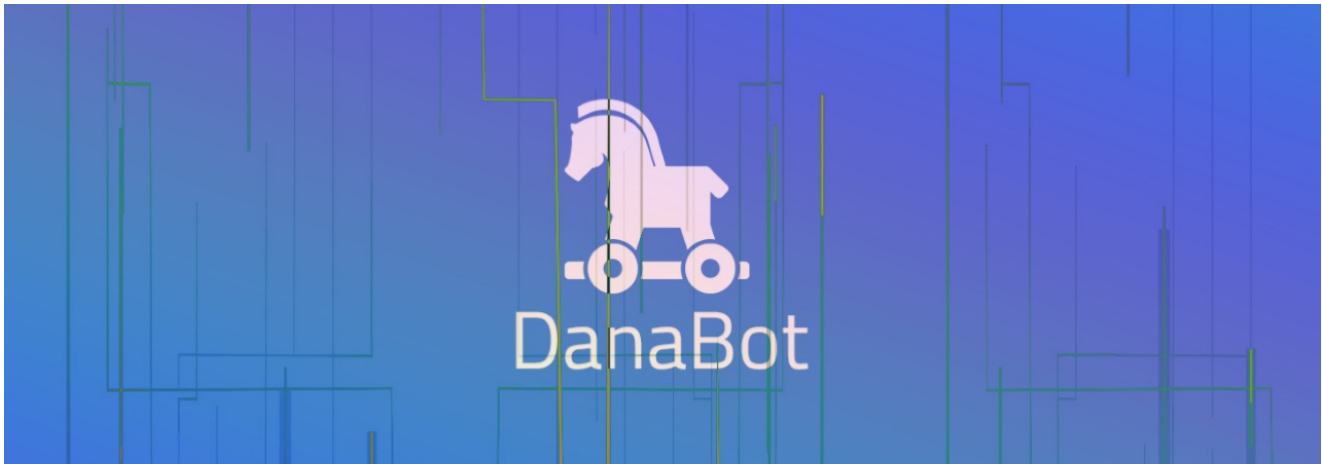


Config Extractor per DanaBot (PARTE 1)

malverse.it/costruiamo-un-config-extractor-per-danabot-parte-1



Introduzione

Ciao a tutti, oggi volevo analizzare la sfida bi-settimanale lanciata di Daniel di Zero2Auto che consiste questa volta nel scrivere un Config Extractor che funzioni per le diverse versioni di **DanaBot**, un malware scritto in **Delphi**.

Da Malpedia:

Proofpoints describes DanaBot as the latest example of malware focused on persistence and stealing useful information that can later be monetized rather than demanding an immediate ransom from victims. The social engineering in the low-volume DanaBot campaigns we have observed so far has been well-crafted, again pointing to a renewed focus on “quality over quantity” in email-based threats. DanaBot’s modular nature enables it to download additional components, increasing the flexibility and robust stealing and remote monitoring capabilities of this banker.

Ci vengono forniti questi quattro link, che ci permettono di ottenere diverse versioni del sample:

Logicamente possiamo ottenere altri sample classificati come Danabot, ad esempio [qui](#) e [qui](#).

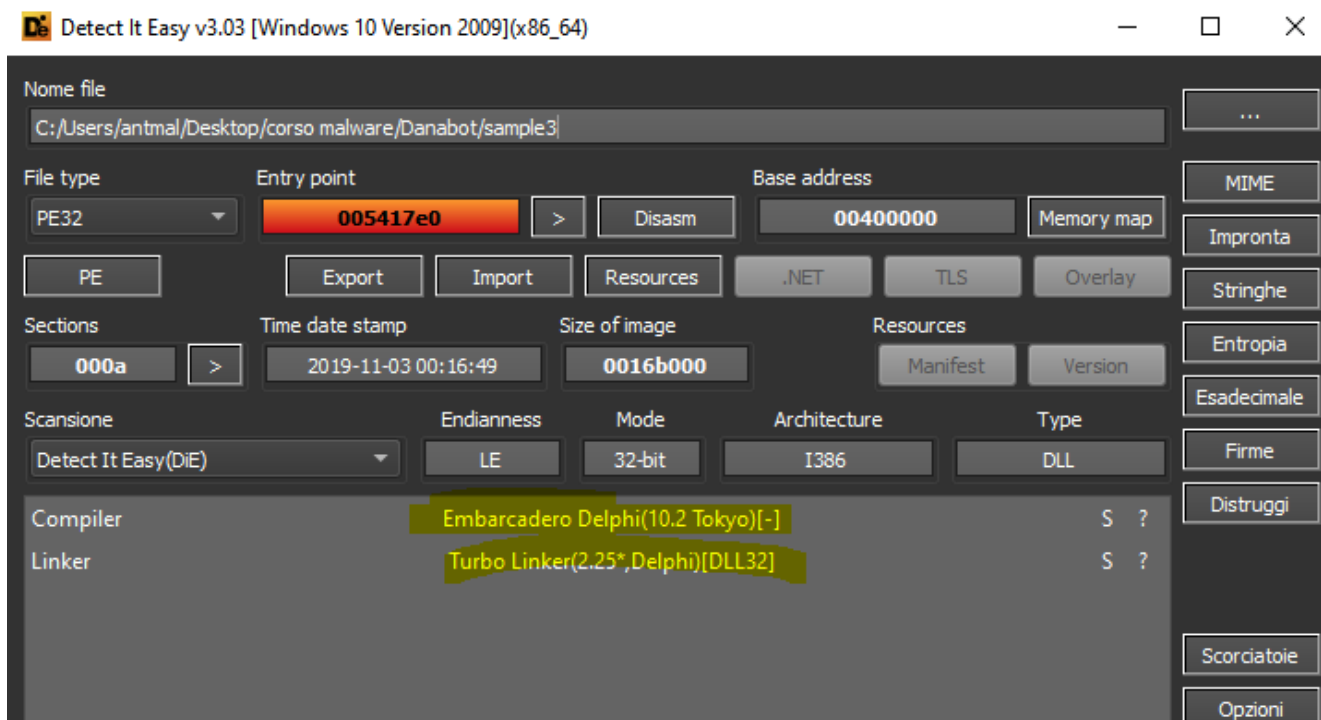
In particolare, in questo post analizzeremo tre sample (**MD5: 6b448c6851f3235c9b3d0c24353c480f, 5c0be4a5273dec6b3ebb180a90f337f2, 611c2bf7aa7bb62e90f3a92f3682c0b5**), realizzando un semplice script per estrarre gli IP del

C&C; nei prossimi post analizzeremo come avviene la comunicazione con il C&C, identificheremo le funzioni di cifratura, estrarremo la chiave RSA e realizzeremo lo script finale che funziona sui diversi sample forniti.

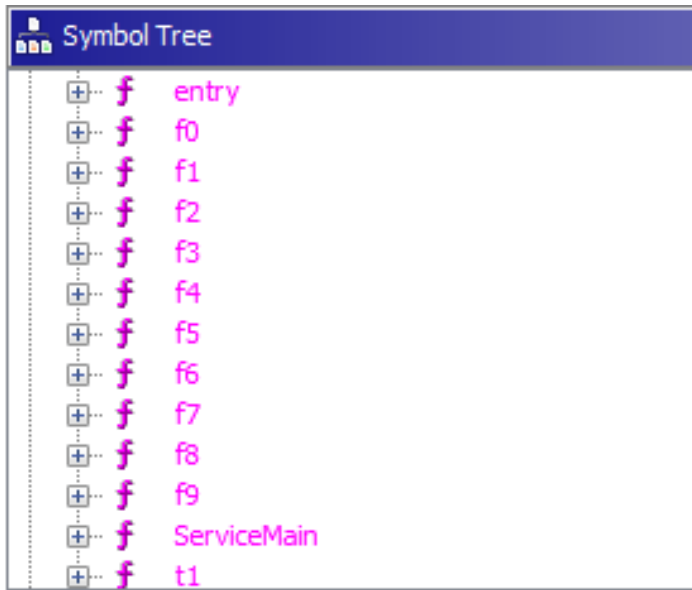
Partiremo da analizzare il primo sample, si tratta del **Main Component** di DanaBot, successivamente analizzeremo dei sample più complessi che contengono al loro interno il Main Component.

Analisi Main Component Danabot

Partiamo analizzando il primo sample (**MD5: 6b448c6851f3235c9b3d0c24353c480f**); si tratta del Main Component di DanaBot, sviluppato in **Delphi** ed esporta diverse funzioni (f0, f1, ... , f9):



DIE rileva che si tratta di una DLL scritta in Delphi



Funzioni esportate dalla DLL

Analizzando la funzione F0, dopo la decifratura di diverse stringhe e la creazione di un altro thread, troviamo la creazione di un thread che contiene diverse chiamate per effettuare operazioni con i **socket** (per chi volesse maggiori informazioni sul funzionamento dei socket può consultare [questa](#) ottima guida); tracciando i parametri passati a queste funzioni, riusciamo ad ottenere dove effettivamente avviene la creazione del config.

```

else {
    FUN_005243a4();
    local_40 = local_44 + 0x26e;
    local_3c = local_44 + 0x4a5;
    local_48 = local_18 * 0x32a;
    FUN_00523cfc();
    local_48 = local_18 * 0x2ff;
    *PTR_DAT_005458a8 = 1;
    CreateThread((LPSECURITY_ATTRIBUTES) 0x0, 0, InternetOperation, (LPVOID) 0x0, 0, &local_8);
    local_38 = local_40 + 0x1fa;
    local_3c = local_40 + 0x23e;
    pOVar2 = L"蚘\x01/\0摩00";
    capa::anti-analysis::anti-debugging::debugger-detection::fun.FUN_00532d28();
    local_34 = 0x18e - local_38;
    local_30 = 0x13b - local_34;
    counter = 0;
}

