code analysis is the internal structure of the SWF format itself. Having the disassembled SWF it is possible to apply a series of analysation patterns such as a pattern matching for a function call or its parameters. For each of the attacks described in section 2 an attempt is made to define a set of analysation patterns recognising the attack method.

### 3.2 SWF File Format

According to Adobe's SWF specification[21] the file starts with the string "FWS" or "CWS", followed by an 8-bit version number and 32-bit file length field. In case of CWS all the remaining file contents are ZLIB compressed:

```
[FWS] [Version] [Length] [Data]
```

or

```
[CWS] [Version] [Length] [Zlib Data]
```

The uncompressed data part starts with a header followed by a list of tags.

```
[Header] [Tag] [Tag] ...
```

Each tag acts as a container for a data type, e.g. for a JPEG image, RGB colour or an ActionScript bytecode. A tag starts with a tag type identifier and the tag's length, followed by arbitrary data.

```
[tag code and length (16 bits)] [data (length bytes)]
```

When decoding SWF files from the real world, undocumented or unknown tag codes may be encountered. There can be several reasons for these mysterious tag codes, for example the file could be corrupted or our parser could be incomplete. More likely, however, is either that a commonly used - yet undocumented - tag was used correctly according to the programmer's point of view.

The complete SWF looks like this:

```
[FWS/CWS] [Version] [Length] [ [Header] [[Tag Header] [Tag Contents]] ... [0] ]
```

As indicated, the last tag is a tag with tag type 0 and length 0 hence resulting in a 16 bit representation of 0.

Tags containing the program logic are of special interest to the static code analysis. Bytecode referenced as ActionScript 2 bytecode in this paper means an ACTIONRECORD, which is a list of actions. ACTIONRECORDs can reside in one of the following tags: `DoInitAction`, `DoAction`, `PlaceObject2`, `PlaceObject3`, `DefineButton`, `DefineButton2`. ActionScript 3 bytecode is contained in `DoABC` tags and is separately specified by the ABC Specification [22].

### 3.3 Analysation Patterns

Patterns are defined using Examples written in ActionScript 2 bytecode only. All patterns, with the exception of the described obfuscation method, have equivalents in ActionScript 3 bytecode. These have been omitted, since they are conceptually identical.

**Obfuscation.** In order to prevent exactly the static code analysis the bytecode could have been obfuscated. Some programmers like to obfuscate their code just to create the illusion of a property protection. For the purpose of a security analysis, disguised or obfuscated code is to be considered potentially malicious.

One common obfuscation method looks like this: ActionScript 2 bytecode (e.g. as located inside `doAction` tags) can contain branch actions[3] with a relative address offset pointing to an address outside the current tag. Example (written in pseudo code):

```
0x100: tag1 header with unknown tag code
0x104: code in tag 1
...
0x200: doAction tag
0x204: jump -0x100
```

This way the code inside `tag1` is hidden from ordinary SWF analyser tools and can still be executed. In order to make it even harder to find the hidden code, a random bytecode sequence could be inserted in between actual bytecode, or dormant bytecode (which is never executed) could be used as distraction. A variation of this obfuscation method is to hide code not in another tag, but in the same tag after deceivingly adding an end-of-actionlist action (action code 0).

Fortunately this technique is also really easy to detect since a checker only needs to be able to check for uncommon branch offsets. However, most disassemblers, such as Flare, can be fooled.

**Code Inconsistencies.** The SWF format specification was written for building compilers assembling SWF files. Any deviation in the binary code of SWF files either not covered or (explicitly or implicitly) ruled out by the specification is an inconsistency. Examples:

- A string would normally be null-terminated, but could come with a redundant length field. This length may differ from the actual size of the string resulting in an inconsistency.
- Lists of Actions, e.g. in the `DoAction` tag, are terminated by an action with code 0, but the list could go on.
- Negative values of signed numbers could make no sense in a specific context, like for the application frame's geometry.
- Tags or Actions could be incomplete or malformed, e.g. unterminated strings, premature end of tag.
- Unknown tag or action codes can disguise hidden code for obfuscation.

