CERT-AGID

# Gestire Smokeloader e velocizzarne l’analisi

Linee di azione per ottenere rapidamente payload e URL

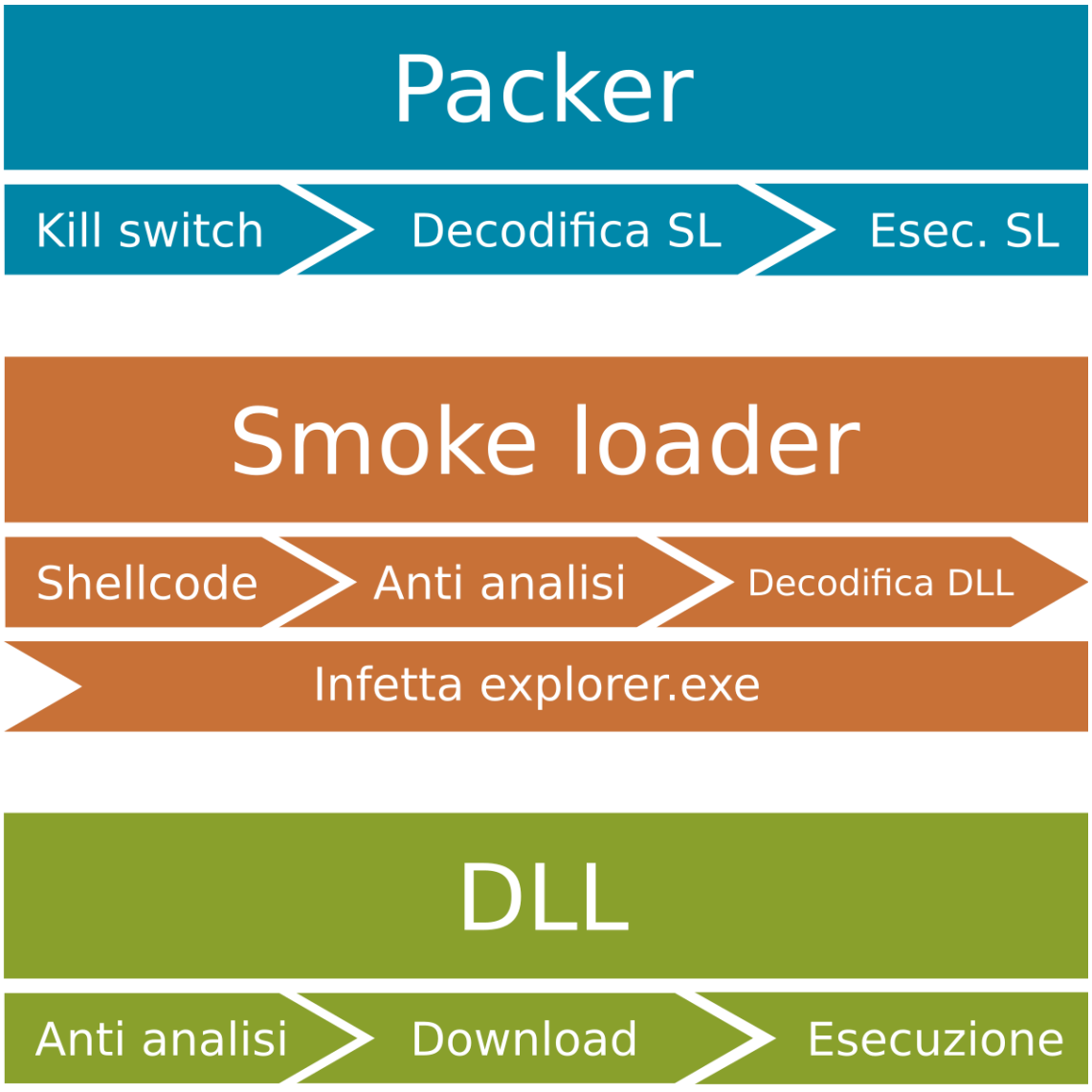
# Table of Contents

# Introduzione

Dato che SmokeLoader (SL in seguito) è già stato ampiamente discusso in letteratura (in particolare si faccia riferimento a Deep Analysis of SmokeLoader di Abdallah Elshinbary, che rimane ancora in larga parte valido) in questo documento non verrà ripetuta la sua analisi completa ma verranno date delle **linee di azione** per ottenere il payload ed gli URL in breve tempo.

Dato che i malware sono in costante cambiamento e che una semplice lista di azioni senza contesto riduce notevolmente l'utilità della stessa, nei capitoli successivi sono forniti maggiori dettagli (sempre restando che l'analisi completa di SmokeLoader è disponibile in letteratura).

Si riportano qui, schematicamente, i componenti e le loro azioni.




La seconda parte di questo documento contiene dettagli più precisi sui componenti della catena di infezione, evitando però di approfondire SL (già trattata altrove).

# Recuperare i gli URL dropper ed i payload

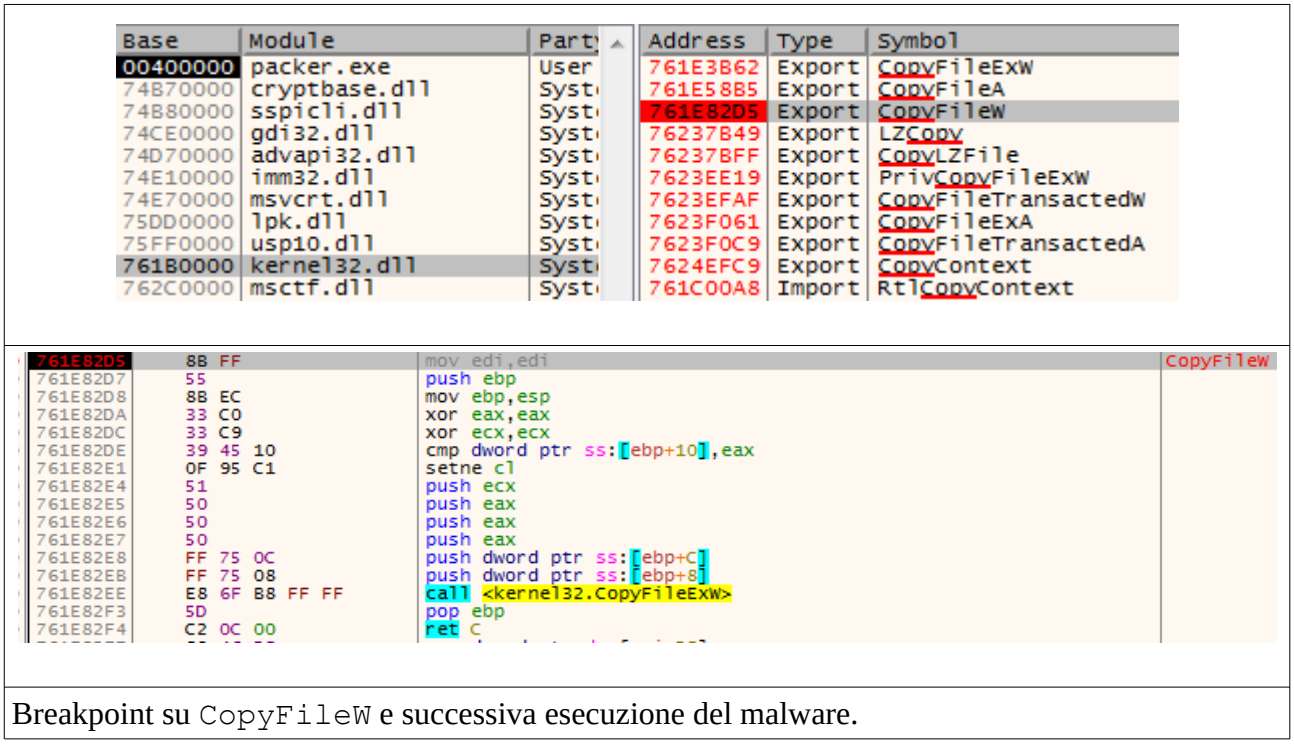
Questi passi si applicano debuggando direttamente il packer.

## 1. Neutralizzare il caricamento della copia di ntdll

E' possibile usare una delle due seguenti alternative.

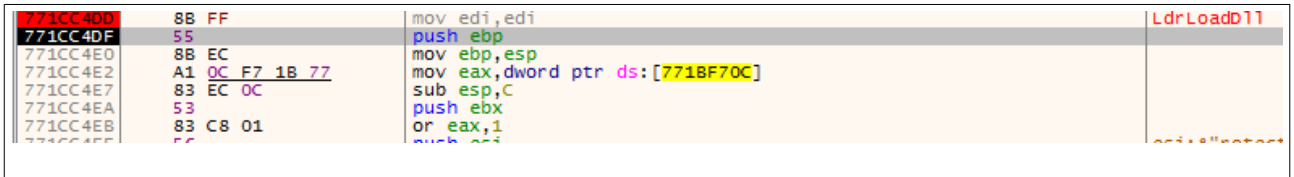
### Alternativa A

1. Mettere un breakpoint su `CopyFileW` (in *kernel32.dll*) ed eseguire il malware.



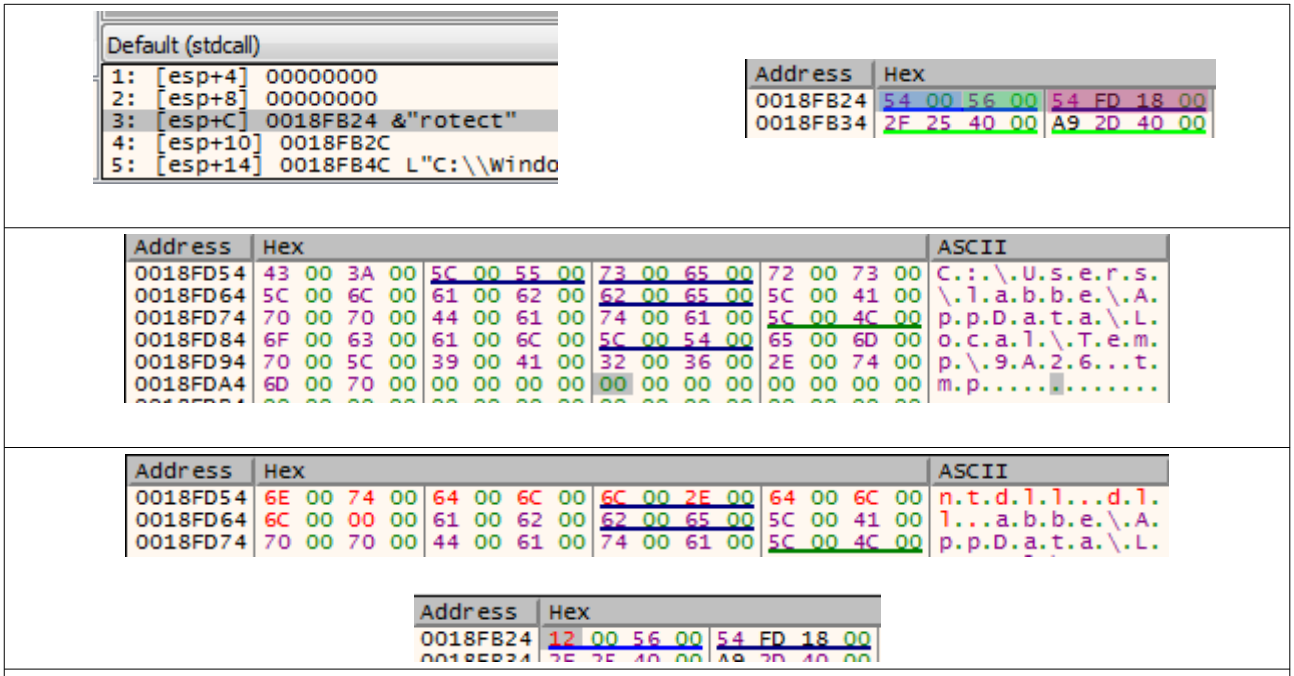

Breakpoint su `CopyFileW` e successiva esecuzione del malware.