```

Creazione del Thread principale che si occupa di comunicare con il C&C

La funzione che ci interessa attualmente è **inet_addr**, essendo che ha come parametro l'IP in formato dotted-decimal; in realtà non troveremo l'IP direttamente in questo formato, ma l'IP in formato decimale verrà prima convertito con una semplice funzione che ho rinominato **IntToIP** e poi passato a `inet_addr`:

```

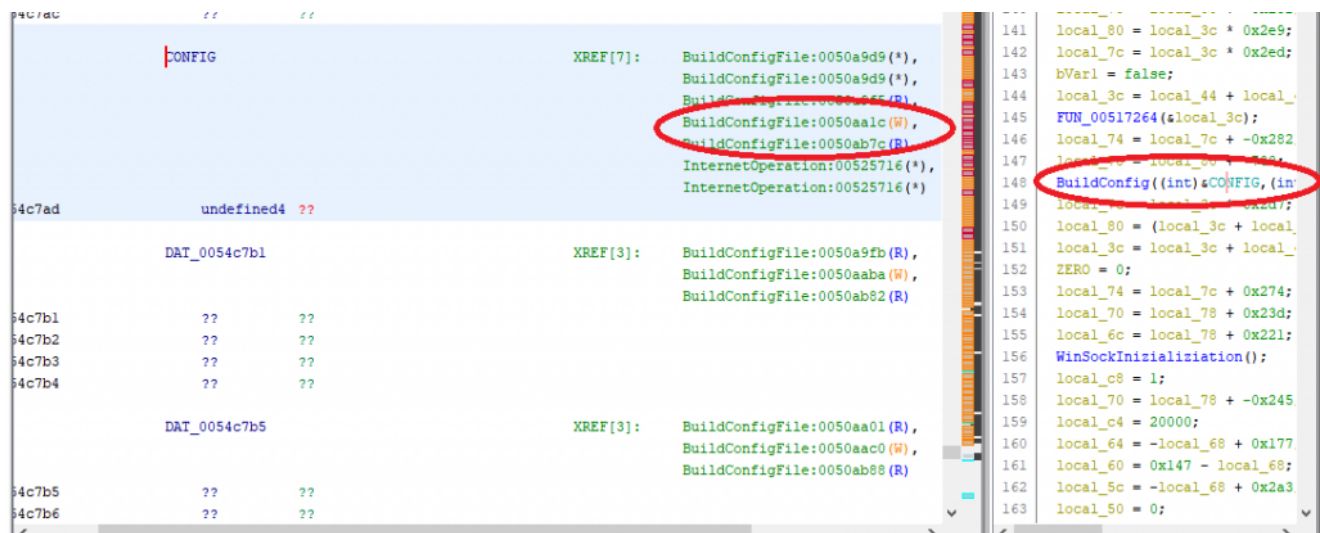
local_44 = local_28 * 0x325;
FUN_004075bc((undefined8 *)local_14,0x10,0);
local_14._0_2_ = 2;
local_38 = local_3c * 0x1b0;
local_3c = local_44 + -0x6c;
local_40 = local_28 * 0x30b + -0x297;
local_44 = local_28 * 0x2c0;
local_14._2_2_ = htons(local_1a);
local_38 = (local_40 + 0x243) * 0x1c6;
local_40 = local_44 + 0x24e;
local_3c = local_44 + 0x471;
IntToIP(intIP,&IP);
cp = (char *)FUN_0040a248((int)IP);
local_10 = inet_addr(cp);
local_40 = local_44 + -0x25a;
if (local_2c < local_28) {
    FUN_0050efa4(&local_28,&local_28);
}
FUN_00510370();
local_40 = local_44 + -0x25a;
iVar1 = connect(local_20,(sockaddr *)local_14,0x10);
if (iVar1 != 0) {
    local_28 = local_28 * 2;
    FUN_00511470(&local_28);
    local_40 = local_44 + -0x28d;
    local_3c = local_44 + -0x58;
    capa::communication::fun.FUN_005150b0(local_20);
    local_28 = local_28 + local_2c;
    FUN_00510850();
}

```

Operazioni con i socket

e conversione dell'IP da int

Il primo parametro della funzione IntToIP è un parametro a sua volta della funzione padre, quindi analizzo le chiamate a questa funzione (solo una) e traccio tale valore; viene referenziato solo in due funzioni e in particolare una è interessante perché come parametro ha una variabile globale:



Funzione che accede in scrittura al Config

Questa variabile è acceduta da diverse funzioni, in particolare una di queste effettua la scrittura in questa zona di memoria; essendo che non contiene dati, viene quindi popolata in runtime, avvio quindi il debugger e confermo che questa zona di memoria contiene proprio il config (inizia con **3C** e termina con **4E**):

Indirizz	Hex	ASCII
007CC7AD	3C 00 00 00 31 BB 00 00 00 00 00 00 6D BB 00 00	<...1>.....m>..
007CC7B0	00 00 00 00 F3 7F 2B 06 40 7E AF 02 82 0F E6 98	...ó.+.@~...æ.
007CC7C0	4A 63 88 C0 F4 0E E2 23 5F B3 A8 25 33 81 4C 08	Jc.Àð.â#_">%3.L.
007CC7D0	97 D2 55 9F 2D 4C 7B B1 4B 39 0E 79 00 00 00 4E	.ÒU.-L{±K9.y...N
007CC7E0	17 15 04 94 17 15 04 AC 17 15 04 C4 17 15 04 DC	...4...L...d...
007CC7FD	17 15 04 94 17 15 04 AC 17 15 04 C4 17 15 04 DC	...ô...\$...<
007CC80D	17 15 04 F4 17 15 04 0C 18 15 04 18 15 04 3C	...T...l...Ä...Û
007CC81D	18 15 04 54 18 15 04 6C 18 15 04 84 18 15 04 9C	...ô...\$...<
007CC82D	18 15 04 B4 18 15 04 CC 18 15 04 E4 18 15 04 FC	...T...l...Ä...Û
007CC83D	18 15 04 14 19 15 04 2C 19 15 04 44 19 15 04 5C	...ô...\$...<
007CC84D	19 15 04 74 19 15 04 8C 19 15 04 88 1A 15 04 00	...T...l...Ä...Û
007CC85D	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	...ô...\$...<
007CC86D	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	...T...l...Ä...Û
007CC87D	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	...ô...\$...<
007CC88D	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	...T...l...Ä...Û
007CC89D	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	...ô...\$...<

Config

ottenuto con il debugger

Confermata che fosse questa la funzione che costruisce il config, trovo infatti a un certo punto una variabile globale che contiene i diversi IP:

The screenshot shows a debugger window with a list of memory addresses and their hex values. A red box highlights a range from 00545768 to 0054578c. The values are: 00545768 f3 7f 2b 06, 0054576c 40 7e af 02, 00545770 82 0f e6 98, 00545774 4a 63 88 c0, 00545778 f4 0e e2 23, 0054577c 5f b3 a8 25, 00545780 33 81 4c 08, 00545784 97 d2 55 9f, 00545788 2d 4c 7b b1, and 0054578c B17B4C2Dh. A red box also highlights a line in the decompiled code: `_DAT_0054c7c1 = DAT_00545768;`

IP in formato int

Vediamo quindi un primo script specifico per questo sample, che poi verrà generalizzato per supportare i vari sample. In questo caso ho effettuato una regex sullo specifico move nella funzione di Config Builder per ottenere l'indirizzo specifico che contiene i diversi IP del C&C.

```

import pefile, ipaddress, binascii, re, struct

pe = None
imageBase = None

def GetRVA(va):
    return pe.get_offset_from_rva(va - imageBase)

def GetVA(raw):
    return imageBase + pe.get_rva_from_offset(raw)

def main():

    global pe, imageBase

    filename = "sample3"

    with open(filename, 'rb') as sample:
        data = bytearray(sample.read())

    pe = pefile.PE(filename)
    imageBase = pe.OPTIONAL_HEADER.ImageBase

    copy_operation = b'\xa1\x68\x57\x54\x00'

    for m in re.compile(copy_operation).finditer(data):
        addrStart = int(hex(struct.unpack("<L", data[m.start() + 1:m.start() + 1 +
4])[0]), 16)

        for i in range(10):

            start = int(hex(addrStart + i*4),16)
            end = int(hex(addrStart + (i+1)*4),16)

            ip = binascii.hexlify(data[GetRVA(start):GetRVA(end)])

            print(str(ipaddress.IPv4Address(int(ip, 16))))

if __name__ == "__main__":
    main()

```

Lo script ci permette di ottenere i diversi IP utilizzati dal malware come C&C:

243.127.43.6
64.126.175.2
130.15.230.152
74.99.136.192
244.14.226.35
95.179.168.37
51.129.76.8
151.210.85.159
45.76.123.177
75.57.14.121