**Constant Pool Pattern Matching.** The action `ConstantPool` defines a list of string constants. If referenced accordingly, they can have the meaning of function names, function parameters or parts of function parameters. The occurrence

[3] aka. jump or goto

of well known names of functions or objects provided by the Flash Player could indicate the use of a referenced object. Additionally a detailed analysis of all references to the constant pool can reveal the purpose of each constant individually. A matching can be applied to the constant's value in both cases, with or without its context.

**Tag and Action Matching** Tags and Actions can both be matched by code, which is documented in the format specification.

### 3.4 Sets of Analysation Patterns

The following case by case study is an attempt to define criteria for the detection of attacks discussed above. Each criterion is composed of one or more analysation patterns and relates directly to one of the attacks as described in 2.

Criteria can serve as a basis for automated filters such as filtering proxy servers or browser plug-ins. The general idea is to analyse all data requested by a web-browser, then detect SWF files and apply the criteria on the fly.

For each attack all of the following questions should be answered:

- Can the attack be recognised by using any combination of analysation patterns, and how?
- Assuming an ordinary Flash banner advertisement is being analysed, what are the security implications? This point is of particular interest, since anyone can place arbitrary Flash files onto web-pages just by booking ad space. This way banners equipped with appropriately prepared bytecode could reach and exploit a vast number of target systems.

**Redirections.** Redirections can only be accomplished by loading a URL, which involves any of the following calls: `LoadMovie`, `LoadMovieNum`. These can be found using the constant pool pattern matching. In addition there is the Action `getURL`.

Finding an occurrence of any such function is not enough here to assume security implications. Ads, for example, regularly load data such as context or location sensitive information into the running application. That means the URL to be loaded would have to be processed further, e.g. matched against a whitelist or blacklist of URLs. Unfortunately this URL could also be put together on the fly incorporating external arguments. In fact it happens to be a common way for banner ads to supply the embedded Flash with the redirection URL as an argument. Since the argument is not known at the time of the analysis only the mere existence of non-constant parameters given to the function calls mentioned can be matched against safely.

**ExternalInterface and DOM injections.** The use of the `ExternalInterface` can be matched by using the constant pool pattern matching. Unless there is a reason why the Flash application should be allowed to access the underlying

DOM or any other JavaScript function, the use of this interface alone may be an indication for security implications. As far as banner ads are concerned there is no such reason.

**Socket functions.** The use of socket functions can be matched by using the constant pool pattern matching. While it can happen that banner may take external data for displaying (i.e. stock exchange information or special offers) to be able to easily update information it is most likely that the existence of such functions indicates a security risk. Ads would never have to use sockets.

**Attacking the parser and analysis tools.** Any code inconsistency not anticipated by the programmer of a format parser may result in an undetermined behaviour of the parser. If this includes granting the program access to memory or functions normally not used by the program, it may be exploitable.

It is possible to check a SWF file for well known inconsistencies such as the examples mentioned in section 3.3/Code Inconsistencies. However yet undiscovered flaws in parsers could still pose a vulnerability. A common technique to harden parsers against such attacks is format fuzzing.

### 3.5 Evaluation

The following list of freely available tools shows how the analysation patterns described in section 3.3 can be put into practical use. It includes disassemblers for SWF, ActionScript 2 and ActionScript 3 bytecode.