2. Mettere un breakpoint su `LdrLoadDll` (in *ntdll.dll*) e continuare l'esecuzione.




Il malware raggiunge `LdrLoadDll`.

3. Cambiare la stringa unicode passata a `LdrLoadDll` dal percorso del file temporaneo a "*ntdll.dll*".




**In altro a sinistra:** I parametri di LdrLoadDll, il terzo è il puntatore alla stringa unicode del file da caricare. **In alto a destra:** la struttura della stringa, in blu la sua lunghezza, in verde la dimensione allocata e in rosso il puntatore ai dati. **Al centro:** i dati della stringa unicode prima e dopo (sotto) il patching. Infine, la dimensione della stringa è aggiustata per riflettere i nuovi dati.

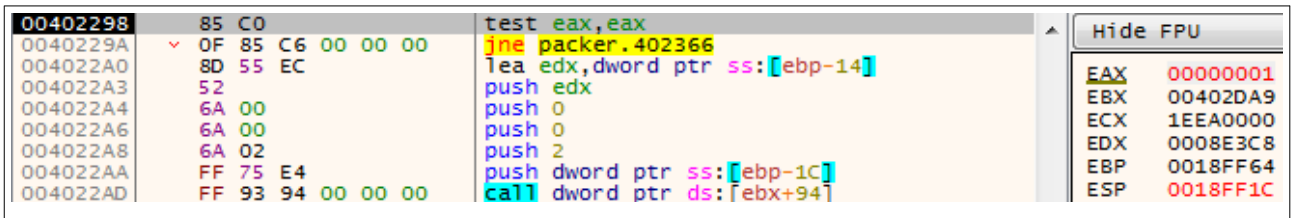
Alternativa B

1. Mettere un breakpoint su `LdrLoadDll` ed eseguire il malware.
2. Saltare tutte le chiamate che non riguardano un file temporaneo.
3. Cambiare la stringa unicode passata a `LdrLoadDll` dal percorso del file temporaneo a "*ntdll.dll*".

**Prima di proseguire al passo successivo, disabilitare i breakpoint inseriti e non continuare l'esecuzione del malware.**

## 2. Neutralizzare le tecniche anti VM

1. Mettere un breakpoint in `NtOpenKey` (in **ntdll.dll**) e continuare l'esecuzione.
2. Ritornare al codice del malware e cambiare il valore di `eax` in uno non zero.



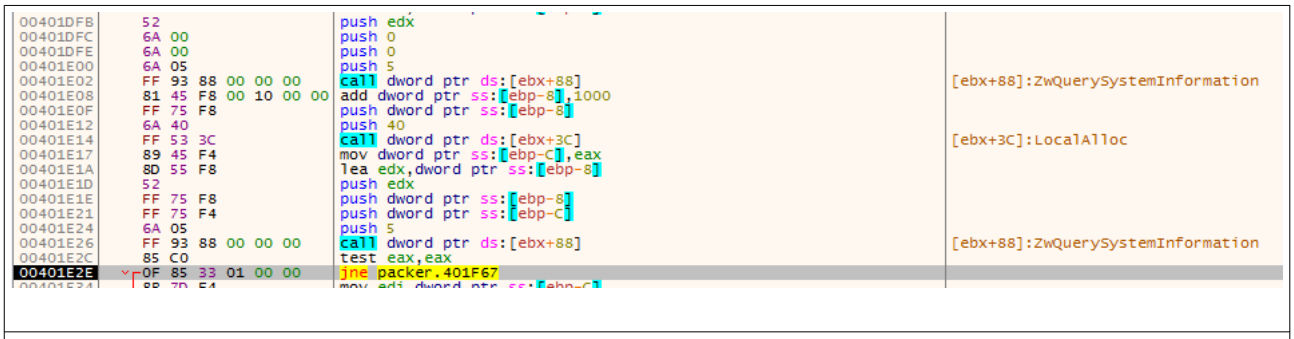

Il codice del malware subito dopo la chiamata a `NtOpenKey` e il valore di `eax` (a destra, in rosso) modificato da zero a uno.

3. Eseguire il malware e ripetere dal punto 2 una seconda volta.

Prima di proseguire, disabilitare i breakpoint inseriti.

## 3. Neutralizzare le tecniche anti tool

1. Mettere un breakpoint in `ZwQuerySystemInformation` (in *ntdll.dll*) e continuare l'esecuzione.
2. Ritornare al codice del malware e cambiare il valore di eax in uno non zero.




Il malware chiama `ZwQuerySystemInformation` due volte per ottenere la lista di processi e due volte per ottenere la lista dei moduli. Sopra le due chiamate (la prima per l'allocazione del buffer dei risultati) per ottenere la lista dei processi.

3. Eseguire il malware e ripetere dal punto 2 altre tre volte.

**Prima di proseguire al passo successivo, disabilitare i breakpoint inseriti e non continuare l'esecuzione del malware.**

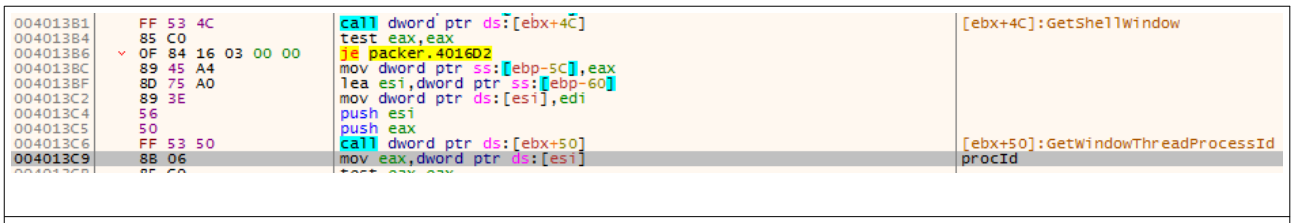
## 4. Impedire l'infezione di explorer.exe

L'infezione di explorer.exe è problematica perchè il suo debug è reso complicato dal fatto che explorer.exe è necessario per un corretto utilizzo dell'ambiente desktop di Windows.
Conviene quindi far infettare un altro processo.

SL crea due sezioni di memoria: una per la DLL ed una per il buffer di lavoro di questa. Queste due sezioni sono mappate sia sul processo di SL sia su *explorer.exe*. Infine un thread viene creato in explorer.exe.
Per impedirne l'infezione dobbiamo prima recuperare i quattro base address (due in SL e due in *explorer.exe)* delle sezioni e poi cambiare i parametri della creazione del thread.

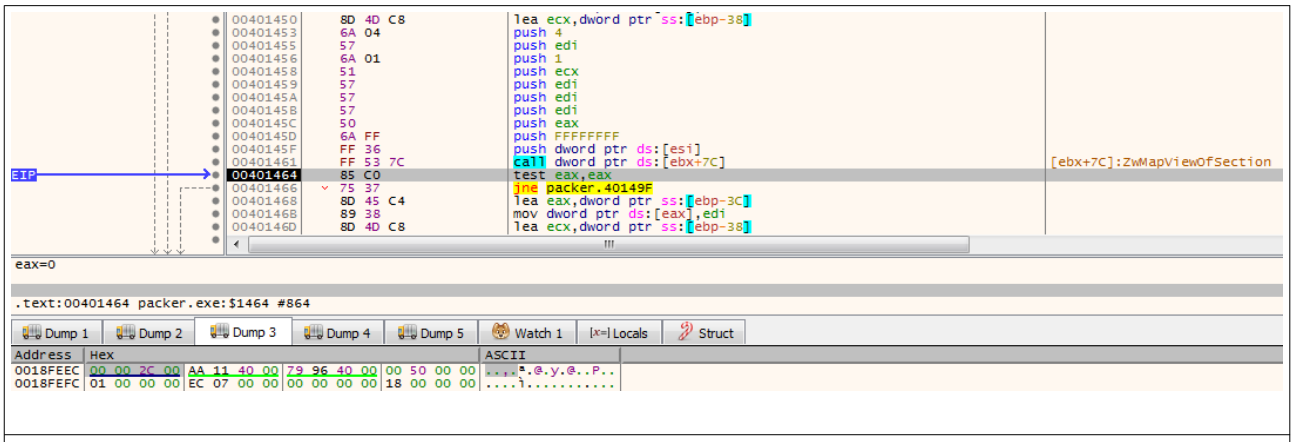
1. Mettere un breakpoint su `GetWindowThreadProcessId` e continuare l'esecuzione.
2. Il secondo parametro è un puntatore dove verrà scritto il PID del processo che SL infetterà. Eseguire l'API e poi cambiare il PID inserendo quello di un processo a 64 bit che si vuole

infettare (più scarno e semplice è il processo, più semplice sarà il suo debug; vedi appendice A).




Cambiare il PID che SL usa per iniettare la DLL può essere fatto prendendo nota, al momento del breakpoint su `GetWindowThreadProcessId`, del puntatore in cui viene scritto o analizzando il codice del malware dopo la chiamata. Sopra, la riga evidenziata mostra come il PID venga messo in eax, dando all'analista la possibilità immediata di modificarlo.

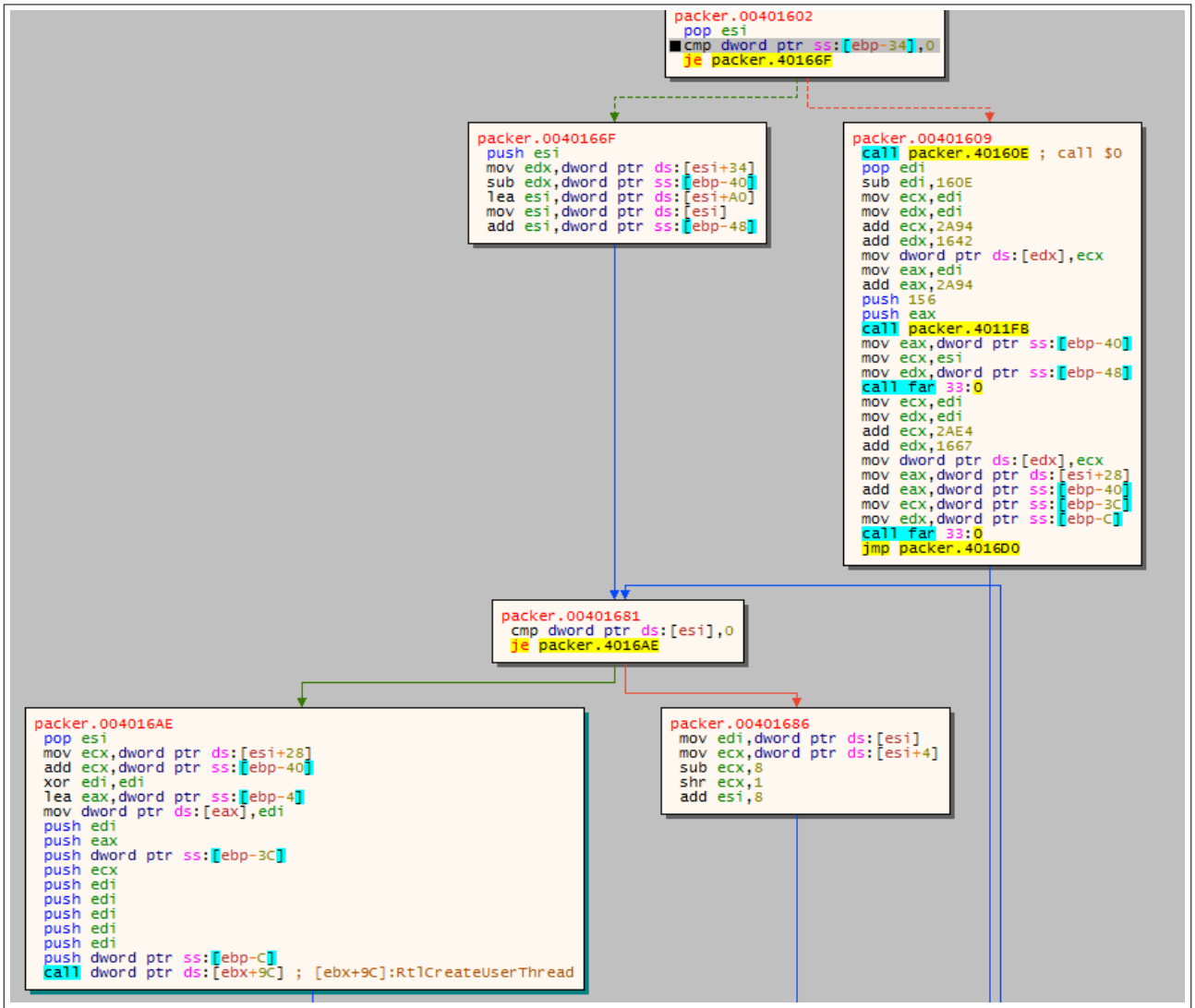
3. Mettere un breakpoint su `ZwMapViewOfSection` (in *ntdll.dll*) e continuare l'esecuzione. Quando il malware arriva ad eseguire quest'API, il terzo parametro è un puntatore che conterrà (una volta ritornati dall'API) il base address del buffer allocato.
   Prendere nota dell'indirizzo del buffer allocato.