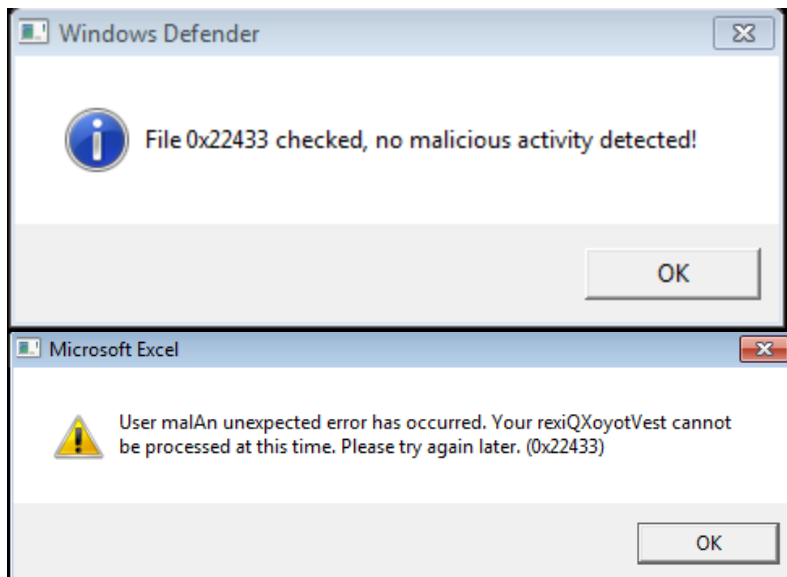
Analisi Loader Danabot

Nel secondo sample (MD5: 611c2bf7aa7bb62e90f3a92f3682c0b5) abbiamo un VBS script molto offuscato:

```
Function ReadText(path)::Const ForReading=1::Set OyuajDIIy=objgiVksCc.OpenTextFile(path, ForReading)::ReadText=OyuajDIIy.ReadAll::OyuajDIIy.Close::End function:::Function GetReportName UFRvGIZL,ubZSSU1XpVZ1V1RSpyNoP,ubZSSU1XpVZ1V1RSpyNoP,ubZSSU1XpVZ1V1RSpyNoP,mKcTtTtoSErwdFp1DV1pmbFKJX,zbZSSU1XpVZ1V1RSpyNoP,uDnnvhMKDCNlZQVI,mMgGwhWoospvghGuzQz,xKcTtTtoSErwdFp1DV1pmbFKJX,cdF cSYHnFXbRsmHujXoqVYgAmdV,udfPnREUHFkEdbbh,kxJAYEdUhbQKOGZTtNgqJnd,rTQ1WFEAeYUFRvGIZL,nKcTtTtoSErwdFp1DV1pmbFKJX,yMgGwhWoospvghGuzQz,oTQ1WFEAeYUFRvGIZL,zbZSSU1XpVZ1V1RSpyNoP,yMgGwhWoospvghGuzQz, pDnnvhMKDCNlZQVI,kDnnvhMKDCNlZQVI,hxJAYEdUhbQKOGZTtNgqJnd,xDnnvhMKDCNlZQVI,1KkgEaKMcSMWZS,hdfPnREUHFkEdbbh,sKcTtTtoSErwdFp1DV1pmbFKJX,rDnnvhMKDCNlZQVI,fcSYHnFXbRsmHujXoqVYgAmdV,hDnnvhMKDCNlZQVI, sKcTtTtoSErwdFp1DV1pmbFKJX,ydfPnREUHFkEdbbh,tCSYHnFXbRsmHujXoqVYgAmdV,jbZSSU1XpVZ1V1RSpyNoP,vTQ1WFEAeYUFRvGIZL,hMgGwhWoospvghGuzQz,aTQ1WFEAeYUFRvGIZL,eKcTtTtoSErwdFp1DV1pmbFKJX,rbZSSU1XpV TtTtoSErwdFp1DV1pmbFKJX,cdFpNREUHFkEdbbh,oTQ1WFEAeYUFRvGIZL,aKcTtTtoSErwdFp1DV1pmbFKJX,gTQ1WFEAeYUFRvGIZL,iUYnIFVcIF1C1L10zcQLTnzId,bcSYHnFXbRsmHujXoqVYgAmdV,iUYnIFVcIF1C1L10zcQLTnzId,jKkgEaKMcSM ynoP,yMgGwhWoospvghGuzQz,tKcTtTtoSErwdFp1DV1pmbFKJX,1KkgEaKMcSMWZS,cMgGwhWoospvghGuzQz,vbZSSU1XpVZ1V1RSpyNoP,nKkgEaKMcSMWZS,ebZSSU1XpVZ1V1RSpyNoP,nKkgEaKMcSMWZS,uTQ1WFEAeYUFRvGIZL,sbZSSU1 rwdFp1DV1pmbFKJX,rbZSSU1XpVZ1V1RSpyNoP,dfPnREUHFkEdbbh,eTQ1WFEAeYUFRvGIZL):::Function cZyYqUFXQu()::::cZyYqUFXQu=Array(hcSYHnFXbRsmHujXoqVYgAmdV,gbZSSU1XpVZ1V1RSpyNoP,1DnnvhMKDCNlZQVI,bKcTtTt bRsmHujXoqVYgAmdV,tbZSSU1XpVZ1V1RSpyNoP,kTQ1WFEAeYUFRvGIZL,rKcTtTtoSErwdFp1DV1pmbFKJX,vxJAYEdUhbQKOGZTtNgqJnd,iKcTtTtoSErwdFp1DV1pmbFKJX,1KkgEaKMcSMWZS,iKcTtTtoSErwdFp1DV1pmbFKJX,tMgGwhWoospv gS,iKcTtTtoSErwdFp1DV1pmbFKJX,mMgGwhWoospvghGuzQz,rDnnvhMKDCNlZQVI,iKcTtTtoSErwdFp1DV1pmbFKJX,rbZSSU1XpVZ1V1RSpyNoP,yTQ1WFEAeYUFRvGIZL,oMgGwhWoospvghGuzQz,cbZSSU1XpVZ1V1RSpyNoP,1bZSSU1XpVZ1V1RS yYnIFVcIF1C1L10zcQLTnzId,iUYnIFVcIF1C1L10zcQLTnzId,1bZSSU1XpVZ1V1RSpyNoP,ucSYHnFXbRsmHujXoqVYgAmdV,uKcTtTtoSErwdFp1DV1pmbFKJX,vTQ1WFEAeYUFRvGIZL,xMgGwhWoospvghGuzQz,bdfPnREUHFkEdbbh,iKcTtTtoSE rwdFp1DV1pmbFKJX,xUYnIFVcIF1C1L10zcQLTnzId,udfPnREUHFkEdbbh,qMgGwhWoospvghGuzQz,hbZSSU1XpVZ1V1RSpyNoP,xUYnIFVcIF1C1L10zcQLTnzId,zKcTtTtoSErwdFp1DV1pmbFKJX,aTQ1WFEAeYUFRvGIZL,xCSYHnFXbRsmHujXoqVYgAmdV,ednn EaeYUFRvGIZL,rKcTtTtoSErwdFp1DV1pmbFKJX,eKkgEaKMcSMWZS,rKkgEaKMcSMWZS,tbZSSU1XpVZ1V1RSpyNoP,pUYnIFVcIF1C1L10zcQLTnzId,wxJAYEdUhbQKOGZTtNgqJnd,jdfPnREUHFkEdbbh,vdfPnREUHFkEdbbh,nTQ1WFEAeYUFRvGI duhbQKOGZTtNgqJnd,udfPnREUHFkEdbbh,qKkgEaKMcSMWZS,mKkgEaKMcSMWZS,hxJAYEdUhbQKOGZTtNgqJnd,yTQ1WFEAeYUFRvGIZL,jMgGwhWoospvghGuzQz,iUYnIFVcIF1C1L10zcQLTnzId,akkgEaKMcSMWZS,1xJAYEdUhbQKOGZTtNgqJ IFC1L10zcQLTnzId,uDnnvhMKDCNlZQVI,qbZSSU1XpVZ1V1RSpyNoP,1dfPnREUHFkEdbbh,vTQ1WFEAeYUFRvGIZL,kTQ1WFEAeYUFRvGIZL,cMgGwhWoospvghGuzQz,dKkgEaKMcSMWZS,ocSYHnFXbRsmHujXoqVYgAmdV,1dfPnREUHFkEdb UhbQKOGZTtNgqJnd,udfPnREUHFkEdbbh,eKcTtTtoSErwdFp1DV1pmbFKJX,wKkgEaKMcSMWZS,sMgGwhWoospvghGuzQz,cSYHnFXbRsmHujXoqVYgAmdV,tKkgEaKMcSMWZS,iUYnIFVcIF1C1L10zcQLTnzId,1DnnvhMKDCNlZQVI,pKcTtTtoSE NlZQVI,fxJAYEdUhbQKOGZTtNgqJnd,kKcTtTtoSErwdFp1DV1pmbFKJX,bMgGwhWoospvghGuzQz,eTQ1WFEAeYUFRvGIZL,pcSYHnFXbRsmHujXoqVYgAmdV,eMgGwhWoospvghGuzQz,uYnIFVcIF1C1L10zcQLTnzId,zcSYHnFXbRsmHujXoqVYgAmd Jnd,kTQ1WFEAeYUFRvGIZL,rxJAYEdUhbQKOGZTtNgqJnd,1DnnvhMKDCNlZQVI,yxJAYEdUhbQKOGZTtNgqJnd,gKkgEaKMcSMWZS,cbZSSU1XpVZ1V1RSpyNoP,dKcTtTtoSErwdFp1DV1pmbFKJX,oKcTtTtoSErwdFp1DV1pmbFKJX,yxJAYEdUhbQKOG L10zcQLTnzId,wKkgEaKMcSMWZS,pxJAYEdUhbQKOGZTtNgqJnd,oDnnvhMKDCNlZQVI,rcSYHnFXbRsmHujXoqVYgAmdV,ydfPnREUHFkEdbbh,bBZSSU1XpVZ1V1RSpyNoP,rKcTtTtoSErwdFp1DV1pmbFKJX,kTQ1WFEA RSpynoP,uYnIFVcIF1C1L10zcQLTnzId,hcSYHnFXbRsmHujXoqVYgAmdV,iKcTtTtoSErwdFp1DV1pmbFKJX,1KcTtTtoSErwdFp1DV1pmbFKJX,scSYHnFXbRsmHujXoqVYgAmdV,dfPnREUHFkEdbbh,uYnIFVcIF1C1L10zcQLTnzId,eDnnvhMKDCNl bbb,mDnnvhMKDCNlZQVI,gKcTtTtoSErwdFp1DV1pmbFKJX,yKcTtTtoSErwdFp1DV1pmbFKJX,uYnIFVcIF1C1L10zcQLTnzId,csYHnFXbRsmHujXoqVYgAmdV,exJAYEdUhbQKOGZTtNgqJnd,sMgGwhWoospvghGuzQz,fDnnvhMKDCNlZQVI,jMgG CnlZQVI,iTQ1WFEAeYUFRvGIZL,qxJAYEdUhbQKOGZTtNgqJnd,vbZSSU1XpVZ1V1RSpyNoP,nKcTtTtoSErwdFp1DV1pmbFKJX,aTQ1WFEAeYUFRvGIZL,uKcTtTtoSErwdFp1DV1pmbFKJX,xDnnvhMKDCNlZQVI,abZSSU1XpVZ1V1RSpyNoP,bMgGwhN ZSSU1XpVZ1V1RSpyNoP,iKcTtTtoSErwdFp1DV1pmbFKJX,rbZSSU1XpVZ1V1RSpyNoP,udfPnREUHFkEdbbh,iCSYHnFXbRsmHujXoqVYgAmdV,rMgGwhWoospvghGuzQz,uKkgEaKMcSMWZS,gMgGwhWoospvghGuzQz,qdfPnREUHFkEdbbh,qKcTt TnzId,mdfPnREUHFkEdbbh,mMgGwhWoospvghGuzQz,sbZSSU1XpVZ1V1RSpyNoP,wdfPnREUHFkEdbbh,iMgGwhWoospvghGuzQz,bMgGwhWoospvghGuzQz,rMgGwhWoospvghGuzQz,uKcTtTtoSErwdFp1DV1pmbFKJX,nKcTtTtoSErwdFp1DV1pmbF NgqJnd,kUYnIFVcIF1C1L10zcQLTnzId,kDnnvhMKDCNlZQVI,bcSYHnFXbRsmHujXoqVYgAmdV,qDnnvhMKDCNlZQVI,pbZSSU1XpVZ1V1RSpyNoP,mKcTtTtoSErwdFp1DV1pmbFKJX,ucSYHnFXbRsmHujXoqVYgAmdV,gcSYHnFXbRsmHujXoqVYgAmd YnIFVcIF1C1L10zcQLTnzId,rbZSSU1XpVZ1V1RSpyNoP,kTQ1WFEAeYUFRvGIZL,iMgGwhWoospvghGuzQz,1xJAYEdUhbQKOGZTtNgqJnd,tbZSSU1XpVZ1V1RSpyNoP,gbZSSU1XpVZ1V1RSpyNoP,vTQ1WFEAeYUFRvGIZL,gDnnvhMKDCNlZQVI,rKc
```

Primo VBS offuscato

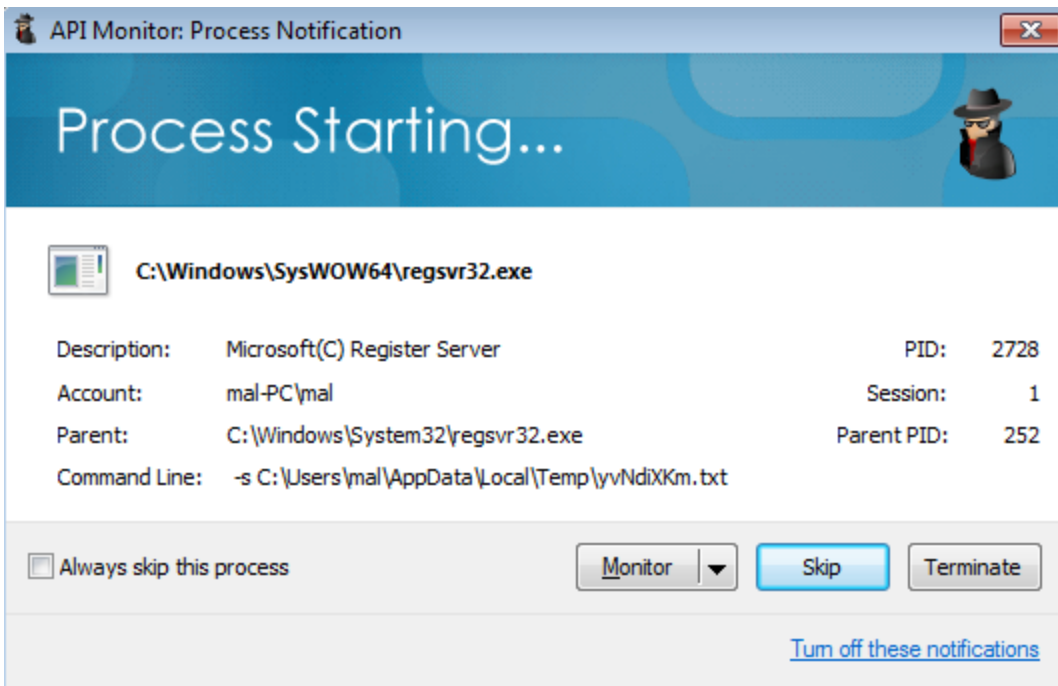
Una volta avviato si ottengono due messaggi:



Primo messaggio dello script

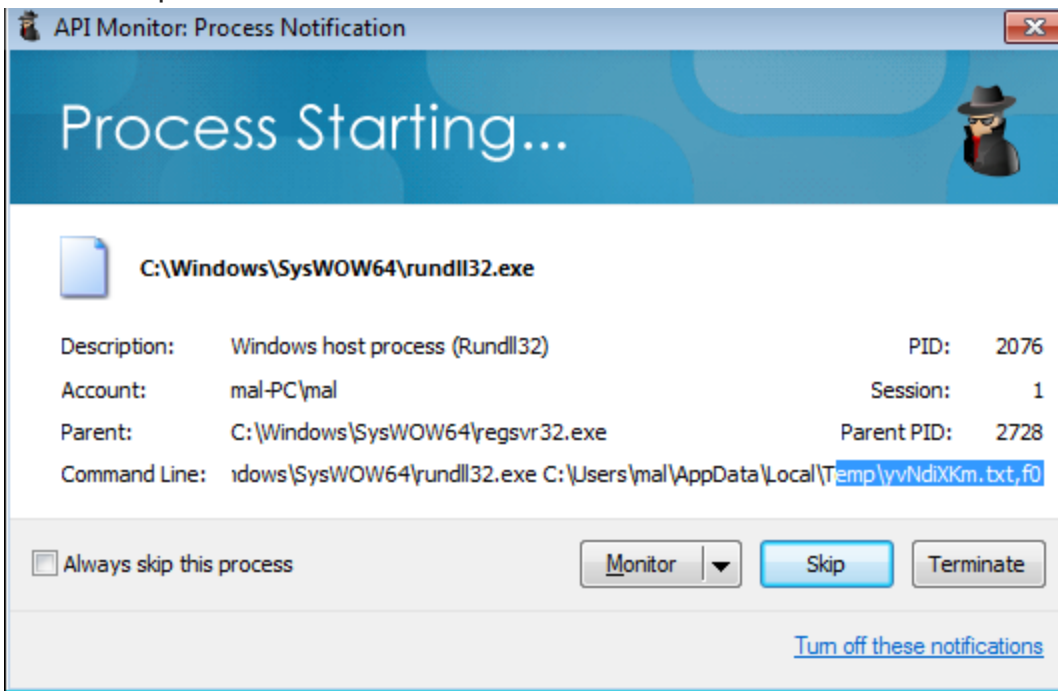
Secondo messaggio dello script

Lo script salva la DLL **yvNdiXKm.txt** in TEMP e avvia la funzione F0, che in realtà non viene esportata dalla DLL e quindi viene avviato l'entry:



Il VBS estrae la

DLL in temp e la avvia



Rundll32 avvia la

funzione F0 della DLL (entry)

Effettuiamo una prima analisi del sample con Resource Hacker, PE Studio e Detect It Easy:

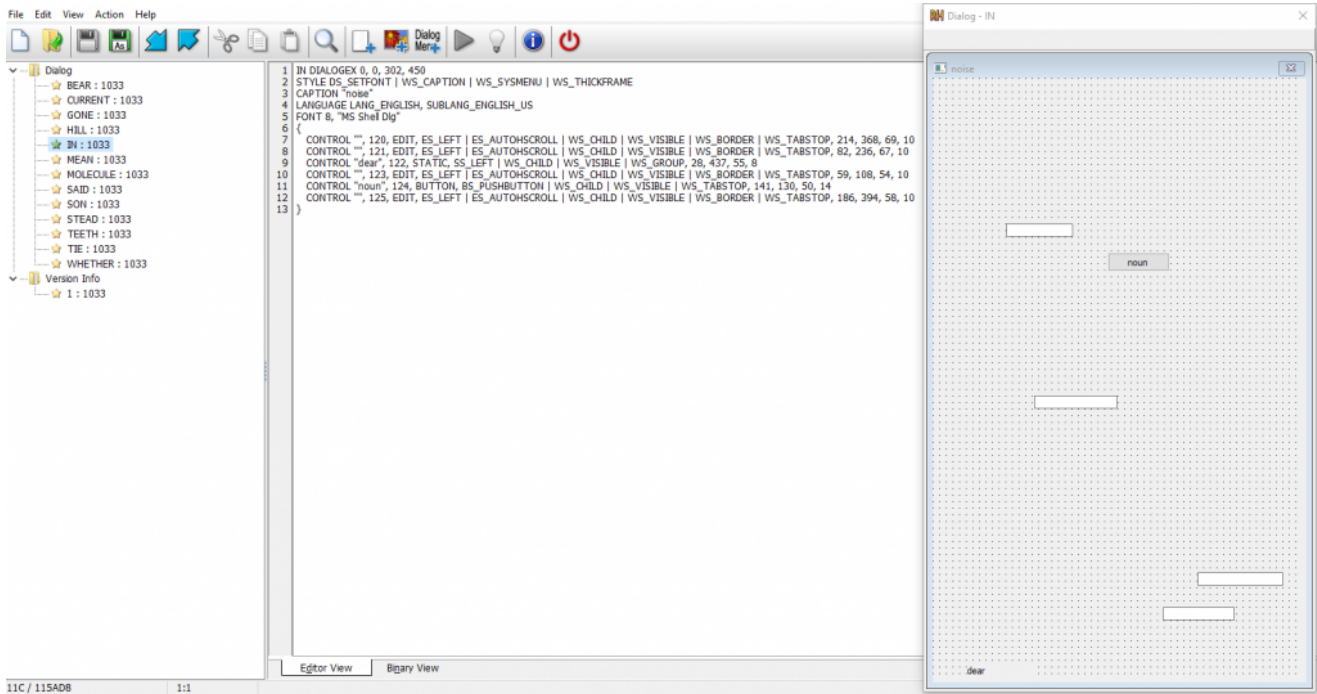
Type	Total	Status
PE32	6.63218	packed(82%)

Entropia	Bytes			
Regions				
Offset	Size	Entropia	Status	
00000000	00001000	0.80597	not packed	PE Header
00001000	000d8000	6.75698	packed	Section(0)['.text']
000d9000	0002e000	6.60150	packed	Section(1)['.rdata']
00107000	0000e000	5.99332	not packed	Section(2)['.data']
00115000	00002000	2.25154	not packed	Section(3)['.rsrc']
00117000	00008000	1.14783	not packed	Section(4)['.reloc']

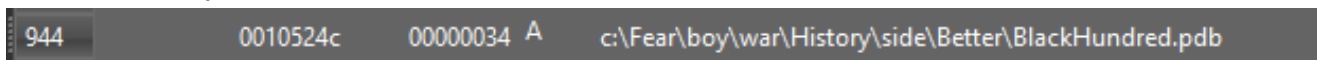
Il sample risulta packed

ATT&CK Tactic	ATT&CK Technique
DISCOVERY	File and Directory Discovery:: T1083
EXECUTION	Shared Modules:: T1129
MBC Objective	MBC Behavior
PROCESS	Allocate Thread Local Storage:: [C0040]
	Terminate Process:: [C0018]
CAPABILITY	NAMESPACE
contains PDB path	executable/pe/pdb
contain a resource (.rsrc) section	executable/pe/section/rsrc
get common file path	host-interaction/file-system
allocate thread local storage	host-interaction/process
terminate process	host-interaction/process/terminate
link many functions at runtime	linking/runtime-linking

Le pochi capability trovate da capa confermano sia un packer

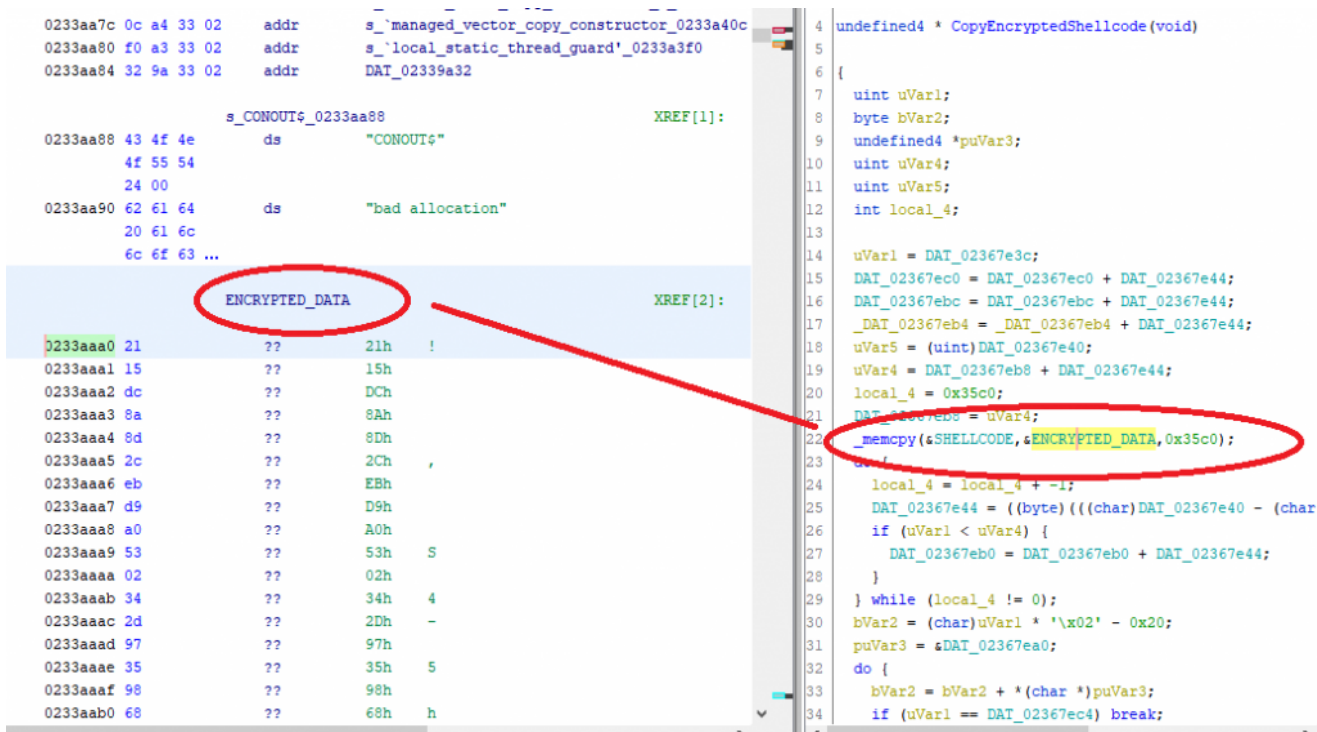


La DLL è composta da diversi form



Path con riferimenti alla guerra e Russia

Analizzando questa DLL non trovo le funzioni socket viste in precedenza essendo il packer; metto come breakpoint le funzioni **VirtualProtect**, **VirtualAlloc** e **CreateThread**. Viene raggiunto VirtualProtect e all'indirizzo **base_address + 0x115f50** è presente la shellcode, che viene copiata dall'indirizzo **base_address + 0xdaaa0**:



Copia della shellcode cifrata

In particolare, la decifratura della shellcode è molto semplice, infatti nonostante siano presente molte operazioni, viene solo modificata da una operazione, che aggiunge per ogni 4 byte il valore **0x1828308** e questa somma viene fatta per **0x0135910C** volte:

```

    uVar2 = 35_VALUE * 2 - 3;
}
E4E91D0A_VALUE = 35_VALUE + E4E91D0A_VALUE * -0x3377;
}
if (DAT_03fe7ec8 - _DAT_03fe7eac == 0xefef) {
    uVar2 = E4E91D0A_VALUE + DAT_03fe7e34 * 0x5f;
}
else {
    DAT_03fe7e34 = E4E91D0A_VALUE + DAT_03fe7e34 * -0x3377;
}
VALUE_ASSIGNED_TO_SHELLCODE = *pSourceShellcode + 0x1828308;
E4E91D0A_VALUE = (uVar2 - (uVar2 & 0xff)) + 7;
*pSourceShellcode = VALUE_ASSIGNED_TO_SHELLCODE;
if (DAT_03fe7ea4 < E4E91D0A_VALUE) {
    _DAT_03fe7eac = DAT_03fe7e34;
}
pSourceShellcode = pSourceShellcode + 1;
iVar1 = E4E91D0A_VALUE - 27444_VALUE;
counter = counter + -1;
} while (counter != 0);
iVar3 = 35_VALUE - (DAT_03fe7e34 & 0xff);
27444_VALUE = iVar3 + 0x26;
DAT_03fe7e44 = iVar1 + 0x21;
DAT_03fe7e9c = iVar3 + 7;
return;
}

```

Decifratura della shellcode

The screenshot shows a debugger window with assembly code on the left and registers/stack on the right. A red box highlights the instruction `push dword ptr ds:[496F5A6]` with a label "pointer to shellcode address". Another red box highlights the `ret` instruction. A red arrow points from the `ret` instruction to the `push` instruction. The register window shows `EIP` at `04028846` and `EAX` at `0496F5A6`. The stack window shows `Predefinito (stdcall)` with addresses `0000F7DC`, `FFFC546`, `0000F8AC`, `76A71880`, and `DBEA7FF9`.

