Recurity Labs GmbH

http://www.recurity-labs.com

2010-07-05

Preventing Adobe Flash Exploitation

Blitzableiter - a signature-less protection tool

Abstract

Adobe Flash is the most widely deployed Rich Internet Application (RIA) platform. The large population, coupled with design and implementation weaknesses, makes Flash an attractive target for client-side exploitation as well as online fraud using Flash advertisements. The protection tool “Blitzableiter” implements a new approach to counter both attack types by using file format normalization and dynamic Flash code modifications. While the current implementation targets Adobe Flash, the author believes that the general approach is applicable to other complex file formats that are often used to carry out client-side attacks.

Author

Felix 'FX' Lindner
Head of Recurity Labs

1 Introduction

Rich Internet Application platforms, such as Adobe Flash, Microsoft Silverlight and Sun JavaFX currently play an important role in the dynamic presentation of online content. The Adobe Flash run-time environment, also known as Flash Player, is installed on 97%[1] of computers and mobile devices used for web browsing and many popular web sites employ Flash for implementing dynamic content presentation.

The large distribution of the Flash run-time makes it an attractive target for client-side attacks. Most commonly, the attacker identifies and exploits a vulnerability in the parsing code of Adobe Flash by creating an intentionally malformed file, which results in an exploitable memory corruption when viewed with a Flash enabled web browser.

Another area of concern is the use of Flash files for banner advertisements and third party applications on modular web sites. In this case, the native functionality of Flash is used to present apparently innocent content when submitting the application to the web site or advertisement network for review. After the Flash file is accepted and distributed on the site or advertisement network, it will change its behavior and pull personal information from its surrounding web site or forward the user to a malware distribution site. Several popular online news media sites[2] suffered from incidents with malicious banner advertisements.

Therefore, a defense approach must cover both types of attacks in order to be applicable for both end users and web site operators.

2 Overview of the Attack Surface

The file format used by Adobe Flash is called the SWF (pronounced “swiff”) format. Originally developed by FutureWave Software, it was subsequently expanded by Macromedia and Adobe. The file format can carry a variety of content types, including graphic objects, type sets for font rendering, sound and video data as well as code for the virtual machines provided by the Flash run-time. The current specification[3] of the file format covers SWF version 10, which still supports all data structures from previous versions.

Internally, the format is split into so-called Tags, which then carry a certain type of content. Tags are simple Type-Length-Value container data structures. Adobe specifies 63 different Tag types, each carrying a variety of often complex and nested data structures. Flash authoring tools or SWF generators often use additional undocumented Tag types to include proprietary data in the output file.

Since SWF is a multimedia presentation format, it can include a range of other media formats, such as MP3, various video formats as well as PNG, JPEG and GIF images.

The Flash Player provides two independent and incompatible virtual machines, for which the SWF format can carry byte code. The virtual machines are called Adobe Virtual Machine (AVM) 1 and 2 respectively. The AVM1 is historically grown since SWF version 3 with the subsequent expansion of Flash's functionality. SWF version 9 introduced the

[1] http://riastats.com/, July 2010

[2] Handelsblatt.de and Zeit.de (http://www.heise.de/security/meldung/Schaedliche-Werbebanner-auf-Handelsblatt-de-und-Zeit-de-2-Update-921139.html), New York Times (http://www.nytimes.com/2009/09/15/technology/internet/15adco.html?_r=3)

[3] http://www.adobe.com/devnet/swf/pdf/swf_file_format_spec_v10.pdf

new AVM2, which is based on ECMA-262 with some modifications made by Adobe. Both AVMs are stack machines, which interpret and run byte code provided within the SWF file. AVM2 is capable of Just-in-Time compilation of byte code into native machine code. A flag in the File Attributes Tag of a SWF file indicates which virtual machine is to be used when the the file is loaded for rendering.