Al ritorno da `ZwMapViewOfSection`, recuperare il base address B del buffer allocato. In questo caso B è `2c0000h` come evidenziato in basso nella figura.

4. Ripetere il punto 1 altre tre volte.
5. Chiamiamo, in ordine di incontro, i base address recuperati come: *SL.data*, *EXP.data*, *SL.dll* e *EXP.dll*.
6. Eseguire il codice del malware passo-passo fino a che non si intravede una chiamata a `RtlCreateUserThread` o due chiamate far (Heaven's gate).
   Mettere un breakpoint **hardware** su `RtlCreateUserThread` o sulla **seconda** chiamata far e continuare l'esecuzione.
   Nel caso si arrivi alla chiamata a `RtlCreateUserThread`, il quartultimo parametro indica l'indirizzo dell'entry-point nel processo infettato.
   Nel caso si arrivi alla chiamata far, l'indirizzo è in eax.

7. Attaccare un debugger al processo infettato e mettere un breakpoint sull'entry-point appena trovato.
8. Continuare con l'esecuzione del malware (che creerà il thread e terminerà).




Nel ramo di destra il malware esegue due chiamate far: la prima riloca la DLL e la seconda chiama `RtlCreateUserThread` (una volta recuperato da ntdll usando il PEB). A sinistra invece la rilocazione e la chiamata a `RtlCreateUserThread` sono nel codice a 32 bit.

## 5. Rimuovere i controlli anti analisi

L'analisi prosegue nel processo infettato.

1. Mettere un breakpoint su `CreateThread` e continuare l'esecuzione.
2. Saltare l'esecuzione dell'API impostando `rip` direttamente sull'istruzione `ret`.
3. Ripetere da 1 un'altra volta.

## 6. Recupero dei dropper URL

1. Mettere un breakpoint su `WinHttpConnect` (in winhttp.dll).
2. Il malware proverà gli URL in sequenza, facendo fallire la chiamata a `WinHttpConnect` è possibile costringerlo a provare il prossimo URL.

## 7. Recupero del payload

1. Anzichè seguire il punto 5, mettere un breakpoint in WinHttpReadData (in winhttp.dll).
2. Il payload sarà nel buffer di output dell'API indicata e verrà decodificato successivamente dal malware (non si tratta di un PE, almeno nel nostro caso, ma di codice che verrà inettato direttamente in una nuova instanza di *explorer.exe*, a tal fine dei breakpoint su `CreateProcessInternal`, `NtMapViewOfSection` e `ResumeThread` aiutano ad arrivare direttamente alle parti di interesse).

# Il packer

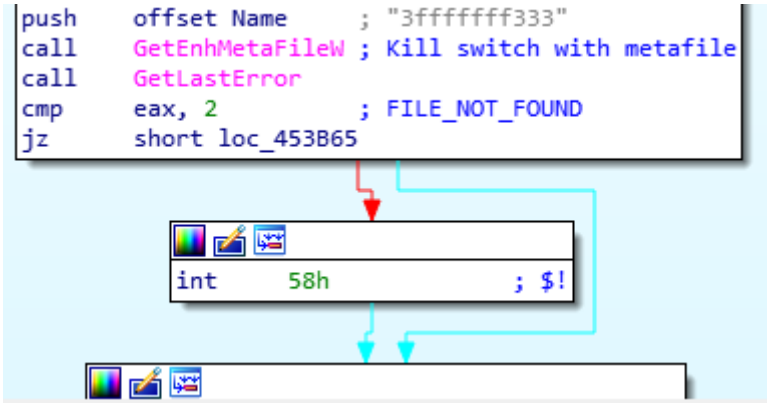
Il file eseguibile che conteneva SL ottenuto dal CERT-AGID è risultato avere un primo stadio in cui un PE veniva decodificato e mappato nel processo stesso.
Ai fini di questa analisi lo abbiamo trattato come un packer, seguendo la nostra nomenclatura per cui ogni eseguibile il cui unico scopo è quello di decodificare ed eseguire un PE è un packer.

Questo packer non presenta alcuna tecnica anti-analisi o offuscamento, è di semplice analisi sebbene l'utilizzo diffuso di variabili globali e l'assenza totale di ottimizzazioni lo renda un po' meno "scorrevole" alla lettura.

Consiste principalmente di tre fasi: un kill switch, la decodifica del payload e la sua mappatura ed esecuzione.


## Kill switch

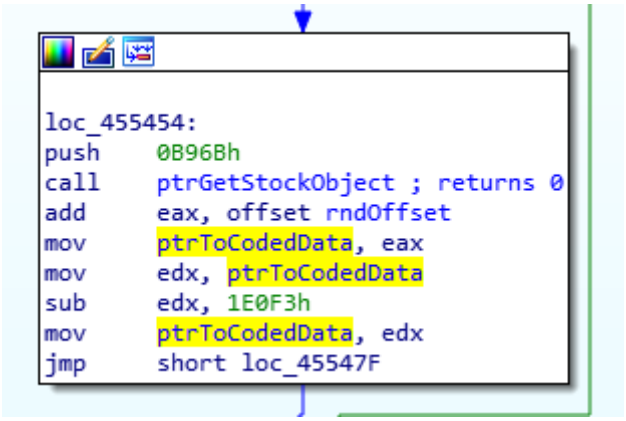
Appena ad inizio esecuzione il packer prova a aprire un metafile di nome `3fffffff333` ed in caso positivo termina generando un'eccezione.




Considerando il nome del file e l'esito di una sua eventuale presenza, siamo più inclini a classificare questa azione come un kill switch piuttosto che una tecnica anti analisi.

## Decodifica del payload (SL)

Poco dopo il packer ottiene l'indirizzo del payload codificato, questo è ottenuto tramite costanti infisse nelle istruzioni[1] (in particolare è ottenuto come delta dall'indirizzo di una variabile nota).



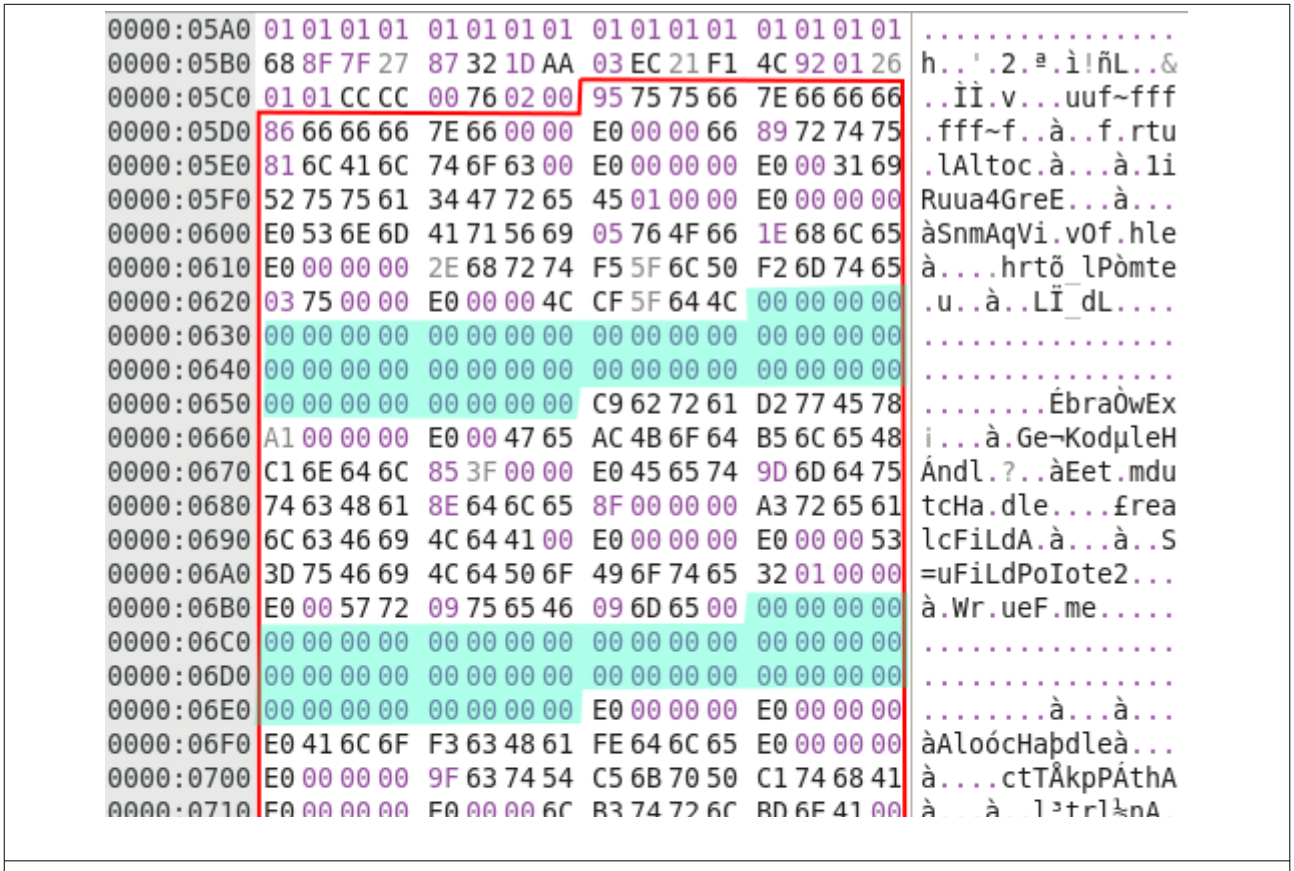

Il codice che ottiene il puntatore al payload codificato. Notare come questo si semplifichi nella sottrazione di `1e0f3h` dal VA della variabile `rndOffset`.

Per qualche motivo, probabilmente per occultazione, il codice mostrato sopra si trova in una routine che legge il valore di default della chiave `iNTerface\\{b196b287-bab4-101a-b69c-00aa00341d07}` senza poi usarlo.

La decodifica del payload avviene in due fasi: nella prima vengono rimossi i byte in eccesso e nella seconda viene eseguito un algoritmo di decodifica.