Esecuzione della shellcode attraverso RET

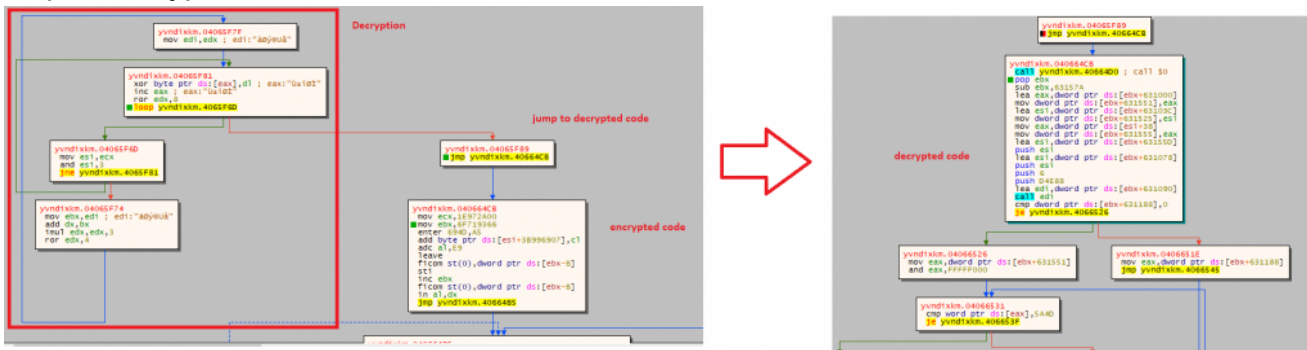
Dopo la decifratura viene avviata la shellcode che si occupa di decifrare la restante shellcode:

```

02365F50 81C1 D9B1EDD8 | add ecx,D8EDB1D9
02365F56 E8 00000000 | call yvndixkm.2365F5B
02365F5B 5B | pop ebx
02365F5F BF C8A66453 | mov edi,5364A6C8
02365F64 B9 64090000 | mov ecx,964
02365F69 89FA | mov edx,edi
02365F68 31DB | xor ebx,ebx
02365F6D 89CE | mov esi,ecx
02365F6F 83E6 03 | and esi,3
02365F72 75 0D | jne yvndixkm.2365F81
02365F74 89F8 | mov ebx,edi
02365F76 66:01DA | add dx,bx
02365F79 68D2 03 | imul edx,edx,3
02365F7C C1CA 04 | ror edx,4
02365F7F 89D7 | mov edi,edx
02365F81 3010 | xor byte ptr ds:[eax],dl
02365F83 40 | inc eax
02365F84 C1CA 08 | ror edx,8
02365F87 ^ E4 | loop yvndixkm.2365F6D
02365F89 v E9 3D050000 | jmp yvndixkm.23664CB
02365F8F FF |
02365F8F FF |
02365F90 FF |
02365F91 FFA2 09000030 | jmp dword ptr ds:[edx+30000009]
02365F97 4F | dec edi
02365F98 0C 00 | or al,0
02365F9A 0087 2F2993A2 | add byte ptr ds:[edi-5D6CD6D1],al
02365FA0 BC 87CA5763 | mov esp,6357CA87
02365FA5 v E9 709CC5CB | jmp CDFBFC1A
02365FAA 27 | daa

```

loop di decryption che decifra la restante shellcode e le funzioni della nuova DLL



Control Flow prima e dopo la decifatura

Successivamente avviene qualcosa di molto interessante, la shellcode cambia i i permessi della varie sezioni con **VirtualProtect** in scrittura (0x4) riscrivendo parte di queste e quella in .data in execution (0x40); questa tecnica si chiama **Reflective DLL Loading** e consiste nel caricare direttamente la DLL dalla memoria senza passare dal disco.

021A0000	00001000	yvndixkm.txt			
021A1000	000D8000	".text"	Executable code	IMG	-R---
02279000	0002E000	".rdata"	Read-only initialized data	IMG	ER---
022A7000	00919000	".data"	Initialized data	IMG	-RW--
028C0000	00002000	".rsrc"	Resources	IMG	-R---
028C2000	00008000	".reloc"	Base relocations	IMG	-R---
02DB0000	00003000			PRV	-RW--

Permessi iniziali della DLL

022D0000	00001000	yvndixkm.txt			
022D1000	000D8000	".text"	Executable code	IMG	-R---
023A9000	0002E000	".rdata"	Read-only initialized data	IMG	-RW--
023D7000	00919000	".data"	Initialized data	IMG	-RW--
02CF0000	00002000	".rsrc"	Resources	IMG	-R---
02CF2000	00008000	".reloc"	Base relocations	IMG	-R---
02E50000	00003000			PRV	-RW--

Cambio dei permessi in scrittura

022D0000	00001000	yvndixkm.txt			
022D1000	000D8000	".text"	Executable code	IMG	-R---
023A9000	0002E000	".rdata"	Read-only initialized data	IMG	ER---
023D7000	00919000	".data"	Initialized data	IMG	ER---
02CF0000	00002000	".rsrc"	Resources	IMG	-R---
02CF2000	00008000	".reloc"	Base relocations	IMG	-R---
02E50000	00003000			PRV	-RW--

Cambio dei permessi in esecuzione

```

EAX 02420000 "Embarcadero Delphi for win32 compiler version 32.0 (25.0.26309.314"
EBX FFB7F000
ECX 00001000
EDX 001B0559
EBP 002EF7C8
ESP 002EF784 "\\t\t\x1B"
ESI 022D0310 ".rdata"
EDI 00000002

EIP 769143CE <kernel32.VirtualProtect>

EFLAGS 00000344
ZF 1 PF 1 AF 0
OF 0 SF 0 DF 0
CF 0 TF 1 IF 1

```

Riferimento a Delphi

Successivamente viene eseguito **CreateThread**, passando come indirizzo un indirizzo presente in .data, ricordiamo decifrato in precedenza dalla shellcode.

```

Nascondi FPU

EAX 0032F940
EBX 00000000
ECX 0032F914
EDX FFFF781D
EBP 0032F944 "%u2"
ESP 0032F8C4 "»rD\x02"
ESI 001E021C
EDI 00000000

EIP 769124E4 <kernel32.CreateThread>

EFLAGS 00200304
ZF 0 PF 1 AF 0
OF 0 SF 0 DF 0
CF 0 TF 1 IF 1

LastError 00000000 (ERROR_SUCCESS)
LastStatus C0000139 (STATUS_ENTRYPOINT_NOT_FOUND)

GS 002B FS 0053
ES 002B DS 002B
CS 0023 SS 002B

Predefinito (stdcall) 5 Sbloccato
1: [esp+4] 00000000
2: [esp+8] 00000000
3: [esp+C] 02436184 "U<` ('"
4: [esp+10] 00000000
5: [esp+14] 00000000

```

Creazione del thread con la

funzione decifrata dalla shellcode

Inseriamo nuovamente come breakpoint la funzione connect e otteniamo la funzione dove si effettua la connessione al C&C:

023093D8	7D 16	ige yvndixkm.23093F0
023093DA	8D45 80	lea eax,dword ptr ss:[ebp-80]
023093DD	50	push eax
023093DE	8D45 C8	lea eax,dword ptr ss:[ebp-38]
023093E1	50	push eax
023093E2	8D4D 8C	lea ecx,dword ptr ss:[ebp-74]
023093E5	8D55 98	lea edx,dword ptr ss:[ebp-68]
023093E8	8D45 A8	lea eax,dword ptr ss:[ebp-58]
023093EB	E8 C05FFBFF	call yvndixkm.22BF3B0
023093F0	8845 C0	mov eax,dword ptr ss:[ebp-40]
023093F3	2B45 B4	sub eax,dword ptr ss:[ebp-4C]
023093F6	8945 C8	mov dword ptr ss:[ebp-38],eax
023093F9	66:BA BB01	mov dx,1BB
023093FD	8845 F4	mov eax,dword ptr ss:[ebp-C]
02309400	E8 8BC1FEFF	call yvndixkm.22F5590
02309405	8945 E8	mov dword ptr ss:[ebp-18],eax
EIP 02309408	6985 74FFFFFF 3B020000	imul eax,dword ptr ss:[ebp-8C],23B
02309412	8945 80	mov dword ptr ss:[ebp-80],eax

Anche questa volta vediamo che il config inizia con **3C** e termina con **4E** ma il terzo e il quarto byte sono differenti rispetto al config precedente:

Indirizz	Hex	ASCII
025287AD	3C 00 00 00 C1 FD 00 00 00 00 00 00 FD FD 00 00	...Áí.....ýí
025287BD	00 00 00 00 B5 3F 2C C2 CF 94 53 6C 2D 4D 28 47	...µ?,Áí.SI-M(G
025287CD	57 73 8A A9 18 E5 30 07 74 6F CE 1B 2D C4 8F CB	ws. @.á.ó.tóí.-Á.É
025287DD	DA 41 03 C7 83 3B 6E BA 71 51 61 60 00 00 00 4E	ÚA.Ç.;n°qqa'...N
025287ED	AA 88 00 7C 3D D5 00 34 3B D5 00 AC 3D D5 00 24	..Ï.Ï.Ï.Ï.Ï.Ï.Ï.Ï
025287FD	9D D5 00 DC 9D D5 00 F4 9D D5 00 0C 9E D5 00 24	.Ï.Ï.Ï.Ï.Ï.Ï.Ï.Ï
0252880D	9E D5 00 3C 9E D5 00 54 9E D5 00 6C 9E D5 00 84	.Ï.<.Ï.T.Ï.Ï.Ï.
0252881D	9E D5 00 9C 9E D5 00 B4 9E D5 00 CC 9E D5 00 E4	.Ï...Ï.Ï.Ï.Ï.Ï.Ï
0252882D	9E D5 00 FC 9E D5 00 14 9F D5 00 2C 9F D5 00 44	.Ï.Ï.Ï.Ï.Ï.Ï.Ï.Ï
0252883D	9F D5 00 5C 9F D5 00 74 9F D5 00 8C 9F D5 00 A4	.Ï.\.Ï.t.Ï.Ï.Ï.Ï
0252884D	9F D5 00 BC 9F D5 00 D4 9F D5 00 E8 A0 D5 00 00	.Ï.¼.Ï.Ï.Ï.Ï.è Ï..
0252885D	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

Config ottenuto con il debugger

Ho iniziato quindi a tracciare le diverse VirtualAlloc per capire dove effettivamente fosse il config; a un certo punto viene allocata una zona di memoria e il suo indirizzo salvato in **[ebx+631549]** e questo puntatore viene utilizzato per effettuare diverse operazioni sul PE per deoffuscarlo:

025C06A4	61	popad
025C06A5	8B4E 08	mov ecx,dword ptr ds:[esi+8]
025C06A8	034E 0C	add ecx,dword ptr ds:[esi+C]
025C06AB	034E 04	add ecx,dword ptr ds:[esi+4]
025C06AE	6A 40	push 40
025C06B0	68 00300000	push 3000
025C06B5	51	push ecx
025C06B6	6A 00	push 0
025C06B8	FF93 71156300	call dword ptr ds:[ebx+631571]
025C06BE	8983 49156300	mov dword ptr ds:[ebx+631549],eax

alloca la zona di memoria per il PE

025C06BE	8983 49156300	mov dword ptr ds:[ebx+631549],eax
025C06C4	60	pushad
025C06C5	FFB3 4D156300	push dword ptr ds:[ebx+63154D]
025C06CB	50	push eax
025C06CC	FFB3 31156300	push dword ptr ds:[ebx+631531]
025C06D2	FFB3 88116300	push dword ptr ds:[ebx+631188]
025C06D8	E8 58FBFFFF	call <ExtractPEto631549Pointer>
025C06DD	83C4 10	add esp,10
025C06E0	61	popad
025C06E1	68 00800000	push 8000
025C06E6	6A 00	push 0
025C06E8	FFB3 4D156300	push dword ptr ds:[ebx+63154D]
025C06EE	FF93 5D156300	call dword ptr ds:[ebx+63155D]
025C06F4	8883 49156300	mov eax,dword ptr ds:[ebx+631549]
025C06FA	8983 4D156300	mov dword ptr ds:[ebx+63154D],eax
025C0700	8883 55156300	mov eax,dword ptr ds:[ebx+631555]
025C0706	83E0 04	and eax,4
025C0709	85C0	test eax,eax
025C0708	74 1B	je 25C0728
025C070D	60	pushad
025C070E	8D56 1C	lea edx,dword ptr ds:[esi+1C]
025C0711	8D4E 2A	lea ecx,dword ptr ds:[esi+2A]
025C0714	51	push ecx
025C0715	52	push edx
025C0716	FF76 04	push dword ptr ds:[esi+4]
025C0719	FFB3 49156300	push dword ptr ds:[ebx+631549]
025C071F	E8 B9FBFFFF	call <UnkOperationPE>
025C0724	83C4 10	add esp,10
025C0727	61	popad
025C0728	60	pushad
025C0729	8883 55156300	mov eax,dword ptr ds:[ebx+631555]
025C072F	83E0 02	and eax,2
025C0732	85C0	test eax,eax
025C0734	74 05	je 25C0738
025C0736	8B46 0C	mov eax,dword ptr ds:[esi+C]
025C0739	EB 03	jmp 25C073E
025C073B	8B46 08	mov eax,dword ptr ds:[esi+8]
025C073E	FF76 18	push dword ptr ds:[esi+18]
025C0741	50	push eax
025C0742	FFB3 49156300	push dword ptr ds:[ebx+631549]
025C0748	E8 60FBFFFF	call <DecryptPE>
025C074D	61	popad
025C074E	8883 55156300	mov eax,dword ptr ds:[ebx+631555]
025C0754	83E0 02	and eax,2
025C0757	85C0	test eax,eax
025C0759	74 18	je 25C0773
025C075B	888B 49156300	mov edi,dword ptr ds:[ebx+631549]
025C0761	89FA	mov edx,edi
025C0763	0356 0C	add edx,dword ptr ds:[esi+C]
025C0766	8993 49156300	mov dword ptr ds:[ebx+631549],edx
025C076C	52	push edx
025C076D	57	push edi
025C076E	E8 19FAFFFF	call <DecompressPE>
025C0773	888B 49156300	mov edi,dword ptr ds:[ebx+631549]

Pointer to Obfuscated PE

Deobfuscation del PE tramite decifratura e decompressione

La prima funzione che viene eseguita estrae il PE offuscato copiando il contenuto in EDI; il PE offuscato si trova nell'indirizzo di memoria **base_address + 0xbecc**.

00730288	57	push eax
0073028C	51	push ecx
0073028D	8B7424 20	mov esi,dword ptr ss:[esp+20]
00730291	8B7C24 1C	mov edi,dword ptr ss:[esp+1C]
00730295	8B4C24 34	mov ecx,dword ptr ss:[esp+34]
00730299	F3:A4	rep movsb
0073029B	59	pop ecx
0073029C	5F	pop edi
0073029D	5E	pop esi
0073029E	01E8	add eax,ebp
007302A0	8D342A	lea esi,dword ptr ds:[edx+ebp]
007302A3	EB 9F	jmp 730244
007302A5	5F	pop edi
007302A6	5E	pop esi
007302A7	5D	pop ebp
007302A8	5B	pop ebx
007302A9	83C4 08	add esp,8
007302AC	C3	ret
007302AD	55	push ebp
007302AE	89E5	mov ebp,esp
007302B0	8845 08	mov eax,dword ptr ss:[ebp+8]
007302B3	884D 0C	mov ecx,dword ptr ss:[ebp+C]
007302B6	8855 10	mov edx,dword ptr ss:[ebp+10]
007302B9	31DB	xor ebx,ebx
007302BB	89CE	mov esi,ecx
007302BD	83E6 03	and esi,3
007302C0	75 0F	jne 7302D1
007302C2	885D 10	mov ebx,dword ptr ss:[ebp+10]
007302C5	66:01DA	add dx,bx
007302C8	68D2 03	imul edx,edx,3
007302CB	C1CA 04	ror edx,4
007302CE	8955 10	mov dword ptr ss:[ebp+10],edx
007302D1	3010	xor byte ptr ds:[eax],dl
007302D3	40	inc eax
007302D4	C1CA 08	ror edx,8
007302D7	E2 E2	loop 7302BB
007302D9	C9	leave

source addr
dest add
num movsb

Indirizz	Hex	ASCII
000DF7E0	00 00 CA 04 1C BE 0E 04 DD 06 73 00 00 00 0E 04	. .É.İ%.ÿ.s....
000DF7F0	00 A0 A2 00 00 00 CA 04 5B 09 00 00 00 00 74 00	. c...É.[...t.
000DF800	3C 00 73 00 40 F8 0D 00 1C F8 0D 00 00 F0 0F 00	<.s.o...o...ö.
000DF810	00 00 CA 04 00 00 D5 B2 00 00 CA 04 DC E7 0D 00	.É...ö...É.ü+
000DF820	46 C5 FF FF AC F8 0D 00 80 1B A7 76 B2 E5 38 C2	FÄÿÿ-ø...sv=ä8À
000DF830	FE FF FF FF D5 03 00 00 08 00 00 00 91 34 00 00	pyÿÿÿ...4.
000DF840	22 02 A3 05 DD 15 0E 04 00 00 00 00 50 5F 1F 04	".É.ÿ...P...
000DF850	60 08 C4 04 4B F6 00 00 68 F8 0D 00 45 F6 00 00	".Ä.Kö...hø..Eö..
000DF860	4B F6 00 00 00 00 00 00 DE B2 00 00 82 8E 1B 04	Kö...p...%
000DF870	00 00 00 00 01 00 00 00 BC F8 0D 00 00 00 0E 04)...%o....
000DF880	29 23 0E 04 00 00 0E 04 01 00 00 00 00 00 00 00)#...%o....
000DF890	D3 FF DE CF B3 23 0E 04 D8 F8 0D 00 00 00 00 00	ÖÿB!#...öo.
000DF8A0	01 00 00 00 00 F8 0D 00 B3 23 0E 04 20 F9 0D 00	...o...#...ü..
000DF8B0	30 3D 0E 04 BF 54 CD CB 00 00 00 00 E4 F8 0D 00	0=...¿TIE...äo..
000DF8C0	D0 23 0E 04 00 00 0E 04 96 29 8D 77 00 00 0E 04	D#...).W...#

Funzione che copia il PE offuscato da ESI in EDI

03F5BEC0	03F5 57	add ebx,dword ptr ds:[esi]
03F5BEC4	F5	crc
03F5BEC8	66:00F9	add cl,bh
03F5BED0	F9	stc
03F5BED4	F9	stc
03F5BED8	00F9	add cl,bh
03F5BEDC	E8 0000F183	CALL 87E6BDC
03F5BED8	AF	scasd
03F5BEE0	A5	movsd
03F5BEE4	EA 3A4671E8 00F9	mov far ptr ds:[edi],far ptr ds:[esi]
03F5BEE8	F9	stc
03F5BEEC	90	nop
03F5BEF0	6300	arpl word ptr ds:[eax],word ptr ds:[ebx]
03F5BEF4	8B A2AB510	mov ebx,1082AA2
03F5BEF8	F9	stc
03F5BEFC	F9	stc
03F5BF00	3111	xor dword ptr ds:[eax],dword ptr ds:[ebx]
03F5BF04	218E 29668631	and dword ptr ds:[eax],dword ptr ds:[ebx]
03F5BF08	F9	stc
03F5BF0C	20 1EFFF0F9	sub eax,F900FF1E
03F5BF10	0000	add byte ptr ds:[eax],al
03F5BF14	F9	stc
03F5BF18	0075 75	add byte ptr ss:[ebx],al
03F5BF1C	192B	sbb dword ptr ds:[edi],dword ptr ds:[ebx]
03F5BF20	6299 FED10000	bound ebx,dword ptr ds:[edi]
03F5BF24	FF	stc
03F5BF28	8000 CA	add byte ptr ds:[eax],al
03F5BF2C	44	inc esp
03F5BF30	52	push edx
03F5BF34	2ACB	sub cl,al
03F5BF38	4C	dec esp
03F5BF3C	66:E8 00FF	CALL 8E18
03F5BF40	F9	stc
03F5BF44	F9	stc
03F5BF48	E8 E800CC00	CALL 8C1C00A
03F5BF4C	CC	int3
03F5BF50	F9	stc
03F5BF54	EC	in al,dx
03F5BF58	AC	lodsb
03F5BF5C	35 3D01F900	mov eax,F9013D
03F5BF60	26	mov eax,dword ptr ds:[eax]
03F5BF64	C1B 42	rcr dword ptr ds:[eax],cl

area of memory pointed by ESI

PE Offuscato ottenuto staticamente

Questo PE compresso viene decifrato da una semplice funzione:

```

025C02AD <Decr 55          push ebp
025C02AE          89E5          mov ebp,esp
025C02B0          8B45 08       mov eax,dword ptr ss:[ebp+8]
025C02B3          8B4D 0C       mov ecx,dword ptr ss:[ebp+C]
025C02B6          8B55 10       mov edx,dword ptr ss:[ebp+10]
025C02B9          31DB         xor ebx,ebx
-> 025C02BB          89CE         mov esi,ecx
025C02BD          83E6 03       and esi,3
025C02C0          75 0F        jne 25C02D1
025C02C2          8B5D 10       mov ebx,dword ptr ss:[ebp+10]
025C02C5          66:01DA      add dx,bx
025C02C8          6BD2 03       imul edx,edx,3
025C02CB          C1CA 04       ror edx,4
025C02CE          8955 10       mov dword ptr ss:[ebp+10],edx
-> 025C02D1          3010         xor byte ptr ds:[eax],dl
025C02D3          40           inc eax
025C02D4          C1CA 08       ror edx,8
- 025C02D7          E2 E2        loop 25C02BB
025C02D9          CA           leave