Both virtual machines provide access to an API and Flash internal classes, which allow the byte code to communicate with the web browser, manage browser independent local storage and initiate direct network communication with third party systems. The AVM2 also allows dynamic loading of additional byte code, which can be used to emit code at run time. Neither of the two virtual machines allow self modifying code.

The sheer amount of parsing code required to handle this complex file format, coupled with the complexity and power of the virtual machines, creates a very large attack surface. Since the Flash Player is written in an unmanaged language, out-of-bounds memory operations caused by corrupted data structures in an SWF file often lead to exploitable vulnerabilities, which in turn allow the execution of arbitrary code within the process space of the Flash Player or web browser.

Additionally, the Flash Player is implemented in a way that tries to ignore most parsing and byte code execution issues. This is apparently done in order to allow slightly malformed files to still display content so that the user experience is unharmed. This approach has allowed a wide range of incorrectly formatted SWF files to be used online, which complicates the task of identifying maliciously malformed files even further.

3 File Format Normalization

To protect a fragile parser from exploitation through malformed input files, the well-formedness of the input must be guaranteed. The approach is to implement a parser for the file format in a managed language and employing the strictest verification of the input file. The managed language environment provides automatic protection against out-of-bounds operations, such as buffer overflows, as well as integer overflows or sign issues. The strict validation of the specified components of the file format allow to detect and prevent utilization of undocumented aspects of the format.

Once the input file is parsed in its entirety, the original file is discarded. This leaves only well understood and correctly formatted data structures and byte code, which are then used to generate an output file.

Is the input a well-formed file, the process of the format normalization produces an output that is functionally equivalent to the input file. If the input file exhibits slight format violations, such as using reserved bits and fields, the output generation will correct those. Input files with significant format violations are rejected during the initial parsing pass, since there is a high likelihood that they will cause the consumer parser to fail and potentially represent attempts at exploiting a vulnerability in the same.

The more thorough the normalization parser is, the better is the protection provided. Therefore, it is required that the normalization parser implements all documented aspects of the file format. If the file format can carry other data formats, those must be parsed as well, as they could also be used to exploit the consumer parser.

While the creation of a defensive, secure and strict parser for a complex file format, such as SWF, requires a large initial effort, it has the benefit of not requiring constant updates or specific attack signatures. The approach follows a white-listing strategy, only allowing known-to-be-good data to pass verification and be placed in the output file.

In cases, where the input passes verification but still triggers a vulnerability in the final consumer parser, the process of recreation provides a minor additional exploit mitigation layer, since the attacker cannot anticipate the structure of the generated output with complete certainty. While this obstacle can certainly be dealt with by an experienced attacker, it still prevents less sophisticated exploits from executing their payload, as most of the offsets within the output file will differ from the input file.

4 Byte Code Modifications

The prevention of misusing Flash's functionality, as in the case of malicious banner advertisements, requires the defense tool to analyze the byte code within the SWF file.

While AVM1 and AVM2 byte code have specific Tag types in the SWF file, several other components of the file format, such as buttons or graphical shapes, can carry independent AVM1 code embedded within their respective data structures. All of these code locations must be inspected to prevent malicious code from passing through the defense layer.

After the initial parsing is completed and the well-formedness of the byte code is verified, all byte code instructions that invoke potentially unwanted functionality can be easily identified. Static code flow analysis is then employed to determine whether the arguments of an instruction are static or not.

Instructions that invoke functionality and have static arguments can be handled directly. The input file can be rejected entirely or the offending code is removed from the byte code, leaving all other functionality intact.

When encountering instructions whose arguments cannot be determined with static analysis, the inspection code will emit AVM byte code that carries out the defined check at run time. Through this patch of the AVM code, the check is executed by the final consumer run-time environment, hereby preventing unwanted functionality.

Using this two-sided approach, the number of code locations that must be modified is reduced by the static analysis step, while the modified code locations ensure that all instructions are verified.