Prima di essere inserito nel packer il payload è stato alterato inserendo 44 byte nulli (o comunque inutili) ogni 100 byte originali.
Questo è immediatamente visibile, oltre che dal codice che li rimuove, anche aprendo il packer con un editor esadecimale.

1 In terminologia Intel, sono detti *immediate*.



```
0000:05A0 01 01 01 01  01 01 01 01  01 01 01 01  01 01 01 01  ................
0000:05B0 68 8F 7F 27  87 32 1D AA  03 EC 21 F1  4C 92 01 26  h..'.2.ª.ì!ñL..&
0000:05C0 01 01 CC CC  00 76 02 00  95 75 75 66  7E 66 66 66  ..ÌÌ.v...uuf~fff
0000:05D0 86 66 66 66  7E 66 00 00  E0 00 00 66  89 72 74 75  .fff~f..à..f.rtu
0000:05E0 81 6C 41 6C  74 6F 63 00  E0 00 00 00  E0 00 31 69  .lAltoc.à...à.1i
0000:05F0 52 75 75 61  34 47 72 65  45 01 00 00  E0 00 00 00  Ruua4GreE...à...
0000:0600 E0 53 6E 6D  41 71 56 69  05 76 4F 66  1E 68 6C 65  àSnmAqVi.vOf.hle
0000:0610 E0 00 00 00  2E 68 72 74  F5 5F 6C 50  F2 6D 74 65  à....hrtõ_lPòmte
0000:0620 03 75 00 00  E0 00 00 4C  CF 5F 64 4C  00 00 00 00  .u..à..LÏ_dL....
0000:0630 00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ................
0000:0640 00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ................
0000:0650 00 00 00 00  00 00 00 00  C9 62 72 61  D2 77 45 78  ........ÉbraÒwEx
0000:0660 A1 00 00 00  E0 00 47 65  AC 4B 6F 64  B5 6C 65 48  ¡...à.Ge¬KodµleH
0000:0670 C1 6E 64 6C  85 3F 00 00  E0 45 65 74  9D 6D 64 75  Ándl.?..àEet.mdu
0000:0680 74 63 48 61  8E 64 6C 65  8F 00 00 00  A3 72 65 61  tcHa.dle....£rea
0000:0690 6C 63 46 69  4C 64 41 00  E0 00 00 00  E0 00 00 53  lcFiLdA.à...à..S
0000:06A0 3D 75 46 69  4C 64 50 6F  49 6F 74 65  32 01 00 00  =uFiLdPoIote2...
0000:06B0 E0 00 57 72  09 75 65 46  09 6D 65 00  00 00 00 00  à.Wr.ueF.me.....
0000:06C0 00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ................
0000:06D0 00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ................
0000:06E0 00 00 00 00  00 00 00 00  E0 00 00 00  E0 00 00 00  ........à...à...
0000:06F0 E0 41 6C 6F  F3 63 48 61  FE 64 6C 65  E0 00 00 00  àAloócHaþdleà...
0000:0700 E0 00 00 00  9F 63 74 54  C5 6B 70 50  C1 74 68 41  à....ctTÅkpPÁthA
0000:0710 E0 00 00 00  E0 00 00 6C  B3 74 72 6C  BD 6E 41 00  à...à..l³trl½nA.
```

L’inizio del payload del packer contornato in rosso ed i byte nulli evidenziati in verde chiaro.

Una volta ricompattato il payload è trasformato secondo l’algoritmo seguente.

```
push    ebp
mov     ebp, esp
push    ecx
push    esi
mov     eax, [ebp+varI]
mov     [ebp+locVarI], eax
mov     ecx, bufferOffsetInDec
mov     [ebp+arg_0], ecx
push    24C4h           ; lpIconName
push    0               ; hInstance
call    LoadIconA
mov     esi, eax
add     esi, [ebp+locVarI]
push    24C4h           ; lpIconName
push    0               ; hInstance
call    LoadIconA
add     esi, eax        ; esi = VarI
mov     edx, [ebp+arg_0]
add     esi, [edx]      ; i + buffer[i]
mov     eax, [ebp+arg_0]
mov     [eax], esi      ; buffer[i] += i
pop     esi
mov     esp, ebp
pop     ebp
retn
```

```
push    ebp
mov     ebp, esp
push    edi
mov     eax, lastVarI
mov     xorValue, eax
mov     ecx, lastOffsetBuffer
mov     edx, [ecx]
mov     dword_4DBAB4, edx
mov     ecx, xorValue
mov     eax, dword_4DBAB4
mov     dword_4DBAB8, eax
mov     eax, 8AE38h
mov     eax, 8AE38h
mov     eax, dword_4DBAB8
jmp     short $+2
```

```
loc_4538DD:
mov     edi, eax
xor     dword_4DBAB8, ecx
mov     dword_4DBAB4, 0
mov     edi, dword_4DBAB8
add     dword_4DBAB4, edi
mov     edi, edi
mov     eax, lastOffsetBuffer
mov     ecx, dword_4DBAB4
mov     [eax], ecx
```

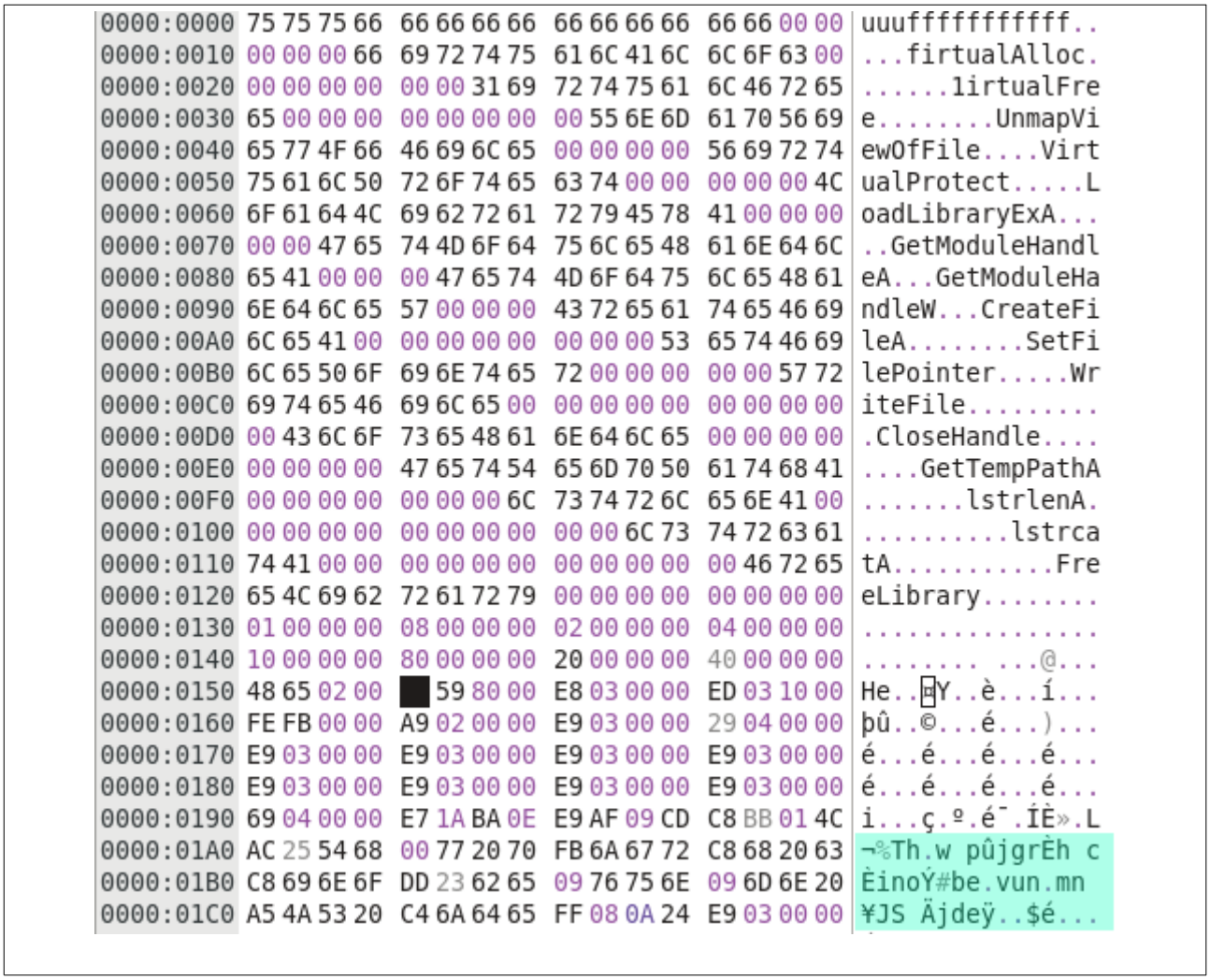
```
lea     ecx, [eax+edx+0DEh] ; ecx = varI + 0xe0
mov     [ebp+var_4], ecx
mov     edx, bufferOffsetInDec
mov     lastOffsetBuffer, edx
mov     eax, [ebp+var_4]
mov     lastVarI, eax
```

```
void decode(uint32_t* payload, size_t size)
{
     for (size_t i = 0; i < size / 4; i++)
          payload[i] = (payload[i] + i * 4) ^ (0xe0 + i * 4)
}
```

L’algoritmo di decodifica del payload e i frammenti di codice assembly per la sua ricostruzione.

La dimensione del payload è contenuta nella DWORD subito prima di questo; nel sample mostrato il payload si trova all’offset 5c8h e la sua dimensione all’offset 5c4h.
Più avanti descriveremo un’euristica per trovare il payload all’interno del packer.

Il risultato della decodifica contiene il nome di alcune API e quello che sembra un PE codificato ed appena riconoscibile.

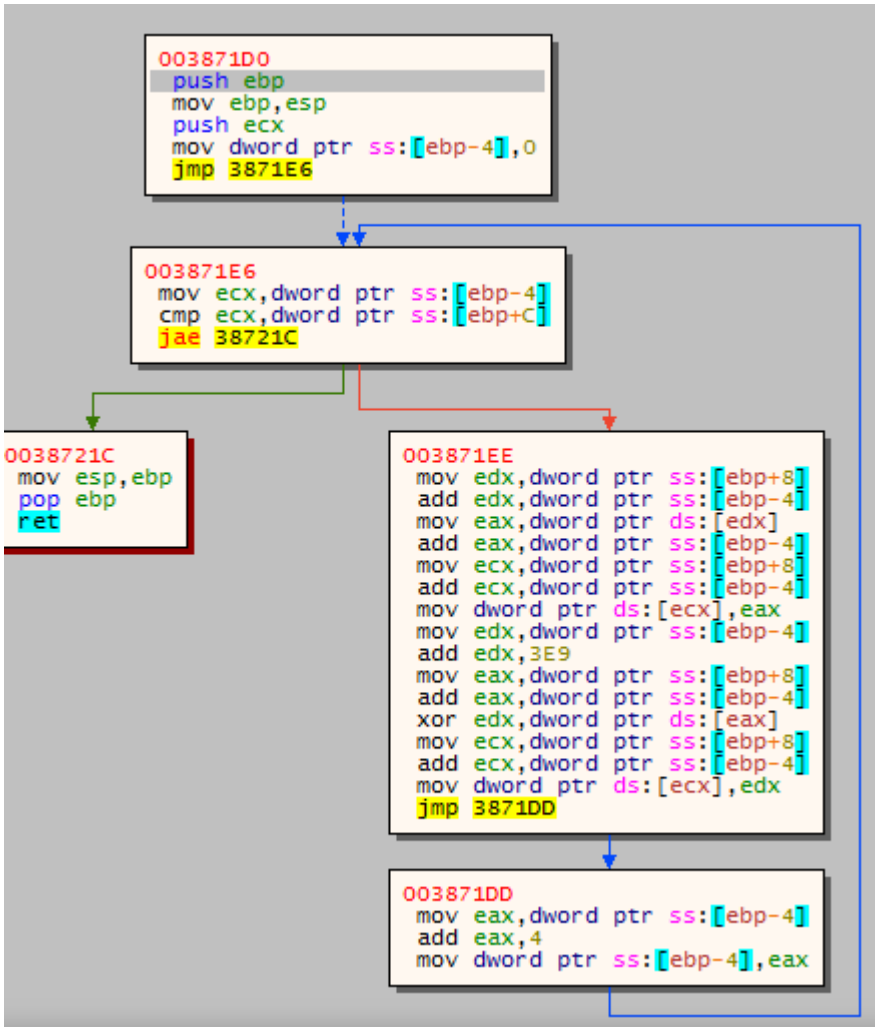


Il payload decodificato e la stringa “*This program cannot be run in*” appena riconoscibile.