```

Funzione che decifra il PE

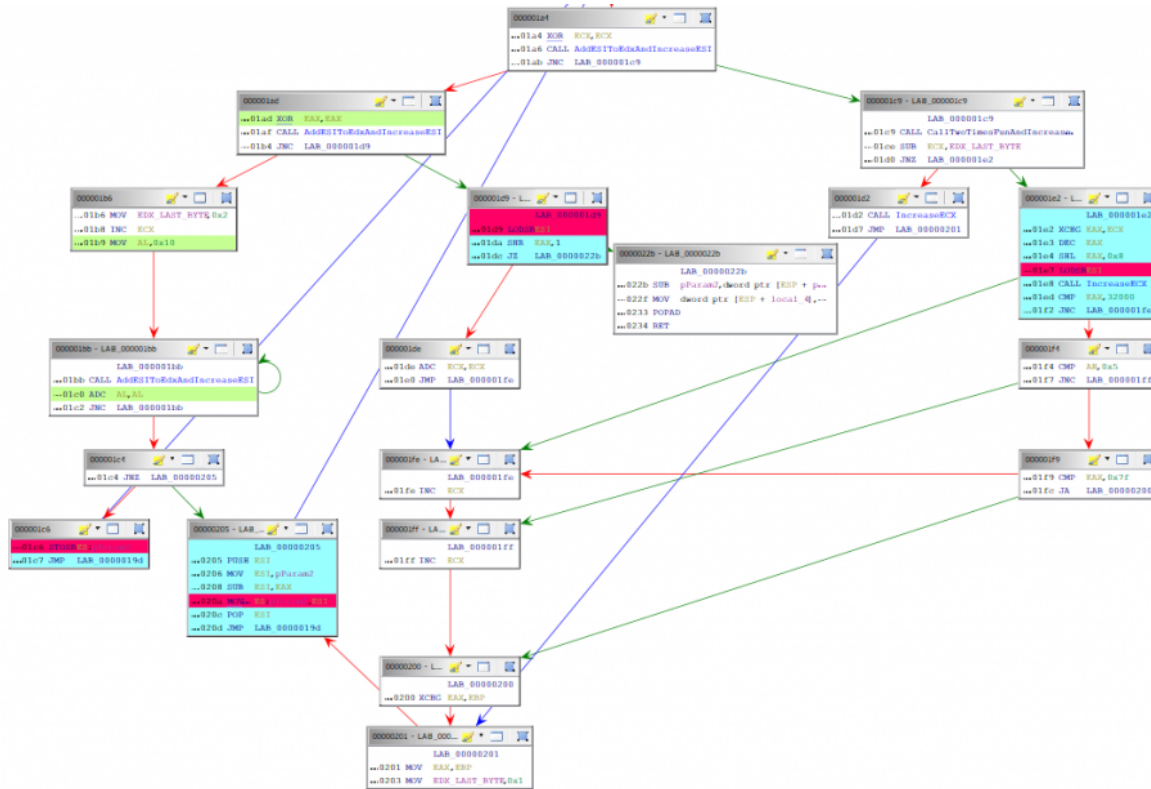
Per quanto riguarda la decompressione, la funzione prende un sottoinsieme di byte dalla zona di memoria puntata da ESI e li trasferisce nella zona di memoria puntata da EDI attraverso **movsb**, **stosb** e **lodsb**; per chi volesse approfondire come funziona il trasferimento attraverso queste istruzioni si può leggere l'ottimo articolo presente [qui](#). In particolare, il PE è compresso con **APLib** (la signature del PE è **M8Z**) e successivamente possiamo utilizzare quindi l'ottimo [tool](#) di herrcore.

```

005B0188          0000         add byte ptr ds:[eax],al
005B018A          FF03         inc dword ptr ds:[ebx]
005B018C          60           pushad
005B018D          8B7424 24     mov esi,dword ptr ss:[esp+24]
005B0191          8B7C24 28     mov edi,dword ptr ss:[esp+28]
005B0195          FC           cld
005B0196          B2 80        mov dl,80
005B0198          31DB         xor ebx,ebx
-> 005B019A          A4           movsb
005B019B          B3 02        mov bl,2
-> 005B019D          E8 6D000000 call 5B020F
-- 005B01A2          73 F6        jae 5B019A
005B01A4          31C9         xor ecx,ecx
005B01A6          E8 64000000 call 5B020F
-- 005B01AB          73 1C        jae 5B01C9
005B01AD          31C0         xor eax,eax
005B01AF          E8 5B000000 call 5B020F
-- 005B01B4          73 23        jae 5B01D9
005B01B6          B3 02        mov bl,2
005B01B8          41           inc ecx
005B01B9          B0 10        mov al,10
-> 005B01BB          E8 4F000000 call 5B020F
005B01C0          10C0         adc al,al
-- 005B01C2          73 F7        jae 5B01B8
-- 005B01C4          75 3F        jne 5B0205
005B01C6          AA           stosb
005B01C7          EB D4        jmp 5B019D
-> 005B01C9          E8 4D000000 call 5B021B
005B01CE          29D9         sub ecx,ebx
-- 005B01D0          75 10        jne 5B01E2
005B01D2          E8 42000000 call 5B0219
005B01D7          EB 28        jmp 5B0201
-> 005B01D9          AC           lodsb
005B01DA          D1E8         shr eax,1
-- 005B01DC          74 4D        je 5B022B
005B01DE          11C9         adc ecx,ecx
005B01E0          EB 1C        imo 5B01FE

```

Funzione che si occupa di decomprimere il PE



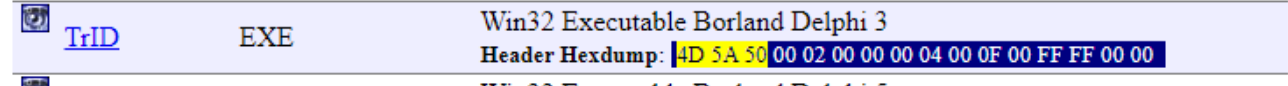
Function Graph della funzione che si occupa della decompressione del PE

Al ritorno della funzione in EDX avremo il puntatore al PE completamente deoffuscato, dove infatti troviamo gli IP estratti in precedenza:

Indirizz	Hex	ASCII
048FA18E	4D 5A 50 00 02 00 00 00 04 00 0F 00 FF FF 00 00	MZP.....yy..
048FA19E	88 00 00 00 00 00 00 00 40 00 1A 00 00 00 00 00@.....
048FA1AE	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
048FA18E	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
048FA1CE	8A 10 00 0E 1F B4 09 CD 21 88 01 4C CD 21 90 90iLiL..
048FA1DE	54 68 69 73 20 70 72 6F 67 72 61 6D 20 6D 75 73	This program mus
048FA1EE	74 20 62 65 20 72 75 6E 20 75 6E 64 65 72 20 57	t be run under W
048FA1FE	69 6E 33 32 0D 0A 24 37 00 00 00 00 00 00 00 00	in32.\$7.....
048FA20E	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
048FA21E	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
048FA22E	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
048FA23E	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
048FA24E	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
048FA25E	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
048FA26E	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
048FA27E	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

Comando:

PE completamente deoffuscato zona di memoria puntata da EDX



Header della nuova DLL estratta corrisponde a Delphi 3

Indirizz	Data
04D3D0F6	B5 3F 2C C2

Ricerca dell'IP nell'area di

memoria puntata da EDX

Questo PE viene poi utilizzato per sovrascrivere le attuali sezioni come visto in precedenza.

Nella seconda parte dell'articolo realizzerò un video per vedere praticamente questa parte tramite debugger per poi scrivere lo script che si occupa in automatico di rimuovere i diversi layer di obfuscation; per ora, a scopo "didattico", possiamo effettuare il dump della DLL e proseguire; dopo aver effettuato il fixing con Shylla, forzo la decompilazione nella sezione .data ed ecco la funzione di Decryption Config, simile al primo sample che abbiamo ottenuto:

```

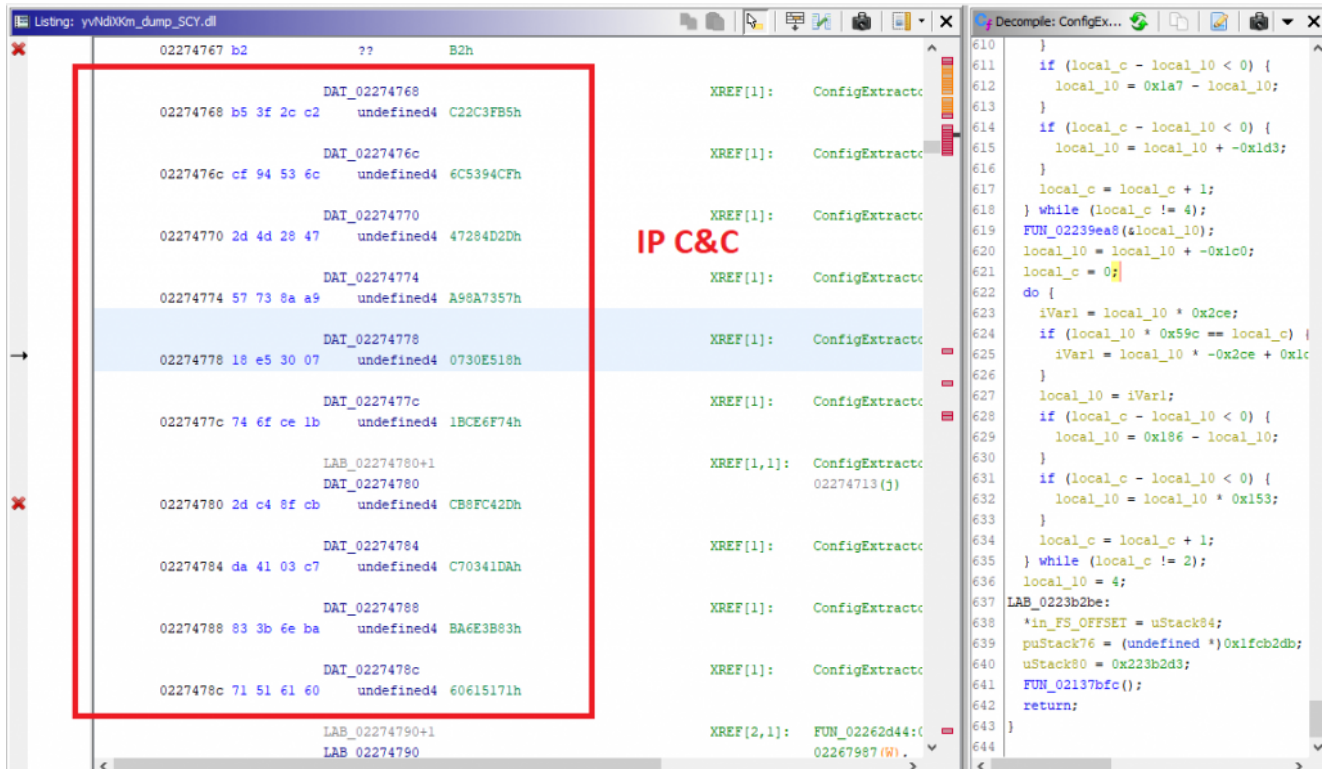
312     }
313     local_10 = iVar1;
314     if (local_c < local_10 * 2) {
315         local_10 = local_10 + 0x2d3;
316     }
317     if (local_c == 0) {
318         local_10 = local_10 + -0x280;
319     }
320     } while (local_c < 2);
321     local_c = (local_10 + -400) * (local_10 + -400);
322     if (((int)uRam0200b7ad >> 0x1f) + iRam0200b7b5 + (uint)CARRY4(uRam0200b7ad,uRam0200b7b1) ==
323         iRam0200b7bd && uRam0200b7ad + uRam0200b7b1 == iRam0200b7b9) goto LAB_022eb2be;
324 }
325 uRam0200b7ad = 0x3c;
326 local_c = 0;
327 do {
328     iVar1 = -local_10;
329     local_10 = iVar1 + 0x3a2;
330     if (local_c == 0) {
331         local_10 = iVar1 + 0x613;
332     }
333     if (local_c < local_10 * 2) {
334         local_10 = 0x2df - local_10;
335     }
336     if (local_c - local_10 < 0) {
337         local_10 = 0x1ee - local_10;
338     }
339     if (local_c - local_10 < 0) {
340         local_10 = local_10 * 0x1bb;

```

Config Builder

Offset	Name	Value	Meaning
105590	Characteristics	3F0458B	
105594	TimeStamp	558BEC45	giovedì, 25.06.2015 11:55:49 UTC
105598	MajorVersion	3E8	
10559A	MinorVersion	E455	
10559C	Name	B7DC23B	
1055A0	Base	8DDC558D	
1055A4	NumberOfFunc...	6DE8E445	
1055A8	NumberOfNames	8BFFFE15	
1055AC	AddressOfFunc...	452BF045	
1055B0	AddressOfNames	DC4589EC	
1055B4	AddressOfNam...	4589C033	

Caratteristiche nuova DLL sovrascritta dalla shellcode
 E anche questa volta otteniamo la lista degli IP come variabili globali:



IP del C&C in formato int

Possiamo facilmente cambiare l'espressione regolare dell'operazione di copy (\xa1\x68\x47\x00\x02) per ottenere:

181.63.44.194
 207.148.83.108
 45.77.40.71
 87.115.138.169
 24.229.48.7
 116.111.206.27
 45.196.143.203
 218.65.3.199
 131.59.110.186
 113.81.97.96

Vediamo come adattare ora lo script precedente per farlo funzionare per i due sample visti fino ad ora, iniziamo ad analizzare le due funzioni di Config Builder; dovendo generalizzare nella regex la destination essendo l'indirizzo dove son presenti gli IP diversi, mantenendo solo come statica la source (registro EAX) otteniamo un numero molto elevato di MOV, è necessario quindi rendere più specifica la regex.