- Flare and Flasm: Flare[13] and the Open Source tool Flasm[14] can disassemble SWF up to Version 8 including ActionScript 2 bytecode. The resulting data is saved in human readable pseudocode.
  They can be used for the analysation patterns 'Constant Pool Pattern Matching' and 'Tag and Action Matching'. In case of obfuscated SWF files the resulting output is incomplete.
- Tamarin abcdump: The abcdump tool is part of the Tamarin project[15]. ActionScript 3 bytecode can be decompiled into pseudocode source code. As such it is suitable for the ActionScript 3 equivalent of the 'Constant Pool Pattern Matching'. Obfuscation as described in 3.3 is not known for ActionScript 3 bytecode by now.
- erlswf: The erlswf [12] toolkit provides an Erlang library for disassembly of SWF files up to version 9. It has been designed with security as its main focus. Unknown tags and the use of the branch offset obfuscation method explained in section 3.3 can be recognised using the provided command line tool. Up to date the analysis of ActionScript 2 bytecode is limited to the `DoAction` and `DoInitAction` tags.
  Erlang is a programming language designed for process oriented and concurrent software design. With this and the idea of a proxy as introduced in section 3.4 in mind, the erlswf toolkit could be used to implement real-time analysis of SWF files.

## 4 Conclusion

This paper reflects the current state of research in malware ads based on Adobe Flash and has to be considered "work in progress". So the findings in this paper are far from being complete. While most static tools for analysing SWF were not developed to be used for security testing and analysis, erlswf is known to be the first with this intention (SWFIntruder is intended to be used for ActionScript 2 based SWF runtime analysis). However, erlswf is still missing some abilities, for example full AVM2/ActionScript 3 support and support for ActionScript 2 tags other than `DoAction` and `DoInitAction`.

With some of the tools mentioned in 3.5 (Flare, Flasm) static analysis is much easier to use for a tester who is aware of critical ActionScript functions. But neither Flare nor Flasm help much in building a tool chain for a quick real time analysis or even a proxy and they are not supporting ActionScript 3 at all.

The conditions where exploitation through banners are possible depend very much on the way how the SWF is embedded into the calling page and configuration of the player. Ad networks are advised to supply the parameter `allowScriptAccess` with the value `never` in order to make exploitation much harder.

Adobe should introduce a way for users to be notified when cross domain calls are triggered from inside a SWF and the ability to block this requests before they happen. One way could be another option for general configuration.

In general there are no known *official* guidelines for securely testing and deploying banner ads by now beside some tips from Adobe covering general Flash security considerations [25]. Such guideline should contain for example:

- A description how SWF has to be tested in a safe environment, what tools need to be used and how the results have to be interpreted by the testers
- Functions which are not allowed inside a SWF for safe deployment (i.e. Socket, ExternalInterface, eval'd ActionScript 2 code)
- How to safely embed SWF into the ad code

Another interesting topic which goes beyond the scope covered by this paper is obfuscation of SWF. While some Flash developers think it is important for them to obfuscate their work in order to protect their intellectual property from decompiling and analysis, from the security perspective all obfuscated SWF should be completely rejected, front and foremost when being used as a banner ad.

## References

1. John Viega, J.T. Bloch, Tadayoshi Kohno, and Gary McGraw. Its4: A static vulnerability scanner for C and C++ code. In Proceedings of the 16th Annual Computer Security Applications Conference, 2000.
2. Sandi Hardmeier. Heise.de hit by malicious banner advertisement. Website, https://msmvps.com/blogs/spywaresucks/archive/2008/01/15/1463622.aspx, January 2008.