Oltre al PE ed alle stringhe con i nomi delle API da importare, il payload contiene anche del codice[2] che ha il compito di proseguire con l’infezione.

Anche in questo caso non vi sono tecniche di anti-analisi nè codice offuscato per cui il reverse engineering dell’algoritmo di decodifica del PE è piuttosto semplice.

All’offset 150h è presente una DWORD che indica la dimensione del PE e subito dopo il PE stesso. Questo è copiato in un buffer apposito e decodificato come mostrato.

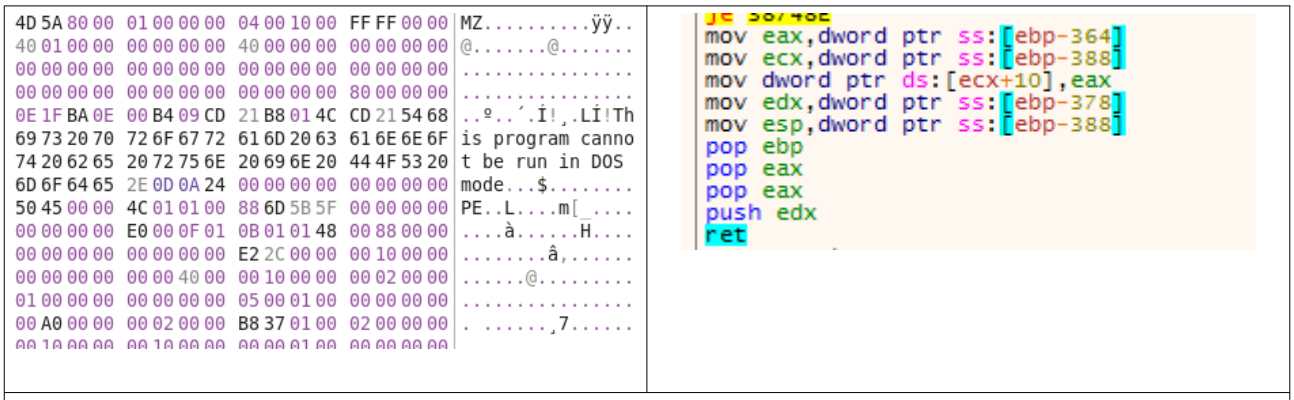



```
void decode(uint32_t* payload, size_t size)
{
     for (size_t i = 0; i < size / 4; i++)
          payload[i] = (payload[i] + i * 4) ^ (0x3e9 + i * 4)
}
```

L’algoritmo di decodifica, in assembly ed in C.

2 E’ prassi chiamare questo genere di codice “shellcode” ma data la sua posizione e modalità di esecuzione, facciamo fatica a considerarlo tale.

Il risultato è un PE (si tratta di SL) che viene mappato nell'address space dello stesso processo sovrascrivendo codice e dati del packer. Infine il packer salta all'entry-point del PE.




Il PE decodificato (a sinistra) e il codice che passa il controllo all'entry-point (a destra, notare il `push` di `edx` prima di `ret`).

## Estrazione automatica

Gli algoritmo di decodifica sono stati completamente reversati per cui l'unico ostacolo rimasto è trovare l'offset del payload nel file del packer.
Per fortuna, avendo il file PE una struttura ben nota, è alla peggio possibile provate tutti i possibili offset.

Possiamo però ridurre la lista di offset candidati considerando che la dimensione del payload deve essere minore di quella del file e maggiore di una soglia minima (ad esempio 50KiB).
Questo riduce notevolmente il numero di offset da provare.

Di quelli che rimangono è possibile vedere quanti hanno 44 byte di zero ogni 100 byte e scartare quelli che ne hanno un numero diverso. Questo test risulta essere più insidioso perchè un byte nullo può comparire tra quelli originali del payload e quindi possiamo solo scartare gli offset che indicherebbero un payload in cui ci sono **meno** di 44 byte nulli ogni 100 byte.
Siamo costretti a tenere i payload con più di 44 byte nulli (ogni 100 byte) tra i candidati.

A questo punto, dei candidati rimasti, possiamo prendere quello che ha meno zeri extra. Questo elimina i problemi dati da grosse sezioni di zeri precedute da una DWORD non zero.

Di seguito viene fornito un PoC in C per l'estrazione automatica del payload. *Si avverte il lettore che la qualità del codice è incredibilmente scarsa, paragonabile, in bruttezza, solo a quello riscontrabile nelle gare di competitive programming.*

```
[vmtest@localhost tmp]$ ./slp 1.exe
Payload at 5c8
[vmtest@localhost tmp]$ file slp.output
slp.output: PE32 executable (GUI) Intel 80386, for MS Windows
```

L'utilizzo del programma di estrazione automatica e il suo file di output (`slp.output`).

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>

typedef unsigned char uchar;
typedef unsigned int uint;

int count_zeros(uchar* buffer, int i, int size)
{
  int count;
  for (count = 0; i + count < size && ! buffer[i + count]; count++);
  return count;
}

int is_chunked(uchar* buffer, int i, int size, float* over)
{
  uint csize = *(uint*)(buffer + i - 4);
  uint j = csize;
  int zeros = 0;
  int c;

  if (csize >= (uint)size || csize == 0 || csize > (1<<20) || csize < (50 << 10))
    return 0;

  while (j >= 100)
    if ((c = count_zeros(buffer, i + 100, size)) < 44)
```

```
      break;
    else
    {
      i += 144;
      j -= 100;
      zeros += (c - 44);
    }

  *over = zeros/csize;
  return  j < 100;
}

int main(int argc, char** argv)
{
  if (argc != 2)
    return fprintf(stderr, "Usage: sl INJECTED_DLL\n"), 1;

  FILE* f = fopen(argv[1], "rb");
  if ( ! f)
    return fprintf(stderr, "Cannot open %s: %s (%d)\n", argv[1], strerror(errno),
errno), 2;

  fseek(f, 0, SEEK_END);
  long size = ftell(f);
  fseek(f, 0, SEEK_SET);

  unsigned char* buffer = malloc(size);
  if ( ! buffer)
    return fprintf(stderr, "Cannot allocate %ld bytes: %s (%d)\n", size,
strerror(errno), errno), 3;

  if (fread(buffer, size, 1, f) != 1)
    return fprintf(stderr, "Cannot read %s: %s (%d)\n", argv[1], strerror(errno),
errno), 4;

  float min_z = -1;
  int off = 0;
  float c = 0;

  //Find the chuncked section with least extra zeros
  for (int i = 0; i < size - 144; i++)
  {
    //if (i != 0x5c8)
    //  continue;

    if ( ! is_chunked(buffer, i, size, &c))
      continue;

    if (min_z < 0 || min_z > c)
    {
      off = i;
      min_z = c;
    }

  }

  if (off == 0)
    return fprintf(stderr, "Payload not found :(\n"), 5;

  uint psize = *(uint*)(buffer + off - 4);
  uchar* payload = malloc(psize);

  if ( ! payload)
    return fprintf(stderr, "Cannot allocate %d bytes: %s (%d)\n", psize,
strerror(errno), errno), 6;

  int i = 0, j = 0, k = psize;

  //Remove the null bytes
  while (k > 0)
  {
    memcpy(payload + i, buffer + off + j, k >= 100 ? 100 : k);
    i += 100;
    j += 144;
    k -= 100;
  }
```

```
  //Decode the payload
  uint* ptr = (uint*)payload;
  for (uint i = 0; i < psize / 4; i++)
  {
    ptr[i] = (ptr[i] + 4*i) ^ (0xe0 + 4*i);
  }

  //Decode the PE
  uint pe_size = ptr[0x150 / 4];
  uint* ptr2 = (uint*)(payload + 0x154);

  for (uint i = 0; i < pe_size / 4; i++)
  {
    ptr2[i] = (ptr2[i] + 4*i) ^ (0x3e9 + 4*i);
  }

  FILE* g = fopen("slp.output", "wb");
  fwrite(ptr2, pe_size, 1, g);
  fclose(g);

  printf("Payload at %x\n", off);
  return 0;
}
```

## Estrazione tramite debug

Un metodo veloce per ottenere il payload è mettere un breakpoint su `VirtualAlloc` e successivamente un breakpoint hardware sull'accesso al buffer appena allocato.

In alternativa anche `VirtualProtect` può essere usata per un breakpoint, questa è chiamata al momento della mappatura del PE, per cui il codice del packer ha l'indirizzo del PE in chiaro nei registri o in qualche variabile locale.
E' anche possibile, a questo punto, semplicemente cercare la firma di un PE nel working set del packer.

# Smoke Loader

Rimanendo fedeli al nostro proposito di non creare un altro documento di analisi di SL, in questa sezione ci proponiamo solo di condividere alcuni consigli.

## Anti disassembly

SL utilizza una tecnica molto semplice, ma ancora efficace, per confondere i disassemblatori; si faccia ad esempio riferimento all'immagine sotto.

```
                public start
start:
                call    $+5
                jnz     short near ptr loc_402CEC+1
                jz      short near ptr loc_402CEC+1
                push    ecx

loc_402CEC:                             ; CODE XREF: .text:00402CE7↑j
                                        ; .text:00402CE9↑j
                mov     eax, ds:0C80AEB5Bh
                sub     ebx, 2CE7h
                jmp     short loc_402CFE
; ---------------------------------------------------------------------------
                db 8Ah, 0EBh, 0F5h
                db 0C8h, 8Ah
; ---------------------------------------------------------------------------