Dopo il primo MOV vediamo che è presente un altro MOV che salva il valore di EAX in un'altra variabile globale.


```

0223ab8e a1 8c 47      MOV      EAX, [0200478c]=>DAT_0227478c
          00 02
0223ab93 a3 e5 b7      MOV      [0200b7e5]=>DAT_0227b7e4+1,EAX
          00 02
03efac7e a1 8c 57      MOV      param_1,[DAT_03f3578c]
          f3 03
03efac83 a3 e5 c7      MOV      [DAT_03f3c7e4+1],param_1
          f3 03

```

MOV del primo sample

MOV del secondo sample

Inoltre all'inizio della funzione sono presenti delle inizializzazioni di registri quasi uguali (si noti che questa parte non è strettamente necessaria, infatti anche rimuovendo il regex per questa parte lo script funziona comunque essendo che la prima modifica già permette di ottenere solo quell'indirizzo):

```

0223a044 64 ff 30      PUSH    dword ptr FS:[EAX]
0223a047 64 89 20      MOV     dword ptr FS:[EAX],ESP
0223a04a 33 c0        XOR     EAX,EAX
0223a04c 89 45 f8      MOV     dword ptr [EBP + local_c],EAX

```

Operazioni del

primo sample

```

03efa42a 64 ff 30      PUSH    dword ptr FS:[param_1]
03efa42d 64 89 20      MOV     dword ptr FS:[param_1],ESP
03efa430 33 c0        XOR     param_1,param_1
03efa432 89 45 f4      MOV     dword ptr [EBP + local_10],param_1

```

Operazioni

del secondo sample

Dopo queste due considerazioni la regex diventa quindi:

```

header = b'\x64\xff\x30\x64\x89\x20\x33\xc0\x89\x45.'
copy_operation = b'\xa1\x68...\xa3\xc1...'
regex = header + copy_operation

```

Provando il nuovo script funziona su entrambi i sample:

```

C:\Users\antmal\Desktop\corso malware\Danabot>python decryptDanabot.py 1ef8b148b1b51343c3150d5dad342d3e\yvNdiXKm_dump_SC
Y.dll
181.63.44.194
207.148.83.108
45.77.40.71
87.115.138.169
24.229.48.7
116.111.206.27
45.196.143.203
218.65.3.199
131.59.110.186
113.81.97.96

C:\Users\antmal\Desktop\corso malware\Danabot>python decryptDanabot.py sample3
243.127.43.6
64.126.175.2
130.15.230.152
74.99.136.192
244.14.226.35
95.179.168.37
51.129.76.8
151.210.85.159
45.76.123.177
75.57.14.121

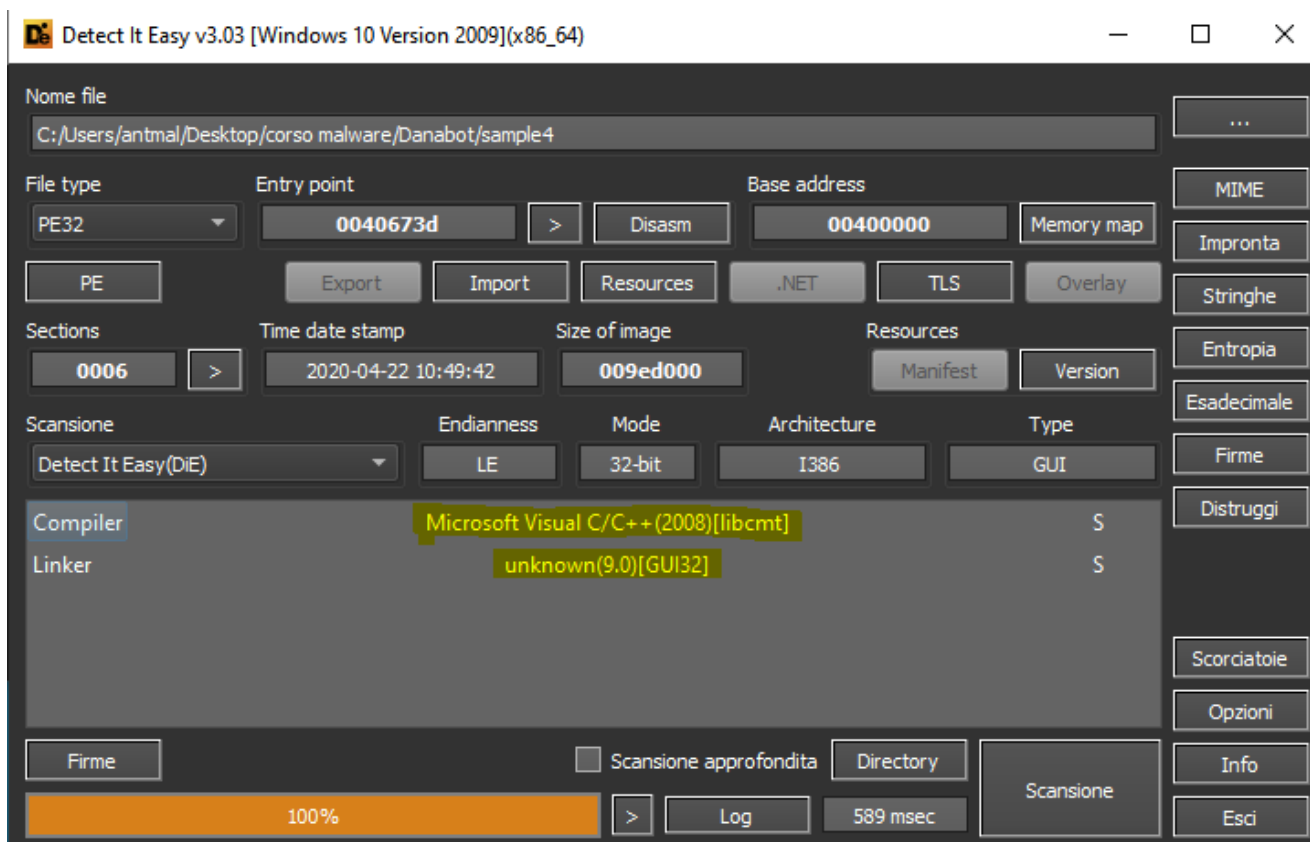
```

IP decifrati

Si noti che questo script funziona solo sul Main Loader di DanaBot; nel prossimo post vedremo come aggiornare lo script per farlo funzionare direttamente sul dropper (vbs, exe) e automatizzare la decifrazione effettuata dalla shellcode.

Analisi secondo Loader

Iniziamo ora l'analisi del terzo sample (**MD5: 5c0be4a5273dec6b3ebb180a90f337f2**), questa volta è un EXE sviluppato in C:



In questo caso l'EXE si occupa di estrarre la DLL nella cartella corrente e avviarla; questa DLL a sua volta avvia la stessa DLL passando un parametro casuale (quindi verrà avviato in realtà l'entry); infatti come possiamo vedere oltre gli export canonici essendo una DLL scritta in Delphi, non abbiamo altro:

Exported Functions [3 entries]					
Offset	Ordinal	Function RVA	Name RVA	Name	Forwarder
57DA28	1	57F634	588084	dbkFCallWrapp...	
57DA2C	2	10688	588070	__dbk_fcall_wra...	
57DA30	3	3FF08C	588051	TMethodImple...	

■ C:\Windows\SysWOW64\rundll32.exe

```
C:\Windows\system32\rundll32.exe C:\Users\Admin\AppData\Local\Temp\9D68E9~1.DLL,Z C:\Users\Admin\AppData\Local\Temp\9D68E9~1.EXE
```

■ C:\Windows\SysWOW64\RUNDLL32.EXE

```
C:\Windows\system32\RUNDLL32.EXE C:\Users\Admin\AppData\Local\Temp\9D68E9~1.DLL,YR1E
```

Applico la conoscenza precedente cercando riferimenti a socket, non trovando niente.

Sospetto quindi che in realtà sia un packer e tramite il debugger allora analizzo le varie chiamate **VirtualAlloc**, **VirtualProtect** e **CreateThread** si vede come in realtà la DLL abbia al suo interno un'altra DLL, questa volta con un export **FunDLLData**:

Exported Functions [4 entries]					
Offset	Ordinal	Function RVA	Name RVA	Name	Forwarder
501C28	1	502634	61F099	dbkFCallWrapp...	
501C2C	2	116BC	61F085	__dbk_fcall_wra...	
501C30	3	667E4	61F066	TMethodImple...	
501C34	4	47B744	61F05B	FunDLLData	

In questa DLL invece si trovano i riferimenti alla comunicazione tramite socket e tracciando i

parametri passati riusciamo a raggiungere il Config Builder:

```
00873040 int32_t ConfigBuilder()
00873048 data_a180d8 = 3
00873053 data_a180dc = sub_866d30()
0087305b data_a180e0 = sub_871134()
00873063 data_a180ec = sub_871400()
00873066 char* esi = " B2585F6479280F48B64C99F950BBF36..."
0087306b char* edi = &data_a18121
00873073 for (int32_t ecx = 8; ecx != 0; ecx = ecx - 1)
00873073     *edi = *esi
00873073     edi = &edi[4]
00873073     esi = &esi[4]
00873075 *edi = *esi
00873076 data_a180f4 = 0x6e5
0087307f data_a180f8 = 0
00873085 int32_t eax_4 = data_a180e4 + 1
00873086 data_a180e8.d = eax_4
00873089 data_a180f0 = 0x57e40
00873090 data_a18142 = 0x621a03c0
00873097 data_a1814a = 0x6b1a03c0
0087309e data_a18152 = 0x530a1c0
008730a5 data_a1815a = 0xcb92ecc0
008730af data_a18146 = 0x1bb
008730b6 data_a1814e = 0x1bb
008730bd data_a18156 = 0x1bb
008730c4 data_a1815e = 0x1bb
008730d1 return eax_4
```

Per questo post è tutto, nei prossimi continueremo l'analisi, analizzeremo le altre informazioni presenti nel config, estrarremo la chiave RSA utilizzata per la comunicazione e generalizzeremo lo script per i restanti sample 😊 Per qualunque consiglio o richiesta, scrivete pure nei commenti, grazie! 😊

Si ringrazia bleepingcomputer.com per l'immagine di copertina

Share this content:

-
-
-
-
-