3. Sandi Hardmeier. Malicious advertisements and advertising fraud. What do we know? Website, http://msmvps.com/blogs/spywaresucks/archive/2007/12/08/1386804.aspx, December 2007.
4. Petko D. Petkov. Hacking the interwebs. Website, http://www.gnucitizen.org/blog/hacking-the-interwebs/, January 2008.
5. fukami. DOM manipulation with Flash using ExternalInterface. Demo, http://dom.flashsec.org/, March 2008.
6. Princeton Attack: DNS Attack Scenario. Website, http://www.cs.princeton.edu/sip/news/dns-scenario.html, February 1996.
7. Martin Johns. (somewhat) breaking the same-origin policy by undermining DNS pinning. Website, http://shampoo.antville.org/stories/1451301/, August 2006.
8. Kanatoko Anvil. Anti-DNS Pinning + Socket in Flash. Website, http://www.jumperz.net/, August 2006.
9. Collin Jackson, Adam Barth, Andrew Bortz, Weidong Shao, Dan Boneh. Protecting Browsers from DNS Rebinding Attacks. Paper, http://crypto.stanford.edu/dns/dns-rebinding.pdf, October 2007.
10. David Neu, fukami. Design flaw in AS3 socket handling allows port probing. Website, http://scan.flashsec.org, July 2007.
11. FlashSec: Advisories regarding Adobe Flash. Website, https://www.flashsec.org/wiki/Advisories
12. Ben Fuhrmannek. erlswf: Toolkit for diassembling SWF up to version 9. Tool, http://code.google.com/p/erlswf/
13. Igor Kogan. Flare: ActionScript decompiler. Tool, http://www.nowrap.de/flare.html
14. Igor Kogan. Flasm: Command line assembler/disassembler of ActionScript bytecode. Tool, http://www.nowrap.de/flasm.html
15. Tamarin project:Open-Source implementation of the ECMAScript 4th edition (ES4) language specification. Tool, http://www.mozilla.org/projects/tamarin/
16. Stefano Di Paola. SWFIntruder: Runtime analyzer for SWF. Tool, http://code.google.com/p/swfintruder/, December 2007
17. Wade Alcorn. BeEF: Browser Exploitation Framework. Tool, http://www.bindshell.net/tools/beef/
18. Petko D. Petkov. Backframe: Attack console for exploiting web browsers. Tool, http://www.gnucitizen.org/projects/backframe/, October 2006
19. David A. Wheeler. Flawfinder. Tool, http://www.dwheeler.com/flawfinder, 2001.
20. Inc. Secure Software. Rats - rough auditing tool for security. Tool, http://www.securesoftware.com/resources/download_rats.html, 2001.
21. SWF Format Specification by Adobe (requires registration). Specification, http://www.adobe.com/licensing/developer/
22. ABC Format Specification. Specification, http://www.adobe.com/devnet/actionscript/articles/avm2overview.pdf
23. Socket connection timing can reveal information about network configuration Website, http://kb.adobe.com/selfservice/viewContent.do?externalId=kb402956
24. Adobe Flash Player Administration Guide Paper, http://www.adobe.com/devnet/flashplayer/articles/flash_player_admin_guide/flash_player_admin_guide.pdf
25. Adobe Flash Player Player 9 Security Guide Paper,http://www.adobe.com/devnet/flashplayer/articles/flash_player_9_security.pdf

26. ActionScript 2 reference: MovieClip.getURL. http://livedocs.adobe.com/flash/8/main/wwhelp/wwhimpl/common/html/wwhelp.htm?context=LiveDocs_Parts&file=00001730.html
27. ActionScript 2 reference: MovieClip.loadMovie. http://livedocs.adobe.com/flash/8/main/wwhelp/wwhimpl/common/html/wwhelp.htm?context=LiveDocs_Parts&file=00002479.html
28. ActionScript 2 reference: flash.external.ExternalInterface. http://livedocs.adobe.com/flash/8/main/wwhelp/wwhimpl/common/html/wwhelp.htm?context=LiveDocs_Parts&file=00002200.html
29. Flex 2 Language Reference: flash.net.navigateToURL. http://livedocs.adobe.com/flex/2/langref/flash/net/package.html#navigateToURL()
30. Flex 2 Language Reference: flash.net.URLRequest. http://livedocs.adobe.com/flex/2/langref/flash/net/URLRequest.html
31. Flex 2 Language Reference: flash.external.ExternalInterface. http://livedocs.adobe.com/flex/2/langref/flash/external/ExternalInterface.html
32. Flex 2 Language Reference: flash.net.Socket. http://livedocs.adobe.com/flex/2/langref/flash/net/Socket.html
33. Flex 2 Language Reference: flash.net.XMLSocket. http://livedocs.adobe.com/flex/2/langref/flash/net/ExternalInterface.html