loc_402CFE:                             ; CODE XREF: .text:00402CF7↑j
                jz      short near ptr loc_402D04+3
                jnz     short near ptr loc_402D04+3
                outs    dx, byte ptr es:[esi]
```

IDA da precedenza al ramo di esecuzione principale, piuttosto che ai branch, e quindi non risce a decodificare correttamente la funzione.

Le istruzioni `jnz` e `jz` (seconda e terza istruzione della funzione) saltano in mezzo all'istruzione `mov eax, ds:0c80aeb5bh` alla posizione `loc_402cec`.
Siccome queste due istruzioni, in coppia, sono equivalenti ad un salto incondizionato, il vero flusso di esecuzione è quello che parte in mezzo all'istruzione `mov eax, ds:0c80aeb5bh` ma il disassemblatore ha già analizzato questo percorso quando ha considerato entrambi i salti non presi.
Ovviamente questa è una condizione impossibile ma il disassemblatore (IDA nello specifico) non è in grado di accorgesene e si trova quindi con due rami di esecuzione sovrapposti e di cui uno inizia nel mezzo di un'istruzione dell'altro.

Il risultato finale è che il codice mostrato non è quello realmente eseguito.

E' possibile sistemare il codice manualmente[3] o **azzardando** una soluzione automatica.
Il codice problematico ha sempre la forma di una coppio di istruzioni `jz X` / `jnz X` o `jnz X` / `jz X` , per cui potremmo cercare queste istruzioni e trasformare in un salto incondizionale.
L'azzardo stra proprio nella ricerca: il codice x86 ha lunghezza variabile e non ha alcuna proprietà

3 Ad esempio con IDA, usando il tasto *u* e poi il tasto *c* all'indirizzo giusto.

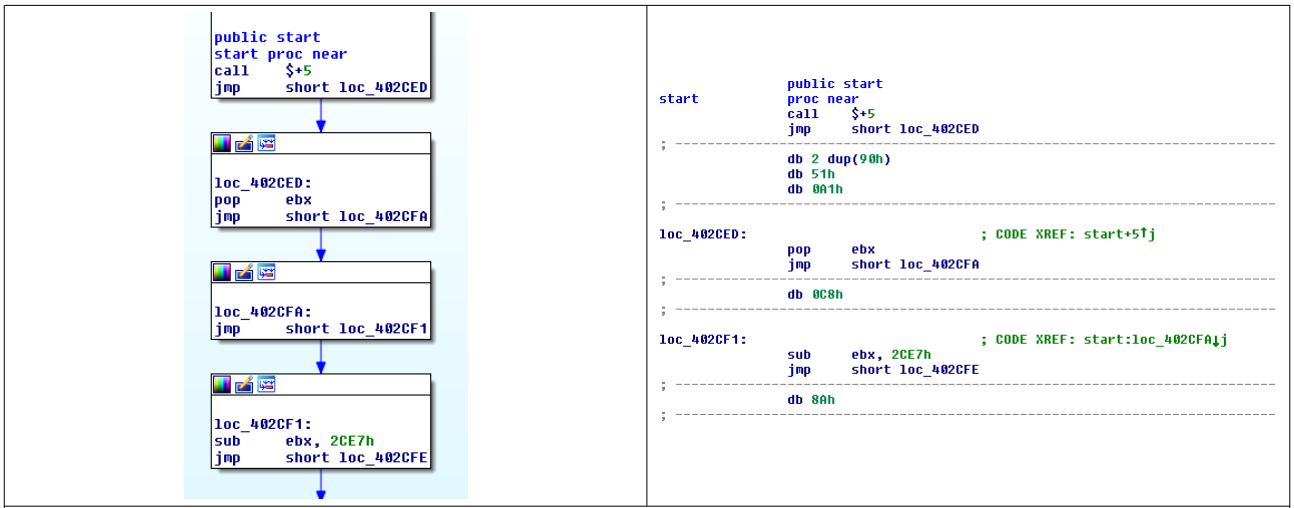
di sincronizzazione[4] per cui non è possibile cercare un'istruzione senza disassemblarle tutte e seguire tutti i possibili flussi di esecuzione; un lavoro troppo oneroso.

Il pattern da cercare (si faccia riferimento al secondo manual Intel per la codifica delle istruzioni) è tuttavia piuttosto specifico (si veda sotto) e si può quindi azzardare l'ipotesi che non compaia mai in altre instanze.

| | | | | |
|---|---|---|---|---|
| **74** | **x + 2** | **75** | **x** | `jz target / jnz target` |
| **75** | **x + 2** | **74** | **x** | `jnz target / jz target` |
| **EB** | **x + 2** | **90** | **90** | `jmp target / nop / nop` |

I byte da cercare (in valori sono in esadecimale, x indica un valore qualsiasi) nelle prime due righe. Nella riga evidenziata i byte da scrivere in sostituzione degli originali. Nella colonna a destra le relative istruzioni.

Dopo il patching IDA non ha problemi ad analizzare il codice.




L'entry-point di SL dopo il patching. IDA mostra il codice reale e riesce persino a crearne la relativa funzione (a sinistra).

Il patching può essere fatto con lo strumento che si ritiene più idoneo, nel nostro caso abbiamo usato un primitivo programma C.

```
#include <stdio.h>
#include <stdlib.h>

#define PTR(x, y) (x*)(y)

void patch(unsigned char* buf, size_t start, size_t size)
{
  for (size_t i = start; i < start + size; i++)
    if ((buf[i] & 0xfe) == 0x74 && (buf[i] ^ buf[i+2]) == 1 && buf[i+1] == buf[i+3]+2)
    {
        buf[i] = 0xeb;
```

4 Ad esempio non richiede che le istruzioni siano allineate (sebbene farlo migliori le prestazioni, specie per i cicli, compatibilmente con la microarchitettura in uso) ed è impossibile disassemblarlo al contrario (si provi a disassemblare `mov eax, 90909090h / add eax, 1` partendo da add eax, 1 ed andando indietro).

```
        buf[i+2] = buf[i+3] = 0x90;

        //FIXME: Use the PE to get the real image base address and the RVA of the .text
section
        printf("Patched %lx\n", 0x401000 + i - start);
        i+=3;
    }

}

size_t get_first_sec(unsigned char* buf, size_t* out_size)
{
  unsigned char* pe = buf + *PTR(unsigned int, buf + 0x3c);
  unsigned short size_opt = *PTR(unsigned short, pe + 0x14);
  unsigned int* secs = PTR(unsigned int, pe + 0x18 + size_opt);

  *out_size = secs[4];
  return secs[5];
}

int main(int argc, char** argv)
{
    if (argc != 3)
      return fprintf(stderr, "Usage: rep INPUT OUTPUT\n"), 1;

    FILE* f = fopen(argv[1], "rb");
    if ( ! f)
      return fprintf(stderr, "Cannot open %s\n", argv[1]), 2;

    fseek(f, 0, SEEK_END);
    long s = ftell(f);
    fseek(f, 0, SEEK_SET);

    unsigned char* buf = malloc(s);
    if (fread(buf, s, 1, f) != 1)
      return fprintf(stderr, "Cannot read input file\n"), 3;
    fclose(f);

    //FIXME: Actually find the .text section, not just the first section
    size_t sec_size, sec_start = get_first_sec(buf, &sec_size);

    patch(buf, sec_start, sec_size);

    f = fopen(argv[2], "wb");
    if ( ! f)
      return fprintf(stderr, "Cannot open %s\n", argv[2]), 4;
    if (fwrite(buf, s, 1, f) != 1)
      return fprintf(stderr, "Cannot write output file\n"), 5;
    fclose(f);

    return 0;
}
```

## Decifratura al volo del codice

Come noto, SL utilizza una tecnica di decifratura al volo del codice.
Quando una funzione viene eseguita, la prima e l'ultima azione compiute sono quelle di decifrare e ricifrare il proprio codice (l'algoritmo di cifratura è un semplice xor, quindi simmetrico).

La funzione che effettua la cifratura è facilmente riconoscibile perchè:

1. E' la prima e l'ultima funzione chiamata in molte procedure.
2. Ha come argomenti l'RVA del codice da cifrare/decifrare (in `eax`) e la sua dimensione (in `ecx`).

Qui sotto è possibile vedere la chiamata alla suddetta funzione (rinominata `dec_after`) dove si può notare che il codice da decifrare è immediatamente successivo alla chiamata della funzione (l'RVA della chiamata è 270dh, sommati i 5 byte dell'istruzione call si ottiene che la prossima istruzione ha RVA 2712h, che è quello passato in input). Questo è un pattern che si ripete sempre in SL: il codice da decifrare è immediatamente successivo alla prima chiamata a `dec_after` (da cui il nome).

Questo ha due conseguenze: la prima è che è necessario essere cauti quando si fa il debug di SL; la seconda che è possibile riconoscere le chiamate che decifrano da quelle che ricifrano. Quest'ultime infatti avranno in ingresso un RVA che non è quello dell'istruzione successiva.

```
loc_4026EB:                             ; CODE XREF: .text:loc_4026F6↓j
                mov     eax, 2712h
                jmp     short loc_4026F9
; ---------------------------------------------------------------------------
                dw 65D7h
                db 3Dh, 0B4h
; ---------------------------------------------------------------------------

loc_4026F6:                             ; CODE XREF: .text:004026E5↑j
                jmp     short loc_4026EB
; ---------------------------------------------------------------------------
                db 0C4h
; ---------------------------------------------------------------------------

loc_4026F9:                             ; CODE XREF: .text:004026F0↑j
                jmp     short loc_40270A
; ---------------------------------------------------------------------------

loc_4026FB:                             ; CODE XREF: .text:0040271C↓j
                sub     esp, [ebp+edi-4Ch]

loc_4026FF:                             ; CODE XREF: .text:loc_40270A↓j
                mov     ecx, 55h
                jmp     short loc_40270D
; ---------------------------------------------------------------------------
                sub     esp, [ebp+edi-4Ch]

loc_40270A:                             ; CODE XREF: .text:loc_4026F9↑j
                jmp     short loc_4026FF
; ---------------------------------------------------------------------------
                db 8
; ---------------------------------------------------------------------------

loc_40270D:                             ; CODE XREF: .text:00402704↑j
                call    dec_after
```

La chiamta alla funzione di decifratura del codice, con il flusso di esecuzione semplificato e in cui sono in evidenza i parametri di ingresso.

Grazia al secondo punto è possibile creare un script IDAPython per decifrare il codice.

```
loc_40270D:                             ; CODE XREF: .text:00402704↑j
                call    dec_after
                stosb
                push    esp
                sub     eax, 0A48CD6A8h
                loope   loc_402770
                aas
                jno     short near ptr loc_4026FB+1
                push    esp
                sub     ecx, ecx
                retf    0DEDFh
; ---------------------------------------------------------------------------
                dd 55E1A4DEh, 0DD64E62Bh, 21212120h, 0E6C5CA
                dd 2ECA2121h, 951C454Bh, 21063399h, 4B26CA21
                db 45h, 1Ch
```

```
loc_40270D:                                             ; COD
                nop
                nop
                nop
                nop
                nop
                mov     esi, [ebp+0Ch]
                mov     edi, esi

loc_402717:                                             ; COD
                lodsd
                test    eax, eax
                jz      short loc_40273A
                push    eax
                push    dword ptr [ebp+8]
                call    loc_402610
                test    eax, eax
                jz      short loc_402733
                mov     dword ptr [ebp-4], 1
                stosd
                jmp     short loc_402717
```

Sopra il codice cifrato, a destra quello decifrato (con la chiamata a `dec_after` rimossa)

```
from idaapi import *
from idautils import *
from idc import *

#Get the previous instruction
def prev(ea):
       #Is this ea referred from anoter ea (i.e. by a jump)?
       #The filter hack is used to remove some IDA misdetections
    xr = [x for x in filter(lambda x: x != ea - 1, [x.frm for x in XrefsTo(ea, 0)])]

       #We take the first xref if any, otherwise we take the prev instruction
(spatially)
    if len(xr) > 0:
        return xr[0]
    else:
        return PrevHead(ea)



#Get the EA of dec_after
da_ea = None
for (ea, name) in Names():
    if name == "dec_after":
        da_ea = ea

#Get all the call to dec_after
xr = XrefsTo(da_ea, 0)
for x in xr:

    ea = x.frm
    size = 0
    addr = 0
       #Get back max 10 instructions (from the call)
    for y in range(10):
        ea = prev(ea)

             #Is this a mov to ecx with an immediate?
        if GetMnem(ea) == "mov" and GetOpType(ea, 1) == 5 and GetOpnd(ea, 0) == "ecx"
and size==0:
            size = GetOperandValue(ea, 1)

        #Ditto but for eax?
        if GetMnem(ea) == "mov" and GetOpType(ea, 1) == 5 and GetOpnd(ea, 0) == "eax"
and addr==0:
            addr = 0x400000 + GetOperandValue(ea, 1)

        if size > 0 and addr > 0:
            break

    if size <= 0 or addr == 0:
        print("Call at " + hex(x.frm) + " not patched!")
        continue