To minimize the risk of a semantic deviation between the static analysis and the modified AVM code, the process is implemented as an minimal stack machine, which executes individual fine-granular steps of the code flow analysis. If the stack machine runs into a state where it can no longer guarantee that the value of an argument can be reliably determined, it switches its mode of operation and executes the same sequence of steps again, this time emitting the functionally equivalent AVM code.

This approach also simplifies the development of the rules that are to be enforced, as it allows a certain level of abstraction. The user or developer can specify what API function is of concern to him and declare conditions that the arguments to this API function must fulfill.

5 Implementation and Use

The defense approach described above has been implemented in the tool “Blitzableiter”, which is published[4] as open source under the GNU Public License, version 3. Providing the source code for the tool allows for a maximum of transparency regarding its real value in specific scenarios and prevents users from getting a false sense of security, as no protection mechanism will ever reach 100% coverage of the attack surface.

The managed language of choice is the .NET language C#, due to the superior security properties of the .NET Common Language Run-time (CLR) and the clear structure of the source code. Blitzableiter is build to target the .NET CLR 2.0, which allows it to be binary compatible to both Microsoft Windows as well as open source operating systems using the Mono run-time environment for .NET.

Blitzableiter can be used as a pure command line tool, allowing for easy integration into central systems, such as proxy servers.

Thanks to the cooperation of Giorgio Maone, author of the popular NoScript add-on for the Mozilla Firefox browser, NoScript now supports arbitrary content filters for specific MIME types of objects embedded in web pages. This allows a larger user base of security minded people to use Blitzableiter on a daily basis. NoScript will invoke Blitzableiter for every allowed Flash object on a page and enables the user to selectively disable the content filter in case compatibility issues arise (see 6).

Web site or advertisement network operators can customize Blitzableiter in order to enforce contractual requirements placed on the Flash content provided to them. For example, a banner advertisement submitted for a specific campaign can be ensured to only forward users to a specific URL, which prevents the Flash banner ad from later changing the destination, a common trick used to perform so-called click-fraud.

6 Challenges

The primary challenge the approach faces is that of compatibility and user acceptance. Since the Adobe Flash Player will accept many malformed types of SWF files, programs that emit SWF files often produce files that deviate far from the specification of the format. While web site operators can require well-formed SWF content and verify the same using Blitzableiter, the application as content filter within a web browser could be hindered by Flash content no longer working. This can only be overcome by a large number of samples, which then must be inspected individually and decided upon.

Another challenge is the amount of embedded third party and proprietary formats, for which no or insufficient format specifications are published. Formats with published specifications will subsequently be added to Blitzableiter, while undocumented formats may be either allowed as pure data blocks or filtered out, depending on the user's preferences.

The approach also suffers from undecidable cases, which again only affect the application within a web browser. The looseness of the native Flash Player byte code execution offers a variety of tricks commonly employed by Flash obfuscation software. Those attempts to protect the intellectual property of the Flash code result in AVM byte code that fails verification, as it will include invalid byte codes and intentionally convoluted code flow. A popular example of such Flash content is the video player of the Hulu.com site. It can only be left to the user to disable Blitzableiter for specific sites, which the NoScript add-on supports in a convenient way.

[4] http://blitzableiter.recurity.com

7 Conclusions

The developers of Blitzableiter believe the file format normalization approach, coupled with the adaptive modification of AVM byte code, is a sound approach to counter a large fraction of the existing Flash attack landscape.

The release of the first official beta version of Blitzableiter at BlackHat USA 2010 will allow end users to integrate the tool with their Mozilla Firefox browsers and the NoScript add-on. This will hopefully aid the identification and remedy of remaining compatibility issues, which could deter users from employing the tool. It will also allow security researchers to verify the protection value of Blitzableiter by testing its operation on known exploits.

The possibility to automatically test Flash files as well as the enforcement of AVM code properties and behavior should enable web site operators to tighten their Flash content submission policies, while reducing manual work at the same time.

The further development of Blitzableiter will entirely depend on the feedback the team receives from users and the security community.