       #Is this an encrypting call?
    if addr == x.frm + 5:
        ptr = addr
        while size > 0:
            v = Byte(ptr) ^ 0x21
            PatchByte(ptr, v)
            ptr += 1
            size -= 1

       #nop the call
    for i in range(5):
        PatchByte(x.frm + i, 0x90)
        pass
```

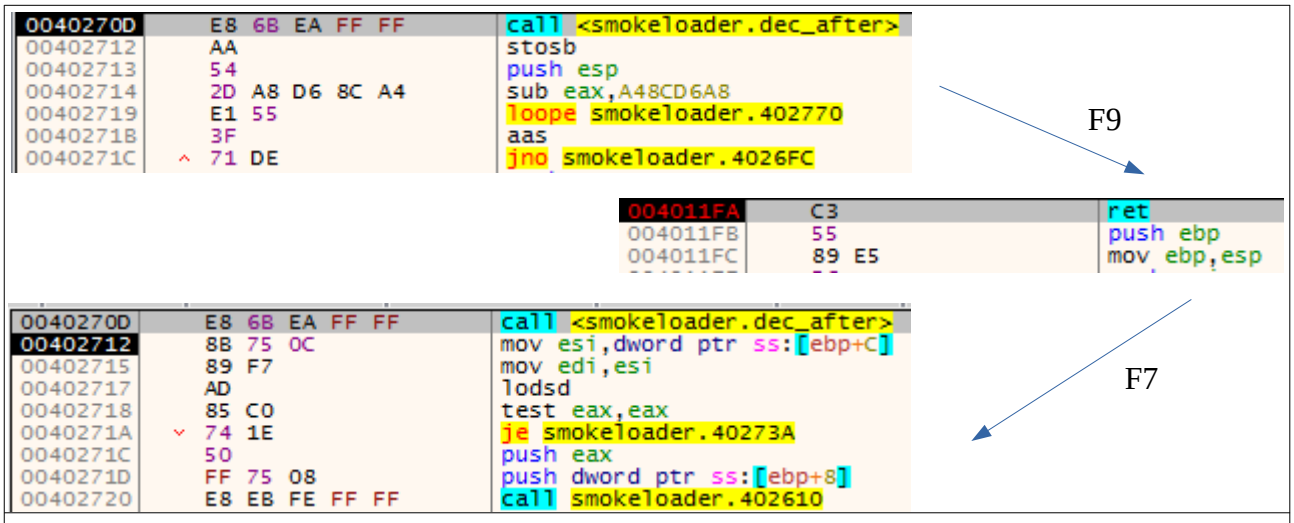
Infine, durante il debug di SL è necessario identificare quanto prima la funzione `dec_after` poichè non è possibile farvi step-over in quanto modifica il codice dopo di sè.
La funzionalità di step-over (o in generale di qualsiasi breakpoint sul codice che non sia hardware

nè un single-step) infatti altera il codice del programma inserendo un byte di valore `0cch` (l'opcode corto di `int 03h`) per richiamare il debugger.

```
0040270D    E8 6B EA FF FF    call <smokeloader.dec_after>
00402712    AA                stosb
00402713    54                push esp
00402714    2D A8 D6 8C A4    sub eax,A48CD6A8
00402719    E1 55             loope smokeloader.402770
0040271B    3F                aas
0040271C  ^ 71 DE             jno smokeloader.4026FC
```

Premere *F8* (step-over) in questo punto comprometterebbe il codice del malware, rendendolo non analizzabile. Infatti lo step-over è implementato scrivendo `0cch` al posto del byte `0aah` a `402712h`, cambiando quindi il codice cifrato del malware e compromettendone la decodifica.

Il modo corretto[5] di continuare con il debug è **mettere un breakpoint** alla fine di `dec_after` e una volta arrivati alla chiamata proseguire con la combinazione *F9* (continue) e *F7* (single-step) per tornare al codice appena decifrato.
Il breakpoint in `dec_after` è scomodo se si vogliono saltare porzioni del malware ma molto utile se lo si vuole analizzare passo passo (infatti è chiamato prima del codice vero e proprio di ogni funzione e prima dell'uscita da ogni funzione).




I passi per il debug del codice cifrato di SL

5 Altri metodi sono possibili ovviamente.

## Recuperare la DLL

Per recuperare la DLL è sufficiente mettere un breakpoint su `RtlDecompressBuffer` ed eseguire i passi 1 e 2, mostrati ad inizio documento, per eludere le tecniche anti VM e anti tool. Il secondo parametro dell'API conterrà, una volta eseguita, la DLL in chiaro.

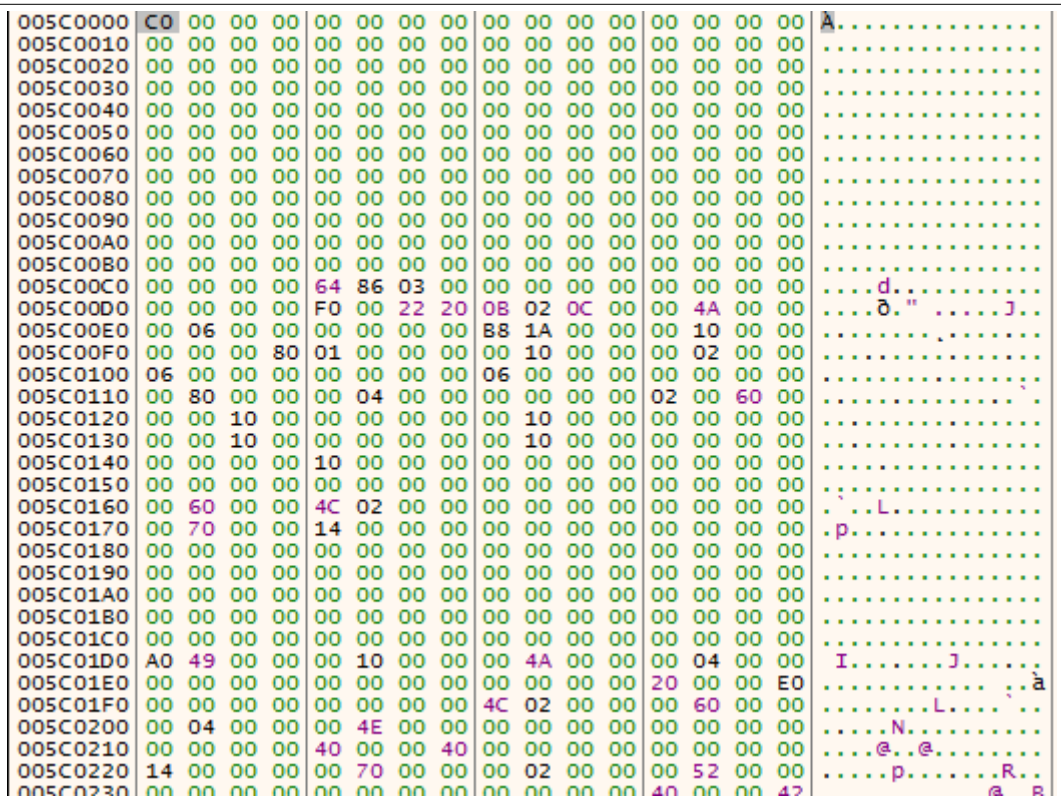
La DLL è poco riconoscibile perchè l'header MZ, lo stub DOS, la firma PE ed i nomi delle sezioni sono stati rimossi. Ma il modo in cui è mappata successivamente dal codice del malware indica chiaramente che si tratta di un PE.

E' possibile ricostruire la sua struttura, e poi analizzarlo, semplicemente sovrascrivendo i primi bytes (massimo fino a 0c0h bytes[6]) con un qualsiasi header MZ, sistemare in questo il campo che punta all'header PE (campo che si trova all'offset 3ch) e poi aggiungere la firma "PE" all'offset 0c0h.

```
004012E8    6A 04                 push 4
004012EA    68 00 30 00 00        push 3000
004012EF    52                    push edx
004012F0    51                    push ecx
004012F1    50                    push eax
004012F2    6A FF                 push FFFFFFFF
004012F4    FF 93 80 00 00 00     call dword ptr ds:[ebx+80]          [ebx+80]:ZwAllocateVirtualMemory
004012FA    85 C0                 test eax,eax
004012FC    75 25                 jne packer.401323
004012FE    8B 45 F4              mov eax,dword ptr ss:[ebp-C]
00401301    8D 4D FC              lea ecx,dword ptr ss:[ebp-4]
00401304    8D 55 F8              lea edx,dword ptr ss:[ebp-8]
00401307    51                    push ecx
00401308    57                    push edi
00401309    56                    push esi
0040130A    FF 32                 push dword ptr ds:[edx]
0040130C    50                    push eax
0040130D    6A 02                 push 2
0040130F    FF 93 A0 00 00 00     call dword ptr ds:[ebx+A0]          [ebx+A0]:RtlDecompressBuffer
00401315    85 C0                 test eax,eax
00401317    75 0A                 jne packer.401323
```

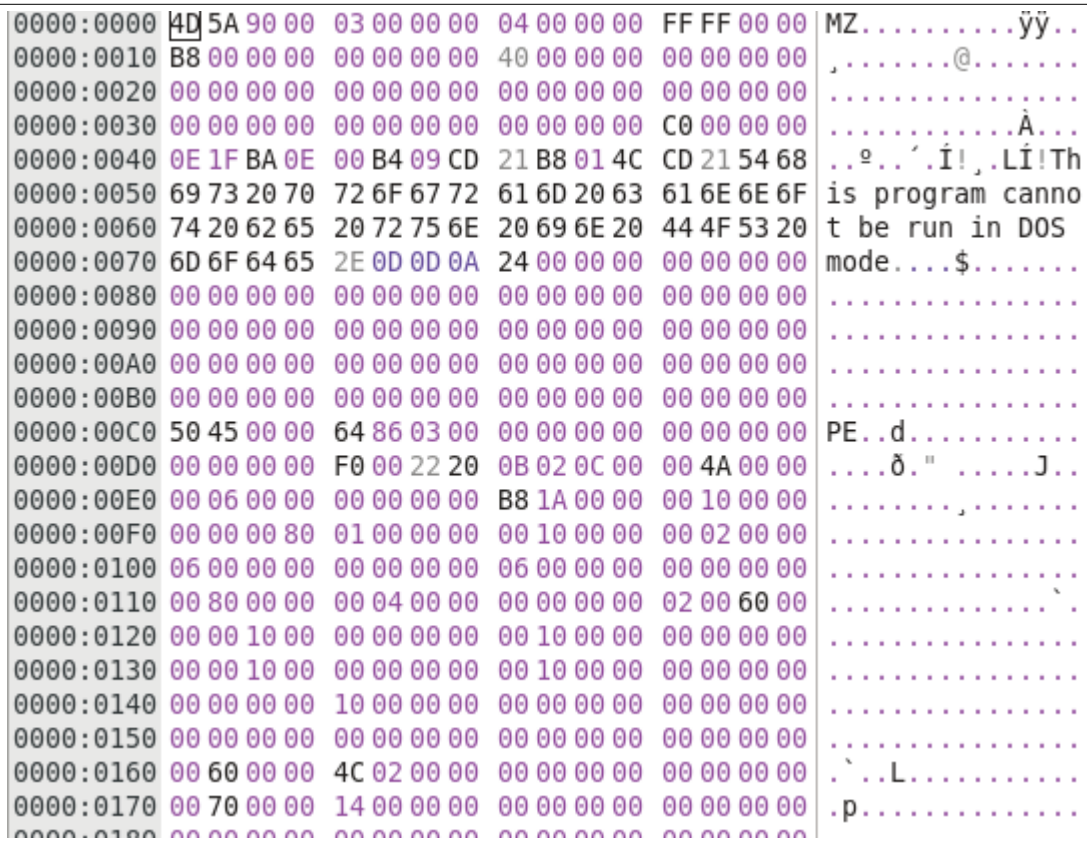
Il codice che decomprime la DLL da iniettare in *explorer.exe*.

6 Si prende ad esempio la DLL trovata nel sample usato per la redazione di questo documento. La prima DWORD nel buffer indica la posizione dell'header PE e quindi anche il numero di byte liberi ad inizio buffer.



```
005C0000 C0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 À...............
005C0010 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
005C0020 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
005C0030 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
005C0040 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
005C0050 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
005C0060 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
005C0070 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
005C0080 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
005C0090 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
005C00A0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
005C00B0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
005C00C0 00 00 00 00 64 86 03 00 00 00 00 00 00 00 00 00 ....d...........
005C00D0 00 00 00 00 F0 00 22 20 0B 02 0C 00 00 4A 00 00 ....ð." .....J..
005C00E0 00 06 00 00 00 00 00 00 B8 1A 00 00 00 10 00 00 ........¸.......
005C00F0 00 00 00 80 01 00 00 00 00 10 00 00 00 02 00 00 ................
005C0100 06 00 00 00 00 00 00 00 06 00 00 00 00 00 00 00 ................
005C0110 00 80 00 00 00 04 00 00 00 00 00 00 02 00 60 00 ..............`.
005C0120 00 00 10 00 00 00 00 00 00 10 00 00 00 00 00 00 ................
005C0130 00 00 10 00 00 00 00 00 00 10 00 00 00 00 00 00 ................
005C0140 00 00 00 00 10 00 00 00 00 00 00 00 00 00 00 00 ................
005C0150 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
005C0160 00 60 00 00 4C 02 00 00 00 00 00 00 00 00 00 00 .`..L...........
005C0170 00 70 00 00 14 00 00 00 00 00 00 00 00 00 00 00 .p..............
005C0180 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
005C0190 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
005C01A0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
005C01B0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
005C01C0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
005C01D0 A0 49 00 00 00 10 00 00 00 4A 00 00 00 04 00 00  I.......J......
005C01E0 00 00 00 00 00 00 00 00 00 00 00 00 20 00 00 E0 ............ ..à
005C01F0 00 00 00 00 00 00 00 00 4C 02 00 00 00 60 00 00 ........L....`..
005C0200 00 04 00 00 00 4E 00 00 00 00 00 00 00 00 00 00 .....N..........
005C0210 00 00 00 00 40 00 00 40 00 00 00 00 00 00 00 00 ....@..@........
005C0220 14 00 00 00 00 70 00 00 00 02 00 00 00 52 00 00 .....p.......R..
005C0230 00 00 00 00 00 00 00 00 00 00 00 00 40 00 00 42                @..B
```

La DLL così come decompressa. La prima DWORD indica dove inizia l'header PE. Poichè l'header MZ è stato semplicemente cancellato è possibile "re-incollarlo" direttamente sopra i primi byte del buffer (avendo cura di aggiustarne il campo che punta all'header PE).
Va inoltre aggiunta la firma "PE".




La DLL con l’header MZ, lo stub DOS e la firma PE ripristinati. Notare come i byte da 0c0h in poi non siano stati modificati.

# La DLL

La DLL può essere più facilmente debuggata se si elimina SL dalla catena di infezione.

Infatti questa è caricata ed eseguita normalmente eccetto che il parametro passato a `DllMain` non è il base address della DLL ma il base address di un buffer (di 5 pagine) di lavoro.

Risulta quindi sufficiente creare un semplice programma che carica la DLL senza eseguire DllMain in modo da eseguirla con i parametri giusti.

Ad esempio, il seguente programma assembly (da assemblare con *NASM* e linkare con un linker qualsiasi, consigliamo *golink*) può essere usato per il debug della DLL.

```
BITS 64

%define DONT_RESOLVE_DLL_REFERENCES 1

GLOBAL start

EXTERN LoadLibraryExA

SECTION .bss
	buffer resb 5000h

SECTION .data
	strDll db "injected.dll", 0

SECTION .text

start:

	lea rcx, [REL strDll]
	xor edx, edx
	mov r8d, DONT_RESOLVE_DLL_REFERENCES
	call LoadLibraryExA

	mov ebx, DWORD [rax + 3ch]
	add rbx, rax
	 mov ebx, DWORD [rbx + 28h]
	add rax, rbx

	and rsp, -16
	lea rcx, [REL buffer]
	call rax

	ret
```

A tal fine:

1. Assemblare e linkare il programma.
2. Rinominare la DLL in *injected.dll*.
3. Debuggare il programma ed eseguirlo fino all'istruzione `call rax`.
4. Continuare normalmente con il debug.

## Stringhe

Le stringhe sono codificate come descritto in letteratura, in particolare i domini sono xorati con una chiave di un byte (diversa per dominio) e le altre stringhe sono cifrate con RC4 con una chiave di 4 byte salvata prima delle stringhe stesse.

E' quindi possibile scorrere ogni offset del file e vedere se considerandolo l'inizio della sezione delle stringhe, si ottiene del testo i cui caratteri sono nel range stampabile (per ogni stringa).
In caso positivo si può essere ragionevolmente sicuri di aver trovato e decodificato le stringhe della DLL.

Per i domini si può considerare l'offset corrente come l'inizio di un dominio e poi semplicemente provare a xorare la stringa "http:" (da provare anche "https:", compito lasciato al lettore) con i primi 5 byte del dominio cifrato.
Il risultato è la chiave da usare[7] per decifrare il dominio.

Il seguente, primitivo, programma C effettua la decodifica.

```
[vmtest@localhost tmp]$ ./sl injected.dll
https://dns.google/resolve?name=microsoft.com
Software\Microsoft\Internet Explorer
advapi32.dll
Location:
plugin_size
\explorer.exe
user32
advapi32
urlmon
ole32
winhttp
ws2_32
dnsapi
shell32
svcVersion
Version
<?xml version="1.0"?><scriptlet><registration classid="{00000000-0000-0000-
0000-00000000%04X}"><script language="jscript"><!
[CDATA[GetObject("winmgmts:Win32_Process").Create("%ls",null,null,null);]]></
script></registration></scriptlet>
.bit
%sFF
%02x
%s%08X%08X
%s\%hs
%s%s
regsvr32 /s %s
regsvr32 /s /n /u /i:"%s" scrobj
%APPDATA%
%TEMP%
.exe
.dll
.bat
:Zone.Identifier
POST
Content-Type: application/x-www-form-urlencoded
open
Host: %s
PT10M
1999-11-30T00:00:00
```

7 In realtà è la chiave ripetuta cinque volte, poichè la chiave è di un solo byte.

```
NvNgxUpdateCheckDaily_{%08X-%04X-%04X-%04X-%08X%04X}
Accept: */*
Referer: %S
http://the3rd.ml/
http://xieliorn.com/
http://kavauelo.co/
http://piesislel.is/
http://danae.to/
http://3rdcamp.ga/
```

Le stringhe nella DLL decodificate ed i domini usati da SL.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>

static unsigned char ID[256];

void init()
{
  for (int i = 0; i < 256; i++)
    ID[i] = (unsigned char)i;


}

void schedule(unsigned char* key, int len, unsigned char* output)
{
  memcpy(output, ID, 256);

  unsigned int j = 0, aux;
  for (int i = 0; i < 256; i++)
  {
    j = ((j + output[i] + key[i % len])) % 256;

    aux = output[i];
    output[i] = output[j];
    output[j] = aux;
  }
}

void xor_rollover(unsigned char* key, unsigned char* output, int len)
{
  int j = 0, i = 0, aux;
  for(int x = 0 ; x < len ; x++)
  {
      i = (i+1) % 256;
      j = (j + key[i])% 256;


      aux = key[j];
      key[j]= key[i];
      key[i] = aux;


      output[x]= output[x]^key[(key[i] + key[j]) % 256];
  }
}




int count_strings(unsigned char* buffer, int next_str, int size)
{
    int count = 0;

    for (int len = buffer[next_str]; len > 0 && next_str < size; next_str += (len + 1),
len = buffer[next_str])
      count++;

    return count;
```

```
}

int decrypt_string(unsigned char* buffer, int k, int i, unsigned char* key, unsigned
char* data, int* next, int* olen)
{
    int len = buffer[i];
    *olen = len;
    *next = i + len + 1;

    memcpy(data, buffer + i + 1, len);

    schedule(buffer + k, 4, key);
    xor_rollover(key, data, len);
    data[len] = 0;
    int zero=0;

    for (int k = 0; k < len; k++)
    {
      zero |= !data[k];
      if ((data[k] < 0x20 && (data[k] != 0xa && data[k] != 0xd && data[k] != 0)) ||
data[k] > 0x7e)
      {
        return 0;
      }
    }

    if (zero)
    {
      int o = 0;

      for (int y = 0; y < len; y++)
        if (data[y])
          data[o++] = data[y];
      data[o] = 0;
      *olen = o;
    }

    return 1;
}

int main(int argc, char** argv)
{
  if (argc != 2)
    return fprintf(stderr, "Usage: sl INJECTED_DLL\n"), 1;

  FILE* f = fopen(argv[1], "rb");
  if ( ! f)
    return fprintf(stderr, "Cannot open %s: %s (%d)\n", argv[1], strerror(errno),
errno), 2;

  fseek(f, 0, SEEK_END);
  long size = ftell(f);
  fseek(f, 0, SEEK_SET);

  unsigned char* buffer = malloc(size);
  if ( ! buffer)
    return fprintf(stderr, "Cannot allocate %ld bytes: %s (%d)\n", size,
strerror(errno), errno), 3;

  if (fread(buffer, size, 1, f) != 1)
    return fprintf(stderr, "Cannot read %s: %s (%d)\n", argv[1], strerror(errno),
errno), 4;


  init();
  unsigned char key[256], data[257];


  for (int i = 0x400; i < size; i++)
  {

    int count = count_strings(buffer, i, size);

    if (count < 20)
      continue;
```

```
    int test = i;
    int k, len;
    for (k = 0; k < count && decrypt_string(buffer, i-4, test, key, data, &test, &len);
k++);

    if (k < 5)
      continue;

    test = i;
    while (decrypt_string(buffer, i-4, test, key, data, &test, &len))
      if (len)
        printf("%s\n", data);


  }

  char* pref = "http";

  //Domains
  for (int i = 0x400; i < size - 5; i++)
  {

    int len = buffer[i];
    if (len < 5)
      continue;

    int possible_key = buffer[i+1] ^ 'h';

    memcpy(data, buffer + i + 1, len);

    for (int j = 0; j < len; j++)
      data[j] ^= possible_key;
    data[len] = 0;

    if (strncmp(data, "http:", 5) == 0)
      printf("%s\n", data);

    continue;

    memcpy(data, buffer + i, len);
    unsigned int key = *(unsigned int*)pref;
    unsigned char* bytes = (unsigned char*)&key;

    for (int j = 0; j < 4; j++)
      bytes[j] ^= data[j];

    for (int j = 0; j < len; j++)
      data[j] ^= bytes[j % 4];
    data[len] = 0;

    int no = 0;
    for (int j = 0; j < len; j++)
      if (data[j] >= 'a' && data[j] <= 'z' ||
          data[j] >= 'z' && data[j] <= 'Z' ||
          data[j] >= '0' && data[j] <= '9' ||
          data[j] == '-' ||  data[j] == '.' ||
          data[j] == '/' || data[j] == ':' ||
          data[j] == '?' || data[j] == '_')
      {
      }
      else
      {
        no = 1;
      }

    if ( ! no)
      printf("%s\n", data);

  }

  return 0;

}
```

# Appendice A

Un semplice programma a 64 bit per l’infezione da parte di SL.

```
BITS 64

GLOBAL start

SECTION .text

start:
  pause
 jmp start
```